The GPars Quick Reference

The Whole GPars Team <gpars-developers@googlegroups.com>

Version 2.0, 2017-02-01

Table of Contents

To download this document as a PDF - click here

Actor

Concepts

Actors are independent isolated active objects, which mutually share no data and communicate solely by messages passing. Avoiding shared mutable state relieves developers from many typical concurrency problems, like live-locks or race conditions. The body (code) of each **actor** gets executed by a random thread from a thread pool and so **actors** can proceed concurrently and independently. Since **Actors** can share a relatively small thread pool, they avoid the threading limitations of the JVM and don't require excessive system resources even in cases when your application consists of thousands of **actors**.

Actors typically perform three basic types of operations on top of their usual tasks:

- Create a new **actor**
- Send a message to another **actor**
- Receive a message

Actors can be created as subclasses of an particular **actor** class or using a factory method supplying the **actor**'s body as a closure parameter. There are various ways to send a message, either using the >> operator or any of the *send()* , *sendAndWait()* or *sendAndContinua()* methods.

Receiving a message can be performed either in a blocking or a non-blocking way, when the physical thread is returned to the pool until a message is available.

Actors can be orchestrated into various sorts of algorithms, potentially leveraging architectural patterns similar to those known from the enterprise messaging systems.

Lifecycle

Creating an Actor Using Factory Methods

Creating an **Actor**

```
Actors.actor {
    println "actor1 has started"
    delegate.metaClass {
        afterStop = {List undeliveredMessages ->
            println "actor1 has stopped"
        }
        onInterrupt = {InterruptedException e ->
            println "actor1 has been interrupted"
        }
        onTimeout = {->
            println "actor1 has timed out"
        }
        onException = {Exception e ->
            println "actor1 threw an exception"
        }
    }
    println("Running actor1")
    ...
}
```

Sub-classing the DefaultActor class

Sub-class an **Actor**

```
class PooledLifeCycleSampleActor extends DefaultActor {
    protected void act() {
        println("Running actor2")
        ...
    }
    private void afterStart() {
        println "actor2 has started"
    }
    private void afterStop(List undeliveredMessages) {
        println "actor2 has stopped"
    }
    private void onInterrupt(InterruptedException e) {
        println "actor2 has been interrupted"
    }
    private void onTimeout() {
        println "actor2 has timed out"
    }
    private void onException(Exception e) {
        println "actor2 threw an exception"
    }
}
```

Usage

Creating an Actor Using a Factory Method

An **Actor** *from The Factory*

```
import static groovyx.gpars.actor.Actors.actor

def console = actor {
    loop {
        react {
            println it
        }
    }
    ...
}
```

Sub-classing the DefaultActor class

Sub-class a **DefaultActor**

```
class CustomActor extends DefaultActor {
    @Override protected void act() {
        loop {
            react {
                println it
            }
        }
    }
}

def console=new CustomActor()
console.start()
```

Sending Messages

Messages for **Actors**

```
console.send('Message')
console << 'Message'
console.sendAndContinue 'Message', {reply -> println "I received reply: $reply"}
console.sendAndWait 'Message'
```

Timeouts

How To Handle Timing Issues

```
import static groovyx.gpars.actor.Actors.actor

def me = actor {
    friend.send('Hi')
    react(30.seconds) {msg ->
        if (msg == Actor.TIMEOUT) {
            friend.send('I see, busy as usual. Never mind.')
            stop()
        } else {
            //continue conversation
        }
    }
}
me.join()
```

When a timeout expires when waiting for a message, the *Actor.TIMEOUT* message arrives instead. Also the *onTimeout()* handler is invoked, if present on the **actor**:

What Happens When an Actor Times-out

```
import static groovyx.gpars.actor.Actors.actor

def me = actor {
    delegate.metaClass.onTimeout = {->
        friend.send('I see, busy as usual. Never mind.')
        stop()
    }
    friend.send('Hi')
    react(30.seconds) {
        // Continue conversation.
    }
}
me.join()
```

Actor Groups

A Group of **Actors** *Is Called What ?*

```
def coreActors = new NonDaemonPGroup(5)  //5 non-daemon threads pool
def helperActors = new DefaultPGroup(1)  //1 daemon thread pool
def priceCalculator = coreActors.actor {
...
}
def paymentProcessor = coreActors.actor {
...
}
def emailNotifier = helperActors.actor {
...
}
def cleanupActor = helperActors.actor {
...
}

// Increase size of the core actor group.
coreActors.resize 6

// Shutdown the group's pool once you no longer need the group to release resources.
helperActors.shutdown()
```

DynamicDispatchActor

Dynamic Dispatch

```
final Actor actor = new DynamicDispatchActor({
    when {String msg -> println 'A String'; reply 'Thanks'}
    when {Double msg -> println 'A Double'; reply 'Thanks'}
    when {msg -> println 'A something ...'; reply 'What was that?'}
})
actor.start()
```

Reactor

When **Actors** *React*

```
import groovyx.gpars.actor.Actors

final def doubler = Actors.reactor {
    2 * it
}.start()
```

Agent

Concepts

In the *Clojure* programing language you can find a concept of **Agents**, which essentially behave like actors accepting code (functions) as messages. After reception, the received function is run against the internal state of the **Agent** and the return value of the function is considered to be the new internal state of the **Agent**. Essentially, **agents** safe-guard mutable values by allowing only a single *agent-managed thread* to make modifications to them. The mutable values are *not directly accessible* from outside, but instead requests have to be sent to the **agent** and the **agent** guarantees to process the requests sequentially on behalf of the callers. **Agents** guarantee sequential execution of all requests and so consistency of the values.

Usage

Agent Implements a Clojure-like Agent Concept

An **Agent** *Example*

```
import groovyx.gpars.agent.Agent

def jugMembers = new Agent<List>(['Me'])  // Add Me.
jugMembers.send {it.add 'James'}  // Add James.

final Thread t1 = Thread.start{
    jugMembers {it.add 'Jo'}  // Add Jo --- using the implicit call() method to send
the function.
}

final Thread t2 = Thread.start{
    jugMembers << {it.add 'Dave'}  // Add Dave.
    jugMembers << {it.add 'Alice'}  // Add Alice.
}

[t1, t2]*.join()

println jugMembers.val
jugMembers.valAsync {println "Current members: $it"}
System.in.read()
jugMembers.stop()
```

Communicating Seq. Procs

Concepts

The **CSP** (*Communicating Sequential Processes*) concurrency concept provides a message-passing model with synchronous rendezvous-type communication.

It is valued mainly for its high level of determinism and the ability to compose parallel processes.

GPars *GroovyCSP* wraps the JCSP library from The University of Canterbury and builds on the work of *Jon Kerridge*. Please review his works here.

For more information about the **CSP** concurrency model, checkout the **CSP** section of the User Guide or refer to the links below:

- **CSP** definition : Wiki **CSP**
- Google's **Go** programming language with **CSP**-style concurrency : Go with Google

---

Usage

GroovyCSP

Take a look at this example of the Groovy API for CSP-style concurrency :

How Do You Code This One ?

```
import groovyx.gpars.csp.PAR

import org.jcsp.lang.CSProcess
import org.jcsp.lang.Channel
import org.jcsp.lang.ChannelOutput
import org.jcsp.lang.One2OneChannel

import groovyx.gpars.csp.plugAndPlay.GPrefix
import groovyx.gpars.csp.plugAndPlay.GPCopy
import groovyx.gpars.csp.plugAndPlay.GPairs
import groovyx.gpars.csp.plugAndPlay.GPrint

class FibonacciV2Process implements CSProcess {
    ChannelOutput outChannel

    void run() {
        One2OneChannel a = Channel.createOne2One()
        One2OneChannel b = Channel.createOne2One()
        One2OneChannel c = Channel.createOne2One()
        One2OneChannel d = Channel.createOne2One()
        new PAR([
            new GPrefix(prefixValue: 0, inChannel: d.in(), outChannel: a.out()),
            new GPrefix(prefixValue: 1, inChannel: c.in(), outChannel: d.out()),
            new GPCopy(inChannel: a.in(), outChannel0: b.out(), outChannel1:
outChannel),
            new GPairs(inChannel: b.in(), outChannel: c.out()),
        ]).run()
    }
}

One2OneChannel N2P = Channel.createOne2One()

new PAR([
    new FibonacciV2Process(outChannel: N2P.out()),
    new GPrint(inChannel: N2P.in(), heading: "Fibonacci Numbers")
]).run()
```

Dataflow Concurrency

Concepts

Dataflow Concurrency offers an alternative concurrency model, which is inherently safe and robust. It puts an emphasis on the data and their flow though your processes instead of the actual processes that manipulate the data. **Dataflow** algorithms relieve developers from dealing with live-locks, race-conditions and make dead-locks deterministic and thus 100% reproducible. If you don't get dead-locks in tests you won't get them in production.

Dataflow Variable

A single-assignment multi-read variable offering a thread-safe data-exchange among threads.

Dataflows Class

A virtual infinite map of **Dataflow Variables** with on-demand creation policy.

Dataflow Stream

A thread-safe unbound deterministic blocking stream with a **Dataflow Variable**-compatible interface.

Dataflow Queue

A thread-safe unbound blocking queue with a **Dataflow Variable**-compatible interface.

Dataflow Task

A lightweight thread of execution, which gets assigned a physical thread from a thread pool to execute the body of the task. Tasks should typically exchange data using **Dataflow Variables* and *Streams**.

Dataflow Operator

A cornerstone of a more thorough *dataflow concurrency algorithm*. Such algorithms typically define a number of operators and connect them with channels, represented by *Dataflow Streams, Queues* or *Variables*.

Each operator specifies its input and output channels to communicate with other operators. Repeatedly, whenever all input channels of a particular operator contain data, the operator's body is executed and the produced output is sent into the output channels.

Usage

Dataflow Variables

A Sample

```
import static groovyx.gpars.dataflow.Dataflow.task

final def x = new DataflowVariable()
final def y = new DataflowVariable()
final def z = new DataflowVariable()
task{
    z << x.val + y.val
    println "Result: ${z.val}"
}

task{
    x << 10
}

task{
    y << 5
}
```

Dataflows

A Sample

```
import static groovyx.gpars.dataflow.Dataflow.task
final def df = new Dataflows()

task{
    df.z = df.x + df.y
    println "Result: ${df.z}"
}

task{
    df.x = 10
}

task{
    df.y = 5
}
```

Dataflow Queues

A Sample

```
import static groovyx.gpars.dataflow.Dataflow.task

def words = ['Groovy', 'fantastic', 'concurrency', 'fun', 'enjoy', 'safe', 'GPars',
'data', 'flow']
final def buffer = new DataflowQueue()

task{
    for (word in words) {
        buffer << word.toUpperCase()  // Add to the buffer.
    }
}

task{
    while(true) println buffer.val  // Read from the buffer in a loop.
}
```

Bind Handlers

A Sample

```
def a = new DataflowVariable()
a >> {println "The variable has just been bound to $it"}

a.whenBound{println "Just to confirm that the variable has been really set to $it"}
```

Dataflow Operators

A Sample

```
operator(inputs: [a, b, c], outputs: [d]) {x, y, z ->
    ...
    bindOutput 0, x + y + z
}
```

Fork/Join

Concepts

Fork/Join, or *Divide and Conquer*, is a very powerful abstraction to solve hierarchical problems. When talking about hierarchical problems, think about quick sort, merge sort, file system or general tree navigation and such.

- **Fork / Join** algorithms essentially split a problem at hands into several smaller sub-problems and recursively apply the same algorithm to each of the sub-problems.
- Once the sub-problem is small enough, it is solved directly.
- The solutions of all sub-problems are combined to solve their parent problem, which in turn helps solve its own parent problem.

Usage

Using the Fork-Join Builder

Feel free to experiment with the number of fork/join threads in the pool

A Sample

```
withPool(1){pool ->

    println """Number of files: ${

        runForkJoin(new File("./src")) {file ->
            long count = 0

            file.eachFile {
                if (it.isDirectory()) {
                    println "Forking a child task for $it"
                    // Fork a child task.
                    forkOffChild(it)
                } else {
                    count++
                }
            }

            // Use results of children tasks to calculate and store own result.
            return count + (childrenResults.sum(0))
        }
    }"""
}
```

Extending the *AbstractForkJoinWorker* class

A Sample

```
public final class FileCounter extends AbstractForkJoinWorker<Long> {
    private final File file;

    def FileCounter(final File file) {
        this.file = file
    }

    protected void compute() {
        long count = 0;
        file.eachFile{
            if (it.isDirectory()) {
                println "Forking a thread for $it"
                // Fork a child task.
                forkOffChild(new FileCounter(it))
            }
            else {
                count++
            }
        }

        // Use results of children tasks to calculate and store own result.
        setResult(count + ((childrenResults)?.sum() ?: 0))
    }
}

withPool(1){pool ->  // Feel free to experiment with the number of fork/join threads
in the pool.
    println "Number of files: ${orchestrate(new FileCounter(new File("..")))}"
}
```

Fork/Join Pool

Concepts

Dealing with data frequently involves manipulating collections. Lists, arrays, sets, maps, iterators, strings and lot of other data types can be viewed as collections of items. The common pattern to process such collections is to take elements sequentially, one-by-one, and make an action for each of the items in row.

Take, for example, the min() function, which is supposed to return the smallest element of a collection. When you call the min() method on a collection of numbers, the caller thread will create an accumulator or so-far-the-smallest-value initialized to the minimum value of the given type, let say to zero. And then the thread will iterate through the elements of the collection and compare them with the value in the accumulator. Once all elements have been processed, the minimum value is stored in the accumulator.

This algorithm effectively wastes 75% of the computing power of the chip

This algorithm, however simple, is totally wrong on multi-core hardware. Running the *min()* function on a dual-core chip can leverage at most 50% of the computing power of the chip. On a quad-core it would be only 25%. Correct, this algorithm effectively wastes 75% of the computing power of the chip.

Tree-like Structures Should Be Used Here

Tree-like structures proved to be more appropriate for parallel processing. The *min()* function in our example doesn't need to iterate through all the elements in row and compare their values with the accumulator. What it can do, instead, is relying on the multi-core nature of your hardware.

A *parallel_min()* function could, for example, compare pairs (or tuples of certain size) of neighboring values in the collection and promote the smallest value from the tuple into a next round of comparison.

Searching for minimum in different tuples can safely happen in parallel and so tuples in the same round can be processed by different cores at the same time without races or contention among threads.

Usage

Parallel Collection Processing

The following methods are currently supported on all objects in **Groovy**:

- eachParallel()
- eachWithIndexParallel()

- collectParallel()
- findAllParallel()
- findParallel()
- everyParallel()
- anyParallel()
- grepParallel()
- groupByParallel()
- foldParallel()
- minParallel()
- maxParallel()
- sumParallel()

Summarize Numbers Concurrently with This Sample

```
ForkJoinPool.withPool{
    final AtomicInteger result = new AtomicInteger(0)
    [1, 2, 3, 4, 5].eachParallel{result.addAndGet(it)}
    assert 15 == result
}

// Multiply numbers asynchronously.
ForkJoinPool.withPool{
    final List result = [1, 2, 3, 4, 5].collectParallel{it * 2}
    assert ([2, 4, 6, 8, 10].equals(result))
}
```

Meta-class Enhancer

A Sample

```
import groovyx.gpars.ParallelEnhancer

def list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

ParallelEnhancer.enhanceInstance(list)

println list.collectParallel{it * 2 }
```

Transparently Parallel Collections

This selectImportantNames() method Processes the Name Collections Concurrently.

```
ForkJoinPool.withPool{

    assert ['ALICE', 'JASON'] == selectImportantNames(['Joe', 'Alice', 'Dave', 'Jason
'].makeConcurrent())
}

/**
 * A function implemented using standard sequential collect() and findAll() methods.
 */
def selectImportantNames(names) {
    names.collect{it.toUpperCase()}.findAll{it.size() > 4}
}
```

Map/Reduce

Available methods:

- map()
- reduce()
- filter()
- size()
- sum()
- min()
- max()

The *collection* property will return all elements wrapped in a Groovy collection instance.

A Sample

```
println 'Number of occurrences of the word GROOVY today: ' + urls.parallel
        .map{it.toURL().text.toUpperCase()}
        .filter{it.contains('GROOVY')}
        .map{it.split()}
        .map{it.findAll{word -> word.contains 'GROOVY'}.size()}
        .sum()
```

Software Trans. Memory

Concepts

Software Transactional Memory or **STM**, gives developers transactional semantics for accessing in-memory data. When multiple threads share data in memory, by marking blocks of code as transactional (atomic) the developer delegates the responsibility for data consistency to the Stm engine. **GPars** leverages the Multiverse STM engine.

Atomic Closures

GPars allows developers to structure their concurrent code into atomic blocks (closures), which are then performed as single units, preserving the transactional ACI (*Atomicity, Consistency, Isolation*) attributes.

Usage

Running a Piece of Code Atomically

An Atomic Sample

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.references.TxnInteger

import static org.multiverse.api.StmUtils.newTxnInteger

public class Account {
    private final TxnInteger amount = newTxnInteger(0);

    public void transfer(final int a) {
        GParsStm.atomic {
            amount.increment(a);
        }
    }

    public int getCurrentAmount() {
        GParsStm.atomicWithInt {
            amount.get();
        }
    }
}
```

Customizing the Transactional Properties

A Sample

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.AtomicBlock
import org.multiverse.api.PropagationLevel

final TxnExecutor block = GParsStm.createTxnExecutor(maxRetries: 3000, familyName:
'Custom', PropagationLevel: PropagationLevel.Requires, interruptible: false)

assert GParsStm.atomicWithBoolean(block) {
    true
}
```

ThreadPool

Concepts

On multi-core systems, you can benefit from having some tasks run asynchronously in the background, and so off-load your main thread of execution. The *ThreadPool* class allows you to easily start tasks in the background to be performed asynchronously and collect the results later.

Usage

Use of ThreadPool - the Java Executors' Based Concurrent Collection Processor

Closures Enhancements

A Sample

```
GParsExecutorsPool.withPool() {
    Closure longLastingCalculation = {calculate()}

    // Create a new closure, which starts the original closure on a thread pool.
    Closure fastCalculation = longLastingCalculation.async()

    // Returns almost immediately.
    Future result=fastCalculation()

    // Do stuff while calculation performs...
    println result.get()
}
```

Another Sample

```
GParsExecutorsPool.withPool() {
    /**
     * The callAsync() method is an asynchronous variant of the default call() method
     * to invoke a closure. It will return a Future for the result value.
     */
    assert 6 == {it * 2}.call(3).get()
    assert 6 == {it * 2}.callAsync(3).get()
}
```

Executor Service Enhancements

A Sample

```
GParsExecutorsPool.withPool {ExecutorService executorService ->
    executorService << {println 'Inside parallel task'}
}
```

Asynchronous Function Processing

A Sample

```
GParsExecutorsPool.withPool {

    // Waits for results.
    assert [10, 20] == AsyncInvokerUtil.doInParallel({calculateA()}, {calculateB()})

    // Returns a Future and doesn't wait for results to be calculated.
    assert [10, 20] == AsyncInvokerUtil.executeAsync({calculateA()}, {calculateB()})
*.get()
}
```