**ESP-IDF console 组件**

它包含了开发基于串口的交互式控制终端所需要的所有模块，主要支持以下功能：

**行编辑**，由 linenoise 库具体实现，它支持处理退格键和方向键，支持回看命令的历史记录，支持命令的自动补全和参数提示。

将命令行拆分为参数列表。

**参数解析**，由 argtable3 库具体实现，该库提供解析 GNU 样式的命令行参数的 API。

**用于注册和调度命令的函数**。

帮助创建 REPL (Read-Evaluate-Print-Loop) 环境的函数。

配置

Linenoise 库不需要显式地初始化，但是在调用行编辑函数之前，可能需要对某些配置的默认值稍作修改。

linenoiseClearScreen()

使用转义字符清除终端屏幕，并将光标定位在左上角。

linenoiseSetMultiLine()

在单行和多行编辑模式之间进行切换。单行模式下，如果命令的长度超过终端的宽度，会在行内滚动命令文本以显示文本的结尾，在这种情况下，文本的开头部分会被隐藏。单行模式在每次按下按键时发送给屏幕刷新的数据比较少，与多行模式相比更不容易发生故障。另一方面，在单行模式下编辑命令和复制命令将变得更加困难。默认情况下开启的是单行模式。

linenoiseAllowEmpty()

设置 linenoise 库收到空行的解析行为，设置为 true 时返回长度为零的字符串 ("") ，设置为 false 时返回 NULL。默认情况下，将返回长度为零的字符串。

linenoiseSetMaxLineLen()

设置 linenoise 库中每行的最大长度，默认长度为 4096 字节，可以通过更新该默认值来优化 RAM 内存的使用。

主循环

linenoise()

在大多数情况下，控制台应用程序都会具有相同的工作形式——在某个循环中不断读取输入的内容，然后解析再处理。 linenoise() 是专门用来获取用户按键输入的函数，当回车键被按下后会便返回完整的一行内容。因此可以用它来完成前面循环中的“读取”任务。

linenoiseFree()

必须调用此函数才能释放从 linenoise() 函数获取的命令行缓冲区。

提示和补全

linenoiseSetCompletionCallback()

当用户按下制表键时， linenoise 会调用 补全回调函数 ，该回调函数会检查当前已经输入的内容，然后调用 linenoiseAddCompletion() 函数来提供所有可能的补全后的命令列表。启用补全功能，需要事先调用 linenoiseSetCompletionCallback() 函数来注册补全回调函数。

console 组件提供了一个现成的函数来为注册的命令提供补全功能 esp\_console\_get\_completion() （见下文）。

linenoiseAddCompletion()

补全回调函数会通过调用此函数来通知 linenoise 库当前键入命令所有可能的补全结果。

linenoiseSetHintsCallback()

每当用户的输入改变时， linenoise 就会调用此回调函数，检查到目前为止输入的命令行内容，然后提供带有提示信息的字符串（例如命令参数列表），然后会在同一行上用不同的颜色显示出该文本。

linenoiseSetFreeHintsCallback()

如果 提示回调函数 返回的提示字符串是动态分配的或者需要以其它方式回收，就需要使用 linenoiseSetFreeHintsCallback() 注册具体的清理函数。

历史记录

linenoiseHistorySetMaxLen()

该函数设置要保留在内存中的最近输入的命令的数量。用户通过使用向上/向下箭头来导航历史记录。

linenoiseHistoryAdd()

Linenoise 不会自动向历史记录中添加命令，应用程序需要调用此函数来将命令字符串添加到历史记录中。

linenoiseHistorySave()

该函数将命令的历史记录从 RAM 中保存为文本文件，例如保存到 SD 卡或者 Flash 的文件系统中。

linenoiseHistoryLoad()

与 linenoiseHistorySave 相对应，从文件中加载历史记录。

linenoiseHistoryFree()

释放用于存储命令历史记录的内存。当使用完 linenoise 库后需要调用此函数。

**将命令行拆分成参数列表**

console 组件提供 esp\_console\_split\_argv() 函数来将命令行字符串拆分为参数列表。该函数会返回参数的数量（argc）和一个指针数组，该指针数组可以作为 argv 参数传递给任何接受 argc，argv 格式参数的函数。

根据以下规则来将命令行拆分成参数列表：

参数由空格分隔

如果参数本身需要使用空格，可以使用 \ （反斜杠）对它们进行转义

其它能被识别的转义字符有 \\ （显示反斜杠本身）和 \" （显示双引号）

可以使用双引号来引用参数，引号只可能出现在参数的开头和结尾。参数中的引号必须如上所述进行转义。参数周围的引号会被 esp\_console\_split\_argv() 函数删除

示例：

abc def 1 20 .3 ⟶ [ abc, def, 1, 20, .3 ]

abc "123 456" def ⟶ [ abc, 123 456, def ]

`a\ b\\c\" ⟶ [ a b\c" ]

**参数解析**

对于参数解析，console 组件使用 argtable3 库。

下面列出argtable3 库使用说明。

**命令的注册与调度**

console 组件包含了一些工具函数，用来注册命令，将用户输入的命令和已经注册的命令进行匹配，使用命令行输入的参数调用命令。

应用程序首先调用 esp\_console\_init() 来初始化命令注册模块，然后调用 esp\_console\_cmd\_register() 函数注册命令处理程序。

对于每个命令，应用程序需要提供以下信息（需要以 esp\_console\_cmd\_t 结构体的形式给出）：

命令名字（不含空格的字符串）

帮助文档，解释该命令的用途

可选的提示文本，列出命令的参数。如果应用程序使用 Argtable3 库来解析参数，则可以通过提供指向 argtable 参数定义结构体的指针来自动生成提示文本

**命令处理函数**

命令注册模块还提供了其它函数：

esp\_console\_run()

该函数接受命令行字符串，使用 esp\_console\_split\_argv() 函数将其拆分为 argc/argv 形式的参数列表，在已经注册的组件列表中查找命令，如果找到，则执行其对应的处理程序。

esp\_console\_register\_help\_command()

将 help 命令添加到已注册命令列表中，此命令将会以列表的方式打印所有注册的命令及其参数和帮助文本。

esp\_console\_get\_completion()

与 linenoise 库中的 linenoiseSetCompletionCallback() 一同使用的回调函数，根据已经注册的命令列表为 linenoise 提供补全功能。

esp\_console\_get\_hint()

与 linenoise 库中 linenoiseSetHintsCallback() 一同使用的回调函数，为 linenoise 提供已经注册的命令的参数提示功能。

**初始化 REPL 环境**

除了上述的各种函数，console 组件还提供了一些 API 来帮助创建一个基本的 REPL 环境。

在一个典型的 console 应用中，你只需要调用 esp\_console\_new\_repl\_uart()，它会为你初始化好构建在 UART 基础上的 REPL 环境，其中包括安装 UART 驱动，基本的 console 配置，创建一个新的线程来执行 REPL 任务，注册一些基本的命令（比如 help 命令）。

之后你可以使用 esp\_console\_cmd\_register() 来注册其它命令。REPL 环境在初始化后需要再调用 esp\_console\_start\_repl() 函数才能开始运行。

**Argtable3使用**

**Argtable3介绍**

Argtable3 使用 NetBSD getopt 执行实际解析，遵循POSIX约定、大多数的 UNIX 程序和部分 Windows遵循该约定。它支持短选项（例如-abc和-o myfile）、长选项（例如–-scalar=7和–-verbose）以及未标记的参数（例如<file> [<file>]）。

**快速开始**

Argtable3 提供了一组arg\_xxx结构化函数来定义输入命令的格式。

1. 结构对应它支持的每种类型的参数（文字、整数、双精度、字符串、文件名等）。
2. 结构中每条命令都可以在输入时重复几次(出现的次数在程序中定义一个范围)。
3. 每个选项都可以设定一个长输入或短输入解析字符串，也可以两种都有。
4. 也可以不为选项定义标签 ( <file>)，在这种情况下，需要在输入时对应上它的位置即可 （带标签的选项可以出现在命令行上的任何位置）。

必需用arg\_xxx结构化参数函数生成参数实例。

**实例讲解**

下面是运行时实现命令行解析输入输出的结果 ：

$> util.exe --help

Usage: util.exe [-v] [--help] [--version] [--level=<n>] [-o myfile] <file> [<file>]...

Demonstrate command-line parsing in argtable3.

--help display this help and exit

--version display version information and exit

--level=<n> foo value

-v, --verbose verbose output

-o myfile output file

<file> input files

以下为Argtable3 实现的代码：

#include "argtable3.h"

/\* global arg\_xxx structs \*/

struct arg\_lit \*verb, \*help, \*version;

struct arg\_int \*level;

struct arg\_file \*o, \*file;

struct arg\_end \*end;

int main(int argc, char \*argv[])

{

/\* the global arg\_xxx structs are initialised within the argtable \*/

void \*argtable[] = {

help = arg\_litn(NULL, "help", 0, 1, "display this help and exit"),

version = arg\_litn(NULL, "version", 0, 1, "display version info and exit"),

level = arg\_intn(NULL, "level", "<n>", 0, 1, "foo value"),

verb = arg\_litn("v", "verbose", 0, 1, "verbose output"),

o = arg\_filen("o", NULL, "myfile", 0, 1, "output file"),

file = arg\_filen(NULL, NULL, "<file>", 1, 100, "input files"),

end = arg\_end(20),

};

生成结构化数据实例。

int exitcode = 0;

char progname[] = "util.exe";

int nerrors;

nerrors = arg\_parse(argc,argv,argtable);

/\* special case: '--help' takes precedence over error reporting \*/

if (help->count > 0)

{

printf("Usage: %s", progname);

arg\_print\_syntax(stdout, argtable, "\n");

printf("Demonstrate command-line parsing in argtable3.\n\n");

arg\_print\_glossary(stdout, argtable, " %-25s %s\n");

exitcode = 0;

goto exit;

}

/\* If the parser returned any errors then display them and exit \*/

if (nerrors > 0)

{

/\* Display the error details contained in the arg\_end struct.\*/

arg\_print\_errors(stdout, end, progname);

printf("Try '%s --help' for more information.\n", progname);

exitcode = 1;

goto exit;

}

exit:

/\* deallocate each non-null entry in argtable[] \*/

arg\_freetable(argtable, sizeof(argtable) / sizeof(argtable[0]));

return exitcode;

}

例如，我们可以需要一个输入一个int型数据如-–scalar=7.，此时需要用arg\_int结构体并实例化这个结构体。

下面是结构体定义：

struct arg\_int

{

struct arg\_hdr hdr;

int count;

int \*ival;

};

struct arg\_hdr hdr 保存Argtable3 库函数使用的私有数据。

int \*ival 指向一个整数数组，用来保存用户从命令行输入的值。（如果输入值不连续或存在空格则会被判断为两次输入同一个标签，次数将保存在count中）

int count保存用户输入值的次数。

这个结构体要用arg\_int函数初始化分配参数才能生效使用。

struct arg\_hdr hdr结构体如下，它启示录的系统调用的一些数据

struct arg\_int \* arg\_intn(

const char\* shortopts,

const char\* longopts,

const char \*datatype,

int mincount,

int maxcount,

const char \*glossary);

这些数据会占用一定内存，要用arg\_xxx构造函数来生成数据，如果不想用了，可以用free释放内存。

注意，一定要用arg\_xxx构造函数来生成数据。

以arg\_int为例，下面代码片将构造一个整数类型选项，其形式为--scalar=<n>

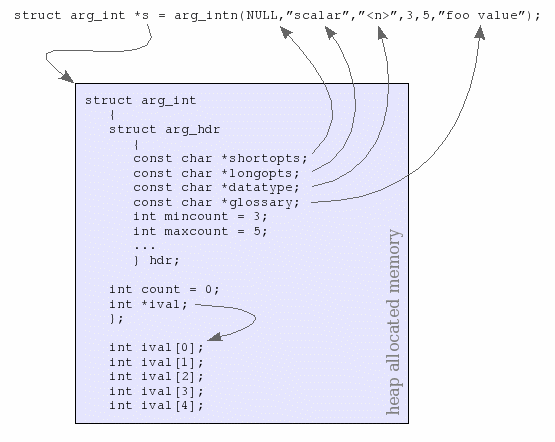
此时必须在命令行上输入 3~5 个数（含）。多或少都会报错。

struct arg\_int \*s;

s = arg\_intn(NULL, "scalar", "<n>", 3, 5, "foo value");

完成之后s将指向一个包含该arg\_int结构的内存块，基中包含5 个int ival型元素的数组。

对应结果如下：



从图中可以看出，定义时输入数据保存的位置。

count被初始化为0，\*ival指向我们定义的数据。Ival数据空间由hdr.maxcount值决定。

NULL在此示例中，\*shortopts 数据可以膦NULL，也可以为"k"：如下所示

s = arg\_intn("k", "scalar", "<n>", 3, 5, "foo value");

其结果也是一样的，只是多一个短输入选项标签。注意短选项一定是一个单字符串。

其实还可以定义多个不同标签，但他们功能都是一样的。

短的定义方法是，输入几个字符，每一个单字符就是一标签。

长标签则是用”,”分隔开。例如，

s = arg\_intn("kKx", "scalar,foo", "<n>", 3, 5, "foo value");

输入时可以使用下面任何方式其结果一样：-k<n> -K<n> -x<n> --scalar=<n> --foo=<n>

除了arg\_int结构类型，arg\_xxx还有以下结构类型：

struct arg\_lit

{

struct arg\_hdr hdr;

int count;

};

struct arg\_dbl

{

struct arg\_hdr hdr;

int count;

double \*dval;

};

struct arg\_str

{

struct arg\_hdr hdr;

int count;

const char \*\*sval;

};

struct arg\_rex

{

struct arg\_hdr hdr;

int count;

const char \*\*sval;

};

struct arg\_file

{

struct arg\_hdr hdr;

int count;

const char \*\*filename;

const char \*\*basename;

const char \*\*extension;

};

struct arg\_date

{

struct arg\_hdr hdr;

const char \*format;

int count;

struct tm \*tm\_val;

};

我们用arg\_xxx构造了以下参数表

命令行参数：[-a] [-b] [-c] [--scalar=<n>] [-v|--verbose] [-o myfile] <file> [<file>]

构造示例：

struct arg\_lit \*a = arg\_litn("a", NULL, 0, 1, "the -a option");

struct arg\_lit \*b = arg\_litn("b", NULL, 0, 1, "the -b option");

struct arg\_lit \*c = arg\_litn("c", NULL, 0, 1, "the -c option");

struct arg\_int \*scal = arg\_intn(NULL, "scalar", "<n>", 0, 1, "foo value");

struct arg\_lit \*verb = arg\_litn("v", "verbose", 0, 1, "verbose output");

struct arg\_file \*o = arg\_filen("o", NULL,"myfile", 0, 1, "output file");

struct arg\_file \*file = arg\_filen(NULL, NULL, "<file>", 1, 2, "input files");

struct arg\_end \*end = arg\_end(20);

void \*argtable[] = {a, b, c, scal, verb, o, file, end};

arg\_end结构是一种特殊的结构，它不用任何的输入。它主要表示argtable数组结束，也用来存储处理命令行参数时遇到的错误。传递给arg\_end的整数参数是它能存储的最大错误数量。 如20，多出的错误将被丢弃，并给出“错误满”消息。

我们将所有参数表用其构造函数分配成功后可以对数据进行默认值设置。

首先需检测argtable 数组中没有出现错误，我们使用arg\_nullcheck检查 argtable 中的数据。如果结构表在末尾之前返回NULL，则它返回NULLarg\_end

if (arg\_nullcheck(argtable) != 0)

printf("error: insufficient memory\n");

没有错误，现在可以为我们需要输入的参数设置一个默认值，如果命令行有数据输入，输入的数据将替换它们。只需将默认值直接写入arg\_xxx结构体。下面分别设置repeat和outfile参数的默认值3和“-“示例。

repeat->ival[0] = 3;

outfile->filename[0] = "-";

**解析命令行**

使用arg\_parse该函数。

nerrors = arg\_parse(argc, argv, argtable);

nerrors为错误数量，

0为成功

如果成功可以arg\_xxx结构体中判断需要处理的任务。

if (nerrors == 0)

{

int i;

printf("-a = %d\n", a->count);

printf("-b = %d\n", b->count);

printf("-c = %d\n", c->count);

printf("--verbose = %d\n", verb->count);

if (scal->count > 0)

printf(“--scalar=%d\n”, scal->ival[0]);

if (o->count > 0)

printf(“-o %s\n”, o->filename[0]);

for (i = 0; i < file->count; i++)

printf(“file[%d]=%s\n”, i, file->filename[i]);

};

**错误处理**

如果 arg\_parse 函数报告错误，只需调用 arg\_print\_errors 函数即可按错误顺序打印出来。

错误存储在参数表的 arg\_end 结构中。

void arg\_print\_errors(FILE\* fp, struct arg\_end\* end, const char\* progname);

只需向函数传递一个arg\_end结构体的指针和当前程序的函数数名。

progname 可以为NULL。

If (nerrors > 0)

arg\_print\_errors(stdout, end, "myprog");

下面是一个打印错误的例子：

$ ./myprog -x -y -z --scalar=hello --verby

myprog: invalid option "-x"

myprog: invalid option "-y"

myprog: invalid option "-z"

myprog: invalid argument "hello" to option --scalar=<n>

myprog: invalid option "--verby"

myprog: missing option <file>

arg\_parse函数不打印错误消息的原因是，可以多次调用它来使用替代参数表解析命令行，而不会过早显示无关的错误消息。因此，我们可以为那些具有互斥命令行选项集的程序定义单独的参数表，依次尝试每个参数表，直到找到成功的候选者。如果所有参数表都无法满足，那么我们可以选择打印所有参数表的错误消息，或者只显示最接近匹配的错误消息。无论如何，我们控制显示哪些消息。

**显示命令输入语法**

如果您希望在输入时显示帮助信息，可以使用arg\_print\_syntax函数输入时命令行语法。

该函数有两种：

void arg\_print\_syntax(FILE \*fp, void \*\*argtable, const char \*suffix);

void arg\_print\_syntaxv(FILE \*fp, void \*\*argtable, const char \*suffix);

arg\_print\_syntaxv显示更详细的数据,每个参数表条目显示其替代的短选项和长选项，由字符分隔，|后跟其数据类型字符串。例如，

arg\_int0("kKx", "scalar,foo", "<n>", "foo value");

将显示为[-k|-K|-x|--scalar|--foo=<n>]。

arg\_print\_syntax仅显示每个参数表条目的第一个选项来缩写，如 [-k <n>]。并将参数表中的所有短选项连接成标准 GNU 样式（例如：-a -b -c显示为-abc）。

arg\_print\_syntax显示为：

[-abcv] [--scalar=<n>] [-o myfile] <file> [<file>]

arg\_print\_syntaxv显示为：

[-a] [-b] [-c] [--scalar=<n>] [-o myfile] [-v|--verbose] <file> [<file>]

请注意，可选项用方括号“[]”，强制参数则不会。最多只显示三个如“[]”，多出的将会被省略号取代，如“[]...”。

**显示项目标签表**

arg\_print\_glossary函数显示参数表的各个项目包括语法和定义的字符串。

void arg\_print\_glossary(FILE \*fp, void \*\*argtable, const char \*format);

arg\_print\_glossary以下是打印前面示例的结果：

-a the -a option

-b the -b option

-c the -c option

--scalar=<n> foo value

-v, --verbose verbose option

-o myfile output file

<file> input files

**结构数据内存释放**

释放申请的arg\_xxx结构体内存。可以通过 free 来释放，但用arg\_freetable函数更方便。

arg\_freetable(argtable, sizeof(argtable) / sizeof(argtable[0]));

它将遍历其每个元素的并 free 。

注意，第参数sizeof(argtable)/sizeof(argtable[0])仅仅表示 argtable 数组中的元素数量。释放后所有 argtable 数组指针设置为NULL。

#include <argtable3.h>

/\* global arg\_xxx structs \*/

struct arg\_lit \*a, \*b, \*c, \*verb;

struct arg\_int \*scal;

struct arg\_file \*o, \*file;

struct arg\_end \*end;

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

{

/\* the global arg\_xxx structs are initialised within the argtable \*/

void \*argtable[] = {

a = arg\_lit0(“a”, NULL, ”the -a option”),

b = arg\_lit0(“b”, NULL, ”the -b option”),

c = arg\_lit0(“c”, NULL, ”the -c option”),

scal = arg\_int0(NULL, ”scalar”,”<n>”, ”foo value”),

verb = arg\_lit0(“v”, ”verbose, ”verbose output”),

o = arg\_file0(“o”, NULL,”myfile”, ”output file”),

file = arg\_filen(NULL,NULL,”<file>”,1,2, ”input files”),

end = arg\_end(20),

};

...

return 0;

};