EBook Gratis

APRENDIZAJE qml

Free unaffiliated eBook created from
Stack Overflow contributors.

#qml

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con qml

Observaciones

QML es un acrónimo que significa **Q** t **M** eta-object **L** angles. Es un lenguaje de programación declarativo que forma parte del marco Qt. El objetivo principal de QML es la creación rápida y sencilla de interfaces de usuario para sistemas de escritorio, móviles e integrados. QML permite una integración perfecta de JavaScript , ya sea directamente en el código QML o mediante la inclusión de archivos JavaScript.

Versiones

Versión Qt	Versión QtQuick	Fecha de lanzamiento
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

Instalación

QML viene con una versión más reciente del framework de aplicaciones multiplataforma Qt . Puede encontrar la versión más reciente de Qt en la sección de Descargas .

Para crear un nuevo proyecto QML en el IDE de Qt Creator , seleccione "Archivo -> Nuevo ..." y en "Aplicaciones" seleccione "Aplicación rápida de Qt". Después de hacer clic en "seleccionar",

ahora puede nombrar y establecer la ruta para este proyecto. Después de pulsar "siguiente", puede seleccionar qué componentes desea utilizar, si no está seguro, simplemente deje el valor predeterminado y haga clic en "siguiente". Los dos pasos siguientes le permitirán configurar un kit y un control de fuente si lo desea, de lo contrario, mantenga la configuración predeterminada.

Ahora ha creado una aplicación QML simple y lista para usar.

Hola Mundo

Una aplicación sencilla que muestra el texto "Hola mundo" en el centro de la ventana.

```
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
    }
}
```

Creando un simple botón

Puede transformar fácilmente cada componente en un botón pulsable usando el componente MouseArea. El siguiente código muestra una ventana de 360x360 con un botón y un texto en el centro; presionando el botón cambiará el texto:

```
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";
            }
        }
    }
}
```

Mostrar una imagen

Este ejemplo muestra el uso más simple del componente de imagen para mostrar una imagen.

La propiedad `source` imagen es un tipo de URL que puede ser un archivo con una ruta absoluta o relativa, una URL de Internet (`http://`) o un recurso Qt (`qrc:/`)

```
import QtQuick 2.3

Rectangle {
    width: 640
    height: 480

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

Evento del ratón

Este ejemplo muestra cómo se usa el evento del mouse en 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.")
            }

        }
    }


}
```

Lea Empezando con qml en línea: https://riptutorial.com/es/qml/topic/653/empezando-con-qml

Capítulo 2: Animación

Examples

Animación de números simples

Una de las animaciones muy básicas que podrías encontrar es la `NumberAnimation` . Esta animación funciona cambiando el valor numérico de una propiedad de un elemento de un estado inicial a un estado final. Considere el siguiente ejemplo completo:

```
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'
        }
    }
}
```

Animación basada en el comportamiento

Una animación basada en el comportamiento le permite especificar que cuando una propiedad cambia, el cambio debería animarse con el tiempo.

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

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

En este ejemplo, si algo cambia el valor de la barra de progreso, el cambio se animará más de 250 ms

Lea Animación en línea: https://riptutorial.com/es/qml/topic/1281/animacion

Capítulo 3: Creando elementos personalizados en C ++

Examples

Creando elementos personalizados en C ++

QML vino con un rico conjunto de elementos visuales. Usando solo QML podemos construir aplicaciones complejas con estos elementos. También es muy fácil crear su propio elemento basado en un conjunto de elementos estándar como Rectángulo, Botón, Imagen, etc. Además, podemos usar elementos como Canvas para crear elementos con pintura personalizada. Parece que podemos construir una variedad de aplicaciones solo en QML, sin tocar las capacidades de C ++. Y en realidad es cierto, pero aún así a veces nos gustaría que nuestra aplicación sea más rápida o queremos extenderla con el poder de Qt o agregar alguna oportunidad que no esté disponible en QML. Y ciertamente hay tal posibilidad en *QML* . Básicamente, *QtQuick* utiliza *Scene Graph* para pintar su contenido en un motor de renderizado de alto rendimiento basado en *OpenGL* . Para implementar nuestro propio elemento visual podemos usar 2 formas:

1. La forma tradicional para Qt usando QPainter (QQuickPaintedItem).
2. La forma QML común mediante el uso de la funcionalidad QQuickItem y OpenGL.

Es posible que el primer método parezca más fácil, pero vale la pena considerar que también es más lento que el primero, ya que QtQuick pinta el contenido del elemento en una superficie y luego lo inserta en el gráfico de la escena para que la representación sea una operación de dos pasos. Por lo tanto, usar la API de gráficos de escena directamente es siempre mucho más rápido.

Para explorar ambos métodos más cerca, creemos nuestro propio elemento que definitivamente no existe en QML, por ejemplo, un triángulo.

Declaración de clase

```
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();
};
```

Agregamos la macro Q_OBJECT para trabajar con señales. También agregamos propiedades personalizadas para especificar el color de nuestro rectángulo. Para que funcione, todo lo que necesitamos es volver a implementar la función virtual QQuiclItem :: updatePaintNode () .

Implementación de la clase.

En primer lugar definimos un constructor.

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

Tenga en cuenta que la llamada a la función setFlag () es obligatoria, de lo contrario su objeto no se agregará al gráfico de escena. A continuación, definimos una función para el dolor.

```
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;
}
```

En la primera llamada a la función, nuestro nodo aún no está creado, por lo que `oldNode` será NULL. Entonces creamos el nodo y le asignamos geometría y material. Aquí usamos *GL_TRIANGLE_FAN* para nuestra geometría para pintar un rectángulo sólido. Este punto es el

mismo que en OpenGL. Por ejemplo, para dibujar un marco de triángulo podemos cambiar el código a:

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

Puede consultar el manual de *OpenGL* para verificar otras formas. Entonces, todo lo que queda es definir el setter / getter para nuestra propiedad:

```
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();
    }
}
```

Ahora solo hay un pequeño detalle para que funcione. Necesitamos notificar a *QtQuick* del nuevo artículo. Por ejemplo, puede agregar este código a su main.cpp:

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

Y aquí está nuestro archivo de prueba 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);
    }
}
```

Como ves, nuestro artículo se comporta como todos los demás elementos de QML. Ahora vamos a crear el mismo elemento usando QPainter :

Todo lo que necesitamos es reemplazar

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

con

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

y, por supuesto, heredamos nuestra clase de `QQuickPaintedItem` lugar de `QQuickItem` . Aquí está nuestra función de pintura:

```
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);
}
```

Todo lo demás permanece sin cambios.

Lea Creando elementos personalizados en C ++ en línea:
https://riptutorial.com/es/qml/topic/6509/creando-elementos-personalizados-en-c-plusplus

Capítulo 4: Integración con C ++

Examples

Creando una vista QtQuick desde C ++

Es posible crear una vista QtQuick directamente desde C ++ y exponer a las propiedades definidas de QML C ++. En el código a continuación, el programa C ++ crea una vista QtQuick y expone a QML el alto y el ancho de la vista como propiedades.

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
    }
}
```

Creando una ventana QtQuick desde C ++

A partir de Qt 5.1 y posterior, puede usar QQmlApplicationEngine en lugar de QQuickView para cargar y renderizar un script QML.

Con QQmlApplicationEngine, necesita usar un tipo de ventana QML como su elemento raíz.

Puede obtener el contexto raíz del motor desde donde puede agregar propiedades globales al contexto al que puede acceder el motor cuando procesa 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
    }
}
```

Creando un modelo simple para TreeView

Desde Qt 5.5 tenemos un nuevo TreeView maravilloso, un control que todos hemos estado esperando. Un TreeView implementa una representación de árbol de los elementos de un modelo. En general, se parece a otras vistas QML: ListView o TableView . Pero la estructura de datos de TreeView es más compleja.

Un dato en ListView o TableView está representado por una matriz unidimensional de nodos. En TreeView cada nodo puede contener su propia matriz de nodos. Por lo tanto, a diferencia de las otras vistas en TreeView para obtener un nodo específico, debemos conocer el nodo principal, no solo la fila o columna de elemento.

Otra diferencia importante es que TreeView no admite ListModel . Para proporcionar un dato debemos subclase QAbstractItemModel . En Qt hay listas para usar clases modelo, como QFileSystemModel, que proporciona acceso al sistema de archivos local, o QSqlTableModel, que proporciona acceso a una base de datos.

En el siguiente ejemplo crearemos dicho modelo derivado de QAbstractItemModel . Pero para hacer el ejemplo más realista, sugiero hacer el modelo como ListModel pero especificado para árboles para que podamos agregar nodos desde QML. Es necesario aclarar que el modelo en sí no contiene ningún dato, solo proporciona acceso a él. Por lo tanto, proporcionar y organizar los datos es responsabilidad nuestra.

Dado que los datos del modelo se organizan en un árbol, la estructura de nodo más simple se ve de la siguiente manera, en pseudo código:

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

En C++ la declaración de nodo debe ser como sigue:

```
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;
};
```

Derivamos nuestra clase de QObject para poder crear un nodo en `QML` . Todos los nodos secundarios se agregarán a la propiedad de `nodes` , por lo que las siguientes 2 partes del código son las mismas:

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

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

Vea este artículo para saber más sobre la propiedad por defecto.

Implementación de clase de nodo:

```
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;
}
```

Para exponer una propiedad de tipo lista a QML a través de QQmlListProperty necesitamos la siguiente función 4:

```
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();
}
```

Ahora declaremos la clase modelo:

```
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();
};
```

Dado que derivamos la clase de modelo del resumen QAbstractItemModel debemos redefinir la siguiente función: datos () , indicadores () , índice () , padre () , columnCount () y rowCount () . Para que nuestro modelo pueda funcionar con `QML` definimos roleNames () . Además, así como en la clase de nodo, definimos la propiedad predeterminada para poder agregar nodos al modelo en `QML` . `roles` propiedad `roles` tendrá una lista de nombres de roles.

La implementación:

```
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 general, este código no es muy diferente de la implementación estándar, por ejemplo, ejemplo de árbol simple

En lugar de definir roles en C++ , proporcionamos una forma de hacerlo desde QML . Los eventos y métodos TreeView básicamente funcionan con QModelIndex . Personalmente, no veo mucho sentido pasarlo a qml, ya que lo único que puede hacer con él es devolvérselo al modelo.

De todos modos, nuestra clase proporciona una forma de convertir índice a nodo y viceversa. Para poder utilizar nuestras clases en QML necesitamos registrarlo:

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

Y finalmente, en un ejemplo de cómo podemos usar nuestro modelo con TreeView en 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"
      }
}
```

Haga doble clic para agregar un nodo, mantenga presionado para obtener información sobre el nodo.

Lea Integración con C ++ en línea: https://riptutorial.com/es/qml/topic/2254/integracion-con-c-plusplus

Capítulo 5: Propiedad vinculante

Observaciones

A la propiedad de un objeto se le puede asignar un valor estático que permanece constante hasta que se le asigna explícitamente un nuevo valor. Sin embargo, para hacer el uso más completo de QML y su soporte integrado para comportamientos de objetos dinámicos, la mayoría de los objetos QML utilizan enlaces de propiedad.

Los enlaces de propiedad son una característica central de QML que permite a los desarrolladores especificar relaciones entre diferentes propiedades de objetos. Cuando las dependencias de una propiedad cambian de valor, la propiedad se actualiza automáticamente de acuerdo con la relación especificada.

Examples

Conceptos básicos sobre los enlaces de propiedad

Considera este simple ejemplo:

```
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"
    }
}
```

En el ejemplo anterior, el ancho del `Rectangle` está vinculado al de su padre. Si cambia el ancho de la ventana de la aplicación en ejecución, el ancho del rectángulo también cambia.

Un ejemplo más complicado.

En el ejemplo simple, simplemente establecemos el ancho del rectángulo al de su padre. Consideremos un ejemplo más complicado:

```
ApplicationWindow {
    visible: true
    width: 400
    height: 640
```

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

En el ejemplo, realizamos una operación aritmética en el valor que se está enlazando. Si cambia el tamaño de la ventana de la aplicación en ejecución al ancho máximo, el espacio entre el rectángulo y la ventana de la aplicación será más amplio y viceversa.

Crear enlaces con archivos QML creados dinámicamente

Cuando se utilizan instancias de archivos QML declarándolos directamente, cada `property` crea un enlace. Esto se explica en los ejemplos anteriores.

Así es como creas dinámicamente los componentes:

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

Cuando cambia el tamaño de `mainWindow` , el tamaño del `PopUp` creado no se ve afectado. Para crear un enlace, establezca el tamaño de la `popup` esta manera:

```
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);
```

Ahora el tamaño del `PopUp` dependerá de `mainWindow` .

Lea Propiedad vinculante en línea: https://riptutorial.com/es/qml/topic/1967/propiedad-vinculante

Creditos

S. No	Capítulos	Contributors
1	Empezando con qml	Akash Agarwal, Beriol, Community, CroCo, dangsonbk, jpnurmi, Mailerdaimon, Massimo Callegari, Mitch, Violet Giraffe
2	Animación	Akash Agarwal, Eluvatar, Violet Giraffe
3	Creando elementos personalizados en C ++	folibis
4	Integración con C ++	Beriol, Brad van der Laan, folibis, Violet Giraffe
5	Propiedad vinculante	Akash Agarwal, Eluvatar, Furkanzmc, Violet Giraffe