Описание архитектуры интерпретатора командной строки

Интерпретатор

Интерпретатор, получая от пользователя ввод построчно, запускает его, выводя результат и поддерживая внутреннее состояние. Интерпретатор за время жизни получает множество отдельных самодостаточных команд от пользователя – *пользовательских запросов.* Предполагается, что когда пользователь нажал кнопку Enter, он написал полный текст своего запроса – интерпретатор может попытаться запустить этот запрос, показав либо ошибку, либо результат, и зависнуть в ожидании следующего пользовательского запроса.

Состояние

Состояние интерпретатора (`InterpreterState`) объединяет в себе окружение, в котором интерпретируются команды. Оно передается по ссылке узлам AST и командам при их интерпретации (см. *Интерпретация узлов AST*, *Реализация встроенных команд* и *Запуск внешних программ*), что делает возможным запуск каждой команды в разных окружениях.

Состояние инициализируется при запуске стартового метода приложения (с помощью `getPwd()` вычисляется `pwd`, копируются значения переменных окружения и т.д). В нем хранится набор переменных окружения, текущая директория, потоки ввода-вывода.

Класс `InterpreterState` имеет набор методов, позволяющих читать (`getVar`, `getStdin`, …), записывать (`setVar`, `setStdin`, …), а также копировать все состояние, чтобы запускать команды в разных окружениях. При копировании состояние методом `clone()` делается глубокая копия всего состояния, кроме полей `stdin`, `stdout`, `stderr`, которые копируются по ссылке.

Разбор и структура AST

Утверждается, что для того, чтобы корректно разобрать входную строку пользовательского запроса, необходимо иметь эту строку и состояние интерпретатора (в том числе переменные окружения). Пользовательский запрос разбирается целиком (см. предположения в *Интерпретатор*), если разобрать не удалось, то пользователю выводится сообщение об ошибке.

За разбор пользовательского ввода отвечает класс `Parser`, лексический анализ в данной модели для простоты не рассматривается, хотя его можно добавить как часть парсинга. `Parser` посимвольно читает текст пользовательского запроса целиком слева направо, заменяя вхождения переменных окружения на их значения,

обрабатывая одинарные и двойные кавычки. В результате возвращается AST дерево или бросается исключение, если пользовательский запрос некорректен. Более подробно алгоритм разбора описан в *Более подробно об алгоритме парсинга.*

AST выражено тремя узлами: **`Pipe`**, **`Assign`**, **`Cmd`**. Pipe соответствует оператору “ | ” и хранит список соединенных узлов, **`Assign`** – операция присваивания значения в переменную окружения, хранит название переменной и строковое значение. **`Cmd`** – вызов команды по имени с списком аргументов.

Таким образом строка **`a=5 | cat f.txt | exit`** будет представлена узлом **`Pipe`**, имеющем 3 потомка: **`Assign(name=a, value=5)`**, **`Cmd(path=cmd, args=[“f.txt”])`** и **`Cmd(path=exit, args=[])`**

Интерпретация

Интерпретация узлов AST

За интерпретацию каждого узла отвечает метод **`eval(state: InterpreterState)`**, реализованный в классе этого узла. Далее идет речь о реализации этих методов.

Assign

Изменяет полученное **`InterpreterState`** при помощи метода **`setVar(name: String, value: String)`**.

Pipe

Создает список **`subStates`** из **`InterpreterState`** длины **`n`**, где **`n`** – количество вложенных узлов **`parts`**. При этом клонируются все переменные окружения, а для IO потоков в списке выполняются следующие условия:

- **`subStates[0].stdin`** = **`state.stdin`**
- **`subStates[n - 1].stdout`** = **`state.stdout`**
- **`subStates[i].stdin`** = **`subStates[i - 1].stdout`**
- **`subStates[i].stderr`** = **`state.commonStderr`**

После инициализации **`Pipe`** запускает интерпретацию каждого вложенного узла в отдельном потоке, передавая *i*-ому узлу **`subStates[i]`**. В каждом потоке обрабатывается **`ExitException`**. При этом если исключение кинул любой поток кроме последнего, то исключение игнорируется, а в случае последнего прокидывается выше. Текущий поток ждет завершения интерпретации последнего узла и возвращает его код возврата.

Cmd

С помощью метода фабрики **`CommandFactory.getCommand(state: InterpreterState, path: String)`**, получает реализацию класса **`Command`**,

который содержит логику данной команды и вызывает у нее метод `exec(args: String[], state: InterpreterState)`, передавая список аргументов и состояние. Если метод фабрики вернул `null`, то выводится ошибка о невозможности найти команду.
(См. далее *Фабрика CommandFactory* и *Реализация встроенных команд*)

Фабрика-одиночка `CommandFactory`

Одиночка. Имеет предопределенную таблицу встроенных команд и один публичный метод, позволяющий получить правильную реализацию интерфейса `Command`, содержащую бизнес-логику требуемой команды.

Фабрика опирается на строку path при выборе реализации. Если path – это имя одной из встроенных команд, записанных в таблице builtins, то фабрика возвращает готовый объект Command (т.е. для `path` равного `"echo"` возвращается объект, класса `EchoCommand`).

В противном случае, если `path` не указывает ни на одну встроенную команду, выполняется поиск бинарного файла, опираясь на полученное состояние интерпретатора `InterpreterState` (на `pwd`, переменную `PATH` и т.д.). Перебираются все пути `pwd + path` и `PATH[i] + path` в порядке возрастания *i* до первого существующего файла. Если такой файл найден, будет создан экземпляр `BinaryCommand` со ссылкой на этот файл типа `File`, иначе будет брошено исключение и выведена ошибка пользователю.

Реализация встроенных команд

Все встроенные команды представлены классами, реализующими интерфейс `Command`. Бизнес-логика каждой команды находится в методе `exec(args: String[*], state: InterpreterState)`, соответствующего ей класса. Аргумент `args` – это набор строковых аргументов команды, подобно *argv.* Через состояние интерпретатора у команды есть доступ к состоянию переменных окружения, к потокам ввода вывода, к `pwd` и т.д. Метод `exec()` возвращает число – код возврата команды.

Далее более кратко опишем реализации метода `exec(args: String[], state: InterpreterState)` некоторых встроенный команд, остальные реализуются тривиально.

`cat [FILE]`

- Если полученный список аргументов `args` имеет длину, отличную от 0 и 1, то в поток `state.stderr` пишется сообщение об ошибке и команда завершается с кодом возврата 1;
- Если полученный список аргументов пуст, то данные читаются из `state.stdin` и пишутся в `state.stdout`

- Если файл переданный в качестве аргумента не найден, то, аналогично, пишется сообщение об ошибке и возвращает ненулевой код возврата;
- Если файл найден, то его содержимое начинает записываться в **`stdout`**, когда конец файла будет достигнут, команда завершается с нулевым кодом возврата

`echo [ARG …]`

- В **`state.stdout`** выводится строка, состоящая из полученных аргументов, разделенных пробелами.

`wc [FILE]`

- В зависимости от полученных аргументов команда либо анализирует входной поток, либо набор полученных файлов и выводит в stdout результат;
- Ошибки обрабатываются аналогично команде cat

`pwd`

- Получает путь до папки из **`state`** и выводит его в **`state.stdout`**

`exit`

- Бросает исключение **`ExitException`** которое отлавливается **`ReplApplication`** и останавливает REPL цикл

Запуск внешних программ

У экземпляра класса **`BinaryCommand`** есть ссылка на бинарный файл **`executable`**. При запуске команды (вызове метода **`exec(String[*], InterpreterState)`**) системным вызовом запускается бинарный файл, ему передается полученный список аргументов, переменные окружения из **`InterpreterState`**, текущая директория, устанавливаются полученные потоки ввода и вывода. После завершения процесса метод **`exec`** возвращает его код возврата.

Примечания

(стоит читать, только если есть вопросы)

Более подробно об алгоритме парсинга

Состояние парсера хранит не разобранную часть пользовательского запроса и флаг, показывающий разрешены ли подстановки переменных окружения и экранирование. Этот флаг требуется, чтобы отличить два контекста: в одинарных кавычках запрещены экранирование и подстановки переменных, в остальных местах они разрешены. Хранение буфера пользовательского ввода позволяет выполнять препроцессинг – заменять части этого буфера, так обрабатываются кавычки и подстановки переменных окружения.

Парсер читает текст слева направо. Начальное состояние содержит весь пользовательский запрос и разрешает подстановки и экранирование.

Когда встречается не_экранированный знак доллара и подстановки разрешены, парсер ищет конец вхождения переменной окружения, подставляет ее значение в буфер не разобранного пользовательского запроса.

Когда встречается не экранированная двойная кавычка, парсер начинает поглощать строку до вхождения второй двойной не_экранированной кавычки, разрешая подстановки и экранирования. После чего результат подставляется в буфер вместо строкового литерала.

Когда встречается не экранированная одинарная кавычка, поглощается вся строка до второго вхождения одинарной кавычки, как в случае с двойными кавычками, только запрещаются подстановки и экранирования. Результат подставляется в буфер.