

Traduction dynamique

Qt by Nokia

par Johan Thelin traducteur : Guillaume Belz Qt Quarterly

Date de publication : 15/04/2010

Dernière mise à jour : 23/10/2010


Permettre la traduction facile des interfaces utilisateur est l'un des points forts de Qt. Le support de l'Unicode dès les fondations et une infrastructure pour l'internationalisation permettent de travailler facilement dans un environnement internationalisé.

Cet article est une traduction autorisée de **Dynamic Translation**, par Johan Thelin.

N'hésitez pas à commenter cet article !

I - L'article original.....	3
II - Introduction.....	3
III - Compliquer un monde parfait.....	3
IV - Traduction dynamique automatique.....	5
V - Extraction de messages pour les paresseux.....	6
VI - Conclusion.....	6

I - L'article original

Qt Quarterly est une revue trimestrielle électronique proposée par Nokia à destination des développeurs et utilisateurs de Qt. Vous pouvez trouver les  **versions originales**.

Nokia, Qt, Qt Quarterly et leurs logos sont des marques déposées de *Nokia Corporation* en Finlande et/ou dans les autres pays. Les autres marques déposées sont détenues par leurs propriétaires respectifs.

Cet article est la traduction de l'article **Dynamic Translation** de Johan Thelin paru dans la Qt Quarterly Issue 33.

Cet article est une traduction d'un des tutoriels écrits par **Nokia Corporation and/or its subsidiary(-ies)** incluse dans la documentation de Qt, en anglais. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia.

II - Introduction

Permettre la traduction facile des interfaces utilisateur est l'un des points forts de Qt. Le support de l'unicode dès les fondations et une infrastructure pour l'internationalisation permettent de travailler facilement dans un environnement multi-langue.




Les règles de travail sont simples : toutes les chaînes visibles par les utilisateurs finaux ont besoin de passer par la fonction **QObject::tr()**. Cette fonction **tr()** peut traiter les formes du singulier et du pluriel lorsque les textes l'utilisent. Elle permet aussi de fournir des descriptions non ambiguës aux traducteurs, pour éviter les erreurs.

Parfois les messages ne sont pas utilisés dans un contexte permettant d'utiliser **tr()**. Par exemple dans une liste de chaînes de caractères avec des messages qui sont sélectionnés en fonction d'un index. Dans ce cas, les macros **QT_TR_NOOP**, **QT_TRANSLATE_NOOP** et **QT_TRANSLATE_NOOP3** peuvent être utilisées pour marquer la chaîne que l'utilisateur pourra voir. Assurez-vous simplement de passer la chaîne par **tr()** avant de la montrer à l'utilisateur.

Tous les messages visibles pour l'utilisateur sont ensuite extraits à l'aide de **lupdate**, traduit en utilisant **Qt Linguist** et compilés pour être intégré dans une application en utilisant **lrelease**. A partir de là, utiliser les traductions se fait facilement en créant un objet **QTranslator** dans l'objet **QApplication** utilisé.


Lorsque vous installez un nouveau traducteur, un événement **QEvent::LanguageChange** se propage à travers le système. Cela permet de supporter les changements de langue à la volée, ce que tous **Designer** générant des interfaces utilisateur peut proposer grâce à l'utilisation de la fonction **retranslateUi()** implémentée dans les classes **Ui::**. Utiliser **Qt Creator** pour créer une classe de formulaire pour Qt Designer (Qt Designer Form Class) permet de générer automatiquement le code de la méthode **changeEvent()** et l'appel à **retranslateUi()**.


III - Compliquer un monde parfait

Les  **modèles** sont une des choses qui ne sont pas mis à jour dynamiquement. Il y a plusieurs façons d'aborder ce problème : soit en implémentant un modèle proche de **tr()** pour nos données, soit en implémentant une  **délégation** de **tr()** qui applique la traduction au moment de mettre à jour l'affichage. Ici, nous allons utiliser une troisième voie : une  **délégation** de **tr()**, **TranslatorProxyModel**, qui traduit les données du modèle au moment où on les transmet.

L'ajout de traductions aux modèles a un avantage par rapport aux délégations. Les modèles peuvent intercepter les événements **QEvent::LanguageChange** et appeler la fonction **reset()**, c'est à dire faire en sorte que la vue recharge l'ensemble des données à nouveau.

Lors de la création d'un modèle proche de **tr()** fournissant les données actualisées, utiliser **tr()** directement n'est généralement pas un problème. Cependant, **tr()** s'appuie sur un contexte. Le contexte est généralement la classe

contenant l'appel à **tr()**. Il est donc possible d'utiliser la même expression dans différents dialogues et de la traduire différemment selon le contexte. Cela signifie également qu'on doit donner un contexte à un modèle  **délégation**, qui ne contient pas toutes les chaînes en interne.


En conséquence, le constructeur du modèle  **délégation** prend un contexte comme argument. Ce contexte est ensuite utilisé dans les appels directs à la fonction **QCoreApplication::translate()** au lieu de **tr()**. La déclaration de la classe entière est indiquée ci-dessous.

```
class TranslatorProxyModel : public QSortFilterProxyModel
{
    Q_OBJECT
public:
    explicit TranslatorProxyModel(const char *translatorContext,
        QObject *parent = 0);

    QVariant data(const QModelIndex &index,
        int role=Qt::DisplayRole) const;
    QVariant headerData(int section,
        Qt::Orientation orientation,
        int role=Qt::DisplayRole) const;

protected:
    bool event(QEvent *e);

private:
    QByteArray m_translatorContext;
};
```

Comme vous pouvez le voir, la  **délégation** surcharge les fonctions **data()** et **headerData()**. Ces deux fonctions sont très similaires. Tout ce qu'elles font, c'est de traduire les données **DisplayRole** du modèle source. Cela signifie que le rôle **EditRole** est laissé non traduit. Tout comme **ToolTipRole**, **StatusTipRole** et **WhatsThisRole**. Je laisse comme exercice, pour le lecteur intéressé, d'ajouter des traductions pour les rôles supplémentaires, ainsi que de s'assurer que les données, autre que les chaînes qui sont passées par cette classe, le sont d'une manière propre. Mais comme point de départ, c'est à cela que la fonction **data()** doit ressembler.

```
QVariant TranslatorProxyModel::data(const QModelIndex &index,
    int role) const
{
    if(role == Qt::DisplayRole)
        return QCoreApplication::translate(
            m_translatorContext.constData(),
            QSortFilterProxyModel::data(index, role)
                .toString().toAscii().constData());
    else
        return QSortFilterProxyModel::data(index, role);
}
```

Une utilisation habituelle du traducteur  **par délégation** serait de déclarer toutes les données en utilisant **QT_TRANSLATE_NOOP** puis de placer le modèle Proxy entre le modèle et la vue réelle. Comme le modèle Proxy intercepte et réagit de lui-même à l'événement **QEvent::LanguageChange**, cette solution est du type *tire et oublie*.

```
QStandardItemModel *model = new QStandardItemModel(this);
model->setHorizontalHeaderLabels( QStringList()
    << QT_TRANSLATE_NOOP("CapitalModel", "Country")
    << QT_TRANSLATE_NOOP("CapitalModel", "Capital"));
model->appendRow(QList<QStandardItem*>()
    << new QStandardItem(QT_TRANSLATE_NOOP("CapitalModel", "Denmark"))
    << new QStandardItem(QT_TRANSLATE_NOOP("CapitalModel", "Copenhagen")));

// Add more data here

TranslatorProxyModel *proxyModel =
    new TranslatorProxyModel("CapitalModel", this);
proxyModel->setSourceModel(model);
```

```
QTableView *view = new QTableView();  
view->setModel(proxyModel);
```

Le reste du code source de la classe TranslatorProxyModel peut être téléchargé avec cet article (voir en fin d'article).

IV - Traduction dynamique automatique

Etre capable de changer de langue à la volée peut être une fonctionnalité très complexe à mettre en place dans certains cas. En utilisant Qt Designer, même le changement dynamique de langue est une tâche facile. Cependant implémenter l'interface utilisateur avec du code pur peut être assez fastidieux.

La raison est que vous devez garder une trace de tous les widgets contenant des messages traduisibles. Cela correspond en particulier à toutes les petites étiquettes, boîtes de groupe et autres parties passives de l'interface utilisateur que la gestion de la mémoire de Qt vous permet généralement d'oublier. Toutefois, depuis la version 4.2 de Qt, les objets **QObject** peuvent être paramétrés avec des propriétés dynamiques. Ceci, combiné avec un algorithme récursif, peut être utilisé pour fournir la traduction dynamique automatique.

La méthode se déroule comme suit. Chaque widget peut être paramétré avec une propriété dynamique appelé **Originaltext**. Cette propriété contient un texte qui est ensuite utilisé pour définir les propriétés **text**, **title** et **windowTitle** des widgets.

La traduction dynamique est assurée par la fonction **dynamicRetranslateUi()**, qui est destinée à être appelée lorsqu'un événement **QEvent::LanguageChange** est reçu. Dans ce cas, la fonction est implémentée dans une classe **DynamicTranslatorWidget**. Pour réutiliser cette solution, il faut tout simplement hériter de cette classe lors de la création d'une nouvelle fenêtre. La classe contient également une fonction **changeEvent()** pour faire en sorte que la fonction soit appelée lorsque c'est nécessaire.

L'implémentation de la fonction peut être divisée en trois parties : la récurrence, la traduction et la modification du texte. La récurrence visite simplement tous les widgets dans la hiérarchie de **QObject** à l'aide des fonctions **children()** et **isWidgetType()**. Cela peut être étudié dans le code source téléchargeable. La traduction, cependant, n'est pas aussi simple qu'on pourrait le croire.

Comme la classe **DynamicTranslatorWidget** est la base de la classe appelant la fonction **tr()**, cette classe n'est pas le contexte approprié pour l'appel à **tr()**. Au lieu de cela, nous devons utiliser les informations du méta-objet sur le nom de classe en cours afin de déterminer un contexte plus plausible. Cela donne le code suivant :

```
QString text = QApplication::translate(  
    this->metaObject()->className(),  
    w->property("originalText")  
        .toString().toAscii().constData());
```

Une autre approche aurait été de fournir à la classe un contexte, comme pour **TranslatorProxyModel**. Si le contexte n'est pas défini, l'approche méta-objet peut être utilisée comme solution de repli. Encore une fois, le lecteur intéressé est invité à donner son avis.

Lorsque nous avons la traduction d'une chaîne de caractères, il faut l'attribuer à la propriété **text** visible par les utilisateurs. Cette propriété n'est pas la même pour tous les widgets. La plupart ont une propriété **text** (**étiquettes**, **boutons**, etc.), certains ont une propriété **title** (**zone de groupe**) tandis que d'autres ont une propriété **format**. Tous les widgets peuvent également être affichés comme une fenêtre et donc avoir une propriété **windowTitle**.

Dans la solution présentée ici, seules les propriétés **text** et **title** sont utilisées. Ajouter du support pour d'autres propriétés est triviale. Par ailleurs, la fenêtre est supposée n'avoir aucune de ces propriétés. Quelque chose qui est vrai si la base de toutes les fenêtres est la classe **DynamicTranslatorWidget**.

```
if (w->metaObject()->indexOfProperty("text") != -1)  
    w->setProperty("text", text);  
else if (w->metaObject()->indexOfProperty("title") != -1)
```

```
w->setProperty("title", text);  
else if (w->isWindow())  
    w->setWindowTitle(text);
```

Notez que l'existence des propriétés est vérifiée par l'intermédiaire des méta-objets.

La solution présentée dans le code source en fin d'article ajoute une autre caractéristique de la traduction dynamique. Il est possible d'ajouter des propriétés appelées `textValue`, `textValue1`, `textValue2` et ainsi de suite. Ils sont utilisés pour remplacer `%n`, `%1`, `%2`, etc. dans la chaîne en cours de traduction. Ceci est expliqué dans les commentaires du code donc je ne vais pas entrer plus dans les détails ici. Toutefois, il convient de noter que l'appel à `%n` est possible pour les formes du singulier et du pluriel des chaînes de traduction.

V - Extraction de messages pour les paresseux

Etant moi-même un développeur, je me rends compte que la plupart des développeurs ne se réjouissent pas d'ajouter des lignes supplémentaires permettant de définir la propriété `Originaltext` de tous les widgets de l'interface. Ce n'est pas un travail amusant et ça peut être difficile à automatiser correctement. Cependant, il y a un truc pour cela.

La classe `DynamicTranslatorWidget` contient une fonction protégée `dynamicExtractStrings()`. Elle utilise la même approche récursive que `dynamicRetranslateUi()` mais elle extrait le texte original de tous les widgets qu'elle peut trouver.

Le code d'extraction n'est pas très intéressant en lui-même. Il prend simplement les propriétés `text`, `title` `windowTitle` et les copie dans `Originaltext`. Elle prend également soin de ne pas remplacer une propriété `Originaltext` déjà existante. Utiliser cette approche signifie qu'il faut prendre certaines précautions.

- Assurez-vous qu'aucun traducteur est en place avant son appel si vous utilisez `tr()`, ou remplacez tous vos appels à `tr()` par `QT_TR_NOOP` ;
- les textes utilisant `%n`, `%1`, `%2`, etc. ne fonctionneront pas et `%n` ne peut être utilisé avec `QT_TR_NOOP`. Dans ces cas, vous devez faire un appel fictif à `tr()` (pour que `lupdate` enregistre la chaîne), puis assigner manuellement la propriété `Originaltext` ;
- en assignant une chaîne vide à la propriété `Originaltext`, vous pouvez vous assurer que le texte ne sera pas extrait et que le widget ne sera pas traduit. Cela permet de gérer les changements de langue à partir de votre propre code ;
- n'oubliez pas d'appeler `dynamicRetranslateUi()` lorsque tous les widgets ont été mis en place et tous les textes extraits.

La nécessité d'ajouter manuellement `Originaltext` quand on utilise `%n` n'est pas vraiment un obstacle, tout comme il est nécessaire de continuer à assigner la propriété `textValue`.

VI - Conclusion

Utiliser les classes `DynamicTranslatorWidget` et `TranslatorProxyModel` dans votre application est facile. Ces classes peuvent avoir besoin de quelques ajustements, car elles sont plus des démonstrations techniques que des implémentations matures. Le résultat final est simplement une application où la traduction dynamique fonctionne correctement.

Le code source de l'exemple présenté dans cet article est disponible au téléchargement à l'adresse suivante : [Source qq33-dynamic-translation.zip](http://source.qq33-dynamic-translation.zip).

Johan Thelin est l'auteur du livre *Foundations of Qt Development* de chez Apress et a un faible pour les systèmes embarqués. Il est également impliqué sur le site QtCentre ainsi que dans le développement de logiciels et la rédaction technique.

Merci à [dourouc05](#) et à [Lyche](#) pour leur relecture et leurs conseils.

Au nom de toute l'équipe Qt, j'aimerais adresser le plus grand remerciement à Nokia pour nous avoir autorisé la traduction de cet article !