# **SONiC Buildimage Guide**

概述：

Sonic buildimage 是一个基于自动构建过程环境的一个GUN（GUN是一个自由OS，以GPL（通用公共许可证，用来试图保证共享和修改自由软件的自由）的方式发布。其中包含两个主要的部分：

①Makefile的后端收集和其他定义通用目标组的帮助，被方法使用

②前端-配方的集合，定义每个构建目标的元数据

结构“

Sonic buildimage的文件结构如下：

sonic-buildimage/

Makefile 【规定了编译规则，具体如何实现是在同一目录下的Makefile.work】

slave.mk 【被Makefile.work引用，也是用来定义编译的规则，规定构建路径】

sonic-slave/

Dockerfile 【这个文件中是安装一系列各个模块需要用到的软件】

rules/ 【rule下面是各种的mk文件，对应着各个模块】

config 【基础的配置信息，比如默认登录名 密码 路由栈什么的】

Functions【添加新得到的包路径以及额外的包，定义打印的红黄灰绿等颜色】

recipe1.mk

..

dockers/

docker1/

Dockerfile.template

..

src/ 【下面存放的是各个功能模块

submodule1/

..

package1/

Makefile 这个Makefile规定的就只当前这个模块的编译规则

..

platform/ 【存放的是各个平台的信息，其中如果添加一个平台就是主要在这做修改】

vendor1/

..

target/ 【目标文件夹，各种编译目标的存放文件夹】

debs/ 【存放编译好的deb包】

python-wheels/

后端：Makefile 、slave.mk 、sonic-slave、

Makefile，slave.mk和sonic-slave/Dockerflile属于构建镜像的后端文件（buildimage）。Slave.mk是一个真正的编译脚本。它给目标群组定义了一系列的规则。在slave文件中，能够找到任何一个定义在recipe中的目标的编译规则。Makefile是一个sonic-slave Docker镜像的一个封装。【就是说，makefile这个文件就是slave这个文件的一个外壳，执行的还是slave这个真正的编译文件】

构造的每个部分都在一个叫做sonic-slave的容器中执行，特别是这个环境的技巧。【可能就是说的那些功能吧】，如果是第一次在一个特定的主机上开始构造镜像，那么这台机器上的sonic-slave/Dockerfile会生成一个新的sonic-slave镜像。这会花费一个时间，要有耐心。在所有的准备工作结束后，后面的编译命令会在这个容器内执行。Makefile会将目标传送给编译命令并且并且代表它作为一个指向一个容器的入口，使运行容器透明处理。

前端：

Rules/这个文件夹有一个平台独立目标recipe的收集。**每个recipe是都是一个文件，这个recipe文件描述了一个特定目标的元数据，这个元数据对这个目标的构造是必须的。**

Rules/config是很有用的。因为它是构造系统的一个配置，它能够打开或者关闭一些调整。

在dockers/路径下，能够找到通用Docker镜像的Docker文件。

src/用来存放通用包源代码。其中有子模块（简单样例，只在dpke-buildpackage去构建的时候运行）和复杂部件的路径，由它们呢自己的编译文件提供这个路径。

Platform/包括所有特定供应商的recipes以及子模块等。

每个platform/供应商名/ 这个路径是buildimage前端的派生部分，定义了规则和具体供应商的目标。

构造输出：

target/这个路径是构造结果的基本输出路径。在这个路径下，能找到构造的所有结果。

Recipe和目标组（Target group）：

Recipe是一个小编译文件，用来定义Target和去构造这个target所需要的变量。如果想要去添加一个新的target去构造镜像的话，（deb包或者Docker镜像），就必须去为了这个target创建一个recipe。

Target group指的是这一系列要编译的目标都是依据一个rules。【比如cx532p平台的编译和cx系列其他平台的编译是一个rule】。每个recipe会设置一个target所属的目标组。

Recipe范例：

Swsscommon的recipe：

# libswsscommon package

LIBSWSSCOMMON = libswsscommon\_1.0.0\_amd64.deb

$(LIBSWSSCOMMON)\_SRC\_PATH = $(SRC\_PATH)/sonic-swss-common

$(LIBSWSSCOMMON)\_DEPENDS += $(LIBHIREDIS\_DEV) $(LIBNL3\_DEV) $(LIBNL\_GENL3\_DEV) \$(LIBNL\_ROUTE3\_DEV) $(LIBNL\_NF3\_DEV) \

$(LIBNL\_CLI\_DEV)

$(LIBSWSSCOMMON)\_RDEPENDS += $(LIBHIREDIS) $(LIBNL3) $(LIBNL\_GENL3) \

$(LIBNL\_ROUTE3) $(LIBNL\_NF3) $(LIBNL\_CLI)SONIC\_DPKG\_DEBS += $(LIBSWSSCOMMON)

LIBSWSSCOMMON\_DEV = libswsscommon-dev\_1.0.0\_amd64.deb

$(eval $(call add\_derived\_package,$(LIBSWSSCOMMON),$(LIBSWSSCOMMON\_DEV)))

Note：

我们第一次定义swsscommon这个包。我们将这个过程分为SRC\_PATH(资源路径),DEPENDS(构造依赖)，和RDEPENDS(Docker安装时运行时所需依赖)。

然后将我们的target添加到SONIC\_DPKG\_DEBS目标组。

最后我们为swsscommon定义设备包并从主要设备中做编译。使用add\_derived\_package这个方法做包元数据的深层拷贝，这样做我们就省去了不必要的重复。

【所以cx532p这这一步执行也就是做深层次的拷贝】

目标组：

Sonic\_dpkg\_debs:

编译deb包的主目标组，定义如下：

SOME\_NEW\_DEB = some\_new\_deb.deb # 所需要编译deb包的名字 $(SOME\_NEW\_DEB)\_SRC\_PATH = $(SRC\_PATH)/project\_name # path to directory with sources指出这个新包需要的资源目录

$(SOME\_NEW\_DEB)\_DEPENDS = $(SOME\_OTHER\_DEB1) $(SOME\_OTHER\_DEB2) ... # build dependencies构造这个包所依赖的其他包资源

$(SOME\_NEW\_DEB)\_RDEPENDS = $(SOME\_OTHER\_DEB1) $(SOME\_OTHER\_DEB2) ... # runtime dependencies运行时所需要的依赖

SONIC\_DPKG\_DEBS += $(SOME\_NEW\_DEB) # 将这个包加入到目标组中

Sonic\_python\_stdeb\_debs

和sonic\_dpkg\_debs一样，但是不是使用dpkg-buildpackage构建包，而是执行python setup.py --command-packages = stdeb.command bdist\_deb。 定义如下：

SOME\_NEW\_DEB = some\_new\_deb.deb # name of your package$(SOME\_NEW\_DEB)\_SRC\_PATH = $(SRC\_PATH)/project\_name # path to directory with sources

$(SOME\_NEW\_DEB)\_DEPENDS = $(SOME\_OTHER\_DEB1) $(SOME\_OTHER\_DEB2) ... # build dependencies

$(SOME\_NEW\_DEB)\_RDEPENDS = $(SOME\_OTHER\_DEB1) $(SOME\_OTHER\_DEB2) ... # runtime dependencies

SONIC\_PYTHON\_STDEB\_DEBS += $(SOME\_NEW\_DEB) # add package to this target group将包加入到目标组中

Sonic\_make\_debs

这个有一些灵活。如果必须要做一些特定的构造类型，或者应用之前路径做构造，只需要定义你自己的makefile【编译文件】并把他加入到buildimage中即可。定义如下：

SOME\_NEW\_DEB = some\_new\_deb.deb # name of your package$(SOME\_NEW\_DEB)\_SRC\_PATH = $(SRC\_PATH)/project\_name # path to directory with sources

$(SOME\_NEW\_DEB)\_DEPENDS = $(SOME\_OTHER\_DEB1) $(SOME\_OTHER\_DEB2) ... # build dependencies

$(SOME\_NEW\_DEB)\_RDEPENDS = $(SOME\_OTHER\_DEB1) $(SOME\_OTHER\_DEB2) ... # runtime dependencies

SONIC\_MAKE\_DEBS += $(SOME\_NEW\_DEB) # add package to this target group

如有一些包由于法律问题必须在本地构造，或者他们早就被预构造并可以在线获取，那么下面这四个目标组很有用：sonic\_copy\_files,sonic\_online\_debs,sonic\_online\_files,sonic\_online\_files

Sonic\_copy\_debs

这些包只从从本机的特定位置拷贝，定义如下：

SOME\_NEW\_DEB = some\_new\_deb.deb # name of your package$(SOME\_NEW\_DEB)\_PATH = path/to/some\_new\_deb.deb # path to file

SONIC\_COPY\_DEBS += $(SOME\_NEW\_DEB) # add package to this target group

第二步就是指出deb所在本机位置

Sonic\_copy\_files:

除了规律文件的应用，其他的和上面一样。当你需要一些regular文件在Docker容器中做安装时使用这个目标组。定义如下：

SOME\_NEW\_FILE = some\_new\_file # name of your file

$(SOME\_NEW\_FILE)\_PATH = path/to/some\_new\_file # path to file

SONIC\_COPY\_FILES += $(SOME\_NEW\_FILE) # add file to this target group

【第二步大概是指向指向这个要拷贝的文件的源路径】

Sonic\_online\_debs

Deb包目标组应该从网上获取的。定义如下：

SOME\_NEW\_DEB = some\_new\_deb.deb # name of your package$(SOME\_NEW\_DEB)\_URL = https://url/to/this/deb.deb # path to file # URL for downloading

SONIC\_ONLINE\_DEBS += $(SOME\_NEW\_DEB) # add file to this target group

Sonic\_online\_files

Regular文件目标组应该从网上获取的，定义如下：

SOME\_NEW\_FILE = some\_new\_file # name of your file$(SOME\_NEW\_FILE)\_URL = https://url/to/this/file # URL for downloading

SONIC\_ONLINE\_FILES += $(SOME\_NEW\_FILE) # add file to this target group

Docker镜像也有他们的目标组

Sonic\_simple\_docker\_images

正如从组名称看到的那样，旨在从常规（regular）dockerfile文件构建docker镜像。定义如下：

SOME\_DOCKER = some\_docker.gz # name of your docker

$(SOME\_DOCKER)\_PATH = path/to/your/docker # path to your Dockerfile

SONIC\_SIMPLE\_DOCKER\_IMAGES += $(SOME\_DOCKER) # add docker to this group

Sonic\_docker\_images

这个稍微复杂一些。你能够从buildimage中定义deb包 ，这个buildimage会安装这个deb包，相应的dockerfile文件会从模板一般性生成。

SOME\_DOCKER = some\_docker.gz # name of your docker

$(SOME\_DOCKER)\_PATH = path/to/your/docker # path to your Dockerfile

$(SOME\_DOCKER)\_DEPENDS += $(SOME\_DEB1) $(SOME\_DEB2) # .deb packages to install into image

$(SOME\_DOCKER)\_PYTHON\_WHEELS += $(SOME\_WHL1) $(SOME\_WHL2) # python wheels to install into image

$(SOME\_DOCKER)\_LOAD\_DOCKERS += $(SOME\_OTHER\_DOCkER) # docker image from which this one is built

SONIC\_DOCKER\_IMAGES += $(SOME\_DOCKER) # add docker to this group

以上这个目标组的第二步都是在指出所需要资源或者包、文件的文件所在位置。

提示和技巧：

虽然每个target都是在sonic-slave这个容器中创建的，sonic-slave这个容器会在构造结束时退出，你还是能够进入sonic-slave的bash中使用下面这个命令：

$ make sonic-slave-bash

当你添加一个新的目标时并且遇到了问题时，这个命令很有用。

Sonic-slave环境只能被构建一次，但是如果sonic-slave/Dockerfile更新了，可以使用下面这个命令去重建:

$ make sonic-slave-build

通过执行下面这个命令，可以打印输出所有的变量目标

$ make list

所有的目标组会被一个或者另一个recipe所使用，所以当要添加一个新东西时，可以使用这个recipes作为参考。

构造调试docker并且调试sonic安装镜像：

使用 'INSTALL\_DEBUG\_TOOLS=y'构造调试镜像。

**例如:INSTALL\_DEBUG\_TOOLS=y make target/sonic-broadcom.bin**

①构造调试dockerfile镜像

②调试镜像带有后缀’-dbg’

例如：target/docker-orchagent-dbg.gz

③target/sonic-braodcom.bin是通过使用调试镜像构造的

选择性源存档并在/ src下可用

创建一个空/调试目录，以便在调试会话期间使用。

所有调试docker都使用/ src：ro和/ debug：rw挂载

登录横幅将简要描述这些功能。

Note:target/sonic-broadcom.bin这个名字和是否使用了调试镜像无关。

Recommend：使用INSTALL\_DEBUG\_TOOLS=y

重命名镜像防止冲突。可以使用mv target/sonic-broadcom.bin target/sonic-broadcom-dbg.bin

【就是说如果要调试镜像 就在编译命令前加上INSTALL\_DEBUG\_TOOLS=y】

Debug docker

①使用所有变量调试标志构造

②使用调试要求的许多基本包安装

* + gdb
  + gdbserver
  + vim
  + strace
  + openssh-client
  + sshpass

③可加载到支持docker的任何环境中

④外部的sonic镜像，可以使用后面这个命令运行docker

--entrypoint=/bin/bash

⑤使用-v这个参数指出你的主机目录

⑥在支持docker的非sonic环境中调试内核文件

* docker load -i docker-<name>-dbg.gz
* copy your unzipped core file into ~/debug
* docker run -it -entrypoint=/bin/bash -v ~/debug:/debug <image id>
* gdb /usr/bin/<your binary> -c /debug/<your core>

【这一部分应该很快就会需要】

Debug sonic image

①将这个镜像安装到支持这个镜像的交换机

对于与平台无关的二进制文件，您可以使用调试映像进行虚拟交换

②如果需要调试的源码，在/src中打开存档

③每个打开docker的debug使用只读/src挂载

④主机有/ debug 路径，它被映射到每个可调试的docker中作为/ debug并具有读写权限。

⑤调试内核

将内核拷贝到主机的/debug并且解压。

随意根据需要创建和使用子目录。

整个/debug安装在docker中

进入Docker 使用docker -it exec <name> bash

gdb /usr/bin/<binary> -c /debug/<core file path>【这个就是在Docker这个容器中做内核文件调试。

在gdb中使用set-directory去映射src下合适的源路径

能够设置gdb进入/debug

⑥用于实时调试

将次作为常规开关使用

进入docker，能够做

在dbg下开始进程

将gdb放到运行进程中

如果需要，在src下设置请求源路径

可以使用/ debug记录所有geb日志或来自调试会话的任何输出。

提升debug Dockers

①添加到<docker name> \_DBG\_IMAGE\_PACKAGES，这是将在构建期间预安装的其他调试工具。

例如 $（DOCKER\_ORCHAGENT）\_DBG\_IMAGE\_PACKAGES + = perl

Build将使用“apt”安装这些工具

②添加到<docker name> \_DBG\_DEPENDS，将在构建期间预安装的其他调试.deb软件包。

例如 $（DOCKER\_ORCHAGENT）\_DBG\_DEPENDS + = $（SWSS\_DBG） $（LIBSWSSCOMMON\_DBG）$（LIBSAIREDIS\_DBG）

③添加到DBG\_SRC\_ARCHIVE，源路径进行存档

此目录树下的源文件（.c，.cpp，.h和.hpp）已存档。

例如 rules / swss.mk有DBG\_SRC\_ARCHIVE + = swss