Guide To Agents

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

The **Agent** class is a thread-safe non-blocking shared mutable state wrapper implementation inspired by **Agents** in **Clojure**.

Shared Mutable State can’t be avoided

A lot of the concurrency problems disappear when you eliminate the need for *Shared Mutable State* with your architecture. Indeed, concepts like **actors**, **CSP** or **dataflow concurrency** avoid or isolate mutable state completely.

In some cases, however, sharing mutable data is either inevitable or makes the design more natural and understandable. For example, think of a shopping cart in a typical e-commerce application, when multiple [blue]AJAX requests may hit the cart with read or write requests concurrently.

Introduction

In the **Clojure** programing language, you can find a concept of **Agents**, the purpose of which is to protect mutable data that need to be shared across threads. **Agents** hide the data and protect it from direct access. Clients can only send commands (functions) to the **agent**. The commands will be serialized and processed against the data one-by-one in turn.

With the commands being executed serially, the commands do not need to care about concurrency and can assume the data is all theirs when run. Although implemented differently, **GPars Agents**, called *Agent* , fundamentally behave like **actors**. They accept messages and process them asynchronously. The messages, however, must be commands (functions or **Groovy** closures) and will be executed inside the **agent**. 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.

Schematically -

```
agent = new Agent(0)  //created a new Agent wrapping an integer with initial value 0
agent.send {increment()}  //asynchronous send operation, sending the increment() function
...

//after some delay to process the message the internal Agent's state has been updated
...

assert agent.val== 1
```

To wrap integers, we can certainly use AtomicXXX types on the **Java** platform, but when the state is a more complex object we need more support.

Concepts

GPars provides an **Agent** class, which is a special-purpose, thread-safe, non-blocking implementation inspired by **Agents** in **Clojure**.

An **Agent** wraps a reference to mutable state, held inside a single field, and accepts code (closures or commands) as messages, which can be sent to the **Agent** just like to any other actor using the '<<' operator, the *send()* methods or the *implicit call()* method.

At some point after reception of a closure / command, the closure is invoked against the internal mutable field and can make changes to it. The closure is guaranteed to be run without intervention from other threads and so may freely alter the internal state of the **Agent** held in the internal *data* field.

The whole update process is of the `fire-and-forget` type since, once the message (closure) is sent to the Agent, the caller thread can go off to do other things and come back later to check the current value with **Agent.val** or **Agent.valAsync(closure)**.

Basic Rules

- When executed, the submitted commands obtain the **agent**'s state as a parameter.
- The submitted commands /closures can call any methods on the *agent8's state.
- Replacing the state object with a new one is also possible and is done using the **updateValue()** method.
- The *return value* of the submitted closure doesn't have a special meaning and is ignored.
- If the message sent to an **Agent** is `not a closure`, it is considered to be a new value for the internal reference field.
- The *val* property of an **Agent** will wait until all preceding commands in the agent's queue are consumed and then safely return the value of the **Agent**.
- The *valAsync()* method will do the same **without** blocking the caller.
- The *instantVal* property will return an immediate snapshot of the internal **agent**'s state.
- All **Agent** instances share a default daemon thread pool. Setting the *threadPool* property of an **Agent** instance will allow it to use a different thread pool.
- Exceptions thrown by the commands can be collected using the *errors* property.

Examples

Shared List of Members

The **Agent** wraps a list of members, who have been added to the club. To add a new member, a message (command to add a member) has to be sent to the *clubMembers* Agent.

A Sample -

```
import groovyx.gpars.agent.Agent import
java.util.concurrent.ExecutorService import java.util.concurrent.Executors

/**
 * Create a new Agent wrapping a list of strings
 */
def clubMembers = new Agent<List<String>>(['Me'])  //add Me

clubMembers.send {it.add 'James'}  //add James

final Thread t1 = Thread.start {
    clubMembers.send {it.add 'Joe'}  //add Joe
}

final Thread t2 = Thread.start {
    clubMembers << {it.add 'Dave'}  //add Dave
    clubMembers {it.add 'Alice'}    //add Alice (using the implicit call() method)
}

[t1, t2]*.join()
println clubMembers.val
clubMembers.valAsync {println "Current members: $it"}

clubMembers.await()
```

Shared Conference Counting Number of Registrations

The **Conference** class allows registration and un-registration, however these methods can only be called from the commands sent to the *conference* **Agent**.

A Conference Sample -

```
import groovyx.gpars.agent.Agent

/**
 * Conference stores number of registrations and allows parties to register and
unregister.
 * It inherits from the Agent class and adds the register() and unregister() private
methods,
 * which callers may use it the commands they submit to the Conference.
 */
class Conference extends Agent<Long> {
    def Conference() { super(0) }
    private def register(long num) { data += num }
    private def unregister(long num) { data -= num }
}

final Agent conference = new Conference()  //new Conference created

/**
 * Three external parties will try to register/unregister concurrently
 */

final Thread t1 = Thread.start {
    conference << {register(10L)}              //send a command to register 10 attendees
}

final Thread t2 = Thread.start {
    conference << {register(5L)}               //send a command to register 5 attendees
}

final Thread t3 = Thread.start {
    conference << {unregister(3L)}             //send a command to unregister 3
attendees
}

[t1, t2, t3]*.join()

assert 12L == conference.val
```

Factory Methods

Agent instances can also be created using the *Agent.agent()* factory method.

A Sample to Make an Agent Instance

```
def clubMembers = Agent.agent ['Me']  //add Me
```

Listeners and Validators

Agents allow the user to add listeners and validators. While listeners are notified each time the internal state changes, validators get a chance to reject or veto a coming change by throwing an exception.

A Concrete Example -

```
final Agent counter = new Agent()

counter.addListener {oldValue, newValue -> println "Changing value from $oldValue to
$newValue"}
counter.addListener {agent, oldValue, newValue -> println "Agent $agent changing value
from $oldValue to $newValue"}

counter.addValidator {oldValue, newValue -> if (oldValue > newValue) throw new
IllegalArgumentException('Things can only go up in Groovy')}
counter.addValidator {agent, oldValue, newValue -> if (oldValue == newValue) throw new
IllegalArgumentException('Things never stay the same for $agent')}

counter 10
counter 11
counter {updateValue 12}
counter 10  //Will be rejected

counter {updateValue it - 1}  //Will be rejected
counter {updateValue it}  //Will be rejected
counter {updateValue 11}  //Will be rejected
counter 12  //Will be rejected

counter 20
counter.await()
```

Both listeners and validators are essentially closures taking two or three arguments. Exceptions thrown from the validators will be logged inside the **agent** and can be tested using the *hasErrors()* method or retrieved through the *errors* property.

Testing for Errors Sample

```
assert counter.hasErrors()
assert counter.errors.size() == 5
```

Validator Gotchas

Groovy is not very strict on variable data types and immutability, so **agent** users should be aware of potential bumps on the road.

If the submitted code modifies the state directly, validators will not be able to un-do the change in case of a validation rule violation. There are two possible solutions available:

- Make sure you never change the supplied object representing current agent state
- Use custom copy strategy on the agent to allow the agent to create copies of the internal state

In both cases you need to call *updateValue()* to set and validate the new state properly.

The problem as well as both of the solutions follows :

A Validator Sample -

```
//Create an agent storing names, rejecting 'Joe'
final Closure rejectJoeValidator = {oldValue, newValue -> if ('Joe' in newValue) throw
new IllegalArgumentException('Joe is not allowed to enter our list.')}

Agent agent = new Agent([])
agent.addValidator rejectJoeValidator

agent {it << 'Dave'}                    //Accepted
agent {it << 'Joe'}                     //Erroneously accepted, since by-passes the
validation mechanism
println agent.val

//Solution 1 - never alter the supplied state object
agent = new Agent([])
agent.addValidator rejectJoeValidator

agent {updateValue(['Dave', * it])}      //Accepted
agent {updateValue(['Joe', * it])}       //Rejected
println agent.val

//Solution 2 - use custom copy strategy on the agent
agent = new Agent([], {it.clone()})
agent.addValidator rejectJoeValidator

agent {updateValue it << 'Dave'}        //Accepted
agent {updateValue it << 'Joe'}         //Rejected, since 'it' is now just a copy of the
internal agent's state
println agent.val
```

Grouping

By default, all **Agent** instances belong to the same group sharing its daemon thread pool.

Custom groups can also create instances of **Agent**. These instances will belong to the group, which created them, and will share a thread pool. To create an **Agent** instance belonging to a group, call the *agent()* factory method on the group. This way you can organize and tune performance of agents.

Create Groups Around a Thread Pools

```
final def group = new NonDaemonPGroup(5)  //create a group around a thread pool
def clubMembers = group.agent(['Me'])  //add Me
```

Custom Thread Pools for Agents

The default thread pool for **agents** contains daemon threads. Make sure that your custom thread pools either use daemon threads, too, which can be achieved either by using **DefaultPGroup** or by providing your own thread factory to a *thread pool constructor*.

Alterntively, in case your thread pools use non-daemon threads, such as when using the **NonDaemonPGroup** group class, make sure you shutdown the group or the thread pool explicitly by calling its `shutdown()` method, otherwise your applications will [red]never exit.

Direct Pool Replacement

Alternatively, by calling the *attachToThreadPool()* method on an **Agent** instance, a custom thread pool can be specified for it.

attachToThreadPool() *Example*

```
def clubMembers = new Agent<List<String>>(['Me'])  //add Me

final ExecutorService pool = Executors.newFixedThreadPool(10)
clubMembers.attachToThreadPool(new DefaultPool(pool))
```

Remember, like **actors**, a single **Agent** instance (aka agent) can never use more than one thread at a time

The Shopping Cart Example

A Sample -

```
import groovyx.gpars.agent.Agent

class ShoppingCart {
    private def cartState = new Agent([:])
//----------------- public methods below here ---------------------------------
    public void addItem(String product, int quantity) {
        cartState << {it[product] = quantity}  //the << operator sends
                                               //a message to the Agent
    }    public void removeItem(String product) {
        cartState << {it.remove(product)}
    }    public Object listContent() {
        return cartState.val
    }    public void clearItems() {
        cartState << performClear
    }

    public void increaseQuantity(String product, int quantityChange) {
        cartState << this.&changeQuantity.curry(product, quantityChange)
    }
//----------------- private methods below here --------------------------------
    private void changeQuantity(String product, int quantityChange, Map items) {
        items[product] = (items[product] ?: 0) + quantityChange
    }    private Closure performClear = { it.clear() }
}
//----------------- script code below here ------------------------------------
final ShoppingCart cart = new ShoppingCart()
cart.addItem 'Pilsner', 10
cart.addItem 'Budweisser', 5
cart.addItem 'Staropramen', 20

cart.removeItem 'Budweisser'
cart.addItem 'Budweisser', 15

println "Contents ${cart.listContent()}"

cart.increaseQuantity 'Budweisser', 3
println "Contents ${cart.listContent()}"

cart.clearItems()
println "Contents ${cart.listContent()}"
```

You might have noticed two implementation strategies in the code.

1. Public methods may internally just send the required code off to the **Agent**, instead of executing the same functionality directly

And so Typically Sequential Code Like This

```
public void addItem(String product, int quantity) {
    cartState[product]=quantity
}
```

Becomes

```
public void addItem(String product, int quantity) {
    cartState << {it[product] = quantity}
}
```

1. Public methods may send references to internal private methods or closures, which hold the desired functionality to perform the deed.

A Public-to-Private Sample

```
public void clearItems() {
    cartState << performClear
}

private Closure performClear = { it.clear() }
```

Currying might be necessary, if the closure takes other arguments besides the current internal state instance. See the *increaseQuantity* method.

The Printer Rervice Example

Another example, suppose a not thread-safe printer service is shared by multiple threads. The printer needs to have the document and quality properties set before printing, so obviously we have a potential for race conditions if not guarded properly. Callers don't want to block until the printer is available, which the `fire-and-forget` nature that **actors** solve very elegantly.

A Sample Printer Service

```
import groovyx.gpars.agent.Agent

/**
 * A non-thread-safe service that slowly prints documents on at a time
 */
class PrinterService {
    String document
```

```
    String quality

    public void printDocument() {
        println "Printing $document in $quality quality"
        Thread.sleep 5000
        println "Done printing $document"
    }
}

def printer = new Agent<PrinterService>(new PrinterService())

final Thread thread1 = Thread.start {
    for (num in (1..3)) {
        final String text = "document $num"
        printer << {printerService ->
            printerService.document = text
            printerService.quality = 'High'
            printerService.printDocument()
        }
        Thread.sleep 200
    }
    println 'Thread 1 is ready to do something else. All print tasks have been submitted'
}

final Thread thread2 = Thread.start {
    for (num in (1..4)) {
        final String text = "picture $num"
        printer << {printerService ->
            printerService.document = text
            printerService.quality = 'Medium'
            printerService.printDocument()
        }
        Thread.sleep 500
    }
    println 'Thread 2 is ready to do something else. All print tasks have been submitted'
}

[thread1, thread2]*.join()
printer.await()
```

For the latest updates, see the respective Demos

Reading The Value

To follow the **Clojure** philosophy closely, the **Agent** class gives reads higher priority than to writes. By using the *instantVal* property, your read request will bypass the incoming message queue of the **Agent** and returns the current snapshot of the internal state. The *val* property will wait in the message queue for processing, just like the non-blocking variant *valAsync(Clojure cl)* , which will invoke the provided closure with the internal state as a parameter.

You have to bear in mind that the *instantVal* property might return although correct, but randomly looking results, since the internal state of the **Agent** at the time of *instantVal* execution is non-deterministic and depends on the messages that have been processed before the thread scheduler executes the body of *instantVal* .

The *await()* method lets you wait for the processing of all the messages submitted to the **Agent** before and so may block the calling thread.

State Copy Strategy

To avoid leaking the internal state, the **Agent** class can specify a `copy strategy` as the second constructor argument. With the `copy strategy` specified, the internal state is processed by the `copy strategy` closure and the output value of the `copy strategy` value is returned to the caller instead of the actual internal state. This applies to *instantVal*, *val* as well as to *valAsync()* .

Error Handling

Exceptions thrown from within the submitted commands are stored inside the **agent** and can be obtained from the *errors* property. The property gets cleared once read.

A Sample of Error Handling

```
def clubMembers = new Agent<List>()
assert clubMembers.errors.empty

    clubMembers.send {throw new IllegalStateException('test1')}
    clubMembers.send {throw new IllegalArgumentException('test2')}
    clubMembers.await()

    List errors = clubMembers.errors
    assert 2 == errors.size()
    assert errors[0] instanceof IllegalStateException
    assert 'test1' == errors[0].message
    assert errors[1] instanceof IllegalArgumentException
    assert 'test2' == errors[1].message

    assert clubMembers.errors.empty
```

Fair and Non-fair Agents

Agents can be either fair or non-fair. Fair **agents** give up the thread after processing each message, unfair **agents** keep a thread until their message queue is empty. As a result, non-fair **agents** tend to perform better than fair ones.

The default setting for all **Agent** instances is to be **non-fair**, however by calling its *makeFair()* method the instance can be made fair.

A Sample To Make It Fair

```
def clubMembers = new Agent<List>(['Me'])  //add Me
clubMembers.makeFair()
```