Tips on Compiler Optimization

Useful tips and tricks for programming in Go.

WE'LL COVER THE FOLLOWING

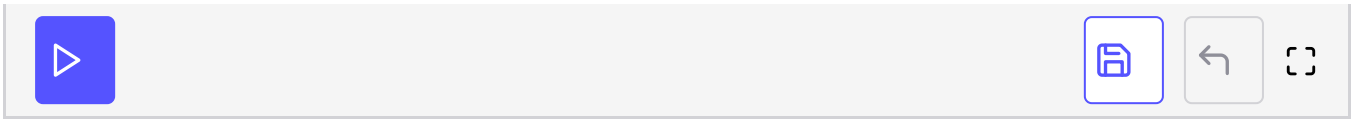
Compiler Optimizations

You can pass specific compiler flags to see what optimizations are being applied as well as how some aspects of memory management. This is an advanced feature, mainly for people who want to understand some of the compiler optimizations in place.

Let's take the following code example from an earlier chapter:

```
package main

import "fmt"

type User struct {
        Id             int
        Name, Location string
}

func (u *User) Greetings() string {
        return fmt.Sprintf("Hi %s from %s",
                u.Name, u.Location)
}

func NewUser(id int, name, location string) *User {
        id++
        return &User{id, name, location}
}

func main() {
        u := NewUser(42, "Matt", "LA")
```

```
        fmt.Println(u.Greetings())
}
```



Build your file (here called `t.go`) passing some `gcflags`:

```
$ go build -gcflags=-m t.go
# command-line-arguments
./t.go:15: can inline NewUser
./t.go:21: inlining call to NewUser
./t.go:10: leaking param: u
./t.go:10: leaking param: u
./t.go:12: (*User).Greetings ... argument does not escape
./t.go:15: leaking param: name
./t.go:15: leaking param: location
./t.go:17: &User literal escapes to heap
./t.go:15: leaking param: name
./t.go:15: leaking param: location
./t.go:21: &User literal escapes to heap
./t.go:22: main ... argument does not escape
```

The compiler notices that it can inline the `NewUser` function defined on line 15 and inline it on line 21. Dave Cheney has a great post about why Go's inlining is helping your programs run faster.

Basically, the compiler moves the body of the `NewUser` function (L15) to where it's being called (L21) and therefore avoiding the overhead of a function call but increasing the binary size.

The compiler creates the equivalent of:

```
func main() {
        id := 42 + 1
        u := &User{id, "Matt", "LA"}
        fmt.Println(u.Greetings())
}
```

On a few lines, you see the potentially alarming `leaking param` message. It doesn't mean that there is a memory leak but that the param is kept alive even after returning. The "leaked params" are:

- On the `Greetings`'s method: `u` (receiver)
- On the `NewUser`'s functon: `name`, `location`

The reason why `u` "leaks" in the `Greetings` method is because it's being used

The reason why `u` "leaks" in the `Greetings` method is because it's being used in the `fmt.Sprintf` function call as an argument. `name` and `location` are also

"leaked" because they are used in the `User`'s literal value. Note that `id` doesn't leak because it's a value, only references and pointers can leak.

X `argument does not escape` means that the argument doesn't "escape" the function, meaning that it's not used outside of the function so it's safe to store it on the stack.

On the other hand, you can see that `&User literal escapes to heap`. What it means is that the address of a literal value is used outside of the function and therefore can't be stored on the stack. The value *could* be stored on the stack, except a pointer to the value escapes the function, so the value has to be moved to the heap to prevent the pointer referring to incorrect memory once the function returns. This is always the case when calling a method on a value and the method uses one or more fields.

Expvar

TODO package

Setting the Build ID using git's SHA

It's often very useful to burn a build id in your binaries. I personally like to use the SHA1 of the git commit I'm committing. You can get the short version of the sha1 of your latest commit by running the following `git` command from your repo:

```
git rev-parse --short HEAD
```

The next step is to set an exported variable that you will set at compilation time using the `-ldflags` flag.

Environment Variables

Key:	Value:
GOPATH	/go

```
package main

import "fmt"
```

```
// compile passing -ldflags "-X main.Build <build sha1>"
var Build string


func main() {
        fmt.Printf("Using build: %s\n", Build)
}
```

Save the above code in a file called `example.go`. If you run the above code, `Build` won't be set, for that you need to set it using `go build` and the `-ldflags`.

```
$ go build -ldflags "-X main.Build a1064bc" example.go
```

Now run it to make sure:

```
$ ./example
Using build: a1064bc
```

Now, hook that into your deployment compilation process, I personally like Rake to do that, and this way, every time I compile, I think of Jim Weirich.

How to see what packages my app imports

\label{sec:list_imported_go_packages}

It's often practical to see what packages your app is importing. Unfortunately there isn't a simple way to do that, however it is doable via the `go list` tool and using templates.

Go to your app and run the following.

```
$ go list -f '{{join .Deps "\n"}}' |
  xargs go list -f '{{if not .Standard}}{{.ImportPath}}{{end}}'
```

Here is an example with the clirescue refactoring example:

```
$ cd $GOPATH/src/github.com/GoBootcamp/clirescue
$ go list -f '{{join .Deps "\n"}}' |
  xargs go list -f '{{if not .Standard}}{{.ImportPath}}{{end}}'
github.com/GoBootcamp/clirescue/cmdutil
github.com/GoBootcamp/clirescue/trackerapi
github.com/GoBootcamp/clirescue/user
github.com/codegangsta/cli
```

If you want the list to also contain standard packages, edit the template and use:

```
$ go list -f '{{join .Deps "\n"}}' |  xargs go list -f '{{.ImportPath}}'
```