

实时操作系统基本知识

- [实时操作系统基本知识](#)
 - [Linux实时操作系统](#)
 - [双内核操作系统基本知识](#)
 - [Xenomai操作系统](#)
 - [rros](#)
 - [rros编译](#)
 - [docker安装及拉取代码](#)
 - [rros编译过程](#)
 - [使用Qemu进行模拟同时使用gdb对进行Debug](#)
 - [引用](#)

Linux实时操作系统

Linux是一个通用操作系统，而不是一个实时操作系统。目前有多种方法可以让Linux成为一个实时操作系统或者具备实时操作系统的能力：

- 一个广泛使用的方案是让给实时操作系统提供兼容POSIX的API接口来改善生态问题，但是这个方法在面对大型项目的移植时的效果不好，也没有解决性能问题；
- 另一条技术路线是可以对Linux内核采用抢占补丁，但是这个方法只能让Linux成为一个软实时的操作系统，同时没有任何的隔离措施，不利于稳定性；
- 还可以采用虚拟机来同时在一个主机上同时运行多个内核，一个是较为简单的硬实时内核，另一个是Linux内核，这个方法的稳定性很强，但是由于采用了硬件虚拟化，实时性和性能受损，同时在Linux内核上运行的应用不具备实时性能，在实时内核上运行的应用不能享受到Linux生态的好处，两者之间的数据交互比较困难；
- 最后一条路线是采用双内核的方法，在一个内核空间里面同时运行两个内核，实时内核和Linux同时并行，实时内核的优先级更高，Linux内核作为一个idle任务在实时内核中调度，这个方法同时兼顾了实时性，性能和应用生态，实时任务可以同时利用Linux的生态和实时应用的能力，但是这个方法的缺点是稳定性不足，因为两个内核同处一个地址空间，所以如果Linux内核出现故障，没有任何的隔离措施，很容易导致操作系统崩溃；

我们的rros（rust-based real operating system）就是采用了双内核的技术路线，下面首先介绍一下双内核操作系统。

双内核操作系统基本知识

双内核操作系统主要分为两部分，一个是硬件虚拟层（HAL），主要完成中断虚拟化等相关工作；第二个是实时内核，主要负责处理实时请求，和Linux内核在逻辑上是并列的关系，但是优先级要比Linux内核高。

双内核的实现主要分为RTLinux，RTAI，Xenomai三个项目。双内核最早的实现RTLinux是在[1]这篇论文中提出的。硬件虚拟层是双内核路线中的关键一环，RTHAL是在RTLinux的论文中提出来的，被另一个实时操作系统RTAI仿照实现，但是RTLinux的项目组后来申请了专利，RTAI被迫采用了其他硬件虚拟层技术ADEOS。ADEOS是在[2]这篇论文中提出的，采用不同方法换了RT-Linux提出的RTHAL，规避了专利问题。所以后来RTAI和Xenomai社区采用了ADEOS。

拓展阅读 ADEOS的实现细节在[3]中可以看到；
ADEOS和RTHAL两种硬件虚拟层技术的比较在[4]中可以看到；

RTLinux自从被同类竞品VxWorks收购后，已经从开源逐步走向关停。而Xenomai和RTAI两个项目在一段时间有过短暂的合并。但是后来因为开发的目标不同，两个项目又逐渐分离。rros主要仿照的就是Xenomai操作系统。

Xenomai操作系统

Xenomai目前已经进展到4.0了。Xenomai4.0主体是两个部分，硬件中断层dovetail和实时内核evl。dovetail主要是以代码树形式提供，直接修改了Linux内核的代码，作用是根据优先级将中断分发给cobalt内核和evl内核。实时内核evl则作为一个module插入到linux系统中，和Linux内核一起启动，启动后接管整个操作系统。

rros

rros采用了dovetail硬件中断层，用rust重写了实时内核evl，下面我们将会介绍rros的基本情况。

项目的代码树主体结构如下，相对于Linux代码树新增的文件用*标出，一些重要的目录或者文件我们加以解释：

```

.
├── arch
├── block
├── certs
├── COPYING
├── CREDITS
├── crypto
├── .config
├── Documentation
├── drivers
├── fs
├── gr
├── include
├── init
├── io_uring
├── ipc
├── Kbuild
├── Kconfig
├── kernel
├── acct.c
├── ...
├── rros
├── built-in.a
├── clock.rs
├── clock_test.rs
├── crossing.rs
├── double_linked_list_test.rs
├── factory.rs
├── fifo.rs
├── fifo_test.rs
├── file.rs
├── idle.rs

```

这个文件中包含了Linux编译时Linux项目的文档

* 包含了运行rust-gdb命令的文件

Linux内核的主要代码

* rros实时内核的主要代码

```

├── init.o
├── init.rs
├── libinit.rmeta
├── list.rs
├── list_test.rs
├── lock.rs
├── Makefile
├── memory.rs          内存子系统
├── modules.order
├── monitor.rs
├── net.rs
├── queue.rs
├── sched.rs          调度子系统
├── sched_test.rs
├── stat.rs
├── stat_test.rs
├── syscall.rs
├── test.rs
├── thread.rs        线程子系统
├── thread_test.rs
├── tick.rs          tick子系统
├── timeout.rs
├── timer.rs         时钟子系统
├── timer_test.rs
├── wait.rs
├── weak.rs
├── ...
├── workqueue.o
├── lib
├── LICENSES
├── MAINTAINERS
├── Makefile
├── mm
├── modules.builtin
├── modules.builtin.modinfo
├── modules.order
├── Module.symvers
├── net
├── README
├── rust              * rust-for-linux的代码
├── samples          包含了各个子系统的一些示例代码
├── scripts
├── security
├── sound
├── System.map
├── tools
├── usr
├── virt
├── vmlinux
├── vmlinux.a
├── vmlinux.o

```

因为dovetail硬件中断层已经合入到代码树中，通过[git log](#)的历史记录是看不出来的，如果要知道dovetail修改了哪些内容，可以从[patch-5.15.9-dovetail1.patch](#)这个patch中看到。

rust的支持是通过rust-for-linux (rfl) 项目[5]，rfl目前已经合入linux主线，由于历史原因，我们项目中rfl的支持是通过补丁的形式进行的，和目前主线上的rfl不兼容。rfl项目支持我们用rust写linux的驱动，我们的rros就是以驱动的形式加载到linux内核中的。

rros编译

为了方便大家做实验，我们以docker的形式提供一个可用的环境，大家只需要安装docker并拉取镜像，然后按照我们的编译说明进行编译，就可以做后续的实验了。

docker安装及拉取代码

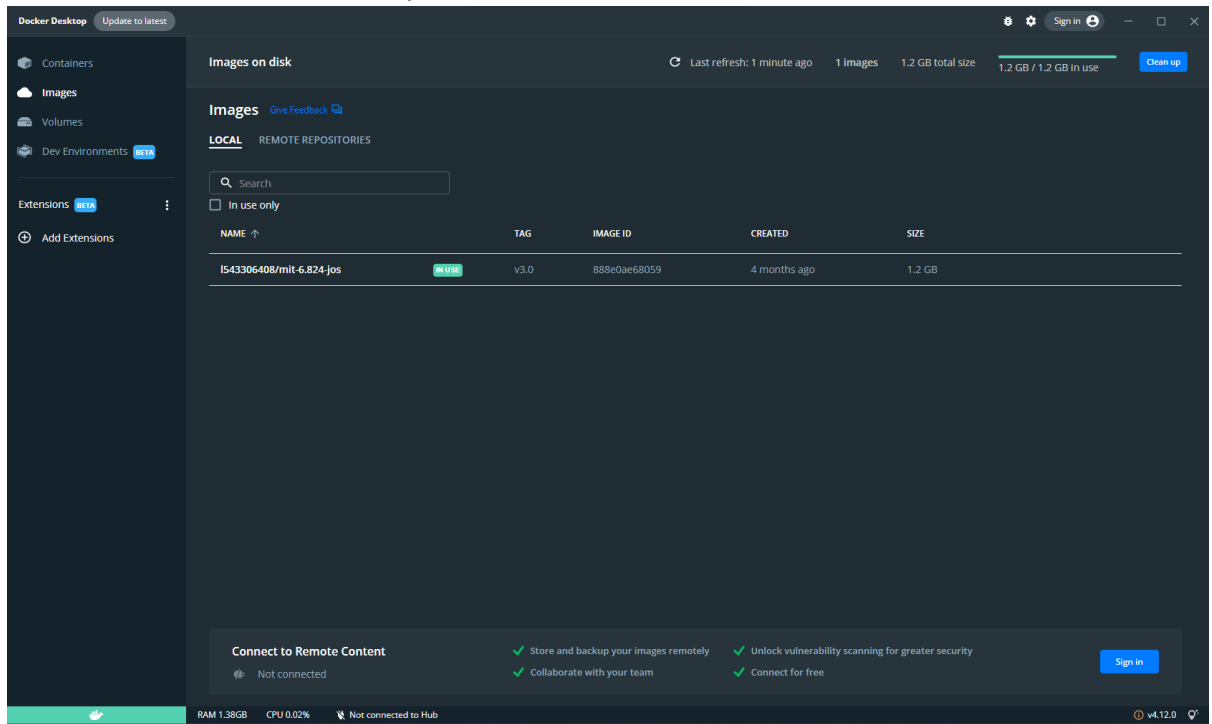
docker在windows/linux/mac上都可以直接安装，主要参考[官方的文档](#)，具体步骤方法如下：

- 在windows上安装
 - 下载[docker desktop](#)，并点击安装。
 - 安装完docker后，如果提示因为wsl2没有安装不能正常启动的话，这是因为在windows上使用docker需要开启wsl2或者hyper-v相关的组件，我们这里采用wsl2，这部分内容参考微软的[官方说明](#)。
 - 以管理员身份打开 PowerShell (“开始”菜单 > “PowerShell” > 单击右键 > “以管理员身份运行”)，然后输入以下命令：

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart
dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /norestart
```

- 下载[wsl更新包](#)，并安装执行。

- 重启电脑，并启动docker desktop，就可以正常启动了



- 在linux ubuntu/mac上安装
 - 这里考虑到linux上安装时可能没有图形化界面，所以下面用命令行说明
 - linux上运行docker的原理是使用kvm虚拟化技术，可以使用下列命令检测linux是否满足docker的条件

```
lsmod | grep kvm
```

```
lhy@Phoenix22:~$ lsmod | grep kvm
kvm_intel          253952    0
kvm                659456    1 kvm_intel
```

正确的输出如下:

- 对linux软件包安装地址进行换源

```
sudo add-apt-repository "deb [arch=amd64]
https://mirrors.aliyun.com/docker-ce/linux/ubuntu $(lsb_release -cs)
stable"
sudo apt-get update
```

- 安装docker

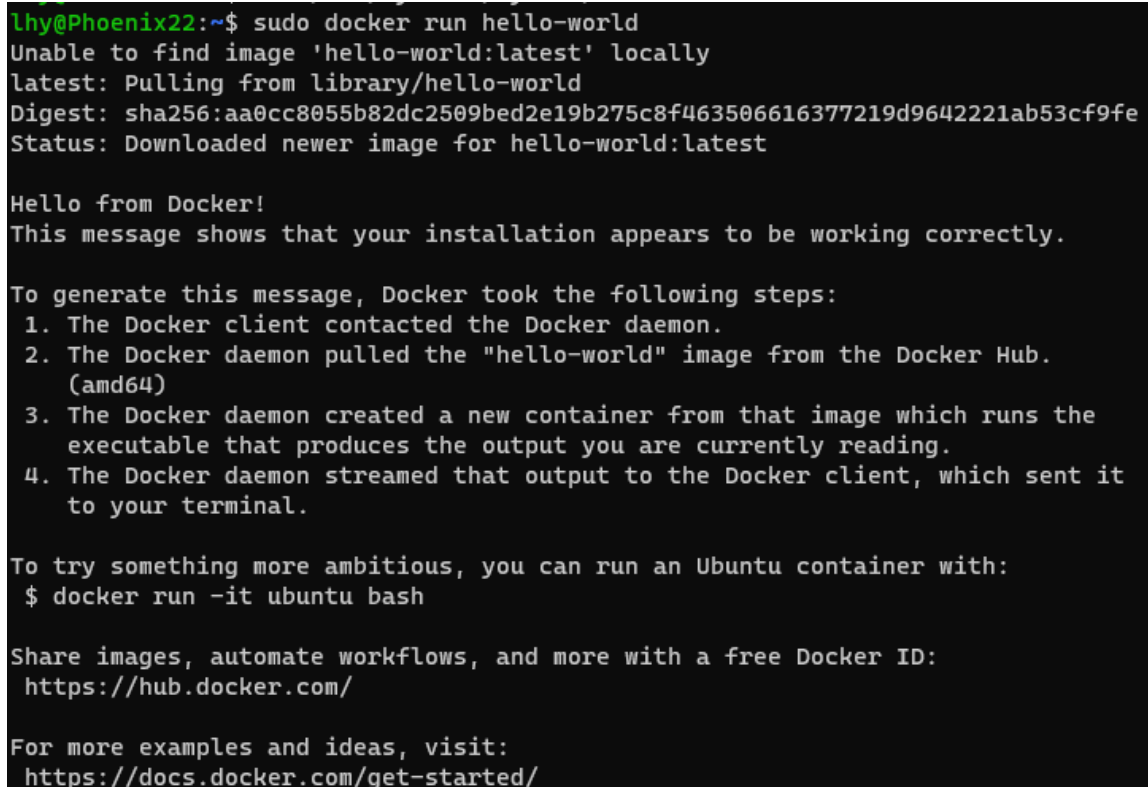
```
sudo apt-get install docker-ce
sudo docker run hello-world
```

- docker换源

```
vim /etc/docker/daemon.json
# 加入下面的内容
# {
#   "registry-mirrors": ["https://akchsm1h.mirror.aliyuncs.com"]
# }
```

- 检查是否可以正常执行

```
sudo docker run hello-world
```



```
lhy@Phoenix22:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
Digest: sha256:aa0cc8055b82dc2509bed2e19b275c8f463506616377219d9642221ab53cf9fe
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (amd64)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

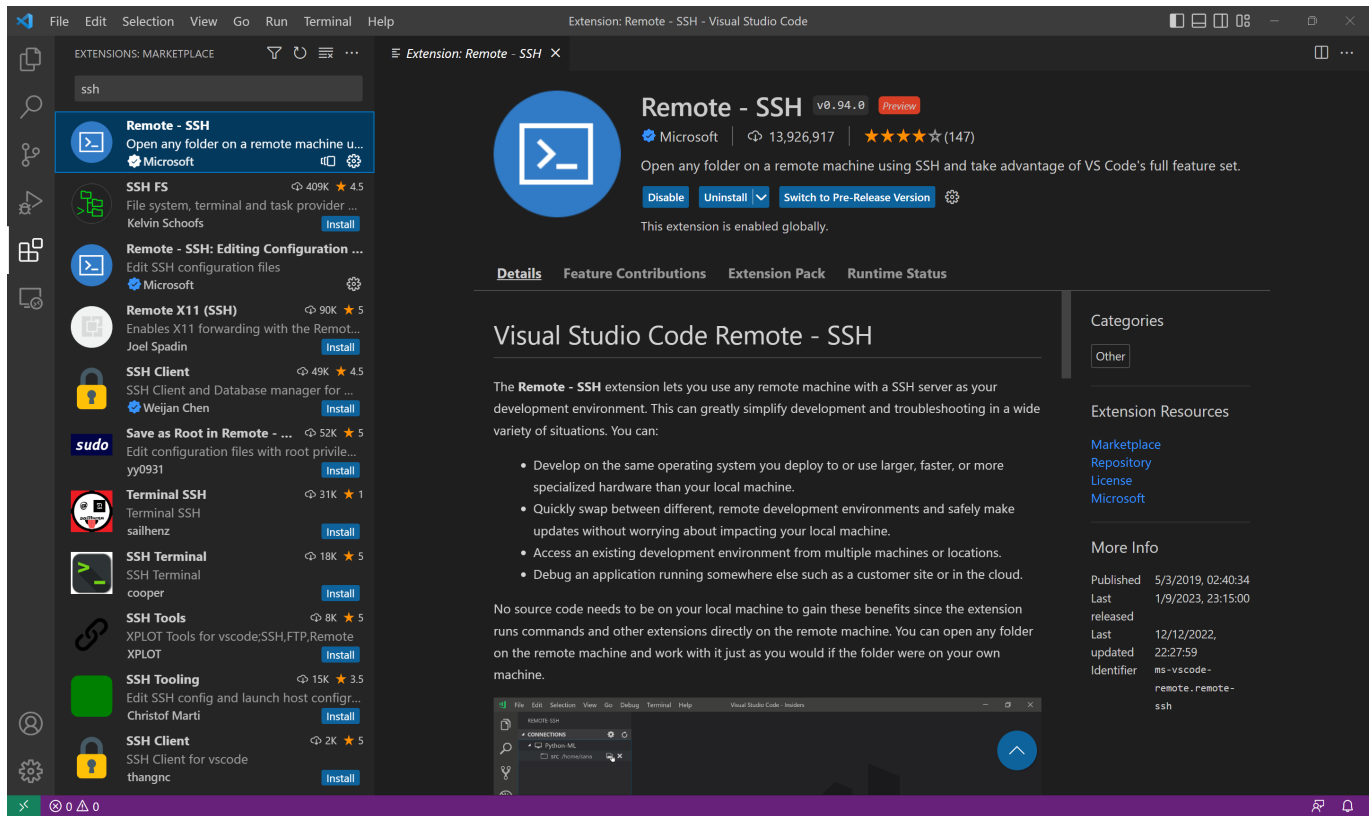
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

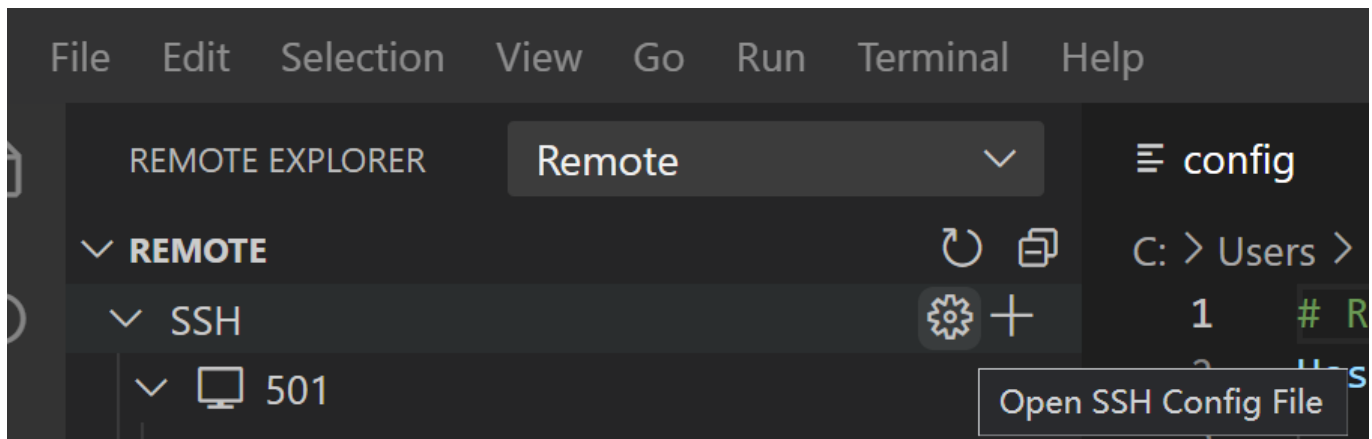
[TODO: 我这里没有mac的环境，但是应该和windows/linux安装的过程是相似的，之后补充一下]

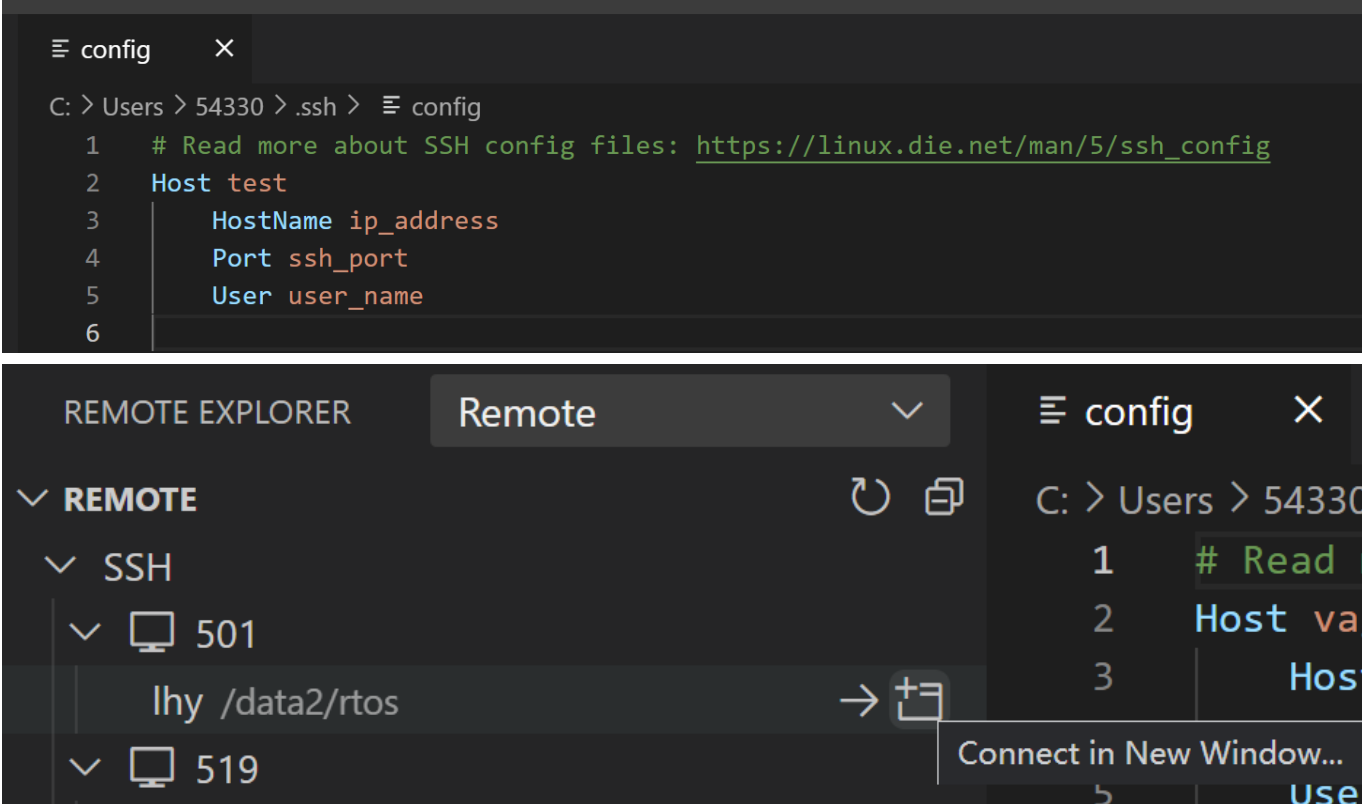
安装完成后，打开命令行窗口，使用 `docker pull 1543306408/bupt-rtos:v0.3` 命令来拉取 rros docker 的镜像 image。接着使用 `docker run -itd --name rros_lab 1543306408/bupt-rtos:v0.3` 来运行一个名为 rros_lab 的 container。

最后我们利用vscode来完成后续实验。首先在vscode中安装remote-ssh插件,

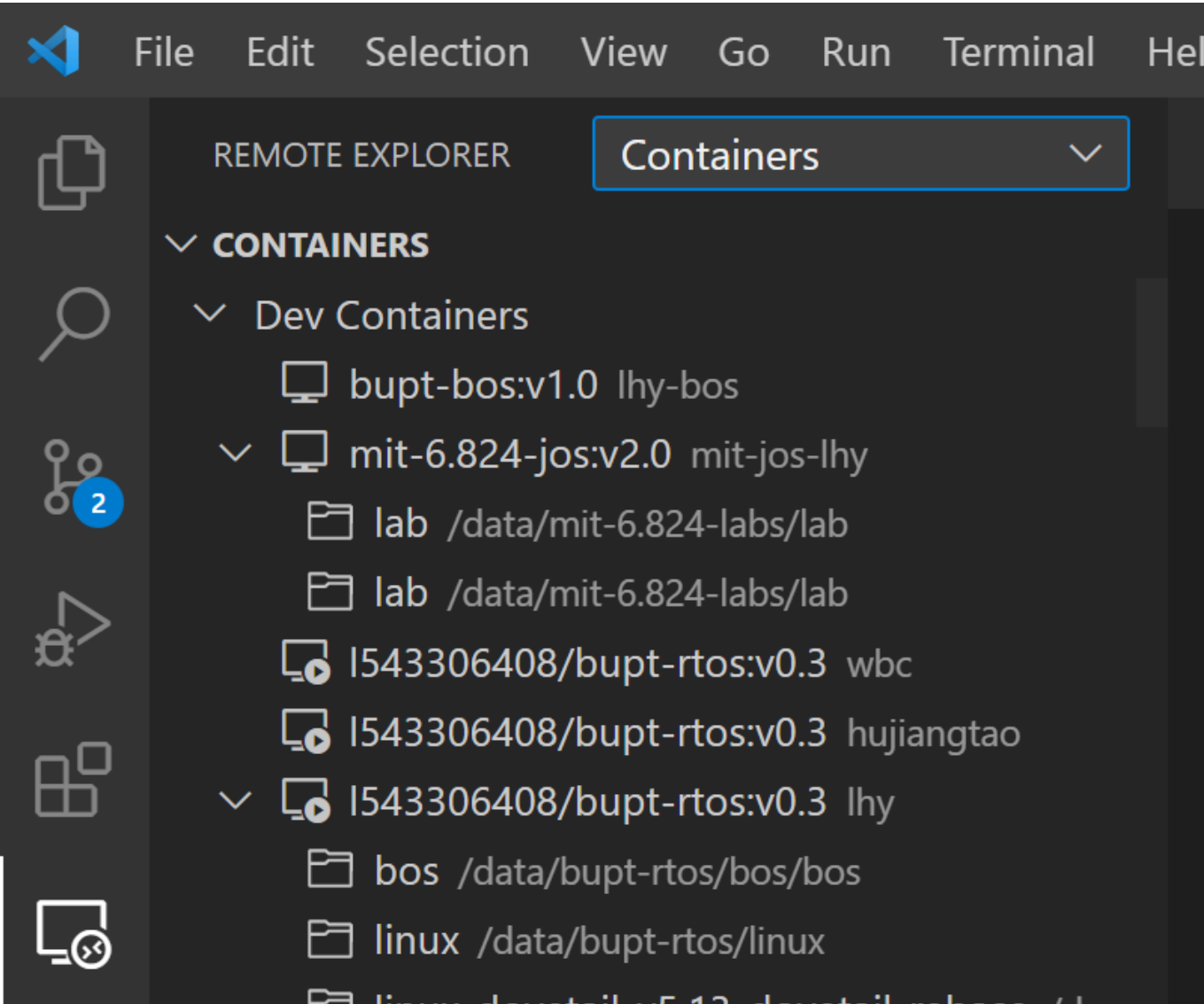


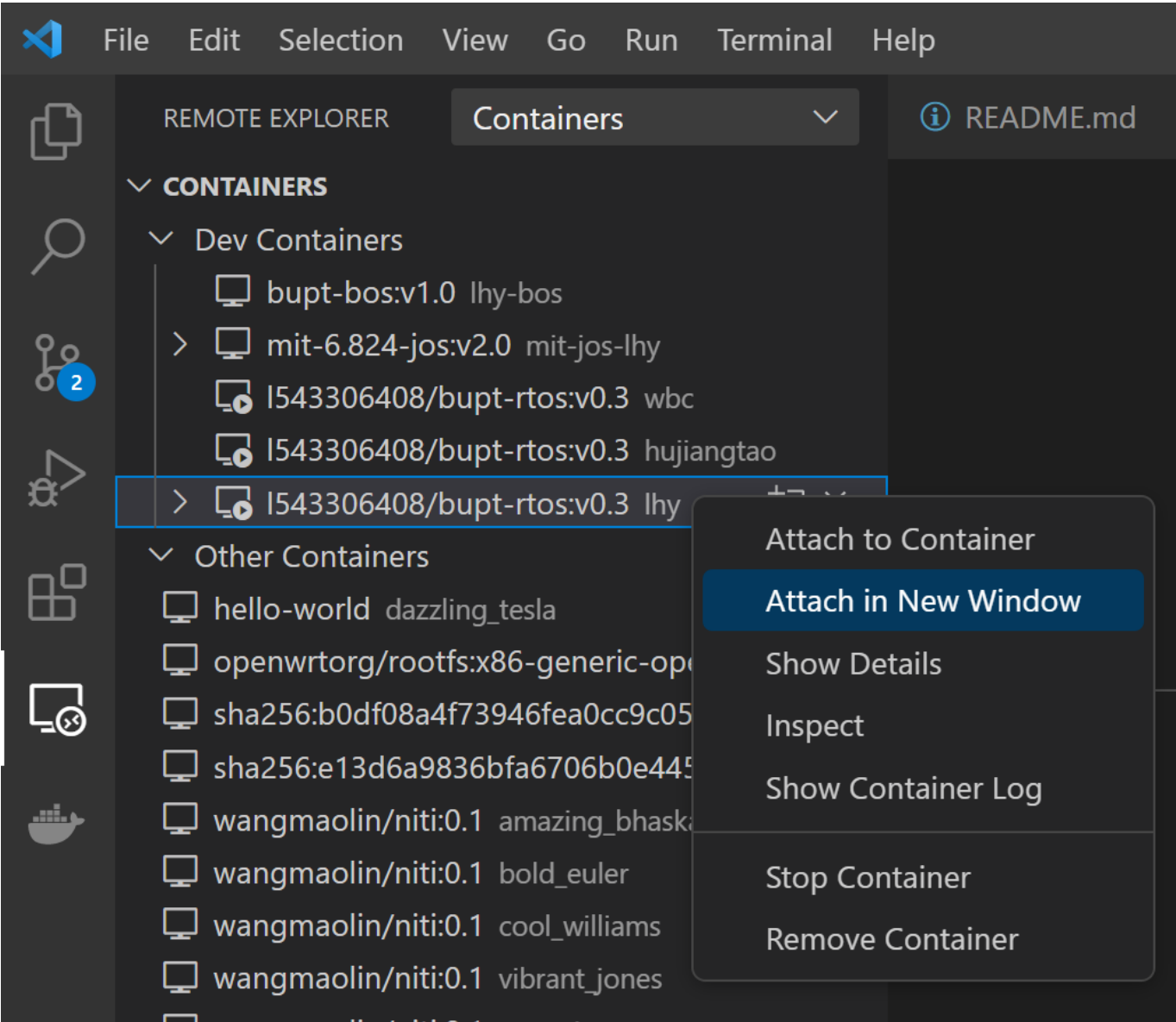
然后我们编辑一下linux环境的相关配置信息，就可以通过ssh连接这个linux环境了

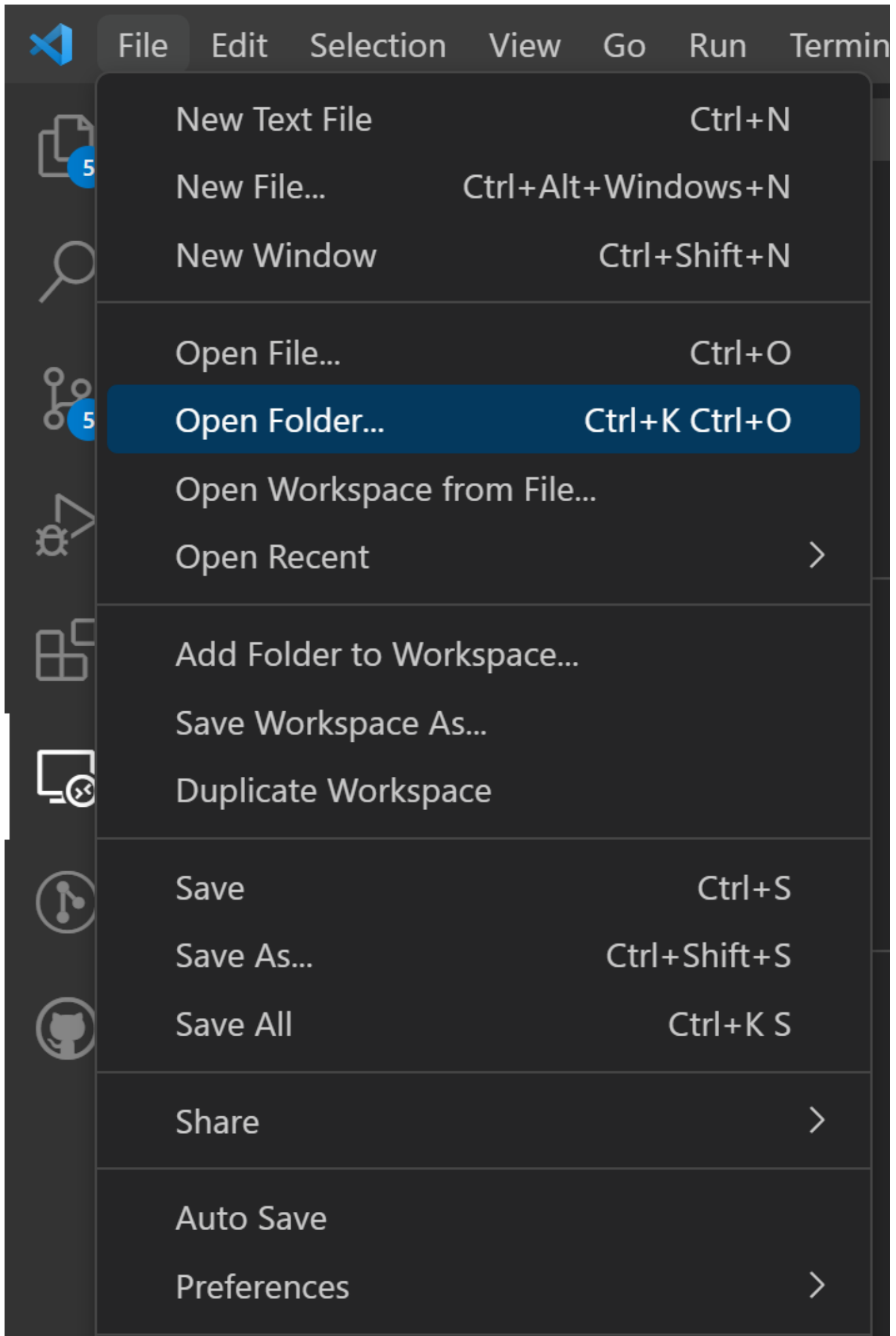


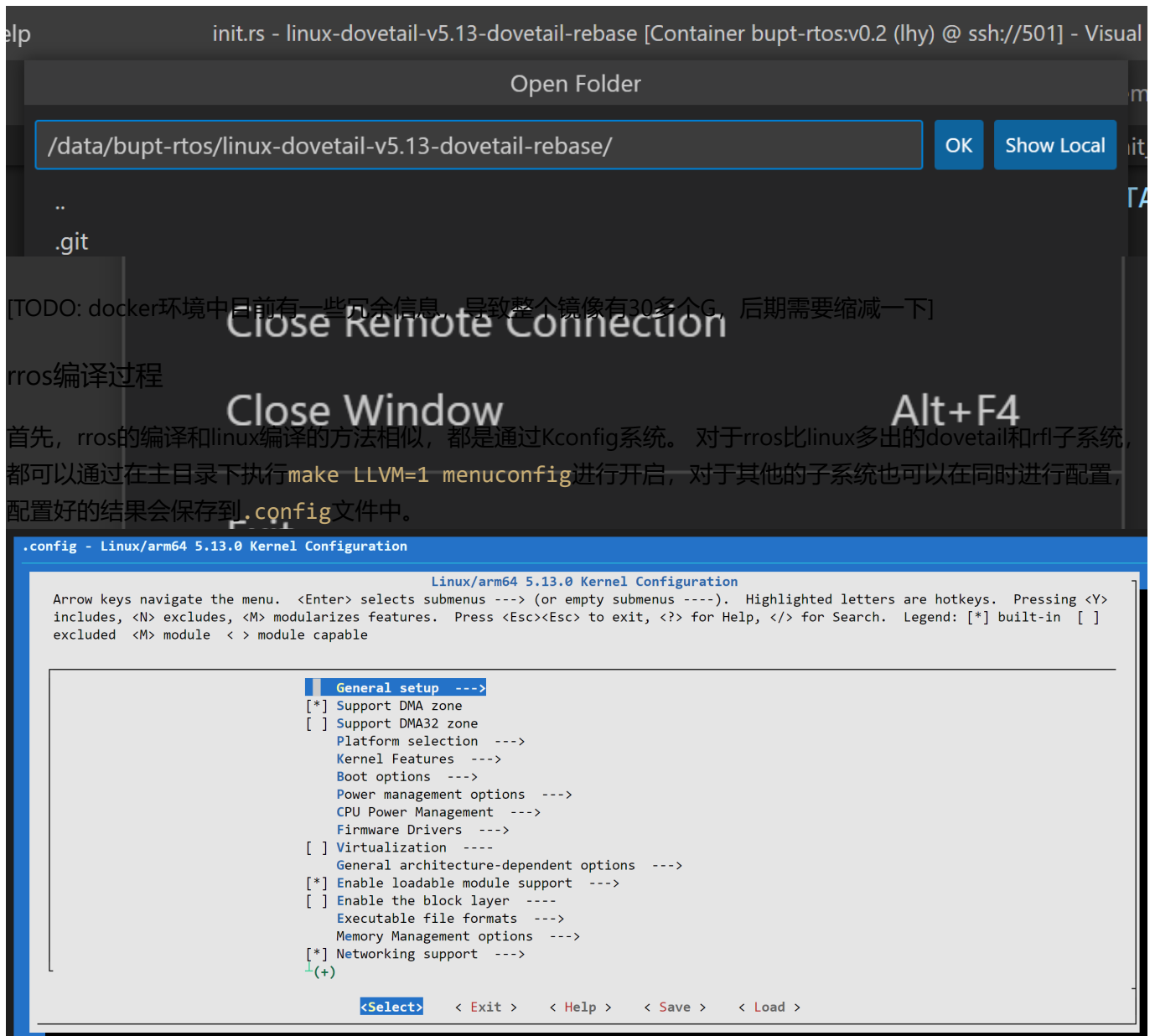


在新窗口中打开linux环境后，我们就可以看到我们运行起来的docker了，然后点击进入，打开我们项目的文件夹



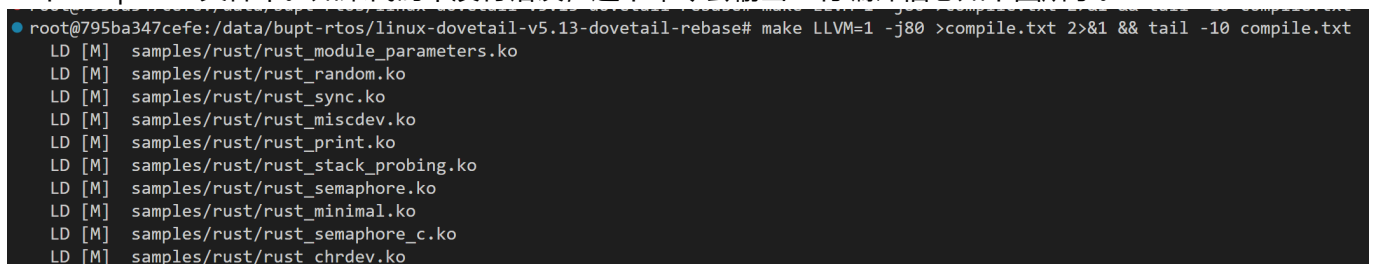






为了方便大家做后面的实验，我们提供一个已经配置好的`.config`文件，不需要大家手动配置。并且这个`.config`中的选项经过了剪裁，所以操作系统编译的速度会大大加快，把这个文件放到rrros的主目录下面就可以了。如果大家想要体会手动配置config的过程，可以参考下面的编译tips中第四点。

然后，可以用`make LLVM=1 -j80 >compile.txt 2>&1 && tail -10 compile.txt`对整个操作系统进行编译，这个命令后面的重定向是由于目前项目中有大量的warning没有被消除，所以我们最好把编译的结果保存到一个`compile.txt`文件中。如果代码中没有错误，这个命令会输出10行编译信息如下图所示。



如果代码中有错误，可以用`finderr.py`脚本对错误进行过滤，执行命令是`python finderr.py compile.txt`,

最后利用`cat result`查看`result`文件中的错误信息。

```
root@795ba347cefe:/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase# make LLVM=1 -j80 >compile.txt 2>&1 && tail -10 compile.txt
root@795ba347cefe:/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase# python finderr.py compile.txt
root@795ba347cefe:/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase# cat result
error[E0425]: cannot find value `res` in this scope
--> kernel/rros/init.rs:139:11
139 |         match res {
    |         ^^^ not found in this scope

error: aborting due to previous error; 197 warnings emitted
```

最后，如果编译成功了，我们就可以在主目录下看到最新生成的`vmlinux`文件，然后用`qemu`去模拟运行这个操作系统，使用`qemu`运行操作系统的部分，会在[使用Qemu进行模拟](#)小节进行讲解。

编译tips:

1. 可以注意到`make`命令中使用了`LLVM=1`，这个参数会让编译过程中使用`llvm`而不是`gcc`，这是因为我们需要`rfl`项目的支持，而`rfl`需要通过`llvm`才能成功编译`rust`。所以我们在`rros`的大部分编译命令中都需要加入`LLVM=1`。
2. `rros`目标的平台是在`arm64`，所以涉及到交叉编译的知识，交叉编译就是编译代码的环境和执行代码的环境不在一个平台上，比如在`x86_64`平台下，编译`arm64`的目标文件，我们在编译时通过使用两个环境变量来说明这两个信息，`rros`编译时会自动读取这两个环境变量来获得这部分信息。

```
root@795ba347cefe:/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase# echo $ARCH
arm64
root@795ba347cefe:/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase# echo $CROSS_COMPILE
aarch64-linux-gnu-
```

3. 一些`docker`环境中隐藏的细节：`docker`环境中配置了可以支持交叉编译环境的`gcc`，`llvm`，`qemu`，`gdb`，`objdump`等编译相关的工具，以及`cmake`，`rust`等开发相关的工具，有些软件是从源码编译安装的，因为`ubuntu`/`centos`的`apt-get`和`yum`包管理工具会限制这些软件的版本。
4. `config`如何手动配置：`config`中需要手动配置的主要是分为三部分：如何开启`dovetail`和`rfl`，如何开启`debug`相关的选项，如何裁剪和`rros`内核无关的`config`，下面介绍前两部分。

- Kernel Features中开启Dovetail interface

```
General setup ---> file(s)
Platform selection --->
Kernel Features --->
Boot options --->
Power management options --->
CPU Power Management --->
Firmware Drivers --->

[*] NUMA Memory Allocation and Scheduler Support
(4) Maximum NUMA Nodes (as a power of 2)
Timer frequency (250 HZ) --->
[*] Dovetail interface
[ ] Enable paravirtualization code (NEW)
[ ] Paravirtual steal time accounting (NEW)
```

- General setup中开启Rust support

```
.config - Linux/arm64 5.13.0 Kernel Configuration
Linux/arm64 5.13.0 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?
<M> module < > module capable

General setup --->
Platform selection --->
Kernel Features --->
Boot options --->
Power management options --->
CPU Power Management --->

[ ] Harden stack tree with STX
[ ] Page allocator randomization
[*] SLUB per cpu partial caching
[*] Profiling support
[*] Rust support
```

- debug相关的config

```
.config - Linux/arm64 5.13.0 Kernel Configuration
> Kernel hacking

Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ---). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
excluded <M> module < > module capable

printk and dmesg options --->
Compile-time checks and compiler options --->
Generic Kernel Debugging Instruments --->
*- Kernel debugging
[ ] Miscellaneous debug code
Memory Debugging --->
[ ] Debug shared IRQ handlers
[*] Debug IRQ pipeline
[ ] Torture tests for IRQ pipeline
[*] Debug Dovetail interface
Debug Oops, Lockups and Hangs --->
Scheduler Debugging --->
[ ] Enable extra timekeeping sanity checking
[ ] Debug preemptible kernel
Lock Debugging (spinlocks, mutexes, etc...) --->
[ ] Debug IRQ flag manipulation
[ ] Stack backtrace support
[ ] Warn for all uses of unseeded randomness
[ ] kobject debugging
+(<+>)

<Select> < Exit > < Help > < Save > < Load >
```

```
.config - Linux/arm64 5.13.0 Kernel Configuration
> Kernel hacking

Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ---). Highlighted letters are hotkeys. Pressing <Y>
includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
excluded <M> module < > module capable

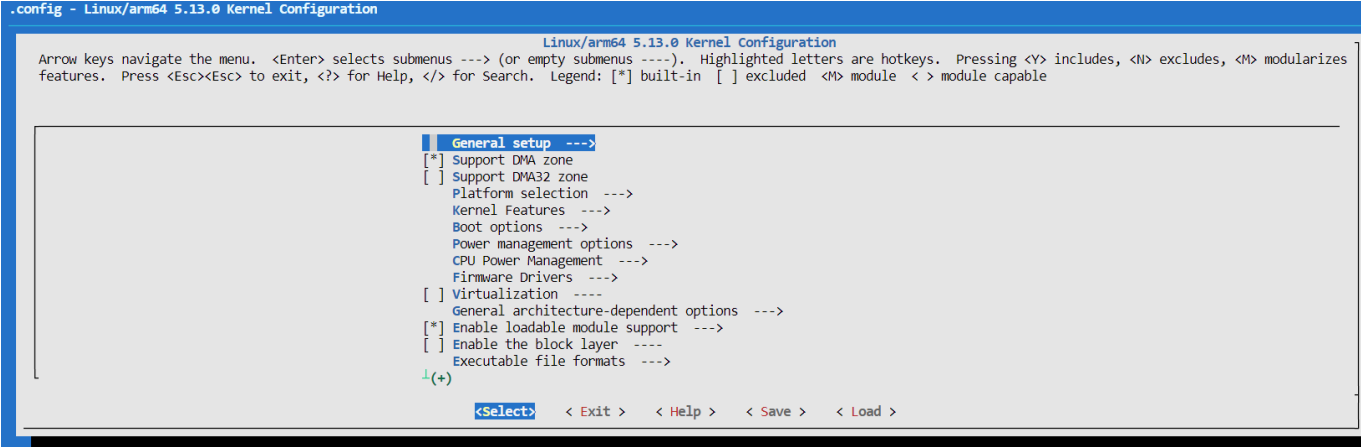
printk and dmesg options --->
Compile-time checks and compiler options --->
Generic Kernel Debugging Instruments --->
*- Kernel debugging
[ ] Miscellaneous debug code
Memory Debugging --->
[ ] Debug shared IRQ handlers
*- Debug IRQ pipeline
[ ] Torture tests for IRQ pipeline
[*] Debug Dovetail interface
Debug Oops, Lockups and Hangs --->
Scheduler Debugging --->
[ ] Enable extra timekeeping sanity checking
[ ] Debug preemptible kernel
Lock Debugging (spinlocks, mutexes, etc...) --->
[ ] Debug IRQ flag manipulation
[ ] Stack backtrace support
[ ] Warn for all uses of unseeded randomness
[ ] kobject debugging
+(<+>)

<Select> < Exit > < Help > < Save > < Load >
```

编译等级

调试时可能需要将编译等级调低。我们所给的配置文件.config应该已经配置，若没有，你可以手动配置一下。

输入menuconfig

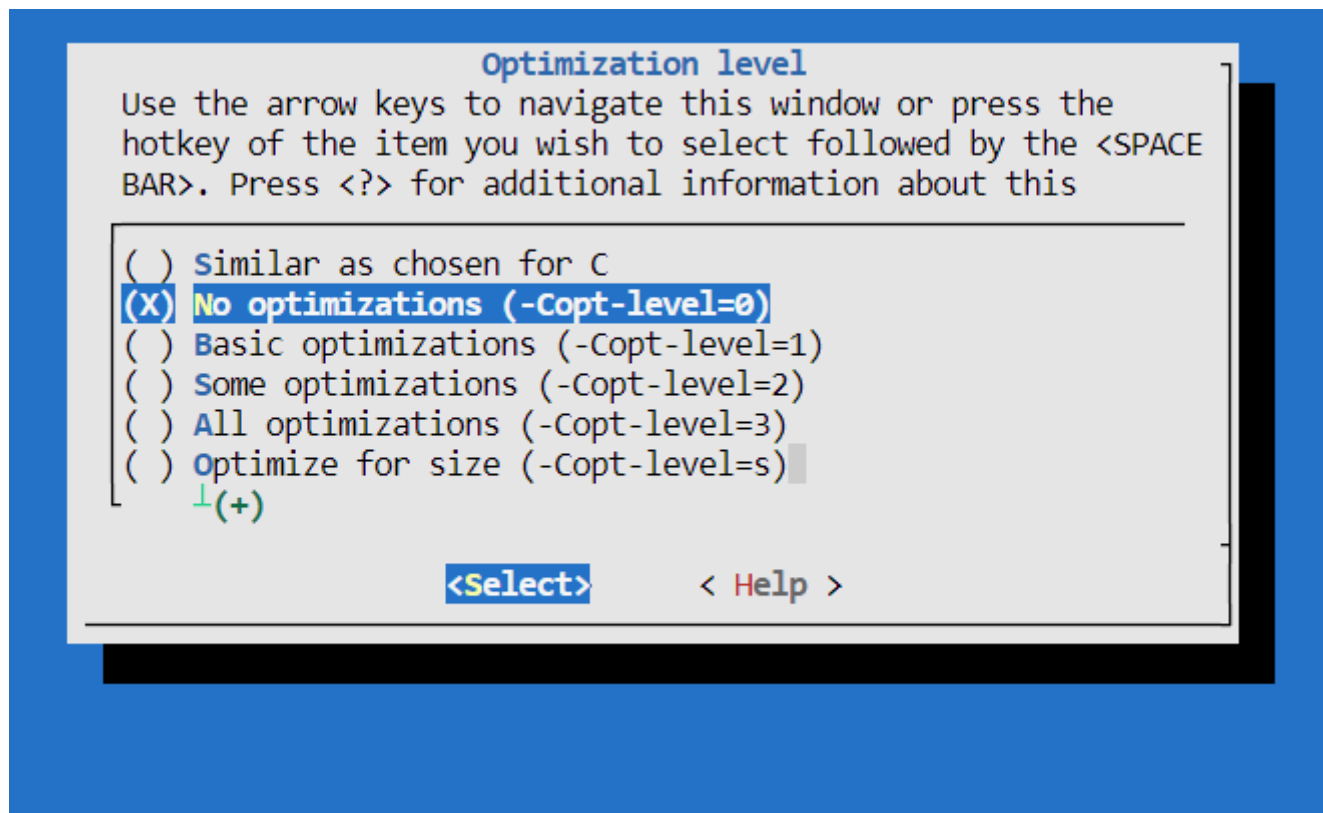


如果出现问题，可能有两种情况：

- 把窗口拉大一点
- 看bash环境变量是否有\$CROSS_COMPILE，\$ARCH。这些在之前应该已经配置过。

```
# 在~/.bashrc里添加
export CROSS_COMPILE=aarch64-linux-gnu-
export ARCH=arm64
```

把kernel hacking > Rust Hacking > Optimization level > debug-level 调到最低



按空格确定向右选择Exit，按回车退出，选择保存Yes.

使用rust-gdb调试

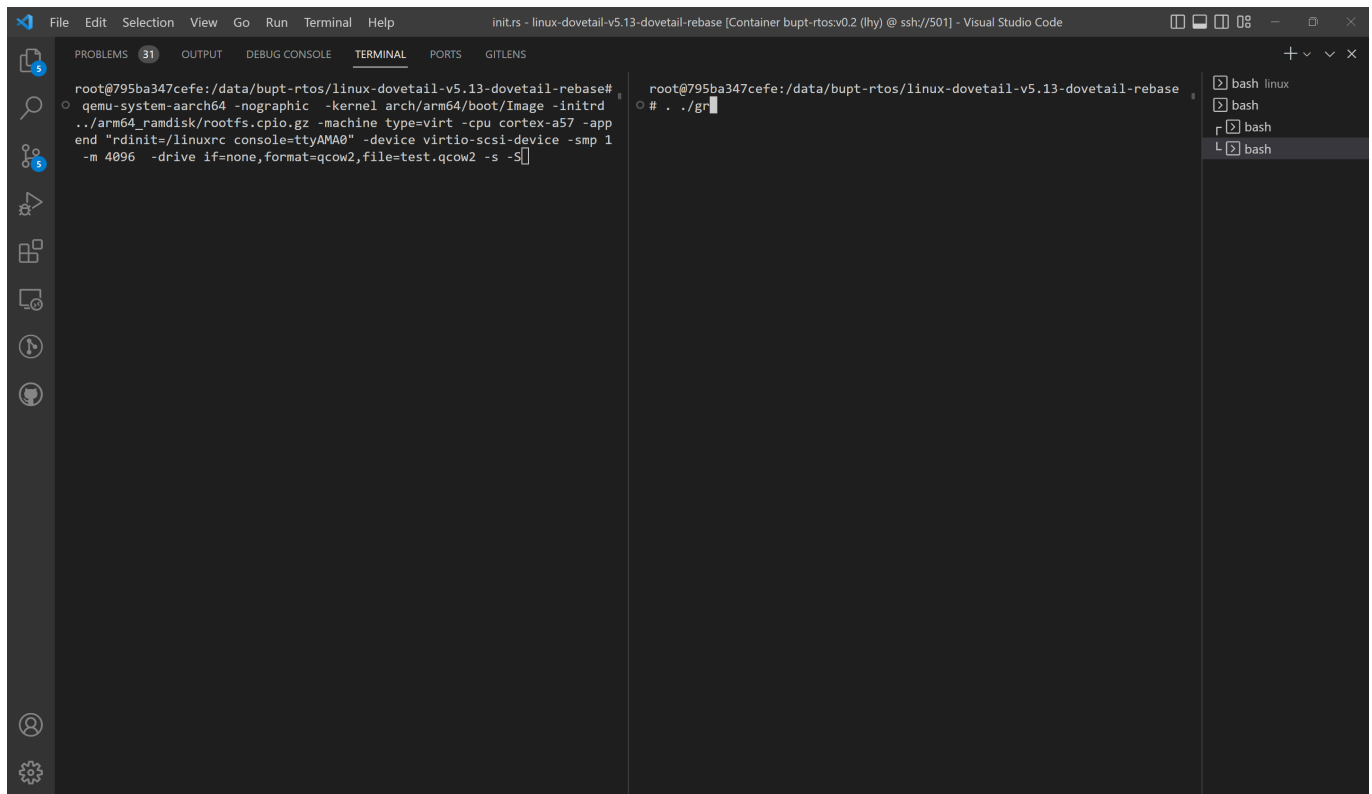
在docker中我们已经安装好qemu了，所以可以直接使用。并且qemu启动所需要的文件系统已经在docker中准备完成。

我们只需要同时打开两个命令行窗口，然后左边运行qemu的命令，右边运行gdb的命令，我们就可以完成对rros的debug工作。

```
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd
../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append
"rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096 -
drive if=none,format=qcow2,file=test.qcow2 -s -S
```

最后的两个标志 -s 表示启动gdb server， -S表示不要立刻执行指令，按c可以开始执行。

```
. ./gr
```

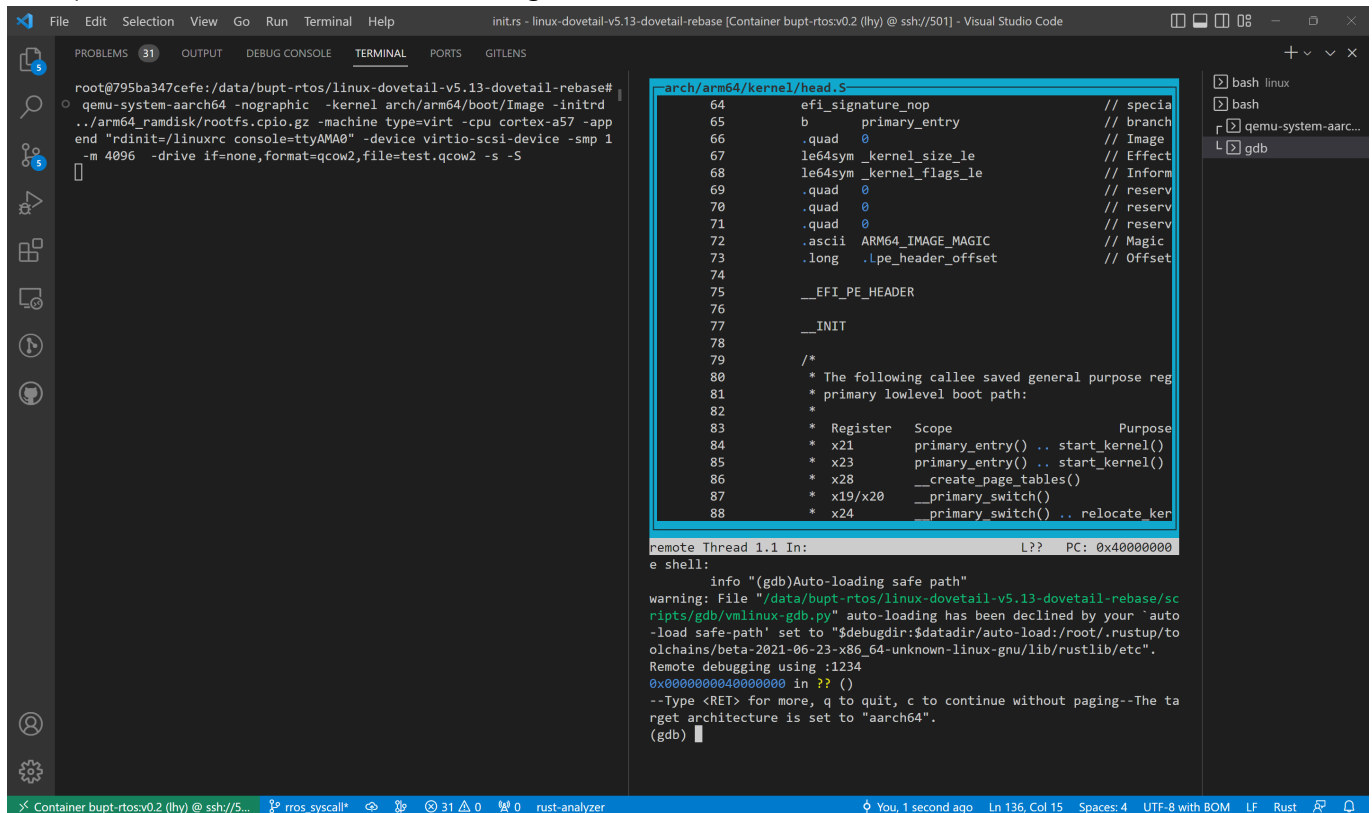
如果没有`gr`在路径下面，也可以手动启动一个rust-gdb进程：

```
rust-gdb \  
--tui vmlinux \  
-ex "target remote localhost:1234" \  
-ex "set architecture aarch64" \  
-ex "set auto-load safe-path" \  
-ex "set lang rust"
```

如果不想debug，只想用qemu对操作系统进行模拟运行，那么只需要打开一个窗口，然后去掉-s -S这两个qdb相关的参数，运行下列命令即可

```
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd
../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append
"rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096
-drive if=none,format=qcow2,file=test.qcow2
```

先在qemu所在窗口执行上述命令，然后在gdb窗口执行上述命令，就可以成功运行



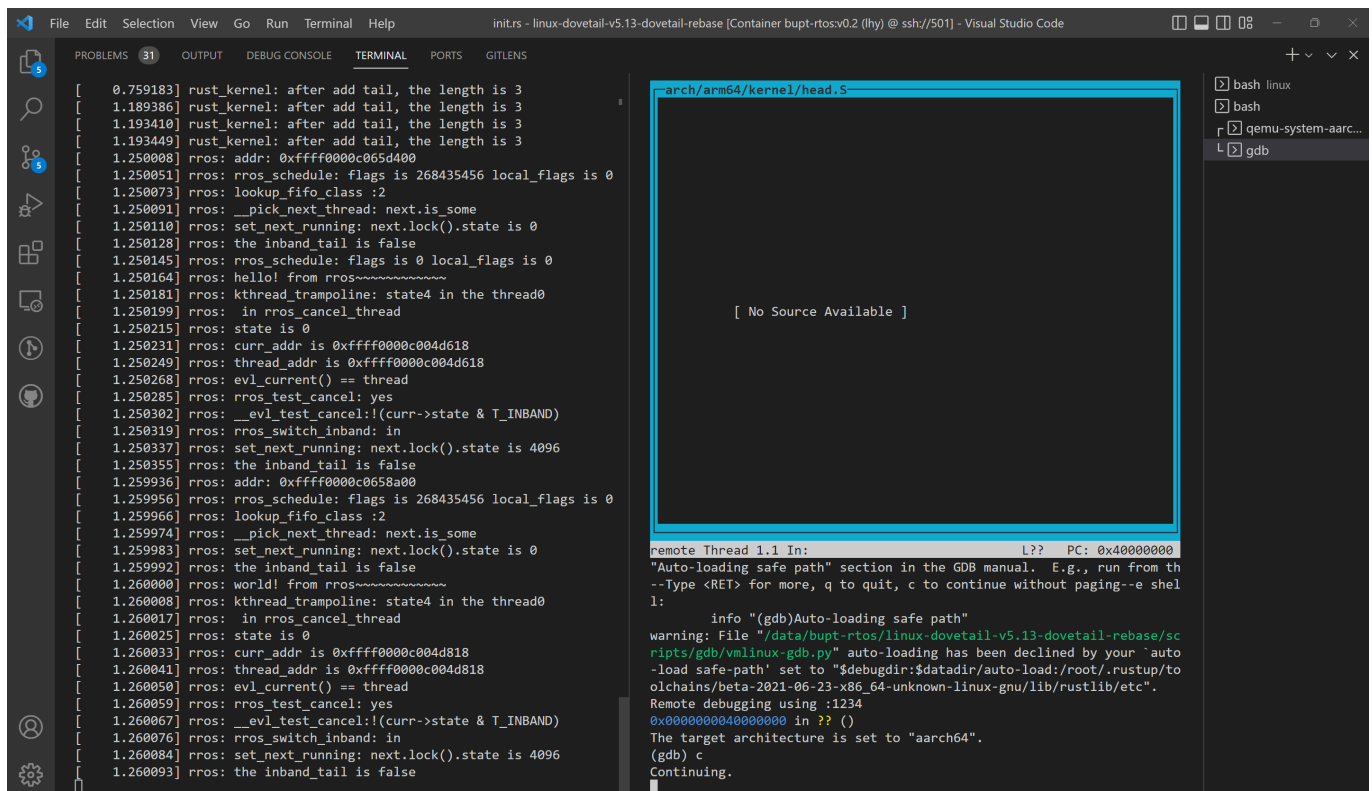
The screenshot shows a Visual Studio Code interface with two main panes. The left pane is a terminal window with the following content:

```
root@795ba347cfe:/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase#  
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd  
../arm64_ranisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -app  
end "rdinit/linuxrc console=tttyAMA0" -device virtio-ccsi-device -smp 1  
-m 4096 -drive if=none,format=qcow2,file=test.qcow2 -s -S
```

The right pane is a GDB window showing the kernel source code for `arch/arm64/kernel/head.S`. The code includes comments and assembly instructions for the primary entry point. The GDB console at the bottom shows the following output:

```
remote Thread 1.1 In: L?? PC: 0x40000000  
e shell:  
info "(gdb)Auto-loading safe path"  
warning: File "/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase/sc  
ripts/gdb/vmlinux-gdb.py" auto-loading has been declined by your `auto  
-load safe-path' set to "$debugdir:$datadir/auto-load:/root/.rustup/to  
olchains/beta-2021-06-23-x86_64-unknown-linux-gnu/lib/rustlib/etc".  
Remote debugging using :1234  
0x0000000040000000 in ?? ()  
--Type <RET> for more, q to quit, c to continue without paging--The ta  
rget architecture is set to "aarch64".  
(gdb)
```


在gdb窗口中按c并回车，可以看到rros操作系统就可以正常执行了。



The screenshot shows a Visual Studio Code interface with two main panes. The left pane is a terminal window showing the output of the rros kernel. The output includes various log messages and system initialization steps. The right pane is a GDB window showing the kernel source code for `arch/arm64/kernel/head.S`. The GDB console at the bottom shows the following output:

```
remote Thread 1.1 In: L?? PC: 0x40000000  
"Auto-loading safe path" section in the GDB manual. E.g., run from th  
--Type <RET> for more, q to quit, c to continue without paging--e shel  
l:  
info "(gdb)Auto-loading safe path"  
warning: File "/data/bupt-rtos/linux-dovetail-v5.13-dovetail-rebase/sc  
ripts/gdb/vmlinux-gdb.py" auto-loading has been declined by your `auto  
-load safe-path' set to "$debugdir:$datadir/auto-load:/root/.rustup/to  
olchains/beta-2021-06-23-x86_64-unknown-linux-gnu/lib/rustlib/etc".  
Remote debugging using :1234  
0x0000000040000000 in ?? ()  
The target architecture is set to "aarch64".  
(gdb) c  
Continuing.
```

gdb的具体指令和上学期bos lab中的相关指令一致，具体内容可以回看上学期ppt。

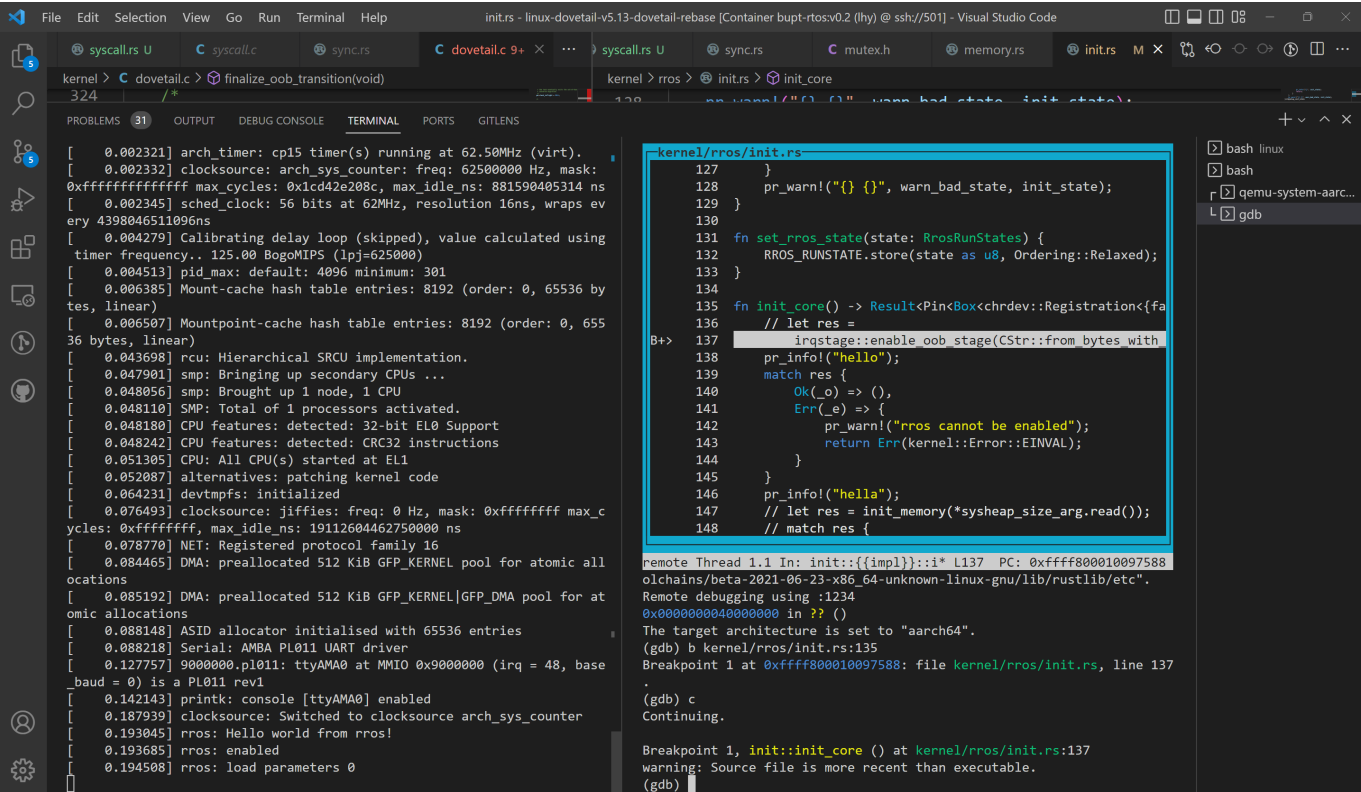


- 基本用法: `b/r/c/n/s/i/help`
- 命令行下的图形化: `ctrl+x a` `ctrl+x o`
- 同时查看多种信息: `layout`
- 观察内存信息: `x/Nx ptr`

下面给出了gdb的一些常见命令

命令	作用
c	跳到下一个断点
b 文件名:行号	设置断点
p 变量名	打印变量
x/	打印地址下的数据
finish	跳到当前函数的结尾
frame	查看栈帧
n	next, 下一步, 不进入函数
s	step in, 下一步, 但是可能进入函数

举一个例子, 如果想要在操作系统执行之前打一个断点, 可以用**b kernel/rros/init.rs:135**在 kernel/rros/init.rs这个文件的135行打一个断点, 然后再按c就能执行到这一行了。



执行结束时，退出qemu时，先按ctrl+a+x；退出gdb时，按ctrl+d。

使用vscode的调试

也可以给vscode添加配置文件。

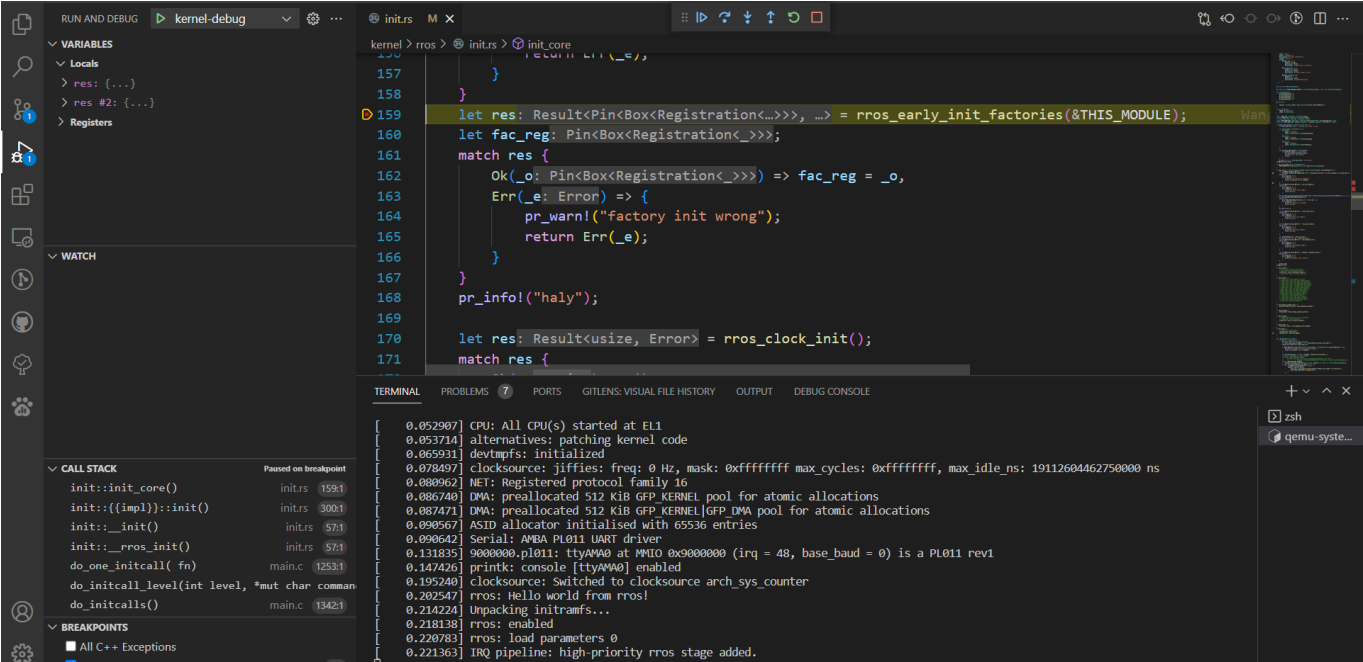
首先，同样还是在命令行启动调试的qemu：

```
qemu-system-aarch64 -nographic -kernel arch/arm64/boot/Image -initrd
../arm64_ramdisk/rootfs.cpio.gz -machine type=virt -cpu cortex-a57 -append
"rdinit=/linuxrc console=ttyAMA0" -device virtio-scsi-device -smp 1 -m 4096 -
drive if=none,format=qcow2,file=test.qcow2 -s -S
```

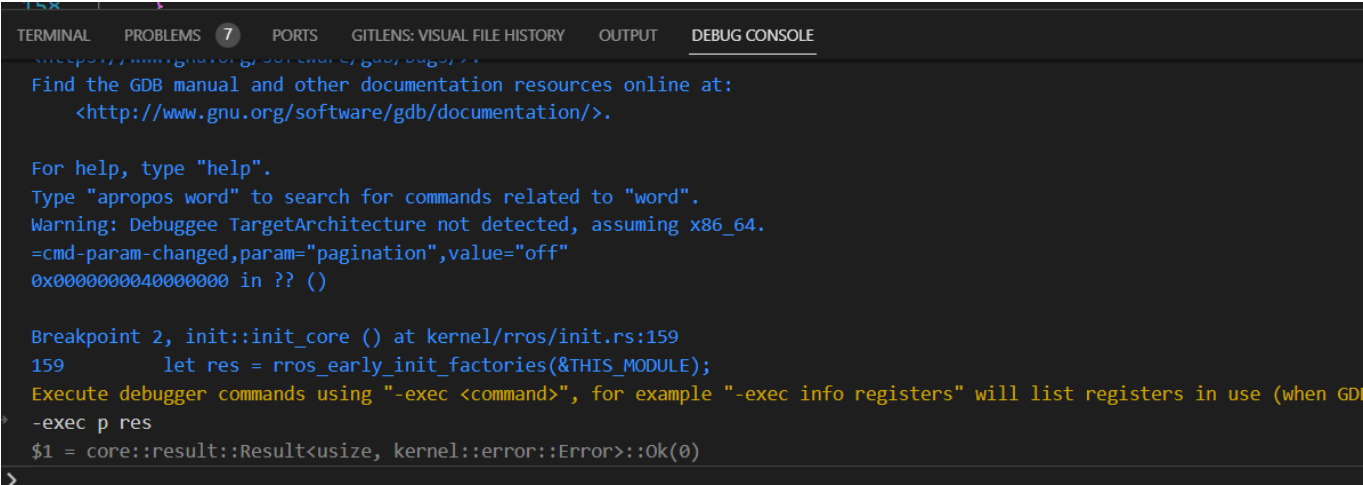
然后，在项目根目录的.vscode文件夹中，打开.vscode/launch.json(没有的话新建一个)，把下面的配置粘贴进去：

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "kernel-debug",
      "type": "cppdbg",
      "request": "launch",
      "miDebuggerServerAddress": "127.0.0.1:1234",
      "program": "${workspaceFolder}/vmlinux",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "logging": {
        "engineLogging": false
      },
      "MIMode": "gdb",
      "miDebuggerPath": "/root/.cargo/bin/rust-gdb",
      // "miDebuggerPath": "/usr/bin/gdb-multiarch",
      "setupCommands": [
        {
          "description": "set language rust",
          "text": "set lang rust",
          "ignoreFailures": true
        }
      ]
    }
  ]
}
```

在行号处点击断点，按F5开始调试



如果需要使用gdb命令，可以在下面DEBUG CONSOLE，输入-exec {gdb命令}执行



引用

- [1] A linux-based real-time operating system
- [2] Adaptive domain environment for operating systems
- [3] Life with adoes
- [4] Study and Comparison of the RTHAL-based and ADEOS-based RTAI Real-time Solutions for Linux
- [5] <https://github.com/Rust-for-Linux/linux>