Faster Groovy in 1.8

Jochen „blackdrag“ Theodorou

SpringSource/VMWare

About blackdrag

- Working on Groovy Core for about 5 years now
- Almost as long as that Tech Lead of Groovy
- Currently Employed at VMWare, in the SpringSource Devision
- Responsible for most of the technical side of Groovy

Email: blackdrag@gmx.org

Groovy is a strong and dynamic typed language with static elements

Groovy in Keywords

- Dynamic Language
- Has an MOP (add/remove/update methods)
- Instance based multimethods
- Multi threaded (uses java threads)
- Runtime class generation or compilation to file
- Joint compilation of Groovy and Java (or for example Scala)
- Compiles to normal Java classes with all signatures visible

Groovy in Keywords

- Tight integration with Java (Groovy extends Java extends Groovy)
- Support for generics signatures
- Support for annotations
- Inner classes
- Overloaded Methods
- Support for closures
- Duck typing

Groovy in Keywords

- Dynamic typing
- Static typing possible, but with a different concept
- Supports java security model
- Native Java Bean property support

Differences to Java

- Array init syntax is not supported
- Semis are optional
- No generics testing in expressions
- Braces are partially optional
- Native lists and maps
- Additional loop constructs
- Additional methods on standard classes

Disclaimer

Should I mention closures, so I am talking about *groovy.lang.Closure* and what we do with it. I am not talking about closures in the strict functional sense - more about a mix between open blocks and anonymous functions, implemented using inner classes.

Groovy has closures

Some Important Projects

Grails for Web Applications

Griffon for Swing Applications

Gradle for Buildsystems

Gparalizer for Parallel Computing

Fibonacci – A stupid Example

```
int fib(int n) {
  if (n<2) return 1
  return   fib(n-1)
         + fib(n-2)
}
```

- Fib calls cannot be easily inlined or eliminated
- The performance of this depends on 1 compare, 1 plus, 2 minus and 2 method calls per fib-run
- All these are quite fast on the JVM, if you do it like Java

Fibonacci – A stupid Example

```
int fib(int n) {
  if (n<2) return 1
  return   fib(n-1)
         + fib(n-2)
}
```

- In Groovy n<2 is a method call
- n-1/n-2 are method calls
- a+b is a method call
- fib(x) is a method call
- All method calls are „dynamic“

Fibonacci – A stupid Example

- While integer math can be easily converted into a few simple CPU codes, a method call often reuqires a jump
- If the JIT cannot inline those method calls, we will jump a lot
- In many cases this is no problem for Groovy or other dynamic languages since a big part of the computation is done in Java world
- For the Fibonacci example this is different unless you use for example BigInteger

Groovy 1.0.x / Groovy 1.5.x

```
public int fib(int n) { //GROOVY 1.5.x
    Integer nInt = n;
    if (ScriptBytecodeAdapter.compareLessThan(nInt, 2)) return 1;
    Object n_1 = ScriptBytecodeAdapter.invokeMethodN(
                Fib2.class, nInt, "minus", new Object[]{1});
    Object fib_n_1 = ScriptBytecodeAdapter.invokeMethodN(
                Fib2.class, this, "fib", new Object[]{n_1});

    Object n_2 = ScriptBytecodeAdapter.invokeMethodN(
                Fib2.class, nInt, "minus", new Object[]{2});
    Object fib_n_2 = ScriptBytecodeAdapter.invokeMethodN(
                Fib2.class, this, "fib", new Object[]{n_2});

    Object ret = ScriptBytecodeAdapter.invokeMethodN(
                Fib2.class, fib_n_1, "plus", new Object[]{fib_n_2});
    return DefaultTypeTransformation.intUnbox(ret);
}
```

Weak Point: Centralized Invocation

```
public int fib(int n) { //GROOVY 1.5.x
      Integer nInt = n;
      if (ScriptBytecodeAdapter.compareLessThan(nInt, 2)) return 1;
      Object n_1 = ScriptBytecodeAdapter.invokeMethodN(
                  Fib2.class, nInt, "minus", new Object[]{1});
      Object fib_n_1 = ScriptBytecodeAdapter.invokeMethodN(
                  Fib2.class, this, "fib", new Object[]{n_1});

      Object n_2 = ScriptBytecodeAdapter.invokeMethodN(
                  Fib2.class, nInt, "minus", new Object[]{2});
      Object fib_n_2 = ScriptBytecodeAdapter.invokeMethodN(
                  Fib2.class, this, "fib", new Object[]{n_2});

      Object ret = ScriptBytecodeAdapter.invokeMethodN(
                  Fib2.class, fib_n_1, "plus", new Object[]{fib_n_2});
      return DefaultTypeTransformation.intUnbox(ret);
}
```

ScriptBytecodeAdapter.invokeMethodN is used to invoke the methods minus, fib and plus using the MetaClass – and in the end Reflection. The receiver and arguments types do differ.

InvokeMethodN and the invoking methods in MetaClass will be megamorphic... Reflection is bad anyway

Weak Point: Central Handle all Cases Method

```
public int fib(int n) { //GROOVY 1.5.x
    Integer nInt = n;
    if (ScriptBytecodeAdapter.compareLessThan(nInt, 2))
        return 1;
    [...]
}
```

„Handle all Cases Methods“:
are typically methods taking Objects and then inside branch off by using for example instanceof

Weak Point: Central Handle all Cases Method

```
private static int compareToWithEqualityCheck(Object left, Object right, boolean equalityCheckOnly) {
    if (left == right) {
        return 0;
    }
    if (left == null) {
        return -1;
    } else if (right == null) {
        return 1;
    }
    if (left instanceof Comparable) {
        if (left instanceof Number) {
            if (isValidCharacterString(right)) {
                return DefaultGroovyMethods.compareTo((Number) left, (Character) box(castToChar(right)));
            }
            if (right instanceof Character || right instanceof Number) {
                return DefaultGroovyMethods.compareTo((Number) left, castToNumber(right));
            }
        } else if (left instanceof Character) {
            if (isValidCharacterString(right)) {
                return DefaultGroovyMethods.compareTo((Character)left,(Character)box(castToChar(right)));
            }
            if (right instanceof Number) {
                return DefaultGroovyMethods.compareTo((Character)left,(Number)right);
            }
        } else if (right instanceof Number) {
            if (isValidCharacterString(left)) {
                return DefaultGroovyMethods.compareTo((Character)box(castToChar(left)),(Number) right);
            }
        } else if (left instanceof String && right instanceof Character) {
            return ((String) left).compareTo(right.toString());
        } else if (left instanceof String && right instanceof GString) {
            return ((String) left).compareTo(right.toString());
        }
        if (!equalityCheckOnly || left.getClass().isAssignableFrom(right.getClass())
                || (right.getClass() != Object.class && right.getClass().isAssignableFrom(left.getClass())) //GROOVY-4046
                || (left instanceof GString && right instanceof String)) {
            Comparable comparable = (Comparable) left;
            return comparable.compareTo(right);
        }
    }

    if (equalityCheckOnly) {
        return -1; // anything other than 0
    }
    throw new GroovyRuntimeException("Cannot compare " + left.getClass().getName() + " with value '" +
            left + "' and " + right.getClass().getName() + " with value '" + right + "'");
}
```

Weak Point: Central Handle all Cases Method

```
private static int compareToWithEqualityCheck(
                    Object left, Object right, boolean equalityCheckOnly)
{
    [...]
    if (left instanceof Comparable) {
        if (left instanceof Number) {
            [...]
        } else if (left instanceof Character) {
            [...]
        } else if (right instanceof Number) {
            [...]
        } else if (left instanceof String && right instanceof Character) {
            [...]
        } else if (left instanceof String && right instanceof GString) {
            [...]
        }
        [...]
    }
    [...]
}
```

Methods like these do get easily too big to be efficiently handled

Back to stupid Fibonacci

- Java: fib(42) takes about 3.4s
- Groovy 1.5.x: fib(42) takes about 10 minutes (Java*193)

So we encouraged people to write such „hot spots“ in Java

Groovy 1.6.x / Groovy 1.7.x

```
private static CallSiteArray csa =
            new CallSiteArray(Fib.class,"fib","minus","plus");
private static Integer const$1 = 1, const$2 = 2;

public int fib(int n) { // GROOVY 1.6.x / 1.7.x
     Integer nInt = n;
     CallSite[] cs = csa.array;
     if (ScriptBytecodeAdapter.compareLessThan(nInt, const$2))
                                                  return const$1;
     Object n_1 = cs[1].call(nInt,const$1);
     Object fib_n_1 = cs[0].call(this,n_1);
     Object n_2 = cs[1].call(nInt,const$2);
     Object fib_n_2 = cs[0].call(this,n_2);
     Object ret = cs[2].call(fib_n_1,fib_n_2);
     return ret;
}
```

Weak Point: Central Handle all Cases Method

```
public int fib(int n) { //GROOVY 1.0.x - GROOVY 1.7.x
     [...]
     if (ScriptBytecodeAdapter.compareLessThan(nInt, const$2))
          return 1;
     [...]
}
```

compareLessThan is still the same – so same problems.

CallSite and CallSiteArray

- Each class has a CallSiteArray, storing all CallSite objects
- Each invocation through a CallSite may replace the entry in that CallSiteArray
- Runtime generated methods can be used to call the target method for this CallSite directly
- Next call will then pickup the optimized method
- Inlining becomes partially available
- **No Reflection for method invocation anymore**

CallSite and CallSiteArray

- We still often go through the meta class to invoke methods
- If the user provides a custom meta class we have no other chance than to do that
- Runtime bytecode generation for method call stubs eats up permgen fast
- We rely partially on Unsafe to avoid method verification and other things

CallSite and CallSiteArray

CallSite and CallSiteArray have mostly the same functional potential as MethodHandles do, but we do not make use of the full potential

not yet

Back to stupid Fibonacci

- 6-7x times faster than Groovy 1.0.x/1.5.x
- Java for fib(42) takes about 3.4s
- Groovy 1.6.x/1.7.x for fib(42) takes about 97s (still Java*28)
- If BigInteger is used, Groovy is only about 57% slower

So we encourage people to write such „hot spots“ in Java

Groovy 1.8 (not yet released)

```
public int fib(int n) { //GROOVY 1.8 - fastpath
    if (intDefault.valid) {
        if (n<2) return 1;
        if (intDefault.valid && thisDefault.valid) {
            return fib2(n-1)+fib2(n-2);
        } else {
            goto label;
        }
    } else {//GROOVY 1.6 style fall back - slowpath
        Integer nInt = n;
        CallSite[] cs = csa.array;
        if (ScriptBytecodeAdapter.compareLessThan(nInt, 2)) return 1;
        label: return 1;
        Integer n_1 = (Integer) cs[1].call(nInt,const$1);
        Integer fib_n_1 = (Integer) cs[0].call(this,n_1);
        Integer n_2 = (Integer) cs[1].call(nInt,const$2);
        Integer fib_n_2 = (Integer) cs[0].call(this,n_2);
        Integer ret = (Integer) cs[2].call(fib_n_1,fib_n_2);
        return ret;
    }
}
```

Groovy 1.8 (not yet released)

```
public int fib(int n) { //GROOVY 1.8
     if (intDefault.valid) {
         if (n<2) return n;
         if (intDefault.valid && thisDefault.valid) {
     [...]
}
```

- intDefault and thisDefault are holders for booleans without any synchronization
- They get flagged if the according MetaClass is no longer the default (intDefault for the MetaClass of Integer/int)
- The current thread may or may not pick up the change. If the user wants to be sure, synchronization has to be provided by the user

Groovy 1.8 (not yet released)

The basic idea is to guard each statement by those boolean flag holders, if the expressions inside are optimized

```
int fib(int n) {
     if (n<2) return 1
     return fib(n-1) + fib(n-2)
}
```

- The green parts are optimized to integer math operations
- The red parts are optimized to direct method calls

Groovy 1.8 (not yet released)

- This allows the JIT to work on very „local“ data
- We avoid big concurrent data structures, called for each method invocation
- Many calls become possible without need for an actual MetaClass
- Groovy does often BigDecimal based calculations in the background that can be mapped to double operations in the fastpath

Groovy 1.8 (not yet released)

```
int count(List<List> listOfLists) {
     int size = 0
     listOfLists.each { it?.each { size++ } }
     return size
}
```

- The green parts are optimized to integer math operations
- The red parts are optimized to direct method calls, but we don't know the exact type, so the compiler may has to guess
- Problem 1: the closures here do contain ExpressionStatements, which are to be handled like statements.
- Problem 2: Do we really want all those types?

Expression Statement

```
int count(List<List> listOfLists) { // fastpath only
    int size = 0
    Closure c1 = { List it ->
        Closure c2 = {if (integerDefault.valid {size++} else ...}
        if (it?.getClass()==ArrayList.class
                && arrayListDefault.valid)
        {
            DefaultGroovyMethod.each(listOfLists, c2)
        } else ...
    }
    if (listOfList?.getClass()==ArrayList.class
        && arrayListDefault.valid)
    {
        DefaultGroovyMethod.each(listOfLists, c1)
    } else ...
    return size
}
```

partial static compilation problem

- The already big bytecode gets doubled in size for the same code
- If the type cannot be infered or is not specific enough, we may not be able to optimize
- Statically optimizing multimethods is tough
- A slow compiler is no option if you do runtime compilation
- Calling for example DefaultGroovyMethods#each(Collection,Closure) turns into a problem if a more specific method is given in a new version of Groovy

partial static compilation problem

Still Groovy 1.8 will get an optional partial static compiler to finally make use of „optional typing“

partial static compilation speed

- Java for fib(42) takes about 3.4s
- Groovy 1.8.x prototype for fib(42) takes about 3.8s (12% slower than Java, over a hundred times faster than Groovy 1.0)

So we may no longer encourage people to write such „hot spots“ in Java

Where it is of absolutely no help

- A custom MetaClass on Integer, destroys all integer math optimizations
- We rely on a certain meta class all the time, but for example Grails uses the non default, mutable MetaClass ExpandoMetaClass. So no optimizations here
- Categories (thread scoped method additons to classes) are like setting a custom MetaClass for a short time. So no optimizations here

Groovy 1.8 (not yet released)

Other things to optimize:

- MetaClass requires much memory. Maybe partial MetaClasses are possible (load additional information on demand, rather than upfront)
- Groovy startup time should be reduced (faster MetaClass creation will help here, avoiding to read in DefaultGroovyMethods and creating wrappers for each method too)

Groovy 2.0 (maybe end of 2011)

- Groovy 1.8 is really a testing ground to find out as of what has to be changed to get Groovy fast in most cases, but makes Groovy incompatible to older versions

Possible changes are:

- Change MetaClass to deliver „executables“ instead of having custom MetaClasses
- Those maybe be internally realized by MethodHandles or CallSites

Comment on MethodHandles

- MethodHandles are cool, but only in Java7
- Groovy wants to be fast on Java5 too
- Remi's backport is cool, but will not help us much, since Groovy wants to run in environments where no Agents are allowed

Q/A!?