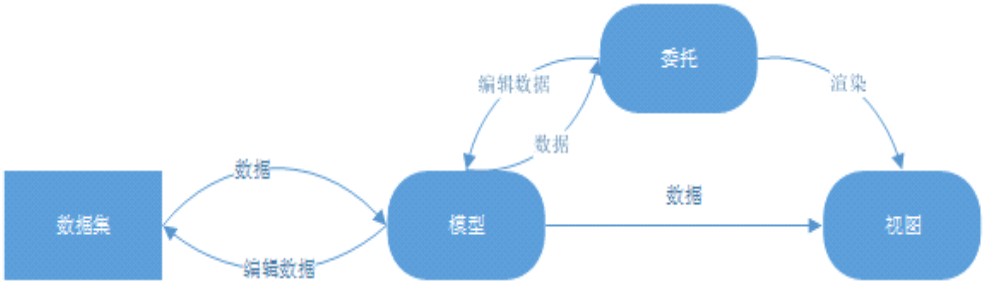
模型、视图、委托

2019年4月7日 星期日　　22:38

模型/视图架构基于MVC设计模式发展而来。MVC中，模型(Model)用来表示数据；视图(View)是界面，用来显示数据；控制(Controller)定义界面对用户输入的反应方式。

Qt中视图和控制结合在一起形成新的模型/视图架构。这同样将数据的存储和数据向用户展示进行了分离，但提供了更为简单的框架。数据和界面分离，使得相同的数据可以在不同的视图显示，而且可以扩展新的视图，而不需要改变底层的数据框架。为了灵活处理用户输入，引入了委托，也称为代理，使用它可以定制数据的渲染和编辑方式。



模型与数据通信，为其它组件提供接口。视图通过模型索引(Model Index)从模型中获取数据，模型索引用来表示数据项。委托渲染数据项，编辑项目时，委托使用模型索引直接与模型通信。

它们之间的关系为：

(1).当数据源发生变化时，模型发出信号通知视图。

(2).用户对界面进行操作时，视图发出信号提供交互信息。

(3).用户编辑数据时，代理发出信号告知模型和视图编辑器的状态。

1.　模型

模型用来提供数据，并提供了一个标准的接口供视图和委托来访问数据。这个接口由QAbstractItemModel类来定义，不管数据项是什么结构，它都会以层次结构来表示数据，这个结构中包含了数据项表。视图按照这种约定来访问模型中的数据，但这不会影响数据的显示，视图可以使用任何形式将数据显示出来。当数据发生变化时，模型会通过信号和槽机制通知关联的视图。

(1)　　模型索引 QModelIndex

模型索引使数据的表示与数据的获取相分离，每一块可以获取的数据都用一个模型索引来表示，视图和委托使用模型索引来请求数据项并显示。

模型索引包含一个模型指针，指向创建它们的模型，使用多个模型时可以避免出错。

由于模型中的数据在随时变化，因此模型引随时会变化，不需要也不应当保存一个模型索引，如果需要长时间引用一块数据，则必须使用QPersistentModelIndex创建模型索引。

QAbstractItemModel类中提供了 index接口获取模型索引，QModelIndex index(int row, int column, const QModelIndex &parent = QModelIndex())，通常可将数据看成是一个表格结构，提供行和列的索引，对于树形结构，则提供其父索引。

(2)　　数据项 QStandardItem

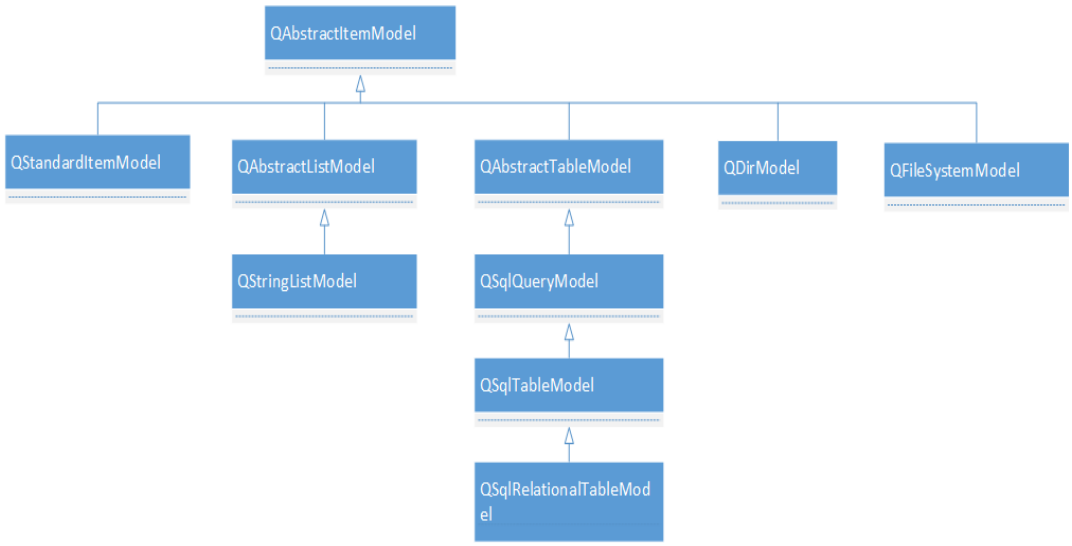
模型中的数据项可以作为各种角色在其它组件中使用，允许为其它组件提供不同类型的数据。通过为每个角色提供合适的数据，模型可以告知视图和委托如何显示数据。

Qt::DisplayRole	数据为QString类型，渲染为文本

Qt::DecorationRole	数据被渲染为图标等装饰，类型通常为QColor,QIcon,QPixmap等
Qt::EditRole	数据为QString类型，可在编辑器中编辑
Qt::ToolTipRole	数据为QString类型，显示在工具提示中
Qt::StatusTipRole	数据为QString类型，显示在状态栏
Qt::WhatThisRole	数据为QString类型，显示在“What’s This？”模式下
Qt::SizeHintRole	数据类型为QSize，表示数据项的大小，将会应用到视图

(3) QAbstractItemModel 提供了灵活的接口，可以将数据表示为列表、表格、数等形式。其子类 QAbstractListModel 和 QAbstractTableModel 为列表和表格结构的数据提供了一些常见的默认实现。

Qt定义了许多默认的模型实现，在应用中可以直接拿来使用。



QStandardItemModel：一个可被当作表模型、表格模型、数模型使用的通用模型，可用来管理复杂的树型结构树据，每一个数据项都可以包含任意数据。但它有个缺点：加载大数据时较慢。

QStringListModel：用来存储一个简单的QString项目的列表模型。

QFileSystemModel：提供本地文件系统中的文件和目录信息。

QSqlQueryModel：访问数据库。

(4) 自定义模型

当要为一个已经存在的数据结构创建一个新的模型时，需要考虑使用哪种类型的模型来为数据提供接口，如果数据结构可以表示为项目列表或表格，则可以考虑子类化QAbstractListModel 和 QAbstractTableModel来实现。

```
QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;            // 为不同角色提供数据
bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole);      // 设置对应模型索引下的数据
QVariant headerData(int section, Qt::Orientation orientation, int role = Qt::DisplayRole) const;     // 表头数据
Qt::ItemFlags flags(const QModelIndex &index) const;    // 返回模型索引是否可编辑，是否可用，编辑，选择等等状态
int rowCount(const QModelIndex &parent = QModelIndex()) const;    // 当前父索引下的行数
int columnCount(const QModelIndex &parent = QModelIndex()) const;     // 当前父索引下的列数
bool insertRows(int row, int count, const QModelIndex &parent = QModelIndex());     // 添加行 添加前需要调用 beginInsertRows，添加完成后调用 endInsertRows
```

bool removeRows(int row, int count, const QModelIndex &parent = QModelIndex());　　// 删除行　删除前需要调用 *beginRemoveRows，删除完成后调用 endRemoveRows*

2.　视图

视图用来显示数据。

视图中的标准接口由 QAbstractItemView 类提供。

视图通常管理从模型获取的数据的整体布局，它们可以自己渲染独立的数据项，也可以使用委托来渲染和编辑。

除了呈现数据，视图还处理项目间的导航以及项目选择的某些方面，如设置选择行为 SelectionBehavior、选择模式SelectionMode，上下文菜单和拖放等。

(1)　QItemSelectionModel 选择模型

这里的选择模型和前面的模型是不同的概念，在视图中被选择项目的信息都存储在 QItemSelectionModel 实例中，这样被选择项目的模型索引便保持在一个独立的模型中，与所有的视图都是独立的。在一个模型上设置多个视图时就可以实现多个视图共享。QItemSelectionModel 对象可以通过视图的接口获取和重新设置。

QItemSelectionModel *selectionModel() const;

void setSelectionModel(QItemSelectionModel *selectionModel);

QItemSelectionModel 对象会保存当前模型的指针，也可以返回操作的模型索引列表。

QModelIndexList selectedIndexes() const;

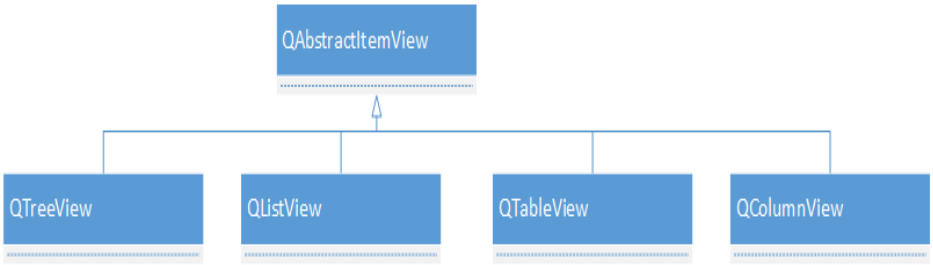
(2) Qt提供了几种常用的视图，都是使用规范的格示来显示数据，如果还要实现条形图、饼状图或更复杂的图形，就要重新实现视图类。

QListView: 将数据显示为一个列表。

QTableView: 将模型中的数据显示在一个表格中。

QTreeView: 将模型中的数据项显示在具有层次的列表中。

QColumnView: 提供一个多级视图(每点开一选项都会在它旁边出现一个菜单)。



3. 委托

委托提供特殊显示和编辑功能。委托的标准接口由 QAbstractItemDelegate 类提供。委托通过 paint() 和 sizeHint() 函数来使它们可以渲染自身的内容。

委托的编辑功能可以通过两种方式来实现，一种是使用部件来管理编辑过程，另一种是直接处理事件（通过子类化输入控件完成）。

QItemDelegate 和QStyledItemDelegate 是委托Qt提供的两种委托实现。

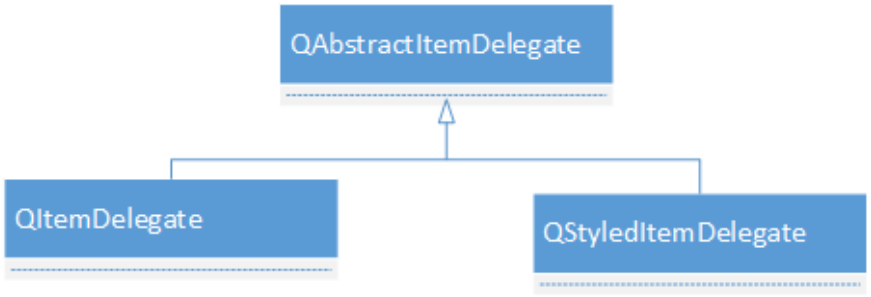
QStyledItemDelegate使用当前的样式来绘制项目，当要自定义的委托要使用样式表一起应用时，建议使用它作为基类。

QListView、QTableView和QTreeView都使用QItemDelegate来提供编辑功能，这使得它们只有普通风格的渲染。

视图可以获取和设置委托。

```
void setItemDelegate(QAbstractItemDelegate *delegate);
```

```
QAbstractItemDelegate *itemDelegate() const;
```



(1)　　自定义委托示例

当视图需要编辑器时，它会告知委托为被修改的项目提供一个编辑器部件，委托会调用 createEditor 函数提供一个合适的部件。

在自定义委托时，createEditor返回一个可编辑输入的控件对象，如果不需要编辑，则返回 nullptr，返回的指针对象不需要保存，因为视图在不需要的时候会销毁它。

setEditorData 函数将模型中的数据渲染到编辑器中。

setModelData 函数在用户完成了输入之后，将数据存储到模型中。

updateEditorGeometry 函数用来调整编辑器的位置和大小，QStyleOptionViewItem对象提供了几何布局相关的信息。

代理对象会在完成编辑后发射 closeEditor 信号来告知视图。

```
class SpinDelegate : public QItemDelegate
{
    Q_OBJECT
public:
    using QItemDelegate::QItemDelegate;
    // 创建编辑器
    QWidget* createEditor(QWidget* parent, const QStyleOptionViewItem& option, const QModelIndex& index) const override;
    // 为编辑器设置数据
    void setEditorData(QWidget* editor, const QModelIndex& index) const override;
    // 将数据写入到模型
    void setModelData(QWidget* editor, QAbstractItemModel* model, const QModelIndex& index) const override;
    // 更新编辑器的几何布局
    void updateEditorGeometry(QWidget* editor, const QStyleOptionViewItem& option, const QModelIndex& index) const
override;
};

QWidget* SpinDelegate::createEditor(QWidget* parent, const QStyleOptionViewItem& option, const QModelIndex& index) const
{
    QSpinBox* editor = new QSpinBox{parent};
    editor->setRange(0, 40);
    return editor;
}
```

```
void SpinDelegate::setEditorData(QWidget* editor, const QModelIndex& index) const
{
    QSpinBox* spinBox = dynamic_cast<QSpinBox*>(editor);
    if (spinBox)
    {
        bool isOK = false;
        int value = index.model()->data(index, Qt::EditRole).toInt(&isOK);
        if (isOK)
        {
            spinBox->setValue(value);
        }
    }
}

void SpinDelegate::setModelData(QWidget* editor, QAbstractItemModel* model, const QModelIndex& index) const
{
    QSpinBox* spinBox = dynamic_cast<QSpinBox*>(editor);
    if (spinBox)
    {
        spinBox->interpretText();  // 确保取得最新的数值
        int value = spinBox->value();
        model->setData(index, value, Qt::EditRole);
    }
}

void SpinDelegate::updateEditorGeometry(QWidget* editor, const QStyleOptionViewItem& option, const QModelIndex& index)
const
{
    editor->setGeometry(option.rect);
}
```