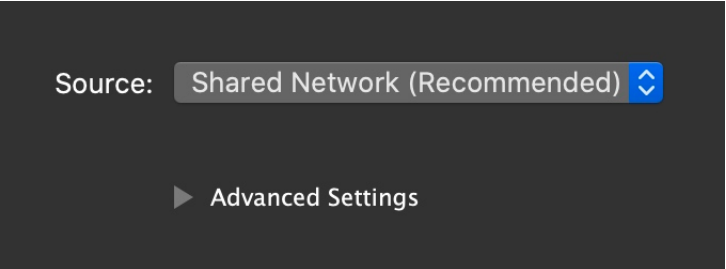


使用QEMU搭建本地分析环境

0x00 安装QEMU

说明一下我的环境，我的环境是mbp上通过pd来起kali虚拟机，在kali虚拟机里我再安装QEMU的虚拟机。kali虚拟机和mbp之间是通过



配置的网络

下面安装QEMU的相关程序

```
apt-get install binfmt-support qemu-user-static qemu qemu-system
```

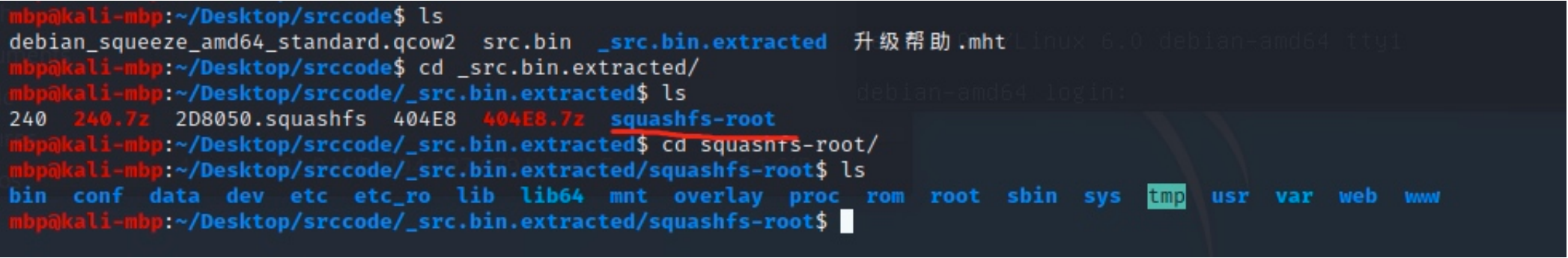
全装上总没错！有文章推荐安装buildroot，可能是后续交叉编译什么的使用，但目前看来也用不到所以先不装了。这样装完qemu应该就可以用了。

0x01 QEMU用户模式

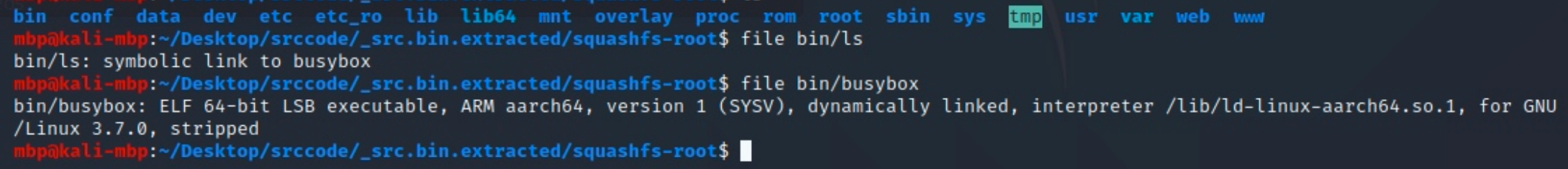
用户模式我的理解上就是直接使用qemu的命令行执行某个依赖于其他架构的程序，比如mips架构的二进制程序需要在mips系统里运行，但是如果只是单纯运行他，我们通过用户模式的方式去直接运行就可以不用开启一个类似于vmware的完整的虚拟机。下面演示一下用户模式时的qemu使用：

1.正常流程

先进入需要执行的二进制文件夹，比如这里我binwalk出了一个文件系统：



比如我们要执行bin里面的busybox命令，我们先看看他是什么架构的文件



可以看到“ARM aarch64”


```
/Linux 5.7.0, stripped
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root$ qemu-aarch64-static
qemu-aarch64_be-static      qemu-mipsn32-static      qemu-system-alpha
qemu-aarch64-static        qemu-mips-static        qemu-system-arm
qemu-alpha-static          qemu-nbd                 qemu-system-avr
qemu-armeb-static          qemu-nios2-static       qemu-system-cris
qemu-arm-static            qemu-or1k-static        qemu-system-hppa
qemu-cris-static           qemu-ppc64le-static     qemu-system-i386
qemu-debootstrap           qemu-ppc64-static       qemu-system-m68k
qemu-hppa-static           qemu-ppc-static         qemu-system-microblaze
qemu-i386-static           qemu-pr-helper          qemu-system-microblazeel
qemu-img                   qemu-riscv32-static     qemu-system-mips
qemu-io                     qemu-riscv64-static     qemu-system-mips64
qemu-m68k-static           qemu-s390x-static       qemu-system-mips64el
qemu-make-debian-root      qemu-sh4eb-static       qemu-system-mipsel
qemu-microblazeel-static   qemu-sh4-static         qemu-system-moxie
qemu-microblaze-static     qemu-sparc32plus-static qemu-system-nios2
qemu-mips64el-static       qemu-sparc64-static     qemu-system-or1k
qemu-mips64-static        qemu-sparc-static       qemu-system-ppc
qemu-mipsel-static         qemu-storage-daemon     qemu-system-ppc64
qemu-mipsn32el-static      qemu-system-aarch64     qemu-system-ppc64le
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root$ qemu-aarch64-static
```

我们可以看到qemu-aarch64-static这个命令是有的，用它可以进行用户模式。

```
cp $(which qemu-aarch64-static) ./
```

这是把qemu的用于执行arm架构的命令复制到当前目录

```
sudo chroot . ./qemu-aarch64-static ./bin/ls
```

```
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root$ cp $(which qemu-aarch64-static) ./
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root$ sudo chroot . ./qemu-aarch64-static ./bin/ls
[sudo] password for mbp:
bin      etc      mnt      rom      tmp      www
conf     etc_ro  overlay  root
data     lib     proc     sbin
dev       lib64   qemu-aarch64-static sys      web
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root$ sudo chroot . ./qemu-aarch64-static ./bin/busybox
BusyBox v1.19.4 (2020-10-22 10:32:16 CST) multi-call binary.
Copyright (C) 1998-2011 Erik Andersen, Rob Landley, Denys Vlasenko
and others. Licensed under GPLv2.
See source distribution for full notice.

Usage: busybox [function] [arguments] ...
or: busybox --list[-full]
or: function [arguments] ...

BusyBox is a multi-call binary that combines many common Unix
utilities into a single executable. Most people will create a
link to busybox for each function they wish to use and BusyBox
will act like whatever it was invoked as.
```

可以看到执行成功。
正常情况下我们没有安装qemu，直接执行这个busybox会报错，提示架构不同不能执行。

2.非正常使用？

安装了qemu后，如果我们直接进入bin执行busybox会提示

```
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root$ cd bin
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root/bin$ ./busybox
qemu-aarch64-static: Could not open '/lib/ld-linux-aarch64.so.1': No such file or directory
mbp@kali-mbp:~/Desktop/srccode/_src.bin.extracted/squashfs-root/bin$
```

这个so文件在这个文件系统的lib目录下是存在的，换句话说我们没有chroot导致他找不到，这里我们没有用qemu就执行了busybox，可能是因为我安装了qemu后默认自动对非x86架构的可执行文件以用户模式来启动。那么我们可以试着直接chroot到这个目录或者是把lib文件复制到根目录的lib目录下，应该也是可以执行的


```
mbp@kali-mbp:~/Desktop/src/extracted/squashfs-root/bin$ cd ..
mbp@kali-mbp:~/Desktop/src/extracted/squashfs-root$ sudo chroot .
chroot: failed to run command '/bin/bash': No such file or directory
mbp@kali-mbp:~/Desktop/src/extracted/squashfs-root$ sudo chroot . ./bin/sh

BusyBox v1.19.4 (2020-10-22 10:32:16 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ # busybox ls
bin      data     etc      lib      mnt      proc     root     sys      usr      web
conf     dev      etc_ro   lib64    overlay  rom      sbin     tmp      var      www
/ #
```

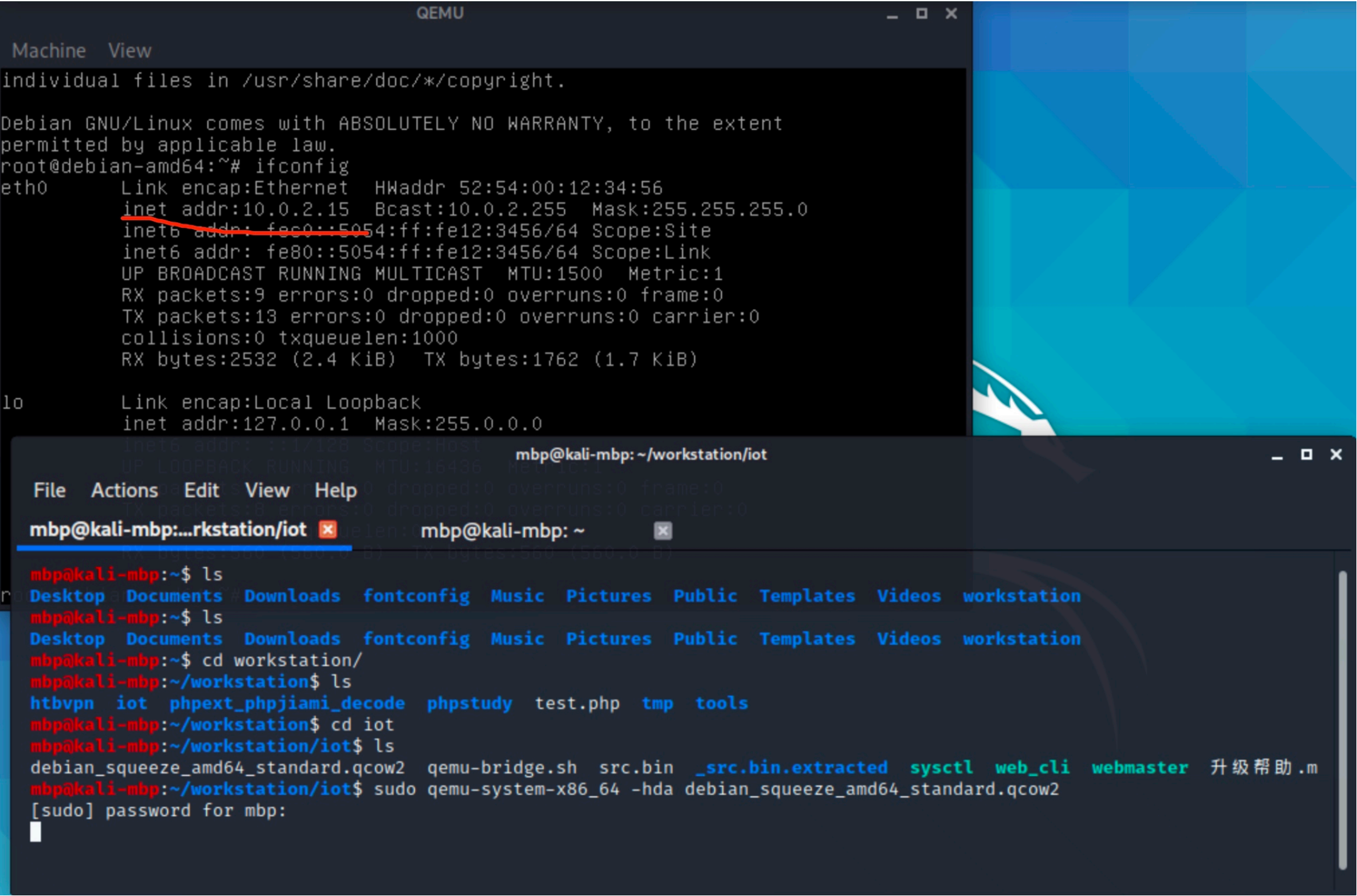
这个图里注意一个小细节：直接chroot提示没有bash，应该是chroot后我们没有给定运行的程序所以默认是以chroot后的bin里的bash为运行命令，因此找不到bash就报错了，这个bin里面有sh命令，所以我们指定/bin/sh为chroot后运行的命令。于是乎就进来了。

0x02 QEMU系统模式

系统模式就是类似于起一个VM虚拟机，在使用系统模式之前通常我们需要配置一下网桥以便在其他主机可以访问到qemu主机。

1.配置网桥

先说一下为什么要配置网桥，如果我们不配置网桥直接下载对应的系统运行，会变成下图



可以看到qemu的ip是一个10.0.2.15，无论是在kali还是在mbp里都ping不通，我们需要和他通信，以便后续的远程调试等操作。

配置网桥参考的文章：[参考smilejay.com](#)

看了一圈就这篇靠谱。

1.安装依赖

```
apt-get install bridge-utils uml-utilities
```

2.配置网桥

安装完毕后，我们直接使用一个sh脚本在host主机上开启网桥

```
brctl addbr br0      #添加br0这个bridge
brctl addif br0 eth0  #将br0与eth0绑定起来
brctl stp br0 on     #将br0加入到STP协议中
dhclient br0         #将br0的网络配置好
```

将脚本保存为.sh文件，这里保存为qemu-bridge.sh
直接在host主机上运行脚本

```
sudo sh qemu-bridge.sh
```

运行完毕后

```
mbp@kali-mbp:~/workstation/iot$ sudo sh qemu-bridge.sh
[sudo] password for mbp:
mbp@kali-mbp:~/workstation/iot$ ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 10.211.55.4  netmask 255.255.255.0  broadcast 10.211.55.255
    inet6 fdb2:2c26:f4e4:0:21c:42ff:fe16:7965  prefixlen 64  scopeid 0x0<global>
    inet6 fe80::21c:42ff:fe16:7965  prefixlen 64  scopeid 0x20<link>
    ether 00:1c:42:16:79:65  txqueuelen 1000  (Ethernet)
    RX packets 73  bytes 24573 (23.9 KiB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 12  bytes 1624 (1.5 KiB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 10.211.55.3  netmask 255.255.255.0  broadcast 10.211.55.255
    inet6 fdb2:2c26:f4e4:0:21c:42ff:fe16:7965  prefixlen 64  scopeid 0x0<global>
    inet6 fe80::21c:42ff:fe16:7965  prefixlen 64  scopeid 0x20<link>
    ether 00:1c:42:16:79:65  txqueuelen 1000  (Ethernet)
    RX packets 199  bytes 60862 (59.4 KiB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 240  bytes 22689 (22.1 KiB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
    inet 127.0.0.1  netmask 255.0.0.0
    inet6 ::1  prefixlen 128  scopeid 0x10<host>
    loop txqueuelen 1000  (Local Loopback)
    RX packets 20  bytes 996 (996.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 20  bytes 996 (996.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

可以看到多了个br0，并且ip地址为10.211.55.4

3.准备qemu-ifup文件

复制备份原来的qemu-ifup文件后，修改qemu-ifup文件内容为如下：

```
#!/bin/bash
#This is a qemu-ifup script for bridging.
#You can use it when starting a KVM guest with bridge mode network.
#set your bridge name
switch=br0
if [ -n "$1" ]; then
#create a TAP interface; qemu will handle it automatically.
#tunctl -u $(whoami) -t $1
#start up the TAP interface
ip link set $1 up
sleep 1
#add TAP interface to the bridge
brctl addif ${switch} $1
exit 0
else
echo "Error: no interface specified"
exit 1
fi
```

保存即可。
如果重启的话，网桥的脚本需要重新运行一下。

2.配置qemu系统模式虚拟机












在启用系统模式前，我们必须搞清楚对应目标文件系统的系统架构。我这边的目标架构是arm64

```
mbp@kali-mbp:~/workstation/iot$ ls
amd64VM  arm64VM  armelVM  armhfVM  targetfs
mbp@kali-mbp:~/workstation/iot$ cd targetfs/
mbp@kali-mbp:~/workstation/iot/targetfs$ ls
qemu-bridge.sh  src.bin  _src.bin.extracted  sysctl  web_cli  webmaster  升级帮助.mht
mbp@kali-mbp:~/workstation/iot/targetfs$ cd _src.bin.extracted/squashfs-root/bin/
mbp@kali-mbp:~/workstation/iot/targetfs/_src.bin.extracted/squashfs-root/bin$ file busybox
busybox: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64
.so.1, for GNU/Linux 3.7.0, stripped
mbp@kali-mbp:~/workstation/iot/targetfs/_src.bin.extracted/squashfs-root/bin$
```

这个架构我一开始以为可以通过[debian.org](#)里面提供的armel或者是armhf来解决，但经过尝试并不可行。这一小节下面的部分截图并不是使用的arm64的系统截图，不过并不影响整体的使用方式，所以就不改了。假设我们可以使用armhf来运行qemu系统模式。

首先，到这个地方下载对应的文件[debian.org](#)







Index of /~aurel32/qemu

Name	Last modified	Size	Description
 Parent Directory	-		
 amd64/	2014-01-06 18:29	-	
 armel/	2014-01-06 18:29	-	
 armhf/	2014-01-06 18:29	-	
 i386/	2014-01-06 18:29	-	
 kfreebsd-amd64/	2014-01-06 18:29	-	
 kfreebsd-i386/	2014-01-06 18:29	-	
 mips/	2015-03-15 19:07	-	
 mipsel/	2014-06-22 09:55	-	
 powerpc/	2014-01-06 18:29	-	
 sh4/	2014-01-06 18:29	-	
 sparc/	2014-01-06 18:29	-	

Apache Server at people.debian.org Port 443

这里假设我们用armhf，那就点进去armhf，可以看到这些

Index of /~aurel32/qemu/armhf

Name	Last modified	Size	Description
 Parent Directory	-		
 README.txt	2014-01-06 18:29	3.4K	
 debian_wheezy_armhf_desktop.qcow2	2013-12-17 02:43	1.7G	
 debian_wheezy_armhf_standard.qcow2	2013-12-17 00:04	229M	
 initrd.img-3.2.0-4-vexpress	2013-12-17 01:57	2.2M	
 vmlinuz-3.2.0-4-vexpress	2013-09-20 18:33	1.9M	

-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1,SHA256

在下面已经写了如何选择对应的版本，什么有桌面没桌面什么的，不是很重要

Both images are 25GiB images in QCOW2 format on which a Debian Wheezy system has been installed. The standard images correspond to a "Standard system" installation, while the desktop images correspond to a "Standard system" + "Desktop environment" with Gnome, KDE and Xfce environments. The original default desktop environment is Gnome and the default display manager is GDM. These can not work in QEMU, as Gnome needs an accelerated graphics card, which is not provided by QEMU. Moreover both Gnome and GDM need more than the default 128MiB memory provided on QEMU. They have therefore been replaced respectively by Xfce and LightDM on the desktop image. Both Gnome and GDM are still installed on the image, and the original default can be restored using the following commands:

- update-alternatives --auto x-session-manager
- echo /usr/sbin/gdm3 > /etc/X11/default-display-manager

The other installation options are the following:

这里我们选择没有桌面的，然后虚拟机账号密码、启动虚拟机执行的命令也告诉你了

```
- echo /usr/sbin/gdm3 > /etc/X11/default-display-manager
```

The other installation options are the following:

- Keyboard: US
- Locale: en_US
- Mirror: ftp.debian.org
- Hostname: debian-armel
- Root password: root
- User account: user
- User password: user

To use these images, you need to install QEMU 1.1.0 (or later). Start

QEMU with the following arguments:

- qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd initrd.img-3.2.0-4-vexpress -drive if=sd,file=debian_wheezy_armhf_standard.qcow2 -append "root=/dev/mmcblk0p2"
- qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd initrd.img-3.2.0-4-vexpress -drive if=sd,file=debian_wheezy_armhf_desktop.qcow2 -append "root=/dev/mmcblk0p2"

By default QEMU emulates a machine with 128MiB of RAM. The -m option increases or decreases the size of the RAM. It is however limited to 1024MiB. If you don't want to start QEMU in graphic mode, you can use the -nographic option. The image is configured to display a login prompt on the first serial port (ttyAMA0). If you want to switch the boot messages to the serial port, you need to add "console=ttyAMA0" after "root=/dev/mmcblk0p2".

这里我们选择第一条，因此要下载

1. vmlinuz-3.2.0-4-vexpress
2. initrd.img-3.2.0-4-vexpress
3. debian_wheezy_armhf_standard.qcow2

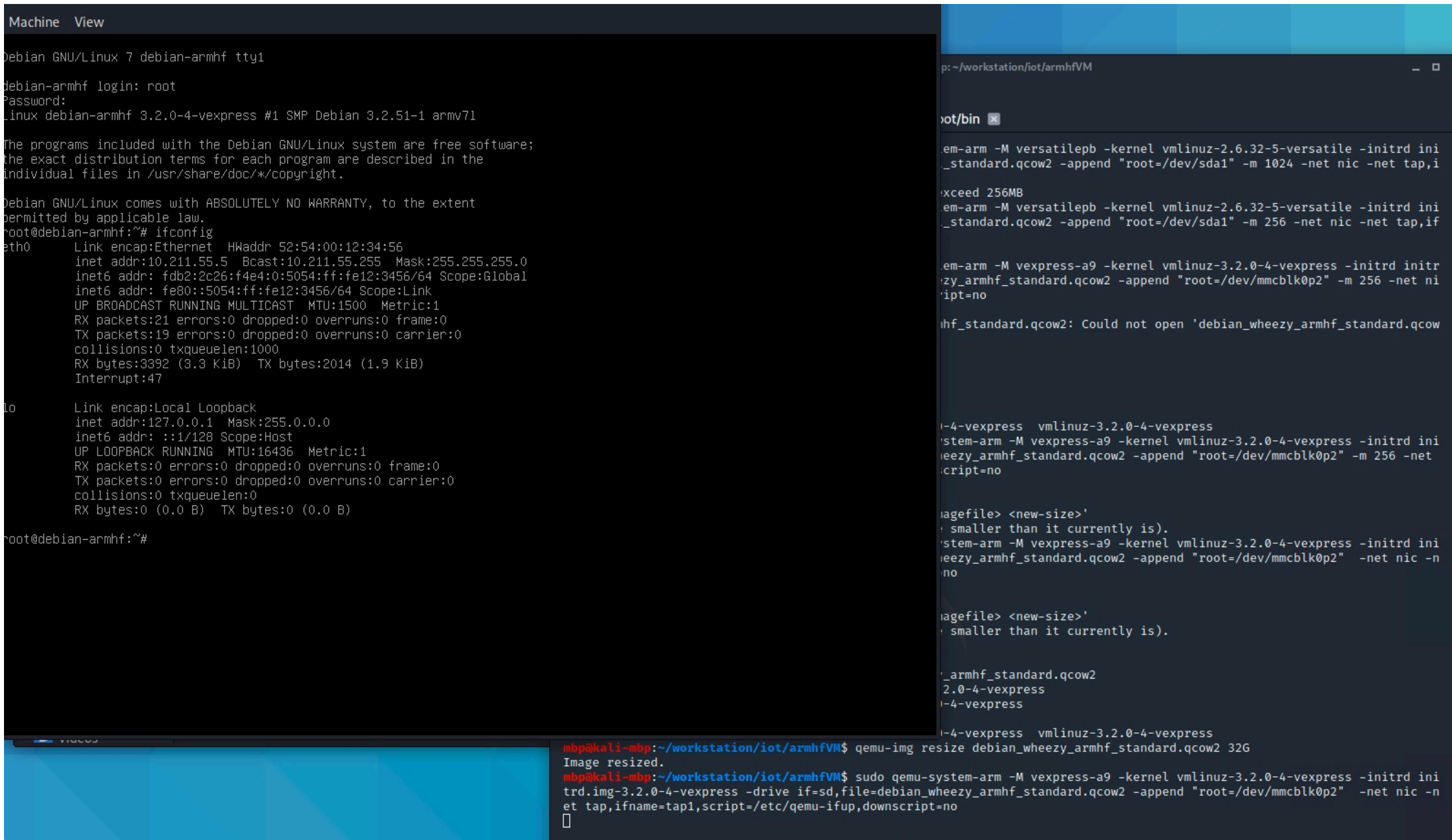
然后执行

```
sudo qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd initrd.img-3.2.0-4-vexpress -drive if=sd,file=debian_wheezy_armhf_standard.qcow2 -append "root=/dev/mmcblk0p2" -m 256 -net nic -net tap,ifname=tap1,script=/etc/qemu-ifup,downscript=no
```

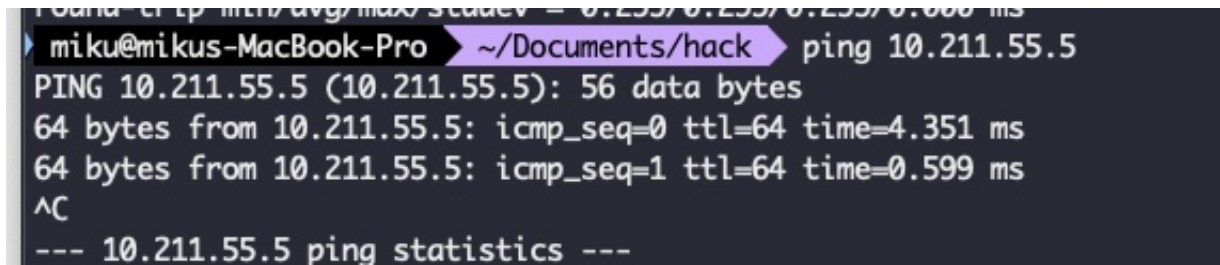
结果报错了，我们按照提示使用qemu-img resize命令试试

```
(note that this will lose data if you make the image smaller than it currently is).
mbp@kali-mbp:~/workstation/iot/armhfVM$ sudo qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd initrd.img-3.2.0-4-vexpress -drive if=sd,file=debian_wheezy_armhf_standard.qcow2 -append "root=/dev/mmcblk0p2" -net nic -net tap,ifname=tap1,script=/etc/qemu-ifup,downscript=no
qemu-system-arm: Invalid SD card size: 25 GiB
SD card size has to be a power of 2, e.g. 32 GiB.
You can resize disk images with 'qemu-img resize <imagefile> <new-size>'
(note that this will lose data if you make the image smaller than it currently is).
mbp@kali-mbp:~/workstation/iot/armhfVM$ ls -lh
total 233M
-rw-r--r-- 1 mbp mbp 229M Jan 29 18:59 debian_wheezy_armhf_standard.qcow2
-rw-r--r-- 1 mbp mbp 2.2M Jan 29 18:59 initrd.img-3.2.0-4-vexpress
-rw-r--r-- 1 mbp mbp 2.0M Jan 29 18:59 vmlinuz-3.2.0-4-vexpress
mbp@kali-mbp:~/workstation/iot/armhfVM$ ls
debian_wheezy_armhf_standard.qcow2  initrd.img-3.2.0-4-vexpress  vmlinuz-3.2.0-4-vexpress
mbp@kali-mbp:~/workstation/iot/armhfVM$ qemu-img resize debian_wheezy_armhf_standard.qcow2 32G
Image resized.
mbp@kali-mbp:~/workstation/iot/armhfVM$
```

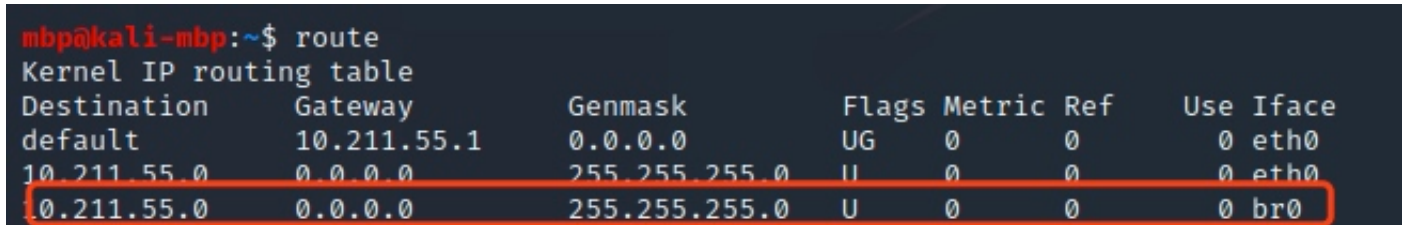
将debian_wheezy_armhf_standard.qcow2文件进行resize到32G，再重新运行命令，可以看到成功启动QEMU虚拟机



这里注意一下，在虚拟机里启动QEMU的虚拟机，可能会遇到光标切换不到虚拟机里的问题，使用快捷键 `ctl+option+g` 可以切出来mbp去ping kali虚拟机里的qemu虚拟机也是ping的通的，同样qemu的主机也可以上网

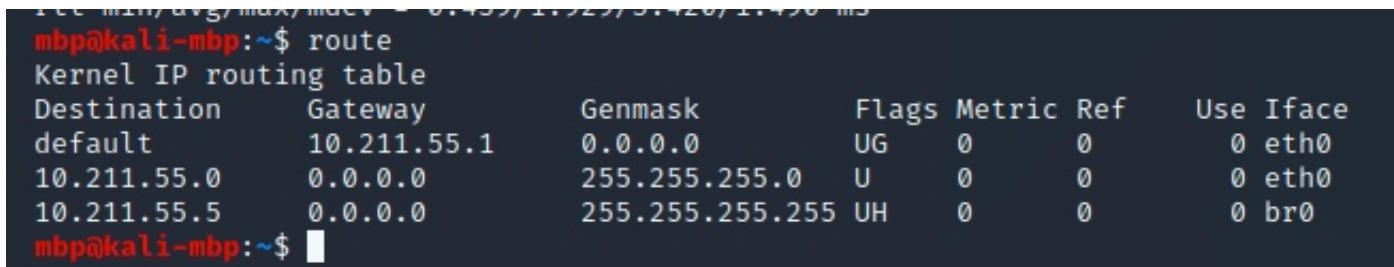


如果还有问题可能是路由的问题，之前看到有多余的路有条目



把这条删掉，然后单纯的加上到qemu的路由

```
sudo route del -net 10.211.55.0 netmask 255.255.255.0 gw 0.0.0.0 dev eth0
sudo route add -net 10.211.55.5 netmask 255.255.255.255 gw 0.0.0.0 dev br0
```



然后就正常了，这时候qemu虚拟机ping百度和pinghost主机都是ping的通的

0x03 挂载文件系统

我们qemu启动的系统是以下载到的那个系统启动的，我们还需要将自己需要分析的文件系统挂载到这个虚拟系统中进行运行。在做这个挂载之前，我们要判断一下如何进行挂载，一般有两种比较简单的方式：

1. 通过nfs网络共享进行挂载
2. 通过制作img镜像直接挂载

首先，我们创建空的img文件并格式化，这里为了确定要格式化成什么样的文件格式，我们可以先进入到qemu的虚拟机里进行查看

```
bin boot dev etc home lib lost+found media mnt opt pr
root@debian-armhf:/# mount
mount          mount.nfs  mount.nfs4  mountpoint  mountstats
root@debian-armhf:/# mount
```

可以看到mount支持nfs，这意味着我们只能通过nfs共享来挂载。

###1. 通过nfs共享进行挂载

nfs是网络共享，因此我们只需要起一个nfs服务把我们的target文件夹加入共享然后在qemu里通过网络远程挂载即可。先确保qemu虚拟机到host虚拟机的网络是通的，如果ping不通，回到前面的网络配置。

先安装nfs服务 `sudo apt-get install nfs-kernel-server` 安装完毕后我们配置一下nfs的配置文件，使得两个虚拟机间的网断可以访问nfs共享

```
# /etc/exports: the access control list for filesystems which may be exported
#                to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes      hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subt
#
# Example for NFSv4:
# /srv/nfs4       gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes gss/krb5i(rw,sync,no_subtree_check)
#
/home/mbp/workstation/iot 10.0.0.0/8(rw,sync) *(ro)
~
```

修改配置文件/etc/exports

这里我加入了我要共享的目录、哪个网段有权限访问、非该网段的用户访问权限，实验环境无所谓，只要能挂载就行。

修改完毕后，我们通过命令

```
sudo systemctl start nfs-server.service
```

等nfs服务启动后，我们切到qemu的虚拟机里，通过mount命令远程挂载nfs目录

```
mount -t nfs 10.211.55.3:/home/mbp/workstation/iot /mnt
```

挂载成功后，我们进入/mnt目录查看

```
root@debian-armel:~#
root@debian-armel:~# mount -t nfs 10.211.55.3:/home/mbp/workstation/iot /mnt
[ 246.094896] RPC: Registered udp transport module.
[ 246.096260] RPC: Registered tcp transport module.
[ 246.097180] RPC: Registered tcp NFSv4.1 backchannel transport module.
[ 246.163881] Slow work thread pool: Starting up
[ 246.165788] Slow work thread pool: Ready
[ 246.169030] FS-Cache: Loaded
[ 246.268722] FS-Cache: Netfs 'nfs' registered for caching
root@debian-armel:~# cd /mnt
root@debian-armel:/mnt# ls
amd64VM  armVM  targetfs
root@debian-armel:/mnt# cd targetfs/
root@debian-armel:/mnt/targetfs# ls
qemu-bridge.sh  _src.bin.extracted  web_cli  ♦ ♦ ♦ ♦ .mht
src.bin         sysctl              webmaster
root@debian-armel:/mnt/targetfs# cd _src.bin.extracted/
root@debian-armel:/mnt/targetfs/_src.bin.extracted# ls
240      2D8050.squashfs  404E8.7z  share.img
240.7z   404E8          mntcode  squashfs-root
root@debian-armel:/mnt/targetfs/_src.bin.extracted# cd squashfs-root/
root@debian-armel:/mnt/targetfs/_src.bin.extracted/squashfs-root# ls
bin  data  etc    lib  mnt    proc  root  sys  usr  web
conf dev  etc_ro lib64 overlay rom  sbin  tmp  var  www
root@debian-armel:/mnt/targetfs/_src.bin.extracted/squashfs-root#
```

可以看到我们的目录已经挂载到虚拟机里去了，并且可以访问。

2. 通过制作img镜像进行挂载

我们需要把文件系统打包成img镜像，然后通过命令参数在启动虚拟机时挂载到虚拟机里。

PS：这里我用的是amd64的虚拟机来做的测试，不用太在意前面的内容，主要是后面跟的参数不同

```
sudo qemu-system-x86_64 -hda debian_squeeze_amd64_standard.qcow2 -smp 2 -m 1024 -net nic -net tap,ifname=tap1,script=
/etc/qemu-ifup,downscript=no -drive file=./share.img
```



```
//增加了-drive参数来指定挂载img
```

那么我们先来制作一个空的img镜像，至于这个镜像要格式化成什么格式一般可以参考前面说的qemu里面支持的挂载方式，下面以ext4为例

```
dd if=/dev/zero of=share.img bs=1M count=1024
这相当于创建了1个G大的空的镜像文件，如果不够可以再改大
sudo mkfs.ext4 share.img
格式化成ext4文件系统
```

完成后我们将其挂载到host主机上，然后将目标文件夹拷贝进去

```
mkdir mntcode
sudo mount share.img mntcode
sudo cp -rfp targetfs/ mntcode/
sudo umount mntcode
```

现在我们把目标文件夹已经放入到share.img里了，但是因为我们创建的img总容量1g，我们需要把它调整成和文件一样大的

```
sudo e2fsck -p -f share.img
sudo resize2fs -M share.img
```

好了，img文件制作完毕，我们启动qemu

```
sudo qemu-system-x86_64 -hda debian_squeeze_amd64_standard.qcow2 -smp 2 -m 1024 -net nic -net tap,ifname=tap1,script=
/etc/qemu-ifup,downscript=no -drive file=./share.img
```

0x04 通过chroot复现原系统

现在我们已经在qemu虚拟机里挂载了目标文件系统，剩下的工作就比较简单了，由于我们想要以目标文件系统的结构来最终复现目标的原系统。那么我们的思路就是通过chroot的方式将目标文件系统作为根目录来运行。

在chroot前，我们需要将现有的qemu里的/dev、/sys、/proc三个目录挂载到目标文件系统里的对应目录，因为这三个文件夹和系统本身运行时有关。我们进入到需要chroot的目标文件系统的根目录，然后进行挂载

```
mount -t proc /proc proc/
mount --rbind /sys sys/
mount --rbind /dev dev/
```

```
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root# mount --rbind /sys sys/
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root# mount --rbind /dev dev/
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root# ls
bin  data  etc    lib    mnt    proc  root  sys  usr  web
conf dev  etc_ro lib64  overlay rom  sbin  tmp  var  www
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root# cd proc/
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root/proc# ls
1      1285  197  314  856      driver      loadavg  swaps
10     1287  2    32   9        execdomains locks    sys
1005   1288  20   33   903      fb          meminfo  sysrq-trigger
1017   1289  200  34   946      filesystems misc     sysvipc
11     1293  201  35   958      fs          modules  timer_list
12     13    21   36   acpi     interrupts  mounts   timer_stats
1234   1344  22   4    asound   iomem       mtrr     tty
1252   1377  23   462  buddyinfo ioports     net      uptime
1253   14    236  463  bus      irq         pagetypeinfo version
1254   15    24   491  cgroups  kallsyms    partitions vmallocinfo
1255   16    25   5    cmdline  kcore       sched_debug vmstat
1256   17    28   6    cpuinfo  keys        scsi      zoneinfo
1257   18    29   7    crypto   key-users   self
1258   19    3    734  devices  kmsg        slabinfo
1259   195   30   746  diskstats kpagecount  softirqs
1284   196   31   8    dma      kpageflags  stat
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root/proc# cd ..
root@debian-amd64:/mnt/_src.bin.extracted/squashfs-root# _
```

可以看到我们挂载了proc后，目标文件目录里的proc也有了内容，这时候我们再chroot

```
chroot . /bin/sh
```

0x05 arm64支持

前面的部分截图我是在非arm64环境下完成的，针对我目标文件系统的arm64的二进制文件就算完成了chroot也无法执行二进制文件，因为架构不一样。
所以我这边补充一下面对这种下载不到相关文件时候的方法

1. 自己编译内核和相关文件系统

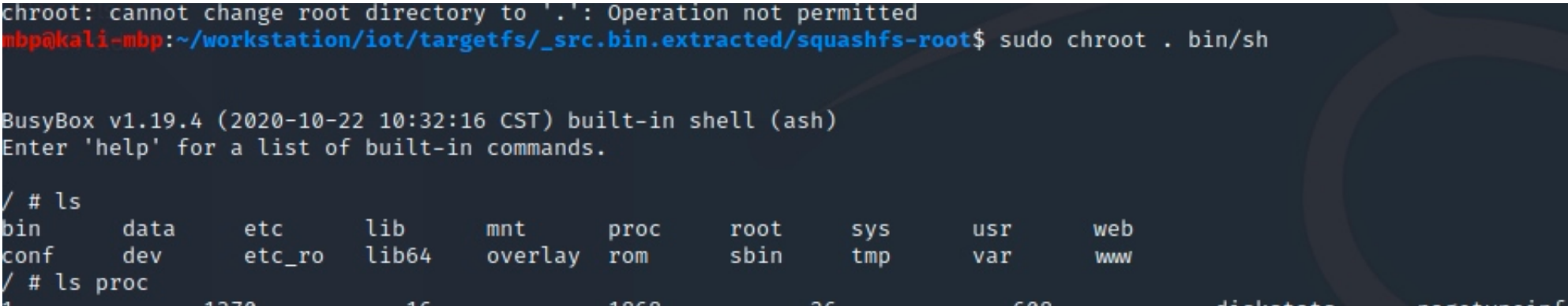
编译内核、制作初始化文件和文件系统等，最终作出一个环境，参考下面文章
主要参考：[利用Qemu-4.0虚拟ARM64实验平台](#)

2. 使用chroot+用户模式

这个其实之前就有提到，也简单，主要流程：

- 1. 进入到目标文件系统根目录
- 2. 执行mount
- 3. 按照前面说的使用用户模式的方式进行操作
- 4. 最后chroot就好

```
mount -t proc /proc proc/  
mount --rbind /sys sys/  
mount --rbind /dev dev/
```



可能的问题就是会和宿主机公用硬件、内存等，可能会导致一些问题，环境不够隔离

3. 使用docker

可以考虑使用docker来操作，网上有封装好的docker拿来用就行，不管他是通过系统模式还是用户模式，但至少有一层docker的隔离，基本上可以当作虚拟机来用了。
比如这个项目[docker-arm64](#)