
O'REILLY®
Concurrent Programming in Go
With Interactivity

Learning Objectives

By the end of this course, you will understand:

- Learn how to correctly reason about concurrent code in Go.
- Gain familiarity with Go's concurrency primitives along with multiple examples and non-examples.
- Become aware of common pitfalls and mistakes that can happen in Go and understand the dangers involved.
- Learn how to apply best practices for concurrent programming in Go.
- Learn common concurrent programming patterns such as fan-out, fan-in, and map-reduce.

Course Outcomes

By the end of this course, you should be able to:

- Understand the behavior of channels and goroutines in Go.
- Make use of select statements, and contexts to cancel running goroutines
- Analyze and avoid resource leaks and deadlocks in concurrent code.
- Understand the use of wait groups, mutexes, and conditions.
- Understand the in-memory behavior of buffered channels.
- Apply common concurrency patterns such as fan-in, fan-out, and map-reduce.

Live-Coding

Goroutines, channels, and fan-out

Playground Exercise

(5 minutes)

Using the Go playground link provided by the instructor, add the following WaitGroup calls at the correct locations.

- wg.Add(1)
- wg.Done()
- wg.Wait()

Run your code to test it. Do you see the results which you expected?

Using the 'happens-before' relationship and the locations you selected, convince yourself that it is *impossible* for the main() func to complete until every message has been received.

Live-Coding

Cancellation & Timeouts with context

Context Q&A

Break

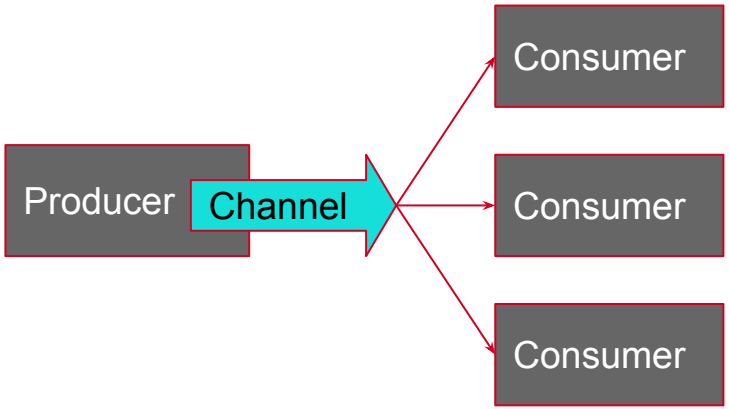
(5 minutes)

Fan-in Challenge

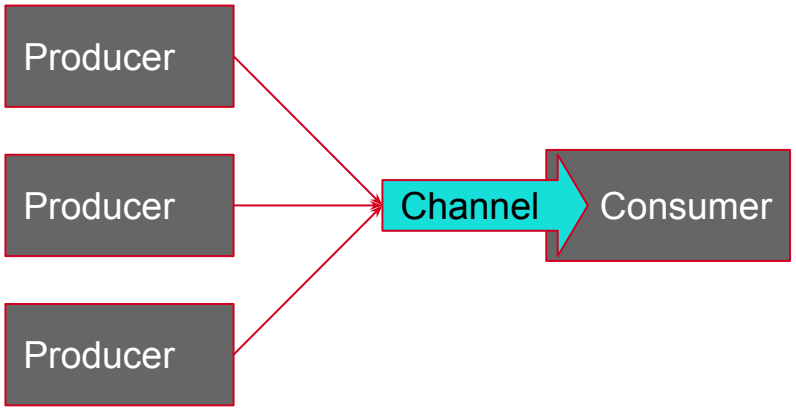
Lab

Fan-out vs. Fan-in

Fan-out



Fan-in



Lab: Fan-in Pattern

(25 minutes)

Apply what you've learned to the starter code provided to construct the fan-in pattern. Use one channel which accepts integers generated from 10 producer goroutines and sends them to a single consumer goroutine. The consumer can just print out the integers in order.

Keep the following in mind:

- Sending and receiving to/from a channel may always block.
- Range loops over a channel block until the channel has been closed.
- Range loop capture is dangerous: pass variables to anonymous functions explicitly.
- "Never start a goroutine you don't know how to stop."
    - Convince yourself that all goroutines return before the main func ends using the 'happens-before' relationship between WaitGroup calls.

Don't hesitate to ask questions in the chat and share work-in-progress playground links. I will be happy to address them live.

Live-Coding

Fan-in solution

Break

(5 minutes)

Intro

Circular buffers

Circular Buffers

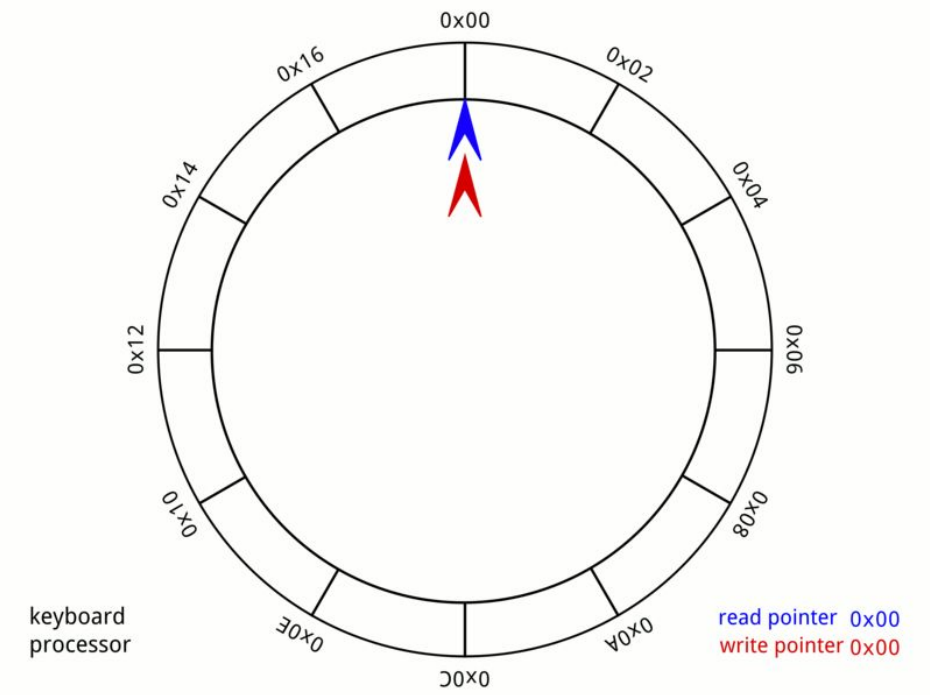


Image credit: Wikipedia "Circular Buffer" article.

Live-Coding

Buffered Channels from scratch

Playground Exercise

(5 minutes)

Using the Go playground link provided by the instructor, implement the 'Receive' operation using the starter code at the playground link.

- If no elements are available to receive in the buffer, return an ErrorEmpty error.

Run your code to test it. Do you see the results which you expected?

Live-Coding

Buffered Channels from scratch

Q&A

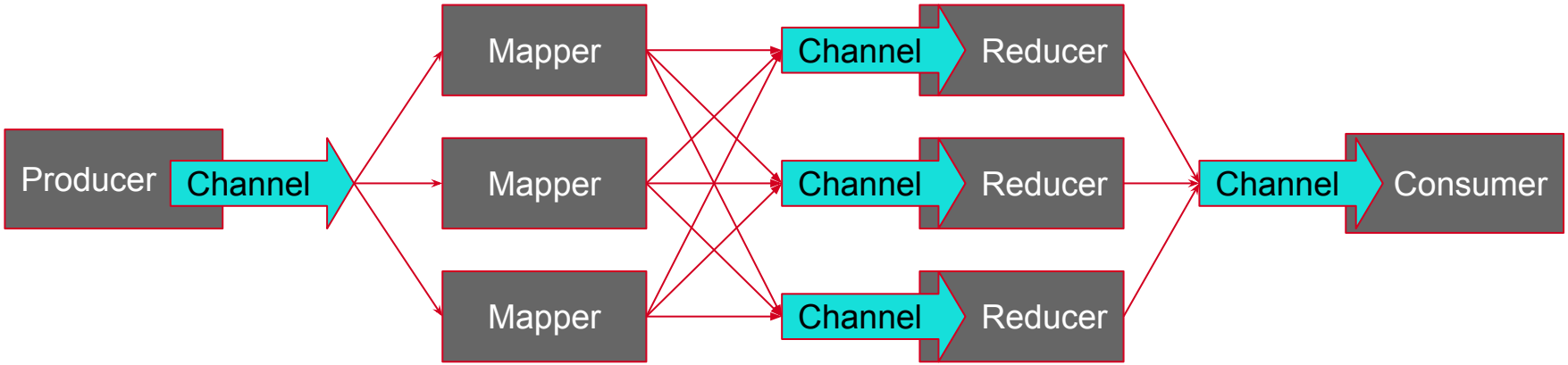
Break

(5 minutes)

Intro

Map-Reduce

Map-Reduce


Producer
Channel
Mapper
Mapper
Mapper
Channel
Reducer
Channel
Reducer
Channel
Reducer
Channel
Consumer

Live-Coding

Map-Reduce

Lab Challenge: Channel Closure & Shutdown

(15 minutes)

Apply what you've learned to the starter code provided to close all channels properly and achieve graceful shutdown before the main() func ends. Use the 'happens-before' relationship to convince yourself that channels are closed only after all data has been processed.

Keep the following in mind:

- Sending and receiving to / from a channel may always block.
- Range loops over a channel block until the channel has been closed.

Don't hesitate to ask questions in the chat and share work-in-progress playground links. I will be happy to address them live.

Q&A