Boomer

Aug 05, 2021

Contents

CHAPTER 1

What is Boomer?

Boomer is a golang library and works with Locust.

Using goroutines to run you code concurrently will outperform the gevent implementation in Locust. That's why I created this project.

Remember, use it as a library, not a general-purpose benchmarking tool.

1.1 Features

- **Write user test scenarios in golang**

  Just put you test scenarios in a normal function, boomer will spawn goroutines to run the function for many times to produce stress.

- **Build-in rate limit support**

  You can put rate limit on each boomer instance, which is useful when you just want to evaluate if the target is able to handle specific requests per second, instead of exhausting the target.

- **Different output destination**

  You can write you own output implementation to collect the test result.

CHAPTER 2

Installation

Boomer can be installed and updated with the “go get” command.

install:

```
$ go get github.com/myzhan/boomer
```

update:

```
$ go get -u github.com/myzhan/boomer
```

If you want to point to a particular revision of boomer, you should use a dependency management tool like dep or go module.

2.1 The goczmq dependency

Locust uses the zeromq protocol, so boomer depends on a zeromq client. Boomer uses gomq by default, which is a pure Go implementation.

Because of the instability of gomq, you can switch to goczmq.

Once install goczmq successfully, then you can build with goczmq instead of gomq.

```
$ go build -tags 'goczmq' your-code.go
```

CHAPTER 3

Quickstart

3.1 Code

This is a quick example of writing test scenarios with boomer.

```go
package main

import "time"
import "github.com/myzhan/boomer"

func foo(){
    start := time.Now()
    time.Sleep(100 * time.Millisecond)
    elapsed := time.Since(start)

    /*
    Report your test result as a success
    */
    boomer.RecordSuccess("http", "foo", elapsed.Nanoseconds()/int64(time.Millisecond),
↪ int64(10))
}

func bar(){
    start := time.Now()
    time.Sleep(100 * time.Millisecond)
    elapsed := time.Since(start)

    /*
    Report your test result as a failure
    */
    boomer.RecordFailure("udp", "bar", elapsed.Nanoseconds()/int64(time.Millisecond),
↪"udp error")
}
```

(continues on next page)

(continued from previous page)

```go
func main(){
    task1 := &boomer.Task{
        Name: "foo",
        // The weight is used to distribute goroutines over multiple tasks.
        Weight: 10,
        Fn: foo,
    }

    task2 := &boomer.Task{
        Name: "bar",
        Weight: 20,
        Fn: bar,
    }

    boomer.Run(task1, task2)
}
```

Here we define two tasks, task1 reports a success to boomer every 100 milliseconds, and meanwhile task2 reports a failure. The weight of task1 is 10, and the weight of task2 is 20, if the locust master asks boomer to spawn 30 users, then task1 will get 10 goroutines to run and task2 will get 20. The numbers of users can be specified in the Web UI.

3.2 Test

Because task1 is named foo and tasks2 is named bar, you can run them without connecting to the master.

```
$ go run --run-tasks foo,bar
```

In this case, task1 and task2 will be run for one time with no output.

You can add logs to ensure your tasks running correctly.

3.3 Build

```
$ go build -o you-code you-code.go
```

3.4 Run

1. Start the locust master with the included dummy.py.

```
$ locust --master -f dummy.py
```

So far, dummy.py is necessary when starting a master, because locust needs such a file.

Don't worry, dummy.py has nothing to do with your test.

2. Start you test program.

```
$ chmod +x ./you-code && ./you-code
```

Note: To see all available options type: `you-code --help`

3.5 Open up Locust's web interface

Once you've started Locust and boomer, you should open up a browser and point it to http://127.0.0.1:8089 (if you are running Locust locally).

CHAPTER 4

Running Mode

Currently, boomer has two running mode, standalone and distributed.

4.1 Distributed

When running in distributed mode, boomer will connect to a locust master and running as a slave. It's the default running mode of boomer.

4.2 Standalone

When running in standalone mode, boomer doesn't need to connect to a locust master and start testing immediately.

By default, the standalone mode works with a ConsoleOutput, which will print the test result to the console, you can write you own output and add more by calling boomer.AddOutput().

```go
package main

import (
	"log"
	"time"

	"github.com/myzhan/boomer"
)

func foo() {
	start := time.Now()
	time.Sleep(100 * time.Millisecond)
	elapsed := time.Since(start)

	// Report your test result as a success, if you write it in python, it will looks like this
```

(continues on next page)

(continued from previous page)

```
        // events.request_success.fire(request_type="http", name="foo", response_
↪time=100, response_length=10)
        globalBoomer.RecordSuccess("http", "foo", elapsed.Nanoseconds()/int64(time.
↪Millisecond), int64(10))
}

var globalBoomer *boomer.Boomer

func main() {
        log.SetFlags(log.LstdFlags | log.Lshortfile)

        task1 := &boomer.Task{
                Name:    "foo",
                Weight: 10,
                Fn:      foo,
        }

        numClients := 10
        spawnRate := 10
        globalBoomer = boomer.NewStandaloneBoomer(numClients, spawnRate)
        globalBoomer.Run(task1)
}
```

CHAPTER 5

Custom output

You can write you own output to deal with the test result.

```
type Output interface {
    OnStart()
    OnEvent(data map[string]interface{})
    OnStop()
}
```

All the OnXXX function will be call in a separated goroutine, just in case some output will block. But it will wait for all outputs return to avoid data lost.

It works like:

```
wg := sync.WaitGroup{}
wg.Add(len(outputs))
for _, output := range outputs {
    go func(o Output) {
        o.OnXXXX()
        wg.Done()
    }(output)
}
wg.Wait()
```

5.1 OnStart

OnStart will be call before the test starts.

5.2 OnEvent

By default, each output receive stats data from runner every three seconds. OnEvent is responsible for dealing with the data.

Don’t write to the origin data! Because all outputs share the same reference.

5.3 OnStop

OnStop will be called before the test ends. If you are writing to a disk file, it’s time to flush.

CHAPTER 6

API

See godoc.

CHAPTER 7

Options

For convenience, boomer supports several command line options.

Since it may conflict with user's code, I'm planning to remove this feature and allow users to set options programmatically.

7.1 `--master-host`

Host or IP address of locust master for distributed load testing.

Defaults to 127.0.0.1.

7.2 `--master-port`

The port to connect to that is used by the locust master for distributed load testing.

Defaults to 5557.

7.3 `--run-tasks`

Run tasks without connecting to the master, multiply tasks is separated by comma.

7.4 `--max-rps`

Max RPS that boomer can generate, disabled by default.

–max-rps=100 means the max RPS is limit to 100.

Defaults to 0.

7.5 `--request-increase-rate`

Request increase rate, disabled by default.

–request-increase-rate=100/1s means the threshold will ramp up.

7.6 `--cpu-profile`

Enable CPU profiling and specify a file path to save the result.

7.7 `--cpu-profile-duration`

CPU profile duration.

The timer will start when the process starts.

Defaults to 30 seconds.

7.8 `--mem-profile`

Enable memory profiling and specify a file path to save the result.

7.9 `--mem-profile-duration`

Memory profile duration.

The timer will start when the process starts.

Defaults to 30 seconds.

CHAPTER 8

Profiling

You may think there are bottlenecks in your load generator, don't hesitate to do profiling.

Both CPU and memory profiling are supported.

It's not suggested to run CPU profiling and memory profiling at the same time.

8.1 CPU Profiling

```
# 1. run locust master.
# 2. run boomer with cpu profiling for 30 seconds.
$ go run main.go -cpu-profile cpu.pprof -cpu-profile-duration 30s
# 3. start test in the WebUI.
# 4. run pprof.
$ go tool pprof cpu.pprof
Type: cpu
Time: Nov 14, 2018 at 8:04pm (CST)
Duration: 30.17s, Total samples = 12.07s (40.01%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) web
```

8.2 Memory Profiling

```
# 1. run locust master.
# 2. run boomer with memory profiling for 30 seconds.
$ go run main.go -mem-profile mem.pprof -mem-profile-duration 30s
# 3. start test in the WebUI.
# 4. run pprof and try 'go tool pprof --help' to learn more.
$ go tool pprof -alloc_space mem.pprof
Type: alloc_space
Time: Nov 14, 2018 at 8:26pm (CST)
```

(continues on next page)

(continued from previous page)

```
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search