The complete guide to Go net/http timeouts

Filippo Valsorda

When writing an HTTP server or client in Go, timeouts are amongst the easiest and most subtle things to get wrong: there's many to choose from, and a mistake can have no consequences for a long time, until the network glitches and the process hangs.

HTTP is a complex multi-stage protocol, so there's no one-size fits all solution to timeouts. Think about a streaming endpoint versus a JSON API versus a Comet endpoint. Indeed, the defaults are often not what you want.

In this post I'll take apart the various stages you might need to apply a timeout to, and look at the different ways to do it, on both the Server and the Client side.

SetDeadline

First, you need to know about the network primitive that Go exposes to implement timeouts: Deadlines.

Exposed by `net.Conn` with the `Set[Read|Write]Deadline(time.Time)` methods, Deadlines are an absolute time which when reached makes all I/O operations fail with a timeout error.

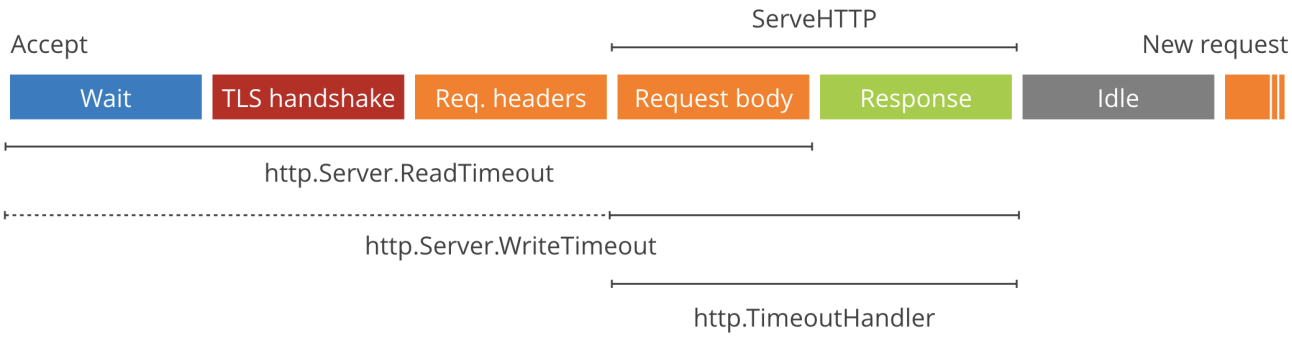
Deadlines are not timeouts. Once set they stay in force forever (or until the next call to `SetDeadline`), no matter if and how the connection is used in the

meantime. So to build a timeout with `SetDeadline` you'll have to call it before *every* `Read`/`Write` operation.

You probably don't want to call `SetDeadline` yourself, and let `net/http` call it for you instead, using its higher level timeouts. However, keep in mind that all timeouts are implemented in terms of Deadlines, so they **do NOT reset every time data is sent or received**.

Server Timeouts

The "So you want to expose Go on the Internet" post has more information on server timeouts, in particular about HTTP/2 and Go 1.7 bugs.



It's critical for an HTTP server exposed to the Internet to enforce timeouts on client connections. Otherwise very slow or disappearing clients might leak file descriptors and eventually result in something along the lines of:

```
http: Accept error: accept tcp [::]:80: accept4: too many open files; retrying in 5ms
```

There are two timeouts exposed in `http.Server`: `ReadTimeout` and `WriteTimeout`. You set them by explicitly using a Server:

```
srv := &http.Server{
    ReadTimeout: 5 * time.Second,
    WriteTimeout: 10 * time.Second,
}
```

```
log.Println(srv.ListenAndServe())
```

`ReadTimeout` covers the time from when the connection is accepted to when the request body is fully read (if you do read the body, otherwise to the end of the headers). It's implemented in `net/http` by calling `SetReadDeadline` immediately after Accept.

`WriteTimeout` normally covers the time from the end of the request header read to the end of the response write (a.k.a. the lifetime of the ServeHTTP), by calling `SetWriteDeadline` at the end of readRequest.

However, when the connection is HTTPS, `SetWriteDeadline` is called immediately after Accept so that it also covers the packets written as part of the TLS handshake. Annoyingly, this means that (in that case only) `WriteTimeout` ends up including the header read and the first byte wait.

You should set both timeouts when you deal with untrusted clients and/or networks, so that a client can't hold up a connection by being slow to write or read.

Finally, there's `http.TimeoutHandler`. It's not a Server parameter, but a Handler wrapper that limits the maximum duration of `ServeHTTP` calls. It works by buffering the response, and sending a *504 Gateway Timeout* instead if the deadline is exceeded. Note that it is broken in 1.6 and fixed in 1.6.2.

http.ListenAndServe is doing it wrong

Incidentally, this means that the package-level convenience functions that bypass `http.Server` like `http.ListenAndServe`, `http.ListenAndServeTLS` and `http.Serve` are unfit for public Internet servers.

Those functions leave the Timeouts to their default off value, with no way of enabling them, so if you use them you'll soon be leaking connections and run

out of file descriptors. I've made this mistake at least half a dozen times.

Instead, create a `http.Server` instance with ReadTimeout and WriteTimeout and use its corresponding methods, like in the example a few paragraphs above.

About streaming

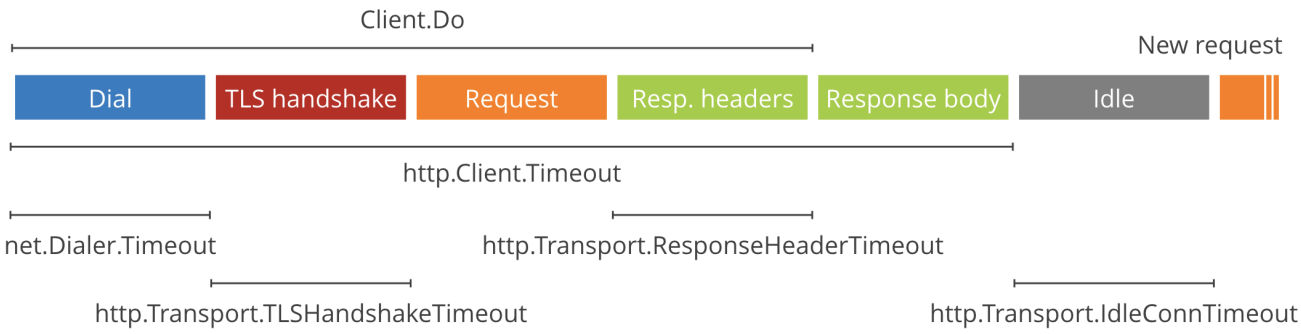
Very annoyingly, there is no way of accessing the underlying `net.Conn` from `ServeHTTP` so a server that intends to stream a response is forced to unset the `WriteTimeout` (which is also possibly why they are 0 by default). This is because without `net.Conn` access, there is no way of calling `SetWriteDeadline` before each `Write` to implement a proper idle (not absolute) timeout.

Also, there's no way to cancel a blocked `ResponseWriter.Write` since `ResponseWriter.Close` (which you can access via an interface upgrade) is not documented to unblock a concurrent Write. So there's no way to build a timeout manually with a Timer, either.

Sadly, this means that streaming servers can't really defend themselves from a slow-reading client.

I submitted an issue with some proposals, and I welcome feedback there.

Client Timeouts



Client-side timeouts can be simpler or much more complex, depending which

ones you use, but are just as important to prevent leaking resources or getting stuck.

The easiest to use is the `Timeout` field of `http.Client`. It covers the entire exchange, from Dial (if a connection is not reused) to reading the body.

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
resp, err := c.Get("https://blog.filippo.io/")
```

Like the server-side case above, the package level functions such as `http.Get` use a Client without timeouts, so are dangerous to use on the open Internet.

For more granular control, there are a number of other more specific timeouts you can set:

- `net.Dialer.Timeout` limits the time spent establishing a TCP connection (if a new one is needed).

- `http.Transport.TLSHandshakeTimeout` limits the time spent performing the TLS handshake.

- `http.Transport.ResponseHeaderTimeout` limits the time spent reading the headers of the response.

- `http.Transport.ExpectContinueTimeout` limits the time the client will wait between sending the request headers *when including an* `Expect: 100-continue` and receiving the go-ahead to send the body. Note that setting this in 1.6 will disable HTTP/2 (`DefaultTransport` is special-cased from 1.6.2).

```
c := &http.Client{
    Transport: &http.Transport{
        Dial: (&net.Dialer{
                Timeout:   30 * time.Second,
                KeepAlive: 30 * time.Second,
        }).Dial,
        TLSHandshakeTimeout:   10 * time.Second,
        ResponseHeaderTimeout: 10 * time.Second,
        ExpectContinueTimeout: 1 * time.Second,
    }
}
```

As far as I can tell, there's no way to limit the time spent sending the request specifically. The time spent reading the request body can be controlled manually with a `time.Timer` since it happens after the Client method returns (see below for how to cancel a request).

Finally, new in 1.7, there's `http.Transport.IdleConnTimeout`. It does not control a blocking phase of a client request, but how long an idle connection is kept in the connection pool.

Note that a Client will follow redirects by default. `http.Client.Timeout` includes all time spent following redirects, while the granular timeouts are specific for each request, since `http.Transport` is a lower level system that has no concept of redirects.

Cancel and Context

`net/http` offers two ways to cancel a client request: `Request.Cancel` and, new in 1.7, Context.

`Request.Cancel` is an optional channel that when set and then closed causes the request to abort as if the `Request.Timeout` had been hit. (They are actually implemented through the same mechanism, and while writing this post I found a bug in 1.7 where all cancellations would be returned as timeout errors.)

We can use `Request.Cancel` and `time.Timer` to build a more granular timeout that allows streaming, pushing the deadline back every time we successfully read some data from the Body:

```
package main

import (
    "io"
    "io/ioutil"
    "log"
    "net/http"
    "time"
)

func main() {
    c := make(chan struct{})
    timer := time.AfterFunc(5*time.Second, func() {
        close(c)
    })

        // Serve 256 bytes every second.
    req, err := http.NewRequest("GET", "http://httpbin.org/range/2048?duration=8&chunk_s
    if err != nil {
        log.Fatal(err)
    }
    req.Cancel = c

    log.Println("Sending request...")
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
```

In the example above, we put a timeout of 5 seconds on the Do phases of the request, but then we spend at least 8 seconds reading the body in 8 rounds, each time with a timeout of 2 seconds. We could go on streaming like this forever without risk of getting stuck. If we were not to receive body data for more than 2 seconds, then io.CopyN would return `net/http: request canceled`.

In 1.7 the `context` package graduated to the standard library. There's a lot to learn about Contexts, but for our purposes you should know that they replace and deprecate `Request.Cancel`.

To use Contexts to cancel a request we just obtain a new Context and its `cancel()` function with `context.WithCancel` and create a Request bound to it with `Request.WithContext`. When we want to cancel the request, we cancel the Context by calling `cancel()` (instead of closing the Cancel channel):

```
ctx, cancel := context.WithCancel(context.TODO())
timer := time.AfterFunc(5*time.Second, func() {
    cancel()
})

req, err := http.NewRequest("GET", "http://httpbin.org/range/2048?duration=8&chunk_size=
if err != nil {
    log.Fatal(err)
}
req = req.WithContext(ctx)
```

Contexts have the advantage that if the parent context (the one we passed to `context.WithCancel`) is canceled, ours will be, too, propagating the command down the entire pipeline.

This is all. I hope I didn't exceed your `ReadDeadline`!

If this kind of deep dive into the Go standard libraries sound entertaining to you, know that we are hiring in London, Austin (TX), Champaign (IL), San Francisco and Singapore.

Reliability API JSON Programming Go

RELATED POSTS

November 29, 2018 6:45PM

Logs from the Edge

With Cloudflare Workers, it is possible to send traffic logs to arbitrary locations. In this post we are going to discuss an example Worker implementation on how to achieve this. So if you are building or maintaining your own traffic logging/analytics environment, read on....

By Michael Tremante

Reliability, Cloudflare Workers, Serverless, Logs

August 06, 2018 5:45PM

Additional Record Types Available with Cloudflare DNS

Cloudflare recently updated the authoritative DNS service to support nine new record types. Since these records are less commonly used than what we previously supported, we thought it would be a good idea to do a brief explanation of each record type and how it is used....

By Sergi Isasi, Etienne Labaume

Reliability, Speed & Reliability, DNS, DNSSEC, Product News

June 01, 2018 2:13AM

Today we mitigated 1.1.1.1

Cloudflare is protected from attacks by the Gatebot DDoS mitigation pipeline. Gatebot performs hundreds of mitigations a day, shielding our infrastructure and our customers from L3 and L7 attacks....

By Marek Majkowski

Reliability, Post Mortem, Mitigation, 1.1.1.1, DNS

January 18, 2018 12:06PM

However improbable: The story of a processor bug

Processor problems have been in the news lately, due to the Meltdown and Spectre vulnerabilities. But generally, engineers writing software assume that computer hardware operates in a reliable, well-understood fashion, and that any problems lie on the software side of the software-hardware divide....

By David Wragg

Reliability, Bugs, Vulnerabilities, NGINX

Dima Shapiro • 3 years ago • edited

I wonder, on a client side, what would be the advantages of using context.WithTImeout() instead of http.Client.Timeout, assuming I don't need a fine-grained control over timeout phases (TLS etc)?

36 • Reply • Share ›

Inanc Gumus • 3 years ago

I created a very simple utility library for this: https://github.com/inancgum...

9 • Reply • Share ›

Rob Fielding • 2 years ago

Is there a way to _lengthen_ the timeout? Example: I begin an http call, and I haven't looked closely enough at it to know what it's going to do. So, generically just arbitrarily assign something like 5sec as a deadline time. But when I figure out exactly what they are asking for, I realize that my return content-length will be 4GB. I would then like to be able to take a supported minimumThroughput to CALCULATE what a reasonable deadline should be.

This drives me nuts every time I read an article about a fixed deadline. This only works if the request and response bodies are of small nearly fixed lengths.

4 • Reply • Share ›

Dominic Mitchell • 4 years ago

The use of contexts can be simplified. To start, you should use context.Background().

```
ctx, cancel := context.WithCancel(context. Background())
timer := time.AfterFunc(5*time.Second, cancel)
```

You should always ensure the context is cancelled:

```
ctx, cancel := context.WithCancel(context. Background())
```