qemu qom模型

读过qemu源码的同学都会发现里面很多Type，object，class这些东西，这就是qemu的qom的组件。如果不能很好的理解qom模型就很难阅读源码。

qemu是c语言写的，但是qemu需要模拟很多复杂的设备总线这些东西，而这些设备总线之间是存在一些共性和继承关系的，也就是需要一些面向对象的描述。为了用c来模拟面向对象的东西就创造出了qom（qemu object model）模型。

在qom模型中有很多类似的概念，如果不加区分很容易搞混。

type，class，xxxstate的区别与联系

在qemu的设备模拟中充斥着xx_info，并有一个type_init函数紧随其后，例如：

```
static const TypeInfo virt_machine_info = {
    .name          = TYPE_VIRT_MACHINE,
    .parent        = TYPE_MACHINE,
    .abstract      = true,
    .instance_size = sizeof(VirtMachineState),
    .class_size    = sizeof(VirtMachineClass),
    .class_init    = virt_machine_class_init,
    .instance_init = virt_instance_init,
    .interfaces = (InterfaceInfo[]) {
         { TYPE_HOTPLUG_HANDLER },
         { }
    },
};

static void machvirt_machine_init(void)
{
    type_register_static(&virt_machine_info);
}
type_init(machvirt_machine_init);
```

这是arm64的virt主板的type info，称为virt_machine_info，由type_init来注册。这个type info类似于c++的一个类的概念，但是并不一样，它更像是一个类的描述，因为它并不包含具体的数据成员，只是隐含了静态数据（class所指）和非静态数据（initance所指，也就是xxxstate）。这里最重要的三类概念都出现了，type_info，class，xxxstate。搞清楚这三个东西才能理解qom，如果你用c++的类的模型来理解的话，type_info可以视作类的描述，class可视作静态成员，xxxstate可视作对象。但是这三者在某种程度上是独立的，xxxstate并不是有type info定义的，他们各自有自己的父子节点体系，而且互不干扰。比如virt_machine_type的父类型是machine_type；VirtMachineClass的父类是MachineClass，VirtMachineState的父结构是MachineState。他们之间有联系，但是是平行的对应的。

比如virt_machine_info的类型层次为TYPE_VIRT_MACHINE->TYPE_MACHINE->TYPE_OBJECT。

VirtMachineClass的层次为：VirtMahcineClass->MachineClass->ObjectClass。

VirtMachineState的层次为：VirtMachineState->MachineState->Object。

他们各自有自己的继承关系，在c语言中，这种继承是通过包含来实现的。在子结构体中的第一个元素往往是父结构。

好了，现在我们搞清楚了type，class，state这三者之间的关系，这是了解qom最重要的一步，我看到很多资料都没有讲的很清楚，而这又是非常容易混淆的东西。综合【1】【2】结合我自己的理解，有了上面的论述，当然我不一定是对的，但是我觉得很接近事实。

qom模型的初始化包括三个阶段，类型的注册（type init），静态类型初始化（type initialize），对象初始化（object new）。下面分别讲解。

类型的注册

这里的类型就是上面提到的type。注册是指将需要编译的代码中有关type的描述注册成一系列对应的数据结构并且组织起来，方便后面使用。每个type info之后的type_init宏的作用就是注册。

```
#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);                                   \
}
```

type_init最终会调用register_module_init。__attribute__((constructor))说明这个宏是在进入main函数之前调用的。module_init的入参是MODULE_INIT_QOM，代表type都是qom类型的

< 202

日	一	二
27	28	29
3	4	5
10	11	12
17	18	19
24	25	26
31	1	2

导航

博客园

首页

新随笔

联系

订阅

管理

统计

随笔 - 97
文章 - 0
评论 - 0
阅读 -

36954

公告

昵称： 半山随笔
园龄： 1年11个月
粉丝： 5
关注： 3
+加关注

搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

bug解决

ebpf(7)

操作系统

工具介绍

环境搭建

虚拟化技

```
static ModuleTypeList *find_type(module_init_type type)
{
    init_lists();

    return &init_type_list[type];
}

void register_module_init(void (*fn)(void), module_init_type type)
{
    ModuleEntry *e;
    ModuleTypeList *l;

    e = g_malloc0(sizeof(*e));
    e->init = fn;
    e->type = type;

    l = find_type(type);

    QTAILQ_INSERT_TAIL(l, e, node);
}
```

find_type会返回init_type_list中对应qomt类型的链表。init_type_list是一个全局链表数组。对应的type类型会在registe_module_init函数中分配内存，主要是将function设置到init成员，之后挂到qom链表中。

在进入main函数之后，会调用module_call_init来初始化type。调用链为:main->qemu_init->qemu_init_subsystems->module_call_init。

```
void module_call_init(module_init_type type)
{
    ModuleTypeList *l;
    ModuleEntry *e;

    if (modules_init_done[type]) {
        return;
    }

    l = find_type(type);

    QTAILQ_FOREACH(e, l, node) {
        e->init();
    }

    modules_init_done[type] = true;
}
```

module_call_init会遍历type_init注册的类型，然后调用他们的init回调函数。对virt_machine_info这个例子，init回调就是machvirt_machine_init，也就是type_init的入参。

一般这个init回调会调用到type_register_internal函数来初始化这个type。

```
static TypeImpl *type_register_internal(const TypeInfo *info)
{
    TypeImpl *ti;

    if (!type_name_is_valid(info->name)) {
        fprintf(stderr, "Registering '%s' with illegal type name\n", info->name);
        abort();
    }

    ti = type_new(info);

    type_table_add(ti);
    return ti;
}
```

type_new会创建一个TypeImpl类型的结构，将type info中的信息复制过去，然后将其加入到哈希表中。

```
static void type_table_add(TypeImpl *ti)
{
    assert(!enumerating_types);
    g_hash_table_insert(type_table_get(), (void *)ti->name, ti);
}

static GHashTable *type_table_get(void)
{
    static GHashTable *type_table;
```

```
    if (type_table == NULL) {
        type_table = g_hash_table_new(g_str_hash, g_str_equal);
    }

    return type_table;
}
```

这里要注意，type_table是一个静态变量，在第一次分配内存之后就会一直存在，所以在之后的调用中都会返回第一次分配的tpye_table，这样每次调用type_table_add都会加入到一个哈希表中。想要找到这个哈希表也仅仅需要调用type_table_get即可。

这样type info就转变为TypeImpl结构了。之后就可以从TypeImpl结构去查找type。

类型的进一步初始化

上一步仅仅是将type最基本的信息记录下来，下一步要全面初始化type类型。这是在type_initialize函数中做的。对于virt_machine_info的初始化，调用链为main->qemu_init->qemu_create_machine->select_machine->object_class_get_list->object_class_foreach->object_class_foreach_tramp->type_initialize

type_initialize比较长，主要包含两部分，设置相关的成员，给class成员分配内存。

```
static void type_initialize(TypeImpl *ti)
{
    TypeImpl *parent;

    if (ti->class) {
        return;
    }

    ti->class_size = type_class_get_size(ti);
    ti->instance_size = type_object_get_size(ti);
    ti->instance_align = type_object_get_align(ti);
    /* Any type with zero instance_size is implicitly abstract.
     * This means interface types are all abstract.
     */
    if (ti->instance_size == 0) {
        ti->abstract = true;
    }
    if (type_is_ancestor(ti, type_interface)) {
        assert(ti->instance_size == 0);
        assert(ti->abstract);
        assert(!ti->instance_init);
        assert(!ti->instance_post_init);
        assert(!ti->instance_finalize);
        assert(!ti->num_interfaces);
    }
    ti->class = g_malloc0(ti->class_size);
```

之后会对parent进行初始化。

```
    parent = type_get_parent(ti);
    if (parent) {
        type_initialize(parent);
        GSList *e;
        int i;

        g_assert(parent->class_size <= ti->class_size);
        g_assert(parent->instance_size <= ti->instance_size);
        memcpy(ti->class, parent->class, parent->class_size);
        ti->class->interfaces = NULL;

        for (e = parent->class->interfaces; e; e = e->next) {
            InterfaceClass *iface = e->data;
            ObjectClass *klass = OBJECT_CLASS(iface);

            type_initialize_interface(ti, iface->interface_type, klass->type);
        }

        for (i = 0; i < ti->num_interfaces; i++) {
            TypeImpl *t = type_get_by_name_noload(ti->interfaces[i].typename);
            if (!t) {
                error_report("missing interface '%s' for object '%s'",
                             ti->interfaces[i].typename, parent->name);
                abort();
            }
            for (e = ti->class->interfaces; e; e = e->next) {
                TypeImpl *target_type = OBJECT_CLASS(e->data)->type;

                if (type_is_ancestor(target_t
                    break;
                }
```


阅读排行榜

t rootfs失败问题(48

ency mode"的解决

k8s(2043)

l源码分析(1831)

_module section si
rnel's built struct n
e的解决办法(1415)

推荐排行榜

（十）- 页面回收（二

析(1)

his_module sectior
kernel's built struc
me的解决办法(1)

（五）- 缺页处理(1)

（一）物理内存的组织

```
            }

            if (e) {
                continue;
            }

            type_initialize_interface(ti, t, t);
        }
    }

    ti->class->properties = g_hash_table_new_full(g_str_hash, g_str_equal, NULL,
                                                  object_property_free);

    ti->class->type = ti;
```

最后会调用父类型的class_base_init函数和当前类型的class_init函数。

```
    while (parent) {
        if (parent->class_base_init) {
            parent->class_base_init(ti->class, ti->class_data);
        }
        parent = type_get_parent(parent);
    }

    if (ti->class_init) {
        ti->class_init(ti->class, ti->class_data);
    }
}
```

上面还会初始化interface等还没有将到的部分内容。

type_initialize会出现在很多地方，但是type只需要初始化一次。

object_class_get_list会遍历所有之前注册的类型，每到一个类型都会调用type_initialize函数进行初始化。

对象的构造

构造对象就需要调用type对应的非静态成员的初始化函数，对于virt_machine_type，就是调用virt_instance_init。下面是调用virt_instance_init的调用栈。

在qemu_create_machine中会根据命令行传入的machine类型来找到之前注册的对应的type，根据这个type调用object_new_with_class，之后会调用object_new_with_type。这是一个更为基础的函数，使用它的更普遍的方法是调用object_new。

```
Object *object_new_with_class(ObjectClass *klass)
{
    return object_new_with_type(klass->type);
}

Object *object_new(const char *typename)
{
    TypeImpl *ti = type_get_or_load_by_name(typename, &error_fatal);

    return object_new_with_type(ti);
}
```

object_new_with_type创建一个Object对象并分配内存之后会做进一步初始化。

```
static Object *object_new_with_type(Type type)
{
    Object *obj;
    size_t size, align;
    void (*obj_free)(void *);

    g_assert(type != NULL);
    type_initialize(type);

    size = type->instance_size;
    align = type->instance_align;

    /*
     * Do not use qemu_memalign unless required.  Depending on the
     * implementation, extra alignment implies extra overhead.
     */
    if (likely(align <= __alignof__(qemu_max_
        obj = g_malloc(size);
        obj_free = g_free;
```

```
    } else {
        obj = qemu_memalign(align, size);
        obj_free = qemu_vfree;
    }

    object_initialize_with_type(obj, size, type);
    obj->free = obj_free;

    return obj;
}
```

object_initialize_with_type会将object用0填充，初始化property，最后调用object_init_with_type。

```
static void object_init_with_type(Object *obj, TypeImpl *ti)
{
    if (type_has_parent(ti)) {
        object_init_with_type(obj, type_get_parent(ti));
    }

    if (ti->instance_init) {
        ti->instance_init(obj);
    }
}
```

object_init_with_type最终调用type指定的instance_init回调。至此Object对象就创建完成了。

前文提到，vrit_machine_info type对应的对象是 VirtMachineState，但是前面只是创建了Object对象，VirtMachineState在哪里？其实Object是所有对象的祖先，因此可以转化为任何子类。在initant_init回调中会将Object转化为VirtMahcine_State并初始化。

```
static void virt_instance_init(Object *obj)
{
    VirtMachineState *vms = VIRT_MACHINE(obj);
    VirtMachineClass *vmc = VIRT_MACHINE_GET_CLASS(vms);

    /* EL3 is disabled by default on virt: this makes us consistent
     * between KVM and TCG for this board, and it also allows us to
     * boot UEFI blobs which assume no TrustZone support.
     */
    vms->secure = false;

    /* EL2 is also disabled by default, for similar reasons */
    vms->virt = false;

...
```

所以，调用万initant_init回调，其实我们得到就是最终的VirtMachineState对象。

属性

属性可以帮助对象实现类似C++多态的性质。属性分为两部分，静态和非静态。静态属性存在于ObjectClass中，非静态存在于Object中，在Object结构体中有一个properties用来存放property。

```
struct Object
{
    /* private: */
    ObjectClass *class;
    ObjectFree *free;
    GHashTable *properties;
    uint32_t ref;
    Object *parent;
};
```

ObjectClass

```
struct ObjectClass
{
    /* private: */
    Type type;
    GSList *interfaces;

    const char *object_cast_cache[OBJECT_CLASS_CAST_CACHE];
    const char *class_cast_cache[OBJECT_CLASS
                                 字节旗下的 AI IDE
    ObjectUnparent *unparent;
```

```
    GHashTable *properties;
};
```

两者都包含了properties域，都是hash类型。

属性类由ObjectProperty表示。

```
struct ObjectProperty
{
    char *name;
    char *type;
    char *description;
    ObjectPropertyAccessor *get;
    ObjectPropertyAccessor *set;
    ObjectPropertyResolve *resolve;
    ObjectPropertyRelease *release;
    ObjectPropertyInit *init;
    void *opaque;
    QObject *defval;
};
```

name是属性名，type是属性的类型，有bool，link，string等。每一种属性类都有对应的结构体，opaque用来存放具体的属性类。

```
typedef struct {
    union {
        Object **targetp;
        Object *target; /* if OBJ_PROP_LINK_DIRECT, when holding the pointer  */
        ptrdiff_t offset; /* if OBJ_PROP_LINK_CLASS */
    };
    void (*check)(const Object *, const char *, Object *, Error **);
    ObjectPropertyLinkFlags flags;
} LinkProperty;

typedef struct BoolProperty
{
    bool (*get)(Object *, Error **);
    void (*set)(Object *, bool, Error **);
} BoolProperty;

typedef struct StringProperty
{
    char *(*get)(Object *, Error **);
    void (*set)(Object *, const char *, Error **);
} StringProperty;
```

还有相关的函数负责查找添加属性。object_property_add, object_property_find。

对于设备类对象，最重要的属性是realized，它是bool类型。在qemu代码中常常可以看到类似object_property_set_bool(OBJECT(dev), true, "realized", &err);

参考文献

【1】QEMU/KVM源码解析与应用 李强

【2】https://martins3.github.io/qemu/qom.html

【3】https://martins3.github.io/

« 上一篇： cpu0 SCHED_SOFTIRQ异常升高的问题解释
» 下一篇： linux定时器迁移分析

posted on
2025-05-07 23:03
半山随笔
阅读(54)
评论(0)

登录后才能查看或发表评论，立即 登录 或者 逛逛 博客园首页

相关博文：
· armv8虚拟化原理笔记
· 在qemu中配置pci bus和numa node亲和性
· Qt MetaTypeInterface
· Qt源码解析——元对象系统热身
· 【QML qmlRegisterType】qmlRegisterType 的功能以及用法

阅读排行：
· 博客园众包：再次诚征3D影像景深延拓实时处理方案（预算8-15万，需求有调整）
· 扣子(Coze)，开源了！Dify 天塌了
· 精选 5 款 .NET 开源、功能强大的工作流系统，告别重复造轮子！
· 爆肝2月，我的 AI 代码生成平台上线了！
· 从经典产品看大模型方向

历史上的今天：
2024-05-07 使用libvirt绑定numa node