eBook Gratuit

APPRENEZ qml

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#qml

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: qml

It is an unofficial and free qml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official qml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec qml

Remarques

QML est un acronyme qui signifie **Q** t **M** eta-object **L** anguage. C'est un langage de programmation déclaratif qui fait partie du framework Qt. L'objectif principal de QML est la création rapide et facile d'interfaces utilisateur pour les systèmes de bureau, mobiles et intégrés. QML permet une intégration transparente de JavaScript , soit directement dans le code QML, soit en incluant des fichiers JavaScript.

Versions

Version Qt	Version QtQuick	Date de sortie
4.7	1.0	2010-09-21
4.8	1.1	2011-12-15
5.0	2.0	2012-12-19
5.1	2.1	2013-06-03
5.2	2.2	2013-12-12
5.3	2.3	2014-05-20
5.4	2.4	2014-12-10
5.5	2,5	2015-07-01
5.6	2.6	2016-03-15
5.7	2.7	2016-06-16
5.8	2.7	2017-01-23

Examples

Installation

QML est livré avec une version plus récente de l'infrastructure d'application multiplate-forme Qt . Vous pouvez trouver la version la plus récente de Qt dans la section Téléchargements .

Pour créer un nouveau projet QML dans l' EDI de Qt Creator , sélectionnez "Fichier -> Nouveau ..." et sous "Applications", sélectionnez "Application rapide Qt". Après avoir cliqué sur

"sélectionner", vous pouvez maintenant nommer et définir le chemin pour ce projet. Après avoir cliqué sur "suivant", vous pouvez sélectionner les composants que vous souhaitez utiliser. En cas de doute, laissez simplement la valeur par défaut et cliquez sur "Suivant". Les deux étapes suivantes vous permettront de configurer un kit et un contrôle de source si vous le souhaitez, sinon conservez les paramètres par défaut.

Vous avez maintenant créé une application QML simple et prête à l'emploi.

Bonjour le monde

Une application simple montrant le texte "Hello World" au centre de la fenêtre.

```
import QtQuick 2.3
import QtQuick.Window 2.0

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World") //The method qsTr() is used for translations from one language
to other.

    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
}
```

Créer un simple bouton

Vous pouvez facilement transformer chaque composant dans un bouton cliquable à l'aide du composant MouseArea. Le code ci-dessous affiche une fenêtre 360x360 avec un bouton et un texte au centre; En appuyant sur le bouton, le texte change:

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360

    Rectangle {
        id: button

        width: 100
        height: 30
        color: "red"
        radius: 5     // Let's round the rectangle's corner a bit, so it resembles more a
button
        anchors.centerIn: parent

        Text {
            id: buttonText
            text: qsTr("Button")
            color: "white"
            anchors.centerIn: parent
```

```
        }

        MouseArea {
            // We make the MouseArea as big as its parent, i.e. the rectangle. So pressing
anywhere on the button will trigger the event
            anchors.fill: parent

            // Exploit the built-in "clicked" signal of the MouseArea component to do
something when the MouseArea is clicked.
            // Note that the code associated to the signal is plain JavaScript. We can
reference any QML objects by using their IDs
            onClicked: {
                buttonText.text = qsTr("Clicked");
                buttonText.color = "black";
            }
        }
    }
}
```

Afficher une image

Cet exemple montre l'utilisation la plus simple du composant Image pour afficher une image.

La propriété `source` Image est un type d'URL qui peut être un fichier avec un chemin absolu ou relatif, une URL Internet (`http://`) ou une ressource Qt (`qrc:/`)

```
import QtQuick 2.3

Rectangle {
    width: 640
    height: 480

    Image {
         source: "image.png"
    }
}
```

Événement souris

Cet exemple montre comment l'événement de souris est utilisé dans QML.

```
import QtQuick 2.7
import QtQuick.Window 2.2

Window {
    visible: true
    Rectangle {
        anchors.fill: parent
        width: 120; height: 240
        color: "#4B7A4A"

        MouseArea {
            anchors.fill: parent // set mouse area (i.e. covering the entire rectangle.)
            acceptedButtons:  Qt.AllButtons
            onClicked: {
                // print to console mouse location
```

```
                console.log("Mouse Clicked.")
                console.log("Mouse Location: <",mouseX,",",mouseY,">")

                //change Rectangle color
                if ( mouse.button === Qt.RightButton )
                    parent.color = 'blue'
                if ( mouse.button === Qt.LeftButton )
                    parent.color = 'red'
                if ( mouse.button === Qt.MiddleButton )
                    parent.color = 'yellow'
            }
            onReleased: {
                // print to console
                console.log("Mouse Released.")
            }
            onDoubleClicked: {
                // print to console
                console.log("Mouse Double Clicked.")
            }

        }
    }


}
```

Lire Démarrer avec qml en ligne: https://riptutorial.com/fr/qml/topic/653/demarrer-avec-qml

Chapitre 2: Animation

Examples

Animation de numéro simple

Une des animations les plus élémentaires que vous pourriez trouver est l' `NumberAnimation` . Cette animation fonctionne en changeant la valeur numérique d'une propriété d'un élément d'un état initial à un état final. Prenons l'exemple complet suivant:

```
import QtQuick 2.7
import QtQuick.Controls 2.0

ApplicationWindow {
    visible: true
    width: 400
    height: 640

    Rectangle{
        id: rect
        anchors.centerIn: parent
        height: 100
        width: 100
        color: "blue"
        MouseArea{
            anchors.fill: parent
            onClicked: na.running = true
        }

        NumberAnimation {
            id: na    //ID of the QML Animation type
            target: rect    //The target item on which the animation should run
            property: "height"    //The property of the target item which should be changed by
the animator to show effect
            duration: 200    //The duration for which the animation should run
            from: rect.height    //The initial numeric value of the property declared in
'property'
            to: 200    //The final numeric value of the property declared in 'property'
        }
    }
}
```

Animation basée sur le comportement

Une animation basée sur le comportement vous permet de spécifier que, lorsqu'une propriété change, la modification doit être animée au fil du temps.

```
ProgressBar {
    id: progressBar
    from: 0
    to: 100
    Behavior on value {
        NumberAnimation {
```

```
            duration: 250
        }
    }
}
```

Dans cet exemple, si quelque chose modifie la valeur de la barre de progression, la modification sera animée sur 250 ms

Lire Animation en ligne: https://riptutorial.com/fr/qml/topic/1281/animation

Chapitre 3: Création d'éléments personnalisés en C ++

Examples

Création d'éléments personnalisés en C ++

QML est venu avec un riche ensemble d'éléments visuels. En utilisant uniquement QML, nous pouvons créer des applications complexes avec ces éléments. En outre, il est très facile de créer votre propre élément en fonction d'un ensemble d'éléments standard tels que Rectangle, Button, Image, etc. De plus, nous pouvons utiliser des éléments tels que Canvas pour créer des éléments avec une peinture personnalisée. Il semblerait que nous puissions créer diverses applications en QML uniquement, sans toucher aux fonctionnalités de C ++. Et c'est en fait vrai, mais parfois nous aimerions rendre notre application plus rapide ou nous voulons l'étendre avec la puissance de Qt ou pour ajouter des opportunités qui ne sont pas disponibles dans QML. Et il y a certainement une telle possibilité dans *QML* . Fondamentalement, *QtQuick* utilise *Scene Graph* pour peindre son contenu en un moteur de rendu hautes performances basé sur *OpenGL* . Pour implémenter notre propre élément visuel, nous pouvons utiliser 2 manières:

1. La méthode traditionnelle pour Qt utilisant QPainter (QQuickPaintedItem).
2. La méthode QML commune utilisant les fonctionnalités QQuickItem et OpenGL.

Il est possible que la première méthode semble plus facile, mais cela vaut la peine de la considérer comme plus lente que la première, car QtQuick peint le contenu de l'élément sur une surface, puis l'insère dans un graphe de scène. L'utilisation directe de l'API de graphes de scènes est donc toujours beaucoup plus rapide.

Pour explorer les deux méthodes de plus près, créons notre propre élément qui n'existe pas dans QML, par exemple un triangle.

Déclaration de classe

```
class QQuickCustomItem : public QQuickItem
{
    Q_OBJECT
    Q_PROPERTY(QColor color READ color WRITE setColor NOTIFY colorChanged)
public:
    QQuickCustomItem(QQuickItem *parent = Q_NULLPTR);

protected:
    QSGNode *updatePaintNode(QSGNode *oldNode, UpdatePaintNodeData *updatePaintNodeData);

    QColor color() const;
    void setColor(const QColor &color);

private:
    QColor m_color;
    bool m_needUpdate;
```

```
signals:
    void colorChanged();
};
```

Nous ajoutons la macro Q_OBJECT pour travailler avec les signaux. Nous ajoutons également une propriété personnalisée pour spécifier la couleur de notre rectangle. Pour que cela fonctionne, il suffit de réimplémenter la fonction virtuelle QQuiclItem :: updatePaintNode () .

Implémentation de classe.

Nous définissons tout d'abord un constructeur.

```
QQuickCustomItem::QQuickCustomItem(QQuickItem *parent) :
    QQuickItem(parent),
    m_color(Qt::red),
    m_needUpdate(true)
{
    setFlag(QQuickItem::ItemHasContents);
}
```

Veuillez noter que l'appel de la fonction setFlag () est obligatoire, sinon votre objet ne sera pas ajouté au graphique de la scène. Ensuite, nous définissons une fonction pour la douleur.

```
QSGNode *QQuickCustomItem::updatePaintNode(QSGNode *oldNode, QQuickItem::UpdatePaintNodeData
*updatePaintNodeData)
{
    Q_UNUSED(updatePaintNodeData)
    QSGGeometryNode *root = static_cast<QSGGeometryNode *>(oldNode);

    if(!root) {
        root = new QSGGeometryNode;
        QSGGeometry *geometry = new QSGGeometry(QSGGeometry::defaultAttributes_Point2D(), 3);
        geometry->setDrawingMode(GL_TRIANGLE_FAN);
        geometry->vertexDataAsPoint2D()[0].set(width() / 2, 0);
        geometry->vertexDataAsPoint2D()[1].set(width(), height());
        geometry->vertexDataAsPoint2D()[2].set(0, height());

        root->setGeometry(geometry);
        root->setFlag(QSGNode::OwnsGeometry);
        root->setFlag(QSGNode::OwnsMaterial);
    }

    if(m_needUpdate) {
        QSGFlatColorMaterial *material = new QSGFlatColorMaterial;
        material->setColor(m_color);
        root->setMaterial(material);
        m_needUpdate = false;
    }

    return root;
}
```

Au premier appel à la fonction, notre noeud n'est pas encore créé, donc `oldNode` sera NULL. Nous créons donc le nœud et lui assignons géométrie et matériel. Ici, nous utilisons *GL_TRIANGLE_FAN* pour notre géométrie pour peindre un rectangle plein. Ce point est le même

que dans OpenGL. Par exemple, pour dessiner un triangle, nous pouvons changer le code pour:

```
geometry->setDrawingMode(GL_LINE_LOOP);
geometry->setLineWidth(5);
```

Vous pouvez vous référer au manuel *OpenGL* pour vérifier d'autres formes. Donc, il ne reste plus qu'à définir setter / getter pour notre propriété:

```
QColor QQuickCustomItem::color() const
{
    return m_color;
}

void QQuickCustomItem::setColor(const QColor &color)
{
    if(m_color != color) {
        m_color = color;
        m_needUpdate = true;
        update();
        colorChanged();
    }
}
```

Maintenant, il n'y a qu'un petit détail pour que cela fonctionne. Nous devons notifier *QtQuick* du nouvel élément. Par exemple, vous pouvez ajouter ce code à votre main.cpp:

```
qmlRegisterType<QQuickCustomItem>("stackoverflow.qml", 1, 0, "Triangle");
```

Et voici notre fichier de test QML:

```
import QtQuick 2.7
import QtQuick.Window 2.0
import stackoverflow.qml 1.0

Window {
    width: 800
    height: 800
    visible: true

    Rectangle {
        width: 200
        height: 200
        anchors.centerIn: parent
        color: "lightgrey"

        Triangle {
            id: rect
            width: 200
            height: 200
            transformOrigin: Item.Top
            color: "green"
            onColorChanged: console.log("color was changed");
            PropertyAnimation on rotation {
                from: 0
                to: 360
                duration: 5000
```

```
                loops: Animation.Infinite
            }
        }
    }
    Timer {
        interval: 1000
        repeat: true
        running: true
        onTriggered: rect.color = Qt.rgba(Math.random(),Math.random(),Math.random(),1);
    }
}
```

Comme vous voyez notre article se comporte comme tous les autres éléments QML. Maintenant, créons le même élément en utilisant QPainter :

Tout ce dont nous avons besoin est de remplacer

```
QSGNode *updatePaintNode(QSGNode *oldNode, UpdatePaintNodeData *updatePaintNodeData);
```

avec

```
void paint(QPainter *painter);
```

et, bien sûr, hériter de notre classe de `QQuickPaintedItem` au lieu de `QQuickItem` . Voici notre fonction de peinture:

```
void QQuickCustomItem::paint(QPainter *painter)
{
    QPainterPath path;
    path.moveTo(width() / 2, 0);
    path.lineTo(width(), height());
    path.lineTo(0, height());
    path.lineTo(width() / 2, 0);
    painter->fillPath(path, m_color);
}
```

Tout le reste reste inchangé.

Lire Création d'éléments personnalisés en C ++ en ligne:
https://riptutorial.com/fr/qml/topic/6509/creation-d-elements-personnalises-en-c-plusplus

Chapitre 4: Intégration avec C ++

Examples

Créer une vue QtQuick à partir de C ++

Il est possible de créer une vue QtQuick directement à partir de C ++ et d'exposer aux propriétés définies par QML C ++. Dans le code ci-dessous, le programme C ++ crée une vue QtQuick et expose à QML la hauteur et la largeur de la vue en tant que propriétés.

main.cpp

```
#include <QApplication>
#include <QQmlContext>
#include <QQuickView>

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

    // Creating the view and manually setting the QML file it should display
    QQuickView view;
    view.setSource(QStringLiteral("main.qml"));

    // Retrieving the QML context. This context allows us to expose data to the QML components
    QQmlContext* rootContext = view.rootContext();

    // Creating 2 new properties: the width and height of the view
    rootContext->setContextProperty("WINDOW_WIDTH", 640);
    rootContext->setContextProperty("WINDOW_HEIGHT", 360);

    // Let's display the view
    view.show();

    return app.exec();
}
```

main.qml

```
import QtQuick 2.0

Rectangle {
    // We can now access the properties we defined from C++ from the whole QML file
    width: WINDOW_WIDTH
    height: WINDOW_HEIGHT

    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
}
```

Créer une fenêtre QtQuick à partir de C ++

A partir de Qt 5.1 et versions ultérieures, vous pouvez utiliser QQmlApplicationEngine au lieu de QQuickView pour charger et afficher un script QML.

Avec QQmlApplicationEngine, vous devez utiliser un type de fenêtre QML comme élément racine.

Vous pouvez obtenir le contexte racine du moteur où vous pouvez ensuite ajouter des propriétés globales au contexte auquel le moteur peut accéder lors du traitement des scripts QML.

main.cpp

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

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

    QQmlApplicationEngine engine;

    QQmlContext* rootContext = engine.rootContext();
    rootContext->setContextProperty("WINDOW_WIDTH", 640);
    rootContext->setContextProperty("WINDOW_HEIGHT", 360);

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

    return app.exec();
}
```

main.qml

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window { // Must be this type to be loaded by QQmlApplicationEngine.
    visible: true
    width: WINDOW_WIDTH   //Accessing global context declared in C++
    height: WINDOW_HEIGHT //Accessing global context declared in C++
    title: qsTr("Hello World")
    Component.onCompleted: {
        // We can access global context from within JavaScript too.
        console.debug( "Width: " + WINDOW_WIDTH )
        console.debug( "Height: " + WINDOW_HEIGHT )
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
    }

    Text {
        text: qsTr("Hello World")
        anchors.centerIn: parent
    }
}
```

Créer un modèle simple pour TreeView

Depuis Qt 5.5, nous avons un nouveau TreeView merveilleux, un contrôle que nous attendions tous. Un TreeView implémente une représentation arborescente des éléments d'un modèle. En général, cela ressemble à d'autres vues QML - ListView ou TableView . Mais la structure de données de TreeView est plus complexe.

Une donnée dans ListView ou TableView est représentée par un tableau de nœuds à une dimension. Dans TreeView, chaque nœud peut contenir son propre tableau de nœuds. Par conséquent, contrairement aux autres vues de TreeView pour obtenir le nœud spécifié, nous devons connaître le nœud parent, pas seulement la ligne ou la colonne de l'élément.

Une autre différence majeure est que TreeView ne supporte pas ListModel . Pour fournir une donnée, nous devons sous- classer QAbstractItemModel . Dans Qt, il existe des classes de modèle prêtes à l'emploi comme QFileSystemModel qui donne accès au système de fichiers local ou QSqlTableModel qui donne accès à une base de données.

Dans l'exemple suivant, nous allons créer un tel modèle dérivé de QAbstractItemModel . Mais pour rendre l'exemple plus réaliste, je suggère de rendre le modèle comme ListModel mais spécifié pour les arbres afin que nous puissions ajouter des nœuds à partir de QML. Il est nécessaire de préciser que le modèle lui-même ne contient aucune donnée, mais seulement d'y accéder. La fourniture et l'organisation de données relèvent donc entièrement de notre responsabilité.

Comme les données de modèle sont organisées dans un arbre, la structure de noeud la plus simple est vue comme suit, en pseudo-code:

```
Node {
    var data;
    Node parent;
    list<Node> children;
}
```

En C++ la déclaration de noeud doit être la suivante:

```
class MyTreeNode : public QObject
{
    Q_OBJECT
public:
    Q_PROPERTY(QQmlListProperty<MyTreeNode> nodes READ nodes)
    Q_CLASSINFO("DefaultProperty", "nodes")
    MyTreeNode(QObject *parent = Q_NULLPTR);

    void setParentNode(MyTreeNode *parent);
    Q_INVOKABLE MyTreeNode *parentNode() const;
    bool insertNode(MyTreeNode *node, int pos = (-1));
    QQmlListProperty<MyTreeNode> nodes();

    MyTreeNode *childNode(int index) const;
    void clear();

    Q_INVOKABLE int pos() const;
```

```
    Q_INVOKABLE int count() const;

private:
    QList<MyTreeNode *> m_nodes;
    MyTreeNode *m_parentNode;
};
```

Nous dérivons notre classe de QObject pour pouvoir créer un nœud dans `QML` . Tous les nœuds enfants seront ajoutés à la propriété `nodes` afin que la partie suivante du code soit identique:

```
TreeNode {
    nodes:[
        TreeNode {}
        TreeNode {}
    ]
}

TreeNode {
    TreeNode {}
    TreeNode {}
}
```

Voir cet article pour en savoir plus sur la propriété par défaut.

Implémentation de classe de noeud:

```
MyTreeNode::MyTreeNode(QObject *parent) :
    QObject(parent),
    m_parentNode(nullptr) {}

void MyTreeNode::setParentNode(MyTreeNode *parent)
{
    m_parentNode = parent;
}

MyTreeNode *MyTreeNode::parentNode() const
{
    return m_parentNode;
}

QQmlListProperty<MyTreeNode> MyTreeNode::nodes()
{
    QQmlListProperty<MyTreeNode> list(this,
                                      0,
                                      &append_element,
                                      &count_element,
                                      &at_element,
                                      &clear_element);
    return list;
}

MyTreeNode *MyTreeNode::childNode(int index) const
{
    if(index < 0 || index >= m_nodes.length())
        return nullptr;
    return m_nodes.at(index);
}
```

```
void MyTreeNode::clear()
{
    qDeleteAll(m_nodes);
    m_nodes.clear();
}

bool MyTreeNode::insertNode(MyTreeNode *node, int pos)
{
    if(pos > m_nodes.count())
        return false;
    if(pos < 0)
        pos = m_nodes.count();
    m_nodes.insert(pos, node);
    return true;
}

int MyTreeNode::pos() const
{
    MyTreeNode *parent = parentNode();
    if(parent)
        return parent->m_nodes.indexOf(const_cast<MyTreeNode *>(this));
    return 0;
}

int MyTreeNode::count() const
{
    return m_nodes.size();
}

MyTreeNode *MyTreeModel::getNodeByIndex(const QModelIndex &index)
{
    if(!index.isValid())
        return nullptr;
    return static_cast<MyTreeNode *>(index.internalPointer());
}

QModelIndex MyTreeModel::getIndexByNode(MyTreeNode *node)
{
    QVector<int> positions;
    QModelIndex result;
    if(node) {
        do
        {
            int pos = node->pos();
            positions.append(pos);
            node = node->parentNode();
        } while(node != nullptr);


        for (int i = positions.size() - 2; i >= 0 ; i--)
        {
            result = index(positions[i], 0, result);
        }
    }
    return result;
}

bool MyTreeModel::insertNode(MyTreeNode *childNode, const QModelIndex &parent, int pos)
{
    MyTreeNode *parentElement = getNode(parent);
    if(pos >= parentElement->count())
```

```
        return false;
    if(pos < 0)
        pos = parentElement->count();

    childNode->setParentNode(parentElement);
    beginInsertRows(parent, pos, pos);
    bool retValue = parentElement->insertNode(childNode, pos);
    endInsertRows();
    return retValue;
}

MyTreeNode *MyTreeModel::getNode(const QModelIndex &index) const
{
    if(index.isValid())
        return static_cast<MyTreeNode *>(index.internalPointer());
    return m_rootNode;
}
```

Pour exposer une propriété de type liste à QML via QQmlListProperty, nous avons besoin des 4 prochaines fonctions:

```
void append_element(QQmlListProperty<MyTreeNode> *property, MyTreeNode *value)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    value->setParentNode(parent);
    parent->insertNode(value, -1);
}

int count_element(QQmlListProperty<MyTreeNode> *property)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    return parent->count();
}

MyTreeNode *at_element(QQmlListProperty<MyTreeNode> *property, int index)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    if(index < 0 || index >= parent->count())
        return nullptr;
    return parent->childNode(index);
}

void clear_element(QQmlListProperty<MyTreeNode> *property)
{
    MyTreeNode *parent = (qobject_cast<MyTreeNode *>(property->object));
    parent->clear();
}
```

Maintenant, déclarons la classe de modèle:

```
class MyTreeModel : public QAbstractItemModel
{
    Q_OBJECT
public:
    Q_PROPERTY(QQmlListProperty<MyTreeNode> nodes READ nodes)
    Q_PROPERTY(QVariantList roles READ roles WRITE setRoles NOTIFY rolesChanged)
    Q_CLASSINFO("DefaultProperty", "nodes")
```

```
    MyTreeModel(QObject *parent = Q_NULLPTR);
    ~MyTreeModel();

    QHash<int, QByteArray> roleNames() const Q_DECL_OVERRIDE;
    QVariant data(const QModelIndex &index, int role) const Q_DECL_OVERRIDE;
    Qt::ItemFlags flags(const QModelIndex &index) const Q_DECL_OVERRIDE;
    QModelIndex index(int row, int column, const QModelIndex &parent = QModelIndex()) const
Q_DECL_OVERRIDE;
    QModelIndex parent(const QModelIndex &index) const Q_DECL_OVERRIDE;
    int rowCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;
    int columnCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;
    QQmlListProperty<MyTreeNode> nodes();

    QVariantList roles() const;
    void setRoles(const QVariantList &roles);

    Q_INVOKABLE MyTreeNode * getNodeByIndex(const QModelIndex &index);
    Q_INVOKABLE QModelIndex getIndexByNode(MyTreeNode *node);
    Q_INVOKABLE bool insertNode(MyTreeNode *childNode, const QModelIndex &parent =
QModelIndex(), int pos = (-1));

protected:
    MyTreeNode *getNode(const QModelIndex &index) const;

private:
    MyTreeNode *m_rootNode;
    QHash<int, QByteArray> m_roles;

signals:
    void rolesChanged();
};
```

Puisque nous avons dérivé la classe de modèle de l'abstraction QAbstractItemModel, nous devons redéfinir la fonction suivante: data () , flags () , index () , parent () , columnCount () et rowCount () . Pour que notre modèle puisse fonctionner avec `QML` nous définissons roleNames () . En outre, ainsi que dans la classe de nœud, nous définissons la propriété default pour pouvoir ajouter des nœuds au modèle dans `QML` . `roles` propriété `roles` contiendra une liste de noms de rôles.

La mise en oeuvre:

```
MyTreeModel::MyTreeModel(QObject *parent) :
    QAbstractItemModel(parent)
{
    m_rootNode = new MyTreeNode(nullptr);
}
MyTreeModel::~MyTreeModel()
{
    delete m_rootNode;
}

QHash<int, QByteArray> MyTreeModel::roleNames() const
{
    return m_roles;
}

QVariant MyTreeModel::data(const QModelIndex &index, int role) const
{
```

```
    if (!index.isValid())
        return QVariant();

    MyTreeNode *item = static_cast<MyTreeNode*>(index.internalPointer());
    QByteArray roleName = m_roles[role];
    QVariant name = item->property(roleName.data());
    return name;
}

Qt::ItemFlags MyTreeModel::flags(const QModelIndex &index) const
{
    if (!index.isValid())
        return 0;

    return QAbstractItemModel::flags(index);
}

QModelIndex MyTreeModel::index(int row, int column, const QModelIndex &parent) const
{
    if (!hasIndex(row, column, parent))
        return QModelIndex();

    MyTreeNode *parentItem = getNode(parent);
    MyTreeNode *childItem = parentItem->childNode(row);
    if (childItem)
        return createIndex(row, column, childItem);
    else
        return QModelIndex();
}

QModelIndex MyTreeModel::parent(const QModelIndex &index) const
{
    if (!index.isValid())
        return QModelIndex();

    MyTreeNode *childItem = static_cast<MyTreeNode*>(index.internalPointer());
    MyTreeNode *parentItem = static_cast<MyTreeNode *>(childItem->parentNode());

    if (parentItem == m_rootNode)
        return QModelIndex();

    return createIndex(parentItem->pos(), 0, parentItem);
}

int MyTreeModel::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    MyTreeNode *parentItem = getNode(parent);
    return parentItem->count();
}

int MyTreeModel::columnCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return 1;
}

QQmlListProperty<MyTreeNode> MyTreeModel::nodes()
{
    return m_rootNode->nodes();
```

```
}

QVariantList MyTreeModel::roles() const
{
    QVariantList list;
    QHashIterator<int, QByteArray> i(m_roles);
    while (i.hasNext()) {
        i.next();
        list.append(i.value());
    }

    return list;
}

void MyTreeModel::setRoles(const QVariantList &roles)
{
    static int nextRole = Qt::UserRole + 1;
    foreach(auto role, roles) {
        m_roles.insert(nextRole, role.toByteArray());
        nextRole ++;
    }
}

MyTreeNode *MyTreeModel::getNodeByIndex(const QModelIndex &index)
{
    if(!index.isValid())
        return nullptr;
    return static_cast<MyTreeNode *>(index.internalPointer());
}

QModelIndex MyTreeModel::getIndexByNode(MyTreeNode *node)
{
    QVector<int> positions;
    QModelIndex result;
    if(node) {
        do
        {
            int pos = node->pos();
            positions.append(pos);
            node = node->parentNode();
        } while(node != nullptr);


        for (int i = positions.size() - 2; i >= 0 ; i--)
        {
            result = index(positions[i], 0, result);
        }
    }
    return result;
}


bool MyTreeModel::insertNode(MyTreeNode *childNode, const QModelIndex &parent, int pos)
{
    MyTreeNode *parentElement = getNode(parent);
    if(pos >= parentElement->count())
        return false;
    if(pos < 0)
        pos = parentElement->count();

    childNode->setParentNode(parentElement);
```

```
    beginInsertRows(parent, pos, pos);
    bool retValue = parentElement->insertNode(childNode, pos);
    endInsertRows();
    return retValue;
}

MyTreeNode *MyTreeModel::getNode(const QModelIndex &index) const
{
    if(index.isValid())
        return static_cast<MyTreeNode *>(index.internalPointer());
    return m_rootNode;
}
```

En général, ce code n'est pas très différent de l'implémentation standard, par exemple Exemple d' arborescence simple

Au lieu de définir des rôles en C++ nous fournissons un moyen de le faire à partir de QML . Les événements et méthodes TreeView fonctionnent essentiellement avec QModelIndex . Personnellement, je ne vois pas beaucoup de sens à transmettre cela à qml, car la seule chose que vous pouvez faire est de le renvoyer au modèle.

Quoi qu'il en soit, notre classe offre un moyen de convertir l'index en nœud et vice versa. Pour pouvoir utiliser nos classes dans QML, nous devons l'enregistrer:

```
qmlRegisterType<MyTreeModel>("qt.test", 1, 0, "TreeModel");
qmlRegisterType<MyTreeNode>("qt.test", 1, 0, "TreeElement");
```

Et enfin, en exemple, comment utiliser notre modèle avec TreeView dans QML :

```
import QtQuick 2.7
import QtQuick.Window 2.2
import QtQuick.Dialogs 1.2
import qt.test 1.0


Window {
    visible: true
    width: 800
    height: 800
    title: qsTr("Tree example")

    Component {
        id: fakePlace
        TreeElement {
            property string name: getFakePlaceName()
            property string population: getFakePopulation()
            property string type: "Fake place"
            function getFakePlaceName() {
                var rez = "";
                for(var i = 0;i < Math.round(3 + Math.random() * 7);i ++) {
                    rez += String.fromCharCode(97 + Math.round(Math.random() * 25));
                }
                return rez.charAt(0).toUpperCase() + rez.slice(1);
            }
            function getFakePopulation() {
                var num = Math.round(Math.random() * 100000000);
```

```
                num = num.toString().split("").reverse().join("");
                num = num.replace(/(\d{3})/g, '$1,');
                num = num.split("").reverse().join("");
                return num[0] === ',' ? num.slice(1) : num;
            }
        }
    }

    TreeModel {
        id: treemodel
        roles: ["name","population"]

        TreeElement {
            property string name: "Asia"
            property string population: "4,164,252,000"
            property string type: "Continent"
            TreeElement {
                property string name: "China";
                property string population: "1,343,239,923"
                property string type: "Country"
                TreeElement { property string name: "Shanghai"; property string population:
"20,217,700"; property string type: "City" }
                TreeElement { property string name: "Beijing"; property string population:
"16,446,900"; property string type: "City" }
                TreeElement { property string name: "Chongqing"; property string population:
"11,871,200"; property string type: "City" }
            }
            TreeElement {
                property string name: "India";
                property string population: "1,210,193,422"
                property string type: "Country"
                TreeElement { property string name: "Mumbai"; property string population:
"12,478,447"; property string type: "City" }
                TreeElement { property string name: "Delhi"; property string population:
"11,007,835"; property string type: "City" }
                TreeElement { property string name: "Bengaluru"; property string population:
"8,425,970"; property string type: "City" }
            }
            TreeElement {
                property string name: "Indonesia";
                property string population: "248,645,008"
                property string type: "Country"
                TreeElement {property string name: "Jakarta"; property string population:
"9,588,198"; property string type: "City" }
                TreeElement {property string name: "Surabaya"; property string population:
"2,765,487"; property string type: "City" }
                TreeElement {property string name: "Bandung"; property string population:
"2,394,873"; property string type: "City" }
            }
        }
        TreeElement { property string name: "Africa"; property string population:
"1,022,234,000"; property string type: "Continent" }
        TreeElement { property string name: "North America"; property string population:
"542,056,000"; property string type: "Continent" }
        TreeElement { property string name: "South America"; property string population:
"392,555,000"; property string type: "Continent" }
        TreeElement { property string name: "Antarctica"; property string population: "4,490";
property string type: "Continent" }
        TreeElement { property string name: "Europe"; property string population:
"738,199,000"; property string type: "Continent" }
        TreeElement { property string name: "Australia"; property string population:
```

```
"29,127,000"; property string type: "Continent" }
    }

    TreeView {
        anchors.fill: parent
        model: treemodel
        TableViewColumn {
            title: "Name"
            role: "name"
            width: 200
        }
        TableViewColumn {
            title: "Population"
            role: "population"
            width: 200
        }

        onDoubleClicked: {
            var element = fakePlace.createObject(treemodel);
            treemodel.insertNode(element, index, -1);
        }
        onPressAndHold: {
            var element = treemodel.getNodeByIndex(index);
            messageDialog.text = element.type + ": " + element.name + "\nPopulation: " +
element.population;
            messageDialog.open();
        }
    }
    MessageDialog {
          id: messageDialog
          title: "Info"
      }
}
```

Double-cliquez pour ajouter un nœud, maintenez la touche enfoncée pour obtenir des informations sur le nœud.

Lire Intégration avec C ++ en ligne: https://riptutorial.com/fr/qml/topic/2254/integration-avec-c-plusplus

Chapitre 5: Liaison de propriété

Remarques

La propriété d'un objet peut se voir attribuer une valeur statique qui reste constante jusqu'à ce qu'une nouvelle valeur lui soit explicitement attribuée. Cependant, pour tirer le meilleur parti de QML et de sa prise en charge intégrée des comportements d'objets dynamiques, la plupart des objets QML utilisent des liaisons de propriétés.

Les liaisons de propriétés sont une fonctionnalité essentielle de QML qui permet aux développeurs de spécifier des relations entre différentes propriétés d'objet. Lorsque les dépendances d'une propriété changent de valeur, la propriété est automatiquement mise à jour en fonction de la relation spécifiée.

Examples

Notions de base sur les liaisons de propriétés

Considérons cet exemple simple:

```
import QtQuick 2.7
import QtQuick.Controls 2.0

ApplicationWindow {
    visible: true
    width: 400
    height: 640

    Rectangle{
        id: rect
        anchors.centerIn: parent
        height: 100
        width: parent.width
        color: "blue"
    }
}
```

Dans l'exemple ci-dessus, la largeur de `Rectangle` est liée à celle de son parent. Si vous modifiez la largeur de la fenêtre de l'application en cours d'exécution, la largeur du rectangle change également.

Un exemple plus compliqué

Dans l'exemple simple, nous définissons simplement la largeur du rectangle à celle de son parent. Considérons un exemple plus compliqué:

```
ApplicationWindow {
    visible: true
    width: 400
```

```
    height: 640

    Rectangle{
        id: rect
        anchors.centerIn: parent
        height: 100
        width: parent.width/2 + parent.width/3
        color: "blue"
    }
}
```

Dans l'exemple, nous effectuons une opération arithmétique sur la valeur liée. Si vous redimensionnez la fenêtre de l'application en cours au maximum, l'espace entre le rectangle et la fenêtre de l'application sera plus large et inversement.

Créer des liaisons avec des fichiers QML créés dynamiquement

Lorsque vous utilisez des instances de fichiers QML en les déclarant directement, chaque `property` crée une liaison. Ceci est expliqué dans les exemples ci-dessus.

Voici comment créer dynamiquement des composants:

```
var component = Qt.createComponent("Popup.qml");
var popup = component.createObject(parent, {"width": mainWindow.width, "height":
mainWindow.height});
```

Lorsque la taille de la `mainWindow` change, la taille du `PopUp` créé n'est pas affectée. Pour créer une liaison, vous devez définir la taille de la `popup` comme `popup` :

```
var component = Qt.createComponent("Popup.qml");
var options = {
    "width": Qt.binding(function() { return mainWindow.width }),
    "height": Qt.binding(function() { return mainWindow.height }),
};
var popup = component.createObject(parent, options);
```

Maintenant, la taille du `PopUp` dépendra de `mainWindow` .

Lire Liaison de propriété en ligne: https://riptutorial.com/fr/qml/topic/1967/liaison-de-propriete

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec qml	Akash Agarwal, Beriol, Community, CroCo, dangsonbk, jpnurmi, Mailerdaimon, Massimo Callegari, Mitch, Violet Giraffe
2	Animation	Akash Agarwal, Eluvatar, Violet Giraffe
3	Création d'éléments personnalisés en C ++	folibis
4	Intégration avec C ++	Beriol, Brad van der Laan, folibis, Violet Giraffe
5	Liaison de propriété	Akash Agarwal, Eluvatar, Furkanzmc, Violet Giraffe