

QEMU 设备模拟

转载 winceos 最后发布于2015-11-19 17:24:51 阅读数 4381 ☆ 收藏



1



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……）

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

```
1 static const MemoryRegionOps pit_ioport_ops = {
2     .read = pit_ioport_read,
3     .write = pit_ioport_write,
4     .impl = {
5         .min_access_size = 1,
6         .max_access_size = 1,
7     },
8     .endianness = DEVICE_LITTLE_ENDIAN,
9 };
10
11 static int pit_initfn(PITCommonState *pit)
12 {
13     PITChannelState *s;
14
15     s = &pit->channels[0];
16     /* the timer 0 is connected to an IRQ */
17     //这里有个irq_timer，用于qemu_set_irq提供中断注入
18     s->irq_timer = qemu_new_timer_ns(vm_clock, pit_irq_timer, s);
19     qdev_init_gpio_out(&pit->dev.qdev, &s->irq, 1);
20
21     memory_region_init_io(&pit->iports, &pit_ioport_ops, pit, "pit", 4);
22     qdev_init_gpio_in(&pit->dev.qdev, &pit_irq_control, 1);
23     return 0;
24 }
```

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;
```

```
1 typedef enum {
2     MODULE_INIT_BLOCK,
3     MODULE_INIT_MACHINE,
4     MODULE_INIT_QAPI,
5     MODULE_INIT_QOM,
6     MODULE_INIT_MAX
7 } 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_types) IDEDeviceClass //IDE硬盘 type_init(ide_register_types) ISADeviceClass //鼠标键盘 type_init(i8042_register_types), RTC时钟 type_init(pit_register) SysBusDeviceClass



举报



```
1 IDEDeviceClass //IDE硬盘或CD-ROM type_init(ide_register_types)
2 ISADeviceClass //鼠标键盘type_init(i8042_register_types), RTC时钟type_init(pit_register)
3 SysBusDeviceClass//MMIO IDE(IDE设备直接连接CPU bus而不是连接IDE controller)type_init(mmio_ide_register_types)
4 构
5 -> 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); \ }
```

```
1 #define type_init(function) module_init(function, MODULE_INIT_QOM)
2 #define module_init(function, type) \
3 staticvoid __attribute__((constructor))do_qemu_init_## function(void) { \
4 register_module_init(function,type);\
5 }
```

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); }
```

```
1 staticvoid __attribute__((constructor))do_qemu_init_pit_register_types(void)
2 {
3 register_module_init(pit_register_types,MODULE_INIT_QOM);
4 }
```

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util/module.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; l = find_type(type); QTAILQ_INSERT_TAIL(l, e, node); }
```

```
1 voidregister_module_init(void(*fn)(void),module_init_typetype)
2 {
3 ModuleEntry*e;
4 ModuleTypeList*l;
5 e=g_malloc0(sizeof(*e));
6 e->init=fn;//init指针被设置为fn
7 l=find_type(type);
8 QTAILQ_INSERT_TAIL(l,e,node);
9 }
```

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 }
```

```
1 voidmain()
2 {
3 module_call_init(MODULE_INIT_QOM);//初始化设备
4 qemu_add_opts//初始化默认选项
5 module_call_init(MODULE_INIT_MACHINE);//初始化机器类型
6 machine=find_default_machine();//这里对machine赋值，下面还会通过参数更改machine
7 vtp_script_execute(g_qemu_start_hook_path,g_fairsched_string,TYPE_START);//开机启动脚本的调用
8 深度分析启动参数
9 bdrv_init_with_whitelist->bdrv_init->module_call_init(MODULE_INIT_BLOCK);//初始化BLOCK设备
10 machine->init(&args);//初始化machine
11
12 qemu_run_machine_init_done_notifiers();//初始化成功回调通知
13 qemu_system_reset(VMRESET_SILENT);//system reset 启动运行
14 if(loadvm){
15 load_vmstate(loadvm);
16 }elseif(loadstate){
17 load_state_from_blockdev(loadstate);
18 }
19 }
```



举报



22 }



1



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(l, &module_init_type_list, next) { e = l->entry; module_call_init(e, type); } }
```

```
1 void module_call_init(module_init_type type)
2 {
3     ModuleTypeList *l;
4     ModuleEntry *e;
5     l = find_type(type);
6     QTAILQ_FOREACH(l, &module_init_type_list, next) {
7         e = l->entry; //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
8     }
9 }
```

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parent TypeInfo的class_size .instance_size = sizeof(PITCommonState), //分配实例的大小 .class_init = pit_class_init };
static void pit_register_types(void) { type_register_static(&pit_info); }
```

```
1 static const TypeInfo pit_info = {
2     .name = "isa-pit", //做为type_table的key
3     .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parent TypeInfo的class_size
4     .instance_size = sizeof(PITCommonState), //分配实例的大小
5     .class_init = pit_class_init, //初始化函数
6 };
7
8 static void pit_register_types(void)
9 {
10     type_register_static(&pit_info);
11 }
```

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qom/object.c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TypeInfo为VALUE，在以name为KEY回溯parent时需要这个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"

qom/object.c

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

```
1 static TypeInfo object_info = {
2     .name = "object",
3     .instance_size = sizeof(Object),
4     .instance_init = object_instance_init,
5     .abstract = true,
6 };
```

hw/qdev.c

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

```
1 static const TypeInfo device_type_info = {
2     .name = "device",
3     .parent = "object",
4     .instance_size = sizeof(DeviceState),
5     .instance_init = device_instance_init,
6     .instance_finalize = device_finalize,
7     .class_base_init = device_class_base_init,
8     .class_init = device_class_init,
9     .abstract = true,
10    .class_size = sizeof(DeviceClass),
11 };
```



举报



hw/isa-bus.c

```
1 staticconstTypeInfoisa_device_type_info={
2     .name="isa-device",
3     .parent="device",
4     .instance_size=sizeof(ISADevice),
5     .abstract=true,
6     .class_size=sizeof(ISADeviceClass),
7     .class_init=isa_device_class_init,
8 };
```



1



hw\i8254_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, };
```



```
1 staticconstTypeInfopit_common_type={
2     .name      ="pit-common",
3     .parent    ="isa-device",
4     .instance_size=sizeof(PITCommonState),
5     .class_size  =sizeof(PITCommonClass),
6     .class_init  =pit_common_class_init,
7     .abstract    =true,
8 };
```

hw\i8254.c

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

```
1 staticconstTypeInfopit_info={
2     .name      ="isa-pit",
3     .parent    ="pit-common",
4     .instance_size=sizeof(PITCommonState),
5     .class_init  =pit_class_initfn,
6 };
```

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]; };
```

```
1 structTypeImpl
2 {
3     constchar*name;
4     size_tclass_size;
5     size_tinstance_size;
6     void(*class_init)(ObjectClass*klass,void*data);
7     void(*class_base_init)(ObjectClass*klass,void*data);
8     void(*class_finalize)(ObjectClass*klass,void*data);
9     void*class_data;
10    void(*instance_init)(Object*obj);
11    void(*instance_finalize)(Object*obj);
12    boolabstract;
13    constchar*parent;
14    TypeImpl*parent_type;
15    ObjectClass*class;
16    intnum_interfaces;
17    InterfaceImplinterfaces[MAX_INTERFACES];
18 };
```

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 = 0x55555579b790, class_base_init = 0x55555579b790, class_finalize = 0, class_data = 0x0, instance_init = 0, instance_finalize = 0, abstract = false, parent = 0x5555566e5e50 "isa-c", parent_type = 0x5555566d8bd0, class = 0x555556a50e50, num_interfaces = 0, interfaces = {{typename = 0x0}} }
```

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。



举报



- 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 *vmstate_desc; /* Private to qdev / bus. */ qdev_initfn init; /* TODO remove, once users are converted to realize */ qdev_eventunplug; qdev_eventexit; const char *bus_type; } DeviceClass; typedef struct ISADeviceClass { DeviceClass parent_class; int (*init)(ISADevice *dev); } ISADeviceClass; typedef struct PITCommonClass { DeviceClass 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;
```

```
1 struct ObjectClass
2 {
3     /*< private >*/
4     Type type;
5     GSList *interfaces;
6     ObjectUnparent *unparent;
7 };
8 typedef struct DeviceClass {
9     /*< private >*/
10    ObjectClass parent_class;
11    /*< public >*/
12    const char *fw_name;
13    const char *desc;
14    Property *props;
15    int no_user;
16    /* callbacks */
17    void (*reset)(DeviceState *dev);
18    DeviceRealize realize;
19    DeviceUnrealize unrealize;
20    /* device state */
21    const struct VMStateDescription *vmstate_desc;
22    /* Private to qdev / bus. */
23    qdev_initfn init; /* TODO remove, once users are converted to realize */
24    qdev_eventunplug;
25    qdev_eventexit;
26    const char *bus_type;
27 } DeviceClass;
28 typedef struct ISADeviceClass {
29     DeviceClass parent_class;
30     int (*init)(ISADevice *dev);
31 } ISADeviceClass;
32 typedef struct PITCommonClass {
33     DeviceClass parent_class;
34     int (*init)(PITCommonState *s);
35     void (*set_channel_gate)(PITCommonState *s, PITChannelState *sc, int val);
36     void (*get_channel_info)(PITCommonState *s, PITChannelState *sc,
37         PITChannelInfo *info);
38     void (*pre_save)(PITCommonState *s);
39     void (*post_load)(PITCommonState *s);
40 } 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; DeviceState { /*< private >*/ Object parent_obj; /*< public >*/ const char *id; bool realized; QemuOpts *opts; int hotplugged; BusState *parent; 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; int alias_required_for_version; }; struct ISADevice { DeviceState qdev; uint32_t isa_irq[2]; int nirqs; int ioport_id; }; typedef struct PITCommonState { ISADevice dev; MemoryRegion ioports; uint32_t iobase; PITChannelState channels[3]; } PITCommonState;
```

```
1 struct Object
2 {
3     /*< private >*/
4     ObjectClass *class;
5     ObjectFree *free;
6     QTAILQ_HEAD(, ObjectProperty) properties;
```

```
9    };
10   struct DeviceState{
11       /*< private >*/
12       Objectparent_obj;
13       /*< public >*/
14       const char*id;
15       boolrealized;
16       QemuOpts*opts;
17       inthotplugged;
18       BusState*parent_bus;
19       intnum_gpio_out;
20       qemu_irq*gpio_out;
21       intnum_gpio_in;
22       qemu_irq*gpio_in;
23       QLIST_HEAD(,BusState)child_bus;
24       intnum_child_bus;
25       intinstance_id_alias;
26       intalias_required_for_version;
27   };
28   struct ISADevice{
29       DeviceStateqdev;
30       uint32_tisairq[2];
31       intnirqs;
32       intioport_id;
33   };
34   typedef struct PITCommonState{
35       ISADevicedev;
36       MemoryRegioniports;
37       uint32_ttiobase;
38       PITChannelStatechannels[3];
39   }PITCommonState;
```



1



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Object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, 就像你妈给你做衣服做裤腰带, 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 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
DEVICE(object_new_with_type(type_get_by_name(typename))) if (!dev) { return NULL; } if (!bus) { bus = sysbus_get_default(); } qdev_set_
bus); object_unref(OBJECT(dev)); return dev; } ObjectClass *object_class_by_name(const char *typename) { //之前在type_register_static中
TypeInfo.name, 例如"isa-pit"为keys的TypeImpl TypeImpl *type = type_get_by_name(typename); if (!type) { return NULL; } type_initialize(t
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 *pare
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"
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 = "
.parent = "isa-device", .instance_size = sizeof(PITCommonState), .class_size = sizeof(PITCommonClass), .class_init = pit_common_class_in
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
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
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 *type_get_b
>interfaces[i].typename); for (e = ti->class->interfaces; e; e = e->next) { TypeImpl *target_type = OBJECT_CLASS(e->data)->
if
(type_is_ancestor(target_type, ti)) { break; } } if (e) { continue; } type_initialize_interface(ti, ti->interfaces[i].typename); } ti-
->type = ti
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); } pa
type_get_
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_
这个函数主
```

```
1 DeviceState*qdev_try_create(BusState*bus,constchar*type)
2 {
3     DeviceState*dev;
4     //这个type为TypeInfo.name, 例如"isa-pit"
5     if(object_class_by_name(type)==NULL){
6         returnNULL;
7     }
8     //type_initialize完成后, object_new用来实例化一个instance
9     dev=DEVICE(object_new(type));// = DEVICE(object_new_with_type(type_get_by_name(typename)))
10    if(!dev){
11        returnNULL;
12    }
13    if(!bus){
14        bus=sysbus_get_default();
15    }
16    qdev_set_parent_bus(dev,bus);
17    object_unref(OBJECT(dev));
18    returndev;
19 }
20
21 ObjectClass*object_class_by_name(constchar*typename)
22 {
23     //之前在type_register_static的时候, 注册了TypeInfo.name, 例如"isa-pit"为key的TypeImpl
24     TypeImpl*type=type_get_by_name(typename);
25     if(!type){
26         returnNULL;
27     }
28     type_initialize(type);//这里面, 初始化class,
29     returntype->class;
30 }
31
32 //其实这个函数更应该叫做new_TypeInfo_class()
33 staticvoidtype_initialize(TypeImpl*ti)
34 {
35     TypeImpl*parent;
36     if(ti->class){
37         return;
38     }
39     /*
40     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
41     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
42     static const TypeInfo pit_common_type = {
43         .name      = "pit-common",
44         .parent     = "isa-device",
45         .instance_size = sizeof(PITCommonState),
46         .class_size  = sizeof(PITCommonClass),
47         .class_init   = pit_common_class_init,
48         .abstract     = true,
49     };
50     */
51     ti->class_size=type_class_get_size(ti);
52     ti->instance_size=type_object_get_size(ti);
53     ti->class=g_malloc0(ti->class_size);
54     parent=type_get_parent(ti);
55     if(parent){
56         //1, 保证parent初始化了
57         type_initialize(parent);
58         GSList*e;
59         inti;
60
61         //2, 将parent的class内容memcpy一份给自己的对应的parent区域
62         g_assert(parent->class_size<=ti->class_size);
63         memcpy(ti->class,parent->class,parent->class_size);
64
65         //3, 将parent里面的class的interfaces做一次深度复制, 复制给自己
66         for(e=parent->class->interfaces;e;e=e->next){
67             ObjectClass*iface=e->data;
68             type_initialize_interface(ti,object_class_get_name(iface));
```



1



举报




```
71 //4.如果本类型有自己的interfaces, 初始化
72 for(i=0;inum_interfaces;i++){
73     TypeImpl*t=type_get_by_name(ti->interfaces[i].typename);
74     for(e=ti->class->interfaces;e=e->next){
75         TypeImpl*target_type=OBJECT_CLASS(e->data)->type;
76         if(type_is_ancestor(target_type,t)){
77             break;
78         }
79     }
80     if(e){
81         continue;
82     }
83     type_initialize_interface(ti,ti->interfaces[i].typename);
84 }
85 }
86
87 ti->class->type=ti;
88 while(parent){
89     if(parent->class_base_init){
90         //回溯回调parent的class_base_init函数
91         parent->class_base_init(ti->class,ti->class_data);
92     }
93     parent=type_get_parent(parent);
94 }
95
96 if(ti->class_init){
97     /*
98     如果本类设置了class_init, 回调它, ti->class_data是一个void*的参数
99     比如"isa-pit"我们设置了pit_class_initfn
100     这个函数主要干啥? 主要填充class里的其他该填充的地方。
101     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)
102     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
103     */
104     ti->class_init(ti->class,ti->class_data);
105 }
106 }
```



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); 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; }

object_initialize_with_type(void *data, TypeImpl *type) { Object *obj = data; g_assert(type != NULL); type_initialize(type); g_assert(type->i sizeof(Object)); g_assert(type->abstract == false); memset(obj, 0, type->instance_size); obj->class = type->class; //instance的类型通过class指 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函数挂 instance的构造函数 }

```
1 Object*object_new_with_type(Typetype)
2 {
3     Object*obj;
4     g_assert(type!=NULL);
5     type_initialize(type);
6     obj=g_malloc(type->instance_size);//这个instance_size是初始化TypeInfo的时候设置的sizeof(PITCommonState)
7     object_initialize_with_type(obj,type);
8     obj->free=g_free;
9     returnobj;
10 }
11 voidobject_initialize_with_type(void*data,TypeImpl*type)
12 {
13     Object*obj=data;
14     g_assert(type!=NULL);
15     type_initialize(type);
16     g_assert(type->instance_size>=sizeof(Object));
17     g_assert(type->abstract==false);
18     memset(obj,0,type->instance_size);
19     obj->class=type->class;//instance的类型通过class指针指定
20     object_ref(obj);
21     QTAILQ_INIT(&obj->properties);
22     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的构造函数
23 }
```




```
13$33 = {name = 0x5555566e40b0 "interface", class_size = 32, instance_size = 0, class_init = 0, class_base_init = 0, class_finalize = 0, class_data = 0x0,
14 instance_init = 0, instance_finalize = 0, abstract = true, parent = 0x0, parent_type = 0x0, class = 0x0, num_interfaces = 0, interfaces = {{
15     typename = 0x0 }}
16 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"interface"), 那么要考
17这里不是, 且type->num_interfaces为0
18 */
19 if(type->num_interfaces&&type_is_ancestor(target_type,type_interface)){
20     intfound=0;
21     GSList*i;
22     for(i=class->interfaces;i;i->next){
23         ObjectClass*target_class=i->data;
24         if(type_is_ancestor(target_class->type,target_type)){
25             ret=target_class;
26             found++;
27         }
28     }
29     /* The match was ambiguous, don't allow a cast */
30     if(found>1){
31         ret=NULL;
32     }
33 }elseif(type_is_ancestor(type,target_type)){
34     /*
35     判断type="mc146818rtc"的祖先是否是target_type="isa-device",
36     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
37     */
38     ret=class;
39 }
40 returnret;
}
```

GDB显示其内部数据为:

(gdb) p *obj //Object *obj

\$11 = {class = 0x555556a50e50, free = 0x7ffff7424020, properties = {tqh_first = 0x555556a17da0, tqh_last = 0x555556a50fd0}, ref = 1, parent

(gdb) p *obj->class //ObjectClass * class

\$12 = {type = 0x5555566e5cb0, interfaces = 0x0, unparent = 0x555556b1ac0 }

(gdb) p *obj->class->type //struct TypeImpl * type

\$13 = {name = 0x5555566e5e30 "mc146818rtc", class_size = 128, instance_size = 664, class_init = 0x55555679b790, class_base_init = 0, class_finalize = 0, class_data = 0x0, instance_init = 0, instance_finalize = 0, abstract = false, parent = 0x5555566e5e50 "isa-device", parent_type = 0x5555566d8bd0, class = 0x555556a50e50, num_interfaces = 0, interfaces = {{typename = 0x0}} }

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PitCo应该类似如下所示:

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

```
1 classPITCommonClass:publicISADeviceClass{
2 public:
3     virtualintinit(PITCommonState*s)=0;
4 };
5
6 classISADevice:publicDeviceState{
7 public:
8     intnirqs;
9     intioport_id;
10 };
11
12 classPITCommonState:publicISADevice,publicPITCommonClass{
13     intinit(PITCommonState*s);
14 };
```

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好的状态, 所以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.pc_machine_init函数注册的第一个machine, 即:

```
qemu_register_machine(&pc_i440fx_machine_v1_4);
1 staticQEMUMachinepc_i440fx_machine_v1_4={
2     .name="pc-i440fx-1.4",
3     .alias="pc",
4     .desc="Standard PC (i440FX + PIIX, 1996)",
5     .init=pc_init_pci,
6     .max_cpus=255,
7     .is_default=1,
8     .default_machine_opts=KVM_MACHINE_OPTIONS,
9     DEFAULT_MACHINE_OPTIONS,
10 };
11 qemu_register_machine(&pc_i440fx_machine_v1_4);
```



1



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 - init1，这个关PC硬件：

```
static void pc_init1(MemoryRegion *system_memory, MemoryRegion *system_io, ram_addr_t ram_size, const char *boot_device, const ch
*kernel_filename, const char *kernel_cmdline, const char *initrd_filename, const char *cpu_model, int pci_enabled, int kvmclock_enable
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.am
(!xen_enabled()) { //ROM, BIOS, RAM相关初始化 fw_cfg = pc_memory_init(system_memory, kernel_filename, kernel_cmdline, initrd_filena
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,
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_
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 **rt
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_al
口，并口 //初始化vmouse ps2_mouse }
```

```
1 staticvoidpc_init1(MemoryRegion*system_memory,
2     MemoryRegion*system_io,
3     ram_addr_t ram_size,
4     constchar*boot_device,
5     constchar*kernel_filename,
6     constchar*kernel_cmdline,
7     constchar*initrd_filename,
8     constchar*cpu_model,
9     intpci_enabled,
10    intkvmclock_enabled)
11 {
12     //CPU类型初始化-> cpu_x86_init -> mce_init/qemu_init_vcpu，初始化VCPU
13     pc_cpus_init(cpu_model);
14     //初始化acpi_tables
15     pc_acpi_init("acpi-dsdt.aml");
16     if(!xen_enabled()){
17         //ROM, BIOS, RAM相关初始化
18         fw_cfg=pc_memory_init(system_memory,
19             kernel_filename,kernel_cmdline,initrd_filename,
20             below_4g_mem_size,above_4g_mem_size,
21             rom_memory,&ram_memory);
22     }
23     //IRQ，初始化
24     //VGA初始化
25     pc_vga_init(isa_bus,pci_enabled?pci_bus=NULL);
26     /* init basic PC hardware */
27     pc_basic_device_init(isa_bus,gsi,&rtc_state,&floppy,xen_enabled());//这里调用pit_init
28     //初始化网卡
29     pc_nic_init(isa_bus,pci_bus);
30     //初始化硬盘，音频设备
31     //初始化cmos数据，比如设置cmos rtc时钟，是否提供PS/2设备
32     pc_cmos_init(below_4g_mem_size,above_4g_mem_size,boot_device,
33         floppy,idebus[0],idebus[1],rtc_state);
34     //初始化USB
35     if(pci_enabled&&usb_enabled(false)){
36         pci_create_simple(pci_bus,piix3_devfn+2,"piix3-usb-uhci");
37     }
38 }
39
40 voidpc_basic_device_init(ISABus*isa_bus,qemu_irq*gsi,
```



举报



```
43     boolno_vmpport)
44 {
45     //初始化HPET
46     //初始化mc146818 rtc
47     //初始化i8042 PIT
48     pit=pit_init(isa_bus,0x40,pit_isa_irq,pit_alt_irq);
49     //初始化串口，并口
50     //初始化vmmouse ps2_mouse
51 }
```



1



接下来的流程是pit_init -> isa_create(bus, "isa-pit") -> qdev_create -> qdev_try_create，qdev_try_create的实现在前面已经讨论过了，如上图所示，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(), };
1 staticPropertymc146818rtc_properties[]={
2     DEFINE_PROP_INT32("base_year",RTCState,base_year,1980),
3     DEFINE_PROP_LOSTTICKPOLICY("lost_tick_policy",RTCState,
4     lost_tick_policy,LOST_TICK_DISCARD),
5     DEFINE_PROP_END_OF_LIST(),
6 };
```

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 = 0x5555556afc60, release = 0,
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 = 0x5555556af400, set = 0x5555556af990,
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 = 0x555556a50f10}}
(gdb) p *obj->properties.tqh_first.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);

```
1 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", (Object **)&dev->parent_bus, NULL);

```
1 object_property_add_bool(obj,"realized",
2     device_get_realized,device_set_realized,NULL);
3 bject_property_add_link(OBJECT(dev),"parent_bus",TYPE_BUS,
4     (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？

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_bool，此函数定



举报



```
== 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); } } 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 }

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name的属性
    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, dev, &s->io, base);
```

```
1 static const MemoryRegionOps cmos_ops = {
2     .read = cmos_ioport_read,
3     .write = cmos_ioport_write,
4     .impl = {
5         .min_access_size = 1,
6         .max_access_size = 1,
7     },
8     .endianness = DEVICE_LITTLE_ENDIAN,
9 };
10 void isa_register_ioport(ISADevice *dev, MemoryRegion *io, uint16_t start)
11 {
12     memory_region_add_subregion(isabus->address_space_io, start, io);
13     isa_init_ioport(dev, start);
14 }
15
16 memory_region_init_io(&s->io, &cmos_ops, s, "rtc", 2);
17 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->address_space_io 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已经存储到MemoryRegion的size里面了。

再看看isabus内容，有个更深入的性的认识：



举报



```
parent = 0x55555698d740}, parent = 0x55555698d740, name = 0x555556a14ff0 "isa.0", allow_hotplug = 0, max_index = 3, c
0x555556a50f30,
tqh_last = 0x555556a16450}, sibling = {le_next = 0x0, le_prev = 0x55555698d7b8}}, address_space_io = 0x555556708930, ir
(gdb) p *isabus->address_space_io
$9 = {ops = 0x0, opaque = 0x0, parent = 0x0, size = {lo = 65536, hi = 0}, addr = 0, destructor = 0x5555557cb8e0, ram_addr = 0
subpage = false, terminates = false, readable = true, ram = false, readonly = false, enabled = true, rom_device = false, warni
flush_coalesced_mmio = false, alias = 0x0, alias_offset = 0, priority = 0, may_overlap = false, subregions = {tqh_first = 0x555
tqh_last = 0x55555698bf60}, subregions_link = {tqe_next = 0x0, tqe_prev = 0x0}, coalesced = {tqh_first = 0x0, tqh_last = 0x5
name = 0x5555566f7220 "io", dirty_log_mask = 0 '\000', ioeventfd_nb = 0, ioeventfds = 0x0, updateaddr = 0, updateopaque
}
```

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...里处理，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...ason为KVI...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; }
```

```
1  uint8_tcpu_inb(pio_addr_taddr)
2  {
3      uint8_tval;
4      val=ioport_read(0,addr);
5      returnval;
6  }
```

read的地址比较重要，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, write = 0x7fca49fe1050, destructor = 0x7fca49fdfc0} 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 (gdb) p ioport_read_table[0][0x71] $68 = (IOPortReadFunc *) 0x5555557c6980 */ return func(ioport_opaque[address], address); }
```

```
1  staticuint32_tioport_read(intindex,uint32_taddress)
2  {
3      staticIOPortReadFunc*constdefault_func[3]={
4          default_ioport_readb,
5          default_ioport_readw,
6          default_ioport_readl
7      };
8      IOPortReadFunc*func=ioport_read_table[index][address];
9      if(!func)
10         func=default_func[index];
11     /*
12     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*
13     这个ioport_opaque的每个值，都存储的该端口对应的IORange*
14     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
15     (gdb) p *(IORange *)ioport_opaque[0x71]
16     $22 = {ops = 0x7fca4a51c380, base = 112, len = 2}
17     (gdb) p *(IORangeOps *)0x7fca4a51c380
18     $25 = {read = 0x7fca49fe1190, write = 0x7fca49fe1050,
19     destructor = 0x7fca49fdfcb0 }
20
21     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
22     (gdb) p ioport_read_table[0][0x70]
23     $67 = (IOPortReadFunc *) 0x5555557c6980
24     (gdb) p ioport_read_table[0][0x71]
25     $68 = (IOPortReadFunc *) 0x5555557c6980
26     */
27     returnfunc(ioport_opaque[address],address);
```



举报



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；

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

```
1 staticIOPortReadFunc*ioport_read_table[3][64*1024]
2 staticuint32_tioport_readb_thunk(void*opaque,uint32_taddr)
3 {
4     IORange*ioport=opaque;
5     uint64_tdata;
6     //read is memory_region_iorange_read when input char to ps/2 keyboard
7     //比如，mc146818的时候，addr为x71，ioport->base为x71，ioport->len=2
8     ioport->ops->read(ioport,addr-ioport->base,1,&data);
9     returndata;
10 }
```

上面的回调为memory_region_iorange_read (iorange=0x55555680b1a0, offset=1, width=1, data=0x7ffec1fdc00)

static const MemoryRegionOps cmos_ops = { .read = cmos_ioport_read, .write = cmos_ioport_write, .impl = { .min_access_size = 1, .max_a
.endianness = DEVICE_LITTLE_ENDIAN, }; static void memory_region_iorange_read(IORange *iorange, uint64_t offset, unsigned width, uin
MemoryRegionIORange *mrrio = container_of(iorange, MemoryRegionIORange, iorange); /* 一段MemoryRegionIORange里包含了IORange ic
MemoryRegion* mr (gdb) p *mrrio \$58 = {iorange = {ops = 0x555555d16200, base = 0x70, len = 2}, mr = 0x555556a17b80, offset = 0} (gdb) p *r
>iorange.ops \$59 = {read = 0x5555557ccf50, write = 0x5555557cce10, destructor = 0x5555557cba70} (gdb) p *mrrio->mr \$60 = {ops = 0x5555
opaque = 0x555556a17ae0, parent = 0x555556708930, size = {lo = 2, hi = 0}, addr = 112, destructor = 0x5555557cb910, ram_addr = 1844674
subpage = false, terminates = true, readable = true, ram = false, readonly = false, enabled = true, rom_device = false, warning_printed = fals
flush_coalesced_mmio = false, alias = 0x0, alias_offset = 0, priority = 0, may_overlap = false, subregions = {tqh_first = 0x0, tqh_last = 0x5555
subregions_link = {tqe_next = 0x555556a50d80, tqe_prev = 0x555556a1c1f8}, coalesced = {tqh_first = 0x0, tqh_last = 0x555556a17c08}, nam
0x55555680b300 "rtc", dirty_log_mask = 0 '\000', ioeventfd_nb = 0, ioeventfds = 0x0, updateaddr = 0, updateopaque = 0x0} (gdb) p *mrrio->
{read = 0x55555579ea20, write = 0x55555579e040, endianness = DEVICE_LITTLE_ENDIAN, valid = {min_access_size = 0, max_access_size =
false, accepts = 0}, impl = {min_access_size = 1, max_access_size = 1, unaligned = false}, old_portio = 0x0, old_mmio = { read = {0, 0, 0}, writ
MemoryRegion *mr = mrrio->mr; //如果mrrio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
上加上x70，但是他没加 offset += mrrio->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 - mrrio->offset, width, false); *data = ((uint64_t)1 << (width * 8)) - 1; if (mrp) { *data = mrp
>opaque, offset); } else if (width == 2) { mrp = find_portio(mr, offset - mrrio->offset, 1, false); assert(mrp); *data = mrp->read(mr->opaque, of
>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, v
>impl.min_access_size, //这个min_access_size和max_access_size是在设置ops的时候定义的，见cmos_ops mr->ops->impl.max_access_siz
memory_region_read_accessor, mr); }

```
1 staticconstMemoryRegionOpscmos_ops={
2     .read=cmos_ioport_read,
3     .write=cmos_ioport_write,
4     .impl={
5         .min_access_size=1,
6         .max_access_size=1,
7     },
8     .endianness=DEVICE_LITTLE_ENDIAN,
9 };
10
11 staticvoidmemory_region_iorange_read(IORange*iorange,
12     uint64_toffset,
13     unsignedwidth,
14     uint64_t*data)
15 {
16     MemoryRegionIORange*mrrio
17     =container_of(iorange,MemoryRegionIORange,iorange);
18     /*
19     一段MemoryRegionIORange里包含了IORange iorange和MemoryRegion* mr
20 (gdb) p *mrrio
21 $58 = {iorange = {ops = 0x555555d16200, base = 0x70, len = 2}, mr = 0x555556a17b80, offset = 0}
22 (gdb) p *mrrio->iorange.ops
23 $59 = {read = 0x5555557ccf50, write = 0x5555557cce10,
24     destructor = 0x5555557cba70 }
25 (gdb) p *mrrio->mr
26 $60 = {ops = 0x555555d14ba0, opaque = 0x555556a17ae0, parent = 0x555556708930, size = {lo = 2, hi = 0}, addr = 112,
```



```
29 may_overlap = false, subregions = {tqh_first = 0x0, tqh_last = 0x555556a17be8}, subregions_link = {tqe_next = 0x555556a50d80, tqe_pre
30 coalesced = {tqh_first = 0x0, tqh_last = 0x555556a17c08}, name = 0x55555680b300 "rtc", dirty_log_mask = 0 '\000', ioeventfd_nb = 0, ioe
31 updateaddr = 0, updateopaque = 0x0}
32 (gdb) p *mrio->mr->ops
33 $61 = {read = 0x5555579ea20, write = 0x5555579e040, endianness = DEVICE_LITTLE_ENDIAN, valid = {min_access_size = 0,
34 max_access_size = 0, unaligned = false, accepts = 0}, impl = {min_access_size = 1, max_access_size = 1, unaligned = false}, old_portio =
35 read = {0, 0, 0}, write = {0, 0, 0}}
36 */
37 MemoryRegion*mr=mrio->mr;
38
39 //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的基础上加上x70,
40 offset+=mrio->offset;
41 if(mr->ops->old_portio){//对"mc146818rtc"已经没有old_portio的CALLBACK了,跳过
42     constMemoryRegionPortio*mrp=find_portio(mr,offset-mrio->offset,
43         width,false);
44
45     *data=((uint64_t)1<<(width*8))-1;
46     if(mrp){
47         *data=mrp->read(mr->opaque,offset);
48     }elseif(width==2){
49         mrp=find_portio(mr,offset-mrio->offset,1,false);
50         assert(mrp);
51         *data=mrp->read(mr->opaque,offset)|
52             (mrp->read(mr->opaque,offset+1)<<8);
53     }
54     return;
55 }
56 *data=0;//这是read后的返回值存储区域,提前清零
57 access_with_adjusted_size(offset,data,width,
58     mr->ops->impl.min_access_size,//这个min_access_size和max_access_size是在设置ops的时候定义的,见cmos_ops
59     mr->ops->impl.max_access_size,
60     memory_region_read_accessor,mr);
61 }
```

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=0x7ffec1fdc00, size=1, access_size_min=, access_size_max=, access=access@entry=0x555557cbf70, 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 *opaque, hwaddr addr, uint64_t *value, unsigned size, unsigned shift, uint64_t mask), void *opaque) { uint64_t access_mask; 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_min), 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_size区间为[1,4]字节 access_mask = -1ULL >> (64 - access_size); 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, 这里一个字节的结果但是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 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就有值,只占用了bit,不会超过bit */ access(opaque, addr + i, value, access_size, i * 8, access_mask); }

```
1 staticvoidaccess_with_adjusted_size(hwaddraddr,
2     uint64_t*value,
3     unsignedsize,
4     unsignedaccess_size_min,
5     unsignedaccess_size_max,
6     void(*access)(void*opaque,
7     hwaddraddr,
8     uint64_t*value,
9     unsignedsize,
10    unsignedshift,
11    uint64_tmask),
12    void*opaque)
13 {
14     uint64_taccess_mask;
15     unsignedaccess_size;
16     unsignedi;
17
18     if(!access_size_min){
19         access_size_min=1;
20     }
21     if(!access_size_max){
22         access_size_max=4;
```

55556a1c1ff

s = 0x0,

1

```
25 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
26 for(i=0;i<size;i+=access_size){<="" div="" style="" word-wrap: break-word;">
27     /*
28     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
29     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，
30     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
31     第二步Read 0x103-0x104返回个字节的结果，存储到value的高字节
32     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
33     */
34     access(opaque,addr+i,value,access_size,i*8,access_mask);
35 }
36 }
```



1



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=0x7ffec1fdc00, size=1, shift=0, mask=255)

(gdb) p *mr

```
$52 = {ops = 0x555555d14ba0, opaque = 0x555556a17ae0, parent = 0x555556708930, size = {lo = 2, hi = 0}, addr = 112,
destructor = 0x5555557cb910, ram_addr = 18446744073709551615, subpage = false, terminates = true, readable = true, ram = false,
readonly = false, enabled = true, rom_device = false, warning_printed = false, flush_coalesced_mmio = false, alias = 0x0, alias_offset = 0, pr
may_overlap = false, subregions = {tqh_first = 0x0, tqh_last = 0x555556a17be8}, subregions_link = {tqe_next = 0x555556a50d80, tqe_prev =
0x555556a1c1f8},
coalesced = {tqh_first = 0x0, tqh_last = 0x555556a17c08}, name = 0x55555680b300 "rtc", dirty_log_mask = 0 '\000', ioeventfd_nb = 0, ioever
updateaddr = 0, updateopaque = 0x0}
```

(gdb) p *mr->ops

```
$53 = {read = 0x55555579ea20, write = 0x55555579e040, endianness = DEVICE_LITTLE_ENDIAN, valid = {min_access_size = 0,
max_access_size = 0, unaligned = false, accepts = 0}, impl = {min_access_size = 1, max_access_size = 1, unaligned = false}, old_portio = 0x0,
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=0x555556a17b80, addr=1, size=1)

```
static void memory_region_read_accessor(void *opaque, hwaddr addr, uint64_t *value, unsigned size, unsigned shift, uint64_t mask) { Me
= opaque; uint64_t tmp; if (mr->flush_coalesced_mmio) { qemu_flush_coalesced_mmio_buffer(); } tmp = mr->ops->read(mr->opaque, add
(tmp & mask) << shift; }
```

```
1 static void memory_region_read_accessor(void *opaque,
2     hwaddr addr,
3     uint64_t *value,
4     unsigned size,
5     unsigned shift,
6     uint64_t mask)
7 {
8     MemoryRegion *mr = opaque;
9     uint64_t tmp;
10
11     if (mr->flush_coalesced_mmio) {
12         qemu_flush_coalesced_mmio_buffer();
13     }
14     tmp = mr->ops->read(mr->opaque, addr, size);
15     *value |= (tmp & mask) << shift; <="" div="" style="" word-wrap: break-word;">
16 }
```

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

👍 点赞 1 ☆ 收藏 ➦ 分享 ...



winceos

发布了52 篇原创文章 · 获赞 14 · 访问量 31万+



开发网站,电子商务系统设计,推广+网站制作

网站建设开发



举报



qemu2 machine的注册和的选择

在qemu里面，machine代表一台要虚拟的硬件机器，那么qemu是如何注册和选择机器的？我们今天就来分析一下…

博文

来自: [woai11012](#)

imx6ull-qemu 裸机教程1: GPIO,IOMUX,I2C

无意间搜到了韦东山老师的6ul网站，上面有一个6ul的qemu仿真器，下载下来用了用，非常好用，有UI，比原装的…

博文

来自: [u01128071](#)

qemu无界面启动，并重定向输出到终端

qemu-system-x86_64 -kernel bzImage -initrd /mnt/rootfs.cpio.gz /dev/zero -m 2G -nographic -append …

博文

来自: [weixin_34](#)

qemu QOM(qemu object model)和设备模拟

本文所用qemu为1.5版本的，不是android emulator的。之前几篇文章介绍的都是android emulator中的设备模拟…

博文

来自: [ayu_ag的专栏](#)



传统ERP已经过时，2019流行的ERP系统是这一款！

主流erp系统

虚拟机libvirt os machine取值

阅读数 2226

nova代码里获取libvirt里 os type 的machine的，如果这里获取和qemu不一致，会报如下错误：libvirtError: intern…

博文

来自: [wllabs的专栏](#)

KVM虚拟机和QEMU（命令行选项）

阅读数 2万+

KVM安装RHEL/Fedora/CentOSyum install bridge-utils kvmbridge-utils是网卡桥接工具，示例1：Redhat系统KV…

博文

来自: [少帅的天空](#)

学习qemu相关网址

阅读数 106

<https://blog.csdn.net/zhqh100/article/details/51173275><https://www.bennee.com/~alex/blog/2014/05/09/r...>

博文

来自: [Mouse的专栏](#)

qemu参数解析

阅读数 4516

代码版本： qemu1.5static QemuOptsList *vm_config_groups[32]; qemu_add_opts(&qemu_drive_opts); qemu_…

博文

来自: [ayu_ag的专栏](#)

QEMU使用简介

阅读数 1万+

QEMU使用简介。

博文

来自: [jiangwei0512的博客](#)

QEMU 设备模拟_C/C++_万能的终端和网络-CSDN博客

...可以模拟任何硬件设备的模拟器_weixin_34087503的博客-CSDN博客



开发网站,电子商务系统设计,推广+网站制作

网站建设开发

QEMU测试环境搭建

阅读数 402

本人的github仓库：<https://github.com/rikeyone/qemu.git>仓库中集成了整个QEMU环境，包含install、build、st…

博文

来自: [程序猿的日常干货](#)

Android Porting and Qemu - 万能的终端和网络 - CSDN博客

KVM CPU虚拟化 - 万能的终端和网络 - CSDN博客

Qemu 简述

阅读数 213

Qemu 架构Qemu 是纯软件实现的虚拟化模拟器，几乎可以模拟任何硬件设备，我们最熟悉的的就是能够模拟一台能够…

博文

来自: [weixin_33921089](#)



TangGeeA

130篇文章

排名:千里之外

关注



FRAWSCCC

22篇文章

排名:千里之外

关注



weixin_34221073

4659篇文章

排名:千里之外

关注



ayu_ag

1篇文章

排名:千里之外

关注

Versatile Express support in QEMU - 万能的终端和网络 - CSDN博客

QEMU 4.0.0 发布，几乎可以模拟任何硬件设备的模拟器

QEMU4.0.0发布了，此版本更新亮点包括：ARM：实现了一批ARMv8.X的扩展，包括SB、PredInv、HPD、LOR、F...

博文 来自： [weixin_33](#)



1

445



199

2.1 Qemu用户态 Machine与cpu管理

本节主要分析PC机在Qemu中的构成结构(QEMU的对象模型)，特别是CPU的相关结构

博文 来自： [wanthelpin](#)

Android Porting and Qemu_移动开发_万能的终端和网络-CSDN博客

如何在模拟器中使用串口上的设备？

1、我编译了Gingerbread的代码，make及make sdk，生成了自己的模拟器。模拟器可以正常运行。模拟器运行时使用 /d...



做一个小程序大概要多少钱

小程序开发多少钱

qemu-system-i386模拟的x86系统硬件架构是怎样的？比如用的什么类型的串口，硬件怎么连接的？

本人想用qemu(2.5.0)调试linux-2.6.10内核，想知道qemu模拟出来的硬件的拓扑结构，从qemu文档得知使用的是i440fx/...

问答

【转载】qemu源码分析-添加默认虚拟设备（以串口为例）

阅读量 278

在vl.c中的main函数： if (display_type == DT_NOGRAPHIC) { if (default_parallel) add_devic...

博文 来自： [Ubuntu 14.04配置](#)

emulator @模拟器名称 -qemu -serial COM3 关于模拟器与串口通信的连接

阅读量 71

客户端的串口通信已搭建成功，具体根据网上提供的开源SerialPortAPI来搭建串口参考文章：http://lpcjrflsa.iteye.c...

博文 来自： [weixin_30627381](#)

Xen/KVM中解决鼠标移动问题 <VNC>

阅读量 1222

原文：http://smilejay.com/2012/06/xen-kvm-cursor-movement/听同事提起，某云计算公司前阵子在线上云服务...

博文 来自： [lianliange85的专栏](#)

在Qemu/KVM下虚拟Windows XP中的鼠标位置偏移问题

阅读量 8344

在Qemu/KVM中虚拟一个windows xp操作系统，发现用VNC（使用kvm -vnc :1参数启动qemu的VNC）登录windo...

博文 来自： [Be the best myself](#)

kvm qemu键盘鼠标问题

阅读量 694

qemu, kvm, kqemu的键盘方向键问题解决 可能是因为我使用的非标准键盘或者usb键盘的原因，我用qemu和kvm...

博文 来自： [hypercube2的专栏](#)

Qemu&KVM 创建虚拟机之第一篇（5） QEMU Machine Protocol

阅读量 21

QEMU Machine ProtocolThe QEMU Machine Protocol (QMP) is aJSON-based protocol which allows applicatio...

博文 来自： [weixin_34318272](#)

Qemu Vhost Block架构分析（下）

阅读量 3314

一. 概述在上半部已经将VirtIO-blk

博文 来自： [u012377031的专栏](#)

GPIO模拟I2C程序实现

阅读量 1万+

GPIO模拟I2C程序实现.I2C是由Philips公司发明的一种串行数据通信协议，仅使用两根信号线：SerialClock（简称S...

博文 来自： [zhenwenxian的专栏](#)

GPIO模拟串口

阅读量 2317

模拟串口 串口默认的1byte数据格式为（暂不考虑校验位）：1bit起始位+8bit数据位+1bit停止位，起始位(S)为...

博文 来自： [li13158的博客](#)

使用 qemu 来模拟 nvme 设备

阅读量 715

使用 qemu 模拟 nvme 设备，本篇可以参考。引用本文请注明出处: https://blog.csdn.net/Hello_NB1/article/detai...

博文 来自： [Hello_NB1的博客](#)

qemu intel i6300esb watchdog虚拟外设分析

阅读量 317

本文介绍qemu是如何模拟和使用intel 6300esb芯片组的watchdog功能的

博文 来自： [daxiatou的专栏](#)

QEMU PCIe设备实现

PCIe 设备虚拟化QEMU中的实现 包括处理中断的硬件以及Linux如何响应和处理终端。技术分析分享



10

举报

下载



做移动端设备的开发测试同学都知道，兼容性是个很头疼的问题，测试或者复现兼容问题时借设备也是很痛苦的事情…

博文 来自： 简言

在qemu中增加pci设备并用linux驱动验证

声明本文主要针对x86架构进行说明。使用的qemu版本是：qemu-kvm-1.2.0-rc21)PCI结构简介每个PCI设备都有一…

博文 来自： XscKernel

Linux利器：QEMU！用它模拟开发板能替代真开发板？

QEMU，搞嵌入式开发的一定不陌生，最近各大群里都讨论疯了，说它是Linux利器一点也不夸张。它是一款知名的…

博文 来自： chungle201

qemu问题

kvm的linux已成功用qemu安装了windows系统。 windows也已经用了一段时间，现在想通过qemu引导进入windows系…

QEMU设备模拟

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…

博文 来自： tycoon的专

5.4 QEMU block & AIO

本节通过ide磁盘读写的情景，分析block driver和底层Aio的工作原理

博文 来自： wanthehelping的博文

QEMU虚拟网卡设备的创建流程

基于qemu-kvm-0.12.1.2-2.160.el6_1.8.src.rpm虚拟网卡类型为virtio-net-pcivirtio网卡设备对应的命令行参数为 -d…

博文 来自： sdulibh的专栏

QEMU建模之设备创建总体流程

(这里的设备创建以中断控制器即openpic为例)1.main函数之前执行type_init1> 在vl.c文件的main函数执行前会先执…

博文 来自： sinat_38205774的…

qemu 中是怎么模拟的新的设备

kvm_cpu_exec 和 d e m o 中演示的一样转载于:https://www.cnblogs.com/honpey/p/8063875.html…

博文 来自： weixin_30244681…

从零使用qemu模拟器搭建arm运行环境

本文从零开始介绍如何搭建qemu+arm的运行环境

博文 来自： 海枫的专栏

KVM浅析&基于Qemu创建Guest OS的测试

《KVM虚拟化技术：实战与原理解析》作者写作过程草稿连载http://smilejay.com/kvm_theory_practice/ KVM简介…

博文 来自： Celeste7777的博客

KVM中的网络IO设备虚拟化方式

在KVM虚拟化的架构里，对CPU的虚拟化采用的是硬件辅助的方式（Intel VT-x，AMD-V），效率比较高，内存的虚…

博文 来自： Summer的小屋

Java C语言 Python C++ C# Visual Basic .NET JavaScript PHP SQL Go语言 R语言 Assembly language Swift Ruby
MATLAB PL/SQL Perl Visual Basic Objective-C Delphi/Object Pascal Unity3D



winceos

TA的个人主页>

原创 粉丝 获赞 评论 访问
52 22 14 18 31万+

等级: 博客 S 周排名: 16万+
积分: 2655 总排名: 3万+

关注

私信



举报



最新文章

KVM CPU虚拟化

ernel 3.10内核源码分析--KVM相关--虚拟机运行

kernel 3.10代码分析--KVM-KVM_SET_USER_MEMORY_REGION流程

KVM vCPU创建过程

kernel 3.10代码分析--KVM相关--虚拟机创建

分类专栏

	SELinux/Android	6篇
	Linux	14篇
	Android	33篇
	网站开发	5篇
	magento	1篇

展开

归档

2015年11月	7篇
2015年7月	4篇
2015年1月	3篇
2014年12月	1篇
2014年8月	2篇
2014年7月	19篇
2014年6月	7篇
2014年5月	3篇

展开

热门文章

Git撤销git commit 但是未git push的修改
阅读数 88446

手动为Android 4.x 手机添加自己的根证书 (CA 证书)
阅读数 19633

使用git clone error: RPC failed
阅读数 18507

Repo 详解
阅读数 14097

tar 压缩打包时排除或忽略某个子目录或文件
阅读数 11263

最新评论

Git撤销git commit 但...

weixin_41276238: 1.选择要撤销的commit上一个的commitId, 2.最好日常使用使用--soft 选项 ...



1



举报



????????????????????? ...

使用git cherry-pick...

weixin_45989618:

????????????????????? ...

Git撤销git commit但...

weixin_43464805: commit_id是撤销commit的前一个id

Git撤销git commit但...

weixin_43464805: 收藏了.2019.12.24

🗨️ QQ客服

✉️ kefu@csdn.net

🗣️ 客服论坛

☎️ 400-660-0108

工作时间 8:30-22:00

关于我们 招聘 广告服务 网站地图

京ICP备19004658号 经营性网站备案信息

🇨🇳 公安备案号 11010502030143

©1999-2020 北京创新乐知网络技术有限公司
网络110报警服务

北京互联网违法和不良信息举报中心

中国互联网举报中心 家长监护

版权与免责声明 版权申诉



1



举报

