CR1

// This sample program demonstrates the basic channel mechanics

// for goroutine signaling.

package main

import (

"context"

"fmt"

"math/rand"

"runtime"

"sync"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

func main() {

waitForResult()

// fanOut()

// waitForTask()

// pooling()

// Advanced patterns

// fanOutSem()

// boundedWorkPooling()

// drop()

// Cancellation Pattern

// cancellation()

// Retry Pattern

// ctx, cancel := context.WithTimeout(context.Background(), 5\*time.Second)

// defer cancel()

// retryTimeout(ctx, time.Second, func(ctx context.Context) error { return errors.New("always fail") })

}

// waitForResult: You are a manager and you hire a new employee. Your new

// employee knows immediately what they are expected to do and starts their

// work. You sit waiting for the result of the employee's work. The amount

// of time you wait on the employee is unknown because you need a

// guarantee that the result sent by the employee is received by you.

func waitForResult() {

ch := make(chan string)

go func() {

time.Sleep(time.Duration(rand.Intn(500)) \* time.Millisecond)

ch <- "paper"

fmt.Println("employee : sent signal")

}()

p := <-ch

fmt.Println("manager : recv'd signal :", p)

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// fanOut: You are a manager and you hire one new employee for the exact amount

// of work you have to get done. Each new employee knows immediately what they

// are expected to do and starts their work. You sit waiting for all the results

// of the employees work. The amount of time you wait on the employees is

// unknown because you need a guarantee that all the results sent by employees

// are received by you. No given employee needs an immediate guarantee that you

// received their result.

func fanOut() {

emps := 2000

ch := make(chan string, emps)

for e := 0; e < emps; e++ {

go func(emp int) {

time.Sleep(time.Duration(rand.Intn(200)) \* time.Millisecond)

ch <- "paper"

fmt.Println("employee : sent signal :", emp)

}(e)

}

for emps > 0 {

p := <-ch

emps--

fmt.Println(p)

fmt.Println("manager : recv'd signal :", emps)

}

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// waitForTask: You are a manager and you hire a new employee. Your new

// employee doesn't know immediately what they are expected to do and waits for

// you to tell them what to do. You prepare the work and send it to them. The

// amount of time they wait is unknown because you need a guarantee that the

// work your sending is received by the employee.

func waitForTask() {

ch := make(chan string)

go func() {

p := <-ch

fmt.Println("employee : recv'd signal :", p)

}()

time.Sleep(time.Duration(rand.Intn(500)) \* time.Millisecond)

ch <- "paper"

fmt.Println("manager : sent signal")

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// pooling: You are a manager and you hire a team of employees. None of the new

// employees know what they are expected to do and wait for you to provide work.

// When work is provided to the group, any given employee can take it and you

// don't care who it is. The amount of time you wait for any given employee to

// take your work is unknown because you need a guarantee that the work your

// sending is received by an employee.

func pooling() {

ch := make(chan string)

g := runtime.GOMAXPROCS(0)

for e := 0; e < g; e++ {

go func(emp int) {

for p := range ch {

fmt.Printf("employee %d : recv'd signal : %s\n", emp, p)

}

fmt.Printf("employee %d : recv'd shutdown signal\n", emp)

}(e)

}

const work = 100

for w := 0; w < work; w++ {

ch <- "paper"

fmt.Println("manager : sent signal :", w)

}

close(ch)

fmt.Println("manager : sent shutdown signal")

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// fanOutSem: You are a manager and you hire one new employee for the exact amount

// of work you have to get done. Each new employee knows immediately what they

// are expected to do and starts their work. However, you don't want all the

// employees working at once. You want to limit how many of them are working at

// any given time. You sit waiting for all the results of the employees work.

// The amount of time you wait on the employees is unknown because you need a

// guarantee that all the results sent by employees are received by you. No

// given employee needs an immediate guarantee that you received their result.

func fanOutSem() {

emps := 2000

ch := make(chan string, emps)

g := runtime.GOMAXPROCS(0)

sem := make(chan bool, g)

for e := 0; e < emps; e++ {

go func(emp int) {

sem <- true

{

time.Sleep(time.Duration(rand.Intn(200)) \* time.Millisecond)

ch <- "paper"

fmt.Println("employee : sent signal :", emp)

}

<-sem

}(e)

}

for emps > 0 {

p := <-ch

emps--

fmt.Println(p)

fmt.Println("manager : recv'd signal :", emps)

}

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// boundedWorkPooling: You are a manager and you hire a team of employees. None of

// the new employees know what they are expected to do and wait for you to

// provide work. The amount of work that needs to get done is fixed and staged

// ahead of time. Any given employee can take work and you don't care who it is

// or what they take. The amount of time you wait on the employees to finish

// all the work is unknown because you need a guarantee that all the work is

// finished.

func boundedWorkPooling() {

work := []string{"paper", "paper", "paper", "paper", "paper", 2000: "paper"}

g := runtime.GOMAXPROCS(0)

var wg sync.WaitGroup

wg.Add(g)

ch := make(chan string, g)

for e := 0; e < g; e++ {

go func(emp int) {

defer wg.Done()

for p := range ch {

fmt.Printf("employee %d : recv'd signal : %s\n", emp, p)

}

fmt.Printf("employee %d : recv'd shutdown signal\n", emp)

}(e)

}

for \_, wrk := range work {

ch <- wrk

}

close(ch)

wg.Wait()

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// drop: You are a manager and you hire a new employee. Your new employee

// doesn't know immediately what they are expected to do and waits for

// you to tell them what to do. You prepare the work and send it to them. The

// amount of time they wait is unknown because you need a guarantee that the

// work your sending is received by the employee. You won't wait for the

// employee to take the work if they are not ready to receive it. In that case

// you drop the work on the floor and try again with the next piece of work.

func drop() {

const cap = 100

ch := make(chan string, cap)

go func() {

for p := range ch {

fmt.Println("employee : recv'd signal :", p)

}

}()

const work = 2000

for w := 0; w < work; w++ {

select {

case ch <- "paper":

fmt.Println("manager : sent signal :", w)

default:

fmt.Println("manager : dropped data :", w)

}

}

close(ch)

fmt.Println("manager : sent shutdown signal")

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// cancellation: You are a manager and you hire a new employee. Your new

// employee knows immediately what they are expected to do and starts their

// work. You sit waiting for the result of the employee's work. The amount

// of time you wait on the employee is unknown because you need a

// guarantee that the result sent by the employee is received by you. Except

// you are not willing to wait forever for the employee to finish their work.

// They have a specified amount of time and if they are not done, you don't

// wait and walk away.

func cancellation() {

duration := 150 \* time.Millisecond

ctx, cancel := context.WithTimeout(context.Background(), duration)

defer cancel()

ch := make(chan string, 1)

go func() {

time.Sleep(time.Duration(rand.Intn(200)) \* time.Millisecond)

ch <- "paper"

}()

select {

case d := <-ch:

fmt.Println("work complete", d)

case <-ctx.Done():

fmt.Println("work cancelled")

}

time.Sleep(time.Second)

fmt.Println("-------------------------------------------------------------")

}

// retryTimeout: You need to validate if something can be done with no error

// but it may take time before this is true. You set a retry interval to create

// a delay before you retry the call and you use the context to set a timeout.

func retryTimeout(ctx context.Context, retryInterval time.Duration, check func(ctx context.Context) error) {

for {

fmt.Println("perform user check call")

if err := check(ctx); err == nil {

fmt.Println("work finished successfully")

return

}

fmt.Println("check if timeout has expired")

if ctx.Err() != nil {

fmt.Println("time expired 1 :", ctx.Err())

return

}

fmt.Printf("wait %s before trying again\n", retryInterval)

t := time.NewTimer(retryInterval)

select {

case <-ctx.Done():

fmt.Println("timed expired 2 :", ctx.Err())

t.Stop()

return

case <-t.C:

fmt.Println("retry again")

}

}

}

CR2

// Sample program to show how to use an unbuffered channel to

// simulate a game of tennis between two goroutines.

package main

import (

"fmt"

"math/rand"

"sync"

"time"

)

func init() {

rand.Seed(time.Now().UnixNano())

}

func main() {

// Create an unbuffered channel.

court := make(chan int)

// wg is used to manage concurrency.

var wg sync.WaitGroup

wg.Add(2)

// Launch two players.

go func() {

player("Serena", court)

wg.Done()

}()

go func() {

player("Venus", court)

wg.Done()

}()

// Start the set.

court <- 1

// Wait for the game to finish.

wg.Wait()

}

// player simulates a person playing the game of tennis.

func player(name string, court chan int) {

for {

// Wait for the ball to be hit back to us.

ball, wd := <-court

if !wd {

// If the channel was closed we won.

fmt.Printf("Player %s Won\n", name)

return

}

// Pick a random number and see if we miss the ball.

n := rand.Intn(100)

if n%13 == 0 {

fmt.Printf("Player %s Missed\n", name)

// Close the channel to signal we lost.

close(court)

return

}

// Display and then increment the hit count by one.

fmt.Printf("Player %s Hit %d\n", name, ball)

ball++

// Hit the ball back to the opposing player.

court <- ball

}

}

CR3

// Sample program to show how to use an unbuffered channel to

// simulate a relay race between four goroutines.

package main

import (

"fmt"

"sync"

"time"

)

// wg is used to wait for the program to finish.

var wg sync.WaitGroup

func main() {

// Create an unbuffered channel.

track := make(chan int)

// Add a count of one for the last runner.

wg.Add(1)

// First runner to his mark.

go Runner(track)

// Start the race.

track <- 1

// Wait for the race to finish.

wg.Wait()

}

// Runner simulates a person running in the relay race.

func Runner(track chan int) {

// The number of exchanges of the baton.

const maxExchanges = 4

var exchange int

// Wait to receive the baton.

baton := <-track

// Start running around the track.

fmt.Printf("Runner %d Running With Baton\n", baton)

// New runner to the line.

if baton < maxExchanges {

exchange = baton + 1

fmt.Printf("Runner %d To The Line\n", exchange)

go Runner(track)

}

// Running around the track.

time.Sleep(100 \* time.Millisecond)

// Is the race over.

if baton == maxExchanges {

fmt.Printf("Runner %d Finished, Race Over\n", baton)

wg.Done()

return

}

// Exchange the baton for the next runner.

fmt.Printf("Runner %d Exchange With Runner %d\n",

baton,

exchange)

track <- exchange

}

CR4

// This sample program demonstrates how to use a buffered

// channel to receive results from other goroutines in a guaranteed way.

package main

import (

"fmt"

"log"

"math/rand"

"time"

)

// result is what is sent back from each operation.

type result struct {

id int

op string

err error

}

func init() {

rand.Seed(time.Now().UnixNano())

}

func main() {

// Set the number of routines and inserts.

const routines = 10

const inserts = routines \* 2

// Buffered channel to receive information about any possible insert.

ch := make(chan result, inserts)

// Number of responses we need to handle.

waitInserts := inserts

// Perform all the inserts.

for i := 0; i < routines; i++ {

go func(id int) {

ch <- insertUser(id)

// We don't need to wait to start the second insert

// thanks to the buffered channel. The first send

// will happen immediately.

ch <- insertTrans(id)

}(i)

}

// Process the insert results as they complete.

for waitInserts > 0 {

// Wait for a response from a goroutine.

r := <-ch

// Display the result.

log.Printf("N: %d ID: %d OP: %s ERR: %v", waitInserts, r.id, r.op, r.err)

// Decrement the wait count and determine if we are done.

waitInserts--

}

log.Println("Inserts Complete")

}

// insertUser simulates a database operation.

func insertUser(id int) result {

r := result{

id: id,

op: fmt.Sprintf("insert USERS value (%d)", id),

}

// Randomize if the insert fails or not.

if rand.Intn(10) == 0 {

r.err = fmt.Errorf("Unable to insert %d into USER table", id)

}

return r

}

// insertTrans simulates a database operation.

func insertTrans(id int) result {

r := result{

id: id,

op: fmt.Sprintf("insert TRANS value (%d)", id),

}

// Randomize if the insert fails or not.

if rand.Intn(10) == 0 {

r.err = fmt.Errorf("Unable to insert %d into USER table", id)

}

return r

}

CR5

// This sample program demonstrates how to use a channel to

// monitor the amount of time the program is running and terminate

// the program if it runs too long.

package main

import (

"errors"

"log"

"os"

"os/signal"

"time"

)

// Give the program 3 seconds to complete the work.

const timeoutSeconds = 3 \* time.Second

func main() {

// sigChan receives os signals.

sigChan := make(chan os.Signal, 1)

// timeout limits the amount of time the program has.

timeout := time.After(timeoutSeconds)

// complete is used to report processing is done.

complete := make(chan error)

// shutdown provides system wide notification.

shutdown := make(chan struct{})

log.Println("Starting Process")

// We want to receive all interrupt based signals.

signal.Notify(sigChan, os.Interrupt)

// Launch the process.

log.Println("Launching Processors")

go processor(complete, shutdown)

ControlLoop:

for {

select {

case <-sigChan:

// Interrupt event signaled by the operation system.

log.Println("OS INTERRUPT")

// Close the channel to signal to the processor

// it needs to shutdown.

close(shutdown)

// Set the channel to nil so we no longer process

// any more of these events.

sigChan = nil

case <-timeout:

// We have taken too much time. Kill the app hard.

log.Println("Timeout - Killing Program")

os.Exit(1)

case err := <-complete:

// Everything completed within the time given.

log.Printf("Task Completed: Error[%s]", err)

break ControlLoop

}

}

// Program finished.

log.Println("Process Ended")

}

// processor provides the main program logic for the program.

func processor(complete chan<- error, shutdown <-chan struct{}) {

log.Println("Processor - Starting")

// Variable to store any error that occurs.

// Passed into the defer function via closures.

var err error

// Defer the send on the channel so it happens

// regardless of how this function terminates.

defer func() {

// Capture any potential panic.

if r := recover(); r != nil {

log.Println("Processor - Panic", r)

}

// Signal the goroutine we have shutdown.

complete <- err

}()

// Perform the work.

err = doWork(shutdown)

log.Println("Processor - Completed")

}

// doWork simulates task work.

func doWork(shutdown <-chan struct{}) error {

log.Println("Processor - Task 1")

time.Sleep(2 \* time.Second)

if checkShutdown(shutdown) {

return errors.New("Early Shutdown")

}

log.Println("Processor - Task 2")

time.Sleep(1 \* time.Second)

if checkShutdown(shutdown) {

return errors.New("Early Shutdown")

}

log.Println("Processor - Task 3")

time.Sleep(1 \* time.Second)

return nil

}

// checkShutdown checks the shutdown flag to determine

// if we have been asked to interrupt processing.

func checkShutdown(shutdown <-chan struct{}) bool {

select {

case <-shutdown:

// We have been asked to shutdown cleanly.

log.Println("checkShutdown - Shutdown Early")

return true

default:

// If the shutdown channel was not closed,

// presume with normal processing.

return false

}

}

CRa

// Sample program to show the order of channel communication for unbuffered,

// buffered and closing channels based on the specification.

// https://golang.org/ref/mem#tmp\_7

package main

import "fmt"

func main() {

unBuffered()

buffered()

closed()

}

// With unbuffered channels, the receive happens before the corresponding send.

// The write to a happens before the receive on c, which happens before the

// corresponding send on c completes, which happens before the print.

func unBuffered() {

c := make(chan int)

var a string

go func() {

a = "hello, world"

<-c

}()

c <- 0

// We are guaranteed to print "hello, world".

fmt.Println(a)

}

// With buffered channels, the send happens before the corresponding receive.

// The write to a happens before the send on c, which happens before the

// corresponding receive on c completes, which happens before the print.

func buffered() {

c := make(chan int, 10)

var a string

go func() {

a = "hello, world"

c <- 0

}()

<-c

// We are guaranteed to print "hello, world".

fmt.Println(a)

}

// With both types of channels, a close happens before the corresponding receive.

// The write to a happens before the close on c, which happens before the

// corresponding receive on c completes, which happens before the print.

func closed() {

c := make(chan int, 10)

var a string

go func() {

a = "hello, world"

close(c)

}()

<-c

// We are guaranteed to print "hello, world".

fmt.Println(a)

}