Effective Go

Introduction

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

This document gives tips for writing clear, idiomatic Go code. It augments the language specification, the Tour of Go, and How to Write Go Code, all of which you should read first.

Examples

The Go package sources are intended to serve not only as the core library but also as examples of how to use the language. Moreover, many of the packages contain working, self-contained executable examples you can run directly from the golang.org web site, such as this one (if necessary, click on the word "Example" to open it up). If you have a question about how to approach a problem or how something might be implemented, the documentation, code and examples in the library can provide answers, ideas and background.

Formatting

Formatting issues are the most contentious but the least consequential. People can adapt to different formatting styles but it's better if they don't have to, and less time is devoted to the topic if everyone adheres to the same style. The problem is how to approach this Utopia without a long prescriptive style guide.

With Go we take an unusual approach and let the machine take care of most formatting issues. The `gofmt` program (also available as `go fmt`, which operates at the

package level rather than source file level) reads a Go program and emits the source in a standard style of indentation and vertical alignment, retaining and if necessary reformatting comments. If you want to know how to handle some new layout situation, run `gofmt`; if the answer doesn't seem right, rearrange your program (or file a bug about `gofmt`), don't work around it.

As an example, there's no need to spend time lining up the comments on the fields of a structure. `Gofmt` will do that for you. Given the declaration

```
type T struct {
    name string // name of the object
    value int // its value
}
```

`gofmt` will line up the columns:

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

All Go code in the standard packages has been formatted with `gofmt`.

Some formatting details remain. Very briefly:

Indentation
: We use tabs for indentation and `gofmt` emits them by default. Use spaces only if you must.

Line length
: Go has no line length limit. Don't worry about overflowing a punched card. If a line feels too long, wrap it and indent with an extra tab.

Parentheses
: Go needs fewer parentheses than C and Java: control structures (`if`, `for`, `switch`) do not have parentheses in their syntax. Also, the operator precedence hierarchy is shorter and clearer, so

  ```
  x<<8 + y<<16
  ```

  means what the spacing implies, unlike in the other languages.

Commentary

Go provides C-style `/* */` block comments and C++-style `//` line comments. Line comments are the norm; block comments appear mostly as package comments, but are useful within an expression or to disable large swaths of code.

The program—and web server—`godoc` processes Go source files to extract documentation about the contents of the package. Comments that appear before top-level declarations, with no intervening newlines, are extracted along with the declaration to serve as explanatory text for the item. The nature and style of these comments determines the quality of the documentation `godoc` produces.

Every package should have a *package comment*, a block comment preceding the package clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the `godoc` page and should set up the detailed documentation that follows.

```
/*
Package regexp implements a simple library for regular expressions.

The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

If the package is simple, the package comment can be brief.

```
// Package path implements utility routines for
// manipulating slash-separated filename paths.
```

Comments do not need extra formatting such as banners of stars. The generated

output may not even be presented in a fixed-width font, so don't depend on spacing for alignment—`godoc`, like `gofmt`, takes care of that. The comments are uninterpreted plain text, so HTML and other annotations such as `_this_` will reproduce *verbatim* and should not be used. One adjustment `godoc` does do is to display indented text in a fixed-width font, suitable for program snippets. The package comment for the `fmt` package uses this to good effect.

Depending on the context, `godoc` might not even reformat comments, so make sure they look good straight up: use correct spelling, punctuation, and sentence structure, fold long lines, and so on.

Inside a package, any comment immediately preceding a top-level declaration serves as a *doc comment* for that declaration. Every exported (capitalized) name in a program should have a doc comment.

Doc comments work best as complete sentences, which allow a wide variety of automated presentations. The first sentence should be a one-sentence summary that starts with the name being declared.

```
// Compile parses a regular expression and returns, if successful,
// a Regexp that can be used to match against text.
func Compile(str string) (*Regexp, error) {
```

If every doc comment begins with the name of the item it describes, the output of `godoc` can usefully be run through `grep`. Imagine you couldn't remember the name "Compile" but were looking for the parsing function for regular expressions, so you ran the command,

```
$ godoc regexp | grep -i parse
```

If all the doc comments in the package began, "This function...", `grep` wouldn't help you remember the name. But because the package starts each doc comment with the name, you'd see something like this, which recalls the word you're looking for.

```
$ godoc regexp | grep parse
    Compile parses a regular expression and returns, if successful, a
Regexp
    parsed. It simplifies safe initialization of global variables
holding
    cannot be parsed. It simplifies safe initialization of global
variables
$
```

Go's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables. Since the whole declaration is presented, such a comment can often be perfunctory.

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = errors.New("regexp: internal error")
    ErrUnmatchedLpar = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar = errors.New("regexp: unmatched ')'")
    ...
)
```

Grouping can also indicate relationships between items, such as the fact that a set of variables is protected by a mutex.

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

Names

Names are as important in Go as in any other language. They even have semantic effect: the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

Package names

When a package is imported, the package name becomes an accessor for the contents. After

```
import "bytes"
```

the importing package can talk about `bytes.Buffer`. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be

typing that name. And don't worry about collisions *a priori*. The package name is only the default name for imports; it need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

Another convention is that the package name is the base name of its source directory; the package in `src/encoding/base64` is imported as `"encoding/base64"` but has name `base64`, not `encoding_base64` and not `encodingBase64`.

The importer of a package will use the name to refer to its contents, so exported names in the package can use that fact to avoid stutter. (Don't use the `import .` notation, which can simplify tests that must run outside the package they are testing, but should otherwise be avoided.) For instance, the buffered reader type in the `bufio` package is called `Reader`, not `BufReader`, because users see it as `bufio.Reader`, which is a clear, concise name. Moreover, because imported entities are always addressed with their package name, `bufio.Reader` does not conflict with `io.Reader`. Similarly, the function to make new instances of `ring.Ring`—which is the definition of a *constructor* in Go—would normally be called `NewRing`, but since `Ring` is the only type exported by the package, and since the package is called `ring`, it's called just `New`, which clients of the package see as `ring.New`. Use the package structure to help you choose good names.

Another short example is `once.Do`; `once.Do(setup)` reads well and would not be improved by writing `once.DoOrWaitUntilDone(setup)`. Long names don't automatically make things more readable. A helpful doc comment can often be more valuable than an extra long name.

Getters

Go doesn't provide automatic support for getters and setters. There's nothing wrong with providing getters and setters yourself, and it's often appropriate to do so, but it's neither idiomatic nor necessary to put `Get` into the getter's name. If you have a field called `owner` (lower case, unexported), the getter method should be called `Owner` (upper case, exported), not `GetOwner`. The use of upper-case names for export provides the hook to discriminate the field from the method. A setter function, if needed, will likely be called `SetOwner`. Both names read well in practice:

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

Interface names

By convention, one-method interfaces are named by the method name plus an -er suffix or similar modification to construct an agent noun: `Reader`, `Writer`, `Formatter`, `CloseNotifier` etc.

There are a number of such names and it's productive to honor them and the function names they capture. `Read`, `Write`, `Close`, `Flush`, `String` and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method `String` not `ToString`.

MixedCaps

Finally, the convention in Go is to use `MixedCaps` or `mixedCaps` rather than underscores to write multiword names.

Semicolons

Like C, Go's formal grammar uses semicolons to terminate statements, but unlike in C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them.

The rule is this. If the last token before a newline is an identifier (which includes words like `int` and `float64`), a basic literal such as a number or string constant, or one of the tokens

```
break continue fallthrough return ++ -- ) }
```

the lexer always inserts a semicolon after the token. This could be summarized as, “if the newline comes after a token that could end a statement, insert a semicolon”.

A semicolon can also be omitted immediately before a closing brace, so a statement such as

```
    go func() { for { dst <- <-src } }()
```

needs no semicolons. Idiomatic Go programs have semicolons only in places such as `for` loop clauses, to separate the initializer, condition, and continuation elements. They

are also necessary to separate multiple statements on a line, should you write code that way.

One consequence of the semicolon insertion rules is that you cannot put the opening brace of a control structure (`if`, `for`, `switch`, or `select`) on the next line. If you do, a semicolon will be inserted before the brace, which could cause unwanted effects. Write them like this

```
if i < f() {
    g()
}
```

not like this

```
if i < f()  // wrong!
{           // wrong!
    g()
}
```

Control structures

The control structures of Go are related to those of C but differ in important ways. There is no `do` or `while` loop, only a slightly generalized `for`; `switch` is more flexible; `if` and `switch` accept an optional initialization statement like that of `for`; `break` and `continue` statements take an optional label to identify what to break or continue; and there are new control structures including a type switch and a multiway communications multiplexer, `select`. The syntax is also slightly different: there are no parentheses and the bodies must always be brace-delimited.

If

In Go a simple `if` looks like this:

```
if x > 0 {
    return y
}
```

Mandatory braces encourage writing simple `if` statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a `return` or `break`.

Since `if` and `switch` accept an initialization statement, it's common to see one used to set up a local variable.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

In the Go libraries, you'll find that when an `if` statement doesn't flow into the next statement—that is, the body ends in `break`, `continue`, `goto`, or `return`—the unnecessary `else` is omitted.

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```

This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in `return` statements, the resulting code needs no `else` statements.

```
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Redeclaration and reassignment

An aside: The last example in the previous section demonstrates a detail of how the `:=` short declaration form works. The declaration that calls `os.Open` reads,

```
f, err := os.Open(name)
```

This statement declares two variables, `f` and `err`. A few lines later, the call to `f.Stat` reads,

```
d, err := f.Stat()
```

which looks as if it declares `d` and `err`. Notice, though, that `err` appears in both statements. This duplication is legal: `err` is declared by the first statement, but only *re-assigned* in the second. This means that the call to `f.Stat` uses the existing `err` variable declared above, and just gives it a new value.

In a `:=` declaration a variable `v` may appear even if it has already been declared, provided:

- this declaration is in the same scope as the existing declaration of `v` (if `v` is already declared in an outer scope, the declaration will create a new variable §),
- the corresponding value in the initialization is assignable to `v`, and
- there is at least one other variable in the declaration that is being declared anew.

This unusual property is pure pragmatism, making it easy to use a single `err` value, for example, in a long `if-else` chain. You'll see it used often.

§ It's worth noting here that in Go the scope of function parameters and return values is the same as the function body, even though they appear lexically outside the braces that enclose the body.

For

The Go `for` loop is similar to—but not the same as—C's. It unifies `for` and `while` and there is no `do-while`. There are three forms, only one of which has semicolons.

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
```