内容简介

关于我

- 谭小凡
- 软件开发工程师，主要从事RISCV软件开发与移植工作

关于内容

在进行发行版软件包的编译、根文件系统构建以及开发板镜像制作时，开发人员常常面临如下痛点：目标板硬件稀缺或性能不足、不同开发者之间环境不统一导致复现困难、反复试错消耗大量时间。为了解决这些问题，本次议题聚焦 **qemu-user 与 docker 的协同应用**，为跨架构容器制作、软件包编译、开发板系统的制作提供高效的解决方案。

qemu-user 作为跨架构用户态软件的模拟工具，可在 x86 架构上通过翻译目标指令与系统调用的机制直接运行 RISCV、ARM、MIPS 等目标架构的二进制程序，与 docker 配合则能结合容器化技术，直接在x86架构的开发机上便捷运行目标架构的容器。

议题将从实际开发场景出发，逐步讲解如何安装qemu-user、执行目标架构的容器、制作目标架构的发行版容器、快速移植开发板系统等问题。帮助开发人员跳过不必要的试错环节，提升软件和系统移植的效率。

QEMU常见的三种工作场景

模式	架构要求	功能	性能
qemu-system + kvm	Host和Guest同架构	虚拟机运行完整操作系统	很好
qemu-system	可跨架构	模拟器运行完整操作系统	较差
qemu-user	可跨架构	模拟器运行用户态程序，将系统调用翻译到Host Kernel上	中等

在跨架构模拟执行时，通常qemu-user的性能优于qemu-system的性能。

因为 qemu-system 内需运行一个完整的操作系统内核，当不使用KVM加速时，就只能使用TCG引擎来做二进制翻译，性能较差。

qemu-user 只负责用户态程序的模拟，拦截Guest Program的syscall，将其翻译为对本机内核的系统调用。虽然也需要使用TCG引擎做二进制翻译，但无需在QEMU中模拟一个完整的内核，所以性能相对qemu-system要好一些

模拟执行方式的性能测试

在openEuler-24.03-lts下，通过rpmbuild构建软件包，比较qemu-system和qemu-user的性能差距

编译软件包	qemu-system-riscv64	qemu-user-riscv64
bash-5.2.15-9.oe2403	real 10m17.595s user 22m45.558s sys 11m30.601s	real 7m28.796s user 17m0.117s sys 0m52.560s
nginx-1.24.0-1.oe2403	real 6m17.300s user 39m10.105s sys 12m59.054s	real 3m8.487s user 19m45.362s sys 0m49.696s

解释执行静态程序

```
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ cat hello.c
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 0;
}
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ riscv64-linux-gnu-gcc -
static hello.c -o hello
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ qemu-riscv64 ./hello
hello world
```

注

- 静态链接的C程序**只依赖于内核的syscall**，不依赖于其他的运行时环境
- **qemu-riscv64** 是riscv64作为target的qemu-user
- **qemu-riscv64**对用户态程序hello做模拟执行，将对riscv64内核的系统调用翻译为对本机的系统调用

解释执行动态程序

```
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ cat hello.c
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 0;
}
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ riscv64-linux-gnu-gcc
hello.c -o hello
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ qemu-riscv64 ./hello
qemu-riscv64: Could not open '/lib/ld-linux-riscv64-lp64d.so.1': No such file or
directory
```

这里遇到第一个问题，缺少riscv64的动态链接器。

动态链接的程序需要动态链接器才能运行，这个动态链接器的路径是以绝对路径的形式写到ELF文件的**program header**中

```
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ readelf -l hello

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x5b0
There are 10 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x0000000000000230 0x0000000000000230  R      0x8
  INTERP         0x0000000000000270 0x0000000000000270 0x0000000000000270
                 0x0000000000000021 0x0000000000000021  R      0x1
      [Requesting program interpreter: /lib/ld-linux-riscv64-lp64d.so.1]
  RISCV_ATTRIBUT 0x0000000000001033 0x0000000000000000 0x0000000000000000
                 0x0000000000000053 0x0000000000000000  R      0x1
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x000000000000073c 0x000000000000073c  R E    0x1000
  LOAD           0x0000000000000db0 0x0000000000001db0 0x0000000000001db0
                 0x0000000000000258 0x0000000000000260  RW     0x1000
  DYNAMIC        0x0000000000000dc8 0x0000000000001dc8 0x0000000000001dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000294 0x0000000000000294 0x0000000000000294
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x00000000000006cc 0x00000000000006cc 0x00000000000006cc
                 0x000000000000001c 0x000000000000001c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000db0 0x0000000000001db0 0x0000000000001db0
                 0x0000000000000250 0x0000000000000250  R      0x1
```

但是在x86-64的环境下，并不存在**/lib/ld-linux-riscv64-lp64d.so.1**这个文件，所以无法直接执行动态链接的程序。

好在这个文件名不会和x86-64的动态链接器名字冲突，所以我们可以从riscv64的sysroot中拷贝这个文件到系统的/lib目录下

```
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ sudo cp /usr/riscv64-
linux-gnu/lib/ld-linux-riscv64-lp64d.so.1 /lib
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$ qemu-riscv64 ./hello
./hello: error while loading shared libraries: libc.so.6: cannot open shared
object file: No such file or directory
```

除了动态链接器以外，还需要一个 libc.so.6 这个动态库。这个名字和x86-64环境中的libc的名字相同，我们难以直接通过cp指令拷贝riscv64的libc.so.6到系统动态库的目录下。但我们可以设置LD_LIBRARY_PATH来帮助动态链接器搜索到riscv64版本的libc.so.6。

```
xiaofan@xfan-ubuntu2404-devel:~/workspace/riscv-compile$
LD_LIBRARY_PATH=/usr/riscv64-linux-gnu/lib qemu-riscv64 ./hello
hello world
```

至此成功通过qemu-user运行了一个riscv64的动态链接的程序。

linux的binfmt_misc机制

我们现在已知了两个问题

- 执行目标可执行文件时，常会遇到宿主机上缺少动态链接器和依赖的so库等问题，虽然可以通过一些技术手段来满足，但实际操作过程很麻烦。
- 执行目标可执行文件时，需要手动执行qemu-riscv64这个可执行文件，比较麻烦。

考虑这两个问题的解决办法

- chroot到一个目标架构的根文件系统中，在这个根文件系统中的动态链接器、动态库都是目标架构的文件
- 考虑使用linux的binfmt_misc机制，Linux 的 `binfmt_misc` 允许注册多条规则（每条包含 `magic`、`mask`、`interpreter` 等），内核在 `exec` 某个文件时会用规则的 `mask` 对文件头按位与，再判断 `(file_header & mask) == magic`，若匹配则由该规则指定的解释器来运行该文件。（如果不容易理解参照IP地址和子网掩码来计算网段的方式进行理解）

还会有一个额外的问题

- 在choot环境下，也需要qemu-user的可执行文件、qemu-uesr依赖到的动态链接器、动态库。虽然可以提前将这些放进目标根目录下，但仍然不是很方便

这个问题的解决

- 使用 qemu-user-static ，它是静态链接的可执行文件，不会依赖到额外的动态链接器、动态链接库。
- binfmt_misc的规则中允许设置一个F标志(fix binary)，其原理是在注册解释器时不仅仅是登记解释器的路径，内核会将这个解释器载入内核中（实际上是持有fd），即使在chroot环境中，解释器路径不存也不影响，因为内核已经提前将解释器载入到内核中，无需依赖解释器的路径就能执行解释器。

```
F - fix binary
The usual behaviour of binfmt_misc is to spawn the binary lazily when the misc
format file is invoked. However, this doesn't work very well in the face of mount
namespaces and changeroots, so the F mode opens the binary as soon as the
emulation is installed and uses the opened image to spawn the emulator, meaning
it is always available once installed, regardless of how the environment changes.
```

binfmt_misc示例

```
:qemu-
riscv64:M::\x7f\x45\x4c\x46\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x
00\xf3\x00:\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff\xff\xff\xff\xff\xfe\x
ff\xff\xff:/usr/libexec/qemu-binfmt/riscv64-binfmt-P:OPF
```

在这个例子中

字段	值	解释
name	qemu-riscv64	支持多个binfmt_misc规则，使用名字作为ID

字段	值	解释
magic	\x7f\x45\x4c\x46\x02\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\xf3\x00	参考ELF文件头部
mask	\xff\xff\xff\xff\xff\xff\xff\x00\xff\xff\xff\xff\xff\xff\xff\xff\xfe\xff\xff\xff	参考ELF文件头部
interpreter	/usr/libexec/qemu-binfmt/riscv64-binfmt-P	解释器路径，实际指向qemu-riscv64-static
flags	OPF	

flags

- P: 将解释器路径作为argv[0]，然后把原始命令作为额外的参数传给解释器
- O: 内核打开要被解释的目标程序，通过fd的方式直接传递给解释器。如果直接传文件可能会出现遇到只有可执行权限没有读权限，但无法运行的情况。
- C: 将进程凭据按照被执行的二进制来计算，而不是根据模拟器的二进制来计算。模拟执行sudo/su等依赖于SUID/SGID的程序来说需要设置。
- F: 将解释器直接加载到内核中，启动时无需通过解释器路径去启动。

到这里，我们已经满足了直接在chroot环境中或mount namespaces环境中无感使用qemu-user-static的准备工作

- 可以直接执行riscv64的elf文件，无需手动调用qemu解释器程序。内核自动通过binfmt_misc机制识别哪些文件由哪个解释器执行
- 通过binfmt_misc的F标志，将解释器加载到内核中，即使在chroot/mount namespaces环境下找不到解释器路径，也不妨碍解释器执行
- qemu-user采用静态链接，不依赖任何用户态的运行时依赖，保证在chroot/mount namepsaces下可以正常工作

与docker的配合使用

准备

在Ubuntu的开发机上，我们可以直接使用

```
apt-get install qemu-user-static
```

来安装各种架构的qemu解释器，并自动设置binfmt_misc。

验证

```
$ docker run --rm -it xfan1024/openeuler:24.03-x86_64 uname -m
x86_64
$ docker run --rm -it xfan1024/openeuler:24.03-aarch64 uname -m
WARNING: The requested image's platform (linux/arm64) does not match the detected
host platform (linux/amd64/v4) and no specific platform was requested
aarch64
$ docker run --rm -it xfan1024/openeuler:24.03-riscv64 uname -m
WARNING: The requested image's platform (linux/riscv64) does not match the
detected host platform (linux/amd64/v4) and no specific platform was requested
riscv64
```

其他安装方式

既然qemu-user-static是一组静态的可执行文件，并且在设置了F标志时它们只需要注册到内核中，而不需要在使用它的进程的文件系统视角下能找到它。

所以我们还可以考虑直接通过 docker 直接安装这一组解释器到内核中

```
docker run --rm --privileged multiarch/qemu-user-static # host支持 x86-64
docker run --rm --privileged xfan1024/qemu-user-static  # host支持 x86-
64/arm64/riscv64，如果在MACOS(M系列CPU)中使用，可以使用这个
```

这样一来，即使你的发行版上找不到 qemu-user-static 软件包，也可以通过docker容器来安装qemu-user-static二进制到内核中

qemu-user-static容器镜像的制作方法

- **multiarch/qemu-user-static** https://github.com/multiarch/qemu-user-static
- **xfan1024/qemu-user-static** https://github.com/xfan1024/qemu-user-static

制作openEuler-riscv64的容器

```
# 先从一个本机架构的镜像出发，作为目标架构的根文件系统准备环境
FROM openeuler/openeuler:24.03-lts AS build

# 注意 --forcearch riscv64 与 --installroot /target
# 当使用 --forcearch riscv64 时，需要借助qemu-user-static的能力来做安装工作
#   Force the use of an architecture. Any architecture can be
#   specified.  However, use of an architecture not supported
#   natively by your CPU will require emulation of some kind.
#   This is usually through QEMU
RUN dnf --setopt=install_weak_deps=False --releasever 24.03LTS --forcearch
riscv64 --installroot /target \
    install -y coreutils rpm dnf yum bash findutils procps tar && \
    dnf clean all --installroot /target && \
    rm -rf /target/var/cache/yum && \
    rm -rf /target/var/log/* && \
    rm /target/var/lib/dnf/history.sqlite-*

# 从一个空白镜像出发，将build镜像中的/target目录作为当前镜像的根目录
FROM --platform=linux/riscv64 scratch
COPY --from=build /target /
CMD [ "/bin/bash" ]
```

详细制作方法可参考

- 从零制作docker镜像：openEuler(riscv64) https://zhuanlan.zhihu.com/p/636350939

我自行维护了openeuler多种架构的容器镜像，可以直接取用

- xfan1024/openeuler https://hub.docker.com/r/xfan1024/openeuler

快速移植开发板系统

文件系统制作

```
FROM --platform=linux/riscv64 xfan1024/openeuler:24.03-lts-riscv64
RUN dnf install -y kernel linux-firmware dracut grub2 grub2-efi-riscv64-modules \
    dbus NetworkManager systemd-timesyncd \
    && systemctl enable systemd-timesyncd \
    && echo "root:openEuler12#$" | chpasswd
```

镜像制作

mkimage.sh

这个脚本需要xfan1024/genimage的容器中执行，也就是说创建可启动镜像的过程也是在一个容器化的环境中完成的。

使制作镜像的工具不必在开发机中直接安装，也避免了开发机环境不一致导致脚本执行失败。

```
#!/bin/bash
# NOTE: this file used in docker container
#       don't execute it directly
#                                -- xiaofan

set -xe

# now we are in /work
# /work/rootfs.tar mounted from host (read only)
# /work/board mounted from host (read only)
# /work/output mounted from host (read/write)
# /work/rootfs is temporary directory, store all files in rootfs partition
# /work/efi is temporary directory, store all files in efi partition

export GRUB_DISABLE_OS_PROBER=true

chroot_prepare() {
    mount --bind /dev $1/dev
    mount --bind /proc $1/proc
    mount --bind /sys $1/sys
    mount -t tmpfs tmpfs $1/tmp
}

chroot_cleanup() {
    umount $1/dev
    umount $1/proc
    umount $1/sys
    umount $1/tmp
}

mkdir -p rootfs
tar -C rootfs -xf rootfs.tar

ESP_UUID=0045-5350
ROOT_UUID=78662d6f-706f-6570-6f72-742d726f6f74
```

```
rm -f rootfs/.dockerenv
rm -f rootfs/etc/resolv.conf
rm -f rootfs/etc/fstab

cat >rootfs/etc/fstab <<EOF
UUID=$ROOT_UUID /           ext4    defaults    0   1
UUID=$ESP_UUID /boot/efi    vfat    defaults    0   2
EOF

echo "oe2403lts-rv64" >rootfs/etc/hostname
mv rootfs/boot/efi efi
mkdir rootfs/boot/efi
cp efi/EFI/openEuler/grubriscv64.efi efi/EFI/BOOT/BOOTRISCV64.EFI

genimage --config board/genimage.cfg --input . --rootpath rootfs

LOOPDEV=$(losetup -fP --show images/oe2403lts-rv64-qemu.img)
mkdir -p mnt
mount ${LOOPDEV}p2 -o ro mnt
mount ${LOOPDEV}p1 -o rw mnt/boot/efi
chroot_prepare mnt
chroot mnt grub2-mkconfig -o /boot/efi/EFI/openEuler/grub.cfg
chroot_cleanup mnt
umount mnt/boot/efi
umount mnt
losetup -d $LOOPDEV

# compress and set ownership (if provided)
qemu-img convert -f raw -O qcow2 images/oe2403lts-rv64-qemu.img output/oe2403lts-
rv64-qemu.qcow2
cp board/{*.fd,start_vm.sh} output/
if [ -n "$HOSTUID" -a -n "$HOSTGID" ]; then
    cd output/
    chown -R $HOSTUID:$HOSTGID oe2403lts-rv64-qemu.qcow2 RISCV_VIRT_CODE.fd
RISCV_VIRT_VARS.fd start_vm.sh
fi
```

genimage.cfg

```
image esp.vfat {
    vfat {
        label = "ESP"
        extraargs = "-i 00455350"
        files = {
            "efi/EFI",
        }
    }
    size = 1GB
}

image rootfs.ext4 {
    ext4 {
        label = "rootfs"
```

```
        features = "filetype"
        use-mke2fs = true
        extraargs = "-U 78662d6f-706f-6570-6f72-742d726f6f74"
    }
    size = 10G
}


image oe2403lts-rv64-qemu.img {
    hdimage {
        partition-table-type = "gpt"
    }

    partition esp {
        offset = 2M
        image = "esp.vfat"
        partition-type-uuid = U
        bootable = true
    }

    partition rootfs {
        image = "rootfs.ext4"
        partition-type-uuid = L
    }
}
```

示例项目开源

- xfan1024/mkoervimg https://github.com/xfan1024/mkoervimg
- xfan1024/openeuler-d1 https://github.com/xfan1024/openeuler-d1
- xfan1024/openeuler-rk3568-rock3a https://github.com/xfan1024/openeuler-rk3568-rock3a

软件包编译

几乎所有的发行版的软件包都可以在这个发行版的系统环境下构建（例如openEuler，当然也有例外，如OpenWRT）。

我们可以直接利用qemu-user + docker来搭建这个发行版的容器镜像制作对应的编译环境。

实现在开发机上以用户态模拟的方式，方便的构建各种架构各种发行版的软件包。

曾经一些riscv64的构建平台基础设施使用的是qemu-system环境模拟的完整系统，

这种方式进行模拟的话理论上能获得更好的兼容性，其代价是更慢的编译性能。