

Stream Processing with Go Channels

...

Vladimir Vivien
March 30 2016

Overview

About Me

Motivation

Pipeline Patterns

Automi

The Project

About Me

About Me

Work for EMC {code}

Recovering Java Developer

Go Programmer 1.5 yrs

Lover of Tech

twitter/@vladimirvivien

medium/@vladimirvivien

Motivation

Motivation

Pipeline Patterns

Sameer Ajmani, 2014

<https://blog.golang.org/pipelines>

Pipeline Patterns

Pipeline Patterns in Go

Divide a process into stages (or operators)

Each stage is connected via channels

Channels serve as conduit for data elements

An operation is applied to element at each stage

A Simple Pipeline:
ingest() -> process() -> store()

Stage 1

Ingest some data

```
func ingest() <-chan []string {  
    out := make(chan []string)  
    go func() {  
        out <- []string{"aaaa", "bbb"}  
        out <- []string{"ccccc", "dddddd"}  
        out <- []string{"e", "ffffff", "g"}  
        close(out)  
    }()  
    return out  
}
```

Stage 2

Process the data

```
func process(concurrency int, in <-chan []string) <-
chan int {
    var wg sync.WaitGroup
    wg.Add(concurrency)
    out := make(chan int)
    work := func() {
        for data := range in {
            for _, word := range data {
                out <- len(word)
            }
        }
        wg.Done()
    }
    go func() {
        for i := 0; i < concurrency; i++ {
            go work()
        }
    }()
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

Stage 3

Some persistent step

```
func store(in <-chan int) <-chan struct{} {  
    done := make(chan struct{})  
    go func() {  
        defer close(done)  
        for data := range in {  
            fmt.Println(data)  
        }  
    }()  
    return done  
}
```

Driver

Brings it all together

```
func main() {  
    // stage 1 ingest data from source  
    in := ingest()  
  
    // stage 2 - process data  
    reduced := process(4, in)  
  
    // stage 3 - store  
    <-store(reduced)  
}
```

Works, but

Noisy

Business mixed with channel
communication primitives

Synchronization becomes complex as
number of stages grow

Observations

Go pipelines patterns is a form of stream processing of data, a well-understood subject

Data flows through an in-memory conduit

Apply operations on data elements as it flows

Higher level of abstraction will help

Automi

Automi

API for stream processing

Uses pipeline patterns internally

Hides channel communication and
synchronization primitives

Lazy builder fluent API

Example

Abbreviated code snippet

```
type scientist struct {...}
in := src.New().WithFile("./data.txt")
out := snk.New().WithFile("./result.txt")

stream := stream.New().From(in)
stream.Map(func(cs []string) scientist {
    yr, _ := strconv.Atoi(cs[3])
    return scientist{
        FirstName: cs[1],
        LastName:  cs[0],
        Title:     cs[2],
        BornYear:  yr,
    }
})
stream.Filter(func(cs scientist) bool {
    if cs.BornYear > 1930 {
        return true
    }
    return false
})
stream.Map(func(cs scientist) []string {

    return
    []string{cs.FirstName, cs.LastName, cs.Title}
})
stream.To(out)
<-stream.Open() // wait for completion
```

Set Source/Ingest

Sets a CSV source which emits
slices []string for each row

```
//----- data.txt -----//  
Lovelace, Ada, Mathematician, 1815  
Turing, Alan, Computer Scientist, 1912  
Knuth, Donald, Mathematician, 1938  
Berners-Lee, Tim, Computer Scientist, 1955  
Von Neumann, John, Mathematician, 1903  
...
```

```
//-----//
```

```
// Create source for new stream  
in := src.New().WithFile("./data.txt")  
stream := stream.New().From(in)
```

Stage 1

Map incoming slice from source
to struct scientist

```
type scientist struct {  
    FirstName string  
    LastName  string  
    Title     string  
    BornYear  int  
}  
...  
  
// map slices to struct scientist  
stream.Map(func(cs []string) scientist {  
    yr, _ := strconv.Atoi(cs[3])  
    return scientist{  
        FirstName: cs[1],  
        LastName:  cs[0],  
        Title:     cs[2],  
        BornYear:  yr,  
    }  
})
```

Stage 2

Filters out scientist where
scientist.BornYear > 1930

```
// Returns true when BornYear > 1039
stream.Filter(func(cs scientist) bool {
    if cs.BornYear > 1930 {
        return true
    }
    return false
})
```

Stage 3

Map each scientist element back
to a slice []string

```
// Prep scientist for next stage
stream.Map(func(cs scientist) []string {
    return []string{
        cs.FirstName,
        cs.LastName, cs.Title,
    }
})
```

Set Sink/Persist

Write slice []string stream items
to CSV file

```
stream.To(out)  
out := snk.New().WithFile("./result.txt")
```

Open Stream

Execute the stream and wait for completion/handle error

```
<-stream.Open() // wait for completion
```

Built in Stream Operators

Comes with several built-in operators

Process() *

Filter()

Map()

FlatMap()

Reduce() *

GroupBy()

More to come...

The Project

The Automi Project

Star it on GitHub

Apache 2 License

github.com/vladimirvivien/automi

API getting more stable

Release 0.1 any day now

Future Operators

Targeting a rich set of operators

Support both batch- and stream-style

Simplify API wherever possible

Inspirations from projects such as
Spark Streaming and Apache Flink

Future Sources & Sinks

SocketSource Network socket stream source

SocketSink Network socket stream sink

CsvSource: Source for CSV files

CsvSink: Sink component

HttpSource: Sources stream data from http

HttpSink: Sink for posting data via http

DbSource: Database source for streaming data items

DbSink: A sink component for streaming data

And more...

More Sources and Sinks

Distributed FS (HDFS, S3, etc)

Cassandra

Messaging systems

Logging systems

Whatever source/sink users find useful

Thank You

Automi - <https://github.com/vladimirvivien/automi>

Article - <https://blog.gopheracademy.com/advent-2015/automi-stream-processing-over-go-channels/>

Vladimir Vivien - [twitter/@vladimirvivien](https://twitter.com/vladimirvivien)