Easy Concurrency from Sequential Functions

John Scott

Unix/Database Consultant
Founder, SetSpace, Inc, 1998-now
Founder, august.{com,net}, Inc, 1998-2006

Presented at Viverae
Dallas, Texas, USA
June 21, 2016

@dreid

Programming computers is an exercise in constantly making things whose complexity just slightly exceeds your own comprehension.

Go Concurrency Patterns: Pipelines and cancellation

blog.golang.org/pipelines (https://blog.golang.org/pipelines)

What is a Sequential Process?

```
package main
import "fmt"

func fib(i int) int {

    if i <= 2 {
        return 1
    }

    return fib(i - 1) + fib(i - 2)
}

func main() {

    fmt.Println(fib(43))

}
```

Run

Single Fibonacci Call

Sequential Processing is Slow!

```
package main
import "fmt"

func fib(i int) int {

    if i <= 2 {
        return 1
    }

    return fib(i - 1) + fib(i - 2)
}

func main() {

    fmt.Println(

        fib(43),   fib(43),   fib(43),   fib(43),
        fib(43),   fib(43),   fib(43),   fib(43),

    )
}
```

Run

Eight Fibonacci Calls, Sequentially

Concurrent Calls to Fibonacci Much Quicker

```
package main
import . "fmt"

func fib(i int) int {
    if i <= 2 {
        return 1
    }
    return fib(i - 1) + fib(i - 2)
}

func main() {

    out := make(chan int)
    fibber := func(i int) {
        out <- fib(i)
    }

    go fibber(43);   go fibber(43);   go fibber(43);   go fibber(43);
    go fibber(43);   go fibber(43);   go fibber(43);   go fibber(43);

    Println(<-out,  <-out,  <-out,  <-out,  <-out,  <-out,  <-out,  <-out)
}
```

Run

Eight Fibonacci Calls, Concurrently

Flow Oriented Computing

- receive values from upstream via inbound channels
- perform some function on that data, usually producing new values
- send values downstream via outbound channels

Build a Fibonacci Flow "Op"

```
func fib_flow(in chan int) (out chan int) {

    out = make(chan int)

    go func() {                         //  Listen in Background on 'in' channel
        for i := range in {
            out <- fib(i)
        }
        close(out)
    }()

    return out
}

func main() {

    in := make(chan int)
    out := fib_flow(in)

    in <- 43
    fmt.Println(<- out)
}
```

Run

Build a Square Flow "Op"

```
func sqr_flow(in chan int) (out chan int) {

    out = make(chan int)

    go func() {                    //  Listen in Background on 'in' channel
        for i := range in {
            out <- i * i
        }
        close(out)
    }()

    return out
}

func main() {

    in := make(chan int)
    out := sqr_flow(in)

    in <- 43
    fmt.Println(<- out)
}
```

Run

Compose Flow Operations

- data in pipeline, not on heap or stack
- scheduling based on flow not time sliced
- pipelines can be connected into *Directed Acyclic Graph* (DAG)
- flows (with no side effects) always terminate!

Easy to Compose New Flow Operations

```
func sqr_fib_flow(in chan int) (out chan int) {

    return sqr_flow(fib_flow(in))
}

func main() {

    in := make(chan int)

    //  Compose twice
    out := sqr_fib_flow(sqr_fib_flow(in))

    in <- 5
    fmt.Println(<- out)

}
```

Run

In/Out Channels Same Data Type!

Merge Many Channels into One

```
func merge(in ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)

    io := func(c <-chan int) {    //
        for n := range c {
            out <- n
        }
        wg.Done()            // End of input so decrement semaphore counter
    }
    wg.Add(len(in))

    for _, c := range in {        //  Start listeners
        go io(c)
    }

    go func() {                //  Wait for all listeners to finish
        wg.Wait()
        close(out)
    }()

    return out
}
```

Flow 8 Requests to 4 Fibbers

```
func pump(iv ...int) (chan int) {

    out := make(chan int)

    go func() {
        for _, n := range iv {
            out <- n
        }
        close(out)
    }()
    return out
}

func main() {

    in := pump(43, 43, 43, 43, 43, 43, 43, 43)
    out := merge(sqr_fib_flow(in), sqr_fib_flow(in), sqr_fib_flow(in), sqr_fib_flow(in))

    for n := range out {
        fmt.Println(n)
    }
}
```

Run

Thank you

John Scott
jmscott@setspace.com (mailto:jmscott@setspace.com)
https://github.com/jmscott (https://github.com/jmscott)

Unix/Database Consultant
Founder, SetSpace, Inc, 1998-now
Founder, august.{com,net}, Inc, 1998-2006

Presented at Viverae
Dallas, Texas, USA
June 21, 2016