a tale of two stacks: optimizing gin's panic recovery handler

A programming article by Efron Licht

more articles

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

gin's panic handler

The popular go web framework Gin has a middleware that allows you to recover from and log panics while serving HTTP.

Here's an example program that will panic whenever we hit the route GET /panic

```
 1  package main
 2
 3  import (
 4     "fmt"
 5     "net/http"
 6
 7     "github.com/gin-gonic/gin"
 8  )
 9
10  func main() {
11     engine := gin.New()
12     engine.Use(gin.Recovery())
13     engine.GET("/panic", func(c *gin.Context)
    {fmt.Fprintf(c.Writer, "%s", f())})
```

```
14      http.ListenAndServe(":8080", engine)
15  }
16  func f() string {
17      panic("this function panics!")
18  }
```

We run the program:

```
1   go run main.go
```

and in another shell provoke the panic:

```
1   curl http://localhost:8080
```

Getting this nicely formatted result:P

```
 1  2023/01/17 13:45:01 [Recovery] 2023/01/17 - 13:45:01 panic
    recovered:
 2  GET /panic HTTP/1.1
 3  Host: localhost:8080
 4  Accept: */*
 5  User-Agent: curl/7.81.0
 6
 7
 8  this function panics!
 9  /home/efron/go/src/gitlab.com/efronlicht/gin-ex/main.go:19
    (0x7200a6)
10          f: panic("this function panics!")
11  /home/efron/go/src/gitlab.com/efronlicht/gin-ex/main.go:14
    (0x720094)
12          main.func1: fmt.Fprintf(c.Writer, "%s", f())
13  /home/efron/go/pkg/mod/github.com/gin-gonic/gin@v1.8.2/
    context.go:173 (0x71a601)
14          (*Context).Next: c.handlers[c.index](c)
15  /home/efron/go/pkg/mod/github.com/gin-gonic/gin@v1.8.2/
    recovery.go:101 (0x71a5ec)
16          CustomRecoveryWithWriter.func1: c.Next()
17  /home/efron/go/pkg/mod/github.com/gin-gonic/gin@v1.8.2/
    context.go:173 (0x719470)
18          (*Context).Next: c.handlers[c.index](c)
19  /home/efron/go/pkg/mod/github.com/gin-gonic/gin@v1.8.2/
    gin.go:616 (0x7190d8)
20          (*Engine).handleHTTPRequest: c.Next()
21  /home/efron/go/pkg/mod/github.com/gin-gonic/gin@v1.8.2/
    gin.go:572 (0x718d9c)
22          (*Engine).ServeHTTP: engine.handleHTTPRequest(c)
23  /usr/local/go/src/net/http/server.go:2947 (0x620a6b)
24          serverHandler.ServeHTTP: handler.ServeHTTP(rw, req)
25  /usr/local/go/src/net/http/server.go:1991 (0x61d366)
26          (*conn).serve:
    serverHandler{c.server}.ServeHTTP(w, w.req)
27  /usr/local/go/src/runtime/asm_amd64.s:1594 (0x4672e0)
28          goexit: BYTE    $0x90   // NOP
```

As well as the usual elements of a stack trace: file, line, program counter, function name; Gin's recovery handler somehow gives you the **actual line of the source code** where that frame of the stack trace happened.

For programmers more used to dynamic languages, this seems pretty natural: you need the source code to run the file. But Go is a statically compiled language that doesn't carry around it's source code.

I had two thoughts when I saw this:

- "wow, cool!"
- "this is probably incredibly slow".

Let's take a look into the implementation to find out *how* Gin does it, and see if we can do it better. When we're done, let's run some benchmarks to compare the solutions, and then talk about if this source-code magic is a good idea in the first place.

Investigating Gin

Luckily, the stack trace itself gives us nearly all the clues we need:

> `/home/efron/go/pkg/mod/github.com/gin-gonic/gin@v1.8.2/recovery.go:101 (0x71a5ec)

```
1       CustomRecoveryWithWriter.func1: c.Next()`
```

which contains a function `stack()`, which is simple and clearly documented. From now on, I'm going to call it `slowstack()`, and we'll call our eventual optimized version `faststack()`.
Starting at `skip`, we walk frame-by-frame up the stack.

for each frame, we use `runtime.Caller` to get the stack information for each one, using the `file` name to look up up a file with the same name on the host system.

We read the entire file into memory, caching the most-recent file, so multiple calls to the same function don't have to re-read the file, and print out the annotated stack frame.

```
1   // stack returns a nicely formatted stack frame, skipping
    skip frames.
2   func stack(skip int) []byte { // from now on, called
    `slowstack()`
3    // As we loop, we open files and read them. These
    variables record the currently
4    // loaded file.
5    var lines [][]byte
6    var lastFile string
7    for i := skip; ; i++ { // Skip the expected number of
    frames
```

```
 8     pc, file, line, ok := runtime.Caller(i)
 9     if !ok {
10      break
11     }
12     // Print this much at least.  If we can't find the
       source, it won't show.
13     fmt.Fprintf(buf, "%s:%d (0x%x)\n", file, line, pc)
14     if file != lastFile {
15      data, err := os.ReadFile(file)
16      if err != nil {
17       continue
18      }
19      lines = bytes.Split(data, []byte{'\n'})
20      lastFile = file
21     }
22     fmt.Fprintf(buf, "\t%s: %s\n", function(pc),
       source(lines, line))
23    }
24    return buf.Bytes()
25   }
26
27   func source(lines [][]byte, n int) {
28    n-- // in stack trace, lines are 1-indexed but our array
       is 0-indexed
29    if n < 0 || n >= len(lines) {
30     return dunno
31    }
32    return bytes.TrimSpace(lines[n])
33   }
34   // function returns, if possible, the name of the function
       containing the PC.
35   func function(pc uintptr) []byte{ // body omitted
36   }
37
```

What are the performance limitations of `slowstack`?

- unnecessary work in the `runtime` package
- always reads a whole file instead of a single line
- can re-read the same file multiple times
- opens many file handles
- unbounded allocations
- always attempts to read files even if it's impossible

unnecessary work in the `runtime` package

`runtime.Caller()` is a specalized invocation of runtime.CallersFrames for a single function: it ascends the callstack to `skip` using runtime magic, builds a `runtime.Frame` struct, and gives you a couple of that struct's fields:

Let's look at the source:

```
 1  type Frame struct {
 2   // PC is the program counter for the location in this
    frame.
 3   // For a frame that calls another frame, this will be the
 4   // program counter of a call instruction. Because of
    inlining,
 5   // multiple frames may have the same PC value, but
    different
 6   // symbolic information.
 7   PC uintptr
 8
 9   // Func is the Func value of this call frame. This may be
    nil
10   // for non-Go code or fully inlined functions.
11   Func *Func
12
13   // Function is the package path-qualified function name of
14   // this call frame. If non-empty, this string uniquely
15   // identifies a single function in the program.
16   // This may be the empty string if not known.
17   // If Func is not nil then Function == Func.Name().
18   Function string
19
20   // File and Line are the file name and line number of the
21   // location in this frame. For non-leaf frames, this will
    be
22   // the location of a call. These may be the empty string
    and
23   // zero, respectively, if not known.
24   File string
25   Line int
26
27   // Entry point program counter for the function; may be
    zero
28   // if not known. If Func is not nil then Entry ==
29   // Func.Entry().
30   Entry uintptr
31   // contains filtered or unexported fields
32  }
33  // package runtime
34  func Caller(skip int) (pc uintptr, file string, line int,
    ok bool) {
35   rpc := make([]uintptr, 1)
36   n := callers(skip+1, rpc[:])
37   if n < 1 {
38    return
39   }
40   frame, _ := CallersFrames(rpc).Next() // a runtime.Frame
41   return frame.PC, frame.File, frame.Line, frame.PC != 0
42  }
```

`slowstack` then calls `function(pc)` on the returned program counter to lookup the `*runtime.Func`, then get it's `Name`.

We could could simply grab the `(runtime.Frame).Function` field directly. Better yet, we can call `runtime.Callers()` and `runtime.CallersFrames()` exactly once and ascend the callstack by repeatedly calling `frames.Next()`, which avoids having to repeatedly jump up and down the stack.

```
 1  // stack returns a nicely formatted stack frame, skipping
    skip frames.
 2  func stack_01(skip int) []byte {
 3      // grab the stack frame
 4      pc := make([]uintptr, 64)
 5      n := runtime.Callers(skip, pc)
 6      if n == 0 { // no callers: e.g, skip > len(callstack).
 7          return nil
 8      }
 9      pc = pc[:n] // pass only valid pcs to runtime.Caller
10      buf := new(bytes.Buffer)
11      frames := runtime.CallersFrames(pc)
12      for {
13          frame, more := frames.Next()
14          fmt.Fprintf(buf, "%s:%d (0x%x)\n", frame.File,
    frame.Line, frame.PC)
15      if file != lastFile {
16          data, err := os.ReadFile(file)
17          if err != nil {
18              continue
19          }
20          lines = bytes.Split(data, []byte{'\n'})
21          lastFile = file
22           fmt.Fprintf(buf, "\t%s: %s\n", frame.Function,
    source(lines, line))
23      }
24      if !more {
25          return buf.Bytes()
26      }
27      }
28      panic("unreachable!)
29  }
```

This approach has one limitation: we have to specify a max stack size to pass to `runtime.Callers`. I think this is a *good thing*, since it places bounds on the resources this function can use, but it *is* a limitation. You could mitigate this by making the max depth a parameter of the function and configuring it at runtime, e.g, via an environment variable, but 64 seems like a good number to me.

always reads a whole file instead of a single line

Even with an incredibly simple handler, like ours, `slowstack` reads 7 files into memory, totalling 276KiB.

- main.go (.3 KiB)
- gin/context.go (37 KiB)

- gin/recovery.go (6 Kib)
- gin/context.go again (37 KiB)
- gin/gin.go (23 KiB)
- net/http/server.go (114 KiB)
- asm_amd64.s (59KiB)

The longest lines of a `.go` or `.s` file are roughly 200 bytes, so this is using roughly 1300x the bytes it needs to.

reading a line at a time

We can read one line at a time instead by using `bufio.Scanner`. We'll need a new scanner per file, but we can share a buffer between them.
This has two benefits: first, we allocate less memory. Secondly, we can *stop* reading a file once we've hit the appropriate line without having to read to the end.

```
 1  func stack_02(skip int) []byte {
 2      // grab the stack frame
 3      pc := make([]uintptr, 64)
 4      n := runtime.Callers(skip, pc)
 5      if n == 0 { // no callers: e.g, skip > len(callstack).
 6          return nil
 7      }
 8      pc = pc[:n] // pass only valid pcs to runtime.Caller
 9      buf := new(bytes.Buffer)
10      frames := runtime.CallersFrames(pc)
11      scanBuf := make([]byte, 0, 256)
12      FRAME:
13      for {
14          frame, more := frames.Next()
15          fmt.Fprintf(buf, "%s:%d (0x%x)\n", file, line, pc)
16          f, err := os.Open(file)
17          if err != nil {
18              continue
19          }
20          f, err := os.Open(frame.File)
21          if err != nil {
22              continue FRAME
23          }
24          scanner := bufio.NewScanner(f)
25          scanner.Buffer(scanBuf[:0], bufio.MaxScanTokenSize)
26          var source []byte
27          for i := 0; scanner.Scan() && i < Line; i++ {
28              if i == line {
29                  fmt.Fprintf(buf, "\t%s: %s\n",
    function(pc),  bytes.TrimSpace(scanner.Bytes())
30                  f.Close()
31                  continue FRAME
32              }
33          }
34          // hit EOF early:
```

```
35              fmt.Fprintf(buf, "\t%s: %s\n", function(pc),
    "???")
36              f.Close()
37          }
38          return buf.Bytes()
39      }
```

Two caveats: while `stack_02` uses less memory *per file* than `stack_01` or `slowstack` we still have to open the same file each time it appears on the callstack. `slowstack` had an optimization for a common case: if the same file appeared twice in a row on the callstack, it would re-use the memory it had previously read.

Second, `stack_02` has a hidden if unlikely bug: a `bufio.Scanner` can panic if it hits too many empty tokens in a row (in this case, dozens of newlines). In our next iteration, we'll wrap this in a `recover()` to protect ourselves from weird sourcefiles: we don't want to panic during a panic recovery handler! We'll handle this in our final solution.

reading each file *exactly once*

While our solution uses less memory *per file*, we still have to open the same file each time it appears on the callstack. Pretty bad if we hit a recursive function:

```
1
2   func FibStack(n int) int {
3       if n < 0 {
4           panic("expected n >= 0")
5       }
6       if n == 0 || n == 1 {
7           fmt.Println(string(stack(0)))
8           return 1
9       } else {
10          return FibStack(n-1) + FibStack(n-2)
11      }
12      default:
13          return
14      }
```

But `slowstack`'s solution was no prize, either: callstacks that rapidly bounce between the same files with at least one different file in-between get no savings from that approach.

We can build our own pathological example (we'll use this for benchmarking later)

```
1   // ping.go
2   func PingStack(n int) []byte {
3       if n < 0 {
4           panic("expected n >= 0")
5       }
6       if n == 0 {
7           return stack(0)
```

```
 8       }
 9       return Pong(n-1)
10   }
11   // in pong.go
12   func Pong(n int) []byte {return Ping(n)}
```

We'd like to open each file exactly once, which means grouping our stack frames by filename. There's a few approaches you could use, but I prefer this one:
If we sort the frames by file-and-line, we can populate each frame with the source in exactly one pass thorough each file. We can then format the annotated frames.

This will require a new datastructure:

```
1   type debugInfo struct {
2           *runtime.Frame // hold a pointer rather than
    allocate a new one
3           Source
    string // the line of source code holding the frame
4           Depth int // the frame's original depth in the stack
5   }
```

This requires a few structural changes: rather than formatting the frames as we ascend the callstack, we put them all in a slice, populate them all, resort-them with their original order, and *then* format them into our output buffer.

If there are few or no repeated files, this will use more peak memory than the previous approach, but it's a solid solution that saves us from the pathological cases we outlined above. And since our minimal test case proves *any* Gin handler will have some repeated cases, we always save *some* work.

```
 1   func stack_03(skip int) (formatted []byte) {
 2       // grab the stack frame
 3       pc := make([]uintptr, 64)
 4       n := runtime.Callers(skip, pc)
 5       if n == 0 { // no callers: e.g, skip > len(callstack).
 6           return nil
 7       }
 8       pc = pc[:n] // pass only valid pcs to runtime.Caller
 9       frames := runtime.CallersFrames(pc)
10       // allocate a 4KiB reusable buffer, exactly once. we
     will use this both to read the input and format the output.
11       buf := make([]byte, 0, 4096)
12       type debugInfo struct {
13           *runtime.Frame
14           Source string
15           Depth int
16       }
17       // populate the debuginfo with the lines of code that
     appear in the stack  trace.
18       di := make([]debugInfo, 0, n)
19       for depth := 0; ; depth++ {
20           frame, more := frames.Next()
```

```
21          di = append(di, debugInfo{Frame: &frame, Depth:
    depth})
22          if !more {
23              break
24          }
25      }
26
27      // group the debuginfo by file and line: we'll resort
    them by depth later.
28      sort.Slice(di, func(i, j int) bool {
29          return di[i].File < di[j].File || (di[i].File ==
    di[j].File && di[i].Line < di[j].Line)
30      })
31      // populate debug info with source.
32      func() {
33          line := 0
34          lastFile := di[0].File
35          f, err := os.Open(lastFile)
36          defer func() {
37              // scanner.Scan() can panic if it hits too many
    newlines.
38              // in that case, we immediately abandon trying
    to get more sourceInfo and just write as much as we can.
39              if p := recover(); p != nil {
40                  log.Println("panic while formatting stack:
    too many empty lines in sourcefile?", p, debug.Stack())
41                  f.Close()
42              }
43
44          }()
45
46          scanner := bufio.NewScanner(f)
47          scanner.Buffer(buf[:0], bufio.MaxScanTokenSize)
48          for i := range di {
49              if di[i].File != lastFile {
50                  if err == nil {
51                      f.Close()
52                  }
53                  // reset: we're at the beginning of a new
    file
54                  line = 0
55                  f, err = os.Open(di[i].File)
56                  if err != nil {
57                      continue
58                  }
59                  scanner = bufio.NewScanner(f)
60                  scanner.Buffer(buf[:0],
    bufio.MaxScanTokenSize)
61              }
62              if err != nil {
63                  continue
64              }
65              // it's possible that we have multiple calls to
    the same function in the stack, such as during recursion,
```

```
66                  // so we check that we haven't gone past BEFORE
    we advance the scanner.
67                  for ; line < di[i].Line; line++ {
68                      scanner.Scan()
69                  }
70                  di[i].Source = scanner.Text()
71
72                  i++
73              }
74              _ = f.Close()
75          }()
76
77          // put the debuginfo back in depth-first order
78          sort.Slice(di, func(i, j int) bool { return
    di[i].Depth < di[j].Depth })
79          // format it all into the buffer. we're safe to reuse
    buf, since we're done with all the scanners.
80          out := bytes.NewBuffer(buf[:0])
81          for i := range di {
82              fmt.Fprintf(out, "%s:%d (0x%x)\t%s:%s\n",
    di[i].File, di[i].Line, di[i].PC,
    trimFunction(di[i].Function),
    strings.TrimSpace(di[i].Source))
83
84          }
85
86          return out.Bytes()
87      }
```

Pretty good! We're nearly done. When there are files to read, we read them efficiently.

what if we can't read the files?

There's plenty of situations where we can't read the files at all. Off the top of my head:

- The executable was built with `go build -trimpath`.
- It's running in a binary without access to the source, like a minimal container.
- The running user doesn't have permission to access the source.

In this case, our implementation does a ton of work that's completely unnecessary, juggling around a ton of data just before it's inevitable failure.

If we had some kind of heuristic for "can read source files", we'd know whether or not it was worth the attempt. We could then build an fast-path for that case.

Here, we use a `sync.Once` to lazily-check whether or not we can access the file where we defined `faststack()`:

```
1   // local source is not always available. for example, the
    executable may be running on a system without the source
```

```
 2    // or the -trimpath buildflag could have been provided to
    go tool.
 3    // if we can't find the source for THIS file, we are
    unlikely to be able to find it for any file.
 4    var localSourceOnce sync.Once
 5    var localSourceNotFound bool
 6    func localSourceUnavailable() bool {
 7            localSourceOnce.Do(func() {
 8                _, file, _, _ := runtime.Caller(3)
 9                if _, err := os.Lstat(file); err != nil {
10                    localSourceNotFound = true
11                }
12
13            })
14            return localSourceNotFound
15        }
```

and then we can put it all together:

finished faststack

```
 1    // stack returns a formatted stack frame, skipping debug
    frames
 2    func faststack(skip int) (formatted []byte) {
 3        // grab the stack frame
 4        pc := make([]uintptr, 64)
 5        n := runtime.Callers(skip, pc)
 6        if n == 0 { // no callers: e.g, skip > len(callstack).
 7            return nil
 8        }
 9        pc = pc[:n] // pass only valid pcs to runtime.Caller
10        frames := runtime.CallersFrames(pc)
11        // allocate a 4KiB reusable buffer, exactly once. we
    will use this both to read the input and format the output.
12        buf := make([]byte, 0, 4096)
13        if localSourceUnavailable() {
14            // fast path: just format the frames in the order
    they occur without looking up the source.
15            out := bytes.NewBuffer(buf)
16            for {
17                frame, more := frames.Next()
18                fmt.Fprintf(out, "%s:%d (0x%x)\t%s:%s\n",
    frame.File, frame.Line, frame.PC,
    trimFunction(frame.Function), "")
19                if !more {
20                    return out.Bytes()
21                }
22            }
23        }
24        // slow path: at least some local source is available,
    so we want to populate the debuginfo with the lines of code
    that appear in the stack  trace.
```

```
25      di := make([]debugInfo, 0, n)
26      for depth := 0; ; depth++ {
27          frame, more := frames.Next()
28          di = append(di, struct {
29              *runtime.Frame
30              Source string // line of code where the call
    appeared
31              Depth  int
32          }{Frame: &frame, Depth: depth})
33          if !more {
34              break
35          }
36      }
37
38      // group the debuginfo by file and line: we'll resort
    them by depth later.
39      sort.Slice(di, func(i, j int) bool {
40          return di[i].File < di[j].File || (di[i].File ==
    di[j].File && di[i].Line < di[j].Line)
41      })
42      // populate debug info with source.
43      func() {
44          line := 0
45          lastFile := di[0].File
46          f, err := os.Open(lastFile)
47          defer func() {
48              // scanner.Scan() can panic if it hits too many
    newlines.
49              // in that case, we immediately abandon trying
    to get more sourceInfo and just write as much as we can.
50              if p := recover(); p != nil {
51                  log.Println("panic while formatting stack:
    too many empty lines in sourcefile?", p, debug.Stack())
52                  f.Close()
53              }
54
55          }()
56
57          scanner := bufio.NewScanner(f)
58          scanner.Buffer(buf[:0], bufio.MaxScanTokenSize)
59          for i := range di {
60              if di[i].File != lastFile {
61                  if err == nil {
62                      f.Close()
63                  }
64                  // reset: we're at the beginning of a new
    file
65                  line = 0
66                  f, err = os.Open(di[i].File)
67                  if err != nil {
68                      continue
69                  }
70                  scanner = bufio.NewScanner(f)
```

```
71                  scanner.Buffer(buf[:0],
    bufio.MaxScanTokenSize)
72              }
73              if err != nil {
74                  continue
75              }
76              // it's possible that we have multiple calls to
    the same function in the stack, such as during recursion,
77              // so we check that we haven't gone past BEFORE
    we advance the scanner.
78              for ; line < di[i].Line; line++ {
79                  scanner.Scan()
80              }
81              di[i].Source = scanner.Text()
82
83              i++
84          }
85          _ = f.Close()
86      }()
87
88      // put the debuginfo back in depth-first order
89      sort.Slice(di, func(i, j int) bool { return
    di[i].Depth < di[j].Depth })
90      // format it all into the buffer. we're safe to reuse
    buf, since we're done with all the scanners.
91      out := bytes.NewBuffer(buf[:0])
92      for i := range di {
93          fmt.Fprintf(out, "%s:%d (0x%x)\t%s:%s\n",
    di[i].File, di[i].Line, di[i].PC,
    trimFunction(di[i].Function),
    strings.TrimSpace(di[i].Source))
94
95      }
96
97      return out.Bytes()
98  }
```

benchmarks

This behavior is hard to benchmark, since how it performs depends wildly on the structure of the callstack and actual files it's reading, which could be wildly different. Ideally, we'd collect a wide sample set of real Go code, insert calls to `faststack` and `slowstack`, and run it on dozens or hundreds of combinations of hardware, operating system, etc, etc.

I'm way too lazy for that and this article's already taken eight hours longer than I thought it would, so instead we'll make a couple simple synthetic benchmarks at a variety of stack depths and try and draw some conclusions.

We'll run the benchmarks on the same computer under four different circumstances:

linux (wsl), source code on SSD
linux (wsl), source code on SSD, compiled with `-trimpath`

Please see the source code in bench_test.go, `[ping_test.go]`(ping_test.go), and `[pong_test.go]`(pong_test.go) for additional details.

bench: caveats

These are *synthetic benchmarks* and probably don't reflect real-world performance.

- These benchmarks are somewhat unfair to `slowstack`, since they're constructed to attack a known worst-case for it's performance.
- It's always hard to measure filesystem performance, since modern filesystems cache recent reads in memory.
- We artificially cap `faststack` at a depth of 64.

Let's examine the first case:

(These benchmarks are formatted using `[fmtbench]`, a tool I created for my simple byte hacking article)

bench: linux(wsl): SSD

name	runs	ns/op	%/max	bytes	%/max	allocs	%/max
FastStack/depth=16-24	7.76e+03	1.42e+05	0.0624	1.66e+04	0.0194	242	0.42
FastStack/depth=32-24	4.93e+03	2.18e+05	0.0959	3.3e+04	0.0385	403	0.7
FastStack/depth=64-24	4.21e+03	3.04e+05	0.133	4.59e+04	0.0536	651	1.13
FastStack/depth=128-24	3.85e+03	2.98e+05	0.131	4.59e+04	0.0536	651	1.13
FastStack/depth=256-24	4.3e+03	2.77e+05	0.122	4.59e+04	0.0536	651	1.13
FastStack/depth=512-24	4.31e+03	2.75e+05	0.121	4.59e+04	0.0536	651	1.13
FastStack/depth=1024-24	4.19e+03	3.06e+05	0.134	4.59e+04	0.0536	651	1.13
FastStack/depth=2048-24	3.45e+03	2.92e+05	0.128	4.59e+04	0.0536	651	1.13
FastStack/depth=4096-24	4.03e+03	2.98e+05	0.131	4.59e+04	0.0536	651	1.13
SlowStack/depth=0016-24	4.95e+03	2.42e+05	0.106	5.32e+05	0.62	313	0.543
SlowStack/depth=0032-24	2.74e+03	3.99e+05	0.175	8.66e+05	1.01	538	0.934
SlowStack/depth=0064-24	1.72e+03	7.07e+05	0.311	1.53e+06	1.79	988	1.71
SlowStack/depth=0128-24	814	1.44e+06	0.631	2.87e+06	3.35	1.89e+03	3.28
	384	3.29e+06	1.44	5.54e+06	6.46	3.68e+03	6.4

name	runs	ns/op	%/ max	bytes	%/ max	allocs	%/ max
SlowStack/ depth=0256-24							
SlowStack/ depth=0512-24	152	7.44e+06	3.27	1.09e+07	12.7	7.28e+03	12.6
SlowStack/ depth=1024-24	52	2.07e+07	9.1	2.16e+07	25.2	1.45e+04	25.1
SlowStack/ depth=2048-24	18	6.6e+07	29	4.29e+07	50.1	2.88e+04	50.1
SlowStack/ depth=4096-24	5	2.28e+08	100	8.57e+07	100	5.76e+04	100

We note the following:

- `faststack` is always faster than `slowstack`; it starts out roughly 40% faster and gets better from there.
- `faststack` always uses *significantly* less memory.
- faststack's resource usage stops increasing after depth 64, as we'd expect (since we hard-limited the callstack).
- slowstack's memory usage and runtime increase linearly with the depth of the callstack, using nearly a MiB per call for even a 32-deep callstack.

bench: Windows (HDD)

name	runs	ns/op	%/max	bytes	%/max	allocs	%/max
FastStack/depth=16-24	861	1.34e+06	0.29	3e+04	0.0346	250	0.434
FastStack/depth=32-24	532	2.25e+06	0.487	4.66e+04	0.0538	410	0.711
FastStack/depth=64-24	322	3.63e+06	0.784	8e+04	0.0922	653	1.13
FastStack/depth=128-24	324	3.61e+06	0.78	8e+04	0.0923	653	1.13
FastStack/depth=256-24	328	3.69e+06	0.798	8e+04	0.0923	653	1.13
FastStack/depth=512-24	326	3.58e+06	0.774	8e+04	0.0923	653	1.13
FastStack/depth=1024-24	338	3.61e+06	0.78	8e+04	0.0923	653	1.13
FastStack/depth=2048-24	322	3.64e+06	0.786	8e+04	0.0923	653	1.13
FastStack/depth=4096-24	327	3.67e+06	0.794	8e+04	0.0923	653	1.13
SlowStack/depth=0016-24	752	1.6e+06	0.346	5.38e+05	0.621	317	0.55
SlowStack/depth=0032-24	427	2.84e+06	0.613	8.76e+05	1.01	542	0.94
SlowStack/depth=0064-24	232	5.11e+06	1.11	1.55e+06	1.79	992	1.72
SlowStack/depth=0128-24	120	1.02e+07	2.22	2.9e+06	3.35	1.89e+03	3.28
SlowStack/depth=0256-24	57	2.02e+07	4.37	5.61e+06	6.47	3.69e+03	6.4
SlowStack/depth=0512-24	31	4e+07	8.66	1.1e+07	12.7	7.28e+03	12.6
SlowStack/depth=1024-24	13	8.56e+07	18.5	2.18e+07	25.2	1.45e+04	25.1

name	runs	ns/op	%/max	bytes	%/max	allocs	%/max
SlowStack/depth=2048-24	6	1.93e+08	41.8	4.35e+07	50.1	2.89e+04	50.1
SlowStack/depth=4096-24	3	4.62e+08	100	8.68e+07	100	5.76e+04	100

This is slower, but not by as much as we might expect. Probably the operating system's in-memory read cache is doing the heavy lifting here.

Only the truly deep calls start taking nearly a second (though that's *really* slow for a http response!)

Is it worth optimizing a panic recovery handler?

Everything comes at a cost. In this case, `faststack` cuts down on the memory usage and clock time significantly by using a more efficient algorithm and limiting the total stack depth, at the cost of doubling the length of the code and increasing it's complexity. `slowstack` was trivial to understand, and `faststack` is definitely not; it requires a new data structure, global variables, lazy initialization, it's *own* panic recovery handler, and two separate sorts.

Why do we have a panic recovery handler in the first place? To provide continuous service, even in the presence of software bugs. That is, a panic recovery handler is supposed to be a last-ditch protection against a bug that never should have made it into production.
These panics are *supposed* to be rare; prevented by a suite of tests, CI, etc. In practice, well, all software is buggy, and Go programs can panic a lot. When I worked at an ISP, I had production bugs that triggered 20K recovery handlers, per minute, per server (on ~5 servers). If those stacks have 20 frames each, on average, and the files those frames came from average 20K, *8GiB* of allocations a minute per server, or 40GiB of allocations per minute: - that's enough to bring most containers to a crawl, and even a relatively beefy modern PC might struggle to clean up all that garbage. **This** server only has 256MiB of RAM (as of my last edit.) Admittedly, that's a pretty niche scenario.

More worryingly, there's two performance problems that *faststack* can't help with: first, most storage mediums can't do concurrent IO. If *something else* on the server needs to touch the disks the panic handler is reading, then they'll block each other for as long as those reads take.

Secondly, most operating systems place a limit on the number of file handles a single process can open (`1024` by default on linux, iirc). Concurrent calls to `faststack()` might quickly eat up this limit. While these conditions are rare, they're likely to occur for at least some of Gin's users, who may struggle to diagnose the problem.

There are technical solutions for *those* problems: for example, you could cut down on IO by having a concurrency-safe cache for common file:line couples, and you could limit the total memory usage of *that* by having it be a LRU cache, but that's adding *even more complexity*.

Which leads us to an important question: what benefit are we getting?
Is having a single line of source code annotation really helpful?

closing thoughts

How much benefit are we really getting from this implementation? How likely is a single line of source code to help us to fix a bug, if we already have the **file**, **line**, and **function name**? Generally speaking, we'll need more context than a single line to diagnose the bug: we'll have to look at the source itself, not just a pinhole window into it. While it's a cool trick, this source code annotation is just that: a neat parlor trick, not a particularly helpful debugging tool. And there's one corner case where it could be really damaging: if the source code present on the host system differs from the version used to compile the executable, the source code annotations will be *wrong*.

At some point, it's important to take a step back and ask yourself *whether or not all this work is worth it in the first place*. I don't think `slowstack` or `faststack` solve a problem that needs to be solved. I think think the Gin project would be better served by just using `runtime.Stack()` and forgetting about source code annotation. Sometimes simpler is better.

We'll talk about panics, logging, and recovery more in a later article: stay tuned.

MORE ARTICLES