Concurrency in Clojure and Go

Stephen Adams

May 3rd, 2012

Clojure

The Clojure programming language was released in 2007 by the software developer Rich Hickey. Clojure was designed with four features in mind:

Clojure

The Clojure programming language was released in 2007 by the software developer Rich Hickey. Clojure was designed with four features in mind:

The Four Features of Clojure

- A Lisp
- Functional programming
- Symbiosis with an established platform (Java)
- Designed for concurrency

Go

Go was first made available in 2009 by Google Inc. after two years of in house development.
“Provide the efficiency of a statically-typed compiled language with the ease of programming of a dynamic language"

Go

Go was first made available in 2009 by Google Inc. after two years of in house development.
“Provide the efficiency of a statically-typed compiled language with the ease of programming of a dynamic language"

Other Goals for Go

- Type and memory safety
- Good concurrency support
- Efficient and latency-free garbage collection
- High-speed compilation

Go is a systems language designed in 2012

Hello World

```
package main
import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

Go's type system

```
x := 12
```

Go's type system

```
x := 12

var x int
x = 12
```

Go's type system

```
x := 12

var x int
x = 12

//These all fail!
var x
x = 12
```

Intro to Clojure Concurrency

“Clojure’s main tenet isn’t the facilitation of concurrency. Instead, Clojure at its core is concerned with the sane management of state, and facilitating concurrent programming naturally falls out of that."

-From The Joy of Clojure

Software Transactional Memory

Analogous to database transactions for controlling shared memory.

STM Transaction Properties

- Atomic - Multiple updates in a transaction appear to happen simultaneously outside of the transaction
- Consistent - If a ref's validation function fails the entire transaction will fail
- Isolated - Transactions cannot see other currently running transactions

Software Transactional Memory

Databases are also typically durable, since Clojure transactions are in-memory durability is not provided. These four properties are known as ACID. Databases provide ACID, Clojure provides ACI.

Four reference types

	Ref	Agent	Atom	Var
Coordinated	✓			
Asynchronous		✓		
Retriable	✓		✓	
Thread-local				✓

Reference type features

- coordinated - Reads and writes to refs happen without race conditions
- asynchronous - Changes are queued to happen in another thread and run at a later time
- retriable - Work done updating a ref's value is speculative and may have to be done again
- thread-local - Changes in state are isolated to a single thread

When to use Refs

Do you need:

- To coordinate multiple updates at once (Transactions)
- Synchronize multiple transactions

When to use Atoms

Do you need:

- Apply updates regardless of anything else in the world (Uncoordinated)
- Still synchronize multiple transactions

When to use Atoms

Do you need:

- Apply updates regardless of anything else in the world (Uncoordinated)
- Still synchronize multiple transactions

Atoms are not wrapped in transactions this is what makes them uncoordinated.

When to use Agents

Do you need:

- To run a job that is independent of other currently running tasks

Tasks sent to an agent are placed in a queue to be run later on a thread from a thread pool.

What are vars?

Vars are thread-local symbol to value bindings
Vars are created when you use the def, defn, and other similar macros

Goroutines

"A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space. It is lightweight, costing little more that the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required."

From Effective Go

Starting new Goroutines

```
package main
import (
    "fmt"
    "runtime"
)

func say(s string) {
    for i := 0; i < 5; i++ {
        fmt.Println(s)
        runtime.Gosched()
    }
}
```

Starting new Goroutines

```
func main() {
    go say("world")
    say("hello")
}
```

Starting new Goroutines

```
hello
world
hello
world
hello
world
hello
world
hello
world
```

Channels

Channel creation:

```
testChannel := make(chan string)
```

Channels

Channel creation:

```
testChannel := make(chan string)
```

Sending and receiving from a channel

```
testString := "test"
\\Sending to a channel
testChannel <- testString
\\Receiving from a channel
recieveValue := <- testChannel
```

Concurrent Fibonacci

```
package main
import ("fmt")
func fibonacci(c chan int) {
        x, y := 1, 1
        n := cap(c)
        for i := 0; i < n; i++ {
                c <- x
                x, y = y, x + y
        }
        close(c)
}
```

Concurrent Fibonacci

```
func main() {
        c := make(chan int, 10)
    go fibonacci(c)
        for i := range c {
                fmt.Println(i)
        }
}
```

Conclusion

These two languages approach the concurrency problem differently:

- Clojure manages state
- Go automates locking and waiting, and carefully protects scope

Which language is for me?

Don’t choose between these two languages for their concurrency features.
These languages are very dissimilar and choose concurrency control that fits their other design goals.

References

- Fogus, M., and Houser, C. The Joy of Clojure, Manning Publications.
- Halloway S. Programming Clojure, Pragmatic Bookshelf