# 31. 开发套件构建系统

DPDK 需要一个构建系统用于编译等操作。 本节介绍 DPDK 框架中使用的约束和机制。

这个框架有两个使用场景：

* 编译DPDK库和示例应用程序，该框架生成特定的二进制库，包含文件和示例应用程序。
* 使用安装的DPDK二进制编译外部的应用程序或库。

## 31.1. 编译DPDK二进制文件

以下提供了如何构建DPDK二进制文件。

### 31.1.1. 建立目录概念

安装之后，将创建一个构建目录结构。 每个构件目录包含文件、库和应用程序。

构建目录特定于配置的体系结构、执行环境、工具链。 可以存在几个构建目录共享源码，但是配置不一样的情况。

例如，要使用配置模板 config/defconfig\_x86\_64-linuxapp 创建一个名为 my\_sdk\_build\_dir 的构建目录，我们使用如下命令：

cd ${RTE\_SDK}

make config T=x86\_64-native-linuxapp-gcc O=my\_sdk\_build\_dir

这会创建一个新的 new my\_sdk\_build\_dir 目录，之后，我们可以使用如下的命令进行编译：

cd my\_sdk\_build\_dir

make

相当于：

make O=my\_sdk\_build\_dir

目录 my\_sdk\_build\_dir 的内容是：

-- .config # used configuration

-- Makefile # wrapper that calls head Makefile

# with $PWD as build directory

-- build #All temporary files used during build

+--app # process, including . o, .d, and .cmd files.

| +-- test # For libraries, we have the .a file.

| +-- test.o # For applications, we have the elf file.

| `-- ...

+-- lib

+-- librte\_eal

| `-- ...

+-- librte\_mempool

| +-- mempool-file1.o

| +-- .mempool-file1.o.cmd

| +-- .mempool-file1.o.d

| +-- mempool-file2.o

| +-- .mempool-file2.o.cmd

| +-- .mempool-file2.o.d

| `-- mempool.a

`-- ...

-- include # All include files installed by libraries

+-- librte\_mempool.h # and applications are located in this

+-- rte\_eal.h # directory. The installed files can depend

+-- rte\_spinlock.h # on configuration if needed (environment,

+-- rte\_atomic.h # architecture, ..)

`-- \\*.h ...

-- lib # all compiled libraries are copied in this

+-- librte\_eal.a # directory

+-- librte\_mempool.a

`-- \\*.a ...

-- app # All compiled applications are installed

+ --test # here. It includes the binary in elf format

请参阅 [Development Kit Root Makefile Help](https://link.jianshu.com?t=dev_kit_root_make_help.html" \l "development-kit-root-makefile-help" \t "_blank) 获取更详细的信息。

## 31.2. 构建外部应用程序

由于DPDK本质上是一个开发工具包，所以最终用户的第一个目标就是使用这个SDK创建新的应用程序。 要编译应用程序，用户必须设置 RTE\_SDK 和 RTE\_TARGET 环境变量。

export RTE\_SDK=/opt/DPDKexport RTE\_TARGET=x86\_64-native-linuxapp-gcccd /path/to/my\_app

对于一个新的应用程序，用户必须创建新的 Makefile 并包含指定的 .mk 文件，如 ${RTE\_SDK}/mk/rte.vars.mk 和 ${RTE\_SDK}/mk/rte.app.mk。 这部分内容描述请参考 [Building Your Own Application](https://link.jianshu.com?t=build_app.html" \l "building-your-own-application" \t "_blank) .

根据 Makefile 所选定的目标（架构、机器、执行环境、工具链）或环境变量，应用程序和库将使用适当的h头文件进行编译，并和适当的a库链接。 这些文件位于 ${RTE\_SDK}/arch-machine-execenv-toolchain，由 ${RTE\_BIN\_SDK} 内部引用。

为了编译应用程序，用户只需要调用make命令。编译结果将置于 /path/to/my\_app/build 目录。

示例应用程序在example目录中提供。

## 31.3. Makefile 描述

### 31.3.1. DPDK Makefiles 的通用规则

在DPDK中，Makefiles始终遵循相同的方案：

起始处包含 $(RTE\_SDK)/mk/rte.vars.mk 文件。

为RTE构建系统定义特殊的变量。

包含指定的 $(RTE\_SDK)/mk/rte.XYZ.mk 文件，其中 XYZ 可以是 app、lib、extapp, extlib、obj、gnuconfigure等等，取决于要编译什么样的目标文件。 请参阅

[See Makefile Types](" \l "makefile-types" \t "_blank)

描述。

包含用户定义的规则及变量。

以下是一个简单的例子，用于便于一个外部应用程序：

include $(RTE\_SDK)/mk/rte.vars.mk

# binary name

APP = helloworld

# all source are stored in SRCS-y

SRCS-y := main.c

CFLAGS += -O3

CFLAGS += $(WERROR\_FLAGS)

include $(RTE\_SDK)/mk/rte.extapp.mk

### 31.3.2. Makefile 类型

根据Makefile最后包含的 .mk 文件，Makefile将具有不同的角色。 注意到，并不能在同一个Makefile文件中同时编译库和应用程序。 因此，用户必须创建两个独立的Makefile文件，最好是置于两个不同的目录中。

无论如何，rte.vars.mk 文件必须包含用户Makefile。

#### 31.3.2.1. 应用程序

这些 Makefiles 生成一个二进制应用程序。

* rte.app.mk: DPDK框架中的应用程序。
* rte.extapp.mk: 外部应用程序。
* rte.hostapp.mk: 建立DPDK的先决条件和工具。

#### 31.3.2.2. 库

创建一个 .a 库。

* rte.lib.mk: DPDK中的库。
* rte.extlib.mk: 外部库。
* rte.hostlib.mk: DPDK中的host库。

#### 31.3.2.3. 安装

* rte.install.mk: 不构建任何东西，只是用于创建链接或者将文件复制到安装目录。 这对于开发包框架中包含的文件非常有用。

#### 31.3.2.4. 内核模块

* rte.module.mk: 构建DPDK内核模块。

#### 31.3.2.5. 对象

* rte.obj.mk: DPDK中的目标文件聚合（合并一些o文件成一个）。
* rte.extobj.mk: 外部目标文件聚合（合并外部的一些o文件）。

#### 31.3.2.6. 杂

* rte.doc.mk: DPDK中的文档。
* rte.gnuconfigure.mk: 构建一个基于配置的应用程序。
* rte.subdir.mk: 构建几个目录。

### 31.3.3. 内部生成的构建工具

app/dpdk-pmdinfogen

dpdk-pmdinfogen 扫描各种总所周知的符号名称对象文件。这些目标文件由各种宏定义，用于导出关于pmd文件的硬件支持和使用的重要信息。 例如宏定义：

RTE\_PMD\_REGISTER\_PCI(name, drv)

创建以下的符号：

static char this\_pmd\_name0[] \_\_attribute\_\_((used)) = "<name>";

将被 dpdk-pmdinfogen 扫描。使用这个虚拟系，可以从目标文件中导出其他相关位信息，并用于产生硬件支持描述， 然后 dpdk-pmdinfogen 按照以下格式编码成 json 格式的字符串：

static char <name\_pmd\_string>="PMD\_INFO\_STRING=\"{'name' : '<name>', ...}\"";

然后可以通过外部工具搜索这些字符串，以确定给定库或应用程序的硬件支持。

### 31.3.4. 构建系统提供的有用变量

* RTE\_SDK: DPDK源码绝对路径。 编译DPDK时，该变量由框架自动设置。 如果编译外部应用程序，它必须由用户定义为环境变量。
* RTE\_SRCDIR: DPDK源码根路径。 当编译DPDK时，RTE\_SRCDIR = RTE\_SDK。 当编译外部应用程序时，该变量指向外部应用程序源码的跟目录。
* RTE\_OUTPUT: 输出文件的路径。 通常情况下，他是 $(RTE\_SRCDIR)/build，但是可以通过make命令中的 O= 选项来重新指定。
* RTE\_TARGET: 一个字符串，用于我们正在构建的目标。 格式是arch-machine-execenv-toolchain。 当编译DPDK时，目标是有构建系统从配置(.config)中推导出来的。 当构建外部应用程序时，必须由用户在Makefile中指定或作为环境变量。
* RTE\_SDK\_BIN: 参考 $(RTE\_SDK)/$(RTE\_TARGET)。
* RTE\_ARCH: 定义架构(i686, x86\_64)。 它与 CONFIG\_RTE\_ARCH 相同，但是没有字符串的双引号。
* RTE\_MACHINE: 定义机器。 它与 CONFIG\_RTE\_MACHINE 相同，但是没有字符串的双引号。
* RTE\_TOOLCHAIN: 定义工具链 (gcc , icc)。 它与 CONFIG\_RTE\_TOOLCHAIN 相同，但是没有字符串的双引号。
* RTE\_EXEC\_ENV: 定义运行环境 (linuxapp)。 它与 CONFIG\_RTE\_EXEC\_ENV 相同，但是没有字符串的双引号。
* RTE\_KERNELDIR: 这个变量包含了将被用于编译内核模块的内核源的绝对路径。 内核头文件必须与目标机器（将运行应用程序的机器）上使用的头文件相同。 默认情况下，变量设置为 /lib/modules/$(shell uname -r)/build，当目标机器也是构建机器时，这是正确的。
* RTE\_DEVEL\_BUILD: 更严格的选项（停止警告）。 它在默认的git树中。

### 31.3.5. 只能在Makefile中设置/覆盖的变量

* VPATH: 构建系统将搜索源码的路径列表。默认情况下，RTE\_SRCDIR将被包含在VPATH中。
* CFLAGS: 用于C编译的标志。用户应该使用+=在这个变量中附加数据。
* LDFLAGS: 用于链接的标志。用户应该使用+=在这个变量中附加数据。
* ASFLAGS: 用于汇编的标志。用户应该使用+=在这个变量中附加数据。
* CPPFLAGS: 用于给C预处理器赋予标志的标志（仅在汇编.S文件时有用）。用户应该使用+=在这个变量中附加数据。
* LDLIBS: 在应用程序中，链接的库列表（例如，-L / path / to / libfoo -lfoo）。用户应该使用+=在这个变量中附加数据。
* SRC-y: 在应用程序，库或对象Makefiles的情况下，源文件列表（.c，.S或.o，如果源是二进制文件）。源文件必须可从VPATH获得。
* INSTALL-y-$(INSTPATH): 需要安装到 $(INSTPATH) 的文件列表。 这些文件必须在VPATH中可用，并将复制到 $(RTE\_OUTPUT)/$(INSTPATH)。几乎可以在任何 RTE Makefile 中可用。
* SYMLINK-y-$(INSTPATH): 需要安装到 $(INSTPATH) 的文件列表。 这些文件必须在VPATH中可用并将链接到 (symbolically) 在 $(RTE\_OUTPUT)/$(INSTPATH)。 几乎可以在任何 RTE Makefile 中可用。
* PREBUILD: 构建之前要采取的先决条件列表。用户应该使用+=在这个变量中附加数据。
* POSTBUILD: 主构建之后要执行的操作列表。用户应该使用+=在这个变量中附加数据。
* PREINSTALL: 安装前要执行的先决条件操作的列表。 用户应该使用+=在这个变量中附加数据。
* POSTINSTALL: 安装后要执行的操作列表。用户应该使用+=在这个变量中附加数据。
* PRECLEAN: 清除前要执行的先决条件操作列表。用户应该使用+=在这个变量中附加数据。
* POSTCLEAN: 清除后要执行的先决条件操作列表。用户应该使用+=在这个变量中附加数据。
* DEPDIRS-$(DIR): 仅在开发工具包框架中用于指定当前目录的构建是否依赖于另一个的构建。这是正确支持并行构建所必需的。

### 31.3.6. 只能在命令行上由用户设置/覆盖的变量

一些变量可以用来配置构建系统的行为。在文件 [Development Kit Root Makefile Help](https://link.jianshu.com?t=dev_kit_root_make_help.html" \l "development-kit-root-makefile-help" \t "_blank) 及 [External Application/Library Makefile Help](https://link.jianshu.com?t=ext_app_lib_make_help.html" \l "external-application-library-makefile-help" \t "_blank) 中有描述。

* WERROR\_CFLAGS: 默认情况下，它被设置为一个依赖于编译器的特定值。 鼓励用户使用这个变量，如下所示：

CFLAGS += $(WERROR\_CFLAGS)

这避免了根据编译器（icc或gcc）使用不同的情况。而且，这个变量可以从命令行覆盖，这允许绕过标志用于测试目的。

### 31.3.7. 可以在Makefile或命令行中由用户设置/覆盖的变量

* CFLAGS\_my\_file.o: 为my\_file.c的C编译添加的特定标志。
* LDFLAGS\_my\_app: 链接my\_app时添加的特定标志。
* EXTRA\_CFLAGS: 在编译时，这个变量的内容被附加在CFLAGS之后。
* EXTRA\_LDFLAGS: 链接后，将此变量的内容添加到LDFLAGS之后。
* EXTRA\_LDLIBS: 链接后，此变量的内容被添加到LDLIBS之后。
* EXTRA\_ASFLAGS: 组装后这个变量的内容被附加在ASFLAGS之后。
* EXTRA\_CPPFLAGS: 在汇编文件上使用C预处理器时，此变量的内容将附加在CPPFLAGS之后。

作者：半天妖  
链接：https://www.jianshu.com/p/7aff658c93e8  
来源：简书  
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。