Qt: Contexte, événements (signaux/slots) et propriétés

François Delobel

Master TechMed et Setsys, Université Clermont-Auvergne

Plan

Aperçu des spécificités des `QObject`

- Organisés en forêt d'objets (ensemble d'arbres).
- La destruction d'un `QObject` entraîne la destruction de tous ses fils.
- Bénéficient de capacité d'introspection (voir `QObject::metaObject` et `QMetaObject`).
- Peuvent avoir des *propriétés*, y compris des propriétés *dynamiques*. Voir `property()` et `setProperty()`.

UN QObject n'est pas copiable

- Un QObject n'est pas copiable.
- *Un QObject n'est pas copiable.*
- **Un QObject n'est pas copiable.**
- Ses dérivés ne seront pas non plus copiables!
- On ne passe donc pas un `QObject` par copie! (ni retour)
- Choix?

Signaux et slots

- Le *métacompilateur* (MOC) permet de doter les `QObject` de *signaux* et *slots*.
- Mécanisme fondamental pour communication UI/métier. Présent en QML, C++, Javascript.

Signal

- Ce qui peut être signalé, événement.
- Peut avoir des arguments
- Comme un prototype de méthode

Slot

- Ce qui peut être appelé par un signal.
- Comme une méthode *avec un corps*. Peut être appelé comme n'importe quelle méthode.
- Peut être virtuel.

Connexion d'un signal à un slot

- On connecte le signal *d'un objet* au slot d*'un objet*.
- On peut connecter un signal à plusieurs slots, ou à aucun.
  - ▷ On peut bien sûr ne pas connecter un signal.
- On peut déconnecter un slot d'un signal.
- On peut connecter un signal à un slot qui a moins d'arguments (on en ignore).
- On peut connecter un signal à un autre signal.
- L'appel des slots est *synchrone*
  - ▷ Ne pas faire des slots qui durent longtemps!
- L'appelé ignore l'identité de l'appelant.
  - ▷ Favorise le *découplage*
    - → Mais peut être contourné!

Définition de signaux et slots en C++

```
#include <QObject>
// Stolen from official Qt Doc :)
class Counter : public QObject   // Heritage direct ou indirect
{
    Q_OBJECT // !!!!! NE PAS OUBLIER !!!

public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }

public slots:
    void setValue(int value); // Il y a un corps pour setValue

signals:
    void valueChanged(int newValue); // Mais pas pour valueChanged

private:
    int m_value;
};
```

Envoi d'un signal

Le corps du *slot* `setValue` est utilisé pour émettre un signal.

```
void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}
```

Connexion d'un signal à un slot

```
Counter a, b;

// Syntaxe moderne du connect
QObject::connect(&a, &Counter::valueChanged,
                 &b, &Counter::setValue);

// Ancienne syntaxe (<= Qt5 )
QObject::connect(&a, SIGNAL(valueChanged(int)),
                 &b, SLOT(setValue(int)));


a.setValue(12);     // a.value() == 12, b.value() == 12
b.setValue(48);     // a.value() == 12, b.value() == 48
```

Signaux, slots et C++11

Connexion possible d'un slot a une lambda.

- Version de `connect` à trois paramètres (objet sender, signal, lamda).
- Au passage: `destroyed` est le signal émis à la destruction de tout objet Qt.

```
connect(sender, &QObject::destroyed,
    [this](){ delete this->otherObject;
              this->otherObject=nullptr; });
```

Définir des signaux dans QML

- On peut définir des signaux en QML.

```
Rectangle {
    signal trigger
    signal send (string notice)
    signal perform (string task, variant object)
}
```

Traitement des signaux en QML

- L'ajout d'un signal génère automatiquement l'ajout des *signal handlers* correspondants
- *handler* nommé on+*nomSignal*

```
onTrigger: console.log("trigger signal emitted")

onSend: {
    console.log("send signal emitted with notice: " + notice)
}

onPerform: console.log("perform signal emitted")
```

Signaux de modification des propriétés

- En cas de modification d'une *propriété* d'un objet QML, un signal est automatiquement émis.

```
Rectangle {
    id: sprite
    width: 25; height: 25
    x: 50; y: 15

    onXChanged: console.log("x prop changed, emitted xChanged signal")
    onYChanged: console.log("y prop changed, emitted yChanged signal")
}
```

Émission de signaux en QML

- Similaire à l'appel d'une méthode.
- Utiliser l'`id` de l'objet.

```
Rectangle {
    id: messenger

    signal send( string person, string notice)

    onSend: {
        console.log("For " + person + ", the notice is: " + notice)
    }

    Component.onCompleted: messenger.send("Tom", "the door is open.")
}
```

`Component.onCompleted` est automatiquement émis quand un composant a fini de s'initialiser.

Connecter un signal à une méthode en QML

- Toute méthode peut-être connectée en QML: pas besoin de slots.
- Tout signal a une méthode `connect`
- On utilise des méthodes javascript.

```
Rectangle {
    id: relay

    signal sendNote( string person, string notice)
    onSendNote: console.log("Send signal to: " + person + ", " + notice)

    Component.onCompleted: {
        relay.sendNote.connect(sendToPost)
        relay.sendNote.connect(sendToEmail)
        relay.sendNote("Tom", "Happy Birthday")
    }

    function sendToPost(person, notice) {
        console.log("Sending to post: " + person + ", " + notice)
    }

    function sendToEmail(person, notice) {
        console.log("Sending to email: " + person + ", " + notice)
    }
}
```

Q_INVOKABLE: Appeler des méthodes depuis QtQuick

- Simplement préfixer le prototype de la méthode par `Q_INVOKABLE`
- `Q_INVOKABLE void getDetails(QString id);`
- Le MOC va alors rajouter *ce qu'il faut* au code pour que la méthode soit invocable depuis QML.

But du jeu: appeler des méthodes C++ depuis QML

- Problème: pour appeler une méthode, il faut un objet!!
- Idée: rendre l'objet C++ visible depuis QML
- Solution: récupérer un *contexte* et y injecter la valeur désirée, avec un nom.

```
class CallableClass : public QObject
{
 Q_OBJECT
 // ...
public slots:
 void cppMethod() { qDebug("C++ method called!"); }
};

// ...
CallableClass callMe;
context->setContextProperty("cppObject", &callMe);
```

Dans QML :

```
MouseArea {
 onClicked: { cppObject.cppMethod();  }
}
```

Les vues QML depuis C++: QQuickView

- `QQuickView`: vue d'un fichier QML.
- *contexte*: ensemble d'objets visibles depuis les différents langages.
- Exemple: Afficher un objet QML et fixer une propriété.
- Une propriété peut contenir des `QObject` ou des `QVariant`.

```
// MyItem.qml
import QtQuick 2.0

Text { text: currentDateTime }
```

```
// main.cpp
QQuickView view;
view.rootContext()->setContextProperty("currentDateTime",
                                       QDateTime::currentDateTime());
view.setSource(QUrl::fromLocalFile("MyItem.qml"));
view.show();
```

Exporter ses propres objets dans le contexte

```
class ApplicationData : public QObject { // La classe est bien un QObject
    Q_OBJECT                             // Vous n'avez pas oublie, hein?
signals:
    void dataChanged();
public:
    Q_INVOKABLE QDateTime getCurrentDateTime() const {  // Notez le prefixe
        return QDateTime::currentDateTime();
    }
};

int main(int argc, char *argv[]) {
    QGuiApplication app(argc, argv);

    QQuickView view;  // On cree une vue

    ApplicationData data;    // L'objet a partager
    view.rootContext()->setContextProperty("applicationData", &data); // !!!

    view.setSource(QUrl::fromLocalFile("MyItem.qml"));
    view.show();

    return app.exec();
}
```

Invocation des méthodes et connexions aux signaux de C++

- Crée un composant QML qui accède à l'objet du contexte et invoque une méthode
- Connecte le signal `dataChanged` de l'objet `applicationData` à un bout de javascript.

```
Text {
    text: applicationData.getCurrentDateTime()
                        // L'objet est dans le contexte
                       // et la methode est invokable

    Connections {      // On declare une nouvelle connexion
        target: applicationData       // Objet emetteur
        onDataChanged: console.log("The application data changed!")
    }
}
```

Créer une application QML depuis C++ avec `QQmlApplicationEngine`

- Crée à la fois un `QMLEngine`, et un `QMLComponent`.
- Hérite de `QMLEngine`.
- Gère automatiquement les traduction.
- La fermeture de la vue ferme l'application.
- Méthode `rootContext()` pour récupérer le contexte.

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine("main.qml");
    return app.exec();
}
```

Exemple volé à wisol.ch (“Connect Qt QML and C++”)

```
// L'objet metier Receiver.h/cpp
#ifndef RECEIVER_H
#define RECEIVER_H
#include <QObject>
class Receiver : public QObject
{
  Q_OBJECT
  public:
    explicit Receiver(QObject *parent = 0);
  signals:
    void sendToQml(int count);
  public slots:
    void receiveFromQml(int count);
};
#endif // RECEIVER_H
```

Côté QML

```
// main.qml
import QtQuick 2.2
import QtQuick.Window 2.1

Window {
  Connections {
    target: receiver
    onSendToQml: {
    console.log("Received in QML from C++: " + count)
    }
  }
  MouseArea {
    anchors.fill: parent
    onClicked: {
      receiver.receiveFromQml(42);
    }
  }
}
```

Le main, l'application, le contexte

```cpp
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include "receiver.h"

int main(int argc, char *argv[])
{
  QGuiApplication app(argc, argv);
  QQmlApplicationEngine engine;

  Receiver receiver;

  QQmlContext* ctx = engine.rootContext();
  ctx->setContextProperty("receiver", &receiver);

  engine.load(QUrl(QStringLiteral("qrc:///main.qml")));

  return app.exec();
}
```

Propriétés

- Comme des attributs, mais plus puissant.
- Réservés aux `QObjects`. Propriétés liées à la métaclasse, avec du code automatiquement ajouté par la méta-compilation.
  - Ajout automatique d'accesseurs, de signaux...
  - Beaucoup de mots clé car beaucoup de subtilités sémantiques.
- Permet (entre autres) de communiquer entre QML et C++.

Les propriétés: Q_PROPERTY

- Macro, surcouche aux langages (C++: MOC)

```
Q_PROPERTY(type name // Declaration attibut
  (READ getFunction // Getter function
   [WRITE setFunction]  // Setter function
  |MEMBER memberName  // Association avec variable si pas READ !!!
                 //
      [(READ getFunction | WRITE setFunction)])
               // Mais qui peuvent etre redefinis
  [RESET resetFunction] // Remet la valeur par defaut
  [NOTIFY notifySignal] // Signal envoye qd changt
  [REVISION int=0]     // Revision number
  [DESIGNABLE bool=t] // Visible dans designer
  [SCRIPTABLE bool=t] // Accessible par script
  [STORED bool=t] // Stocke si sauvegarde etat objet
  [USER bool=f]  // Si manipulee par utilisateur
  [CONSTANT] // propriete constante
  [FINAL]) // Pas redefinissable
```

Les propriétés: Example

```
class Session : public QObject {
  Q_OBJECT
  Q_PROPERTY ( QString username  // Une propriete username
      READ username  // un getter username
      WRITE setUsername // un setter setUsername
      NOTIFY usernameChanged ) // un signal envoye quand change
public:
  QString username() const;
  void setUsername(const QString &newusername);
signals:
  void usernameChanged(QString username);
private:
  QString m_username;
};
```

Les propriétés: implémentation des méthodes

```
QString Session::username() const {
  return m_username;
}

void Session::setUsername(const QString &newusername) {
  if ( newusername != m_username ) {
    m_username = newusername;
    emit usernameChanged(m_username);
  }
}
```

Utilisation des propriétés

Appel des getters et setters définis

```
Session *s = new Session;
s->setUsername(tr("Evangelion"));  // setter
qDebug() << s->username();             // getter
```

Utilisation de setProperty et property

```
Session *s = new Session;
s->setProperty( "username",tr("Icari") );
```

- Attention à bien respecter l'orthographe de l'attribut, casse comprise.

Qt5 introduit MEMBER

Ce que n'est *pas* `MEMBER`

- N'ajoute pas de getter et de setter automatiquement pour la propriété!

`MEMBER`, un outil de communication inter-paradigme

- Sert à partager la propriété entre les différents langages de Qt.
  - Nécessaire (avec `NOTIFY`) quand on communique avec QML sans passer par des méthodes.
- Envoi tout seul le `NOTIFY` quand la propriété est altérée.
- Du côté `C++`, on utilise directement la variable.
- Ont peut ajouter un `WRITE` ou un `READ` mais pas les deux en même temps.
- Mais il est possible d'utiliser l'accès dynamique aux propriétés (`setProperty`).

Utilisation de MEMBER

```
    Q_PROPERTY(QColor color MEMBER m_color NOTIFY colorChanged)
    Q_PROPERTY(qreal spacing MEMBER m_spacing NOTIFY spacingChanged)
    Q_PROPERTY(QString text MEMBER m_text NOTIFY textChanged)
    ...
signals:
    void colorChanged();
    void spacingChanged();
    void textChanged(const QString &newText);

private:
    QColor  m_color;
    qreal   m_spacing;
    QString m_text;
```

Propriétés dynamiques

- Les propriétés peuvent être crées dynamiquement (exécution).
- Aussi avec `setProperty` si le nom de la propriété n'existe pas.
- Problème du type...

Utilisation de `setProperty` et `property` pour créer une propriété

```
Session *s = new Session;
s->setProperty( "nouvellepropriete",tr("babus") );
qDebug() << s->property("nouvellepropriete");
// QVariant(QString, "babus")
Session *s = new Session;
s->setProperty( "nouv2",42 );
qDebug() << s->property("nouv2");
// QVariant(int, 42)
```

- Attention à bien respecter l'orthographe de l'attribut, casse comprise.

Petit inconvénient des propriétés dynamiques: `QVariant`

- `QVariant`: type “générique” permettant de stocker la plupart des types usuels de Qt.
- Utilisé pour copier/stocker des données dont ont ne connaît pas le type.

```
Session *s = new Session;
QObject *o = s;
qDebug() << o->property("username");
// QVariant(QString, "Icari")
qDebug() << o->property("username").toString();
// "Icari"
```

Le QVariant en bref (Extrait de la doc)

```
QDataStream out(...);
QVariant v(123);                // The variant now contains an int
int x = v.toInt();              // x = 123
out << v;                       // Writes a type tag and an int to out
v = QVariant("hello");          // The variant now contains a QByteArray
v = QVariant(tr("hello"));      // The variant now contains a QString
int y = v.toInt();              // y = 0 since v can't be converted to an int
QString s = v.toString();       // s = tr("hello")  (see QObject::tr())
out << v;                       // Writes a type tag and a QString to out
...
QDataStream in(...);            // (opening the previously written stream)
in >> v;                        // Reads an Int variant
int z = v.toInt();              // z = 123
qDebug("Type is %s",            // prints "Type is int"
       v.typeName());
v = v.toInt() + 100;            // The variant now hold the value 223
v = QVariant(QStringList());
```

Que peut-on mettre dans un QVariant?

```
QRegExp, QRegularExpression , QUrl , QEasingCurve , QUuid ,
QModelIndex ,QJsonValue , QJsonObject , QJsonArray ,
QJsonDocument , QVariant &, Type, int typeId, void *,
QVariant , QDataStream , int, uint, qlonglong, qulonglong,
bool, double, float, char *, QByteArray , QBitArray ,
QString , QLatin1String, QStringList , QChar, QDate , QTime,
QDateTime , QList<QVariant> , QMap<QString, QVariant> ,
QHash<QString, QVariant> , QSize , QSizeF , QPoint , QPointF,
QLine , QLineF , QRect , QRectF , QLocale
```

Vues et modèle, l'autre façon de coopérer

- Modèle en C++ (ou éventuellement en QML)
  - `QStringList`
  - `QObjectList`
  - `QAbstractItemModel` sous-classé.
- Vues en QML, chargées de la présentation
  - `ListView` avec délégué.

Exemple d'utilisation d'une QStringList

```
QStringList dataList;   // On meuble de donnees
dataList.append("Item 1");
dataList.append("Item 2");
dataList.append("Item 3");
dataList.append("Item 4");

// On partage avec QML
QQuickView view;
QQmlContext *ctxt = view.rootContext();
ctxt->setContextProperty("myModel", QVariant::fromValue(dataList));
```

Comment afficher le modèle en QML?

- `model` est l'objet qui contient les données.
- `delegate` est le composant chargé de faire affichage d'une valeur.
  - modelData contiendra une valeur

```
ListView {
    width: 100; height: 100

    model: myModel
    delegate: Rectangle {
        height: 25
        width: 100
        Text { text: modelData }
    }
}
```

Une classe, une liste d'objets

```
class DataObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString name READ name WRITE setName NOTIFY nameChanged)
    Q_PROPERTY(QString color READ color WRITE setColor NOTIFY colorChanged)
};
int main(int argc, char ** argv)
{
    QGuiApplication app(argc, argv);

    QList<QObject*> dataList;
    dataList.append(new DataObject("Item 1", "red"));
    dataList.append(new DataObject("Item 2", "green"));

    QQuickView view;
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    QQmlContext *ctxt = view.rootContext();
    ctxt->setContextProperty("myModel", QVariant::fromValue(dataList));
    // Notez le besoin de construire un QVariant^^
```

Une view...

... C'est tout simple!

```
ListView {
    width: 100; height: 100

    model: myModel
    delegate: Rectangle {
        height: 25
        width: 100
        color: model.modelData.color
        Text { text: name }
    }
}
```