Semperante

Backend Developer в IBM

20.06.2018 82941 35

Java @Аннотации. Что это и как этим пользоваться?

Данная статья предназначена для людей, которые никогда не работали с Аннотациями, но хотели бы разобраться, что это и с чем его едят. Если же вы имеете опыт в данной сфере, не думаю, что эта статья как-то расширит ваши знания (да и, собственно, такую цель я не преследую).

Также статья не подходит для тех, кто только начинает изучать язык Java. Если Вы не понимаете что такое `Map<>` или `HashMap<>` или не знаете что означает запись `static{ }` внутри определения класса, либо же никогда не работали с рефлекцией – Вам рано читать эту статью и пытаться понять, что такое аннотации. **Сам по себе этот инструмент не создан для использования новичками, так как требует уже не совсем базовых пониманий взаимодействия классов и объектов** (моё мнение) (спасибо комментариям за то, что показали необходимость этой приписки).

Итак, приступим.

Аннотации в Java являются своего рода метками в коде, описывающими метаданные для функции/класса/пакета. Например, всем известная Аннотация @Override, обозначающая, что мы собираемся переопределить метод родительского класса. Да, с одной стороны, можно и без неё, но если у родителей не окажется этого метода, существует вероятность, что мы зря писали код, т.к. конкретно этот метод может и не вызваться никогда, а с Аннотацией @Override компилятор нам скажет, что: "Я не нашел такого метода в родителях... что-то здесь нечисто".

Однако Аннотации могут нести в себе не только смысл "для надежности": в них можно хранить какие-то данные, которые после будут использоваться.

Для начала рассмотрим простейшие аннотации предоставляемые стандартной библиотекой.

(снова же спасибо комментариям, вначале не подумал что этот блок нужен)

Сначала обсудим, какие бывают аннотации. Каждая из них имеет 2 главных **обязательных** параметра:

- Тип хранения (Retention);
- Тип объекта над которым она указывается (Target).

Тип хранения

Под "типом хранения" понимается стадия до которой "доживает" наша аннотация внутри класса. Каждая аннотация имеет **только один** из возможных "типов хранения" указанный в классе `RetentionPolicy`:

- **SOURCE** - аннотация используется только при написании кода и игнорируется компилятором (т.е. не сохраняется после компиляции). Обычно используется для каких-либо препроцессоров (условно), либо указаний компилятору
- **CLASS** - аннотация сохраняется после компиляции, однако игнорируется JVM (т.е. не может быть использована во время выполнения). Обычно используется для каких-либо сторонних сервисов, подгружающих ваш код в качестве plug-in приложения
- **RUNTIME** - аннотация которая сохраняется после компиляции и подгружается JVM (т.е. может использоваться во время выполнения самой программы). Используется в качестве меток в коде, которые напрямую влияют на ход выполнения программы (пример будет рассмотрен в данной статье)

Тип объекта над которым указывается

Данное описание стоит понимать практически буквально, т.к. в Java аннотации могут указываться над чем угодно (Поля, класса, функции, т.д.) и для каждой аннотации указывается, над чем конкретно она может быть задана. Здесь уже нет правила "что-то одно", аннотацию можно указывать над всем ниже перечисленным, либо же выбрать только нужные элементы класса `ElementType`:

- **ANNOTATION_TYPE** - другая аннотация
- **CONSTRUCTOR** - конструктор класса
- **FIELD** - поле класса
- **LOCAL_VARIABLE** - локальная переменная
- **METHOD** - метод класса
- **PACKAGE** - описание пакета `package`
- **PARAMETER** - параметр метода `public void hello(@Annontation String param){}`
- **TYPE** - указывается над классом

Всего на момент версии Java SE 1.8 стандартная библиотека языка предоставляет нам 10 аннотаций. В данной статье рассмотрим самые часто встречающиеся из них (кому интересны они все Welcome to Javadoc):

@Override

Retention: SOURCE;
Target: METHOD.

Данная аннотация показывает, что метод над котором она прописана, наследован у родительского класса.

Первая аннотация с которой сталкивался каждый начинающий Java-программист, при использовании IDE, которая настойчиво пихает эти @Override. Зачастую учителя с ютуба рекомендуют либо: "сотрите чтобы не мешало", либо: "оставьте не задумываясь зачем оно здесь". На самом деле аннотация более чем полезна, она не только позволяет понять, какие методы были определены в этом классе впервые, а какие уже есть у родителей, что бесспорно повышает читаемость вашего кода, но также данная аннотация служит "самопроверкой", что вы не ошиблись при определении перегружаемой функции.

@Deprecated

Retention: Runtime;

Данная аннотация указывает на методы, классы или переменные, которые является "устаревшими" и могут быть убраны в последующих версиях продукта.

С данной аннотацией обычно сталкиваются те, кто читает документацию каких-либо API, либо той же стандартной библиотеки Java. Иногда эту аннотацию игнорируют, т.к. она не вызывает никаких ошибок и в принципе сама по себе сильно жить не мешает. Однако главный посыл, который несет в себе данная аннотация – "мы придумали более удобный метод реализации данного функционала, используй его, не используй старый" - ну, либо же - "мы переименовали функцию, а это так, для легаси оставили..." (что тоже в общем-то неплохо). Короче говоря, если видите @Deprecated - лучше стараться не использовать то, над чем она висит, если в этом нет прям крайней необходимости и, возможно, стоит перечитать документацию, чтобы понять каким образом теперь реализуется задача, выполняемая устаревшим элементом. Например вместо использований `new Date().getYear()` рекомендуется использовать `Calendar.getInstance().get(Calendar.YEAR)`.

@SuppressWarnings

Retention: SOURCE;
Target: TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE

Данная аннотация отключает вывод предупреждений компилятора, которые касаются элемента над которым она указана. Является SOURCE аннотацией указываемой над полями, методами, классами.

@Retention

Retention: RUNTIME;
Target: ANNOTATION_TYPE;

Данная аннотация задает "тип хранения" аннотации над которой она указана. Да эта аннотация используется даже для самой себя... магия да и только.

@Target

Retention: RUNTIME;
Target: ANNOTATION_TYPE;

Данная аннотация задает тип объекта над которым может указываться создаваемая нами аннотация. Да и она тоже используется для себя же, привыкайте...

Думаю, на это можно завершить ознакомление со стандартными аннотациями библиотеки Java, т.к. остальные используются достаточно редко и, хоть и несут свою пользу, сталкиваться с ними приходится редко и совершенно необязательно. *Если же вы хотите чтобы я рассказал о какой-то конкретной аннотации из стандартной библиотеки (либо, возможно, аннотации типа @NotNull и @Nullable которые в STL не входят) напишите в комментариях - либо вам там ответят добрые пользователи, либо я когда увижу. Если уж много людей будут просить какую-то аннотацию - также внесу её в статью.*

Практическое применение RUNTIME аннотаций

Собственно, думаю, хватит теоретической болтавни: давайте перейдем к практике на примере бота.

Допустим вы хотите написать бота для какой-то соцсети. У всех крупных сетей, таких как ВК, Facebook, Discord, есть свои API, которые позволяют написать бота. Для этих же сетей есть уже написанные библиотеки для работы с API, на языке Java в том числе. Поэтому не будем углубляться в работу какого-либо API или библиотеки. Всё, что нам нужно знать в данном примере — то, что наш бот умеет реагировать на сообщения, отправленные в чат, в котором, собственно, наш бот находится.

Т.е допустим, у нас есть класс `MessageListener` с функцией:

```
1  public class MessageListener
2  {
3      public void onMessageReceived(MessageReceivedEvent event)
4      {
```

Она отвечает за обработку принятого сообщения. Всё что нам нужно от класса `MessageReceivedEvent` — строка полученного сообщения (например, "Привет" или "Бот, привет"). Стоит учесть: в разных библиотеках эти классы называются по-разному. Я использовал библиотеку для Discord.

И вот мы хотим сделать так, чтобы бот реагировал на какие-то команды, начинающиеся с "Бот" (с запятой или без — решайте сами: для урока предположим, что запятой там быть не должно.

То есть, уже наша функция будет начинаться с чего-то вроде:

```
public void onMessageReceived(MessageReceivedEvent event)
{
    //Убираем чувствительность к регистру (БоТ, бОт и т.д.)
    String message = event.getMessage().toLowerCase();
    if (message.startsWith("бот"))
    {

    }
}
```

И вот теперь перед нами есть множество вариантов реализации той или иной команды. Бесспорно, для начала нужно отделить команду от её аргументов, т.е разбить на массив.

```
public void onMessageReceived(MessageReceivedEvent event)
{
    //Убираем чувствительность к регистру (БоТ, бОт и т.д.)
    String message = event.getMessage().toLowerCase();
    if (message.startsWith("бот"))
    {
        try
        {
            //получим массив {"Бот", "(команду)", "аргумент1", "аргумент2",... "аргументN"};
            String[] args = message.split(" ");
            //Для удобства уберем "бот" и отделим команду от аргументов
            String command = args[1].toLowerCase();
            String[] nArgs = Arrays.copyOfRange(args, 2, args.length);
            //Получили command = "(команда)"; nArgs = {"аргумент1", "аргумент2",..."аргументN"};
            //Данный массив может быть пустым
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            //Вывод списка команд или какого-либо сообщения
            //В случае если просто написать "Бот"
        }
    }
}
```

Данного куска кода нам никак не избежать, потому что отделение команды от аргументов нужно всегда. А вот дальше уже у нас есть выбор:

- Сделать if(command.equalsIngnoreCase("..."))
- Сделать switch(command)
- Сделать ещё какой-то способ обработки...

И вот мы наконец дошли до практической части использования Аннотаций.

Давайте рассмотрим код аннотации для нашей задачи (он может отличаться, конечно же).

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

//Указывает, что наша Аннотация может быть использована
//Во время выполнения через Reflection (нам как раз это нужно).
@Retention(RetentionPolicy.RUNTIME)

//Указывает, что целью нашей Аннотации является метод
//Не класс, не переменная, не поле, а именно метод.
@Target(ElementType.METHOD)
public @interface Command //Описание. Заметим, что перед interface стоит @;
{
    //Команда за которую будет отвечать функция (например "привет");
    String name();

     //Аргументы команды, использоваться будут для вывода списка команд
    String args();

     //Минимальное количество аргументов, сразу присвоили 0 (логично)
    int minArgs() default 0;

    //Описание, тоже для списка
    String desc();

     //Максимальное число аргументов. В целом не обязательно, но тоже можно использовать
    int maxArgs() default Integer.MAX_VALUE;

     //Показывать ли команду в списке (вовсе необязательная строка, но мало ли, пригодится!)
    boolean showInHelp() default true;

    //Какие команды будут считаться эквивалентными нашей
    //(Например для "привет", это может быть "Здаров", "Прив" и т.д.)
    //Под каждый случай заводить функцию - не рационально
    String[] aliases();

}
```

Важно! Каждый параметр описывается как функция (с круглыми скобками). В качестве параметров могут быть использованы только примитивы, `String`, `Enum`. Нельзя написать `List<String> args();` — ошибка.

Теперь, когда мы описали Аннотацию, давайте заведем класс, назовем его `CommandListener`.

```
public class CommandListener
{
    @Command(name = "привет",
             args = "",
             desc = "Будь культурным, поздоровайся",
```

```
    public void hello(String[] args)
    {
        //Какой-то функционал, на Ваше усмотрение.
    }

    @Command(name = "пока",
            args = "",
            desc = "",
            aliases = {"удачи"})
    public void bie(String[] args)
    {
         // Функционал
    }

    @Command(name = "помощь",
            args = "",
            desc = "Выводит список команд",
            aliases = {"help", "команды"})
    public void help(String[] args)
    {
        StringBuilder sb = new StringBuilder("Список команд: \n");
        for (Method m : this.getClass().getDeclaredMethods())
        {
            if (m.isAnnotationPresent(Command.class))
            {
                Command com = m.getAnnotation(Command.class);
                if (com.showInHelp()) //Если нужно показывать команду в списке.
                {
                    sb.append("Бот, ")
                       .append(com.name()).append(" ")
                       .append(com.args()).append(" - ")
                       .append(com.desc()).append("\n");
                }
            }
        }
        //Отправка sb.toString();

    }
}
```

Стоит отметить одно небольшое неудобство: т.к. мы сейчас боремся за универсальность, все функции должны иметь одинаковый список формальных параметров, поэтому даже если у команды нет аргументов, у функции должен быть параметр `String[] args`.

Мы сейчас описали 3 команды: привет, пока, помощь.

Теперь давайте модифицируем наш `MessageListener` так, чтобы он как-то с этим работал. Для удобства и скорости работы, будем сразу хранить наши команды в `HashMap`:

```
public class MessageListner
{
    //Map который хранит как ключ команду
    //А как значение функцию которая будет обрабатывать команду
```

```
    //Объект класса с командами (по сути нужен нам для рефлекции)
    private static final CommandListener listener = new CommandListener();

    static
    {
       //Берем список всех методов в классе CommandListener
        for (Method m : listener.getClass().getDeclaredMethods())
        {
            //Смотрим, есть ли у метода нужная нам Аннотация @Command
            if (m.isAnnotationPresent(Command.class))
            {
                //Берем объект нашей Аннотации
                Command cmd = m.getAnnotation(Command.class);
                //Кладем в качестве ключа нашей карты параметр name()
                //Определенный у нашей аннотации,
                //m – переменная, хранящая наш метод
                commands.put(cmd.name(), m);

                //Также заносим каждый элемент aliases
               //Как ключ указывающий на тот же самый метод.
                for (String s : cmd.aliases())
                {
                    commands.put(s, m);
                }
            }
        }
    }

    public void onMessageReceived(MessageReceivedEvent event)
    {

        String message = event.getMessage().toLowerCase();
        if (message.startsWith("бот"))
        {
            try
            {
                String[] args = message.split(" ");
                String command = args[1].toLowerCase();
                String[] nArgs = Arrays.copyOfRange(args, 2, args.length);
                Method m = commands.get(command);
                if (m == null)
                {
                    //(вывод помощи)
                    return;
                }
                Command com = m.getAnnotation(Command.class);
                if (nArgs.length < com.minArgs())
                {
                    //что-то если аргументов меньше чем нужно
                }
                else if (nArgs.length > com.maxArgs())
                {
                    //что-то если аргументов больше чем нужно
                }
```

НАЧАТЬ ОБУЧЕНИЕ

```
63                  //String[] args – иначе она просто не будет найдена;
64                  m.invoke(listener, nArgs);
65              }
66              catch (ArrayIndexOutOfBoundsException e)
67              {
68                  //Вывод списка команд или какого-либо сообщения
69                  //В случае если просто написать "Бот"
70              }
71          }
72      }
73  }
```

Вот собственно и всё, что нужно, чтобы наши команды работали. Теперь добавление новой команды — это не новый if, не новый case, в которых нужно было бы заново переучесть количество аргументов, также пришлось бы переписывать help, добавляя в него новые строки. Теперь же, чтобы добавить команду, нам нужно просто в классе CommandListener добавить новую функцию с аннотацией @Command и всё — команда добавлена, случаи учтены, help дополнен автоматически.

Абсолютно бесспорно, что данную задачу можно решить множеством других путей. Да, всё что можно сделать при помощи аннотаций/рефлекций можно сделать и без них, вопрос лишь в удобстве, оптимальности и размерах кода, конечно же, совать Аннотацию везде где есть малейший намек на то, что получится её использовать - тоже не самый рациональный вариант, во всем нужно знать меру =). Но при написании API, Библиотек или программ, в которых возможно повторение однотипного (но не совсем одинакового) кода, аннотации - бесспорно оптимальное решение.

Semperante

Backend Developer в IBM

Анна Уровень 36, Russian Federation — 27 апреля, 14:05

Спасибо за полезную статью.
public void bie(String[] args), наверное, bye?

Ответить — 0

Egogo Уровень 4, Москва, Russian Federation — 1 ноября 2021, 13:01

спасибо автору за аннотацию)

Ответить — +3

Виталий Работает в **поте лица** — 12 октября 2021, 00:23

Что делать, если System.out.println(a), где a - аннотация, выдает текст содержимого полей аннотации в виде юникода?

Ответить — +1

Денис Кайдунов Уровень 38, Гомель, Беларусь — 2 июля 2021, 21:36

Также статья не подходит для тех, кто только начинает изучать язык Java. Если Вы не понимаете что такое Map<> или HashMap<> или не знаете что означает запись static{ } внутри определения класса, либо же никогда не работали с рефлекцией
->рефлексией

Ответить — 0

Денис Кайдунов Уровень 38, Гомель, Беларусь — 2 июля 2021, 21:35

Тип объекта над которым указывается
Данное описание стоит понимать практически буквально, т.к. в Java аннотации могут указываться над чем угодно (Поля, класса,
->классы

Ответить — 0

Лёхансан Junior Java Developer в **Senla** — 24 февраля 2021, 22:42

Мне было интересно. Автору - спасибо!
Для понимания материала, требуется хотя бы базовое представление об аннотациях.
Интересно, а на JavaRush есть какое-то задание по написанию своего бота?

Ответить — 0

Алексей Уровень 6, Москва, Россия — 10 февраля 2021, 16:13

Что-то я не понял, как MessageListner будет понимать какой метод вызвать? Если все методы аннотированы одной аннотацией, только присвоение переменных разное, что-то не понял где проходит разбор по присвоенным переменным или как вообще это происходит?

Ответить — 0

Edffom Уровень 33, Мирный, Россия — 23 февраля 2021, 05:32

все методы хранятся в мапе commands, метод соответственно будет получен после обработки входящего сообщения и вычленения названия метода (String command = args[1].toLowerCase();) Method m = commands.get(command); Сама аннотация это просто метка в коде которая позволяет легко расширять логику программы - просто добавив в класс CommandListener очередной блок @Command(name = "newCommand".....) {function}

Ответить — +3

Показать еще комментарии

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский