

Blog

Subscribe and stay up to date with our latest posts.

📌 **SUBSCRIBE** ([HTTPS://WWW.ARDANLABS.COM/SUBSCRIBE](https://www.ardanlabs.com/subscribe))

X



LIVE

Courses Available

LIVE STREAM TRAINING

ENROLL NOW!

(https://www.eventbrite.com/o/ardan-labs-7092394651?utm_source=ardan_website&utm_medium=blog_banner&utm_campaign=website_livestream_promo)

☰ List All Posts (<https://www.ardanlabs.com/all-posts>)

📡 RSS

Pool Go Routines To Process Task Oriented Work

William Kennedy September 14, 2013

✉ (<mailto:bill@ardanlabs.com>)

🐙 (<https://github.com/ardanlabs/gotraining>)

🐦 (<https://twitter.com/goinggodotnet>)

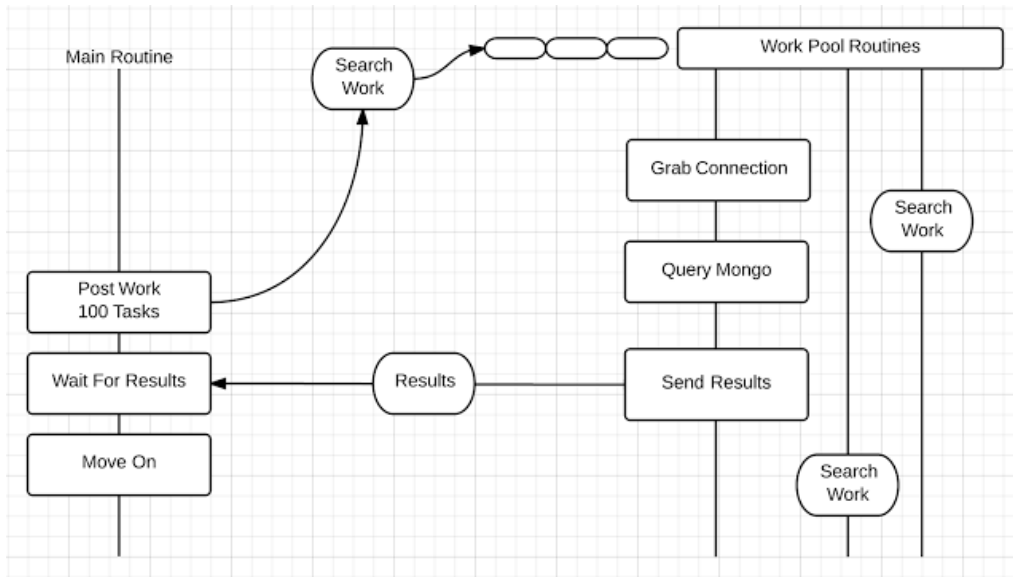
After working in Go for some time now, I learned how to use an unbuffered channel to build a pool of goroutines. I like this implementation better than what is implemented in this post. That being said, this post still has value in what it describes.

(<https://github.com/goinggo/work>)<https://github.com/goinggo/work> (<https://github.com/goinggo/work>)

On more than one occasion I have been asked why I use the Work Pool pattern. Why not just start as many Go routines as needed at any given time to get the work done? My answer is always the same. Depending on the type of work, the computing resources you have available and the constraints that exist within the platform, blindly throwing Go routines to perform work could make things slower and hurt overall system performance and responsiveness.

Every application, system and platform has a breaking point. Resources are not unlimited, whether that is memory, CPU, storage, bandwidth, etc. The ability for our applications to reduce and reuse resources is important. Work pools provide a pattern that can help applications manage resources and provide performance tuning options.

Here is the pattern behind the work pool:



(../images/goinggo/Screen+Shot+2013-09-14+at+10.15.08+AM.png)

In the diagram above, the Main Routine posts 100 tasks into the Work Pool. The Work Pool queues each individual task and once a Go routine is available, the task is dequeued, assigned and performed. When the task is finished, the Go routine becomes available again to process more tasks. The number of Go routines and the capacity of the queue can be configured, which allows for performance tuning the application.

With Go don't think in terms of threads but in Go routines. The Go runtime manages an internal thread pool and schedules the Go routines to run within that pool. Thread pooling is key to minimizing load on the Go runtime and maximizing performance. When we spawn a Go routine, the Go runtime will manage and schedule that Go routine to run on its internal thread pool. No different than the operating system scheduling a thread to run on an available CPU. We can gain the same benefits out of a Go routine pool as we can with a thread pool. Possibly even more.

I have a simple philosophy when it comes to task oriented work, Less is More. I always want to know what is the least number of Go routines, for a particular task, I need that yields the best result. The best result must take into account not only how fast all the tasks are getting done, but also the total impact processing those tasks have on the application, system and platform. You also have to look at the impact short term and long term.

We might be able to yield very fast processing times in the beginning, when the overall load on the application or system is light. Then one day the load changes slightly and the configuration doesn't work anymore. We may not realize that we are crippling a system we are interacting with. We could be pushing a database or a web server too hard and eventually, always at the wrong time, the system shuts down. A burst run of 100 tasks might work great, but sustained over an hour might be deadly.

A Work Pool is not some magic pixie dust that will solve the worlds computing problems. It is a tool you can use for your task oriented work inside your applications. It provides options and some control on how your application performs. As things change, you have flexibility to change with it.

Let's prove the simple case that a Work Pool will process our task oriented work faster than just blindly spawning Go routines. The test application I built runs a task that grabs a MongoDB connection, performs a Find on that MongoDB and retrieves the data. This is something the average business application would do. The application will post this task 100 times into a Work

Pool and do this 5 times to get an average runtime.

To download the code, open a Terminal session and run the following commands:

```
export GOPATH=$HOME/example
go get github.com/goinggo/workpooltest
cd $HOME/example/bin
```

Let's start with a work pool of 100 Go routines. This will simulate the model of spawning as many routines and as we have tasks.

```
./workpooltest 100 off
```

The first argument tells the program to use 100 Go routines in the pool and the second parameter turns off the detailed logging.

Here is the result of using 100 Go routines to process 100 tasks on my Macbook:

```
CPU[8] Routines[100] AmountOfWork[100] Duration[4.599752] MaxRoutines[100] MaxQueued[3]
CPU[8] Routines[100] AmountOfWork[100] Duration[5.799874] MaxRoutines[100] MaxQueued[3]
CPU[8] Routines[100] AmountOfWork[100] Duration[5.325222] MaxRoutines[100] MaxQueued[3]
CPU[8] Routines[100] AmountOfWork[100] Duration[4.652793] MaxRoutines[100] MaxQueued[3]
CPU[8] Routines[100] AmountOfWork[100] Duration[4.552223] MaxRoutines[100] MaxQueued[3]
Average[4.985973]
```

The output tells us a few things about the run:

```
CPU[8]           : The number of cores on my machine
Routines[100]    : The number of routines in the work pool
AmountOfWork[100] : The number of tasks to run
Duration[4.599752] : The amount of time in seconds the run took
MaxRoutines[100] : The max number of routines that were active during the run
MaxQueued[3]     : The max number of tasks waiting in queued during the run
```

Next let's run the program using 64 Go routines:

```
CPU[8] Routines[64] AmountOfWork[100] Duration[4.574367] MaxRoutines[64] MaxQueued[35]
CPU[8] Routines[64] AmountOfWork[100] Duration[4.549339] MaxRoutines[64] MaxQueued[35]
CPU[8] Routines[64] AmountOfWork[100] Duration[4.483110] MaxRoutines[64] MaxQueued[35]
CPU[8] Routines[64] AmountOfWork[100] Duration[4.595183] MaxRoutines[64] MaxQueued[35]
CPU[8] Routines[64] AmountOfWork[100] Duration[4.579676] MaxRoutines[64] MaxQueued[35]
Average[4.556335]
```

Now using 24 Go routines:

```
CPU[8] Routines[24] AmountOfWork[100] Duration[4.595832] MaxRoutines[24] MaxQueued[75]
CPU[8] Routines[24] AmountOfWork[100] Duration[4.430000] MaxRoutines[24] MaxQueued[75]
CPU[8] Routines[24] AmountOfWork[100] Duration[4.477544] MaxRoutines[24] MaxQueued[75]
CPU[8] Routines[24] AmountOfWork[100] Duration[4.550768] MaxRoutines[24] MaxQueued[75]
CPU[8] Routines[24] AmountOfWork[100] Duration[4.629989] MaxRoutines[24] MaxQueued[75]
Average[4.536827]
```

Now using 8 Go routines:

```
CPU[8] Routines[8] AmountOfWork[100] Duration[4.616843] MaxRoutines[8] MaxQueued[91]
CPU[8] Routines[8] AmountOfWork[100] Duration[4.477796] MaxRoutines[8] MaxQueued[91]
CPU[8] Routines[8] AmountOfWork[100] Duration[4.841476] MaxRoutines[8] MaxQueued[91]
CPU[8] Routines[8] AmountOfWork[100] Duration[4.906065] MaxRoutines[8] MaxQueued[91]
CPU[8] Routines[8] AmountOfWork[100] Duration[5.035139] MaxRoutines[8] MaxQueued[91]
Average[4.775464]
```

Let's collect the results of the different runs:

```
100 Go Routines : 4.985973 :
64  Go Routines : 4.556335 : ~430 Milliseconds Faster
24  Go Routines : 4.536827 : ~450 Milliseconds Faster
8   Go Routines : 4.775464 : ~210 Milliseconds Faster
```

This program seems to run the best when we use 3 Go routines per core. This seems to be a magic number because it always yields pretty good results for the programs I write. If we run the program on a machine with more cores, we can increase the Go routine number and take advantage of the extra resources and processing power. That's to say if the MongoDB can handle the extra load for this particular task. Either way we can always adjust the size and capacity of the Work Pool.

We have proved that for this particular task, spawning a Go routine for each task is not the best performing solution. Let's look at the code for the Work Pool and see how it works:

The Work Pool can be found under the following folder if you downloaded the code:

```
cd $HOME/example/src/github.com/goinggo/workpool
```

All the code can be found in a single Go source code file called `workpool.go`. I have removed all the comments and some lines of code to let us focus on the important pieces. Not all the functions are listed in this post as well.

Let's start with the types that make up the Work Pool:

```
type WorkPool struct {
    shutdownQueueChannel chan string
    shutdownWorkChannel  chan struct{}
    shutdownWaitGroup    sync.WaitGroup
    queueChannel         chan poolWork
    workChannel         chan PoolWorker
    queuedWork           int32
    activeRoutines       int32
    queueCapacity        int32
}

type poolWork struct {
    Work          PoolWorker
    ResultChannel chan error
}

type PoolWorker interface {
    DoWork(workRoutine int)
}
```

The `WorkPool` structure is a public type that represents the Work Pool. The implementation uses two channels to run the pool.

The `WorkChannel` is at the heart of the Work Pool. It manages the queue of work that needs to be processed. All of the Go routines that will be performing the work will wait for a signal on this channel.

The `QueueChannel` is used to manage the posting of work into the `WorkChannel` queue. The `QueueChannel` provides acknowledgments to the calling routine that the work has or has not been queued. It also helps to maintain the `QueuedWork` and `QueueCapacity` counters.

The `PoolWork` structure defines the data that is sent into the `QueueChannel` to process enqueueing requests. It contains an interface reference to the users `PoolWorker` object and a channel to receive a confirmation that the task has been enqueued.

The `PoolWorker` interface defines a single function called `DoWork` that has a parameter that represents an internal id for the Go routine that is running the task. This is very helpful for logging and other things that you may want to implement at a per Go Routine level.

The `PoolWorker` interface is the key for accepting and running tasks in the Work Pool. Look at this sample client implementation:

```
type MyTask struct {
    Name string
    WP *workpool.WorkPool
}

func (mt *MyTask) DoWork(workRoutine int) {
    fmt.Println(mt.Name)

    fmt.Printf("*****> WR: %d QW: %d AR: %d\n",
        workRoutine,
        mt.WP.QueuedWork(),
        mt.WP.ActiveRoutines())

    time.Sleep(100 * time.Millisecond)
}

func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())

    workPool := workpool.New(runtime.NumCPU() * 3, 100)

    task := MyTask{
        Name: "A" + strconv.Itoa(i),
        WP: workPool,
    }

    err := workPool.PostWork("main", &task)

    ...
}
```

I create a type called `MyTask` that defines the state I need for the work to be performed. Then I implement a member function for `MyTask` called `DoWork`, which matches the signature of the `PoolWorker` interface. Since `MyTask` implements the `PoolWorker` interface, objects of type `MyTask` are now considered objects of type `PoolWorker`. Now we can pass an object of

type MyTask into the PostWork call.

To learn more about interfaces and object oriented programming in Go read this blog post:

(<http://www.goinggo.net/2013/07/object-oriented-programming-in-go.html>)
<http://www.goinggo.net/2013/07/object-oriented-programming-in-go.html>
<http://www.goinggo.net/2013/07/object-oriented-programming-in-go.html>)

In main I tell the Go runtime to use all of the available CPUs and cores on my machine. Then I create a Work Pool with 24 Go routines. On my current machine I have 8 cores and as we learned above, three Go routines per core is a good starting place. The last parameter tells the Work Pool to create a queue capacity for 100 tasks.

Then I create a MyTask object and post it into the queue. For logging purposes, the first parameter of the PostWork function is a name you can give to the routine making the call. If the err variable is nil after the call, the task has been posted. If not, then most likely you have reached queue capacity and the task could not be posted.

Let's look at the internals of how a WorkPool object is created and started:

```
func New(numberOfRoutines int, queueCapacity int32) *WorkPool {
    workPool = WorkPool{
        shutdownQueueChannel: make(chan string),
        shutdownWorkChannel:  make(chan struct{}),
        queueChannel:          make(chan poolWork),
        workChannel:           make(chan PoolWorker, queueCapacity),
        queuedWork:            0,
        activeRoutines:        0,
        queueCapacity:         queueCapacity,
    }

    for workRoutine := 0; workRoutine < numberOfRoutines; workRoutine++ {
        workPool.shutdownWaitGroup.Add(1)
        go workPool.workRoutine(workRoutine)
    }

    go workPool.queueRoutine()
    return &workPool
}
```

The New function accepts the number of routines and the queue capacity as we saw in the above sample client code. The WorkChannel is a buffered channel which is used as the queue for storing the work we need to process. The QueueChannel is an unbuffered channel used to synchronize access to the WorkChannel buffer, guarantee queuing and to maintain the counters.

To learn more about buffered and unbuffered channels read this web page:

(http://golang.org/doc/effective_go.html#channels)
http://golang.org/doc/effective_go.html#channels
http://golang.org/doc/effective_go.html#channels)

Once the channels are initialized we are able to spawn the Go routines that will perform the work. First we add 1 to the wait group for each Go routine so we can shut it down the pool cleanly it is time. Then we spawn the Go routines so we can process work. The last thing we do is start up the QueueRoutine so we can begin to accept work.

To learn how the shutdown code and WaitGroup works read this web page:

(<http://dave.cheney.net/2013/04/30/curious-channels>)
<http://dave.cheney.net/2013/04/30/curious-channels>
 (<http://dave.cheney.net/2013/04/30/curious-channels>)

Shutting down the Work Pool is done like this:

```
func (wp *WorkPool) Shutdown(goRoutine string) {
    wp.shutdownQueueChannel <- "Down"
    <-wp.shutdownQueueChannel

    close(wp.queueChannel)
    close(wp.shutdownQueueChannel)

    close(wp.shutdownWorkChannel)
    wp.shutdownWaitGroup.Wait()

    close(wp.workChannel)
}
```

The Shutdown function brings down the QueueRoutine first so no more requests can be accepted. Then the ShutdownWorkChannel is closed and the code waits for each Go routine to decrement the WaitGroup counter. Once the last Go routine calls Done on the WaitGroup, the call to Wait will return and the Work Pool is shutdown.

Now let's look at the PostWork and QueueRoutine functions:

```
func (wp *WorkPool) PostWork(goRoutine string, work PoolWorker) (err error) {
    poolWork := poolWork{work, make(chan error)}

    defer close(poolWork.ResultChannel)

    wp.queueChannel <- poolWork
    return <-poolWork.ResultChannel
}
```

```
func (wp *WorkPool) queueRoutine() {
    for {
        select {
        case <-wp.shutdownQueueChannel:
            wp.shutdownQueueChannel <- "Down"
            return

        case queueItem := <-wp.queuechannel:
            if atomic.AddInt32(&wp.queuedWork, 0) == wp.queueCapacity {
                queueItem.ResultChannel <- fmt.Errorf("Thread Pool At Capacity")
                continue
            }

            atomic.AddInt32(&wp.queuedWork, 1)

            wp.workChannel <- queueItem.Work

            queueItem.ResultChannel <- nil
            break
        }
    }
}
```

The idea behind the PostWork and QueueRoutine functions are to serialize access to the

WorkChannel buffer, guarantee queuing and to maintain the counters. Work is always placed at the end of the WorkChannel buffer by the Go runtime when it is sent into the channel.

The highlighted code shows all the communication points. When the QueueChannel is signaled, the QueueRoutine receives the work. Queue capacity is checked and if there is room, the user PoolWorker object is queued into the WorkChannel buffer. Finally the calling routine is signaled back that everything is queued.

Last let's look at the WorkRoutine functions:

```
func (wp *WorkPool) workRoutine(workRoutine int) {
    for {
        select {
            case <-wp.shutdownworkchannel:
                wp.shutdownWaitGroup.Done()
                return

            case poolWorker := <-wp.workChannel:
                wp.safelyDoWork(workRoutine, poolWorker)
                break
        }
    }
}
```

```
func (wp *WorkPool) safelyDoWork(workRoutine int, poolWorker PoolWorker) {
    defer catchPanic(nil, "workRoutine", "workpool.WorkPool", "SafelyDoWork")
    defer atomic.AddInt32(&wp.activeRoutines, -1)

    atomic.AddInt32(&wp.queuedWork, -1)
    atomic.AddInt32(&wp.activeRoutines, 1)

    poolWorker.DoWork(workRoutine)
}
```

The Go runtime takes care of assigning work to a Go routine in the pool by signaling the WorkChannel for a particular Go routine that is waiting. When the channel is signaled, the Go runtime passes the work that is at the head of the channel buffer. The channel buffer acts as a queue, FIFO.

If all the Go routines are busy, then none will be waiting on the WorkChannel, so all remaining work has to wait. As soon as a routine completes its work, it returns to wait again on the WorkChannel. If there is work in the channel buffer, the Go runtime will signal the Go routine to wake up again.

The code uses the SafelyDo pattern for processing work. At this point the code is calling into user code and could panic. You don't want anything to cause the Go routine to terminate. Notice the use of the first defer statement. It catches any panics and stops them in their tracks.

The rest of the code safely increments and decrements the counters and calls into the user routine via the Interface.

To learn more about catch panics read this blog post:

(<http://www.goinggo.net/2013/06/understanding-defer-panic-and-recover.html>)<http://www.goinggo.net/2013/06/understanding-defer-panic-and-recover.html>

(<http://www.goinggo.net/2013/06/understanding-defer-panic-and-recover.html>)

That's the heart of the code and how it implements the pattern. The WorkPool really shows the elegance and grace of channels. With very little code I was able to implement a pool of Go routines to process work. Adding guaranteed queuing and maintain the counters was a breeze.

Download the code from the GoingGo repository on Github and try it for yourself.

Go Training

We have taught Go to thousands of developers all around the world since 2014. There is no other company that has been doing it longer and our material has proven to help jump start developers 6 to 12 months ahead of their knowledge of Go. We know what knowledge developers need in order to be productive and efficient when writing software in Go.

Our classes are perfect for both experienced and beginning engineers. We start every class from the beginning and get very detailed about the internals, mechanics, specification, guidelines, best practices and design philosophies. We cover a lot about "if performance matters" with a focus on mechanical sympathy, data oriented design, decoupling and writing production software.


Interested in Ultimate Go Corporate Training and special pricing?

Let's Talk Corporate Training! (<mailto:hello@ardanlabs.com>)
Subject=Let's%20Talk%20Ultimate%20Go%20Corporate%20Training%20and%20special%20pricing!

Join Our Online Education Program


Our courses have been designed from training over 4,000 engineers since 2013 and they go beyond just being a language course. Our goal is to challenge every student to think about what they are doing and why.

✚ENROLL NOW (HTTPS://EDUCATION.ARDANLABS.COM)

 (/) [Training \(/training\)](/training)
[Development \(/development\)](/development)
[DevOps \(/devops-consulting\)](/devops-consulting)
[Staffing \(/staffing\)](/staffing)
[UI/UX \(/ui-ux\)](/ui-ux)
[Machine Learning \(/machine-learning\)](/machine-learning)

[About \(/about\)](/about)
[Blog \(/blog\)](/blog)
[Contact \(/my/contact-us\)](/my/contact-us)
[Careers \(/careers\)](/careers)
[My Lab \(/my/lab\)](/my/lab)
[Terms \(/terms-service\)](/terms-service)
[Privacy \(/privacy-policy\)](/privacy-policy)

Reach out to us

 (https://www.instagram.com/ardan_labs/)
 (<https://twitter.com/ardanlabs>)
 (<https://github.com/ardanlabs>)
 (888) 72 ARDAN
888 722-7326
info@ardanlabs.com (<mailto:info@ardanlabs.com>)

Ardan labs Copyrights © 2020