Signaux et slots avec Qt4

par Matthieu Brucher (http://matthieu-brucher.developpez.com/) (Blog)

Date de publication : 25/09/2007

Dernière mise à jour : 20/12/2007

Le mécanisme des signaux et slots de Qt est très avancé et permet de communiquer entre les threads. Cela est unique pour le moment et est aussi la cause de l'existence de moc et de l'utilisation de qmake pour construire les projets Qt.

Introduction

Qt propose depuis son origine un mécanisme de communication entre les objets appelé signaux et slots. Cela a nécessité la présence d'un préprocesseur appelé moc car des mots-clés supplémentaires ont été ajoutés. L'intérêt de qmake réside dans le fait que les fichiers nécessitant le préprocesseur sont automatiquement détectés avec cet outil de construction.

I - Création et utilisation des signaux et slots

Le mécanisme nécessite de vivre dans un QObject. C'est lui qui prend en charge le dispatching des signaux aux différents slots existants.

Ceci entraîne une restriction dans l'utilisation des QObject, il n'est pas possible d'hériter de deux classes héritant de QObject. De plus, dans le cas d'un héritage multiple, il faut hériter de QObject (ou d'un descendant) en premier.

La déclaration des outils nécessaires se fait par la macro Q_OBJECT. C'est grâce à sa présence que qmake saura que le préprocesseur devra être utilisé. Une fois la macro utilisée, les signaux et les slots peuvent être créés.

Un bloc de signaux est déclaré par le mot-clé **signals**. Chaque signal déclaré est un simple prototype, l'implémentation étant réalisée par le préprocesseur moc.

Les slots sont aussi regroupés en bloc à l'aide du mot-clé **slots**, pouvant être public, protégé ou private. Un slot est une fonction qui sera appelée lorsqu'un signal qui lui est relié est déclanché.

Le fait qu'une classe dérivant de QObject déclare Q_OBJECT entraîne la virtualisation des signaux. En fait, si une classe déclare un slot et qu'une sous-classe surcharge ce slot et définit Q_OBJECT, quoiqu'il arrive, c'est ce dernier slot qui sera utilisé.

*Il est possible d'émettre un signal à l'aide de l'instruction **emit**. Cela permet de tester le fonctionnement des signaux.*

La création de la glue entre signal et slot est réalisée par la fonction connect(). Celle-ci prend en paramètre un pointeur vers l'objet contenant le signal, la fonction de signal elle-même appelée par la macro SIGNAL(), un pointeur vers l'objet contenant le slot et le slot appelé par la macro SLOT(). Un signal peut avoir plusieurs arguments auquel cas le slot associé ne peut pas avoir plus d'arguments que le signal. Il est aussi possible de connecter un signal sur un autre signal.

La déconnexion d'un signal se fait avec la fonction disconnect(). Il est aussi possible d'auto-connecter des signaux à des slots d'un objets en utilisant comme nom de slot on_(nom de l'objet)_(signal émanent de l'objet). Cela est fait lors de l'appel à QMetaObject::connectSlotsByName(objet).

L'exemple suivant crée une simple application avec une fenêtre et un bouton. Lorsque le bouton est appuyé, le signal **clicked()** est envoyé à la fenêtre et le slot **s_cliked()** est exécuté. Dans ce slot, un texte est demandé et un nouveau signal est envoyé, avec du texte cette fois-ci, à un autre slot qui va afficher ce texte.

exemple.h

```
#include <QMainWindow>
#include <QString>

class QPushButton;
class QDialogButtonBox;

class Exemple: public QMainWindow
{
  Q_OBJECT

public:
  Exemple();
```

exemple.h

```
signals:
  void pseudoclick(QString& texte);

public slots:
  void s_clicked();
  void s_clicked_texte(QString& texte);

private:
  QPushButton* button;
};
```

exemple.cpp

```
#include "exemple.h"
#include <QApplication>
#include <QPushButton>
#include <QInputDialog>
#include <QMessageBox>

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

  Exemple exemple;

  exemple.show();
  return app.exec();
}

Exemple::Exemple()
  :QMainWindow()
{
  button = new QPushButton("Hello!", this);

  connect(button, SIGNAL(clicked()), this, SLOT(s_clicked()));
  connect(this, SIGNAL(pseudoclick(QString&)), this, SLOT(s_clicked_texte(QString&)));
}

void Exemple::s_clicked()
{
  QString texte = QInputDialog::getText(this, "Demande", "Entrez du texte");
  emit pseudoclick(texte);
}

void Exemple::s_clicked_texte(QString& texte)
{
  QMessageBox message(QMessageBox::Information, "Ceci est un test d'information", "Le texte\n" +
 texte + "\na été entréé");
  message.exec();
}
```

*Voici les fichiers : **exemple.cpp** et **exemple.h**.*

Pour compiler l'exemple, exécutez qmake -project && qmake && make (ou nmake avec Visual Studio).

Il est possible de connecter plusieurs signaux à plusieurs slots. La seule chose à savoir est que l'ordre d'appel n'est pas assuré.

II - Utilisation multithreadée des signaux et slots

Contrairement aux signaux et slots de Boost, les signaux et slots de Qt peuvent être utilisés entre threads.

Tout d'abord, il faut savoir qu'un objet appartient à un thread particulier. Il est possible de changer l'affinité de l'objet. C'est cette affinité qui permet de savoir la méthode d'appel à utiliser. Lorsqu'un signal est connecté à un slot, si l'objet du signal appartient au même thread que l'objet du slot, la connexion est réalisée à l'aide d'un simple appel de la fonction. Lorsque ce n'est pas le même thread, le signal est posté à l'autre thread qui sera chargé de l'activation du slot lorsque la pompe à message du thread sera interrogée.

Ce genre de connexion est très utile lors de la réception et l'envoi de données sur le réseau, pour la communication entre GUI et un thread de calcul, ...

Il est possible de forcer le type de connexion à l'aide d'un cinquième argument à connect(). Qt::AutoConnection détecte automatiquement quelle connexion utiliser, Qt::DirectConnection fait automatiquement un appel standard au slot, Qt::QueuedConnection poste un message dans la pompe à message du thread de l'objet cible et Qt::BlockingQueuedConnection bloque le thread appelant en attendant que le message ait été effectivement validé.

Lors de la communication entre threads avec l'envoi d'un message, les arguments sont ***sérialisés*** *automatiquement. Pour un type usuel ou Qt, l'opération est déjà prévue, pour les autres types, il suffit d'enregistrer le type grâce à la fonction template qRegisterMetaType<>(nom).*