

50 | 计算虚拟化之CPU（上）：如何复用集团的人力资源？

2019-07-22 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 18:23 大小 16.84M



上一节，我们讲了一下虚拟化的基本原理，以及 qemu、kvm 之间的关系。这一节，我们就来看一下，用户态的 qemu 和内核态的 kvm 如何一起协作，来创建虚拟机，实现 CPU 和内存虚拟化。

这里是上一节我们讲的 qemu 启动时候的命令。


复制代码

```
1 qemu-system-x86_64 -enable-kvm -name ubuntutest -m 2048 -hda ubuntutest.qcow2 -vnc :19
```

接下来，我们在[这里下载](#)qemu 的代码。qemu 的 main 函数在 vl.c 下面。这是一个非常非常长的函数，我们来慢慢地解析它。

1. 初始化所有的 Module


第一步，初始化所有的 Module，调用下面的函数。

 复制代码

```
1 module_call_init(MODULE_INIT_QOM);
```

上一节我们讲过，qemu 作为中间人其实挺累的，对上面的虚拟机需要模拟各种各样的外部设备。当虚拟机真的要使用物理资源的时候，对下面的物理机上的资源要进行请求，所以它的工作模式有点儿类似操作系统对接驱动。驱动要符合一定的格式，才能算操作系统的一个模块。同理，qemu 为了模拟各种各样的设备，也需要管理各种各样的模块，这些模块也需要符合一定的格式。

定义一个 qemu 模块会调用 type_init。例如，kvm 的模块要在 accel/kvm/kvm-all.c 文件里面实现。在这个文件里面，有一行下面的代码：

 复制代码


```
1 type_init(kvm_type_init);
2
3 #define type_init(function) module_init(function, MODULE_INIT_QOM)
4
5 #define module_init(function, type) \
6 static void __attribute__((constructor)) do_qemu_init_ ## function(void) \
7 { \
8     register_module_init(function, type); \
9 }
10
11 void register_module_init(void (*fn)(void), module_init_type type)
12 {
13     ModuleEntry *e;
14     ModuleTypeList *l;
15
16     e = g_malloc0(sizeof(*e));
17     e->init = fn;
18     e->type = type;
19
20     l = find_type(type);
21
22     QTAILQ_INSERT_TAIL(l, e, node);
23 }
```

从代码里面的定义我们可以看出来，`type_init` 后面的参数是一个函数，调用 `type_init` 就相当于调用 `module_init`，在这里函数就是 `kvm_type_init`，类型就是 `MODULE_INIT_QOM`。是不是感觉和驱动有点儿像？

`module_init` 最终要调用 `register_module_init`。属于 `MODULE_INIT_QOM` 这种类型的，有一个 `Module` 列表 `ModuleTypeList`，列表里面是一项一项的 `ModuleEntry`。KVM 就是其中一项，并且会初始化每一项的 `init` 函数为参数表示的函数 `fn`，也即 KVM 这个 `module` 的 `init` 函数就是 `kvm_type_init`。

当然，`MODULE_INIT_QOM` 这种类型会有很多很多的 `module`，从后面的代码我们可以看到，所有调用 `type_init` 的地方都注册了一个 `MODULE_INIT_QOM` 类型的 `Module`。


了解了 `Module` 的注册机制，我们继续回到 `main` 函数中 `module_call_init` 的调用。

 复制代码

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

在 `module_call_init` 中，我们会找到 `MODULE_INIT_QOM` 这种类型对应的 `ModuleTypeList`，找出列表中所有的 `ModuleEntry`，然后调用每个 `ModuleEntry` 的 `init` 函数。这里需要注意的是，在 `module_call_init` 调用的这一步，所有 `Module` 的 `init` 函数都已经被调用过了。

后面我们会看到很多的 `Module`，当你看到它们的时候，你需要意识到，它的 `init` 函数在这里也被调用过了。这里我们还是以对于 `kvm` 这个 `module` 为例子，看看它的 `init` 函数都做了哪些事情。你会发现，其实它调用的是 `kvm_type_init`。

 复制代码

```
1 static void kvm_type_init(void)
2 {
```

```

3     type_register_static(&kvm_accel_type);
4 }
5
6 TypeImpl *type_register_static(const TypeInfo *info)
7 {
8     return type_register(info);
9 }
10
11 TypeImpl *type_register(const TypeInfo *info)
12 {
13     assert(info->parent);
14     return type_register_internal(info);
15 }
16
17 static TypeImpl *type_register_internal(const TypeInfo *info)
18 {
19     TypeImpl *ti;
20     ti = type_new(info);
21
22     type_table_add(ti);
23     return ti;
24 }
25
26 static TypeImpl *type_new(const TypeInfo *info)
27 {
28     TypeImpl *ti = g_malloc0(sizeof(*ti));
29     int i;
30
31     if (type_table_lookup(info->name) != NULL) {
32     }
33
34     ti->name = g_strdup(info->name);
35     ti->parent = g_strdup(info->parent);
36
37     ti->class_size = info->class_size;
38     ti->instance_size = info->instance_size;
39
40     ti->class_init = info->class_init;
41     ti->class_base_init = info->class_base_init;
42     ti->class_data = info->class_data;
43
44     ti->instance_init = info->instance_init;
45     ti->instance_post_init = info->instance_post_init;
46     ti->instance_finalize = info->instance_finalize;
47
48     ti->abstract = info->abstract;
49
50     for (i = 0; info->interfaces && info->interfaces[i].type; i++) {
51         ti->interfaces[i].typename = g_strdup(info->interfaces[i].type);
52     }
53     ti->num_interfaces = i;
54

```

```

55     return ti;
56 }
57
58 static void type_table_add(TypeImpl *ti)
59 {
60     assert(!enumerating_types);
61     g_hash_table_insert(type_table_get(), (void *)ti->name, ti);
62 }
63
64 static GHashTable *type_table_get(void)
65 {
66     static GHashTable *type_table;
67
68     if (type_table == NULL) {
69         type_table = g_hash_table_new(g_str_hash, g_str_equal);
70     }
71
72     return type_table;
73 }
74
75 static const TypeInfo kvm_accel_type = {
76     .name = TYPE_KVM_ACCEL,
77     .parent = TYPE_ACCEL,
78     .class_init = kvm_accel_class_init,
79     .instance_size = sizeof(KVMState),
80 };

```

每一个 Module 既然要模拟某种设备，那应该定义一种类型 `TypeImpl` 来表示这些设备，这其实是一种面向对象编程的思路，只不过这里用的是纯 C 语言的实现，所以需要变相实现一下类和对象。

`kvm_type_init` 会注册 `kvm_accel_type`，定义上面的代码，我们可以认为这样动态定义了一个类。这个类的名字是 `TYPE_KVM_ACCEL`，这个类有父类 `TYPE_ACCEL`，这个类的初始化应该调用函数 `kvm_accel_class_init`（看，这里已经直接叫类 `class` 了）。如果用这个类声明一个对象，对象的大小应该是 `instance_size`。是不是有点儿 Java 语言反射的意思，根据一些名称的定义，一个类就定义好了。

这里的调用链为：`kvm_type_init->type_register_static->type_register->type_register_internal`。

在 `type_register_internal` 中，我们会根据 `kvm_accel_type` 这个 `TypeInfo`，创建一个 `TypeImpl` 来表示这个新注册的类，也就是说，`TypeImpl` 才是我们想要声明的那个

class。在 qemu 里面，有一个全局的哈希表 type_table，用来存放所有定义的类。在 type_new 里面，我们先从全局表里面根据名字找这个类。如果找到，说明这个类曾经被注册过，就报错；如果没有找到，说明这是一个新的类，则将 TypeInfo 里面信息填到 TypeImpl 里面。type_table_add 会将这个类注册到全局的表里面。到这里，我们注意，class_init 还没有被调用，也即这个类现在还处于纸面的状态。

这点更加像 Java 的反射机制了。在 Java 里面，对于一个类，首先我们写代码的时候要写一个 class xxx 的定义，编译好就放在.class 文件中，这也是出于纸面的状态。然后，Java 会有一个 Class 对象，用于读取和表示这个纸面上的 class xxx，可以生成真正的对象。

相同的过程在后面的代码中我们也可以看到，class_init 会生成 XXXClass，就相当于 Java 里面的 Class 对象，TypeImpl 还会有一个 instance_init 函数，相当于构造函数，用于根据 XXXClass 生成 Object，这就相当于 Java 反射里面最终创建的对象。和构造函数对应的还有 instance_finalize，相当于析构函数。

这一套反射机制放在 qom 文件夹下面，全称 QEMU Object Model，也即用 C 实现了一套面向对象的反射机制。

说完了初始化 Module，我们还回到 main 函数接着分析。


2. 解析 qemu 的命令行

第二步我们就要开始解析 qemu 的命令行了。qemu 的命令行解析，就是下面这样一长串。还记得咱们自己写过一个解析命令行参数的程序吗？这里的 opts 是差不多的意思。

 复制代码

```
1  qemu_add_opts(&qemu_drive_opts);
2  qemu_add_opts(&qemu_chardev_opts);
3  qemu_add_opts(&qemu_device_opts);
4  qemu_add_opts(&qemu_netdev_opts);
5  qemu_add_opts(&qemu_nic_opts);
6  qemu_add_opts(&qemu_net_opts);
7  qemu_add_opts(&qemu_rtc_opts);
8  qemu_add_opts(&qemu_machine_opts);
9  qemu_add_opts(&qemu_accel_opts);
10 qemu_add_opts(&qemu_mem_opts);
11 qemu_add_opts(&qemu_smp_opts);
12 qemu_add_opts(&qemu_boot_opts);
13 qemu_add_opts(&qemu_name_opts);
14 qemu_add_opts(&qemu_numa_opts);
```


为什么有这么多的 opts 呢？这是因为，我们上一节给的参数都是简单的参数，实际运行中创建的 kvm 参数会复杂 N 倍。这里我们贴一个开源云平台软件 OpenStack 创建出来的 KVM 的参数，如下所示。不要被吓坏，你不需要全部看懂，只需要看懂一部分就行了。具体我来给你解析。

 复制代码

```
1 qemu-system-x86_64
2 -enable-kvm
3 -name instance-00000024
4 -machine pc-i440fx-trusty,accel=kvm,usb=off
5 -cpu SandyBridge,+erms,+smep,+fsgsbase,+pdpe1gb,+rdrand,+f16c,+osxsave,+dca,+pcid,+pdc
6 -m 2048
7 -smp 1,sockets=1,cores=1,threads=1
8 .....
9 -rtc base=utc,driftfix=slew
10 -drive file=/var/lib/nova/instances/1f8e6f7e-5a70-4780-89c1-464dc0e7f308/disk,if=none,i
11 -device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x4,drive=drive-virtio-disk0,id=virtio-d:
12 -netdev tap,fd=32,id=hostnet0,vhost=on,vhostfd=37
13 -device virtio-net-pci,netdev=hostnet0,id=net0,mac=fa:16:3e:d1:2d:99,bus=pci.0,addr=0x3
14 -chardev file,id=charserial0,path=/var/lib/nova/instances/1f8e6f7e-5a70-4780-89c1-464dc0
15 -vnc 0.0.0.0:12
16 -device cirrus-vga,id=video0,bus=pci.0,addr=0x2
```

-enable-kvm：表示启用硬件辅助虚拟化。

-name instance-00000024：表示虚拟机的名称。

-machine pc-i440fx-trusty,accel=kvm,usb=off：machine 是什么呢？其实就是计算机体系结构。不知道什么是体系结构的话，可以订阅极客时间的另一个专栏《深入浅出计算机组成原理》。

qemu 会模拟多种体系结构，常用的有普通 PC 机，也即 x86 的 32 位或者 64 位的体系结构、Mac 电脑 PowerPC 的体系结构、Sun 的体系结构、MIPS 的体系结构，精简指令集。如果使用 KVM hardware-assisted virtualization，也即 BIOS 中 VD-T 是打开的，则参数中 accel=kvm。如果不使用 hardware-assisted virtualization，用的是纯模拟，则有参数 accel = tcg，-no-kvm。

-cpu

SandyBridge,+erms,+smep,+fsgsbase,+pdpe1gb,+rdrand,+f16c,+osxsave,+dca,+

pcid,+pdcml,+xtpr,+tm2,+est,+smx,+vmx,+ds_cpl,+monitor,+dtes64,+pbe,+tm,+ht,+ss,+acpi,+ds,+vme : 表示设置 CPU , SandyBridge 是 Intel 处理器 , 后面的加号都是添加的 CPU 的参数 , 这些参数会显示在 /proc/cpuinfo 里面。

-m 2048 : 表示内存。

-smp 1,sockets=1,cores=1,threads=1 : SMP 我们解析过 , 叫对称多处理器 , 和 NUMA 对应。qemu 仿真了一个具有 1 个 vcpu , 一个 socket , 一个 core , 一个 threads 的处理器。

socket、core、threads 是什么概念呢 ? socket 就是主板上插 cpu 的槽的数目 , 也即常说的 “路” , core 就是我们平时说的 “核” , 即双核、4 核等。thread 就是每个 core 的硬件线程数 , 即超线程。举个具体的例子 , 某个服务器是 : 2 路 4 核超线程 (一般默认为 2 个线程) , 通过 cat /proc/cpuinfo , 我们看到的是 242=16 个 processor , 很多人也习惯成为 16 核了。

-rtc base=utc,driftfix=slew : 表示系统时间由参数 -rtc 指定。

-device cirrus-vga,id=video0,bus=pci.0,addr=0x2 : 表示显示器用参数 -vga 设置 , 默认为 cirrus , 它模拟了 CL-GD5446PCI VGA card。

有关网卡 , 使用 -net 参数和 -device。

从 HOST 角度 : -netdev tap,fd=32,id=hostnet0,vhost=on,vhostfd=37。

从 GUEST 角度 : -device virtio-net-pci,netdev=hostnet0,id=net0,mac=fa:16:3e:d1:2d:99,bus=pci.0,addr=0x3。

有关硬盘 , 使用 -hda -hdb , 或者使用 -drive 和 -device。

从 HOST 角度 : -drive file=/var/lib/nova/instances/1f8e6f7e-5a70-4780-89c1-464dc0e7f308/disk,if=none,id=drive-virtio-disk0,format=qcow2,cache=none

从 GUEST 角度 : -device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x4,drive=drive-virtio-disk0,id=virtio-disk0,bootindex=1

-vnc 0.0.0.0:12 : 设置 VNC。

在 main 函数中 , 接下来的 for 循环和大量的 switch case 语句 , 就是对于这些参数的解析 , 我们不一一解析 , 后面真的用到这些参数的时候 , 我们再仔细看。

3. 初始化 machine

回到 main 函数 , 接下来是初始化 machine。


```

1 machine_class = select_machine();
2 current_machine = MACHINE(object_new(object_class_get_name(
3     OBJECT_CLASS(machine_class))));

```

这里的 `machine_class` 是什么呢？这还得从 `machine` 参数说起。

```

1 -machine pc-i440fx-trusty,accel=kvm,usb=off

```

这里的 `pc-i440fx` 是 x86 机器默认的体系结构。在 `hw/i386/pc_piix.c` 中，它定义了对应的 `machine_class`。

```

1 DEFINE_I440FX_MACHINE(v4_0, "pc-i440fx-4.0", NULL,
2     pc_i440fx_4_0_machine_options);
3
4 #define DEFINE_I440FX_MACHINE(suffix, name, compatfn, optionfn) \
5     static void pc_init_##suffix(MachineState *machine) \
6     { \
7         .....
8         pc_init1(machine, TYPE_I440FX_PCI_HOST_BRIDGE, \
9             TYPE_I440FX_PCI_DEVICE); \
10    } \
11    DEFINE_PC_MACHINE(suffix, name, pc_init_##suffix, optionfn)
12
13
14 #define DEFINE_PC_MACHINE(suffix, namestr, initfn, optsfn) \
15     static void pc_machine_##suffix##_class_init(ObjectClass *oc, void *data
16 ) \
17     { \
18         MachineClass *mc = MACHINE_CLASS(oc); \
19         optsfn(mc); \
20         mc->init = initfn; \
21    } \
22    static const TypeInfo pc_machine_type_##suffix = { \
23        .name      = namestr TYPE_MACHINE_SUFFIX, \
24        .parent     = TYPE_PC_MACHINE, \
25        .class_init = pc_machine_##suffix##_class_init, \
26    }; \
27    static void pc_machine_init_##suffix(void) \
28    { \

```

```
29     type_register(&pc_machine_type_##suffix); \
30 } \
31 type_init(pc_machine_init_##suffix)
```

为了定义 `machine_class`，这里有一系列的宏定义。入口是 `DEFINE_I440FX_MACHINE`。这个宏有几个参数，`v4_0` 是后缀，`"pc-i440fx-4.0"` 是名字，`pc_i440fx_4_0_machine_options` 是一个函数，用于定义 `machine_class` 相关的选项。这个函数定义如下：

 复制代码

```
1 static void pc_i440fx_4_0_machine_options(MachineClass *m)
2 {
3     pc_i440fx_machine_options(m);
4     m->alias = "pc";
5     m->is_default = 1;
6 }
7
8 static void pc_i440fx_machine_options(MachineClass *m)
9 {
10     PCMachineClass *pcmc = PC_MACHINE_CLASS(m);
11     pcmc->default_nic_model = "e1000";
12
13     m->family = "pc_piix";
14     m->desc = "Standard PC (i440FX + PIIX, 1996)";
15     m->default_machine_opts = "firmware=bios-256k.bin";
16     m->default_display = "std";
17     machine_class_allow_dynamic_sysbus_dev(m, TYPE_RAMFB_DEVICE);
18 }
```

我们先不看 `pc_i440fx_4_0_machine_options`，先来看 `DEFINE_I440FX_MACHINE`。

这里面定义了一个 `pc_init_##suffix`，也就是 `pc_init_v4_0`。这里面转而调用 `pc_init1`。注意这里这个函数只是定义了一下，没有被调用。


接下来，`DEFINE_I440FX_MACHINE` 里面又定义了 `DEFINE_PC_MACHINE`。它有四个参数，除了 `DEFINE_I440FX_MACHINE` 传进来的三个参数以外，多了一个 `initfn`，也即初始化函数，指向刚才定义的 `pc_init_##suffix`。

在 `DEFINE_PC_MACHINE` 中，我们定义了一个函数 `pc_machine_##suffix##class_init`。从函数的名字 `class_init` 可以看出，这是 `machine_class` 从纸面上的 `class` 初始化为 `Class` 对象的方法。在这个函数里面，我们可以看到，它创建了一个 `MachineClass` 对象，这个就是 `Class` 对象。`MachineClass` 对象的 `init` 函数指向上面定义的 `pc_init##suffix`，说明这个函数是 `machine` 这种类型初始化的一个函数，后面会被调用。

接着，我们看 `DEFINE_PC_MACHINE`。它定义了一个 `pc_machine_type_##suffix` 的 `TypeInfo`。这是用于生成纸面上的 `class` 的原材料，果真后面调用了 `type_init`。

看到了 `type_init`，我们应该能够想到，既然它定义了一个纸面上的 `class`，那上面的那句 `module_call_init`，会和我们上面解析的 `type_init` 是一样的，在全局的表里面注册了一个全局的名字是“`pc-i440fx-4.0`”的纸面上的 `class`，也即 `TypeImpl`。

现在全局表中有这个纸面上的 `class` 了。我们回到 `select_machine`。

 复制代码

```
1 static MachineClass *select_machine(void)
2 {
3     MachineClass *machine_class = find_default_machine();
4     const char *optarg;
5     QemuOpts *opts;
6     .....
7     opts = qemu_get_machine_opts();
8     qemu_opts_loc_restore(opts);
9
10    optarg = qemu_opt_get(opts, "type");
11    if (optarg) {
12        machine_class = machine_parse(optarg);
13    }
14    .....
15    return machine_class;
16 }
17
18 MachineClass *find_default_machine(void)
19 {
20     GSList *el, *machines = object_class_get_list(TYPE_MACHINE, false);
21     MachineClass *mc = NULL;
22     for (el = machines; el; el = el->next) {
23         MachineClass *temp = el->data;
24         if (temp->is_default) {
25             mc = temp;
26             break;
27         }
28     }
```

```

29     g_slist_free(machines);
30     return mc;
31 }
32
33 static MachineClass *machine_parse(const char *name)
34 {
35     MachineClass *mc = NULL;
36     GSList *el, *machines = object_class_get_list(TYPE_MACHINE, false);
37
38     if (name) {
39         mc = find_machine(name);
40     }
41     if (mc) {
42         g_slist_free(machines);
43         return mc;
44     }
45     .....
46 }

```

在 `select_machine` 中，有两种方式可以生成 `MachineClass`。一种方式是 `find_default_machine`，找一个默认的；另一种方式是 `machine_parse`，通过解析参数生成 `MachineClass`。无论哪种方式，都会调用 `object_class_get_list` 获得一个 `MachineClass` 的列表，然后在里面找。 `object_class_get_list` 定义如下：


 复制代码

```

1  GSList *object_class_get_list(const char *implements_type,
2                               bool include_abstract)
3  {
4      GSList *list = NULL;
5
6      object_class_foreach(object_class_get_list_tramp,
7                           implements_type, include_abstract, &list);
8      return list;
9  }
10
11 void object_class_foreach(void (*fn)(ObjectClass *klass, void *opaque), const char *imp.
12                           void *opaque)
13 {
14     OCFData data = { fn, implements_type, include_abstract, opaque };
15
16     enumerating_types = true;
17     g_hash_table_foreach(type_table_get(), object_class_foreach_tramp, &data);
18     enumerating_types = false;
19 }

```

在全局表 `type_table_get()` 中，对于每一项 `TypeImpl`，我们都执行 `object_class_foreach_tramp`。

 复制代码


```
1 static void object_class_foreach_tramp(gpointer key, gpointer value,
2                                       gpointer opaque)
3 {
4     OCData *data = opaque;
5     TypeImpl *type = value;
6     ObjectClass *k;
7
8     type_initialize(type);
9     k = type->class;
10    .....
11    data->fn(k, data->opaque);
12 }
13
14 static void type_initialize(TypeImpl *ti)
15 {
16     TypeImpl *parent;
17    .....
18    ti->class_size = type_class_get_size(ti);
19    ti->instance_size = type_object_get_size(ti);
20    if (ti->instance_size == 0) {
21        ti->abstract = true;
22    }
23    .....
24    ti->class = g_malloc0(ti->class_size);
25    .....
26    ti->class->type = ti;
27
28    while (parent) {
29        if (parent->class_base_init) {
30            parent->class_base_init(ti->class, ti->class_data);
31        }
32        parent = type_get_parent(parent);
33    }
34
35    if (ti->class_init) {
36        ti->class_init(ti->class, ti->class_data);
37    }
38 }
```

在 `object_class_foreach_tramp` 中，会调用 `type_initialize`，这里面会调用 `class_init` 将纸面上的 `class` 也即 `TypeImpl` 变为 `ObjectClass`，`ObjectClass` 是所有 `Class` 类的祖先，`MachineClass` 是它的子类。

因为在 machine 的命令行里面，我们指定了名字为"pc-i440fx-4.0"，就肯定能够找到我们注册过了的 TypeImpl，并调用它的 class_init 函数。

因而 pc_machine_##suffix##class_init 会被调用，在这里面，pc_i440fx_machine_options 才真正被调用初始化 MachineClass，并且将 MachineClass 的 init 函数设置为 pc_init##suffix。也即，当 select_machine 执行完毕后，就有一个 MachineClass 了。

接着，我们回到 object_new。这就很好理解了，MachineClass 是一个 Class 类，接下来应该通过它生成一个 Instance，也即对象，这就是 object_new 的作用。

 复制代码

```
1 Object *object_new(const char *typename)
2 {
3     TypeImpl *ti = type_get_by_name(typename);
4
5     return object_new_with_type(ti);
6 }
7
8 static Object *object_new_with_type(Type type)
9 {
10     Object *obj;
11     type_initialize(type);
12     obj = g_malloc(type->instance_size);
13     object_initialize_with_type(obj, type->instance_size, type);
14     obj->free = g_free;
15
16     return obj;
17 }
```

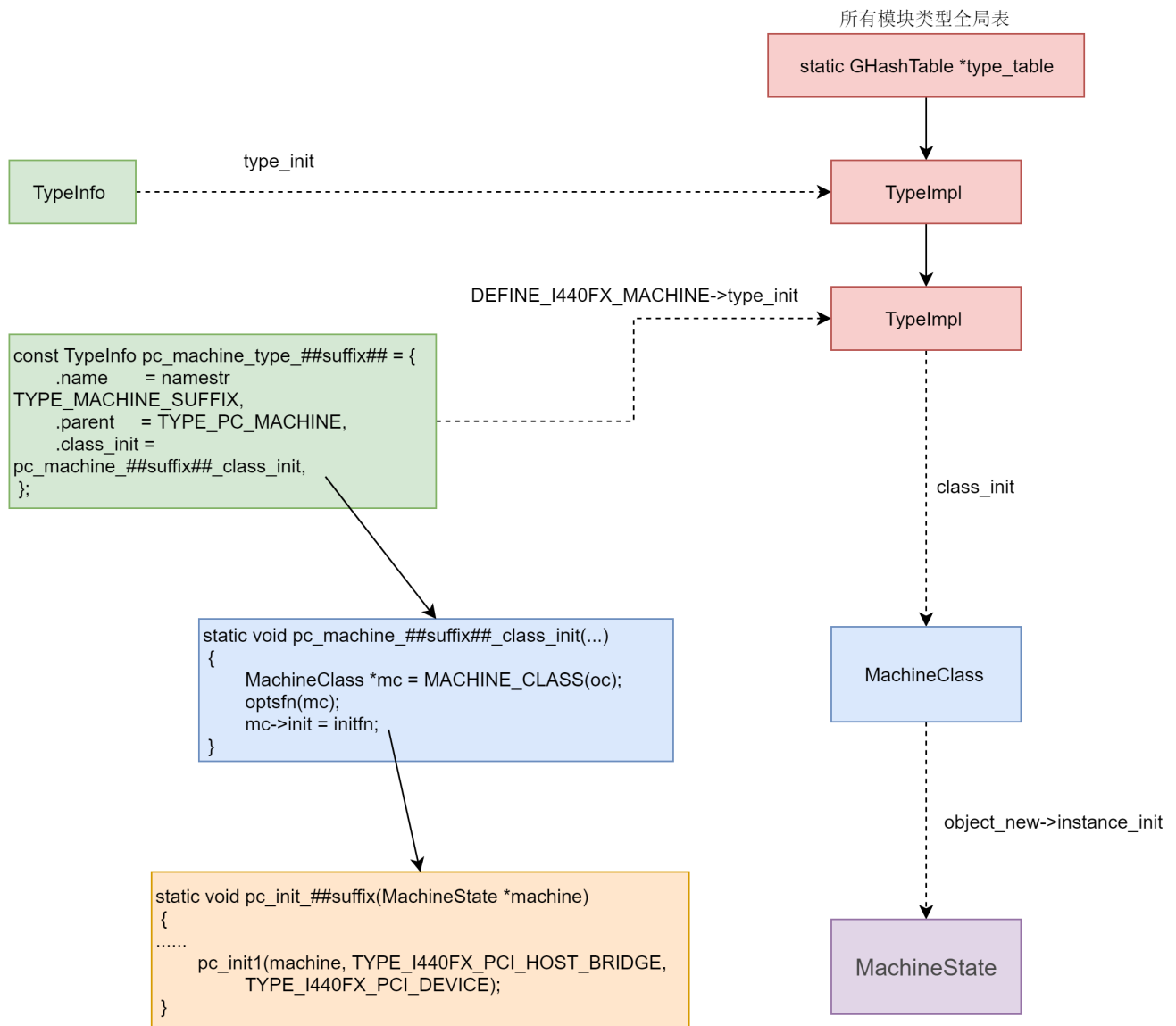
object_new 中，TypeImpl 的 instance_init 会被调用，创建一个对象。current_machine 就是这个对象，它的类型是 MachineState。

至此，绕了这么大一圈，有关体系结构的对象才创建完毕，接下来很多的设备的初始化，包括 CPU 和内存的初始化，都是围绕着体系结构的对象来的，后面我们会常常看到 current_machine。

总结时刻

这一节，我们学到，虚拟机对于设备的模拟是一件非常复杂的事情，需要用复杂的参数模拟各种各样的设备。为了能够适配这些设备，qemu 定义了自己的模块管理机制，只有了解了这种机制，后面看每一种设备的虚拟化的时候，才有一个整体的思路。

这里的 MachineClass 是我们遇到的第一个，我们需要掌握它里面各种定义之间的关系。



每个模块都会有一个定义 TypeInfo，会通过 type_init 变为全局的 TypeImpl。TypeInfo 以及生成的 TypeImpl 有以下成员：

name 表示当前类型的名称

parent 表示父类的名称

class_init 用于将 TypeImpl 初始化为 MachineClass

instance_init 用于将 MachineClass 初始化为 MachineState

所以，以后遇到任何一个类型的时候，将父类和子类之间的关系，以及对应的初始化函数都要看好，这样就一目了然了。

课堂练习

你可能会问，这么复杂的 qemu 命令，我是怎么找到的，当然不是我一个字一个字打的，这是著名的云平台管理软件 OpenStack 创建虚拟机的时候自动生成的命令行。所以，给你留一道课堂练习题，请你看一下 OpenStack 的基本原理，看它是通过什么工具来管理如此复杂的命令行的。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，**反复研读**。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

 极客时间

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 49 | 虚拟机：如何成立子公司，让公司变集团？

下一篇 51 | 计算虚拟化之CPU（下）：如何复用集团的人力资源？



大王叫我来巡山

2019-07-22

感觉设计这个软件的真厉害，怪不得我们自己做的业务系统自己都信不过，差距实在是太远，国内很多大公司在分享技术的时候也就是个PPT，根本不敢把代码放出来给大家看，也没有把实际用的效果展示给大家，只是给别人的感觉很牛逼而已。我感觉我从事这个工作这么久，真没遇到过这种大神

展开 ∨

💬 1

👍 1



石维康

2019-07-23

请问老师pc_machine_type_##suffix所对应的TypeImpl的instance_init是在哪初始化的？也就是从代码里如何体现从MachineClass生成MachineState？

展开 ∨

💬

👍



落大雨起泡泡

2019-07-22

请问代码是用的哪个内核版本

展开 ∨

💬 1

👍