start fast: booting go programs quickly with `inittrace` and `nonblocking[T]`

A software article by Efron Amber Licht
June 2023

The only thing a program does every time is boot, so it should boot as fast as possible. This used to be a given, but modern programs are often extraordinarily slow to start. It's not unusual for simple programs to take 10s of seconds to start: it's expensive, frustrating, and totally preventable. In this article, I'll show you how to make your programs start in milliseconds, not seconds.

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

- start fast: booting go programs quickly with `inittrace` and `nonblocking[T]`
  - 1. Table-setting
  - 2. Init Time
  - 3. Measuring init time
    - 3.1. Measuring ready time
  - 4. Assets
    - 4.1. Load less
    - 4.2. Prefer disk to network, and embedding to both
    - 4.3. Load assets of the same kind in parallel
    - 4.4. Store assets in the form you can use them most immediately
      - 4.4.1. use code as a data-description language
      - 4.4.2. pre-encode or render assets to their use format
      - 4.4.3. avoid extra encoding/decoding steps
    - 4.5. Lazy evaluation
    - 4.6. Nonblocking eager initialization
    - 4.7. Generics to the rescue: `nonblocking[T]`

1. Table-setting

A few notes before we begin: we're going to optimize for ordinary, **wall-clock time**. For our purpose, **Boot Time** is the time from the start of the program to the program being ready to accept user input. For a web server, that's time from `./server` to `tcp: listening on :8080`. For a game, that's time from double-clicking the `.exe` to `PRESS ANY KEY TO CONTINUE`. We'll divide **Boot Time** into two halves. **Init Time** is the time it takes to get to the first line of `main()`. **Ready Time** is the time from the first line of `main()` to READY.

> **Boot Time** = **Init Time** + **Ready Time**

There's only four ways to reduce the time it takes to do something:

description	**formal name**	**example**
do less work	algorithmic optimization	`btree` vs `map`
do the same work faster	micro-optimization	loading from cache vs loading from disk
do work at the same time	parallelization	decoding mp3s in parallel
do it in an order that involves less waiting	scheduling/pipeline optimization	dial postgres DB in a separate goroutine

The first two are covered in many, many other articles on software performance, including here and here on this very blog. We'll mostly focus on the last two.

2. Init Time

Go programs form a tree of packages, with `main()` at the root. Starting from the leaves (that is, packages without any imports), each package is initialized by running `init()`. `init()` is a special function that is called by the runtime, *even if you don't explicitly define it*. It has two phases:

- evaluating all the package's globals (`var` statements). (I'll call this **implicit** `init`)
- running the developer-provided `init()` function, if it exists. I'll call this **explicit** `init`

```
1  package somepackage
2
3  var a = rand.Int() // implicit initialization happens first
4  var b int
5
```

```
6   func init() {b = rand.Int()} // explicit initialization
    happens second
```

All of these `package.init()` functions share a single goroutine, so they run sequentially.

See the [go spec on package initialization] for a more precise explanation.

3. Measuring init time

The go runtime provides a way to trace initialization with the environment variable `GODEBUG=inittrace=1`. [spec]
Let's try tracing the initialization of `eblog`, the server you're reading this on.

IN

```
1   GODEBUG=inittrace=1 go run .
```

OUT

```
1   init internal/bytealg @0.003 ms, 0 ms clock, 0 bytes, 0
    allocs
2   init runtime @0.020 ms, 0.020 ms clock, 0 bytes, 0 allocs
3   init errors @0.19 ms, 0 ms clock, 0 bytes, 0 allocs
4   init sync @0.21 ms, 0.001 ms clock, 16 bytes, 1 allocs
5   # many more lines
6   init gitlab.com/efronlicht/blog/server/static @1.5 ms,
    0.017 ms clock, 17400 bytes, 187 allocs
7   init gitlab.com/efronlicht/enve @1.6 ms, 0 ms clock, 48
    bytes, 1 allocs
8   init main @1.6 ms, 0.32 ms clock, 31688 bytes, 353 allocs
```

> "main.init() finished 1.6ms after the start of the program, and took 0.32ms. It required 31688 bytes of memory, spread across 353 calls to the allocator.".

3.1. Measuring ready time

Tracing ready time is even simpler: just use `time.Now()` on the first line of `main()` and right before you start the meat of the program.

```
1   func main() {
2       var startTime = time.Now()
3       // other initialization
4       // ...
5       log.Printf("main() ready in %s", time.Since(startTime))
6       go server.ListenAndServe()
7       log.Printf("app listening on %s", server.Addr)
8   }
```

Using the blog again, we find:

> main() ready in 611.478µs
>
> app listening on :6483`

My blog starts in **<3ms**! Can't say I don't practice what I preach.

This suggests 3 approaches to reducing init time:

- eliminate packages
- optimize `init()` functions, slowest to fastest
- (if necessary) parallelize `init()` functions

4. Assets

Assets (images, text files, configuration files, etc) are the most common culprit for slow initialization. As programs mature, they tend to accumulate more and more assets. Shorten load times by:

4.1. Load less

Don't add a zillion assets that don't do anything. Use one font instead of dozens. Use one image, or none. Most importantly, **don't load assets you don't need**. It is wild how many programs load literal hundreds of MiB of assets they never use. Please do not do this.

4.2. Prefer disk to network, and embedding to both

Loading from disk is (usually) faster & more reliable than loading from the network. Loading from the binary is ALWAYS faster than opening a file to read it. Syscalls are slow. Disks are slow. Networks are slow. Filesystems (can) be slow: NTFS in particular struggles with small files. (this is mostly a library problem, apparently).

Avoid both by getting necessary assets at *compile time* and embedding them directly in your binary.

That is, instead of this:

```
var data []byte
func init() {
    var err error
    data, err = io.ReadFile("assets/data.json")
    if err != nil {
        log.Fatal(err)
    }
    // do something with data
```

```
}
```

Or even worse, this:

```
var data []byte
func init() {
    var buf bytes.Buffer
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    s3Downloader.Get("assets/data.json", &data, &s3.GetObjectInput{
        Bucket: aws.String("assets")
        Key:    aws.String("data.json")
    })
    data = buf.Bytes()
}
```

Just do this:

```
import _ "embed" // must be imported to use `//go:embed`

//go:embed assets/data.json
var data []byte
```

Simple & fast.

Don't be the guy who needs to call seven different cloud APIs to serve a web page.

4.3. Load assets of the same kind in parallel

Instead of initializing similar assets ad-hoc, use classic fork-join parallelism to do them all at once.

Let's use how my game **Tactical Tapir** loads fonts from disk as an example.

We use a `sync.WaitGroup` to wait for all the fonts to load, and a `map[string]font.Face` to store them.

Populating a map in parallel is a little tricky, because maps aren't thread-safe. There's a number of approaches you could use:

- use a `sync.Mutex` to synchronize access to the map
- use a channel to send the fonts as they're loaded to a single writer goroutine
- use `sync.Map`
- **populate a slice in parallel, then load the array into the map after the synchronization point**

We'll use the last approach, because it has no locks. Any of the four are unlikely to cause bottlenecks: use whatever you're most comfortable with.

Once `static.init` returns, all the fonts are loaded and ready to use, so all other packages can just read the map without worrying about synchronization.

```
package static
import "embed"
import "sync"
"github.com/golang/freetype/truetype"
canvasfont "github.com/tdewolff/canvas/font"

//go:embed fonts/*.woff2
var fs embed.FS
var Fonts = loadFont()
func loadFont() map[string]font.Face {
    wg := new(sync.WaitGroup)
    dir := must(fs.ReadDir("font"))
    tmp := make([]font.Face, len(dir)) // temporary slice to hold the fonts as they're loaded
    wg.Add(len(dir))
    for i, entry := range dir {
        i, entry := i, entry
        // needed until https://github.com/golang/go/wiki/LoopvarExperiment is implemented
        wg.Add(1)
        go func() {
            defer wg.Done()
            b := must(assets.ReadFile("font/" + d.Name()))
            if strings.Contains(d.Name(), "woff2") {
                b = must(canvasfont.ParseWOFF2(b))
            }
            ttf := must(truetype.Parse(b))
            tmp[i] = truetype.NewFace(ttf, &truetype.Options{Size: 16})
        }()


    }
    wg.Wait() // synchronization point: wait for all fonts to load

    // load the fonts into the map
    fonts := make(map[string]font.Face, len(tmp))
    for i, face := range tmp {
        if face != nil {
            fonts[dir[i].Name()] = face
        }
    }
    return fonts
}
func must[T any](t T, err error) {if err != nil {log.Fatal(err)} return t}

```

We can generalize this idea for other asset types like audio, images, and shaders, and use another layer of fork-join parallelism to load all **those** at once:

```
var (
    Audio  = make(map[string][]byte, len(must(assets.ReadDir("audio"))))
    Img    = make(map[string]*ebiten.Image, len(must(assets.ReadDir("img"))))
    Fonts  = make(map[string]font.Face, len(must(assets.ReadDir("font"))))
    Shader = make(map[string]*ebiten.Shader, len(must(assets.ReadDir("shader"))))
)
func init() {
    wg := new(sync.WaitGroup)
    wg.Add(4)
    go func() {Fonts = loadFont(); wg.Done() }()
    go func() { Audio = loadAudio(); wg.Done() }()
    go func() { Img = loadImg(); wg.Done() }()
    go func() { Shader = loadShader(); wg.Done() }()
    wg.Wait()
}
```

Let's see how this performs:

IN (serial)

```
git checkout serial-init
go build -o bin/tacticaltapir-serial
./bin/tacticaltapir-serial
```

OUT

```
# FULLY SERIAL
static: 14:50:04 static.go:64: audio    loading 33 files...
static: 14:50:06 static.go:78: initmap: static.loadAudio audio    loaded 33 files in 1.3351982s
static: 14:50:06 static.go:53: initMap: static.loadFonts: begin
static: 14:50:06 static.go:64: font     loading 61 files...
static: 14:50:06 static.go:78: initmap: static.loadFonts font     loaded 61 files in 234.0474ms
static: 14:50:06 static.go:53: initMap: static.loadShader: begin
static: 14:50:06 static.go:64: shader   loading 2 files...
static: 14:50:06 static.go:78: initmap: static.loadShader shader   loaded 2 files in 1.0396ms
static: 14:50:06 static.go:53: initMap: static.loadImg: begin
static: 14:50:06 static.go:64: img      loading 1 files...
static: 14:50:06 static.go:78: initmap: static.loadImg img      loaded 1 files in 1.5193ms
```

```
13  static: 14:50:06 static.go:165: all      loaded 97 files
    in 1.5723292s
```

1572ms

IN (parallel)

```
1  git checkout parallel-init
2  go build -o bin/tacticaltapir-parallel
3  ./bin/tacticaltapir-parallel
```

OUT

```
1  static: 14:48:52 static.go:78: initmap: static.loadShader
   shader   loaded 2 files in 24.4905ms
2  static: 14:48:52 static.go:78: initmap: static.loadImg
   img      loaded 1 files in 13.1632ms
3  static: 14:48:52 static.go:78: initmap: static.loadFonts
   font     loaded 61 files in 171.8455ms
4  static: 14:48:52 static.go:78: initmap: static.loadAudio
   audio    loaded 33 files in 249.365ms
5  static: 14:48:52 static.go:165: all      loaded 97 files in
   250.4947ms
```

1572ms / 250.49ms ≈ 6.275 . ***That's over 6x faster!*** While the difference between 1.5s and 250ms may not seem like much, as we scale the program, the difference will become more and more pronounced. 30s vs 5s is night and day.

4.4. Store assets in the form you can use them most immediately

4.4.1. use code as a data-description language

Instead of storing your 100000x item list as JSON, XML, TOML, or some other serialization format, just describe it in code. Why add extra steps?

To use **Tactical Tapir** again as an example, here's how I define guns: I just have a big flat array of structs, indexed by `GunID`, an enum:

```
1  type GunID byte
2  const (
3   PISTOL GunID = iota                  // single pistol,
   held in the right hand.
4
5       // many omitted
6
7   RIFLE                    // sniper rifle. slow to fire,
   large clip, pierces enemies and walls.
8   GUN_ID_N                 // number of guns; must be last
9  )
10
```

```
// gun definitions defined at compile time
var Defs = [GUN_ID_N]Def{
        // many omitted
    PISTOL: {
        GunID:    PISTOL,
        AmmoID:   AMMO_9MM,
        FireType: SEMIAUTO,
        HoldType: HOLD_RHAND,

        AmmoPerClip:               10,
        AmmoPerShot:               1,
        CanPutRoundInChamber:      true,
        // many omitted
        PPierce:      0,
        PSpread:      0.03,
        ReloadFrames: 10,
    },
}
```

4.4.2. pre-encode or render assets to their use format

For example, instead of rendering HTML via text templates on boot, you can render the HTML pre-compile and bake it into the binary. **That's how you're reading this page right now.**

4.4.3. avoid extra encoding/decoding steps

Sometimes you *do* need to store data in a format that isn't immediately usable: e.g, to avoid binary bloat. In this case, choose a format that encodes and decodes quickly. A potential avenue to improve **Tactical Tapir's** boot times by storing audio in formats that decode faster (like WAV or direct PCM rather than mp3), and storing our fonts directly as TTF files instead of WOFF2. When considering alternatives for compression or encoding, always **measure**, don't just assume. Performance can be counterintuitive: sometimes a compressed format is *faster* to load than an uncompressed one, because the compressed format makes better use of cache. If I end up doing this, I'll update this post with the results.

4.5. Lazy evaluation

Booting our program often involves setting up a bunch of dependencies (Database, Logger, Redis, etc) that are not required for the 'basic' functionality of the program. We'd like some way to get these up and running without waiting for every single one to be ready. After all, if we aren't going to use a component, we don't care if it's ready or not.

Lazy evaluation means waiting to initialize a dependency until you need it. You can use a `sync.Once` for lazy initialization. Here's an example of a lazy-initialized database:

```
package postgres
import (
    "database/sql"
    _ "github.com/lib/pq" // import enables postgres driver
    "sync"
)



var db, dbErr *sql.DB // as before
var once sync.Once // as before



// DB returns the database connection, initializing it if necessary.
func DB() (*sql.DB, error) { once.Do(initDB); return db, dbErr}

// MustDB is as DB, but panics on error.
func MustDB() *sql.DB {
    once.Do(initDB)
    if dbErr != nil {
        panic(dbErr)
    }
    return db
}
func initDB() {db, dbErr = sql.Open("postgres", "...")}
```

The user simply calls `DB()` or `MustDB()` anywhere they need a database connection. The first call initializes the database, and subsequent calls return the same connection: the runtime will make sure you block until the connection is ready.

4.6. Nonblocking eager initialization

Lazy initialization is helpful if we don't need a dependency right away, but what if we do? After all, waiting until you're hungry to order lunch makes lunch *later*, not sooner. This analogy suggests it's own solution: call in an order for lunch, then pick it up when you need it.

In programming terms, we **eagerly initialize dependencies**, but we refrain from blocking on them until we need them. Implementation is trivial: just call `sync.Once.Do` in it's own goroutine during `init()`.

```
func init() { go once.Do(initDB) }
```

This isn't a lot of work, but you'll start needing a lot of scaffolding: each dependency will need it's own line in `init()`, two functions, and a handful of variables.

4.7. Generics to the rescue: `nonblocking[T]`

We can cleanly abstract both 'classic' lazy initialization and the nonblocking eager kind with a new type using Go's generics.

```
// nonblocking[T] is a lazy-initialized value of type T.
// build a nonblocking[T] with NewLazy[T]() or NewEager[T]()
type nonblocking[T any] struct {
    once sync.Once          // guards initialization
    val  T                  // result of initialization, once initialized
    err  error              // error from initialization, once initialized
    fn   func() (T, error) // initializing function, called with Once().
}

// initialize the nonblocking[T] by evaluating fn() and storing the result.
func (nb *nonblocking[T]) initialize() { nb.once.Do(func() { nb.val, nb.err = nb.fn() }) }

// NewEager returns a *nonblocking[T] that will be initialized immediately in its own goroutine.
func NewEager[T any](f func() (T, error)) *nonblocking[T] {
    nb := &nonblocking[T]{fn: f}
    go nb.initialize()
    return nb
}

// NewLazy returns a *nonblocking[T] that will be initialized on first call to Get().
func NewLazy[T any](f func() (T, error)) *nonblocking[T] { return &nonblocking[T]{fn: f} }

// Get returns the value of the nonblocking[T], initializing it if necessary.
func (nb *nonblocking[T]) Get() (T, error) { nb.initialize(); return nb.val, nb.err }

// MustGet is as Get, but panics on error.
func (nb *nonblocking[T]) MustGet() T {
    nb.initialize()
    if nb.err != nil {
        panic(nb.err)
    }
    return nb.val
}
```

This new API is much simpler to use:

```
// see https://go.dev/play/p/VvSh3C4RVvK
```

```
 2   var DB = NewEager(newDB)
 3
 4   // i've added some sleeps to demonstrate the
 5   // interleaving of initialization and use.
 6   func main() {
 7       time.Sleep(160 * time.Millisecond)
 8
 9       fmt.Println("main: 1")
10       DB.MustGet()
11       fmt.Println("main: 2")
12   }
13
14   func newDB() (*sql.DB, error) {
15       time.Sleep(100 * time.Millisecond)
16
17       fmt.Println("newDB: 1")
18       time.Sleep(100 * time.Millisecond)
19       fmt.Println("newDB: 2")
20       return new(sql.DB), nil
21   }
```

IN:

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

OUT:

```
1   newDB: 1
2   main: 1
3   newDB: 2
4   main: 2
```

Don't make everything Eager - 'regular' lazy evaluation is often the best choice, and most things initialize so fast there's no need to do anything at all. But I highly recommend using this pattern for any dependency that takes more than a few milliseconds to initialize.

5. putting it all together

These techniques are orthogonal. For example, we could use eager nonblocking initialization *on* our parallelized static asset loading, so that the rest of the program can start up while the assets are loading. We could also lazily initialize some assets that are unlikely to be used rather than loading them all at once, etc. Keep in mind, if you find a particularly sticky package that boots slowly, **you'll need to rely on traditional programming techniques to speed it up**. Parallelism cannot fix algorithmic complexity, only paper over it.

6. conclusion

Hope you found this helpful! A quick-booting program should be a source of pride. Your program may not do anything, but by god, it'll do nothing fast.

Next we're going to talk about how to make your program *compile* and *deploy* quickly. After all, as a developer, you probably build nearly as often as you run: shouldn't that process be as fast and painless as possible?

After that, I plan on an article about *why* boot time and compile time matter.
If you liked this article, check out more, like this one on lesser-known go features.

MORE ARTICLES