



The Go Blog

Go Concurrency Patterns: Pipelines and cancellation

13 March 2014

Introduction

Go's concurrency primitives make it easy to construct streaming data pipelines that make efficient use of I/O and multiple CPUs. This article presents examples of such pipelines, highlights subtleties that arise when operations fail, and introduces techniques for dealing with failures cleanly.

What is a pipeline?

There's no formal definition of a pipeline in Go; it's just one of many kinds of concurrent programs. Informally, a pipeline is a series of *stages* connected by channels, where each stage is a group of goroutines running the same function. In each stage, the goroutines

- receive values from *upstream* via *inbound* channels
- perform some function on that data, usually producing new values
- send values *downstream* via *outbound* channels

Each stage has any number of inbound and outbound channels, except the first and last stages, which have only outbound or inbound channels, respectively. The first stage is sometimes called the *source* or *producer*; the last stage, the *sink* or *consumer*.

We'll begin with a simple example pipeline to explain the ideas and techniques. Later, we'll present a more realistic example.

Squaring numbers

[Next article](#)

[The Go Gopher](#)

[Previous article](#)

[Go talks at FOSDEM 2014](#)

Links

[golang.org](#)

[Install Go](#)

[A Tour of Go](#)

[Go Documentation](#)

[Go Mailing List](#)

[Go on Google+](#)

[Go+ Community](#)

[Go on Twitter](#)

[Blog index](#)

Consider a pipeline with three stages.

The first stage, `gen`, is a function that converts a list of integers to a channel that emits the integers in the list. The `gen` function starts a goroutine that sends the integers on the channel and closes the channel when all the values have been sent:

```
func gen(nums ...int) <-chan int {
    out := make(chan int)
    go func() {
        for _, n := range nums {
            out <- n
        }
        close(out)
    }()
    return out
}
```

The second stage, `sq`, receives integers from a channel and returns a channel that emits the square of each received integer. After the inbound channel is closed and this stage has sent all the values downstream, it closes the outbound channel:

```
func sq(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}
```

The `main` function sets up the pipeline and runs the final stage: it receives values from the second stage and prints each one, until the channel is closed:

```
func main() {
    // Set up the pipeline.
    c := gen(2, 3)
    out := sq(c)

    // Consume the output.
```

```

    fmt.Println(<-out) // 4
    fmt.Println(<-out) // 9
}

```

Since `sq` has the same type for its inbound and outbound channels, we can compose it any number of times. We can also rewrite `main` as a range loop, like the other stages:

```

func main() {
    // Set up the pipeline and consume the output.
    for n := range sq(sq(gen(2, 3))) {
        fmt.Println(n) // 16 then 81
    }
}

```

Fan-out, fan-in

Multiple functions can read from the same channel until that channel is closed; this is called *fan-out*. This provides a way to distribute work amongst a group of workers to parallelize CPU use and I/O.

A function can read from multiple inputs and proceed until all are closed by multiplexing the input channels onto a single channel that's closed when all the inputs are closed. This is called *fan-in*.

We can change our pipeline to run two instances of `sq`, each reading from the same input channel. We introduce a new function, `merge`, to fan in the results:

```

func main() {
    in := gen(2, 3)

    // Distribute the sq work across two goroutines that both read from
    // in.
    c1 := sq(in)
    c2 := sq(in)

    // Consume the merged output from c1 and c2.
    for n := range merge(c1, c2) {
        fmt.Println(n) // 4 then 9, or 9 then 4
    }
}

```

The `merge` function converts a list of channels to a single channel by starting a goroutine for each inbound channel that copies the values to the sole outbound channel. Once all the output goroutines have been started, `merge` starts one more goroutine to close the outbound channel after all sends on that channel are done.

Sends on a closed channel panic, so it's important to ensure all sends are done before calling `close`. The [sync.WaitGroup](#) type provides a simple way to arrange this synchronization:

```
func merge(cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)

    // Start an output goroutine for each input channel in cs.  output
    // copies values from c to out until c is closed, then calls wg.Done.
    output := func(c <-chan int) {
        for n := range c {
            out <- n
        }
        wg.Done()
    }
    wg.Add(len(cs))
    for _, c := range cs {
        go output(c)
    }

    // Start a goroutine to close out once all the output goroutines are
    // done.  This must start after the wg.Add call.
    go func() {
        wg.Wait()
        close(out)
    }()
    return out
}
```

Stopping short

There is a pattern to our pipeline functions:

- stages close their outbound channels when all the send operations are done.
- stages keep receiving values from inbound channels until those channels are closed.

This pattern allows each receiving stage to be written as a range loop and ensures that all goroutines exit once all values have been successfully sent downstream.

But in real pipelines, stages don't always receive all the inbound values. Sometimes this is by design: the receiver may only need a subset of values to make progress. More often, a stage exits early because an inbound value represents an error in an earlier stage. In either case the receiver should not have to wait for the remaining values to arrive, and we want earlier stages to stop producing values that later stages don't need.

In our example pipeline, if a stage fails to consume all the inbound values, the goroutines attempting to send those values will block indefinitely:

```
// Consume the first value from output.
out := merge(c1, c2)
fmt.Println(<-out) // 4 or 9
return
// Since we didn't receive the second value from out,
// one of the output goroutines is hung attempting to send it.
}
```

This is a resource leak: goroutines consume memory and runtime resources, and heap references in goroutine stacks keep data from being garbage collected. Goroutines are not garbage collected; they must exit on their own.

We need to arrange for the upstream stages of our pipeline to exit even when the downstream stages fail to receive all the inbound values. One way to do this is to change the outbound channels to have a buffer. A buffer can hold a fixed number of values; send operations complete immediately if there's room in the buffer:

```
c := make(chan int, 2) // buffer size 2
c <- 1 // succeeds immediately
c <- 2 // succeeds immediately
c <- 3 // blocks until another goroutine does <-c and receives 1
```

When the number of values to be sent is known at channel creation time, a buffer can simplify the code. For example, we can rewrite `gen` to copy the list of integers into a buffered channel and avoid creating a new goroutine:

```
func gen(nums ...int) <-chan int {
    out := make(chan int, len(nums))
    for _, n := range nums {
```

```

        out <- n
    }
    close(out)
    return out
}

```

Returning to the blocked goroutines in our pipeline, we might consider adding a buffer to the outbound channel returned by `merge`:

```

func merge(cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int, 1) // enough space for the unread inputs
    // ... the rest is unchanged ...

```

While this fixes the blocked goroutine in this program, this is bad code. The choice of buffer size of 1 here depends on knowing the number of values `merge` will receive and the number of values downstream stages will consume. This is fragile: if we pass an additional value to `gen`, or if the downstream stage reads any fewer values, we will again have blocked goroutines.

Instead, we need to provide a way for downstream stages to indicate to the senders that they will stop accepting input.

Explicit cancellation

When `main` decides to exit without receiving all the values from `out`, it must tell the goroutines in the upstream stages to abandon the values they're trying to send. It does so by sending values on a channel called `done`. It sends two values since there are potentially two blocked senders:

```

func main() {
    in := gen(2, 3)

    // Distribute the sq work across two goroutines that both read from
    // in.
    c1 := sq(in)
    c2 := sq(in)

    // Consume the first value from output.
    done := make(chan struct{}, 2)
    out := merge(done, c1, c2)

```

```

fmt.Println(<-out) // 4 or 9

// Tell the remaining senders we're leaving.
done <- struct{}{}
done <- struct{}{}
}

```

The sending goroutines replace their send operation with a `select` statement that proceeds either when the send on `out` happens or when they receive a value from `done`. The value type of `done` is the empty struct because the value doesn't matter: it is the receive event that indicates the send on `out` should be abandoned. The output goroutines continue looping on their inbound channel, `c`, so the upstream stages are not blocked. (We'll discuss in a moment how to allow this loop to return early.)

```

func merge(done <-chan struct{}, cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)

    // Start an output goroutine for each input channel in cs.  output
    // copies values from c to out until c is closed or it receives a
    value
    // from done, then output calls wg.Done.
    output := func(c <-chan int) {
        for n := range c {
            select {
            case out <- n:
            case <-done:
            }
        }
        wg.Done()
    }
    // ... the rest is unchanged ...
}

```

This approach has a problem: *each* downstream receiver needs to know the number of potentially blocked upstream senders and arrange to signal those senders on early return. Keeping track of these counts is tedious and error-prone.

We need a way to tell an unknown and unbounded number of goroutines to stop sending their values downstream. In Go, we can do this by closing a channel, because [a receive operation on a closed channel can always proceed immediately, yielding the element type's zero value.](#)

This means that `main` can unblock all the senders simply by closing the `done` channel. This close is effectively a broadcast signal to the senders. We extend `each` of our pipeline functions to accept `done` as a parameter and arrange for the close to happen via a `defer` statement, so that all return paths from `main` will signal the pipeline stages to exit.

```
func main() {
    // Set up a done channel that's shared by the whole pipeline,
    // and close that channel when this pipeline exits, as a signal
    // for all the goroutines we started to exit.
    done := make(chan struct{})
    defer close(done)

    in := gen(done, 2, 3)

    // Distribute the sq work across two goroutines that both read from
    in.
    c1 := sq(done, in)
    c2 := sq(done, in)

    // Consume the first value from output.
    out := merge(done, c1, c2)
    fmt.Println(<-out) // 4 or 9

    // done will be closed by the deferred call.
}
```

Each of our pipeline stages is now free to return as soon as `done` is closed. The `output` routine in `merge` can return without draining its inbound channel, since it knows the upstream sender, `sq`, will stop attempting to send when `done` is closed. `output` ensures `wg.Done` is called on all return paths via a `defer` statement:

```
func merge(done <-chan struct{}, cs ...<-chan int) <-chan int {
    var wg sync.WaitGroup
    out := make(chan int)

    // Start an output goroutine for each input channel in cs.  output
    // copies values from c to out until c or done is closed, then calls
    // wg.Done.
    output := func(c <-chan int) {
        defer wg.Done()
        for n := range c {
            select {
```

```

        case out <- n:
        case <-done:
            return
        }
    }
// ... the rest is unchanged ...

```

Similarly, `sq` can return as soon as `done` is closed. `sq` ensures its `out` channel is closed on all return paths via a `defer` statement:

```

func sq(done <-chan struct{}, in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for n := range in {
            select {
            case out <- n * n:
            case <-done:
                return
            }
        }
    }()
    return out
}

```

Here are the guidelines for pipeline construction:

- stages close their outbound channels when all the send operations are done.
- stages keep receiving values from inbound channels until those channels are closed or the senders are unblocked.

Pipelines unblock senders either by ensuring there's enough buffer for all the values that are sent or by explicitly signalling senders when the receiver may abandon the channel.

Digesting a tree

Let's consider a more realistic pipeline.

MD5 is a message-digest algorithm that's useful as a file checksum. The command line utility `md5sum` prints digest values for a list of files.

```
% md5sum *.go
d47c2bbc28298ca9befdfbc5d3aa4e65  bounded.go
ee869af31f83cbb2d10ee81b2b831dc  parallel.go
b88175e65fdcbc01ac08aaf1fd9b5e96  serial.go
```

Our example program is like `md5sum` but instead takes a single directory as an argument and prints the digest values for each regular file under that directory, sorted by path name.

```
% go run serial.go .
d47c2bbc28298ca9befdfbc5d3aa4e65  bounded.go
ee869af31f83cbb2d10ee81b2b831dc  parallel.go
b88175e65fdcbc01ac08aaf1fd9b5e96  serial.go
```

The main function of our program invokes a helper function `MD5All`, which returns a map from path name to digest value, then sorts and prints the results:

```
func main() {
    // Calculate the MD5 sum of all files under the specified directory,
    // then print the results sorted by path name.
    m, err := MD5All(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var paths []string
    for path := range m {
        paths = append(paths, path)
    }
    sort.Strings(paths)
    for _, path := range paths {
        fmt.Printf("%x %s\n", m[path], path)
    }
}
```

The `MD5All` function is the focus of our discussion. In [serial.go](#), the implementation uses no concurrency and simply reads and sums each file as it walks the tree.

```
// MD5All reads all the files in the file tree rooted at root and returns
// a map
// from file path to the MD5 sum of the file's contents. If the
// directory walk
```

```
// fails or any read operation fails, MD5All returns an error.
func MD5All(root string) (map[string][md5.Size]byte, error) {
    m := make(map[string][md5.Size]byte)
    err := filepath.Walk(root, func(path string, info os.FileInfo, err
error) error {
        if err != nil {
            return err
        }
        if !info.Mode().IsRegular() {
            return nil
        }
        data, err := ioutil.ReadFile(path)
        if err != nil {
            return err
        }
        m[path] = md5.Sum(data)
        return nil
    })
    if err != nil {
        return nil, err
    }
    return m, nil
}
```

Parallel digestion

In [parallel.go](#), we split MD5All into a two-stage pipeline. The first stage, sumFiles, walks the tree, digests each file in a new goroutine, and sends the results on a channel with value type result:

```
type result struct {
    path string
    sum  [md5.Size]byte
    err  error
}
```

sumFiles returns two channels: one for the results and another for the error returned by `filepath.Walk`. The walk function starts a new goroutine to process each regular file, then checks `done`. If `done` is closed, the walk stops immediately:

```
func sumFiles(done <-chan struct{}, root string) (<-chan result, <-chan
error) {
```

```
// For each regular file, start a goroutine that sums the file and
// sends
// the result on c.  Send the result of the walk on errc.
c := make(chan result)
errc := make(chan error, 1)
go func() {
    var wg sync.WaitGroup
    err := filepath.Walk(root, func(path string, info os.FileInfo,
err error) error {
        if err != nil {
            return err
        }
        if !info.Mode().IsRegular() {
            return nil
        }
        wg.Add(1)
        go func() {
            data, err := ioutil.ReadFile(path)
            select {
            case c <- result{path, md5.Sum(data), err}:
            case <-done:
            }
            wg.Done()
        }()
        // Abort the walk if done is closed.
        select {
        case <-done:
            return errors.New("walk canceled")
        default:
            return nil
        }
    })
    // Walk has returned, so all calls to wg.Add are done.  Start a
    // goroutine to close c once all the sends are done.
    go func() {
        wg.Wait()
        close(c)
    }()
    // No select needed here, since errc is buffered.
    errc <- err
}()
return c, errc
}
```

MD5All receives the digest values from c. MD5All returns early on error, closing done via a defer:

```
func MD5All(root string) (map[string][md5.Size]byte, error) {
    // MD5All closes the done channel when it returns; it may do so
    before
    // receiving all the values from c and errc.
    done := make(chan struct{})
    defer close(done)

    c, errc := sumFiles(done, root)

    m := make(map[string][md5.Size]byte)
    for r := range c {
        if r.err != nil {
            return nil, r.err
        }
        m[r.path] = r.sum
    }
    if err := <-errc; err != nil {
        return nil, err
    }
    return m, nil
}
```

Bounded parallelism

The MD5All implementation in [parallel.go](#) starts a new goroutine for each file. In a directory with many large files, this may allocate more memory than is available on the machine.

We can limit these allocations by bounding the number of files read in parallel. In [bounded.go](#), we do this by creating a fixed number of goroutines for reading files. Our pipeline now has three stages: walk the tree, read and digest the files, and collect the digests.

The first stage, walkFiles, emits the paths of regular files in the tree:

```
func walkFiles(done <-chan struct{}, root string) (<-chan string, <-chan
error) {
    paths := make(chan string)
    errc := make(chan error, 1)
```

```

go func() {
    // Close the paths channel after Walk returns.
    defer close(paths)
    // No select needed for this send, since errc is buffered.
    errc <- filepath.Walk(root, func(path string, info os.FileInfo,
err error) error {
        if err != nil {
            return err
        }
        if !info.Mode().IsRegular() {
            return nil
        }
        select {
        case paths <- path:
        case <-done:
            return errors.New("walk canceled")
        }
        return nil
    })
}()
return paths, errc
}

```

The middle stage starts a fixed number of digester goroutines that receive file names from paths and send results on channel c:

```

func digester(done <-chan struct{}, paths <-chan string, c chan<- result)
{
    for path := range paths {
        data, err := ioutil.ReadFile(path)
        select {
        case c <- result{path, md5.Sum(data), err}:
        case <-done:
            return
        }
    }
}

```

Unlike our previous examples, digester does not close its output channel, as multiple goroutines are sending on a shared channel. Instead, code in MD5All arranges for the channel to be closed when all the digesters are done:

```
// Start a fixed number of goroutines to read and digest files.
c := make(chan result)
var wg sync.WaitGroup
const numDigesters = 20
wg.Add(numDigesters)
for i := 0; i < numDigesters; i++ {
    go func() {
        digester(done, paths, c)
        wg.Done()
    }()
}
go func() {
    wg.Wait()
    close(c)
}()


```

We could instead have each digester create and return its own output channel, but then we would need additional goroutines to fan-in the results.

The final stage receives all the results from c then checks the error from errc. This check cannot happen any earlier, since before this point, walkFiles may block sending values downstream:

```
m := make(map[string][md5.Size]byte)
for r := range c {
    if r.err != nil {
        return nil, r.err
    }
    m[r.path] = r.sum
}
// Check whether the Walk failed.
if err := <-errc; err != nil {
    return nil, err
}
return m, nil
}
```

Conclusion

This article has presented techniques for constructing streaming data pipelines in Go. Dealing with failures in such pipelines is tricky, since each stage in the pipeline may block attempting to send values downstream, and the downstream stages may no longer care

about the incoming data. We showed how closing a channel can broadcast a "done" signal to all the goroutines started by a pipeline and defined guidelines for constructing pipelines correctly.

Further reading:

- [Go Concurrency Patterns \(video\)](#) presents the basics of Go's concurrency primitives and several ways to apply them.
- [Advanced Go Concurrency Patterns \(video\)](#) covers more complex uses of Go's primitives, especially select.
- Douglas McIlroy's paper [Squinting at Power Series](#) shows how Go-like concurrency provides elegant support for complex calculations.

By *Sameer Ajmani*

Related articles

- [Go Concurrency Patterns: Context](#)
- [Introducing the Go Race Detector](#)
- [Advanced Go Concurrency Patterns](#)
- [Concurrency is not parallelism](#)
- [Go videos from Google I/O 2012](#)
- [Go Concurrency Patterns: Timing out, moving on](#)
- [Share Memory By Communicating](#)

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License,

and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#) | [View the source code](#)