Originally published at https://rakyll.org/custom-profiles/.

Go provides several pprof profiles out of the box to gather profiling data from Go programs.

The builtin profiles provided by the runtime/pprof package:

- **profile**: CPU profile determines where a program spends its time while actively consuming CPU cycles (as opposed while sleeping or waiting for I/O).
- **heap**: Heap profile reports the currently live allocations; used to monitor current memory usage or check for memory leaks.
- **threadcreate**: Thread creation profile reports the sections of the program that lead the creation of new OS threads.
- **goroutine**: Goroutine profile report the stack traces of all current goroutines.
- **block**: Block profile show where goroutines block waiting on synchronization primitives (including timer channels). Block profile is not enabled by default; use runtime.SetBlockProfileRate to enable it.
- **mutex**: Mutex profile reports the lock contentions. When you think your CPU is not fully utilized due to a mutex contention, use this profile. Mutex profile is not enabled by default, see runtime.SetMutexProfileFraction to enable.

Additional to the builtin profiles, runtime/pprof package allows you to export your custom profiles, and instrument your code to record execution stacks that contributes to this profile.

Imagine we have a blob server, and we are writing a Go client for it. And our users want to be able to profile the opened blobs on the client. We can create a profile and record the events of blob opening and closing, so the user can tell how many open blobs they are at any time.

Here is a blobstore package that allows you to open some blobs. We will create a new custom profile and start recording execution stacks that contributes to opening of blobs:

```
package blobstore

import "runtime/pprof"

var openBlobProfile = pprof.NewProfile("blobstore.Open")

// Open opens a blob, all opened blobs need
// to be closed when no longer in use.
func Open(name string) (*Blob, error) {
    blob := &Blob{name: name}
    // TODO: Initialize the blob...
```

```
    openBlobProfile.Add(blob, 2) // add the current execution stack to the profile
    return blob, nil
}
```

And once users want to close the blob, we need to remove the execution stack associated with the current blob from the profile:

```
// Close closes the blob and frees the
// underlying resources.
func (b *Blob) Close() error {
    // TODO: Free other resources.
    openBlobProfile.Remove(b)
    return nil
}
```

And now, from the programs using this package, we should be able to retrieve `blobstore.Open` profile data and use our daily pprof tools to examine and visualize them.

Let's write a small main program than opens some blobs:

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    _ "net/http/pprof" // as a side effect, registers the pprof endpoints.
    "time"

    "myproject.org/blobstore"
)

func main() {
    for i := 0; i < 1000; i++ {
        name := fmt.Sprintf("task-blob-%d", i)
        go func() {
            b, err := blobstore.Open(name)
            if err != nil {
                // TODO: Handle error.
            }
            defer b.Close()

            // TODO: Perform some work, write to the blob.
        }()
    }
    http.ListenAndServe("localhost:8888", nil)
```

```
}
```

Start the server, then use go tool to read and visualize the profile data:

```
$ go tool pprof http://localhost:8888/debug/pprof/blobstore.Open
(pprof) top
Showing nodes accounting for 800, 100% of 800 total
      flat  flat%   sum%        cum   cum%
       800   100%   100%        800   100%  main.main.func1 /Users/jbd/src/hello/main.go
```

You will see that there are 800 open blobs and all openings are coming from main.main.func1. In this small example, there is nothing more to see, but in a complex server you can examine the hottest spots that works with an open blob and find out bottlenecks or leaks.