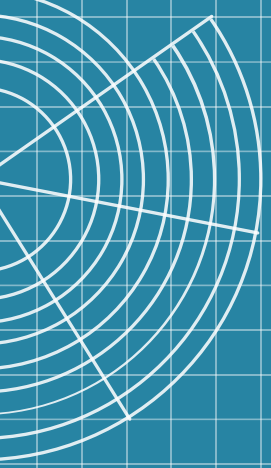
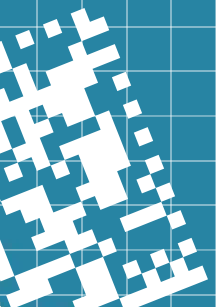
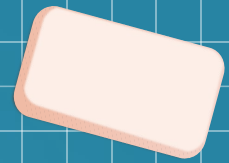
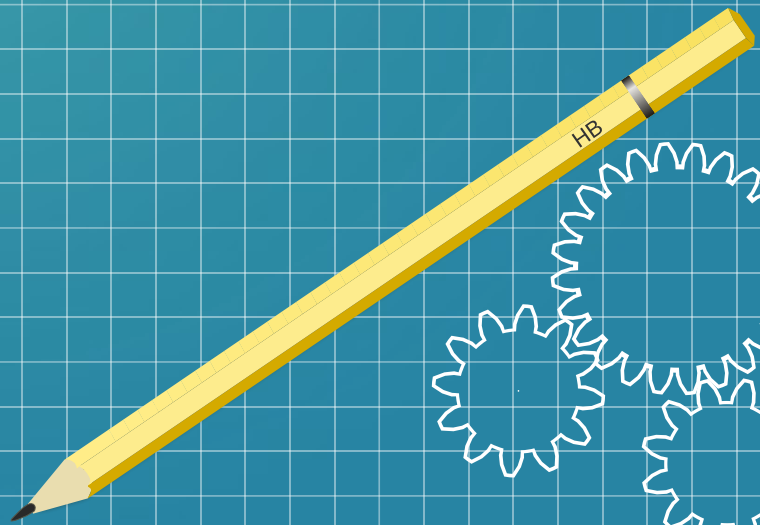
QEMU：宏魔法与赛博积木

SignKirigami









一些常识

- QEMU 是一个通用的开源机器模拟器和虚拟化器，由 Fabrice Bellard 于 2003 年首次发布。
- 可以模拟不同的计算机架构，让一个架构的系统能够运行另一个架构的操作系统和程序；同时，在支持硬件虚拟化技术的情况下，能以接近原生的性能运行虚拟机。
- 可以模拟各种常见的硬件设备，如硬盘（IDE、SATA、SCSI 等接口）、网卡（E1000、virtio 等）、显卡、USB 设备等。
- 支持全系统模拟、单个用户程序的模拟。

查看 QEMU 支持模拟的机器和设备

```
prcups@yuno-laptop [ ~ ] $ qemu-system-aarch64 -M help |grep raspi
raspi0               Raspberry Pi Zero (revision 1.2)
raspi1ap             Raspberry Pi A+ (revision 1.1)
raspi2b              Raspberry Pi 2B (revision 1.1)
raspi3ap             Raspberry Pi 3A+ (revision 1.0)
raspi3b              Raspberry Pi 3B (revision 1.2)
raspi4b              Raspberry Pi 4B (revision 1.5)
```

```
prcups@yuno-laptop [ ~ ] ! qemu-system-loongarch64 -device help |grep net
name "e1000", bus PCI, alias "e1000-82540em", desc "Intel Gigabit Ethernet"
name "e1000-82544gc", bus PCI, desc "Intel Gigabit Ethernet"
name "e1000-82545em", bus PCI, desc "Intel Gigabit Ethernet"
name "i82550", bus PCI, desc "Intel i82550 Ethernet"
name "i82551", bus PCI, desc "Intel i82551 Ethernet"
name "i82557a", bus PCI, desc "Intel i82557A Ethernet"
name "i82557b", bus PCI, desc "Intel i82557B Ethernet"
name "i82557c", bus PCI, desc "Intel i82557C Ethernet"
name "i82558a", bus PCI, desc "Intel i82558A Ethernet"
name "i82558b", bus PCI, desc "Intel i82558B Ethernet"
```

话题介绍

- 在 QEMU 中如何模拟机器和设备?
- 如何实现 LoongArch 板卡的模拟?有哪些实际困难?

QEMU 对象模型（QOM）

- QEMU 本体是由 C 语言实现的。为了方便开发，QEMU 实现了一套面向对象的编程框架，即 QOM 。
- 使用 QOM ，可以动态的添加、删除一个对象，或是从一个对象派生出另一个对象并继承其方法。
- QEMU 中的机器、设备都是 QOM 。一个 QOM 在建立的过程中可以创建更多子 QOM ，形成一棵树。可以在 monitor 中查看当前的 QOM 树，如右图。

```
QEMU 9.2.2 monitor - type 'help' for more information
(qemu) info qom-tree
/machine (virt-machine)
  /fw_cfg (fw_cfg_mem)
    /\x2from@etc\x2facpi\x2frsdp[0] (memory-region)
    /\x2from@etc\x2facpi\x2ftables[0] (memory-region)
    /\x2from@etc\x2ftable-loader[0] (memory-region)
    /fwcfg.ctl[0] (memory-region)
    /fwcfg.data[0] (memory-region)
    /fwcfg.dma[0] (memory-region)
  /iocsr[0] (memory-region)
  /iocsr_misc[0] (memory-region)
  /peripheral (container)
  /peripheral-anon (container)
  /unattached (container)
    /device[0] (la464-loongarch-cpu)
      /unnamed-gpio-in[0] (irq)
      /unnamed-gpio-in[10] (irq)
      /unnamed-gpio-in[11] (irq)
      /unnamed-gpio-in[12] (irq)
      /unnamed-gpio-in[1] (irq)
      /unnamed-gpio-in[2] (irq)
      /unnamed-gpio-in[3] (irq)
      /unnamed-gpio-in[4] (irq)
      /unnamed-gpio-in[5] (irq)
      /unnamed-gpio-in[6] (irq)
      /unnamed-gpio-in[7] (irq)
      /unnamed-gpio-in[8] (irq)
      /unnamed-gpio-in[9] (irq)
    /device[10] (virtio-net-pci)
```

示例:定义一个设备

- 右图是一个设备类型的典型定义。
- 我们通过type_init 宏定义了一个类型,这个类型派生自DEVICE 。

```c
#include "qdev.h"

#define TYPE_MY_DEVICE "my-device"

typedef DeviceClass MyDeviceClass;
typedef struct MyDevice
{
    DeviceState parent_obj;

    int reg0, reg1, reg2;
} MyDevice;

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
};

static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}

type_init(my_device_register_types)
```

Class 与 Instance

- QOM 中与类型相关的概念有类（Class）和实例（Instance）。
- 类在 QEMU 初始化时注册，全局唯一；实例通过形如 object_new、qdev_new 之类的函数动态创建和销毁。

```
static void ls7a_rtc_class_init(ObjectClass *klass, const void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);   Incompatible integer to
    dc->vmsd = &vmstate_ls7a_rtc;
    dc->realize = ls7a_rtc_realize;
    device_class_set_legacy_reset(dc, ls7a_rtc_reset);
    dc->desc = "ls7a rtc";
}

static const TypeInfo ls7a_rtc_info = {
    .name          = TYPE_LS7A_RTC,
    .parent        = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(LS7ARtcState),
    .class_init    = ls7a_rtc_class_init,
};

static void ls7a_rtc_register_types(void)
{
    type_register_static(&ls7a_rtc_info);
}

type_init(ls7a_rtc_register_types)
```

Class 与 Instance

```c
static void loongarch_extioi_class_init(ObjectClass *klass, const void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);  Incompatible integer to pointe
    LoongArchExtIOIClass *lec = LOONGARCH_EXTIOI_CLASS(klass);  Incompatib
    LoongArchExtIOICommonClass *lecc = LOONGARCH_EXTIOI_COMMON_CLASS(klass);

    device_class_set_parent_realize(dc, loongarch_extioi_realize,
                                    &lec->parent_realize);
    device_class_set_parent_unrealize(dc, loongarch_extioi_unrealize,
                                      &lec->parent_unrealize);
    device_class_set_legacy_reset(dc, loongarch_extioi_reset);
    lecc->post_load = vmstate_extioi_post_load;
}

static const TypeInfo loongarch_extioi_types[] = {
    {
        .name          = TYPE_LOONGARCH_EXTIOI,
        .parent        = TYPE_LOONGARCH_EXTIOI_COMMON,
        .instance_size = sizeof(LoongArchExtIOIState),
        .class_size    = sizeof(LoongArchExtIOIClass),
        .class_init    = loongarch_extioi_class_init,
    }
};

DEFINE_TYPES(loongarch_extioi_types)  Unknown type name 'typeof'; did you
```

```c
#define TYPE_LOONGARCH_EXTIOI "loongarch.extioi"
OBJECT_DECLARE_TYPE(LoongArchExtIOIState, LoongArchExtIOIClass, LOONGARCH_EXTIOI)

struct LoongArchExtIOIState {
    LoongArchExtIOICommonState parent_obj;
};

struct LoongArchExtIOIClass {
    LoongArchExtIOICommonClass parent_class;

    DeviceRealize parent_realize;
    DeviceUnrealize parent_unrealize;
};
```

```c
#define OBJECT_DECLARE_TYPE(InstanceType, ClassType, MODULE_OBJ_NAME) \
    typedef struct InstanceType InstanceType; \
    typedef struct ClassType ClassType; \
    \
    G_DEFINE_AUTOPTR_CLEANUP_FUNC(InstanceType, object_unref) \
    \
    DECLARE_OBJ_CHECKERS(InstanceType, ClassType, \
                         MODULE_OBJ_NAME, TYPE_##MODULE_OBJ_NAME)
```

接口与继承

```c
typedef struct MyState MyState;

typedef void (*MyDoSomething)(MyState *obj);

typedef struct MyClass {
    ObjectClass parent_class;

    MyDoSomething do_something;
} MyClass;

static void my_do_something(MyState *obj)
{
    // do something
}

static void my_class_init(ObjectClass *oc, const void *data)
{
    MyClass *mc = MY_CLASS(oc);

    mc->do_something = my_do_something;
}

static const TypeInfo my_type_info = {
    .name = TYPE_MY,
    .parent = TYPE_OBJECT,
    .instance_size = sizeof(MyState),
    .class_size = sizeof(MyClass),
    .class_init = my_class_init,
};
```

```c
typedef struct DerivedClass {
    MyClass parent_class;

    MyDoSomething parent_do_something;
} DerivedClass;

static void derived_do_something(MyState *obj)
{
    DerivedClass *dc = DERIVED_GET_CLASS(obj);

    // do something here
    dc->parent_do_something(obj);
    // do something else here
}

static void derived_class_init(ObjectClass *oc, const void *
{
    MyClass *mc = MY_CLASS(oc);
    DerivedClass *dc = DERIVED_CLASS(oc);

    dc->parent_do_something = mc->do_something;
    mc->do_something = derived_do_something;
}

static const TypeInfo derived_type_info = {
    .name = TYPE_DERIVED,
    .parent = TYPE_MY,
    .class_size = sizeof(DerivedClass),
    .class_init = derived_class_init,
};
```

助手宏

- 为了简化编写，QEMU 提供了一些宏，如 DEFINE_TYPES、DEFINE_MACHINE、OBJECT_DEFINE_SIMPLE_TYPE 等。

```
/**
 * DEFINE_TYPES:
 * @type_array: The array containing #TypeInfo structures to register
 *
 * @type_array should be static constant that exists for the life time
 * that the type is registered.
 */
#define DEFINE_TYPES(type_array)                                        \
static void do_qemu_init_ ## type_array(void)                           \
{                                                                       \
    type_register_static_array(type_array, ARRAY_SIZE(type_array));     \
}                                                                       \
type_init(do_qemu_init_ ## type_array)
```

```
static const TypeInfo device_types_info[] = {
    {
        .name = TYPE_MY_DEVICE_A,
        .parent = TYPE_DEVICE,
        .instance_size = sizeof(MyDeviceA),
    },
    {
        .name = TYPE_MY_DEVICE_B,
        .parent = TYPE_DEVICE,
        .instance_size = sizeof(MyDeviceB),
    },
};

DEFINE_TYPES(device_types_info)
```

定义 2K1000 开发板

```c
static void ls2k1000_class_init(ObjectClass *oc, const void* data)
{
    MachineClass *mc = MACHINE_CLASS(oc);  Incompatible integer

    mc->desc = "ls2k1000 platform";
    mc->init = ls2k1000_init;
    mc->default_cpus = LS2K1000_CPUS;
    mc->block_default_type = IF_IDE;
    mc->default_cpu_type = LOONGARCH_CPU_TYPE_NAME("la464");
    mc->default_ram_id = "loongarch.ram";
    mc->default_ram_size = 1 * GiB;
}

static const TypeInfo ls2k1000_machine_typeinfo = {
    .name       = MACHINE_TYPE_NAME("ls2k1000"),
    .parent     = TYPE_MACHINE,
    .class_init = ls2k1000_class_init,
    .instance_size = sizeof(LS2K1000MachineState),
};

static void ls2k1000_machine_init_register_types(void)
{
    type_register_static(&ls2k1000_machine_typeinfo);
}

type_init(ls2k1000_machine_init_register_types)
```

```c
#define LS2K1000_CPUS 2

typedef struct LS2K1000MachineState {
    /*< private >*/
    MachineState parent_obj;

    MemoryRegion bios;
    MemoryRegion cpu_config;
    MemoryRegion ram1, ram2, ram3;
    MemoryRegion ddr_config;
    uint64_t reg_ddr;
    uint8_t ddr_level_reg[0x1000];

    DeviceState *ipi;
    DeviceState *liointc;
    DeviceState *spi;

    PCIBus *pci_bus;

    LoongArchCPU* cpu[LS2K1000_CPUS];
} LS2K1000MachineState;

#define TYPE_LS2K1000_MACHINE MACHINE_TYPE_NAME("ls2k1000")
#define LS2K1000_MACHINE(obj) \
OBJECT_CHECK(LS2K1000MachineState, (obj), TYPE_LS2K1000_MACHINE)
```

实现 CPU、中断控制器

- 在机器的初始化函数中，通过 qdev_new 实现各个需要的设备。
- 实现方法因设备而异。有的设备需要设置好参数后调用形如 qdev_realize 的方法。

```
LS2K1000MachineState* ls2k1000ms = LS2K1000_MACHINE(ms);
int i;

if (ms->smp.cpus != 2) {
    error_report("This machine can only be used with 2 Core");  | Call to undeclared fu
    exit(EXIT_FAILURE);
}

ls2k1000ms->ipi = qdev_new(TYPE_LS2K_IPI);
qdev_prop_set_uint32(ls2k1000ms->ipi, "num-cpu", ms->smp.cpus);
sysbus_realize_and_unref(SYS_BUS_DEVICE(ls2k1000ms->ipi), &error_fatal);  | Call to unde

ls2k1000ms->liointc = qdev_new(TYPE_LS2K_LIOINTC);
qdev_prop_set_uint32(ls2k1000ms->liointc, "num-cpu", ms->smp.cpus);
sysbus_realize_and_unref(SYS_BUS_DEVICE(ls2k1000ms->liointc), &error_fatal);  | Incompat

for (i = 0; i < ms->smp.cpus; i++) {
    Object *cpuobj = NULL;
    cpuobj = object_new(LOONGARCH_CPU_TYPE_NAME("la464"));
    if (cpuobj == NULL) {
        error_report("Fail to create object with type %s ",  | Call to undeclared funct
                     LOONGARCH_CPU_TYPE_NAME("la464"));
        exit(EXIT_FAILURE);
    }

    ls2k1000ms->cpu[i] = LOONGARCH_CPU(cpuobj);  | Incompatible integer to pointer conve
    qdev_realize_and_unref(DEVICE(cpuobj), NULL, &error_fatal);  | Call to undeclared fu
    qdev_connect_gpio_out(ls2k1000ms->ipi, i, qdev_get_gpio_in(DEVICE(cpuobj), IRQ_IPI));
    for (int j = 0; j < 4; j++) {
        qdev_connect_gpio_out(ls2k1000ms->liointc, PARENT_COREx_IPy(i, j),
                              qdev_get_gpio_in(DEVICE(cpuobj), 2 + j));  | Incompatible
    }
}
```

设备内存与地址空间

- QEMU 提供了 MemoryRegion 类型和一系列 api，实现内存的管理。
- 通过 memory_region_init_io 初始化设备内存，设定其大小、处理函数，再通过 memory_region_add_subregion 函数将其插入到机器的地址空间的指定位置。
- 机器的物理内存也要手动映射到地址空间。

```
static void mips_qemu_write (void *opaque, hwaddr addr,
                             uint64_t val, unsigned size)
{
    if ((addr & 0xffff) == 0 && val == 42)
        qemu_system_reset_request(SHUTDOWN_CAUSE_GUEST_RESET);
    ...
}

static uint64_t mips_qemu_read (void *opaque, hwaddr addr,
                                unsigned size)
{
    return 0;
}

static const MemoryRegionOps mips_qemu_ops = {
    .read = mips_qemu_read,
    .write = mips_qemu_write,
    .endianness = DEVICE_NATIVE_ENDIAN,
};

void mips_r4k_init(MachineState *machine)
{
    ...
    MemoryRegion *iomem = g_new(MemoryRegion, 1);

    memory_region_init_io(iomem, NULL, &mips_qemu_ops, NULL, "mips-qemu", 0x10000);
    memory_region_add_subregion(get_system_memory(), 0x1fbf0000, iomem);
}
```

中断的实现和处理

- QEMU 提供了 qemu_irq 类型，代表中断线。
- 使用 qemu_init_gpio_out 初始化 qemu_irq 。
- 使用 qdev_connect_gpio_out 和 qdev_get_gpio_in 连接中断线。
- 中断管理是 qdev api 的一部分。

```
static void ls2k_liointc_realize(DeviceState *dev, Error **errp)
{
    LS2KLIOINTCState *s = LS2K_LIOINTC(dev);

    ...
    qdev_init_gpio_in(dev, irq_handler, 64);

    s->parent_irq = g_new0(qemu_irq, s->num_cpu * NUM_IPS);
    qdev_init_gpio_out(dev, s->parent_irq, s->num_cpu * NUM_IPS);
}

...

static void ls2k1000_init(MachineState *ms)
{
    ...
    for (i = 0; i < ms->smp.cpus; i++) {
        ...
        for (int j = 0; j < 4; j++) {
            qdev_connect_gpio_out(ls2k1000ms->liointc, PARENT_COREx_IPy(i, j),
                                  qdev_get_gpio_in(DEVICE(cpuobj), 2 + j));
        }
    }
}
```

更多设备的实现

- bios rom
- pci 总线
- serial 、rtc 、spi
- 硬盘、网卡、显卡、声卡
- ……

2k1000 实现过程中的问题

- 实现新设备 / 与旧设备磨合?

  目前 qemu 主线实现了对 LoongArch64 及 MIPS64 的 3A 平台的支持，与 LoongArch64 2K 平台有着微妙的差异。

- 与 Uboot 固件联动

  如 Uboot 要求 bios rom 以 spi flash 方式存在于机器中，且 spi flash 访问方式与主线支持的 spi flash 设备不同。

- 最怀念 C++ 的一集

  C 语言实现的面向对象一定少不了与宏魔法打交道的过程，需要耐下性子。

  ~~当然，看久了 C 代码可以看点 Qt 程序放松一下~~

谢谢