Exploiting Concurrency with Dynamic Languages

Tobias Ivarsson
Neo Technology
tobias@thobe.org
@thobe on twitter
$ whoami
tobias

> MSc in Computer Science from Linköping University, Sweden
> Jython compiler zealot
> Hacker at Neo Technology
  • Home of the Neo4j graph database
  • Check out http://neo4j.org
> Follow me on twitter: @thobe
  • Low traffic, high tech
> Website: http://www.thobe.org (#include blog)
Exploiting concurrency with dynamic languages

> Background
> The languages
> The benefits of these languages
> Example
> Summary
Background
What Java brought to the table

> Built in, simple support for multi threaded programming

> Synchronized blocks and methods in the language

> `wait()` and `notify()`

> A good, sound memory model for concurrent situations

> Executors, Thread pools, Concurrent collection, Atomic variables, Semaphores, Mutexes, Barriers, Latches, Exchangers, Locks, fine grained timing...
Why use a language other than Java?

> More expressive - put the good stuff to better use
> Higher order constructs abstract away tedious boilerplate
> Closures
Why use a language other than Java?

> Closures provide clean ways of expressing tasks and thread entries

> Meta-programmability can encapsulate conventions
  • Operator overloading
  • Macros and function modifiers

> Capabilities for building DSLs

> Built in ways of dealing with modern problems
  • Support for concurrency best practices?
The Languages
Jython

- I’m biased
- Python is used a lot in scientific computing
  - Could benefit a lot from better concurrency offered by Jython being on the Java platform
- Highly dynamic
- Strict, explicit, but compact and readable syntax
JRuby

> It is a great language implementation
> It is a popular, highly productive language
> Highly dynamic
> Powerful syntax for closures
Scala

> Geared towards scalability in concurrency
> Interesting Actor model

• Provide for scalability by restricting data sharing between concurrently executing code
Clojure

> Built at core for being good at concurrency
> Provides for concurrency by using persistent data structures
  • Most objects are immutable
> Objects that are mutable are mutated explicitly
  • Software transactional memory
“Why not Groovy” (or another language)

> Good and popular dynamic language for the JVM
> Can do anything that Java, Jython or JRuby can
> Not interesting - All examples would be “me too”

> Countless other languages have their specialties, including them all would be too much
  • Ioke, Kawa, Rhino, Java FX script, Kahlua, Fortress...
What these languages add
Automatic resource management - Jython

```python
with synchronization_on(shared_resource):
    data = shared_resource.get_data(key)
    if data is not None:
        data.update_time = current_time()
        data.value = "The new value"
    shared_resource.set_data(key, data)
```
Closures - JRuby

```ruby
shared_resource.synchronized do
  data = shared_resource.get_data(key)
  if data.nil?
    data.update_time = current_time()
    data.value = "The new value"
    shared_resource.set_data key, data
  end
end
end
```
Closures - JRuby

chunk_size = large_array.size[NUM_THREADS]
threads[NUM_THREADS] do |i|
  start = chunk_size * i
  (start..start+chunk_size).each do |j|
    large_array[j] += 1
  end
end
Meta functions - Jython ("function decorators")

@future # start thread, return future ref
def expensive_computation(x, y):
    z = go_do_a_lot_of_stuff(x)
    return z + y

# use the function and the future
future_value = expensive_computation(4, 3)
go_do_other_stuff_until_value_needed()
value = future_value()
Meta functions - Jython

@forkjoin  # fork on call, join on iter
def recursive_iteration(root):
    if root.is_leaf:
        yield computation_on(root)
    left = recursive_iteration(root.left)
    right = recursive_iteration(root.right)
    for node in left:  yield node
    for node in right: yield node
    for node in recursive_iteration(tree): ...
(defn transfer [amount from to]
  (dosync ; isolated, atomic transaction
   (if (>= (ensure from) amount)
     (do (alter from - amount)
         (alter to + amount))
     (throw (new java.lang.RuntimeException "Insufficient funds"))))))

(transfer 1000 your-account my-account)
Transactions as a managed resource - Jython

```python
import neo4j
neo = neo4j.NeoService("/var/neodb")
persons = neo.index("person")
with neo.transaction:
    tobias = neo.node(name="Tobias")
    persons[tobias['name']] = tobias
    emil = neo.node(name="Emil", CEO=True)
    persons[emil['name']] = emil
    tobias.Knows(emil, how="Buddies")
```
Actor Execution - Scala

> Concurrent processes communicating through messages
> Receive and act upon messages
  • Send message
  • Create new actors
  • Terminate
  • Wait for new message
> Message objects are immutable
Actor Execution - Scala

case class IncredibleProblem(d: String)
class SimpleSolution(d: String)
object RichardDeanAnderson extends Actor{
  def act = {
    loop {
      react {
        case IncredibleProblem(d) =>
          reply(SimpleSolution(d))}
    }}
}
Example
Adding two numbers

# Jython
def adder(a,b):
    return a+b

# JRuby
def adder(a,b)
a+b
end
Counting the number of additions - Jython

```python
from threading import Lock
count = 0
lock = Lock()

def adder(a,b):
    global count
    with lock:
        count += 1
    return a+b
```
class Counting
  def adder(a,b)
    @mutex.synchronize { 
      @count = @count + 1
    } 
    a + b
  end
end
Adding two numbers - Clojure

(defn adder [a b]
  (+ a b))

(let [count (ref 0)]
  (defn counting-adder [a b]
    (dosync (alter count + 1))
    (+ a b))
  (defn get-count [] (deref count)))
class Adder {
    def add(a: Int, b: Int) = a + b
}

Actors Adding numbers - Scala
Actors Adding numbers - Scala

class Cntr(var count: Int) extends Actor {
  def act = {
    loop {
      react {
        case Inc => count = count + 1
        case GetCount => reply(count); exit
      }
    }
  }
}
class CountingAdder(times: Int, adder: Adder, counter: Cntr) extends Actor {
    def act = {
        loop {
            react {
                case Add(a,b) => {
                    val start = nanoTime()
                    for (i <- 0 until times) {
                        counter ! Inc
                        adder.add(a,b)
                    }
                    val time = nanoTime() - start
                    react {
                        case GetTime => reply(time / 1000000.0)
                    }
                }
                case Done => exit
            }
        }
    }
}
Performance figures

<table>
<thead>
<tr>
<th>CODE</th>
<th>RĪGA</th>
<th>STRĒLNIEMS</th>
<th>MEŽOTNES LS</th>
</tr>
</thead>
<tbody>
<tr>
<td>04.41</td>
<td>05.11</td>
<td>15.35</td>
<td>08.25</td>
</tr>
<tr>
<td>06.21</td>
<td>E 06.38</td>
<td>E 18.38</td>
<td>E 21.38</td>
</tr>
<tr>
<td>07.51</td>
<td>E 08.08</td>
<td>19.31</td>
<td></td>
</tr>
<tr>
<td>E 09.28</td>
<td>E 09.48</td>
<td>E 17.28</td>
<td></td>
</tr>
<tr>
<td>E 11.28</td>
<td>11.51</td>
<td>E 17.48</td>
<td></td>
</tr>
<tr>
<td>E 12.48</td>
<td>13.11</td>
<td>E 20.28</td>
<td></td>
</tr>
<tr>
<td>14.01</td>
<td>E 14.18</td>
<td>14.31</td>
<td></td>
</tr>
<tr>
<td>E 15.38</td>
<td>15.51</td>
<td>E 16.08</td>
<td></td>
</tr>
<tr>
<td>E 16.58</td>
<td>17.11</td>
<td>E 16.28</td>
<td></td>
</tr>
<tr>
<td>18.11</td>
<td>E 18.38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table likely contains performance figures or codes, possibly related to locations or codes, with specific numerical values.
Execution times (ms for 400000 additions)

- Jython
- JRuby
- Clojure
Execution times (ms for 400000 additions)

- Jython (Lock)
- Jython (AtomicInteger)
- JRuby (Mutex)
- Clojure (STM)

Without counter
Execution times (ms for 400000 additions)

- Mutex (JRuby)
- STM (Clojure)
- Actors (Scala)
Jython

> Nice resource management syntax
> Implementation has a few things left to work out
> Suffer from high mutability in core datastructures
JRuby

- Closures help a lot
- Great implementation
- Suffer from high mutability in core data structures
Scala

> Actors have good characteristics
  • Decouple work tasks with no shared state
  • Asynchronous communication is great
>
> Syntax is more on the verbosity level of Java
Clojure

> Immutability frees you from thinking about synchronization
> Transactions for mutation enforces thinking about state
> The combination enforces best practices, but in a way much different from “plain old Java”
Tobias Ivarsson

http://twitter.com/thobe

http://Neo4j.org