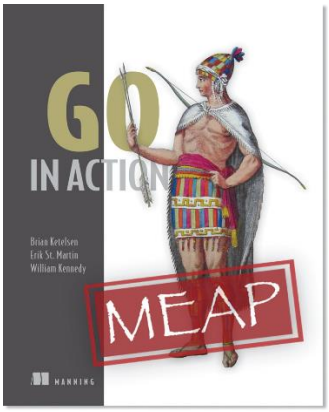
Go in Action: Exploring the Work Package

By Brian Ketelsen, Erik St. Martin, and William Kennedy



In this article we share section 7.3 from the Go in Action book. This section explores a package named work that provides a concurrency pattern for pooling a set of goroutines to perform and control work.

The purpose of the work package is to show how you can use an unbuffered channel to create a pool of goroutines to perform and control the amount of work that gets done concurrently. This is a better approach that using a buffered channel of some arbitrary static size that acts as a queue of work and throwing a bunch of goroutines at it.

Unbuffered channels provide a guarantee that data has been exchanged between two goroutines. The approach this package takes by using an unbuffered channel allows the user to know when the pool is performing the work and pushes back when it can't accept any more work because it is busy. No work is ever lost or stuck in queue that has no guarantee it will ever be worked on.

Let's take a look at the `work.go` code file from the `work` package:

Listing 1

```
chapter7/patterns/work/work.go

01 // Example provided with help from Jason Waldrip.
02 // Package work manages a pool of goroutines to perform work.
03 package work
04
05 import "sync"
06
07 // Worker must be implemented by types that want to use
08 // the work pool.
09 type Worker interface {
10    Task()
11 }
12
13 // Pool provides a pool of goroutines that can execute any Worker
14 // tasks that are submitted.
15 type Pool struct {
16    work chan Worker
17    wg sync.WaitGroup
18 }
19
```

```
20 // New creates a new work pool.
21 func New(maxGoroutines int) *Pool {
22    p := Pool{
23          tasks: make(chan Worker),
24    }
25
26    p.wg.Add(maxGoroutines)
27    for i := 0; i < maxGoroutines; i++ {
28          go func() {
29                for w := range p.work {
30                      w.Task()
31                }
32                p.wg.Done()
33          }()
34    }
35
36    return &p
37 }
38
39 // Run submits work to the pool.
40 func (p *Pool) Run(w Worker) {
41    p.work <- w
42 }
43
44 // Shutdown waits for all the goroutines to shutdown.
45 func (p *Pool) Shutdown() {
46    close(p.tasks)
47    p.wg.Wait()
48 }
```

The work package in listing 2 starts off with the declaration of an interface named `Worker` and a struct named `Pool`:

Listing 2

```
07 // Worker must be implemented by types that want to use
08 // the work pool.
09 type Worker interface {
10    Task()
11 }
12
13 // Pool provides a pool of goroutines that can execute any Worker
14 // tasks that are submitted.
15 type Pool struct {
16    work chan Worker
17    wg sync.WaitGroup
18 }
```

The `Worker` interface in listing 1 on line 10 declares a single method called *Task*. Then on line 15, a struct named `Pool` is declared which is the type that implements the pool of goroutines and will have methods that process the work. The type declares two fields, one named `work` which is a channel of the `Worker` interface type and a `sync.WaitGroup` named *wg*.

Next, let's look at the factory function for the `work` package:

Listing 3

```
20 // New creates a new work pool.
21 func New(maxGoroutines int) *Pool {
22    p := Pool{
23          work: make(chan Worker),
24    }
25
26    p.wg.Add(maxGoroutines)
27    for i := 0; i < maxGoroutines; i++ {
28          go func() {
29                for w := range p.work {
30                      w.Task()
31                }
32                p.wg.Done()
33          }()
34    }
35
36    return &p
37 }
```

Listing 3 shows the `New` function that is used to create work pool that is configured with a fixed set number of goroutines. The number of goroutines is passed in as a parameter to the `New` function. On line 22, a value of type `Pool` is created and the *work* field is initialized with an unbuffered channel.

Then on line 26, the `WaitGroup` is initialized and on lines 27 through 34 the specified number of goroutines are created. The goroutine just receives interface values of type `Worker` and calls the *Task* method on those values:

Listing 4

```
28          go func() {
29                for w := range w.work {
30                      w.Task()
31                }
32                p.wg.Done()
33          }()
```

The `for range` loop blocks until there is a `Worker` interface value to receive on the `work` channel. When a value is received, the `Task` method is called. Once the `work` channel is closed, the `for range` loop ends and the call to `Done` on the `WaitGroup` is called. Then the goroutine terminates.

Now that we can create a pool of goroutines that can wait for and execute work, let's look at how work is submitted into the pool:

Listing 5

```
39 // Run submits work to the pool.
40 func (p *Pool) Run(w Worker) {
41    w.work <- w
```

```
42 }
```

Listing 5 shows the `Run` method. This method is used to submit work into the pool. It accepts an interface value of type `Worker` and sends that value through the `work` channel. Since the `work` channel is an unbuffered channel, the caller must wait for a goroutine from the pool to receive it. This is what we want because the caller needs the guarantee that the work being submitted is being worked on once the call to `Run` returns.

At some point the work pool need to be shutdown. This is where the `Shutdown` method comes in:

Listing 6

```
44 // Shutdown waits for all the goroutines to shutdown.
45 func (p *Pool) Shutdown() {
46    close(p.work)
47    p.wg.Wait()
48 }
```

The `Shutdown` method in listing 6 does two things. First, it closes the `work` channel which causes all of the goroutines in the pool to shut down and call the `Done` method off the `WaitGroup`. Then the `Shutdown` method calls the `Wait` method on the `WaitGroup` which causes the `Shutdown` method to wait for all the goroutines to report they have terminated.

Now that we have seen the code for the work package and learned how it works, let's review the test program in the main.go source code file:

Listing 7

```
chapter7/patterns/work/main/main.go

01 // This sample program demonstrates how to use the work package
02 // to use a pool of goroutines to get work done.
03 package main
04
05 import (
06    "log"
07    "sync"
08    "time"
09
10    "github.com/goinaction/code/chapter7/patterns/work"
11 )
12
13 // names provides a set of names to display.
14 var names = []string{
15    "steve",
16    "bob",
17    "mary",
18    "therese",
19    "jason",
20 }
21
22 // namePrinter provides special support for printing names.
23 type namePrinter struct {
24    name string
```

```
25 }
26
27 // Task implements the Worker interface.
28 func (m *namePrinter) Task() {
29    log.Println(m.name)
30    time.Sleep(time.Second)
31 }
32
33 // main is the entry point for all Go programs.
34 func main() {
35    // Create a work pool with 2 goroutines.
36    p := work.New(2)
37
38    var wg sync.WaitGroup
39    wg.Add(100 * len(names))
40
41    for i := 0; i < 100; i++ {
42          // Iterate over the slice of names.
43          for _, name := range names {
44                // Create a namePrinter and provide the
45                // specific name.
46                np := namePrinter{
47                      name: name,
48                }
49
50                go func() {
51                      // Submit the task to be worked on. When RunTask
52                      // returns we know it is being handled.
53                      p.Run(&np)
54                      wg.Done()
55                }()
56          }
57    }
58
59    wg.Wait()
60
61    // Shutdown the work pool and wait for all existing work
62    // to be completed.
63    p.Shutdown()
64 }
```

Listing 7 shows the test program that uses the package `work` to perform the displaying of names. The code starts out on line 14 with the declaration of a package level variable named `names` which is declared as a slice of strings. The slice is also initialized with five names. Then a type named `namePrinter` is declared:

```
22 // namePrinter provides special support for printing names.
23 type namePrinter struct {
24    name string
25 }
26
27 // Task implements the Worker interface.
28 func (m *namePrinter) Task() {
29    log.Println(m.name)
30    time.Sleep(time.Second)
31 }
```

On line 23 in listing 7, the type `namePrinter` is declared and the implementation of the `Worker` interface follows. The purpose of the work is to display names to the screen. The type contains a single field named `name` which will contain the name to display. The implementation of the `Worker` interface uses the `log.Println` function to display the name and then waits a second before returning. The second wait is just to slow the test program down so we can see the concurrency is action.

With the implementation of the `Worker` interface, we can look at the code inside the `main` function:

```
33 // main is the entry point for all Go programs.
34 func main() {
35    // Create a work pool with 2 goroutines.
36    p := work.New(2)
37
38    var wg sync.WaitGroup
39    wg.Add(100 * len(names))
40
41    for i := 0; i < 100; i++ {
42          // Iterate over the slice of names.
43          for _, name := range names {
44                // Create a namePrinter and provide the
45                // specific name.
46                np := namePrinter{
47                      name: name,
48                }
49
50                go func() {
51                      // Submit the task to be worked on. When RunTask
52                      // returns we know it is being handled.
53                      p.Run(&np)
54                      wg.Done()
55                }()
56          }
57    }
58
59    wg.Wait()
60
61    // Shutdown the work pool and wait for all existing work
62    // to be completed.
63    p.Shutdown()
64 }
```

On line 36 in listing 7, the `New` function from the `work` package is called to create the work pool. The number 2 is passed into the call indicating the pool should only contain two goroutines. Then on lines 38 and 39 a `WaitGroup` is declared and initialized to each goroutine that is going to be created. In this case, a goroutine for each name in the `names` slice is going to be created 100 times. This is to create a lot of goroutines competing to submit work to the pool.

Then on line 41 and 43, inner and outer `for` loops are declared to create all the goroutines. Inside each iteration of the inner loop, a value of type `namePrinter` is created and provided with a name to print. Then on line 50, an anonymous function is declared and created as a goroutine. The goroutine calls the `Run` method against the work pool to submit the `namePrinter` value to the pool. Once a goroutine from the work pool receives the value, the call to `Run` returns. This is turn causes the goroutine to decrement the `WaitGroup` count and terminate.

Once all the goroutines are created, the *main* function calls `Wait` on the `WaitGroup`. The function will wait until all the goroutines that were created submit their work. Once `Wait` returns, the work pool is shut down by calling the `Shutdown` method. This method will not return until all the work is complete. In our case, there would only be two outstanding pieces of work by this time.