# Busybox源码解析

Busybox是一个遵循GPLv2协议的开源项目。Busybox将众多的Linux命令集合进一个很小的可执行程序中。Busybox为各种小型的或嵌入式系统提供了一个比较完全的工具集，包括了Linux中的大部分命令。

## 1应用busybox

用户可从busybox官网https://busybox.net/下载源码压缩包，作者下载的是busybox-1.32.0。解压后源码根目录下包含以下主要文件夹及文件：

|  |  |  |
| --- | --- | --- |
| 目录 | 描述 | 备注 |
| applets | 主要是实现applets框架的文件 |  |
| archival | 与压缩有关的命令源文件，例如：bzip2、gzip等 |  |
| configs | 自带的一些默认配置文件 |  |
| console-tools | 与控制台相关的一些命令，例如：setconsole等 |  |
| coreutils | 常用核心命令，例如：cat、rm等 |  |
| editors | 常用编辑命令，例如：vi、diff等 |  |
| findutils | 用于查找的命令，例如：find、grep等 |  |
| include | 头文件 |  |
| init | init进程的实现源文件 |  |
| libbb | busybox中各命令使用的公共函数，appletlib.c中包含main()函数 |  |
| networking | 与网络相关的命令，例如：telnetl、arp等 |  |
| shell | 与shell相关的实现，例如：ash、msh等 |  |
| scripts | 配置、构建过程中使用的脚本程序 |  |
| util-linux | Linux下常用的命令，主要是与文件系统相关的，例如：mkfs\_ext2等 |  |
| Config.in | 顶层配置文件 |  |
| Makefile | 顶层构建文件 |  |

Busybox源码的配置、构建与Linux内核相似，在源码根目录下存在Makefile文件，以及Config.in配置文件（如同Linux内核源码中的Kbuild文件）。

Busybox配置文件名称为Config.in，相当于Linux内核中的Kconfig文件。构建文件名称为Kbuild，相当于Linux内核中的Makefile文件。注意busybox中的配置、构建文件是动态生成的，而不是静态编写的，这与Linux内核不同。

Busybox源码解压后，各文件夹下存在Config.src和Kbuild.src文件，Config.src是生成Config.in文件的模板（往里面插入内容生成Config.in文件），Kbuild.src是生成Kbuild文件的模板。

在对busybox源码进行配置、构建前，可以通过修改源码树根目录下的Makefile文件，设置交叉编译工具前缀和体系结构，如下图所示：

busybox

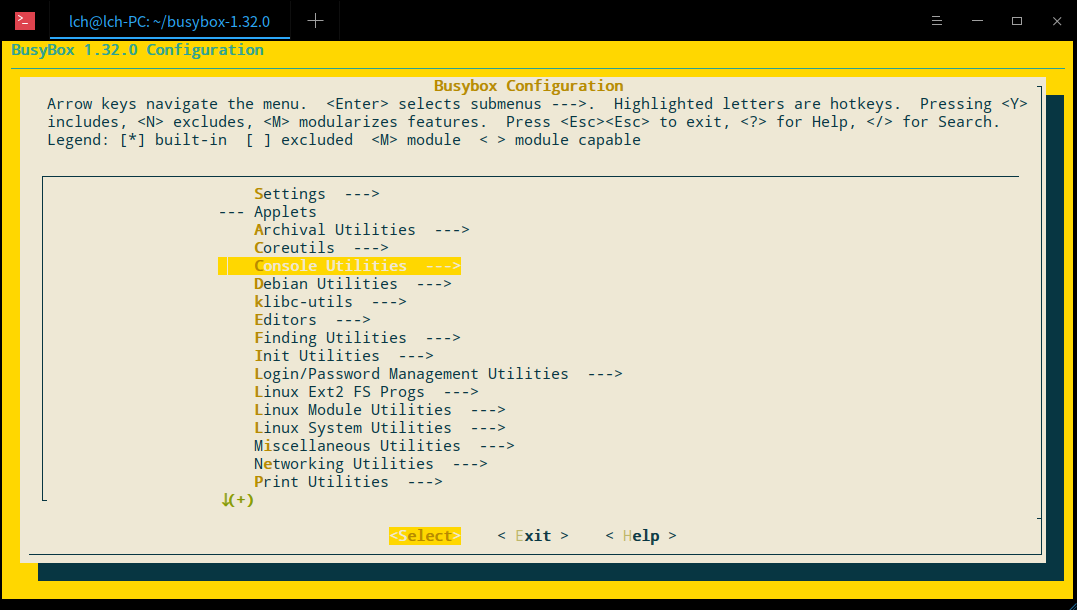
busybox0

注：交叉编译工具前缀可以在配置选项中指定，也可以在构建时通过CROSS\_COMPILE命令行参数指定，三者任选其一即可。但是，如果有两个以上同时设置了，则Makefile文件内的设置具有最高优先权。

Busybox源码配置、构建步骤如下：

**（1）make menuconfig**

配置busybox，用于选择/设置配置选项，配置界面如下：



Settings菜单下主要包含busybox特性选项、构建选项（编译链接选项）、安装路径选项、调试选项等，Applets下各菜单选择构建的命令。

在配置操作中将自动生成各配置文件Config.in和构建文件Kbuild，以及以下头文件：

●**include/applets.h** /\*以include/applets.src.h为模板生成的头文件，包含bb\_applet结构体实例数组等，

/\*用于生成include/applet\_tables.h 头文件，见下一小节\*/

●**.config** /\*头文件，包含由配置选项生成的宏定义\*/

●**include/autoconf.h** /\*包含依据.config生成的宏定义\*/

以上头文件将应用于源代码的编译、链接，更详细的信息见下一小节。

**（2）make**

构建操作，对源代码进行编译、链接，最终在源代码树根目录下生成一个名称为**“busybox”**的可执行文件。

构建操作除了生成busybox可执行目标文件外，还将成生以下头文件或主机可执行目标文件：

●生成applets/usage主机工具（用于输出命令定义的帮助信息）。

●生成applets/applet\_tables主机工具（用于生成include/applet\_tables.h头文件）。

●由applet\_tables主机工具依据/include/applets.h头文件生成**include/applet\_tables.h**头文件。

●生成applets/usage\_pod主机工具（用于输出命令定义的帮助信息）。

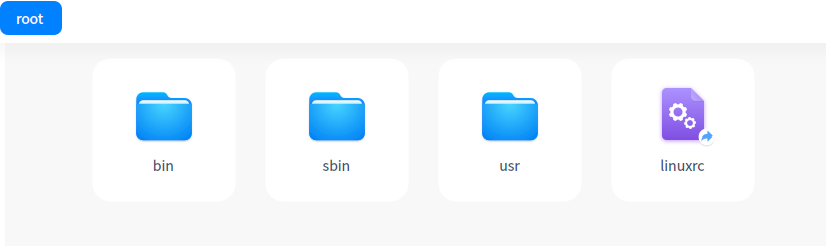
注意：在源代码树applets目录下生成的applet\_tables、usage、usage\_pod是在主机上运行的可执行目标文件，而不是在目标机上运行的。

**（3）make install**

Busybox编译链接后生成一个单一的可执行文件，安装操作将为各命令创建到busybox可执行目标文件的符号链接，符号链接名称为命令名称，如rm、cp等。

在Settings配置菜单下可设置安装路径，保存在CONFIG\_PREFIX配置选项中。

例如：作者设置的路径为/home/lch/root，执行make install命令生成的文件夹如下所示：



bin、sbin、usr文件夹下的命令，包括linuxrc文件都是到bin/busybox可执行目标文件的符号链接。

在以上目录中再增加其它所需的文件（夹），即可制作出目标机的根文件系统。将此根文件系统复制到目标机存储介质（由目标机挂载为根文件系统），即可使用busybox命令了。

在目标机中，可以通过两种方式来执行busybox命令：

（1）直接使用busybox，例如：busybox ls，执行ls命令，ls作为busybox的参数。

（2）使用符号链接，例如：ls，执行ls命令。

## 2 Busybox框架

前一小节简要介绍了busybox的应用，这一小节先介绍busybox配置、构建机制，然后介绍busybox如何执行各命令，即main()函数的实现。

### 2.1配置与构建

Busybox的配置、构建机制与Linux内核类似，但又有所不同，主要区别是其配置、构建文件是根据源文件内容动态生成的，而不是静态编写的。

配置文件Config.in以Config.src文件为模板生成，构建文件Kbuild以Kbuild.src文件为模板生成。配置、构建文件内容来源于busybox源文件。

#### ■配置

下面以coreutils/link.c文件为例，看看文件内与配置和构建文件相关的内容：

/\*以下内容写入动态生成的配置文件coreutils/Config.in\*/

//config:config LINK

//config: bool "link (3.2 kb)"

//config: default y

//config: help

//config: link creates hard links between files.

/\*以下内容写入动态生成的include/applets.h头文件\*/

//applet:IF\_LINK(APPLET\_NOFORK(link, link, BB\_DIR\_BIN, BB\_SUID\_DROP, link))

/\*以下内容写入动态生成的构建文件coreutils/Kbuild\*/

//kbuild:lib-$(CONFIG\_LINK) += link.o

/\*以下内容写入动态生成的include/usage.h头文件，

\*由主机工具applets/usage、applets/usage\_pod输出的信息。

\*同时保存在动态生成的include/usage\_compressed.h头文件，用于在目标机中输出命令帮助信息。\*/

//usage:#define link\_trivial\_usage

//usage: "FILE LINK"

//usage:#define link\_full\_usage "\n\n"

//usage: "Create hard LINK to FILE"

#include "libbb.h"

/\*link命令执行函数\*/

int link\_main(int argc, char \*\*argv) MAIN\_EXTERNALLY\_VISIBLE;

int link\_main(int argc UNUSED\_PARAM, char \*\*argv)

{

...

}

以上文件内容中“//config:”开头的行，其后的内容将写入动态生成的coreutils/Config.in文件，即增加LINK配置选项。

“//kbuild:”开头的行，其后的内容将写入coreutils/Kbuild文件，表示如果选择了LINK配置选项，则编译链接link.c文件。

“//applet:”开头的行，其后的内容将写入动态生成的include/applets.h头文件。

“//usage:”开头的行，其后的内容将写入动态生成的include/usage.h头文件，用于输出帮助信息。

配置、构建文件读者应该都比较熟悉了，这里不再介绍了，下面介绍一个include/applets.h头文件。

include/applets.h头文件是以include/applets.src.h头文件为模板动态生成的文件，主要就是将源文件中“//applet:”标记开头的行写入到applets.src.h头文件，从而生成applets.h头文件。

applets.h头文件内容简列如下：

/\*PROTOTYPES在libbb/appletlib.c文件内定义\*/

**#if defined(PROTOTYPES)**

# define APPLET(name,l,s) int **name##\_main(int argc, char \*\*argv)** MAIN\_EXTERNALLY\_VISIBLE;

# define APPLET\_ODDNAME(name,main,l,s,help) int main##\_main(int argc, char \*\*argv) \ MAIN\_EXTERNALLY\_VISIBLE;

# define APPLET\_NOEXEC(name,main,l,s,help) int main##\_main(int argc, char \*\*argv) \

MAIN\_EXTERNALLY\_VISIBLE;

# define APPLET\_NOFORK(name,main,l,s,help) int main##\_main(int argc, char \*\*argv) \

MAIN\_EXTERNALLY\_VISIBLE;

# define APPLET\_SCRIPTED(name,main,l,s,help)

**#elif defined(NAME\_MAIN)**

# define APPLET(name,l,s) name name##\_main

# define APPLET\_ODDNAME(name,main,l,s,help) name main##\_main

# define APPLET\_NOEXEC(name,main,l,s,help) name main##\_main

# define APPLET\_NOFORK(name,main,l,s,help) name main##\_main

# define APPLET\_SCRIPTED(name,main,l,s,help) name scripted\_main

/\*MAKE\_USAGE在applets/usage.c、applets/usage\_pod.c文件内定义\*/

**#elif defined(MAKE\_USAGE) && ENABLE\_FEATURE\_VERBOSE\_USAGE**

# define APPLET(name,l,s) MAKE\_USAGE(#name, name##\_trivial\_usage name##\_full\_usage)

# define APPLET\_ODDNAME(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage \

help##\_full\_usage)

# define APPLET\_NOEXEC(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage \

help##\_full\_usage)

# define APPLET\_NOFORK(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage \

help##\_full\_usage)

# define APPLET\_SCRIPTED(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage \

help##\_full\_usage)

/\*MAKE\_USAGE在applets/usage.c、applets/usage\_pod.c文件定义\*/

**#elif defined(MAKE\_USAGE) && !ENABLE\_FEATURE\_VERBOSE\_USAGE**

# define APPLET(name,l,s) MAKE\_USAGE(#name, name##\_trivial\_usage)

# define APPLET\_ODDNAME(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage)

# define APPLET\_NOEXEC(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage)

# define APPLET\_NOFORK(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage)

# define APPLET\_SCRIPTED(name,main,l,s,help) MAKE\_USAGE(#name, help##\_trivial\_usage)

**#elif defined(MAKE\_LINKS)**

# define APPLET(name,l,c) LINK l name

# define APPLET\_ODDNAME(name,main,l,s,help) LINK l name

# define APPLET\_NOEXEC(name,main,l,s,help) LINK l name

# define APPLET\_NOFORK(name,main,l,s,help) LINK l name

# define APPLET\_SCRIPTED(name,main,l,s,help) LINK l name

**#elif defined(MAKE\_SUID)**

# define APPLET(name,l,s) SUID s l name

# define APPLET\_ODDNAME(name,main,l,s,help) SUID s l name

# define APPLET\_NOEXEC(name,main,l,s,help) SUID s l name

# define APPLET\_NOFORK(name,main,l,s,help) SUID s l name

# define APPLET\_SCRIPTED(name,main,l,s,help) SUID s l name

**#elif defined(MAKE\_SCRIPTS)**

# define APPLET(name,l,s)

# define APPLET\_ODDNAME(name,main,l,s,help)

# define APPLET\_NOEXEC(name,main,l,s,help)

# define APPLET\_NOFORK(name,main,l,s,help)

# define APPLET\_SCRIPTED(name,main,l,s,help) SCRIPT name

**#else**  /\*bb\_applet实例数组\*/

static struct **bb\_applet applets[]** = { /\* name, main, location, need\_suid \*/

# define APPLET(name,l,s) { #name, #name, l, s },

# define APPLET\_ODDNAME(name,main,l,s,help) { #name, #main, l, s },

# define APPLET\_NOEXEC(name,main,l,s,help) { #name, #main, l, s, 1 }, /\*最后成员为数字1\*/

# define APPLET\_NOFORK(name,main,l,s,help) { #name, #main, l, s, 1, 1 }, /\*最后2个成员为数字1\*/

# define APPLET\_SCRIPTED(name,main,l,s,help) { #name, #main, l, s },

**#endif**

#if ENABLE\_INSTALL\_NO\_USR

# define BB\_DIR\_USR\_BIN BB\_DIR\_BIN

# define BB\_DIR\_USR\_SBIN BB\_DIR\_SBIN

#endif

/\*以下是头文件实际的内容\*/

**IF\_AR(APPLET(ar, BB\_DIR\_USR\_BIN, BB\_SUID\_DROP))**

**IF\_UNCOMPRESS(APPLET(uncompress, BB\_DIR\_BIN, BB\_SUID\_DROP))**

**IF\_GUNZIP(APPLET(gunzip, BB\_DIR\_BIN, BB\_SUID\_DROP))**

**IF\_ZCAT(APPLET\_ODDNAME(zcat, gunzip, BB\_DIR\_BIN, BB\_SUID\_DROP, zcat))**

**...**

#if !defined(PROTOTYPES) && !defined(NAME\_MAIN) && !defined(MAKE\_USAGE) \

&& !defined(MAKE\_LINKS) && !defined(MAKE\_SUID)

};

#endif

...

include/applets.h头文件的实际内容是由形如IF\_AR()的宏定义的。IF\_*XXX*()宏定义在动态生成的头文件include/autoconf.h中。

IF\_*XXX*()宏的参数都是形如APPLET\_*XXX*()的宏，这些宏定义在include/applets.h头文件。但是根据不同的宏定义（条件）APPLET\_*XXX*()宏将被解释成不同的内容。

Busybox源代码中在包含include/applets.h头文件前，会先定义PROTOTYPES、NAME\_MAIN等宏，在包含头文件代码后将会取消PROTOTYPES、NAME\_MAIN等宏的定义。也就是说，include/applets.h头文件在不同的位置可表示不同的内容。

例如，假设源代码在定义了**PROTOTYPES**宏后再包含include/applets.h头文件，则头文件内容表示工具执行函数指针列表，执行函数名称通常为***nam*e\_main(int argc, char \*\*argv)**，name为工具名称，如link。

如果包含/include/applets.h头文件前什么宏都没定义，applets.h头文件内容为**bb\_applet**结构体实例数组**applets[]**。

在busybox源码树根目录下执行make menuconfig命令，完成配置退出后，终端显示内容简列如下：

lch@lch-PC:~/busybox-1.32.0$ make menuconfig

HOSTCC scripts/basic/fixdep

HOSTCC scripts/basic/split-include

HOSTCC scripts/basic/docproc

GEN include/applets.h /\*生成include/applets.h头文件，以include/applets.src.h头文件为模板\*/

GEN **include/usage.h** /\*生成include/usage.h头文件，以include/usage.src.h为模板\*/

GEN libpwdgrp/Kbuild /\*以下是生成各目录下的Config.in、Kbuild文件\*/

GEN util-linux/Kbuild

GEN util-linux/Config.in

GEN util-linux/volume\_id/Kbuild

GEN util-linux/volume\_id/Config.in

GEN archival/Kbuild

GEN archival/Config.in

GEN archival/libarchive/Kbuild

GEN coreutils/Kbuild

GEN coreutils/Config.in

GEN coreutils/libcoreutils/Kbuild

...

HOSTCC scripts/kconfig/conf.o

HOSTCC scripts/kconfig/kxgettext.o

HOSTCC scripts/kconfig/mconf.o

SHIPPED scripts/kconfig/zconf.tab.c

SHIPPED scripts/kconfig/lex.zconf.c

SHIPPED scripts/kconfig/zconf.hash.c

HOSTCC scripts/kconfig/zconf.tab.o

HOSTLD **scripts/kconfig/mconf**  /\*生成脚本程序\*/

HOSTCC scripts/kconfig/lxdialog/checklist.o

HOSTCC scripts/kconfig/lxdialog/inputbox.o

HOSTCC scripts/kconfig/lxdialog/lxdialog.o

HOSTCC scripts/kconfig/lxdialog/menubox.o

HOSTCC scripts/kconfig/lxdialog/msgbox.o

HOSTCC scripts/kconfig/lxdialog/textbox.o

HOSTCC scripts/kconfig/lxdialog/util.o

HOSTCC scripts/kconfig/lxdialog/yesno.o

HOSTLD scripts/kconfig/lxdialog/lxdialog

scripts/kconfig/mconf Config.in

#

# using defaults found in /dev/null

#

\*\*\* End of configuration.

\*\*\* Execute 'make' to build the project or try 'make help'.

配置完成后将在根目录下生成.config头文件，并依此生成include/autoconf.h头文件，autoconf.h内容简列如下：

#define AUTOCONF\_TIMESTAMP "2021-06-26 08:14:39 CST"

#define CONFIG\_HAVE\_DOT\_CONFIG 1

#define ENABLE\_HAVE\_DOT\_CONFIG 1

#ifdef MAKE\_SUID

# define IF\_HAVE\_DOT\_CONFIG(...) \_\_VA\_ARGS\_\_ "CONFIG\_HAVE\_DOT\_CONFIG"

#else

# define IF\_HAVE\_DOT\_CONFIG(...) \_\_VA\_ARGS\_\_

#endif

#define IF\_NOT\_HAVE\_DOT\_CONFIG(...)

/\*

\* Settings

\*/

#define CONFIG\_DESKTOP 1

#define ENABLE\_DESKTOP 1

#ifdef MAKE\_SUID

# define IF\_DESKTOP(...) \_\_VA\_ARGS\_\_ "CONFIG\_DESKTOP"

#else

# define IF\_DESKTOP(...) \_\_VA\_ARGS\_\_

#endif

...

在源文件中使用的如同ENABLE\_DESKTOP的宏，以及在include/applets.h头文件中使用的**IF\_XXX()**宏都在include/autoconf.h头文件内定义。

小结：配置操作执行结果简列如下：

（1）生成include/applets.h头文件（以applets.src.h为模板生成）。

（2）生成include/usage.h头文件（以include/usage.src.h为模板，保存命令定义的帮助信息）。

（3）生成各目录下Config.in文件（以Config.src为模板）。

（4）生成各目录下Kbuild文件（以Kbuild.src为模板）。

（5）根据配置选项生成**.config**头文件。

（6）依据.config头文件生成**include/autoconf.h**头文件。

#### ■构建

配置完成后，用户可执行make命令，执行构建操作，终端输出结果简列如下：

lch@lch-PC:~/busybox-1.32.0$ make

SPLIT include/autoconf.h -> include/config/\*

/\*include/autoconf.h头文件内容写入include/config/目录下\*/

GEN include/bbconfigopts.h

GEN include/common\_bufsiz.h

GEN include/embedded\_scripts.h

HOSTCC **applets/usage**  /\*生成主机可执行目标文件usage，用于输出命令信息\*/

GEN **include/usage\_compressed.h** /\*命令帮助信息，在目标机中输出的信息\*/

HOSTCC applets/applet\_tables /\*生成主机可执行目标文件applet\_tables\*/

GEN **include/applet\_tables.h** include/NUM\_APPLETS.h

/\*由applet\_tables生成applet\_tables.h头文件\*/

GEN **include/applet\_tables.h** include/NUM\_APPLETS.h

CC applets/applets.o

LD applets/built-in.o

HOSTCC **applets/usage\_pod** /\*生成主机可执行目标文件usage\_pod，用于输出命令信息\*/

LD archival/built-in.o /\*以下是编译链接各源文件\*/

CC archival/bbunzip.o

CC archival/bzip2.o

CC archival/cpio.o

... /\*最终生成busybox可执行目标文件\*/

构建操作中首先会生成3个在主机上的可执行目标文件，分别是usage、applet\_tables、usage\_pod，这3个目标文件及其源文件都位于applets/目录下。applet\_tables可执行目标文件，在构建时将被执行，它利用/include/applets.h头文件生成头文件**include/applet\_tables.h**。

applet\_tables.h头文件是busybox编译时真正使用的非常重要的头文件，前面介绍的applets.src.h、applets.h头文件都是用于生成applet\_tables.h头文件的，下面重点介绍一下此头文件的内容。

applet\_tables可执行目标文件由applets/applet\_tables.c源文件生成，在此文件内定义了bb\_applet结构体，如下所示：

struct bb\_applet {

const char \***name**; /\*命令名称（可能是老式的名称）\*/

const char \*main; /\*执行函数中使用的命令名称（新式的名称）\*/

enum bb\_install\_loc\_t **install\_loc**; /\*命令安装目录\*/

enum bb\_suid\_t **need\_suid**; /\*运行命令时需要的权限\*/

unsigned char noexec;

unsigned char nofork;

};

bb\_applet结构体主要成员简介如下：

●**name：**命令名称（可能是老式的名称），其执行函数通常为*name*\_main()。

●**main**：执行函数为*main*\_main()，老式的命令名称保存在name中，新式的名称保存在main中。

●**install\_loc：**命令安装的目录，由枚举类型定义（include/applet\_metadata.h）：

typedef enum bb\_install\_loc\_t {

BB\_DIR\_ROOT = 0, /\*根目录，0\*/

BB\_DIR\_BIN, /\*/bin目录，1\*/

BB\_DIR\_SBIN, /\*/sbin目录，2\*/

#if ENABLE\_INSTALL\_NO\_USR

BB\_DIR\_USR\_BIN = BB\_DIR\_BIN,

BB\_DIR\_USR\_SBIN = BB\_DIR\_SBIN,

#else

BB\_DIR\_USR\_BIN, /\*/usr/bin目录，3\*/

BB\_DIR\_USR\_SBIN, /\*/usr/sbin目录，4\*/

#endif

} bb\_install\_loc\_t;

●**need\_suid：**表示执行命令是否需要超级用户权限，由枚举类型定义（include/applet\_metadata.h）：

typedef enum bb\_suid\_t {

BB\_SUID\_DROP = 0, /\*执行命令前降低进程权限，不需要超级用户权限\*/

BB\_SUID\_MAYBE, /\*可能需要\*/

BB\_SUID\_REQUIRE /\*需要超级用户权限\*/

} bb\_suid\_t;

●**noexec、nofork：**在执行命令时，shell可以不需要为其创建进程，而是在shell中（或其它命令中）直接以函数调用的方式执行命令。noexec、nofork命令执行函数需满足一些限制条件，如不能使用共享的全局变量等，详见busybox说明文档docs/nofork\_noexec.txt。

applets/applet\_tables.c源文件中将包含include/applets.h头文件，且其内容为bb\_applet结构体实例数组。在构建操作中将运行applet\_tables可执行目标文件，依据bb\_applet实例数组生成include/applet\_tables.h头文件。

applet\_tables.h头文件内容简列如下：

#define NUM\_APPLETS 390 /\*命令数量\*/

#define KNOWN\_APPNAME\_OFFSETS 8

/\*applet\_nameofs[]数组项数为7，其值在applet\_tables.c中定义\*/

/\*applet\_nameofs[]数组项表示各段起始名称字符串在applet\_names[]的偏移量，用于执行快速查找\*/

const uint16\_t applet\_nameofs[] ALIGN2 = { /\*数组项数为(KNOWN\_APPNAME\_OFFSETS-1)\*/

299,

648,

980,

1304,

1652,

2027,

2349,

};

const char **applet\_names[]** ALIGN1 = "" /\*命令名称字符数组（名称字符串），按字母顺序排序\*/

"[" "\0" /\*编号0\*/

"[[" "\0" /\*编号1，依此类推\*/

"acpid" "\0"

"add-shell" "\0"

"addgroup" "\0"

"adduser" "\0"

"adjtimex" "\0"

"arch" "\0"

...

;

/\*命令编号\*/

#define APPLET\_NO\_acpid 2

#define APPLET\_NO\_addgroup 4

#define APPLET\_NO\_adduser 5

#define APPLET\_NO\_adjtimex 6

#define APPLET\_NO\_arch 7

#define APPLET\_NO\_arp 8

#define APPLET\_NO\_arping 9

#define APPLET\_NO\_ash 10

#define APPLET\_NO\_awk 11

...

/\*命令执行函数指针数组\*/

#ifndef SKIP\_applet\_main

int (\*const **applet\_main[]**)(int argc, char \*\*argv) = {

test\_main, /\*编号0\*/

test\_main, /\*编号1，依此类推\*/

acpid\_main,

add\_remove\_shell\_main,

addgroup\_main,

adduser\_main,

adjtimex\_main,

uname\_main,

arp\_main,

...

};

#endif

/\*bb\_applet结构体need\_suid成员值，每个成员值由2个比特位表示。\*/

const uint8\_t **applet\_suid[]** ALIGN1 = {

0x00,

0x00,

0x00,

0x00,

0x00,

...

};

/\*bb\_applet结构体install\_loc成员值，每个字节表示2个成员值，每个成员值由4比特表示\*/

const uint8\_t **applet\_install\_loc[]** ALIGN1 = {

0x33,

0x42,

0x44,

0x12,

0x42,

0x31,

0x31,

...

};

applet\_tables.h头文件中主要包含以下内容：

●**applet\_names[]：**命令名称字符数组（名称字符串）。applet\_tables保证各名称字符串是按字母顺序排序的。

●**命令编号列表**：命令编号与applet\_names[]中名称字符串是一一对应的。

●**applet\_main[]：**命令执行函数指针数组。applet\_main[]数组与applet\_names[]中名称字符串、命令编号是一一对应的。

**●applet\_suid[]：**表示执行命令是否需要超级用户权限，每个命令由2个比特位表示。

●**applet\_install\_loc[]：**命令安装位置，每个命令由4个比特位（实际使用了3位）表示。

在applet\_tables.h头文件中，还需要说明一下KNOWN\_APPNAME\_OFFSETS常量和applet\_nameofs[]数组。为了方便查找，busybox将命令按编号进行分段，KNOWN\_APPNAME\_OFFSETS常量表示分段的段数。例如，如上面的头文件所示，命令数量为390，KNOWN\_APPNAME\_OFFSETS常量为8，各段中命令起始编号为i\*390/8（i为段编号，除为整除），编号分段如下图所示：



applet\_nameofs[]数组项数为KNOWN\_APPNAME\_OFFSETS-1=7。applet\_nameofs[]数组项表示的是各段起始命令名称在applet\_names[]字符数组中的偏移量（从第一段开始，第0段始终为0，不需要记录）。例如，上图中applet\_nameofs[0]记录的是48号命令名称在applet\_names[]字符数组中的偏移量。以上分段将在由名称查找工具编号的函数中使用，详见下文。

在busybox的main()函数中，依命令名称在applet\_names[]数组中查找匹配的项，获取命令编号，然后调用applet\_main[]数组中对应的执行函数，执行命令。

小结：构建操作最后在busybox源码根目录下生成单一的可执行目标文件（目标机）busybox，构建操作执行结果简列如下：

（1）生成applets/usage主机工具（用于输出工具定义的帮助信息）。

（2）生成include/usage\_compressed.h头文件，保存命令帮助信息，在目标机中输出的信息。

（3）生成applets/applet\_tables主机工具（用于生成include/applet\_tables.h头文件）。

（4）由applet\_tables主机工具依据include/applets.h头文件生成**include/applet\_tables.h**头文件。

（5）生成applets/usage\_pod主机工具（用于输出工具定义的帮助信息）。

（6）编译链接源文件，最终生成**busybox**可执行目标文件。

### 2.2 main()函数

Busybox最终生成一个用户可执行目标文件，用户程序的入口函数一般为main()，下面介绍busybox中main()函数的定义。

如果busybox在配置时没有选择BUILD\_LIBBUSYBOX选项（构建选项，将busybox构建成一个动态库，目前不支持），在生成的include/autoconf.h头文件中，将ENABLE\_BUILD\_LIBBUSYBOX宏定义为0。此时，busybox的入口函数mian()为libbb/appletlib.c中定义的main()函数，而不是applets/applets.c中定义的main()函数。下面以没有选择BUILD\_LIBBUSYBOX选项为例，介绍main()函数的实现。

用户可通过以下形式执行busybox中命令：

**$busybox ls**  /\*bb\_basename()函数返回busybox\*/

**$/bin/busybox ls**  /\*bb\_basename()函数返回busybox\*/

**$/bin/ls**  /\*bb\_basename()函数返回ls\*/

**$ls**  /\*bb\_basename()函数返回ls\*/

用户可直接通过命令名称执行命令，也可以运行busybox目标文件，将命令作为其参数。用户在shell中输入以上命令时，shell将读取用户输入的命令行参数，并传递给main()函数。

例如，用户输入“**$/bin/busybox ls”**命令，shell向main()函数传递的命令行参数如下：



argv指向指针数组，每一项指向一个命令行参数。

Busybox中main()函数相当于一个分配器，它从命令行参数从获取命令名称，在applet\_tables.h头文件applet\_names[]命令名称列表（按字母顺序排序）中查找匹配的项，获取命令编号，然后依编号调用命令执行函数指针applet\_main[]数组中的执行函数，执行命令。

Busybox中bb\_basename()函数用于获取参数（字符串）中最后‘/’字符后的分量指针，如果不存在‘/’字符，则直接返回参数指针。这个函数用于从命令行参数中获取用户输入的命令名称。

Busybox中main()函数定义如下（libbb/appletlib.c）：

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

{

...

#if defined(SINGLE\_APPLET\_MAIN)

... /\*配置时只选择了一个命令\*/

#elif !ENABLE\_BUSYBOX && NUM\_APPLETS == 0

... /\*配置时一个命令都有没选\*/

#else /\*正常的处理\*/

/\*如果配置选择了FEATURE\_INDIVIDUAL选项（依赖于BUILD\_LIBBUSYBOX选项），

\*则为每个工具创建一个二进制文件。

\*如果没有选择FEATURE\_INDIVIDUAL选项，ENABLE\_FEATURE\_INDIVIDUAL宏为0，

\*lbb\_prepare()函数则基本不用做什么工作（需设置全局指针applet\_name = applet）。

\*/

lbb\_prepare("busybox" IF\_FEATURE\_INDIVIDUAL(, argv));

/\*选择BUSYBOX选项（默认选择），则定义ENABLE\_BUSYBOX宏，表示生成busybox命令，

\*可运行如下命令：

\*$ busybox

\*此命令将输出busybox帮助信息，以及命令列表等。

\*/

# if !ENABLE\_BUSYBOX

if (argv[1] && is\_prefixed\_with(bb\_basename(argv[0]), "busybox"))

argv++;

# endif

**applet\_name** = argv[0]; /\*命令名称赋予全局指针**applet\_name**\*/

if (applet\_name[0] == '-') /\*跳过名称前的'-'字符\*/

applet\_name++;

applet\_name = **bb\_basename**(applet\_name); /\*/lib/get\_last\_path\_component.c\*/

/\*获取命令名称，赋予全局指针**applet\_name**\*/

if (ENABLE\_FEATURE\_SH\_STANDALONE

/\*依赖FEATURE\_SH\_STANDALONE选项（/shell/Config.in），默认不选\*/

|| ENABLE\_FEATURE\_PREFER\_APPLETS

/\*依赖FEATURE\_PREFER\_APPLETS选项（/Config.in），默认不选\*/

|| !BB\_MMU /\*!NOMMU\*/

) {

if (NUM\_APPLETS > 1)

set\_task\_comm(applet\_name); /\*/libbb/vfork\_daemon\_rexec.c，需定义“\_\_linux\_\_”宏\*/

}

parse\_config\_file(); /\*/libbb/appletlib.c\*/

**run\_applet\_and\_exit**(applet\_name, argv); /\*调用命令执行函数\*/

#endif

}

mian()函数从命令行参数中获取命令名称及其它参数，调用run\_applet\_and\_exit(applet\_name, argv)函数执行命令，applet\_name参数指向命令名称。

run\_applet\_and\_exit()函数定义如下（libbb/appletlib.c）。

static NORETURN void run\_applet\_and\_exit(const char \*name, char \*\*argv)

{

# if ENABLE\_BUSYBOX

if (is\_prefixed\_with(name, "busybox")) /\*以“$busybox ls”形式执行命令\*/

exit(**busybox\_main**(/\*unused:\*/ 0, argv));

/\*需要处理$busybox、$busybox --help等命令，对于一般的命令调用run\_applet\_and\_exit()函数\*/

# endif

/\*以“$ls”形式执行命令\*/

# if NUM\_APPLETS > 0

{

int applet = **find\_applet\_by\_name**(name); /\*由名称查找命令编号，返回编号值\*/

if (applet >= 0)

**run\_applet\_no\_and\_exit**(applet, name, argv);

/\*调用命令执行函数，进程退出，不再返回\*/

}

# endif

/\*执行到这里说明出错了\*/

full\_write2\_str(applet\_name);

full\_write2\_str(": applet not found\n");

exit(127); /\*进程退出\*/

}

run\_applet\_and\_exit()函数中需要判断用户是以“$busybox ls”形式还是“$ls”形式执行命令，前者由busybox\_main()函数处理，对于后者先调用find\_applet\_by\_name()函数以名称查找命令编号，然后调用函数run\_applet\_no\_and\_exit()调用命令执行函数。

busybox\_main()函数处理过程类似，只不过需要处理“$busybox”、“$busybox --help”等特殊的命令形式，最终回调run\_applet\_and\_exit()函数执行普通的命令。

#### ■由名称查找命令

前面已经介绍了busybox对命令编号的分段，各段起始命令编号为（i为段号）：

i\*NUM\_APPLETS / KNOWN\_APPNAME\_OFFSETS /\*整除，NUM\_APPLETS 表示总的工具数量\*/

applet\_nameofs[]数组表示各段起始命令名称在applet\_names[]字符数组中的偏移量。

run\_applet\_and\_exit()函数中调用find\_applet\_by\_name()函数，以命令名称在applet\_names[]字符数组中按段搜匹配字符串，返回命令编号。

find\_applet\_by\_name()函数定义如下（libbb/appletlib.c）：

int FAST\_FUNC find\_applet\_by\_name(const char \*name)

{

unsigned i, max;

int j;

const char \*p;

...

p = applet\_names; /\*指向名称字符数组开始处\*/

i = 0;

#if KNOWN\_APPNAME\_OFFSETS <= 0

max = NUM\_APPLETS;

#else

max = NUM\_APPLETS \* KNOWN\_APPNAME\_OFFSETS;

for (j = ARRAY\_SIZE(applet\_nameofs)-1; j >= 0; j--) {

/\*名称字符数组从后往前，确定命令所在段\*/

const char \*pp = applet\_names + applet\_nameofs[j]; /\*段内起始命令，名称字符所在位置\*/

if (**strcmp(name, pp) >= 0**) {

**p = pp**; /\*命令位于pp指示的段内\*/

i = max - NUM\_APPLETS;

break;

}

max -= NUM\_APPLETS;

}

max /= (unsigned)KNOWN\_APPNAME\_OFFSETS; /\*命令所在段的下一段的起始命令编号\*/

i /= (unsigned)KNOWN\_APPNAME\_OFFSETS; /\*命令所在段的起始命令编号\*/

#endif

#if 0

...

#else

while (i < max) { /\*i表示命令编号所在段的起始命令编号，max表示下一段的起始编号\*/

char ch;

j = 0;

/\*比对段内名称字符（串）与参数name指示的名称字符（串）是否相等\*/

while ((ch = \*p) == name[j]) {

if (ch == '\0') {

**return i**; /\*返回命令编号\*/

}

p++;

j++;

}

/\*段内下一个名称字符（串）\*/

while (ch != '\0')

ch = \*++p;

p++;

**i++;** /\*编号加1\*/

}

return -1;

#endif

}

find\_applet\_by\_name()函数从编号分段的末尾段往前查找，确定name名称命令所在的段（段内名称是按字母大小顺序排列的，因此可以比较）。局部变量i表示name所在段的起始命令编号，max表示下一段的起始命令编号（或命令总数量），函数在i表示起始编号的段内依次搜索名称字符串（每搜索一个名称字符串i加1，表示命令编号），并与name比对，若相等则返回此名称字符串编号值，即i的当前值。

#### ■执行命令

run\_applet\_and\_exit()函数在获取命令编号后，调用run\_applet\_no\_and\_exit()函数执行命令，函数定义如下（libbb/appletlib.c）：

void FAST\_FUNC run\_applet\_no\_and\_exit(int applet\_no, const char \*name, char \*\*argv)

/\*applet\_no：命令编号，name：命令名称，argv：命令行参数，argv[0]指向命令名称字符串\*/

{

int argc = string\_array\_len(argv); /\*命令行参数数量\*/

**applet\_name** = name; /\*命令名称赋予全局指针applet\_name\*/

if (1

# if defined APPLET\_NO\_test

&& applet\_no != APPLET\_NO\_test

# endif

# if defined APPLET\_NO\_true

&& applet\_no != APPLET\_NO\_true

# endif

# if defined APPLET\_NO\_false

&& applet\_no != APPLET\_NO\_false

# endif

) {

if (argc == 2 && strcmp(argv[1], "--help") == 0) { /\*输出命令帮助信息\*/

xfunc\_error\_retval = 0;

**bb\_show\_usage()**; /\*libbb/appletlib.c\*/

/\*命令帮助信息保存在动态生成的include/usage\_compressed.h头文件，

\*每个命令的帮助信息由一个字符串表示（按编号依次保存），函数内根据applet\_name

\*指示的命令名称查找命令编号，以此编号搜索命令帮助信息字符串，并输出。\*/

}

}

if (ENABLE\_FEATURE\_SUID)

check\_suid(applet\_no); /\*权限检查，libbb/appletlib.c\*/

xfunc\_error\_retval = **applet\_main**[**applet\_no**](argc, argv); /\*调用命令执行函数\*/

xfunc\_die(); /\*退出进程，/libbb/xfunc\_die.c\*/

}

执行命令比较简单，run\_applet\_and\_exit()函数根据命令编号调用applet\_main[]数组项中指定的执行函数即可，最后退出进程。如果是形如“$ls --help”的命令，则调用bb\_show\_usage()函数输出命令帮助信息。

### 2.3添加命令

Busybox提供了非常便利的配置、构建机制，使添加命令非常容易，所有添加命令的工作可以在一个源文件内完成。下面以一个非常简单的例子为例，说明添加命令的过程。

现在作者要向busybox添加hello命令，命令执行的结果是输出"Hello World!"字符串。作者在init/目录下创建一个名称为hello.c的源文件（也可以在其它目录下创建），文件内容如下。

/\*注意下面标红的地方\*/

//config:config HELLO

//config: bool "Print Hello World!"

//config: default y

//config: help

//config: Print Hello World!

//applet:IF\_HELLO(APPLET(hello, BB\_DIR\_BIN, BB\_SUID\_DROP))

//kbuild:lib-$(CONFIG\_HELLO) += hello.o

//usage:#define hello\_trivial\_usage

//usage: "Print Hello World!"

//usage:#define hello\_full\_usage "\n\n"

//usage: "Print Hello World!"

#include <stdio.h>

#include "libbb.h"

int hello\_main(int argc, char \*\*argv) MAIN\_EXTERNALLY\_VISIBLE;

int hello\_main(int argc UNUSED\_PARAM, char \*\*argv)

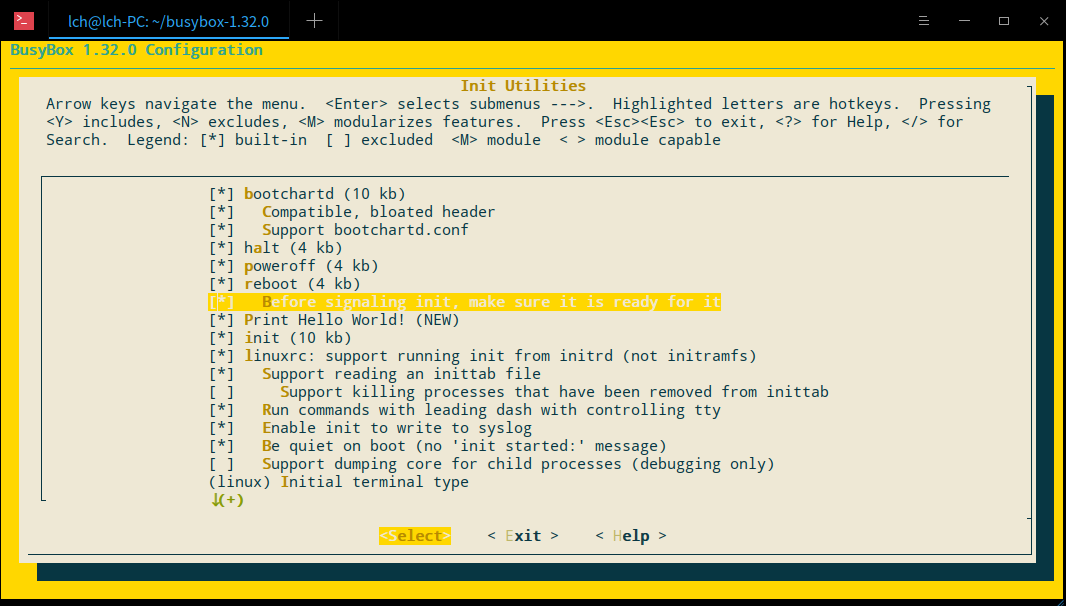
{

printf("Hello World!\n");

return EXIT\_SUCCESS;

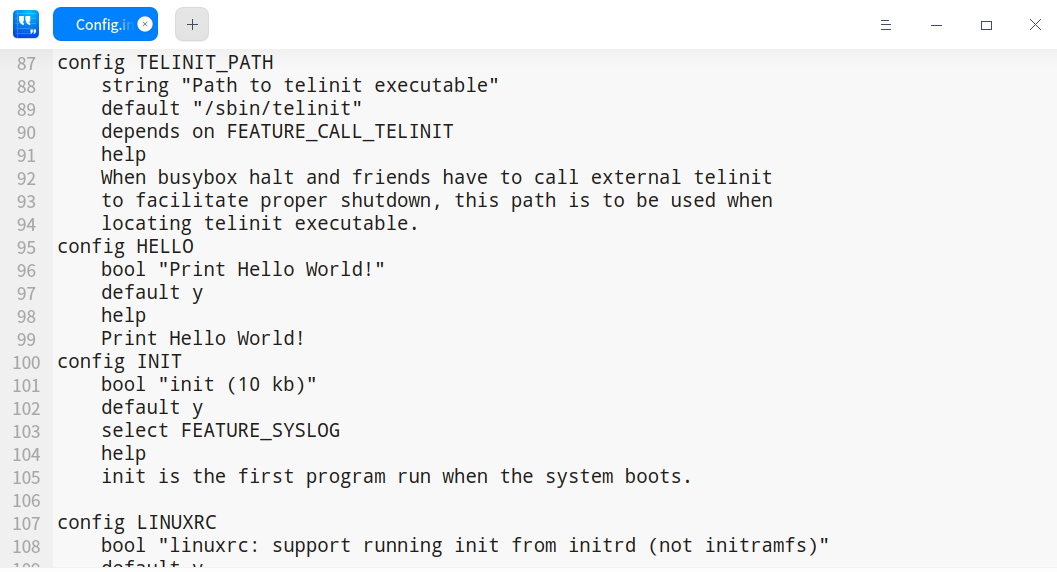
}

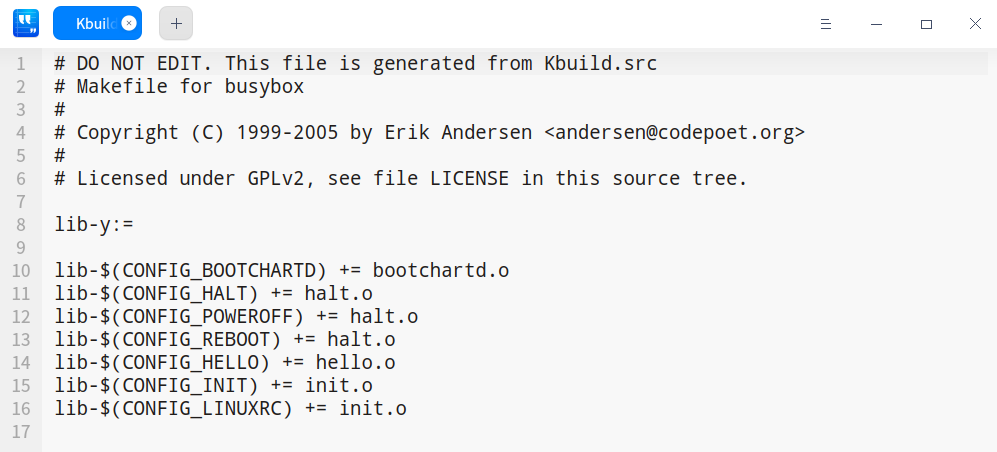
执行配置命令make menuconfig，在Init Utilities菜单下新出现了“Print Hello World!”选项，默认选择，如下图所示：



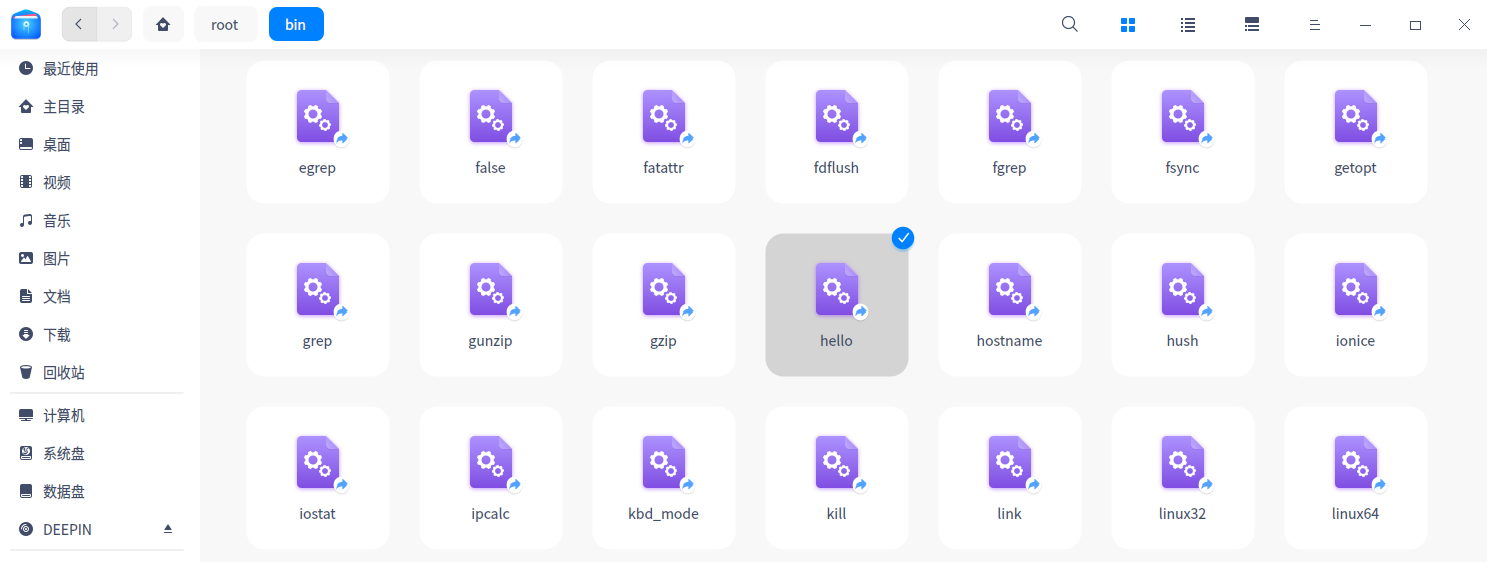
注：如果没有出现“Print Hello World!”选项，则先执行make clean命令，再执行make menuconfig。

配置完成后可以查看init/目录下的Config.in和Kbuild文件，看是否添加了相应的内容，如下图所示：





执行make命令对busybox进行构建操作，最后执行make install安装命令。在安装目录bin/下创建了hello命令的符号链接，如下图所示：



至此，添加hello命令完成。这只是一个非常简单的例子，读者可依照此方法添加自己的命令。