

**QEMU**（Quick Emulator）是一款免费开源模拟器，由[法布里斯·贝拉](https://zh.wikipedia.org/wiki/%E6%B3%95%E5%B8%83%E9%87%8C%E6%96%AF%C2%B7%E8%B4%9D%E6%8B%89" \o "法布里斯·贝拉)（Fabrice Bellard）等人编写。其与[Bochs](https://zh.wikipedia.org/wiki/Bochs" \o "Bochs)，[PearPC](https://zh.wikipedia.org/wiki/PearPC" \o "PearPC)类似，但拥有高速（配合[KVM](https://zh.wikipedia.org/wiki/%E5%9F%BA%E4%BA%8E%E5%86%85%E6%A0%B8%E7%9A%84%E8%99%9A%E6%8B%9F%E6%9C%BA" \o "基于内核的虚拟机)）、跨平台的特性。

QEMU是一个托管的虚拟机，它使用动态二进制转换技术来模拟处理器，并且提供多种硬件和外设模型，这使它能够运行多种未修改的客户机操作系统，能与[KVM](https://zh.wikipedia.org/wiki/%E5%9F%BA%E4%BA%8E%E5%86%85%E6%A0%B8%E7%9A%84%E8%99%9A%E6%8B%9F%E6%9C%BA" \o "基于内核的虚拟机)配合以接近本地速度运行虚拟机（接近真实电脑的速度）。

QEMU可以执行用户级的进程仿真，从而可以使为某一架构编译的程序在另一架构上运行（通过 VMM 的形式实现）。

**基本原理**

QEMU作为系统模拟器时，会模拟出一台能够独立运行操作系统的虚拟机。如下图所示，每个虚拟机对应主机(Host)中的一个QEMU进程，而虚拟机的vCPU对应QEMU进程的一个线程。

系统虚拟化最主要是虚拟出CPU、内存及I/O设备。虚拟出的CPU称之为vCPU，QEMU为了提升效率，借用KVM、XEN等虚拟化技术，直接利用硬件对虚拟化的支持，在主机上安全地运行虚拟机代码(需要硬件支持)。虚拟机vCPU调用KVM的接口来执行任务的流程如下(代码源自QEMU开发者Stefan的技术博客)：

open("/dev/kvm")

ioctl(KVM\_CREATE\_VM)

ioctl(KVM\_CREATE\_VCPU)

for (;;) {

ioctl(KVM\_RUN)

switch (exit\_reason) {

case KVM\_EXIT\_IO: /\* ... \*/

case KVM\_EXIT\_HLT: /\* ... \*/

}

}

QEMU发起ioctrl来调用KVM接口，KVM则利用硬件扩展直接将虚拟机代码运行于主机之上，一旦vCPU需要操作设备寄存器，vCPU将会停止并退回到QEMU，QEMU去模拟出操作结果。

虚拟机内存会被映射到QEMU的进程地址空间，在启动时分配。在虚拟机看来，QEMU所分配的主机上的虚拟地址空间为虚拟机的物理地址空间。

QEMU在主机用户态模拟虚拟机的硬件设备，vCPU对硬件的操作结果会在用户态进行模拟，如虚拟机需要将数据写入硬盘，实际结果是将数据写入到了主机中的一个镜像文件中。

**三、创建及使用虚拟机**

使用qemu-img创建虚拟机镜像，虚拟机镜像用来模拟虚拟机的硬盘，在启动虚拟机之前需要创建镜像文件。

qemu-img create -f qcow2 test-vm-1.qcow2 10G

-f 选项用于指定镜像的格式，qcow2 格式是 Qemu 最常用的镜像格式，采用来写时复制技术来优化性能。test-vm-1.qcow2 是镜像文件的名字，10G是镜像文件大小。镜像文件创建完成后，可使用 qemu-system-x86 来启动x86 架构的虚拟机：

使用 qemu-system-x86 来启动 x86 架构的虚拟机

qemu-system-x86\_64 test-vm-1.qcow2

因为 test-vm-1.qcow2 中并未给虚拟机安装操作系统，所以会提示 “No bootable device”，无可启动设备。

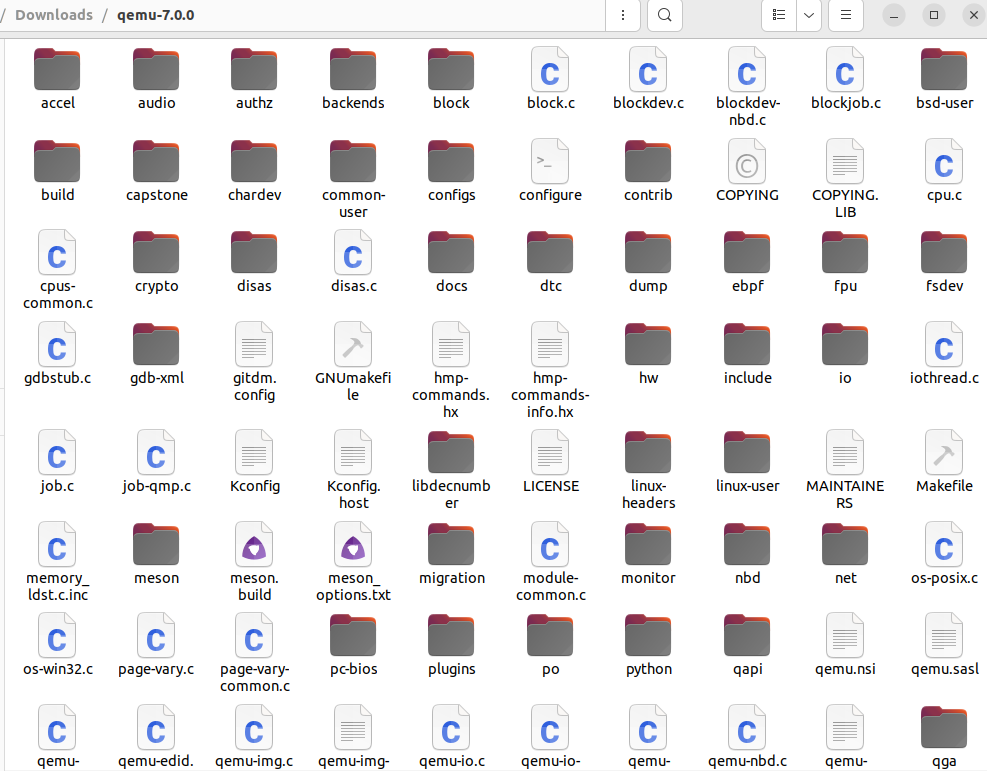
启动 VM 安装操作系统镜像

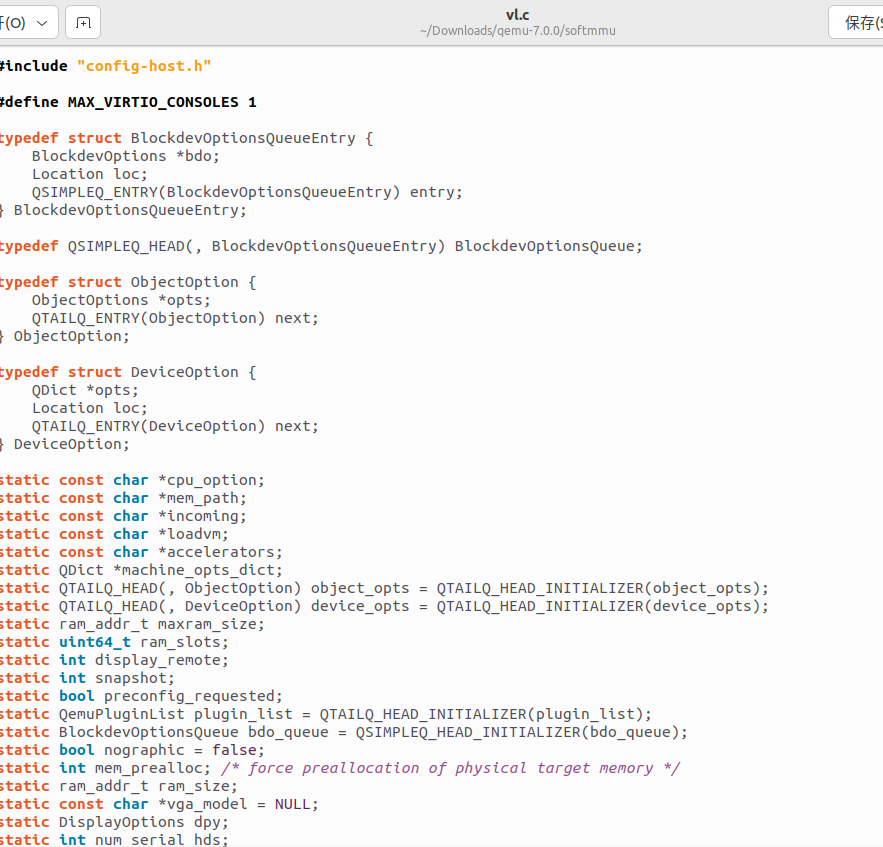
qemu-system-x86\_64 -m 2048 -enable-kvm test-vm-1.qcow2 -cdrom ./Centos-Desktop-x86\_64-20-1.iso

-m 指定虚拟机内存大小，默认单位是 MB， -enable-kvm 使用 KVM 进行加速，-cdrom 添加 fedora 的安装镜像。可在弹出的窗口中操作虚拟机，安装操作系统，安装完成后重起虚拟机便会从硬盘 ( test-vm-1.qcow2 ) 启动。之后再启动虚拟机只需要执行：

qemu-system-x86\_64 -m 2048 -enable-kvm test-vm-1.qcow2

qemu-img 支持非常多种的文件格式，可以通过 qemu-img -h 查看，其中 raw 和 qcow2 是比较常用的两种，raw 是 qemu-img 命令默认的，qcow2 是 qemu 目前推荐的镜像格式，是功能最多的格式。





**(1)开始执行**

主要比较重要的c文件有：/vl.c,/cpus.c, /exec-all.c, /exec.c, /cpu-exec.c。

QEMU的main函数定义在/vl.c中，它也是执行的起点，这个函数的功能主要是建立一个虚拟的硬件环境。它通过参数的解析，将初始化内存，需要的模拟的设备初始化，CPU参数，初始化KVM等等。接着程序就跳转到其他的执行分支文件如：/cpus.c, /exec-all.c, /exec.c, /cpu-exec.c。

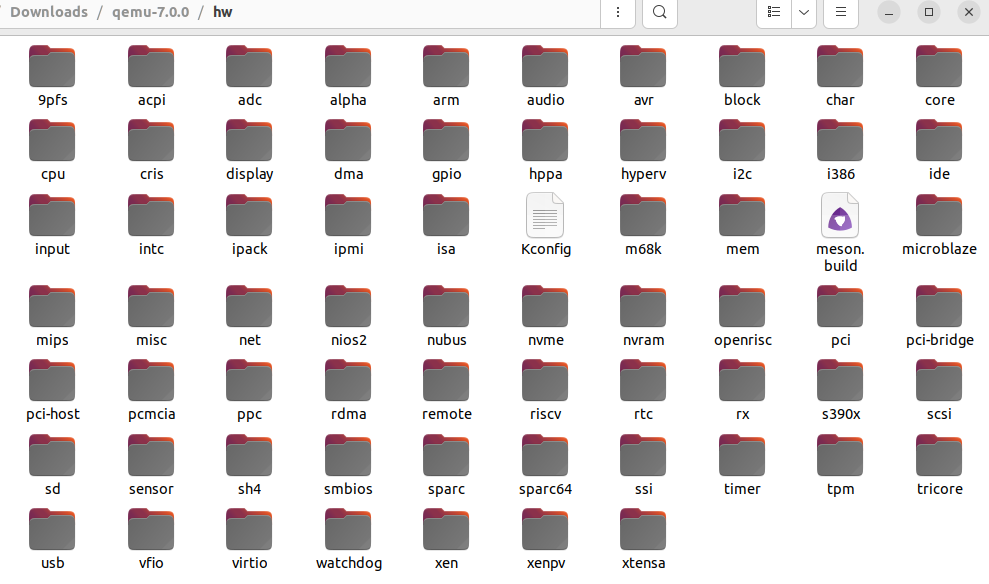
**(2)硬件模拟**

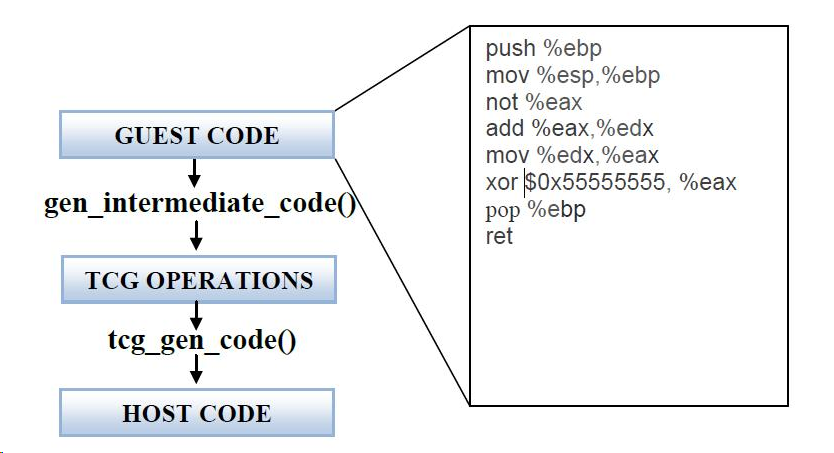
所有的硬件设备都在/hw/ 目录下面，所有的设备都有独自的文件，包括总线，串口，网卡，鼠标等等。它们通过设备模块串在一起，在vl.c中的machine \_init中初始化。这里就不讲每种设备是怎么实现的了。

**(3)目标机器**

现在QEMU模拟的CPU架构有：Alpha, ARM, Cris, i386, M68K, PPC, Sparc, Mips, MicroBlaze, S390X and SH4。

我们在QEMU中使用./configure 可以配置运行的架构，这个脚本会自动读取本机真实机器的CPU架构，并且编译的时候就编译对应架构的代码。对于不同的QEMU做的事情都不同，所以不同架 构下的代码在不同的目录下面。/target-arch/目录就对应了相应架构的代码，如/target-i386/就对应了x86系列的代码部分。虽然 不同架构做法不同，但是都是为了实现将对应客户机CPU架构的TBs转化成TCG的中间代码。这个就是TCG的前半部分





 **初始化阶段**

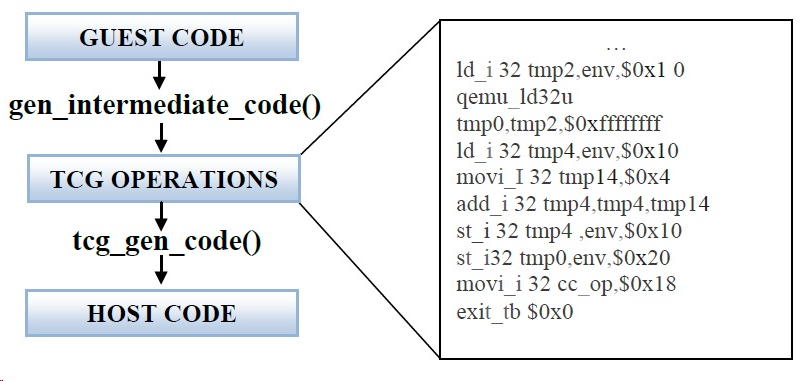
* 调用 vl\_init()，该函数完成内部数据结构的分配、默认参数设定以及各个虚拟设备的注册。
* 设置内存映射和 I/O 队列，为后续的虚拟机运行做好准备。

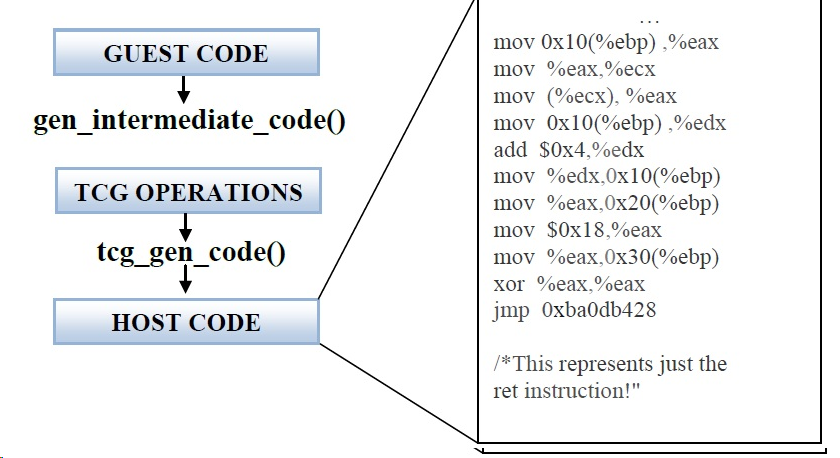
 **虚拟机运行期间**

* 当来宾执行某条 I/O 指令时，会触发 VM exit，此时 VMM（或 VMM 中负责虚拟层的模块）调用 ioctl(KVM\_RUN) 后，进入“vl.c”中预先注册的 I/O 处理函数。
* 该函数根据 VM exit 中返回的端口号和数据大小，识别出目标设备，调用相应的读写处理逻辑。
* 例如，若设备为串口输出，则将数据复制到宿主标准输出；若是虚拟设备（如网络或存储设备），则可能触发更复杂的操作，如数据缓冲、请求转发等。

 **事件循环与退出处理**

* “vl.c”中的核心模块一般会放在一个无限循环中，不断调用 KVM\_RUN 让虚拟 CPU 执行来宾代码，处理各种退出原因。
* 当遇到 KVM\_EXIT\_HLT（halt 指令）时，虚拟机运行结束，模块会调用相应的清理函数，释放资源并退出。





基于内核的虚拟机（KVM）

QEMU 源码中的 hw 文件夹主要用于实现硬件设备的模拟。它包含了各种设备模型的代码，使得 QEMU 能够向来宾操作系统呈现与真实硬件相似的接口和行为。下面详细介绍其主要内容和结构：

1. 目录结构和作用

硬件设备模拟层

hw 文件夹里的代码负责模拟多种硬件设备，涵盖从传统 PC 硬件到嵌入式系统的各类设备。这些设备模型帮助 QEMU 构建完整的虚拟化环境，使得来宾操作系统可以直接访问和控制虚拟硬件。

常见子目录

hw/isa：实现了基于 ISA 总线的设备，如老式的 ISA 声卡、网卡、键盘控制器等。

hw/pc：针对 PC 平台的设备模拟，例如 PCI 设备、南桥芯片、ACPI 支持等。

hw/usb：USB 控制器和 USB 设备模型的实现，支持不同版本的 USB 协议。

hw/net：网络设备模拟，如网卡（例如 e1000、rtl8139、virtio-net 等）的实现。

hw/block：存储设备相关的代码，包括 IDE、SCSI、Virtio 块设备等的模拟。

hw/display：图形输出设备模拟，如 VGA、QXL、Cirrus Logic 显卡等。

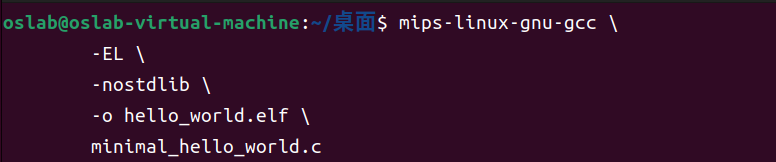
hw/intc：中断控制器模拟（如 PIC、APIC、IOAPIC）的实现，负责处理中断信号。

hw/arm、hw/mips、hw/riscv 等目录：针对不同架构特定的设备模型和平台板级支持。



需要注意，我们在源代码中没有定义 main 函数。\_\_start 函数才是程序的入口。

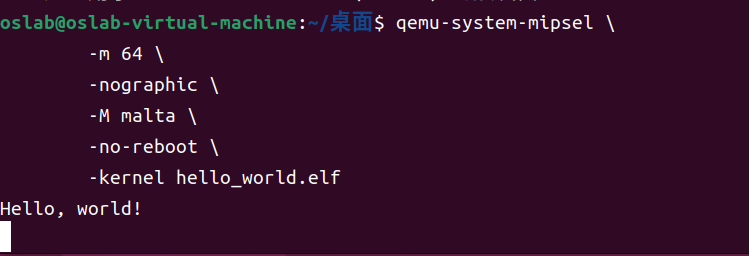
之后我们就可以编译该文件了。只需要使用交叉编译器 mips-linux-gnu-gcc 执行如下命令



编译产生的目标程序为 hello\_world.elf。接下来我们使用 QEMU 运行该文件。

在这里要简单介绍一下 QEMU 的命令。所有的 QEMU 命令都为 qemu-\* 的形式。对于某一体系架构下的模拟，需要使用 qemu-system-\* 命令。如对于小端序的 mips 架构，对应的命令为 qemu-system-mipsel。此外 QEMU 还提供了其他的一些命令行工具，如 qemu-img 就用于创建、转换和修改磁盘镜像。

为了运行我们的目标文件，需要使用 qemu-system-mipsel。运行下面的命令，我们就可以得到 Hello, world! 的输出。



介绍一下上面命令所用到的选项：

* -m：用于指定虚拟机内存的大小。
* -nographic：表示模拟中不使用图形界面，而是使用串口输出
* -M：用于指定要模拟的目标机器，这里模拟的是 MIPS Malta 开发板
* -no-reboot：虚拟机会直接退出而不是重启
* -kernel：指定要启动的内核，这里就是我们的 hello\_world.elf

这段代码是一个典型的裸机（bare-metal）“Hello, world!”程序，它直接操作硬件而不依赖任何操作系统。下面逐行解释其功能：

printch 函数

c

复制

编辑

void printch(char ch) { \*((volatile char \*)(0xB80003f8U)) = ch; }

目的：将一个字符写入特定的内存地址，从而在屏幕上显示该字符。

内存地址 0xB80003f8：通常在x86架构下，VGA文本模式的显存起始地址是0xB8000，但这里的地址有偏移（0x3f8），可能是为了使用特定的显示区域或硬件映射。

volatile 关键字：告诉编译器该内存地址可能会在程序外部改变，防止优化删除或重排对该地址的写操作，确保每次调用时都执行写操作。

print 函数

c

复制

编辑

void print(char \*str) {

while (\*str != '\0') {

printch(\*str);

str++;

}

}

目的：逐字符输出一个以空字符（'\0'）结尾的字符串。

实现细节：循环遍历字符串，每次调用 printch 输出当前字符，直到遇到字符串结束符。

\_\_start 函数

c

复制

编辑

void \_\_start() {

print("Hello, world!\n");

while (1) {

}

}

目的：程序的入口点，类似于 main 函数，但在裸机编程中通常直接定义一个入口符号。

实现细节：

调用 print 函数输出 "Hello, world!\n"。

进入无限循环（while (1)），防止程序结束后跳转到未定义区域或进行其他操作，因为裸机环境没有操作系统来回收程序资源。

总结：

这段代码直接操作特定内存地址，将字符串 "Hello, world!" 输出到显示设备上，是裸机编程的入门示例。程序没有使用操作系统支持，因此所有的输出和控制逻辑都直接通过内存操作和无限循环来实现。