Дескрипторы метода в Java

Типа “reflection” но с возможностью менять типы и порядок параметров в методах, но более “мутный” и с меньшим количеством механизмов. При этом может быть намного быстрее за счет проверок доступа во время создания а не выполнения (т.к. может быть многократное использование созданного одного и того же).

*Комменты:*

*… и только совсем непонятно зачем они нужны… В статье об этом ни слова. Или я плохо читал?*

*Мы используем для улучшения производительности. Создавая Method handle единожды, избегаем постоянных проверок во время вызова, которые есть в java.lang.reflect.Method.*

*Насколько я помню в 8+ java java.lang.reflect.Method (если отключить проверки) немного быстрее methodHandle, к тому же можно пойти дальше и сделать из methodHandle лямбду используя LambdaMetafactory.metafactory что даст еще больше удобства и производительности*

*Да, мы LambdaMetafactory.metafactory тоже юзаем, генерируем Consumer<T> на лету — [UiControllerReflectionInspector](https://github.com/cuba-platform/cuba/blob/ad111be9b07f1ebfa142410a69a82fd34e13e13e/modules/gui/src/com/haulmont/cuba/gui/sys/UiControllerReflectionInspector.java" \l "L184)*

**1. Вступление**

В этой статье мы собираемся исследовать важный API, который был представлен в Java 7 и улучшен в следующих версиях, *[java.lang.invoke.MethodHandles](https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/MethodHandles.html)*.

В частности, мы узнаем, что такое дескрипторы методов, как их создавать и как их использовать.

Дескрипторы методов были введены для работы вместе с существующим API-интерфейсом *[java.lang.reflect](https://docs.oracle.com/javase/9/docs/api/java/lang/reflect/package-summary.html)*, поскольку они служат разным целям и имеют разные характеристики.

**2. Что такое дескрипторы методов?**

Дескрипторы методов - это низкоуровневый механизм поиска, адаптации и вызова методов \*\* .

Дескрипторы метода неизменны и не имеют видимого состояния.

Для создания и использования *MethodHandle*требуется 4 шага:

а). Создание поиска

б). Создание типа метода

с). Создание дескриптора метода

д). Вызов дескриптора метода

**2.1. Методы обработки против отражения**

С точки зрения производительности API ***MethodHandles*может быть намного быстрее, чем API Reflection, поскольку проверки доступа выполняются во время создания, а не во время выполнения***(т.е. когда начинается выполнение доступы уже не нужны)*. Эта разница усиливается, если присутствует менеджер безопасности, так как поиск членов и классов подвергается дополнительным проверкам.

Однако, учитывая, что производительность - не единственная мера пригодности для задачи, мы также должны учитывать, что API *MethodHandles*труднее использовать из-за отсутствия таких механизмов, как перечисление классов членов, проверка флагов доступности и другие.

Тем не менее, API *MethodHandles*предоставляет возможность каррировать методы, изменять типы параметров и менять их порядок.

Имея четкое определение и цели API *MethodHandles*, теперь мы можем начать работать с ними, начиная с поиска.

а).**3. Создание *Lookup (тут проходит проверка доступа)***

Первое, что нужно сделать, когда мы хотим создать дескриптор метода, - это получить поиск, фабричный объект, который отвечает за создание дескрипторов методов для методов, конструкторов и полей, которые видимы для класса поиска.

Через API *MethodHandles*можно создать объект поиска с различными режимами доступа.

Давайте создадим поиск, который предоставляет доступ к *public*методам:

MethodHandles.Lookup publicLookup = MethodHandles.publicLookup();

Однако в случае, если мы хотим иметь доступ также к методам *private*и *protected*, мы можем вместо этого использовать метод *lookup ()*:

MethodHandles.Lookup lookup = MethodHandles.lookup();

б).**4. Создание *MethodType***

Чтобы создать *MethodHandle*, созданный объект поиска требует определения типа его входящих и выходящих параметров, и это достигается с помощью класса *MethodType*.

Так же, как *MethodHandle*, даже экземпляры *MethodType*являются неизменяемыми.

Давайте посмотрим, как можно определить *MethodType*, который определяет класс *java.util.List*в качестве возвращаемого типа *(может быть только один)* и массив *Object*в качестве входного типа*(может быть много)*:

MethodType mt = MethodType.methodType(List.class, Object[].class);

В случае, если метод возвращает тип примитива или *void*в качестве возвращаемого типа, мы будем использовать класс, представляющий эти типы (void.class, int.class …​).

Давайте определим *MethodType*, который возвращает значение типа int и принимает *Object*:

MethodType mt = MethodType.methodType(int.class, Object.class);

Теперь мы можем приступить к созданию *MethodHandle*.

с).**5. Создание *MethodHandle***

Чтобы создать *MethodHandle,*мы должны найти его через объект *lookup*или *publicLookup*, предоставив класс источника, имя метода и тип(.methodType).

Пункты 5.1., 5.2., 5.3. для классов принадлежащих Java. Пункты 5.4., 5.5. для собственных классов.

**5.1. Дескриптор метода для методов**

Использование метода *findVirtual ()*позволяет нам создать MethodHandle для метода объекта. Давайте создадим его на основе метода *concat ()*класса *String*:

MethodType mt = MethodType.methodType(String.class, String.class);

MethodHandle concatMH = publicLookup.findVirtual(String.class, "concat", mt);

**5.2. Дескриптор метода для статических методов**

Когда мы хотим получить доступ к статическому методу, мы можем вместо этого использовать метод *findStatic ()*:

MethodType mt = MethodType.methodType(List.class, Object[].class);

MethodHandle asListMH = publicLookup.findStatic(Arrays.class, "asList", mt);

В этом случае мы создали дескриптор метода, который преобразует массив *Objects*в *List*из них.

**5.3. Дескриптор метода для Конструкторов**

Получить доступ к конструктору можно с помощью метода *findConstructor ()*.

Давайте создадим дескриптор метода, который ведет себя как конструктор класса *Integer*, принимая атрибут *String*:

MethodType mt = MethodType.methodType(void.class, String.class);

MethodHandle newIntegerMH = publicLookup.findConstructor(Integer.class, mt);

**5.4. Дескриптор метода для частных методов**

Создать дескриптор метода для частного метода можно с помощью *[java.lang .reflect](https://docs.oracle.com/javase/9/docs/api/java/lang/reflect/package-summary.html)*API.

Давайте начнем с определения класса *Book*:

public class Book {

String id;

String title;

//constructor

private String formatBook() {

return id + " > " + title;

}

}

Теперь мы можем создать дескриптор метода, который будет вести себя точно так же, как метод *formatBook ()*:

Method formatBookMethod = Book.class.getDeclaredMethod("formatBook");

formatBookMethod.setAccessible(true);

MethodHandle formatBookMH = lookup.unreflect(formatBookMethod);

**5.5. Дескриптор метода для полей**

Используя дескриптор метода, можно получить доступ и к полям.

Имея в качестве предварительного условия прямой видимость доступа между дескриптором метода и объявленным свойством, мы можем создать дескриптор метода, который ведет себя как получатель:

MethodHandle getTitleMH = lookup.findGetter(Book.class, "title", String.class);

Для получения дополнительной информации об обработке переменных/полей, посмотрите на ссылку:/java-variable-handles[Демистифицированные дескрипторы переменных Java 9], где мы обсудим <https://docs.oracle.com/javase/9/docs/API>api/java/lang/invoke/VarHandle.html[java.lang.invoke.VarHandle], добавленный в Java 9.

д).**6. Вызов дескриптора метода**

Класс *MethodHandle*предоставляет 3 различных способа выполнения дескриптора метода: *invoke (), invokeWithArugments ()*и *invokeExact ()*.

**6.1. Вызов Точный**

В случае, если мы хотим быть более строгими в способе выполнения дескриптора метода (количество аргументов и их тип), мы должны использовать метод *invokeExact ()*.

Фактически, он не обеспечивает приведение к предоставленному классу и требует фиксированного количества аргументов.

Давайте посмотрим, как мы можем *суммировать два*int\_\_ значения, используя дескриптор метода:

MethodType mt = MethodType.methodType(int.class, int.class, int.class);

MethodHandle sumMH = lookup.findStatic(Integer.class, "sum", mt);

int sum = (int) sumMH.invokeExact(1, 11);

assertEquals(12, sum);

Если в этом случае мы решим передать методу *invokeExact*число, которое не является *int*, вызов приведет к *WrongMethodTypeException.*

**6.2. Вызов дескриптора метода только с фиксацией количества аргументов**

При использовании метода *invoke ()*мы принудительно фиксируем количество аргументов (арность), но разрешаем выполнять приведение и упаковку/распаковку аргументов и возвращаемых типов.

Давайте посмотрим, как можно использовать *invoke ()*с аргументом в рамке:

MethodType mt = MethodType.methodType(String.class, char.class, char.class);

MethodHandle replaceMH = publicLookup.findVirtual(String.class, "replace", mt);

String output = (String) replaceMH.invoke("jovo", Character.valueOf('o'), 'a');

assertEquals("java", output);

В этом случае *replaceMH*требует аргументы *char*, но *invoke ()*выполняет распаковку аргумента *Character*перед его выполнением.

**6.3. Вызов с аргументами**

Вызов дескриптора метода с использованием метода *invokeWithArguments*является наименее ограничительным из трех вариантов.

Фактически, он допускает вызов переменной arity, в дополнение к приведению и упаковке/распаковке аргументов и возвращаемых типов.

На практике это позволяет нам создать *List*из *Integer*, начиная с *array*из *int*значений:

MethodType mt = MethodType.methodType(List.class, Object[].class);

MethodHandle asList = publicLookup.findStatic(Arrays.class, "asList", mt);

List<Integer> list = (List<Integer>) asList.invokeWithArguments(1,2);

assertThat(Arrays.asList(1,2), is(list));

**7. Работа с массивом**

*MethodHandles*не предназначены для работы только с полями или объектами, но также и с массивами. На самом деле, с помощью API *asSpreader ()*можно создать метод расширения массива.

В этом случае дескриптор метода принимает аргумент массива, распространяя его элементы в качестве позиционных аргументов и, необязательно, длину массива.

Давайте посмотрим, как мы можем распространить дескриптор метода, чтобы проверить, равны ли элементы в массиве:

MethodType mt = MethodType.methodType(boolean.class, Object.class);

MethodHandle equals = publicLookup.findVirtual(String.class, "equals", mt);

MethodHandle methodHandle = equals.asSpreader(Object[].class, 2);

assertTrue((boolean) methodHandle.invoke(new Object[]{ "java", "java" }));

**8. Улучшение дескриптора метода**

После того, как мы определили дескриптор метода, можно улучшить его, связав дескриптор метода с аргументом, фактически не вызывая его.

Например, в Java 9 этот тип поведения используется для оптимизации объединения *String*.

Давайте посмотрим, как мы можем выполнить конкатенацию, привязав суффикс к нашему *concatMH*:

MethodType mt = MethodType.methodType(String.class, String.class);

MethodHandle concatMH = publicLookup.findVirtual(String.class, "concat", mt);

MethodHandle bindedConcatMH = concatMH.bindTo("Hello ");

assertEquals("Hello World!", bindedConcatMH.invoke("World!"));

**9. Улучшения Java 9**

В Java 9 было сделано несколько улучшений API *MethodHandles*, чтобы сделать его намного проще в использовании.

Улучшения коснулись 3 основных тем:

* **Функции поиска**- разрешение поиска классов из разных контекстов

и поддерживать неабстрактные методы в интерфейсах Обработка аргументов \*\* - улучшение сворачивания аргумента, аргумента

функции сбора и распространения аргументов Дополнительные комбинации \*\* - добавление циклов ( *loop*, *whileLoop,*

*doWhileLoop …​*) и лучшая поддержка обработки исключений с помощью *tryFinally*

Эти изменения привели к нескольким дополнительным преимуществам:

* Увеличена оптимизация компилятора JVM
* Сокращение использования
* Включена точность в использовании API *MethodHandles*

Подробная информация о внесенных улучшениях доступна по адресу *[MethodHandles](https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/MethodHandles.html)*[API Javadoc](https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/MethodHandles.html).

**10. Заключение**

В этой статье мы рассмотрели API *MethodHandles*, что это такое и как мы можем их использовать.

Мы также обсудили, как он относится к Reflection API, и поскольку дескрипторы метода допускают низкоуровневые операции, лучше избегать их использования, если они не полностью соответствуют объему работы.