利用 qemu 模拟嵌入式系统制作全过程

Wen Pingbo 创作于 2013/08/31 打赏

目录 [隐藏]

by PingboWen of TinyLab.org
2013/08/31

这篇文章，将介绍如何用qemu来搭建一个基于ARM的嵌入式linux系统。通过该文章，你可以学习到如何配置kernel，如何交叉编译kernel，如何配置busybox并编译，如何制作initramfs，如何制作根文件系统，如何定制自己的uboot，如何通过uboot向kernel传递参数等。开始干活！

1 环境搭建

在实现我们的目标之前，我们需要搭建自己的工作环境。在这里，假设你的主机上已经有 gcc 本地编译环境，并运行 Ubuntu 12.10 。但是这并不影响在其他的 linux 平台上进行，只要修改一下对应的命令就可以了。

首先，我们需要下载一个 ARM 交叉工具链。你可以在网上下载源码自己编译，也可以下载已经编译好的工具链。在工具链中有基本的 ARM 编译工具，比如： gcc, gdb, addr2line, nm, objcopy, objdump 等。可能你会问，这些工具本机不是已经有了么？如果不出意外，我想你的主机应该是 x86 架构的。不同的架构，有不同的指令集，你不能拿一个 x86 的执行文件放到一个 ARM 机器上执行。所以我们需要一个能够在 x86 架构上生成 ARM 可执行程序的 GCC 编译器。有很多预先编译好的 ARM 工具链，这里使用的是 CodeSourcery 。更多关于 toolchain 的信息可以在 elinux.org 找到。下载下来后，直接解压，放到某个目录，然后配置一下 PATH 环境变量，这里是这样配置的：

```
export PATH=~/arm-2013.05/bin:$PATH
```

配置完 ARM 交叉工具链后，我们需要下载一些源码，并安装一些软件。 命令如下：

```
# install qemu
sudo apt-get install qemu qemu-kvm qemu-kvm-extras qemu-user qemu-system
# install mkimage tool
sudo apt-get install uboot-mkimage
# install git
sudo apt-get install git
# prepare related directory
mkdir -pv ~/armsource/{kernel,uboot,ramfs,busybox}
# download latest kernel stable code to kernel dir
git clone http://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
 ~/armsource/kernel
# download latest u-boot code to uboot dir
git clone git://git.denx.de/u-boot.git ~/armsource/uboot
# download latest busybox code to busybox dir
git clone git://busybox.net/busybox.git ~/armsource/busybox
```

2 配置 kernel

环境搭建完后，我们就正式进入主题了。现在我们需要配置 kernel 源码，编译，并用 qemu 运行我们自己编译的 kernel 。这样我们就能够对我们的 kernel 进行测试，并做出对应的修改。

进入 kernel 源码目录，我们需要找最新的 kernel 稳定版本。在写这篇文章的时候，最新的稳定版本是 3.10.10 。我们可以通过 git 切换到 3.10.10 。由于我们编译的内核需要运行在 ARM 上，所以我们应该到 arch/arm/configs 下找到对应我们设备的 kernel 配置文件。但是我们没有实际意义上的设备，而是用 qemu 模拟的设备，所以我们应该选择 qemu 能够模拟的设备的配置文件。这里我们选择常用的 versatile_defconfig 。 对应的命令如下：

```
cd ~/armsource/kernel
# checkout a tag and create a branch
git checkout v3.10.10 -b linux-3.10.10
# create .config file
make versatile_defconfig ARCH=arm
```

配置完了，我们就可以编译了。编译的时候，我们可以用多个线程来加速编译，具体用多少个就要看你主机的配置了。这里我们用 12 个线程编译，命令如下：

```
make -j12 ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

注意，如果交叉编译环境没有配置好，这个地方会提示找不到对应的 gcc 编译器。这里 `-j12` 是指定编译线程为 12 个， ARCH 是指定目标架构为 arm ，所用的交叉编译器 arm-none-linux-gnueabi- 。

OK， kernel 已经编译好了，那么我们需要用 qemu 把它跑起来。关于 qemu 的具体使用，请看 qemu 的官方文档 ，这里直接给出命令：

```
qemu-system-arm -M versatilepb -kernel arch/arm/boot/zImage -nographic
```

这里 -M 是指定模拟的具体设备型号， versatile 系列的 pb 版本， -kernel 指定的是对应的内核， -nographic 是把 qemu 输出直接导向到当前终端。

好，命令成功执行了。但是，好像没有任何有效输出。我们通过 C-a x 来退出 qemu 。编译的 kernel 好像不怎么好使，配置文件肯定有问题。打开 .config 配置文件，发现传递给 kernel 的参数没有指定 console ， 难怪没有输出。我们定位到 CMDLINE ， 并加入 console 参数：

```
CONFIG_CMDLINE="console=ttyAMA0 root=/dev/ram0"
```

保存 .config ， 重新编译 kernel ， 并用 qemu 加载。现在终于有输出了。如果不出意外，kernel 应该会停在找不到根文件系统，并跳出一个 panic 。为什么会找不到根文件系统？因为我们压根就没有给它传递过，当然找不到。

那现在是不是应该制作我们自己的根文件系统了。先别急，为了让后面的路好走一点，我们这里还需对内核进行一些配置。首先，我们需要用 ARM EABI 去编译 kernel ， 这样我们才能让 kernel 运行我们交叉编译的用户态程序，因为我们所有的程序都是用 gnueabi 的编译器编译的。具体可以看 wikipedia 相关页面 4 ， 你也可以简单的理解为嵌入式的 ABI 。其次，我们需要把对 kernel module 的支持去掉，这样可以把相关的驱动都编译到一个文件里，方便我们之后的加载。

当然，你可以使能 kernel 的 debug 选项，这样就可以调试内核了，并打印很多调试信息。这里就不再说了，如果感兴趣，可以看我之前写的关于 kernel 调试的文章 5 。

总结起来，这一次我们对 .config 做了如下修改：

```
# CONFIG_MODULES is not set
CONFIG_AEABI=y
CONFIG_OABI_COMPAT=y
CONFIG_PRINTK_TIME=y
CONFIG_EARLY_PRINTK=y
CONFIG_CMDLINE="earlyprintk console=ttyAMA0 root=/dev/ram0"
```

3 通过 busybox 制作 initramfs 镜像

如果你注意到了之前传递给 kernel 的参数，你会发现有一个 root=/dev/ram0 的参数。没错，这就是给 kernel 指定的根文件系统， kernel 检查到这个参数的时候，会到指定的地方加载根文件系统，并执行其中的 init 程序。这样就不会出现刚才那种情况，找不到根文件系统了。

我们的目标就是让 kernel 挂载我们的 ramfs 根文件系统，并且在执行 init 程序的时候，调用 busybox 中的一个 shell ， 这样我们就有一个可用的 shell 来和系统进行交互了。

整个 ramfs 中的核心就是一个 busybox 可执行文件。 busybox 就像是一把瑞士军刀，可以把很多 linux 下的命令（比如： cp, rm, whoami 等）全部集成到一个可执行文件。这为制作嵌入式根文件系统提供了很大的便利，开发者不用单独编译每一个要支持的命令，还不用考虑库的依赖关系。基本上每一个制作嵌入式系统的开发者的首选就是 busybox 。

busybox 也是采用 Kconfig 来管理配置选项，所以配置和编译 busybox 和 kernel 没有多大区别。 busybox 很灵活，你可以自由取舍你想要支持的命令，并且还可以添加你自己写的程序。在编译 busybox 的时候，为了简单省事，我们这里采用静态编译，这样就不用为 busybox 准备其他 libc ， ld 等依赖库了。

具体命令如下：

```
cd ~/armsource/busybox
# using stable version 1.21
git checkout origin/1_21_stable -b busybox-1.21
# using default configure
make defconfig ARCH=arm
# compile busybox in static
make menuconfig
make -j12 ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

编译完后，我们就得到一个 busybox 静态链接的文件。

接下来，我们需要一个 init 程序。这个程序将是 kernel 执行的第一个用户态的程序，我们需要它来产生一个可交互的 shell 。在桌面级别的 linux 发行版本，使用的 init 程序一般是 System V init(传统的 init)， upstart(ubuntu)， systemd(fedora) 等。 busybox 也带有一个 init 程序，但是我们想自己写一个。既然自己写，那有两种实现方式，用 C 和 libc 实现，或者写一个 shell 脚本。

为了简单，这里选择后者，具体脚本如下：

```
#!/bin/sh
echo
echo "###########################################################"
echo "## THis is a init script for initrd/initramfs             ##"
echo "## Author: wengpingbo@gmail.com                           ##"
echo "## Date: 2013/08/17 16:27:34 CST                          ##"
echo "###########################################################"
echo

PATH="/bin:/sbin:/usr/bin:/usr/sbin"

if [ ! -f "/bin/busybox" ];then
  echo "cat not find busybox in /bin dir, exit"
  exit 1
fi

BUSYBOX="/bin/busybox"

echo "build root filesystem..."
$BUSYBOX --install -s

if [ ! -d /proc ];then
  echo "/proc dir not exist, create it..."
  $BUSYBOX mkdir /proc
fi
echo "mount proc fs..."
$BUSYBOX mount -t proc proc /proc

if [ ! -d /dev ];then
  echo "/dev dir not exist, create it..."
  $BUSYBOX mkdir /dev
fi
# echo "mount tmpfs in /dev..."
# $BUSYBOX mount -t tmpfs dev /dev

$BUSYBOX mkdir -p /dev/pts
echo "mount devpts..."
$BUSYBOX mount -t devpts devpts /dev/pts
```

```
if [ ! -d /sys ];then
    echo "/sys dir not exist, create it..."
    $BUSYBOX mkdir /sys
fi
echo "mount sys fs..."
$BUSYBOX mount -t sysfs sys /sys

echo "/sbin/mdev" > /proc/sys/kernel/hotplug
echo "populate the dev dir..."
$BUSYBOX mdev -s

echo "drop to shell..."
$BUSYBOX sh

exit 0
```

我们把这个脚本保存在 ~/armsource 目录下。在这个脚本中，我们通过 busybox –install -s 来构建基本文件系统，挂载相应的虚拟文件系统，然后就调用 busybox 自带的 shell 。

现在我们已经编译好了 busybox ， 并准备好了相应的 init 脚本。我们需要考虑根文件系统的目录结构了。 kenel 支持很多种文件系统，比如： ext4, ext3, ext2, cramfs, nfs, jffs2, reiserfs 等，还包括一些伪文件系统 : sysfs, proc, ramfs 等。而在 kernel 初始化完成后，会尝试挂载一个它所支持的根文件系统。根文件系统的目录结构标准是 FHS ， 由一些 kernel 开发者制定，感兴趣的可以看 wikipedia 相关页面 。

由于我们要制作一个很简单的 ramfs ， 其中只有一个 busybox 可执行文件，所以我们没必要过多的考虑什么标准。只需一些必须的目录结构就 OK 。这里，我们使用的目录结构如下：

```
├── bin
│   ├── busybox
│   └── sh -> busybox
├── dev
│   └── console
├── etc
│   └── init.d
│       └── rcS
├── init
├── sbin
└── usr
    ├── bin
    └── sbin
```

你可以通过如下命令来创建这个文件系统：

```
cd ~/armsource/ramfs
mkdir -pv bin dev etc/init.d sbin user/{bin,sbin}
cp ~/armsource/busybox/busybox bin/
ln -s busybox bin/sh
mknod -m 644 dev/console c 5 1
cp ~/armsource/init .
touch etc/init.d/rcS
chmod +x bin/busybox etc/init.d/rcS init
```

现在我们有了基本的 initramfs ， 万事具备了，就差点东风了。这个东风就是怎样制作 intramfs 镜像，并让 kernel 加载它。

在 kernel 文档中，相应说明有 initramfs 和 initrd 。 initramfs 其实就是一个用 gzip 压缩的 cpio 文件。我们可以把 initramfs 直接集成到 kernel 里，也可以单独加载 initramfs 。在 kernel 源码的 scripts 目录下，有一个 gen_initramfs_list.sh 脚本，专门是用来生成 initramfs 镜像和 initramfs list 文件。你可以通过如下方式自动生成 initramfs 镜像：

```
sh scripts/gen_initramfs_list.sh -o ramfs.gz ~/armsource/ramfs
```

然后修改 kernel 的 .config 配置文件来包含这个文件：

```
CONFIG_INITRAMFS_SOURCE="ramfs.gz"
```

重新编译后， kernel 就自动集成了你制作的 ramfs.gz ，并会在初始化完成后，加载这个根文件系统，并产生一个 shell 。

你也可以用 gen_initramfs_list.sh 脚本生成一个列表文件，然后 CONFIG_INITRAMFS_SOURCE 中指定这个列表文件。也可以把你做的根文件系统自动集成到 kernel 里面。命令如下：

```
sh scripts/gen_initramfs_list.sh ~/armsource/ramfs > initramfs_list
```

对应的内核配置： `CONFIG_INITRAMFS_SOURCE="initramfs_list"`

但是这里并不打算这么做，我们自己手动制作 initramfs 镜像，然后外部加载。命令如下：

```
cd ~/armsource/ramfs
find . | cpio -o -H newc | gzip -9 > ramfs.gz
```

选项 -H 是用来指定生成的格式。

手动生成 ramfs.gz 后，我们就可以通过 qemu 来加载了，命令如下：

```
qemu-system-arm -M versatilepb -kernel arch/arm/boot/zImage -nographic -initrd ramfs
.gz
```

现在我们的系统起来了，并且正确执行了我们自己写的脚本，进入了shell。我们可以在里面执行基本常用的命令。是不是有点小兴奋。

4 配置物理文件系统，切换根文件系统

不是已经配置了根文件系统，并加载了，为什么还需要切换呢？可能你还沉浸在刚才的小兴奋里，但是，很不幸的告诉你。现在制作的小 linux 系统还不是一个完全的系统，因为没有完成基本的初始化，尽管看上去好像已经完成了。

在 linux 中 initramfs 和 initrd 只是一个用于系统初始化的小型文件系统，通常用来加载一些第三方的驱动。为什么要通过这种方式来加载驱动呢？因为由于版权协议的关系，如果要把驱动放到 kernenl 里，意味着你必须要开放源代码。但是有些时候，一些商业公司不想开源自己的驱动，那它就可以把驱动放到 initramfs 或者 initrd 。这样既不违背 kernel 版权协议，又达到不开源的目的。也就是说在正常的 linux 发行版本中， kernel 初始化完成后，会先挂载 initramfs/initrd ，来加载其他驱动，并做一些初始化设置。然后才会挂载真真的根文件系统，通

过一个 switch_root 来切换根文件系统，执行第二个 init 程序，加载各种用户程序。在这中间，linux kernel 跳了两下。

既然他们跳了两下，那我们也跳两下。第一下已经跳了，现在的目标是制作物理文件系统，并修改 initramfs 中的 init 脚本，来挂载我们物理文件系统，并切换 root 文件系统，执行对应的 init 。

为了省事，我们直接把原先的 initramfs 文件系统复制一份，当作物理根文件系统。由于是模拟，所以我们直接利用 dd 来生成一个磁盘镜像。具体命令如下：

```
dd if=/dev/zero of=~/armsource/hda.img bs=1 count=10M
mkfs -t ext2 hda.img
mount hda.img /mnt
cp -r ~/armsource/ramfs/* /mnt
umount /mnt
```

这样 hda.img 就是我们制作的物理根文件系统， ext2 格式。现在我们需要修改原先在 initramfs 中的 init 脚本，让其通过 busybox 的 switch_root 功能切换根文件系统。这里需要注意的是，在切换根文件系统时，不能直接调用 busybox 的 switch_root ，而是需要通过 exec 来调用。这样才能让最终的 init 进程 pid 为 1 。

修改后的 init 脚本如下：

```
#!/bin/sh
echo
echo "###########################################################"
echo "## THis is a init script for sd ext2 filesystem          ##"
echo "## Author: wengpingbo@gmail.com                          ##"
echo "## Date: 2013/08/17 16:27:34 CST                         ##"
echo "###########################################################"
echo

PATH="/bin:/sbin:/usr/bin:/usr/sbin"

if [ ! -f "/bin/busybox" ];then
  echo "cat not find busybox in /bin dir, exit"
  exit 1
fi

BUSYBOX="/bin/busybox"

echo "build root filesystem..."
$BUSYBOX --install -s

if [ ! -d /proc ];then
  echo "/proc dir not exist, create it..."
  $BUSYBOX mkdir /proc
fi
echo "mount proc fs..."
$BUSYBOX mount -t proc proc /proc

if [ ! -d /dev ];then
  echo "/dev dir not exist, create it..."
  $BUSYBOX mkdir /dev
fi
# echo "mount tmpfs in /dev..."
# $BUSYBOX mount -t tmpfs dev /dev

```

```
$BUSYBOX mkdir -p /dev/pts
echo "mount devpts..."
$BUSYBOX mount -t devpts devpts /dev/pts

if [ ! -d /sys ];then
  echo "/sys dir not exist, create it..."
  $BUSYBOX mkdir /sys
fi
echo "mount sys fs..."
$BUSYBOX mount -t sysfs sys /sys

echo "/sbin/mdev" > /proc/sys/kernel/hotplug
echo "populate the dev dir..."
$BUSYBOX mdev -s

echo "dev filesystem is ok now, log all in kernel kmsg" >> /dev/kmsg

echo "you can add some third part driver in this phase..." >> /dev/kmsg
echo "begin switch root directory to sd card" >> /dev/kmsg

$BUSYBOX mkdir /newroot
if [ ! -b "/dev/mmcblk0" ];then
  echo "can not find /dev/mmcblk0, please make sure the sd \
        card is attached correctly!" >> /dev/kmsg
  echo "drop to shell" >> /dev/kmsg
  $BUSYBOX sh
else
  $BUSYBOX mount /dev/mmcblk0 /newroot
  if [ $? -eq 0 ];then
        echo "mount root file system successfully..." >> /dev/kmsg
  else
        echo "failed to mount root file system, drop to shell" >> /dev/kmsg
        $BUSYBOX sh
  fi
fi

# the root file system is mounted, clean the world for new root file system
echo "" > /proc/sys/kernel/hotplug
$BUSYBOX umount -f /proc
$BUSYBOX umount -f /sys
$BUSYBOX umount -f /dev/pts
# $BUSYBOX umount -f /dev

echo "enter new root..." >> /dev/kmsg
exec $BUSYBOX switch_root -c /dev/console /newroot /init

if [ $? -ne 0 ];then
  echo "enter new root file system failed, drop to shell" >> /dev/kmsg
  $BUSYBOX mount -t proc proc /proc
  $BUSYBOX sh
fi
```

现在我们可以通过 qemu 来挂载 hda.img ，为了简单，我们这里把这个设备虚拟为 sd 卡，这也是为什么上面的 init 脚本挂载物理根文件系统时，是找 /dev/mmcblk0 了。具体命令如下：

```
qemu-system-arm -M versatilepb -kernel arch/arm/boot/zImage -nographic -initrd ramfs
.gz -sd hda.img
```

如果不出意外，你可以看到这个自己做的linux系统，通过调用两个init脚本，跳到最终的hda.img上的文件系统。

5 配置 Uboot，加载 kernel

可能到这里，你觉得，终于把整个流程走了一遍了。但是，还差一环。之前我们都是通过 qemu 来直接加载我们的 kernel ， initramfs 和物理镜像，但是在真真的嵌入式设备，这些加载过程都需要你好好考虑。那么在这一节，我们借助 uboot 来模拟加载过程。

我们的目标是让 uboot 来加载 kernel ， initramfs ，并识别 qemu 虚拟的 sd 卡设备。这里我们通过 tftp 来向 uboot 传递 kernel 和 initramfs 镜像。既然要依靠 uboot 来加载系统镜像，那么需要按照 uboot 的镜像格式制作加载的镜像。而 mkimage 工具，就是干这活的。在制作 uboot 镜像时，我们需要指定镜像类型，加载地址，执行地址等，制作 uboot 版的 initramfs 命令如下：

```
mkimage -A arm -O linux -T ramdisk -C none -a 0x00808000 -e 0x00808000 -n ramdisk
-d ramfs.gz ramfs-uboot.img
```

其中 -a 和 -e 分别是指定加载定制和执行地址

而 kernel 的 uboot 版就不需要这么手动生成了，在编译 kernel 的时候，可以通过 make uImage 来制作 uboot 格式镜像，默认的加载地址是 0x00008000 ，你也可以通过 LOADADDR 指定你自己的加载地址，这里用默认的。

镜像准备好之后，需要把这两个镜像拷贝到一个指定的目录，这样在用 tftp 传输的时候，能够找到对应的镜像。这里假设拷贝到~ /armsource/tftp 目录下。

下一步，我们需要交叉编译 uboot 。在编译之前，我们需要对 uboot 进行一些配置。由于我们使用的是 versatilepb ， 它对应的配置文件在 include/configs/versatile.h 中，这里对这个文件的修改如下：

```
#define CONFIG_ARCH_VERSATILE_QEMU
#define CONFIG_INITRD_TAG
#define CONFIG_SYS_PROMPT  "myboard > "
#define CONFIG_BOOTCOMMAND \
  "sete ipaddr 10.0.2.15;"\
  "sete serverip 10.0.2.2;"\
  "set bootargs 'console=ttyAMA0,115200 root=/dev/mmcblk0';"\
  "tftpboot 0x00007fc0 uImage;"\
  "tftpboot 0x00807fc0 ramfs-uboot.img;"\
  "bootm 0x7fc0 0x807fc0"
```

其中 ARCH_VERSATILE_QEMU 是为了让 uboot 为了适应 qemu 做一些配置上的调整。INITRD_TAG 是让 uboot 通过 tag_list 给 kernel 传递 initramfs 的地址，如果没有这个配置选项， kernel 是找不到 uboot 传给他的 initramfs 。 SYS_PROMPT 是指定 uboot 的命令提示符，你可以指定你自己的名字。 BOOTCOMMAND 是指定 uboot 起来后，自动执行的命令，这里是让 uboot 自动设置自己的 ip 和 tftp 服务器的 ip ，然后设定传递给 kernel 的参数，最后三个命令是把 kernel 镜像和 initramfs 镜像装载进来，并从内存指定地址开始执行指令。其实这些命令，也可以在 uboot 起来后，自己输入。

注意：在设置 uboot 的 ip 的时候，一定要和 qemu 给定的 ip 对应。由于这里使用的 qemu 内部自带的 tftp 服务，所以这里的 ip 和 qemu 内部 tftp 服务器的 ip 在同一个网段。

uboot 配置完之后，可以通过如下命令来编译 uboot:

```
make versatilepb_defconfig
make -j12 ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

如果没什么错误，就会生成一个 u-boot 镜像，然后我们就可以通过 qemu 来加载它：

```
sudo qemu-system-arm -M versatilepb -kernel u-boot -m 256M -net nic -net user,tftp
=~/armsource/tftp -sd hda.img -nographic
```

命令执行后，你就可以和之前一样的内核加载，最后经过两次跳转，到我们的 sd 卡上的文件系统。

6 结语

到这里，我们最终完成了 **qemu –> uboot –> kernel –> initramfs –> hda.img** 这一过程。而这也是制作嵌入式系统，甚至一个桌面发行版本的基本流程。如果看完这篇文章后，还对嵌入式系统念念不忘，还是建议你买一块开发板，然后真真走一遍这个过程，毕竟这是用qemu模拟的。现在有很多open source hardware project(Arduino, Beagle Board, Cubieboard, Odroid，PandaBoard，Raspberry Pi)，你可以购买他们的板子，然后移植任何自己喜欢的东西。由于是open source，你可以获取到很多资料，并且有社区支持。

7 参考资料

- Download CodeSourcery
- toolchain
- Qemu User Document
- EABI

- kernel debug
- Busybox

- FHS

- initrd
- Initrd/Initramfs
- Virtual_Development_Board