T-Head 曳影 1520

Yocto 用户指南

版本序号	V1.0.0
保密等级	保密
日期	Sept-16-2022

平头哥上海半导体技术有限公司 T-HEAD Semiconductor Co., LTD

地址: 中国(上海)自由贸易试验区上科路 366 号、川和路 55 弄 2 号 5 层

网址: www.t-head.cn

版本历史

版本	说明	作者	日期
V1.0.0	初稿	T-Head	Sept-16-2022

目录

图表目录

术语与缩略语

缩略语	英文全名	中文解释

1 概述

本文将主要介绍 Yocto 的基本用法，指导用户基于 Yocto 完成日常开发。

2 编译环境

Yocto 编译环境使用 Ubuntu 18.04，推荐使用 Linux + docker 的方式部署，因特殊原因无法使用 docker 的，请参考 Ubuntu 环境章节。

2.1 docker 环境

请先安装 Linux 基础 OS，用户根据自己的需要选择 Ubuntu、Centos 等 Linux 发行版本，具体安装方法本文不做详细介绍。安装完成后，在此 OS 的基础上继续安装 docker，然后构建 docker image，将相关的开发环境构建在 docker 里面，后续的开发都基于 docker 内的环境进行，具体的构建方式如下。

2.1.1 安装 docker

使用官方安装脚本自动安装。

```bash
curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
```

2.1.2 下载 dockerfile

点击下载 linux-dev-master.7z 解压后进入 linux-dev-master 目录，打开 Dockerfile，修改用户名和 ID，“your the same user name asyour host”修改成用户 host os 的用户名：

```bash
ENV DOCKER_USER2 "your the same user name asyour host"
```

“your user id”的值记录在/etc/passwd，打开后搜索自己的用户名所在的行，anonuid 字段的值即是对应的 ID。

```
ENV USER2_ID "your user id"
```

2.1.3 构建 image

运行以下命令构建自己的环境。

docker build -t linux-dev-base:base 。

这个 docker 镜像可以编译 thead 发布的 buildroot、yocto 等 Linux SDK。默认密码为 123。

查看构建的 dokcer image，正常情况下可以看到如下结果：

```
@:~$ docker images
REPOSITORY       TAG       IMAGE ID       CREATED          SIZE
linux-dev-base   base      32207ad97b7a   12 minutes ago   2.13GB
ubuntu           18.04     886eca19e611   2 weeks ago      63.1MB
```

2.1.4 启动 docker

docker 启动可以使用 docker run 命令。

```
docker run -u thead -dt --name linux-dev-{your_name}  -v
{your_lock_home}:{your_home} linux-dev-base:base /bin/bash
```

注意：可以通过-v 选项挂载 host 的一个或多个目录，起到类似共享文件的作用，其中：your_name：docker container 名称，起一个自己的名字，不要和别人重名；your_lock_home：host 本地路径；your_home：本地路径在 docker 里的 mount 路径。

查询启动的 docker container：

```
docker ps |grep linux-dev-base
```

正常情况下能看到：

```
@~$ docker ps |grep linux-dev-base
CONTAINER ID   IMAGE                 COMMAND       CREATED
STATUS         PORTS     NAMES
017e0217cab0   linux-dev-base:base   "/bin/bash"   3 minutes ago   Up 3
minutes   22/tcp    linux-dev
```

2.1.5 登录 docker

执行如下命令登录 docker。

```
docker exec -it linux-dev-{your_name} /bin/bash
```

此时已 ssh 登录到 docker 里的 Ubuntu 18.04，运行以下命令查询系统版本：

```
thead@017e0217cab0:/$ lsb_release -a No LSB modules are available.
Distributor ID: Ubuntu Description:    Ubuntu 18.04.6 LTS Release:
 18.04 Codename:       bionic
```

在启动 docker 一节中，对 host 本地$HOME 目录和 docker guest $HOME 做了映射，因此在 docker 的 home/thead 目录下能看 host $HOME 目录下的所有内容，并且所有在 docker 该目录下的文件与 host 完全同步，docker 关闭或删除都不受影响。

2.1.6 镜像迁移

个别环境下，因网络受限等原因，无法从零开始构建完整 docker image，此时可以通过 docker save 和 docker load 命令完成镜像迁移的过程，即在一台网络不受限制的 PC 先构建好 image，再迁移到目标机器上。

具体步骤为先将镜像保存为压缩包，然后在其他位置再加载压缩包。

在网络不受限制的 host 上执行以下命令生成 docker image：

```
cd linux-dev-master docker build -t linux-dev-base:base .
```

查看生成的 docker image，确认 image 生成。

```
docker images linux-dev-base:base
```

开始保存 image。

```
docker save  linux-dev-base:base| gzip >linux-dev-base:base.tar.gz
```

这步后可以在命令执行的当前目录下生成 linux-dev-base:base.tar.gz。

将 tar 包拷贝 linux-dev-base:base.tar.gz 到目标机上，然后用 docker load 命令导入 image。

```
docker load -i linux-dev-base\:base.tar.gz
```

此时在迁移目标机上就可以看到 image 了。

docker images linux-dev-base:base

迁移完毕后，就可以启动 docker container 了，比如：

```
sudo docker run --network=host -u $user -dt --name $user_docker-ubuntu18
-v /home/$user:/home/$user linux-dev-base:base /bin/bash
```

再执行：

```
sudo docker exec -it $user_docker-ubuntu18  /bin/bash
```

2.2 Ubuntu 环境

本章介绍不基于 docker 环境的编译环境搭建方法。

首先安装 Ubuntu 18.04，安装方法请参考 Ubuntu 官方网站，不再赘述。

安装编译 Yocto 所需的依赖包

Bash 复制代码

```
sudo apt-get update
sudo apt-get install -y --assume-yes --no-install-recommends apt-utils
sudo apt-get install -y --assume-yes --no-install-recommends    automake
\
    autotools-dev \
    axel \
    bash \
    bash-completion \
    bc \
    bison \
    build-essential \
    busybox \
    ccache \
    chrpath \
    cpio \
    curl \
    debianutils \
    device-tree-compiler \
    diffstat \
    exuberant-ctags \
    fakeroot \
    file \
    flex \
    g++ \
    g++-multilib \
    gawk \
    gcc \
    gcc-multilib \
    git \
    gnupg \
    gperf \
    iputils-ping \
    less \
    libglib2.0-0 \
    libncurses-dev \
    libncurses5-dev \
    libssl-dev \
    locales \
    lsb-release \
    netcat-openbsd \
    nfs-common \
    openssh-server \
    pkg-config \
    procps \
    pv \
```

```
45        python-pip \
46        python3-pip \
47        rsync \
48        scons \
49        socat \
50        sshfs \
51        sudo \
52        texinfo \
53        tig \
54        tmux \
55        tree \
56        tzdata \
57        unzip \
58        vim \
59        wget \
60        x11-apps \
61        xz-utils \
62        zip \
63        zlib1g-dev \
64        zsh \
65        libevent-dev \
66        libgnutls-dane0 \
67        libgnutls-openssl27 \
68        libgnutls28-dev \
69        libgnutlsxx28 \
70        libidn2-dev \
71        libjsoncpp-dev \
72        liblog4cpp5-dev \
73        libopus-dev \
74        libunistring-dev \
75        libz-dev \
76        nettle-dev \
77        lib32ncurses5-dev \
78        lib32z-dev \
79        libaio-dev \
80        libattr1-dev \
81        libbluetooth-dev \
82        libbrlapi-dev \
83        libc6-dev-i386 \
84        libcap-dev \
85        libgl1-mesa-dev \
86        libglib2.0-dev \
87        liblzo2-dev \
88        libnuma-dev \
89        libpixman-1-dev \
90        libsnappy-dev \
91        libssl-dev \
92        libvdeplug-dev \
```

```
93        libx11-dev \
94        libxml2-utils \
95        openjdk-8-jdk \
96        x11proto-core-dev \
97        xsltproc \
98        libclang-dev \
99        cmake \
100       libvulkan-dev \
101       gtk-update-icon-cache \
102       module-init-tools \
```

3 编译

本章节介绍 Yocto 使用过程中会用到的常用技巧，如需系统性了解 Yocto 知识可以参考如下手册：

- SDK 提供的中文手册《Yocto 开发手册》
- 英文版《BitBake 用户手册》
- 英文版《Yocto 工程参考手册》

3.1 组件

Yocto 以 package 为单位管理海量的开源软件组件，如需编译某个 package，要编译某个 package，方法如下：

```Bash
bitbake “package-name”
```

每个 packege 都会在自己的 recipes 文件中定义支持哪些 task，列出本 package 支持的所有 task 以及 help 信息：

```Bash
bitbake “package-name” -c listtasks
```

3.1.1 源码目录

Yocto 集成了大量的开源 package，这些 package 编译的时候的工作目录通常在以下目录：

- tmp-glibc/work/riscv64-oe-linux
- tmp-glibc/work/${MACHINE}

以 gnome-shell 为例，目录结果通常如下：

```
docker-ubuntu18:thead-build/light-fm/tmp-glibc/work/riscv64-oe-linux/gnome-shell$ tree -L 2
.
└── 3.34.5-r0
    ├── xxx.patch
    ├── build
    ├── debugsources.list
```

其中

- gnome-shell-3.34.5：源码目录
- image：编译输出
- build：编译目录

查询 package 目录的方法：

```
bitbake -e your_package _name  | grep ^S=
```

3.2 Linux 内核

3.2.1 源码路径

Yocto 在编译的时候将源码下载到如下路径下：

```
tmp-glibc/work/d1-oe-linux/linux-thead
```

以 5.10 内核为例，编译过一次内核后，会看到如下目录结构，其中：

- linux-5.10.y：内核源码，带 git 信息
- linux-light_fm_linux-standard-build：编译中间文件等

- temp：编译过程中日志
- image：安装到文件系统的文件

```
└── 5.10.y-r0
    ├── defconfig
    ├── deploy-debs
    ├── deploy-linux-thead
    ├── image
    ├── license-destdir
    ├── linux-5.10.y
    ├── linux-light_fm_linux-standard-build
    ├── package
    ├── packages-split
    ├── pkgdata
    ├── pkgdata-pdata-input
    ├── pkgdata-sysroot
```

3.2.2 构建 Linux 内核

一般修改 Linux 内核源码后，只需要执行该命令即可。

```
$ bitbake linux-thead -C compile
```

3.2.3 清除 Linux 内核

该命令会清除整个 Linux 内核的构建目录，通常不需要执行。

```
$ bitbake linux-thead -c clean
```

注意：clean 会清除 temp/work 下的内核源码。

3.2.4 menuconfig

```
Bash | 复制代码
$ bitbake linux-thead -c menuconfig
```

更多命令可通过 bitbake linux-thead -c listtasks 查看。

3.3 U-Boot

3.3.1 源码路径

Yocto 在编译的时候将源码下载到如下路径下，其中倒数第二级目录名称为 U-Boot 版本号。

```
Bash | 复制代码
tmp-glibc/work/d1-oe-linux/u-boot/1_2018.05-r0/git
```

3.3.2 构建 U-Boot

一般修改 U-Boot 内核源码后，只需要执行该命令即可。

```
Bash | 复制代码
$ bitbake u-boot -C compile
```

3.3.3 清除 U-Boot

该命令会清除整个 U-Boot 内核的构建目录，通常不需要执行。

```
Bash | 复制代码
$ bitbake u-boot -c clean
```

更多命令可通过 bitbake u-boot -c listtasks 查看。

3.4 构建镜像

构建命令如下：

MACHINE=light-a-public-release bitbake light-fm-image-linux

MACHINE 支持列表：

light-a-public-release：曳影 1520 EVB-A 验证板

light-b-public：曳影 1520 EVB-B 验证板

light-beagle：beagle-board 开发板

3.5 镜像打包

预发布的镜像包含了 light-a-public-release，light-b- public，light-beagle 的多种镜像，为了简化烧写镜像包的打包流程，可以用如下仓库的脚本完成相关工作。

cd ~/xuantie-yocto/build/light-fm/sdk

./sdk.sh

对于一般开发者，如果用于镜像烧写，可以减少一些打包行为，只需执行：./sdk.sh no-deb no-tarball

4 添加组件

4.1 查看组件

如查看名字中带有“perf”的组件是否存在：

```
Bash | 复制代码
1  $ bitbake-layers show-recipes *perf*
```

4.2 增加组件

比如要在镜像中增加 perf 命令，方法如下：

第一步：在 image 对应的 bb 文件中增加配置。

例如在 d1-image-miniapp-dev image 中增加 perf 工具，则：

```
Bash | 复制代码
$ vi meta-d1/recipes-core/images/d1-image-miniapp-dev.bb

IMAGE_INSTALL += " perf "
```

第二步：执行“构建镜像”命令，方法见本文 3.4 章节。

第三步：执行“打包”命令，方法见本文上 3.5 章节。

4.3 进入组件目录

增加组件后，如果需要查看组件源码，可以通过如下命令进入组件的构建目录：

```
Bash | 复制代码
$ bitbake perf -c devshell
```

4.4 构建 SYSROOTS SDK

通过本章节构建出来的 SDK，内包含工具链，是用来开发 Linux 应用程序，比如开发运行在 Linux 上的 helloword，就需要这个 SDK。该 SDK 在提供的 SDK 压缩包中已经存在，一般用户直接使用即可，不需要操作本章节步骤。

由于 Yocto 版本有缺陷，构建 SDK 前需要先删除 coreutils，构建完成后再把其改回来。在 openembedded-core 目录下，修改如下（第 9 行）：

```
diff --git a/meta/recipes-core/meta/target-sdk-provides-dummy.bb
b/meta/recipes-core/meta/target-sdk-provides-dummy.bb
index e3beeb796c..765611ffda 100644
--- a/meta/recipes-core/meta/target-sdk-provides-dummy.bb
+++ b/meta/recipes-core/meta/target-sdk-provides-dummy.bb
@@ -4,7 +4,6 @@ DUMMYPROVIDES_PACKAGES = "\
     busybox \
     busybox-dev \
     busybox-src \
-    coreutils \
     coreutils-dev \
     coreutils-src \
     bash \
```

例如，构建 d1-image-miniapp-dev sysroots SDK：

```
$ bitbake d1-image-miniapp-dev -c populate_sdk
```

构建完成后，生成的 SDK 在如下目录，使用方法可见 D1 Linux 快速上手手册。

```
thead-build/d1-miniapp/tmp-glibc/deploy/sdk/
```

4.5 Makefile 例子

使用 tree 可以看到，其有一个 bb 文件，然后其中还有一个目录放着 Makefile 与 source code：

```
meta-thead/recipes-examples/hellomakefile$ tree
.
├── hello
│   ├── helloYocto.c
│   ├── makefile
│   └── zlibtest.c
├── hello.bb
└── README.md

1 directory, 5 files
```

其中的 bb 文件内容如下：

```
DESCRIPTION = "Hello World and Zlib test"
DEPENDS = "zlib"
SECTION = "libs"
LICENSE = "MIT"
PV = "3"
PR = "r0"

SRC_URI = " \
          file://helloYocto.c \
          file://zlibtest.c \
          file://makefile \
```

可以看到，bb 文件中指定了下面几个变量的值：

- SRC_URI：定义源码。
- LIC_FILES_CHKSUM：这个是 checksum，如果是基于像 git 与 svn 的版本管理 source，则不需要定义。
- FILES_$(PN)：PN 是 Package number，指代软件版本使用的 PV 与 PR 结合表示，即前面 bitbake -s 中看到的 3-r0。

除变量外，该 bb 文件重载了有两个方法：

- do_compile
- do_install

这两个方法，对应了 task list 中的 compile 与 install task。

添加软件包之前，确认好以下信息：

- 原始软件在哪里，git 仓库的地址、分支、revision，tar 包的地址、sum 值。
- 是否有额外的 patch、配置文件？
- 使用哪种方式编译，makefile、cmake、meson、ninja、脚本等，确保已经可以单独编译成功。
- 编译的产物是什么，要放到哪里？

4.6 Cmake 例子

以下是一个基于 cmake 的例子，旨在 demo 如何在 Yocto 下，如何集成基于 cmake 的 package。

Bash 复制代码

```
meta-thead/recipes-examples/hellocmake$ tree
.
├── files
│   ├── A
│   │   ├── CMakeLists.txt
│   │   └── main.c
│   ├── B
│   │   ├── add.c
│   │   ├── add.h
│   │   └── CMakeLists.txt
│   ├── C
│   │   ├── CMakeLists.txt
│   │   ├── sub.c
│   │   └── sub.h
│   └── CMakeLists.txt
├── hellocmake.bb
└── README.md

4 directories, 11 files
```

本例子功能如下：

B：c 语言编写的加法函数，编译成动态库，供 C 和 A 使用，即编译出动态库的例子。

C：c 语言编写的减法函数，以及调用 B 中的减法函数例子，即动态库调用动态库的例子。

A：c 语言编写的可执行程序例子，会调用 B 和 C 编译出的动态库，以及一个 ffmpeg 的例子。

编译方法：

Bash 复制代码

```
biatake hellocmake
```

5 其他

其他关于 Yocto 的高级用法，请参考《Yocto 使用技巧》。