QOM exegesis and apocalypse

Paolo Bonzini
Red Hat, Inc.

KVM Forum 2014

ex·e·ge·sis /ˌeksiˈjēsis/

noun (plural: exegeses)

critical explanation or interpretation of a text

An IMAGE TEN Production

NIGHT OF THE LIVING DEAD

They keep coming back in a bloodthirsty lust for HUMAN FLESH!...

Pits the dead against the living in a struggle for survival!

IONAS

Ἀπό (from, out of) + καλύπτω (to hide)

uncovering, disclosure of what's hidden

Outline

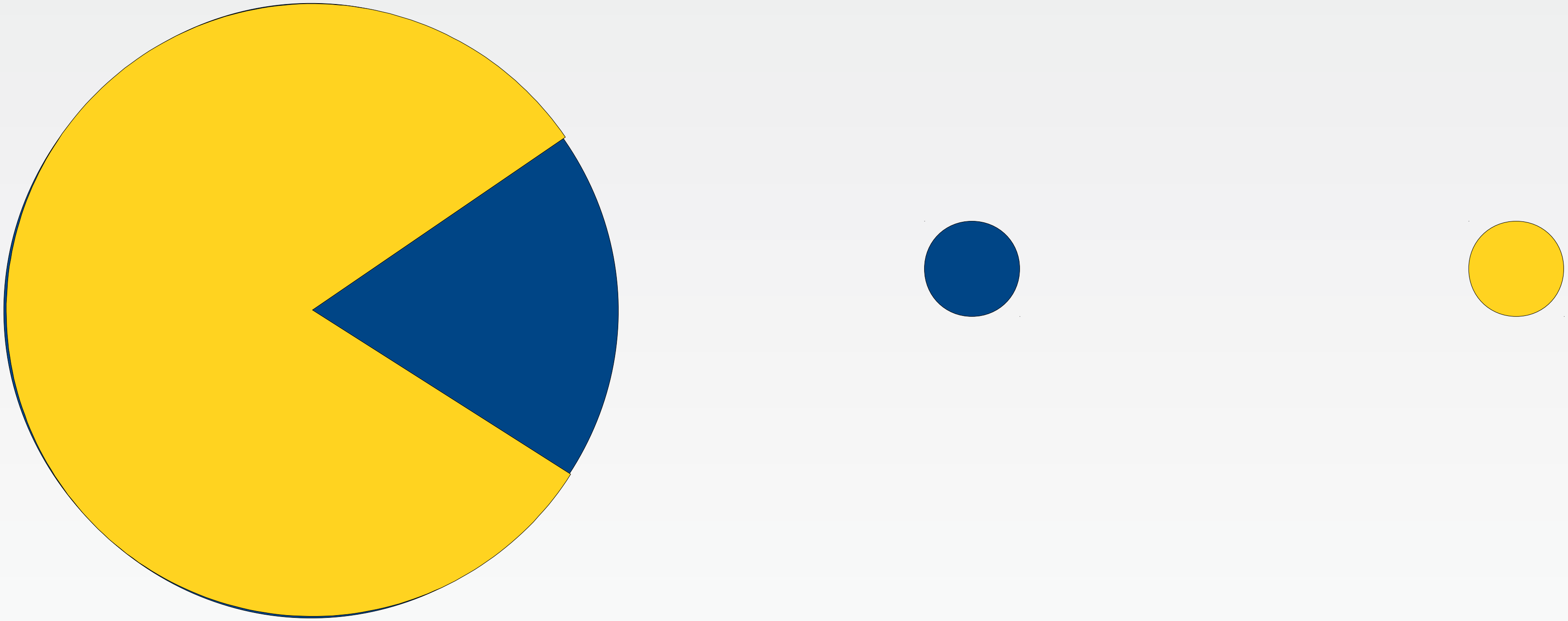
- What is the QEMU Object Model?
- How do you use QOM?
- How could we improve QOM?

Why QOM?

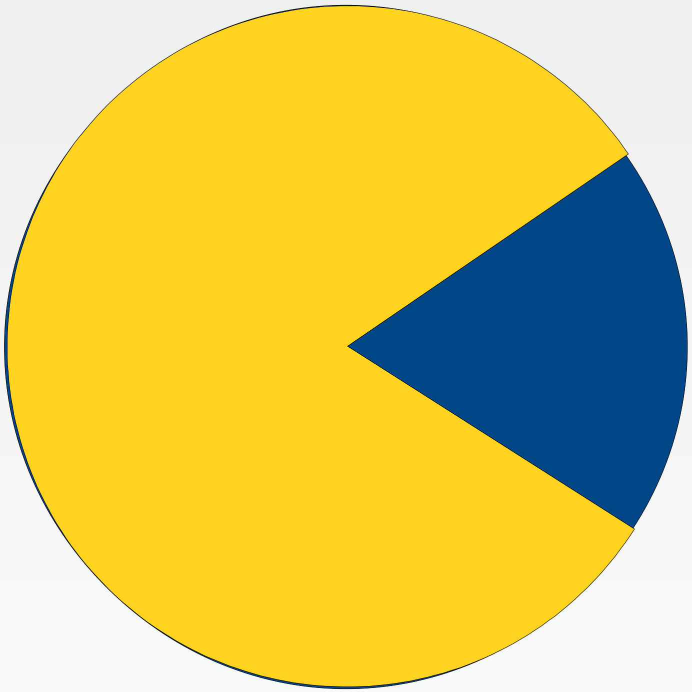
All device creation, device configuration, backend creation and backed configuration done through a single interface

Rigorous support for introspection both of runtime objects and type capabilities

Did it work?



Do pie charts look like Pac Man?



● % of pie charts that look like Pac Man

● % of pie charts that do not look like Pac Man

The QOM reality

- RNG backend
- Memory backend
- Console
- Device
- IRQ
- MemoryRegion
- Machine

The QOM reality

- ✔ New backends use QOM (RNG, memory device)
- ✔ Clear model of object lifetime
- ✔ Simple, type-safe QMP interface

- ✗ Limited type introspection
- ✗ Original intended interface mostly unused

What happened?

- Bad design? No.
  - QOM integrates well with the rest of QEMU
  - All problems are fixable
- Solution in search of a problem? Somewhat.
  - Adding new backends happens rarely
  - Introspection already part of qdev & vmstate
- No transition/completion plan? Totally.

The rest of this talk

- ✔ New backends use QOM (RNG, memory device)
- ✔ Clear model of object lifetime
- ✔ Simple, type-safe QMP interface

- ✗ Limited type introspection
- ✗ Original intended interface mostly unused

QOM properties and introspection

QOM in practice

- Inheritance (single inheritance + interfaces)
- Polymorphic objects (class based)
- Polymorphic properties (prototype based)
- Object enumeration (“composition tree”)
- Generalized factory interface

QOM concepts: properties

- Properties are the external interface to an object
- Different uses of properties:
  - For construction: set before the object is “started”
  - For inspection: read after the object is “started”
  - Very few examples of the second kind :)
- Similar to Linux sysfs, with arbitrary QAPI structs instead of bytes

A step back: the QAPI vision

“QAPI is a framework to move QEMU to the next level of feature, function, and robustness”

More practically...

- Decompose serialization into
  - Marshaling (composite → primitive type)
  - Transport (primitive type ↔ representation)
- Marshaling done by automatically generated code
- Transport done by hand written “visitors”
  - QObject (JSON), QemuOpts (key/value pairs), string
  - “Input” vs. “output” visitors

It works!

Fundamental QAPI data types

- Scalar JSON types: Integer, string, boolean
- Homogeneous arrays (*xyz*List)
  - Non-homogeneous JSON arrays never used
- Enums (JSON String ↔ C enum)
- Records (including discriminated records)
  - Serialized as JSON dictionaries
  - Strongly-typed

QOM property types

- Non-object
  - Example: isa-serial.iobase=0x402
  - QOM property types are QAPI types
- Object
  - child<X> provides the *canonical path* to an object
  - link<X> provides alternative paths
- Aliases
  - Same type as the target, except child<X> → link<X>

QOM properties under the hood

- All properties are accessed through visitors:

```
typedef void (ObjectPropertyAccessor)(Object *obj,
     Visitor *v, void *opaque,
     const char *name, Error **errp);
typedef void (ObjectPropertyRelease)(Object *obj,
     const char *name, void *opaque);
```

- Similar to Linux sysfs, visitors instead of files
- Wrappers for strings and bools
  - Again, think of Linux seqfile
  - Still some boilerplate, but not too bad

Visitors in QOM

- QObject (type-safe!)

```
{ 'execute': 'object-add', 'arguments': {
   'id': 'my-rng', 'type': 'rng-random',
   'props': { 'filename': '/dev/random' } }
```

- QemuOpts (key/value pair)

```
qemu -object rng-random,id=my-rng,filename=/dev/random
```

```
object_add rng-random,id=my-rng,filename=/dev/random
```

- String (scalar-only)
  - -device
  - info qtree (“human” mode)

Creating an object

```
Object *o = object_new(TYPE_RNG_BACKEND_RANDOM);
object_property_set_str(o, “filename”, “/dev/random”, NULL);
object_property_set_bool(o, “opened”, “true”, NULL);

object_property_add_child(container_get(“/somewhere”),
                          “my-rng”, o, NULL);
object_unref(o);
```

Inside properties

```
static bool rng_get_opened(Object *obj, Error **errp)
{
    RngBackend *s = RNG_BACKEND(obj);
    return s->opened;
}


static void rng_set_opened(Object *obj, bool value,
                           Error **errp)
{
    RngBackend *s = RNG_BACKEND(obj);
    RngBackendClass *k = RNG_BACKEND_GET_CLASS(s);

    ...

    if (k->opened) {
        k->opened(s, errp)
    }
}
```

Inside properties

```
static void rng_backend_init(Object *obj)
{
    object_property_add_bool(obj, "opened",
               rng_get_opened, rng_set_opened, NULL);
}


static const TypeInfo rng_backend_info = {
    .name          = TYPE_RNG_BACKEND,
    .parent        = TYPE_OBJECT,
    .instance_size = sizeof(RngBackend),
    .instance_init = rng_backend_init,
    .class_size    = sizeof(RngBackendClass),
    .abstract      = true,
};
```

The two sides of QOM

- Class-based methods/interface polymorphism
  - ➔ Cannot override a method for a single object
- Object-based, dynamic properties
  - ➔ Each instance of a class can have different properties
- **Except for child properties, all properties are usually handled as if they were static**
- So why the difference?

Uses of dynamic properties

- “Child” properties do not “exist” until the embedded object is created with object_new()
  - MemoryRegions in a device
  - e.g. /objects contains /foo after “-object id=foo”
- “Array” properties
  - e.g. pci-host/pci-bus/child[12]
  - *Not* array-typed properties!
  - Usually children or links

Dynamic properties vs. introspection

- Property names and types are an object's schema
- With dynamic properties, the schema is not known in advance
- “Solution”: instantiate a temporary object, examine it, delete it
- Implemented for “-device foo,help”

Towards a QOM schema?

- No QAPI schema introspection in QEMU
  - Patches stuck?
  - Prerequisite for QOM introspection (QOM property types can be arbitrary QAPI types)
- Should we expose a QOM schema via QAPI?
  - Similar to “-device foo,help”, but for objects
  - Dummy object creation, or static properties?

QOM object lifetime and the composition tree

QOM composition tree

```
/machine
    /peripheral
        /serial0   -device isa-serial,id=serial0,iobase=0x3f8,...
    /unattached
        /device[0] (PCI host)
        /device[1] (fw_cfg)
        ...
/objects
    /rng0          -device rng-random,...
/backends
```

The QOM tree keeps an object alive!

Example:

```
(qemu) object-add rng-random,id=rng0,filename=/dev/random
(qemu) device-add virtio-rng-pci,rng=rng0
```

Birth of a QOM object

- Creation (`object_new`)
  - `instance_initialize`
  - No parent
  - Properties initialized to default values
- Preparation
  - `object_property_add_child`
  - Values written to properties

... and here comes the fun part!

- Activation
  - `qdev_init`
  - `user_creatable_complete`
- Deactivation
  - `object_unparent`
- Finalization • • • *suspence* • • •
  - `instance_finalize`
  - `g_free`

`object_unparent`

- Initiated by guest or management
- Deletes the child<X> property
  - Calls the unparent callback
  - Makes the object unreachable from the composition tree
  - Drops a reference to the object
- Usually the last reference goes away
  - All properties are deleted
  - Effect: recursive unparenting of children

What should the `unparent` callback do?

- “Ultimately” cause the object to die
  - **Hide itself from the guest**
  - **Eliminate circular links** by propagating unparent to other objects (e.g. child buses)
  - No circular links? Finalization will handle tear down just fine!
- As soon as the guest finishes using it, the object will be finalized

Pattern: references from child to parent

- Children usually oblivious of parent
- If not, how to avoid dangling pointers?
  - ➔ Parent keeps children alive via composition tree
  - ➔ Children keep parent alive via reference counting
- How to avoid circular references?
  - ➔ **References to the parent should be weak; only take a reference to the parent during guest actions** (e.g. MMIO accesses)
- Guest actions cannot happen after unparent returns → no window for dangling pointers!

Pattern: references from child to parent

- Separate reference-counting APIs
  - `memory_region_ref/unref`
    Guest action, add/remove reference to the QOM parent (device)
  - `object_ref/unref`
    Management action, add/remove reference to the object itself
- `memory_region_ref/unref` implicitly keeps MemoryRegion alive (via child property)

The future: “Owner” vs. “parent”?

- Example: MemoryRegion needs to know its parent device
    - Currently, children do not need to control the lifetime of their grandparents
    - Is that really the rule?
- Counterexample: implementing PCI configuration space as MemoryRegions
    - `/.../pci-device/msix-capability/region`
    - Config space accesses bypass the capability object
    - The MemoryRegion has to keep the device alive

The future: “Owner” vs. “parent”?

- Right now, MemoryRegion always “refs” the QOM parent
- In the future, we could add a new API `memory_region_set_owner`

So, does it work?

Yes!