Why are my Go executable files so large?

Size visualization of Go executables using D3

Raphael ‘kena’ Poss

March 2019

modified: 2021-04-17 07:00:00
slug: go-executable-size-visualization-with-d3
subtitle: Size visualization of Go executables using D3
category: Programming
tags: golang, compilers, analysis, programming languages, c++, tooling, python, cockroachdb

Contents

Overview 3

Background and motivation 4

Building the visualization 4
- Method . . . 4
- Extracting executable entries . . . 5
  - Decomposing Go symbols . . . 5
  - Decomposing C/C++ symbols . . . 6
- Organizing the data as a tree . . . 6
- Visualization using D3 . . . 7

Example visualization for a simple program 8
- Example program . . . 8
- Transformation process . . . 8
- Interactive visualization . . . 8
- Initial impressions . . . 9

What's inside a CockroachDB executable binary? 10
- Visual exploration . . . 10
- Analysis . . . 11
- Comparison between CockroachDB versions . . . 11

What's this `runtime.pclntab` anyway? 12

Of size/performance trade-offs and use cases 12

Other oddities 13

Summary and Conclusion 14

Copyright and licensing 14

Note

The latest version of this document can be found online at https://dr-knz.net/go-executable-size-visualization-with-d3.html. Alternate formats: Source, PDF.

Note

Erratum (2019-04-02): The increase in binary size from CockroachDB v1.0 to v19.2 is 94%/125%, not 194%/225% as initially written. The increase in source code is 40%, not 140%. The rest of the argument remains the same. Thanks to commenter Antonio D'souza for pointing this out.

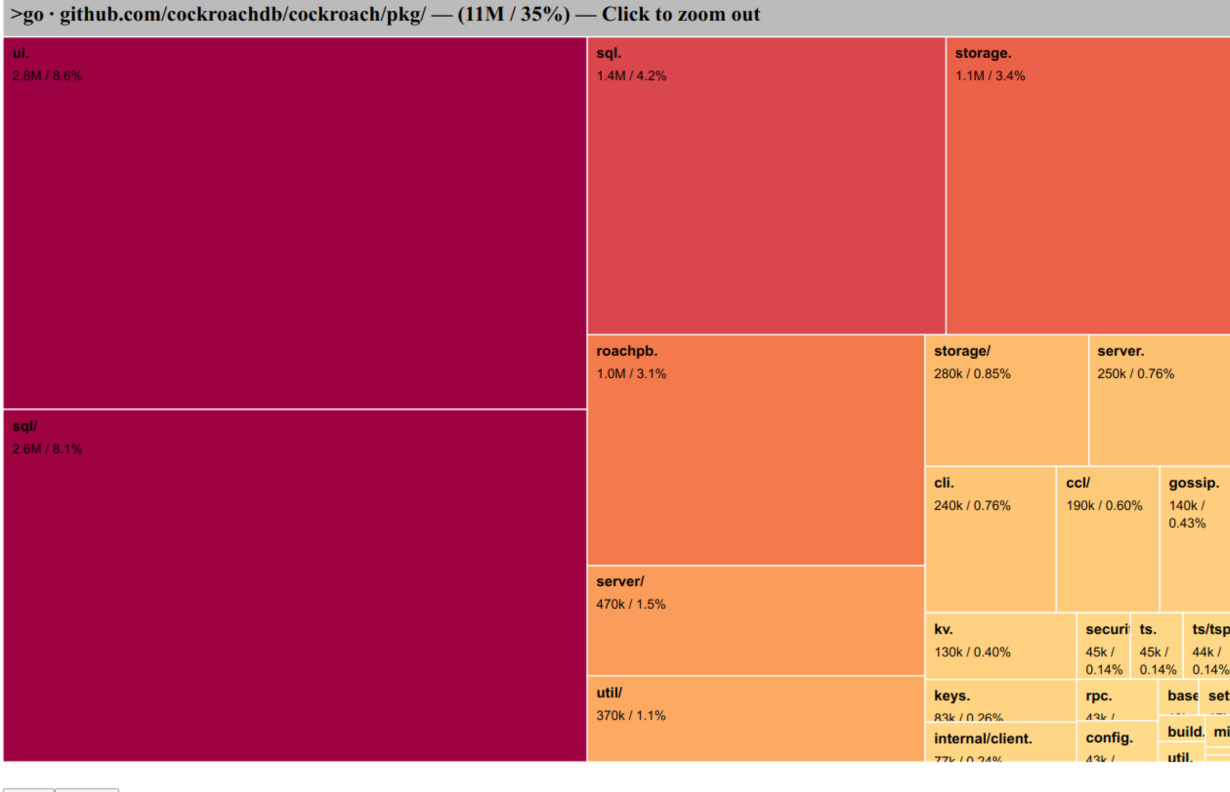
Note

A followup analysis is also available which revisits the findings with Go 1.15/1.16, in 2021.

Overview

I built some tooling to extract details about the contents of a Go executable file, and a small D3 application to visualize this information interactively as zoomable tree maps.

Here's a static screenshot of how the app illustrates the size of the compiled code, in this example for a group modules in CockroachDB:



The reason why I built this was to help me discover and learn what makes Go executable programs larger than I would expect. Thanks to this tool, I made several discoveries about how Go builds executable files.

AnD YoU wOnT BeLiEVe WhAt I fOuND InSiDE! (Read more below.)

The source code is public on GitHub: https://github.com/knz/go-binsize-viz

Background and motivation

My co-workers and I are busy preparing the latest release of CockroachDB, version 19.1. CockroachDB is released as a single program executable file containing all functionality.

Today, the latest build is 123MB large, 88MB stripped. This is a 94% (resp. 125%) increase since CockroachDB v1.0 was released, a bit more than two years ago. What happened?

This is especially puzzling given that:

- there is about 70MB of source code currently in CockroachDB 19.1, and there was 50MB of source code in CockroachDB v1.0. The increase in source was just ~40%. *How come did the binary size increase by a larger factor?*
- typically, compiled code is smaller than the source code it is compiled from. There is 70MB of source, a lot of which is just comments. Yet the binary is 88MB. *What makes the executable size larger than the source code?*

These questions alone made me curious.

Meanwhile, I also personally care about program size for practical reasons: smaller binaries cause less cache thrashing. They are easier to distribute and deploy. They make container orchestration more nimble. For these additional reasons, I would prefer if CockroachDB release binaries could become smaller. Figuring out what they contain might suggest how to achieve that.

Building the visualization

Method

My goal was to find some clarity into 123MB of inscrutable executable data. I started without knowing exactly how to achieve that.

I knew about the standard Unix utility nm which can display the size of individual entries in an executable file, and I knew that Go provides its own half-baked re-implementation (`go tool nm`). However, even a small program in Go will contain dozens of entries, and there were tens of thousands in the particular files I was looking at. So I needed an overview.

I also knew about tree maps, ever since the movie *Jurassic Park* featured the fsn 3D browser in 1993. This visualization represents sized hierarchical entries—for example files on disk, and in my case also like entries inside an executable binary—using visual elements whose size on the screen is proportional to their size in bytes on disk.

I thus decided to connect the two: **visualize Go binaries using tree maps**.

Furthermore, reminding myself that I have potentially tens of thousands of entries to visualize, I decided upfront that it would do me no good to attempt to represent them all simultaneously on screen. I thus went looking for **zoomable** tree maps.

Finally, I already had learned some D3 basics and wanted to learn more, so I decided I would use D3 for this exercise too.

So I went to search for "zoomable d3 tree map" on my favorite search engine and discovered that D3 has native supports for tree maps, provided some input data in a suitable format.

I initially started to tinker with Mike Bostok's zoomable treemaps but then quickly ran into some issue where some D3 feature I wanted to use was not available: Mike's code uses D3 V3, "modern" D3 runs at V5, and there were major API changes between V3 and V4. Converting the V3 example to V4 (or even V5) seemed non-trivial. Instead I set out to find some examples built for V4+. I then discovered this example from Jahnichen Jacques, itself inspired from Mike Bostok's, and finally this simpler example from Guglielmo Celata inspired from both but with a simpler implementation.

All these examples worked using D3 hierarchical data sets loaded from CSV or JSON with a particular schema.

The main thinking exercise was thus to massage the output of `nm` into a format suitable to load into D3. The rest of the work was simpler, to adapt D3 examples I found online into something that made sense for the type of data I was working with.

Extracting executable entries

A Go executable binaries contains, as per `go tool nm -size`, two types of entries:

- entries compiled from Go. These look like this:

```
10ddac0         17 t github.com/cockroachdb/cockroach/pkg/base.(*ClusterIDContainer).Unlock
```

- entries compiled from C/C++ via `cgo`, or linked into the Go program from external C/C++ libraries. These look like this (modulo filtering using `c++filt`):

```
28404a0         44 T rocksdb::PosixDirectory::~PosixDirectory()
```

The first column is the address, and is of no interest here. The second column is the size. The third column is the entry type, and of no interest here either. The last part is the symbol for the entry.

To build a *tree* visualization we thus need to decompose each symbol into *name components* that group the symbols into a hierarchy.

Decomposing Go symbols

Intuitively, a Go symbol contains a hierarchical package path (e.g. `github.com/lib/pq`) and some package-local name. The package-name name is either global in the package (e.g. `main`), or a method name with some receiver type prefix (e.g. `(*File).Write`).

This model accurately describes a large majority of symbols, and can be readily decomposed using a simple regular expression. However, we quickly come across exotic names that do not fit this model:

```
5250978         13 d crypto/tls..gobytes.1
3823b40         48 r go.itab.*compress/flate.byLiteral,sort.Interface
     aa3740        113 t go.(*struct { io.Reader; io.Closer }).Close
     e79cb0         10 t database/sql.glob..func1
     8ce580        123 t encoding/json.floatEncoder.encode-fm
     73aed0         82 t runtime.gcMarkDone.func1.1
```

I thus iterated to expand a simple regular expression to properly decompose the variety of names found in Go binaries. The result was a bit gnarly can be found here.

For the examples above, my program produces the following:

Size	Path	Name
13	`['crypto/', 'tls.', '.gobytes.']`	`1`
48	`['compress/', 'flate.']`	`go.itab.*byLiteral,sort.Interface`
113	`['go.', '(*struct { io.Reader; io.Closer }).']`	`Close`
10	`['database/', 'sql.', 'glob..']`	`func1`
123	`['encoding/', 'json.', 'floatEncoder.']`	`encode-fm`
82	`['runtime/', 'gcMarkDone.', 'func1.']`	`1`

Decomposing C/C++ symbols

Intuitively, a C/C++ symbol contains a hierarchical namespace path (e.g. `std::` or `google::protobuf::`) and then a variable or function name.

Again, looking at real-world binaries, we easily find many items that don't fit the simple model:

```
37ee1d0          8 r $f64.c05eb8bf2d05ba25
26abe20        100 T void rocksdb::JSONWriter::AddValue<int>(int const&)
28388f0         71 t rocksdb::(anonymous namespace)::PosixEnv::NowNanos()
2821ae0        231 T rocksdb::NumberToString[abi:cxx11](unsigned long)
 5b8c5c         34 t rocksdb::PosixRandomRWFile::Sync() [clone .cold.88]
265a740        211 T google::protobuf::internal::LogMessage::operator<<(long)
```

Using the same technique, I iterated from a simple regular expression to decompose the variety of symbols encountered. I even went one step further and chose to decompose identifiers at underscore boundaries. The result regular expressions are again rather complex and can be found here.

For the examples above, my program produces the following:

Size	Path	Name
8	`['$f64.']`	`c05eb8bf2d05ba25`
100	`['rocksdb::', 'JSONWriter::']`	`void AddValue<int>(int const&)`
71	`['rocksdb::', '(anonymous namespace)::', 'PosixEnv::']`	`NowNanos()`
231	`['rocksdb::', 'PosixRandomRWFile::']`	`Sync() [clone .cold.88]`
211	`['google::', 'protobuf::', 'internal::', 'LogMessage::']`	`operator<<(long)`

Some difficulty arises from the fact that C++ symbols can contain an arbitrarily amount of nested template parameters or parentheses in types, and regular expressions cannot match recursively.

My current implementation is thus restricted to only supports 6 levels of nesting. This appears to be insufficient to capture all symbols in my program of interest (where some symbols contain 10+ levels of nesting!) but I chose to exclude a few symbols to keep my regular expression simple(r). In my target analysis, the size of these excluded symbols is negligible anyway.

Organizing the data as a tree

After decomposing the *path components* of each symbol, my program creates nested Python dictionaries using a simple recursive function.

However, the result of this strategy for the path `a,b,c` is something like this:

```
{'children':{
```

```
        'a':{'children':{
                'b':{'children':{
                      'c': ...
                      }}
             }}
}}
```

Whereas the D3 visualization code really wants this:

```
{'children':[
        {'name':'a', 'children':[
                {'name':'b', 'children':[
                       {'name':'c', ... }
                      ]}
             ]}
]}
```

For this, I built a separate simplification program that turns the former format into the latter.

The reason why I separated the code into two programs is that the decomposition of symbols is rather expensive, and once I was satisfied with the decomposition I wanted the ability to iterate quickly on the tree transform without having to decompose over and over again.

Additionally, the simplification program collapses ("flattens") multiple hierarchy levels with a single child into just one level with a combined name. For example, the hierarchy `a/ → b/ → c/ → x,y` becomes `a/b/c/ → x,y`.

Visualization using D3

The original D3 tree map example as initially designed by Mike Bostok, and modified by Jahnichen Jacques and Guglielmo Celata operates thus:

1. it prepares a SVG canvas in a named HTML "chart" entity;
2. it defines a display function which, given a computed D3 tree map layout, creates a 3-level (grandparent-parent-child) visualization in the SVG;
3. the display function internally defines transition logic to zoom in and out when clicking on the canvas;
4. it loads the data from JSON, attaches it to a D3 tree map layout, and renders it using the aforementioned facilities.

On top of this logic by the previous authors, I added the following:

- displaying sizes using both absolute values and as percentages;
- a stable color map;
- the ability to switch betwen visualization of sizes and visualization of counts;
- the ability to view multiple data sets inside the same web page.

Example visualization for a simple program

Example program

We'll use the following Go code:

```
package main

import "fmt"

var x = struct { x [10000]int }{}

func main() {
        fmt.Println("hello world", x)
}
```

I choose to use a large struct for variable `x` so that the `main` package becomes significant in size compared to the imported `runtime` objects.

The program can be compiled as usual:

```
$ go build hello.go
```

Transformation process

I then use the following commands:

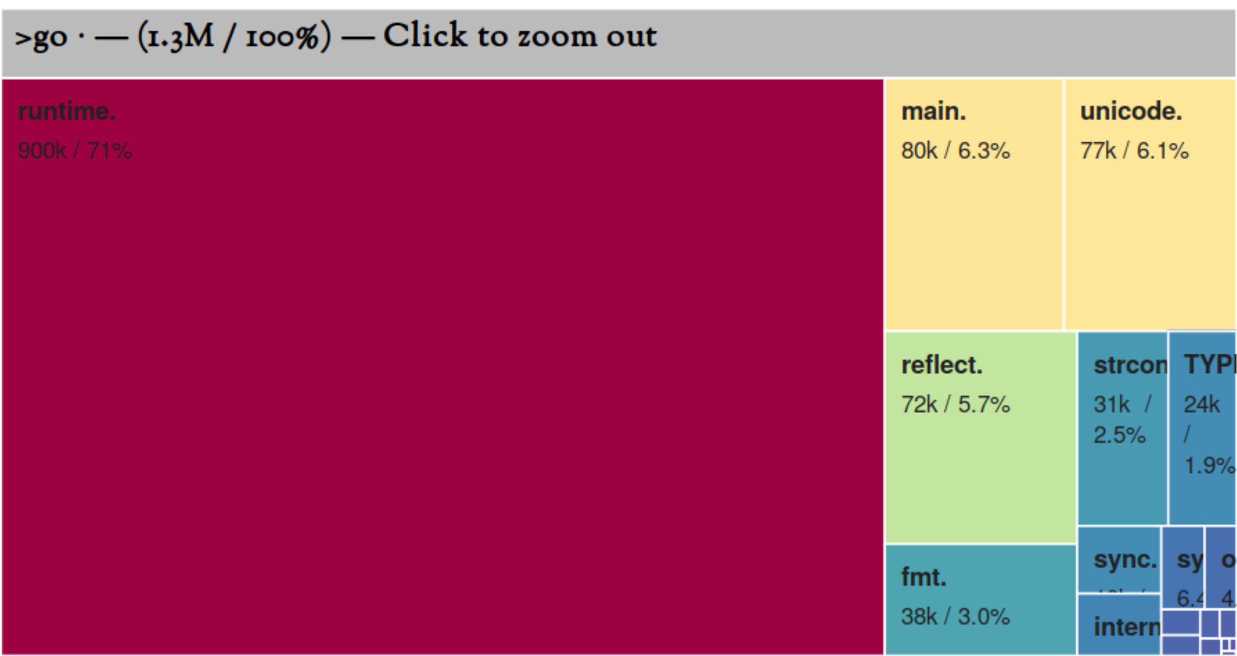
```
$ go tool nm -size hello            >hello.symtab
$ python3 tab2pydic.py hello.symtab >hellodic.py
$ python3 simplify.py hellodic.py   >hello.js
```

Interactive visualization

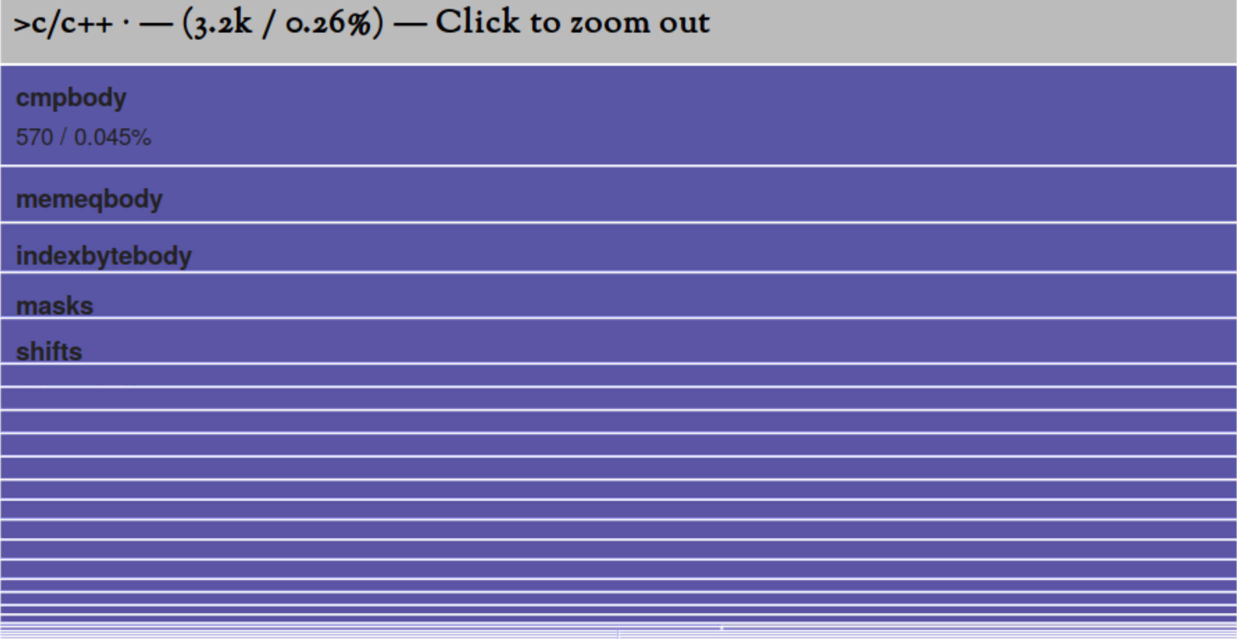
The following HTML code is sufficient:

```
<html>
  <head>
       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
       <link rel="stylesheet" type="text/css" href="treemap.css">
  </head>
  <body>
       <p class="chart" id="chart"></p>
       <script src="js/d3.v4.min.js" type="text/javascript"></script>
       <script src="js/d3-color.v1.min.js"></script>
       <script src="js/d3-interpolate.v1.min.js"></script>
       <script src="js/d3-scale-chromatic.v1.min.js"></script>
       <script src="app3.js" type="text/javascript"></script>
       <script type="text/javascript">
         viewTree("chart", "example-data/hello.js");
       </script>
  </body>
</html>
```

Which renders as follows (ensure Javascript is enabled):



Although this simple executable file appears to only contain Go symbols, it actually does contain C/C++ symbols too. However, their size is negligible and they initially appear as a mere line on the right side of the tree map. By clicking on that line, you may be able to zoom into them and obtain this:



Initial impressions

The small programm above contains 6 lines of source code and compiles to a 1.3MB binary. The breakdown of sizes is as follows:

Package	Size
runtime	900K / 71%
main	80K / 6.3%
unicode	77K / 6.1%
reflect	72K / 5.7%
fmt	38K / 3.0%
strconv	31K / 2.5%
sync	10K / 0.8%
internal	9K / 0.7%
syscall	6K / 0.5%
(others)	(remainder)

In addition to code compiled from source, 24K (1.9% of size) are compiler-generated type equality and hashing functions. These are accumulated in the box TYPEINFO in the tree map.

The following becomes clear quickly:

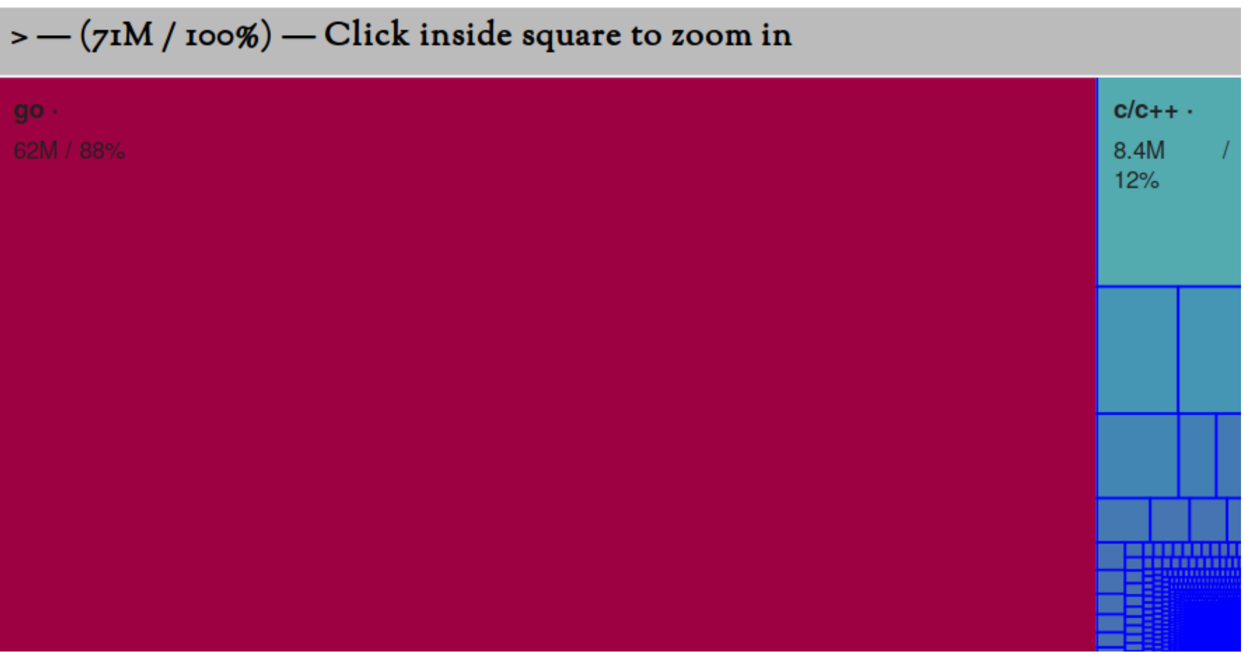
- the Go standard library is not well modularized; importing just one function (`fmt.Println`) pulls in about 300KB of code.
- even accounting for `fmt.Println` and its dependencies and the 80K of predictable code from the main program, we are still left to wonder about 900K (71%) of code from the `runtime` package.
- Zooming into there, we see that 450K (35%) are taken by just one single entry with name `runtime.pclntab`. This is more than the code for the main program and all the supporting code for `fmt.Println` combined.

We'll come back to this below.

What's inside a CockroachDB executable binary?

Visual exploration

Applying the tool on a recent CockroachDB build, one can find the following:



Analysis

By exploring this visualization, we discover:

- 71M of total entries;
- 8.4M (12% of size) in C/C++ entries, including 3M (4.3%) from RocksDB;
- 62M (88%) in Go entries, including:
  - 32M (45%) compiled directly from CockroachDB source code or its dependencies;
  - 26M (36%) from the `runtime` package.

... wait, what?

We established above, in the simple example, that `runtime` was about 900K in size. Now it has become 26M—a 28X increase! What is going on?

Zooming in, the mystery is revealed: all of the increase went into that single object `runtime.pclntab`. The other entries in package `runtime` do not appear to differ between programs.

We will come back to this `pclntab` object later below.

Comparison between CockroachDB versions

The visualization above was for a pre-release build of CockroachDB v19.1. For reference, here is the data for CockroachDB v1.0:

This has:

- 32M of total entries;
- 2.5M (7.8%) in C/C++ entries, including 1.5M (4.7%) from RocksDB;

- 30M (92%) in Go entries, including:
  - 16M (50%) compiled from CockroachDB sources and dependencies;
  - **7.9M (25%) from the `runtime` package, of which 7.3M (23%) comes** directly from `pclntab`.

Recalling the initial problem statement, CockroachDB increased about 40% in source code size between v1.0 and v19.1. Thanks to the visualization, we observe that the compiled code that directly originates from the CockroachDB sources increased from 16M to 32MB, which is about a 100% increase.

Meanwhile, `runtime.pclntab` increased from 7.9M to 26M—a 230% increase!

What's this `runtime.pclntab` anyway?

It is not too well documented however this comment from the Go source code suggests its purpose:

> // A LineTable is a data structure mapping program counters to line numbers.

The purpose of this data structure is to enable the Go runtime system to produce descriptive stack traces upon a crash or upon internal requests via the `runtime.GetStack` API.

So it seems useful. *But why is it so large?*

The URL https://golang.org/s/go12symtab hidden in the aforelinked source file redirects to a document that explains what happened between Go 1.0 and 1.2. To paraphrase:

- prior to 1.2, the Go linker was emitting a *compressed* line table, and the program would decompress it upon initialization at run-time.
- in Go 1.2, a decision was made to pre-expand the line table in the executable file into its final format suitable for direct use at run-time, without an additional decompression step.

In other words, **the Go team decided to make executable files larger to save up on initialization time**.

Also, looking at the data structure, it appears that its overall size in compiled binaries is super-linear in the **number of functions** in the program, in addition to how large each function is. We will come back to this below.

Of size/performance trade-offs and use cases

The change in design between Go pre-1.2 and go 1.2 is a classic trade-off between performance (time) and memory (space).

When is this trade-off warranted?

- if a program is *executed often* (e.g. microservice, system tools), then it is worth accelerating its start-up time at the cost of an increase in size.

  If, moreover, the program is *small*, with relatively few functions, (a reasonable assumption for programs executed often, like microservices or system tools), the increase in size incurred by `runtime.pclntab` will be negligible and thus have no significant impact on usability: file transfers, deployments, orchestration, etc.

  In that case, the Go 1.2 design is sound and warranted.

- if, in contrast, a program is *executed rarely* (e.g. a system service that runs continuously in the background, like, say … a database server), then the start-up time can be amortized throughout the lifetime of the process. The performance benefit of accelerating start-up is then not so clear.

  If, moreover, the program is *large*, with tens of thousands of of functions (a reasonable assumption for complex, feature-driven enterprise software like … database servers), the increase in size incurred by `runtime.pclntab` becomes inconveniently significant.

  This makes the Go 1.2 design … not so advantageous.

In the case of CockroachDB, `runtime.pclntab` could soon exceed the entirety of code compiled from sources, even with a conservative assumption of linear growth in compiled code size:

Year	**2017**	**2019**	**2021 (projected)**	**2023**
Total size	32M	71M	157M	350M
CockroachDB code	16M (50%)	32M (45%)	64M (41%)	128M (37%)
`runtime.pclntab`	7.3M (23%)	26M (36%)	85M (54%)	281M (80%)

Other oddities

- the Go compiler and linker always produce and keep the following entries, even when they are only used in functions elided by the linker because they are not used:

  - **Go interface conversion tables (`go.itab.`) between every pair of** interface ever mentioned in the source code.
    This accounts for about 1% of the CockroachDB 19.1 binary, and is predicted to increase with the introduction of more inter-component interfaces for testing in 19.2.
  - Type equality and hashing functions (`type..`).

    This accounts for about 1.2% of the CockroachDB 19.1 binary, and is predicted to increase with the increasing use of code generation for optimizations inside CockroachDB 19.2.

- as discussed in my previous article, Go uses memory instead of registers to pass arguments and return values across function calls. On x86/x86-64, memory-accessing instructions use longer machine encodings than register-only instructions.

  In my experience, this is specifically the inflection point for the ratio between source code size and compiled code size in a monomorphic C-like language: when targeting fixed-length instruction encodings and/or a memory-heavy calling convention, the size of the compiled code grows larger than the source code (excluding comments and whitespace). We can see this with both C on ARMv6 (no Thumb) or Go on x86(-64).

  When targeting variable-length instruction encodings and the calling convention suitably utilizes the more compact instructions, the size of the compiled code becomes smaller than the source code. We can see this with C on x86(-64) with a decent compiler, but, as examined here, not with Go.

Summary and Conclusion

To understand the internal structure of executable files compiled from Go, I built a D3 visualization using zoomable tree maps. This visualization is published on GitHub and has been tested to work with any executable produced by Go versions between 1.4 and 1.12.

Using this tool, I have analyzed the space usage inside the monolithic CockroachDB binary, `cockroach`. I discovered that the majority of the growth of `cockroach` over time is concentrated in one object, `runtime.pclntab`.

This object is automatically generated by the Go compiler to support the generation of textual stack traces at run-time, for example in error reports.

A design choice made in Go 1.2 causes this object to grow super-linearly in the *number* of functions in a program, in addition to the sum of their sizes. Between CockroachDB 1.0 and 19.1, `runtime.pclntab` grew by 230% while the total amount of source code only increased by 40% and compiled code by 100%.

This design choice was intended to lower the start-up time of programs, and contrasts with the previous design using compressed tables—which is, incidentally, the industry standard for other C-like languages, even in contemporary compilers. This performance goal is not relevant to server software with long-running processes, like CockroachDB, and its incurred space cost is particularly inconvenient for large, feature-rich programs.

> **Note**
> In 2021, the situation is slightly different but arguably worse. Read the followup analysis for more details.

Copyright and licensing

Copyright © 2014-2021, Raphael 'kena' Poss. Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.