

InvokeDynamic – Der neue Star unter den Byte-Code-Befehlen

Bernd Müller
Ostfalia









Speaker

- ▶ Prof. Computer Science (Ostfalia, HS Braunschweig/Wolfenbüttel)
- ▶ Book author (JSF, JPA, Seam, ...)



- ▶ Member EGs JSR 344 (JSF 2.2) und JSR 338 (JPA 2.1)
- ▶ Jakarta EE WG Guest Member, Microprofile WG Guest Member
- ▶ Managing Director PMST GmbH
- ▶ JUG Ostfalen Co-Organizer
- ▶ bernd.mueller@ostfalia.de
- ▶  @berndmuller.bsky.social
- ▶  @berndmuller@fosstodon.org
- ▶  @berndmuller
- ▶  BerndMuller

Warum wurde InvokeDynamic eingeführt?

Rückblick: Möglichkeiten für Methodenaufrufe (bis Java 6)

- ▶ `invokestatic`
Aufruf statischer Methoden

Rückblick: Möglichkeiten für Methodenaufrufe (bis Java 6)

- ▶ `invokestatic`
Aufruf statischer Methoden
- ▶ `invokeinterface`
Aufruf von Interface-Methoden

Rückblick: Möglichkeiten für Methodenaufrufe (bis Java 6)

- ▶ `invokestatic`
Aufruf statischer Methoden
- ▶ `invokeinterface`
Aufruf von Interface-Methoden
- ▶ `invokespecial`
Aufruf von Konstruktoren, `super()`, privater Methoden

Rückblick: Möglichkeiten für Methodenaufrufe (bis Java 6)

- ▶ `invokestatic`
Aufruf statischer Methoden
- ▶ `invokeinterface`
Aufruf von Interface-Methoden
- ▶ `invokespecial`
Aufruf von Konstruktoren, `super()`, privater Methoden
- ▶ `invokevirtual`
Aufruf von Objektmethoden

Rückblick: Möglichkeiten für Methodenaufrufe (bis Java 6)

- ▶ `invokestatic`
Aufruf statischer Methoden
- ▶ `invokeinterface`
Aufruf von Interface-Methoden
- ▶ `invokespecial`
Aufruf von Konstruktoren, `super()`, privater Methoden
- ▶ `invokevirtual`
Aufruf von Objektmethoden
- ▶ Es fehlt eine Möglichkeit, sehr dynamische Aufrufe zu realisieren!

Bedarfe dynamisch getypter Sprachen

- ▶ Zu Beginn der 2000-er Jahre war JVM als Laufzeitumgebung sehr populär
- ▶ [List of JVM languages \(Wikipedia\)](#) führt Reihe von Sprachen auf JVM auf
- ▶ Einige davon dynamisch getypt

Bedarfe dynamisch getypter Sprachen

- ▶ Zu Beginn der 2000-er Jahre war JVM als Laufzeitumgebung sehr populär
- ▶ [List of JVM languages \(Wikipedia\)](#) führt Reihe von Sprachen auf JVM auf
- ▶ Einige davon dynamisch getypt

```
def add(a, b):  
    return a + b;
```

Bedarfe dynamisch getypter Sprachen

- ▶ Zu Beginn der 2000-er Jahre war JVM als Laufzeitumgebung sehr populär
- ▶ [List of JVM languages \(Wikipedia\)](#) führt Reihe von Sprachen auf JVM auf
- ▶ Einige davon dynamisch getypt

```
def add(a, b):  
    return a + b;
```

- ▶ JVM kennt nur iadd, ladd, fadd, dadd
- ▶ Realisierung daher aufwendiger als nötig

JSR 292: Supporting Dynamically Typed Languages on the Java Platform

JSR

Community

Expert Group

[Summary](#) | [Proposal](#) | [Detail \(Summary & Proposal\)](#)

JSRs: Java Specification Requests

JSR 292: Supporting Dynamically Typed Languages on the Java™ Platform

Stage	Access	Start	Finish
Final Release	Download page	20 Jul, 2011	
Final Approval Ballot	View results	05 Jul, 2011	18 Jul, 2011
Proposed Final Draft	Download page	24 Jun, 2011	
Public Review Ballot	View results	12 Apr, 2011	18 Apr, 2011
Public Review	Download page	16 Feb, 2011	18 Apr, 2011
Early Draft Review	Download page	19 May, 2008	17 Aug, 2008
Expert Group Formation		14 Mar, 2006	17 Oct, 2007
JSR Review Ballot	View results	28 Feb, 2006	13 Mar, 2006

2.1 Please describe the proposed Specification:

There is **growing interest** in running a variety of programming languages on the the Java platform, and consequently, on the Java virtual machine (JVM). **This interest is increasingly focused on dynamically typed languages**, in particular scripting languages.

To make it easier to produce performant, high quality implementations of such languages, we propose to add support at the virtual machine level.

Specifically, we seek to add a new JVM instruction, `invokedynamic`, designed to support the implementation of dynamically typed object oriented languages. We will also investigate support for hotswapping, the capability to modify the structure of classes at run time.

2.1 Please describe the proposed Specification:

There is growing interest in running a variety of programming languages on the the Java platform, and consequently, on the Java virtual machine (JVM). This interest is increasingly focused on dynamically typed languages, in particular scripting languages.

To make it easier to produce performant, high quality implementations of such languages, we propose to add support at the virtual machine level.

Specifically, we seek to add a new JVM instruction, invokedynamic, designed to support the implementation of dynamically typed object oriented languages. We will also investigate support for hotswapping, the capability to modify the structure of classes at run time.

2.1 Please describe the proposed Specification:

There is growing interest in running a variety of programming languages on the the Java platform, and consequently, on the Java virtual machine (JVM). This interest is increasingly focused on dynamically typed languages, in particular scripting languages.

To make it easier to produce performant, high quality implementations of such languages, we propose to add support at the virtual machine level.

Specifically, we seek to add a new JVM instruction, `invokedynamic`, designed to support the implementation of dynamically typed object oriented languages. We will also investigate support for hotswapping, the capability to modify the structure of classes at run time.

Umgesetzt mit Java 7 (GA Juli 2011)

- ▶ Neues Package `java.lang.invoke`:
„The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.“

Umgesetzt mit Java 7 (GA Juli 2011)

- ▶ Neues Package `java.lang.invoke`:
„The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.“
- ▶ Also: API und JVM

Umgesetzt mit Java 7 (GA Juli 2011)

- ▶ Neues Package `java.lang.invoke`:
„The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.“
- ▶ Also: API und JVM
- ▶ Völlig ungewöhnlich: JavaDoc spezifiziert JVM-Funktionalität

Umgesetzt mit Java 7 (GA Juli 2011)

- ▶ Neues Package `java.lang.invoke`:
„The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.“
- ▶ Also: API und JVM
- ▶ Völlig ungewöhnlich: JavaDoc spezifiziert JVM-Funktionalität
- ▶ Package `java.lang.invoke` Java 7

Umgesetzt mit Java 7 (GA Juli 2011)

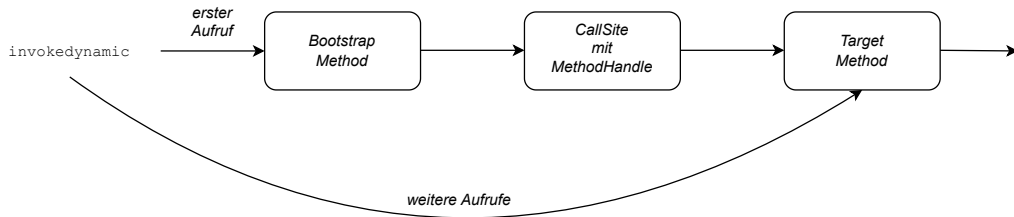
- ▶ Neues Package `java.lang.invoke`:
„The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.“
- ▶ Also: API und JVM
- ▶ Völlig ungewöhnlich: JavaDoc spezifiziert JVM-Funktionalität
- ▶ Package `java.lang.invoke` Java 7
- ▶ Package `java.lang.invoke` Java 23

Umgesetzt mit Java 7 (GA Juli 2011)

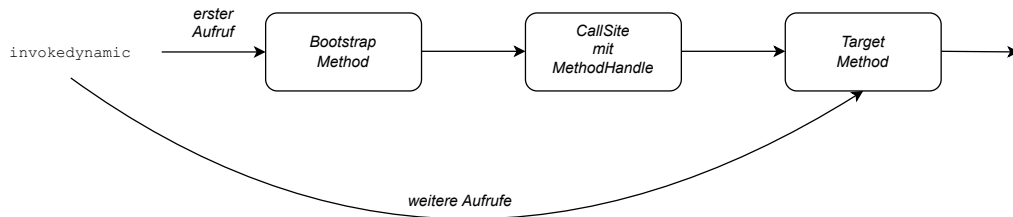
- ▶ Neues Package `java.lang.invoke`:
„The `java.lang.invoke` package contains dynamic language support provided directly by the Java core class libraries and virtual machine.“
- ▶ Also: API und JVM
- ▶ Völlig ungewöhnlich: JavaDoc spezifiziert JVM-Funktionalität
- ▶ Package `java.lang.invoke` Java 7
- ▶ Package `java.lang.invoke` Java 23
- ▶ JVM Specification: 5.4.3.6 Dynamically-Computed Constant and Call Site Resolution

Wurde mit Java 7 eingeführt,
aber **nirgends** verwendet

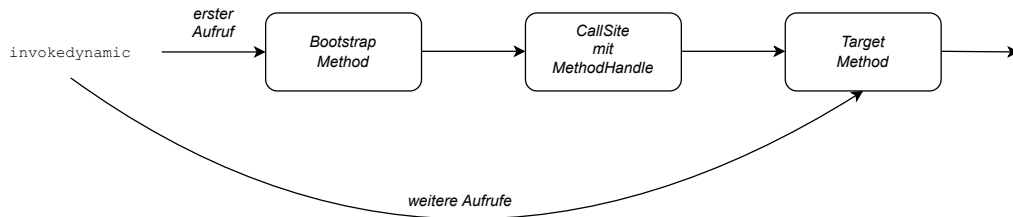
Wie funktioniert es?



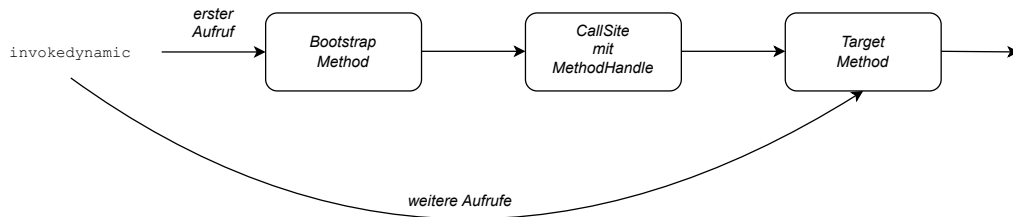
- ▶ Beim ersten Aufruf von `invokedynamic` wird eine *Bootstrap-Methode* (BSM)



- ▶ Beim ersten Aufruf von `invokedynamic` wird eine *Bootstrap-Methode* (BSM)
- ▶ Diese gibt `j.l.i.CallSite` zurück, das `j.l.i.MethodHandle` enthält, das die aktuell aufzurufende Methode enthält



- ▶ Beim ersten Aufruf von `invokedynamic` wird eine *Bootstrap-Methode* (BSM)
- ▶ Diese gibt `j.l.i.CallSite` zurück, das `j.l.i.MethodHandle` enthält, das die aktuell aufzurufende Methode enthält
- ▶ Bei weiteren Aufrufen wird i.d.R. gecachte Methode direkt aufgerufen



- ▶ Beim ersten Aufruf von `invokedynamic` wird eine *Bootstrap-Methode* (BSM)
- ▶ Diese gibt `j.l.i.CallSite` zurück, das `j.l.i.MethodHandle` enthält, das die aktuell aufzurufende Methode enthält
- ▶ Bei weiteren Aufrufen wird i.d.R. gecachte Methode direkt aufgerufen
- ▶ JIT freundlich

Intro MethodHandle

```
MethodType methodType =  
    MethodType.methodType(void.class, String.class);  
MethodHandle methodHandle =  
    MethodHandles.lookup()  
        .findVirtual(PrintStream.class,  
                    "println", methodType);  
methodHandle.invoke(System.out, "Hallo World!");
```

Intro MethodHandle

```
MethodType methodType =  
    MethodType.methodType(void.class, String.class);  
MethodHandle methodHandle =  
    MethodHandles.lookup()  
        .findVirtual(PrintStream.class,  
                    "println", methodType);  
methodHandle.invoke(System.out, "Hallo World!");
```

- ▶ Bei invokedynamic komplizierter: Bootstrap Method mit CallSite und Eintrag in Constant Pool
- ▶ Leider lässt sich kein Beispielprogramm in Java dafür angeben, da Compiler das intern verdrahtet

Wo wird es verwendet?

Fallbeispiele auf Ebene der Bootstrap-Methode

Java 8: Lambdas

- ▶ Lambdas prinzipiell mit anonymen Klassen implementierbar, oder Dynamic Proxy, oder ...


Java 8: Lambdas

- ▶ Lambdas prinzipiell mit anonymen Klassen implementierbar, oder Dynamic Proxy, oder ...
- ▶ Brian Goetz beschreibt in [Translation of Lambda Expressions](#) Alternative mit `invokedynamic`

Java 8: Lambdas

- ▶ Lambdas prinzipiell mit anonymen Klassen implementierbar, oder Dynamic Proxy, oder ...
- ▶ Brian Goetz beschreibt in [Translation of Lambda Expressions](#) Alternative mit `invokedynamic`
- ▶ Dies wurde der erste Einsatz von `invokedynamic` in Java

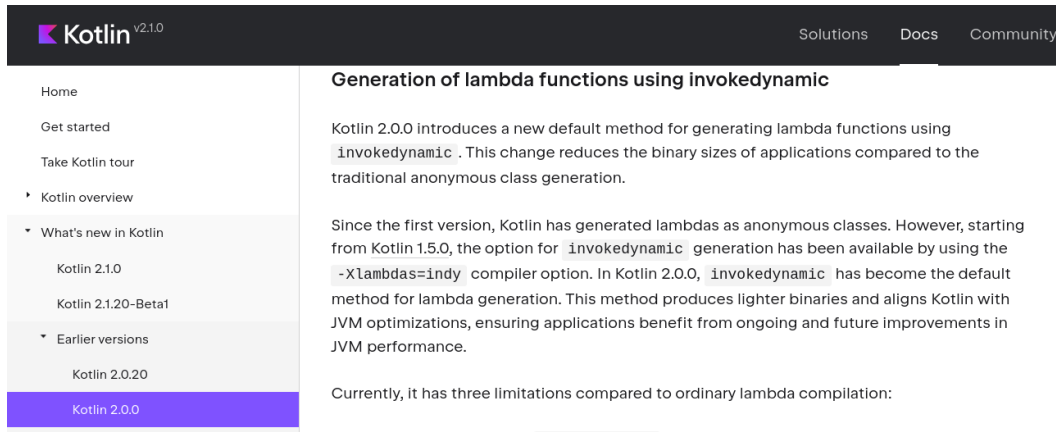
Umsetzung

- ▶ Klasse `java.lang.invoke.LambdaMetafactory` 
- ▶ Mit Methoden `metafactory()` und `altMetafactory()`

JavaDoc der Klasse:

*„Methods to facilitate the creation of simple “function objects” that implement one or more interfaces by delegation to a provided `MethodHandle`, possibly after type adaptation and partial evaluation of arguments. **These methods are typically used as bootstrap methods for invokedynamic call sites, to support the lambda expression and method reference expression features of the Java Programming Language.**“*

Anmerkung: macht Kotlin seit 2.0 auch so



The screenshot shows the Kotlin v2.1.0 documentation website. The header includes the Kotlin logo, version v2.1.0, and navigation links for Solutions, Docs, and Community. The left sidebar contains a navigation menu with links to Home, Get started, Take Kotlin tour, Kotlin overview, What's new in Kotlin, and Earlier versions. Under 'What's new in Kotlin', there are links for Kotlin 2.1.0, Kotlin 2.1.20-Beta1, and Kotlin 2.0.0 (which is highlighted in blue). Under 'Earlier versions', there is a link for Kotlin 2.0.20. The main content area is titled 'Generation of lambda functions using invokedynamic'. The text explains that Kotlin 2.0.0 introduces a new default method for generating lambda functions using `invokedynamic`. This change reduces the binary sizes of applications compared to the traditional anonymous class generation. It also mentions that since the first version, Kotlin has generated lambdas as anonymous classes, but starting from Kotlin 1.5.0, the option for `invokedynamic` generation has been available by using the `-Xlambdas=indy` compiler option. In Kotlin 2.0.0, `invokedynamic` has become the default method for lambda generation. This method produces lighter binaries and aligns Kotlin with JVM optimizations, ensuring applications benefit from ongoing and future improvements in JVM performance. Finally, it states that currently, it has three limitations compared to ordinary lambda compilation.

Generation of lambda functions using invokedynamic

Kotlin 2.0.0 introduces a new default method for generating lambda functions using `invokedynamic`. This change reduces the binary sizes of applications compared to the traditional anonymous class generation.

Since the first version, Kotlin has generated lambdas as anonymous classes. However, starting from Kotlin 1.5.0, the option for `invokedynamic` generation has been available by using the `-Xlambdas=indy` compiler option. In Kotlin 2.0.0, `invokedynamic` has become the default method for lambda generation. This method produces lighter binaries and aligns Kotlin with JVM optimizations, ensuring applications benefit from ongoing and future improvements in JVM performance.

Currently, it has three limitations compared to ordinary lambda compilation:

Java 9: String-Konkatenation

► JEP 280: Indify String Concatenation

„Change the static String-concatenation bytecode sequence generated by javac to use invokedynamic calls to JDK library functions. *This will enable future optimizations of String concatenation* without requiring further changes to the bytecode emitted by javac.“


Java 9: String-Konkatenation

► JEP 280: Indify String Concatenation

„Change the static String-concatenation bytecode sequence generated by javac to use invokedynamic calls to JDK library functions. *This will enable future optimizations of String concatenation* without requiring further changes to the bytecode emitted by javac.“

- Total abgefahren: invokedynamic statt StringBuilder.append() um zukünftige Optimierungen zu ermöglichen

Umsetzung

- ▶ Klasse `java.lang.invoke.StringConcatFactory` 
- ▶ Mit Methoden `makeConcat()` und `makeConcatWithConstants()`

JavaDoc der Klasse:

„Methods to facilitate the creation of String concatenation methods, that can be used to efficiently concatenate a known number of arguments of known types, possibly after type adaptation and partial evaluation of arguments. These methods are typically used as bootstrap methods for invokedynamic call sites, to support the string concatenation feature of the Java Programming Language.“

Demo Time




Sleepy from slides, we are !

Java 14: Records

- ▶ JEP 359, Preview in Java 14
- ▶ JEP 384, Preview in Java 15
- ▶ JEP 395, final in Java 16

Umsetzung

- ▶ Klasse `java.lang.runtime.ObjectMethods` 
- ▶ Mit Methode `bootstrap()`

JavaDoc der Klasse:

„Bootstrap methods for state-driven implementations of core methods, including `Object.equals(Object)`, `Object.hashCode()`, and `Object.toString()`. These methods may be used, for example, by Java compiler implementations to implement the bodies of `Object` methods for record classes.“


Interessant

- ▶ Package `java.lang.runtime` wurde extra hierfür eingeführt

Java 17: Pattern Matching for switch

- ▶ JEP 406, Preview in Java 17
- ▶ JEP 420, Preview in Java 18
- ▶ JEP 427, Preview in Java 19
- ▶ JEP 433, Preview in Java 20
- ▶ JEP 441, final in Java 21

Umsetzung

- ▶ Klasse `java.lang.runtime.SwitchBootstrap` 
- ▶ Mit Methoden `typeSwitch()` und `enumSwitch()`


JavaDoc der Klasse:

„Bootstrap methods for linking invokedynamic call sites that implement the selection functionality of the switch statement. The bootstraps take additional static arguments corresponding to the case labels of the switch, implicitly numbered sequentially from $[0..N)$.“

Java 21: String-Templates

- ▶ JEP 430, Preview in Java 21
- ▶ JEP 459, Preview in Java 22
- ▶ JEP 465, zurückgezogen

Umsetzung

- ▶ Klasse `java.lang.runtime.TemplateRuntime` 
- ▶ Mit Methoden `newStringTemplate()`, `newLargeStringTemplate()`, `processStringTemplate()`


JavaDoc der Klasse:

„Manages string template bootstrap methods. These methods may be used, for example, by Java compiler implementations to create `StringTemplate` instances.“

Java 22: Class-File API

- ▶ JEP 457, Preview in Java 22
- ▶ JEP 466, Preview in Java 23
- ▶ JEP 484, final in Java 24

Umsetzung

- ▶ Klasse `java.lang.classfile.BootstrapMethodEntry` 
- ▶ Mit Methode `bootstrapMethod()`

JavaDoc der Klasse:

„Models an entry in the bootstrap method table. The bootstrap method table is stored in the BootstrapMethods attribute, but is modeled by the Constant, since the bootstrap method table is logically part of the constant pool.“

Zusammenfassung

Zusammenfassung

- ▶ invokedynamic mit Java 7 zur besseren Unterstützung dynamischer Sprachen eingeführt

Zusammenfassung

- ▶ invokedynamic mit Java 7 zur besseren Unterstützung dynamischer Sprachen eingeführt
- ▶ Wurde aber in Java 7 nicht verwendet

Zusammenfassung

- ▶ invokedynamic mit Java 7 zur besseren Unterstützung dynamischer Sprachen eingeführt
- ▶ Wurde aber in Java 7 nicht verwendet
- ▶ Jetzt (seit Java 8) verwendet, um höhere Flexibilität zu haben

Zusammenfassung

- ▶ invokedynamic mit Java 7 zur besseren Unterstützung dynamischer Sprachen eingeführt
- ▶ Wurde aber in Java 7 nicht verwendet
- ▶ Jetzt (seit Java 8) verwendet, um höhere Flexibilität zu haben
- ▶ Auch verwendet, um zukünftige, bessere Implementierungsideen nutzen zu können

Zusammenfassung

- ▶ invokedynamic mit Java 7 zur besseren Unterstützung dynamischer Sprachen eingeführt
- ▶ Wurde aber in Java 7 nicht verwendet
- ▶ Jetzt (seit Java 8) verwendet, um höhere Flexibilität zu haben
- ▶ Auch verwendet, um zukünftige, bessere Implementierungsideen nutzen zu können
- ▶ Und das wirklich Erstaunliche zum Schluss: Es gibt öffentliche Klassen und Methoden, die nicht zur Verwendung durch Entwickler, sondern ausschließlich durch den Compiler vorgesehen sind

