

Connecter plusieurs signaux à un seul

par [Kinji1](#)

Date de publication : 08 février 2009

Dernière mise à jour :

Qt Quarterly est un journal électronique disponible exclusivement aux clients Qt. Chaque trimestre, nous envoyons un e-mail qui, nous l'espérons, ajoutera à votre expérience Qt, avec des articles de qualité écrits par des experts de Qt.

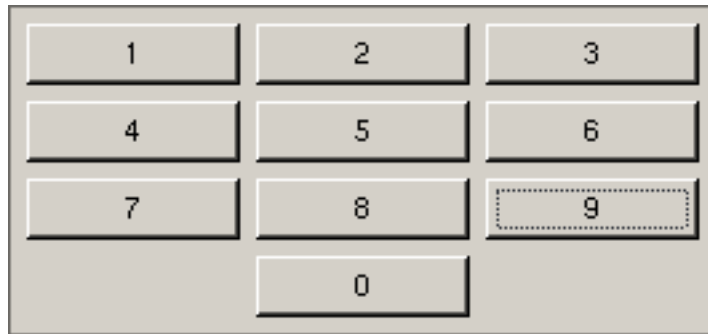
Developpez.com a reçu l'autorisation de Nokia afin de traduire ces articles.

I - Connecter plusieurs signaux à un seul.....	3
I-A - La solution triviale.....	3
I-B - L'approche sender().....	4
I-C - L'approche de la classe dérivée.....	5
I-D - L'approche du Signal Mapper.....	5
I-E - Le layout.....	7

I - Connecter plusieurs signaux à un seul

Par Jasmin Blanchette

Qt nous permet de connecter plusieurs signaux à un même slot. Cela peut être utile pour donner à l'utilisateur différentes manières de réaliser la même action. Cependant, nous voulons parfois que le slot se comporte légèrement différemment selon le widget qui l'a appelé. Dans cet article nous explorons diverses solutions, dont l'utilisation de QSignalMapper.



Pour illustrer le problème, nous implémenterons un widget représentant un clavier numérique (Keypad) qui a dix QPushButton, numérotés de 0 à 9, et un signal digitClicked(int) qui est émis quand l'utilisateur clique sur un bouton. Nous verrons quatre solutions à ce problème et nous présenterons leurs avantages respectifs.

I-A - La solution triviale

La solution la plus simple (et la plus bête) à notre problème est de connecter les dix signaux clicked() des objets QPushButton à dix slots nommés button0Clicked(), ..., button9Clicked(). Chacun de ces slots émettant le signal digitClicked(int) avec le paramètre correspondant, de 0 à 9. Voici la définition de la classe Keypad :

```
class Keypad : public QWidget
{
    Q_OBJECT
public:
    Keypad(QWidget *parent = 0);

signals:
    void digitClicked(int digit);

private slots:
    void button0Clicked();
    void button1Clicked();
    ...
    void button9Clicked();

private:
    void createLayout();

    QPushButton *buttons[10];
};
```

Voici le constructeur de Keypad :

```
Keypad::Keypad(QWidget *parent) : QWidget(parent)
{
    for (int i = 0; i < 10; ++i) {
        QString text = QString::number(i);
        buttons[i] = new QPushButton(text, this);
    }
}
```

```
connect(buttons[0], SIGNAL(clicked()), this, SLOT(button0Clicked()));  
...  
connect(buttons[9], SIGNAL(clicked()), this, SLOT(button9Clicked()));  
  
createLayout();  
}
```

Dans le constructeur nous créons les QPushButton et nous connectons fastidieusement chacun des signaux clicked() au slot privé correspondant.

```
void Keypad::button0Clicked()  
{  
    emit digitClicked(0);  
}  
  
...  
  
void Keypad::button9Clicked()  
{  
    emit digitClicked(9);  
}
```

Chaque slot émet simplement le signal digitClicked(int) avec un argument différent codé en dur.

Inutile de dire que cette approche n'est pas très flexible et est source d'erreurs. Elle est utilisable pour un petit nombre de connexions, mais même pour un clavier de 10 touches, le copier-coller est presque insupportable. Voyons donc une meilleure solution.

I-B - L'approche sender()

La prochaine étape est de fusionner les slots buttonNClicked() en un seul slot privé qui émet le signal digitClicked(int) avec le paramètre correct, selon le bouton qui a été cliqué. Ceci est possible en utilisant la fonction QObject::sender(), comme nous allons bientôt le voir. Le constructeur de Keypad devient alors :

```
Keypad::Keypad(QWidget *parent) : QWidget(parent)  
{  
    for (int i = 0; i < 10; ++i) {  
        QString text = QString::number(i);  
        buttons[i] = new QPushButton(text, this);  
        connect(buttons[i], SIGNAL(clicked()), this, SLOT(buttonClicked()));  
    }  
    createLayout();  
}
```

Et voici le code du slot buttonClicked() :

```
void Keypad::buttonClicked()  
{  
    QPushButton *button = (QPushButton *)sender();  
    emit digitClicked(button->text()[0].digitValue());  
}
```

Nous commençons en appelant sender() pour récupérer un pointeur sur le QObject qui a émis le signal déclenchant ce slot. Dans cet exemple, nous savons que l'émetteur est un QPushButton donc nous convertissons la valeur de retour de sender() en un QPushButton*. Puis nous émettons le signal digitClicked(int) avec la valeur affichée par le bouton.

Le défaut de cette approche est que nous avons besoin d'un slot privé pour réaliser le démultiplexage. Le code de buttonClicked() n'est pas très élégant; si vous remplacez subitement les QPushButton avec un autre type de widget et oubliez de changer la conversion, vous aurez un crash. De même, si vous changez le texte du bouton (par exemple, "NUL" au lieu de "0") le signal digitClicked(int) sera émis avec une valeur incorrecte. Enfin, l'utilisation de

sender() mène à des composants fortement couplés, ce que beaucoup considèrent comme une mauvaise technique de programmation. Ce n'est pas si mauvais dans cet exemple car le clavier connaît déjà les objets boutons, mais si buttonClicked() avait été un slot d'une autre classe, l'utilisation de sender() aurait eu l'effet malheureux de lier cette classe aux détails d'implémentation de la classe Keypad.

I-C - L'approche de la classe dérivée

Notre troisième approche ne requiert aucun slot privé dans la classe Keypad; à la place, nous nous assurons que les boutons émettent eux-mêmes le signal clicked(int) qui peut être directement connecté au signal digitClicked(int) de Keypad (Quand l'on connecte un signal à un autre, le signal cible est émis chaque fois que le premier signal est émis.) Ceci nécessite la dérivation de QPushButton comme cela :

```
class KeypadButton : public QPushButton
{
    Q_OBJECT
public:
    KeypadButton(int digit, QWidget *parent);

signals:
    void clicked(int digit);

private slots:
    void reemitClicked();

private:
    int myDigit;
};

KeypadButton::KeypadButton(int digit, QWidget *parent)
    : QPushButton(parent)
{
    myDigit = digit;
    setText(QString::number(myDigit));
    connect(this, SIGNAL(clicked()), this, SLOT(reemitClicked()));
}

void KeypadButton::reemitClicked()
{
    emit clicked(myDigit);
}
```

Chaque fois que QPushButton émet le signal clicked(), nous l'interceptons dans notre classe dérivée KeypadButton et émettons le signal clicked(int) avec le chiffre correct comme argument.

Le constructeur de Keypad ressemble alors à ceci :

```
Keypad::Keypad(QWidget *parent) : QWidget(parent)
{
    for (int i = 0; i < 10; ++i) {
        buttons[i] = new KeypadButton(i, this);
        connect(buttons[i], SIGNAL(clicked(int)), this, SIGNAL(digitClicked(int)));
    }
    createLayout();
}
```

Cette approche est à la fois plus flexible et plus propre, mais légèrement lourde à écrire, parce qu'elle nous force à dériver la class QPushButton.

I-D - L'approche du Signal Mapper

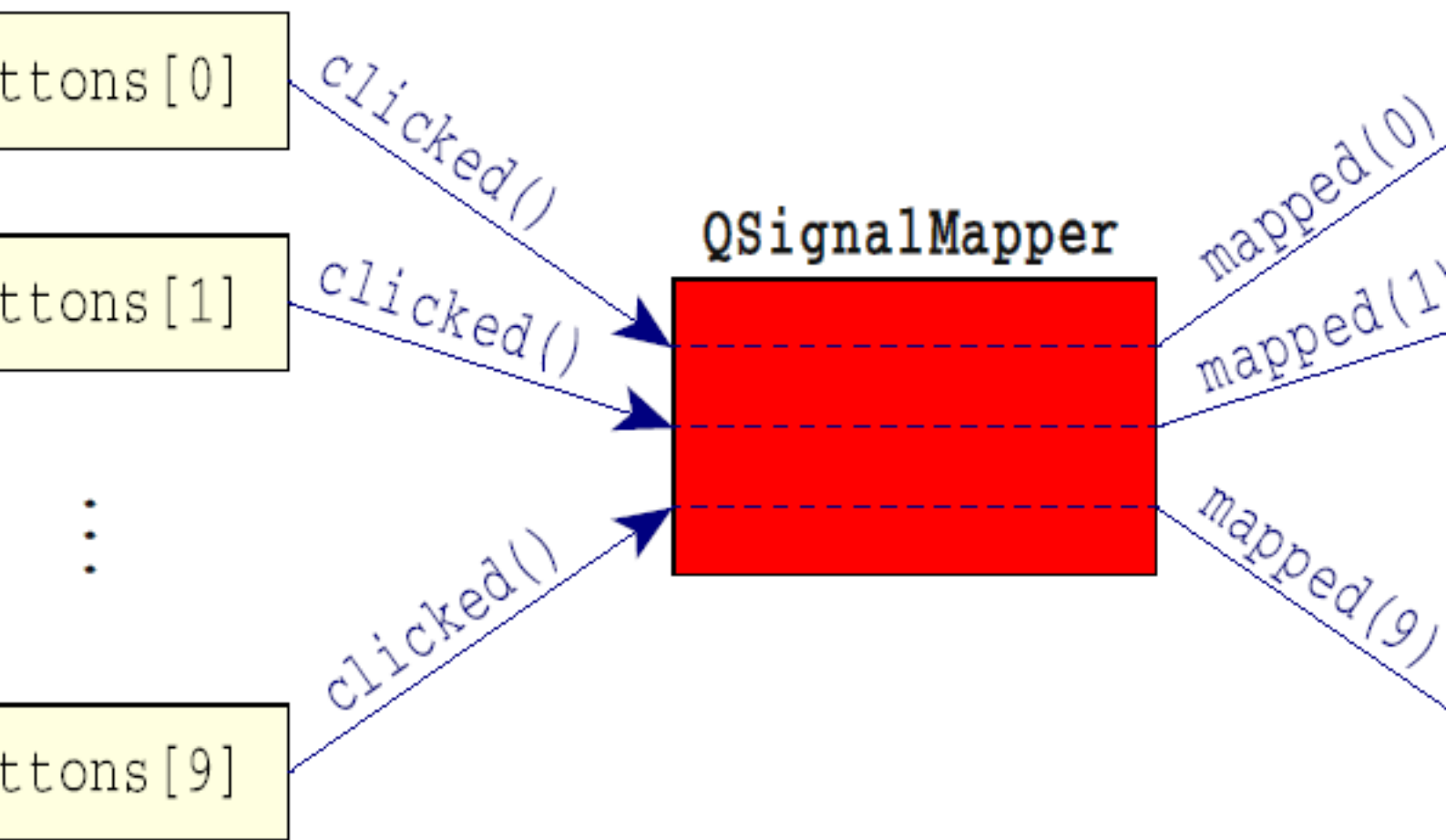
La quatrième et dernière approche ne requiert aucun slot privé, ni de classe dérivée de QPushButton. Au lieu de cela, l'entière logique des signaux est implémentée dans le constructeur de la classe Keypad :

```
Keypad::Keypad(QWidget *parent)
: QWidget(parent)
{
    QSignalMapper *signalMapper = new QSignalMapper(this);
    connect(signalMapper, SIGNAL(mapped(int)), this, SIGNAL(digitClicked(int)));

    for (int i = 0; i < 10; ++i) {
        QString text = QString::number(i);
        buttons[i] = new QPushButton(text, this);
        signalMapper->setMapping(buttons[i], i);
        connect(buttons[i], SIGNAL(clicked()), signalMapper, SLOT(map()));
    }

    createLayout();
}
```

Nous créons premièrement un objet QSignalMapper. QSignalMapper hérite de QObject et fournit un moyen d'établir une relation entre un ensemble de signaux sans paramètre et un signal ou un slot ayant un seul paramètre. L'appel de setMapping() dans la boucle for établit une liaison entre un bouton et une valeur entière; par exemple, buttons[3] est associé à la valeur 3.



Lorsque le signal clicked() d'un bouton est émis, le slot map() du QSignalMapper est invoqué (grâce au connect() dans la boucle for). Si une liaison existe pour l'objet qui a émis le signal, le signal mapped(int) est émis avec la valeur entière définie par setMapping(). Ce signal est à son tour connecté au signal digitClicked(int) du Keypad. La fonction setMapping() existe en deux versions : une qui utilise un int et l'autre une QString comme second argument. Cela rend possible l'association d'une chaîne quelconque avec un objet émetteur, à la place d'un entier. Lorsqu'une telle association existe, QSignalMapper émet le signal mapped(const QString &).



QSignalMapper ne supporte pas directement d'autres types de données. Ce qui signifie, par exemple, que si vous vouliez implémenter une palette permettant à l'utilisateur de choisir une couleur parmi un ensemble de couleurs et aviez besoin d'émettre un signal `colorSelected(const QColor &)`, le meilleur choix serait d'utiliser l'approche `sender()` ou une classe dérivée. Si vous avez déjà une classe dérivée, disons `QPushButton`, pour représenter une couleur, cela ne coûte presque rien de lui ajouter un slot `clicked(const QColor &)`.

I-E - Le layout

Bien que cet exemple ne concerne pas les layouts, le code ne compilera pas si nous n'implémentons pas la fonction `createLayout()` appelée par le constructeur. Voici le code manquant :

```
void Keypad::createLayout()
{
    QGridLayout *layout = new QGridLayout(this, 3, 4);
    layout->setMargin(6);
    layout->setSpacing(6);

    for (int i = 0; i < 9; ++i)
        layout->addWidget(buttons[i + 1], i / 3, i % 3);
    layout->addWidget(buttons[0], 3, 1);
}
```

Le code de la fonction `main()` manque également; nous laissons son implémentation en exercice au lecteur.