交换机

代码仓库，编译和运行

--- 1.2

1.1

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **日期** | **修订人** | **修订内容** |
| 1.2 | 2022/6/15 | 曾毅 | 大幅度更新 |
| 1.1 | 2022/6/15 | 曾毅 | 楠菲微更新了SDK编译框架，本文档用于说明之前SDK版本编译，故作废（新的SDK编译参考后续版本） |
| 1.0 | 2022/4/26 | 曾毅 | 初稿 |
|  |  |  |  |

目 录

[交换机 1](#_Toc2755)

[代码仓库，编译和运行 1](#_Toc31267)

[--- 1.2 1](#_Toc4320)

[1. 目的 4](#_Toc19009)

[2. 代码 5](#_Toc24773)

[2.1 楠菲微代码简述（旧） 5](#_Toc25330)

[2.1.1 交换SDK开发包 5](#_Toc8750)

[2.1.2 内部SOC开发包 5](#_Toc3675)

[2.2 楠菲微SDK 6.1.7 6](#_Toc6818)

[2.3 Connect 7](#_Toc22551)

[2.3.1 CPPF编译平台 7](#_Toc15055)

[2.3.2 通用编译框架 8](#_Toc15244)

[2.3.3 Connect业务组件 15](#_Toc4118)

[2.3.4 通用打包框架 16](#_Toc23019)

[2.4 代码仓库 17](#_Toc773)

[3. 版本控制 18](#_Toc7453)

[4. Gerrit系统 19](#_Toc21265)

[5. 编译和打包 20](#_Toc5933)

[5.1 主机平台环境 20](#_Toc221)

[5.1.1 32位编译器 20](#_Toc10737)

[5.2 编译 20](#_Toc9147)

[5.2.1 编译netforward 20](#_Toc8442)

[5.2.2 编译Connect 22](#_Toc19545)

[6. 升级 24](#_Toc30745)

# 目的

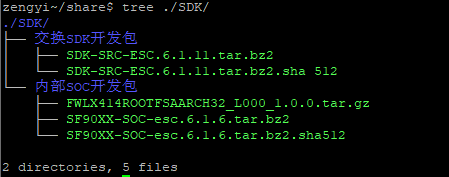
本文档基于总体设计，《代码仓库》和《编译和打包》制定交换机项目的代码仓库说明，代码编译方法。

# 代码

## 楠菲微代码简述（旧）

本章节是楠菲微以前提供的SDK套件的介绍，不知道为什么楠菲微又重新写了一套编译框架将CPU SDK和交换芯片SDK给合到一起编译，所以旧的本章节不再维护。

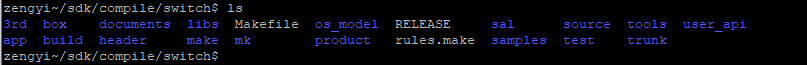
楠菲微提供的代码如下，



其中“交换SDK开发包”就是交换芯片的SDK，“内部SOC开发包”就是SOC芯片中CPU的代码，其中包括了SPL（Flash启动代码），uboot，kernel和好几个rootfs（也不知道为啥提供那么多个），另外编译器也顺带放在其中。

### 交换SDK开发包

解压“SDK-SRC-ESC.6.1.11.tar.bz2”压缩包，里面就是纯的SDK代码，SDK代码的学习可以参考《SDK代码结构介绍.pdf》，《SF90XX\_SDK API手册\_V1.0.pdf》，《SF90XX\_SDK User Guide\_V1.0.pdf》，主要还是要去看看代码。



### 内部SOC开发包

解压“FWLX414ROOTFSAARCH32\_L000\_1.0.0.tar.gz”压缩包和“SF90XX-SOC- esc.6.1.6.tar. bz2”压缩包，里面的内容就比较丰富了，

首先是FWLX414ROOTFSAARCH32\_L000\_1.0.0.tar.gz，里面有rootfs，默认包含了busybox-1.22.1，这个目录主要是用来编译busybox，然后通过内核编译那边的配置最后和内核一起打包成升级镜像（编译参考后续章节）。



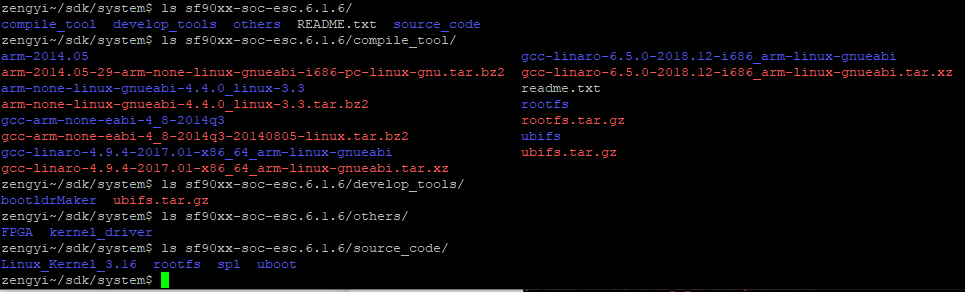
SF90XX-SOC- esc.6.1.6.tar. bz2解压后的一级目录包括，

compile\_tool：存放编译器

develop\_tools：主要放一些协助编译的工具，比如目前只有把SPL.bin和uboot.bin打包的工具

others：里面放的东西大概率都用不到，估计要问问楠菲微电子这部分怎么用

source\_code：里面放了spl，uboot和kernel的代码



这里面值得注意的是楠菲微提供了3个编译器，不同的编译器用来编译不同的代码组件，

gcc-arm-none-eabi-4\_8-2014q3：用来编译SPL代码，SPL里面的代码是在DDR初始化前运行的代码这部分代码的指令可能有一些特别要求。

gcc-linaro-6.5.0-2018.12-i686\_arm-linux-gnueabi：用来编译uboot代码，为啥uboot的编译器要和kernel的编译器分开这就搞不清了，很神奇。

arm-none-linux-gnueabi-4.4.0\_linux-3.3：用来编译内核，同时这个编译器也被用来编译各种各样的业务组件。

## 楠菲微SDK 6.1.7

楠菲微sdk 6.1.7将CPU BSP和交换芯片SDK合在了一起，通过Makefile一次性可以将bootloard，kernel和switch sdk编译完成。

解压sdk-src-all-board\_6.1.7-esc\_6.2.1\_20220512.tar.gz，代码目录如下，



Sdk-src-board\_6.1.7目录中存放BSP代码；

Sdk-src-esc\_6.2.1目录中存放交换芯片SDK代码；

Common 顾名思义就是通用的配置项；

Tools 也是顾名思义主要用于协助编译和打包，比如spl boot和uboot需要打包为bootloader等，另外编译器也在此目录下；

User 按照楠菲微的设计，系统集成公司（比如我司）的代码应该放在这个目录下，不过这是不可能的，系统集成的复杂性和多样性决定了楠菲微的这套编译框架无法使用；

Vendors 针对不同芯片（猜测也可以修改为针对不同客户）的编译configure文件；

Build.sh和makefile 需要注意的是整个系统是由makefile引导，不是从build.sh引导，敲make help可以看到提示。

Sdk-src-board\_6.1.7和sdk-src-esc\_6.2.1中的代码分析不在本文档的范围内。

## Connect

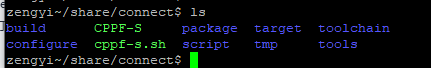
我司针对交换机开发使用Connect系统，Connect系统由多个部分组成，

### CPPF编译平台

一般来说编译平台会和业务组件分离，但是由于我司刚起步为了减少复杂度，减少维护工作量按时将编译平台和组件放在一起。

编译平台只是一个框架，负责引导编译和打包流程（不负责具体的编译工作），有的时候也做点别的（比如检查编译环境，Release版本等），市面上常见的编译平台有buildroot，yocto，openwrt等方便又好用，目前考虑到复杂度和维护工作量所以简单开发了一个CPPF编译平台，CPPF和上述编译平台相比最大的不足是无法建立组件依赖关系，但是根据CPPF编译平台设计后续如果有必要使用buildroot，业务组件包只需要很少的改动即可在buildroot等编译平台上编译。

编译平台目录如下，



每个编译平台其实都大同小异，

Cppf-s.sh 用于引导整个编译；

Package 用于存放组件包，业务根据自身的情况按照组件包的形式进行组织；

Target 用于编译过程中组件包的头文件依赖引用和库依赖引用，需要注意的是target只是用于编译过程中的依赖关系，最后生成的组件包中的动态库以及相关文件由打包框架提供解决方案；

Toolchain 顾名思义；

Tools 主要放协助编译的工具，这里需要分清的是CPPF是编译平台，CPPF平台并不使用tools中的工具；

Script，Configure和tmp 用于协助CPPF编译平台引导编译；

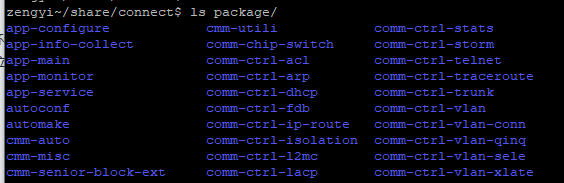
Build 用于存放编译的组件，和其他编译平台一样，CPPF也是将package中的组件拷贝到build目录中再引导编译，大家可以根据具体情况进行覆盖编译或者继续编译；

CPPF-S 文件用于CPPF平台搜索根目录，便于定位各级子目录。

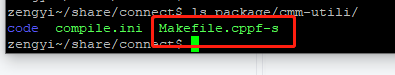
需要反复强调的是CPPF只是编译平台，不负责编译也不负责打包，组件包按照其规则放到package目录中，通过命令（比如 ./cppf-s.sh compile switch）触发组件编译，编译平台将组件拷贝到build目录中，触发组件包中的Makefile，剩下的工作都交由组件包自己完成。

在CPPF编译平台中放组件包的规则如下，

1. 必须放在package子目录中，



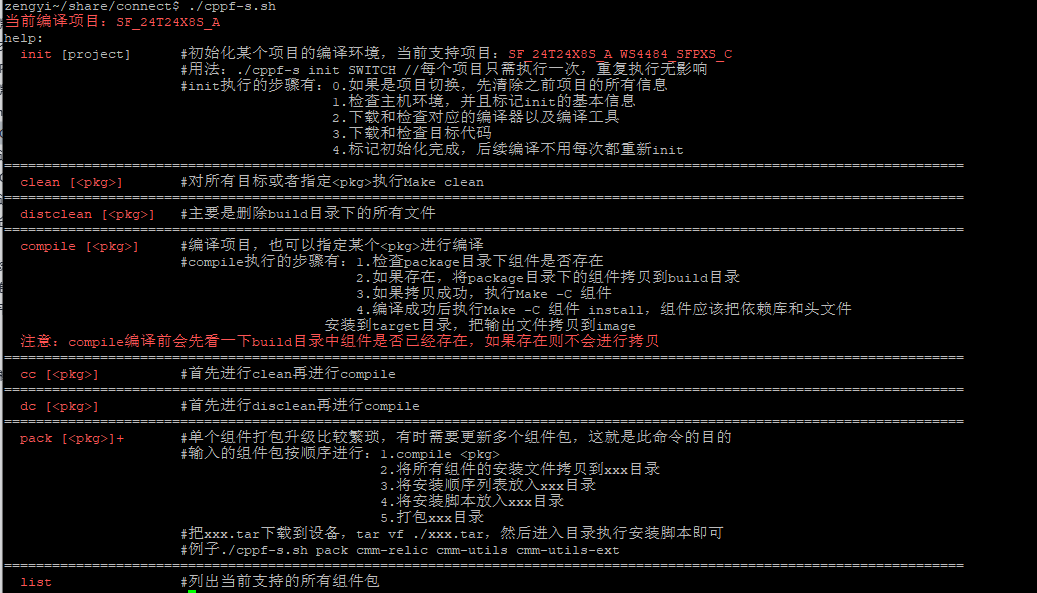
1. 在组件包的根目录下必须有一个Makefile.cppf-s文件或者build.sh文件，



1. 当触发组件包编译时，CPPF平台将组件包拷贝到./build目录中，按照如下顺序依次执行：
2. 如果有Makefile.cppf-s则将Makefile.cppf-s修改为Makefile并执行Make，
3. 如果Makefile.cppf-s不存在则检查build.sh是否存在，如果存在则执行./build.sh，
4. 如果两个文件都不存在则提示错误。

整个编译平台逻辑很简单，其他的事情都是组件包自行完成。

CPPF编译平台提供的功能可以通过输入./cppf-s.sh不带任何参数获得帮助信息，

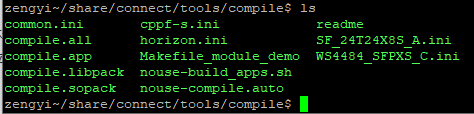


### 通用编译框架

前面介绍了CPPF编译平台，平台只提供编译引导但不提供组件的编译动作，对于一个大型系统或者一个大型项目是由很多个组件包组成，如果让每个组件包都单独编写编译Makefile（其实也可以，只是难度大些）会造成编译的选项和动作不统一存在潜在的风险，比如搜索路径配置错误导致引用了错误的头文件，为了提高质量和开发效率一般情况下会提供一套通用的编译框架，每个组件只要按照一定的规则配置要编译的文件以及生成的目标文件类型就可以用编译框架生成编译结果。

通用编译框架本质上和Linux automake原理一样，只是没那么高级，buildroot，openwrt以及各个大公司，包括楠菲微的SDK编译都是这么个套路。

Connect系统也提供了一套通用编译框架，被放置在./tools/compile目录下，



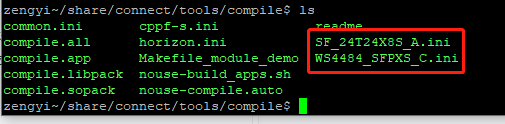
#### 编译框架初始化

通用编译框架的运作原理和引用原理如下，

首先在组件包的Makefile.cppf-s文件中引用cppf-s.ini，用于初始化CPPF编译环境，cppf-s.ini中的内容如下，

|  |
| --- |
| zengyi~/share/connect/tools/compile$ cat ./cppf-s.ini  #这一段通过搜索CPPF-S文件，找到connect框架的根目录  ifndef CPPF\_ROOT  export CPPF\_ROOT = $(shell  ... #内容省略  endif  #通过$(CPPF\_ROOT)/tmp/proj.ini，获得当前编译的项目  include $(CPPF\_ROOT)/tmp/proj.ini  export PATH:=$(PROJ\_COMPILE\_PATH)/bin/:$(shell printenv PATH)  #定义CPPF各级子目录，方便后续引用  export CPPF\_BUILD = $(CPPF\_ROOT)/build  export CPPF\_TARGET = $(CPPF\_ROOT)/target  export CPPF\_TOOLS = $(CPPF\_ROOT)/tools  export CPPF\_COMPILE = $(CPPF\_TOOLS)/compile  export CPPF\_PACKAGE = $(CPPF\_BUILD)/$(CPPF\_PACKAGE\_NAME)  export CPPF\_IMAGE = $(CPPF\_PACKAGE)/image  export CPPF\_TARGET\_SBIN\_DIR = $(CPPF\_TARGET)/usr/sbin  export CPPF\_TARGET\_LIB\_DIR = $(CPPF\_TARGET)/usr/lib  export CPPF\_TARGET\_INC\_DIR = $(CPPF\_TARGET)/usr/include  export CPPF\_TARGET\_SRC\_DIR = $(CPPF\_TARGET)/usr/src  zengyi~/share/connect/tools/compile$ |

在cppf-s.ini文件中引用的proj.ini文件是由CPPF编译平台中执行”./cppf-s.sh ini 项目”生成的，而生成的过程是CPPF平台通过查找./tools/compile/目录下的”项目.ini”获取而来。比如在当前支持WS4484\_SFPXS\_C和SF\_24T24X8S\_A两个项目，在tools/compile目录下则有以下两个ini文件，



proj.ini文件中主要存放项目以及项目相关的变量定义，

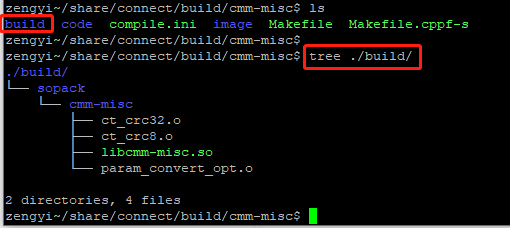
|  |
| --- |
| zengyi~/share/connect/tools/compile$ cat ./WS4484\_SFPXS\_C.ini  ifndef CPPF\_ROOT  $(error "CPPF\_ROOT don't exist.")  Endif  export PROJ\_NAME=WS4484\_SFPXS\_C #项目名称  export PROJ\_VERSION=1.0  #编译器路径  export PROJ\_COMPILE\_PATH=$(CPPF\_ROOT)/toolchain/compile-kernel/  export PROJ\_COMPILE\_CROSS=arm-none-linux-gnueabi-  export PROJ\_COMPILE\_SYSROOT=$(CPPF\_ROOT)/toolchain/compile-kernel/arm-none-linux-gnueabi  export PROJ\_COMPILE\_HOST=arm-none-linux-gnueabi |

接着引用horizon.ini，horizon就是通用编译框架的名字，该文件初始化了编译需要的通用变量和参数，

|  |
| --- |
| zengyi~/share/connect/tools/compile$ cat ./horizon.ini  #This horizon init file must behind cppf-s.ini  ifndef CPPF\_ROOT  $(error "This horizon init file must behind cppf-s.ini")  endif  #获得编译器路径，CPPF系统根目录  export HORIZON\_COMPILER\_DIR = $(PROJ\_COMPILE\_PATH)  export HORIZON\_COMPILE\_CROSS = $(PROJ\_COMPILE\_CROSS)  export HORIZON\_COMPILE\_SYSROOT = $(PROJ\_COMPILE\_SYSROOT)  #获得当前项目以及版本  export HORIZON\_PROJ\_NAME = $(PROJ\_NAME)  export HORIZON\_PROJ\_VERSION = $(PROJ\_VERSION)  #一些通用参数  export HORIZON\_DEBUG = 1  ifdef HORIZON\_DEBUG  HORIZON\_GCC\_DEBUG = -g  else  HORIZON\_GCC\_DEBUG =  endif  export HORIZON\_CFLAGS = $(OFLAGS) $(HORIZON\_GCC\_DEBUG) -D\_\_TRAPS\_\_ -D\_\_LINUX\_\_ -D\_\_IPV4\_\_ \  -D$(HORIZON\_PROJ\_NAME)=1 -DFIRMWARE\_VERSION=$(HORIZON\_PROJ\_VERSION) \  -Wall -Werror -Wno-sizeof-pointer-memaccess -Wno-unused-local-typedefs  export HORIZON\_COMPILE\_DIR = $(CPPF\_COMPILE)  export HORIZON\_INIT\_DIR = $(CPPF\_PACKAGE)  export HORIZON\_CODE\_DIR = $(CPPF\_PACKAGE)/code  export HORIZON\_OUTPUT\_DIR = $(CPPF\_PACKAGE)/build/  export HORIZON\_OUTPUT\_LIB\_DIR = $(HORIZON\_OUTPUT\_DIR)/lib/  export HORIZON\_OUTPUT\_SO\_DIR = $(HORIZON\_OUTPUT\_DIR)/sopack/  export HORIZON\_OUTPUT\_APP\_DIR = $(HORIZON\_OUTPUT\_DIR)/app/  zengyi~/share/connect/tools/compile$ |

通过对cppf-s.ini和horizon.ini引用，编译环境初始化完成，需要编译的项目以及相关的编译器路径已经指定，接着通过compile.all，compile.app，compile.libpack和compile.sopack生成相关的目标文件，当然也可以不引用这些文件自行编写编译。

compile.all，compile.app，compile.libpack和compile.sopack在编译过程中会在组件的一级目录下生成./build目录用于存放对应的目标文件，比如编译cmm-misc组件，



编译总是先生成obj文件，再根据需要将obj打包成为lib（静态或者动态）或者app进程，所以Compile.all主要用于编译obj文件，而compile.libpack用于将obj文件打包成为.a静态库，compile.sopack将obj文件打包成为.so动态库，最后compile.app用于生成进程。

#### Compile.all编译

Compile.all用于引导编译obj文件，具体的Makefile框架如下，

|  |
| --- |
| #以下部分用于填写编译要生成静态库的obj文件和编译选项  LIB\_OPT =  LIB\_INC =  LIB\_OBJS = #用于生成静态库的obj必填选项，比如ct\_crc.o  LIB\_BUILD\_TARGET = #静态库名字，会在build/lib目录下建立对应名字的目录，并且会将obj拷贝到此目录用于下一阶段静态库生成做准备  #以下部分用于填写生成动态库的obj文件和编译选项  SO\_OPT =  SO\_INC =  SO\_OBJS = ct\_crc8.o ct\_crc32.o param\_convert\_opt.o #obj文件  SO\_BUILD\_TARGET = cmm-misc #动态库名字，会在build/sopack目录下建立对应名字的目录，并且会将obj拷贝到此目录用于下一阶段动态库生成做准备  #考虑到实际情况，有可能同一个编译目录下需要同时生成不同的两个动态库，所以以下是另外一个动态库的编译配置。是否存在同一个目录下同时生成三个或者三个以上动态库的情况？雄蝶，如果有这种情况最好回过头看看自己组件的设计是否合理。  SO2\_OPT =  SO2\_INC =  SO2\_OBJS =  SO2\_BUILD\_TARGET =  #以下部分用于编译纯obj文件，一般会被直接生成进程  OBJ\_OPT =  OBJ\_INC =  OBJ\_OBJS = #obj文件  OBJ\_BUILD\_TARGET = #这里一般是进程的名字，会在build/app目录下建立对应名字的目录，并且会将obj拷贝到此目录，用于下一阶段生成进程  #用于递归编译子目录，就是简单的make -c XXX  SUBDIR\_PATH =  #前面都是配置，这里才是引用compile.all进行编译，在compile.all中通过前面定义的变量进行obj编译工作  include $(HORIZON\_COMPILE\_DIR)/compile.all |

compile.all中的内容如下，由于内容较多对其中的一些重复或者不重要的地方做了精简，

|  |
| --- |
| #引用每个组件的compile.ini文件，在这个文件中是每个组件相关的配置，包括编译选项，头文件引用路径等  include $(HORIZON\_INIT\_DIR)/compile.ini  BUILD\_DEPEND = #不支持  #配置输出目录  SOURCE\_DIR := $(shell pwd)  OUTPUT\_DIR := $(patsubst $(HORIZON\_CODE\_DIR)/%, $(HORIZON\_OUTPUT\_DIR)/%, $(SOURCE\_DIR))  #建立编译依赖  O\_PREPARED := $(OUTPUT\_DIR)/.prepared  # LIB编译对象  #O\_LIB\_TARGET := $(addprefix $(OUTPUT\_DIR)/, $(LIB\_NAME))  #O\_LIB\_GNU\_TARGET := $(addprefix $(OUTPUT\_DIR)/, $(addprefix lib, $(LIB\_NAME)))  O\_LIB\_OBJS := $(addprefix $(OUTPUT\_DIR)/, $(LIB\_OBJS))  O\_LIB\_DEPS := $(patsubst %.o, %.d, $(O\_LIB\_OBJS))  O\_LIB\_BUILD\_TARGET := $(addprefix $(HORIZON\_OUTPUT\_LIB\_DIR), $(LIB\_BUILD\_TARGET))  # SO编译对象  O\_SO\_OBJS := $(addprefix $(OUTPUT\_DIR)/, $(SO\_OBJS))  O\_SO\_DEPS := $(patsubst %.o, %.d, $(O\_SO\_OBJS))  O\_SO\_BUILD\_TARGET := $(addprefix $(HORIZON\_OUTPUT\_SO\_DIR), $(SO\_BUILD\_TARGET))  # SO2编译对象  O\_SO2\_OBJS := $(addprefix $(OUTPUT\_DIR)/, $(SO2\_OBJS))  O\_SO2\_DEPS := $(patsubst %.o, %.d, $(O\_SO2\_OBJS))  O\_SO2\_BUILD\_TARGET := $(addprefix $(HORIZON\_OUTPUT\_SO\_DIR), $(SO2\_BUILD\_TARGET))  # OBJ编译对象  O\_OBJ\_OBJS := $(addprefix $(OUTPUT\_DIR)/, $(OBJ\_OBJS))  O\_OBJ\_DEPS := $(patsubst %.o, %.d, $(O\_OBJ\_OBJS))  O\_OBJ\_BUILD\_TARGET := $(addprefix $(HORIZON\_OUTPUT\_APP\_DIR), $(OBJ\_BUILD\_TARGET))  # Main  #O\_TARGETS := $(O\_LIB\_TARGET) so\_target so2\_target obj\_target subdir\_target  O\_TARGETS := lib\_target so\_target so2\_target obj\_target subdir\_target  #编译开始，细节就是标准的Make，这里不再介绍  .PHONY:all  all:$(O\_PREPARED) $(O\_TARGETS)  $(O\_TARGETS): $(O\_PREPARED)  $(O\_PREPARED):  mkdir -p $(OUTPUT\_DIR)  touch $@  #编译细节...省略 |

在compile.all中引用的compile.ini内容如下，

|  |
| --- |
| #基本路径  TOOLS = $(HORIZON\_COMPILER\_DIR)/bin  PATH := $(TOOLS):$(PATH)  #编译器相关，这里可以引用proj.ini中的配置，也可以由组件自行填写其他路径  export CROSS = $(HORIZON\_COMPILE\_CROSS)  CC = $(CROSS)gcc --sysroot=$(HORIZON\_COMPILE\_SYSROOT)  AS = $(CROSS)as  CXX = $(CROSS)g++  AR = $(CROSS)ar  LD = $(CROSS)ld --sysroot=$(HORIZON\_COMPILE\_SYSROOT)  OBJCOPY = $(CROSS)objcopy  RANLIB = $(CROSS)ranlib  ELF2FLT = $(CROSS)elf2flt  STRIPTOOL = $(CROSS)strip  ARFLAGS = rcs  #编译选项相关  CFLAGS = $(HORIZON\_CFLAGS)  COMMON\_CFLAGS =  COMMON\_INCLUDE = $(CMM\_INCLUDE) |

#### Compile.libpack和Compile.sopack

由于Compile.all的工作比较到位，libpack和sopack只需做少量工作即可，上一章节讲到在compile.all中会将obj拷贝到./build/lib/name或者./build/sopack/name目录中，通过以下两条命令即可选中对应的目录并将目录中的obj打包为libname.a或者libname.so，

|  |
| --- |
| #在./build/lib中找到$(LIBNAME)目录，将目录中的obj文件打包为libname.a  $(MAKE) -f $(HORIZON\_COMPILE\_DIR)/compile.libpack LIBPACK=$(LIBNAME)  #在./build/sopack中找到$(SONAME)目录，将目录中的obj文件打包为libname.so  $(MAKE) -f $(HORIZON\_COMPILE\_DIR)/compile.sopack SOPACK=$(SONAME) |

#### Compile.app编译

和上一章节类似，app也只需要在./build/app目录中生成进程即可，只是比起库要稍微复杂一些，因为一个进程除了obj文件一般情况下还会引用其他静态库和动态库，所以需要一些额外的配置，

|  |
| --- |
| #在./build/app中找到$(APPNAME)目录，将目录中的obj文件打包为name进程  $(MAKE) -f $(HORIZON\_COMPILE\_DIR)/compile.app TARGET=$(APPNAME)  zengyi~/share/connect/tools/compile$ cat ./compile.app  include $(HORIZON\_INIT\_DIR)/compile.ini  #编译进程时，需要对组件目录下的$(TARGET)-link.ini进行引用，里面主要填写了需要引用的库文件和路径  include $(HORIZON\_INIT\_DIR)/$(TARGET)-link.ini  #编译部分省略...  zengyi~/share/connect/tools/compile$ cat ../../package/app-main/app-main-link.ini  #定义OBJ文件路径  OBJ\_PATH = $(HORIZON\_OUTPUT\_APP\_DIR)/$(APPNAME)  #定义静态库搜索路径，这里直接引用CPPF提供的标准路径  LIB\_PATH = $(CPPF\_TARGET\_LIB\_DIR)  #定义动态库搜索路径，这里直接引用CPPF提供的标准路径  SO\_PATH = $(CPPF\_TARGET\_LIB\_DIR)  #指定进程需要的静态库  LIB\_LIST = comm-chip-switch-lib comm-ctrl-acl-lib \  comm-ctrl-arp-lib comm-ctrl-dhcp-lib comm-ctrl-fdb-lib \  comm-ctrl-ip-route-lib comm-ctrl-isolation-lib comm-ctrl-l2mc-lib \  comm-ctrl-lacp-lib comm-ctrl-mirror-lib comm-ctrl-misc-lib \  comm-ctrl-packet-lib comm-ctrl-ping-lib comm-ctrl-port-lib \  comm-ctrl-pppoe-lib  #指定进程需要的动态库  SO\_LIST = cmm-utili cmm-misc cmm-senior-block-ext \  cmm-senior-callback cmm-senior-data-abs cmm-senior-dbg-ipc-client \  cmm-senior-dbg-ipc-server cmm-senior-misc cmm-senior-multi-inst cmm-senior-sadbox \  cmm-senior-trust-data cmm-senior-unit cmm-senior-varlen-block \  service-app-mgmt service-app service-apps service-dbg-ipc-client \  service-dbg-ipc-server service-error service-event \  service-ipc service-vfs service-module service-msg |

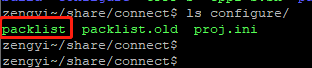
最后，需要注意的是通用编译框架只进行组件的编译，不管生成静态库，动态库还是进程都是单个组件的行为，为了解决组件之间的编译依赖和头文件依赖，当某个组件编译完成时，组件中的Makefile会将头文件和库文件拷贝到CPPF平台下的target目录以便其他组件引用。

### Connect业务组件

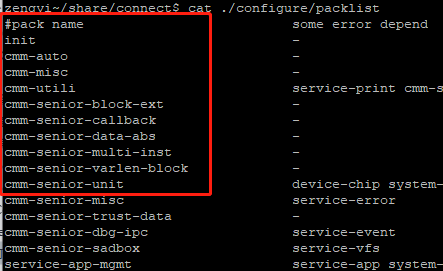
Connect业务组件要参考《Connect系统 总体设计》，在其中介绍了组件之间的关系。

### 通用打包框架

打包框架独立于系统的其它部分，当前为了简单在./configure目录下提供了packlist文件，



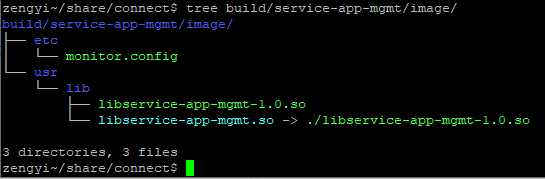
在packlist中存放了组件包名字，如果执行./cppf-s.sh pack all，将会根据packlist按照从上到下的顺序依次编译，最后将packlist中所有组件包的image目录中的内容拷贝到./build/image目录中，并对./build/image打包生成image.tar.gz文件，



目前打包比较简易，后续还需要加入升级包头以及加密的操作。

需要注意的是，打包框架要提取每个组件包的image目录，所以组件包需要在目标系统中运行的文件（进程，动态库以及配置文件）需要按照linux的目录格式在image目录中安放，比如service-app-mgmt组件是一个进程管理组件，它的image目录结构如下，





Monitor.config是service-app-mgmt运行时需要的配置文件，而service-app-mgmt组件作为动态库放在/usr/lib目录下。

## 代码仓库

当前我们的代码托管到http://172.16.1.24/ Gerrit系统中，

Gerrit系统的账号和密码需要申请；

成都如果要访问Gerrit需要额外添加静态路由：（管理员cmd）：route add -p 172.16.1.0 MASK 255.255.255.0 172.31.4.44；

需要注意虚拟机中也要加对应路由：sudo route add -net 172.16.1.0 netmask 255.255.255.0 gw 172.31.4.44

|  |  |
| --- | --- |
| **仓库** | **说明** |
| Connect | Connect系统，即2.3章节介绍的所有内容 |
| Netforward-sdk-6.1.7 | 2.2章节的所有内容 |
| switch | 额外的东西，不用太关心 |
|  |  |

# 版本控制

|  |  |  |
| --- | --- | --- |
| **仓库** | **Develop分支** | **Maintain分支** |
| Connect | 暂无，只有master分支 | 暂无 |
| Netforward-sdk-6.1.7 | Ws4484\_sfpxs\_c |  |
|  |  |  |

# Gerrit系统

各位自行学习，或者参考《Connect系统 代码仓库》文档。

# 编译和打包

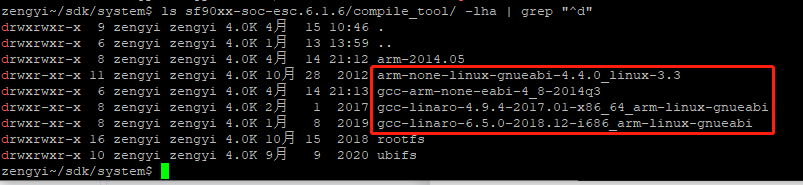
## 主机平台环境

如果在本地进行编译可以参考《虚拟机安装.doc》进行虚拟机的搭建，如果使用服务器编译只需要申请相关账号即可。

文档作者在本地编译使用“ubuntu-16.04.7-desktop-amd64.iso”镜像作为Linux系统主机编译平台，如果对Linux不是特别熟悉建议选用相同或者相近版本。

### 32位编译器

楠菲微电子提供的编译器基本都是32位的，由于主机平台默认是安装的64位系统，所以在编译前要安装32位包支持32位GCC进程。



安装方法（也可以自行在网上搜索）：

|  |
| --- |
| $sudo dpkg --add-architecture i386  $sudo apt-get update  $sudo apt-get install ia32-libs |

## 编译

### 编译netforward

代码的下载参考《Connect系统 代码仓库》，代码下载以后进入目录执行make help可以看到楠菲微的编译提示，

|  |
| --- |
| zengyi~/share/netforward-sdk-6.1.7$ make help  ---------------------------------make help info---------------------------------  Usage:  The following commands require board parameters, such as: make [option] [board=?]  option:  source: download all source  all: build all  spl: build spl  boot: build boot  linux: build linux  rootfs: build rootfs  sdk: build sdk  board=[board name]  eg : make all board=es90soc  The following command does not require board parameters, such as: make [option]  option:  test: build package for Testers  release: build package for release  svninfo: get all svninfo  del\_build: delete build dir  del\_source: delete sdk-src-board sdk-src-esc dir  del\_output: delete output dir  del\_all: delete build\sdk-src-board\sdk-src-esc\output dir  eg : make svninfo  ---------------------------------make help info---------------------------------  zengyi~/share/netforward-sdk-6.1.7$ |

由于本项目在Ws4484\_sfpxs\_c分支上维护，所以在编译之前要进行分支切换，

|  |
| --- |
| zengyi~/share/netforward-sdk-6.1.7$ git branch  \* master  zengyi~/share/netforward-sdk-6.1.7$ git branch -r  origin/HEAD -> origin/master  origin/master  origin/ws4484\_sfpxs\_c  zengyi~/share/netforward-sdk-6.1.7$ git checkout -b ws4484\_sfpxs\_c origin/ws4484\_sfpxs\_c  zengyi~/share/netforward-sdk-6.1.7$ git branch  master  \* ws4484\_sfpxs\_c  zengyi~/share/netforward-sdk-6.1.7$ make all board=es90soc |

输入make all board=es90soc编译SDK所有内容，当前我们选择的是es90soc系列。需要小心的是楠菲微的SDK编译在第一次编译时会将sdk-src-board\_6.1.7和sdk-src-esc\_6.2.1拷贝成为对应的sdk-src-board和sdk-src-esc目录，后续每次编译时都是检查sdk-src-board和sdk-src-esc目录中文件的改动，并将改动拷贝到./build目录中进行编译，不会检查sdk-src-board\_6.1.7和sdk-src-esc\_6.2.1中的内容改动，但是如果大家基于sdk-src-board和sdk-src-esc目录进行修改调试，当修改完成后需要自行将改动拷贝回sdk-src-board\_6.1.7和sdk-src-esc\_6.2.1，否则当执行delete时sdk-src-board和sdk-src-esc目录会被删除，导致修改丢失。

编译生成的结果都在./build目录中，在./build/es90soc/output/中存放了所有的目标文件。这些文件如何使用？请参考《刷机和升级指南-1.0》

### 编译Connect

代码的下载参考《Connect系统 代码仓库》，代码下载以后进入目录执行./cppf-s.sh可以看到编译提示，

|  |
| --- |
| zengyi~/share/connect$ ./cppf-s.sh  当前编译项目：SF\_24T24X8S\_A  help:  init [project] #初始化某个项目的编译环境，当前支持项目：SF\_24T24X8S\_A WS4484\_SFPXS\_C  #用法：./cppf-s init SWITCH //每个项目只需执行一次，重复执行无影响  #init执行的步骤有：0.如果是项目切换，先清除之前项目的所有信息  1.检查主机环境，并且标记init的基本信息  2.下载和检查对应的编译器以及编译工具  3.下载和检查目标代码  4.标记初始化完成，后续编译不用每次都重新init  ===============================================================================================  clean [<pkg>] #对所有目标或者指定<pkg>执行Make clean  ===============================================================================================  distclean [<pkg>] #主要是删除build目录下的所有文件  ===============================================================================================  compile [<pkg>] #编译项目，也可以指定某个<pkg>进行编译  #compile执行的步骤有：1.检查package目录下组件是否存在  2.如果存在，将package目录下的组件拷贝到build目录  3.如果拷贝成功，执行Make -C 组件  4.编译成功后执行Make -C 组件 install，组件应该把依赖库和头文件  安装到target目录，把输出文件拷贝到image  注意：compile编译前会先看一下build目录中组件是否已经存在，如果存在则不会进行拷贝  ================================================================================================  cc [<pkg>] #首先进行clean再进行compile  ================================================================================================  dc [<pkg>] #首先进行disclean再进行compile  ================================================================================================  pack [<pkg>]+ #单个组件打包升级比较繁琐，有时需要更新多个组件包，这就是此命令的目的  #输入的组件包按顺序进行：1.compile <pkg>  2.将所有组件的安装文件拷贝到xxx目录  3.将安装顺序列表放入xxx目录  4.将安装脚本放入xxx目录  5.打包xxx目录  #把xxx.tar下载到设备，tar vf ./xxx.tar，然后进入目录执行安装脚本即可  #例子./cppf-s.sh pack cmm-relic cmm-utils cmm-utils-ext  ================================================================================================  list #列出当前支持的所有组件包 |

使用./cppf-s.sh init [project]切换编译项目，内部就是将./tools/compile/下对应项目的proj.ini文件拷贝到./tmp/proj.ini以便后续引用相关变量和编译器路径。切换项目只做一次，但是不能在编译过程中切换，比如编译了20个组件以后切换项目再继续编译。

单个组件编译的日常操作，由./cppf-s.sh [compile|clean|cc|dc|distclean] <pkg>这几条命令搭配完成。

./cppf-s.sh compile <pkg> 会检查组件包是否再./build目录中，如果不再则从./package中拷贝到./build中进行编译，如果./build目录中已经存在则直接编译；

./cppf-s.sh clean <pkg> 对./build中对应的组件执行make clean；

./cppf-s.sh cc <pkg> 是前面两个命令的结合，因为如果每次都先clean再compile太麻烦，直接用cc就等于先clean，在compile；

./cppf-s.sh distclean <pkg> 会直接删除./build目录下组件包括代码，在删除前会对组件执行make clean或者调用unbuild.sh，所以组件通过相关命令要自行删除./target下的内容，否则会造成删除不干净；

./cppf-s.sh dc <pkg> 这条命令用的相对比较多，就是执行distclean以后重新compile，它的方便之处在于每次改了代码自动删除原来的重新编译，由于组件包代码一般都不多即使重新编译也花不了多少时间。

./cppf-s.sh [distclean|pack] <pkg>+结合可以完成打包动作，这两条命令都可以同时给多个组件，比如 ./cppf-s.sh distclean service-app service-app-mgmt service-apps同时清除这三个组件。

./cppf-s.sh pack all 即对./configure/packlist中的所有组件依次编译，最后提取所有组件的image文件并打包成image.tar.gz。因为我们支持进程化和组件化，在实际调试的过程中不需要每次完整打包，可以指定某几个组件打包然后快速安装，编译，升级和重启业务的时间都会大大缩短，比如通过./cppf-s.sh pack service-app service-app-mgmt service-apps打包三个组件，生成的image.tar.gz文件里面只有这三个组件的动态库和配置文件，只需要把相关动态库解压缩到对应目录重启进程即可。

./cppf-s.sh distclean all 按照./configure/packlist中的组件进行distclean，最后会贴别删除./target/目录下的所有内容。

./cppf-s.sh list 为了方便查看pack all编译组件的顺序，没啥其他用。

使用./cppf-s.sh pack <pkg>+命令会在CPPF编译平台的根目录下生成image.tar.gz文件。Image.tar.gz文件怎么使用？请参考《刷机和升级指南-1.0》。

# 升级

参考《刷机和升级指南》。