xCore-Commander

Руководство по использованию

Vk id: apploidxxx

Kupriyanov Arthur

apploidyakutsk@gmail.com
Java-xCore-Commander

Оглавление

Цели проекта

Основной целью данного проекта является создание микросервиса[1], занимающейся рутинными задачами и предоставляющий легкодоступный интерфейс для пользователя.

Ядро

Java xCore

Центральную часть всей работы составляет модуль, работающий через взаимодействие с помощью команд.

Это позволяет создавать различные прикладные программы для работы с этим модулем, дальше именуемым как **Java xCore**.

Различные решения для различных задач с общими данными.

Преимущества такой архитектуры в том, что мы можем разместить Java xCore в удаленное место (например, в облаке) и создавать программы, отправляющие запрос.

Как итог, мы получаем слабосвязанные между собой модули. Иными словами, Java xCore не знает с кем он работает, ему это и не нужно. Все потому, что это ядро должно предоставлять общий функционал для решения конкретных задач.

Конкретный пример

Чтобы лучше понять принцип, рассмотрим пример Бота, созданного на основе такой архитуры.

Опустим реализацию бота и представим, что он уже работает и обрабатывает сообщения поступающие из социальной сети в Вконтакте. То есть, каждый раз, когда кто-то пишет боту в социальной сети, мы сможем его обработать в программе.

Теперь, необходимо обработать ввод пользователя.

На этом этапе все зависит от приложения. Например, один из них может иметь NAL[2], обрабатывающий запрос и интерпретирующий его в команду, понятную для Java xCore.

Далее, мы отправляем запрос на Java xCore, предоставляющий API для таких запросов и получаем ответ, который мы также можем обработать в нашем приложении, замет отобразив его пользователю.

---

[1] Микросервисы - это путь разбиения большого приложения на слабо связанные модули, которые коммуницируют друг с другом посредством просто API.
[2] NAL - Natural Language Processing

Для разработчиков

Подробная архитектура Java xCore

Рекомендуется просмотреть статью **Создание простой архитектуры бота для внедрения в систему на Java**[3], где подробно расписан пример создания подобной архитектуры.

Далее, будет рассмотрена архитектура исходной программы Java Bot Core. Все его элементы можно разделить на несколько частей:

- Модули
- Команды
- Вспомогательные классы
- Обработчик ввода

Разберем каждый из них.

Модули

Модули – это своего рода библиотеки, к которым обращаются команды. Иными словами, они содержат всю бизнес-логику самой реализации команды.

Команды

Команды – это обрабатывающие ввод пользователя объекты. Они в свою очередь, получив ввод, должны вернуть какое-либо значение или выполнить некую операцию.

Вспомогательные классы

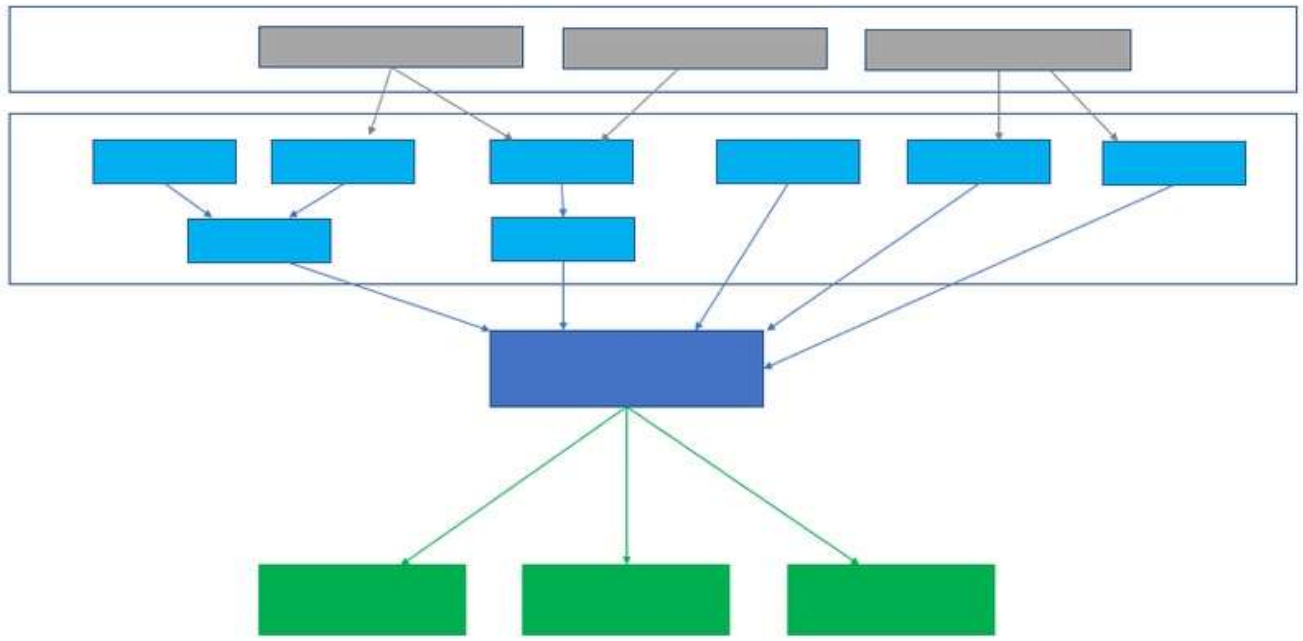
К вспомогательным классам относятся все классы, не входящие в другие, но не менее важные. Базовые классы, относящиеся к этой области – это определитель команд, совершающий выборку команды, и менеджер команд, содержащий все объекты команд. В общем случае эта область исполняет связующую роль между Обработчиком ввода и Командами.

[3] Ссылка - https://vk.com/@apploidxxx-sozdanie-prostoi-arhitektury-bota-dlya-vnedreniya-v-sistemu

Обработчик ввода

Это первое место, куда попадает запрос к Java xCore. Здесь же находятся Сервер приёма данных и Интерфейс ядра, также именуемый как Commander. Основная его задача – это прием и отправка данных, а также их обработка с помощью вспомогательных классов.

Архитектура



Карта цветов:

1. **Серый**. **Модули**. Уровень сторонних и пользовательских библиотек и модулей, предназначенных для решения различных задач. К ним будут ссылаться команды.
2. **Голубой. Команды**. Они идут как отдельные объекты, которые обращаются при необходимости к модулям, тем не менее, они должны иметь как можно меньший объем памяти, так как команд может быть очень много.
3. **Синий. Обработчик ввода и вспомогательные классы**. Общий интерфейс, позволяющий упростить обращение ко множеству команд. Паттерн Facade. Цель внедрения такого интерфейса — это сведение к минимуму зависимости подсистем друг от друга и обмена информацией между ними.
4. **Зеленый. Различные реализации**. Это уже различные системы, обращающиеся к боту, а именно к синему интерфейсу.

Реализация в Java xCore

Рассмотрим реализацию архитектуры в Java xCore

Обработчик ввода и вспомогательные классы

Здесь находятся несколько классов:

- **Commander**
- **CommandDeterminant**
- **CommandManager**

CommandManager (Менеджер команд) – отвечает за список доступных команд и содержит их объекты.
CommandDeterminant (Определитель команд) – отвечает за выборку команды
Commander – выполняет первичную обработку ввода и возвращает ответ

Команды

Команды хранятся как отдельные классы и при запуске их объекты добавляются в CommandManager. Все виды команд наследуются от суперкласса Command и реализуют его метод exec().

Модули

Модули хранятся в отдельных пакетах. Особых требований по ним – нет.

Путь запроса

1. Сначала запрос поступает в Commander.
2. Вызывается метод определителя команд
3. Производиться выборка команды с помощью CommandDeterminant
4. Исполнение команды через CommandManager
5. Возвращение результата выполнения команды

Суперкласс Command

Актуальный исходный код суперкласса доступен на GitHub[4]

Это абстрактный класс-родитель для всех исполняемых в Commander команд.

Основные элементы - это поле name и метод exec()

[4] https://github.com/AppLoidx/Java-xCore-Commander/blob/master/src/xcore/commander/commands/Command.java

Поле name

Поле name – это идентификатор и имя команды. При выполнении команды, определяется первое слово и сравнивается с этим полем каждой команды. В случае совпадения – исполняется эта команда. Следует определить его в методе setName (используя поле commandName), напрямую обращаясь к этому полю.

Метод exec()

Метод, выполняющий инициализацию команды. После выборки команды вызывается этот метод и в аргументах передается запрос в виде массива, разделенный пробелом.

Минимальная реализация пользовательской команды

```
public class Unknown extends Command {

    @Override
    protected void setName() {
        commandName = "unknown";
    }
    @Override
    public String init(String... args) {
        return "Не распознанная команда";
    }
}
```

Использование модуля KeysReader

Класс KeysReader[5] предназначен для обработки ключей команды, вводимых пользователем. На момент версии 1.0.1 содержит два статических метода:

- readKeys(String[] words) - возвращает Map с паттерном: <String ключ, String значение>, где значение может быть пустым. Присутствует JavaDoc. В большинстве случаев достаточно передать ему входной аргумент метода init() и получить карту ключей.
- readOrderedKeys(String[] words) - действует также, как и readKeys, но поддерживает сортировку ключей. Иными словами, он нужен, если важна последовательность введённых ключей. Возвращает TreeMap с паттерном: <Integer index, <String ключ, String значение>>.

[5] https://github.com/AppLoidx/JavaBot/blob/master/src/main/java/core/common/KeysReader.java

```
public String init(String... args) {
    Map<String, String> keysMap = KeysReader.readKeys(args);
    String name;


    if(keysMap.containsKey("-n") || keysMap.containsKey("--name")){
        name = keysMap.get("-n");
    } else {
        return "Не указан обязательный ключ -n [имя_очереди]";
    }
```

Документация к модулям

KeysReader v1.1

Структура модуля:

- public static Map<String, String> **readKeys**(String[] words)
- public static TreeMap<Integer, Map<String, String>> **readOrderedKeys**(String[] words)

readKeys

Версия: 1.2

Считывает ключи в массиве данных и возвращает Map со структурой <ключ>=<значение>. Благодаря тому, что метод получает в качестве аргумента тип строкового массива (получаемого при методе инициализации команды (init()), можно сразу передать в качестве аргумента сам запрос.

Формат кючей:

-[ключ] [его_значение]

Множество ключей:

-[ключ_1] [значение_ключа_1] -[ключ_2] [значение_ключа_2]

-[ключ_1] –[ключ_2] -[значение_ключа_2]

В таком случае ключ_1 получает значение пустой строки (“”)

Совмещение ключей:

-[ключ_1][ключ_2][ключ_3] [значение_ключа_3]

Длинные ключи:

--[ключ_слово] [его_значение]

readOrderedKeys

версия 1.0

Считывание ключей с сохранением порядка. Работает также, как и readKeys(), но имеет структуру TreeMap со структурой в значении Map, таким же как в readKeys().

Этот метод используется в том случае, если важен порядок считывания ключей.

Рекомендуется использовать этот метод лишь при необходимости, так как он затратный, как в памяти, так и в производительности.

Структура TreeMap:

<номер_ключа> = <ключ_со_значением>