码农故事

搜索

搜索

首页

← 上一篇

下一篇 →

QEMU设备模拟

发表于 2014 年 10 月 15 日

设备模拟目的

我们好像不会干一件事而毫无目的，就算不停刷微信朋友圈也是为了打发你无聊的时间。
其实最装B的回答是：设备模拟的目的就是模拟设备。这话是屁话，不过也能说明些什么，确实是模拟设备，用软件的方式提供硬件设备具备的功能。
对于和PC机交互的硬件设备，主要要干两件事，一是提供IRQ中断，二是响应IO输入输出。IO包括PIO/MMIO/DMA等（DMA算不算IO？）
以i8254.c实现的pit为例，主要提供了IRQ注入和PIO响应，见初始化函数pit_initfn：

```
static const MemoryRegionOps pit_ioport_ops = {
    .read = pit_ioport_read,
    .write = pit_ioport_write,
    .impl = {
        .min_access_size = 1,
        .max_access_size = 1,
    },
    .endianness = DEVICE_LITTLE_ENDIAN,
};

static int pit_initfn(PITCommonState *pit)
{
    PITChannelState *s;

    s = &pit->channels[0];
    /* the timer 0 is connected to an IRQ */
    //这里有个irq_timer，用于qemu_set_irq提供中断注入
    s->irq_timer = qemu_new_timer_ns(vm_clock, pit_irq_timer, s);
    qdev_init_gpio_out(&pit->dev.qdev, &s->irq, 1);

    memory_region_init_io(&pit->ioports, &pit_ioport_ops, pit, "pit", 4);
    qdev_init_gpio_in(&pit->dev.qdev, pit_irq_control, 1);
    return 0;
}
```

这里的pit_ioport_ops，主要注册GUEST操作系统读写PIO时候的回调函数。

模块注册

QEMU要模拟模块那么多，以程序员的喜好，至少得来一套管理这些模拟设备模块的接口，以示设计良好。
QEMU将被模拟的模块分为了四类：

```
typedef enum {
    MODULE_INIT_BLOCK,
    MODULE_INIT_MACHINE,
    MODULE_INIT_QAPI,
    MODULE_INIT_QOM,
    MODULE_INIT_MAX
} module_init_type;
```

- **BLOCK**
  比如磁盘IO就属于BLOCK类型，e.g: block_init(bdrv_qcow2_init); block_init(iscsi_block_init);
- **MACHINE**
  PC虚拟machine_init(pc_machine_init); XEN半虚拟化machine_init(xenpv_machine_init); MIPS虚拟

machine_init(mips_machine_init);

- **QAPI**

  QEMU GUEST AGENT模块里面会执行QAPI注册的回调

- **QOM**

  QOM树大枝多，儿孙满堂，应该是这里面最复杂的一个，我们重点介绍。

  e.g:

```
ObjectClass ->  PCIDeviceClass   //显卡type_init(cirrus_vga_register_types)，网卡type_init(rtl8139_register_ty
                IDEDeviceClass   //IDE硬盘或CD-ROM type_init(ide_register_types)
                ISADeviceClass  //鼠标键盘type_init(i8042_register_types)，RTC时钟type_init(pit_register)
                SysBusDeviceClass //MMIO IDE(IDE设备直接连接CPU bus而不是连接IDE controller)type_init(mmio_ide_
                 ->  CRISCPUClass
```

注册QOM设备的时候，使用QEMU提供的宏，type_init宏进行注册：

```
#define type_init(function) module_init(function, MODULE_INIT_QOM)
#define module_init(function, type)
    static void __attribute__((constructor)) do_qemu_init_ ## function(void) {
    register_module_init(function, type);
}
```

这和写LINUX驱动类似，一般写在一个模块实现文件的最底部，以pit为例，写的是type_init(pit_register_types)展开后为：

```
static void __attribute__((constructor)) do_qemu_init_pit_register_types(void)
{
    register_module_init(pit_register_types, MODULE_INIT_QOM);
}
```

那么，这个do_qemu_init_pit_register_types何时调用？
在gcc里面，给函数加上__attribute__((destructor))，表示此函数需要在main开始前自动调用，测试调用顺序是: 全局对象构造函数 -> __attribute__((constructor)) -> main -> 全局对象析构函数 -> __attribute__((destructor))。
调用register_module_init就是将pit_register_types回调函数插入utilmodule.c里定义的init_type_list[MODULE_INIT_QOM]链表内。

```
void register_module_init(void (*fn)(void), module_init_type type)
{
    ModuleEntry *e;
    ModuleTypeList *l;
    e = g_malloc0(sizeof(*e));
    e->init = fn; //init指针被设置为fn
    l = find_type(type);
    QTAILQ_INSERT_TAIL(l, e, node);
}
```

通过下面main函数的部分代码可以看出，模块初始化顺序是QOM->MACHINE->BLOCK，至于QAPI，在这个流程里没看到。

```
void main()
{
    module_call_init(MODULE_INIT_QOM); //初始化设备
    qemu_add_opts //初始化默认选项
    module_call_init(MODULE_INIT_MACHINE); //初始化机器类型
    machine = find_default_machine(); //这里对machine赋值，下面还会通过参数更改machine
    vtp_script_execute(g_qemu_start_hook_path, g_fairsched_string, TYPE_START); //开机启动脚本的调用
    深度分析启动参数
    bdrv_init_with_whitelist -> bdrv_init -> module_call_init(MODULE_INIT_BLOCK); //初始化BLOCK设备
    machine->init(&args); //初始化machine

    qemu_run_machine_init_done_notifiers(); //初始化成功回调通知
    qemu_system_reset(VMRESET_SILENT);//system reset 启动运行
    if (loadvm) {
        load_vmstate(loadvm);
    } else if (loadstate) {
        load_state_from_blockdev(loadstate);
    }

    resume_all_vcpus();
    main_loop(); //进入主循环
}
```

在main函数进来的时候，首先调用module_call_init(MODULE_INIT_QOM);

```
void module_call_init(module_init_type type)
{
    ModuleTypeList *l;
    ModuleEntry *e;
    l = find_type(type);
    QTAILQ_FOREACH(e, l, node) {
        e->init(); //这里，就是调用刚才注册的回调，例如，对于kvm-pit来说，调用的是pit_register
    }
}
```

此module_call_init将依次调用注册的回调，如PIT的pit_register_types：

```
static const TypeInfo pit_info = {
```

```
    .name          = "isa-pit",       //做为type_table的key
    .parent        = "pit-common",    //父类型，这个比较重要，如果本TypeInfo没有设置class_size，会根据parent获取par
    .instance_size = sizeof(PITCommonState),//分配实例的大小
    .class_init = pit_class_init,     //初始化函数
};

static void pit_register_types(void)
{
    type_register_static(&pit_info);
}
```

pit_register_types又进一步调用type_register_static -> type_register -> type_register_internal，这个函数完成的功能其实只是在qomobject.c的type_table里插入了一个HASH键值对，以TypeInfo的name为KEY，malloc了一个TypeInfo结构的超集TypeImpl为VALUE，在以name为KEY回溯parent时需要TypeImpl，其实这个hash也可以做成一个tree。

QOM的Object模型

以pit为例，通过回溯parent你可以看到，其定义TypeInfo最终形成一个继承关系：
"isa-pit" -> "pit-common" -> "isa-device" -> "device" -> "object"

qomobject.c

```
static TypeInfo object_info = {
    .name = "object",
    .instance_size = sizeof(Object),
    .instance_init = object_instance_init,
    .abstract = true,
};
```

hwqdev.c

```
static const TypeInfo device_type_info = {
    .name = "device",
    .parent = "object" ,
    .instance_size = sizeof(DeviceState),
    .instance_init = device_initfn,
    .instance_finalize = device_finalize,
    .class_base_init = device_class_base_init,
    .class_init = device_class_init,
    .abstract = true,
    .class_size = sizeof(DeviceClass),
};
```

hwisa-bus.c

```
static const TypeInfo isa_device_type_info = {
    .name = "isa-device",
    .parent = "device",
    .instance_size = sizeof(ISADevice),
    .abstract = true,
    .class_size = sizeof(ISADeviceClass),
    .class_init = isa_device_class_init,
};
```

hwi8254_common.c

```
static const TypeInfo pit_common_type = {
    .name          = "pit-common",
    .parent        = "isa-device",
    .instance_size = sizeof(PITCommonState),
    .class_size    = sizeof(PITCommonClass),
    .class_init    = pit_common_class_init,
    .abstract      = true,
};
```

hwi8254.c

```
static const TypeInfo pit_info = {
    .name          = "isa-pit",
    .parent        = "pit-common",
    .instance_size = sizeof(PITCommonState),
    .class_init    = pit_class_initfn,
};
```

由于TypeInfo只是注册时临时使用，而TypeImpl是TypeInfo的超集，所以，这层关系也反应了TypeImpl的继承关系。

```
struct TypeImpl
{
    const char *name;
    size_t class_size;
    size_t instance_size;
    void (*class_init)(ObjectClass *klass, void *data);
```

```
    void (*class_base_init)(ObjectClass *klass, void *data);
    void (*class_finalize)(ObjectClass *klass, void *data);
    void *class_data;
    void (*instance_init)(Object *obj);
    void (*instance_finalize)(Object *obj);
    bool abstract;
    const char *parent;
    TypeImpl *parent_type;
    ObjectClass *class;
    int num_interfaces;
    InterfaceImpl interfaces[MAX_INTERFACES];
};
```

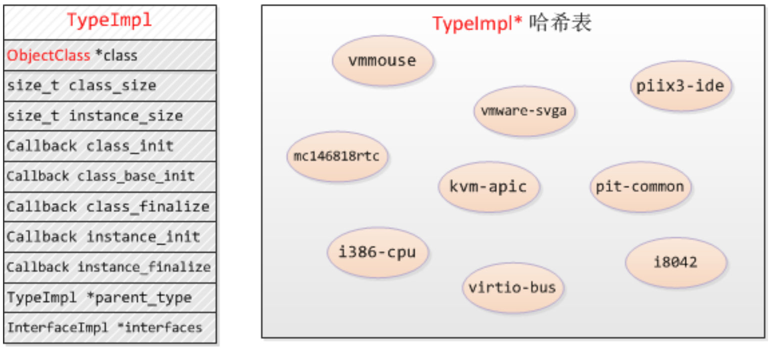

Figure 1 TypeImpl图解
打印查看TypeImpl属性：

```
(gdb) p *obj->class->type //struct TypeImpl * type
$13 = {name = 0x5555566e5e30 "mc146818rtc", class_size = 128, instance_size = 664, class_init = 0x555555
  class_finalize = 0, class_data = 0x0, instance_init = 0, instance_finalize = 0, abstract = false, pare
  parent_type = 0x5555566d8bd0, class = 0x555556a50e50, num_interfaces = 0, interfaces = {{typename = 0x
```

其主要包含如下部分：

- name/parent/parent_type 表示自己的，父亲的KEY和TypeImpl指针。
- class/class_size/class_init/class_base_init/class_finalize/class_data 和ObjectClass联系，组成继承关系。
- instance_init/instance_finalize和ObjectClass有裙带关系的Object，共同完成继承体系。
- num_interfaces/interfaces 用于管理接口。

Object和ObjectClass的关系

还是通过这条继承链来看：
"isa-pit" -> "pit-common" -> "isa-device" -> "device" -> "object"
其中ObjectClass链的定义为：

```
struct ObjectClass
{
    /*< private >*/
    Type type;
    GSList *interfaces;
    ObjectUnparent *unparent;
};
typedef struct DeviceClass {
    /*< private >*/
    ObjectClass parent_class;
    /*< public >*/
    const char *fw_name;
    const char *desc;
    Property *props;
    int no_user;
    /* callbacks */
    void (*reset)(DeviceState *dev);
    DeviceRealize realize;
    DeviceUnrealize unrealize;
    /* device state */
    const struct VMStateDescription *vmsd;
    /* Private to qdev / bus.   */
    qdev_initfn init; /* TODO remove, once users are converted to realize */
    qdev_event unplug;
    qdev_event exit;
    const char *bus_type;
} DeviceClass;
typedef struct ISADeviceClass {
    DeviceClass parent_class;
    int (*init)(ISADevice *dev);
} ISADeviceClass;
typedef struct PITCommonClass {
    ISADeviceClass parent_class;
    int (*init)(PITCommonState *s);
    void (*set_channel_gate)(PITCommonState *s, PITChannelState *sc, int val);
    void (*get_channel_info)(PITCommonState *s, PITChannelState *sc,
        PITChannelInfo *info);
    void (*pre_save)(PITCommonState *s);
    void (*post_load)(PITCommonState *s);
} PITCommonClass;
```

下层定义包含上层，很明显的继承模型，ObjectClass更像C++的CLASS，而Object链的定义为：

```
struct Object
{
    /*< private >*/
    ObjectClass *class;
    ObjectFree *free;
    QTAILQ_HEAD(, ObjectProperty) properties;
    uint32_t ref;
    Object *parent;
};
struct DeviceState {
    /*< private >*/
    Object parent_obj;
    /*< public >*/
    const char *id;
    bool realized;
    QemuOpts *opts;
    int hotplugged;
    BusState *parent_bus;
    int num_gpio_out;
    qemu_irq *gpio_out;
    int num_gpio_in;
    qemu_irq *gpio_in;
    QLIST_HEAD(, BusState) child_bus;
    int num_child_bus;
    int instance_id_alias;
    int alias_required_for_version;
};
struct ISADevice {
    DeviceState qdev;
    uint32_t isairq[2];
    int nirqs;
    int ioport_id;
};
typedef struct PITCommonState {
    ISADevice dev;
    MemoryRegion ioports;
    uint32_t iobase;
    PITChannelState channels[3];
} PITCommonState;
```

有了ObjectClass为什么还要有个Object？从代码看，ObjectClass只有一份实例，而Object是可以多个实例的，Object引用ObjectClass获得ObjectClass的特征，但是同时又节约了初始化和存放ObjectClass的CPU和空间，相同的ObjectClass可以被多个Object引用，例如scsi-disk.c里面有"scsi-hd","scsi-cd","scsi-block","scsi-disk"四种Object共同引用了"scsi-device"。这里可以想象成C++的虚继承，ObjectClass是virtual class而Object是class。其实两者是可以柔和在一起的，Object也有对应的继承关系，用来保存特定属性。

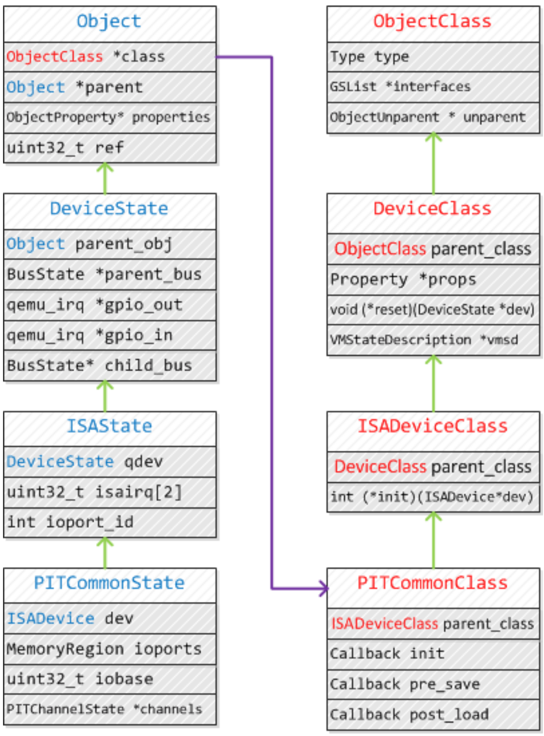

Figure 2 ObjectClass和Object 关系

Object和ObjectClass的初始化

上面讲的Object和ObjectClass主要是完成一个对象继承模型，从代码看QEMU的这个模型实现并不非常很优雅，封装不够彻底，就像你妈给你做了条裤子，却没有做裤腰带，还得提着上路。

Object和ObjectClass的初始化方式并不一致，需要分别初始化，ObjectClass通常使用object_class_by_name获取，此函数会根据提供的KEY去查找TypeImpl并初始化ObjectClass指针；而Object的初始化是使用的object_new，通过参数KEY查找TypeImpl然后

malloc 实例。以qdev_try_create获取"isa-pit"的Object DeviceState实例来说，其获取DeviceState的函数如此定义：

```
DeviceState *qdev_try_create(BusState *bus, const char *type)
{
    DeviceState *dev;
    //这个type为TypeInfo.name，例如"isa-pit"
    if (object_class_by_name(type) == NULL) {
        return NULL;
    }
    //type_initialize完成后，object_new用来实例化一个instance
    dev = DEVICE(object_new(type));// = DEVICE(object_new_with_type(type_get_by_name(typename)))
    if (!dev) {
        return NULL;
    }
    if (!bus) {
        bus = sysbus_get_default();
    }
    qdev_set_parent_bus(dev, bus);
    object_unref(OBJECT(dev));
    return dev;
}

ObjectClass *object_class_by_name(const char *typename)
{
    //之前在type_register_static的时候，注册了TypeInfo.name，例如"isa-pit"为key的TypeImpl
    TypeImpl *type = type_get_by_name(typename);
    if (!type) {
        return NULL;
    }
    type_initialize(type); //这里面，初始化class，
    return type->class;
}

//其实这个函数更应该叫做new_TypeInfo_class()
static void type_initialize(TypeImpl *ti)
{
    TypeImpl *parent;
    if (ti->class) {
        return;
    }
    /*
    type_class_get_size 首先获取自己的class_size变量，如果没有，再找parent类型所指的TypeImpl的class_size，直到找到为止
    比如"isa-pit"没有设置class_size，那么获取的是"pit-common"的class_size, 而type_object_get_size也是类似
    static const TypeInfo pit_common_type = {
        .name          = "pit-common",
        .parent        = "isa-device",
        .instance_size = sizeof(PITCommonState),
        .class_size    = sizeof(PITCommonClass),
        .class_init    = pit_common_class_init,
        .abstract      = true,
    };
    */
    ti->class_size = type_class_get_size(ti);
    ti->instance_size = type_object_get_size(ti);
    ti->class = g_malloc0(ti->class_size);
    parent = type_get_parent(ti);
    if (parent) {
        //1，保证parent初始化了
        type_initialize(parent);
        GSList *e;
        int i;

        //2，将parent的class内容memcpy一份给自己的对应的parent区域
        g_assert(parent->class_size <= ti->class_size);
        memcpy(ti->class, parent->class, parent->class_size);

        //3，将parent里面的class的interfaces做一次深度复制，复制给自己
        for (e = parent->class->interfaces; e; e = e->next) {
            ObjectClass *iface = e->data;
            type_initialize_interface(ti, object_class_get_name(iface));
        }

        //4.如果本类型有自己的interfaces，初始化
        for (i = 0; i < ti->num_interfaces; i++) {
            TypeImpl *t = type_get_by_name(ti->interfaces[i].typename);
            for (e = ti->class->interfaces; e; e = e->next) {
                TypeImpl *target_type = OBJECT_CLASS(e->data)->type;
                if (type_is_ancestor(target_type, t)) {
                    break;
                }
            }
            if (e) {
                continue;
            }
            type_initialize_interface(ti, ti->interfaces[i].typename);
        }
    }

    ti->class->type = ti;
    while (parent) {
        if (parent->class_base_init) {
            //回溯回调parent的class_base_init函数
            parent->class_base_init(ti->class, ti->class_data);
        }
        parent = type_get_parent(parent);
    }

    if (ti->class_init) {
        /*
        如果本类设置了class_init，回调它，ti->class_data是一个void*的参数
        比如"isa-pit"我们设置了pit_class_initfn
        这个函数主要干啥？主要填充class里的其他该填充的地方。
```

```
        malloc之后你总得调用构造函数吧，调用构造函数的第一句都是super(xxx)
        这工作前面,2,3步骤已经做了，然后干你自己的活。见pit_class_initfn定义
        */
        ti->class_init(ti->class, ti->class_data);
    }
}
```

上述代码把object_class_by_name的流程说完了，再看看object_new(type) = object_new_with_type(type_get_by_name(typename))的流程：

```
Object *object_new_with_type(Type type)
{
    Object *obj;
    g_assert(type != NULL);
    type_initialize(type);
    obj = g_malloc(type->instance_size); //这个instance_size是初始化TypeInfo的时候设置的sizeof(PITCommonState)
    object_initialize_with_type(obj, type);
    obj->free = g_free;
    return obj;
}
void object_initialize_with_type(void *data, TypeImpl *type)
{
    Object *obj = data;
    g_assert(type != NULL);
    type_initialize(type);
    g_assert(type->instance_size >= sizeof(Object));
    g_assert(type->abstract == false);
    memset(obj, 0, type->instance_size);
    obj->class = type->class; //instace的类型通过class指针指定
    object_ref(obj);
    QTAILQ_INIT(&obj->properties);
    object_init_with_type(obj, type); //深度递归调用TypeImpl及其parent的instance_init函数指针，相当于new instance的构
}
```

qdev_try_create->object_class_by_name->type_initialize的调用流程，如果父ObjectClass没初始化，会初始化父ObjectClass，此时调用到父ObjectClass对应name的TypeImpl的class_init函数，例如"pit-common"的class_init回调pit_common_class_init，此回调会设置ISADeviceClass的init回调为pit_init_common。

Object类型转换

再解释下Object里常用的一个宏：OBJECT_CHECK，以ISA_DEVICE这段代码为例：

```
static const TypeInfo mc146818rtc_info = {
    .name          = "mc146818rtc",
    .parent        = "isa-device",
    .instance_size = sizeof(RTCState),
    .class_init    = rtc_class_initfn,
};
ISADevice *isa_create(ISABus *bus, const char *name)
{
    DeviceState *dev;
    if (!bus) {
        hw_error("Tried to create isa device %s with no isa bus present.", name);
    }
    dev = qdev_create(&bus->qbus, name);
    return ISA_DEVICE(dev);//毫无疑问，"mc146818rtc"肯定可以转换为"isa-device"，why？见mc146818rtc_info定义
}
ISADevice *dev = isa_create(bus, "mc146818rtc");
```

这里的ISA_DEVICE体现了OBJECT的类型转换功能，宏定义为：

```
#define OBJECT(obj)
    ((Object *)(obj))
#define OBJECT_CHECK(type, obj, name)
    ((type *)object_dynamic_cast_assert(OBJECT(obj), (name)))
#define ISA_DEVICE(obj)
    OBJECT_CHECK(ISADevice, (obj), TYPE_ISA_DEVICE)
```

展开后为：

```
#define ISA_DEVICE(dev) (ISADevice*)object_class_dynamic_cast(((Object *)dev)->class, "isa-device")
```

object.c里面定义了object_class_dynamic_cast函数，其实此函数功能比较简单，就是通过遍历parent看当前class是否有一个祖先是typename，其定义如下：

```
ObjectClass *object_class_dynamic_cast(ObjectClass *class,
                                       const char *typename)
{
    TypeImpl *target_type = type_get_by_name(typename); //找到"isa-device"对应的TypeImpl*
    TypeImpl *type = class->type; //本ObjectClass真实的TypeImpl*，其实这里是"mc146818rtc"
    ObjectClass *ret = NULL;
    /*
    (gdb) p *target_type
$31 = {name = 0x5555566d0b20 "isa-device", class_size = 128, instance_size = 160, class_init = 0x55555568ddd0 <isa
  class_finalize = 0, class_data = 0x0, instance_init = 0, instance_finalize = 0, abstract = true, parent = 0x5555
```

```
  class = 0x555556a14750, num_interfaces = 0, interfaces = {{typename = 0x0} <repeats 32 times>}}
    (gdb) p *type_interface
$33 = {name = 0x5555566e40b0 "interface", class_size = 32, instance_size = 0, class_init = 0, class_base_init = 0,
  instance_init = 0, instance_finalize = 0, abstract = true, parent = 0x0, parent_type = 0x0, class = 0x0, num_int
      typename = 0x0} <repeats 32 times>}}
    type_is_ancestor用于判断，"interface"是否是"isa-device"的祖先（判断方法是递归遍历"isa-device"的parent，比较是否有"i
    */
    if (type->num_interfaces && type_is_ancestor(target_type, type_interface)) {
        int found = 0;
        GSList *i;
        for (i = class->interfaces; i; i = i->next) {
            ObjectClass *target_class = i->data;
            if (type_is_ancestor(target_class->type, target_type)) {
                ret = target_class;
                found++;
            }
         }
        /* The match was ambiguous, don't allow a cast */
        if (found > 1) {
            ret = NULL;
        }
    } else if (type_is_ancestor(type, target_type)) {
        /*
        判断type="mc146818rtc"的祖先是否是target_type="isa-device"，
        如果是，这里表示子类class="mc146818rtc"能成功转换为父类typename="isa-device"所指的ObjectClass
        */
        ret = class;
    }
    return ret;
}
```

GDB显示其内部数据为：

```
(gdb) p *obj //Object *obj
$11 = {class = 0x555556a50e50, free = 0x7ffff7424020 , properties = {tqh_first = 0x555556a17da0, tqh_las
(gdb) p *obj->class //ObjectClass * class
$12 = {type = 0x5555566e5cb0, interfaces = 0x0, unparent = 0x5555556b1ac0 }
(gdb) p *obj->class->type //struct TypeImpl * type
$13 = {name = 0x5555566e5e30 "mc146818rtc", class_size = 128, instance_size = 664, class_init = 0x55555
  class_finalize = 0, class_data = 0x0, instance_init = 0, instance_finalize = 0, abstract = false, pare
  parent_type = 0x5555566d8bd0, class = 0x555556a50e50, num_interfaces = 0, interfaces = {{typename = 0x
```

QOM设备初始化

基于Object和ObjectClass实现的QOM设备，何时触发他的初始化，以PIT为例，将之前的Object和ObjectClass想象成C++，那么PIT对应的PitCommonState定义应该类似如下所示：

```
class PITCommonClass : public ISADeviceClass {
public:
    virtual int init(PITCommonState *s) = 0;
};

class ISADevice : public DeviceState {
public:
    int nirqs;
    int ioport_id;
};

class PITCommonState : public ISADevice, public PITCommonClass {
    int init(PITCommonState *s);
};
```

看吧，QEMU绕了这么大一个圈子，就想实现这样一个结构，所以有的时候用C++还是有好处的（虽然本人生理周期现正处于不太喜欢C++时间）。
那么，何处调用了new PITCommonState()？
这得从main函数开始看，main函数里面，有machine->init(&args);函数调用，这是对注册的machine的初始化，而默认的machine是在pc_piix.c里面pc_machine_init函数注册的第一个machine，即：

```
static QEMUMachine pc_i440fx_machine_v1_4 = {
    .name = "pc-i440fx-1.4",
    .alias = "pc",
    .desc = "Standard PC (i440FX + PIIX, 1996)",
    .init = pc_init_pci,
    .max_cpus = 255,
    .is_default = 1,
    .default_machine_opts = KVM_MACHINE_OPTIONS,
    DEFAULT_MACHINE_OPTIONS,
};
qemu_register_machine(&pc_i440fx_machine_v1_4);
```

当main函数调用machine->init时，我的实验环境默认情况其实就是调用的pc_i440fx_machine_v1_4的初始化回调pc_init_pci -> pc_init1，这个函数主要初始化相关PC硬件：

```
static void pc_init1(MemoryRegion *system_memory,
                     MemoryRegion *system_io,
                     ram_addr_t ram_size,
                     const char *boot_device,
                     const char *kernel_filename,
```

```
                      const char *kernel_cmdline,
                      const char *initrd_filename,
                      const char *cpu_model,
                      int pci_enabled,
                      int kvmclock_enabled)
{
    //CPU类型初始化-> cpu_x86_init -> mce_init/qemu_init_vcpu，初始化VCPU
    pc_cpus_init(cpu_model);
    //初始化acpi_tables
    pc_acpi_init("acpi-dsdt.aml");
    if (!xen_enabled()) {
        //ROM, BIOS, RAM相关初始化
        fw_cfg = pc_memory_init(system_memory,
            kernel_filename, kernel_cmdline, initrd_filename,
            below_4g_mem_size, above_4g_mem_size,
            rom_memory, &ram_memory);
    }
    //IRQ，初始化
    //VGA初始化
    pc_vga_init(isa_bus, pci_enabled ? pci_bus : NULL);
    /* init basic PC hardware */
    pc_basic_device_init(isa_bus, gsi, &rtc_state, &floppy, xen_enabled()); //这里调用pit_init
    //初始化网卡
    pc_nic_init(isa_bus, pci_bus);
    //初始化硬盘，音频设备
    //初始化cmos数据，比如设置cmos rtc时钟，是否提供PS/2设备
    pc_cmos_init(below_4g_mem_size, above_4g_mem_size, boot_device,
        floppy, idebus[0], idebus[1], rtc_state);
    //初始化USB
    if (pci_enabled && usb_enabled(false)) {
        pci_create_simple(pci_bus, piix3_devfn + 2, "piix3-usb-uhci");
    }
}

void pc_basic_device_init(ISABus *isa_bus, qemu_irq *gsi,
                          ISADevice **rtc_state,
                          ISADevice **floppy,
                          bool no_vmport)
{
    //初始化HPET
    //初始化mc146818 rtc
    //初始化i8042 PIT
    pit = pit_init(isa_bus, 0x40, pit_isa_irq, pit_alt_irq);
    //初始化串口，并口
    //初始化vmmouse ps2_mouse
}
```

接下来的流程是pit_init -> isa_create(bus, "isa-pit") -> qdev_create -> qdev_try_create，qdev_try_create的实现在前面已经讲了，如上节所述，它分别使用object_class_by_name和object_new来初始化ObjectClass和Object。

属性设置

Object同ObjectClass的显著区别就是Object提供了属性的概念，以MC146818为例，其定义时设置了"base_year"和"lost_tick_policy"：

```
static Property mc146818rtc_properties[] = {
    DEFINE_PROP_INT32("base_year", RTCState, base_year, 1980),
    DEFINE_PROP_LOSTTICKPOLICY("lost_tick_policy", RTCState,
    lost_tick_policy, LOST_TICK_DISCARD),
    DEFINE_PROP_END_OF_LIST(),
};
```

但是用GDB一看：

```
(gdb) p *obj->properties.tqh_first
$15 = {name = 0x555556a17df0 "type", type = 0x555556a17e10 "string", get = 0x555555727ae0
, set = 0,
  release = 0x555555727980
, opaque = 0x555556a17d80, node = {tqe_next = 0x555556a17e50, tqe_prev = 0x555556a17af0}}
(gdb) p *obj->properties.tqh_first.node.tqe_next
$21 = {name = 0x555556a17ea0 "realized", type = 0x555556a17ec0 "bool", get = 0x5555557279c0
, set = 0x555555727a40
,
  release = 0x555555727940
, opaque = 0x555556a17e30, node = {tqe_next = 0x555556a17ee0, tqe_prev = 0x555556a17dd0}}
(gdb) p *obj->properties.tqh_first.node.tqe_next.node.tqe_next
$22 = {name = 0x555556a17f30 "base_year", type = 0x555556a1be10 "int32", get = 0x5555556af370 , set = 0x
  opaque = 0x555555d4e1a0, node = {tqe_next = 0x555556a50ee0, tqe_prev = 0x555556a17e80}}
(gdb) p *obj->properties.tqh_first.node.tqe_next.node.tqe_next.node.tqe_next
$23 = {name = 0x555556a1be30 "lost_tick_policy", type = 0x555556a1be50 "LostTickPolicy", get = 0x5555556
  release = 0, opaque = 0x555555d4e1c0, node = {tqe_next = 0x555556a50fa0, tqe_prev = 0x555556a17f10}}
(gdb) p *obj->properties.tqh_first.node.tqe_next.node.tqe_next.node.tqe_next.node.tqe_next
$24 = {name = 0x555556a50f80 "parent_bus", type = 0x55555680b1d0 "link", get = 0x555555729e50 ,
  set = 0x55555572a320 , release = 0, opaque = 0x555556a17b30, node = {tqe_next = 0x0, tqe_prev = 0x5555
(gdb) p *obj->properties.tqh_first.node.tqe_next.node.tqe_next.node.tqe_next.node.tqe_next.node.tqe_next
Cannot access memory at address 0x0
```

事实上却多了"type" "realized" "parent_bus"，这些属性都是动态添加的。

在"object"类型的，instance_init = object_instance_init回调处，添加了

```
object_property_add_str(obj, "type", qdev_get_type, NULL, NULL);
```

在"device"类型的instance_init = device_initfn回调处，添加了

```
object_property_add_bool(obj, "realized",
                         device_get_realized, device_set_realized, NULL);
bject_property_add_link(OBJECT(dev), "parent_bus", TYPE_BUS,
                        (Object **)&dev->parent_bus, NULL);
```

设置属性的时候，调用类似qdev_prop_set_int32(&dev->qdev, "base_year", base_year);进行设置，这里，注意第一个参数为什么是DeviceState* dev->qdev而不是ISADevice *dev？
因为rtc_class_initfn里初始化props是给 DeviceClass *dc初始化的，所以对应的应该是DeviceState而不是子类ISADevice。

设置属性是如何实现的？
以"realized"的bool属性设置为例，调用顺序为object_property_set_bool -> object_property_set_qobject -> object_property_set，此函数定义：

```
void object_property_set(Object *obj, Visitor *v, const char *name,
                         Error **errp)
{
    //obj还是"mc146818rtc"的实例，name为"realized"，object_property_find其实就是查找obj的properties链表里是否存在名字为
    ObjectProperty *prop = object_property_find(obj, name, errp);
    if (prop == NULL) {
        return;
    }
    if (!prop->set) {//如果存在，且没有设置过set handler，错误
        error_set(errp, QERR_PERMISSION_DENIED);
    } else {//"realized"的set函数为property_set_bool
        prop->set(obj, v, prop->opaque, name, errp);
    }
}

static void property_set_bool(Object *obj, Visitor *v, void *opaque,
                              const char *name, Error **errp)
{
    BoolProperty *prop = opaque;
    bool value;
    Error *local_err = NULL;
    visit_type_bool(v, &value, name, &local_err);
    if (local_err) {
        error_propagate(errp, local_err);
        return;
    }
    prop->set(obj, value, errp); //对于realized来说，其实就是调用device_set_realized
}
```

而CALLBACK=device_set_realized 又会调用CALLBACK=device_realize。

模块PIO回调流程

注册回调

以mc146818 rtc为例，在rtc_initfn的时候，注册回调的代码如下：

```
static const MemoryRegionOps cmos_ops = {
    .read = cmos_ioport_read,
    .write = cmos_ioport_write,
    .impl = {
        .min_access_size = 1,
        .max_access_size = 1,
    },
    .endianness = DEVICE_LITTLE_ENDIAN,
};
void isa_register_ioport(ISADevice *dev, MemoryRegion *io, uint16_t start)
{
    memory_region_add_subregion(isabus->address_space_io, start, io);
    isa_init_ioport(dev, start);
}

memory_region_init_io(&s->io, &cmos_ops, s, "rtc", 2);
isa_register_ioport(dev, &s->io, base);
```

其中s->io是MemoryRegion类型，MemoryRegion是可以像树一样，多级挂载，比如，现在将rtc的MemoryRegion挂载在isabus的address_space_io这个MemoryRegion下，其start参数为offset在整个isabus->address_space_io MemoryRegion中的偏移，即0x70，那么END呢？END在memory_region_init_io的时候已经存储到MemoryRegion的size里面了。

再看看isabus内容，有个更深入的性感的认识：

```
(gdb) p *isabus
$8 = {qbus = {obj = {class = 0x555556a14e70, free = 0x7ffff7424020 , properties = {tqh_first = 0x555556a
      parent = 0x55555698d740}, parent = 0x55555698d740, name = 0x555556a14ff0 "isa.0", allow_hotplug =
      tqh_last = 0x555556a16450}, sibling = {le_next = 0x0, le_prev = 0x55555698d7b8}}, address_space_io
(gdb) p *isabus->address_space_io
$9 = {ops = 0x0, opaque = 0x0, parent = 0x0, size = {lo = 65536, hi = 0}, addr = 0, destructor = 0x55555
```

```
subpage = false, terminates = false, readable = true, ram = false, readonly = false, enabled = true, r
flush_coalesced_mmio = false, alias = 0x0, alias_offset = 0, priority = 0, may_overlap = false, subreg
  tqh_last = 0x55555698bf60}, subregions_link = {tqe_next = 0x0, tqe_prev = 0x0}, coalesced = {tqh_fir
name = 0x5555566f7220 "io", dirty_log_mask = 0 '00', ioeventfd_nb = 0, ioeventfds = 0x0, updateaddr =
```

回调流程

QEMU通过kvm_cpu_exec -> kvm_vcpu_ioctl(cpu, KVM_RUN, 0) 执行GUEST机CODE，当GUEST遇到IO等操作需要退出，会先在KVM里处理，KVM不能处理，kvm_vcpu_ioctl就返回，给QEMU处理，QEMU根据返回的run->exit_reason进行分派，比如PROT READ 0x71操作，退出时其exit_reason为KVM_EXIT_IO，kvm_handle_io里，根据direction判断是read还是write，根据read的长度，判断该回调哪个函数。比如0x71 read 1字节的时候，调用的是：
stb_p(ptr, cpu_inb(port));
stb_p是将第二个参数cpu_inb(port)的结果转换为一个字节大小赋值给第一个参数ptr所指内存。

cpu_inb (addr=addr@entry=113)
cpu_inb和cpu_inw和cpu_inl是一家人的三兄弟，长得极其神似，我们看cpu_inb，在他调用的ioport_read的时候，第一个参数叫index=0，这是和他的兄弟cpu_inw, cpu_inl区别开来的关键特征。本来这里应该没有index啥事的，但总有人偷懒不设置对应addr的handler，index就是对专门为这种懒人擦屁股，没有handler的时候，给选择一个默认handler，你大概也看明白了，就index的话，三兄弟的差别在于inb=0, inw=1, inl=2。

```
uint8_t cpu_inb(pio_addr_t addr)
{
    uint8_t val;
    val = ioport_read(0, addr);
    return val;
}
```

read 的address比较重要，例如rtc的0x71，index其实是用来选择默认handler的，当在对应的ioport_read_table里面没有注册函数的时候就根据index的值，分别选择readb, readw, readl来做默认操作。

```
static uint32_t ioport_read(int index, uint32_t address)
{
    static IOPortReadFunc * const default_func[3] = {
        default_ioport_readb,
        default_ioport_readw,
        default_ioport_readl
    };
    IOPortReadFunc *func = ioport_read_table[index][address];
    if (!func)
        func = default_func[index];
    /*
     func一般都是ioport_readb_thunk 关键在于ioport_opaque[address]这里面存放的是不同端口的IORange*
     这个ioport_opaque的每个值，都存储的该端口对应的IORange*
     当读写此端口的时候，就会找到之前注册的IORange回调，比如mc146818的IORange为
     (gdb) p *(IORange *)ioport_opaque[0x71]
     $22 = {ops = 0x7fca4a51c380, base = 112, len = 2}
    (gdb) p *(IORangeOps*)0x7fca4a51c380
    $25 = {read = 0x7fca49fe1190 <memory_region_iorange_read>, write = 0x7fca49fe1050 <memory_region_iorange_write
    destructor = 0x7fca49fdfcb0 <memory_region_iorange_destructor>}

    由于x70,0x71都是readb，所以在mc146818设备的时候，这个func其实为ioport_readb_thunk
(gdb) p ioport_read_table[0][0x70]
$67 = (IOPortReadFunc *) 0x5555557c6980 <ioport_readb_thunk>
(gdb) p ioport_read_table[0][0x71]
$68 = (IOPortReadFunc *) 0x5555557c6980 <ioport_readb_thunk>
    */
    return func(ioport_opaque[address], address);
}
```

ioport_readb_thunk (opaque=, addr=)
ioport_register里面，注册了对一个字节的ioport read handler为ioport_readb_thunk，其实这个函数非常简单
就是调用了ops->read的时候，将width设置为1，和ioport_readw_thunk，ioport_readl_thunk之类的就差一个width的区别
为什么要搞这么复杂？这是为了64K的read空间设计的回调，因为不同的offset位置，我们需要知道是应该调用readb还是readw还是readl。

```
static IOPortReadFunc *ioport_read_table[3][64 * 1024]
static uint32_t ioport_readb_thunk(void *opaque, uint32_t addr)
{
    IORange *ioport = opaque;
    uint64_t data;
    //read is memory_region_iorange_read when input char to ps/2 keyboard
    //比如，mc146818的时候，addr为x71，ioport->base为x71, ioport->len=2
    ioport->ops->read(ioport, addr - ioport->base, 1, &data);
    return data;
}
```

上面的回调为memory_region_iorange_read (iorange=0x55555680b1a0, offset=1, width=1, data=0x7fffec1fdc00)

```
static const MemoryRegionOps cmos_ops = {
    .read = cmos_ioport_read,
    .write = cmos_ioport_write,
    .impl = {
```

```
        .min_access_size = 1,
        .max_access_size = 1,
    },
    .endianness = DEVICE_LITTLE_ENDIAN,
};

static void memory_region_iorange_read(IORange *iorange,
                                       uint64_t offset,
                                       unsigned width,
                                       uint64_t *data)
{
    MemoryRegionIORange *mrio
        = container_of(iorange, MemoryRegionIORange, iorange);
    /*
    一段MemoryRegionIORange里包含了IORange iorange和MemoryRegion* mr
(gdb) p *mrio
$58 = {iorange = {ops = 0x555555d16200, base = 0x70, len = 2}, mr = 0x555556a17b80, offset = 0}
(gdb) p *mrio->iorange.ops
$59 = {read = 0x5555557ccf50 <memory_region_iorange_read>, write = 0x5555557cce10 <memory_region_iorange_write>,
  destructor = 0x5555557cba70 <memory_region_iorange_destructor>}
(gdb) p *mrio->mr
$60 = {ops = 0x555555d14ba0, opaque = 0x555556a17ae0, parent = 0x555556708930, size = {lo = 2, hi = 0}, addr = 112
  destructor = 0x5555557cb910 <memory_region_destructor_iomem>, ram_addr = 18446744073709551615, subpage = false,
  readonly = false, enabled = true, rom_device = false, warning_printed = false, flush_coalesced_mmio = false, ali
  may_overlap = false, subregions = {tqh_first = 0x0, tqh_last = 0x555556a17be8}, subregions_link = {tqe_next = 0x
  coalesced = {tqh_first = 0x0, tqh_last = 0x555556a17c08}, name = 0x55555680b300 "rtc", dirty_log_mask = 0 '00',
  updateaddr = 0, updateopaque = 0x0}
(gdb) p *mrio->mr->ops
$61 = {read = 0x55555579ea20 <cmos_ioport_read>, write = 0x55555579e040 <cmos_ioport_write>, endianness = DEVICE_L
    max_access_size = 0, unaligned = false, accepts = 0}, impl = {min_access_size = 1, max_access_size = 1, unalig
    read = {0, 0, 0}, write = {0, 0, 0}}}
    */
    MemoryRegion *mr = mrio->mr;

    //如果mrio还有offset，要加上这个偏移，这个offset其实是当成地址来用的，比如，我认为read 0x71应该在已有offset=1的基础上
    offset += mrio->offset;
    if (mr->ops->old_portio) {//对"mc146818rtc"已经没有old_portio的CALLBACK了，跳过
        const MemoryRegionPortio *mrp = find_portio(mr, offset - mrio->offset,
                                                    width, false);

        *data = ((uint64_t)1 << (width * 8)) - 1;
        if (mrp) {
            *data = mrp->read(mr->opaque, offset);
        } else if (width == 2) {
            mrp = find_portio(mr, offset - mrio->offset, 1, false);
            assert(mrp);
            *data = mrp->read(mr->opaque, offset) |
                    (mrp->read(mr->opaque, offset + 1) << 8);
        }
        return;
    }
    *data = 0;//这是read后的返回值存储区域，提前清零
    access_with_adjusted_size(offset, data, width,
                              mr->ops->impl.min_access_size, //这个min_access_size和max_access_size是在设置ops的时候
                              mr->ops->impl.max_access_size,
                              memory_region_read_accessor, mr);
}
```

从参数看，这里的access参数为memory_region_read_accessor，而这个value参数，用来存放read的返回值。接下来进入access_with_adjusted_size (addr=addr@entry=1, value=value@entry=0x7fffec1fdc00, size=1, access_size_min=, access_size_max=, access=access@entry=0x5555557cbf70 , opaque=opaque@entry=0x555556a17b80)，其access参数非常重要，继续回调的就是access。

```
static void access_with_adjusted_size(hwaddr addr,
                                      uint64_t *value,
                                      unsigned size,
                                      unsigned access_size_min,
                                      unsigned access_size_max,
                                      void (*access)(void *opaque,
                                      hwaddr addr,
                                      uint64_t *value,
                                      unsigned size,
                                      unsigned shift,
                                      uint64_t mask),
                                      void *opaque)
{
    uint64_t access_mask;
    unsigned access_size;
    unsigned i;

    if (!access_size_min) {
        access_size_min = 1;
    }
    if (!access_size_max) {
        access_size_max = 4;
    }
    access_size = MAX(MIN(size, access_size_max), access_size_min);//size其实在参数里面已经指定了，但是为了安全，要确保
    access_mask = -1ULL >> (64 - access_size * 8);//作为mask，对Read的几个mask，确保结果大小为预期大小
    for (i = 0; i < size; i += access_size) {
        /*
          最大返回结果其实只有sizeof(value) = 64bit，这里的设计是，每次取一个字节的返回结果
          但是access_size可以不为bit，比如read 0x100，假设read范围为bit，就是x100-0x104，access_size可以为，
          这样就分两步走，第一步Read 0x100-0x102返回个字节的结果，存储到value的低字节
          第二步Read 0x103-0x104返回个字节的结果，存储到value的高字节
         最后返回的value就存储了两次的Read值，只占用了bit，不会超过bit
        */
        access(opaque, addr + i, value, access_size, i * 8, access_mask);
    }
}
```

上面的access_with_adjusted_size的access参数其实就是memory_region_read_accessor，可以通过GDB打印出来：

```
memory_region_read_accessor (opaque=0x555556a17b80, addr=, value=0x7fffec1fdc00, size=1, shift=0, mask=2
(gdb) p *mr
$52 = {ops = 0x555555d14ba0, opaque = 0x555556a17ae0, parent = 0x555556708930, size = {lo = 2, hi = 0},
  destructor = 0x5555557cb910 , ram_addr = 18446744073709551615, subpage = false, terminates = true, rea
  readonly = false, enabled = true, rom_device = false, warning_printed = false, flush_coalesced_mmio =
  may_overlap = false, subregions = {tqh_first = 0x0, tqh_last = 0x555556a17be8}, subregions_link = {tqe
  coalesced = {tqh_first = 0x0, tqh_last = 0x555556a17c08}, name = 0x55555680b300 "rtc", dirty_log_mask
  updateaddr = 0, updateopaque = 0x0}
(gdb) p *mr->ops
$53 = {read = 0x55555579ea20 , write = 0x55555579e040 , endianness = DEVICE_LITTLE_ENDIAN, valid = {min_
    max_access_size = 0, unaligned = false, accepts = 0}, impl = {min_access_size = 1, max_access_size =
    read = {0, 0, 0}, write = {0, 0, 0}}}
```

绕了这么多，memory_region_read_accessor里的mr->ops->read终于到了我们注册的函数，如mc146818的cmos_ioport_read (opaque=0x555556a17ae0, addr=1, size=1)

```c
static void memory_region_read_accessor(void *opaque,
                                        hwaddr addr,
                                        uint64_t *value,
                                        unsigned size,
                                        unsigned shift,
                                        uint64_t mask)
{
    MemoryRegion *mr = opaque;
    uint64_t tmp;

    if (mr->flush_coalesced_mmio) {
        qemu_flush_coalesced_mmio_buffer();
    }
    tmp = mr->ops->read(mr->opaque, addr, size);
    *value |= (tmp & mask) << shift;
}
```

read/write回调函数，就是纯功能逻辑，比如mc146818主要干注入时钟，写入寄存器，读取日期等。

PDF下载

此条目是由 admin 发表在 未分类 分类目录的。将固定链接 [http://mnstory.net/2014/10/qemu-device-simulation/] 加入收藏夹。