Shutting Down Applications Cleanly

KELSEY HIGHTOWER

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration and distributed systems.
kelsey.hightower@gmail.com

Developers around the world are building new applications at blazing speeds, and with the growing adoption of microservices and continuous delivery, people are shipping applications faster than ever. Most developers focus on getting applications out the door and deployed to production, which can mean neglecting critical parts of an application's life cycle—the shutdown process.

It's easy to understand why the shutdown process is often neglected. The main goal is to deploy applications and keep them running around the clock, not shutting them down. However, the reality is that modern applications shut down as often as they start up thanks to continuous delivery and better deployment tools. As a developer, it's your job to ensure applications are robust and not only start up correctly but also shut down cleanly. Clean shutdowns are the key to zero-downtime deployments and keeping your production systems in a consistent state. Applications that exit abruptly can wreak havoc on system resources caused by leaving network connections open and partial writes to file systems.

Just Give Me the Signal and I'll Do the Rest

On UNIX systems, applications are sent an asynchronous signal to indicate when an application should terminate. The most common signals sent to initiate the shutdown process are SIGINT, SIGTERM, and SIGKILL. SIGINT is sent when the user wants to interrupt a process. For example, say you're running a ping process in the foreground like this:

```
$ ping usenix.org
PING usenix.org (50.56.53.173): 56 data bytes
64 bytes from 50.56.53.173: icmp_seq=0 ttl=51 time=66.741 ms
64 bytes from 50.56.53.173: icmp_seq=1 ttl=51 time=65.001 ms
^C
--- usenix.org ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 65.001/65.871/66.741/0.870 ms
```

Notice how pressing the Ctrl-C key combination causes the ping process to terminate. Also notice how the ping process was able to print a few additional lines before terminating. This works because the ping process caught the SIGINT signal, which made it stop pinging usenix.org and print the results of the ping session.

The SIGTERM signal is sent to a process in order to request that the process terminates. Both the SIGTERM and SIGINT signals can be caught by a running process and cause it to shut down; this is not a hard requirement as a process is free to ignore these signals. The SIGKILL signal is sent to a process to cause it to terminate immediately, and unlike the SIGTERM and SIGINT signals, your process is not given an opportunity to clean up and will be killed—yeah, SIGKILL is brutal. The SIGKILL signal is normally sent after sending a SIGTERM signal and waiting some amount of time before considering the process hung. This is what we want to avoid, so we must make sure the cleanup process happens as quickly as possible.

Don't Forget to Clean Up

Once a termination signal is received it's the application's responsibility to clean up any open network connections, flush pending writes, or complete outstanding client requests before terminating. Let's write a simple Web application that demonstrates how clean shutdowns work in practice. The Go standard library provides everything required to catch UNIX signals, but we'll use a third-party library called manners to simplify the handling of incoming HTTP requests during the shutdown process.

Review the following code sample and save it to a file named main.go.

```
package main
import (
    "log"
    "net/http"
    "os"
    "os/signal"
    "syscall"
    "github.com/braintree/manners"
)

func main() {
    log.Println("Starting up...")

    http.HandleFunc("/", func(w http.ResponseWriter, r
                *http.Request) {
        log.Printf("%s %s - %s", r.Method, r.URL.Path,
                r.UserAgent())
        w.Write([]byte("Hello USENIX!\n"))
    })

    // Catch signals in the background using
    // an anonymous go routine.
    go func() {
        signalChan := make(chan os.Signal, 1)
        signal.Notify(signalChan, syscall.SIGINT,
                syscall.SIGTERM)

        // This blocks until this process receives a SIGINT
        // or SIGTERM signal.
        <-signalChan
        log.Println("Shutting down...")

        // stop accepting new requests and begin shutting down.
        manners.Close()
    }()

    // The call to ListenAndServe() blocks until an error is
    // returned or the manners.Close() function is called.
    err := manners.ListenAndServe(":8080",
                http.DefaultServeMux)
    if err != nil {
        log.Fatal(err)
    }
    log.Println("Shutdown complete.")
}
```

The source code can also be cloned from https://github.com/kelseyhightower/hello-usenix:

```
$ git clone https://github.com/kelseyhightower/hello-usenix.git
$ cd hello-usenix
```

Next, fetch the manners package, which provides a wrapper for Go's standard HTTP server and ensures all active HTTP requests have completed before the HTTP server shuts down.

```
$ go get github.com/braintree/manners
```

Finally, build the hello-usenix server using the go build command:

```
$ go build -o hello-usenix main.go
```

At this point you should have a hello-usenix binary in the current directory that can be used to examine a clean shutdown process in action. The remainder of this tutorial will require three separate terminals where commands can be executed.

Terminal #1

Start the hello-usenix server in the foreground:

```
$ ./hello-usenix
2016/07/03 15:37:30 Starting up...
```

Terminal #2

Use curl to make HTTP requests to the hello-usenix service:

```
$ while true; do curl http://127.0.0.1:8080; sleep 1; done
Hello USENIX!
Hello USENIX!
…
```

Terminal #3

Grab the process ID (PID) of the running hello-usenix process using the ps command:

```
$ ps -u `whoami` | grep hello-usenix
55211 ttys003    0:00.03 ./hello-usenix
```

Use the kill command to send the SIGTERM signal to the hello-usenix process using the correct process ID:

```
$ kill 55211
```

By default the kill command sends the SIGTERM signal to the running process specified by the given process ID (PID). Once the SIGTERM signal reaches the hello-usenix process, the shutdown process will begin by rejecting new HTTP requests, which

will cause the curl command running in Terminal #2 to return connection-refused errors:

Terminal #2

```
$ while true; do curl http://127.0.0.1:8080; sleep .2; done
Hello USENIX!
Hello USENIX!
...
curl: (7) Failed to connect to 127.0.0.1 port 8080: Connection
refused
```

Although handling signals and performing a clean shutdown will give your servers the opportunity to tidy up, it won't prevent clients from continuing to send your services requests. In the case of the hello-usenix application, a front-end load balancer can be used to detect when an instance of hello-usenix is no longer accepting HTTP requests and route incoming requests to another instance.

Once the the final HTTP request has completed, the hello-usenix server will print the "Shutdown complete" log message and terminate:

Terminal #1

```
$ ./hello-usenix
2016/07/03 20:42:06 Starting up...
2016/07/03 20:43:03 GET / - curl/7.43.0
2016/07/03 20:43:04 GET / - curl/7.43.0
...
2016/07/03 20:44:24 Shutting down...
2016/07/03 20:44:24 Shutdown complete.
```

That's what a clean shutdown looks like. Catch a few signals and clean up before terminating. It looks so easy, right? Thanks to manners and the Go standard library, it is.

Conclusion

As a developer it's important to think about the complete application life cycle, and that includes the shutdown process. All applications should perform clean shutdowns and tidy up before terminating. Applications that get this right are much easier to manage and reduce the need for complex deployment scripts that attempt to protect clients from servers which accept requests that will never be processed or to clean up database connections left behind by misbehaving applications.