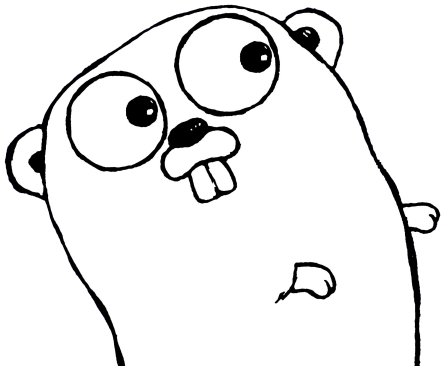
Go Pipelines

Joey Elliott



The Big Question(s):

What tools does Go provide for a pipeline?
How can we use those tools?

Tools Provided By Go

There are a bunch of great tools in the standard library and the sub-repos.

https://golang.org/pkg/

- net/http
- regexp
- sync

https://github.com/golang/go/wiki/SubRepositories

- x/net
- x/image
- x/crypto

Tools (Not) Provided By Go

go get github.com/golang/example/stringutil

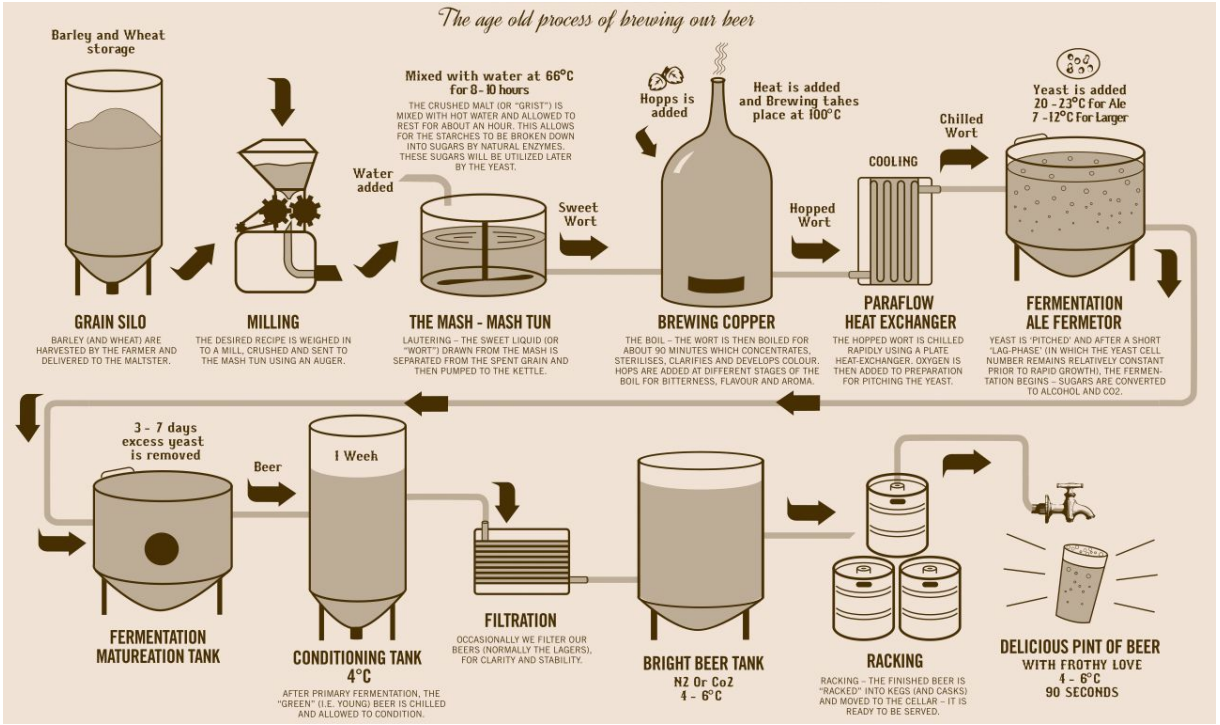
- fetches remote package
- stores in go/bin

```
import (
	"github.com/golang/example/stringutil"
)
```

Wait, I don’t know what a pipeline is.

That’s okay, just:

- Smile and nod
- Think about beer



Tools Provided By Go (Recap)

Goroutines

- Lightweight thread

Channels

- Message passing

```
func main() {
        chProcessedGrain := millHelper()

        fmt.Println(<- chProcessedGrain)
}

func millHelper() <- chan string {
        channel := make(chan string)

        go func() {
                channel <- "processed grain"
        }()

        return channel
}
```

Tools Provided By Go

Multiple Return Values

- Easy way to return multiple channels

sync.WaitGroup

- Use integer value to determine if done
- Provide blocking

```go
func main() {
	chProcessedGrain, chMouse := millHelper()

	for {
		select {
		case processedGrain := <- chProcessedGrain:
			fmt.Println(processedGrain)
		case mouse := <- chMouse:
			if mouse {
				fmt.Println("eww, a mouse!")
			} else {
				fmt.Println("no mice")
			}
			return
		}
	}
}
```

```go
func millHelper() (<- chan string, <- chan bool) {
	chGrainChute, chMouse := make(chan string), make(chan bool)

	go func() {
		var waitGroup sync.WaitGroup

		for i := 0; i < 1000; i++ {
			waitGroup.Add(1)
			go func() {
				chGrainChute <- "finely ground grain"
				waitGroup.Done()
			}()
		}

		waitGroup.Wait()

		// no mice during the processing (thank goodness)
		chMouse <- false
	}()

	return chGrainChute, chMouse
}
```

Tool-Enabled Concepts

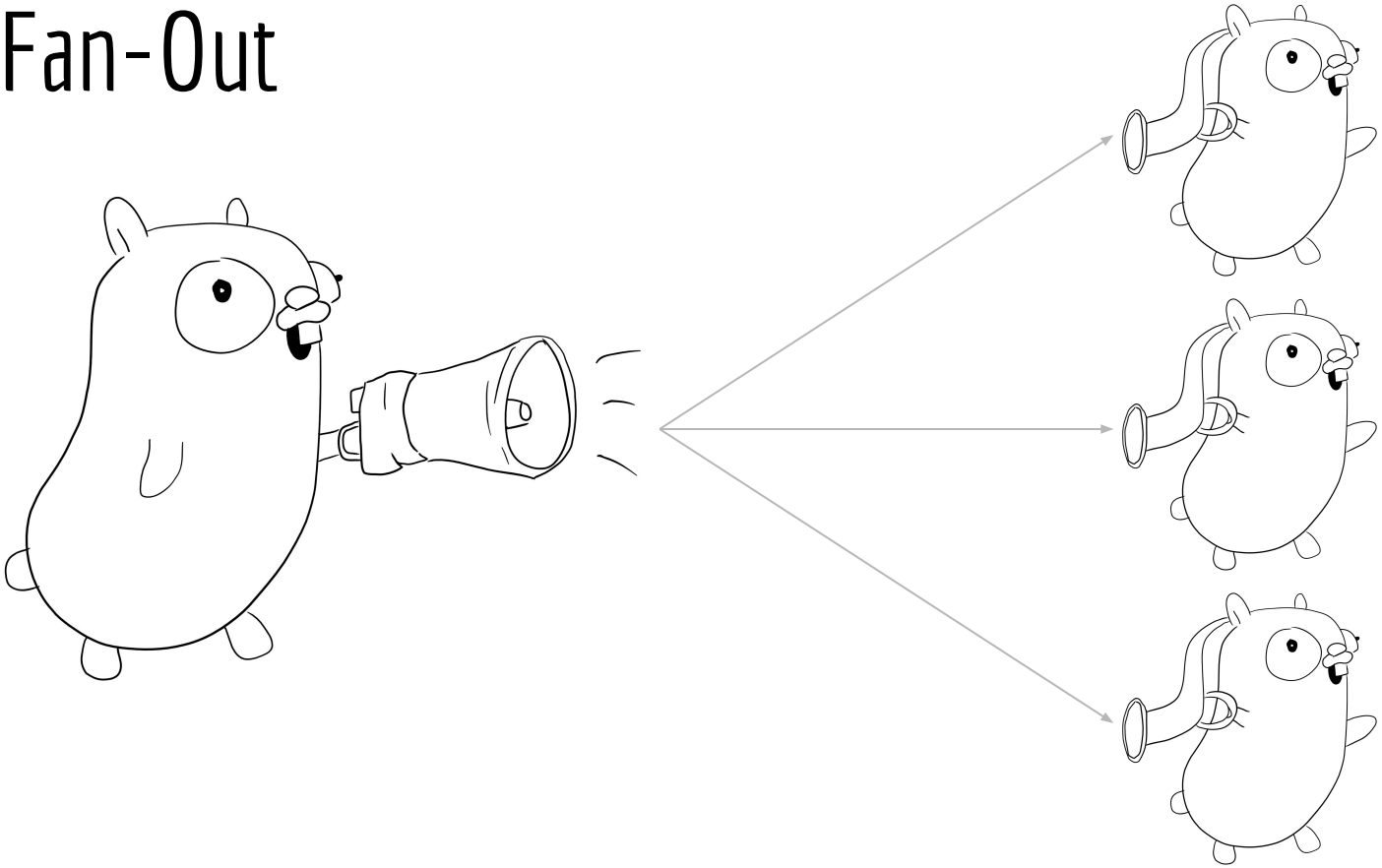
Fan-Out

- Multiple functions read from the same channel

Fan-In

- One function reads from multiple channels

Fan-Out

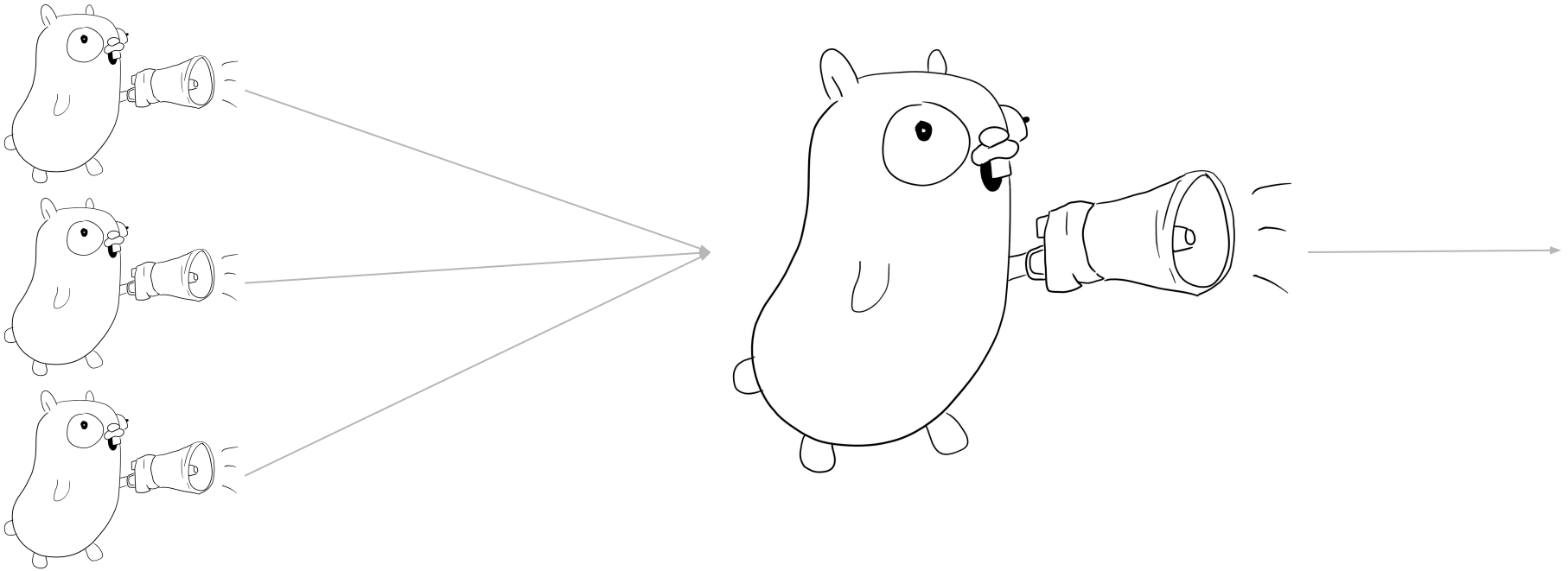


Note: Only one listener picks up a message at a time

```go
func main() {
	// one grain silo
	chGrainSiloOutput := startGrainSilo()

	// 10 millers
	chMillersOutput := startMillers(chGrainSiloOutput, 10)

	// do something with all these channels *cough* fan in *cough*
}

func startGrainSilo() <- chan string {
	chMixedGrains := make(chan string)

	go func() {
		for i := 0; i < 100; i++ {
			if i % 2 == 0 {
				chMixedGrains <- "wheat"
			} else {
				chMixedGrains <- "barley"
			}
		}
		close(chMixedGrains)
	}()

	return chMixedGrains
}
```

```go
func startMillers(chMixedGrains <- chan string, numMillers int) [] <- chan string {
	millers := make([] <- chan string, numMillers)

	for i := range millers {
		millers[i] = millHelper(chMixedGrains)
	}

	return millers
}

func millHelper(chMixedGrains <- chan string) <- chan string {
	chGrainChute := make(chan string)

	go func() {
		for grain := range chMixedGrains {
			chGrainChute <- Grind(grain)
		}
		close(chGrainChute)
	}()

	return chGrainChute
}

func Grind(grain string) string {
	return "finely ground " + grain
}
```

Fan-In



```go
func main() {
	// one grain silo
	chGrainSiloOutput := startGrainSilo()

	// 10 millers
	chMillersOutput := startMillers(chGrainSiloOutput, 10)

	// 1 channel composed of all data from 'chMillersOutput'
	chMergedMillers := mergedMillerChutes(chMillersOutput)

	// one mash tun
	startMashTun(chMergedMillers)
}

func startMashTun(chGroundGrain <- chan string) {
	for i := range chGroundGrain {
		fmt.Printf("Mashing %s as best as I can!\n", i)
	}
}
```

```go
func mergedMillerChutes(millerChutes [] <- chan string) <- chan string {
	merged := make(chan string)

	var waitGroup sync.WaitGroup
	waitGroup.Add(len(millerChutes))

	for _, millerChute := range millerChutes {
		go func(chute <- chan string) {
			for groundGrain := range chute {
				merged <- groundGrain
			}
			waitGroup.Done()
		}(millerChute)
	}

	go func() {
		waitGroup.Wait()
		close(merged)
	}()

	return merged
}
```

Drawbacks

No generics

- Reuse code? No, no, no. Retype code.

Goroutine leak

- Goroutine never finishes; lingers in background
- Can lead to running out of memory

Goroutine Leak

Example:

- Send to channel without a receiver

Other potential leaks:

- Receiving from channel without a sender
- Infinite loops
- API request without timeouts

```
func main() {
        for i := 0; i < 4; i++ {
                start()
                fmt.Printf("Number of goroutines: %d\n", runtime.NumGoroutine())
        }
}

func start() <- chan int {
        channel := make(chan int)

        for i := 0; i < 10; i++ {
                go func() {
                        channel <- i
                }()
        }

        return channel
}
```

Output:

```
Number of goroutines: 11
Number of goroutines: 21
Number of goroutines: 31
Number of goroutines: 41
```

Demo

https://github.com/Joseph-W-E/GolangUrbanDictionaryHelper