Les signaux et slots dans Qt 5

Les signaux-slots dans Qt 4

L'idée des signaux-slots est de créer un “lien” particulier entre deux fonctions de deux classes indépendantes, de façon à ce que lorsque l'on appelle la fonction du premier objet, la fonction du second objet est automatiquement appelée. La première fonction s'appelle “signal”, la seconde “slot”, le lien entre les deux s'appelle une “connexion”.

```
class A {
public:
    void fonction_de_A() const {}
};

class B {
public:
    void fonction_de_B() const {}
};

int main() {
    A un_objet_A;
    B un_objet_B;

    // création d'une connexion entre les deux fonctions
    creer_connexion(un_objet_A, fonction_de_A, un_objet_B,
fonction_de_B);

    // lorsque l'on appelle la fonction de A, la fonction de
B est
    // automatiquement appelée
    un_objet_A.fonction_de_A(); // appelle
un_objet_B.fonction_de_B()
}
```

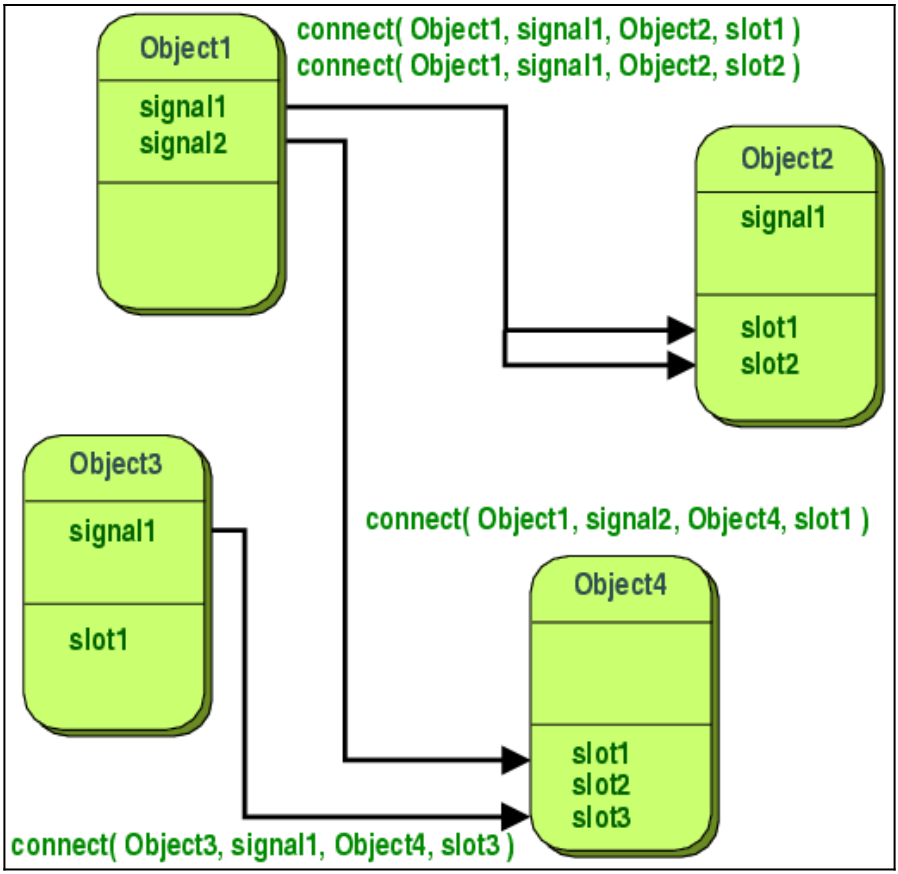
Avec ce type d'approche, vous pouvez utiliser autant de classes et de fonctions que vous souhaitez, celles-ci seront indépendantes. `A` n'a pas

besoin de connaître `B` et `B` n'a pas besoin de connaître `A`. On parle dans cette situation de “couplage faible”. Nous allons voir maintenant comment cela s'utilise dans Qt, mais sachez qu'il est également possible d'utiliser cette technique avec d'autres bibliothèques (par exemple, en utilisant Boost.Signals) ou en utilisant des objet-fonctions comme argument (quelques mots-clés : `std::function`, *functor object*, *lambda function*, etc).

Le système de signaux et slots de Qt est relativement simple : lorsqu'un signal est émis avec le mot clé `emit`, tous les slots qui sont connectés à ce signal sont exécutés. Une connexion est créée en utilisant la fonction `QObject::connect`, en donnant les arguments suivants :

- un pointeur vers l'objet émetteur ;
- le nom du signal et la liste des types d'argument de la fonction signal (en utilisant la macro `SIGNAL`) ;
- un pointeur vers l'objet récepteur ;
- le nom du slot et la liste des types d'arguments de la fonction slot (en utilisant la macro `SLOT`).

Il est possible de connecter plusieurs signaux à un même slot, un signal à plusieurs slots ou un signal avec un signal.



La correspondance entre les arguments des signaux et slots est vérifiée comme une chaîne de caractères, lors de l'exécution. Ainsi, si vous connectez un `signal(float)` avec une `slot(double)`, la connexion sera refusée puisque les arguments ne correspondent pas, malgré le fait qu'un `float` peut être converti en `double` sans problème.

```
QAction* action = new QAction(this);
QWidget* widget = new QWidget(this);
QObject::connect(
    action, SIGNAL(triggered()), // connecte le signal
triggered() de QAction
    widget, SLOT(show()));       // au slot show() de
QWidget.
```

```
// ce code permet donc d'afficher le QWidget lorsque
l'utilisateur active la QAction.
```

La chaîne de caractères utilisée pour identifier les paramètres des signaux et slots est une chaîne avec une signature standardisée (voir la fonction QMetaObject::normalizedSignature). Par exemple :

```
signal(QString const&, float)
// devient
signal(QString,float)
```

Les classes de Qt fournissent de nombreux signaux et slots par défaut (la liste des signaux et slots des classes Qt est indiquée dans la documentation de Qt). Vous pouvez également créer vos propres signaux et slots dans vos classes, en respectant les règles suivantes :

- la classe doit dériver de `QObject` (directement ou indirectement) ;
- il faut appeler la macro `Q_OBJECT` au début de la classe ;
- les slots sont des fonctions classiques déclarées avec le mot-clé `slots` ;
- les signaux sont des fonctions non implémentées déclarées avec le mot-clé `signals`.

Concrètement, une classe doit ressembler à quelque chose comme cela :

```
#include <QObject>

class UneClasse : public QObject
{
    Q_OBJECT

public slots:
    void unSlot() { /* code du slot */ }

signals:
    void unSignal();
```

```
};
```

Comme le principe est de pouvoir connecter n'importe quelle classe et n'importe quels signaux et slots (à partir du moment où les paramètres de fonctions corresponds), vous pouvez sans problème connecter les classes Qt entre elles, avec vos propres classes ou vos propres classes entre elles.

Remarque importante : il faut obligatoirement déclarer les classes dérivant de `QObject` dans un fichier `.h` et non dans un fichier `.cpp`, sinon le pré-compilateur *moc* de Qt ne pourra pas fonctionner.

Les signaux et slots dans Qt 5

Dans Qt 4, il est possible de connecter uniquement les fonctions déclarées comme signaux et slots dans la classe, comme indiqué dans les codes d'exemple précédant. De plus, il faut que les déclarations des fonctions dans les macros `SIGNAL` et `SLOT` correspondent exactement, ce qui interdit l'utilisation de `typedef`/`using`, les espaces de noms ou les conversions implicites des types. Pour terminer, la connexion étant créé lors de l'exécution, il n'est pas possible de savoir dès la compilation si les signaux et slots sont compatibles, ce qui retard le diagnostic des problèmes.

Qt 5 lève ces restrictions, en utilisant des pointeurs de fonctions, ce qui permet une vérification a la compilation. De plus, il est maintenant possible de connecter directement des pointeurs de fonctions ou d'utiliser des fonctions lambdas.

Connecter des pointeurs de fonctions membres

L'utilisation des macros `SIGNAL` et `SLOT` permet de créer une connexion à l'exécution. La vérification de la compatibilité des paramètres entre le signal et le slot est fait à ce moment là (ce qui pose un problème de testabilité).

Avec Qt 5, une nouvelle syntaxe permet de créer des connexions à la compilation, ce qui permet de vérifier la compatibilité des paramètres à la compilation et de gagner en performances.

Pour rappel, la syntaxe pour obtenir un pointeur sur une fonction membre s'écrit :

```
&CLASS::FUNCTION
```

Par exemple pour la fonction `show` de la classe `QWidget` :

```
&QWidget::show
```

Fonction non surchargée

Une fonction n'est pas surchargée lorsqu'elle n'existe qu'en une seule version dans une classe :

```
class Object
{
public:
    void f();
};
```

La syntaxe pour connecter une action à un widget devient donc :

```
auto a = new QAction(this);
auto w = new QWidget(this);
QObject::connect(
    a, &QAction::triggered,  // connecte le signal
triggered() de QAction
    w, &QWidget::show);      // au slot show() de QWidget.
```

Les classes émettrices et réceptrices doivent dériver de `QObject`, mais il n'est plus nécessaire de déclarer les fonctions utilisées comme slots avec le mot clé `slots`. (La macro `Q_OBJECT` n'est pas indispensable dans la classe réceptrice pour les slots, mais il est préférable de la mettre quand même pour les `QMetaObject`).

```
class Sender : public QObject {
    Q_OBJECT

signals:
    void send(int i);
};

class Receiver : public QObject {
    Q_OBJECT

public:
    void receive(int i = 0) { qDebug() << "receive:" << i; }
};

auto sender = new Sender;
auto receiver = new Receiver;
QObject::connect(sender, &Sender::send, receiver, &Receiver
::receive);
```

L'avantage de cette écriture est que la compatibilité des paramètres est effectuée lors de la compilation et une conversion implicite est réalisée si nécessaire. Ainsi, il est possible de connecter un signal `mon_signal(float)` vers un slot `mon_slot(float)`, mais également vers un slot `mon_slot(double)`.

Fonctions surchargées

Des fonctions sont surchargées lorsqu'elles existent en plusieurs versions dans une classe :

```
class Object
{
public:
    void f();
    void f(int);
    void f(QString);
};
```

Lorsqu'il existe plusieurs fonctions surchargées, le compilateur ne sait pas quelle fonction doit être utilisée. Il faut donc donner explicitement la signature de la fonction, en effectuant une conversion.

Il est possible de faire une conversion explicite (*cast*) avec `static_cast` pour spécifier quelle version utiliser :

```
static_cast<void (CLASS::*)(PARAMETERS)>(&CLASS::FUNCTION)
```

Par exemple pour la classe précédente :

```
static_cast<void (Object ::*)()>(&Object ::f)
static_cast<void (Object ::*)(int)>(&Object ::f)
static_cast<void (Object ::*)(QString)>(&Object ::f)
```

Autre exemple, pour connecter les fonctions `QComboBox::currentIndexChanged(int)` ou `QComboBox::currentIndexChanged(QString)`, il est possible d'écrire :

```
auto comboBox = new QComboBox;

// utilisation de : void currentIndexChanged(int index)
connect(
    comboBox,
    static_cast<void (QComboBox::*)(int)>(&QComboBox::
currentIndexChanged),
    /* ... */);

// utilisation de : void currentIndexChanged(const QString &
text)
connect(
    comboBox.data(),
    static_cast<void (QComboBox::*)(const QString &)>(&
QComboBox::currentIndexChanged),
    /* ... */);
```

Il est également possible d'utiliser à partir de Qt 5.7 la fonction qOverload en C++14 ou la classe `QOverload` en C++11.

```
struct Foo {
    void overloadedFunction();
```

```
    void overloadedFunction(int, QString);
};

// C++14
qOverload<>(&Foo:overloadedFunction)
qOverload<int, QString>(&Foo:overloadedFunction)

// C+11
QOverload<>::of(&Foo:overloadedFunction)
QOverload<int, QString>::of(&Foo:overloadedFunction)
```

Lorsqu'il existe des surcharges de fonctions constante et non constante, il faut utiliser les fonctions qConstOverload et qNonConstOverload en C++14 et `QConstOverload` et `QNonConstOverload` en C++11.

```
struct Foo {
    void overloadedFunction(int, QString);
};

// C++14
qConstOverload<>(&Foo:overloadedFunction)
qNonConstOverload<int, QString>(&Foo:overloadedFunction)

// C+11
QConstOverload<>::of(&Foo:overloadedFunction)
QNonConstOverload<int, QString>::of(&Foo:overloadedFunction)
```

Connecter des lambdas, fonctions libres et foncteurs

Il est également possible d'utiliser une fonction lambdas, fonctions libres et foncteurs comme slot dans une connexion. Par exemple, lorsque l'on souhaite connecter un signal `QPushButton::clicked` avec un slot `QLabel::setText(QString)` dans Qt 4, il n'est pas possible de créer directement la connexion.

```
connect(
    button, SIGNAL(clicked()),
```

```
    label,  SLOT(setText(QString));
    // erreur, les paramètres des fonctions ne correspondent
pas
```

Il faut donc écrire un intermédiaire, par exemple :

```
class MyWidget : public QWidget {
    Q_OBJECT

public:
    MyWidget(QObject* parent = 0)
        : QWidget(parent),
          label(new QLabel(this))
    {
        auto button = new QPushButton(this);
        connect(button, SIGNAL(clicked()), this, SLOT(
new_slot()); // Ok
    }

private slots:
    void new_slot() {
        Q_ASSERT(m_label);
        m_label->setText("Mon texte");
    }

private:
    QLabel* m_label = nullptr;
};
```

En utilisant les lambdas, il est maintenant possible d'appeler directement :

```
connect(button, &QPushButton::clicked, [this](){
    Q_ASSERT(label);
    m_label->setText("Mon texte");
});
```

Dans ce code, on capture le pointeur de l'object courant `this` et on appelle dans la fonction lambda la fonction `setText()`. Le résultat obtenu est identique au code précédant, mais il est possible de faire beaucoup d'autres choses dans la fonction lambda (par exemple

déconnecter tous les signaux ou parcourir tous les enfants de l'objet récepteur).

Si le compilateur utilisé ne supporte pas les *variadic template*, les signaux et slots doivent avoir moins de 6 paramètres.

De la même façon, pour les fonctions libres et les foncteurs :

```
QPointer<Sender> sender = new Sender;

// un foncteur
class MyObject {
public:
    void operator()(/* paramètres */) { ... }
};

MyObject object;
QObject::connect(sender, &Sender::unSignal, object);


// une fonction libre
void foo() { ... }

QObject::connect(sender, &Sender::unSignal, foo);
```

Supprimer les connexions

Pour supprimer une connexion, plusieurs méthodes sont possibles :

- appeler `QObject::disconnect`. Cette fonction permet de déconnecter manuellement une connexion en particulier ou plusieurs connexions ;
- supprimer l'émetteur ou le récepteur d'une connexion détruit automatiquement les connexions qui leur sont attachées (sauf cas particulier pour les fonctions lambdas, que l'on va voir ensuite).

Pour appeler `QObject::disconnect`, plusieurs syntaxes sont possibles,

comme indiqué dans la documentation :

```
auto sender = new Sender(this);
auto receiver = new Receiver(this);

// supprimer toutes les connexions liées à tous les signaux
d'un objet :
disconnect(sender, nullptr, nullptr, nullptr);
// ou
sender->disconnect();

// supprimer toutes les connexions liées à un signal en
particulier d'un objet :
disconnect(sender, &Sender::send, nullptr, nullptr);
// ou
sender->disconnect(&Sender::send);

// supprimer un récepteur en particulier :
disconnect(sender, nullptr, receiver.data(), nullptr);
// ou
sender->disconnect(receiver);
```

Une autre approche pour supprimer une connexion est d'utiliser l'objet QMetaObject::Connection retourné lors de la création d'une connexion :

```
QMetaObject::Connection connection = QObject::connect(...);
QObject::disconnect(connection);
```

En C++, il est important de tester la validité des pointeurs avant de les utiliser. C'est pour cela qu'il est préférable d'utiliser `QPointer` pour manipuler les objets dérivant de `QObject`. Cependant, les fonctions lambdas sont un peu spéciales : elles peuvent capturer le pointeur nu `this` dans la lambda et il n'est pas possible de vérifier si le pointeur est valide ou non dans le corps de la lambda.

```
void MyObject::foo() {
    QPointer<MyObject> object = new MyObject;

    connect(this, &MyObject::send, [object](){
        Q_ASSERT(object);      // ok, object est un QPointer,
la validation
```

```
        object->doSomething(); // permet de détecter un
pointeur invalide.
    });

    connect(this, &MyObject::send, [this](){
        Q_ASSERT(this);      // erreur, this est un pointeur
nu, la validation
        this->doSomething(); // ne permet pas de détecter un
pointeur invalide.
    });
}
```

Nouveautés du C++14

Le C++14 apporte quelques nouvelles syntaxes pour écrire des fonctions lambdas : les fonctions lambdas génériques et la capture étendue.

Ces syntaxes sont utilisables avec Qt 5.1 :

```
QPointer<Sender> sender = new Sender;
QPointer<Receiver> receiver = new Receiver;

// lambda générique
connect(sender.data(), &Sender::valueChanged, [receiver](
const auto &newValue) {
    Q_ASSERT(receiver);
    receiver->updateValue("senderValue", newValue);
});

// capture étendue
connect(sender.data(), &Sender::valueChanged, [receiver=
getReceiver()](const auto &newValue) {
    Q_ASSERT(receiver);
    receiver->updateValue("senderValue", newValue);
});
```

Résumé des syntaxes utilisables

```
// Qt 4
QMetaObject::Connection QObject::connect(
    const QObject * sender, const char * signal,
    const QObject * receiver, const char * method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

QMetaObject::Connection QObject::connect(
    const QObject * sender, const char * signal,
    const char * method,
    Qt::ConnectionType type = Qt::AutoConnection) const

// Qt 4.8
QMetaObject::Connection QObject::connect(
    const QObject * sender, const QMetaMethod & signal,
    const QObject * receiver, const QMetaMethod & method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

// Qt 5.0
QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    const QObject * receiver, PointerToMemberFunction method,
    Qt::ConnectionType type = Qt::AutoConnection) [static]

QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    Functor functor) [static]

// Qt 5.2
QMetaObject::Connection QObject::connect(
    const QObject * sender, PointerToMemberFunction signal,
    const QObject * context, Functor functor,
    Qt::ConnectionType type = Qt::AutoConnection) [static]
```

Remarque : toutes ces fonctions sont `static`, sauf la deuxième.

Conclusion

Vous pouvez télécharger un projet d'exemple montrant ces nouvelles

fonctionnalités en action : la page de téléchargement sur GitHub.

Les images et codes d'exemple sont issus en partie de la documentation de Qt5 disponible à cette page : Signals & Slots.

Mises à jour

- 12/09/2014 : correction de la syntaxe des lambdas, suite à une remarque de WinJérôme.
- 27/10/2014 : mise à jour pour Qt 5.4 et suppression de la partie sur le couplage (cela fera partie d'un article plus généraliste sur la communication inter-classes).
- 27/11/2014 : ajout de la partie sur le C++14.
- 21/09/2015 : utilisation de QPointer et Q_ASSERT, suppression des connexions, problème des pointeurs nus dans les fonctions lambdas.
- 05/02/2016 : ajout de l'article “How Qt Signals and Slots Work - Part 3” dans les sources a lire.

Sources

- Signals & Slots
- Differences between String-Based and Functor-Based Connections
- Qt Developer Days 2013 - Olivier Goffart - Signals and Slots in Qt 5
- Les articles de woboq :
  - Signals and Slots in Qt5 ;
  - How Qt Signals and Slots Work ;
  - How Qt Signals and Slots Work - Part 2 - Qt5 New Syntax ;
  - Interlude - QMetatype knows your types ;

    - How Qt Signals and Slots Work - Part 3 - Queued and Inter Thread Connections.
- C++14 : Le C++14 est arrivé et C++14 for Qt programmers.

Pour aller plus loin :

- La documentation des classes QSignalMapper, QSignalBlocker et QSignalSpy
- Benchmark for conception

Qt