DIY moc
Dynamic Meta Objects

Volker Krause
KDAB
volker.krause@kdab.com
@VolkerKrause

Why touch a running system?

- `moc` hidden by the build system
- QObject internals hidden by the Q_OBJECT macro
- Qt5 can statically check signal/slot connections
- What else could we possibly want?

Dynamic Meta Objects

- `moc` requires knowing the object layout at compile time
- What if we don't have that?
  - Inter-process communication (IPC)
  - Bindings to scripting languages
- Examples in Qt
  - D-Bus
  - QML
  - QSignalSpy

Would this be possible?

- Let's look at how static meta objects work
  - Q_OBJECT macro
  - QMetaObject
  - qt_metacall()

Q_OBJECT

```
public:
    static const QMetaObject staticMetaObject;
    virtual const QMetaObject *metaObject() const;
    virtual void *qt_metacast(const char *);
    virtual int qt_metacall(QMetaObject::Call, int, void**);
private:
    static void qt_static_metacall(QObject*,
                    QMetaObject::Call, int, void **);
```

- Static members: not useful for dynamic types
- Relevant for us: just 3 virtual methods!

QMetaObject

- Contains the introspection data
- Read-only API, no virtual methods :-(
- But: Just a struct with a few pointers to data tables!

```
struct {
  const QMetaObject *superdata;
  const QByteArrayData *stringdata;
  const uint *data;
  ...
}
```

- moc generates this, no magic, but quite some busy work to do manually.

qt_metacall()

- Performs various operations accessible via QObject API:
  - Dispatching of method calls
  - Property access
  - …
- `qt_metacall(QMetaObject::Call c, int id, void **a)`
  - c: Type of operation
  - `id`: Method or property index
  - a: Operation specific arguments
  - Returns: negative number if call was handled, index re-based to super class otherwise

qt_metacast()

- Needed for `qobject_cast<>()`
- `void* qt_metacast(const char *name)`
  - If name matches your (fully qualified) class name: return `this`
  - Else: call `qt_metacast()` of base class

Do-it-yourself

- Goal
  - Method and property layout only known at runtime
  - Usable like any other QObject
- Plan
  - Create QMetaObject data tables dynamically
  - Implement generic qt_metacall() to access properties and dispatch calls

QMetaObjectBuilder

- Not public API, but provides exactly what we need
- `#include <private/qmetaobjectbuilder_p.h>`
- API largely symmetric to QMetaObject & friends

```
QMetaObjectBuilder b;
b.setClassName(“MyClass”);
b.setSuperClass(&QObject::staticMetaObject);
…
```

Properties

- QMetaPropertyBuilder – symmetric API to QMetaProperty

```
QMetaObjectBuilder b;
QMetaPropertyBuilder p =
            b.addProperty(“myProperty”, “QString”);
p.setWritable(true);
p.setNotifySignal(...);
...
```

Methods

- QMetaMethodBuilder – symmetric API to QMetaMethod
  - addMethod()
  - addSignal()
  - addSlot()
  - addConstructor()
- Important: add all signals before adding methods/slots!

...and more

- Enumerators
- Class Info
- Static meta call function
- ...

Integrating our meta object

- Create the QMetaObject

```
QMetaObjectBuilder b;
...
QMetaObject *mo = b.toMetaObject();
```

- Implement metaObject()

```
const QMetaObject *MyClass::metaObject() const {
   return mo;
}
```

- Cleanup – free() instead of delete!

```
free(mo);
```

DIY qt_metacall()

Introspection is done, now we need behavior.

```
int qt_metacall(QMetaObject::Call c, int id, void **argv)
{
  switch (c) {
    case QMetaObject::ReadProperty: …
    case QMetaObject::WriteProperty: …
    case QMetaObject::InvokeMetaMethod: …
    case …
  }
}
```

Properties

- Property index:
  - if less than our property count: handle it ourselves and return -1
  - otherwise subtract property count from `id`, and let the base class handle it
- Property value:
  - `argv[0]` is a pointer to the value data
  - `argv[2]` is an int* for the return value of setProperty()

QVariant Data Pointers

- Returning values:
  - Known type: `*reinterpret_cast<T*>(argv[0]) = value`
  - QVariant: `QMetaType::construct(typeid, argv[0], value.data())`
- Receiving values:
  - Known type: `*reinterpret_cast<T*>(argv[0])`
  - QVariant: `QVariant(typeid, argv[0])`
- Special case: QVariant
  - `argv[0]` points to QVariant directly in this case

Calling Methods

- Method index: same as for properties
  - if smaller than method count: handle it ourselves and return -1
  - otherwise: subtract method count and let base class handle it
- Arguments:
  - `argv[0]`: return value
  - `argv[i]`: i-th argument
  - QMetaType::create/destroy can be handy to manage copies

Method Arguments

- Common task: turn void** arguments into a QVariantList
  - Never store `argv` itself, most likely invalid after you return!
- For each argument:
  - Determine its type id
    - Your meta object should have that
    - `QMetaType::type()` is useful if type is only available as a string
    - Alternative for incoming signals: `sender()` + `senderSignalIndex()`
  - Create a copy in a QVariant: `QVariant(typeid, argv[i])`

Emitting Signals

- Usual `moc`-generated signal stubs don't exist
- Connections to signals can nevertheless be established
- Use QMetaObject::activate() to call all connected slots

```
QMetaObject::activate(this, metaObject(), id, argv);
```

...and more

- We only covered the bare minimum, also available when needed:
  - Reset property
  - Query property details
  - Object constructions
  - Meta type registration
  - Find method index for a function pointer (for static connects)

Using a dynamic object

- C++: generic QObject API
  - `property()` / `setProperty()`
  - string-based `connect()`
  - `invokeMethod()`
- QML: no difference to static objects!

Do we really need the meta object?

- Problem: slot signature only known at connect()-time
  - QSignalSpy – generic interception of signal arguments
  - create a more powerful signal mapper
  - ...
- Approach:
  - only override qt_metacall()
  - use index-based QMetaObject::connect(), doesn't verify slot existence!
  - introspect sender object

Warranty void?

- Much of this is used in `moc`-generated code, thus effectively part of the ABI guarantee
- Meta objects have a revision number for backward compatibility
- If really needed, copy QMetaObjectBuilder
  - Use lowest Qt version you want to support

Conclusion

- QObjects are even more powerful than what's in the docs
- Useful for:
  - Bindings to dynamic languages
  - Nice scripting APIs
  - QML-based DSLs
  - Integrating IPC
  - Generic signal interception
  - ...

Questions?