

Linux Lab v0.6 中文手册

泰晓科技 | Tinylab.org

February 8, 2021

目录

1. Linux Lab 概览	8
1.1 项目简介	8
1.2 项目主页	8
1.3 演示视频	9
1.3.1 基本操作	9
1.3.2 炫酷操作	9
1.3.3 视频教程	9
1.4 项目功能	9
1.5 项目历史	10
1.5.1 项目起源	10
1.5.2 项目缘由	10
1.5.3 项目诞生	10
1.6 项目变更	11
1.6.1 v0.1 @ 2019.06.28	11
1.6.2 v0.2 @ 2019.10.30	11
1.6.3 v0.3 @ 2020.03.12	11
1.6.4 v0.4 @ 2020.06.01	12
1.6.5 v0.5 @ 2020.09.12	12
1.6.6 v0.6 @ 2021.02.06	13
2. Linux Lab 安装	14
2.1 软硬件要求	14
2.2 安装 Docker	14
2.3 选择工作目录	15
2.4 切换到普通用户帐号	16
2.5 下载实验环境	16
2.6 运行并登录 Linux Lab	17
2.7 更新实验环境并重新运行	18
2.8 快速上手：启动一个开发板	18

3. Linux Lab 入门	19
3.1 使用开发板	19
3.1.1 列出支持的开发板	19
3.1.2 选择一个开发板	21
3.1.3 以插件方式使用	22
3.1.4 配置开发板	23
3.2 一键自动编译	23
3.3 详细步骤分解	24
3.3.1 下载	24
3.3.2 检出	24
3.3.3 打补丁	25
3.3.4 配置	25
3.3.5 编译	26
3.3.6 保存	26
3.3.7 启动	27
4. Linux Lab 进阶	29
4.1 Linux 内核	29
4.1.1 非交互方式配置	29
4.1.2 使用内核模块	30
4.1.3 使用内核特性	31
4.1.4 新建开发分支	32
4.2 Uboot 引导程序	32
4.3 Qemu 模拟器	33
4.4 Toolchain 工具链	34
4.5 Rootfs 文件系统	35
4.6 Linux 与 Uboot 调试	35
4.6.1 调试 Linux	35
4.6.2 调试 Uboot	36
4.7 自动化测试	37
4.8 文件共享	39

4.8.1 在 rootfs 中安装文件	39
4.8.2 采用 NFS 共享文件	39
4.8.3 通过 tftp 传输文件	39
4.8.4 通过 9p virtio 共享文件	40
4.9 学习汇编	41
4.10 学习 C 语言	42
4.10.1 本地编译和运行	42
4.10.2 交叉编译和运行	42
4.11 运行任意的 make 目标	43
4.12 更多用法	43
5. Linux Lab 开发	44
5.1 选择一个 qemu 支持的开发板	44
5.2 创建开发板的目录	44
5.3 从一个已经支持的开发板中复制一份 Makefile	44
5.4 从头开始配置变量	44
5.5 同时准备 configs 文件	44
5.6 选择 kernel, rootfs 和 uboot 的版本	45
5.7 配置, 编译和启动	46
5.8 保存生成的镜像文件和配置文件	46
5.9 上传所有工作	46
6. 常见问题	48
6.1 Docker 相关	48
6.1.1 docker 下载速度慢	48
6.1.2 Docker 网络与 LAN 冲突	48
6.1.3 本地主机不能运行 Linux Lab	48
6.1.4 非 root 无法运行 tools 命令	48
6.1.5 网络不通	49
6.1.6 Client.Timeout exceeded while waiting headers	49
6.1.7 关机或重启主机后如何恢复运行 Linux Lab	50

6.1.8 the following directives are specified both as a flag and in the configuration file	50
6.1.9 pathspec FETCH_HEAD did not match any file known to git	50
6.1.10 Docker not work in Ubuntu 20.04	50
6.1.11 Error creating aufs mount	51
6.2 Qemu 相关	51
6.2.1 缺少 KVM 加速	51
6.2.2 Guest 关机或重启后挂住	51
6.2.3 如何退出 qemu	52
6.2.4 Boot 时报缺少 sdl2 库	52
6.3 环境相关	52
6.3.1 NFS 与 tftpboot 不工作	52
6.3.2 在 vim 中无法切换窗口	53
6.3.3 长按 Backspace 不工作	53
6.3.4 如何快速切换中英文输入	53
6.3.5 如何调节 Web 界面窗口的大小	53
6.3.6 如何进入全屏模式	54
6.3.7 如何录屏	54
6.3.8 Web 界面无响应	55
6.3.9 登录 WEB 界面时超时或报错	55
6.3.10 Ubuntu Snap 问题	56
6.3.11 如何退出 VNC 客户端全屏模式	56
6.4 Linux Lab 相关	56
6.4.1 No working init found	56
6.4.2 linux/compiler-gcc7.h: No such file or directory	56
6.4.3 linux-lab/configs: Permission denied	57
6.4.4 scripts/Makefile.headersinst: Missing UAPI file	57
6.4.5 unable to create file: net/netfilter/xt_dscp.c	57
6.4.6 如何切到 root 用户	57
6.4.7 提示指定的版本或者配置不存在	57
6.4.8 is not a valid rootfs directory	58



Figure 1: Linux Lab Logo

订阅公众号，关注项目状态：



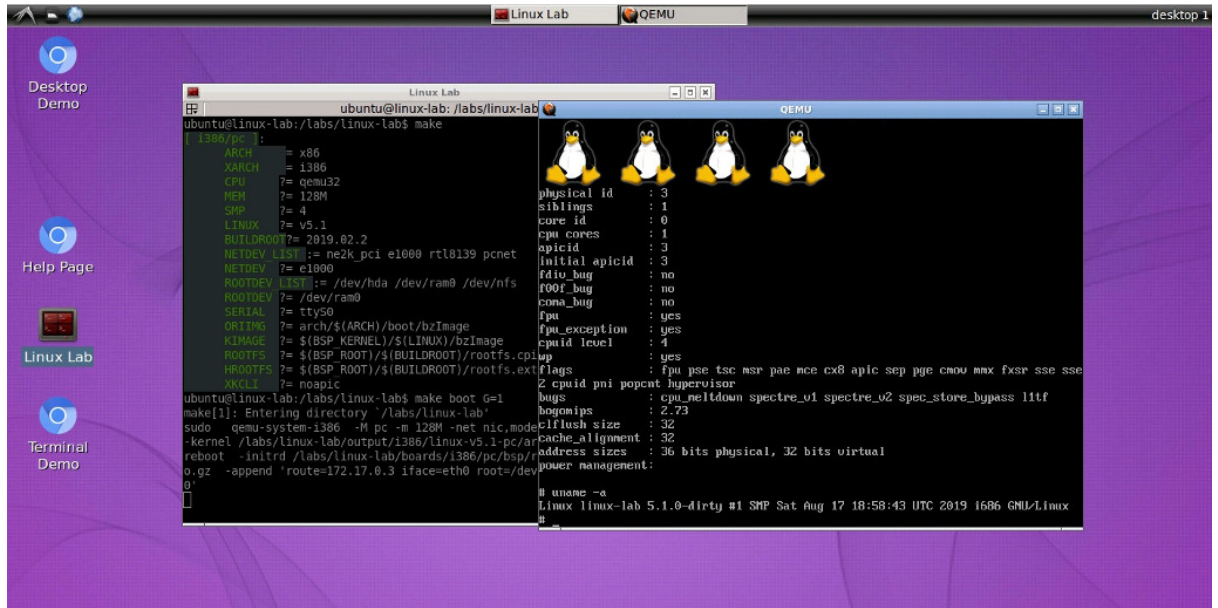
Figure 2: 扫码订阅“泰晓科技”公众号

1. Linux Lab 概览

1.1 项目简介

本项目致力于创建一个基于 Docker + QEMU 的 Linux 实验环境，方便大家学习、开发和测试 Linux 内核。

Linux Lab 是一个开源软件，不提供任何保证，请自行承担使用过程中的任何风险。



1.2 项目主页

- 主页
 - <http://tinylab.org/linux-lab/>
- 仓库
 - <https://gitee.com/tinylab/linux-lab>
 - <https://github.com/tinyclub/linux-lab>

关联项目：

- Cloud Lab
 - Linux Lab 运行环境管理工具
 - <http://tinylab.org/cloud-lab>
- Linux 0.11 Lab
 - 用于 Linux 0.11 学习

- 下载到 `labs/linux-0.11-lab` 后，可直接在 Linux Lab 内使用
- <http://tinylab.org/linux-0.11-lab>
- CS630 Qemu Lab
 - 用于 X86 Linux 汇编学习
 - 下载到 `labs/cs630-qemu-lab` 后，可直接在 Linux Lab 内使用
 - <http://tinylab.org/cs630-qemu-lab>

1.3 演示视频

1.3.1 基本操作

- 基本使用
- 学习 Uboot
- 学习汇编
- 在 Vexpress-a9 开发板上引导启动 ARM Ubuntu 18.04
- 在 ARM64/Virt 开发板上引导启动 Linux v5.1
- 引导启动 Riscv32/virt 和 Riscv64/virt 开发板

1.3.2 炫酷操作

- 一条命令测试某项内核功能
- 一条命令测试多个内核模块
- 批量测试引导启动所有开发板
- 批量测试所有开发板的调试功能

1.3.3 视频教程

- Linux Lab 公开课
 - Linux Lab 简介
 - 龙芯 Linux 内核开发
- 《360° 剖析 Linux ELF》

1.4 项目功能

现在，Linux Lab 已经发展为一个学习、开发和测试 Linux 的集成环境，它支持以下功能：

编号	特性	描述
1	开发板	基于 QEMU，支持 7+ 主流体系架构，15+ 款流行开发板
2	组件	支持 Uboot, Linux, Buildroot, Qemu。支持 Linux v2.6.10 ~ v5.x
3	预置组件	提供上述组件的预先编译版本，并按开发板分类存放，可即时下载使用
4	根文件系统	支持 initrd, harddisk, mmc 和 nfs; ARM 架构提供 Debian 系统
5	Docker	交叉工具链已预先安装，还可灵活配置并下载外部交叉工具链
6	灵活访问	支持通过本地或网络访问，支持 bash, ssh, vnc, web ssh, web vnc
7	网络	内置桥接网络支持，每个开发板都支持网络（Raspi3 是唯一例外）
8	启动	支持串口、Curses（用于 bash/ssh 访问）和图形化方式启动
9	测试	支持通过 <code>make test</code> 命令对目标板进行自动化测试
10	调试	可通过 <code>make debug</code> 命令对目标板进行调试

更多特性和使用方法请看下文介绍。

1.5 项目历史

1.5.1 项目起源

大约九年前，我向 elinux.org 发起了一个 [tinylinux](#) 提案：[Work on Tiny Linux Kernel](#)。该提案最终被采纳，因此我在这个项目上工作了几个月。

1.5.2 项目缘由

在项目开发过程中，编写了几个脚本用于验证一些小特性（譬如：[gc-sections](#)）是否破坏了几个主要的处理器架构上的内核功能。

这些脚本使用 `qemu-system-ARCH` 作为处理器/开发板的模拟器，在模拟器上针对 `Ftrace + Perf` 运行了基本的启动测试和功能测试，并为之相应准备了内核配置文件（`defconfig`）、根文件系统（`rootfs`）以及一些测试脚本。但在当时的条件下，所有的工作只是简单地归档在一个目录下，并没有从整体上将它们组织起来。

1.5.3 项目诞生

这些工作成果在我的硬盘里闲置了好多年，直到有一天我遇到了 `noVNC` 和 `Docker`，并基于这些新技术开发了第一个 [Linux 0.11 Lab](#)，此后，为了将此前开发的那些零散的脚本、内核配置文件、根文件系统和测试脚本整合起来，我开发了 `Linux Lab` 这个系统。

1.6 项目变更

1.6.1 v0.1 @ 2019.06.28

从 2016 年发起，经过数年的开发与迭代，Linux Lab 于 2019 年 6 月 28 日迎来了第 1 个正式版本 v0.1。

- v0.1 rc3
 - 按需加载 prebuilt 并迁移代码仓库到国内，大幅优化了下载体验
- v0.1 rc2
 - 修复了几处基础体验 Bugs
- v0.1 rc1
 - 历史上发布的第 1 个版本，在历史功能上进一步添加了 raspi3 和 risc-v 支持

1.6.2 v0.2 @ 2019.10.30

v0.2 新增原生 Windows 支持、新增龙芯全系支持、新增多个平台外置交叉编译器支持、新增实时 RT 支持、新增 host 侧免 root 支持等，并首次被某线上课程全程采用。

- v0.2 rc3
 - 新增原生 Windows 支持，仅需 Docker，无需安装 Virtualbox 或 Vmware
- v0.2 rc2
 - 龙芯插件新增龙芯教育开发板支持
 - 在 docker 镜像中新增 gdb-multiarch 调试支持，避免为每个平台安装一个 gdb
- v0.2 rc1
 - 携手龙芯实验室，以独立插件的方式新增龙芯全面支持
 - 携手码云，在国内新增 Qemu、U-boot 和 Buildroot 的每日镜像

1.6.3 v0.3 @ 2020.03.12

v0.3 统一了所有组件的公共操作接口更方便记忆，进一步优化了大型仓库的下载体验，通过添加自动依赖关系简化了命令执行并大幅度提升实验效率，为多本知名 Linux 图书新增了 v2.6.10, v2.6.11, v2.6.12, v2.6.14, v2.6.21, v2.6.24 等多个历史版本内核，并发布了首份中文版用户手册。

- v0.3 rc3

- 首次新增中文文档

- v0.3 rc2

- 提升 git 仓库下载体验：所有仓库下载切换为 git init+fetch，更为健壮
- 提升自动化：常规动作都新增了依赖关系，一键自动下载、检出、打补丁、配置、编译、启动

- v0.3 rc1

- 添加多本知名 Linux 图书所用内核支持

1.6.4 v0.4 @ 2020.06.01

v0.4 通过提升镜像下载速度、优化 make 性能、完善登陆方式等进一步完善使用体验，同时首次为 64 位 ARM 架构的 aarch64/virt 新增 Uboot 支持并升级 arm/vexpress-a9 的 Uboot 到当前最新版本，另外，修复了一处新内核下在容器内插入 NFSD 模块导致的系统卡死问题。

- v0.4 rc3

- 新增 aarch64/virt Uboot 支持
- 临时修复新版本内核上容器内插入 NFSD 模块引起的 Sync 卡死问题

- v0.4 rc2

- 新增第 16 块开发板
- 新增 vnc 客户端登陆方法

- v0.4 rc1

- 切换内核镜像到更快的 codeaurora
- 添加本地开发板配置和编辑接口

1.6.5 v0.5 @ 2020.09.12

v0.5 提前升级到新镜像 Ubuntu 20.04，全面导入龙芯系列处理器支持，并进一步完善各种细微体验。

- v0.5 rc3

- 修复 arm/vexpress-a9 因编译器配置问题引起的 Uboot 编译失败
- 进一步完善文档中对普通用户的使用要求，避免使用 root 带来的诸多问题

- v0.5 rc2

- 进一步改善 Qemu 编译体验，在 Gitee 新增 submodules 镜像，不再有挫折感

- 新增 Arch/Manjaro docker 安装文档
- 修复 MacOS 大小写敏感的文件系统镜像制作步骤

- v0.5 rc1

- 全面完善并合并早期对龙芯全系处理器的支持
- 全面升级开发环境基础镜像到 Ubuntu 20.04

1.6.6 v0.6 @ 2021.02.06

v0.6 完善开发镜像，新增首块真实硬件开发板支持。

- v0.6 rc3

- 新增遗漏的 showterm 命令行录制工具
- 进一步完善真实硬件板的支持

- v0.6 rc2

- 新增首块真实硬件开发板 arm/ebf-imx6ull 支持
- 新增 packages-install 目标，允许通过 .labinit 配置环境缺少的软件包
- 新增命令行自动补全脚本，允许直接在命令行补全板子信息，提升使用效率

- v0.6 rc1

- 修复插件中的 BSP 包下载功能
- 修复 x86 架构的内核编译问题
- 修复 aarch64/virt 开发板 Uboot 引导问题

2. Linux Lab 安装

2.1 软硬件要求

Linux Lab 是一套完备的嵌入式 Linux 开发环境，需要预留足够的算力和存储空间，避免后续扩展麻烦，基本硬件配置建议如下：

硬件类型	要求	说明
处理器	X86_64, > 1.5GHz	创建虚拟机时也务必选择 64 位 X86 处理器
磁盘	>= 50G	系统 (25G), Docker 镜像 (~5G), Linux Lab(20G)
内存	>= 4G	过低的内存可能会导致各种卡顿以及异常缓慢

如果平时用的几率比较高，建议把磁盘空间提高到 100G ~ 200G 以上，内存可以提升到 8G 以上。

另外，为了避免走弯路，这里提供了一份验证过的操作系统列表，方便大家参考：

操作系统	系统版本	Docker 版本	内核版本
Ubuntu	16.04, 18.04, 20.04	18.09.4	Linux 4.15, 5.0, 5.3, 5.4
Debian	bullseye	19.03.7	Linux 5.4.42
Arch Linux		19.03.11	Linux 5.4.50, 5.7.4
CentOS	7.6, 7.7	19.03.8	Linux 3.10, 5.2.9
Deepin	15.11	18.09.6	Linux 4.15
Mac OS X	10.15.5	19.03.8	Darwin 19.5.0
Windows	10 PRO, WSL2	19.03.8	MINGW64_NT-10.0-17134
Manjaro		19.03.11	Linux 5.8.3

请查看其他同学 [成功运行过 Linux Lab 的系统](#)，并分享你的情况，例如：

```
1 $ cd /path/to/cloud-lab/
2 $ tools/docker/env
3 System: Ubuntu 16.04.6 LTS
4 Linux: 4.4.0-176-generic
5 Docker: Docker version 18.09.4, build d14af54
```

2.2 安装 Docker

运行 Linux Lab 需要基于 Docker，所以请务必先安装 Docker：

- Linux, Mac OSX, Windows 10
使用 [Docker CE](#)
- 更早的 Windows 版本（含大部分老版本 Windows 10）
通过 Virtualbox 或 Vmware 安装 Ubuntu 后使用

在运行 Linux Lab 之前，请参考 6.1.4 节确保无需 `sudo` 权限也可以正常运行以下命令：

```
1 $ docker run hello-world
```

另外，在国内要正常使用 Docker，请**务必**配置好国内的 Docker 镜像加速服务：

- [阿里云 Docker 镜像使用文档](#)
 - 适合企业和家庭网络，需免费注册帐号并登陆后才能使用
- [USTC Docker 镜像使用文档](#)
 - 适合高校网络

使用 Linux Lab 过程中的常见 Docker 相关问题，请参考常见问题中的 6.1 节，镜像下载慢、下载超时、下载出错等问题都有详细解决方案。

其他问题，请参考 [官方 Docker 文档](#)。

Ubuntu 用户安装手册 - [doc/install/ubuntu-docker.md](#)

Arch 用户安装手册 - [doc/install/arch-docker.md](#)

Manjaro 用户安装手册 - [doc/install/manjaro-docker.md](#)

Windows 用户须知：

- 请参考 [Docker 官方文档](#) 确保所用 Windows 版本支持 Docker
- Linux Lab 当前仅在 Git Bash 验证过，请务必配合 Git Bash 使用
 - 在安装完 [Git For Windows](#) 后，可通过鼠标右键使用“Git Bash Here”

2.3 选择工作目录

如果您是通过 Docker Toolbox 安装，请在 Virtualbox 上进入 `default` 系统的 `/mnt/sda1`，否则，关机后所有数据会丢失，因为缺省的 `/root` 目录是挂载在内存中的。

```
1 $ cd /mnt/sda1
```

对于 Linux 用户，可以简单地在 `~/Downloads` 或 `~/Documents` 下选择一个工作路径。

```
1 $ cd ~/Documents
```

对于 Windows 和 Mac OSX 用户，要正常编译 Linux，请先使能或创建一个区分大小写的文件系统作为工作空间：

Windows：

```
1 (git bash) $ cd /path/to/cloud-lab
2 (git bash) $ fsutil file SetCaseSensitiveInfo ./ enable
```

Mac OSX：

```
1 $ hdiutil create -type SPARSE -size 60g -fs "Case-sensitive Journaled HFS+" -volname
   labspace labspace.dmg
2 $ hdiutil attach -mountpoint ~/Documents/labspace -nobrowse labspace.dmg.sparseimage
3 $ cd ~/Documents/labspace
```

对于 Windows 用户，在安装完 [Git For Windows](#) 后，可通过鼠标右键在选定的工作目录运行“Git Bash Here”。

2.4 切换到普通用户帐号

下载代码前，请务必切到普通用户。Linux Lab 虽未禁用 root 帐号，但是不推荐使用 root 帐号，否则会有各种权限异常问题。

查看当前用户 ID，0 表示 root，非零表示普通用户：

```
1 $ id -u `whoami`
2 1000
```

如果当前为 root，需切到普通用户，请替换 <USER> 为自己的帐号名，下同：

```
1 # id -u `whoami`
2 0
3 # sudo -su <USER>
```

如果目标机器上仅有 root 帐号，则**必须**新建一个普通用户帐号，假设取名为 laber：

```
1 $ sudo useradd --create-home --shell /bin/bash --user-group --groups adm,sudo laber
2 $ sudo passwd laber
3 $ sudo -su laber
4 $ whoami
5 laber
```

2.5 下载实验环境

下载 Cloud Lab，然后再选择 linux-lab 仓库：

```
1 $ git clone https://gitee.com/tinylab/cloud-lab.git
2 $ cd cloud-lab/ && tools/docker/choose linux-lab
```


如果错误使用了 root 帐号来 clone 代码，下载后请**务必**切换到普通用户，并把属主改为普通用户：

```
1 $ sudo -su <USER>
2 $ sudo chown -R <USER>:<USER> -R cloud-lab/{*,.git}
```

2.6 运行并登录 Linux Lab

启动 Linux Lab 并根据控制台上打印的用户名和密码登录实验环境：

```
1 $ tools/docker/run linux-lab
```

通过 Bash 直接登陆：

```
1 $ tools/docker/bash
```

通过 Web 浏览器直接登录实验环境：

```
1 $ tools/docker/webvnc
```

其他登录方式：

```
1 $ tools/docker/vnc
2 $ tools/docker/ssh
3 $ tools/docker/webssh
```

选择某种登陆方式：

```
1 $ tools/docker/login list # 列出并选择，并且记住
2 $ tools/docker/login vnc # 直接选择一种并记住
```

登录方式汇总：

登录方法	描述	缺省用户	登录所在地
bash	docker bash	ubuntu	本地主机
ssh	普通 ssh	ubuntu	本地主机
vnc	普通桌面	ubuntu	本地主机 + VNC client
webvnc	web 桌面	ubuntu	互联网在线即可
webssh	web ssh	ubuntu	互联网在线即可

由于普通的 vnc 客户端五花八门，所以当前建议采用 webvnc，确保可以在各个平台能自动登陆。

如果想使用本地的 vnc 客户端，请先提前安装好客户端，Linux Lab 推荐使用 vinagre。其他的客户端请通过如下方式指定：

```
1 $ tools/docker/vnc vinagre
```

如果上述命令不能正常工作，请根据上述命令打印出来的 VNC 服务器信息，自行配置所用客户端。

2.7 更新实验环境并重新运行

为了更新 Linux Lab 的版本，首先 **必须** 备份所有的本地修改，比如固化容器：

```
1 $ tools/docker/commit linux-lab
2 $ git checkout -- configs/linux-lab/docker/name
```

然后就可以执行更新了：

```
1 $ tools/docker/update linux-lab
```

如果更新失败，请尝试清理当前运行的容器：

```
1 $ tools/docker/rm-all
```

如果有必要的话清理整个环境：

```
1 $ tools/docker/clean-all
```

之后重新运行 Linux Lab：

```
1 $ tools/docker/rerun linux-lab
```

2.8 快速上手：启动一个开发板

进入实验环境，切换目录：

```
1 $ cd /labs/linux-lab
```

输入如下命令，在缺省的 vexpress-a9 开发板上启动预置的内核和根文件系统：

```
1 $ make boot
```

使用 root 帐号登录，不需要输入密码（密码为空），只需要输入 root 然后输入回车即可：

```
1 Welcome to Linux Lab
2
3 linux-lab login: root
4
5 # uname -a
6 Linux linux-lab 5.1.0 #3 SMP Thu May 30 08:44:37 UTC 2019 armv7l GNU/Linux
7 #
8 # poweroff
9 #
```

键入 poweroff 即可关闭板子。部分开发板的关机功能不完善，可通过 CTRL+a x（依次按下 CTRL 和 A，同时释放，再单独按下 x）来退出 Qemu。当然，也可以另开一个控制台，通过 kill 或 pkill 退出 Qemu 进程。

3. Linux Lab 入门

3.1 使用开发板

3.1.1 列出支持的开发板

列出内置支持的开发板:

```

1  $ make list
2  [ aarch64/raspi3 ]:
3      ARCH      = arm64
4      CPU       ?= cortex-a53
5      LINUX     ?= v5.1
6      ROOTDEV_LIST := /dev/mmcblk0 /dev/ram0
7      ROOTDEV   ?= /dev/mmcblk0
8  [ aarch64/virt ]:
9      ARCH      = arm64
10     CPU       ?= cortex-a57
11     LINUX     ?= v5.1
12     ROOTDEV_LIST := /dev/sda /dev/vda /dev/ram0 /dev/nfs
13     ROOTDEV   ?= /dev/vda
14  [ arm/mcimx6ul-evk ]:
15     ARCH      = arm
16     CPU       ?= cortex-a9
17     LINUX     ?= v5.4
18     ROOTDEV_LIST := /dev/mmcblk0 /dev/ram0 /dev/nfs
19     ROOTDEV   ?= /dev/mmcblk0
20  [ arm/versatilepb ]:
21     ARCH      = arm
22     CPU       ?= arm926t
23     LINUX     ?= v5.1
24     ROOTDEV_LIST := /dev/sda /dev/ram0 /dev/nfs
25     ROOTDEV   ?= /dev/ram0
26  [ arm/vexpress-a9 ]:
27     ARCH      = arm
28     CPU       ?= cortex-a9
29     LINUX     ?= v5.1
30     ROOTDEV_LIST := /dev/mmcblk0 /dev/ram0 /dev/nfs
31     ROOTDEV   ?= /dev/ram0
32  [ i386/pc ]:
33     ARCH      = x86
34     CPU       ?= qemu32
35     LINUX     ?= v5.1
36     ROOTDEV_LIST ?= /dev/hda /dev/ram0 /dev/nfs
37     ROOTDEV_LIST[LINUX_v2.6.34.9] ?= /dev/sda /dev/ram0 /dev/nfs
38     ROOTDEV   ?= /dev/hda
39  [ mips64el/ls2k ]:
40     ARCH      = mips
41     CPU       ?= mips64r2
42     LINUX     ?= loongnix-release-1903
43     LINUX[LINUX_loongnix-release-1903] := 04b98684
44     ROOTDEV_LIST := /dev/sda /dev/ram0 /dev/nfs
45     ROOTDEV   ?= /dev/ram0
46  [ mips64el/ls3a7a ]:
47     ARCH      = mips
48     CPU       ?= mips64r2
49     LINUX     ?= loongnix-release-1903
50     LINUX[LINUX_loongnix-release-1903] := 04b98684
51     ROOTDEV_LIST ?= /dev/sda /dev/ram0 /dev/nfs

```

```

52     ROOTDEV ?= /dev/ram0
53 [ mipsel/ls1b ]:
54     ARCH      = mips
55     CPU       ?= mips32r2
56     LINUX     ?= v5.2
57     ROOTDEV_LIST ?= /dev/ram0 /dev/nfs
58     ROOTDEV ?= /dev/ram0
59 [ mipsel/ls232 ]:
60     ARCH      = mips
61     CPU       ?= mips32r2
62     LINUX     ?= v2.6.32-r190726
63     ROOTDEV_LIST := /dev/ram0 /dev/nfs
64     ROOTDEV ?= /dev/ram0
65 [ mipsel/malta ]:
66     ARCH      = mips
67     CPU       ?= mips32r2
68     LINUX     ?= v5.1
69     ROOTDEV_LIST := /dev/hda /dev/ram0 /dev/nfs
70     ROOTDEV ?= /dev/ram0
71 [ ppc/g3beige ]:
72     ARCH      = powerpc
73     CPU       ?= generic
74     LINUX     ?= v5.1
75     ROOTDEV_LIST := /dev/hda /dev/ram0 /dev/nfs
76     ROOTDEV ?= /dev/ram0
77 [ riscv32/virt ]:
78     ARCH      = riscv
79     CPU       ?= any
80     LINUX     ?= v5.0.13
81     ROOTDEV_LIST := /dev/vda /dev/ram0 /dev/nfs
82     ROOTDEV ?= /dev/vda
83 [ riscv64/virt ]:
84     ARCH      = riscv
85     CPU       ?= any
86     LINUX     ?= v5.1
87     ROOTDEV_LIST := /dev/vda /dev/ram0 /dev/nfs
88     ROOTDEV ?= /dev/vda
89 [ x86_64/pc ]:
90     ARCH      = x86
91     CPU       ?= qemu64
92     LINUX     ?= v5.1
93     ROOTDEV_LIST := /dev/hda /dev/ram0 /dev/nfs
94     ROOTDEV_LIST[LINUX_v3.2] := /dev/sda /dev/ram0 /dev/nfs
95     ROOTDEV ?= /dev/ram0
96 [ csky/virt ]:
97     ARCH      = csky
98     CPU       ?= ck810
99     LINUX     ?= v4.9.56
100    ROOTDEV ?= /dev/nfs

```

如果只想查看特定的架构，插件或者模糊匹配，可以使用 ARCH，FILTER:

```

1 $ make list ARCH=arm
2 $ make list FILTER=virt

```

更多用法:

```

1 $ make list-board      # 仅显示 ARCH
2 $ make list-short     # ARCH 和 LINUX
3 $ make list-base      # 不包含插件

```

```
4 $ make list-plugin      # 仅包含插件
5 $ make list-full       # 所有板子信息
6 $ make list-real       # 仅真实硬件
7 $ make list-virt       # 仅虚拟开发板
```

3.1.2 选择一个开发板

3.1.2.1 真实开发板

从 v0.6 版以后，为了方便进一步开展外围设备驱动等实验，Linux Lab 开始支持真实的硬件开发板，选择这类开发板时请务必确保有购买开发板并正确连接到开发主机。

这类开发板可以用 `make list-real` 单独列出来：

```
1 $ make list-real
2 [ arm/ebf-imx6ull ]:
3   ARCH      = arm
4   CPU       ?= cortex-a9
5   LINUX     ?= v4.19.35
6   ROOTDEV_LIST := /dev/mmcblk0 /dev/ram0 /dev/nfs
7   ROOTDEV   ?= /dev/mmcblk0
```

由于不同的真实硬件开发板差异较大，所以在板级目录有提供专门的开发文档，例如：`boards/arm/ebf-imx6ull/README.md`。

所有适配过的开发板会统一放置在如下网店供大家选购，选购完毕后可以加微信号 `tinylab` 中请进入相应的 Linux Lab 开发板群组。

- [泰晓科技自营店](#)

3.1.2.2 虚拟开发板

系统缺省使用的虚拟开发板型号为 `vexpress-a9`，我们也可以自己配置，制作和使用其他的虚拟开发板，具体使用 `BOARD` 选项，举例如下：

```
1 $ make BOARD=malta
2 $ make boot
```

如果存在同名的板子，必须指定架构以示区分，否则系统会默认匹配一个（不一定刚好是你想要的），所以建议明确设定：

```
1 $ make BOARD=mipsel/malta
```

目前同名的有 `virt`，`pc` 等，可以这样查看同名的板子：

```
1 $ make list FILTER=virt
2 [ aarch64/virt ]:
3   ARCH      = arm64
4   CPU       ?= cortex-a57
5   LINUX     ?= v5.1
6   ROOTDEV_LIST := /dev/sda /dev/vda /dev/ram0 /dev/nfs
7   ROOTDEV   ?= /dev/vda
```

```

8 [ riscv32/virt ]:
9     ARCH      = riscv
10    CPU       ?= any
11    LINUX     ?= v5.0.13
12    ROOTDEV_LIST := /dev/vda /dev/ram0 /dev/nfs
13    ROOTDEV ?= /dev/vda
14 [ riscv64/virt ]:
15    ARCH      = riscv
16    CPU       ?= any
17    LINUX     ?= v5.1
18    ROOTDEV_LIST := /dev/vda /dev/ram0 /dev/nfs
19    ROOTDEV ?= /dev/vda
20
21 $ make list FILTER=/pc
22 [ i386/pc ]:
23    ARCH      = x86
24    CPU       ?= qemu32
25    LINUX     ?= v5.1
26    ROOTDEV_LIST ?= /dev/hda /dev/ram0 /dev/nfs
27    ROOTDEV_LIST[LINUX_v2.6.34.9] ?= /dev/sda /dev/ram0 /dev/nfs
28    ROOTDEV ?= /dev/hda
29 [ x86_64/pc ]:
30    ARCH      = x86
31    CPU       ?= qemu64
32    LINUX     ?= v5.1
33    ROOTDEV_LIST := /dev/hda /dev/ram0 /dev/nfs
34    ROOTDEV_LIST[LINUX_v3.2] := /dev/sda /dev/ram0 /dev/nfs
35    ROOTDEV ?= /dev/ram0

```

选择时可以这样：

```

1 $ make BOARD=x86_64/pc
2 $ make BOARD=riscv64/virt

```

如果使用的命令选项是小写的 `board`，这表明创建的开发板的配置不会被保存，提供该选项的目的是为了方便用户同时运行多个开发板而不会相互冲突。

```

1 $ make board=malta boot

```

使用该命令允许在多个不同的终端中或者以后台方式同时运行多个开发板。

检查开发板特定的配置：

```

1 $ cat boards/arm/vexpress-a9/Makefile

```

3.1.3 以插件方式使用

Linux Lab 支持“插件”功能，允许在独立的 git 仓库中添加和维护开发板。采用独立的仓库维护可以确保 Linux Lab 在支持愈来愈多的开发板的同时，自身的代码体积不会变得太大。

该特性有助于支持基于 Linux Lab 学习一些书上的例子以及支持一些采用新的处理器体系架构的开发板，书籍中可能会涉及多个开发板或者是新的处理器架构，并可能会需要多个新的软件包（譬如交叉工具链和架构相关的 qemu 系统模拟器）。

这里列出当前维护的插件：

- [中天微/C-Sky Linux](#)
- [龙芯/Loongson Linux](#)

其中，Loongson 已经在 v5.0 合并进主线。

3.1.4 配置开发板

每块开发板都有特定的配置，部分配置是可以根据需要进行修改的，比如说内存大小、内核版本、文件系统版本、QEMU 版本，以及其他外设配置，比如串口、网络等。

配置方法很简单，参考现有的板级配置（boards/<BOARD>/Makefile）修改即可，以下命令会通过 vim 调出当前开发板的本地配置文件（boards/<BOARD>/labconfig）：

```
1 $ make local-edit
```

建议不要一次性做太大的调整，通常只建议修改内核版本，这样可直接用如下命令达到：

```
1 $ make list-linux
2 v4.12 v4.5.5 v5.0.10 [v5.1]
3 $ make local-config LINUX=v5.0.10
4 $ make list-linux
5 v4.12 v4.5.5 [v5.0.10] v5.1
```

如果想把相关改动提交进上游代码仓库，请使用 board-edit 和 board-config，否则，建议使用 local-edit 和 local-config，这样可以方便同步上游的改动而不产生任何冲突。

3.2 一键自动编译

v0.3 以及之后的版本默认增加了目标依赖支持，所以，如果想编译内核，直接：

```
1 $ make kernel-build
2
3 或
4
5 $ make build kernel
```

它将自动完成所有需要的工作，当然，依然可以跟以前一样手动指定某个目标运行。

更进一步地，通过给每个目标完成情况打上时间戳，完成的目标就不会再运行，从而可以节省时间。如果还想再次执行某个历史目标，可以删掉时间戳文件再运行：

```
1 $ make cleanstamp kernel-build
2 $ make kernel-build
3
4 或
5
6 $ make force-kernel-build
```

下面的命令则删掉所有内核目标的时间戳：

```
1 $ make cleanstamp kernel
```

该功能同样适用于 root, uboot 和 qemu。

3.3 详细步骤分解

3.3.1 下载

下载特定开发板的软件包、内核、buildroot 以及 U-boot 的源码：

```
1 $ make source APP="bsp kernel root uboot"
2 或
3 $ make source APP=all
4 或
5 $ make source all
```

如果需要单独下载这些部分：

```
1 $ make bsp-source
2 $ make kernel-source
3 $ make root-source
4 $ make uboot-source
5
6 或
7
8 $ make source bsp
9 $ make source kernel
10 $ make source root
11 $ make source uboot
```

从 v0.5 开始，所有源代码下载在 Linux Lab 的 `src/` 目录下，历史版本都直接放在 Linux Lab 根目录，请注意该区别。

3.3.2 检出

检出 (checkout) 您需要的 kernel 和 buildroot 版本：

```
1 $ make checkout APP="kernel root"
```

单独检出相关部分：

```
1 $ make kernel-checkout
2 $ make root-checkout
3
4 或
5
6 $ make checkout kernel
7 $ make checkout root
```

如果由于本地更改而导致检出不起作用，请保存更改并做清理以获取一个干净的环境：

```
1 $ make kernel-cleanup
2 $ make root-cleanup
3
4 或
5
6 $ make cleanup kernel
7 $ make cleanup root
```

以上操作也适用于 qemu 和 uboot。

3.3.3 打补丁

给开发板打补丁，补丁包的来源是存放在 `boards/<BOARD>/bsp/patch/linux` 和 `src/patch/linux/` 路径下：

```
1 $ make kernel-patch
2
3 或
4
5 $ make patch kernel
```

3.3.4 配置

3.3.4.1 缺省配置

使用缺省配置（defconfig）配置 kernel 和 buildroot：

```
1 $ make defconfig APP="kernel root"
```

单独配置，缺省情况下使用 `boards/<BOARD>/bsp/` 下的 defconfig：

```
1 $ make kernel-defconfig
2 $ make root-defconfig
3
4 或
5
6 $ make defconfig kernel
7 $ make defconfig root
```

使用特定的 defconfig 配置：

```
1 $ make B=raspb3
2 $ make kernel-defconfig KCFG=bcmrpi3_defconfig
3 $ make root-defconfig KCFG=raspberrypi3_64_defconfig
```

如果仅提供 defconfig 的名字，则搜索所在目录的次序首先是 `boards/<BOARD>`，然后是 `buildroot`，`u-boot` 和 `linux-stable` 各自的缺省配置路径 `src/buildroot/configs`，`src/u-boot/configs` 和 `src/linux-stable/arch/<ARCH>/configs`。

3.3.4.2 手动配置

```
1 $ make kernel-menuconfig
2 $ make root-menuconfig
3
4 或
5
6 $ make menuconfig kernel
7 $ make menuconfig root
```

3.3.4.3 使用旧的缺省配置

```
1 $ make kernel-olddefconfig
2 $ make root-olddefconfig
3 $ make root-olddefconfig
4 $ make uboot-olddefconfig
5
6 或
7
8 $ make olddefconfig kernel
9 $ make olddefconfig root
10 $ make olddefconfig uboot
```

3.3.5 编译

一起编译 kernel 和 buildroot:

```
1 $ make build APP="kernel root"
```

单独编译 kernel 和 buildroot:

```
1 $ make kernel-build # make kernel
2 $ make root-build # make root
3
4 或
5
6 $ make build kernel
7 $ make build root
```

从 v0.5 开始，所有构建结果存放在 Linux Lab 的 build/ 目录下，历史版本都放在 output/ 目录，请注意该区别。

3.3.6 保存

保存所有的配置以及 rootfs/kernel/dtb 的 image 文件:

```
1 $ make saveconfig APP="kernel root"
2 $ make save APP="kernel root"
```

保存配置和 image 文件到 boards/<BOARD>/bsp/:

```
1 $ make kernel-saveconfig
2 $ make root-saveconfig
3 $ make root-save
4 $ make kernel-save
5
6 或
7
8 $ make saveconfig kernel
9 $ make saveconfig root
10 $ make save kernel
11 $ make save root
```

3.3.7 启动

缺省情况下采用非图形界面的串口方式启动，如果要退出可以使用 CTRL+a x, poweroff, reboot 或 pkill qemu 命令（具体参考 6.2.2 节）

```
1 $ make boot
```

图形方式启动（如果要退出请使用 CTRL+ALT+2 quit）:

```
1 $ make b=pc boot G=1 LINUX=v5.1
2 $ make b=versatilepb boot G=1 LINUX=v5.1
3 $ make b=g3beige boot G=1 LINUX=v5.1
4 $ make b=malta boot G=1 LINUX=v2.6.36
5 $ make b=vexpress-a9 boot G=1 LINUX=v4.6.7 // LINUX=v3.18.39 works too
```

注意：真正的图形化方式启动需要 LCD 和键盘驱动的支持，上述开发板可以完美支持 Linux 内核 5.1 版本的运行，raspi3 和 malta 两款开发板支持 tty0 终端但不支持键盘输入。

vexpress-a9 和 virt 缺省情况下不支持 LCD，但对于最新的 qemu，可以通过在启动时指定 G=1 参数然后通过选择“View”菜单切换到串口终端，但这么做无法用于测试 LCD 和键盘驱动。我们可以通过 XOPTS 选项指定额外的 qemu 选项参数。

```
1 $ make b=vexpress-a9 CONSOLE=ttyAMA0 boot G=1 LINUX=v5.1
2 $ make b=raspi3 CONSOLE=ttyAMA0 XOPTS="-serial vc -serial vc" boot G=1 LINUX=v5.1
```

基于 curses 图形方式启动（这么做适合采用 bash/ssh 的登录方式，但不是对所有开发板都有效，退出时需要使用 ESC+2 quit 或 ALT+2 quit）

```
1 $ make b=pc boot G=2 LINUX=v4.6.7
```

使用预编译的内核、dtb 和 Rootfs 启动：

```
1 $ make boot PBK=1 PBD=1 PBR=1
2 or
3 $ make boot k=old d=old r=old
4 or
5 $ make boot kernel=old dtb=old root=old
```

使用新的内核、dtb 和 rootfs 启动：

```
1 $ make boot PBK=0 PBD=0 PBR=0
2 or
3 $ make boot k=new d=new r=new
4 or
5 $ make boot kernel=new dtb=new root=new
```

如果目标内核和 Uboot 不存在，重新编译一个之后再启动：

```
1 $ make boot BUILD="kernel uboot"
```

使用 Uboot 启动（目前仅测试并支持了 versatilepb 和 vexpress-a9 两款开发板）:

```
1 $ make boot U=0
```

使用不同的 rootfs 启动（依赖于开发板的支持，启动后检查 /dev/）

```
1 $ make boot ROOTDEV=/dev/ram // support by all boards, basic boot method
2 $ make boot ROOTDEV=/dev/nfs // depends on network driver, only raspi3 not work
3 $ make boot ROOTDEV=/dev/sda
4 $ make boot ROOTDEV=/dev/mmcblk0
5 $ make boot ROOTDEV=/dev/vda // virtio based block device
```

使用额外的内核命令行参数启动（格式：XKCLI = eXtra Kernel Command Line）：

```
1 $ make boot ROOTDEV=/dev/nfs XKCLI="init=/bin/bash"
```

列出支持的选项：

```
1 $ make list ROOTDEV
2 $ make list BOOTDEV
3 $ make list CCORI
4 $ make list NETDEV
5 $ make list LINUX
6 $ make list UBOOT
7 $ make list QEMU
```

使用 list <xxx> 可以实现更多 <xxx>-list，例如：

```
1 $ make list features
2 $ make list modules
3 $ make list gcc
```

4. Linux Lab 进阶

4.1 Linux 内核

4.1.1 非交互方式配置

Linux 内核提供了一个脚本 `scripts/config`，可用于非交互方式获取或设置内核的配置选项值。基于该脚本，实验环境增加了两个选项 `kernel-getconfig` 和 `kernel-setconfig`，可用于调整内核的选项。基于该功能我们可以方便地实现类似“enable/disable/setstr/setval/getstate”内核选项的操作。

获取一个内核模块的状态：

```
1 $ make kernel-getconfig m=minix_fs
2 Getting kernel config: MINIX_FS ...
3
4 build/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m
```

使能一个内核模块：

```
1 $ make kernel-setconfig m=minix_fs
2 Setting kernel config: m=minix_fs ...
3
4 build/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m
5
6 Enable new kernel config: minix_fs ...
```

更多 `kernel-setconfig` 命令的控制选项：y, n, c, o, s, v：

选项	说明
y	编译内核中的模块或者使能其他内核选项
c	以插件方式编译内核模块，类似 m 选项
o	以插件方式编译内核模块，类似 m 选项
n	关闭一个内核选项
s	RTC_SYSTOHC_DEVICE="rtc0"，设置 rtc 设备为 rtc0
v	v=PANIC_TIMEOUT=5，设置内核 panic 超时为 5 秒

在一条命令中使用多个选项：

```
1 $ make kernel-setconfig m=tun,minix_fs y=ikconfig v=panic_timeout=5 s=DEFAULT_HOSTNAME=
   linux-lab n=debug_info
2 $ make kernel-getconfig o=tun,minix,ikconfig,panic_timeout,hostname
```

4.1.2 使用内核模块

编译所有的内部内核模块：

```
1 $ make modules
2 $ make modules-install
3 $ make root-rebuild // not need for nfs boot
4 $ make boot
```

列出 `src/modules/` 和 `boards/<BOARD>/bsp/modules/` 路径下的所有模块：

```
1 $ make modules-list
```

如果加上 `m` 参数，除了列出 `src/modules/` 和 `boards/<BOARD>/bsp/modules/` 路径下的所有模块外，还会列出 `src/linux-stable/` 下的所有模块：

```
1 $ make modules-list m=hello
2     1 m=hello ; M=$PWD/src/modules/hello
3 $ make modules-list m=tun,minix
4     1 c=TUN ; m=tun ; M=drivers/net
5     2 c=MINIX_FS ; m=minix ; M=fs/minix
```

使能一个内核模块：

```
1 $ make kernel-getconfig m=minix_fs
2 Getting kernel config: MINIX_FS ...
3
4 build/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m
5
6 $ make kernel-setconfig m=minix_fs
7 Setting kernel config: m=minix_fs ...
8
9 build/aarch64/linux-v5.1-virt/.config:CONFIG_MINIX_FS=m
10
11 Enable new kernel config: minix_fs ...
```

编译一个内核模块（例如：`minix.ko`）

```
1 $ make modules M=fs/minix/
2 或
3 $ make modules m=minix
```

安装和清理模块：

```
1 $ make modules-install M=fs/minix/
2 $ make modules-clean M=fs/minix/
```

其他用法：

```
1 $ make kernel-setconfig m=tun
2 $ make kernel x=tun.ko M=drivers/net
3 $ make kernel x=drivers/net/tun.ko
4 $ make kernel-run drivers/net/tun.ko
```

编译外部内核模块（类似编译内部模块）：

```
1 $ make modules m=hello
2 或
3 $ make kernel x=$PWD/src/modules/hello/hello.ko
```

4.1.3 使用内核特性

内核的众多特性都集中存放在 `src/feature/linux/`，其中包括了特性的配置补丁，可以用于管理已合入内核主线的特性和未合入的特性功能。

```
1 $ make feature-list
2 [ /labs/linux-lab/src/feature/linux ]:
3 + 9pnet
4 + core
5   - debug
6   - module
7 + ftrace
8   - v2.6.36
9     * env.g3beige
10    * env.malta
11    * env.pc
12    * env.versatilepb
13   - v2.6.37
14    * env.g3beige
15 + gcs
16   - v2.6.36
17    * env.g3beige
18    * env.malta
19    * env.pc
20    * env.versatilepb
21 + kft
22   - v2.6.36
23    * env.malta
24    * env.pc
25 + uksm
26   - v2.6.38
```

这里列出了针对某项特性验证时使用的内核版本，如果其他条件未改变的话该特性应该可以正常工作。

例如，为了启用内核模块支持，可以执行如下简单的操作：

```
1 $ make feature f=module
2 $ make kernel-olddefconfig
3 $ make kernel
```

为了在 malta 开发板上验证基于 2.6.36 版本的 kft 特性，可以执行如下操作：

```
1 $ make BOARD=malta
2 $ export LINUX=v2.6.36
3 $ make kernel-checkout
4 $ make kernel-patch
5 $ make kernel-defconfig
6 $ make feature f=kft
7 $ make kernel-olddefconfig
8 $ make kernel
9 $ make boot
```

4.1.4 新建开发分支

如果希望新建一个分支来做开发，那么可以参考如下步骤。

首先在 `src/linux-stable` 或配置的其他 `KERNEL_SRC` 目录下基于某个内核版本新建一个 git 分支，假设历史版本是 v5.1：

```
1 $ cd src/linux-stable
2 $ git checkout -b linux-v5.1-dev v5.1
```

然后通过 `kernel-clone` 从 Linux Lab 的 v5.1 克隆一份配置和相应目录：

```
1 $ make kernel-clone LINUX=v5.1 LINUX_NEW=linux-v5.1-dev
```

之后就可以跟往常一样开发。

如果基础版本不是 v5.1，那么可以从支持的版本中挑选一个比较接近的，以 `i386/pc` 为例：

```
1 $ make b=i386/pc list linux
2 v2.6.10 v2.6.11.12 v2.6.12.6 v2.6.21.5 v2.6.24.7 v2.6.34.9 v2.6.35.14 v2.6.36 v4.6.7 [v5
  .1] v5.2
```

例如，想进行 v2.6.38 开发，可以考虑从 v2.6.36 来克隆，就近的配置更接近，出问题可能更少。

```
1 $ cd src/linux-stable
2 $ git checkout -b linux-v2.6.38-dev v2.6.38
3
4 $ make kernel-clone LINUX=v2.6.36 LINUX_NEW=linux-v2.6.38-dev
```

开发过程中，请及时 `commit`，另外，请慎重使用如下命令，避免清除重要变更：

- `kernel-checkout`，检出某个指定版本，可能会覆盖掉当前修改
- `kernel-cleanup`，清理 git 仓库，可能会清理掉当前修改
- `kernel-clean`，清除历史编译记录，可能会自动执行上述 `cleanup` 动作

4.2 Uboot 引导程序

从当前支持 U-boot 的板子：`versatilepb` 和 `vexpress-a9` 中选择一款：

```
1 $ make BOARD=vexpress-a9
```

下载 Uboot：

```
1 $ make uboot-source
```

检出一个特定的版本（版本号在 `boards/<BOARD>/Makefile` 中通过 `UBOOT` 指定）：

```
1 $ make uboot-checkout
```


应用必要的补丁修改，可以指定 `BOOTDEV` 和 `ROOTDEV` 两个选项设置，如果不指定则缺省值使用 `flash`。

```
1 $ make uboot-patch
```

如果要明确指定值为 `tftp`, `sdcard` 或 `flash`，则必须在输入 `uboot-patch` 之前运行 `make uboot-checkout`：

```
1 $ make uboot-patch BOOTDEV=tftp
2 $ make uboot-patch BOOTDEV=sdcard
3 $ make uboot-patch BOOTDEV=flash
```

`BOOTDEV` 用于设定 `uboot` 的存放设备以便从该设备引导，`ROOTDEV` 用于告诉内核从哪里加载 `rootfs`。

配置 U-boot：

```
1 $ make uboot-defconfig
2 $ make uboot-menuconfig
```

编译 U-boot：

```
1 $ make uboot
```

使用 `BOOTDEV` 和 `ROOTDEV` 引导，缺省采用 `flash` 方式：

```
1 $ make boot U=1
```

显式使用 `tftp`, `sdcard` 或 `flash` 方式：

```
1 $ make boot U=1 BOOTDEV=tftp
2 $ make boot U=1 BOOTDEV=sdcard
3 $ make boot U=1 BOOTDEV=flash
```

我们也可以在启动引导阶段改变 `ROOTDEV` 选项，例如：

```
1 $ make boot U=1 BOOTDEV=flash ROOTDEV=/dev/nfs
```

执行清理，更新 `ramdisk`, `dtb` 和 `uImage`：

```
1 $ make uboot-images-clean
2 $ make uboot-clean
```

保存 `uboot` 镜像和配置：

```
1 $ make uboot-save
2 $ make uboot-saveconfig
```

4.3 Qemu 模拟器

内置的 `qemu` 或许不能和最新的 Linux 内核配套工作，为此我们有时不得不自己编译 `qemu`，自行编译 `qemu` 的方法在 `vexpress-a9` 和 `virt` 开发板上已经验证通过。

首先，编译 `qemu-system-ARCH`：

```
1 $ make B=vexpress-a9
2
3 $ make qemu-download
4 $ make qemu-checkout
5 $ make qemu-patch
6 $ make qemu-defconfig
7 $ make qemu
8 $ make qemu-save
```

qemu-ARCH-static 和 qemu-system-ARCH 是不能一起编译的，为了制作 qemu-ARCH-static，请在开发板的 Makefile 中首先使能 QEMU_US=1 然后再重新编译。

如果指定了 QEMU 和 QTOOL，那么实验环境会优先使用 bsp 子模块中的 QEMU 和 QTOOL，而不是已经安装在本地系统中的版本，但会优先使用最近编译的版本，如果最近有编译过的话。

在为新的内核实现移植时，如果使用 2.5 版本的 QEMU，Linux 5.0 在运行过程中会挂起，将 QEMU 升级到 2.12.0 后，问题消失。请在以后内核升级过程中注意相关的问题。

Qemu 每次编译都会检查子仓库是否较新，但是下载通常没那么顺利。如果下载过一次子仓库以后不想再更新，可以通过如下方式禁止更新：

```
1 $ make qemu git_module_status=0
```

4.4 Toolchain 工具链

Linux 内核主线的升级非常迅速，内置的工具链可能无法与其保持同步，为了减少维护上的压力，环境支持添加外部工具链。譬如 ARM64/virt, CCVER 和 CCPATH。

列出支持的预编译工具链：

```
1 $ make gcc-list
```

下载，解压缩和使能外部工具链：

```
1 $ make gcc
```

切换编译器版本，例子如下：

```
1 $ make gcc-switch CCORl=internal GCC=4.7
2
3 $ make gcc-switch CCORl=linaro
```

如果未指定外部工具链，则缺省使用内置的工具链。

如果不存在内置的工具链，则必须指定外部工具链。当前对该特性已经支持 aarch64, arm, riscv, mipsel, ppc, i386, x86_64 多个体系架构。

GCC 的版本可以分别在开发板特定的 Makefile 中针对 Linux, Uboot, Qemu 和 Root 分别指定：

```
1 GCC[LINUX_v2.6.11.12] = 4.4
```

采用以上配置方法，在编译 v2.6.11.12 版本的 Linux 内核时会在 defconfig 时自动切换为使用指定的 GCC 版本。

在编译主机 (host) 的软件时，也需要做相应配置 (需要显式指定 b=i386/pc)：

```
1 $ make gcc-list b=i386/pc
2 $ make gcc-switch CCOR1=internal GCC=4.8 b=i386/pc
```

4.5 Rootfs 文件系统

内置的 rootfs 很小，不足以应付复杂的应用开发，如果需要涉及高级的应用开发，需要使用现代的 Linux 发布包。

环境提供了针对 arm32v7 的 Ubuntu 18.04 的根文件系统，该文件系统已经制作成 Docker 镜像，以后有机会再提供更多更好的文件系统。

可以通过 Docker 直接使用：

```
1 $ docker run -it tinylab/arm32v7-ubuntu
```

可以将文件系统提取出来在 Linux Lab 中使用：

```
1 (host)$ sudo apt-get install -y qemu-user-static
```

ARM32/vexpress-a9 (用户名和密码均为 root)：

```
1 (host)$ tools/root/docker/extract.sh tinylab/arm32v7-ubuntu arm
2 (lab )$ make boot B=vexpress-a9 U=0 V=1 MEM=1024M ROOTDEV=/dev/nfs ROOTFS=$PWD/prebuilt/
    fullroot/tmp/tinylab-arm32v7-ubuntu
```

ARM64/raspi3 (用户名和密码均为 root)：

```
1 (host)$ tools/root/docker/extract.sh tinylab/arm64v8-ubuntu arm
2 (lab )$ make boot B=raspi3 V=1 ROOTDEV=/dev/mmcbk0 ROOTFS=$PWD/prebuilt/fullroot/tmp/
    tinylab-arm64v8-ubuntu
```

其他 Docker 中更多的根文件系统：

```
1 $ docker search arm64 | egrep "ubuntu|debian"
2 arm64v8/ubuntu   Ubuntu is a Debian-based Linux operating system 25
3 arm64v8/debian  Debian is a Linux distribution that's composed 20
```

4.6 Linux 与 Uboot 调试

4.6.1 调试 Linux

使用调试选项编译内核：

```
1 $ make feature f=debug
2 $ make kernel-olddefconfig
3 $ make kernel
```

编译时使用一个线程：

```
1 $ make kernel JOBS=1
```

如果是通过 vnc 或者 webvnc 登陆的，可运行如下命令直接调试：

```
1 $ make debug
```

将打开一个新的终端窗口，从 `.gdb/kernel.default` 加载脚本，自动运行 gdb。

但是，如果是通过 bash、ssh 或者 webssh 登陆的，请通过相应方式再登陆一次 Linux Lab，再次运行如下命令即可：

```
1 $ make debug
```

主要原因是，控制台不支持多窗口，需要手动开多个窗口，方便一边运行，一边调试。

如果想修改调试脚本，可以拷贝一份到 `.gdb/kernel.user`，这样就可以无缝升级：

```
1 $ cp .gdb/kernel.default .gdb/kernel.user
```

以上命令等价于运行如下命令：

```
1 $ make debug linux
2 或
3 $ make boot DEBUG=linux
```

自动测试调试可以运行如下命令：

```
1 $ make test-debug linux
2 或
3 $ make test DEBUG=linux
```

找出内核崩溃出错地址所在的代码行：

```
1 $ make kernel-calltrace func+offset/length
```

4.6.2 调试 Uboot

如果想调试 Uboot（采用 `.gdb/uboot.default` 调试脚本）：

```
1 $ make debug uboot
2 或
3 $ make debug DEBUG=uboot
```

如果是通过控制台登陆的 Linux Lab，不会自动拉起窗口，请按提示再次执行如下命令即可开启调试：

```
1 $ make debug uboot
```

同样可以自动测试调试：

```
1 $ make test-debug uboot
2 或
3 $ make test DEBUG=uboot
```

同样地，如果想修改调试脚本，可以拷贝一份到 `.gdb/uboot.user`，这样就可以无缝升级：

```
1 $ cp .gdb/uboot.default .gdb/uboot.user
```

4.7 自动化测试

以 `aarch64/virt` 作为演示的开发板：

```
1 $ make BOARD=virt
```

为测试做准备，在 `src/system/` 目录下安装必要的文件/脚本：

```
1 $ make rootdir
2 $ make root-install
3 $ make root-rebuild
```

直接引导启动（参考 6.2.2 节）

```
1 $ make test
```

测试完毕后不要关机：

```
1 $ make test TEST_FINISH=echo
```

运行一下客户机的测试用例：

```
1 $ make test TEST_CASE=/tools/ftrace/trace.sh
```

运行客户机的测试用例（`COMMAND_LINE_SIZE` 必须足够大，譬如，4096，查看下文的 `cmdline_size` 特性）

```
1 $ make test TEST_BEGIN=date TEST_END=date TEST_CASE='ls /root,echo hello world'
```

进行重启压力测试：

```
1 $ make test TEST_REBOOT=2
```

注意：reboot 可以有以下几种结果 1) 挂起，2) 继续；3) 超时后被杀死，`TEST_TIMEOUT=30`；4) 超时终止后不报错继续其他测试，`TIMEOUT_CONTINUE=1`

在一个特定的开发板上测试一个特定 Linux 版本的某个功能（`cmdline_size` 特性用于增加 `COMMAND_LINE_SIZE` 为 4096）：

```
1 $ make test f=kft LINUX=v2.6.36 b=malta TEST_PREPARE=board-init,kernel-cleanup
```

注意： board-init 和 kernel-cleanup 用于确保测试自动运行，但是 kernel-cleanup 不安全，请在使用前保存代码！

测试一个内核模块：

```
1 $ make test m=hello
```

测试多个内核模块：

```
1 $ make test m=exception,hello
```

基于指定的 ROOTDEV 测试模块，缺省使用 nfs 引导方式，但注意有些开发板可能不支持网络：

```
1 $ make test m=hello,exception TEST_RD=/dev/ram0
```

在测试内核模块时运行测试用例（在 insmod 和 rmmod 命令之间运行测试用例）：

```
1 $ make test m=exception TEST_BEGIN=date TEST_END=date TEST_CASE='ls /root,echo hello world' TEST_PREPARE=board-init,kernel-cleanup f=cmdline_size
```

在测试内部内核模块时运行测试用例：

```
1 $ make test m=lkdtm TEST_BEGIN='mount -t debugfs debugfs /mnt' TEST_CASE='echo EXCEPTION ">" /mnt/provoke-crash/DIRECT'
```

在测试内部内核模块时运行测试用例，传入内核参数：

```
1 $ make test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

测试时不使用 feature-init（若非必须可以节省时间）

```
1 $ make test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION' TEST_INIT=0
2 或
3 $ make raw-test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

测试模块以及模块的依赖（使用 make kernel-menuconfig 进行检查）：

```
1 $ make test m=lkdtm y=runtime_testing_menu,debug_fs lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

测试时不使用 feature-init，boot-init，boot-finish 以及不带 TEST_PREPARE：

```
1 $ make boot-test m=lkdtm lkdtm_args='cpoint_name=DIRECT cpoint_type=EXCEPTION'
```

测试一个内核模块并且在测试前执行某些 make 目标：

```
1 $ make test m=exception TEST=kernel-checkout,kernel-patch,kernel-defconfig
```

使用一条命令测试所有功能（从下载到关机，如果关机后挂起，请参考 6.2.2）：

```
1 $ make test TEST=kernel,root TEST_PREPARE=board-init,kernel-cleanup,root-cleanup
```

使用一条命令测试所有功能（带 uboot，如果支持的话，譬如：vexpress-a9）：

```
1 $ make test TEST=kernel,root,uboot TEST_PREPARE=board-init,kernel-cleanup,root-cleanup,uboot-cleanup
```

测试引导过程中内核挂起，允许指定超时时间，系统挂起时将发生超时：

```
1 $ make test TEST_TIMEOUT=30s
```

测试过程中如果超时，继续执行后续测试，而不是直接终止：

```
1 $ make test TEST_TIMEOUT=30s TIMEOUT_CONTINUE=1
```

测试内核调试：

```
1 $ make test DEBUG=1
```

4.8 文件共享

缺省支持如下方法在 Qemu 开发板和主机之间传输文件：

4.8.1 在 rootfs 中安装文件

将文件放在 `src/system/` 的相对路径中，安装和重新制作 rootfs：

```
1 $ mkdir src/system/root/
2 $ touch src/system/root/new_file
3 $ make root-install
4 $ make root-rebuild
5 $ make boot
```

上述操作在 `root` 用户目录下新增 `new_file` 文件。

4.8.2 采用 NFS 共享文件

使用 `ROOTDEV=/dev/nfs` 选项启动开发板：

```
1 $ make boot ROOTDEV=/dev/nfs
```

主机 NFS 目录如下：

```
1 $ make env-dump VAR=ROOTDIR
2 ROOTDIR="/labs/linux-lab/boards/<BOARD>/bsp/root/<BUILDRoot_VERSION>/rootfs"
```

4.8.3 通过 tftp 传输文件

在 Qemu 开发板上运行 `tftp` 命令访问主机的 tftp 服务器。

主机侧：

```
1 $ ifconfig br0
2 inet addr:172.17.0.3 Bcast:172.17.255.255 Mask:255.255.0.0
3 $ cd tftpboot/
4 $ ls tftpboot
5 kft.patch kft.log
```

Qemu 开发板：

```
1 $ ls
2 kft_data.log
3 $ tftp -g -r kft.patch 172.17.0.3
4 $ tftp -p -r kft.log -l kft_data.log 172.17.0.3
```

注意：当把文件从 Qemu 开发板发送到主机侧时，必须先在主机上创建一个空的文件，这是一个 bug?!

4.8.4 通过 9p virtio 共享文件

有关如何为一个新的开发板启用 9p virtio，请参考 [qemu 9p setup](#)。编译 qemu 时必须使用 `--enable-virtfs` 选项，同时内核必须打开必要的选项。

重新配置内核如下：

```
1 CONFIG_NET_9P=y
2 CONFIG_NET_9P_VIRTIO=y
3 CONFIG_NET_9P_DEBUG=y (Optional)
4 CONFIG_9P_FS=y
5 CONFIG_9P_FS_POSIX_ACL=y
6 CONFIG_PCI=y
7 CONFIG_VIRTIO_PCI=y
8 CONFIG_PCI_HOST_GENERIC=y (only needed for the QEMU Arm 'virt' board)
```

如果需要使用 qemu 的 `-virtfs` 或 `-device virtio-9p-pci` 选项，需要使能以上 PCI 相关的选项，否则无法工作：

```
1 9pnet_virtio: no channels available for device hostshare
2 mount: mounting hostshare on /hostshare failed: No such file or directory
```

`-device virtio-9p-device` 需要较少的内核选项。

为了使能以上选项，请输入以下命令：

```
1 $ make feature f=9pnet
2 $ make kernel-olddefconfig
```

Docker 主机：

```
1 $ modprobe 9pnet_virtio
2 $ lsmod | grep 9p
3 9pnet_virtio          17519  0
4 9pnet                 72068  1 9pnet_virtio
```

主机：


```
1 $ make BOARD=virt
2
3 $ make root-install      # Install mount/umount scripts, ref: src/system/etc/init.d/
   S50sharing
4 $ make root-rebuild
5
6 $ touch hostshare/test   # Create a file in host
7
8 $ make boot U=0 ROOTDEV=/dev/ram0 PBR=1 SHARE=1
9
10 $ make boot SHARE=1 SHARE_DIR=src/modules # for external modules development
11
12 $ make boot SHARE=1 SHARE_DIR=build/aarch64/linux-v5.1-virt/ # for internal modules
   learning
13
14 $ make boot SHARE=1 SHARE_DIR=src/examples # for c/assembly learning
```

Qemu 开发板：

```
1 $ ls /hostshare/        # Access the file in guest
2 test
3 $ touch /hostshare/guest-test # Create a file in guest
```

使用 Linux v5.1 验证过的开发板：

开发板	支持状态
aarch64/virt	virtio-9p-device (virtio-9p-pci 导致 nfsroot 不工作)
arm/vexpress-a9	仅支持 virtio-9p-device
arm/versatilepb	仅支持 virtio-9p-pci
x86_64/pc	仅支持 virtio-9p-pci
i386/pc	仅支持 virtio-9p-pci
riscv64/virt	同时支持 virtio-9p-pci 和 virtio-9p-dev
riscv32/virt	同时支持 virtio-9p-pci 和 virtio-9p-dev

4.9 学习汇编

Linux Lab 在 `src/examples/assembly` 目录下有许多汇编代码的例子：

```
1 $ cd src/examples/assembly
2 $ ls
3 aarch64 arm mips64el mipsel powerpc powerpc64 README.md x86 x86_64
4 $ make -s -C aarch64/
5 Hello, ARM64!
```

4.10 学习 C 语言

4.10.1 本地编译和运行

以 hello 为例：

```

1 $ cd src/examples/c/hello
2 $ make
3 gcc -fno-stack-protector -fomit-frame-pointer -fno-asynchronous-unwind-tables -fno-pie -
   no-pie -m32 -Wall -Werror -g -o hello hello.c
4 Hello, World!

```

4.10.2 交叉编译和运行

下面简单介绍如何在 Linux Lab 下交叉编译和并运行 C 程序，以 X32 (code for x86-64, int/long/pointer to 32bits), ARM, MIPS, PPC 和 RISCv 为例：

```

1 $ sudo apt-get update -y
2
3 $ sudo apt-get install -y libc6-x32 libc6-dev-x32 libx32gcc-8-dev
4 $ gcc -mx32 -o hello hello.c
5 $ ./hello
6 Hello, World!
7
8 $ sudo apt-get install -y libc6-dev-armel-cross libc6-armel-cross
9 $ arm-linux-gnueabi-gcc -o hello hello.c
10 $ qemu-arm -L /usr/arm-linux-gnueabi/ ./hello
11 Hello, World!
12
13 $ sudo apt-get install -y libc6-dev-mipsel-cross libc6-mipsel-cross
14 $ mipsel-linux-gnu-gcc -o hello hello.c
15 $ qemu-mipsel -L /usr/mipsel-linux-gnu/ ./hello
16 Hello, World!
17
18 $ sudo apt-get install -y libc6-dev-powerpc-cross libc6-powerpc-cross
19 // Linux Lab v0.6 中，必须加 -static，否则运行时会有段错误
20 $ powerpc-linux-gnu-gcc -static -o hello hello.c
21 $ qemu-ppc -L /usr/powerpc-linux-gnu/ ./hello
22 Hello, World!
23
24 $ sudo apt-get install -y libc6-riscv64-cross libc6-dev-riscv64-cross
25 $ riscv64-linux-gnu-gcc -o hello hello.c
26 $ qemu-riscv64 -L /usr/riscv64-linux-gnu/ ./hello
27 Hello, World!

```

上面是通过 `qemu-user` 做指令翻译运行，如果要在目标板子上运行，参考 4.8.1 节复制到对应板子的文件系统即可。

主要的包是 `libc6-dev`, `libc6` 以及 `libgcc`, `x32` 是个例外，包名是 `libx32gcc`。可以通过 `apt-cache search` 检索更详细的列表。

4.11 运行任意的 make 目标

Linux Lab 支持通过形如 `<xxx>-run` 方式访问 Makefile 中定义的目标，譬如：

```
1 $ make kernel-run help
2 $ make kernel-run menuconfig
3
4 $ make root-run help
5 $ make root-run busybox-menuconfig
6
7 $ make uboot-run help
8 $ make uboot-run menuconfig
```

执行这些带有 `-run` 的目标允许我们无需进入相关的构造目录就可以直接运行这些 `make` 目标来制作 `kernel`、`rootfs` 和 `uboot`。

4.12 更多用法

欢迎阅读下述文档学习更多用法：

- [Linux Lab：难以抗拒的十大理由 V1.0](#)
- [Linux Lab：难以抗拒的十大理由 V2.0](#)
- [Linux Lab 龙芯实验手册 V0.2](#)
- Linux Lab 视频公开课
 - [CCTALK](#)
 - [B 站](#)
 - [知乎](#)
- 采用 Linux Lab 作为实验环境的视频课程
 - [《360° 剖析 Linux ELF》](#)

5. Linux Lab 开发

本节介绍如何从头开始为 Linux Lab 添加一块新的开发板。

5.1 选择一个 qemu 支持的开发板

列出支持的开发板，以 arm 架构为例：

```
1 $ qemu-system-arm -M ?
```

5.2 创建开发板的目录

以 vexpress-a9 为例：

```
1 $ mkdir boards/arm/vexpress-a9/
```

5.3 从一个已经支持的开发板中复制一份 Makefile

以 versatilepb 为例：

```
1 $ cp boards/arm/versatilepb/Makefile boards/arm/vexpress-a9/Makefile
```

5.4 从头开始配置变量

先注释掉所有的配置项，然后逐个打开获得一个最小的可工作配置集，最后再添加其他配置。

具体参考 [doc/qemu/qemu-doc.html](#) 或在线说明。

5.5 同时准备 configs 文件

我们需要为 Linux，buildroot 甚至 uboot 准备 config 文件。

Buildroot 已经为 buildroot 和内核配置提供了许多例子：

```
1 buildroot: src/buildroot/configs/qemu_ARCH_BOARD_defconfig
2 kernel: src/buildroot/board/qemu/ARCH-BOARD/linux-VERSION.config
```

Uboot 也提供了许多缺省的配置文件：

```
1 uboot: src/u-boot/configs/vexpress_ca9x4_defconfig
```

内核本身也提供了缺省的配置：

```
1 kernel: src/linux-stable/arch/arm/configs/vexpress_defconfig
```

Linux Lab 也提供许多有效的配置，`xxx-clone` 命令有助于利用现有的配置：

```
1 $ make list kernel
2 v4.12 v5.0.10 v5.1
3 $ make kernel-clone LINUX=v5.1 LINUX_NEW=v5.4
4 $ make kernel-menuconfig
5 $ make kernel-saveconfig
6
7 $ make list root
8 2016.05 2019.02.2
9 $ make root-clone BUILDROOT=2019.02.2 BUILDROOT_NEW=2019.11
10 $ make root-menuconfig
11 $ make root-saveconfig
```

编辑配置文件和 Makefile 直到它们满足我们的需要。

```
1 $ make kernel-menuconfig
2 $ make root-menuconfig
3 $ make board-edit
```

配置文件必须放在 `boards/<BOARD>/` 目录下并且在命名上需要注明必要的版本信息，以 `raspi3` 为例：

```
1 $ make kernel-saveconfig
2 $ make root-saveconfig
3 $ ls boards/aarch64/raspi3/bsp/configs/
4 buildroot_2019.02.2_defconfig linux_v5.1_defconfig
```

2019.02.2 是 buildroot 的版本，v5.1 是内核版本，这两个变量需要在 `boards/<BOARD>/Makefile` 中设置好。

更多 clone 命令的用法如下：

```
1 $ make qemu-clone QEMU=<old_version> QEMU_NEW=<new_version>
2 $ make uboot-clone UBOOT=<old_version> UBOOT_NEW=<new_version>
3 $ make kernel-clone LINUX=<old_version> LINUX_NEW=<new_version>
4 $ make root-clone BUILDROOT=<old_version> BUILDROOT_NEW=<new_version>
```

5.6 选择 kernel, rootfs 和 uboot 的版本

检出版本时请使用 `tag` 命令而非 `branch` 命令，以 kernel 为例：

```
1 $ cd src/linux-stable
2 $ git tag
3 ...
4 v5.0
5 ...
6 v5.1
7 ..
8 v5.1.1
9 v5.1.5
10 ...
```

如果我们需要的是 v5.1 的 kernel，那么可以在 `boards/<BOARD>/Makefile` 添加一行：`LINUX = v5.1`。

或者从旧的版本或者是官方的 defconfig 文件中复制一份内核的配置：

```
1 $ make kernel-clone LINUX_NEW=v5.3 LINUX=v5.1
2
3 或
4
5 $ make B=i386/pc
6 $ pushd src/linux-stable && git checkout v5.4 && popd
7 $ make kernel-clone LINUX_NEW=v5.4 KCFG=i386_defconfig
```

如果不存在对应的 tag，可以直接使用 commit 号同时为它模拟一个 tag 名字，配置方法如下：

```
1 LINUX = v2.6.11.12
2 LINUX[LINUX_v2.6.11.12] = 8e63197f
```

可以配置和 Linux 版本对应的 ROOTFS：

```
1 ROOTFS[LINUX_v2.6.12.6] ?= $(BSP_ROOT)/$(BUILDRoot)/rootfs32.cpio.gz
```

5.7 配置，编译和启动

以 kernel 为例：

```
1 $ make kernel-defconfig
2 $ make kernel-menuconfig
3 $ make kernel
4 $ make boot
```

同样的方法适用于 rootfs，uboot，甚至 qemu。

5.8 保存生成的镜像文件和配置文件

```
1 $ make root-save
2 $ make kernel-save
3 $ make uboot-save
4
5 $ make root-saveconfig
6 $ make kernel-saveconfig
7 $ make uboot-saveconfig
```

5.9 上传所有工作

最后，将 images、defconfigs、patchset 上传到开发板特定的 bsp 子模块仓库。

首先，获取远端 bsp 仓库的地址，方法如下：

```
1 $ git remote show origin
2 * remote origin
3 Fetch URL: https://gitee.com/tinylab/qemu-aarch64-raspi3/
4 Push URL: https://gitee.com/tinylab/qemu-aarch64-raspi3/
5 HEAD branch: master
```

```
6 Remote branch:
7   master tracked
8 Local branch configured for 'git pull':
9   master merges with remote master
10 Local ref configured for 'git push':
11   master pushes to master (local out of date)
```

然后，在 gitee.com 上 fork 这个仓库，上传您的修改，然后发送您的 pull request。

6. 常见问题

6.1 Docker 相关

6.1.1 docker 下载速度慢

为了优化 Docker 镜像的下载速度，请参考 `tools/docker/install` 脚本的内容编辑 `/etc/default/docker` 中的 `DOCKER_OPTS` 以及 6.1.6 节。

6.1.2 Docker 网络与 LAN 冲突

假设 docker 网络为 `10.66.0.0/16`，否则，最好采用如下方式对其进行更改：

```
1 $ sudo vim /etc/default/docker
2 DOCKER_OPTS="$DOCKER_OPTS --bip=10.66.0.10/16"
3
4 $ sudo vim /lib/systemd/system/docker.service
5 ExecStart=/usr/bin/dockerd -H fd:// --bip=10.66.0.10/16
```

请重新启动 docker 服务和 lab 容器以使更改生效：

```
1 $ sudo service docker restart
2 $ tools/docker/rerun linux-lab
```

如果 Linux Lab 的网络仍然无法正常工作，请尝试使用另一个专用网络地址，并最终避免与 LAN 地址冲突。

6.1.3 本地主机不能运行 Linux Lab

Linux Lab 的完整功能依赖于 [Cloud Lab](#) 所管理的完整 docker 环境，因此，请切勿尝试脱离 [Cloud Lab](#) 在本地主机上直接运行 Linux Lab，否则系统会报告缺少很多依赖软件包以及其他奇怪的错误。

Linux Lab 的设计初衷是旨在通过利用 docker 技术使用预先安装好的环境来避免在不同系统中的软件包安装问题，从而加速我们上手的时间，因此 Linux Lab 暂无计划支持在本地主机环境下使用。

6.1.4 非 root 无法运行 tools 命令

如果需要在不使用 `sudo` 的情况下执行 `tools` 目录下的命令，请确保将您的帐户添加到 docker 组并重新启动系统以使其生效：

```
1 $ sudo usermod -aG docker <USER>
2 $ newgrp docker
```


6.1.5 网络不通

如果无法 ping 通，请根据下面列举的方法逐一排查：

- DNS 问题

如果 ping 8.8.8.8 工作正常，请检查 `/etc/resolv.conf` 并确保其与主机配置相同。

- IP 问题

如果 ping 不起作用，请参阅 6.1.2 并更改 docker 容器的 ip 地址范围。

6.1.6 Client.Timeout exceeded while waiting headers

解决方法是选择配置以下 Docker 镜像服务站点中的一个：

- [阿里云 Docker 镜像使用文档](#)
- [USTC Docker 镜像使用文档](#)

Ubuntu 系统下，请根据不同版本情况选择下述**某一种**方法进行 Mirror 站点配置：

`/etc/docker/daemon.json:`

```
1 {  
2   "registry-mirrors": ["<your accelerate address>"]  
3 }
```

`/lib/systemd/system/docker.service:`

```
1 ExecStart=/usr/bin/dockerd -H fd:// --bip=10.66.0.10/16 --registry-mirror=<your  
   accelerate address>
```

`/etc/default/docker:`

```
1 DOCKER_OPTS="\${DOCKER_OPTS} --registry-mirror=<your accelerate address>"
```

注意：以上三种方式不要同时配置，请选择适合 Docker 版本的方式选一种即可，新的 Linux 发行版一般都用 `/etc/docker/daemon.json`。

配置完需要重启 docker 服务才能生效：

```
1 $ sudo service docker restart
```

对于其他 Linux 系统，Windows 和 MacOS 系统，建议优先参考 [阿里云 Docker 镜像使用文档](#)。

如果添加镜像后速度依然很慢，请仔细检查是否配置成功或者是否打错了地址：

```
1 $ docker info | grep -A1 -i Mirrors  
2 Registry Mirrors:  
3   https://XXXXX.mirror.aliyuncs.com/
```

6.1.7 关机或重启主机后如何恢复运行 Linux Lab

如果要恢复容器中已经安装的软件和添加的各类配置，请事先保存好容器：

```
1 $ tools/docker/commit linux-lab
```

在关机或者重启主机（或虚拟机）系统后，通常可以通过点击桌面的“Linux Lab”图标恢复运行，或者通过命令行像第一次运行那样：

```
1 $ tools/docker/run linux-lab
```

当前实现不支持通过 `docker start` 恢复容器，请知悉！

如果上述方式无法恢复，请根据情况执行 6.3.9 节中的相应步骤。

如果是从休眠中的主机（或虚拟机）系统唤醒，那么 Linux Lab 也会自动恢复，可以直接使用，登陆方式请参考 2.4 节中提供的 4 种登陆方式。例如，直接开一个浏览器去使用：

```
1 $ tools/docker/vnc
```

6.1.8 the following directives are specified both as a flag and in the configuration file

如果运行 docker 时遇到如下错误：

```
1 unable to configure the Docker daemon with file /etc/docker/daemon.json: the
2 following directives are specified both as a flag and in the configuration
3 file: registry-mirrors: (from flag: [https://docker.mirrors.ustc.edu.cn/], from
4 file: [https://xxx.mirror.aliyuncs.com])
```

说明同时在 `/etc/docker/daemon.json` 和 `/etc/default/docker` 中配置了 `registry-mirrors`，请注释掉后面的配置后重启 Docker 服务即可。

```
1 $ sudo service docker restart
```

6.1.9 pathspec FETCH_HEAD did not match any file known to git

如果在 `make boot` 时遇到如下错误，说明容器内网络可能不通，请参考 6.1.5 节。

```
1 Could not resolve host: gitee.com
2 error: pathspec 'FETCH_HEAD' did not match any file(s) known to git
```

6.1.10 Docker not work in Ubuntu 20.04

如果在 Ubuntu 20.04 下 Docker 不工作，请尝试使用 `doc/install/daemon.json` 并清理 `dockerd` 的默认参数，更多内容请参考 [docker daemon](#)：

```
1 $ sudo cat /etc/systemd/system/docker.service.d/docker.conf
2 [Service]
3 ExecStart=
4 ExecStart=/usr/bin/dockerd
5
6 $ sudo cp /etc/docker/daemon.json /etc/docker/daemon.json.bak
7 $ sudo cp doc/install/daemon.json /etc/docker/
8 $ sudo service docker restart
```

注意：记得把 registry-mirrors 配置为你希望使用的加速器地址。

6.1.11 Error creating aufs mount

如果遇到类似错误：“error creating aufs mount to ... invalid arguments”，那意味着当前配置的 docker 存储驱动不被支持，可以从 [Storage Driver](#) 选配一个，例如：

```
1 $ sudo vim /etc/docker/daemon.json
2 {
3   "registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"],
4   "storage-driver": "devicemapper"
5 }
```

这里主要是跟内核版本有关，同样的系统升级内核后，Storage Driver 类型可能得相应调整。

6.2 Qemu 相关

6.2.1 缺少 KVM 加速

KVM 当前仅支持 qemu-system-i386 和 qemu-system-x86_64，并且还需要 cpu 和 bios 支持，否则，您可能会看到以下错误日志：

```
1 modprobe: ERROR: could not insert 'kvm_intel': Operation not supported
```

检查 cpu 的虚拟化支持能力，如果没有输出，则说明 cpu 不支持虚拟化：

```
1 $ cat /proc/cpuinfo | egrep --color=always "vmx|svm"
```

如果 cpu 支持，我们还需要确保在 BIOS 中启用了该功能，只需重新启动计算机，按“Delete”键进入 BIOS，请确保“Intel virtualization technology”功能已启用。

6.2.2 Guest 关机或重启后挂住

当前对以下开发板，基于内核版本 5.1 (LINUX=v5.1)，poweroff 和 reboot 命令无法正常工作：

- mipsel/malta (exclude LINUX=v2.6.36)
- mipsel/lsl1b

- mipsel/ls232
- mips64el/ls2k
- mips64el/ls3a7a
- aarch64/raspi3
- arm/versatilepb

在运行 `poweroff` 或 `reboot` 时，系统会直接挂起，为了退出 `qemu`，请使用 `CTRL+a x` 或执行 `shell` 命令 `pkill qemu`。

为了自动化测试这些开发板，请确保设置 `TEST_TIMEOUT`，例如：`make test TEST_TIMEOUT=50`。

欢迎提供修复意见。

6.2.3 如何退出 qemu

停留界面	退出方式
串口控制台	<code>CTRL+a x</code>
基于 Curses 的图形终端	<code>ESC+2 quit</code> 或 <code>ALT+2 quit</code>
基于 X 图形终端	<code>CTRL+ALT+2 quit</code>
通用方法	<code>poweroff</code> , <code>reboot</code> , <code>kill</code> , <code>pkill</code>

6.2.4 Boot 时报缺少 sdl2 库

这是由于 `docker` 的 `image` 没有更新导致，解决的方法是重新运行 `lab`：

```
1 $ tools/docker/pull linux-lab
2 $ tools/docker/rerun linux-lab
3
4 或
5
6 $ tools/docker/update linux-lab
```

使用 `tools/docker/update`，所有的 `docker images` 和源码都会被更新，这是推荐的做法。

6.3 环境相关

6.3.1 NFS 与 tftpboot 不工作

如果 `NFS` 或 `tftpboot` 不起作用，请在主机端运行 `modprobe nfsd` 并通过运行 `/configs/tools/restart-net-servers.sh` 重新启动网络服务，请确保不要使用 `tools/docker/trun`。

6.3.2 在 vim 中无法切换窗口

浏览器和 vim 中都提供了 CTRL+w，为了避免冲突，要从一个窗口切换到另一个窗口，请改用 CTRL+Left 或 CTRL+Right 键，Linux Lab 已将 CTRL+Right 映射为 CTRL+w，将 CTRL+Left 映射为 CTRL+p。

6.3.3 长按 Backspace 不工作

长按键目前在 Web 界面中不起作用，因此，长按“Delete”或“Backspace”键不起作用，请改用 alt+delete 或 alt+backspace 组合键，以下是更多有关组合键的小技巧：

说明	Vim	Bash
行首/行尾	^/\$	Ctrl + a/e
前进/后退一个字	w/b	Ctrl + Home/end
向后剪切一个字	db	Alt + Delete/backspace
向前剪切一个字	dw	Alt + d
剪切光标前所有	d~	Ctrl + u
剪切光标后所有	d\$	Ctrl + k
粘帖剪切的内容	p	Ctrl + y

6.3.4 如何快速切换中英文输入

为了切换英文/中文输入法，请使用 CTRL+s 快捷键，而不是 CTRL+space，以避免与本地系统冲突。

6.3.5 如何调节 Web 界面窗口的大小

有两种方式可以调节 Web 界面窗口的大小，一种是通过 noVNC 左侧边栏设置 Scaling Mode，这样屏幕就可以自适应；另外一种是在启动时指定。

先来介绍第一种，也就是屏幕自适应，这种方法很方便，但是屏幕字体由于拉伸变形后略微有些发虚。

```
1 * 点击 noVNC Web 页面左侧的边栏（左侧有个小箭头）
2 * 断开连接
3 * 点击设置：'Settings -> Scaling Mode: -> Local Scaling -> Apply'
4 * 重新连接
```

接下来介绍第二种方法，即在启动 Lab 时指定一个合适的分辨率。

Linux Lab 的屏幕尺寸是由 xrandr 捕获的，如果不起作用，请检查并自行设置，例如：

获取可用的屏幕尺寸值：

```
1 $ xrandr --current
2 Screen 0: minimum 1 x 1, current 1916 x 891, maximum 16384 x 16384
3 Virtual1 connected primary 1916x891+0+0 (normal left inverted right x axis y axis) 0mm x
   0mm
4   1916x891      60.00*+
5   2560x1600     59.99
6   1920x1440     60.00
7   1856x1392     60.00
8   1792x1344     60.00
9   1920x1200     59.88
10  1600x1200     60.00
11  1680x1050     59.95
12  1400x1050     59.98
13  1280x1024     60.02
14  1440x900      59.89
15  1280x960      60.00
16  1360x768      60.02
17  1280x800      59.81
18  1152x864      75.00
19  1280x768      59.87
20  1024x768      60.00
21  800x600       60.32
22  640x480       59.94
```

更新屏幕尺寸：

```
1 $ cd /path/to/cloud-lab
2 $ tools/docker/resize 1280x1024 # 指定任意一个尺寸
3 $ tools/docker/resize          # 不带参数则设定为主系统同样的屏幕尺寸
```

如果需要做到全屏，可按如下步骤操作：

1. 如果用到虚拟机，先把虚拟机设置为全屏模式
2. 然后执行：`tools/docker/resize`，把 Lab 屏幕大小设定为主机系统屏幕大小
3. 进入到浏览器的 WebVNC 界面，点击左边栏的 `FullScreen` 按钮即可放大

6.3.6 如何进入全屏模式

打开左边的侧边栏，点击“Fullscreen”按钮。

6.3.7 如何录屏

1. 使能录制

打开左侧边栏，按“Settings”按钮，配置“File/Title/Author/Category/Tags/Description”，然后启用“Record Screen”选项。

2. 开始录制

按下“Connect”按钮。

3. 停止录制

按下“Disconnect”按钮。

4. 重放录制的视频

按下“Play”按钮。

5. 分享视频

视频存储在“cloud-lab/recordings”目录下，参考 showdesk.io 的帮助进行分享。

6.3.8 Web 界面无响应

Web 连接可能由于某些未知原因而挂起，导致 Linux Lab 有时可能无法响应，要恢复该状态，请点击 Web 浏览器的刷新按钮或断开连接后重新连接。

6.3.9 登录 WEB 界面时超时或报错

如果登陆 WEB 界面时出现“Disconnect timeout”，请稍等片刻后继续点击左侧“Connect”按钮，如果依然无法成功，请按下述步骤检查。

首先检查 linux-lab 需要的 docker 容器是否正常启动（Up: 正常，Exit: 为不正常）：

```
1 $ docker ps -a
2 CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
3 19a61ba075b5 tinylab/linux-lab "/tools/lab/run" 4 days ago Up 4 days 22/tcp, 5900/tcp
   linux-lab-21575
4 75dae89984c9 tinylab/cloud-ubuntu-web "/startup.sh" 8 days ago Up 8 days ...443/tcp
   cloud-ubuntu-web
```

如果为 Exit，可能是关机后容器自动关闭了，也可能是容器启动失败，如果是容器自动关闭了可以通过如下命令启动：

```
1 $ tools/docker/run linux-lab
```

如果依然无法启动，请检查并分析启动日志：

```
1 $ tools/docker/logs linux-lab
```

如果日志正常，那说明之前保存的帐号和密码可能由于某次异常失效了，需要参考如下步骤重新生成新的帐号和密码。

注意：下述 clean 和 rerun 命令会清理一些容器和数据，请自行做好相应备份，例如固化容器：

```
1 $ tools/docker/commit linux-lab
```

使用不匹配的密码时会导致 Web 登录失败，要解决此问题，请清理环境并重新运行。

```
1 $ tools/docker/clean linux-lab
2 $ tools/docker/rerun linux-lab
```

如果上述命令依然无法启动，请尝试执行下述命令（该命令会整理整个 Cloud Lab 环境，请务必做好必要数据备份）：

```
1 $ tools/docker/clean-all
2 $ tools/docker/rerun linux-lab
```

6.3.10 Ubuntu Snap 问题

用户报告了许多 snap 相关的问题，请改用 apt-get 安装 docker：

- 无法将普通用户添加到 docker 用户组从而导致必须通过 root 用户使用 docker。
- snap 服务会耗尽 /dev/loop 设备从而导致无法挂载文件系统。

6.3.11 如何退出 VNC 客户端全屏模式

在进入 VNC 客户端的全屏模式后，不同客户端软件在不同系统上的退出方式可能五花八门，甚至有些可能有 Bug，如果想切回主机，又没有便捷的方式，就会让人抓狂。

这个时候，就推荐下面的方式，理论上，与 VNC 客户端软件无关，那就是在 Linux Lab 内杀掉 VNC 服务：

```
1 $ sudo pkill x11vnc
```

由于 Linux Lab 会自动恢复掉线的 x11vnc 服务，所以完全不会影响下次登陆。

6.4 Linux Lab 相关

6.4.1 No working init found

这意味着 rootfs.ext2 文件可能已损坏，请删除该文件，然后再次尝试执行 make boot，例如：

```
1 $ rm boards/aarch64/raspi3/bsp/root/2019.02.2/rootfs.ext2
2 $ make boot
```

make boot 命令可以自动创建该映像，请不要中途打断。

6.4.2 linux/compiler-gcc7.h: No such file or directory

这意味着您使用的 gcc 版本不为当前 Linux 内核所支持，可使用 make gcc-switch 命令切换到较旧的 gcc 版本，以 i386 / pc 开发板为例：

```
1 $ make gcc-list
2 $ make gcc-switch CCOR=internal GCC=4.4
```


6.4.3 linux-lab/configs: Permission denied

这个错误会在执行 `make boot` 时报出，原因可能是由于克隆代码仓库时使用了 `root` 权限，解决方式是修改 `cloud-lab/` 目录的所有者：

```
1 $ cd /path/to/cloud-lab
2 $ sudo chown <USER>:<USER> -R ./{*,.git}
3 $ tools/docker/rerun linux-lab
```

为确保环境一致，目前 Linux Lab 仅支持通过普通用户使用，如果是用 `root` 用户下载的代码，请务必确保普通用户可以读写。

6.4.4 scripts/Makefile.headersinst: Missing UAPI file

这是因为 MAC OSX 缺省的文件系统不区分大小写，请使用 `hdiutil` 或 `Disk Utility` 自己创建一个：

```
1 $ hdiutil create -type SPARSE -size 60g -fs "Case-sensitive Journaled HFS+" -volname
   labspace labspace.dmg
2 $ hdiutil attach -mountpoint ~/Documents/labspace -nobrowse labspace.dmg.sparseimage
3 $ cd ~/Documents/labspace
```

6.4.5 unable to create file: net/netfilter/xt_dscp.c

这是因为 Windows 没有使能文件系统的大小写支持，通过 Git Bash 开启它：

```
1 $ cd /path/to/cloud-lab
2 $ fsutil file SetCaseSensitiveInfo ./ enable
```

也可以通过管理员打开 `cmd`，然后执行：

```
1 $ fsutil.exe file SetCaseSensitiveInfo <path/to/cloud-lab> enable
```

6.4.6 如何切到 root 用户

默认情况下，可以免密直接切到 `root`：

```
1 $ sudo -s
```

6.4.7 提示指定的版本或者配置不存在

如果看到如下信息：

```
1 $ make boot ROOTDEV=/dev/vda
2 ERR: /dev/vda not in supported ROOTDEV list: /dev/sda /dev/ram0 /dev/nfs, update may help
   : 'make bsp B=mips64el/ls3a7a'. Stop.
3
4 $ make boot LINUX=v5.8
```

```
5 | Makefile:594: *** ERR: v5.8 not in supported LINUX list: loongnix-release-1903 v5.7,  
   | clone one please: 'make kernel-clone KERNEL_NEW=v5.8'. Stop.  
6  
7 | $ make boot QEMU=loongson-v1.1  
8 | Makefile:606: *** ERR: loongson-v1.1 not in supported QEMU list: loongson-v1.0, clone one  
   | please: 'make qemu-clone QEMU_NEW=loongson-v1.1'.
```

表示当前设置的变量值无效，例如：

- 当前设定的版本还未支持或者还未验证和添加
 - 可以通过 `xxx-clone` 命令克隆出一个新版本，目前支持：`qemu-clone`, `uboot-clone`, `kernel-clone`, `root-clone`
 - 克隆完还需要进行配置和编译验证，验证完才能正确使用
 - 完整添加过程请参考第 5 大节
- 当前设定的变量值未经验证或者根本不支持
 - 比如说某个板子目前的 `ROOTDEV_LIST` 中只有 `sda`, `ram0` 和 `nfs`
 - `vda` 可能根本不支持或者需要重新配置内核后才支持
 - 这个因板子和内核版本而异，需要具体对待

6.4.8 is not a valid rootfs directory

如果当前使用的是预制文件系统，说明操作过程中可能有类似 `CTRL+C` 中断了正常根文件系统目录、`Ramdisk` 或 `Hardisk` 镜像的创建过程，导致文件系统不完整，在确保 `BSP` 目录无其他紧要修改的情况下，可以通过如下命令恢复 `BSP` 仓库为默认设置：

```
1 | $ make bsp-cleanup
```

如果当前使用的是用户自己构建的文件系统，请确保文件系统符合 `Linux` 的规范，确保相关的基础目录均存在。

7. 联系并赞助我们

我们的联系微信是 tinylab，欢迎加入 Linux Lab 的用户和开发人员讨论组。

通过微信扫描下述二维码联系我们或提供赞助：



联系我们



捐赠项目

Figure 3: Linux Lab 需要更多的用户或者赞助 ~ 加入我们吧!