Cloud Profiler

Quickstart: Exploring Profiler

This experiment shows you how to set up and use Cloud Profiler. You download a sample Go program, run it with profiling enabled, and then use the Profiler interface to explore the captured data.

Before you begin

1. In the Google Cloud Console, on the project selector page, select or create a Google Cloud project.

   Note: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

   Go to project selector

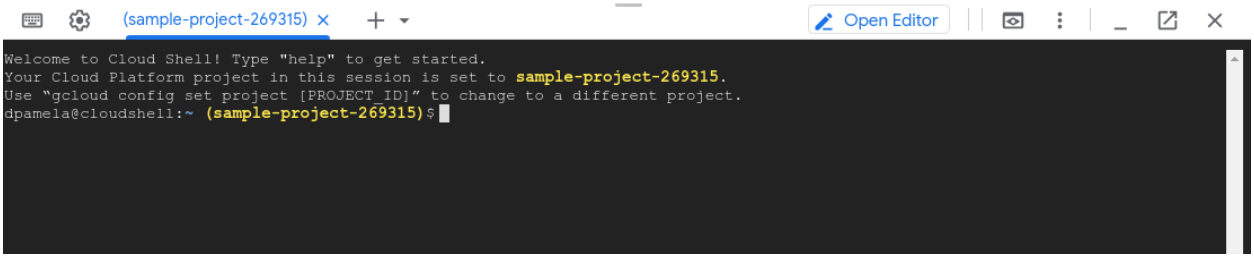
2. To enable the Cloud Profiler API for your project, in the Google Cloud Console navigation pane, click **Profiler**, or use the following button:

   Go to Profiler

3. To open the Cloud Shell, in the Google Cloud Console toolbar, click **Activate Cloud Shell**:

   After a few moments, a Cloud Shell session opens inside the Google Cloud Console:

Get a program to profile

The sample program, main.go, is in the golang-samples repository on GitHub. To get it, retrieve the package of Go samples:

```
git clone https://github.com/GoogleCloudPlatform/golang-samples.git
```

The package retrieval takes a few moments to complete.

Profile the code

Go to the directory of sample code for Profiler in the retrieved package:

```
cd golang-samples/profiler/profiler_quickstart
```

The main.go program creates a CPU-intensive workload to provide data to the profiler.

```
func main() {
        err := profiler.Start(profiler.Config{
                Service:              "hello-profiler",
                NoHeapProfiling:      true,
                NoAllocProfiling:     true,
                NoGoroutineProfiling: true,
                DebugLogging:         true,
                // ProjectID must be set if not running on GCP.
                // ProjectID: "my-project",
        })
        if err != nil {
                log.Fatalf("failed to start the profiler: %v", err)
        }
        busyloop()
}
```

Start the program and leave it running:

```
 go run main.go
```

This program is designed to load the CPU as it runs. It is configured to use Profiler, which collects profiling data from the program as it runs and periodically saves it.

A few seconds after you start the program, you see the message profiler has started. In about a minute, two more messages are displayed:

```
go: downloading golang.org/x/sys v0.0.0-20211124211545-fe61309f8881
go: downloading golang.org/x/oauth2 v0.0.0-20211104180415-d3ed0bb246c8
```

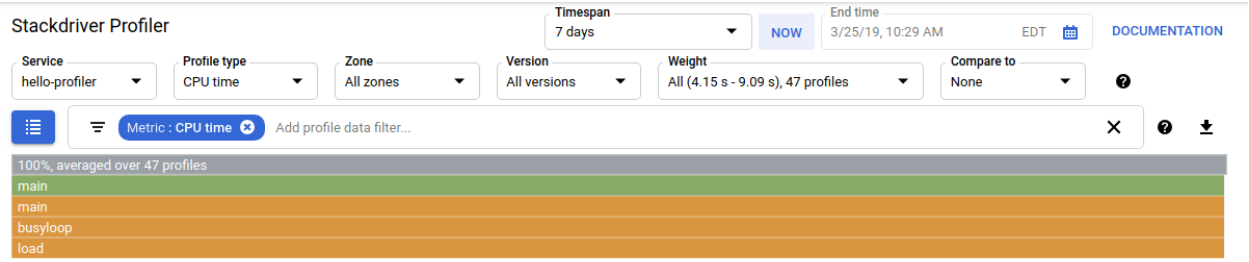
```
go: downloading go.opencensus.io v0.23.0
go: downloading golang.org/x/text v0.3.7
go: downloading github.com/google/go-cmp v0.5.6
go: downloading github.com/golang/groupcache v0.0.0-20210331224755-
41bb18bfe9da
Cloud Profiler: 2022/01/12 15:53:53 Cloud Profiler Go Agent version:
20201104
Cloud Profiler: 2022/01/12 15:53:53 profiler has started
Cloud Profiler: 2022/01/12 15:53:53 creating a new profile via profiler
service
```

These messages indicate that a profile was created and uploaded to your Cloud Storage project. The program continues to emit the last two messages, about one time per minute, for as long as it runs.

If you receive a permission denied error message after starting the service, see Errors for your Google Cloud project configuration.

Profiler interface

A few moments after you start the application, Profiler displays the initial profile data. The interface offers an array of controls and a flame graph for exploring the profiling data:



In addition to time controls, there are options that let you choose the set of profile data to use. When you are profiling multiple applications, you use **Service** to select the origin of the profiled data. **Profile type** lets you choose the kind of profile data to display. **Zone name** and **Version** let you restrict display to data from Compute Engine zones or versions of the application. **Weight** lets you select profiles captured during peak resource consumption.

To refine how the flame graph displays the profiles you've selected to analyze, you add filters. In the previous screenshot, the filter bar filter_list shows one filter. This filter option is Metric and the filter value is CPU time.

Exploring the data

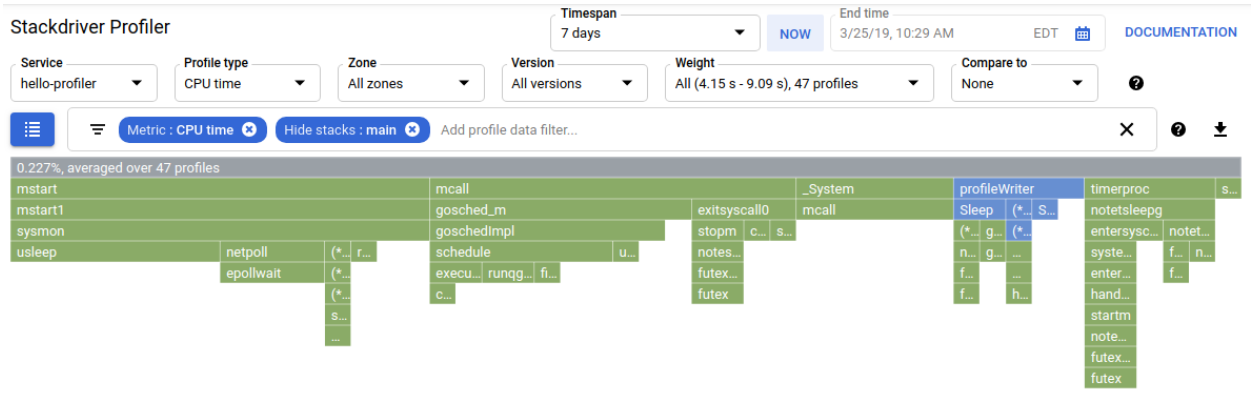
The flame graph displays the call stacks of the program. The flame graph represents each function with a frame. The width of the frame represents the proportion of resource consumption by that function. The top frame represents the entire program. This frame always shows 100% of the resource consumption. This frame also lists how many profiles are averaged together in this graph.

The sample program doesn't have a complicated set of call stacks; in the preceding screenshot, you see 5 frames:

- The gray frame represents the entire executable, which accounts for 100% of the resources being consumed.
- The green `main` frame is the Go `runtime.main`.
- The orange `main` frame is the `main` routine of the sample program.
- The orange `busyloop` frame is a routine called from the sample's `main`.
- The orange `main.load` frame is a routine called from the sample's `main`.

The filter selector lets you do things like filter out functions that match some name. For example, if there is a standard library of utility functions, you can remove them from the graph. You can also remove call stacks originating at a certain method or simplify the graph in other ways. The `main.go` application is simple, so there isn't much to filter out.

Even for a simple application, filters let you hide uninteresting frames so that you can more clearly view interesting frames. For example, in the profiling screenshot for the sample code, the gray frame is slightly larger than the first `main` frame under it. Why? Is there something else going on that's not immediately apparent because the `main` call stack consumes such an overwhelming percentage of the resources? To view what is occurring outside of the application's `main` routine, add a filter that hides the call stack of the `main` routine. Only 0.227% of the resource consumption occurs outside of `main`:



See Using the Profiler interface for much more information on filtering and other ways to explore the profiling data.

Note: If the Profiler agent hasn't uploaded any profiles when you start the interface, Profiler displays the message `No data to show`. The message is automatically replaced with the Profiler interface after profile data is available.

Cleanup

All we have to do is stop the program we have running for the profiler.

Congratulations!

In this experiment, we worked with Cloud Profiler

Explore the how-to guides for more detail

https://cloud.google.com/profiler/docs/how-to?_ga=2.243799365.-1776565748.1641784757