Inside the Go Playground

Andrew Gerrand
12 December 2013

Introduction

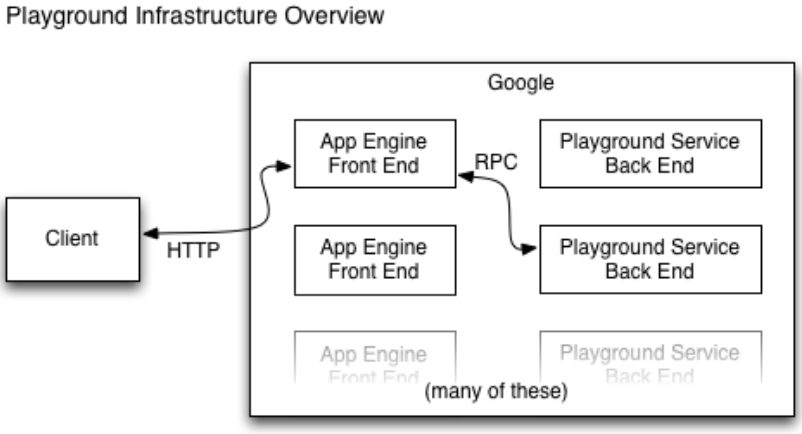
In September 2010 we introduced the Go Playground, a web service that compiles and executes arbitrary Go code and returns the program output.

If you're a Go programmer then you have probably already used the playground by using the Go Playground directly, taking the Go Tour, or running executable examples from the Go documentation.

You may also have used it by clicking one of the "Run" buttons in a slide deck on talks.golang.org or a post on this very blog (such as the recent article on Strings).

In this article we will take a look at how the playground is implemented and integrated with these services. The implementation involves a variant operating system environment and runtime and our description here assumes you have some familiarity with systems programming using Go.

Overview



The playground service has three parts:

- A back end that runs on Google's servers. It receives RPC requests, compiles the user program using the gc tool chain, executes the user program, and returns the program output (or compilation errors) as the RPC response.
- A front end that runs on Google App Engine. It receives HTTP requests from the client and makes corresponding RPC requests to the back end. It also does some caching.
- A JavaScript client that implements the user interface and makes HTTP requests to the front end.

The back end

The back end program itself is trivial, so we won't discuss its implementation here. The interesting part is how we safely execute arbitrary user code in a secure environment while still providing core functionality such as time, the network, and the file system.

To isolate user programs from Google's infrastructure, the back end runs them under Native Client (or "NaCl"), a technology developed by Google to permit the safe execution of x86 programs inside web browsers. The back end uses a special version of the gc tool chain that generates NaCl executables.

(This special tool chain was merged into Go 1.3. To learn more, read the design document.)

NaCl limits the amount of CPU and RAM a program may consume, and it prevents programs from accessing the network or file system. This presents a problem, however. Go's concurrency and networking support are among its key strengths, and access to the file system is vital for many programs. To demonstrate concurrency effectively we need time, and to demonstrate networking and the file system we obviously need a network and a file system.

Although all these things are supported today, the first version of the playground, launched in 2010, had none of them. The current time was fixed at 10 November 2009, `time.Sleep` had no effect, and most functions of the `os` and `net` packages were stubbed out to return an `EINVALID` error.

A year ago we implemented fake time in the playground, so that programs that sleep would behave correctly. A more recent update to the playground introduced a fake network stack and a fake file system, making the playground's tool chain similar to a normal Go tool chain. These facilities are described in the following sections.

Faking time

Playground programs are limited in the amount of CPU time and memory they can use, but they are also restricted in how much real time they can use. This is because each running program consumes resources on the back end and any stateful infrastructure between it and the client. Limiting the run time of each playground program makes our service more predictable and defends us against denial of service attacks.

But these restrictions become stifling when running code that uses time. The Go Concurrency Patterns talk demonstrates concurrency with examples that use timing functions like `time.Sleep` and `time.After`. When run under early versions of the playground, these programs' sleeps would have no effect and their behavior would be strange (and sometimes wrong).

By using a clever trick we can make a Go program *think* that it is sleeping, when really the sleeps take no time at all. To explain the trick we first need to understand how the scheduler manages sleeping goroutines.

When a goroutine calls `time.Sleep` (or similar) the scheduler adds a timer to a heap of pending timers and puts the goroutine to sleep. Meanwhile, a special timer goroutine manages that heap. When the timer goroutine starts it tells the scheduler to wake it when the next pending timer is ready to fire and then sleeps. When it wakes up it checks which timers have expired, wakes the appropriate goroutines, and goes back to sleep.

The trick is to change the condition that wakes the timer goroutine. Instead of waking it after a specific time period, we modify the scheduler to wait for a deadlock; the state where all goroutines are blocked.

The playground version of the runtime maintains its own internal clock. When the modified scheduler detects a deadlock it checks whether any timers are pending. If so, it advances the internal clock to the trigger time of the earliest timer and then wakes the timer goroutine. Execution continues and the program believes that time has passed, when in fact the sleep was nearly instantaneous.

These changes to the scheduler can be found in `proc.c` and `time.goc`.

Fake time fixes the issue of resource exhaustion on the back end, but what about the program output? It would be odd to see a program that sleeps run to completion correctly without taking any time.

The following program prints the current time each second and then exits after three seconds. Try running it.

```
func main() {
    stop := time.After(3 * time.Second)
    tick := time.NewTicker(1 * time.Second)
    defer tick.Stop()
    for {
        select {
        case <-tick.C:
            fmt.Println(time.Now())
        case <-stop:
            return
        }
    }
}
```

Run

How does this work? It is a collaboration between the back end, front end, and client.

We capture the timing of each write to standard output and standard error and provide it to the client. Then the client can "play back" the writes with the correct timing, so that the output appears just as if the program were running locally.

The playground's `runtime` package provides a special `write`function that includes a small "playback header" before each write. The playback header comprises a magic string, the current time, and the length of the write data. A write with a playback header has this structure:

```
0 0 P B <8-byte time> <4-byte data length> <data>
```

The raw output of the program above looks like this:

```
\x00\x00PB\x11\x74\xef\xed\xe6\xb3\x2a\x00\x00\x00\x00\x1
11-10 23:00:01 +0000 UTC
\x00\x00PB\x11\x74\xef\xee\x22\x4d\xf4\x00\x00\x00\x00\x1
11-10 23:00:02 +0000 UTC
\x00\x00PB\x11\x74\xef\xee\x5d\xe8\xbe\x00\x00\x00\x00\x1
11-10 23:00:03 +0000 UTC
```

The front end parses this output as a series of events and returns a list of events to the client as a JSON object:

```
{
    "Errors": "",
    "Events": [
        {
            "Delay": 1000000000,
            "Message": "2009-11-10 23:00:01 +0000
UTC\n"
        },
        {
            "Delay": 1000000000,
            "Message": "2009-11-10 23:00:02 +0000
UTC\n"
        },
        {
            "Delay": 1000000000,
            "Message": "2009-11-10 23:00:03 +0000
UTC\n"
        }
    ]
}
```

The JavaScript client (running in the user's web browser) then plays back the events using the provided delay intervals. To the user it appears that the program is running in real time.

Faking the file system

Programs built with the Go's NaCl tool chain cannot access the local machine's file system. Instead, the `syscall` package's file-related functions (`Open`, `Read`, `Write`, and so on) operate on an in-memory file system that is implemented by the `syscall` package itself. Since package `syscall` is the interface between the Go code and the operating system kernel, user programs see the file system exactly the same way as they would a real one.

The following example program writes data to a file, and then copies its contents to standard output. Try running it. (You can edit it, too!)

```
func main() {
    const filename = "/tmp/file.txt"

    err := ioutil.WriteFile(filename, []byte("Hello,
file system\n"), 0644)
    if err != nil {
        log.Fatal(err)
    }

    b, err := ioutil.ReadFile(filename)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Printf("%s", b)
}
```

Run

When a process starts, the file system is populated with some devices under `/dev` and an empty `/tmp` directory. The program can manipulate the file system as usual, but when the process exits any changes to the file system are lost.

There is also a provision to load a zip file into the file system at init time (see `unzip_nacl.go`). So far we have only used the unzip facility to provide the data files required to run the standard library tests, but we intend to provide playground programs with a set of

files that can be used in documentation examples, blog posts, and the Go Tour.

The implementation can be found in the `fs_nacl.go` and `fd_nacl.go` files (which, by virtue of their `_nacl` suffix, are built into package `syscall` only when `GOOS` is set to `nacl`).

The file system itself is represented by the `fsys` struct, of which a global instance (named `fs`) is created during init time. The various file-related functions then operate on `fs` instead of making the actual system call. For instance, here is the `syscall.Open` function:

```
func Open(path string, openmode int, perm uint32) (fd
int, err error) {
    fs.mu.Lock()
    defer fs.mu.Unlock()
    f, err := fs.open(path, openmode,
perm&0777|S_IFREG)
    if err != nil {
        return -1, err
    }
    return newFD(f), nil
}
```

File descriptors are tracked by a global slice named `files`. Each file descriptor corresponds to a `file` and each `file` provides a value that implements the `fileImpl` interface. There are several implementations of the interface:

- regular files and devices (such as `/dev/random`) are represented by `fsysFile`,
- standard input, output, and error are instances of `naclFile`, which uses system calls to interact with the actual files (these are a playground program's only way to interact with the outside world),
- network sockets have their own implementation, discussed in the next section.

Faking the network

Like the file system, the playground's network stack is an in-process fake implemented by the `syscall` package. It permits playground projects to use the loopback interface (`127.0.0.1`). Requests to other hosts will fail.

For an executable example, run the following program. It listens on a TCP port, waits for an incoming connection, copies the data from that connection to standard output, and exits. In another goroutine, it makes a connection to the listening port, writes a string to the connection, and closes it.

```
func main() {
    l, err := net.Listen("tcp", "127.0.0.1:4000")
    if err != nil {
        log.Fatal(err)
    }
    defer l.Close()

    go dial()

    c, err := l.Accept()
    if err != nil {
        log.Fatal(err)
    }
    defer c.Close()

    io.Copy(os.Stdout, c)
}

func dial() {
    c, err := net.Dial("tcp", "127.0.0.1:4000")
    if err != nil {
        log.Fatal(err)
    }
    defer c.Close()
    c.Write([]byte("Hello, network\n"))
}
```

Run

The interface to the network is more complex than the one for files, so the implementation of the fake network is larger and more complex than the fake file system. It must simulate read and write timeouts, different address types and protocols, and so on.

The implementation can be found in `net_nacl.go`. A good place to start reading is `netFile`, the network socket implementation of the `fileImpl` interface.

The front end

The playground front end is another simple program (shorter than 100 lines). It receives HTTP requests from the client, makes RPC requests to the back end, and does some caching.

The front end serves an HTTP handler at `https://golang.org/compile`. The handler expects a POST request with a `body` field (the Go program to run) and an optional `version` field (for most clients this should be `"2"`).

When the front end receives a compilation request it first checksmemcache to see if it has cached the results of a previous compilation of that source. If found, it returns the cached response. The cache prevents popular programs such as those on the Go home page from overloading the back ends. If there is no cached response, the front end makes an RPC request to the back end, stores the response in memcache, parses the playback events, and returns a JSON object to the client as the HTTP response (as described above).

The client

The various sites that use the playground each share some common JavaScript code for setting up the user interface (the code and output boxes, the run button, and so on) and communicating with the playground front end.

This implementation is in the file `playground.js` in the `go.tools`repository, which can be imported from the`golang.org/x/tools/godoc/static` package. Some of it is clean and some is a bit crufty, as it is the result of consolidating several divergent implementations of the client code.

The `playground` function takes some HTML elements and turns them into an interactive playground widget. You should use this function if you want to put the playground on your own site (see 'Other clients' below).

The `Transport` interface (not formally defined, this being JavaScript) abstracts the user interface from the means of talking to the web front end. `HTTPTransport` is an implementation of `Transport` that speaks the HTTP-based protocol described earlier.`SocketTransport` is another implementation that speaks WebSocket (see 'Playing offline' below).

To comply with the same-origin policy, the various web servers (godoc, for instance) proxy requests to `/compile` through to the playground service at `https://golang.org/compile`. The common `golang.org/x/tools/playground` package does this proxying.

Playing offline

Both the Go Tour and the Present Tool can be run offline. This is great for people with limited internet connectivity or presenters at conferences who cannot (and *should* not) rely on a working internet connection.

To run offline, the tools run their own version of the playground back end on the local machine. The back end uses a regular Go tool chain with none of the aforementioned modifications and uses a WebSocket to communicate with the client.

The WebSocket back end implementation can be found in the`golang.org/x/tools/playground/socket` package. The Inside Present talk discusses this code in detail.

Other clients

The playground service is used by more than just the official Go project (Go by Example is one other instance) and we are happy for you to use it on your own site. All we ask is that you contact us first, use a unique user agent in your requests (so we can identify you), and that your service is of benefit to the Go community.

Conclusion

From godoc to the tour to this very blog, the playground has become an essential part of our Go documentation story. With the recent additions of the fake file system and network stack we are excited to expand our learning materials to cover those areas.

But, ultimately, the playground is just the tip of the iceberg. With Native Client support scheduled for Go 1.3, we look forward to seeing what the community can do with it.