

Improving Performance Using Method Handles



José Paumard

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard <https://github.com/JosePaumard>



Agenda



Performances of the reflection API

What can be done to improve it?

Introducing method handles



Setting up the Problem



```
Person person = Person.class.getConstructor().newInstance();
```

```
→ SecurityManager sm = System.getSecurityManager();  
   if (sm != null) {  
       checkMemberAccess(sm, Member.PUBLIC,  
                           Reflection.getCallerClass(), false);  
   }
```

How does this code work?

- check if the caller has the right to access reflection



```
Person person = Person.class.getConstructor().newInstance();
```

```
→ // some code  
Reflection.ensureMemberAccess(caller, this, this, modifiers);  
// some more code
```

How does this code work?

- check if the caller has the right to access reflection
- and then apply some more security





There are several security checks when accessing a class using reflection

All those checks are checked again each time an access is made

This is a costly and has a noticeable impact on performance...

Introducing the MethodHandle API





The entry point is the **Lookup** object

It encapsulates the security information

From it you can create:

- **MethodHandle** instances to access the class, its methods and fields
- **VarHandle** instance to access fields in a concurrent way

Creating a Lookup Object



```
Lookup lookup = MethodHandles.lookup();
```

The entry point is the object returned by the *lookup* static call

This *lookup* objects carries the security information

- it is a different instance for each caller
- it should not be shared with untrusted code!



Creating a Method Handle on a Class



```
Person person = ...;  
  
Class<?> personClass =  
    MethodHandles.lookup() // returns a Lookup instance  
        .findClass(Person.class.getName());
```

The **entry point** is the object returned by the **lookup** call

Then one can call **findClass()** to get a **reference** on that class

This code works almost the **same** as the **Class.forName(String)**



Creating a Method Handle on a Method





How to get a handle on a method?

- get a **MethodType** object
- call the right method on the lookup object

There is also a bridge from **Method** to **MethodHandle**

```
Lookup lookup = MethodHandles.lookup();  
  
// public String getName() { ... }  
MethodType getterType =  
    MethodType.methodType(String.class);  
  
// public void setName(String name) { ... }  
MethodType setterType =  
    MethodType.methodType(void.class, String.class);
```

First we create a method type for a method that returns a String and takes no argument

Then we create a method type for a method that returns void and takes a String argument



```
Lookup lookup = MethodHandles.lookup();

// public Person(String name, int age) { ... }
MethodType constructorType =
    MethodType.methodType(void.class, String.class, int.class);

// public Person() { ... }
MethodType emptyConstructorType =
    MethodType.methodType(void.class);
```

In the case of a constructor, the returned type is *void.class*





The Lookup class has several methods to find a handle:

- findVirtual()
- findConstructor()
- findStatic()

```
Lookup lookup = MethodHandles.lookup();

// public String getName() { ... }
MethodType getterType =
    MethodType.methodType(String.class);

MethodHandle getterHandle =
    lookup.findVirtual(Person.class, "getName", getterType);
```

Getting a handle on a regular instance method



```
Lookup lookup = MethodHandles.lookup();

// public void setName(String name) { ... }
MethodType setterType =
    MethodType.methodType(void.class, String.class);

MethodHandle setterHandle =
    lookup.findVirtual(Person.class, "setName", setterType);
```

Getting a handle on a regular instance method



```
Lookup lookup = MethodHandles.lookup();  
  
// public Person(String name, int age) { ... }  
MethodType constructorType =  
    MethodType.methodType(void.class, String.class, int.class);  
  
MethodHandle constructorHandle =  
    lookup.findConstructor(Person.class, constructorType);
```

Getting a handle on a constructor



```
Lookup lookup = MethodHandles.lookup();

// public Person() { ... }
MethodType emptyConstructorType =
    MethodType.methodType(void.class); // not Void.class

MethodHandle emptyConstructorHandle =
    lookup.findConstructor(Person.class, emptyConstructorType);
```

Getting a handle on an empty constructor



Creating a Method Handle on a Field





The **Lookup** class has also these two methods:

- **findGetter()**
- **findSetter()**

Those methods return handles on a field, not on a getter / setter!

```
Lookup lookup = MethodHandles.lookup();

MethodHandle nameReader =
    lookup.findGetter(Person.class, "name", String.class); // reads name

MethodHandle nameWriter =
    lookup.findSetter(Person.class, "name", String.class); // writes name
```

A handle returned by a *findGetter* gives read access on a field and does not call the *getter* of that field

A handle returned by a *findSetter* gives write access on a field and does not call the *setter* of that field



Using a Method Handle





Invoking a method using a method handle is the same as with an instance of Method

- it uses the `invoke()` method
- the first argument is the object that holds the invoked method
- the following arguments, if any, are passed to the method

```
Person person = ...;
```

```
public class Person {  
    private String name;  
  
    public String getName() {  
        return this.name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```



```
Person person = ...;
```

```
MethodHandle nameGetter = ...;
```

```
String name = (String)nameGetter.invoke(person);
```

```
MethodHandle nameSetter = ...;
```

```
nameSetter.invoke(person, "John");
```

Invoking a **getter** does not require any argument and returns a **String**

Invoking a **setter** does require a **String** argument and returns nothing



```
Person person = ...;
```

```
MethodHandle nameReader = ...;
```

```
String name = (String)nameReader.invoke(person);
```

```
MethodHandle nameWriter = ...;
```

```
nameWriter.invoke(person, "John");
```

A method handle gives access to non private members...



```
Person person = ...;
```

```
Field nameField = Person.class.getDeclaredField("name");  
nameField.setAccessible(true);
```

```
MethodHandle privateNameReader = lookup.unreflectGetter(field);  
String name = (String) privateNameReader.invoke(person);
```

A pre-Java 9 solution is to use the `unreflect()` methods



```
Person person = ...;
```

```
Field nameField = Person.class.getDeclaredField("name");  
nameField.setAccessible(true);
```

```
MethodHandle privateNameWriter = lookup.unreflectWriter(field);  
privateNameWriter.invoke(person, "John");
```

A pre-Java 9 solution is to use the `unreflect()` methods



```
Person person = ...;

Lookup privateLookup =
    MethodHandles.privateLookupIn(Person.class, lookup);

MethodHandle privateNameReader =
    privateLookup.findGetter(Person.class, "name", String.class);

String name = (String)privateNameReader.invoke(person);
```

Java 9 brings a much cleaner solution

Using a lookup object for the private elements of a class




```
Person person = ...;

Lookup privateLookup =
    MethodHandles.privateLookupIn(Person.class, lookup);

MethodHandle privateNameWriter =
    privateLookup.findSetter(Person.class, "name", String.class);

privateNameWriter.invoke(person, "John");
```

Java 9 brings a much cleaner solution

Using a lookup object for the private elements of a class



Creating and Using a Var Handle





The **VarHandle** class has been added in Java 9

It looks like a **MethodHandle** for fields

But a **MethodHandle** can already access a field...

So why has it been added?





VarHandle gives three types of access

- plain, regular access: read and write
- volatile access
- compare and set access

So it deals with concurrent access!

It you do not need it, you can use method handles



```
Lookup lookup = ...;
Person person = ...;

VarHandle nameVarHandle =
    MethodHandles.privateLookupIn(Person.class, lookup)
        .findVarHandle(Person.class, "name", String.class);

String name = (String)nameVarHandle.get(person);

String name = (String)nameVarHandle.getVolatile(person);
```

The *get()* method invokes a var handle in normal mode

The *getVolatile()* method invokes a var handle in volatile mode



```
Lookup lookup = ...;  
Person person = ...;  
  
VarHandle ageVarHandle =  
    MethodHandles.privateLookupIn(Person.class, lookup)  
        .findVarHandle(Person.class, "age", int.class);  
  
int newAge = (int)ageVarHandle.getAndAdd(person, 1);
```

The *getAndAdd()* method atomically add the value passed and returns the previous value



Demo



Let us see some code!

Let us create method and var handles

And see them in action



Module Wrap Up



What did you learn?

Why the Reflection API may be too slow

Performance needs to be measured!

Introduction to the MethodHandle API

With a hint on VarHandle



Course Wrap Up



Accessing data using the Reflection API
and the MethodHandle API

The technical details

Two applications: object relational
mapping and dependency injection

Hints on performance

Using method handles to fix performance
issues

Using var handles for concurrent access



Course Wrap Up



Thank you!

@JosePaumard

<https://github.com/JosePaumard>

