A shell script for running Go one-liners

bitfield/script is a really neat Go project: it tries to emulate shell scripting using Go chaining primitives, so you can run code like this:

```
script.Stdin().Column(1).Freq().First(10).Stdout()
```

To achieve the same thing as:

```
cat file.txt | cut -f1 | sort | uniq -c | sort -nr | head -10
```

A comment from `jvictor118` on Hacker News:

> If one were actually going to use something like this, I'd think it'd be worth implementing a little shebang script that can wrap a single-file script in the necessary boilerplate and call go run!

This is exactly the kind of thing I can't quite be bothered to write myself, but I'm happy to coach GPT-4 through building.

The result: `goscript.sh`. You can use it like this:

```
cat file.txt | ./goscript.sh -c 'script.Stdin().Column(1).Freq().First(10).Stdout()'
```

Or you can create a script file like this one, saved as `top10.goscript`:

```
script.Stdin().Column(1).Freq().First(10).Stdout()
```

And run:

```
cat file.txt | ./goscript.sh top10.goscript
```

Finally, you can set the shebang line in a script file like this:

```
#!/tmp/goscript.sh
script.Stdin().Column(1).Freq().First(10).Stdout()
```

Then run this:

```
chmod 755 top10.goscript
cat file.txt | ./top10.goscript
```

The script

Here's the `goscript.sh` script that GPT-4 and I came up with:

```
#!/bin/bash

TMPDIR=$(mktemp -d /tmp/goscript.XXXXXX)
SUBDIR="$TMPDIR/goscript_inner"
mkdir -p $SUBDIR
trap "rm -rf $TMPDIR" EXIT
```

```
TMPFILE="$SUBDIR/script.go"

# Write boilerplate to tmpfile
cat > $TMPFILE <<EOF
package main

import (
    "github.com/bitfield/script"
)

func main() {
EOF

# Check for -c flag
if [ "$1" == "-c" ]; then
    # Add the literal string from argument
    echo "$2" >> $TMPFILE
else
    # Add user's code from file
    sed '/^#!/d' "$1" >> $TMPFILE
fi

# Close main function
echo "}" >> $TMPFILE

# Initialize a new module in subdir, fetch dependencies, and run
pushd $SUBDIR > /dev/null 2>&1
go mod init tmp > /dev/null 2>&1
go get github.com/bitfield/script > /dev/null 2>&1
go run script.go
popd > /dev/null 2>&1
```

And the full ChatGPT transcript that lead to the final script presented here.

(Missing from that transcript is the final step where we added the `sed` line to strip out the shebang.)

Here's what I learned from the above code.

The program itself is wrapped in the following boilerplate, using `>>` to write to the temporary file:

```
package main

import (
    "github.com/bitfield/script"
)

func main() {
    // User's code goes here
}
```

With modern Go you need to use the following pattern to get something like this to work with a dependency:

```
go mod init tmp
go get github.com/bitfield/script
go run script.go
```

`go get` downloads the dependency, using a cache if it's already been downloaded.

The shell script runs all of that in a temporary directory, created using:

```
TMPDIR=$(mktemp -d /tmp/goscript.XXXXXX)
SUBDIR="$TMPDIR/goscript_inner"
mkdir -p $SUBDIR
```

That `mktemp -d /tmp/goscript.XXXXXX` line uses the templating feature of `mktemp`, where a sequence of `XXX` is replaced by random characters.

The `trap` call is interesting - see also Running multiple servers in a single Bash script. Effectively it ensures the temporary directory is deleted when the script terminates, no matter why it terminates (success or error):

```
trap "rm -rf $TMPDIR" EXIT
```

I wanted to support two ways of calling the script:

```
./goscript.sh -c 'go code here'
./goscript.sh script.goscript
```

That's handled by this conditional check:

```
if [ "$1" == "-c" ]; then
    # Add the literal string from argument
    echo "$2" >> $TMPFILE
else
    # Add user's code from file
    sed '/^#!/d' "$1" >> $TMPFILE
fi
```

The `sed` line is necessary because if you have a script that looks like this:

```
#!/tmp/goscript.sh
script.Stdin().Column(1).Freq().First(10).Stdout()
```

That first line will be copied into the Go code in a way that breaks syntax. Using `sed` here strips that line out before copying the rest of the file into the `main()` function in the boilerplate.

With this accounted for, running `./top10.goscript` effectively runs the same as calling `./goscript.sh top10.goscript`.

Several of the commands in the script output information to `stdout` or `stderr` - we fixed that with this pattern:

```
go mod init tmp > /dev/null 2>&1
```

(It feels weird to refer to the combination of myself and GPT-4 as "we", but I think it's an honest description of the we we collaborated to build this.)

Related

gpt3 Using ChatGPT to write AppleScript - 2023-03-08

go Installing tools written in Go - 2024-03-25

llms Running nanoGPT on a MacBook M2 to generate terrible Shakespeare - 2023-02-01

python CLI tools hidden in the Python standard library - 2023-06-28

llms Expanding ChatGPT Code Interpreter with Python packages, Deno and Lua - 2023-04-30

gpt3 Using GPT-3 to figure out jq recipes - 2022-08-10

llms Running OpenAI's large context models using llm - 2023-06-13

github-actions GitHub Actions job summaries - 2022-05-17

bash Running multiple servers in a single Bash script - 2023-08-16

gpt3 Using ChatGPT Browse to name a Python package - 2023-06-18

Created 2023-08-20T09:00:31-07:00, updated 2023-08-20T15:27:17-07:00 · History · Edit