Groovy 3
und das neue Meta Object Protocol in Beispielen

Jochen Theodorou, GoPivtoal Germany GmbH
@JochenTheodorou

...natürlich in viel zu wenigen Beispielen

Jochen Theodorou, GoPivtoal Germany GmbH
@JochenTheodorou

> Was nicht vorwärts gehen kann, schreitet zurück.

Johann Wolfgang von Goethe
(1749-1832), dt. Dichter

Ziele:

- **Metaklassen als Persistent Collection und Realms**
  - Bessere Performance bei Multithreading
  - Verschiedene Versionen von Metaklassen
- **Nur noch interne Metaklassen**
  - Kein extends MetaClassImpl mehr, dafür alles quasi ExpandoMetaClass
  - Delegation statt Vererbung
- **MOP Methoden**
  - Namen und Signaturen ändern sich um Konflikte mit normalen Methoden zu vermeiden
  - GroovyObject wird nur noch Markerinterface oder entfernt
  - Alle Methoden dann optional und in verschiedenen Varianten
  - Keine Exceptions für MOP-Steuerung mehr
- **Sonstiges**
  - Alle Operatoren durch Metaprogramming änderbar
  - Basierend auf invokedynamic und ClassValue (Java 7 wird Minimalversion)

MOP Methoden

= Auf Klassenebene definierte Methoden zur Interaktion mit dem MOP

Übersicht:

- **Uncached:**
  - Schwierigkeitsgrad: Anfänger
  - methodMissing, methodIntercept, propertyMissing, propertyIntercept
- **Cached 1:**
  - Schwierigkeitsgrad: Semiprofi
  - cacheMethodMissing, cacheMethodIntercept, cachePropertyMissing, cachePropertyIntercept
- **Cached 2:**
  - Schwierigkeitsgrad: MethodHandles-Profi
  - cacheMethodTarget, cachePropertyTarget

MOP Methoden – Ein Groovy Builder

Groovy 1 und Groovy 2

spring by Pivotal

Ein Groovy Builder in Groovy 1/2

```
class XmlBuilder {
    def out
    def methodMissing(String name, args) {
        out << "<$name>"
        if(args[0] instanceof Closure) {
            args[0].delegate = this
            args[0].call()
        } else {
            out << args[0].toString()
        }
        out << "</$name>"
    }
}
def xml = new XmlBuilder(out:System.out)
xml.html {
    head {
        title "Hello World"
    }
    body {
        p "Welcome!"
    }
}
```

```
<html><head><title>Hello World</title></head><body><p>Welcome!</p></body></html>methodMi
```

Ein paar Erklärungen:

- methodMissing wird immer aufgerufen, wenn eine Method für den Aufruf nicht gefunden wurde
- Das Ergebnis des Methodenaufrufs ist dann das Ergebnis des Aufrufs von methodMissing
- Args enthält die Argumente für den Methodenaufruf

spring by Pivotal

MOP Methoden – Ein Groovy Builder

Groovy 3 Uncached

spring by Pivotal

Ein Groovy Builder in Groovy 3 - Uncached

```
class XmlBuilder {
    def out
    MopResult methodMissing(String name, args) {
        out << "<$name>"
        if(args[0] instanceof Closure) {
            args[0].delegate = this
            args[0].call()
        } else {
            out << args[0].toString()
        }
        out << "</$name>"
       return MopResult.NOTHING //oder return null
    }
}
def xml = new XmlBuilder(out:System.out)
xml.html {
    head {
        title "Hello World"
    }
    body {
        p "Welcome!"
    }
}
```

Ein paar Erklärungen:

- MopResult gibt das Ergebnis des Methodenaufrufs an
- Ähnlich zu Java 8 Optionals kann es aber auch kein Ergebnis (ERROR) bedeuten, was dann in der Regel zu einer MethodMissingException führt
- NOTHING bedeutet schlicht kein Ergebnis und wird zu null
- Wird auch für Properties verwendet, dann natürlich mit PropertyMissingException

MOP Methoden – Ein Groovy Builder

Groovy 3 Cached

spring by Pivotal

Ein Groovy Builder in Groovy 3 - Cached

```
class XmlBuilder {
    def out
    void cacheMethodMissing
            (GroovyCall call, String name, args)
    {
        call.chain {out << "<${name}>"}
        if (args[0] instanceof Closure) {
            call.arg(0).chain {it.delegate = this}
            call.arg(0).execute()
        } else {
            call.arg(0).execute "toString"
        }
        call.chain {out << "</${name}>"}
    }
}
```

Ein paar Erklärungen:

- Um Caching zu ermöglichen wird eine High-Level-API zur verfügung gestellt
- Diese drückt aus wie man zum Ergebnis kommt, nicht was das Ergebnis ist
- Daher ist die Methode immer void
- GroovyCall wird für jeden noch ***nicht*** gecached Aufruf neu erzeugt
- Caching erfolgt pro Callsite (ist Methodenaufrufpunkt eine gute Übersetzung?)
- Die Methode zum Cachen kann mehrfach aufgerufen werde, zum Beispiel wenn sie die Typen der Argumente oder des Receivers ändern
- args steht jetzt nur Exemplarisch für den Methodenaufruf der zum caching führt, nicht für alle. Will man auf verschiedene Werte bei gleichem Typ unterschiedlich reagieren, kann man nicht cachen

spring by Pivotal

MOP Methoden – Ein Groovy Builder

Groovy 3 MethodHandles

spring by Pivotal

Ein Groovy Builder in Groovy 3 – MethodHandles

```
class XmlBuilder {
    def out
    private finalMethodHandle outField = LOOKUP.findGetter....
    void cacheMethodTarget
            (GroovyCall call, String name, args)
    {
        call.chain(
            GroovyInvoker.bind(outField,"leftShift","<${name}>"))
        if (args[0] instanceof Closure) {
            call.arg(0).chain(
                GroovyInvoker.setProperty("delegate",this))
            call.arg(0).execute()
        } else {
            call.arg(0).execute "toString"
        }
        call.chain(
            GroovyInvoker.bind(outField,"leftShift","</${name}>"))

    }
}
```

Ein paar Erklärungen:

- Ziel ist es ein MethodHandle zu erzeugen
- GroovyCall-High-Level-API hilft und kann mit direkt erzeugten handles gemischt werden
- Zwar ist die MethodHandle-API von Java7 effektiv und mächtig, aber leider oft auch schwer zu verstehen – zum Beispiel weil exakt passende Typen notwendig sind
- Ultimativ (und nicht in diesem Beispiel) wird es möglich sein selbst eine Bootstrap-Methode anzugeben und dann selbst zu entscheiden welche Art der Interaktion man mit dem MOP haben will. Auf diese Art kann man auch komplett auf Javalogik umstellen wenn man will (invokedynamic hat natürlich noch immer einen Performance-Overhead)

spring by Pivotal

MOP Methoden – Dynamic Finder

Groovy 1 und Groovy 2

spring by Pivotal

Dynamic Finder in Groovy 1/2

```
class GORM {
    // an array of dynamic methods that use regex
    def dynamicMethods = [...]
    def methodMissing (String name, args) {
        def method = dynamicMethods.find { it.match(name) }
        if (method) {
            GORM.metaClass."$name" = { Object[] varArgs ->
                method.invoke(delegate, name, varArgs)
            }
            return method.invoke(delegate, name, args)
        }
        else throw new
          MissingMethodException(name, delegate, args)
    }
}
```

Ein paar Erklärungen:

- **Fügt der Metaklasse für jeden match eine Methode hinzu**
- **Problem:**
  - Metaklasse kann nicht mehr so einfach freigegeben werden

In Groovy 1&2 ist die Metaklasse zwar einer Klasse oder Instanz zugeordnet, muss aber in einer globalen Struktur gespeichert werden. Diese nutzt SoftReference, was den Speicherbedarf belastet und Garbage-Collection herausfordert

MOP Methoden – Dynamic Finder

Groovy 3 Uncached

spring by Pivotal

Dynamic Finder in Groovy 3

```
class GORM {
    // an array of dynamic methods that use regex
    def dynamicMethods = [...]
    MopResult methodMissing (String name, args) {
        def method = dynamicMethods.find { it.match(name) }
        if (method) {
            GORM.metaClass."$name" = { Object[] varArgs ->
                method.invoke(delegate, name, varArgs)
            }
            return MopResult.of (GroovyInvoker.invoke
                    (method, delegate, name, args))
        }
        else return MopResult.ERROR
    }
}
```

Ein paar Erklärungen:

- **Fügt der Metaklasse für jeden match eine Methode hinzu**

- **ABER:**

  - Dank ClassValue lebt die Metaklasse nicht viel länger als die Klasse selbst

  - Braucht daher keine SoftReferences

  - Erleichtert Garbage-Collection

MOP Methoden – Dynamic Finder

Groovy 3 Cached

spring by Pivotal

Dynamic Finder in Groovy 3

```
class GORM {
    // an array of dynamic methods that use regex
    def dynamicMethods = [...]
    void cacheMethodMissing
          (GroovyCall call, String name, args)
    {
        def method = dynamicMethods.find { it.match(name) }
        if (method) {
            call.chain (method, delegate, name)
        }
        else call.error()
    }
}
```

Ein paar Erklärungen:

- **Die „Methode“ wird am Aufrufpunkt gespeichert**
  - Sobald die aufrufende Klasse Speichermüll ist, kann der gecachte Wert auch entfernt werden
- **Die Metaklasse wird nicht erweitert**

spring by Pivotal

Ein paar Erklärungen:

- **Die „Methode“ wird am Aufrufpunkt gespeichert**
  - Sobald die aufrufende Klasse Speichermüll ist, kann der gecachte Wert auch entfernt werden
- **Die Metaklasse wird nicht erweitert**

Metaklassen Versionen

spring by Pivotal

Metaklassen Versionen und Sichten

Wiederherstellen einer alten Version einer Metaklasse

```
class A{}

A.metaClass.foo = {->1}
assert new A().foo() == 1

def oldMetaClass = A.metaClass
A.metaClass.foo = {->2}
assert new A().foo() == 2

oldMetaClass.restore()
assert new A().foo() == 1
```


spring by Pivotal

Ein paar Erklärungen:

- Vor Groovy 3 wurde immer die Metaklasse mutiert
- Herstellung des alten Zustandes daher schwierig
- Isolation von Änderungen fast unmöglich

spring by Pivotal

Anwendung mit Realm

```
class A{}

A.metaClass.foo = {->1}
assert new A().foo() == 1

realm {
   A.metaClass.foo = {->2}
   assert new A().foo() == 2
}

assert new A().foo() == 1
```

Ein paar Erklärungen:

- **Realm ist eine Sicht auf eine Metaklassenversion vom Aufrufpunkt aus**
- **Änderungen innerhalb des Realms, bleiben im Realm und seine Subrealms**
- **Verlassen des Realms heißt die alte Version der Metaklasse ist wieder Sichtbar**

Anwendung:

- Bibliotheken können Code isolieren und vor Änderungen von außen schützen
- Lexikographische Categorien

spring by Pivotal

Leider ist viel zu wenig Zeit :(