

## 介绍

Qt 4 推出了一组新的 item view 类，它们使用 model/view 结构来管理数据与表示层的关系。这种结构带来的功能上的分离给了开发人员更大的弹性来定制数据项的表示，它也提供一个标准的 model 接口，使得更多的数据源可以被这些 item view 使用。这里对 model/view 的结构进行了描述，结构中的每个组件都进行了解释，给出了一些例子说明了提供的这些类如何使用。

### Model/View 结构

Model-View-Controller(MVC)，是从 Smalltalk 发展而来的一种设计模式，常被用于构建用户界面。经典设计模式的著作中有这样的描述：

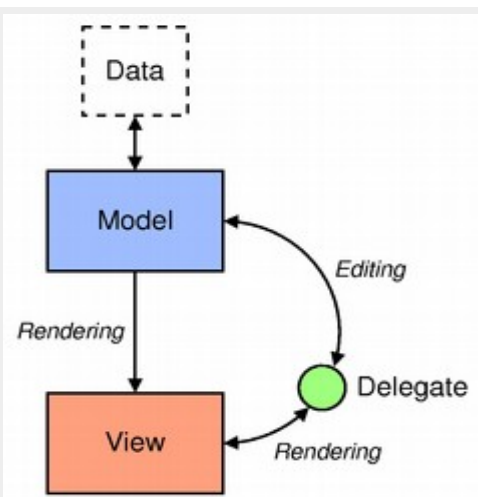
MVC 由三种对象组成。Model 是应用程序对象，View 是它的屏幕表示，Controller 定义了用户界面如何对用户输入进行响应。在 MVC 之前，用户界面设计倾向于三者揉合在一起，MVC 对它们进行了解耦，提高了灵活性与重用性。

假如把 view 与 controller 结合在一起，结果就是 model/view 结构。这个结构依然是把数据存储与数据表示进行了分离，它与 MVC 都基于同样的思想，但它更简单一些。这种分离使得在几个不同的 view 上显示同一个数据成为可能，也可以重新实现新的 view,而不必改变底层的数据结构。为了更灵活的对用户输入进行处理，引入了 delegate 这个概念。它的好处是，数据项的渲染与编程可以进行定制。

#### 模型/视图结构

如上图所示，model 与数据源通讯，并提供接口给结构中的别的组件使用。通讯的性质依赖于数据源的种类与 model 实现的方式。view 从 model 获取 model indexes,后者是数据项的引用。通过把 model indexes 提供给 model,view 可以从数据源中获取数据。

在标准的 views 中，delegate 会对数据项进行渲染，当某个数据项被选中时，delegate 通过 model indexes 与 model 直接进行交流。



总的来说，model/view 相关类可以被分成上面所提到的三组：models,views,delegates。这些组件通过抽象类来定义，它们提供了共同的接口，在某些情况下，还提供了缺省的实现。抽象类意味着需要子类化以提供完整的其他组件希望的功能。这也允许实现定制的组件。

models,views,delegates 之间通过信号，槽机制来进行通讯：

- 从 model 发出的信号通知 view 数据源中的数据发生了改变。
- 从 view 发出的信号提供了有关被显示的数据项与用户交互的信息。

- 从 delegate 发生的信号被用于在编辑时通知 model 和 view 关于当前编辑器的状态信息。

## Models

所有的 item models 都基于 QAbstractItemModel 类，这个类定义了用于 views 和 delegates 访问数据的接口。

数据本身不必存储在 model,数据可被置于一个数据结构或另外的类、文件、数据库或别的程序组件中。

关于 model 的基本概念在 Model Classes 部分中描述。

QAbstractItemModel 提供给数据一个接口，它非常灵活，基本满足 views 的需要，无论数据用以下任何样的形式表现，如 tables,lists,trees。然而，当你重新实现一个 model 时，如果它基于 table 或 list 形式的数据结构，最好从 QAbstractListModel,QAbstractTableModel 开始做起，因为它们提供了适当的常规功能的缺省实现。这些类可以被子类化以支持特殊的定制需求。子类化 model 的过程在 Create New Model 部分讨论。

- QT 提供了一些现成的 models 用于处理数据项：
- QStringListModel 用于存储简单的 QString 列表。
- QStandardItemModel 管理复杂的树型结构数据项，每项都可以包含任意数据。
- QDirModel 提供本地文件系统中的文件与目录信息。
- QSqlQueryModel, QSqlTableModel,QSqlRelationTableModel 用来访问数据库。

假如这些标准 Model 不能满足你的需要，你应该子类化 QAbstractItemModel,QAbstractListModel 或是 QAbstractTableModel 来定制。

## Views

不同的 view 都完整实现了各自的功能：QListView 把数据显示为一个列表，QTableView 把 Model 中的数据以 table 的形式表现，QTreeView 用具有层次结构的列表来显示 model 中的数据。这些类都基于 QAbstractItemView 抽象基类，尽管这些类都是现成的，完整的进行了实现，但它们都可以用于子类化以便满足定制需求。

## Delegates

QAbstractItemDelegate 是 model/view 架构中的用于 delegate 的抽象基类。缺省的 delegate 实现在 QItemDelegate 类中提供。它可以用于 Qt 标准 views 的缺省 delegate。。

## 排序

在 model/view 架构中，有两种方法进行排序，选择哪种方法依赖于你的底层 Model。

假如你的 model 是可排序的，也就是它重新实现了 QAbstractItemModel::sort()函数，QTableView 与 QTreeView 都提供了 API，允许你以编程的方式对 Model 数据进行排序。另外，

你也可以进行交互方式下的排序（例如，允许用户通过点击 view 表头的方式对数据进行排序），可以这样做：把 QHeaderView::sectionClicked() 信号与 QTableView::sortByColumn() 槽或 QTreeView::sortByColumn() 槽进行联结就好了。

另一种方法是，假如你的 model 没有提供需要的接口或是你想用 list view 表示数据，可以用一个代理 model 在用 view 表示数据之前对你的 model 数据结构进行转换。

## 便利类

许多便利类都源于标准的 view 类，它们方便了那些使用 Qt 中基于项的 view 与 table 类，它们不应该被子类化，它们只是为 Qt 3 的等价类提供一个熟悉的接口。

这些类有 QListWidget, QTreeWidget, QTableWidget, 它们提供了如 Qt 3 中的 QListBox, QListView, QTable 相似的行为。

这些类比 View 类缺少灵活性，不能用于任意的 models，推介使用 model/view 的方法处理数据。

## 使用模型和视图

下面的章节解释如何使用 Qt 中模型/视图模式。每个部分包括一个例子，随后的一部分是讨论如何创建新的组件。

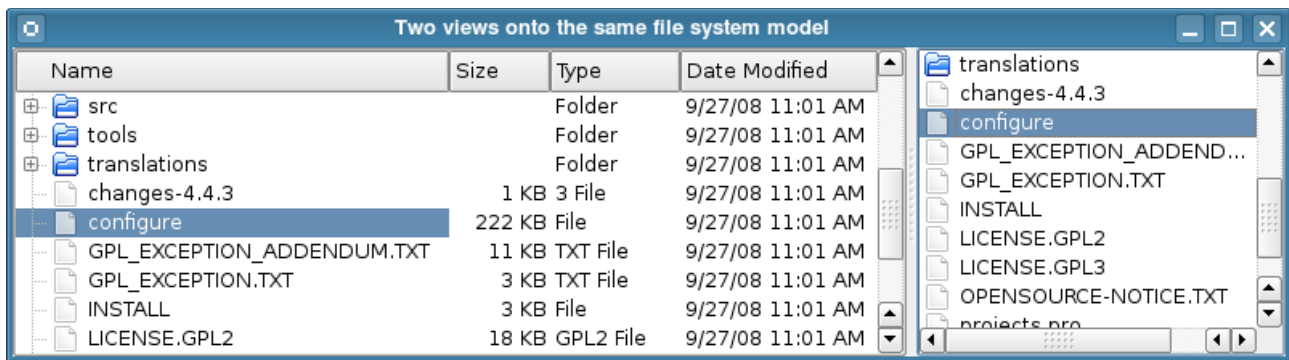
### 包含在 Qt 的两种模式

Qt 提供的两个标准模型 QStandardItemModel 和 QFileSystemModel。QStandardItemModel 是一个多用途的模型，该模型可以用来展示列表、表、树视图的各种不同的数据结构。这种模式也保存数据的项。QFileSystemModel 是一个模型，维护有关目录信息的信息。其结果是，它不保持数据本身，而仅仅是表示本地归档系统中的文件和目录。

QFileSystemModel 提供了一个现成的使用模式进行实验，并可以很容易地配置使用现有的数据。利用这个模型，我们可以展示如何用现成的视图建立一个模型，并探讨了如何使用模型索引来处理数据。

### 使用视图与现有模型

QListView 和 QTreeView 类使用 QFileSystemModel 是最合适的视图。下面介绍的例子显示了树形视图中目录的内容，旁边的列表视图相同的信息。该视图共享用户的选择，这样选择的项目在这两种视图中更突出。



我们建立了一个 `QFileSystemModel`，并创建一些视图来显示一个目录的内容。这显示了使用一个模型的最简单的方法。模型的结构和用途在一个单一的 `main()` 函数中执行：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplitter *splitter = new QSplitter;

    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());
```

该模型被设置为从一个特定的文件系统中使用的数据。`setRootPath()`的调用告诉文件系统哪个驱动器导出给视图。

我们创建了两个视图，让我们可以用两种不同的方式检查模型中的项目：

```
QTreeView *tree = new QTreeView(splitter);
tree->setModel(model);
tree->setRootIndex(model->index(QDir::currentPath()));

QListView *list = new QListView(splitter);
list->setModel(model);
list->setRootIndex(model->index(QDir::currentPath()));
```

视图以与其他部件相同的方式构造。建立一个视图显示项目是简单地以目录模型作为参数调用 `setModel()` 函数。我们通过调用每个视图的 `setRootIndex()` 函数，传递一个合适的模型索引，为当前目录筛选通过模型提供的的数据。

在这种情况下使用的 `index()` 函数是唯一的 `QFileSystemModel`；我们提供它一个目录，并返回一个模型索引。型号指标在模型类的讨论。

该功能的其余部分在一个分离器内显示视图，并运行应用程序的事件循环：

```
splitter->setWindowTitle("Two views onto the same file system model");
splitter->show();
```

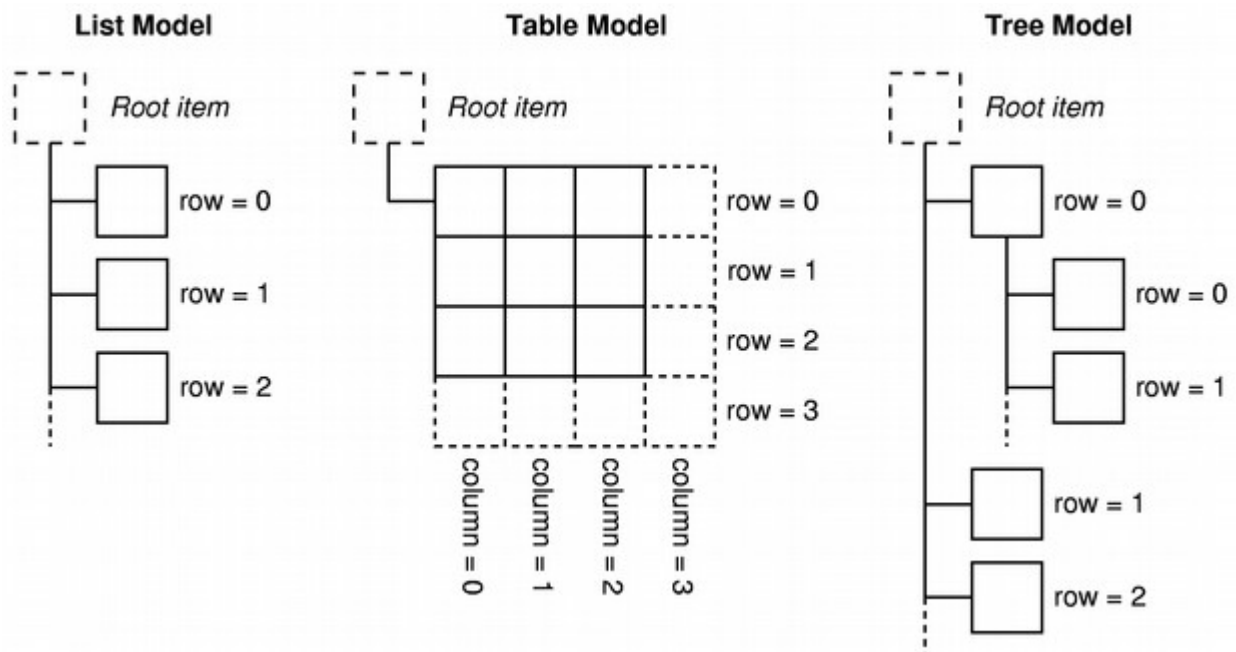
```
return app.exec();  
}
```

在上面的例子中，我们忽略了如何处理项目的选择。这个主题进行了更详细的信息在 [Handling Selections in Item Views](#) 部分。

## Model 类

### 基本概念

在 model/view 构架中，model 为 view 和 delegates 使用数据提供了标准接口。在 Qt 中，标准接口在 QAbstractItemModel 类中被定义。不管数据在底层以何种数据结构存储，QAbstractItemModel 的子类会以层次结构的形式来表示数据，结构中包含了数据项表。我们按这种约定来访问 model 中的数据项，但这个约定不会对如何显示这些数据有任何限制。数据发生改变时，model 通过信号槽机制来通知关联的 views。



### Model Indexes

为了使数据存储与数据访问分开，引入了 model index 的概念。通过 model index，可以引用 model 中的数据项，Views 和 delegates 都使用 indexes 来访问数据项，然后再显示出来。因此，只有 model 需要了解如何获取数据，被 model 管理的数据类型可以非常广泛地被定义。Model indexes 包含一个指向创建它们的 model 的指针，这会在配合多个 model 工作时避免混乱。

```
QAbstractItemModel *model = index.model();
```

model indexes 提供了一项数据信息的临时引用，通过它可以访问或是修改 model 中的

数据。既然 model 有时会重新组织内部的数据结构，这时 model indexes 便会失效，因此不应该保存临时的 model indexes。假如需要一个对数据信息的长期的引用，那么应该创建一个 persistent model index。这个引用会保持更新。临时的 model indexes 由 QModelIndex 提供，而具有持久能力的 model indexes 则由 QPersistentModelIndex 提供。在获取对应一个数据项的 model index 时，需要考虑有关于 model 的三个属性：行数，列数，父项的 model index。

## 行与列

在最基本的形式中，一个 model 可作为一个简单的表来访问，每个数据项由行，列数来定位。这并不意味着 底层的数据用数组结构来存储。行和列的使用仅仅是一种约定，它允许组件之间相互通讯。可以通过指定 model 中的行列数来获取任一项数据，可以得到与数据项一一对应的那个 index。

```
QModelIndex index = model->index(row, column, ...);
```

Model 为简单的、单级的数据结构如 list 与 tables 提供了接口，它们如上面代码所显示的那样，不再需要别的信息被提供。当我们在获取一个 model index 时，我们需要提供另外的信息。

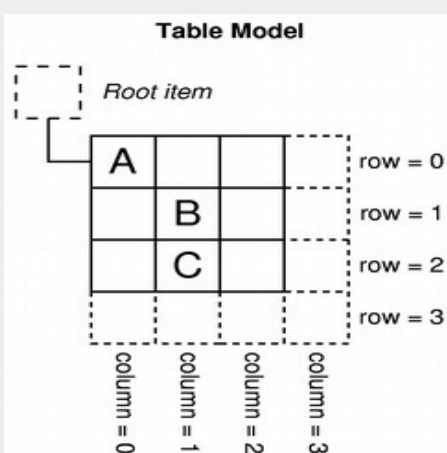
右图代表一个基本的 table model，它的每一项用一对行列数来定位。通过行列数，可以获取代表一个数据项的 model index。

```
QModelIndex indexA = model->index(0, 0, QModelIndex());
```

```
QModelIndex indexB = model->index(1, 1, QModelIndex());
```

```
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

一个 model 的顶级项,由 QModelIndex()取得，它们上式被用作父项。



## 父项

类似于表的接口在搭配使用 table 或 list view 时理想的，这种行列系统与 view 显示的方式是确切匹配的。然则，像 tree views 这种结构需要 model 提供更为灵活的接口来访问数据项。每个数据项可能是别的项的父项，上级的项可以获取下级项的列表。

当获取 model 中数据项的 index 时，我们必须指定关于数据项的父项的信息。在 model 外部，引用一个数据项的唯一方法就是通过 model index,因此需要在求取 model index 时指定父项的信息。

```
QModelIndex index = model->index(row, column, parent);
```



右图中，A 项和 C 项作为 model 中顶层的兄弟项：

```
QModelIndex indexA = model->index(0, 0,
```

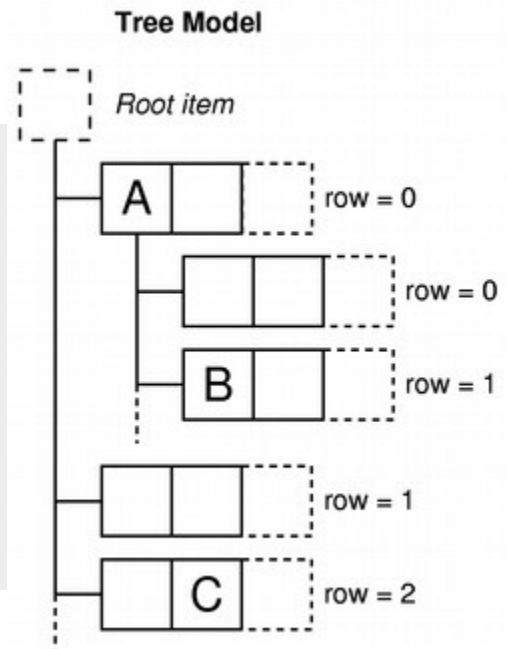
```
QModelIndex());
```

```
QModelIndex indexC = model->index(2, 1,
```

```
QModelIndex());
```

A 有许多孩子，它的一个孩子 B 用以下代码获取：

```
QModelIndex indexB = model->index(1, 0, indexA);
```



## 项角色

model 中的项可以作为各种角色来使用，这允许为不同的环境提供不同的数据。举例来说，Qt::DisplayRole 被用于访问一个字符串，它作为文本会在 view 中显示。典型地，每个数据项都可以为许多不同的角色提供数据，标准的角色在 Qt::ItemDataRole 中定义。我们可以通过指定 model index 与角色来获取我们需要的数据：

```
QVariant value = model->data(index, role);
```

角色指出了从 model 中引用哪种类型的数据。views 可以用不同的形式显示角色，因此为每个角色提供正确的信息是非常重要的。项目数据最常见的用途涵盖了 Qt::ItemDataRole 定义的标准角色。通过为每个角色提供适当数据，model 也为 views 和 delegates 提供了暗示，如何正确地 把这些数据项显给用户。不同的 views 可以自由地解析或忽略这些数据信息，对于特殊的场合，也可以定义一些附加的角色。

## 概念总结：

1. Model indexes 为 views 与 delegates 提供 model 中数据项定位的信息，它与底层的数据结构无关。
2. 通过指定行，列数，父项的 model index 来引用数据项。
3. 依照别的组件的要求，model indexes 被 model 构建。
4. 使用 index()时，如果指定了有效的父项的 model index,那么返回得到的 model index 对应于父项的某个孩子。
5. 使用 index()时，如果指定了无效的父项的 model index,那么返回得到的 model index 对应于顶层项的某个孩子。
6. 角色对一个数据项包含的不同类型的数据给出了区分。

## 使用 Model Indexes

```

QDirModel *model = new QDirModel;
QModelIndex parentIndex = model->index(QDir::currentPath());
int numRows = model->rowCount(parentIndex);
for (int row = 0; row < numRows; ++row)
{
    QModelIndex index = model->index(row, 0, parentIndex);
    tring text = model->data(index, Qt::DisplayRole).toString();
    // Display the text in a widget.
}

```

以上的例子说明了从 model 中获取数据的基本原则：

- model 的尺寸可以从 rowCount() 与 columnCount() 中得出。这些函数通常都需要一个表示父项的 model index。
- model indexes 用来从 model 中访问数据项，数据项用行，列，父项 model index 定位。
- 为了访问 model 顶层项，可以使用 QModelIndex() 指定。
- 数据项为不同的角色提供不同的数据。为了获取数据，除了 model index 之外，还要指定角色。

## 创建新的 Models

model/view 组件之间功能的分离，允许创建 model 利用现成的 views。这也可以使用标准的功能图形用户接口组件像 QListView, QTableView 和 QTreeView 来显示来自各种数据源的数据。

QAbstractListModel 类提供了非常灵活的接口，允许数据源以层次结构的形式来管理信息，也允许以某种方式对数据进行插入、删除、修改和存储。它也提供了对拖拽操作的支持。

QAbstractListModel 与 QAbstractTableModel 为简单的非层次结构的数据提供了接口，对于比较简单的 list 和 table models 来说，这是不错的一个开始点。

## 设计一个 Model

当我们为存在的数据结构新建一个 model 时，首先要考虑的问题是应该选用哪种 model 来为这些数据提供接口。

假如数据结构可以用数据项的列表或表来表示，那么可以考虑子类化

QAbstractListModel 或 QAbstractTableModel，既然这些类已经合理地对许多功能提供缺省实现。

然而，假如底层的数据结构只能表示成具有层次结构的树型结构，那么必须得子类化 QAbstractItemModel。

无论底层的数据结构采取何种形式，在特定的 model 中实现标准的 QAbstractItemModel



API 总是一个不错的主意，这使得可以使用更自然的方式对底层的数据结构进行访问。这也使得用数据构建 model 更为容易，其他的 model/view 组件也可以使用标准的 API 与之进行交互。

### 一个只读 model 示例

这个示例实现了一个简单的，非层次结构的，只读的数据 model，它基于 QStringListModel 类。它有一个 QStringList 作为它内部的数据源，只实现了一些必要的接口。为了简单化，它子类化了 QAbstractListModel，这个基类提供了合理的缺省行为，对外提供了比 QAbstractItemModel 更为简单的接口。当我们实现一个 model 时，不要忘了 QAbstractItemModel 本身不存储任何数据，它仅仅提供了给 views 访问数据的接口。

```
class QStringListModel : public QAbstractListModel
{
    Q_OBJECT
public:
    QStringListModel(const QStringList &strings, QObject *parent = 0)
        : QAbstractListModel(parent), stringList(strings) {}

    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
        int role = Qt::DisplayRole) const;
private:
    QStringList stringList;
};
```

除了构造函数，我们仅需要实现两个函数：rowCount() 返回 model 中的行数，data() 返回与特定 model index 对应的数据项。具有良好行为的 model 也会实现 headerData()，它返回 tree 和 table views 需要的，在标题中显示的数据。

因为这是一个非层次结构的 model，我们不必考虑父子关系。假如 model 具有层次结构，我们也应该实现 index() 与 parent() 函数。

### Model 的尺寸

我们认为 model 中的行数与 string list 中的 string 数目一致：

```
int QStringListModel::rowCount(const QModelIndex &parent) const
{
    return stringList.count();
}
```

在缺省情况下，从 QAbstractListModel 派生的 model 只具有一列，因此不需要实现 columnCount()。

### Model 标题与数据

```

QVariant QStringListModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (index.row() >= stringList.size())
        return QVariant();

    if (role == Qt::DisplayRole)
        return stringList.at(index.row());

    else
        return QVariant();
}

QVariant QStringListModel::headerData(int section, Qt::Orientation orientation,
int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    if (orientation == Qt::Horizontal)
        return QString("Column %1").arg(section);
    else
        return QString("Row %1").arg(section);
}

```

一个数据项可能有多个角色，根据角色的不同输出不同的数据。上例中，model 中的数据项只有一个角色，DisplayRole，然而我们也可以重用提供给 DisplayRole 的数据，作为别的角色使用，如我们可以作为 TooltipRole 来用。

### 可编辑的 model

上面我们演示了一个只读的 model，它只用于向用户显示，对于许多程序来说，可编辑的 list model 可能更有用。我们只需要给只读的 model 提供另外两个函数 flags()与 setData()的实现。下列函数声明被添加到类定义中：

```

Qt::ItemFlags flags(const QModelIndex &index) const;

bool setData(const QModelIndex &index, const QVariant &value,
            int role = Qt::EditRole);

```

### **让 model 可编辑**

delegate 会在创建编辑器之前检查数据项是否是可编辑的。model 必须得让 delegate 知道它的数据项是可编辑的。这可以通过为每一个数据项返回一个正确的标记得到的，在本例中，我们假设所有的数据项都是可编辑可选择的：

```

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

    return QAbstractItemModel::flags(index) | Qt::ItemIsEditable;
}

```

我们不必知道 delegate 执行怎样实际的编辑处理过程，我们只需提供给 delegate 一个方法，delegate 会使用它对 model 中的数据进行设置。这个特殊的函数就是 setData():

```

bool StringListModel::setData(const QModelIndex &index,
const QVariant &value, int role)
{
    if (index.isValid() && role == Qt::EditRole) {
        stringList.replace(index.row(), value.toString());
        emit dataChanged(index, index);
        return true;
    }
    return false;
}

```

当数据被设置后，model 必须得让 views 知道一些数据发生了变化，这可通过发射一个 dataChanged() 信号实现。因为只有一个数据项发生了变化，因此在信号中说明的变化范围只限于一个 model index。

另外，根据 data 函数需要，改变，添加了 Qt::EditRole 测试：

```

QVariant StringListModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (index.row() >= stringList.size())
        return QVariant();

    if (role == Qt::DisplayRole || role == Qt::EditRole)
        return stringList.at(index.row());
    else
        return QVariant();
}

```

## 插入、删除行

在 model 中改变行数与列数是可能的。当然在本列中，只考虑行的情况，我们只需要重新实现插入、删除 的函数就可以了，下面应在类定义中声明：

```
bool insertRows(int position, int rows, const QModelIndex &index = QModelIndex());  
bool removeRows(int position, int rows, const QModelIndex &index = QModelIndex());
```

既然 model 中的每行对应于列表中的一个 string,因此 , insertRows()函数在 string list 中指定位置插入一个空 string, , 父 index 通常用于决定 model 中行列的位置 , 本例中只有一个单独的顶级项 , 因此只需要在 list 中插入空 string。

```
bool StringListModel::insertRows(int position, int rows, const QModelIndex &parent)  
{  
    beginInsertRows(QModelIndex(), position, position+rows-1);  
    for (int row = 0; row < rows; ++row) {  
        stringList.insert(position, "");  
    }  
    endInsertRows();  
    return true;  
}
```

beginInsertRows()通知其他组件行数将会改变。endInsertRows()对操作进行确认与通知。返回 true 表示成功。

删除操作与插入操作类似 :

```
bool StringListModel::removeRows(int position, int rows, const QModelIndex &parent)  
{  
    beginRemoveRows(QModelIndex(), position, position+rows-1);  
    for (int row = 0; row < rows; ++row) {  
        stringList.removeAt(position);  
    }  
    endRemoveRows();  
    return true;  
}
```

## View 类

在 model/view 架构中 , view 从 model 中获得数据项然后显示给用户。数据显示的方式不必与 model 提供的表示方式相同 , 可以与底层存储数据项的数据结构完全不同。

内容与显式的分离是通过由 QAbstractItemModel 提供的标准模型接口 , 由 QAbstractItemView 提供的标准视图接口共同实现的。普遍使用 model index 来表示数据项。view 负责管理从 model 中读取的数据的外观布局。

它们自己可以去渲染每个数据项 , 也可以利用 delegate 来既处理渲染又进行编辑。

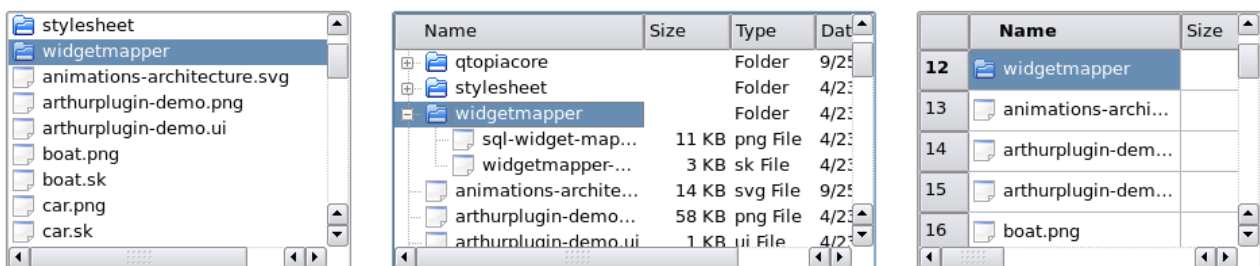
除了显示数据 , views 也处理数据项的导航 , 参与有关于数据项选择的部分功能。view 也实现一些基本的用户接口特性 , 如上下文菜单与拖拽功能。 view 也为数据项提供了缺省的

编程功能，也可搭配 delegate 实现更为特殊的定制编辑的需求。

一个 view 创建时必不需要 model,但在它能显示一些真正有用的信息之前，必须提供一个 model。view 通过使用 selections 来跟踪用户选择的数据项。每个 view 可以维护单独使用的 selections，也可以在多个 views 之间共享。有些 views,如 QTableView 和 QTreeView,除数据项之外也可显示标题(Headers)，标题部分通过一个 view 来实现，QHeaderView。标题与 view 一样总是从相同的 model 中获取数据。从 model 中获取数据的函数是 QabstractItemModel::headerData()，一般总是以表单的形式中显示标题信息。可以从 QHeaderView 子类化，以实现更为复杂的定制化需求。

## 使用现成的 view

Qt 提供了三个现成的 view 类，它们能够以用户熟悉的方式显示 model 中的数据。 QListView 把 model 中的数据项以一个简单的列表的形式显示，或是以经典的图标视图的形式显示。QTreeView 把 model 中的数据项作为具有层次结构的列表的形式显示，它允许以紧凑的深度嵌套的结构进行显示。QTableView 却是把 model 中的数据项以表格的形式展现，更像是一个电子表格应用程序的外观布局。



以上这些标准 view 的行为足以应付大多数的应用程序，它们也提供了一些基本的编辑功能，也可以定制特殊的需求。

## 使用 model

以前的例子中创建过一个 string list model,可以给它设置一些数据，再创建一个 view 把 model 中的内容展示出来：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // Unindented for quoting purposes:
    QStringList numbers;
    numbers << "One" << "Two" << "Three" << "Four" << "Five";
    QAbstractItemModel *model = new QStringListModel(numbers);
    //要注意的是，这里把 QStringListModel 作为一个 QAbstractItemModel 来使用。这样我们就可以
    //使用 model 中的抽象接口，而且如果将来我们用别的 model 代替了当前这个 model,这些代码也会照样工作。
    //QListView 提供的列表视图足以满足当前这个 model 的需要了。
    QListView *view = new QListView;
    view->setModel(model);
```

```

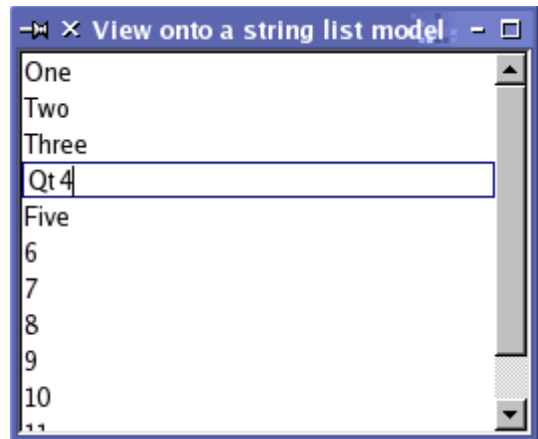
view->show();
return app.exec();
}

```

view 会渲染 model 中的内容，通过 model 的接口来访问它的数据。当用户试图编辑数据项时，view 会使用缺省的 delegate 来提供一个编辑构件。

### 一个 model,多个 views

为多个 views 提供相同的 model 是非常简单的事情，只要为每个 view 设置相同的 model。

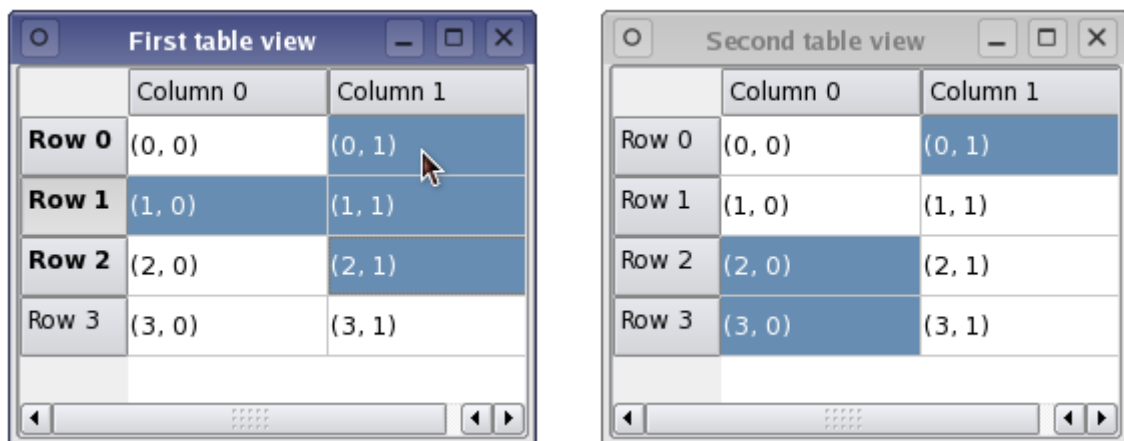


```

QTableView *firstTableView = new QTableView;
QTableView *secondTableView = new QTableView;
firstTableView->setModel(model);
secondTableView->setModel(model);

```

在 model/view 架构中信号、槽机制的使用意味着 model 中发生的改变会传递中联结的所有 view 中，这保证了 不管我们使用哪个 view，访问的都是同样的一份数据。



上面的图展示了一个 model 上的两个不同的 views,尽管在不同的 view 中显示的 model 中的数据是一致的，每个 view 都维护它们自己的内部选择模型，但有时候在某些情况下，共享一个选择模型也是合理的。

### 处理数据项的选择

view 中数据项选择机制由 QItemSelectionModel 类提供。所有标准的 view 缺省都构建它们自己的选择模型，以标准的方式与它们交互。选择模型可以用 selectionModel()函数取得，



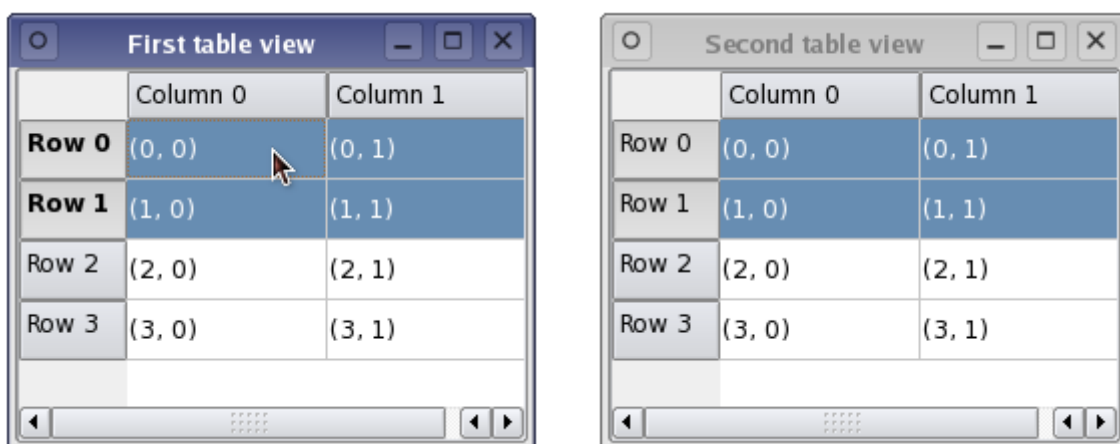
替代的选择模型也可以通过 setSelectionModel()来设置。当我们想在一个 model 上提供多个一致的 views 时，这种对选择模型的控制能力非常有用。通常来讲，除非你子类化一个 model 或 view,你不必直接操纵 selections 的内容。

## 多个 views 之间共享选择

接着上边的例子，我们可以这样：

```
secondTableView->setSelectionModel(firstTableView->selectionModel());
```

现在所有 views 都在同样的选择模型上操作，数据与选择项都保持同步。



上面的例子中，两个 view 的类型是相同的，假如这两个 view 类型不同，那么所选择的数据项在每个 view 中的表现形式会有很大的不同。例如，在一个 table view 中一个连续的选择，在一个 tree view 中表现出来的可能会是几个高亮的数据项片断的组合。

## 在 views 中选择数据项

用于新的 view 类中的选择模型比 Qt3 中的模型有了很大的改进。它为基于 model/view 架构的选择提供了更为全面的描述。尽管对提供了的 views 来说，负责操纵选择的标准类已经足以应付，但是你也可以创建特定的选择模型来满足你特殊的需求。

关于在 view 被选择的数据项的信息保持在 QItemSelectionModel 类的实例中。它也为每个独立的 model 中的数据项维护 model indexes 信息，与任何 views 都关联关系。既然一个 model 可用于多个 views,那么在多个 views 之间共享选择信息也是可以做到的，这使得多个 views 可以以一致的方式进行显示。

选择由多个选择范围组成。通过仅仅记录开始 model indexes 与结束 model indexes，最大化地记录了可以选择的范围。非连续选择数据项由多个选择范围来描述。选择模型记录 model indexes 的集合来描述一个选择。最近选择的数据项被称为 currentselection。应用程序可以通过使用某种类型的选择命令来修改选择的效果。

在进行选择操作时，可以把 QItemSelectionModel 看成是 model 中所有数据项选择状态的

一个记录。一旦建立一个选择模型，所有项的集合都可以选择，撤消选择，或者选择状态进行切换而不需要知道哪个数据项是否已经被选择过。所有被选择的项的 indexes 在任何时候都可以得到，通过信号槽机制可以通知别的组件发生的变化。

## 使用选择模型

标准 view 类提供了缺省的选择模型，它们可以在大次数程序中使用。一个 view 中的选择模型可以通过调用 view 的函数 selectionModel()取得，也可以通过 setSelectionModel()在多个 views 之间共享选择模型，因此总的来说构建一个新的模型一般情况不太必要。

通过给 QItemSelection 指定一个 model,一对 model indexes，可以创建一个选择。 indexes 的用法依赖于给定的 model,这两个 indexes 被解释成选择的区块中的左上角项和右下角项。model 中的项的选择服从于选择模型。

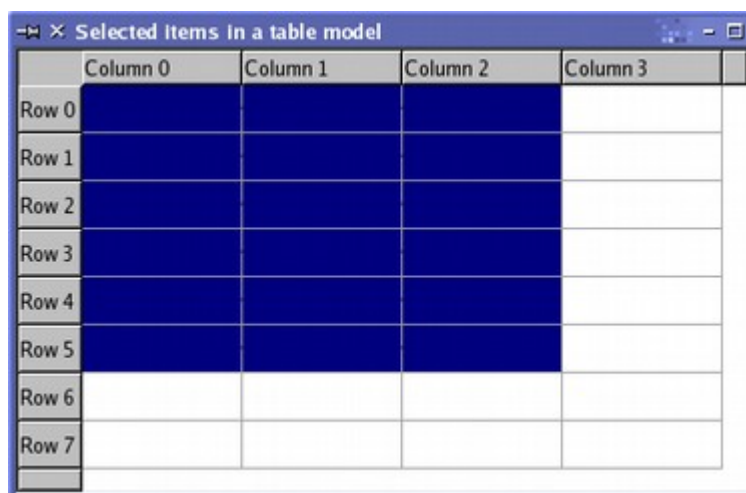
## 选择项

构建一个 table model，它有 32 个项，用一个 table view 进行显示：

```
TableModel *model = new TableModel(8, 4, &app);
QTableView *table = new QTableView(0);
table->setModel(model);
QItemSelectionModel *selectionModel = table->selectionModel();
QModelIndex topLeft;
QModelIndex bottomRight;
topLeft = model->index(0, 0, QModelIndex());
bottomRight = model->index(5, 2, QModelIndex());

QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

结果如下：



|       | Column 0 | Column 1 | Column 2 | Column 3 |
|-------|----------|----------|----------|----------|
| Row 0 |          |          |          |          |
| Row 1 |          |          |          |          |
| Row 2 |          |          |          |          |
| Row 3 |          |          |          |          |
| Row 4 |          |          |          |          |
| Row 5 |          |          |          |          |
| Row 6 |          |          |          |          |
| Row 7 |          |          |          |          |

## 读取选择状态

存储在选择模型中 `indexes` 可以用 `selectionIndexes()` 函数来读取。它返回一个未排序的 `model indexes` 列表，我们可以遍历它，如果我们知道他们关联于哪个 `model` 的话。

```
QModelIndexList indexes = selectionModel->selectedIndexes();
QModelIndex index;
foreach(index, indexes) {
    QString text = QString("(%1,%2").arg(index.row()).arg(index.column());
    model->setData(index, text);
}
```

选择模型在选择发生变化时会发出信号。这用于通知别的组件包括整体与当前焦点项所发生的变化。我们可以连接 `selectionChanged()` 信号到一个槽，检查当信号产生时哪些项被选择或被取消选择。这个槽被调用时带有两个参数，它们都是 `QItemSelection` 对象，一个包含新被选择的项，另一个包含新近被取消选择的项。下面的代码演示了给新选择的项添加数据内容，新近被取消选择的项的内容被清空。

```
void MainWindow::updateSelection(const QItemSelection &selected, const QItemSelection
&deselected)
{
    QModelIndex index;
    QModelIndexList items = selected.indexes();
    foreach (index, items) {
        QString text = QString("(%1,%2").arg(index.row()).arg(index.column());
        model->setData(index, text);
    }
    items = deselected.indexes();
    foreach (index, items)
        model->setData(index, "");
}
```

也可以通过响应 `currentChanged()` 信号来跟踪当前焦点项.对应的槽就有两个接收参数，一个表示之前的焦点，另一个表示当前的焦点。

```
void MainWindow::changeCurrent(const QModelIndex &current,
const QModelIndex &previous)
{
    statusBar()->showMessage(
        tr("Moved from (%1,%2) to (%3,%4)")
        .arg(previous.row()).arg(previous.column())
        .arg(current.row()).arg(current.column()));
}
```

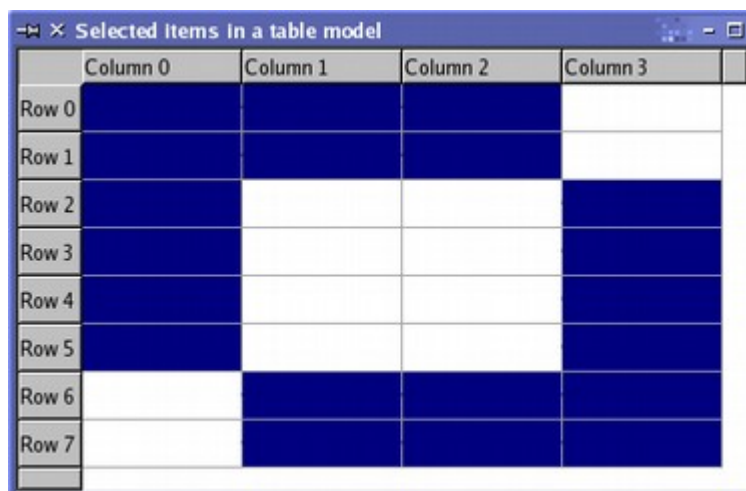
## 更新选择

选择指令是通过选择标志提供的，它被定义在 `QItemSelectionModel::SelectionFlag` 中。常用的有 `Select` 标记，`Toggle` 标记，`Deselect` 标记, `Current` 标记，`Clear` 标记，其意义一目了然。

沿上面例子的结果执行以下代码：

```
QItemSelection toggleSelection;  
topLeft = model->index(2, 1, QModelIndex());  
bottomRight = model->index(7, 3, QModelIndex());  
toggleSelection.select(topLeft, bottomRight);  
selectionModel->select(toggleSelection, QItemSelectionModel::Toggle);
```

结果如下:

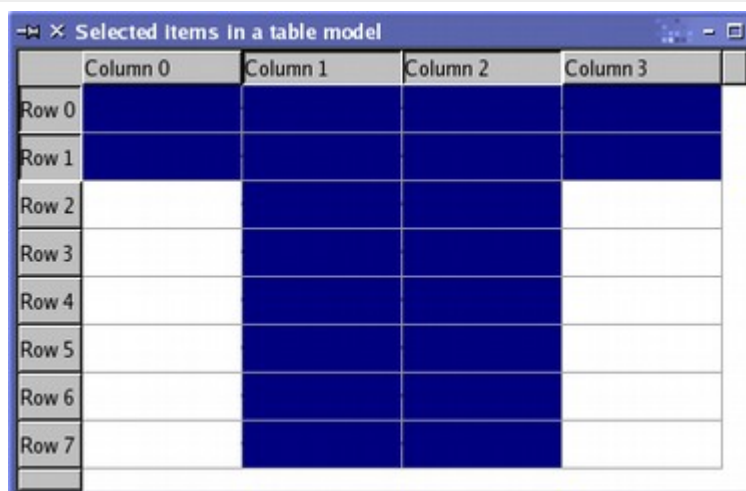


|       | Column 0 | Column 1 | Column 2 | Column 3 |
|-------|----------|----------|----------|----------|
| Row 0 |          |          |          |          |
| Row 1 |          |          |          |          |
| Row 2 |          |          |          |          |
| Row 3 |          |          |          |          |
| Row 4 |          |          |          |          |
| Row 5 |          |          |          |          |
| Row 6 |          |          |          |          |
| Row 7 |          |          |          |          |

缺省情况下，选择指令只针对单个项(由 model indexes 指定)。然而，选择指令可以通过与另外标记的结合来改变整行和整列。举例来说，假如你只使用一个 index 来调用 select(), 但是用 Select 标记与 Rows 标记的组合，那么包括那个项的整行都将被选择。看以下示例：

```
QItemSelection columnSelection;  
topLeft = model->index(0, 1, QModelIndex());  
bottomRight = model->index(0, 2, QModelIndex());  
columnSelection.select(topLeft, bottomRight);  
selectionModel->select(columnSelection,  
QItemSelectionModel::Select | QItemSelectionModel::Columns);  
QItemSelection rowSelection;  
topLeft = model->index(0, 0, QModelIndex());  
bottomRight = model->index(1, 0, QModelIndex());  
rowSelection.select(topLeft, bottomRight);  
selectionModel->select(rowSelection,  
QItemSelectionModel::Select | QItemSelectionModel::Rows);
```

结果如下



|       | Column 0 | Column 1 | Column 2 | Column 3 |
|-------|----------|----------|----------|----------|
| Row 0 |          |          |          |          |
| Row 1 |          |          |          |          |
| Row 2 |          |          |          |          |
| Row 3 |          |          |          |          |
| Row 4 |          |          |          |          |
| Row 5 |          |          |          |          |
| Row 6 |          |          |          |          |
| Row 7 |          |          |          |          |

## 选择模型中所有项

为了选择 model 中的所有项，必须先得创建一个选择，它包括当前层次上的所有项：

```
QModelIndex topLeft = model->index(0, 0, parent);
QModelIndex bottomRight = model->index(model->rowCount(parent)-1,
model->columnCount(parent)-1, parent);

QItemSelection selection(topLeft, bottomRight);
selectionModel->select(selection, QItemSelectionModel::Select);
```

顶级 index 可以这样：

```
QModelIndex parent = QModelIndex();
```

对具有层次结构的 model 来说，可以使用 hasChildren() 函数来决定给定项是否是其它项的父项。

## Delegate 类

与 MVC 模式不同，model/view 结构没有用于与用户交互的完全独立的组件。一般来讲，view 负责把数据展示给用户，也处理用户的输入。为了获得更多的灵活性，交互通过 delegate 执行。它既提供输入功能又负责渲染 view 中的每个数据项。控制 delegates 的标准接口在 QAbstractItemDelegate 类中定义。Delegates 通过实现 paint() 和 sizeHint() 以达到渲染内容的目的。然而，简单的基于 widget 的 delegates 可以从 QItemDelegate 子类化，而不是 QAbstractItemDelegate，这样可以使用它提供的上述函数的缺省实现。delegate 可以使用 widget 来处理编辑过程，也可以直接对事件进行处理。

### 使用现成的 delegate

Qt 提供的标准 views 都使用 QItemDelegate 的实例来提供编辑功能。它以普通的风格来为每个标准 view 渲染数据项。这些标准的 views 包括：QListView, QTableView, QTreeView。所有标准的角色都通过标准 views 包含的缺省 delegate 进行处理。一个 view 使用的 delegate 可以用 itemDelegate() 函数取得，而 setItemDelegate() 函数可以安装一个定制 delegate。

### 一个简单的 delegate

注意这里

这个 delegate 使用 QSpinBox 来提供编辑功能。它主要想用于显示整数的 models 上。尽管我们已经建立了一个基于整数的 table model，但我们也可以使用 QStandardItemModel，因为 delegate 可以控制数据的录入。我们又建了一个 table view 来显示 model 的内容，用我们定制的 delegate 来编辑。

我们从 QItemDelegate 子类化，这样可以利用它缺省实现的显示功能。当然我们必需提

供函数来管理用于编辑的 widget:

```
class SpinBoxDelegate : public QItemDelegate
{
    Q_OBJECT
public:
    SpinBoxDelegate(QObject *parent = 0);
    QWidget *createEditor(QWidget *parent, constQStyleOptionViewItem &option,
        const QModelIndex &index) const;
    voidsetEditorData(QWidget *editor, constQModelIndex &index) const;
    void setModelData(QWidget *editor,QAbstractItemModel *model,
        const QModelIndex &index) const;
    voidupdateEditorGeometry(QWidget *editor,
        const QStyleOptionViewItem &option, constQModelIndex &index) const;
};
```

需要注意的是,当一个 delegate 创建时,不需要安装一个 widget,只有在真正需要时才创建这个用于编辑的 widget。

## 提供编辑器

在这个例子中,当 table view 需要提供一个编辑器时,它要求 delegate 提供一个可用于编辑的 widget,它应该适用于当前正被修改的数据项。这正是 createEditor()函数应该实现的:

```
QWidget *SpinBoxDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem & /* option */,
    const QModelIndex & /* index */) const
{
    QSpinBox *editor = new QSpinBox(parent);
    editor->setMinimum(0);
    editor->setMaximum(100);
    return editor;
}
```

我们不需要跟踪这个 widget 的指针,因为 view 会在不需要时销毁这个 widget。我们也给编辑安装了 delegate 缺省的事件过滤器,这提供了用户期望的标准编辑快捷键。view 通过我们定义相应的函数来保证编辑器的数据与几何布局被正确的设置。我们也可以根据不同的 model index 来创建不同的编辑器,比如,我们有一列整数,一列字符串,我们可以根据哪种列被编辑来创建一个 QSpinBox 或是 QLineEdit。delegate 必需提供一个函数把 model 中的数据拷贝到编辑器中。

```
void SpinBoxDelegate::setEditorData(QWidget *editor,
    const QModelIndex &index) const
{
    int value = index.model()->data(index, Qt::DisplayRole).toInt();
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->setValue(value);
}
```



## 向 model 提交数据

这需要我们实现另外一个函数 `setModelData()`:

```
void SpinBoxDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
const QModelIndex &index) const
{
    QSpinBox *spinBox = static_cast<QSpinBox*>(editor);
    spinBox->interpretText();
    int value = spinBox->value();
    model->setData(index, value);
}
```

标准的 `QItemDelegate` 类当它完成编辑时会发射 `closeEditor()` 信号来通知 view。view 保证编辑器 widget 关闭与销毁。本例中我们只提供简单的编辑功能，因此不需要发送个信号。

## 更新编辑器几何布局

delegate 负责管理编辑器的几何布局。这些几何布局信息在编辑创建时或 view 的尺寸位置发生改变时，都应当被提供。幸运的是，view 通过一个 view option 可以提供这些必要的信息。

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

## 编辑提示

编辑完成后，delegate 会给别的组件提供有关于编辑处理结果的提示，也提供用于后续编辑操作的一些提示。

这可以通过发射带有某种 hint 的 `closeEditor()` 信号完成。这些信号会被安装在 spin box 上的缺省的 `QItemDelegate` 事件过滤器捕获。对这个缺省的事件过滤来讲，当用户按下回车键，delegate 会对 model 中的数据进行提交，并关闭 spin box。

我们可以安装自己的事件过滤器以迎合我们的需要，例如，我们可以发射带有 `EditNextItem` hint 的 `closeEditor()` 信号来实现自动开始编辑 view 中的下一项。