Go Advanced

Go Advanced

Day 1

1. Idiomatic Go
2. Methods and Interfaces
3. Advanced Functions and Methods
4. CSP and Goroutines

Day 2

5. Channels
6. Context
7. Testing
8. Performance

Prerequisites:

- CLI experience
- Basics programming and networking knowledge

Administrative Info

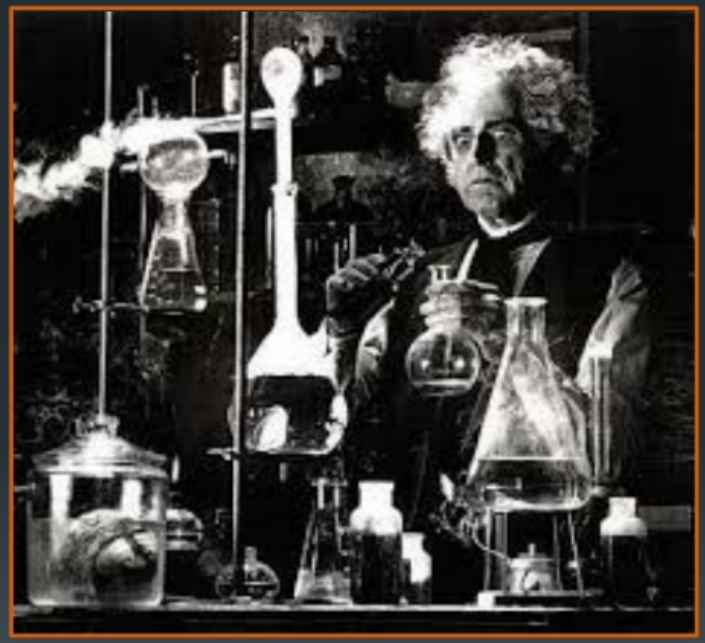
- Length: 8 Modules
- Format: Lecture/Labs/Discussion
- Attendees: Name/Role/Experience/Goals for the Course

- Instructor
- John Kidd
  - john.kidd@kiddcorp.com

Lecture and Lab

- Our Goals in this class are two fold:

1. Introduce concepts and ecosystems
   - Covering concepts and where things fit in the world is the primary purpose of the lecture/discussion sessions
   - The instructor will take you on a tour of the museum
     - Like a museum tour, you should interact with the instructor (tour guide), ask questions, discuss
     - Like a museum tour, you will not have time to read the slides during the tour, instead, the instructor will discuss and point out the highlights of the slides (exhibits) which will be waiting for you to read in depth later should you like to dig deeper
2. Impart practical experience
   - This is the primary purpose of the labs
   - Classes rarely have time for complete real world projects so think of the labs as thought experiments
     - Like hands on exhibits at the museum



Day 1

1. Idiomatic Go
2. Methods and Interfaces
3. Advanced Functions and Methods
4. CSP and Goroutines

Idiomatic Go

Objectives

- Understand the meaning of idiomatic Go
- Examine some idiomatic uses of Go
- Through style, tooling, and thought

Idiomatic Go

- What is Go?
- Go is an open source programming language that makes it easy to build **simple** , **reliable** , and **efficient** software
- What is idiomatic Go?
- The way Go is "spoken" by a native speaker
- Twists of phrase that communicate elegantly but only to those with a non-superficial understanding of the language

- What about dialects? (e.g. New York versus Louisiana)
- Some Go idioms may not hold in all communities/organizations
- Adopt those idioms you find valuable and dispose of the others
- Go is a (sort of) young language and idioms are still evolving
- Sample Idiomatic Go :
- Composing programs principally from functions
- Sample Anti-Idiomatic Go :
- Composing programs from structs with methods, emulating class based languages like Java and C++
- Throughout this and follow on modules we will be looking at various idiomatic approaches to Go coding

Go Program Style

- Indentation, spacing, and other surface-level details of code are automatically standardized by the gofmt or goimport tools
- golint does additional style checks automatically
- Tools and libraries distributed with Go suggest standard approaches to things like:
- API documentation (go doc)
- testing (go test)
- building (go build)
- package management (go get)
- Go enforces rules that are recommendations in other languages
- ***BANNING!***
  - cyclic dependencies
  - unused variables or imports
  - implicit type conversions
- The omission of functional-programming shortcuts like map() and Java-style try/finally blocks tends to encourage an explicit, concrete and imperative programming style

Idiomatic Go

- Happy path is aligned on the left
  - You can quickly scan the left side of

the function body to see what is expected to happen

  - Error handling is indented
  - Don't nest the happy path!
  - Make the happy return the last statement

```
kubernetes % git remote -v
origin          https://github.com/kubernetes/kubernetes.git (fetch)
origin          https://github.com/kubernetes/kubernetes.git (push)

kubernetes % grep -A 40 "Config return a controller manager config objective"
// Config return a controller manager config objective
func (s KubeControllerManagerOptions) Config(allControllers []string, disabled
        if err := s.Validate(allControllers, disabledByDefaultControll
                return nil, err
        }

        if err := s.SecureServing.MaybeDefaultWithSelfSignedCerts("loc
        }               return nil, fmt.Errorf("error creating self-s


        kubeconfig, err := clientcmd.BuildConfigFromFlags(s.Master, s
        if err != nil {
                return nil, err
        }
        kubeconfig.DisableCompression = true
        kubeconfig.ContentConfig.AcceptContentTypes = s.Generic.Client
        kubeconfig.ContentConfig.ContentType = s.Generic.ClientConnect
        kubeconfig.QPS = s.Generic.ClientConnection.QPS
        kubeconfig.Burst = int(s.Generic.ClientConnection.Burst)

        client, err := clientset.NewForConfig(restclient.AddUserAgent(
        if err != nil {
        }               return nil, err


        eventRecorder := createRecorder(client, KubeControllerManagerU

        c := &kubecontrollerconfig.Config{
                        Client:          client,
                        Kubeconfig:      kubeconfig,
                        EventRecorder:   eventRecorder,
        }
        if err := s.ApplyTo(c); err != nil {
                return nil, err
        }
        s.Metrics.Apply()

        s.Logs.Apply()

        return c, nil
}
kubernetes %
```

Idiomatic Go – Some Specifics

- Running gofmt (or goimport) will take care of most style issues
- Single spaces between sentences (even text editors agree!)
- Ex. “Notice the single space. Got it?” over “Here is a double space. Bad form!”
  - https://go-review.googlesource.com/c/go/+/20022/
- Comments for people reading the code always have a single space after the slashes

```
// hi, this is ok
 //                              and this is not, where is the comment
```

- Use singular form for repo/folder names
- E.g. ‘project’ not ‘projects’
- When checking for an empty string (What is ‘s’ data type?)
- use s == “” rather than len(s) == 0

Ugly

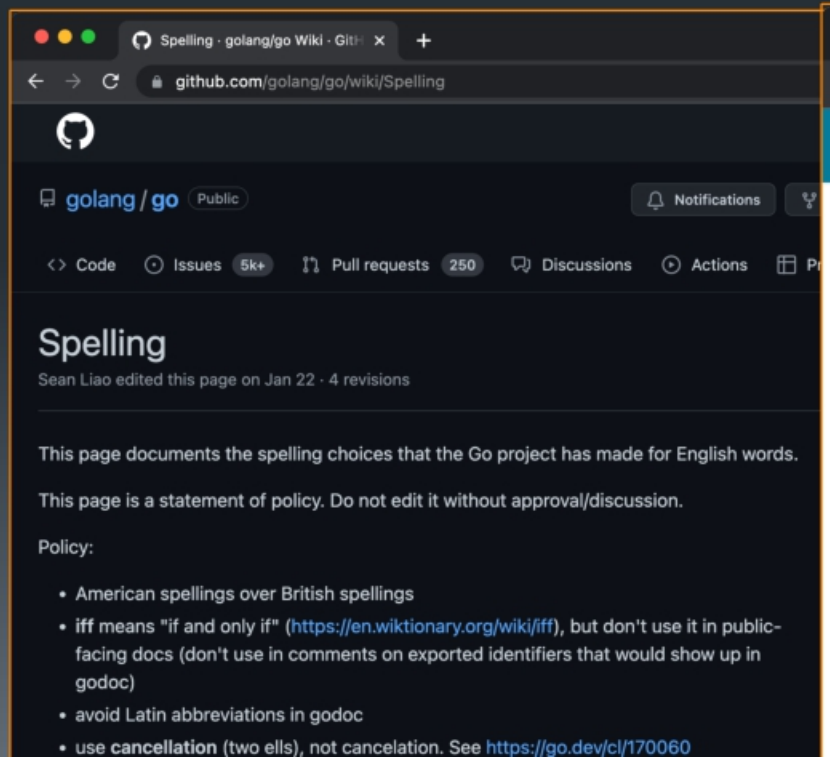
```
% cat formatMe.go
package main

import "fmt"
var a []int
func main(){
a = make([]int,10)
fmt.Println("my bad formatting",a[0:len(a)] ) }
```

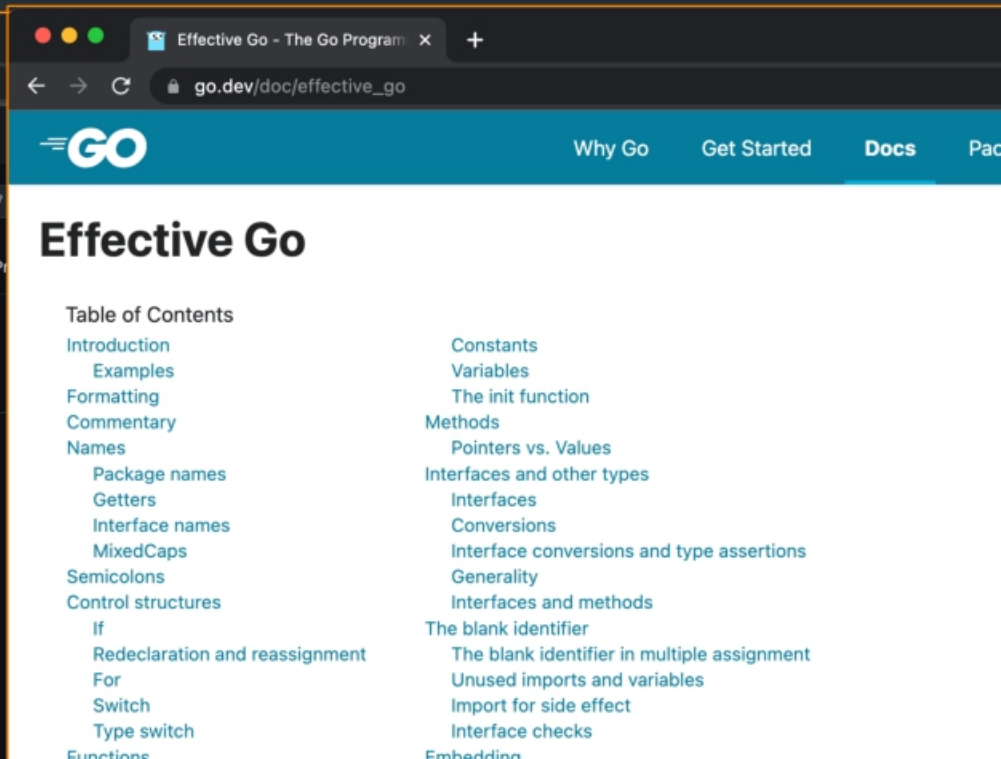
Beautiful

```
% gofmt -r "a[0:len(a)] -> a[0:]" formatMe.go
package main

import "fmt"

var a []int

func main() {
        a = make([]int, 10)
        fmt.Println("my bad formatting", a[0:])
}
```

General formatting considerations

- Consistent spelling of words
- Ex. canceling over cancelling (American over British)
  - https://github.com/golang/go/wiki/Spelling
- Camel-casing
- ex. var MyExportVar … (Export use UPPER camel)
- ex. var myInternallyCoolVar … (Not-exported use LOWER camel)
- Read more specifics here https://go.dev/doc/effective_go





Function Style

- Lines of code packed together within function blocks (e.g. no blank lines in a function body)
- The idea is that if you need multiple "blank line" delimited blocks of code, perhaps you actually just need multiple functions
- Not all agree to this style pattern
- Multiple return arguments used
- Single return functions are the exception
  - Usually single return functions return an error
- The last return argument in a multi return function is usually and error interface called err
- Don't write the smallest number of lines of code, write the clearest number of lines of code

```
func (c *Client) Post(url string, contentType string, body io.Reader) (resp *Response, err error) {
        req, err := NewRequest("POST", url, body)
        if err != nil {
                return nil, err
        }
        req.Header.Set("Content-Type", contentType)
        return c.Do(req)
}
```

General coding considerations

- Use singular names in collections
- Do this
  - github.com/golang/example/hello
  - github.com/golang/example/outyet
  - Not this
  - github.com/golang/examples /hello
  - github.com/golang/examples /outyet

- Error variable naming
  - Declare as “type MyError struct {…} ”
  - Error values should be named
    var ErrBadlogic = errors.New(‘something bad’)

- Use defer (if readability is maintained)

- Avoid unused receivers

```
func (r receiver) func1 {
                    // r unused
}
```

```
func (receiver) func1 {
                    // r unused
}
```

- Mutex hats

```
type X struct {
    quotasMu sync.Mutex // mutex hat, placed on top of data it protects
    quotas [client_names]quota
    smallestAvailbleQuota quota

    added int // place unprotected fields a blank line below
}
```

```
func() {
  for {
    row, err := db.Query("SELECT ...")
    if err != nil {
      ..
    }
    defer row.Close()
  }
}
```

bad

```
func() {
  for {
    row, err := db.Query("SELECT ...")
    if err != nil {
      ..
    }
    row.Close()
  }
}
```

ok

```
func() {
  for {
    func() {
      row, err := db.Query("SELECT ...")
      if row != nil { defer row.Close() }
      if err != nil { ..; return } good
    }()
  }
}
```

Interface contracts, embedding and generality

- Contracts - interfaces are implemented implicitly
- Ex. no extends or implements keywords (i.e. no source syntax requirement)

- Embedding - "prefer composition over inheritance" from GOF (has-a vs is-a)
- Embedding interfaces in interfaces
- Embedding types in structs
- Implementing interfaces provides polymorphism

- Generality - when only the interface behavior matters do not export value type itself

```
1  package main
2
3  import "fmt"
4
5  type Fooer interface{ Foo() string }
6  type Barer interface{ Bar() string }
7
8  type FooBarer interface {
9          Fooer
10         Barer
11 }
12
13 type F struct{}
14
15 func (f F) Foo() string { return "foo" }
16
17 type B struct{}
18
19 func (b B) Bar() string { return "bar" }
20
21 type FB struct {
22         F
23         B
24 }
25
26 func (fb FB) Foo() string { return "foo" }
27 func (fb FB) Bar() string { return "bar" }
28 func (fb FB) FooBar() string { return fb.Foo() + fb.Bar() }
29
30 func f1(f Fooer) { fmt.Printf("%T, %v\n", f, f.Foo()) }
31 func f2(b Barer) { fmt.Printf("%T, %v\n", b, b.Bar()) }
32
33 func f3(fb FooBarer) { fmt.Printf("%T, %v\n", fb, fb.Foo()+fb.Bar())}
34
35 func main() {
36         f := F{}
37         b := B{}
38         fb := FB{}
39         f1(f)
40         f2(b)
41         f3(fb)
42         f1(fb)
43 }
```

interfaces

structs

functions

```
% go run code.go
main.F, foo
main.B, bar
main.FB, foobar
main.FB, foo
```

```
1  package something
2
3  import "fmt"
4
5  type I interface { M() }
6
7  type s struct { }
8  func (s) M() { fmt.Println("got called") }
9
10 func New() I {
11         var i I = s{}
12         return i
13 }
```

```
1  package main
2
3  import "example.com/what/something"
4
5  func main() { something.New().M() }
% go run driver.go
got called
```

Panic/Recover

- Exceptions are not a feature of the Go language
- The exception-like panic feature was initially omitted
- panic/recover was ultimately added in Go circa 2010
- The Go authors advise using panic for:
  - Unrecoverable errors such as those that should halt an entire program
  - For halting an entire server request (with recover)
  - To propagate errors up the stack within a package
  - But not across package boundaries
  - Error returns are the standard API means for propagating errors

```
1   package main
2
3   import "fmt"
4
5   func main() {
6       f()
7       fmt.Println("Returned normally from f")
8   }
9
10  func f() {
11      defer func() {
12          if r := recover(); r != nil {
13              fmt.Println("Recovered in f", r)
14          }
15      }()
16      fmt.Println("Calling g.")
17      g(0)
18      fmt.Println("Returned normally from g")
19  }
20
21  func g(i int) {
22      if i > 3 {
23          fmt.Println("Panicking!")
24          panic(fmt.Sprintf("%v", i))
25      }
26      defer fmt.Println("Defer in g", i)
27      fmt.Println("Printing in g", i)
28      g(i + 1)
29  }
% go run panic.go
Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f
```

Best practices (over the years)

- Per Reddit “Top 11 Golang Best Practices For 2020”
  - Use of HTML Templats (help deal with cross-site scripting/xss)
  - Validate input (similar to above?)
  - gofmt
  - Avoid nesting by handling errors first
  - Avoid SQL injection (HTMLEscapeString, parameterized queries, etc.)
  - Do not capitalize error strings
  - Don’t discard variable with _ (check if valid or not) and avoid panic (return error)
  - DRY
  - Use type switch if unsure of object and method set
  - import . (allows circular dependency testing)
  - Important code goes first
- https://www.reddit.com/r/golang/comments/jn74k7/top_11_golang_best_practices_for_2020/

- Per Francesc Campoy Flores (Go advocate):
  - Avoid nesting by handling errors first (flatten – return early)
  - Avoid repetition when possible (dry)
  - Important code goes first
  - Document your code
  - Shorter is better (early return)
  - Packages with multiple files (if code is looooong)
  - Make your packages "go get"-able
  - Ask for what you need (use interfaces over specific types)
  - Keep independent packages independent
  - Avoid concurrency in your API
  - Use goroutines to manage state
  - Avoid goroutine leaks (use of buffered channels, exit chan)
- https://talks.golang.org/2013/bestpractices.slide#1

...more best practices...

- Per Dave Cheney (Go advocate):
  - Guiding principles
  - Simplicity – “Simplicity is prerequisite for reliability.”
  - Readability – “Readability is essential for maintainability.”
  - Productivity – “Design is the art of arranging code to work today, and be changeable forever.”
  - Identifiers
  - Choose identifiers for clarity, not brevity
  - Identifier length
  - Context is key
  - Don’t name your variables for their types
  - Use a consistent naming style
  - Use a consistent declaration style
  - Be a team player
  - Comments
  - Comments on variables and constants should describe their contents not their purpose
  - Always document public symbols
  - Package Design
  - A good package starts with its name
  - Avoid package names like base, common, or util
  - Return early rather than nesting deeply
  - Make the zero value useful
  - Avoid package level state
  - Project Structure
  - Consider fewer, larger packages
  - Keep package main small as small as possible
  - API Design
  - Design APIs that are hard to misuse
  - Design APIs for their default use case
  - Let functions define the behaviour they requires
  - Error handling
  - Eliminate error handling by eliminating errors
  - Only handle an error once
  - Concurrency
  - Keep yourself busy or do the work yourself
  - Leave concurrency to the caller
  - Never start a goroutine without knowning when it will stop
- https://dave.cheney.net/practical-go/presentations/qcon-china.html

Deliberate omissions

- Go deliberately omits certain features common in other languages
- The designers added only those facilities that all three agreed on
- These omissions guide users away from the features and their semantics
- Features omitted
- Assertions (argued against explicitly: https://golang.org/doc/faq#assertions)
- Pointer arithmetic (explicitly: https://golang.org/doc/faq#no_pointer_arithmetic)
- Implementation Inheritance
- Unions (tagged/untagged)
- Implicit type conversions
- ~~Generic programming~~ (mostly in 1.18)
- Exceptions
- In place of type inheritance users are encouraged to use:
- Interfaces to achieve dynamic dispatch
- Composition to reuse code (has-a)
- Composition and delegation can be automated by struct embedding
- Struct embedding is a feature with many of the drawbacks of inheritance (programmers should avoid overuse for the same reasons inheritance overuse is considered problematic in other languages)
  - Affects the public interface of objects
  - Not fine-grained (i.e, no method-level control over embedding)
  - Methods of embedded objects cannot be hidden
- Achieves polymorphism with composition of interfaces

Programming language values

- Values define actions
- Understanding the values of a programming language and its community can help you understand how code is and should be written
- How libraries and 3rd part software are likely to work
- C++ and Rust
- A programmer should not have to pay for a feature they do not use
- Java and C#
- Everything is an object
- Programs driven by message passing, information hiding, and polymorphism
- Go
- Explicit
- Concrete
- Imperative

Tools to help

- go fmt – *reformats* Go code (comes with Go)
- goimports – gofmt + *import* checks
- go vet – concerned with *correctness* (integrated with Go)
- golint – reports coding *style* errors
- godoc – generate documentation (integrated with Go)

- More tools at https://pkg.go.dev/golang.org/x/tools/cmd

… and there are more techniques!

- Comment Sentences
- Contexts
- Copying
- Crypto Rand
- Declaring Empty Slices
- Doc Comments
- Don't Panic
- Error Strings
- Examples
- Goroutine Lifetimes
- Handle Errors
- Imports
- Import Blank
- Import Dot
- In-Band Errors
- Indent Error Flow
- Initialisms
- Interfaces
- Line Length
- Mixed Caps
- Named Result Parameters
- Naked Returns
- Package Comments
- Package Names
- Pass Values
- Receiver Names
- Receiver Type
- Synchronous Functions
- Useful Test Failures
- Variable Names

Beyond Effective Go blog - https://github.com/golang/go/wiki/CodeReviewComments

Use Modules

- As of Feb-2021 we can finally move to using modules
- Tools still are asked to provide backward compatibility

Summary

- Go offers simple features with great flexibility
- Interfaces use structural typing allowing pre-existing structs to be passed through new interfaces
- Adapters can be used to allow functions and slices to be passed as interfaces
- Anonymous functions can be declared ad hoc anywhere a function of that type is valid
- Closures allow anonymous functions to capture variables from outer scopes

- Functions are first class citizens in Go and can be used as
- Call arguments
- Return parameters
- Struct attributes

- Read Go code

- Simple, readable, maintainable (concise, top down)
- Through style – known syntactic (ex. Formatting) and semantic guidance (ex. Comma, ok)
- Through tooling – tool enforcing the standard
- Through thought – techniques to make code understandable

- Read/try go.dev:
- https://go.dev/ref/spec
- https://go.dev/tour/welcome/1
- https://go.dev/doc/code
- https://go.dev/doc/effective_go

Lab

- Idiomatic Go

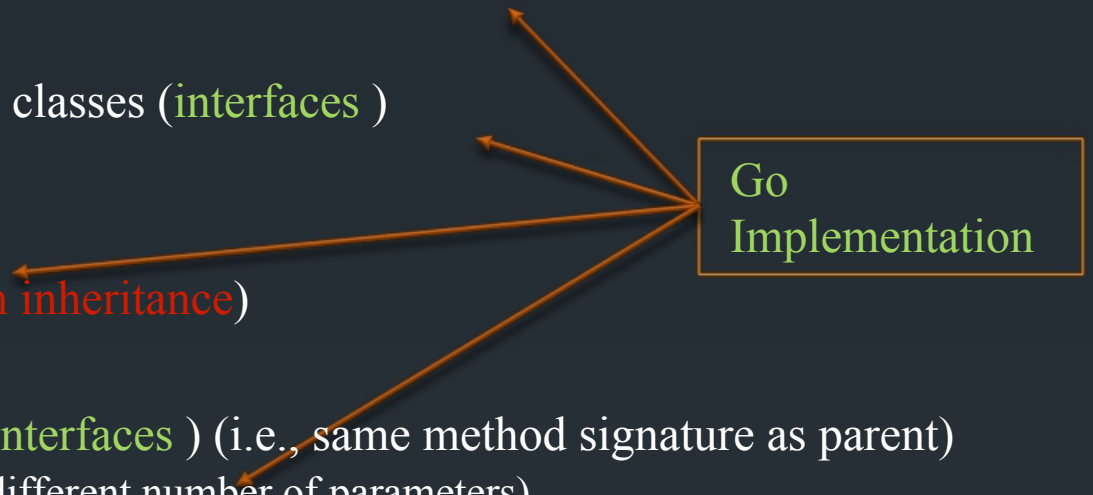
Methods and Interfaces

Objectives

- Explore:
- Methods
- Receivers
- Method binding
- Encapsulation
- Interfaces
- Type Assertions

OO concepts

- Encapsulation
- Hiding of data (exported/unexported identifiers at the package level)
- Abstraction
- Interfaces or abstract classes (interfaces)
- Inheritance
- has-a (composition)
- is-a (implementation inheritance)
- Polymorphism
- Method overriding (interfaces) (i.e., same method signature as parent)
  - Or overloading (i.e., different number of parameters)
- Classification
- Types with fixed interfaces and methods (XXX)



- Why care?
- Many people start in OO before Go
- Many guides (e.g. design patterns) are from the OO world
- Useful to understand the language subtleties when designing software architectures

Methods

- There is no universal definition of object-oriented programming
- In Go, an object is a variable that has methods
- A method is a function associated with a particular type
- An object-oriented program is one that uses methods to express the properties and operations of each data structure so that clients need not access the object's representation directly
- This is one of the key features of object orientation: Encapsulation
- A method is declared much like an ordinary function but with an extra parameter before the function name
- This parameter attaches the function to the type
- The type parameter name is called the receiver and is like the "this" or "self" pointer in some other OO languages
  - receiver names are commonly the first letter of the type name
- Methods are invoked using . notation on the desired object: bike.mm()
- Such and expression is called a selector in Go
- Each type has its own namespace allowing multiple types to have the same method names
- Methods can also use names already taken at the package level
- This allows method names to be compact
- Methods may be declared on any named type defined in the same package, so long as its underlying type is neither a pointer nor an interface

```go
package main

import "fmt"

type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

type digit int

func (i digit) prt() string {
    return fmt.Sprintf("int: %d", i)
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    fmt.Println(bike.mm())

    var x digit = 7
    fmt.Println(x.prt())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:05:30 server.go:73: Using API v1
2017/04/10 22:05:30 debugger.go:68: launching pro
API server listening at: 127.0.0.1:2345
2017/04/10 22:05:31 debugger.go:414: continuing
Honda VFR750
int: 7
```

Pointer Receivers

- Calling a function makes a copy of each argument value
- If a function needs to update a variable or if an argument is large, passing the address of the variable using a pointer is desirable
- Methods that need to update the receiver variable need a pointer receiver
- Even when pointer receivers are defined methods can be called with the name directly
  - bike.ModelSuffix()
  - As opposed to:
  - (&bike).ModelSuffix()
- Nil can be a legal receiver value for types with a nil zero value
- E.g. pointer receiver on a method

```go
package main

import "fmt"

// Moto is for motorcycle
type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

func (m *Moto) ModelSuffix(suf string) {
    m.Model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    fmt.Println(bike.mm())
    bike.ModelSuffix("F")
    fmt.Println(bike.mm())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:16:56 server.go:73: Using API v1
2017/04/10 22:16:56 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/10 22:16:56 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
```

Binding Methods

- It's possible to separate method/receiver binding from invocation
- The selector p.mm yields a function with the method mm bound to variable p
- This function can then be invoked without a receiver value
  - x := p.mm
  - x()
- The selector Moto.mm yields a function invoking the method mm with an additional parameter prepended to accept the receiver
  - x := Moto.mm
  - x(bike)
- Binding methods and objects is shockingly simple compared to the process in other languages (C++98 anyone?)

```go
package main

import "fmt"

// Moto is for motorcycle
type Moto struct {
    Make  string
    Model string
}

func (m Moto) mm() string {
    return m.Make + " " + m.Model
}

func (m *Moto) ModelSuffix(suf string) {
    m.Model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}

    bmm := Moto.mm
    fmt.Println(bmm(bike))

    bms := bike.ModelSuffix
    bms("F")
    fmt.Println(bike.mm())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:37:49 server.go:73: Using API v1
2017/04/10 22:37:49 debugger.go:68: launching proce
API server listening at: 127.0.0.1:2345
2017/04/10 22:37:50 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
```

Encapsulation in Go

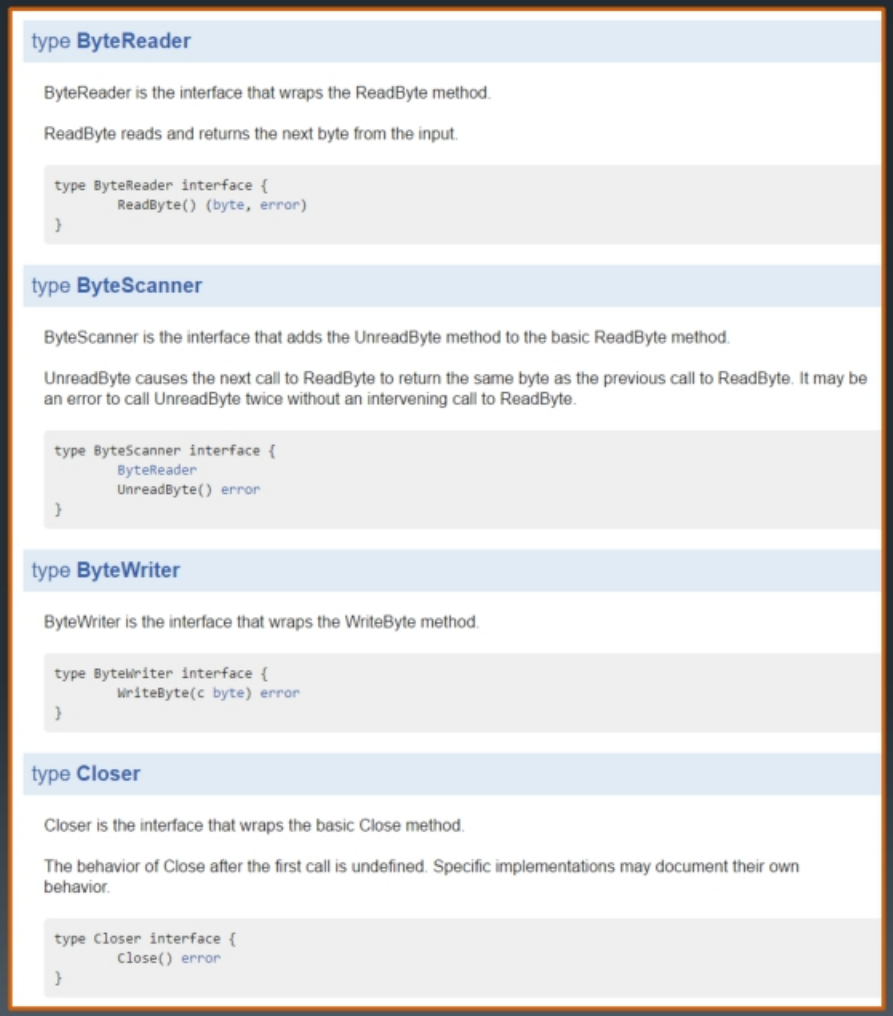
- In Go the unit of encapsulation is the package
  - NOT the type
  - The fields of a struct type are visible to all code within the same package
- Go has only one mechanism to control the visibility of names:
  - Identifier Capitalization
    - Capitalized names are exported from the package
    - Uncapitalized names are not exported from the package
  - This rule also applies to the fields of a struct or the methods of a type
- To encapsulate an object we must make it a struct and give the attribute(s) lowercase names
  - It is common to have structs with a single field for this reason
- Encapsulation provides three benefits
  - one need inspect fewer statements to understand the possible values of variables
  - hiding implementation details prevents clients from depending on things that might change
  - clients are prevented from setting an object's variables arbitrarily

```go
package main

import "fmt"

type Moto struct {
    make  string
    model string
}

func (m Moto) mm() string {
    return m.make + " " + m.model
}

func (m *Moto) ModelSuffix(suf string) {
    m.model += suf
}

func main() {
    bike := Moto{"Honda", "VFR750"}

    bmm := Moto.mm
    fmt.Println(bmm(bike))

    bms := bike.ModelSuffix
    bms("F")
    fmt.Println(bike.mm())
    fmt.Println(bike.make) //we in the same package so this works!
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/10 22:50:37 server.go:73: Using API v1
2017/04/10 22:50:37 debugger.go:68: launching process with args: [d:\dev\g
API server listening at: 127.0.0.1:2345
2017/04/10 22:50:37 debugger.go:414: continuing
Honda VFR750
Honda VFR750F
Honda
```

Interfaces

- Interface types express generalizations or abstractions about the behaviors of other types
- By generalizing, interfaces let us write functions that are more flexible and adaptable because they are not tied to the details of one particular implementation
- Go's interfaces are satisfied implicitly
- There's no need to declare interfaces that a given type satisfies
- Simply possessing the necessary methods is enough
- This is a fairly unique approach to interface/type association
- This design lets you create new interfaces that are satisfied by old types without changing the types
- particularly useful for types defined in packages that you don't control
- A concrete type specifies the exact representation of its values and exposes the intrinsic operations of that representation
- An Interface type is an abstract type and doesn't expose the representation or internal structure of its values, or the set of basic operations supported
- It reveals only some of their methods
- When you have a value of an interface type, you know nothing about what it is; you know only what behaviors are provided by its methods
- The io package offers several key interfaces



type **ByteReader**

ByteReader is the interface that wraps the ReadByte method.

ReadByte reads and returns the next byte from the input.

```
type ByteReader interface {
        ReadByte() (byte, error)
}
```

type **ByteScanner**

ByteScanner is the interface that adds the UnreadByte method to the basic ReadByte method.

UnreadByte causes the next call to ReadByte to return the same byte as the previous call to ReadByte. It may be an error to call UnreadByte twice without an intervening call to ReadByte.

```
type ByteScanner interface {
        ByteReader
        UnreadByte() error
}
```

type **ByteWriter**

ByteWriter is the interface that wraps the WriteByte method.

```
type ByteWriter interface {
        WriteByte(c byte) error
}
```

type **Closer**

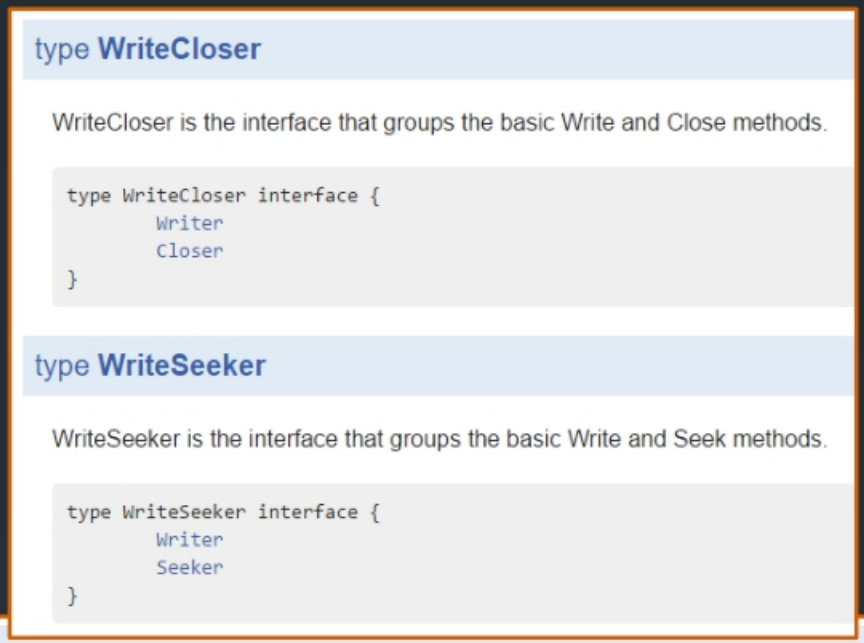
Closer is the interface that wraps the basic Close method.

The behavior of Close after the first call is undefined. Specific implementations may document their own behavior.

```
type Closer interface {
        Close() error
}
```

Interface Combinations

- An interface type specifies a set of methods that a concrete type must possess to be considered an instance of that interface
  - A type satisfies an interface if it possesses all the methods the interface requires.
- The io.Writer type is one of the most widely used interfaces because it provides an abstraction of all the types to which bytes can be written
  - Files
  - Memory buffers
  - Network conns
  - HTTP clients
  - Archivers
  - Hashers
- The io package defines interface types in terms of other interfaces as well



type **WriteCloser**

WriteCloser is the interface that groups the basic Write and Close methods.

```
type WriteCloser interface {
        Writer
        Closer
}
```

type **WriteSeeker**

WriteSeeker is the interface that groups the basic Write and Seek methods.

```
type WriteSeeker interface {
        Writer
        Seeker
}
```

type **Writer**

Writer is the interface that wraps the basic Write method.

Write writes len(p) bytes from p to the underlying data stream. It returns the number of bytes written from p (0 <= n <= len(p)) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns n < len(p). Write must not modify the slice data, even temporarily.

Implementations must not retain p.

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

Implementing an interface

- Imagine you would like to use the fmt.Fprintf method to write formatted text to a buffering type you have
  - The help for Fprintf says it wants an io.Writer which only has one method
  - Implementing that method will satisfy the interface and make it possible for you to use that type with many library functions
  - Note that our receiver is a pointer
  - We need to modify the object
  - Because of this we must pass the pointer to the object (&b) to functions desiring a Writer interface

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

```
1  package main
2
3  import "fmt"
4
5  type StrBuf struct {
6      buf string
7  }
8
9  func (sb *StrBuf) Write(p []byte) (n int, err error) {
10     sb.buf = string(p)
11     return len(sb.buf), nil
12 }
13
14 func main() {
15     var b StrBuf
16     fmt.Fprintf(&b, "Hi")
17     fmt.Println(b.buf)
18 }
19
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
2017/04/10 23:50:05 server.go:73: Using API v1
2017/04/10 23:50:05 debugger.go:68: launching process with args:
API server listening at: 127.0.0.1:2345
2017/04/10 23:50:05 debugger.go:414: continuing
Hi
```

func Fprintf

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

Fprintf formats according to a format specifier and writes to w. It returns the number of bytes written and any write error encountered.

Custom Interfaces

- Interface types are defined with the interface keyword
- Types implement interfaces by offering methods with identical signatures as those in the interface
- Interface variables can point to any object implementing the interface
  - This sets the type and value of the interface variable
  - Known as the dynamic type and value
    - Together these are referred to as the type descriptors
  - The static type of an interface variable is that of the interface
- nil interface calls cause a panic
  - if i == nil //test for unassigned interface variable
- Interface values may be compared using == and !=
  - Two interface values are equal if both are nil, or if their dynamic types are identical and their dynamic values are equal
  - Being comparable, interfaces can be used as map keys

```go
package main

import "fmt"

type Moto struct {
    make, model string
}

func (m Moto) Summary() string {
    return m.make + " " + m.model
}

type Veh interface {
    Summary() string
}

func main() {
    bike1 := Moto{"Honda", "VFR750"}
    bike2 := Moto{"Ducati", "Paso750"}
    var i Veh
    i = bike1
    fmt.Println(i.Summary())
    i = bike2
    fmt.Println(i.Summary())
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/11 00:14:54 server.go:73: Using API v1
2017/04/11 00:14:54 debugger.go:68: launching pr
API server listening at: 127.0.0.1:2345
2017/04/11 00:14:54 debugger.go:414: continuing
Honda VFR750
Ducati Paso750
```

Type Assertion

- Type assertions in Go allows you to test the types supported by the dynamic type associated with an interface
  - i.(T)
    - i is an interface and T is the desired type
  - Type assertions return a tuple with the desired interface type and an ok status
    - i2, ok := i.(T)
    - The type tested need NOT be the current dynamic type of i or a component thereof
  - Much like dynamic casting in other languages
  - A key difference is that the interface supplied and the interface derived need not be related in any way

```go
package main

import "fmt"

type Moto struct {
    make, model string
}

func (m Moto) Summary() string {
    return m.make + " " + m.model
}

type Veh interface {
    Summary() string
}

type Car interface {
    Drive() string
}

func main() {
    bike := Moto{"Honda", "VFR750"}
    var i Veh
    i = bike
    if v, ok := i.(Veh); ok {
        fmt.Println(v.Summary())
    }
    if c, ok := i.(Car); ok {
        fmt.Println(c.Drive())
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
2017/04/11 00:32:57 server.go:73: Using API v1
2017/04/11 00:32:57 debugger.go:68: launching proc
API server listening at: 127.0.0.1:2345
2017/04/11 00:32:57 debugger.go:414: continuing
Honda VFR750
```

Type Switches

- Type assertions can also be used in switch statements (called type switches)
  - An example is when you do not know the format of the source data
  - Another example is when you do not know the source type and which methods to call
- Cases can be ‘type’ or ‘interface’
  - The first match executes

```
 1 package main
 2
 3 import "fmt"
 4
 5 type Moto struct {
 6             make, model string
 7 }
 8
 9 func (m Moto) Summary() string {
10             return m.make + " " + m.model
11 }
12
13 func (m Moto) AnotherMethod() string {
14             return "nonVeh"
15 }
16
17 type Veh interface {
18             Summary() string
19 }
20
21 type Car interface {
22             Drive() string
23 }
24
25 func main() {
26             bike := Moto{"Honda", "VFR750"}
27             var i Veh
28             i = bike
29
30             switch v := i.(type) {
31             case Moto:
32                         fmt.Println("moto", v.AnotherMethod())
33             case Veh:
34                         fmt.Println("veh")
35             default:
36                         fmt.Println("default", v)
37             }
38 }
% go run switch.go
moto nonVeh
```

Interface

- Every type implements the empty interface
- var a any // 1.18+

  or
- var x interface{}

```
 1 package main
 2
 3 import "fmt"
 4
 5 func showMe(pre string, i any) {
 6     s := ""
 7
 8     switch v := i.(type) {
 9         case S:
10         s = fmt.Sprintf("S:                    (%T, %v)", v, v)
11         case T:
12         s = fmt.Sprintf("T:                    (%T, %v, %s)", v, v, v.toString())
13         case any: // place after concrete          types
14         s = fmt.Sprintf("any:(%T, %v)",        v, v)
15         default:
16         s = fmt.Sprintf("default:              (%T, %v)", v, v)
17     }
18
19     fmt.Println("---\n", pre, s)
20 }
21
22 type S struct{ a int }
23
24 type T struct{ s string }
25
26 func (t T) toString() string {
27     return "I am a string from t"
28 }
29
30 func main() {
31     var i any
32     i = 1
33     showMe("int", i)
34
35     m := make(map[string]any)
36     m["imnumberone"] = 1
37     showMe("mapwithint", m)
38
39     m["s1"] = new(S)
40     m["t1"] = new(T)
41     m["s2"] = S{}
42     m["t2"] = T{}
43     showMe("loadedmap", m)
44
45     for _, v := range m {
46         showMe("range", v)
47     }
48 }
```

```
% go run any.go
---
  int any:(int, 1)
---
  mapwithint any:(map[string]interface {},        map[imnumberone:1])
---
  loadedmap any:(map[string]interface {},         map[imnumberone:1 s1:0xc0000b2030 s2:{0} t1:0xc000096270 t2:{}])
---
  range S:                (main.S, {0})
---
  range T:                (main.T, {}, I am a     string from t)
---
  range any:(int, 1)
---
  range any:(*main.S, &{0})
---
  range any:(*main.T, &{})
```

Generics

- Generic programming enables the representation of functions and data structures in a generic form, with types factored out
- Prior to 1.18, Go had only generic like “empty interface”
- Work continues to vet in 1.18 and beyond
- Benefits
  - Additional compile time checks
  - No need to cast
  - Ability to implement generic algorithms (focus more on code then types)
  - Potential for singular optimizations
- Challenges
- Generics are typically implemented where ”is-a” inheritance is used (ex. Java, C++)
  - Go uses “has-a”
- Go demands casting (no auto-casting)

https://gotipplay.golang.org/p/-MJnuC6E13-

```
package main

import (
	"fmt"
)

func Print[T any](s ...T) {
	for _, v := range s {
		fmt.Print(v)
	}
}

func main() {
	Print("Hello, ", "playground\n")
	Print(1, 2)
}
```

```
Hello, playground
12
Program exited.
```

Go Tips

- When designing a new package:
  - Use Interfaces when two or more concrete types must be dealt with in a uniform way
  - Avoid single implementation interfaces
  - These are unnecessary abstractions and have a run-time cost
  - The exception is a single type interface that cannot live in the same package as the type because of dependencies (an interface can decouple the two packages)
  - Keep interfaces minimal, crisp and generic
  - Because interfaces are used when implemented by multiple types they tend to have fewer, simpler methods (often just one)
  - Small interfaces are easier to satisfy when new types come along
  - A good rule of thumb for interface design is ask only for what you need
  - Control method visibility outside a package through exporting

Source: Donovan/Kernighan

Summary

- Go support several key OO features:
- Methods
- Receivers
- Method binding
- Encapsulation
- Interfaces
- Type Assertions

Lab

- Working with methods and interfaces

Advanced Functions and Methods

Objectives

- Gain a deeper understanding of Go functions and their use
- Explore several useful function and interface patterns in Go
- Understand closures

Function Adaptors

- Single method interfaces are common in Go
- Simple, easy to use, widely applicable
- When an interface has only one method it may be desirable to implement the interface with a standalone function
- This will not work without help, anything expecting the interface will require the interface method to be implemented on the object supplied
- Adapters can be used to adapt a stand-alone function to meet the requirements of an interface
- This pattern is generally useful and appears often in the standard library
- The LogFunc type in the example to the right is a log interface adaptor for string returning functions
- Function adaptors are an idiomatic Go pattern

```
% cat funcadapter.go |
nl -b a -w 2 |
sed -e $'s/\t/        /g'
 1  package main
 2
 3  import "fmt"
 4
 5  type log interface {
 6      msg() string
 7  }
 8
 9  func Dump(l log) {
10      fmt.Println(l.msg())
11  }
12  // function adapater
13  type LogFunc func() string
14
15  func (f LogFunc) msg() string {
16      return f()
17  }
18
19  func MyLogger() string {
20      return "Hello World Too!"
21  }
22
23  func DumpFunc(f func() string) {
24      Dump(LogFunc(f))
25  }
26
27  func main() {
28      Dump(LogFunc(func() string {
29          return "Hello World!"
30      }))
29      Dump(LogFunc(MyLogger))
30      DumpFunc(MyLogger)
31  }
% go run funcadapter.go
Hello World!
Hello World Too!
Hello World Too!
```

Function Adaptors - ex. net/http/server.go

/usr/local/go/src/net/http/server.go

```
...
// A Handler responds to an HTTP request.
//
// ServeHTTP should write reply headers and data to the ResponseWriter
// and then return. Returning signals that the request is finished;
// is not valid to use the ResponseWriter or read from the
// Request.Body after or concurrently with the completion of the
// ServeHTTP call.
//
// Depending on the HTTP client software, HTTP protocol version,
// any intermediaries between the client and the Go server, it
// be possible to read from the Request.Body after writing to the
// ResponseWriter. Cautious handlers should read the Request.Body
// first, and then reply.
//
// Except for reading the body, handlers should not modify the
// provided Request.
//
// If ServeHTTP panics, the server (the caller of ServeHTTP)
// that the effect of the panic was isolated to the active request.
// It recovers the panic, logs a stack trace to the server error
// and either closes the network connection or sends an HTTP/2
// RST_STREAM, depending on the HTTP protocol. To abort a handler
// the client sees an interrupted response but the server doesn't
// an error, panic with the value ErrAbortHandler.
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}

... Example usage in same file ...

// NotFound replies to the request with an HTTP 404 not found
func NotFound(w ResponseWriter, r *Request) { Error(w, "404 page

// NotFoundHandler returns a simple request handler
// that replies to each request with a ``404 page not found"
func NotFoundHandler() Handler { return HandlerFunc(NotFound) }
```

it

and

may not

assumes

adapter

HandlerFunc type allowing the use functions as HTTP

```
// The HandlerFunc type is an
// ordinary functions as HTTP
// with the appropriate signature,
// Handler that calls f.
type HandlerFunc func(ResponseWriter,

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w
        f(w, r)
}
```

adapter to allow the use of handlers. If f is a function HandlerFunc(f) is a

*Request)

ResponseWriter, r *Request) {

error.

not found", StatusNotFound) }

reply.

Function Adaptors (another example)

- /usr/local/go/src/crypto/tls/handshake_messages.go

vi +/AddValue $(go env GOROOT)/src/crypto/tls/handshake_messages.go

```
$ vi +/AddValue $(go env GOROOT)/src/crypto/tls/handshake_messages.go
import (
        "fmt"
        "strings"

        "golang.org/x/crypto/ cryptobyte "
)
…
// The marshalingFunction type is an adapter to allow the use of ordinary
// functions as cryptobyte.MarshalingValue.
type marshalingFunction func(b *cryptobyte.Builder) error

func (f marshalingFunction) Marshal (b *cryptobyte.Builder) error {
        return f(b)
}
…
// addBytesWithLength appends a sequence of bytes to the cryptobyte.Builder. If
// the length of the sequence is not the value specified, it produces an error.
func addBytesWithLength(b *cryptobyte.Builder , v []byte, n int) {
        b. AddValue ( marshalingFunction(func(b *cryptobyte.Builder) error {
                if len(v) != n {
                        return fmt.Errorf("invalid value length: expected %d, got %d", n, len(v))
                }
                b.AddBytes(v)
                return nil
        }) )
}
…
```

```
…
func (b *Builder) AddValue (v MarshalingValue) {
        err := v. Marshal (b)
        if err != nil {
                b.err = err
        }
}
…
```

/usr/local/go/src /vendor/golang.org/x/crypto/cryptobyte/builder.go line 332

Logging …

- Package *log* implements simple logging
- It defines the Logger type with methods for formatting output

```
// A Logger represents an active logging object that generates lines of
// output to an io.Writer. Each logging operation makes a single call to
// the Writer's Write method. A Logger can be used simultaneously from
// multiple goroutines; it guarantees to serialize access to the Writer.
type Logger struct {
        mu     sync.Mutex // ensures atomic writes; protects the following fields
        prefix string     // prefix on each line to identify the logger (but see Lmsgprefix)
        flag   int        // properties
        out    io.Writer  // destination for output
        buf    []byte     // for accumulati  text to write
}
```

Details:

- $(go env GOROOT)/src/log/log.go
- go doc log Logger

```
 1  package main
 2
 3  import "log"
 4
 5  func test(s string) {
 6      log.Println("------------")
 7      defer log.Println("------------")
 8
 9      log.Println(s)
10      log.Println(s)
11  }
12
13  func main() {
14      test("call one")
15      test("call two")
16      test("call three")
17  }
```

```
2021/02/23 22:52:09 ------------
2021/02/23 22:52:09 call one
2021/02/23 22:52:09 call one
2021/02/23 22:52:09 ------------
2021/02/23 22:52:09 ------------
2021/02/23 22:52:09 call two
2021/02/23 22:52:09 call two
2021/02/23 22:52:09 ------------
2021/02/23 22:52:09 ------------
2021/02/23 22:52:09 call three
2021/02/23 22:52:09 call three
2021/02/23 22:52:09 ------------
```

... Logging

- It also has a predefined stderr Logger accessible through helper functions:
- Print[f|ln], Fatal[f|ln], and Panic[f|ln]
- The date and time of each logged message is prepended
- If the message being printed does not end in a newline, the logger will add one
- The *Fatal* functions call os.Exit(1) after writing the log message
- The *Panic* functions call panic after writing the log message
- Deferring clean up code and closing methods, like the log sentinel in the example is idiomatic Go

```
% cat log.go
package main

import (
    "fmt"
    "log"
    "os"
)

func main() {
    f, err :=                    os.OpenFile("testlogfile", os.O_RDWR|os.O_CREATE|os.O_APPEND, 0666)
    if err != nil {
        log.Fatalf("error          opening file: %v", err)
    }
    defer f.Close()

    log.SetOutput(f)
    log.Println("This           is a test log entry")
    fmt.Println("This           standard out")

    myLog := log.New(os.S                "cool_prefix", log.Llongfile)
    myLog.Println("myLog          test")

    myLogV2 :=                   log.New(os.Stderr, "cool_prefix", log.Lshortfile|log.Lmsgprefix)
    myLogV2.Println("movi             prefix")
}

% go run log.go
This standard out
                             cool_prefix/Users/ronald.petty/github.com/jwkidd3/go/temp/log.go:21: myLog
log.go:24: cool_prefixmovi               prefix

% echo $?
0
```

```
go doc log.Fatalf
package log // import "log"

func Fatalf(format string, v ...interface{})
    Fatalf is equivalent to Printf()
followed by a call to os.Exit(1).
```

```
% cat logp.go
package main

import "log"

func main() { log.Panicln("panicing") }

% go run logp.go
2020/06/21 14:29:57 panicing
panic: panicing


goroutine 1 [running]:
log.Panicln(0xc00006ef68, 0x1, 0x1)
                    /usr/local/go/src/log/log.go:365 +0xac
main.main()
                    /Users/ronald.petty/github.com/jwkidd3/go/temp/logp.go:5 +0x5d
exit status 2

% echo $?
1
```

```
% cat logf.go
package main

import "log"

func main() { log.Fatalln("exiting") }

% go run logf.go
2020/06/21 14:30:31 exiting
exit status 1

% echo $?
1
```

Returning Functions

- Go functions are first class types
- They can be passed and returned like other types
- Several useful patterns involve returning functions
- Ex. Teardown or Cleanup pattern:
  - When a pair of functions respectively perform some initialization and cleanup
  - The initialization function can return the cleanup function

```
% go run funccleanup.go
2021/02/23 22:59:05 setting things up
2021/02/23 22:59:05 use resources prepared
2021/02/23 22:59:05 tearing complex things down
```

- These function pairs keep related code together
  - Makes it easier to keep the routines in sync
  - Hides details from the user

```
 1  package main
 2
 3  import (
 4      "log"
 5      "math/rand"
 6      "time"
 7  )
 8
 9  func complexThing() bool {
10      rand.Seed(time.Now().UnixNano())
11      return rand.Float32() < 0.5
12  }
13
14  func prepare() func() {
15      var cleanup func() = func() {} // noop cleanup
16
17      //prepare - do complex things maybe, could fail
18      log.Println("setting things up")
19      if complexThing() {
20          return cleanup
21      }
22
23      cleanup = func() {
24          log.Println("tearing complex things down")
25      } // complex cleanup
26
27      return cleanup
28  }
29
30  func main() {
31      cu := prepare()
32      defer cu()
33      log.Println("use resources prepared")
34  }
```

Closures

- Functions can return functions
- Anonymous function objects can refer to variables in their enclosing scope
- A closure is a function value that references variables from outside its body
  - getinc() returns a closure
  - The anonymous inner function access and updates the local variable x of the enclosing getinc() function
- Hidden variable references are why functions are classified as reference types
- Use cases:
- Isolating data – when a function's data shouldn't be accessed
- Wrapping handlers with more logic
- Deferring work

```
 1  package main
 2
 3  import "fmt"
 4
 5  var y int = 0
 6
 7  func getinc(id string) func() string {
 8      var x int
 9
10      return func() string {
11          x++
12          return fmt.Sprintf("%s:%d:%d", id, x, y)
13      }
14  }
15
16  func main() {
17      i1 := getinc("i1")
18      i2 := getinc("i2")
19
20      fmt.Println(i1())
21      fmt.Println(i1())
22      y++
23      fmt.Println(i2())
24      fmt.Println(i2())
25      y++
26      fmt.Println(i1())
27  }
% go run closure.go
i1:1:0
i1:2:0
i2:1:1
i2:2:1
i1:3:2
```

Using closures with returned functions

- Go anonymous functions can form closures
- When a returned function references (closes over) a variable in an outer scope, it forms a closure
- In the example, “t” and “op” are enclosed in the returned function
- Users of the function start do not need to manage this state, it is handled by the closure

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "time"
 6  )
 7
 8  func start(op string) func() {
 9      t := time.Now()
10
11      fmt.Println(op, " start")
12
13      stop := func() {
14          t2 := time.Now().Sub(t)
15          fmt.Println(op, "stop, ", t2)
16      }
17
18      return stop
19  }
20
21  func main() {
22      stop := start("BigOp")
23      defer stop()
24      fmt.Println("perform long running operations")
25  }
% go run funcclose.go
BigOp  start
perform long running operations
BigOp stop,     26.509µs
```

Slices Implementing Interfaces

- Single function interfaces can be implemented by slices of the same interface
- A helper function implements the slice version of the interface by iterating over the slice and invoking the interface method on each element
- This implements two patterns
- Polymorphism
- Aggregation

```
1  package main
2
3  import "fmt"
4
5  type Sizer interface{ Size()            int64 }
6
7  func BufAvail(s Sizer) bool {          return 1024 >= s.Size() }
8
9  type Sizers []Sizer
10
11 func (s Sizers) Size() int64 {
12    var t int64
13    for _, sizer := range s {
14       t += sizer.Size()
15    }
16    return t
17 }
18
19 type FileOb struct{ blks int64          }
20
21 func (f FileOb) Size() int64 {          return f.blks * 512 }
22
23 type MemOb struct{ bytes int64          }
24
25 func (m MemOb) Size() int64 {           return m.bytes }
26
27 func main() {
28    f := FileOb{10}
29    fmt.Println(BufAvail(f))
30    m := MemOb{1024}
31    fmt.Println(BufAvail(m))
32    s := Sizers{f, m}
33    fmt.Println(BufAvail(s))
34 }
```

```
% go run interslice.go
false
true
false
```

Structural Typing

- Imagine a scenario when a library you want to test implements a struct directly without defining an interface for it, yet you want to mock this struct
  - You can not mock a struct, only that type will satisfy parameters of that type
  - However, go uses “duck typing” to apply interfaces to types
  - You can always create your own interface for the library type
  - Then use your interface in your code
  - Mock the interface for testing
  - The library struct implicitly implements the interface so it becomes substitutable with the mock

```
1  package main
2
3  import "fmt"
4
5  type FileOb struct{ blks int64 }
6
7  func (f FileOb) Size() int64 {          return f.blks * 512 }
8  func (f FileOb) Empty() bool {          return f.blks == 0 }
9
10 type EmptySizer interface {
11    Size() int64
12    Empty() bool
13 }
14
15 type FileObMock struct {
16    SizeAttr      func() int64
17    EmptyAttr func() bool
18 }
19
20 func (f FileObMock) Size() int64        { return f.SizeAttr() }
21 func (f FileObMock) Empty() bool        { return f.EmptyAttr() }
22
23 func Dump(f EmptySizer) {
24    fmt.Println(f.Size())
25    fmt.Println(f.Empty())
26 }
27
28 func main() {
29    f := FileOb{10}
30    Dump(f)
31    m := FileObMock{}
32    m.SizeAttr = func() int64 { return      200 }
33    m.EmptyAttr = func() bool { return      true }
34    Dump(m)
35 }
```

```
% go run structinter.go
5120
false
200
true
```

Summary

- Go offers simple features with great flexibility
- Interfaces use structural typing allowing pre-existing structs to be passed through new interfaces
- Adapters can be used to allow functions and slices to be passed as interfaces
- Anonymous functions can be declared ad hoc anywhere a function of that type is valid
- Closures allow anonymous functions to capture variables from outer scopes
- Functions are first class citizens in Go and can be used as
- Call arguments
- Return parameters
- Struct attributes

Lab: Advanced functions

- Functions

Communicating Sequential Processes

Objectives

- Background & Terminology
- Understand Go's
- Concurrency
- Goroutines
- Channels
- Locking

Communicating Sequenctial Processes (CSP)

- Formal language for describing patterns of interaction in concurrent systems
  - Based on message passing via channels
  - First described by Tony Hoare in 1978
    - http://www.usingcsp.com/cspbook.pdf
    - Also known as CSP

- Tool for specifying and verifying the concurrent aspects of a system

- Uses include to find deadlock and livelock scenarios
- deadlock – all tasks waiting for lock release
- livelock – tasks make no progress
- ex. used in International Space Station, smart-cards, etc

- Go implements the channel and message passing aspects of CSP via Channels
  - Go implements to process aspects of CSP via GoRoutines

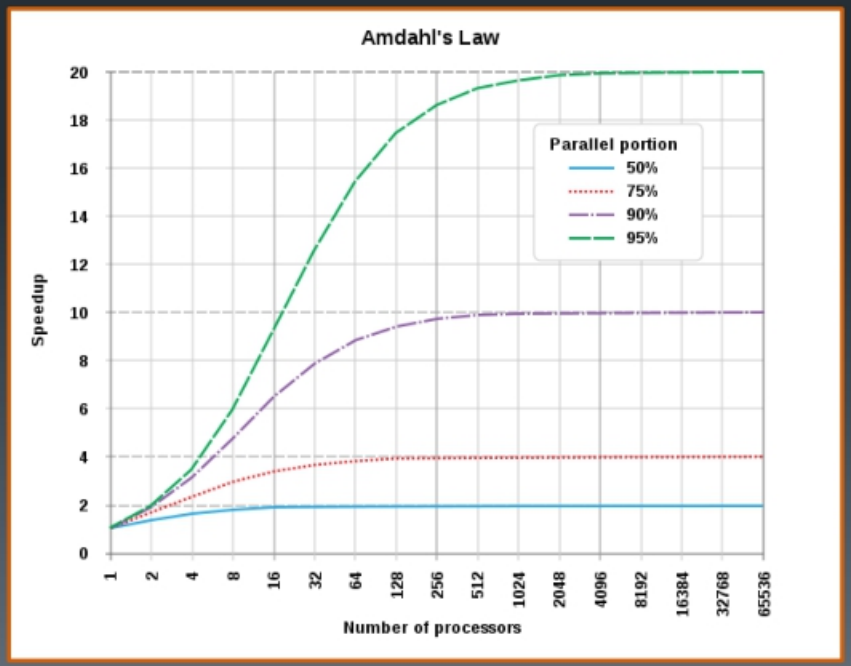
Basic syntax of CSP

$Proc$	::=	STOP	
	|	SKIP	
	|	$e \rightarrow Proc$	(prefixing)
	|	$Proc \Box Proc$	(external choice)
	|	$Proc \sqcap Proc$	(nondeterministic choice)
	|	$Proc \mid\mid\mid Proc$	(interleaving)
	|	$Proc \mid[\{X\}]\mid Proc$	(interface parallel)
	|	$Proc \setminus X$	(hiding)
	|	$Proc; Proc$	(sequential composition)
	|	if b then $Proc$ else $Proc$	(boolean conditional)
	|	$Proc \triangleright Proc$	(timeout)
	|	$Proc \triangle Proc$	(interrupt)

Parallelism

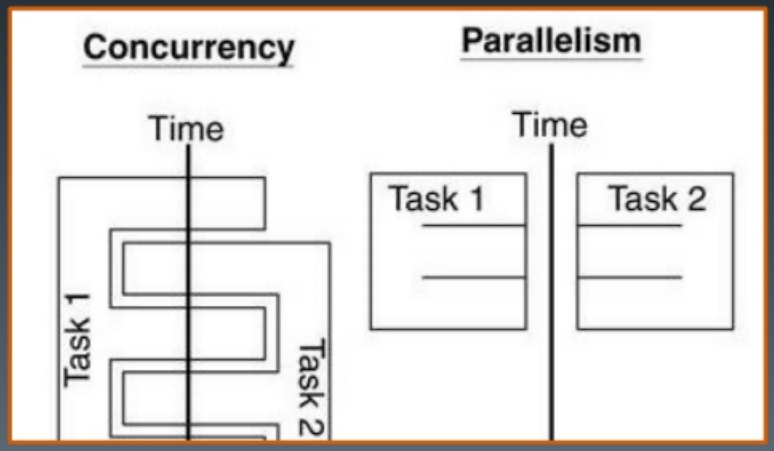
- Parallel computing implies simultaneous execution of processes/threads
  - Large problems can often be divided into smaller ones, which can then be solved at the same time, in parallel
- Types of parallel computing:
  - Bit Parallelism – Increasing cpu word size (the number of parallel bits processed) reduces the instructions the processor must execute to perform an operation
  - Instruction Parallelism – Increasing the number of instructions a computer can execute at the same time (multi core/processor)
  - Data Parallelism – Distributing data across multiple computers allowing them to process the data in parallel
  - Task Parallelism - Distributing tasks concurrently performed by processes or threads across different processors
- Physical constraints preventing frequency scaling have driven multiprocessor and multicore systems
  - Nearly all modern software systems have access to parallel computing (phones and many other embedded systems are typically multicore)
  - This places the burden of designing for task parallelism on developers who wish to take advantage of multiple cores/processors
- Parallel computers can be roughly classified according to the level at which the hardware supports parallelism
  - Multi-core and multi-processor computers having multiple processing elements within a single machine
  - Clusters , MPPs, and grids use multiple computers to work on the same task
- In some cases parallelism is transparent to the programmer
  - E.g. bit-level and instruction-level parallelism
- Parallel algorithms are more difficult to write than sequential ones
  - Concurrency introduces several new classes of software bugs
    - Race conditions are the most common
  - Communication and synchronization between different subtasks are some of the greatest obstacles to good parallel program performance
- Amdahl's law defines the upper bound of program parallel speed-up
  - S_{latency} the theoretical speedup of the execution of the whole task
  - s the speedup of the part that benefits from improved resources
  - p the proportion of execution time that the part benefiting from improved resources originally occupied
  - E.g. Given
    - A program needs 20 hours on a single core to complete
    - A part of the program takes one hour to execute and cannot be parallelized
    - The remaining 19 hours (p = 0.95) of execution can be parallelized
    - At any core count the min execution time cannot be less than 1 hour
    - The theoretical speedup is limited to 20 times (1/(1 − p) = 20)

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$



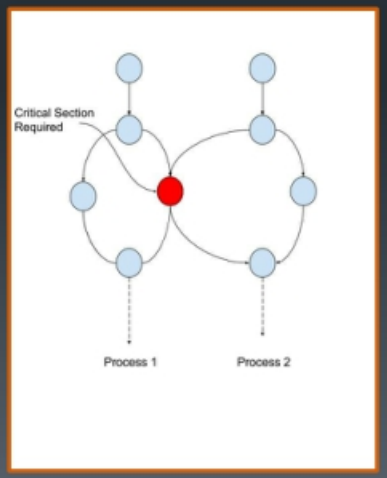
Concurrency

- Concurrency is the composition of independently executing things
- -- Rob Pike
- Parallel computing is closely related to concurrent computing though the two are distinct:
- It is possible to have parallelism without concurrency
  - Such as bit-level parallelism
- It is possible to have concurrency without parallelism
  - Such as multitasking by time-sharing on a single-core CPU
- Concurrency is the decomposability a program into order-independent or partially-ordered components
- This means that even if the concurrent units of a program are executed out-of-order or in partial order, the final outcome will remain the same
- This allows for parallel execution of the concurrent units which can significantly improve overall speed of the execution in multi-processor and multi-core systems



Ordering

- Designing for concurrency involves decomposing a program, algorithm, or problem into order-independent or partially-ordered components
  - If the concurrent units of the program, algorithm, or problem are executed out-of-order or in partial order, the final outcome will remain the same
- The goal of concurrency is to improve overall speed of the execution in multi-processor and multi-core systems
- Mutual exclusion is a property of concurrency control
- Purpose is to prevent race conditions
- One thread of execution can not enter a critical section at the same time as another thread
- A critical section is a piece of code where concurrent access can lead to undefined behavior
- Concurrency does not mean parallel
- 1 CPU can run things concurrently, but not in parallel



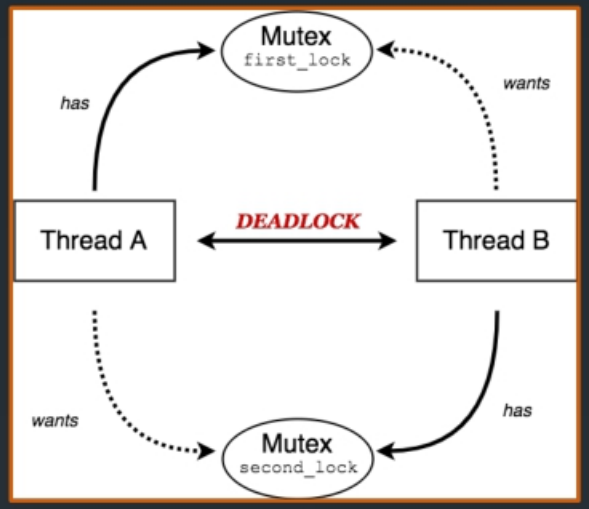
Mutex vs Semaphore

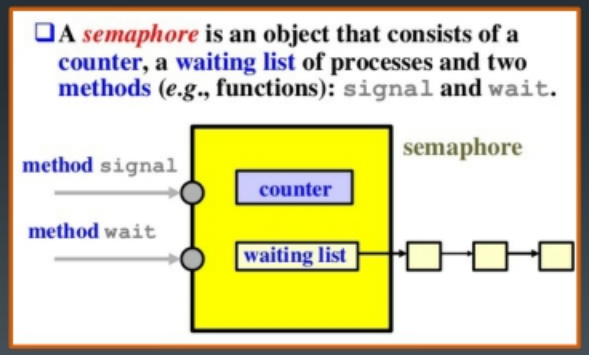
- lock or mutex (from mutual exclusion) is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution
- semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system

- ...seem similar, hum...

- mutex is meant to be taken and released
- semaphore is meant to signal or wait
  - tasks do one (signal) or the other (wait) but not both

- Improper use of mutex can lead to priority inversion
  - Priority inversion in scheduling is a scenario where a high priority task is preempted by a lower priority task
  - Example
  - 3 tasks each with increasing priority (1 is low 3 is high)
  - tasks 1 and 3 use mutex, task 2 does not use mutex
  - task 1 acquires lock, blocking task 3,
  - task 2 preempts task 1, task 3 needs task 2 to release the mutex, thus it waits
  - Mars Pathfinder (1997) suffered a priority inversion error





Coroutines

- Subroutine
  - sequence of program instructions that performs a specific task
  - Coroutine
  - Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.
  - Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.
  - Used in 1958 (Knuth, Conway), explained in 1963
  - Concurrent, not parallel
    - can largely avoid locks as coroutines can only be rescheduled at certain points, unlike threads
  - Goroutine
  - Concurrent and parallel
  - When coroutines are not available, could create a coroutine via a closure (ala keeping state)
  - Threads are an alternative to coroutines
  - Threads are preemptively schedules, coroutines are not
  - Heavier than goroutines, due to management overhead (lighter than processes – ala sharing memory)
    - Aspects of threading derive from Plan 9 --- Rob Pike, Ken Thompson (who are these people!)

Ex. coroutines

```
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

Traditional Tools

C++

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void increment_count() {
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long long get_count() {
    long long c;
    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

mutexlock

nested locking

lockinghierarchies

Java

```
try {
    mutex.acquire();

    try {
        // do something
    } finally {
        mutex.release();
    }
} catch(InterruptedException ie) {
    // ...
}
```

deadlocks

conditionallocking

https://blog.codinghorror.com/

Goroutines

- A goroutine is a function that is capable of running concurrently with other functions
- To create a goroutine we use the keyword go followed by a function invocation:
- go f()
- All programs consist of at least one goroutine
- The first goroutine is implicit and is the main function itself
- Additional goroutines are created when arriving at go statements
- Normally when a function is invoked the program will execute all the statements in a function and then return to the next line following the invocation
- goroutines return immediately and don't wait for the function to complete
- Goroutines are lightweight, programs can create thousands of them without extreme penalties

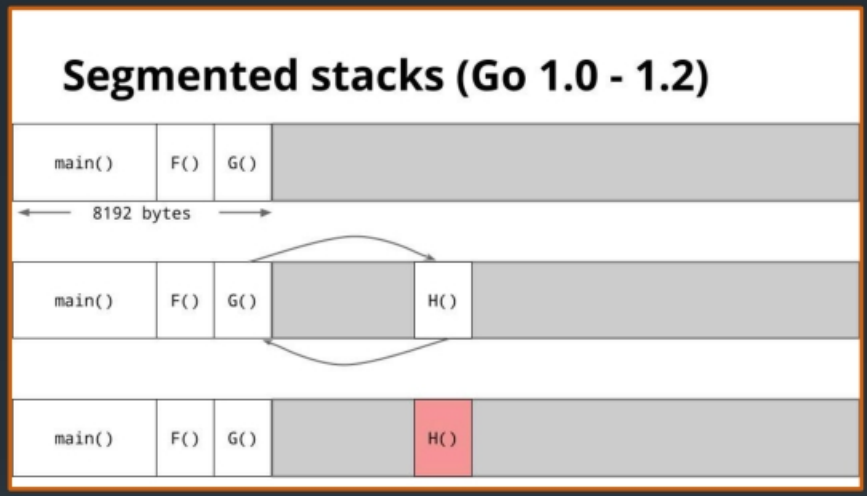
```
% go run gorout.go
1 : 0
0 : 0
2 : 0
1 : 1
2 : 1
0 : 1
1 : 2
2 : 2
0 : 2
```

```
1  package main
2
3  import (
4     "fmt"
5     "math/rand"
6     "time"
7  )
8
9  func f(n int) {
10    for i := 0; i < 3; i++ {
11       fmt.Println(n, ":", i)
12       amt := time.Duration(rand.Intn(250))
13       time.Sleep(time.Millisecond * amt)
14    }
15 }
16
17 func main() {
18    for i := 0; i < 3; i++ {
19       go f(i)
20    }
21
22    var input string
23    fmt.Scanln(&input)
24 }
```

Call Stacks

- goroutine:
- Play on the related term coroutines
- A function that executes concurrently with other goroutines in the same address space (process)
- Lightweight
  - Little more than allocation of stack space
  - Stack memory is small at creation time, so it is cheap
    - 4096 byte initial stack
  - Uses heap to allocate larger stack if required
- Multiplexes onto OS threads , if one blocks another goroutine will take over the OS thread and continue to run
  - Low level threading details are hidden
- To use:
- Prefix your function call with “go”
- Ex. “ go myfunc() ”
- Like “async” in some languages
- Are implemented as follows:
- Small preamble is inserted before each function
- Checks to see if function needs less than available memory (else call runtime-morestack)
- Copies argument to new stack
- Returns control to caller
- When goroutine launched function finishes, memory is reclaimed
- While goroutines make it easy to run parallel code, ordering requires waiting/blocking, hence channels

```
var a string

func f() {
  print(a)
}

func hello() {
  a = "hello, world"
  go f()
}
```



https://dave.cheney.net/wp-content/uploads/2014/06/Gocon-2014-43.jpg

Old ↑

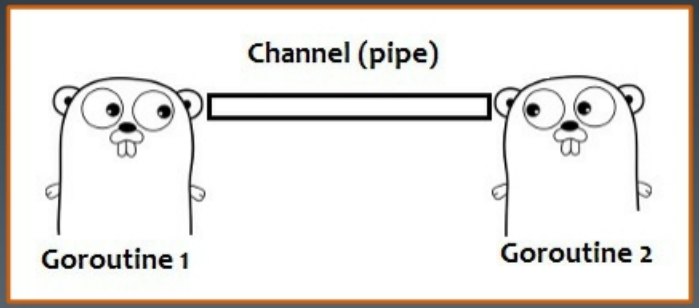
New ↓

Go 1.3 has changed the implementation of goroutine stacks away from the old, "segmented" model to a contiguous model. When a goroutine needs more stack than is available, its stack is transferred to a larger single block of memory. The overhead of this transfer operation amortizes well and eliminates the old "hot spot" problem when a calculation repeatedly steps across a segment boundary.

Channels

- Channels are a typed conduit through which you can send and receive values with the channel operator: <-
  - Send v to channel c c <- v
  - Receive v from channel c v := <- c
  - Channels are allocated via a call to make()
    - Example c := make(chan int)
  - By default, senders and receivers **block** until the both sides are ready
  - Allows for synchronization without locks
  - Channels can be unbuffered or buffered
  - Unbuffered combine exchange of data with synchronization
    - Aka. The sender blocks until the receiver has received the value
  - Buffered can transfer data asynchronously
    - Aka. The sender does not block unless the buffer is full. The receiver empties the buffer.
  - Channels are the main method of communication between goroutines

```
 1  package main
 2
 3  import "fmt"
 4
 5  var c = make(chan string)
 6
 7  func f() {
 8      a := "hello world"
 9      c <- a
10  }
11
12  func main() {
13      go f()
14      x := <-c
15      fmt.Println(x)
16  }
% go run c.go
hello world
```



https://waimengmoan.gitbooks.io/go-applications/content/channel.jpg

Additional Concepts

generator

```
func boring(msg string) <-chan string {
  c := make(chan string)
  go func() {
    for i := 0; ; i++ {
      c <- fmt.Sprintf("%s %d", msg, i)
      time.Sleep(1000* time.Millisecond)
    }
  }()
  return c
}
```

- Pipelines
- No formal definition
  - Informally a series of stages connected by channels
  - General flow:
  - Upstream provides input to goroutines via channel
  - goroutines process data
  - Downstream receives results from processing
  - Each stage can be many to many (m to n) channels
- Generator pattern
- Return a channel
- Channel as reference to a service
- Multiplexing, and more.

```
func main() {
  joe := boring("Joe")
  ann := boring("Ann")

  for i := 0; i < 5; i++ {
    fmt.Println(<-joe) //using generator
    fmt.Println(<-ann) //using generator
  }

  fmt.Println("You're both boring; I'm leaving.")
}
```

Once

- Package “sync” provides a safe

mechanism for initialization in presence of multiple goroutines
  - once.Do(f)
    - Function f is only called once

```
package main

import (
        "fmt"
)       "sync"


var a string
var once sync.Once

func setup() {
        a = "hello, world"
}       fmt.Printf("setting up %s\n", a)


func doSomething() {
        once.Do(setup)
}

func main() {
        go doSomething()
        go doSomething()
        var input string
        fmt.Scanln(&input)
}
```

```
% go run once.go
setting up hello, world
<enter>
%
```

Mutexes and Locks

- Package “sync” provides two lock data types
- sync.Mutex
  - All locks are exclusive
- sync.RWMutex
  - Write locks are exclusive
  - Read locks can be shared
- Provide mutual exclusion support
- func (m *Mutex) Lock()
  - Lock locks m, blocks until the mutex is available
- func (m *Mutex) Unlock()
  - Unlock unlocks m, run-time error if m is not locked

```
var l sync.Mutex

var a string

func f() {
    a = "hello, world"
    l.Unlock()
}

func main() {
    l.Lock()
    go f()
    l.Lock()
    print(a)
}
```

Conditions

- sync package type Cond
  - Cond implements a condition variable
  - A rendezvous point for goroutines waiting for or announcing the occurrence of an event
  - Each Cond has an associated Locker L (usually a *Mutex or *RWMutex) which must be held when changing the condition and when calling the Wait method
  - Can be created as part of other structures
  - Must not be copied after first use
- func NewCond(l Locker) *Cond
  - Returns a new Cond with Locker l
  - func (c *Cond) Broadcast ()
  - Wakes all goroutines waiting on c
  - It is allowed but not required for the caller to hold c.L during the call
  - func (c *Cond) Signal()
  - Wakes one goroutine waiting on c, if there is any
  - It is allowed but not required for the caller to hold c.L during the call
  - func (c *Cond) Wait ()
  - Atomically unlocks c.L and suspends execution of the calling goroutine
  - Wait locks c.L before returning
  - Wait cannot return unless awoken by Broadcast or Signal

```
1   package main
2
3   import "sync"
4   import "log"
5
6   func main() {
7       var m sync.Mutex
8       c := sync.NewCond(&m)
9       n := 2
10      running := make(chan bool, n)
11      awake := make(chan bool, n)
12      for i := 0; i < n; i++ {
13          go func() {
14              m.Lock()
15              running <- true
16              c.Wait()
17              awake <- true
18              m.Unlock()
19          }()
20      }
21      for i := 0; i < n; i++ {
22          <-running
23      }
24      for n > 0 {
25          select {
26          case <-awake:
27              log.Fatal("goroutine not asleep")
28          default:
29          }
30          m.Lock()
31          c.Signal()
32          m.Unlock()
33          <-awake
34          select {
35          case <-awake:
36              log.Fatal("too many goroutines awake")
37          default:
38          }
39          n--
40      }
41      c.Signal()
42
43  }
```

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "sync"
 6  )
 7
 8  // Pool for our struct A
 9  var pool *sync.Pool
10
11  // A dummy struct with a member
12  type A struct {
13      Name string
14  }
15
16  // Func to init pool
17  func initPool() {
18      pool = &sync.Pool{
19          New: func() interface{} {
20              fmt.Println("Returning new A")
21              return new(A)
22          },
23      }
24  }
25
26  // Main func
27  func main() {
28      // Initializing pool
29      initPool()
30      // Get hold of instance one
31      one := pool.Get().(*A)
32      one.Name = "first"
33      fmt.Printf("one.Name = %s\n", one.Name)
34      // Submit back the instance after using
35      pool.Put(one)
36      // Now the same instance becomes usable by another
37      // routine without allocating it again
38  }
% go run pools.go
Returning new A
one.Name = first
```

Pools

- sync package type Pool
  - type Pool struct {
    // Optional function to generate a value (may not be changed concurrently with calls to Get)
    New func() interface{}
    }
  - A Pool is a set of temporary objects that may be individually saved and retrieved
  - Any item stored in the Pool may be removed automatically
    - Safe for use by multiple goroutines simultaneously
- Pool's purpose is to cache allocated but unused items for later reuse, relieving pressure on the garbage collector
  - A thread-safe free lists
- Used to manage a group of temporary items silently shared among and potentially reused by concurrent independent clients of a package
  - Amortizes allocation overhead across many clients
  - e.g. The fmt package maintains a dynamically-sized store of temporary output buffers in a Pool
- Free lists for short-lived objects are not a suitable use for a Pool due to the Pool overhead
  - It is more efficient to have such objects implement their own free list
- A Pool must not be copied after first use
- func (p *Pool) Get() interface{}
  - Selects an arbitrary item from the Pool to return to the caller
  - Get may choose to ignore the pool and treat it as empty
    - If Get would otherwise return nil and p.New is non-nil, Get returns the result of calling p.New
- func (p *Pool) Put(x interface{})
  - Put adds x to the pool

Wait Groups

- The sync package provides the WaitGroup type
- WaitGroup
- A type used to wait for a collection of goroutines to finish
- The main goroutine calls Add to set the number of goroutines to wait for
- Each of the goroutines runs and calls Done when finished
- Must not be copied after first use.
- func (wg *WaitGroup) Add(delta int)
- Adds delta, which may be negative, to the WaitGroup counter
- If the counter becomes zero, all goroutines blocked on Wait are released
- If the counter goes negative, Add panics
- Calls with a positive delta that occur when the counter is zero must happen before a Wait
- Calls with a negative delta, or calls with a positive delta that start when the counter is greater than zero, may happen at any time
  - Thus calls to Add should execute before the statement creating the goroutine or other event to be waited for
  - If a WaitGroup is reused to wait for several independent sets of events, new Add calls must happen after all previous Wait calls have returned
- func (wg *WaitGroup) Done()
- Decrements the WaitGroup counter
- func (wg *WaitGroup) Wait()
- Blocks until the WaitGroup counter is zero

```
1  package main
2
3  import (
4      "fmt"
5      "net/http"
6      "sync"
7      "time"
8  )
9
10 func main() {
11     var wg           sync.WaitGroup
12     var urls =       []string{
13         "http://golang.
14         "http://google.
15                      "http://example.com",
16     }
17     for _, url :=     range urls {
18         wg.Add(1)
19         go func(url    string) {
20             defer      wg.Done()
21
22             var        netClient = &http.Client{
23                 Timeout    time.Second * 1,
24             }
25
26             resp, err     := netClient.Get(url)
27
28             if err ==     nil {
29                         fmt.Println(resp.Status, " ", url)
30             }
31         }(url)
32     }
33     wg.Wait()
34 }
```

76

```
% go run wait.go
200 OK     http://example.com
200 OK     http://google.com
200 OK     http://golang.org
```

Structural Embedding

- In the same way that you can embed other types within a struct, you can embed mutexes
- Embedded types create anonymous fields accessed through the instance itself
  - ex. sync.Mutex
    - Often called a “mutex hat”, as it sits like a hat on top of the elements it protects

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "sync"
 6      "time"
 7  )
 8
 9  type Counter struct {
10      sync.Mutex  //anonymous field
11      v                   map[string]int
12  }
13
14  func (c *Counter) Inc(key string)           {
15      c.Lock() // use of anonymous          field.method
16      c.v[key]++
17      time.Sleep(2000 * time.Millisecond)
18      c.Unlock() //not using defer           , but ok
19  }
20
21  func (c *Counter) Dec(key string)           {
22      c.v[key]-- //no protection!
23  }
24
25  func (c *Counter) Val(key string)           int {
26      c.Lock()
27      defer c.Unlock() //using defer            is the way!
28      return c.v[key]
29  }
30
31  func main() {
32      c := Counter{v: make(map[string]int)
33      go func() {
34          c.Inc("somekey")
35      }()
36      fmt.Println(c.Val("somekey"),           "waiting ...")
37      c.Dec("somekey")
38      fmt.Println("why did inc take          so long ... ", c.Val("somekey"))
39  }
```

```
% go run hat.go
0 waiting ...                    77
fatal error: concurrent map writes
```

Or?

```
% go run hat.go
0 waiting ...
why did inc take so long ...              0
```

sync.map

- Use only for (maybe)
  - Write once read many on a key
  - Multiple goroutines CRUD disjoint keys
- Generally not preferred to over plain Go map (ex. map[k]v)


```
% go doc sync
package sync // import "sync"

Package sync provides basic synchronization primitives such as mutual
exclusion locks. Other than the Once and WaitGroup types, most are intended
for use by low-level library routines. Higher-level synchronization is
better done via channels and communication.

Values containing the types defined in this package should not be copied.

type Cond struct{ ... }
    func NewCond(l Locker) *Cond
type Locker interface{ ... }
type Map struct{ ... }
type Mutex struct{ ... }
type Once struct{ ... }
type Pool struct{ ... }
type RWMutex struct{ ... }
type WaitGroup struct{ ... }
ronaldpetty@Ronalds-MacBook-Pro mymod % go doc sync.Map
package sync // import "sync"

type Map struct {
                    // Has unexported fields.
}
    Map is like a Go map[interface{}]interface{} but is safe for concurrent use
    by multiple goroutines without additional locking or coordination. Loads,
    stores, and deletes run in amortized constant time.

    The Map type is specialized. Most code should use a plain Go map instead,
    with separate locking or coordination, for better type safety and to make it
    easier to maintain other invariants along with the map content.

    The Map type is optimized for two common use cases: (1) when the entry for a
    given key is only ever written once but read many times, as in caches that
    only grow, or (2) when multiple goroutines read, write, and overwrite
    entries for disjoint sets of keys. In these two cases, use of a Map may
    significantly reduce lock contention compared to a Go map paired with a
    separate Mutex or RWMutex.

    The zero Map is empty and ready for use. A Map must not be copied after
    first use.

func (m *Map) Delete(key any)
func (m *Map) Load(key any) (value any, ok bool)
func (m *Map) LoadAndDelete(key any) (value any, loaded bool)
func (m *Map) LoadOrStore(key, value any) (actual any, loaded bool)
func (m *Map) Range(f func(key, value any) bool)
func (m *Map) Store(key, value any)
```

Summary

- Goroutines provide a simper alternative to heavy weight threads in other languages
- Channels provide a way to communicate between goroutines
- Don’t communicate by sharing memory, share memory to communicate

Lab: CSP and goroutines

- Work with goroutines and channels

Day 2

5. Channels
6. Context
7. Testing
8. Performance

Channels

Objectives

- Explore Go Channels
- Examine some common Channel and Go Routing patterns
- Understand the role of channels in synchronization
- Define
- Rendezvous channels
- Buffered channels
- Directional channels
- Look at channels, deadlocks, and closing channels

Channels

- Running Goroutines with "pure functions ", functions which act only upon their inputs, is clean and easy
  - One can launch thousands of pure function goroutines without concerns associated with order of execution or side affects , allowing the Go runtime to schedule the execution of the Goroutines in any order
- Order independence is not a property held by all tasks
  - Real programs often decompose into tasks that require some partial ordering
  - In other words, parts of certain tasks may be able to execute concurrently and other parts may require some ordering with other tasks
- Channels provide a way for two goroutines to communicate with one another and synchronize their execution
- A channel type is represented with the keyword chan followed by the type of the things that are passed on the channel
  - The make built-in allocates and initializes an object of type slice, map, or chan
  - var c chan string = make(chan string)
- The <- (left arrow) operator is used to send and receive messages on the channel
  - c <- "ping" means send "ping"
  - msg := <- c means receive a message and store it in msg
  - fmt.Println(<-c) means print the message received from channel c
- The example to the right uses an unbuffered channel
  - Such a channel blocks readers until a writer writes and blocks writers until a reader reads
  - This is known as a rendezvous, syncing both sides

```
1   package main
2
3   import (
4       "fmt"
5       "time"
6   )
7
8   func pinger(c chan string) {
9       for {
10          c <- "ping"
11      }
12  }
13
14  func printer(c chan string) {
15      for {
16          msg := <-c
17          fmt.Println(msg)
18          time.Sleep(time.Second * 1)
19      }
20  }
21
22  func main() {
23      var c chan string = make(chan string)
24
25      go pinger(c)
26      go printer(c)
27
28      var input string
29      fmt.Scanln(&input)
30  }
% go run chan.go
ping
ping
ping
^Csignal: interrupt
```

Multiple channel readers/writers

- Channels are thread safe
  - Many Goroutines can share a single channel
  - This allows channels to serve as work queues
  - Multiple requestors can post work to the channel and multiple worker can read work requests
  - The example configures two writers and two readers, doubling the message display rate
    - Two printers execute per second

```
% go run chan2.go
ping
pong
ping
pong
pong
ping
pong
ping
^Csignal: interrupt
```

```
1  package main
2
3  import (
4      "fmt"
5      "time"
6  )
7
8  func pinger(c chan string) {
9      for { c <- "ping" }
10 }
11
12 func ponger(c chan string) {
13     for { c <- "pong" }
14 }
15
16 func printer(c chan string) {
17     for {
18         msg := <-c
19         fmt.Println(msg)
20         time.Sleep(time.Second * 1)
21     }
22 }
23
24 func main() {
25     var c chan string = make(chan string)
26
27     go pinger(c)
28     go ponger(c)
29     go printer(c)
30     go printer(c)
31
32     var input string
33     fmt.Scanln(&input)
34 }
```

Channel Direction

- We can specify a direction on a channel type thus restricting it to either sending or receiving
- func pinger(c chan<- string)
- This restricts c to write (send) only operations
- Attempting to receive from c will result in a compiler error
- To restrict a channel to read (receive) only:
- func printer(c <-chan string)
- A channel that doesn't have these restrictions is known as bi-directional
- A bi-directional channel can be passed to a function that takes send-only or receive-only channels, but the reverse is not true

```
 1  package main
 2
 3  func pinger(c chan<- string) { for { c <- "ping" } }
 4
 5  func crash(c <-chan string) { pinger(c) }
 6
 7  func main() {
 8      var c chan string = make(chan string)
 9      go crash(c)
10  }
% go run flat.crash.go
# command-line-arguments
./flat.crash.go:5:37: cannot use c (type <-chan string)
as type chan<- string in argument to pinger
```

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "time"
 6  )
 7
 8  func pinger(c chan<- string) {
 9      for { c <- "ping" }
10  }
11
12  func ponger(c chan<- string) {
13      for { c <- "pong" }
14  }
15
16  func printer(c <-chan string) {
17      for {
18          msg := <-c
19          fmt.Println(msg)
20          time.Sleep(time.Second * 1)
21      }
22  }
23
24  func main() {
25      var c chan string = make(chan string)
26
27      go pinger(c)
28      go ponger(c)
29      go printer(c)
30      go printer(c)
31
32      var input string
33      fmt.Scanln(&input)
34  }
```

Select

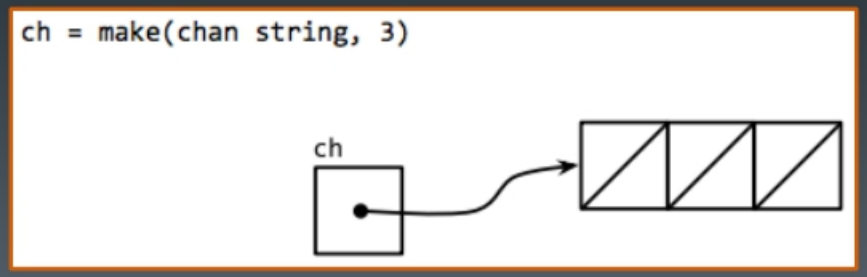
- Go has a special statement called select which allows a task to wait on multiple channels
  - Low level and network programmers will find this similar to the socket/file-descriptor select() function
  - Select picks the first channel that is ready and receives or sends from/to it
  - If more than one of the channels are ready then it randomly picks which one
  - If none of the channels are ready, the statement blocks until one becomes available
  - The select statement is often implement a timeout:
    - time.After creates a channel and after the given duration it will send the current time on it
    - Select also supports a default case which happens immediately if none of the channels are ready

```
% go run flat.select.go
start : 32
from 1 : 34
from 2 : 35
from 1 : 36
timeout : 38
from 1 : 38
from 2 : 38
from 1 : 40
from 2 : 41
from 1 : 42
timeout : 44
from 1 : 44
from 2 : 44
from 1 : 46
from 2 : 47
from 1 : 48
timeout : 50
```

```
1  package main
2
3  import "fmt"
4  import "time"
5
6  func pp(msg string) {
7     fmt.Printf("%8s : %v\n", msg, time.Now().Second())
8  }
9
10 func main() {
11    c1 := make(chan string); c2 := make(chan string)
12
13    f := func(msg string, sd time.Duration, c chan string) {
14       for {
15          time.Sleep(time.Second * sd)
16          c <- msg
17       }
18    }
19
20    pp("start")
21
22    go f("from 1", 2, c1); go f("from 2", 3, c2)
23
24    func() {
25       toLimit := 3
26       for {
27          select {
28          case msg1 := <-c1:
29             pp(msg1)
30          case msg2 := <-c2:
31             pp(msg2)
32          case <-time.After(time.Second * 2):
33             pp("timeout")
34             toLimit--
35             if toLimit <= 0 {
36                return
37             }
38          }
39       }
40    }()
41 }
```

Buffered Channels

- Normally channels are synchronous; both sides of the channel will wait until the other side is ready, forming a rendezvous
- A buffered channel allows send/receive operations to be performed asynchronously without blocking until the channel buffer is full
- A buffered channel can be created by passing a second parameter to the make function when creating a channel:
- `c := make(chan int, 1)`
- This creates a buffered channel with a capacity of 1



Determinism

- A deterministic algorithm is an algorithm which, given a particular input, will always produce the same output
  - The underlying machine always passes through the same sequence of states
- Deterministic algorithms are by far the most studied and familiar kind of algorithm, as well as one of the most practical, since they can be run on real machines efficiently
- A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs
- The ordering and scheduling of goroutines, outside of the synchronization provided by channels, is nondeterministic
  - Expecting a go routine launched first to run first is a mistake
- The example program to the right is nondeterministic
  - In most cases the one second sleep will give the other goroutines time to fill the channel buffer, but not always

Will the buffer have messages?

```
% go run chanbuf.go
0 ,
1 ,
2 ,
3 ,
4 ,
5 ,
6 ,
7 ,
8 ,
9 ,
0 ,
1 ,
2 ,
3 ,
4 ,
5 ,
6 ,
7 ,
8 ,
9 ,
100%
```

```
 1  package main
 2
 3  import "fmt"
 4  import "time"
 5
 6  func main() {
 7      counter := 0
 8      incr := 10
 9      max := 100
10      c := make(chan     int, max)
11
12      runner :=          func() {
13          for i := 0;     i < incr; i++ {
14              counter++
15              c <- i
16          }
17      }
18
19      go runner()
20      go runner()
21      time.Sleep(time.S                  // hydrate channel
22
23      for {
24                         time.Sleep(time.Millisecond * 10)
25          select {
26          case elem      := <-c:
27              fmt.Println(               ",")
28          default:
29              counter++
30
31                         fmt.Print("\033[G\033[K", counter)
32
33              if counter     == max {
34                  return
35              }
36          }
37      }
38  }
```

Deadlocks

- In concurrent computing a deadlock is a state in which each member of a group of actions, is waiting for some other member to release a lock
- Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization
- Go uses system primitives under the covers to serialize channel access
- Required to avoid data corruption
- If a goroutine ever chooses to read/write to a channel at the same time that all other goroutines are waiting on channels the program can no longer progress
- Deadlock
- The only way for a goroutine waiting on a channel to progress is for some other goroutine to send/recv a message
- If all goroutines are waiting on channels the program is effectively frozen
- The use of range in the example

```
 1  package main
 2
 3  import "fmt"
 4
 5  func main() {
 6      c := make(chan int, 50)
 7
 8      runner := func() {
 9          for i := 0; i < 5; i++ {
10              c <- i
11          }
12      }
13
14      go runner()
15      go runner()
16
17      for elem := range c {
18          fmt.Println(elem)
19      }
20  }
```

```
% go run deadlock.go
0
1
2
3
4
0
1
2
3
4
fatal error: all goroutines are asleep -
deadlock!

goroutine 1 [chan receive]:
main.main()

/Users/ronald.petty/go/src/example
s/deadlock.go:17 +0x12c
exit status 2
```

Closing Channels

- Closing a channel indicates that no more values will be sent on it
- close(c)
- This can be useful to communicate completion to the channel's receivers
- Reading from a closed channel produces the channel's Zero value and a False More flag (e.g. "",false for a string channel)
- Writing to a closed channel causes a panic
  - For this reason only single writer channels should ever be closed and only by the single writer!
- A special 2-value form of receive will set the second value to false if the channel has been closed and all values in the channel have already been received
- m, more := <- c

```
% go run closing-channels.go
sent job 1
sent job 2
sent job 3
sent all jobs
received job 1
received job 2
received job 3
received all jobs
```

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      jobs := make(chan int, 5)
7      done := make(chan bool)
8
9      go func() {
10         for {
11             j, more := <-jobs
12             if more {
13                 fmt.Println("received job", j)
14             } else {
15                 fmt.Println("received all jobs")
16                 done <- true
17                 return
18             }
19         }
20     }()
21
22     for j := 1; j <= 3; j++ {
23         jobs <- j
24         fmt.Println("sent job", j)
25     }
26     close(jobs)
27     fmt.Println("sent all jobs")
28
29     <-done
30 }
```

Channels as first class citizens

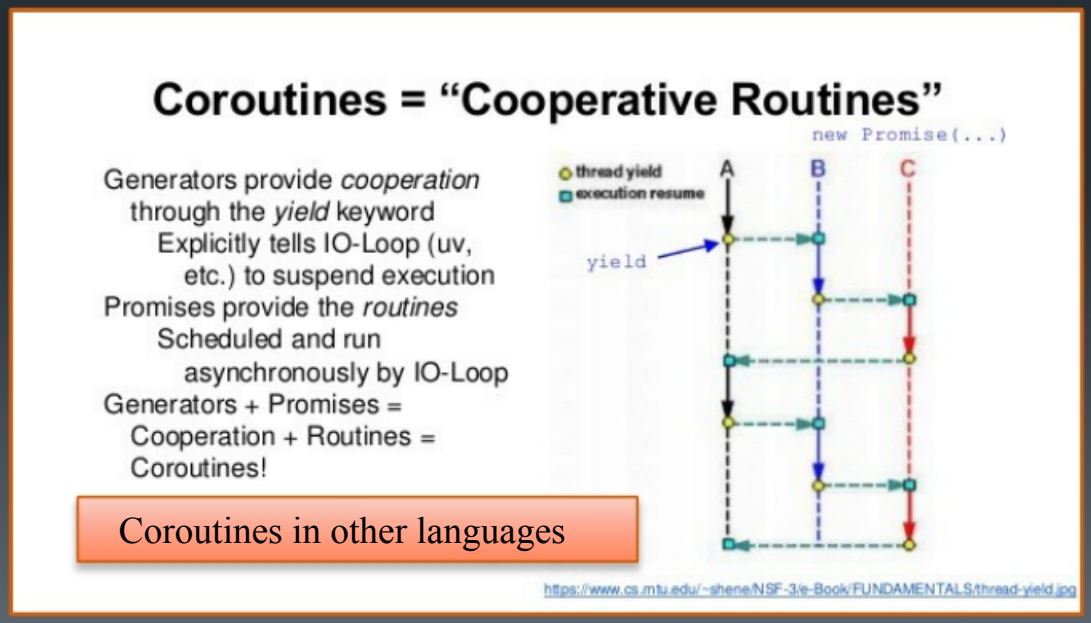
- Channels are just like other types and, like functions, can be used in the same way that other variables are used
  - Passed to functions
  - Returned from functions
  - Used as receivers in method definitions
  - Etc.

```
 1  package main
 2
 3  import "fmt"
 4
 5  func pp(role string, elem int) { if elem%3 == 0          { fmt.Println(role, elem) } }
 6
 7  func producer(msg string, out chan<- int, done           chan<- int) {
 8     for i := 0; i < 10; i++ {
 9        out <- i
10        pp(msg, i)
11     }
12
13     fmt.Println("producer finished")
14     close(out)
15  }
16
17  func commandChannel(wait int) chan int {
18     return make(chan int, wait)
19  }
20
21  func main() {
22     receiver := func(msg string, c <-chan int) {
23        for elem := range c { pp(msg, elem) }
24          done <- 0
25     }
26
27     cTypes := [2]string{"part1", "part2"}
28     wait := len(cTypes)
29     done := commandChannel(wait)
30
31     for i := 0; i < 2; i++ {
32        c := make(chan int, 100)
33
34        go receiver(cTypes[i], c)
35        go producer(cTypes[i], c, done)
36     }
37
38     //the boss is tired of waiting
39     for f := 0; f < wait; f++ { <-done }
40  }
```

```
% go run flat.chanfirst.go
part2 0
part2 3
part2 6
part1 0
part2 9
part1 0
producer finished
part1 3
part1 6
part1 9
producer finished
part1 3
```

Goroutine versus coroutines

- Coroutines
  - Program components that generalize subroutines for non-preemptive multitasking
    - Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes
    - The term coroutine was coined by Melvin Conway in 1958
  - Goroutines are are the Go twist on coroutines
    - Goroutines multiplex independently executing functions (coroutines) onto a set of threads
  - When a go/coroutine blocks the run-time automatically moves an available coroutine onto the underlying OS thread
    - This allows the program to fully utilize the timeslice assigned to the system thread
    - Eliminates an unnecessary and expensive system level context switch
    - The programmer sees none of this, which is the point
  - Goroutines are much cheaper than threads
    - They require no system call to create (the OS does not know about them)
    - They require only a user mode stack
    - They are managed completely by the Go runtime
    - No independent registers, no kernel mode stack, no kernel thread state structures, etc.
  - To make goroutine call stacks small, Go's run-time uses segmented stacks
    - A newly minted goroutine is given a few kilobytes, which is almost always enough
    - When it isn't, the run-time allocates (and frees) extension segments automatically
    - The overhead averages about three cheap instructions per function call
    - It is practical to create hundreds of thousands of goroutines in the same address space
    - If goroutines were just threads, system resources would run out at a much smaller number
  - The Go scheduler uses a parameter called GOMAXPROCS to determine how many OS threads may be actively executing Go code simultaneously
    - Default is the number of CPUs on the machine

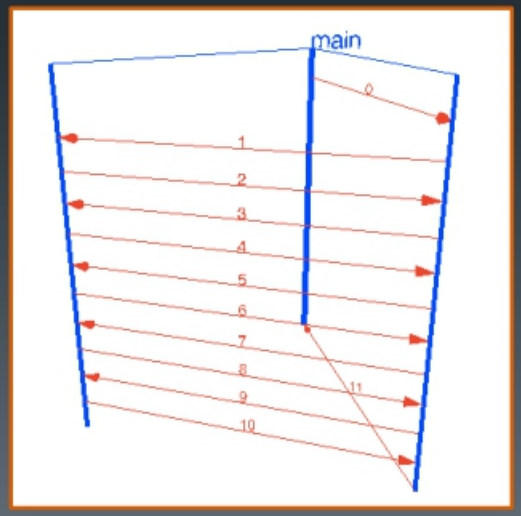


Yielding

- In Go 1 a goroutine looping forever can starve other goroutines on the same thread
- In Go 1.2 the scheduler is invoked occasionally upon entry to a function
  - Inlined functions and loops can still starve the system
- So today the Go Runtime can switch GoRoutines when:
  - A GoRoutine exits
  - A GoRoutine makes a blocking system call (e.g. read from socket)
  - A GoRoutine makes a blocking runtime call (e.g. write to chan)
  - Occasionally when calling a normal Go function
- You can yield to other goroutines by invoking any blocking call
  - time.Sleep(0)
  - runtime.Goschced()
- You can lock a GoRoutine to a specific private OS thread
  - runtime.LockOSThread()
  - Dedicates a real OS thread to this goroutine
  - You need to have a very good reason to consider using this feature

Go Concurrency Patterns

- Rob Pike talk on concurrency patterns in Go
- https://talks.golang.org/2012/concurrency.slide#1
- The Go Blog: Go Concurrency Patterns: Pipelines and cancellation
- https://blog.golang.org/pipelines
- The Go Blog: Advanced Go Concurrency Patterns
- https://blog.golang.org/advanced-go-concurrency-patterns
- Visualizing Go concurrency
- http://divan.github.io/posts/go_concurrency_visualize/



Summary

- Using go routines, channels and select Go provides powerful native first-class concurrency features without:
- Locks
- Semaphores
- Critical Sections
- Etc.
- Channels are one of Go's superpowers

Lab

Working with channels

Context

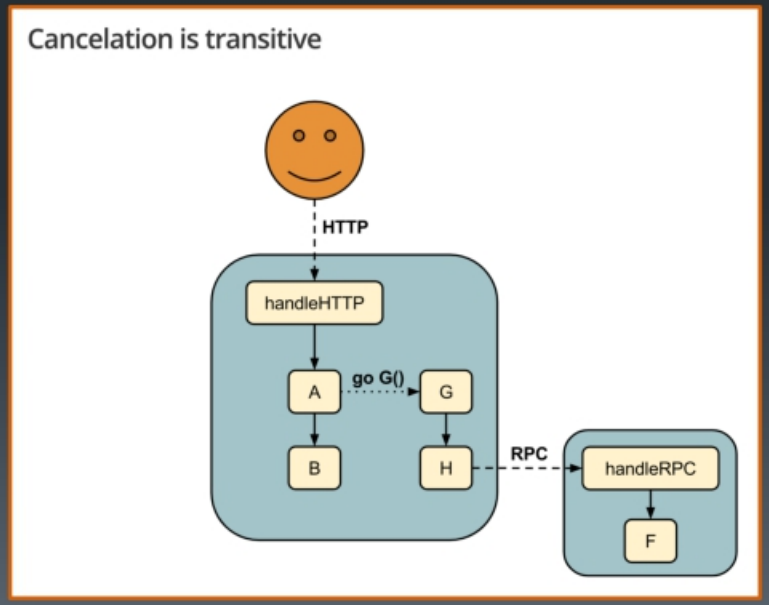
Exploring the Go standard library context package

Objectives

- Understand why we may need to control chains of code
- Explore the context package
- Examine common context use cases
- Explain the purpose of derived context
- Define context features including:
- Deadlines
- Timeouts
- Values
- List the possible context errors

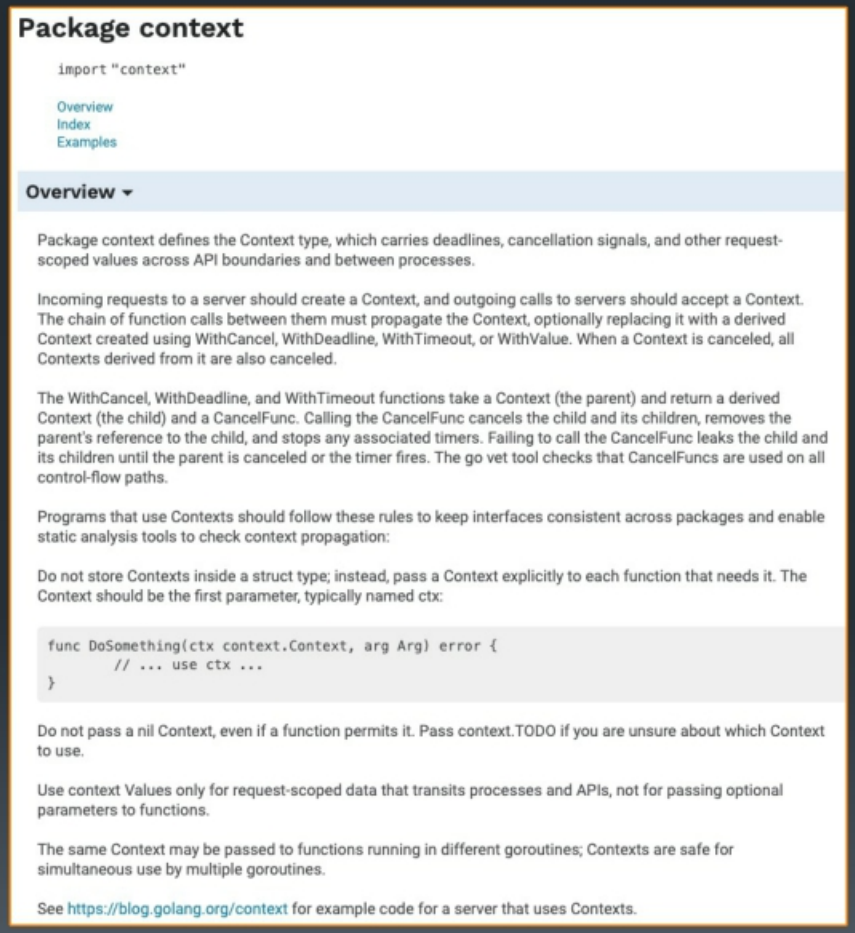
Why Context

- You might you want to cancel a (go)routine
- User navigates away
- User closes the connection
- Results from other routines make further progress in a given routine unnecessary
- This could require many independent activities to stop
- Across database calls
- Across network file systems
- Any remote service (rpc, http, …) (aka distributed system)
- Cancellation and not termination (not forced, orderly shutdown)
- You only want to stop the activities associated with the call being canceled
- A uniform approach to cancelling is desirable
- 3rd parties creating non-standard context and cancellation schemes makes it hard to stop sets of activities across multiple packages with varying approaches to context management
- Context can carry information
- User1 calls API f()
  - Token: user1ab45f
  - Limit: 10
  - Color: red
- User2 calls API f()
  - Token: user2d3b95
  - Limit: 50
  - Color: teal



Context

- Go provides a standard library “context ” package to simplify and standardize context propagation and control
  - Defines the “ Context ” type
  - Implements support for execution deadlines and cancelation within Go programs
  - The Context is propagated (usually as a call parameter called ctx) to all the functions/go-routines to be controlled by that context



Package context

```
import "context"
```

Overview
Index
Examples

Overview

Package context defines the Context type, which carries deadlines, cancellation signals, and other request-scoped values across API boundaries and between processes.

Incoming requests to a server should create a Context, and outgoing calls to servers should accept a Context. The chain of function calls between them must propagate the Context, optionally replacing it with a derived Context created using WithCancel, WithDeadline, WithTimeout, or WithValue. When a Context is canceled, all Contexts derived from it are also canceled.

The WithCancel, WithDeadline, and WithTimeout functions take a Context (the parent) and return a derived Context (the child) and a CancelFunc. Calling the CancelFunc cancels the child and its children, removes the parent's reference to the child, and stops any associated timers. Failing to call the CancelFunc leaks the child and its children until the parent is canceled or the timer fires. The go vet tool checks that CancelFuncs are used on all control-flow paths.

Programs that use Contexts should follow these rules to keep interfaces consistent across packages and enable static analysis tools to check context propagation:

Do not store Contexts inside a struct type; instead, pass a Context explicitly to each function that needs it. The Context should be the first parameter, typically named ctx:

```
func DoSomething(ctx context.Context, arg Arg) error {
        // ... use ctx ...
}
```

Do not pass a nil Context, even if a function permits it. Pass context.TODO if you are unsure about which Context to use.

Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.

The same Context may be passed to functions running in different goroutines; Contexts are safe for simultaneous use by multiple goroutines.

See https://blog.golang.org/context for example code for a server that uses Contexts.

The Context Interface

- The Context interface is at the heart of context management in the context package
- Context is thread (GoRoutine) safe
- Code can pass a single Context to any number of goroutines and cancel that Context to signal all of them
- Done() - returns a channel that acts as a cancelation signal to functions running on behalf of the Context
  - When the channel is closed, the functions should abandon their work and return
- Err() - returns an error indicating why the Context was canceled
- Deadline() – used by code to know when to stop, to set timeouts for I/O operations and to determine whether to start at all
- Value() – allows code to retrieve Context specific request-scoped data
- This data must be safe for simultaneous use by multiple goroutines

```
// A Context carries a deadline, cancelation signal, and request-scoped values
// across API boundaries. Its methods are safe for simultaneous use by multiple
// goroutines.
type Context interface {
    // Done returns a channel that is closed when this Context is canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this context was canceled, after the Done channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

Context package

```
% go doc context | tail -10
var Canceled = errors.New("context canceled")
var DeadlineExceeded error = deadlineExceededError{}

func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)

type CancelFunc func()
type Context interface {…}
```

```
% go doc context.Context | grep -v '/'

type Context interface {
        Deadline() (deadline time.Time, ok bool)
        Done() <-chan struct{}
        Err() error
        Value(key interface{}) interface{}
}
    A Context carries a deadline, a cancellation signal, and other values across
    API boundaries.

    Context's methods may be called by multiple goroutines simultaneously.

func Background() Context
func TODO() Context
func WithValue(parent Context, key, val interface{}) Context
```

Using Context (ctx) objects

- By convention, a context object is passed as the functions first argument
- Typically called "ctx "
- Each context has a Done channel
- This is closed by the cancel() function
- Contexts are hierarchical
- Contexts have no cancel method
  - You should not cancel a context created above you
  - Cancel only Context objects you create (and thus have a cancel() function for)
- Contexts can also pass values along
- Ex. Deadline, trace ids, QoS, and more
- You are not supposed to pass Application level API information via context

```
func DoSomething(ctx context.Context, arg Arg) error {
  // ... use ctx ...
}
```

Derived Context

- A client may call service (A) with a context yet A needs to call another service (B) that it would like to control with a context
- Passing no context to the B service means no one can cancel the B service call
- Passing the outer context to the B service means only the client can cancel the B service call
- Passing a new context to the B service means the call will not automatically be canceled when the outer context is canceled
- Passing a derived context to B ensures the derived context will automatically be canceled if the outer context is canceled but still allows the A service to cancel it independently if needed
- The context package provides functions to derive new Context values from existing ones
- When a Context is canceled, all Contexts derived from it are also canceled
- Background is the root of any Context tree
- The Background() function returns an empty Context
- It is never canceled, has no deadline, and has no values
- Background is typically used in main, init, and tests, and as the top-level Context for incoming service requests
  - func Background() Context
- Derived context
- Derived Context values can be canceled sooner than the parent Context
- The Context associated with an incoming request is typically canceled when the request handler returns
  - func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
  - func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
  - func WithValue(parent Context, key interface{}, val interface{}) Context

Example WithCancel


```go
package main

import "context"
import "fmt"
import "time"
import "math/rand"

func main() {
    tasks := []string{"database", "aws", "google", "salesforce"}
    ctx, cancel := context.WithCancel(context.Background())
    finished := make(chan string)
    ticker := time.NewTicker(1 * time.Second)
    rand.Seed(time.Now().UnixNano())

    for _, v := range tasks {
        go func(ctx context.Context, s string) {
            fmt.Println(s, "starting")
            for {
                select {
                case <-ctx.Done():
                    fmt.Println(s, " canceled")
                    finished <- s
                    return
                case <-ticker.C:
                    if rand.Intn(10) < 2 {
                        fmt.Println(s, " finished")
                        finished <- s
                        return
                    } else {
                        fmt.Println(s, " still running")
                    }
                }
            }
        }(ctx, v)
    }

    for cnt := len(tasks); cnt > 0; {
        select {
        case <-time.After(5 * time.Second):
            fmt.Println("timer expired: cancelling outstanding requests")
            cancel()
        case m := <-finished:
            fmt.Println(m, " completed")
            cnt--
        }
    }
    fmt.Println("All go routines complete")
}
```

```
ubuntu@ip-172-31-22-244:~$ go run can.go
salesforce starting
google starting
database starting
aws starting
salesforce still running
google still running
database still running
aws finished
aws completed
salesforce still running
google still running
database still running
salesforce still running
google still running
timer expired: cancelling outstanding requests
google canceled
salesforce canceled
database canceled
google completed
salesforce completed
database completed
All go routines complete
ubuntu@ip-172-31-22-244:~$
```

Example WithDeadline

```
package main

import (
        "context"
        "fmt"
        "time"
)

func main() {
        d := time.Now().Add(50 * time.Millisecond)
        ctx, cancel := context.WithDeadline(context.Background(), d)

        // Even though ctx will be expired, it is good practice to call its
        // cancellation function in any case. Failure to do so may keep the
        // context and its parent alive longer than necessary.
        defer cancel()

        select {
        case <-time.After(1 * time.Second):
                fmt.Println("overslept")
        case <- ctx.Done():
                fmt.Println("finished:",ctx.Err())
        }
}
```

```
~$ go run f.go
finished: context deadline exceeded
```

Passing Context on the Wire

- Context data can be passed between collaborating services in many ways
- Headers are a good choice for conveying metadata like context in systems that support headers (like HTTP/gRPC/Thrift)
- Custom context solutions are common, however
  in 2020 the W3C introduced Trace Context
  - https://www.w3.org/TR/trace-context-1/
  - This specification defines standard HTTP headers and a value format to propagate context information that enables distributed tracing scenarios
  - Describes how context information is sent and modified between services
  - Uniquely identifies individual requests in a distributed system
  - Defines a means to add and propagate provider-specific context information
- Trace context is split into two fields :
  - traceparent describes the position of the incoming request in its trace graph
  - tracestate extends traceparent with vendor-specific data represented by a set of name/value pairs

3. Trace Context HTTP Headers Format

This section describes the binding of the distributed trace context to `traceparent` and `tracestate` HTTP headers.

3.1 Relationship Between the Headers

The `traceparent` header represents the incoming request in a tracing system in a common format, understood by all vendors. Here's an example of a `traceparent` header.

```
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01
```

The `tracestate` header includes the parent in a potentially vendor-specific format:

```
tracestate: congo=t61rcWkgMzE
```

For example, say a client and server in a system use different tracing vendors: Congo and Rojo. A client traced in the Congo system adds the following headers to an outbound HTTP request.

```
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01
tracestate: congo=t61rcWkgMzE
```

Note: In this case, the `tracestate` value `t61rcWkgMzE` is the result of Base64 encoding the parent ID (`b7ad6b7169203331`), though such manipulations are not required.

The receiving server, traced in the Rojo tracing system, carries over the `tracestate` it received and adds a new entry to the left.

```
traceparent: 00-0af7651916cd43dd8448eb211c80319c-00f067aa0ba902b7-01
tracestate: rojo=00f067aa0ba902b7,congo=t61rcWkgMzE
```

You'll notice that the Rojo system reuses the value of its `traceparent` for its entry in `tracestate`. This means it is a generic tracing system (no proprietary information is being passed). Otherwise, `tracestate` entries are opaque and can be vendor-specific.

If the next receiving server uses Congo, it carries over the `tracestate` from Rojo and adds a new entry for the parent to the left of the previous entry.

```
traceparent: 00-0af7651916cd43dd8448eb211c80319c-b9c7c989f97918e1-01
tracestate: congo=ucfJifl5GOE,rojo=00f067aa0ba902b7
```

Best Practices

- Pass a Context parameter called ctx as the first argument
  - Use context Values only for request-scoped data that transits processes and APIs
  - Not for passing optional parameters to functions
  - Do not pass a nil Context, even if a function permits it
  - Pass context.TODO if you are unsure about which Context to use
  - The same Context may be passed to many goroutines, don't create multiple contexts unnecessarily
  - Contexts are safe for simultaneous use by multiple goroutines
  - Pass context to every function on the call path between incoming and outgoing requests
  - Allows Go code developed by many different teams to interoperate well
  - Provides simple control over timeouts and cancelation
  - Ensures that critical values like security credentials transit Go programs properly
  - Do not store Contexts inside a struct type
  - Pass a Context explicitly to each function that needs it
  - Server frameworks that want to build on Context should provide implementations of Context to bridge between custom package features and those that expect a standard Go Context parameter
  - Client libraries should accept a Context from the calling code
  - By establishing a common interface for request-scoped data and cancelation, Context makes it easier for package developers to share code for creating scalable services

The context package is being considered for widespread use in Go 2.0
Go 1.16 presently uses context in 100 of the 4388 standard library go files
Tools exist to help retrofit existing code to use context
e.g. https://github.com/golang/tools/blob/˚master/cmd/eg/eg.go

Summary

- The Context package provides a standard way for Go programmers to:
  - Cancel operations throughout a call graph, in particular, across go routines
  - Communicate context specific values through a call graph
- The Context package standardizes this commonly required activity and is being used by the go library itself
  - Context objects can be created with:
  - Timeouts/Deadlines
  - Cancelability
  - Key/Value variables

Lab

- Working with Context objects and concurrent routine cancelation

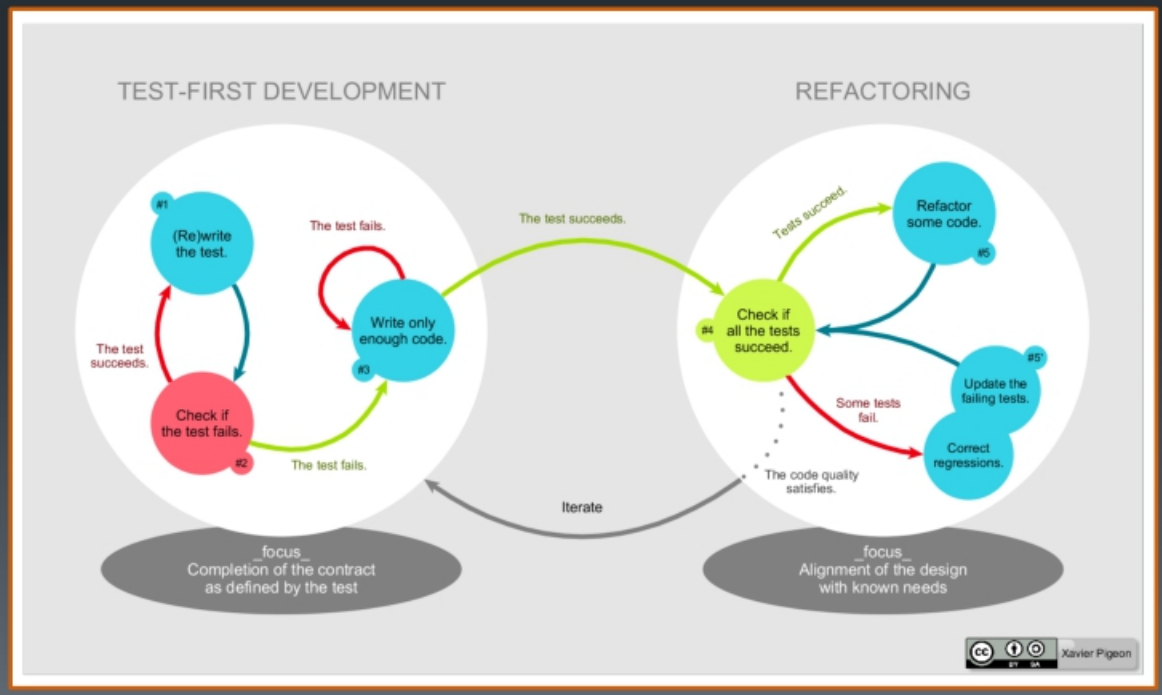
Testing

Objectives

- TDD and BDD
- Concepts
- Examples
- “go test”
- usage
- Package “testing”
- Required syntax
- Examples

Test Driven Development (TDD)

- TDD is a software development process that relies on very short development cycles
- Tests are written first
- Then code is added to make the test pass
- TDD appeared 1999 related to extreme programming (by Kent Beck)
- TDD lifecycle
- Add a test
- Run all test and see if the new test fails
- Write the code
- Run tests to see if new tests pass
- Refactor code
- Repeat
- Tests are generally small and self documenting
- Called “unit tests”
- TDD is not the same as acceptance test driven development (ATDD)
- ATDD is where customer level functionality is tested, often not automated
- ATDD provides guidance on what should be an automated TDD test



TDD Practices

- Individual Tests
- Move setup/teardown into a service (reduce duplication of code)
- Keep focus of each test **oracle** (don't have single oracle across multiple test domains – aka. Source of truth)
- Allow for skew in time based tests (+/-)
- Treat test code like production code

- Anti-patterns (Don't do)
- Depend on previous test results for current test
- Test order should matters
- Not testing time for time dependent code
- All in one oracle
- Testing implementation versus interface
- Leaving slow tests


```
func TestLoop(t *testing.T) {
  cases := []struct {
     in int
     want int
  } {
      {-1,1},
      {3,6},
      {0,0},
  }

   for _, c := range cases {
      got := loop(c.in)
      if got != c.want {
        t.Errorf("loop(%d) == %d,
            want %d", c.in, got, c.want)
      }
   }
}
```

Behavior Driven Development (BDD)

- BDD combines TDD and ATDD
- TDD – write tests first
- ATDD – focus on behavior over implementation
- BDD emerges from TDD
- Allows for management and development to collaborate on what the behavior is for a given program and how to test it
- BDD is mostly facilitated via a domain specific language (DSL)
- BDD uses a combination of unit test (TDD) with acceptance test (ATDD)
- A TDD unit test is often non-specific, BDD focuses the test on desired behavior (aka business goal)
- BDD is considered to be at odds with Go design. The addition of a DSL is considered unnecessary complexity.
- BDD support for Go is provided by 3rd parties
- ginkgo - https://onsi.github.io/ginkgo/

BDD style – ex. Gingko

```
...
var _ = Describe("Book", func() {
    var (
        longBook  Book
        shortBook Book
    )

    BeforeEach(func() {
        longBook = Book{
            Title:  "Les Miserables",
            Author: "Victor Hugo",
            Pages:  1488,
        }

        shortBook = Book{
            Title:  "Fox In Socks",
            Author: "Dr. Seuss",
            Pages:  24,
        }
    })

    Describe("Categorizing book length", func() {
        Context("With more than 300 pages", func() {
            It("should be a novel", func() {
                Expect(longBook.CategoryByLength()).To(Equal("NOVEL"))
            })
        })

        Context("With fewer than 300 pages", func() {
            It("should be a short story", func() {
                Expect(shortBook.CategoryByLength()).To(Equal("SHORT STORY"))
            })
        })
    })
})
```

```
$ ginkgo # or go test
=== RUN TestBootstrap

Running Suite: Books Suite
==========
Random Seed: 1378938274

Will run 2 of 2 specs

..
Ran 2 of 2 Specs in 0.000 seconds
SUCCESS! -- 2 Passed | 0 Failed | 0 Pending | 0 Skipped

--- PASS: TestBootstrap (0.00 seconds)
PASS
ok books 0.025s
```

Package “testing”

- Package testing provides support for automated testing of Go packages
- Used in concert with the “ go test” command
- Provides three primary function types
- Test – code correctness
- Benchmark – code timing
- Example – code examples
- Additional functionality include:
- Subtests and Sub-benchmarks
- Parallelism
- “go test” looks for files that have a suffix “_test.go” and belong to the package of code being tested.

Using Test

- To create an automated test (one ran by "go test")
- Create *name* _test.go (*name* is your program name)
- Define the test
  - Setup
  - Execute
  - Teardown
- Run the test "go test"
  - Fail
  - Create *name*.go (name is your program name)
    - Implement the code
  - Rerun
  - Fix
  - Repeat
- "go test" test functions have names of the form: "Test *Name* "
- "*testing.T" is the parameter used by the test driver (to control the test state)
  - Report failures through the instance of "t *testing.T"

```
user@ubuntu : ~/go/src/gonuts $ go test -run=TestLoop
PASS
ok  gonuts  0.002s
user@ubuntu : ~/go/src/gonuts $
```

loop.go

```
user@ubuntu:~/go/src/gonuts$ cat loop.go
package main


func loop(iter int) int {
   i := 0
   total := 0
   for i <= iter {
      total += i
      i++
   }
   return total
}
user@ubuntu:~/go/src/gonuts$
```

loop_test.go

```
package main

import "testing"

func TestLoop(t *testing.T) {
   cases := []struct {
      in int
      want int
   } {
      {-1,1},
      {3,6},
      {0,0},
   }

   for _, c := range cases {
      got := loop(c.in)
      if got != c.want {
         t.Errorf("loop(%d) == %d,
            want %d", c.in, got, c.want)
      }
   }
}
```

Using Benchmark

- Place benchmark code in *NAME* _test.go (can be same as TestFunc)
- Instead of Test prefix, use Benchmark prefix
- Like Test, context is managed through an object of type "*testing.B"
- In the example "b.N" is calculated by "go test" to find a decent high mark on the number of times to execute the test
- To run, we need to supply the "-bench=*REGEX*" flag
- "." means run anything with "Benchmark" in function name prefix

```
func BenchmarkLoop(b *testing.B) {
    for i := 0; i < b.N; i++ {
        loop(1000)
    }
}
```

```
user@ubuntu : ~/go/src/gonuts $ go test -bench=BenchmarkLoop
BenchmarkLoop-2          5000000               307 ns/op
PASS
ok    gonuts 1.891s
user@ubuntu : ~/go/src/gonuts $
```

Using Example

- Similar to Test and Benchmark, prefix example code (used in documentation) with Example

```
func ExampleFLoop() {
    fmt.Println(loop(3))
    // Output: 6
}
```

- When generating documentation via “godoc”
- If no return value, it will compile
- If return value, it will compile and execute
  - If there is output, using “// Output: 6” will compare to the actual result

- To assist godoc with doc generation we add a suffix to Example:
- Nothing (top level package)
- F function
- T type
- M method

- When used with “go test” it will test our example code!

Fuzzing

- Fuzzing is a type of automated testing which continuously manipulates inputs to a program to find issues such as panics, bugs, or data races to which the code may be susceptible.
- These semi-random data mutations can discover new code coverage that existing unit tests may miss, and uncover edge-case bugs which would otherwise go unnoticed.
- This type of testing works best when able to run more mutations quickly, rather than fewer mutations intelligently.

- Fuzzing was introduced in 1.18
- https://go.googlesource.com/proposal/+/master/design/draft-fuzzing.md
- https://github.com/golang/go/issues/44551
- More on the topic:
- https://owasp.org/www-community/Fuzzing

Using Fuzzing

- A testing technique where a function is called with randomly generated inputs to find bugs not anticipated by unit tests
- func FuzzXxx(*testing.F)
- A provide seed ‘corpus’ and generated one are used to vary inputs (seed prevents regression)
- Fuzzing requires direct selection and termination
- if no errors are found
- https://github.com/golang/go/issues/48127 (more to do in 1.19)

```
func FuzzHex(f *testing.F) {
        for _, seed := range [][]byte{{}, {0}, {9}, {0xa}, {0xf}, {1, 2, 3, 4}} {
            f.Add(seed)
        }
        f.Fuzz(func(t *testing.T, in []byte) {
            enc := hex.EncodeToString(in)
            out, err := hex.DecodeString(enc)
            if err != nil {
                t.Fatalf("%v: decode: %v", in, err)
            }
            if !bytes.Equal(in, out) {
                t.Fatalf("%v: not equal after round trip: %v", in, out)
            }
        })
    }
```

Setup and Teardown

- If a test suite requires setup or teardown; you provide a function definition as follow:
  `func TestMain(m *testing.M) { ... }`
- If TestMain exists, the test runner will execute this function over directly executing tests specified in file
- Trigger tests via call to m.Run()
- TestMain can pass exit code to os.Exit
- Parse arguments via flag.Parse manually; normally done by test runner

```
~$ go doc testing.M
package testing // import "testing"

type M struct {
        // Has unexported fields.
}
    M is a type passed to a TestMain function to run
the actual tests.

func MainStart(deps testDeps, tests []InternalTest,
benchmarks []InternalBenchmark, ...) *M
func (m *M) Run() (code int)
```

```
$ cat g_test.go
package g

import (
        "os"
        "fmt"
        "testing"
)

func TestSomething(t *testing.T) {
        t.Skip("later")
}

func setup() {
        fmt.Println("setting up")
}

func teardown() {
        fmt.Println("tearing down")
}

func TestMain(m *testing.M) {
        setup()
        result := m.Run()
        teardown()
        os.Exit(result)
}
~/g$ go test -v g_test.go
setting up
=== RUN   TestSomething
    g_test.go:10: later
--- SKIP: TestSomething (0.00s)
PASS
tearing down
ok      command-line-arguments  0.002s
```

```
$ cat g_test.go
package g

import (
                "os"
                "fmt"
                 "testing"
                 "flag"
)

func init() {
  if flag.Lookup("test.v") == nil {fmt.Println("normal run") }
  else { fmt.Println("run under go test") }
}

func TestSomething(t *testing.T) {
  if x := flag.Lookup("test.v"); x != nil {fmt.Println("don't test me!")}
                t.Skip("later")
}

func setup() { fmt.Println("setting up") }

func teardown() { fmt.Println("tearing down") }

func TestMain(m *testing.M) {
                setup()
                result := m.Run()
                teardown()
}               os.Exit(result)


~/g$ go test -v g_test.go
normal run
setting up
=== RUN   TestSomething
don't test me!
      g_test.go:22: later
--- SKIP: TestSomething (0.00s)
PASS
tearing down
ok              command-line-arguments          0.002s
```

Detecting Test Runs

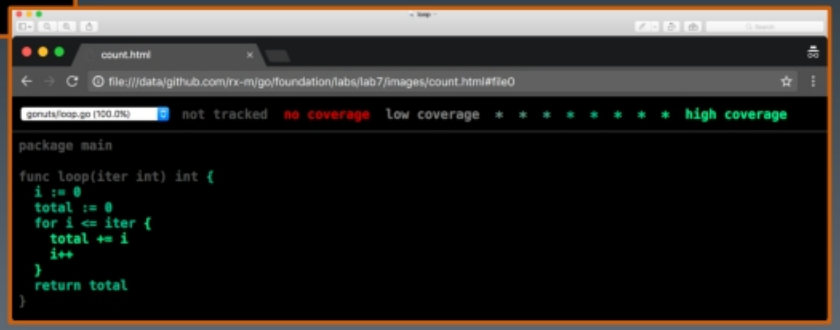
- The testing package modifies the global environment when loaded
  - Registering command-line flags etc.
  - You can determine if you are running under test by

    checking for the “test.v” command line flag

Coverage

- Code coverage is a measure to describe the degree of source code executed as part of a test
- To run "go test -coverprofile=output.txt"
- If you supply the "-covermode=*mode* " to change behavior
- set – does this statement run (bool check)
- count – how many times does this statement run (int check)
- atomic – count for multi-threaded applications (int check, more expensive)

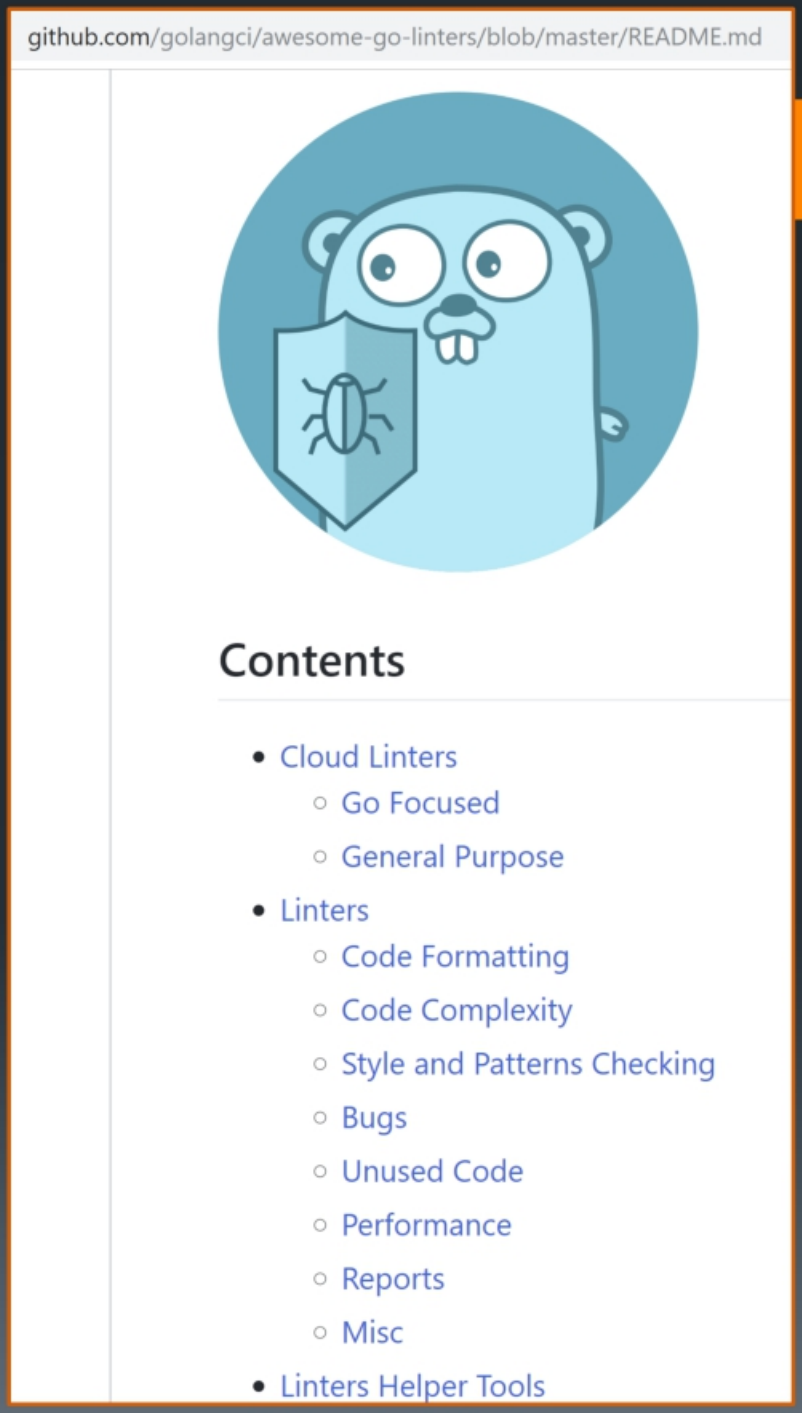
```
$ go help testflag

$ go test -coverprofile=cover.out
PASScoverage: 75.0% of statements
ok                  gonuts          0.002s

$ go tool cover -html=count.out -o count.html
```



Linter Tools

- Cloud Linters
- Go Report Card - Go repo report card
  - https://goreportcard.com/

- GolangCI - Open Source SaaS service for running linters on Github pull requests. Free for Open Source
  - https://golangci.com/ (dead, but!)
    - https://github.com/golangci/golangci-lint-action



Host based linting

- Code Formatting
  - gofmt - Gofmt formats Go programs. Must have for every project. Don't forget to use -s flag.
  - gofumpt - The tool is a modified fork of gofmt, enforcing a stricter format than gofmt, while being backwards compatible.
  - goimports - Goimports does everything that gofmt does. Additionally it checks unused imports.
  - dedupimport - Fix duplicate imports that have the same import path but different import names.
  - unindent - Report code that is unnecessarily indented
- Code Complexity
  - abcgo - ABC metrics for Go source code.
  - depth - Count the maxdepth of go functions. It's helpful to see if a function needs to be splitted into several smaller functions, for readability purpose.
  - funlen - linter that checks for long functions. It can check both the number of lines and the number of statements.
  - gocyclo - Computes and checks the cyclomatic complexity of functions.
  - nakedret - nakedret is a Go static analysis tool to find naked returns in functions greater than a specified function length.
  - splint - It finds any functions that are too long or have too many parameters or results.

Style and pattern checking

- dogsled - Finds assignments/declarations with too many blank identifiers.
- dupl - Tool for code clone detection.
- go-cleanarch - go-cleanarch was created to validate Clean Architecture rules, like a The Dependency Rule and interaction between packages in your Go projects.
- go-consistent - source code analyzer that helps you to make your Go programs more consistent.
- go-namecheck - source code analyzer that helps you to maintain variable/field naming conventions inside your project.
- gochecknoinits - Find init functions, to reduce side effects in code.
- gochecknoglobals - Find global vars, to reduce side effects in code.
- goconst - Find in Go repeated strings that could be replaced by a constant.
- GoLint - Golint is a linter for Go source code.
- gosimple - gosimple is a linter for Go source code that specialises on simplifying code.
- impi - Verify imports grouping and ordering.
- interfacer - Linter that suggests narrower interface types.
- lll - Line length linter, used to enforce line length in files.
- misspell - Finds commonly misspelled English words
- nofuncflags - disallow boolean params to functions (flags).
- predeclared - Find code that shadows Go's built-in identifiers (e.g., append, copy, int).
- revive - ~6x faster, stricter, configurable, extensible, and beautiful drop-in replacement for golint
- unconvert - Remove unnecessary type conversions from Go source.
- usedexports - Find in Go exported variables that could be unexported.
- whitespace - Checks for unnecessary newlines at the start and end of functions

Bugs and Unused Code

- Bugs
  - bodyclose - checks whether HTTP response body is closed and a re-use of TCP connection is not blocked
  - durcheck - durcheck is a very simple linter which detects potential bugs with time.Duration in a Go package.
  - errcheck - Errcheck is a program for checking for unchecked errors in Go programs.
  - gas - Inspects source code for security problems by scanning the Go AST.
  - go vet - Vet examines Go source code and reports suspicious constructs, such as Printf calls whose arguments do not align with the format string. Can check shadowing of variables, but must be enabled explicitly.
  - gosumcheck - Checks all possible cases of type-switch are handled.
  - go-sumtype - Checks all possible cases of type-switch are handled.
  - mulint - Go lint which detects recursive locks, which may lead to dead locks.
  - safesql - Static analysis tool for Golang that protects against SQL injections.
  - scopelint - scopelint checks for unpinned variables in go programs.
  - sqlrows - checks whether Close on sql.Rows is called.
  - staticcheck - staticcheck is go vet on steroids, applying a ton of static analysis checks you might be used to from tools like ReSharper for C#.
- Unused Code
  - deadcode - Finds unused code.
  - ineffassign - Detect when assignments to existing variables are not used.
  - structcheck - Find unused global variables and constants.
  - unparam - Report unused function parameters.
  - unused - unused checks Go code for unused constants, variables, functions and types.
  - varcheck - Find unused global variables and constants.

Others

- Performance
  - Copyfighter - Statically analyzes Go code and reports functions that are passing large structs by value.
  - maligned - Tool to detect Go structs that would take less memory if their fields were sorted.
  - prealloc - Find slice declarations that could potentially be preallocated.
  - rangerdanger - Tool to detect range statements iterating over addressable arrays
- Reports
  - flen - Get info on length of functions in a Go package.
  - GoReporter - A Golang tool that does static analysis, unit testing, code review and generate code quality report.
  - golinters - golinters generates HTML reports about Go linters.
- Misc
  - go-outdated - Console application that displays outdated packages.
  - go-template-lint - go-template-lint is a linter for Go text/template (and html/template) template files.
  - godox - Find all TODO/FIXME comments.
  - lingo - Set of specific checks.
  - megacheck - megacheck runs staticcheck, gosimple and unused at once. Because it is able to reuse work, it will be faster than running each tool separately.
  - go-critic - source code linter that brings checks that are currently not implemented in other linters.
  - tarp - tarp finds functions and methods without direct unit tests in Go source code.
  - go-mnd - Magic number detector for Go.

Food for thought

Per Spencer Baugh at http://catern.com/run.html

When developing a system, it is important to be able to run the system in its entirety.

- "Run the unit tests" doesn't count. The complexity of your system is in the interactions between the units.
- "Run an individual service against mocks" doesn't count. A mock will rarely behave identically to the real dependency, and the behavior of the individual service will be unrealistic. You need to run the actual system.
- "Run an individual service in a shared stateful development environment running all the other services" doesn't count. A shared development environment will be unreliable as it diverges more and more from the real system.
- "Run most services in a mostly-isolated development environment, calling out to a few hard-to-run external services " doesn't count. Those few external services on the edge of the mostly-isolated development environment are often the most crucial ones; without the ability to run modified versions of them, your development process is crippled. Furthermore, being dependent on external services greatly complicates where and how you can run the system; it's much harder to, for example, run tests with the system on every commit if that will access external services.
- "Run all the services that make up the system in an isolated development environment" counts; it's the bare minimum requirement. Bonus points if this can be done completely on localhost, without using an off-host cluster deployment system.
- ... and more ...

Summary

- Go provides a basic TDD package called “testing”
- Exclusion of BDD DSL was by design
- Core Go engineers believe to much TDD abstraction or BSD DSL hides valuable details
- Many additional flags and methods are available
- Simple at first (by design)
- Ability to add your own testing practices on top

Lab: Testing

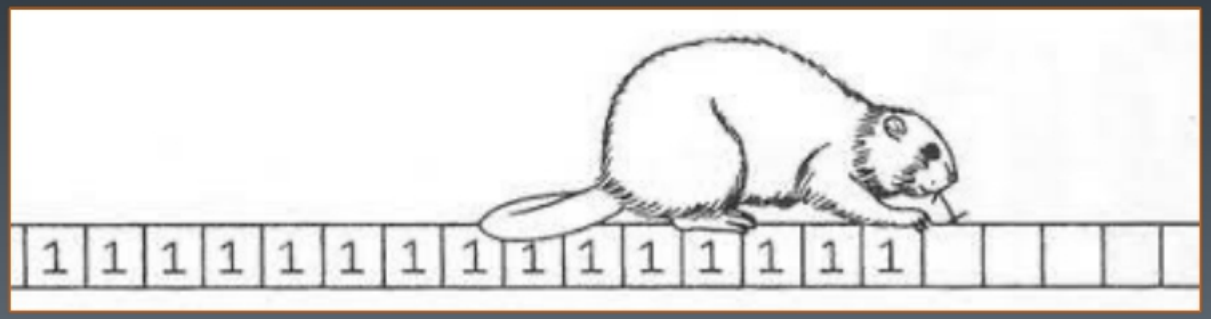
- Develop basic tests and compare performance against algorithms that produce the same output but differ in implementation.

Performance

Profiling, Tracing, Debugging, and more…

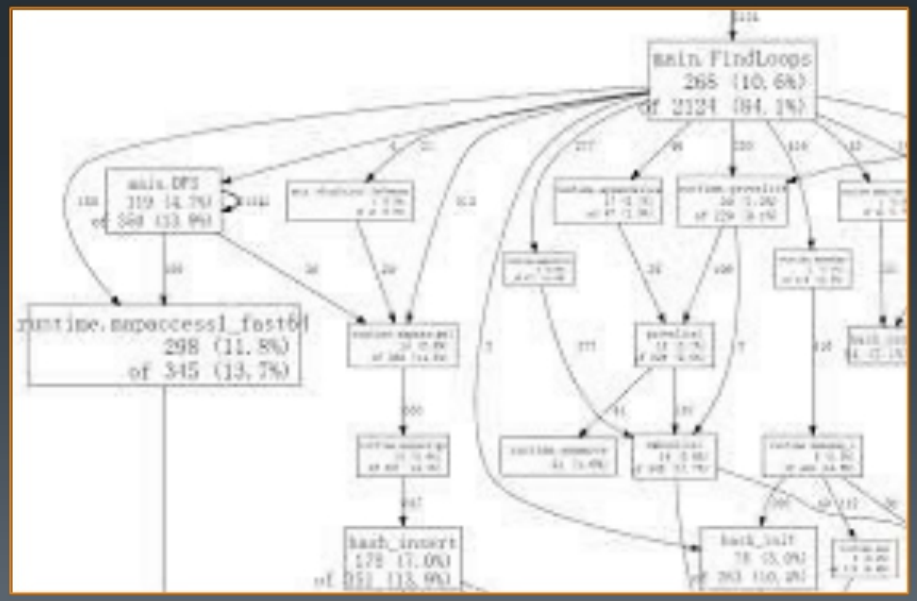
Objectives

- General goals include:
- Resolving performance issues
- Locating memory leaks
- Resolving resource contention
- Consider the various factors when performing diagnostics
- Profiling – measuring resources
  - Explore Go's built in profiling features
  - pprof
- Tracing – understanding threads of execution
- Debugging –making the illogical become logical
  - Explain the challenges of debugging Go programs
  - List some Go Debugging tools
  - Describe the use of Delve



What is Profiling

- Use of statistical profiling
- Ex. Interruptions at fixed intervals
- At each profiling interval we could
- Dump function call stack
- Stats on memory
- At the end of a profiling session
  - We can count the number of function calls or allocations of memory



Profiling in Go

13

- Go ships with a profiling tool and a profiling package:
- go tool pprof # for analyzing profiles
- Package runtime/pprof
- The pprof package also provides a visualization tool
  - The profiler is continually improved so using the latest version of Go pays dividends
  - Go 1.8 adds mutex profiles so you can see mutex contention
  - Go 1.11 adds a new "allocs" profile type that profiles total number of bytes allocated since the program began (identical to the existing "heap" profile viewed in -alloc_space mode)
  - https://github.com/google/pprof
  - pprof lets you analyze CPU profiles, traces, and heap profiles for your Go programs
  - pprof is a tool for visualization and analysis of profiling data~

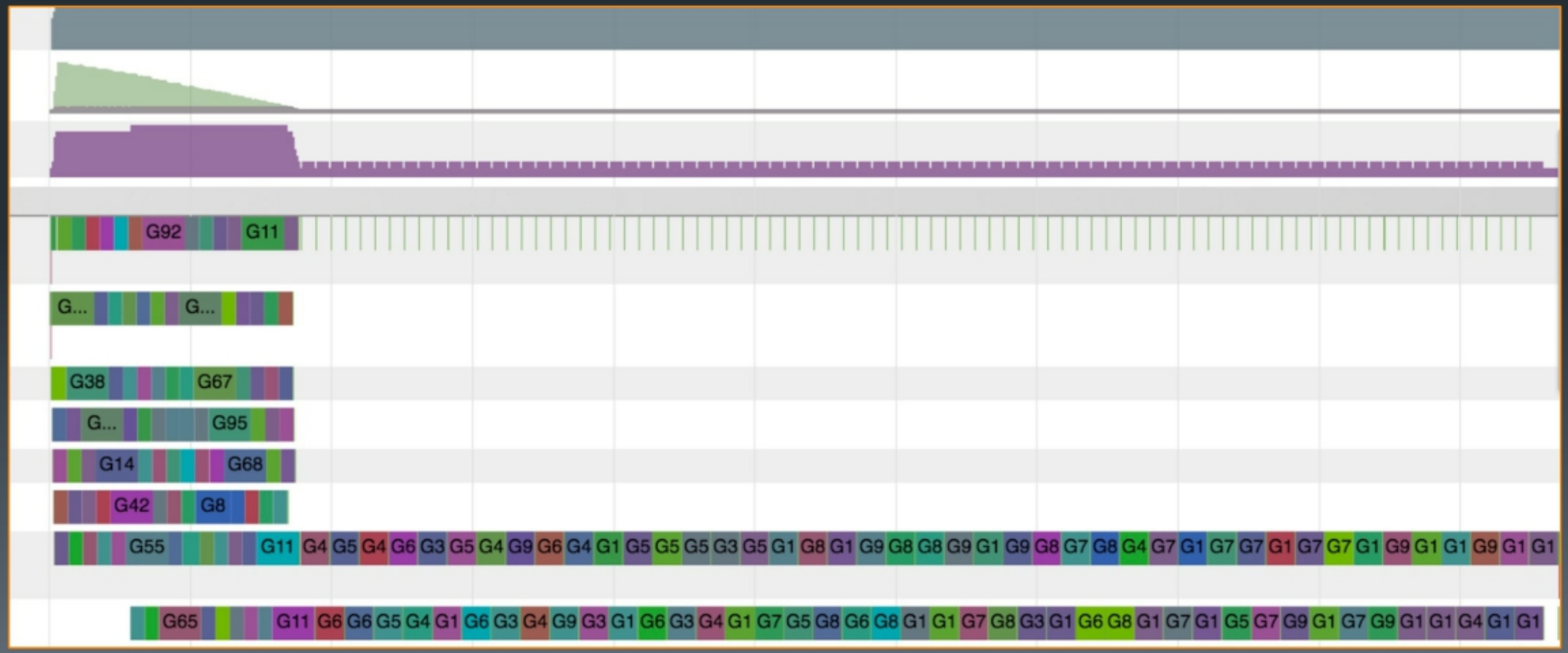
Profiling Details

- pprof generates and analysis data stored in a file known as a profile
- protocol buffers is the file format (ex.profile.proto)
- Data includes sampled callstacks, statistical polling information
- Format - https://github.com/google/pprof/blob/master/proto/profile.proto
- Profiles can be read from a local file or http
- Profiles can be aggregated or compared
- Some profile data can be further investigated by `go tools`
  - addr2line and nm (or native versions of them)
- CPU
- Interrupts triggered via SIGPROF
- At intervals of 10 ms

```
% go tool pprof -h
usage:

Produce output in the specified format.

 pprof <format> [options] [binary] <source> ...

Omit the format to get an interactive shell whose commands can be used
to generate various views of a profile

 pprof [options] [binary] <source> ...

Omit the format and provide the "-http" flag to get an interactive web
interface at the specified host:port that can be used to navigate through
various views of a profile.

 pprof -http [host]:[port] [options] [binary] <source> ...

Details:
 Output formats (select at most one):
   -callgrind       Outputs a graph in callgrind format
   -comments        Output all profile comments
   -disasm          Output assembly listings annotated with samples
   -dot             Outputs a graph in DOT format
   -eog             Visualize graph through eog
   -evince          Visualize graph through evince
   -gif             Outputs a graph image in GIF format
   -gv              Visualize graph through gv
   -kcachegrind     Visualize report in KCachegrind
  -list             Output annotated source for functions matching regexp
  ...
```

What is Tracing

- Tracing traces a line of execution
- Tracing is implemented via instrumentation
- Can be done at compile (rewrite), runtime (inject), or hardware
- Can also trace via external tools (ex. strace)



Tracing in Go

- Package " runtime/trace" contains facilities for programs to generate traces for the Go execution tracer
- import "runtime/trace
- The execution trace captures a wide range of execution events:
- Goroutine lifecycle events like create/block/unblock
- syscall enter/exit/blocked
- GC events
- size of heap changes
- process start/stop

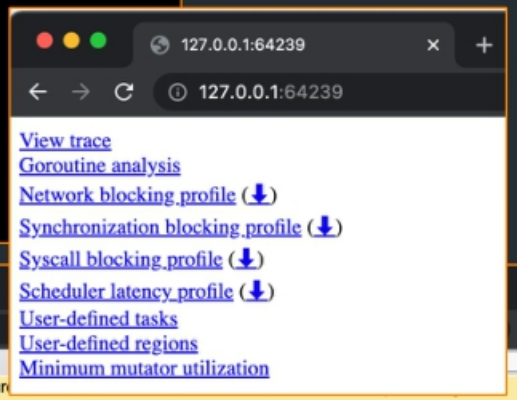
- Run via testing
- go test -trace=test.out

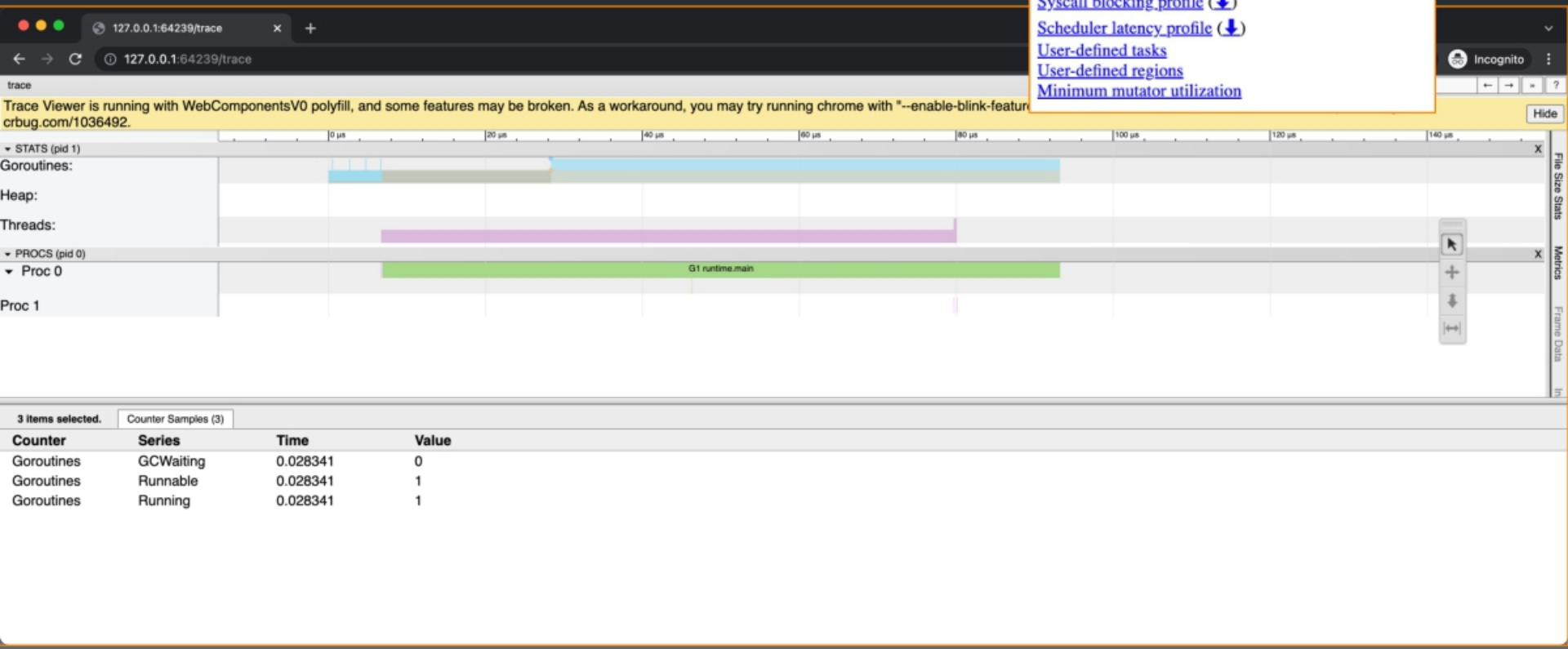
Tracing Example


```
package main

import (
        "fmt"
        "log"
        "os"
        "runtime/trace"
)

// Example demonstrates the use of the trace package to trace
// the execution of a Go program. The trace output will be
// written to the file trace.out
func main() {
        f, err := os.Create("trace.out")
        if err != nil {
                log.Fatalf("failed to create trace output file: %v", err)
        }
        defer func() {
                if err := f.Close(); err != nil {
                        log.Fatalf("failed to close trace file: %v", err)
                }
        }()

        if err := trace.Start(f); err != nil {
                log.Fatalf("failed to start trace: %v", err)
        }
        defer trace.Stop()

        // your program here
        RunMyProgram()
}

func RunMyProgram() {
        fmt.Printf("this function will be traced")
}
```

Example Tracing Session


```
% go build trace.go

% ./trace
this function will be traced

% go tool trace trace.out
2022/02/12 16:05:29 Parsing trace...
2022/02/12 16:05:29 Splitting trace...
2022/02/12 16:05:29 Opening browser. Trace viewer is listening on
http://127.0.0.1:64239
```





Debugging

- Debugging is the process of finding and resolving of defects that prevent correct operation of computer software or a system
  --wikipedia
- Debugging tactics can involve:
- Interactive debugging
  - In Go: gdb or dlv
- Control flow analysis:
  - Trace
- Unit testing
  - Go Test
- Integration testing
  - Go Test
- Log file analysis
  - Package “log”
- Monitoring
  - Various external tools
- Memory dumps
  - gdb or dlv
- Profiling
  - Pprof

GDB

- The GNU Project debugger
- Allows you to see what is going on `inside' another program while it executes
- -- or what another program was doing at the moment it crashed
- GDB can do four main kinds of things:
- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
- The program being debugged can be written in Ada, C, C++, Objective-C, Pascal, Go and many other languages
- Those programs might be executing on the same machine as GDB (native) or on another machine (remote)
- GDB can run on most popular UNIX and Microsoft Windows variants
- GDB does not understand Go programs well
- Go stack management, threading, and the Go runtime differ from the execution model GDB expects
  - This is true even when the program is compiled with gccgo
- GDB can be useful in some situations but it is not a reliable debugger for Go programs
  - It falls particularly short in heavily concurrent debugging because of its lack of recognition of goroutines
- A more Go-centric debugging architecture was required

Delve

- Delve (dlv) is a debugger for the Go programming language
- https://github.com/go-delve/delve
- The goal of the project is to provide a simple, full featured debugging tool for Go
- Delve is easy to invoke and easy to use
- Delve is a "go install" able Go executable package
- go install github.com/go-delve/delve/cmd/dlv@latest
- Can be executed at the command line using dlv or through IDEs
- Visual Studio Code, GoLand, etc. all use dlv under the covers

DELVE

A Debugger for the Go Programming Language

Delve Command Line


```
% $(go env GOPATH)/bin/dlv version
Delve Debugger
Version: 1.8.1
Build: $Id: d85f1f6b736db99a1c239c34fd4a081dcff08a3c $

% $(go env GOPATH)/bin/dlv help
Delve is a source level debugger for Go programs.

Delve enables you to interact with your program by controlling the execution of the process,
evaluating variables, and providing information of thread / goroutine state, CPU register state and more.

The goal of this tool is to provide a simple yet powerful interface for debugging Go programs.

Pass flags to the program you are debugging using `--`, for example:

`dlv exec ./hello -- server --config conf/config.toml`

Usage:
  dlv [command]

Available Commands:
  attach      Attach to running process and begin debugging.
  connect     Connect to a headless debug server with a terminal client.
  core        Examine a core dump.
  dap         Starts a headless TCP server communicating via Debug Adaptor Protocol (DAP).
  debug       Compile and begin debugging main package in current directory, or the package specified.
  exec        Execute a precompiled binary, and begin a debug session.
  help        Help about any command
  run         Deprecated command. Use 'debug' instead.
  test        Compile test binary and begin debugging program.
  trace       Compile and begin tracing program.
  version     Prints version.

Flags:
      --accept-multiclient               Allows a headless server to accept multiple client connections via JSON-RPC or DAP.
      --allow-non-terminal-interactive   Allows interactive sessions of Delve that don't have a terminal as stdin, stdout and stderr
      --api-version int                  Selects JSON-RPC API version when headless. New clients should use v2. Can be reset via RPCServer.SetApiVersion. See Documentation/api/json-rpc/README.md. (default 1)
      --backend string                   Backend selection (see 'dlv help backend'). (default "default")
      --build-flags string               Build flags, to be passed to the compiler. For example: --build-flags="-tags=integration -mod=vendor -cover -v"
      --check-go-version                 Exits if the version of Go in use is not compatible (too old or too new) with the version of Delve. (default true)
      --disable-aslr                     Disables address space randomization
      --headless                         Run debug server only, in headless mode. Server will accept both JSON-RPC or DAP client connections.
  -h, --help                             help for dlv
      --init string                      Init file, executed by the terminal client.
  -l, --listen string                    Debugging server listen address. (default "127.0.0.1:0")
      --log                              Enable debugging server logging.
      --log-dest string                  Writes logs to the specified file or file descriptor (see 'dlv help log').
      --log-output string                Comma separated list of components that should produce debug output (see 'dlv help log')
      --only-same-user                   Only connections from the same user that started this instance of Delve are allowed to connect. (default true)
  -r, --redirect stringArray             Specifies redirect rules for target process (see 'dlv help redirect')
      --wd string                        Working directory for running the program.

Additional help topics:
  dlv backend     Help about the --backend flag.
  dlv log         Help about logging flags.
  dlv redirect    Help about file redirection.

Use "dlv [command] --help" for more information about a command.
```

DelveSession

```
% pwd
/usr/local/go/src/net

net % sudo ~/go/bin/dlv test -- -test.run ^TestReadUnixgramWithUnnamedSocket
Type 'help' for list of commands.
(dlv) b main.main
Breakpoint 1 set at 0x133a4cf for main.main() _testmain.go:623
(dlv) c
> main.main() _testmain.go:623 (hits goroutine(1):1 total:1) (PC: 0x133a4cf)
(dlv) l
> main.main() _testmain.go:623 (hits goroutine(1):1 total:1) (PC: 0x133a4cf)
Command failed: open _testmain.go: no such file or directory
(dlv) b TestReadUnixgramWithUnnamedSocket
Breakpoint 2 set at 0x13070f2 for net.TestReadUnixgramWithUnnamedSocket()  ./unixsock_test.go:20
(dlv) c
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:20 (hits goroutine(35):1  total:1) (PC: 0x13070f2)
    15:                                     "syscall"
    16:                                     "testing"
    17:                                     "time"
    18:             )
    19:
=>  20:             func TestReadUnixgramWithUnnamedSocket(t *testing.T) {
    21:                                     if !testableNetwork("unixgram") {
    22:                                             t.Skip("unixgram test")
    23:                                     }
    24:                                     if runtime.GOOS == "openbsd" {
    25:                                             testenv.SkipFlaky(t, 15157)
(dlv) n
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:21 (PC: 0x1307111)
    16:                                     "testing"
    17:                                     "time"
    18:             )
    19:
    20:             func TestReadUnixgramWithUnnamedSocket(t *testing.T) {
=>  21:                                     if !testableNetwork("unixgram") {
    22:                                             t.Skip("unixgram test")
    23:                                     }
    24:                                     if runtime.GOOS == "openbsd" {
    25:                                             testenv.SkipFlaky(t, 15157)
    26:                                     }
(dlv)
```

next (n) – steps over the next line
list (l) – lists code at the current line
Continue (c) – runs to the next breakpoint
<enter> - reruns the previous command

Delve Commands


```
(dlv) h
The following commands are available:

Running the program:
        call ------------------------ Resumes process, injecting a function call (EXPERIMENTAL!!!)
        continue (alias: c) --------- Run until breakpoint or program termination.
        next (alias: n) ------------- Step over to next source line.
        rebuild --------------------- Rebuild the target executable and restarts it. It does not work if the executable was not built by delve.
        restart (alias: r) ---------- Restart process.
        step (alias: s) ------------- Single step through program.
        step-instruction (alias: si)  Single step a single cpu instruction.
        stepout (alias: so) --------- Step out of the current function.

Manipulating breakpoints:
        break (alias: b) ------- Sets a breakpoint.
        breakpoints (alias: bp)  Print out info for active breakpoints.
        clear ------------------ Deletes breakpoint.
        clearall --------------- Deletes multiple
        condition (alias: cond)  Set breakpoint condition.
        on --------------------- Executes a command
        toggle ----------------- Toggles on or off
        trace (alias: t) ------- Set tracepoint.
        watch ------------------ Set watchpoint.

Viewing program variables and memory:
        args ------------------ Print function argument.
        display --------------- Print value of an
...
```

```
(dlv) help call
Resumes process, injecting a function call (EXPERIMENTAL!!!)

condition.
                call [-unsafe] <function call expression>

Current limitations:
- only pointers to stack-allocated objects can be passed as
- only some automatic type conversions are supported.
- functions can only be called on running goroutines that are
arguments.executing the runtime.
expression- the current goroutine needs to have at least 256 bytes of free
    the stack.
    point.
- calling a function will resume execution of all goroutines.
- only supported on linux's native backend.

(dlv)
```

not

space on

a safe

Navigating

- s – step into function
- n – step over
- stepout – step out of function
- si – single step a machine instruction

```
...
(dlv) l
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:21 (PC: 0x1307111)
    16:                                             "testing"
    17:                                             "time"
    18:                     )
    19:
    20:                     func TestReadUnixgramWithUnnamedSocket(t *testing.T) {
=>  21:                                             if !testableNetwork("unixgram") {
    22:                                                     t.Skip("unixgram test")
    23:                                             }
    24:                                             if runtime.GOOS == "openbsd" {
    25:                                                     testenv.SkipFlaky(t, 15157)
    26:                                             }
(dlv) s
> net.testableNetwork() ./platform_test.go:36 (PC: 0x12bb68a)
    31:                                             }
    32:                     }
    33:
    34:                     // testableNetwork reports whether network is testable on the current
    35:                     // platform configuration.
=>  36:                     func testableNetwork(network string) bool {
    37:                                             net, _, _ := strings.Cut(network, ":")
    38:                                             switch net {
    39:                                             case "ip+nopriv":
    40:                                             case "ip", "ip4", "ip6":
    41:                                                     switch runtime.GOOS {
(dlv) n
> net.testableNetwork() ./platform_test.go:37 (PC: 0x12bb6ad)
    32:                     }
    33:
    34:                     // testableNetwork reports whether network is testable on the current
    35:                     // platform configuration.
    36:                     func testableNetwork(network string) bool {
=>  37:                                             net, _, _ := strings.Cut(network, ":")
    38:                                             switch net {
    39:                                             case "ip+nopriv":
    40:                                             case "ip", "ip4", "ip6":
    41:                                                     switch runtime.GOOS {
    42:                                                     case "plan9":
(dlv) so
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:21 (PC: 0x1307125)
Values returned:
                        ~r0: true

    16:                                             "testing"
    17:                                             "time"
    18:                     )
    19:
    20:                     func TestReadUnixgramWithUnnamedSocket(t *testing.T) {
=>  21:                                             if !testableNetwork("unixgram") {
    22:                                                     t.Skip("unixgram test")
    23:                                             }
    24:                                             if runtime.GOOS == "openbsd" {
    25:                                                     testenv.SkipFlaky(t, 15157)
    26:                                             }
(dlv)
```

Variables

- The print command can display the value of variables
- Print also understands many standard Go expressions
  - print x
  - print x < y
  - print &x
  - print x[4]
- The set command allows you to modify variables
  - set x = 5
- The call command also can modify (ex. when new memory is needed)

```
(dlv) locals
(no locals)
(dlv) args
t = (*testing.T)(0xc00010f380)
(dlv) n
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:29 (PC: 0x13071f5)
    24:                             if runtime.GOOS == "openbsd" {
    25:                                     testenv.SkipFlaky(t, 15157)
    26:                             }
    27:
    28:                             addr := testUnixAddr(t)
=>  29:                             la, err := ResolveUnixAddr("unixgram", addr)
    30:                             if err != nil {
    31:                                     t.Fatal(err)
    32:                             }
    33:                             c, err := ListenUnixgram("unixgram", la)
    34:                             if err != nil {
(dlv) locals
addr = "/tmp/1376813631/sock"
(dlv) print addr
"/tmp/1376813631/sock"
(dlv) print &addr
(*string)(0xc000052cc0)
(dlv) set addr = "broke it"
Command failed: literal string can not be allocated because function calls are not allowed without using 'call'
(dlv) call addr = "broke it"
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:29 (PC: 0x13071f5)
    24:                             if runtime.GOOS == "openbsd" {
    25:                                     testenv.SkipFlaky(t, 15157)
    26:                             }
    27:
    28:                             addr := testUnixAddr(t)
=>  29:                             la, err := ResolveUnixAddr("unixgram", addr)
    30:                             if err != nil {
    31:                                     t.Fatal(err)
    32:                             }
    33:                             c, err := ListenUnixgram("unixgram", la)
    34:                             if err != nil {
(dlv) print addr
"broke it"
(dlv) print &addr
(*string)(0xc000052cc0)
(dlv)
```

Breakpoints


```
(dlv) bp
Breakpoint runtime-fatal-throw (enabled) at 0x103b3c0 for runtime.throw() /usr/local/go/src/runtime/panic.go:982
Breakpoint unrecovered-panic (enabled) at 0x103b780 for runtime.fatalpanic() /usr/local/go/src/runtime/panic.go:1065
        print runtime.curg._panic.arg
Breakpoint 1 (enabled) at 0x133a4cf for main.main() _testmain.go:623 (1)
Breakpoint 2 (enabled) at 0x13070f2 for net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:20 (1)
(dlv) l
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:29 (PC: 0x13071f5)
    24:                                 if runtime.GOOS == "openbsd" {
    25:                                         testenv.SkipFlaky(t, 15157)
    26:                                 }
    27:
    28:                                 addr := testUnixAddr(t)
=>  29:                                 la, err := ResolveUnixAddr("unixgram", addr)
    30:                                 if err != nil {
    31:                                         t.Fatal(err)
    32:                                 }
    33:                                 c, err := ListenUnixgram("unixgram", la)
    34:                                 if err != nil {
(dlv) b 31
Breakpoint 3 set at 0x1307285 for net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:31
(dlv) b 33
Breakpoint 4 set at 0x130737c for net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:33
(dlv) bp
Breakpoint runtime-fatal-throw (enabled) at 0x103b3c0 for runtime.throw() /usr/local/go/src/runtime/panic.go:982
Breakpoint unrecovered-panic (enabled) at 0x103b780 for runtime.fatalpanic() /usr/local/go/src/runtime/panic.go:1065
        print runtime.curg._panic.arg
Breakpoint 1 (enabled) at 0x133a4cf for main.main() _testmain.go:623 (1)
Breakpoint 2 (enabled) at 0x13070f2 for net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:20 (1)
Breakpoint 3 (enabled) at 0x1307285 for net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:31 (0)
Breakpoint 4 (enabled) at 0x130737c for net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:33 (0)
(dlv) c
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:33 (hits goroutine(35):1 total:1) (PC: 0x130737c)
    28:                                 addr := testUnixAddr(t)
    29:                                 la, err := ResolveUnixAddr("unixgram", addr)
    30:                                 if err != nil {
    31:                                         t.Fatal(err)
    32:                                 }
=>  33:                                 c, err := ListenUnixgram("unixgram", la)
    34:                                 if err != nil {
    35:                                         t.Fatal(err)
    36:                                 }
    37:                                 defer func() {
    38:                                         c.Close()
(dlv)
```

(0)

(0)

- T break command sets breakpoints
- Or functions
- Or lines in a file
- Or lines offset from a function
- T bp command displ breakpoints
- T clear command clear a breakpoint
- T clearall command clears al breakpoints

Debugging concurrent applications

- Delve allows you to switch and manage
  - Threads
  - threads – lists available threads
  - thread – switches to the specified thread
  - GoRoutines
    - goroutines – lists available goroutines
    - goroutine – switches to the specified goroutine

```
(dlv) goroutines
  Goroutine 1 - User: /usr/local/go/src/testing/testing.go:1488 testing.(*T).Run (0x11221a5) [chan receive]
  Goroutine 2 - User: /usr/local/go/src/runtime/proc.go:367 runtime.gopark (0x103def2) [force gc (idle)]
  Goroutine 18 - User: /usr/local/go/src/runtime/proc.go:367 runtime.gopark (0x103def2) [GC sweep wait]
  Goroutine 19 - User: /usr/local/go/src/runtime/proc.go:367 runtime.gopark (0x103def2) [GC scavenge wait]
  Goroutine 34 - User: /usr/local/go/src/runtime/proc.go:367 runtime.gopark (0x103def2) [finalizer wait]
* Goroutine 35 - User: ./unixsock_test.go:33 net.TestReadUnixgramWithUnnamedSocket (0x130737c) (thread 494243)
  Goroutine 36 - User: /usr/local/go/src/runtime/debugcall.go:236 runtime.debugCallWrap1 (0x10096f3)
[7 goroutines]
(dlv) l
> net.TestReadUnixgramWithUnnamedSocket() ./unixsock_test.go:33 (hits goroutine(35):1 total:1) (PC: 0x130737c)
     28:                                        addr := testUnixAddr(t)
     29:                                        la, err := ResolveUnixAddr("unixgram", addr)
     30:                                        if err != nil {
     31:                                                t.Fatal(err)
     32:                                        }
=>   33:                                        c, err := ListenUnixgram("unixgram", la)
     34:                                        if err != nil {
     35:                                                t.Fatal(err)
     36:                                        }
     37:                                        defer func() {
     38:                                                c.Close()
(dlv) goroutine 1
Switched from 35 to 1 (thread 494243)
(dlv) l
> runtime.gopark() /usr/local/go/src/runtime/proc.go:367 (PC: 0x103def2)
Warning: debugging optimized function
    362:                                    mp.waittraceev = traceEv
    363:                                    mp.waittraceskip = traceskip
    364:                                    releasem(mp)
    365:                                    // can't do anything that might move the G between Ms here.
    366:                                    mcall(park_m)
=>  367:            }
    368:
    369:            // Puts the current goroutine into a waiting state and unlocks the lock.
    370:            // The goroutine can be made runnable again by calling goready(gp).
    371:            func goparkunlock(lock *mutex, reason waitReason, traceEv byte, traceskip int) {
    372:                                    gopark(parkunlock_c, unsafe.Pointer(lock), reason, traceEv, traceskip)
(dlv) goroutine 35
Switched from 1 to 35 (thread 494243)
                                        c.Close()
(dlv) threads
* Thread 494243 at 0x130737c ./unixsock_test.go:33 net.TestReadUnixgramWithUnnamedSocket
  Thread 494283 at :0
  Thread 494284 at :0
  Thread 494285 at :0
  Thread 494286 at :0
  Thread 494287 at :0
(dlv)
```

Other Commands

- You can display source files associated with the executable
- sources
- You can display the call stack
- stack
- You can display loaded types
- types <optional regex>
- You can display vars defined
- vars <optional regex>
- You can display functions defined
- funcs <optional regex>

```
(dlv) sources .*net/unix.*test.go
/usr/local/go/src/net/unixsock_readmsg_test.go
/usr/local/go/src/net/unixsock_test.go
(dlv)
```

```
(dlv) stack
0  0x000000000130737c in net.TestReadUnixgramWithUnnamedSocket
   at ./unixsock_test.go:33
1  0x00000000011209c3 in testing.tRunner
   at /usr/local/go/src/testing/testing.go:1440
2  0x0000000001122259 in testing.(*T).Run.func1
   at /usr/local/go/src/testing/testing.go:1487
3  0x000000000106dfe1 in runtime.goexit
   at /usr/local/go/src/runtime/asm_amd64.s:1571
(dlv)
```

```
(dlv) types ^sync\.(RWM|M)
sync.Map
sync.Mutex
sync.RWMutex
(dlv)
```

```
(dlv) funcs TestReadUnixgramWithUnnamedSocket
net.TestReadUnixgramWithUnnamedSocket
net.TestReadUnixgramWithUnnamedSocket.func1
net.TestReadUnixgramWithUnnamedSocket.func2
net.TestReadUnixgramWithUnnamedSocket.func2.1
net.TestReadUnixgramWithUnnamedSocket.func2.2
(dlv) funcs main.main
main.main
(dlv) funcs main
main.init.0
main.main
net.TestSpecialDomainName
net.TestSpecialDomainName.func1
net.absDomainName
net.isDomainName
runtime.main
runtime.main.func1
runtime.main.func2
(dlv)
```

```
(dlv) vars ^main
main.tests = []testing.InternalTest len: 241, cap: 241, [...]
main.benchmarks = []testing.InternalBenchmark len: 29, cap: 29, [...]
main.fuzzTargets = []testing.InternalFuzzTarget len: 0, cap: 0, []
main.examples = []testing.InternalExample len: 19, cap: 19, [...]
(dlv) vars ^math
math.useFMA = true
math/rand.globalRand = ("*math/rand.Rand")(0xc0001041b0)
math/rand.rngCooked = [607]int64 [...]
(dlv)
```

Remote Debugging

- Delve supports remote debugging
- Launch the target with dlv using the --headless switch
- $ dlv --headless -l “192.168.0.4:9090” hello.go
- Other useful switches:
  - -l, --listen string Debugging server listen address
    - default "localhost:0"
  - --log Enable debugging server logging
- To attach to the remote debugger:
- $ dlv connect 192.168.0.4:9090

Data Races

- A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write
- Check for data races via:

go run –race race.go

- Helps find:
  - Race on loop counter
  - Accidentally shared vars
  - Unprotected global vars
  - Primitive unprotected vars

```
~$ cat -n race.go | sed -e 's/^ *//g' -e 's/\t/ /g'
1  package main
2
3  import "fmt"
4
5  func main() {
6      c := make(chan bool)
7      m := make(map[string]string)
8      go func() {
9        m["1"] = "a" // First conflicting access
10         c <- true
11     }()
12     m["2"] = "b" // Second conflicting access
13     <-c
14     for k, v := range m {
15         fmt.Println(k, v)
16     }
17 }
```

```
~$ go run -race race.go
==================
WARNING: DATA RACE
Write at 0x00c000124180 by goroutine 7:
  runtime.mapassign_faststr()
      /usr/local/go/src/runtime/map_faststr.go:202 +0x0
  main.main.func1()
      /Users/ronaldpetty/race.go:9 +0x5d

Previous write at 0x00c000124180 by main goroutine:
  runtime.mapassign_faststr()
      /usr/local/go/src/runtime/map_faststr.go:202 +0x0
  main.main()
      /Users/ronaldpetty/ race.go:12 +0xcb

Goroutine 7 (running) created at:
  main.main()
      /Users/ronaldpetty/ race.go:8 +0x9c
==================
2 b
1 a
Found 1 data race(s)
exit status 66
```

What Else?

- Package ‘runtime’ provides many settings to help investigate our application
- Runtime configuration via environment variables
  - GOGC
  - GODEBUG
  - GOMAXPROCS
  - GORACE
  - GOTRACEBACK
- Debug stack via “runtime/debug” package
- Trigger GC – runtime.GC()
- … more …

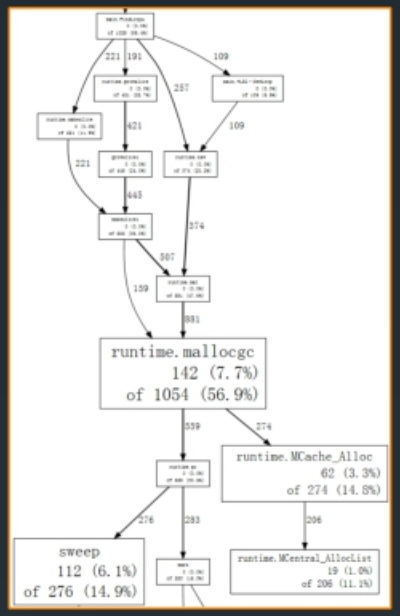
- Other considerations (deeper) are the use of glibc/musl, etc.

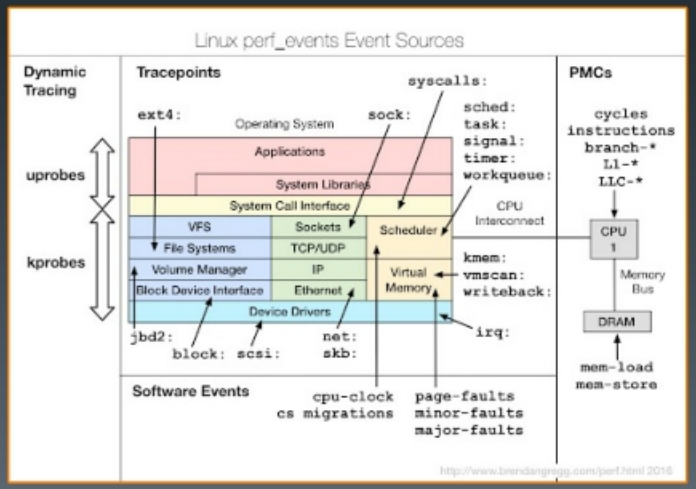
Summary

- Go adds new profiling features with each version
- Go's pprof can also read Linux perf
- https://perf.wiki.kernel.org/index.php/Main_Page

- Go makes it easy to do concurrency
- Which leads to concurrency issues

- There are many tools that can help with debugging Go programs
  - Interactive debuggers are particularly useful
  - Solving problems
  - Also helping developers understand running programs and thus doing a better job of coding
  - GDB is a stop gap debugger only
  - Delve is the most useful Go debugger today





Lab

- Profiling, tracing, and debugging and more!

The End

- Thank you for attending
- If you have any further questions please email me
  - John Kidd
  - jkidd@kiddcorp.com