Go Academy

#6 Concurrency vs. Parallelism, gRPC and protobuf, sql and lexers

Concurrency

- Is about dealing with a lot of things at once (scheduling multiple things)
- In go we should strive to create multiple smaller go routines
- And have work split in smaller chunks that is able to communicate with each other

Parallelism

- Is about doing lots of things at the same time
- Concurrency enables parallelism
- Multiple executors can handle multiple tasks at the same time
- But ultimately depends on number of executors and code

Visualising concurrency (@source)

```
package main

import "time"

func timer(d time.Duration) <-chan int {
    c := make(chan int)
    go func() {
        time.Sleep(d)
        c <- 1
    }()
    return c
}

func main() {
    for i := 0; i < 24; i++ {
        c := timer(1 * time.Second)
        <-c
    }
}
```

Visualising concurrency (server, logging)

```
package main

import "net"

func handler(c net.Conn) {
    c.Write([]byte("ok"))
    c.Close()
}

func main() {
    l, err := net.Listen("tcp", ":5000")
    if err != nil {
        panic(err)
    }

    for {
        c, err := l.Accept()
        if err != nil {
            continue
        }
        go handler(c)
    }
}
```

Additional reading

- More about visualising concurrency
  - https://github.com/divan/gotrace
  - Video https://www.youtube.com/watch?v=KyuFeiG3Y60
- Rob Pike’s presentation
  - https://www.youtube.com/watch?v=cN_DpYBzKso

Protobuf

- A method for serialising structured data
- Well supported in many languages
- Small, fast and simple
- Special package declaration syntax
- Compiles into language specific package
- Helps with future compatibility

Protobuf

```
syntax = "proto3";

message Book {
    message Author {
        string firstName = 1;
        string lastName = 2;
    }

    string title = 1;
    repeated string authors = 2;
    int32 yearOfRelease = 3;
}
```

Protobuf (`01-protobuf`)

- Install `protoc` from https://github.com/google/protobuf/releases

  ```
  brew install protobuf
  ```

- Install protobuf package

  ```
  go get -u github.com/golang/protobuf/protoc-gen-go
  ```

- Compile it

  ```
  protoc --go_out=. 01-protobuf/pb/book.proto
  ```

- Use it `book := &pb.Book{…}`

gRPC

- Google's interpretation of RPC
- Uses protobuf as IDL and message format
- Supports extensive customisation (`DialOption`, `ServerOption`)
- Supports streams (server-side, client-side, bidirectional)

gRPC (02-grpc)

- Install grpc package

```
go get -u google.golang.org/grpc
```

- Define a service

```
service Catalog {
    rpc Search(SearchRequest) returns (SearchResponse);
    rpc RecentPurchases(Empty) returns (stream Book);
}
```

- Rebuild it

```
protoc --go_out=plugins=grpc:. 02-grpc/pb/book.proto
```

SQL (`03-mysql`, `04-sqlite`)

- Golang provides a common interface for SQL databases, the `sql` package
- Many drivers are available / supported (check here)
- Provides:
  - Generic API for all supported databases
  - Takes care of type conversion
  - Possible portability (if only basic SQL is used)

Lexing in go

- Inspired by Rob Pike's talk from 2011 and package `text/template`
- Quick exploration how go can help us solve rather complex problems elegantly in go
- Let's try to apply code from `text/template` to your homework
- And make something like this:

```
> 1 + 2
$1 = 3
> (2 +
Error> (2 +^: Failed to parse calculation
```

Lexing in go

- A single calculation consists of separate tokens (lex items)
- For each of those we need a type, value and position
  - types: number, plus sign, left bracket, space etc.
- How to extract them?
  - Existing tools?
  - Regular expressions?

Lexing in go

- Let's try to build a state machine
- While progressing through our input, state functions we'll be returning new state making it much easier to carry data with it

```
func (state) state
```

- Keep it as simple as possible

Lexing in go (parsing)

- Series of tokens has no value on its own
- Parser reads tokens and puts them in a more execution friendly format
- `"1", "+", "2" -> Calculation(["1", Operation("+", "2)])`
- Executing operations one after another is made much easier again with state functions

Lexing in go (errors)

- One of the goals was to create detailed and unified error messages
  - `Error> 2 +^+ 2: Failed to parse calculation; err = Unexpected operator reached`
  - Without repeating this display logic over and over
- Custom struct (`parsingError`) holds failing token, input string and message
  - `Error()` function satisfies the `error` interface