

Professor Hans Noodles

41 уровень

13.05.2019 30779 76

Динамические прокси

Статья из группы **Java Developer**
42328 участников

Вы в группе

Привет!

Сегодня мы рассмотрим достаточно важную и интересную тему — создание **динамических прокси-классов** в Java. Она не слишком простая, поэтому попробуем разобраться с ней на примерах :)



Итак, самый важный вопрос: **что такое динамические прокси и для чего они нужны?**

Прокси-класс — это некоторая «надстройка» над оригинальным классом, которая позволяет нам при необходимости изменить его поведение.

Что значит «*изменить поведение*» и как это работает?

Рассмотрим простой пример.

Допустим, у нас есть интерфейс **Person** и простой класс **Man**, реализующий этот интерфейс

```
1 public interface Person {
2
3     public void introduce(String name);
4 }
```

НАЧАТЬ ОБУЧЕНИЕ

```
7      public void sayFrom(String city, String country);
8  }
9
10 public class Man implements Person {
11
12     private String name;
13     private int age;
14     private String city;
15     private String country;
16
17     public Man(String name, int age, String city, String country) {
18         this.name = name;
19         this.age = age;
20         this.city = city;
21         this.country = country;
22     }
23
24     @Override
25     public void introduce(String name) {
26
27         System.out.println("Меня зовут " + this.name);
28     }
29
30     @Override
31     public void sayAge(int age) {
32         System.out.println("Мне " + this.age + " лет");
33     }
34
35     @Override
36     public void sayFrom(String city, String country) {
37
38         System.out.println("Я из города " + this.city + ", " + this.country);
39     }
40
41     //..геттеры, сеттеры, и т.д.
42 }
```

У нашего класса `Man` есть 3 метода: представиться, назвать свой возраст, и сказать, откуда ты родом.

Представим, что этот класс мы получили в составе готовой JAR-библиотеки и не можем просто взять и переписать его код.

Тем не менее, нам нужно изменить его поведение. К примеру, мы не знаем, какой именно метод будет вызван у нашего объекта, а потому хотим, чтобы при вызове любого из них человек сначала говорил «Привет!» (никто не любит невежливых).



Как же нам в такой ситуации поступить?

Нам понадобятся несколько вещей:

1. **InvocationHandler**

Что это такое? Можно перевести дословно — «перехватчик вызовов». Это довольно точно опишет его предназначение.

InvocationHandler — это специальный интерфейс, который позволяет перехватить любые вызовы методов к нашему объекту и добавить нужное нам дополнительное поведение.

Нам необходимо сделать собственный перехватчик — то есть, создать класс и реализовать этот интерфейс.

Это довольно просто:

```
1  import java.lang.reflect.InvocationHandler;
2  import java.lang.reflect.Method;
3
4  public class PersonInvocationHandler implements InvocationHandler {
5
6      private Person person;
7
8      public PersonInvocationHandler(Person person) {
9          this.person = person;
10     }
11
12     @Override
13     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
14
15         System.out.println("Привет!");
16         return null;
17     }
18 }
```

Нам нужно реализовать всего один метод интерфейса — **invoke()**. Он, собственно, и делает то что нам нужно — перехватывает все вызовы методов к нашему объекту и добавляет необходимое поведение (здесь мы внутри метода **invoke()** выводим в консоль «Привет!»).

2. Оригинальный объект и его прокси.

```
1  import java.lang.reflect.Proxy;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          //Создаем оригинальный объект
8          Man vasia = new Man("Вася", 30, "Санкт-Петербург", "Россия");
9
10         //Получаем загрузчик класса у оригинального объекта
11         ClassLoader vasiaClassLoader = vasia.getClass().getClassLoader();
12
13         //Получаем все интерфейсы, которые реализует оригинальный объект
14         Class[] interfaces = vasia.getClass().getInterfaces();
15
16         //Создаем прокси нашего объекта vasia
17         Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonInvocatio
18
19         //Вызываем у прокси объекта один из методов нашего оригинального объекта
20         proxyVasia.introduce(vasia.getName());
21
22     }
23 }
```

Выглядит не очень просто!

Я специально написал к каждой строке кода комментарий: давай разберемся подробнее, что там происходит.

В первой строке мы просто делаем оригинальный объект, для которого будем создавать прокси.

Следующие две строки могут вызвать у тебя затруднение:

```
1  //Получаем загрузчик класса у оригинального объекта
2  ClassLoader vasiaClassLoader = vasia.getClass().getClassLoader();
3
4  //Получаем все интерфейсы, которые реализует оригинальный объект
5  Class[] interfaces = vasia.getClass().getInterfaces();
```

Но на самом деле ничего особенного здесь не происходит :)

Для создания прокси нам нужен `ClassLoader` (загрузчик классов) оригинального объекта и список всех интерфейсов, которые реализует наш оригинальный класс (то есть `Man`).

Если ты не знаешь что такое `ClassLoader`, можешь почитать [эту статью](#) о загрузке классов в JVM или [эту на Хабре](#), но пока не особо с этим заморачивайся. Просто запомни, что мы получаем немного дополнительной информации, которая потом будет нужна для создания прокси-объекта.

В четвертой строке мы используем специальный класс `Proxy` и его статический метод `newProxyInstance()`:

```
1  //Создаем прокси нашего объекта vasia
2  Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonInvocatio
```

В метод мы передаем ту информацию об оригинальном классе, которую получили на прошлом шаге (его `ClassLoader` и список его интерфейсов), а также объект созданного нами ранее перехватчика — `InvocationHandler`'а. Главное — не забудь передать перехватчику наш оригинальный объект `vasia`, иначе ему нечего будет «перехватывать» :)

Что же у нас в итоге получилось?

У нас теперь есть прокси-объект `vasiaProxy`. Он может вызывать **любые методы интерфейса** `Person`. Почему?

Потому что мы передали ему список всех интерфейсов — вот здесь:

```
1 //Получаем все интерфейсы, которые реализует оригинальный объект
2 Class[] interfaces = vasia.getClass().getInterfaces();
3
4 //Создаем прокси нашего объекта vasia
5 Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonInvocationHandler(vasia));
```

Теперь он «в курсе» всех методов интерфейса `Person`.

Кроме того, мы передали нашему прокси объект `PersonInvocationHandler`, настроенный на работу с объектом `vasia`:

```
1 //Создаем прокси нашего объекта vasia
2 Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonInvocationHandler(vasia));
```

Теперь, если мы вызовем у прокси-объекта любой метод интерфейса `Person`, наш перехватчик «словит» этот вызов и выполнит вместо него свой метод `invoke()`.

Давай попробуем запустить метод `main()`!

Вывод в консоль:

Привет!

Отлично! Мы видим, что вместо настоящего метода `Person.introduce()` вызван метод `invoke()` нашего `PersonInvocationHandler()`:

```
1 @Override
2 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
3
4     System.out.println("Привет!");
5     return null;
6 }
```

И в консоль было выведено «Привет!»

Но это не совсем то поведение, которое мы хотели получить :/

По нашей задумке сначала должно быть выведено «Привет!», а после — сработать сам метод, который мы вызываем.

Иными словами, вот этот вызов метода:

```
1 proxyVasia.introduce(vasia.getName());
```


Как же нам добиться этого? Ничего сложного: просто придется немного похимичить над нашим перехватчиком и методом `invoke()` :)

Обрати внимание, какие аргументы передаются в этот метод:

```
1 public Object invoke(Object proxy, Method method, Object[] args)
```

У метода `invoke()` есть доступ к методу, вместо которого он вызывается, и ко всем его аргументам (Method method, Object[] args).

Иными словами, если мы вызываем метод `proxyVasia.introduce(vasia.getName())`, и вместо метода `introduce()` вызывается метод `invoke()`, внутри этого метода у нас есть доступ и к оригинальному методу `introduce()`, и к его аргументу!

В результате мы можем сделать вот так:

```
1 import java.lang.reflect.InvocationHandler;
2 import java.lang.reflect.Method;
3
4 public class PersonInvocationHandler implements InvocationHandler {
5
6     private Person person;
7
8     public PersonInvocationHandler(Person person) {
9
10         this.person = person;
11     }
12
13     @Override
14     public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
15         System.out.println("Привет!");
16         return method.invoke(person, args);
17     }
18 }
```

Теперь мы добавили в метод `invoke()` вызов оригинального метода.

Если мы попробуем сейчас запустить код из нашего предыдущего примера:

```
1 import java.lang.reflect.Proxy;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         //Создаем оригинальный объект
8         Man vasia = new Man("Вася", 30, "Санкт-Петербург", "Россия");
9
10        //Получаем загрузчик класса у оригинального объекта
11        ClassLoader vasiaClassLoader = vasia.getClass().getClassLoader();
12    }
```

```
15
16     //Создаем прокси нашего объекта vasia
17     Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonIr
18
19     //Вызываем у прокси объекта один из методов нашего оригинального объекта
20     proxyVasia.introduce(vasia.getName());
21 }
22 }
```

то увидим, что теперь все работает как надо :)

Вывод в консоль:

Привет!

Меня зовут Вася

Где это может тебе понадобится? На самом деле, много где.

Паттерн проектирования «динамический прокси» активно используется в популярных технологиях...а я, кстати, и забыл тебе сказать, что **Dynamic Proxy** — это паттерн!

Поздравляю, ты выучил еще один! :)



Так вот, он активно используется в популярных технологиях и фреймворках, связанных с безопасностью.

Представь, что у тебя есть 20 методов, которые могут выполнять только залогиненные пользователи твоей программы. С помощью изученных приемов ты легко сможешь добавить в эти 20 методов проверку того, ввел ли пользователь логин и пароль, не дублируя код проверки отдельно в каждом методе.

Или, к примеру, если ты хочешь создать журнал, куда будут записываться все действия пользователей, это также легко сделать с использованием прокси.

Можно даже сейчас: просто допиши в пример код, чтобы название метода выводилось в консоль при вызове `invoke()`, и ты получишь простенький журнал логов нашей программы :)

В завершение лекции, **обрати внимание на одно важное ограничение.**

Создание прокси объекта происходит на уровне интерфейсов, а не классов. Прокси создается для интерфейса.

Взгляни на этот код:

```
1 //Создаем прокси нашего объекта vasia
2 Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonInvocationHandler(vasia));
```

Здесь мы создаем прокси именно для интерфейса `Person`.

Если попробуем создать прокси для класса, то есть поменяем тип ссылки и попытаемся сделать приведение к классу `Man`, у нас ничего не выйдет.

```
1 Man proxyVasia = (Man) Proxy.newProxyInstance(vasiaClassLoader, interfaces, new PersonInvocationHandl
2
3 proxyVasia.introduce(vasia.getName());
```

Exception in thread "main" java.lang.ClassCastException: com.sun.proxy.\$Proxy0 cannot be cast to Man

Наличие интерфейса — обязательное требование. Прокси работает на уровне интерфейсов.

На этом на сегодня все :)

В качестве дополнительного материала по теме прокси могу порекомендовать тебе отличное [видео](#) и такую [статью](#).

+208

Комментарии (76)

популярные

новые

старые

JavaCoder

Введите текст комментария

Александр Горохов

Уровень 24, Дятьково, Россия

28 июня, 16:58

Я правильно понимаю, что метод `invoke` может перехватывать только методы, которые есть в реализованных классом интерфейсах? Собственные методы класса он не видит?

Ответить

0

On1k

Уровень 37, Krasnogorsk, Russian Federation

29 июня, 22:43

если класс имплементирует интерфейс, то в интерфейсе не появляются методы, которые вы добавите в свой класс. А так как перехватчик приводится к типу интерфейса, то он не знает о существовании методов класса.
(Думаю, что вы правы, но это лишь мое скромное мнение)

Ответить

+1

Anonymous

Уровень 33, San Francisco

5 апреля, 14:42

прокси [видео](#)

Ответить

+1

Polick Rolick

Уровень 32, Краснодар

24 марта, 18:30

автор, пиши еще)))) спасибо за статью)

Ответить

0

Эдуард

Уровень 2, Санкт-Петербург

28 января, 10:54

Лучше пользоваться в подобных примерах, что были выше АОР

НАЧАТЬ ОБУЧЕНИЕ

LuneFoxинженер по сопровождению в BIFITEXPERT19 декабря 2021, 17:23

То есть, если интересующий меня класс не реализует никаких интерфейсов, то фиг я над ним что надстрою, чтобы с лёгкостью запротоколировать все действия в журнале?

Ответить0

Дмитрий МартыщукУровень 30, Одесса, Украина21 января, 13:31

наверное, в таком случае, нужно будет создать интерфейс и имплементировать его в нужном классе. Не знаю, на сколько это легально

Ответить0

LuneFoxинженер по сопровождению в BIFITEXPERT21 января, 19:42

Так фишка в том, что класс может прийти в неизменяемом виде в составе какой-нибудь библиотеки. Получается нужно будет наследоваться просто ради того, чтобы имплементировать интерйфейс)

Ответить0

At0mJava Developer16 декабря 2021, 01:23

Для классов есть cglib

Ответить0

PineappleУровень 45, Абакан, Россия3 августа 2021, 14:54

```
1 //Создаем прокси нашего объекта vasia
2 Person proxyVasia = (Person) Proxy.newProxyInstance(vasiaClassLoader, interfaces,
3
4 //Вызываем у прокси объекта один из методов нашего оригинального объекта
5 proxyVasia.introduce(vasia.getName());
```

вот зачем тут это

```
1 vasia.getName()
```

только сбивает / путает
зачем вообще в метод который работает с this.name

```
1 @Override
2 public void introduce(String name) {
3
4     System.out.println("Меня зовут " + this.name);
5 }
```

передавать еще какое то имя..
или пусть бы тогда и работал с аргументами, а не с **this**

Ответить+3

Nik GrapeУровень 48, Berkeley, United States10 ноября 2021, 22:25

тоже обратил внимание. Сначала я подумал они хотят сделать что то вроде

```
1 @Override
2 public Object invoke(Object proxy, Method method, Object[] args) throws T
3     System.out.println("Привет " + args[0] + "!");
4     return method.invoke(person, args);
5 }
```

тогда например при

```
1 proxyVasia.introduce("Коля");
```

будет:

Привет Коля!
Меня зовут Вася

Ответить+2

Nik GrapeУровень 48, Berkeley, United States10 ноября 2021, 22:26

но как оказалось никакого смысла в этих аргументах не появилось

Ответить0

On1kУровень 37, Krasnogorsk, Russian Federation29 июня, 22:47

Мне кажется, что если мы не вызовем оригинальный метод объекта класса Map, то перехватчику

НАЧАТЬ ОБУЧЕНИЕ

Антон

Уровень 27, Москва, Россия

12 июля 2021, 15:06

...

Можно менять передаваемые аргументы и вообще - делать, что угодно...
Рефлексия - сильная вещь...

Ответить

PaiMei in J#

Grand Master в Eagles' Claw

18 октября 2021, 11:00

...

Это смотря с какой стороны посмотреть, так то она ломает один из принципов ООП)

Ответить

Eugene Semenov

Уровень 23, Санкт-Петербург, Россия

25 мая 2021, 12:54

...

Такой вопрос: а если мы в основной части программы, имплементировав интерфейс, заменили в нем что-то, как это изменение перехватиться? Мы же можем интерфейс менять, насколько помню?

Ответить

LuneFox

инженер по сопровождению в BIFIT

EXPERT

19 декабря 2021, 17:26

...

Если добавишь новый метод в интерфейс, а в прокси импортируются все методы интерфейса, то ничего не должно поменяться, новый метод интерфейса так же будет среди "проксанутых". Если ты об этом.

Ответить

Vladimir Dubrovsky

Уровень 23, Москва

19 мая 2021, 13:23

...

Всегда этим веселила Java.
Вот, у нас есть инкапсуляция, теперь никто не сможет изменить поведение вашего класса и испортить его!
Но, у нас есть рефлексия и прокси, позволяющие изменять поведение любых классов!

Ответить

Maks Panteleev

Java Developer в Bell Integrator

25 июня 2021, 11:46

...

у нас запрещено множественное наследование, НО ! у нас есть интерфейсы)

Ответить

Игорь

Full Stack Developer в IgorApplications

6 июля 2021, 16:11

...

Зря вы так про рефлексию, этот инструмент очень мощный, к примеру в одной программе есть одни классы, а в другой другие, в txt файле у нас написано какие классы и методы нужно вызвать. Мы пытаемся запустить, хоть какие-то классы и методы, хотя не знаем какие. Это можно решить с помощью Class c = Class.forName("Solution");

В Java нет нормальных обобщений, такие как шаблоны в с++, информация о дженериках существует только во время компиляции, в байт коде нет никакой информации о них. В Java эту проблему решает рефлексия, Object, дженерики

Ответить

↺

Показать еще комментарии

ОБУЧЕНИЕ

- Курсы программирования
- Курс Java
- Помощь по задачам
- Подписки
- Задачи-игры

СООБЩЕСТВО

- Пользователи
- Статьи
- Форум
- Чат
- Истории успеха
- Активности

КОМПАНИЯ

- О нас
- Контакты
- Отзывы
- FAQ
- Поддержка



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.

НАЧАТЬ ОБУЧЕНИЕ

ЯЗЫК ИНТЕРФЕЙСА

 Русский

▼

