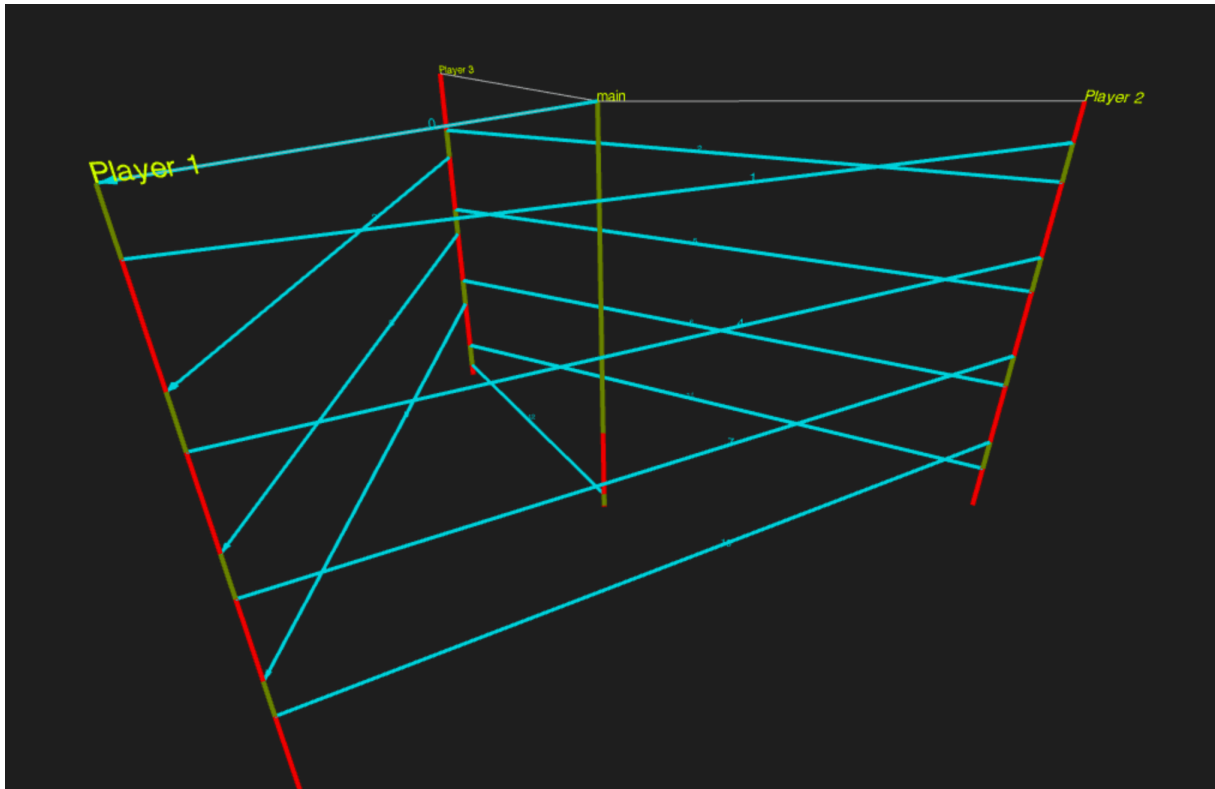

GoTrace - Go Concurrency 3D Tracer

GoTrace is a 3D WebGL visualizer of Go concurrency. It analyzes trace produced by `go tool trace` and renders visualization of concurrency flow.

Original article: https://divan.github.io/posts/go_concurrency_visualize/

Slides from GopherCon'16: <http://divan.github.io/talks/2016/gophercon/>



Intro

This tool generates 3D visualization of Go concurrency flow by analyzing its execution trace. To view the visualization you need a modern browser with WebGL support (pretty much any browser nowadays).

Its primary goal is to be an educational tool for Go concurrency. It works well with small programs that produce short traces (see Scale issues below).

Usage

First, install gotrace:

```
1 go get -u github.com/divan/gotrace
```

Second, use patched Go runtime to produce trace and binary. There are two ways to do it - use a docker container or apply the patch locally.

Quick example using pre-made docker image (jump to detailed instructions):

```
1 docker run --rm -it \  
2     -e GOOS=darwin \  
3     -v $(pwd):/src divan/golang:gotrace \  
4         go build -o /src/binary /src/examples/hello.go  
5 ./binary 2> trace.out  
6 gotrace ./trace.out ./binary
```

Or, using local patched Go installation (jump to detailed instructions):

```
1 gotrace examples/hello.go
```

Prepare your program

Now, **please learn some important things before trying your own code**. Feel free to play first with code in **examples/** folder.

Theoretically, gotrace should do all the magic itself and be able to handle any Go program. That's the goal, but at the present moment, if you want to get good/meaningful visualization, you should follow some rules and suggestions.

Make it short The height of program visualization currently is a fixed value, so any trace fits into the screen height. This means, that example running 1 minute will be visualized at different scale from program running 1 second.

Depending on what you try to see, but rule of thumb is - **the shorter execution time, the better**. See [examples/](#) dir for good samples that produce nice visualizations.

Insert runtime/trace yourself In order to produce the trace, your program should be instrumented with special code. [gotrace](#) can do this automatically for you, but **in some cases it's wiser to put this code by yourself**. Here is a typical example:

```
1 package main  
2  
3 import (  
4     "os"  
5     "runtime/trace"
```

```
6 )
7
8 func main() {
9     trace.Start(os.Stderr)
10    ...
11    trace.Stop()
12 }
```

Currently it's important to write trace into `os.Stderr`. See issue #X if your example uses `stderr` for other needs.

Consider inserting very short time.Sleep() calls If you are trying to visualize some things that happen at nanosecond/microsecond level, it could be wise to insert `time.Sleep(1 * time.Millisecond)` calls to get more clear visualization.

For example, if your code runs 100 goroutines in a loop, their IDs and their start order probably would be different, resulting in slightly messed picture. So, changing:

```
1  for i := 0; i < 100; i++ {
2      go player(table)
3  }
```

to

```
1  for i := 0; i < 100; i++ {
2      time.Sleep(1 * time.Millisecond)
3      go player(table)
4  }
```

will help to make better visualization.

Try to keep number of goroutines/events small The fewer objects that will be rendered, the better. If you have many things to render, WebGL will just hang your browser. Also, keep in mind, that point of visualization is to express something. So running 1024 workers will result in a heavy visualization where you will not see separate goroutines. Setting this value to, say, 36 will produce much more clear picture.

Detailed instructions

The next step is to build your program. You will need to build using the patched Go runtime. So if you patched it yourself (see Appendix A), you just have to run `go build`, or, even simpler, let `gotrace` do it for you. But most people, probably wouldn't want to do this and prefer using Docker for it.

Using Docker You will need Docker installed and running.

Then pull the image from Docker Hub:

```
1 docker pull divan/golang:gotrace
```

or build it yourself:

```
1 docker build -t "divan/golang:gotrace" -f runtime/Dockerfile runtime/
```

If everything went ok, you should have `divan/golang:gotrace` image in your local docker (check with `docker images` command).

Now, use this command to produce the binary: ##### MacOS X `docker run -rm -it`

`-e GOOS=darwin`

`-v $(pwd):/src divan/golang:gotrace`

`go build -o /src/binary /src/examples/hello.go`

Linux

```
1 docker run --rm -it \  
2     -v $(pwd):/src divan/golang:gotrace \  
3         go build -o /src/binary /src/examples/hello.go
```

Windows

```
1 docker run --rm -it \  
2     -e GOOS=windows \  
3     -v $(pwd):/src divan/golang:gotrace \  
4         go build -o /src/binary.exe /src/examples/hello.go
```

3. Run it and save the trace.

Once you have the binary, you can run it and save the trace:

```
1 ./binary 2> trace.out
```

4. Run gotrace (finally)

Now, it's time to run `gotrace` and feed the binary and the trace to produce the visualization:

```
1 gotrace ./trace.out ./binary
```

It should start the browser and render visualization.

Visualization

Colors

Different colors of goroutines represent different states.

- red - blocked state
- green - unblocked
- yellow - unblocked and using CPU

Colors of goroutines' connections and sendings over the channels are the same.

Hotkeys

You can use the mouse/trackpad to zoom/rotate/pan visualization. On MacOS X you use single tap and move to rotate, double-finger touch to zoom, and double-finger tap and move to pan.

You may also try it with a Leap Motion controller for zooming and rotating with hands - switch to **leap** branch.

Also there are some useful hotkeys:

- **r** - restart
- **p** - pause
- **1, 2, 3, 4, 0** - highlight modes (0 - default)
- **+/-** - increase/decrease width of lines
- **s/f** - slower/faster animation

Limits/Known issues

- Value of variable being sent to the channel is supported only for integer types. (see Issue #3)
- Buffered channels doesn't work (yet) (see Issue #2)

Appendix A - patching Go locally

If you really want to play around with gotrace, you may want to patch Go runtime yourself. It will allow you to run gotrace as easy as `gotrace main.go` without all intermediate steps described above.

Here are instructions on how to do it (MacOS X and Linux).

-
1. Assuming your Go installation is in `/usr/local/go` (default), download Go 1.6.3 and unpack it into `/usr/local/go163`.

```
1 sudo -i
2 mkdir -p /usr/local/go163
3 curl https://storage.googleapis.com/golang/go1.6.3.src.tar.gz |
  tar -xz -C /usr/local/go163
```

2. Then, copy patch and apply it:

```
1 sudo patch -p1 -d /usr/local/go163/go < runtime/go1.6.3-tracenew.
  patch
```

3. Build new runtime:

```
1 sudo -i
2 cd /usr/local/go163/go/src
3 export GOROOT_BOOTSTRAP=/usr/local/go # or choose yours
4 ./make.bash
```

4. Finally, export PATH or use `ln -s` command to make this Go version actual in your system:

```
1 export PATH=/usr/local/go163/go/bin:$PATH
```

or (assuming your PATH set to use `/usr/local/go`)

```
1 sudo mv /usr/local/go /usr/local/go-orig
2 sudo ln -nsf /usr/local/go163/go /usr/local/go
```

NOTE: return your previous installation by `sudo ln -nsf /usr/local/go-orig /usr/local/go`

Now, you should be able to run `gotrace main.go` and get the result.

License

MIT License