Дамаскинский К., 3630102/70201

Оглавление

Общие сведения

Библиотека Qt позволяет создать кроссплатформенное приложение с графическим пользовательским интерфейсом. Она реализует систему сигналов и слотов, которая успешно оборачивает как систему сообщений Wintel, так и систему сообщений XWindow.

Для реализации системы интроспекции разработчики Qt реализовали общего для всех Q-классов предка – класс QObject.У каждого типа, наследующегося от QObject, существует статический класс QMetaObject, который хранит метаданные этого **типа**.

Не допускается множественное и виртуальное наследование Q-классов – это запрещается разработчиками библиотеки.

Для обеспечения бинарной совместимости уже собранных приложений с разными версиями библиотеки, в обладающими **одним и тем же интерфейсом, но разными его реализациями**, разработчики Qt используют идиому PIMPL – private implementation. Суть в следующем: все Q-классы представляют собой лишь только интерфейс, а их реализация лежит в классах с суффиксом Private. Методы public-класса представляют собой вызовы методов private-класса. Публичная и приватная ветки имеют симметричную иерархию. При создании публичного класса инициализируется D-pointer – указатель на симметричный приватный класс. В приватном классе также инициализируется Q-pointer – указатель на соответствующий ему публичный класс. Доступ к этим указателям осуществляется с помощью макросов Q_D и Q_Q.

Инициализация GUI–приложения

Первым делом инициализируются ресурсы

Q_INIT_RESOURCE(name) - call qInitResources_<name>()

Далее создаётся экземпляр QApplication – класс, обслуживающий GUI-приложение, поддерживающий жизненный цикл системных сообщений

QApplication инициализирует QCoreApplication, который является синглтоном: в QCoreApplication хранится поле self. Если при вызове init() оно не равно nullptr, то выбрасывается сообщение о фатальной ошибке и программа аварийно завершается.

Далее происходит инициализация диспетчера событий – Qt реализовал его для платформ wintel, unix, symbian. При этом все диспетчеры событий наследуются от абстрактного диспетчера – QAbstractEventDispatcher.

Класс QAbstractEventDispatcher осуществляет регистрацию сообщений, позволяет навешивать обработчики на сообщения конкретного типа, устанавливать на каждый тип сообщений таймер прослушивания.

Запуск цикла сообщений

При вызове метода exec у QApplication вызывается одноимённый метод у QCoreApplication. Перед тем, как попытаться запустить цикл сообщений, производится ряд проверок:

1) Проинициализирован ли QApplication
2) Происходит ли запуск цикла сообщений из главного потока
3) Не запущен ли цикл уже на данном этапе

В случае провала какой-либо из проверок происходит аварийное завершение программы с ненулевым кодом возврата.

Далее создаётся экземпляр класса QEventLoop, у которого вызывается метод exec – в нём вызывается processEvents, который собственно обрабатывает сообщения.

Создание класса TextEdit

После инициализации менеджера графического приложения QApplication, создаётся, собственно, класс текстового редактора TextEdit. Он является наследником класса главного окна – QMainWindow.

Инициализация QMainWindow

Итак, при конструировании TextEdit вызывается конструктор QMainWindow. QMainWindow в свою очередь наследуется от QWidget – базового класса пользовательского интерфейса, который умеет рендерить в клиентскую область и получать сообщению от устройств ввода. В конструкторе виджета производится настройка параметров отображения -- фокусировка на окне, размер рабочей области, родительский виджет и т. п. Самым главным событием является посылка сообщения Create. Далее в теле конструктора QMainWindow прописывается layout – размещение объектов интерфейса (меню, статус бар, тулбар, центральный виджет, вспомогательные виджеты) внутри данного виджета. Ясно, что топология структуры виджетов может быть произвольной - каждый виджет может включать в себя сколь угодно много других. Однако QMainWindow предварительно инициализирует меню, статус-, тул-бар, центральный виджет, которые можно получить для дальнейшего заполнений с помощью соответствующих методов.

Инициализация элементов пользовательского интерфейса

После вызова конструктора QMainWindow следует серия инициализаций пунктов панели меню – за каждый пункт меню отвечает экземпляр класса QMenu – и элементов тулбара, который во многом повторяет пункты меню -- за каждый элемент отвечает экземпляр класса QToolBar. Для каждой кнопки инициализируются QAction – кликабельные элементы пользовательского интерфейса, которые можно подключить к конкретным пользовательским методам с помощью сигнала triggered(), который излучается отслеживающим клики мыши экземпляром класса QMenu, в котором и сложены все экземпляры QAction, привязанные к данному меню. К каждой кнопке меню привязывается метод класса TextEdit, исполняющий соответствующее действие.

На каждый элемент панели меню написан свой метод:

– на пункт File и сопряжённые с ним элементы тулбара – setupFileActions;
– на пункт Edit и сопряжённые с ним элементы тулбара – setupEditActions;
– на пункт Format и сопряжённые с ним элементы тулбара – setupTextActions.

Класс QTextEdit

Затем создаётся экземпляр QTextEdit – это класс, который представляет собой виджет, позволяющий отображать plain-text и rich-text (с HTML-форматированием) и происходит связывание сигналов QTextEdit, излучаемых при изменении формата текста и позиции курсора, с соответствующими слотами.

QTextEdit является наследником QAbstractScrollArea – низкоуровневая абстракция прокручиваемой области. QAbstractScrollArea в свою очередь наследуется от QFrame – виджета с рамкой заданного стиля

Класс QTextEditControl

При инициализации QTextEdit происходит инициализация класса QTextEditControl, который позволяет производить операции вставки и создания plain- и rich-текста (класс QMimeData), соединение сигналов, отвечающих за фокусировку, изменения размеров текст и т. д., с соответствующими слотами.

В конструкторе QTextControl, наследником которого является QTextEditControl, инициализируется экземпляр класса QTextCursor.

Класс QTextCursor

Этот класс предоставляет информацию о позиции курсора и анкера (места, где начинается область выделения текста).

Далее создаётся класс QTextDocument.

Класс QTextDocument.

В конструкторе вызывается метод insertBlock, которые создаёт **корневой блок**

Здесь мы впервые сталкиваемся с понятием **блока**. Блок – это относительно крупная логическая единица, на которые можно разбить текст. Обычно в качестве блока выступает абзац.

Здесь же мы встречаемся с понятием **фрагмента**. Фрагмент – это максимальная область текста с единым форматированием. Весь текст состоит из фрагментов.

В классе QTextDocument хранится дерево блоков и фрагментов. Эти деревья являются красно-чёрными. (QFragmentMap<QTextFragmentData> – FragmentMap.

Ключом в обоих деревьях является позиция блока и фрагмента соответственно.

Итак, QTextEditControl проинициализирован.

Метод load

После инициализации QTextEditControl вызывается метод load, который осуществляет загрузку текста.

Если файл с передаваемым именем (initialFile == ":/example.html" – является ресурсом приложения, компилируется компилятором ресурсов в двоичном виде непосредственно в исполняемый код программы) не существует в файловой системе, то загрузка считается проваленной, возвращается false.

Иначе – всё содержимое файла считывается в байтовый массив.

Далее мы пытаемся с помощью QTextCodec::codecForHtml распознать кодировку для представленного в файле текста. Кодировка распознаётся с помощью специальной сигнатуры в начале файла.

Если выясняется, что файл не является html-деревом, то возвращается кодек для кодировки Latin-1.

Далее с помощью mightBeRichText проверяется, является ли файл подозрительным на то, чтобы содержать rich-текст. Проверяется по литералу <?xml, <!doc, либо выражению, находящемуся внутри угловых скобок. Если выражение удовлетворяет одному из стандартных тегов (QTextHtmlParser::lookupElement), значит, это rich-текст.

Далее полученный текст переводится в Unicode.

Построение DOM-дерева. Класс QHtmlParser

Парсер

С помощью метода setHtml, который вызывается у экземпляра QTextEditControl, происходит построение DOM-дерева.

В процесса работы данного метода создаётся экземпляр класса QTextHtmlImporter, отнаследованный от QTextHtmlParser. В конструкторе импортёра вызывается HTML-парсер (метод parse), который строит DOM-дерево – дерево XML-узлов (QTextHtmlParserNode). Каждый узел имеет вид:

<имя_тега атрибут1=»значение1» атрибут2=»значение2» атрибут3=»значение3»>текст</имя_тега> -- *парный тег*

<имя_тега атрибут1=»значение1» атрибут2=»значение2» атрибут3=»значение3» /> -- непарный тег

При этом логика хранения следующая: в классе QTextHtmlParser хранится QVector<QTextHtmlParserNode> – список абсолютно всех узлов. А у каждого узла хранятся списки **номеров** детей, соответствующих общему списку, и **номер** родителя.

Пример DOM-дерева:

<html>

```
<body>
        <p>Text1</p>
        <div>
                <b>T</b>ext2
                <div>Te<i>xt</i>3</div>
                <p>Text4</p>
        </div>
</body>
</html>
```

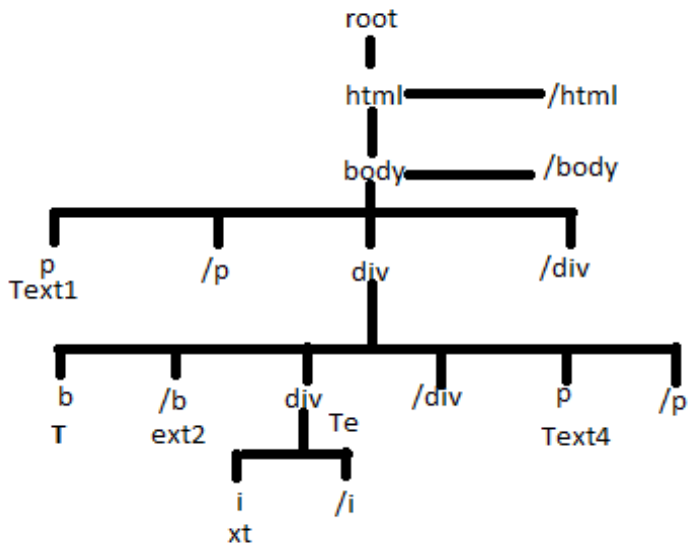

Рисунок 1. DOM-дерево

Импорт в дерево блоков и фрагментов

После того, как парсинг завершился, начинается импорт в дерево блоков.

Программа входит в цикл по всем узлам DOM-дерева (в порядке появления в «общей свалке» - по сути этот порядок соответствует обходу дерева в глубину, начиная с самого левого ребёнка, далее направо до конца списка детей и затем самого себя).

Процесс происходит следующим образом:

Из текста убираются все теги – оставляем голый plain-текст.

Если мы находимся на закрывающем теге каких-то специальным образом отображаемых структур наподобие таблиц, то надо завершить их формирование и обработать специальным образом.

Если по результатам обработки должен появиться новый блок, мы его добавляем.

Добавление происходит посредством проброса большого числа вызовов сквозь множество классов, которое в итоге приводит к вызову insert_block.

Первым делом происходит разделение дерева фрагментов по фрагменту, внутрь которого будет вставляться новый блок (метод QTextDocumentPrivate::split). Далее, в случае создания нового блока, в соответствующее место текста вставляется **сепаратор** и прописываются параметры нового фрагмента: смещение и формат, в случае необходимости создаётся новый блок. Сепаратор – это специальный символ, который вставляется на границах блоков.

Сами деревья блоков и фрагментов организованы следующим образом: ключом является смещение в строке, которая содержит только текст вместе с сепараторами. В каждом узле дерева хранится смещение до текущего фрагмента (size_left_array[0]) и размер текущего фрагмента (size_array[0]). Исходя из метода QFragmentMapData::position, становится понятно, что size_left_array[0] и size_array[0] содержат соответствующие смещения и размеры с учётом сепаратора:

```
inline uint position(uint node, uint field = 0) const {
        …
        offset += f->size_left_array[field] + f->size_array[field];
        …
        return offset;
}
```

При создании первого фрагмента в блоке сепаратор добавляется отдельным фейковым элементом:

```
int QTextDocumentPrivate::insert_block(int pos, uint strPos, int format, int blockFormat,
QTextUndoCommand::Operation op, int command) {
        …

        uint x = fragments.insert_single(pos, 1);
        /* 1 – это размер фрагменита – см. шапку uint insert_single(int key, uint length) */

        …

}
```

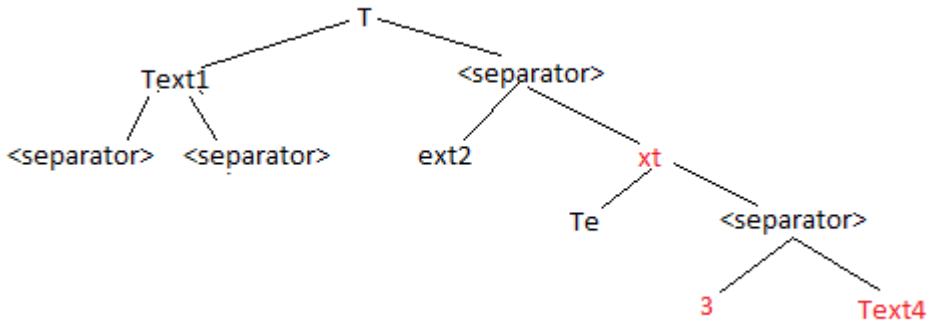

Рисунок 2. Красно-чёрное дерево фрагментов

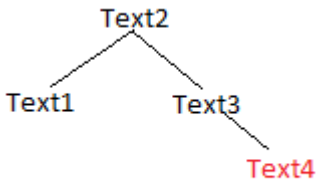

Рисунок 3. Красно-чёрное дерево блоков

Добавление текста

Основные действия происходят в QTextDocumentPrivate::insertText. Если уже есть выделенный текст (позиция анкера/якоря не совпадает с позицией курсора), то текст удаляется. В случае если форматы соседних фрагментов, получившихся при вставке, совпадают, необходимо объединить фрагменты.

Удаление текста

Основные действия происходят в QTextDocumentPrivate::move.

Если удаляемая часть текста задана таким образом, что концы её находятся не на границе фрагментов, то нужно добавить новый фрагмент. Далее идёт проверка на то, происходит удаление куска целиком внутри блока либо на стыке. Если проверка происходит на стыке, то нужно создать новый блок, иначе такой необходимости нет.

Изменение формата

QTextDocumentPrivate::setCharFormat

Создаём новый фрагмент по выделенному куску. Мёржим со старым форматированием (добавляем к нему).