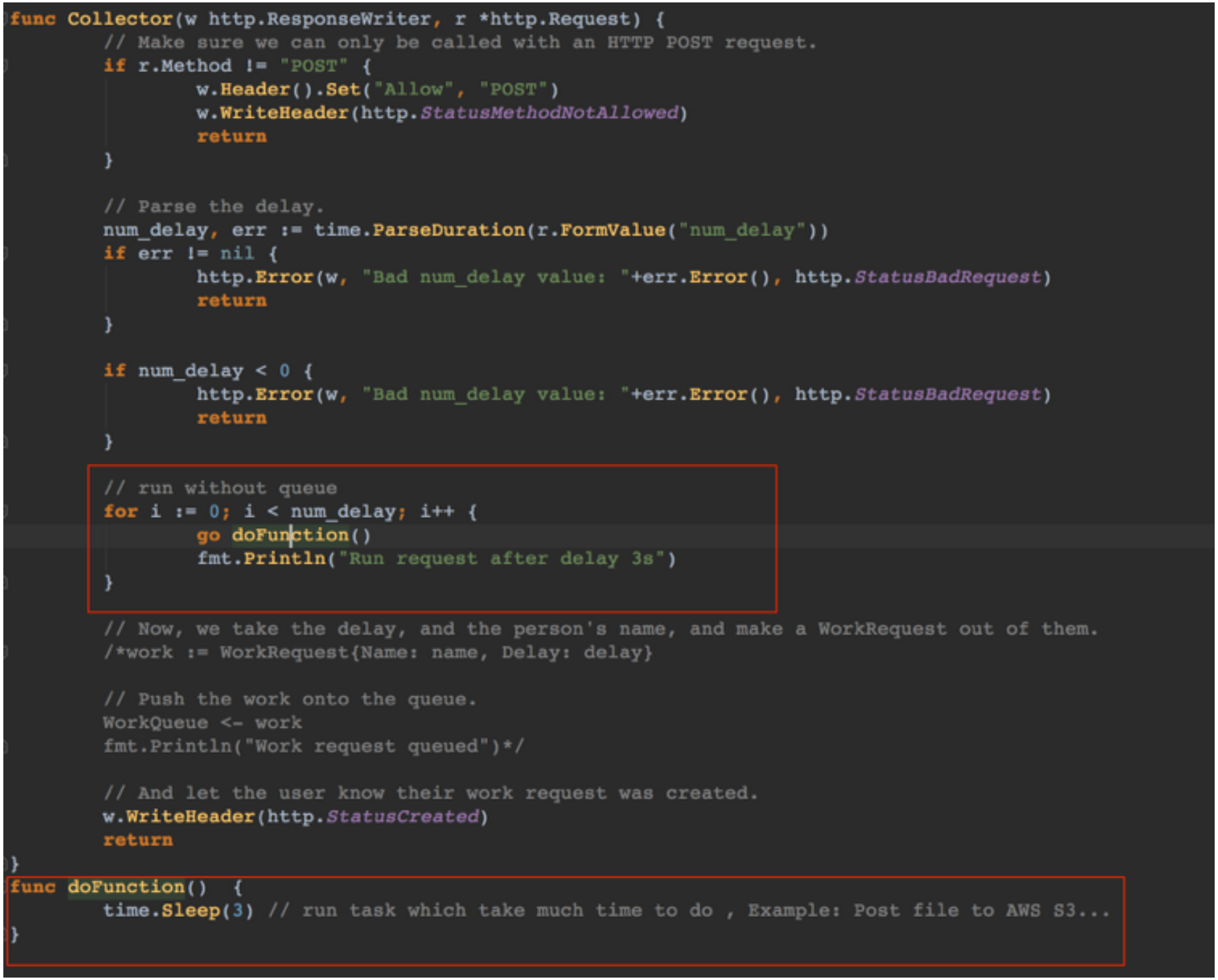
USE QUEUE IN GOLANG TO SCALE SERVER ABILITY AND PERFORMANCE

Author: Michael
Email: michael@cinnamon.is

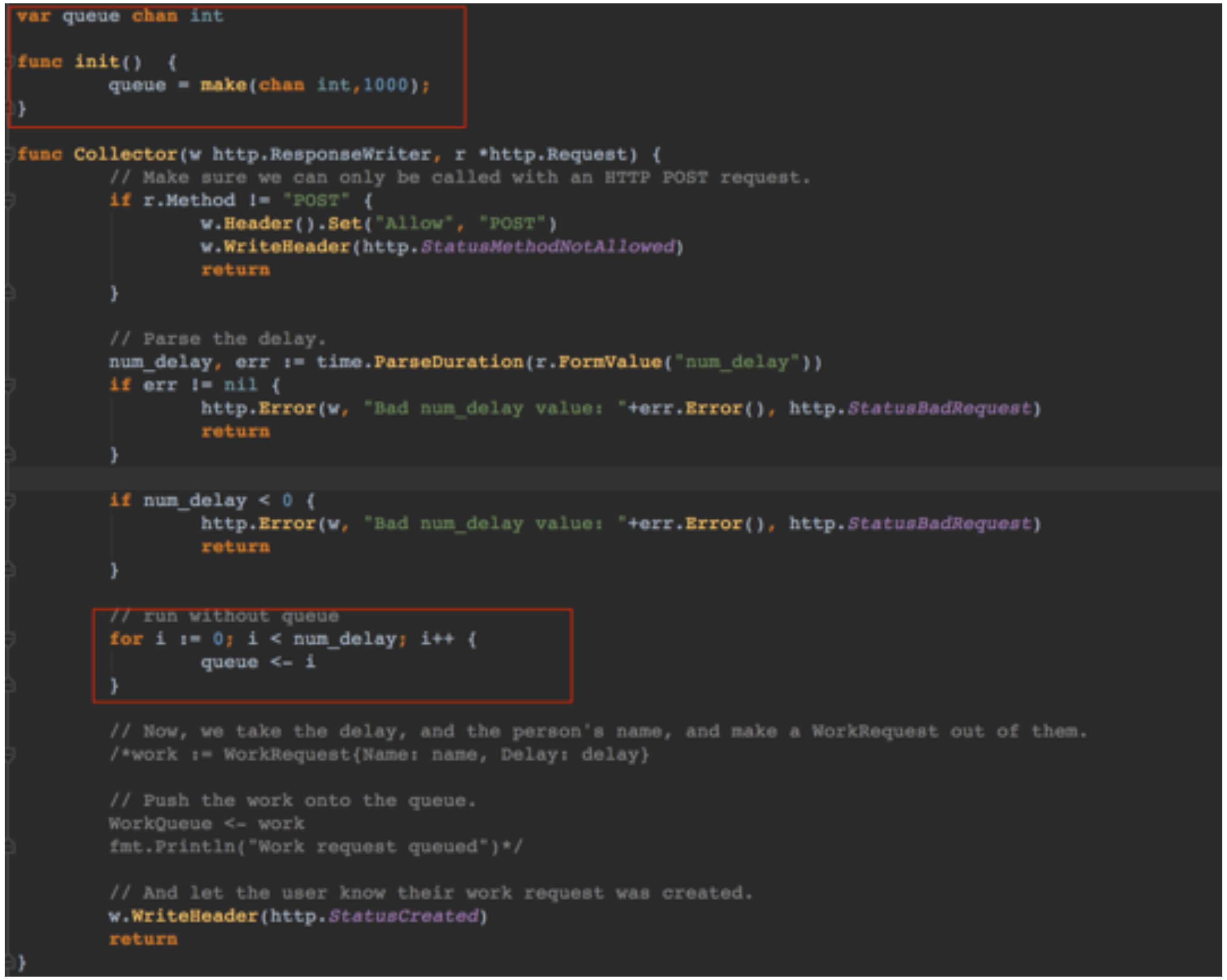
TRY THIS SOURCE CODE



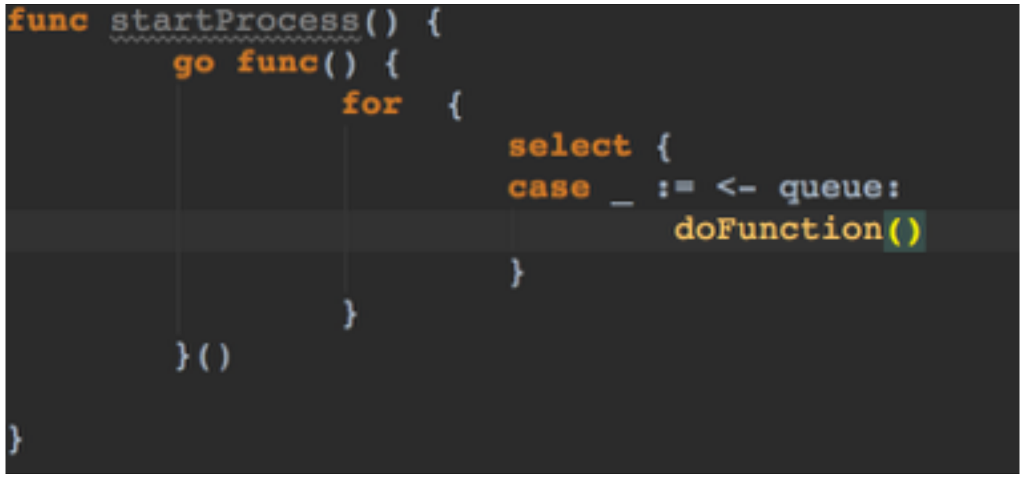
```
func Collector(w http.ResponseWriter, r *http.Request) {
        // Make sure we can only be called with an HTTP POST request.
        if r.Method != "POST" {
                w.Header().Set("Allow", "POST")
                w.WriteHeader(http.StatusMethodNotAllowed)
                return
        }

        // Parse the delay.
        num_delay, err := time.ParseDuration(r.FormValue("num_delay"))
        if err != nil {
                http.Error(w, "Bad num_delay value: "+err.Error(), http.StatusBadRequest)
                return
        }

        if num_delay < 0 {
                http.Error(w, "Bad num_delay value: "+err.Error(), http.StatusBadRequest)
                return
        }

        // run without queue
        for i := 0; i < num_delay; i++ {
                go doFunction()
                fmt.Println("Run request after delay 3s")
        }

        // Now, we take the delay, and the person's name, and make a WorkRequest out of them.
        /*work := WorkRequest{Name: name, Delay: delay}

        // Push the work onto the queue.
        WorkQueue <- work
        fmt.Println("Work request queued")*/

        // And let the user know their work request was created.
        w.WriteHeader(http.StatusCreated)
        return
}
func doFunction()  {
        time.Sleep(3) // run task which take much time to do , Example: Post file to AWS S3...
}
```

- No way to control how many go routines we are spawning!!!
- Since we were getting 1 million POST request per 1 minute ==> of course this code CRASHED very quickly.
- => NEVER DO IT!!!!
- what is better solution?

TRYING AGAIN



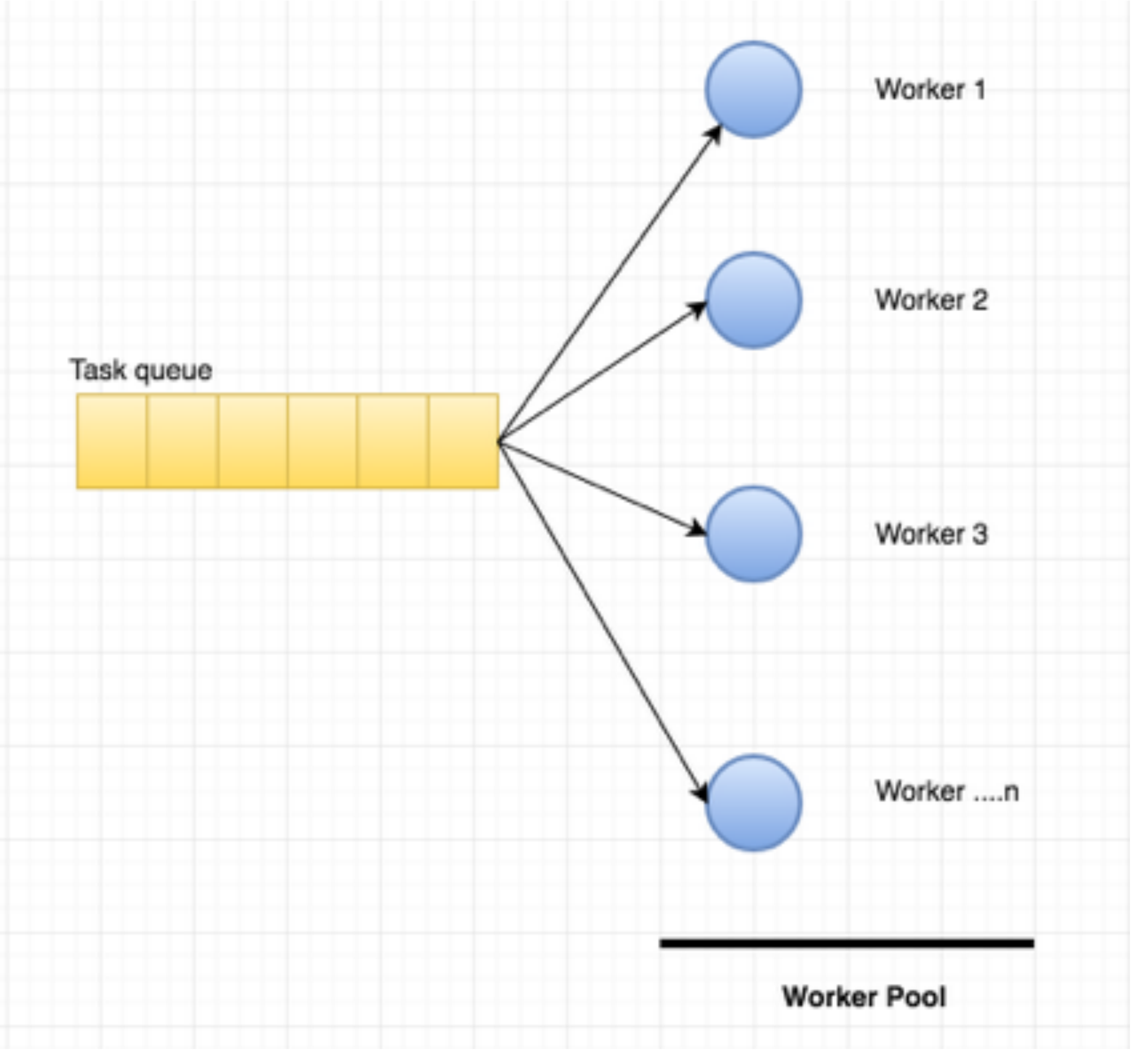
```
var queue chan int

func init()  {
        queue = make(chan int,1000);
}

func Collector(w http.ResponseWriter, r *http.Request) {
        // Make sure we can only be called with an HTTP POST request.
        if r.Method != "POST" {
                w.Header().Set("Allow", "POST")
                w.WriteHeader(http.StatusMethodNotAllowed)
                return
        }

        // Parse the delay.
        num_delay, err := time.ParseDuration(r.FormValue("num_delay"))
        if err != nil {
                http.Error(w, "Bad num_delay value: "+err.Error(), http.StatusBadRequest)
                return
        }

        if num_delay < 0 {
                http.Error(w, "Bad num_delay value: "+err.Error(), http.StatusBadRequest)
                return
        }

        // run without queue
        for i := 0; i < num_delay; i++ {
                queue <- i
        }

        // Now, we take the delay, and the person's name, and make a WorkRequest out of them.
        /*work := WorkRequest{Name: name, Delay: delay}

        // Push the work onto the queue.
        WorkQueue <- work
        fmt.Println("Work request queued")*/

        // And let the user know their work request was created.
        w.WriteHeader(http.StatusCreated)
        return
}
```



```
func startProcess() {
        go func() {
                for  {
                        select {
                        case _ := <- queue:
                                doFunction()
                        }
                }
        }()

}
```

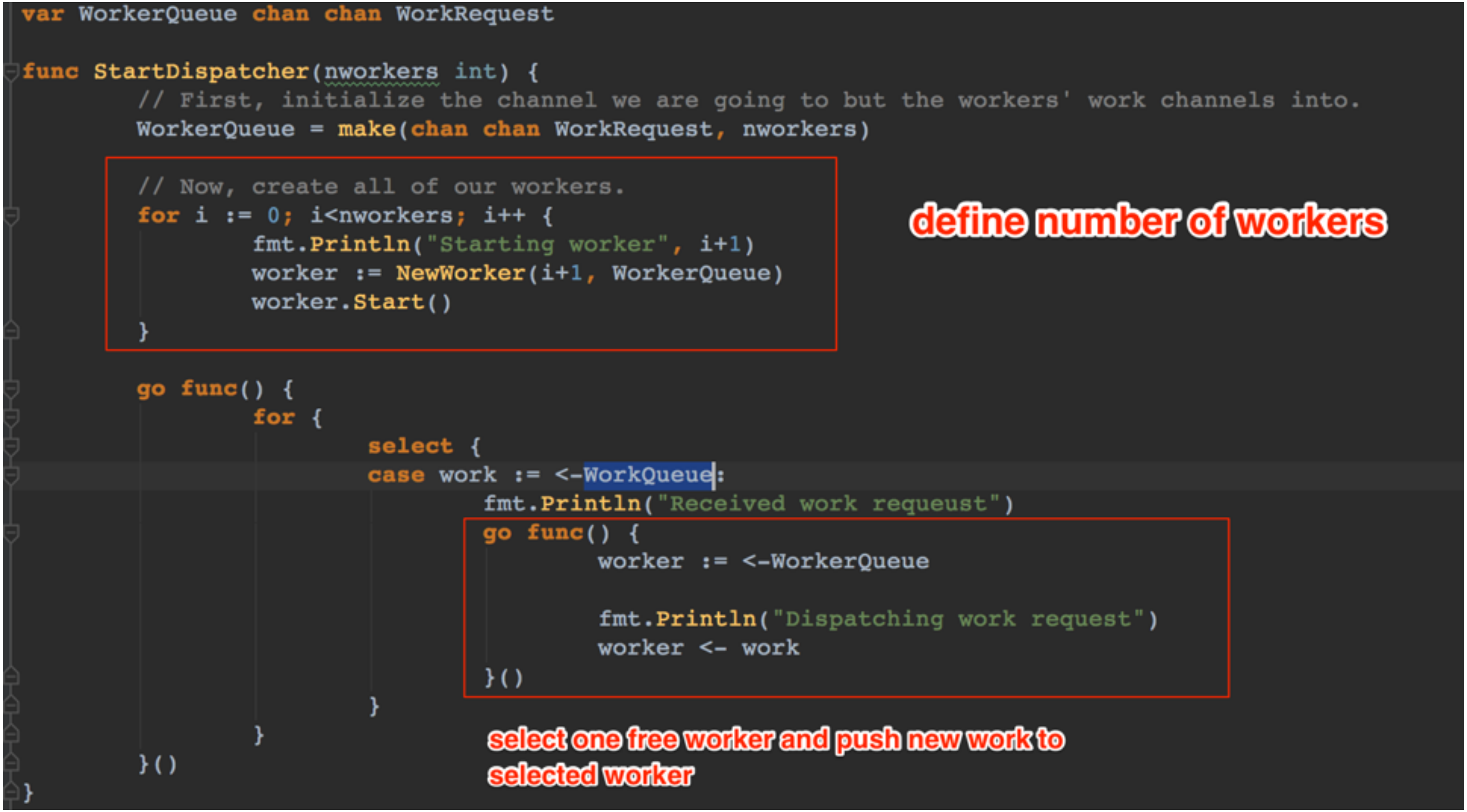
- ➤ performance is better
- ➤ Our server was only run “doFunction” method at a time.
- ➤ the buffered Chanel may be full soon and blocking the request handler ability to queue more items
- ➤ STILL NOT GOOD

AN OTHER SOLUTION?



- Solution with one queue and a lot of worker working together
- Number worker depends on your server resource.
- one worker pool will be create to manage workers.

WORKER POOL AND DISPATCHER

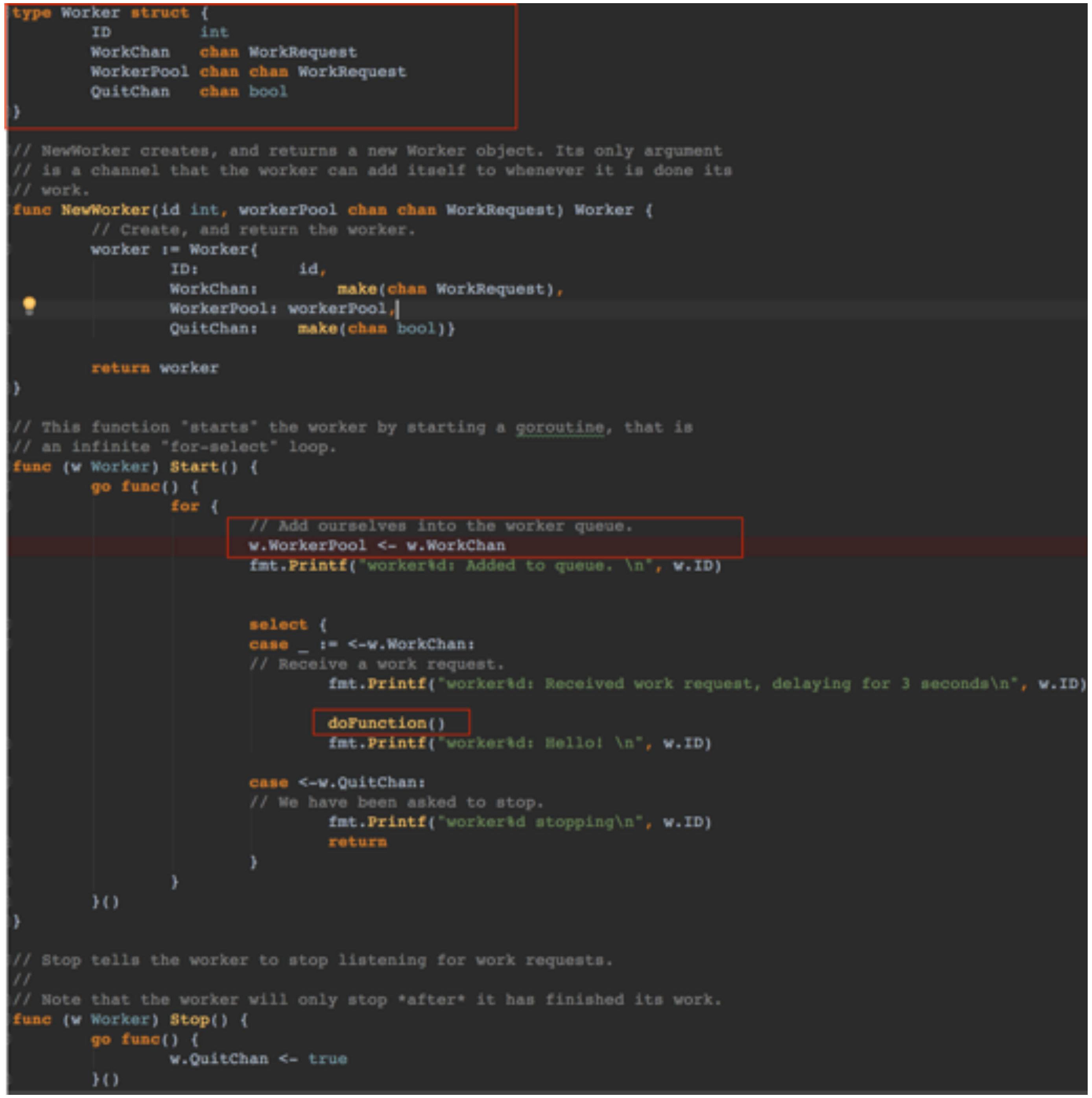


```
var WorkerQueue chan chan WorkRequest

func StartDispatcher(nworkers int) {
    // First, initialize the channel we are going to but the workers' work channels into.
    WorkerQueue = make(chan chan WorkRequest, nworkers)

    // Now, create all of our workers.
    for i := 0; i<nworkers; i++ {
        fmt.Println("Starting worker", i+1)
        worker := NewWorker(i+1, WorkerQueue)
        worker.Start()
    }

    go func() {
        for {
            select {
            case work := <-WorkQueue:
                fmt.Println("Received work requeust")
                go func() {
                    worker := <-WorkerQueue

                    fmt.Println("Dispatching work request")
                    worker <- work
                }()
            }
        }
    }()
}
```

define number of workers

select one free worker and push new work to selected worker

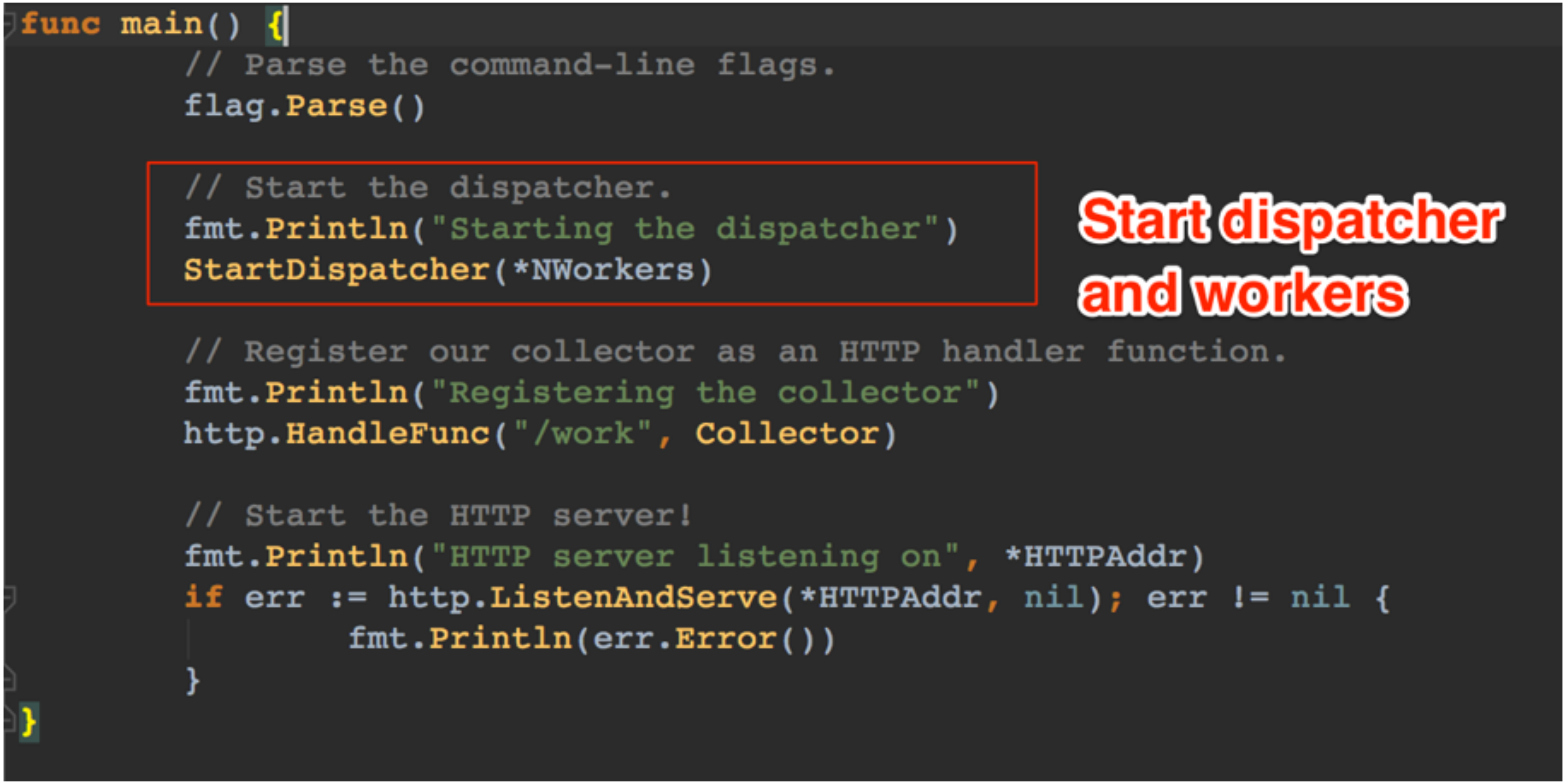
CREATE WORKER



```
type Worker struct {
        ID          int
        WorkChan    chan WorkRequest
        WorkerPool chan chan WorkRequest
        QuitChan    chan bool
}

// NewWorker creates, and returns a new Worker object. Its only argument
// is a channel that the worker can add itself to whenever it is done its
// work.
func NewWorker(id int, workerPool chan chan WorkRequest) Worker {
        // Create, and return the worker.
        worker := Worker{
                ID:          id,
                WorkChan:        make(chan WorkRequest),
                WorkerPool: workerPool,
                QuitChan:    make(chan bool)}

        return worker
}

// This function "starts" the worker by starting a goroutine, that is
// an infinite "for-select" loop.
func (w Worker) Start() {
        go func() {
                for {
                        // Add ourselves into the worker queue.
                        w.WorkerPool <- w.WorkChan
                        fmt.Printf("worker%d: Added to queue. \n", w.ID)


                        select {
                        case _ := <-w.WorkChan:
                        // Receive a work request.
                                fmt.Printf("worker%d: Received work request, delaying for 3 seconds\n", w.ID)

                                doFunction()
                                fmt.Printf("worker%d: Hello! \n", w.ID)

                        case <-w.QuitChan:
                        // We have been asked to stop.
                                fmt.Printf("worker%d stopping\n", w.ID)
                                return
                        }
                }
        }()
}

// Stop tells the worker to stop listening for work requests.
//
// Note that the worker will only stop *after* it has finished its work.
func (w Worker) Stop() {
        go func() {
                w.QuitChan <- true
        }()
```

- ➤ alway register to worker pool one it is free.
- ➤ alway handler new job and quit signals.

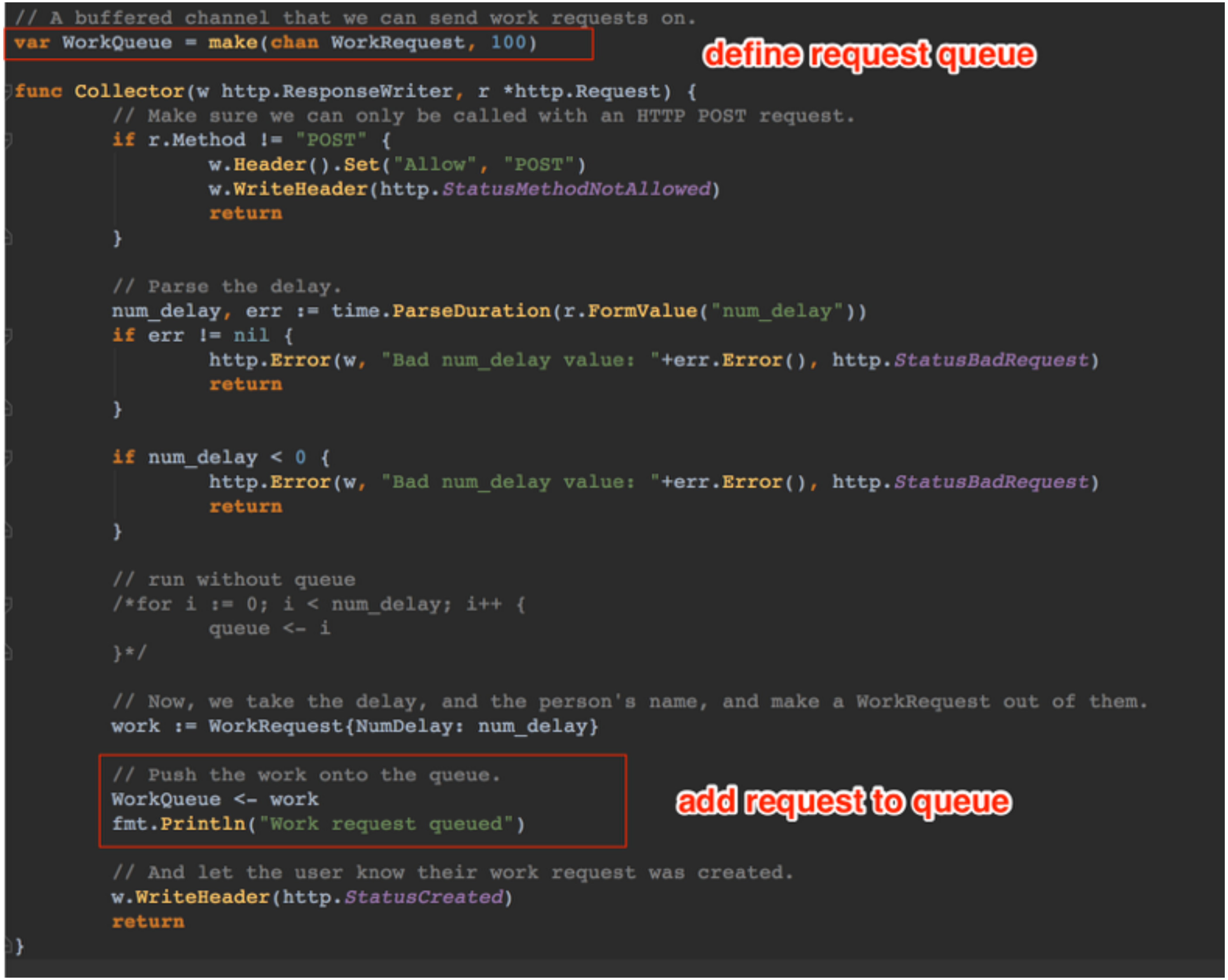
HOW TO USE



```go
func main() {
    // Parse the command-line flags.
    flag.Parse()

    // Start the dispatcher.
    fmt.Println("Starting the dispatcher")
    StartDispatcher(*NWorkers)

    // Register our collector as an HTTP handler function.
    fmt.Println("Registering the collector")
    http.HandleFunc("/work", Collector)

    // Start the HTTP server!
    fmt.Println("HTTP server listening on", *HTTPAddr)
    if err := http.ListenAndServe(*HTTPAddr, nil); err != nil {
        fmt.Println(err.Error())
    }
}
```

Start dispatcher and workers

HOW TO USE



```
// A buffered channel that we can send work requests on.
var WorkQueue = make(chan WorkRequest, 100)

func Collector(w http.ResponseWriter, r *http.Request) {
        // Make sure we can only be called with an HTTP POST request.
        if r.Method != "POST" {
                w.Header().Set("Allow", "POST")
                w.WriteHeader(http.StatusMethodNotAllowed)
                return
        }

        // Parse the delay.
        num_delay, err := time.ParseDuration(r.FormValue("num_delay"))
        if err != nil {
                http.Error(w, "Bad num_delay value: "+err.Error(), http.StatusBadRequest)
                return
        }

        if num_delay < 0 {
                http.Error(w, "Bad num_delay value: "+err.Error(), http.StatusBadRequest)
                return
        }

        // run without queue
        /*for i := 0; i < num_delay; i++ {
                queue <- i
        }*/

        // Now, we take the delay, and the person's name, and make a WorkRequest out of them.
        work := WorkRequest{NumDelay: num_delay}

        // Push the work onto the queue.
        WorkQueue <- work
        fmt.Println("Work request queued")

        // And let the user know their work request was created.
        w.WriteHeader(http.StatusCreated)
        return
}
```

define request queue

add request to queue

RESULT

- we can control how many go routines created
- optimise free worker by worker pool
- optimised your server speed and ability.
- much better if working together with CACHE also.