АЛЬТЕРНАТИВНЫЕ ЯЗЫКИ ДЛЯ JVM

Лекция 12

ПЛАН

- JVM и динамическая типизация
- Метапрограммирование на Groovy

КАК ЭТО СДЕЛАТЬ В JVM

- Вопрос: как в JVM реализовать duck typing ?
- Или динамический полифорфизм ?
- Чтобы мы правильно исполнили

```
a + 5
```

- Пришла ли нам в a строка, число или что-то еще

ЧТО ВНУТРИ

- Смотря в какой версии JVM
- До JDK-7 было 4 JVM-инструкции для вызова
- `invokestatic` - для статических методов
- `invokespecial` - для конструкторов, и приватных

ЧТО ВНУТРИ

- `invokeinterface` - для вызовов через интерфейс
- `invokevirtual` - для обычных методов
- Ближе всего к тому, что надо - `invokevirtual`
- Приглядимся

ПРИМЕР

```
 1 class Q {
 2     public void m() {}
 3 }
 4
 5
 6 class Q2 extends Q {
 7     public void  m() {}
 8 }
 9
10 class Q3 {
11     static void m3(Q q) {
12         q.m();
13     }
14 }
```

БАЙТКОД

```
 1   #5 = Utf8               <init>
 2    #6 = Utf8               ()V
 3    #7 = Methodref          #8.#9          // Q.m:()V
 4    #8 = Class              #10            // Q
 5    #9 = NameAndType        #11:#6         // m:()V
 6   #10 = Utf8               Q
 7   #11 = Utf8               m
 8   #12 = Class              #13            // Q3
 9   #13 = Utf8               Q3
10 // ........................
```

БАЙТКОД

```
 1 // ........................
 2   #14 = Utf8               Code
 3   #15 = Utf8               LineNumberTable
 4   #16 = Utf8               m3
 5   #17 = Utf8               (LQ;)V
 6   #18 = Utf8               SourceFile
 7   #19 = Utf8               Q.java
 8 {
 9
10     descriptor: ()V
11     flags: (0x0000)
12 // ........................
```

БАЙТКОД

```
 1 // ........................
 2     Code:
 3       stack=1, locals=1, args_size=1
 4          0: aload_0
 5
 6          1: invokespecial #1
 7          // Method java/lang/Object."<init>":()V
 8
 9          4: return
10       LineNumberTable:
11         line 11: 0
12 // ........................
```

БАЙТКОД

```
 1 // ........................
 2   static void m3(Q);
 3     descriptor: (LQ;)V
 4     flags: (0x0008) ACC_STATIC
 5     Code:
 6       stack=1, locals=1, args_size=1
 7          0: aload_0
 8          1: invokevirtual #7       // Method Q.m:()V
 9          4: return
10       LineNumberTable:
11         line 13: 0
12         line 14: 4
13 }
```

INVOKEVIRTUAL

- Байткод символьно ссылается на `Q.m`
- Загрузка класса `Q3` предполагает загрузку класса `Q`
- И проверку того, что там есть метод `m`
- И если его нет - это проблема

И ТУТ ПОЯВЛЯЕТСЯ GROOVY

- И вот мы пишем:

```
def f(v) { v.m() }
```

- `def` должен превратиться в какой-то JVM-метод
- И у него должен быть тип параметра
- И это должен быть `Object`

И ТУТ ПОЯВЛЯЕТСЯ GROOVY

- И надо как-то реализовать вызов `m`
- С учетом того, что `v` может быть `java.lang.Integer`
- Или `String`, или `List`
- Беда в том, что у нас нет метода `Object.m`

ВАРИАНТ РЕШЕНИЯ

- Может завести статический runtime-метод
- В него передавать объект и сигнатуру метода
- А он через через рефлексию поищет такой метод
- И если найдет - вызовет

ВАРИАНТ РЕШЕНИЯ

- Это медленно само по себе
- И это плохо стыкуется с JIT
- В JDK7 введен новый механизм вызова
- `invokedynamic` и сопутствующее API в `java.lang.invoke`

INVOKEDYNAMIC

- Ускоряем поиск нужного метода
- И его вызов
- Вводит ряд новых абстракций
- Часть из них полезна и в рамках Java

ПРИМЕР

```
 1 import java.lang.invoke.MethodHandles;
 2 import java.lang.invoke.MethodType;
 3
 4 class C {
 5     public static Object m() {
 6         System.out.println("hello from m");
 7         return "";
 8     }
 9 }
10 // ..............................
```

ПРИМЕР

```
1  // ..............................
2  public class Main {
3      public static void main(String[] args)
4          throws Throwable {
5          MethodHandles.Lookup mhl = MethodHandles.lookup();
6          var t = MethodType.methodType(Object.class);
7          var mh = mhl.findStatic(C.class, "m", t);
8          System.out.println("mh: " + mh);
9          mh.invoke();
10 // ..............................
```

ПРИМЕР

```
1 // ..............................
2         var t2 = MethodType.methodType(String.class);
3         var mh2 = mhl.findVirtual(String.class,
4                                   "trim", t2);
5         System.out.println("mh2: " + mh2);
6         System.out.println("---" + mh2.invoke(" gege ")
7                                  + "-----");
8     }
9 }
```

METHODHANDLE

- Абстракция, объединяющая в себя методы
- И многое еще, сводимое к ним
- Конструкторы, установка и чтение полей
- Ее можно хранить, передавать и вызывать

METHODHANDLES.LOOKUP

- `MethodHandles.Lookup` - factory для `MethodHandler`
- Создается через factory
- Работает с учетом точки создания
- Видит то, что видно из нее

CALLSITE

- Контайнер для экземпляра `MethodHandler`
- Каждый исполнившийся экземпляр invokedynamic связан с экземпляром `CallSite`
- Связь устанавливается при первом исполнении
- И остается, пока JVM работает

CALLSITE

- `CallSite` - абстрактный класс
- Есть три реализации: `ConstantCallSite`, `MutableCallSite`, `VolatileCallSite`
- `ConstantCallSite` намертво привязывает `MethodHandle`
- `MutableCallSite/VolatileCallSite` позволяет менять `MethodHandle`
- Но надо хранить знание о местоположении `invokedynamic`

INVOKEDYNAMIC

- `invokedynamic` указывает на bootstrap-метод
- Его задача - найти и вернуть `CallSite`
- Ему передается `MethodHadle.Lookup`, имя метода и сигнатура
- Возможно, зафиксировать изменяемый `CallSite` для будущего изменения

INVOKEDYNAMIC

- Это универсальный механизм, не только для Groovy
- В Groovy `CallSite` будет дополнительной прослойкой
- Она должна по классу объекта понять, что делать дальше
- И вызвать какой-то `MethodHandle`

МОТИВИРУЮЩИЕ ПРИМЕРЫ

- А зачем это все нужно ?
- Зачем нужны динамически определяемые свойста ?
- И этот ваш duck typing ?
- Чем не устраивает Java/Kotlin ?

CSV

- Мы хотим прочитать таблицу с полем `userid`
- Получить записи из этой таблицы в какой-то структуре
- А еще в кодовой базе есть класс `User`
- С методом `getUserId()`

CSV

- И есть где-то еще метод, который принимает объект класса `User`
- И все, что он с ним делает - это вызов `getUserId()`
- А в Java проблема не только в том, что у нас нет изначально общим методом интерфейса с методом `getUserId()`
- А и в том, что у условного класса `CSVRecord` не будет метода `getUserId`
- А только что-то типа `getField('userid')`

CSV

- А может быть так, что есть две похожих таблицы
- В одной есть поле `userid`, а в другой - просто `id`
- И мы знаем, что это по сути одно и то же
- И мы хотим куда-то передать смесь записей из двух этих таблиц
- Но выковыривать эти поля специально для этого мы не хотим

CSV

- И тут будет очень кстати создать поле-алиас на другое поле
- В Java тоже можно выкрутиться, но посложнее
- Какие-то поля могут быть в целом числовыми, но иногда проскакивают не-числа
- В Groovy в таких случаях можно попытаться сконвертировать поле к более ограниченному типу
- А если не получилось - хранить строкой

CSV

- И такие записи могут не попасть в числовые агрегации
- А на случая попадания - может организовать динамическую обработку
- В Kotlin/Java тоже можно что-то подобное - но с большими приседаниями
- И фактически - это будет тоже динамика

XML, JSON

- Хочется каждый узел воспринимать как объект с атрибутами
- И сюда duck typing тоже
- И не всегда мы уверены, что в одном поле значения одного типа
- И это серьезнее, чем в CSV

HTTP-ЗАГОЛОВКИ

- Переменный набор атрибутов
- Сложная типизация полей
- Интерпретация тела в зависимости от размера
- И от `Content-type`

ПРОФИЛИРОВАНИЕ, ОТЛАДКА

- Через метапрограммирование легко отслеживать вызовы методов
- Считать, профилировать, делать прокси и моки
- Даже то, что в Java делается аннотациями - в Groovy проще
- Порог входа ниже

AST

- Аннотации в Groovy тоже есть
- Но с другим смыслом
- Они позволяют делать преобразования во время компиляции
- В том числе - порождать байткод

ПЕРЕХВАТ ВЫЗОВОВ

- Есть разные способы перехватить вызов метода
- Самый радикальный - реализовать интерфейс `GroovyInterceptable`
- У объектов такого класса будут "замаскированы" все методы
- Даже `println` нельзя будет вызвать

ПРИМЕР

```groovy
class Interception implements GroovyInterceptable {
    def definedMethod() { }

    def invokeMethod(String name, Object args) {
        System.out.println("name: " + name)
        System.out.println("args: " + args)
    }
}
class C {
    static def void main(String[] args) {
        new Interception().m()
        new Interception().definedMethod()
    }
}
```

ВЫЗОВ "СВОЕГО" МЕТОДА

- Можно добраться до метакласса, вызвать `getMetaMethod`
- И вызвать `invoke`
- Можно найти метод через Java-рефлексию
- Первый вариант - более Groovy-style

ПРИМЕР

```groovy
class Interception implements GroovyInterceptable {

    def definedMethod() {
        System.out.println("aaaaa")
    }
// ..............................
```

ПРИМЕР

```
1  // ..............................
2      def invokeMethod(String name, Object args) {
3          if (name == "definedMethod") {
4              System.out.println("meta: " + metaClass)
5              metaClass.getMetaMethod("definedMethod")
6                       .invoke(this)
7              def v = super.getClass()
8                           .getMethods()
9                           .find { it.name == name }
10             v.invoke(this)
11         }
12     }
13 }
14 // ..............................
```

ПРИМЕР

```
1 // ..............................
2 class C {
3     static def void main(String[] args) {
4         new Interception().m()
5         new Interception().definedMethod()
6     }
7 }
```

РАСШИРЕНИЕ "ЧУЖИХ" КЛАССОВ

- Есть несколько способов
- Можно написать вручную классы со статическими методами
- С первыми параметром типа расширяемого класса
- И задействовать JVM-механизм `ServiceLoader`

КАТЕГОРИИ

- Можно воспользоваться уже готовыми расширениями
- И есть механизм локазизации их действия
- По сути - обертка над метапрограммным механизмом Groovy
- Расширения действуют в пределах блока кода

ПРИМЕР

```
 1 class C {
 2     static def void main(String[] args) {
 3         String invoking = 'ha'
 4         invoking.metaClass
 5           .invokeMethod = { String name, Object argv ->
 6             if (name == "length") {
 7                 return delegate.class
 8                                .metaClass
 9                               .getMetaMethod( name, args )
10                                ?.invoke( delegate, args )
11             }
12             'invoked'
13         }
14 // ..................
```

ПРИМЕР

```
1 // ..................
2
3         System.out.println(invoking.length())
4         System.out.println(invoking.replace('a', 'b'))
5         System.out.println("qwerty".length())
6         System.out.println("qwerty".replace('e', 'i'))
7 // ..................
```

ПРИМЕР

```
1  // ....................
2          System.out.println(invoking.metaClass)
3          System.out.println("qwerty".metaClass)
4          System.out.println("abc".metaClass)
5
6        System.out.println(invoking.metaClass.invokeMethod)
7        System.out.println("qwerty".metaClass.invokeMethod)
8        System.out.println("abc".metaClass.invokeMethod)
9      }
10 }
```

ВСПОМОГАТЕЛЬНЫЕ МЕТАКЛАССЫ

- `DelegatingMetaClass` - делегирует "настоящим" методам
- Его `invokeMethod` можно перекрыть
- И выполнять нужную логику до и после вызова

ПРИМЕР

```
1 class TrackerClass extends DelegatingMetaClass {
2     TrackerClass(MetaClass metaClass) {
3         super(metaClass)
4     }
5     TrackerClass(Class theClass) {
6         super(theClass)
7     }
8 // ..............................
```

ПРИМЕР

```
// ..............................
    Object invokeMethod(Object object,
                        String methodName,
                        Object[] args) {
        System.out.println("before call: " + methodName)
        def result = super.invokeMethod(
            object,methodName.toLowerCase(), args)
        System.out.println("after call: " + methodName)
        result
    }
}
// ..............................
```

ПРИМЕР

```
1 // ..............................
2
3 def mc =  new TrackerClass(String.metaClass)
4 mc.initialize()
5
6 s = "hello"
7 s.metaClass = mc
8
9 println(s.substring(1))
```

ПРИМЕР

```
 1 package groovy.runtime.metaclass.java.lang;
 2
 3 class IntegerMetaClass extends DelegatingMetaClass {
 4     IntegerMetaClass(MetaClass metaClass) {
 5         super(metaClass)
 6     }
 7     IntegerMetaClass(Class theClass) {
 8         super(theClass)
 9     }
10 // ..........................
```

ПРИМЕР

```groovy
// ..........................
    Object invokeMethod(
        Object object, String name, Object[] args) {
        if (name =~ /isBiggerThan/) {
            def other = name.split(/isBiggerThan/)[1]
                                .toInteger()
            object > other
        } else {
            return super.invokeMethod(object,name, args);
        }
    }
}

// ................
```

ПРИМЕР

```
1 // ................
2 def i = 10
3
4 assert i.isBiggerThan5()
5 assert !i.isBiggerThan15()
6
7 println i.isBiggerThan5()
```

ПРИМЕР

```
 1 class Book {
 2    String title
 3 }
 4
 5 Book.metaClass.titleInUpperCase << {->
 6     title.toUpperCase()
 7 }
 8
 9 def b = new Book(title:"The Stand")
10
11 assert "THE STAND" == b.titleInUpperCase()
```

ПРИМЕР

```
 1 class Book {
 2   String title
 3 }
 4
 5 def properties = Collections.synchronizedMap([:])
 6
 7 Book.metaClass.setAuthor = { String value ->
 8    properties[System.identityHashCode(delegate) +
 9               "author"] = value
10 }
11 Book.metaClass.getAuthor = {->
12    properties[System.identityHashCode(delegate) +
13               "author"]
14 }
```

ПРИМЕР

```
println(new StringBuilder("11111"))
String.metaClass.constructor = { StringBuilder sb ->
    "qqqqqqqqqq"
}
println(new StringBuilder("11111"))
String.metaClass.constructor = { int v ->
    new String(Integer.toString(v))
}
println(new String(123))
String.metaClass.constructor = { int v ->
    new String(Integer.toString(-v))
}
println(new String(123))
```

ПРИМЕР

```groovy
import groovy.transform.ToString

@ToString
class Book {
    String title
}

Book.metaClass.static.of << { String title ->
    new Book(title:title)
}
// ..............................
```

ПРИМЕР

```
1 // ..............................
2
3 book = Book.of("The Stand")
4 println(book)
5 String.metaClass.static.of << { Book book ->
6     book.title
7 }
8 println(String.of(book))
```

ПРИМЕР

```
 1 class C {
 2     def m1() {
 3         println("C.m1")
 4     }
 5 }
 6
 7 class D extends C {
 8     def m1() {
 9         println("D.m1")
10     }
11 }
12 // ..................
```

ПРИМЕР

```
1 // ....................
2 def f1(C c) {
3     c.m1()
4 }
5
6 f1(new C())
7 f1(new D())
8 // ....................
```

ПРИМЕР

```
 1 // ....................
 2 C.metaClass.m2 = {
 3     println("C.m2")
 4 }
 5 D.metaClass.m2 = {
 6     println("D.m2")
 7 }
 8
 9 def f2(C c) {
10     c.m2()
11 }
12
13 f2(new C())
14 f2(new D())
```

ПРИМЕР

```groovy
class Person {
    String name = "Fred"
}
def methodName = "Bob"

Person.metaClass."changeNameTo${methodName}" = {->
    delegate.name = "Bob"
}

def p = new Person()

assert "Fred" == p.name
p.changeNameToBob()
assert "Bob" == p.name
```

ПРИМЕР

```
 1 import java.lang.reflect.Modifier
 2
 3 def p() {
 4     println(String.metaClass.getMethods().size())
 5     println("".metaClass.getMethods().size())
 6     println("".getClass().getDeclaredMethods()
 7             .toList().stream()
 8             .filter {
 9                 Modifier.isPublic(it.getModifiers())
10             }
11             .count()
12 // ....................
```

ПРИМЕР

```
1 // ..................
2     )
3     println()
4 }
5
6 p()
7 String.metaClass.m1 = {}
8 p()
9 "".metaClass.m2 = {}
10 p()
```

ПРИМЕР

```groovy
import groovy.time.TimeCategory

use(TimeCategory)  {
    println 1.minute.from.now
    println 10.hours.ago

    def someDate = new Date()
    println someDate - 3.months
}
```