электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.512

БИБЛИОТЕКА КЛАССОВ ДЛЯ АНАЛИЗА ПАРАМЕТРОВ КОМАНДНОЙ СТРОКИ

***Сидякин А.А.**, студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Виноградова М.В., к.т.н., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
bauman@bmstu.ru

Как известно, программа – это последовательность инструкций, предназначенная для исполнения устройством управления вычислительной машины, процессором компьютера. Из этого определения следует, что программа способна выполнять только один сценарий действий, заложенный в неё при разработке. Однако, это не так – именно для возможности влияния пользователя на ход выполнения программы предусматривается интерфейс пользователя для взаимодействия с ней.

Программу, имеющую графический интерфейс пользователя (GUI), будем называть приложением с графическим интерфейсом, а программу без графического интерфейса с текстовым интерфейсом (CLI) – утилитой.

Графический интерфейс пользователя приложений, безусловно, более удобен для пользователя. Однако, если взаимодействовать с программой будет не пользователь, а другая программа, то в этом случае в графическом интерфейсе нет никакой надобности. Кроме того, он более требователен к ресурсам системы, чем текстовый интерфейс.

В связи с этим, текстовый интерфейс более предпочтителен при использовании маломощного оборудования, а в некоторых специфических сферах и вовсе является единственно возможным, особенно если в системе нет дисплея для отображения графики, нет возможности его размещения. Примером могут служить микросхемы автономного оборудования, работающего в автоматическом режиме, орбитальные спутники, электроника управления полётом ракеты и т.д.

http://sntbul.bmstu.ru/doc/640934.html

Взаимодействие с утилитой через её текстовый интерфейс осуществляется, в основном, посредством командной строки в среде операционной системы (ОС), будь то ОС семейства Microsoft Windows или Unix-подобная ОС – принцип везде одинаковый. Всё, что записано в командной строке при запуске утилиты является аргументами командной строки. Самым первым аргументом всегда является имя самой утилиты. После него следуют остальные аргументы, разделённые пробелами.

В C/C++-подобных языках обращение к аргументам командной строки, полученным в качестве параметров запуска, происходит через стандартные переменные argc, счётчик количества аргументов и *argv[], массив указателей на аргументы.

По стандарту POSIX аргументы командной строки могут являться опциями или их параметрами. Опции могут быть короткими, состоящими из одного символа, и длинными, более одного символа. Перед короткими опциями ставится символ дефиса "-", а перед длинными – два дефиса "--", что обеспечивает однозначную трактовку этих аргументов как опций. Опции могут иметь параметры. Если опция не имеет параметров, то она называется "флаговой".

Пример короткой опции с параметром и без параметров:

program.exe -c 25 -n

Здесь h является опцией, а 25 – её параметром; n – флаговая опция без параметров.

Пример длинной опции:

program.exe –count=25

Здесь count является опцией, а 25 – её параметром.

Данный стандарт не является обязательным. К примеру, параметры запуска утилит из состава ОС Microsoft Windows имеют несколько другой формат. Однако, во многих источниках рекомендуется при разработке ПО придерживаться именно приведённого выше стандарта POSIX.

Итак, при разработке утилиты программист должен описать механизм реакции утилиты на аргументы командной строки, переданные ей в качестве параметров запуска. Когда утилита работает с одной-двумя опциями, которые являются флаговыми, то есть не имеющими параметров, это не является такой сложной задачей. Можно просто поставить набор логических условий по числу опций и искать их в массиве argv.

Однако, когда опций достаточно много и они могут иметь по несколько параметров каждая, разбор аргументов командной строки превращается в серьёзную проблему.

Конечно, здесь тоже можно написать систему логических условий, срабатывающих в результате вычислений невероятной сложности и глубины вложенности логических

выражений, разобраться в которой уже спустя месяц не сможет даже сам разработчик. Однако, помимо очевидных недостатков такого подхода, есть ещё один – что делать, если решать проблему разбора аргументов командной строки нужно не только в этот единственный раз в жизни, а и далее, в последующих проектах?

Становится понятно, что в этом случае нельзя ограничиться одноразовым решением, ведь тогда потери временных и трудовых ресурсов повторятся снова, когда проблема возникнет вновь. Намного более рационально будет, однажды написав универсальную систему разбора, использовать её во всех последующих проектах.

Разумеется, проблема возникла не вчера, и уже существует ряд решений.

Во-первых, это функция getopt() [1][2], входящая в состав стандартной библиотеки языка C unistd.h. Функция принимает на вход переменные argc и argv, а также строку шаблона, по которой будет осуществляться разбор. Среди достоинств стоит отметить то, что она является стандартной и доступна для использования уже "из коробки". Но функция предназначена лишь для разбора аргументов командной строки с отделением опций от их параметров – результаты её работы никак не организуются, эта задача остаётся за разработчиком.

Далее следует упомянуть библиотеку Boost::Program_options [3]. Это очень мощное решение по разбору аргументов командной строки, предоставляющее широкие возможности по их обработке. Однако, подключение этой библиотеки к проекту и использование её функций влияет на размер утилиты – не вся библиотека может быть подключена лишь из заголовочных файлов, требуется линковка. А так как весь проект не стоит на месте и развивается, то динамическая линковка не подходит, приходится использовать статическую. А как было сказано выше, в некоторых сферах применения размер утилиты может иметь решающее значение.

Кроме этого, заголовочные файлы библиотеки в некоторых дистрибутивах разбиты на модули и не установлены по умолчанию, а значит, кроме загрузки самой библиотеки Boost нужно будет ещё дополнительно найти недостающие модули и их добавить в свой проект. Вместе с этим, со сложностью библиотеки ухудшается и её переносимость – сложно предсказать, как отразится на функционале перенос проекта в среду другой операционной системы.

Другие существующие решения в той или иной степени повторяют функционал библиотеки Boost, однако являются менее известными и, с некоторой долей вероятности, менее проверенными и надёжными.

Таким образом, поставлена задача разработать собственное программное средство, которое будет предоставлять больше возможностей, чем стандартная функция getopt() и, в то же время, будет менее громоздким, чем библиотека Boost.

Диаграмма классов показана на рис.1.

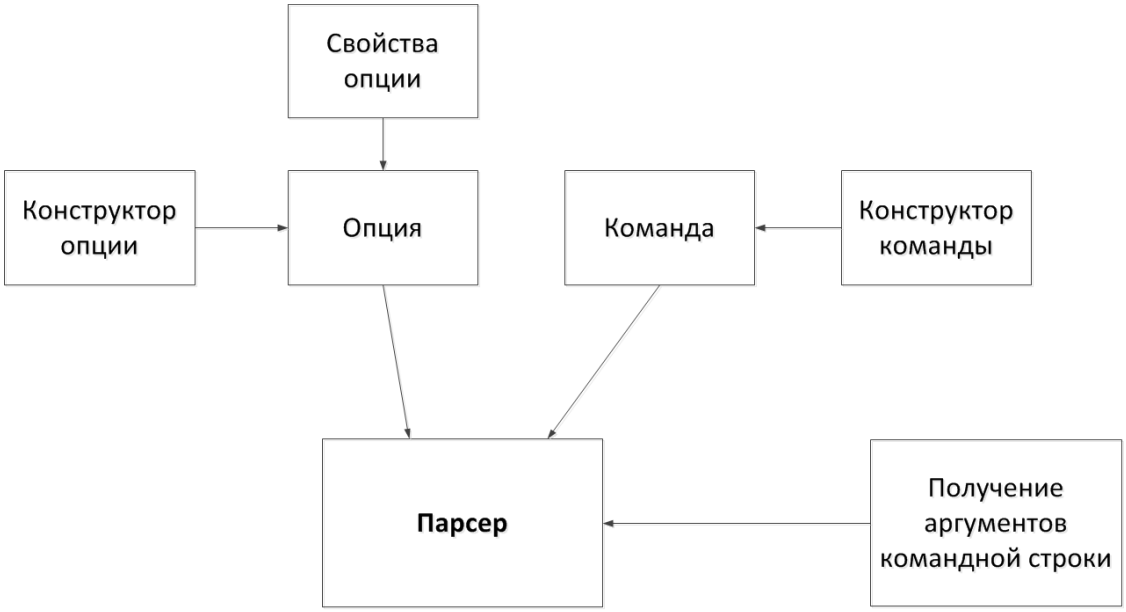

Рис.1. Диаграмма классов

Так как основным назначением программы является выполнение той или иной задачи (команды), то был создан класс команды, метод run() которого осуществляет требуемый функционал. Этот метод определяется программистом. Функционал метода зависит от полученных опций и их параметров, которые содержатся в соответствующих полях этого класса. Каждая команда имеет своё уникальное имя, то есть на ход работы утилиты может влиять несколько команд, и каждая по-своему, в зависимости от функционала метода run() – как раз то, чего мы и хотим добиться.

У команды есть конструктор, который выполняет единственную функцию – создаёт новый объект команды. Конструктор команды входит в состав класса свойств команды, который вызывает его при для создание объекта команды, если её имя присутствует в командной строке.

Таким образом, нам необходимы опции. Опции представлены отдельным классом, поля которого содержат имя опции, под которым она представлена в командной строке, состоящее из одного символа, и так называемое "мнемоническое имя" – осмысленное название опции. Такое "раздвоение" имени опции было введено для того, чтобы в случае необходимости изменить имена опций в командной строке, например переход от коротких

опций к длинным, это не отразилось на функционировании системы. Также имеется поле признака, является ли данная опция флаговой.

Для построения каждой опции требуется конструктор опции, который соотносит обычное и "мнемоническое" имя опции. Конструктор добавляет опции в команду.

Также создан класс свойств опции. Эта структура содержит следующую информацию о свойствах опции: является ли она обязательной, какие опции совместимы с ней, а какие – нет и какие опции должны присутствовать в командной строке помимо данной.

Все вышеперечисленные классы относятся к описанию команды и её опций. Однако, до сих пор ни слова не было сказано про, собственно, получение опций и их параметров из командной строки. Этим занимается отдельный класс, который содержит в своём составе ряд методов. В основе разбора аргументов командной строки лежит стандартная функция getopt(), о которой шла речь выше. Вокруг функции построен разбор результатов её выполнения, организующий их в ассоциативный массив. Ключами в этом массиве являются имена опций, а значениями – их параметры. Причём, массив позволяет назначать на один и тот же ключ – опцию несколько значений - её параметров. Также есть метод для получения имени команды, которую требуется выполнить.

Все классы объединены в главном – классе обработчика командной строки – парсера. Он содержит список команд, которые могут быть переданы утилите в командной строке. За каждой командой закреплён список опций, которые она может иметь, а за каждой опцией, в свою очередь, закреплены её свойства. Первым делом, парсер определяет, какая из имеющихся у него команд передана в командной строке. После этого он извлекает из списка нужную и, просмотрев список возможных опций, формирует строку шаблона, по которой будут обработаны все аргументы командной строки, полученные утилитой при запуске.

В результате этого разбора – парсинга – будет создан объект команды, которому будут переданы обнаруженные опции и их параметры. После этого можно будет вызывать основной метод этой команды, который выполнит свой функционал исходя их полученных опций.

Для оповещения об ошибках разбора, как-то: отсутствующие необходимые опции, отсутствие параметров у опций, неверные команды, неизвестные опции и другие, добавлен отдельный класс ошибки. Большинство методов во всех класса системы возвращают булевское значение, определяющее успех выполнения метода. В случае неудачи выполнения метода вызывается объект ошибки и в него добавляется информация о произошедшем сбое в работе. Каждое новое сообщение добавляется в хвост списка

подобных сообщений, поэтому можно проследить всю цепочку ошибки от места её возникновения.

Для использования описываемой системы в своём проекте необходимо включить в него основной заголовочный файл, в который входят все остальные. Затем следует произвести предварительную настройку в проекте: унаследовать собственные классы команд и задать для каждой набор опций, которые она будет использовать. Также для каждой команды необходимо переопределить метод run(), который отвечает за функционал команды.

Сначала создаются объекты конструкторов опций:

```
OptBuilder opt11("h"), opt12("s");
```

Затем создаются объекты самих опций:

```
Opt0 option11("help-mode", &opt11);
OptN option12("source", &opt12);
```

После этого создаются свойства опций для команды:

```
OptProperty optprops11(&option11, true), optprops12(&option12, true),
```

Созданные опции объединяются в список:

```
QList<OptProperty> opts1;
opts1 << optprops11 << optprops12;
```

После чего можно создавать конструктор команды и саму команду:

```
CmdBuilderSpec1 cmdbld1;
CmdOpt comanda1("copy", &cmdbld1);
```

Последним этапом является создание парсера:

```
Parser parser;
parser.addCmdAndOpts(&comanda1, opts1);
```

Теперь парсер готов к использованию. При запуске будет проверяться набор аргументов командной строки, и если будет обнаружена команда, то она будет выполнена в соответствии с переданными опциями и их параметрами.

Благодаря работе парсера, утилита будет реагировать только на те команды и опции в командной строке, которые были заданы на этапе предварительной настройки. Все остальные будут восприниматься как неверные команды и параметры и будут игнорироваться функциями разбора.

В результате разработана универсальная система классов, которую можно использовать в проектах, где требуется обрабатывать аргументы командной строки, передаваемые приложению в качестве параметров запуска. От существующих решений

данная система отличается простотой использования и меньшим размером, что, в свою очередь, не приведёт к увеличению объёма исполняемого файла или необходимости распространения вместе с приложением дополнительных файлов библиотек. Также стоит отметить, что классы написаны с использованием кроссплатформенного инструментария для разработки ПО – системы классов и библиотек Qt [4], что позволяет с одинаковым успехом использовать её как в среде операционных систем семейства Microsoft Windows, так и в Unux-подобных, включая Mac OS X.

Список литературы

1. Крис Херборт – Обработка команд при помощи getopt(). URL: http://www.ibm.com/developerworks/ru/library/au-unix-getopt/ (дата публикации 19.06.2009, дата обращения 12.10.2012).
2. Средства по работе с параметрами запуска программ в The GNU Project // The GNU Project – Parsing program options using getopt URL: http://www.gnu.org/software/libc/manual/html_node/Getopt.html (дата обращения 12.10.2012).
3. Документация Boost // Boost C++ Libraries – Program_options. URL: http://www.boost.org/doc/libs/1_49_0/doc/html/program_options.html (дата обращения 07.09.2012).
4. Документация Qt // Qt Documentation. URL:http://doc.qt.nokia.com/4.6/index.html (дата обращения 17.08.2012).