15 Plain sailing: Running Docker in production

This chapter covers

- Your options for logging container output
- Monitoring your running containers
- Managing your containers' resource usage
- Using Docker's capabilities to help manage traditional sysadmin tasks

In this chapter we're going to cover some of the subjects that come up when running in production. Running Docker in production is a big subject, and production use of Docker is still an evolving area. Many major tools are in the early stages of development and were changing as we wrote this book's first and second editions.

In this chapter we'll focus on showing you some of the key things you should consider when going from volatile environments to stable ones.

15.1 Monitoring

When you run Docker in production, one of the first things you'll want to consider is how to track and measure what your containers are up to. In this section you're going to learn how you can get an operational view of both your live containers' logging activity and their performance.

This is still a developing aspect of the Docker ecosystem, but some tools and techniques are emerging as more mainstream than others. We'll look at redirecting application logs to the host's syslog, at redirecting the output of the `docker logs` command to a single place, and at Google's container-oriented performance monitoring tool, cAdvisor.

TECHNIQUE 101 Logging your containers to the host's syslog

Linux distributions typically run a syslog daemon. This daemon is the server part of the system-logging functionality—applications send messages to this daemon, along with metadata like the importance of the message, and the daemon will decide where to save the message (if at all). This functionality is used by a range of applications, from network connection managers to the kernel itself dumping information if it encounters an error.

Because it's so reliable and widely used, it's reasonable for applications you write yourself to log to syslog. Unfortunately, this will stop working once you containerize your application (because there's no syslog daemon in containers, by default). If you do decide to start a syslog daemon in all of your containers, you'll need to go to each individual container to retrieve the logs.

PROBLEM

You want to capture syslogs centrally on your Docker host.

SOLUTION

Run a service container that acts as the syslog daemon for Docker containers.

The basic idea of this technique is to run a service container that runs a syslog daemon, and share the logging touchpoint (/dev/log) via the host's filesystem. The log itself can be retrieved by querying the syslog Docker container, and it's stored in a volume.

Figure 15.1 illustrates how /tmp/syslogdev on the host's filesystem can be used as a touchpoint for all syslogging taking place on containers on the host. The logging containers mount and write their syslog to that location, and the syslogger container collates all those inputs.

TIP The syslog daemon is a process that runs on a server, collecting and managing messages sent to a central file, which is normally a Unix domain socket. It generally uses /dev/log as a file to receive log messages, and it logs out to /var/log/syslog.

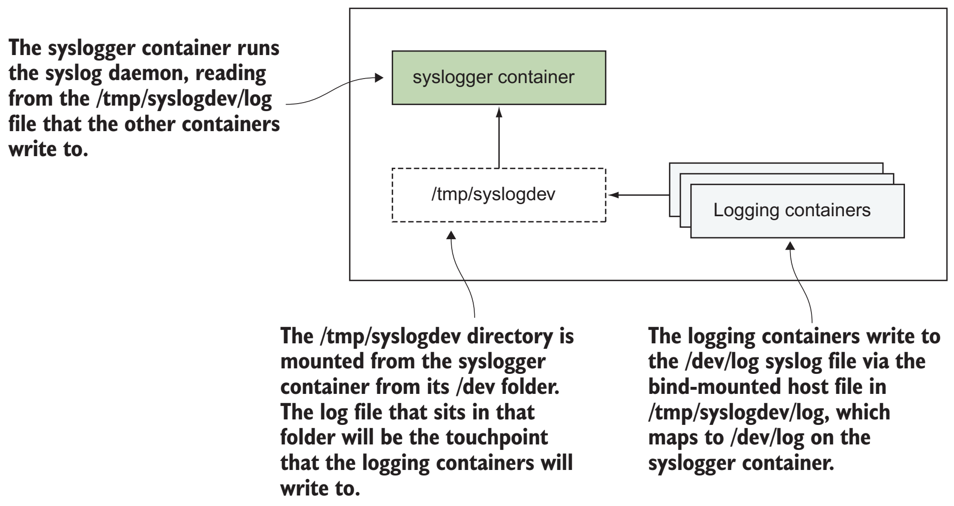
The syslogger container runs the syslog daemon, reading from the /tmp/syslogdev/log file that the other containers write to.
syslogger container
/tmp/syslogdev
Logging containers
The /tmp/syslogdev directory is mounted from the syslogger container from its /dev folder. The log file that sits in that folder will be the touchpoint that the logging containers will write to.
The logging containers write to the /dev/log syslog file via the bind-mounted host file in /tmp/syslogdev/log, which maps to /dev/log on the syslogger container.

Figure 15.1 Overview of centralized syslogging of Docker containers

The syslogger container can be created with this straightforward Dockerfile.

Listing 15.1 Building a syslogger container

```
FROM ubuntu:14.043
RUN apt-get update && apt-get install rsyslog
 VOLUME /dev
 VOLUME /var/log
 CMD rsyslogd -n
```

Creates the /dev volume to share with other containers

Installs the rsyslog package, which makes the rsyslogd daemon program available. The “r” stands for “reliable.”

Creates the /var/log volume to allow the syslog file to persist

Runs the rsyslogd process on startup

Next, you build the container, tagging it with the syslogger tag, and run it:

```
docker build -t syslogger .
docker run --name syslogger -d -v /tmp/syslogdev:/dev syslogger
```

You bind-mounted the container’s /dev folder to the host’s /tmp/syslogdev folder so you can mount a /dev/log socket into each container as a volume, as you’ll see shortly. The container will continue running in the background, reading any messages from the /dev/log file and handling them.

On the host, you’ll now see that the /dev folder of the syslog container has been mounted to the host’s /tmp/syslogdev folder:

```
$ ls -1 /tmp/syslogdev/
fd
full
fuse
kcore
```

```
log
null
ptmx
random
stderr
stdin
stdout
tty
urandom
zero
```

For this demonstration, we're going to start up 100 daemon containers that log their own starting order from 0 to 100 to the syslog, using the `logger` command. Then you'll be able to see those messages by running a `docker exec` on the host to look at the syslogger container's syslog file.

First, start up the containers.

Listing 15.2 Starting up the logger containers

```
for d in {1..100}
do
    docker run -d -v /tmp/syslogdev/log:/dev/log ubuntu logger hello_$d
done
```

The preceding volume mount links the container's syslog endpoint (/dev/log) to the host's /tmp/syslogdev/log file, which in turn is mapped to the syslogger container's /dev/log file. With this wiring, all syslog outputs are sent to the same file.

When that's complete, you'll see something similar to this (edited) output:

```
$ docker exec -ti syslogger tail -f /var/log/syslog
May 25 11:51:25 f4fb5d829699 logger: hello
May 25 11:55:15 f4fb5d829699 logger: hello_1
May 25 11:55:15 f4fb5d829699 logger: hello_2
May 25 11:55:16 f4fb5d829699 logger: hello_3
[...]
May 25 11:57:38 f4fb5d829699 logger: hello_97
May 25 11:57:38 f4fb5d829699 logger: hello_98
May 25 11:57:39 f4fb5d829699 logger: hello_99
```

You can use a modified `exec` command to archive these syslogs if you wish. For example, you could run the following command to get all logs for hour 11 on May 25th archived to a compressed file:

```
$ docker exec syslogger bash -c "cat /var/log/syslog | \
grep '^May 25 11'" | xz - > /var/log/archive/May25_11.log.xz
```

NOTE For the messages to show up in the central syslog container, your programs need to log to syslog. We ensure this here by running the `logger` command, but your applications should do the same for this to work. Most modern logging methods have a means to write to the locally visible syslog.

DISCUSSION

You may be wondering how you can distinguish between different containers' log messages with this technique. Here you have a couple of options. You can change the application's logging to output the hostname of the container, or you can see the next technique to have Docker do this heavy lifting for you.

> **NOTE** This technique looks similar to the next one, which uses a Docker syslog driver, but it's different. This technique keeps the output of containers' running processes as the output of the `docker logs` command, whereas the next one takes over the `logs` command, rendering this technique redundant.

TECHNIQUE 102 Logging your Docker logs output

As you've seen, Docker offers a basic logging system that captures the output of your container's start command. If you're a system administrator running many services off one host, it can be operationally tiresome to manually track and capture logs using the `docker logs` command on each container in turn.

In this technique, we're going to cover Docker's log driver feature. This lets you use the standard logging systems to track many services on a single host, or even across multiple hosts.

PROBLEM

You want to capture `docker logs` output centrally on your Docker host.

SOLUTION

Use the `--log-driver` flag to redirect logs to the desired location.

By default, Docker logs are captured within the Docker daemon, and you can access these with the `docker logs` command. As you're probably aware, this shows you the output of the container's main process.

At the time of writing, Docker gives you several choices for redirecting this output to multiple *log drivers*, including

- syslog
- journald
- json-file

The default is json-file, but others can be chosen with the `--log-driver` flag. The syslog and journald options send the log output to their respective daemons of the same name. You can find the official documentation on all available log drivers at https://docs.docker.com/engine/reference/logging/.

> **WARNING** This technique requires Docker version 1.6.1 or higher.

The syslog daemon is a process that runs on a server, collecting and managing messages sent to a central file (normally a Unix domain socket). It generally uses /dev/log as a file to receive log messages on, and logs out to /var/log/syslog.

Journald is a system service that collects and stores logging data. It creates and maintains a structured index of logs received from a variety of sources. The logs can be queried with the `journalctl` command.

Logging to syslog

To direct your output to the syslog, use the `--log-driver` flag:

```
$ docker run --log-driver=syslog ubuntu echo 'outputting to syslog'
outputting to syslog
```

This will record the output in the syslog file. If you have permission to access the file, you can examine the logs using standard Unix tools:

```
$ grep 'outputting to syslog' /var/log/syslog
Jun 23 20:37:50 myhost docker/6239418882b6[2559]: outputting to syslog
```

Logging to journald

Outputting to a journal daemon looks similar:

```
$ docker run --log-driver=journald ubuntu echo 'outputting to journald'
outputting to journald
$ journalctl | grep 'outputting to journald'
Jun 23 11:49:23 myhost docker[2993]: outputting to journald
```

> **WARNING** Ensure you have a journal daemon running on your host before running the preceding command.

Applying across all containers

It can be laborious to apply this argument to all containers on your host, so you can change your Docker daemon to log by default to these supported mechanisms.

Change the daemon /etc/default/docker, or /etc/sysconfig/docker, or whichever Docker config file your distribution has set up, such that the `DOCKER_OPTS=""` line is activated and includes the log-driver flag. For example, if the line was

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
```

change it to this:

```
DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4 --log-driver syslog"
```

> **TIP** See appendix B for details on how to change the Docker daemon's configuration on your host.

If you restart your Docker daemon, containers should then log to the relevant service.

Discussion

Another common choice worth mentioning in this context (but not covered here) is that you can use containers to implement an ELK (Elasticsearch, Logstash, Kibana) logging infrastructure.

WARNING Changing this daemon setting to anything other than `json-file` or `journald` will mean that the standard `docker logs` command will no longer work by default. Users of this Docker daemon may not appreciate this change, especially because the /var/log/syslog file (used by the `syslog` driver) is typically not accessible to non-root users.

TECHNIQUE 103 Monitoring containers with cAdvisor

Once you have a serious number of containers running in production, you'll want to monitor their resource usage and performance exactly as you do when you have multiple processes running on a host.

The sphere of monitoring (both generally, and with respect to Docker) is a wide field with many candidates. cAdvisor has been chosen here as it's a popular choice. Open-sourced by Google, it has quickly gained in popularity. If you already use a traditional host-monitoring tool such as Zabbix or Sysdig, then it's worth seeing whether it already offers the functionality you need—many tools are adding container-aware functionality as we write.

PROBLEM

You want to monitor the performance of your containers.

SOLUTION

Use cAdvisor as a monitoring tool.

cAdvisor is a tool developed by Google for monitoring containers. It's open-sourced on GitHub at https://github.com/google/cadvisor.

cAdvisor runs as a daemon that collects performance data on running containers. Among other things, it tracks

- Resource isolation parameters
- Historical resource usage
- Network statistics

cAdvisor can be installed natively on the host or run as a Docker container.

Listing 15.3 Running cAdvisor

```
$ docker run \
--volume /:/rootfs:ro \
  --volume /var/run:/var/run:rw \
  --volume /sys:/sys:ro \
  --volume /var/lib/docker/:/var/lib/docker:ro \
  -p 8080:8080 -d --name cadvisor \
  --restart on-failure:10 google/cadvisor
```

Gives cAdvisor read-only access to the root filesystem so it can track information about the host

Mounts the /var/run folder with read-write access. At most, one instance of cAdvisor is expected to run per host.

Gives cAdvisor read-only access to the host's /sys folder, which contains information about the kernel subsystems and devices attached to the host

Gives cAdvisor read-only access to Docker's host directory

cAdvisor's web interface is served on port 8080 of the container, so we publish it to the host on the same port. The standard Docker arguments to run the container in the background and give the container a name are also used.

Restarts the container on failure, up to a maximum of 10 times. The image is stored on the Docker Hub within Google's account.

Once you've started the image, you can visit http://localhost:8080 with your browser to start examining the data output. There's information about the host, but by clicking on the Docker Containers link at the top of the homepage, you'll be able to examine graphs of CPU, memory, and other historical data. Just click on the running containers listed under the Subcontainers heading.

The data is collected and retained in memory while the container runs. There is documentation for persisting the data to an InfluxDB instance on the GitHub page. The GitHub repository also has details about the REST API and a sample client written in Go.

TIP InfluxDB is an open source database designed to handle the tracking of time-series data. It's therefore ideal for recording and analyzing monitoring information that's provided in real time.

DISCUSSION

Monitoring is a fast-evolving and splintering space, and cAdvisor is just one component among many now. For example, Prometheus, the fast-emerging standard for Docker, can receive and store data produced by cAdvisor rather than placing it directly in InfluxDB.

Monitoring is also a subject that developers can get very passionate about. It can pay to develop a strategy for monitoring that can be flexible to meet changing fashions.

15.2 Resource control

One of the central concerns of running services in production is the fair and functional allocation of resources. Under the hood, Docker uses the core operating system concept of cgroups to manage containers' resource usage. By default, a simple and equal-share algorithm is used when containers contend for resources, but sometimes this isn't enough. You might want to reserve or limit resources for a container, or class of containers, for operational or service reasons.

In this section you'll learn how to tune containers' usage of CPU and memory.

TECHNIQUE 104 Restricting the cores a container can execute on

By default, Docker allows containers to execute on any cores on your machine. Containers with a single process and thread will obviously only be able to max out one core, but multithreaded programs in a container (or multiple single-threaded programs) will be able to use all your CPU cores. You might want to change this behavior if you have a container that's more important than others—it's not ideal for customer-facing applications to have to fight for the CPU every time your internal daily reports run. You could also use this technique to prevent runaway containers from locking you out of SSH to a server.

PROBLEM

You want a container to have a minimum CPU allocation, have a hard limit on CPU consumption, or otherwise want to restrict the cores a container can run on.

SOLUTION

Use the --cpuset-cpus option to reserve CPU cores for your container.

To properly explore the --cpuset-cpus option, you'll need to follow this technique on a computer with multiple cores. This may not be the case if you're using a cloud machine.

> **TIP** Older versions of Docker used the flag --cpuset, which is now deprecated. If you can't get --cpuset-cpus to work, try using --cpuset instead.

To look at the effects of the --cpuset-cpus option, we're going to use the htop command, which gives a useful graphical view of the core usage of your computer. Make sure this is installed before continuing—it's typically available as the htop package from your system package manager. Alternatively, you can install it inside an Ubuntu container started with the --pid=host option to expose process information from the host to the container.

If you now run htop, you'll probably see that none of your cores are busy. To simulate some load inside a couple of containers, run the following command in two different terminals:

```
docker run ubuntu:14.04 sh -c 'cat /dev/zero >/dev/null'
```

Looking back at htop, you should see that two of your cores now show 100% use. To restrict this to one core, docker kill the previous containers and then run the following command in two terminals:

```
docker run --cpuset-cpus=0 ubuntu:14.04 sh -c 'cat /dev/zero >/dev/null'
```

Now htop will show that only your first core is being used by these containers.

The --cpuset-cpus option permits multiple core specification as a comma-separated list (0,1,2), a range (0-2), or a combination of the two (0-1,3). Reserving a CPU for the host is therefore a matter of choosing a range for your containers that excludes a core.

DISCUSSION

You can use this functionality in numerous ways. For example, you can reserve specific CPUs for the host processes by consistently allocating the remaining CPUs to running containers. Or you could restrict specific containers to run on their own dedicated CPUs so they don't interfere with the compute used by other containers.

In a multi-tenant environment, this can be a godsend for ensuring that workloads don't interfere with each other.

TECHNIQUE 105 Giving important containers more CPU

Containers on a host will normally share CPU usage equally when they compete for it. You've seen how to make absolute guarantees or restrictions, but these can be a little inflexible. If you want a process to be able to use more CPU than others, it's a waste to constantly reserve an entire core for it, and doing so can be limiting if you have a small number of cores.

Docker facilitates multi-tenancy for users who want to bring their applications to a shared server. This can result in the *noisy neighbor* problem well known to those experienced with VMs, where one user eats up resources and affects another user's VM that happens to be running on the same hardware.

As a concrete example, while writing this book we had to use this functionality to reduce the resource use of a particularly hungry Postgres application that ate CPU cycles, robbing a web server on the machine of the ability to serve end users.

PROBLEM

You want to be able to give more important containers a bigger share of CPU or mark some containers as less important.

SOLUTION

Use the `-c`/`--cpu-shares` argument to the `docker run` command to define the relative share of CPU usage.

When a container is started up, it's given a number (1024 by default) of *CPU shares*. When only one process is running, it will have access to 100% of the CPU if necessary, no matter how many CPU shares it has access to. It's only when competing with other containers for CPU that the number is used.

Imagine we have three containers (A, B, and C) all trying to use all available CPU resources:

- If they've all been given equal CPU shares, they will each be allocated one third of the CPU.
- If A and B are given 512 and C is given 1024, C will get half of the CPU, and A and B will get a quarter each.
- If A is given 10, B is given 100, and C is given 1000, A will get under 1% of the available CPU resources and will only be able to do anything resource-hungry if B and C are idle.

All of this assumes that your containers can use all cores on your machine (or that you only have one core). Docker will spread the load from containers across all cores where possible. If you have two containers running single-threaded applications on a two-core machine, there's obviously no way to apply relative weighting while maximally using the available resources. Each container will be given a core to execute on, regardless of its weight.

If you want to try this out, run the following:

Listing 15.4 Starving a Docker shell of CPU

```
docker run --cpuset-cpus=0 -c 10000 ubuntu:14.04 \
sh -c 'cat /dev/zero > /dev/null' &
docker run --cpuset-cpus=0 -c 1 -it ubuntu:14.04 bash
```

Now see how doing anything in the bash prompt is sluggish. Note that these numbers are relative—you can multiply them all by 10 (for example) and they would mean exactly the same thing. But the default granted is still 1024, so once you start changing these numbers, it's worth considering what will happen to processes that start without a CPU share specified in the command and that run on the same CPU set.

TIP Finding the right CPU share levels for your use case is something of an art. It's worth looking at the output of programs such as top and vmstat to determine what's using CPU time. When using top, it's particularly useful to hit the "1" key to display what each CPU core is doing separately.

DISCUSSION

Although we haven't seen this technique directly used in the real world very often, and its use is generally seen on the underlying platform, it's good to understand and play with the underlying mechanism to know how it works when tenants are complaining about lack of access (or apparent lack of access) to resources. This happens often in real-world environments, especially if the tenants' workloads are sensitive to fluctuations in infrastructure availability.

TECHNIQUE 106 Limiting the memory usage of a container

When you run a container, Docker will allow it to allocate as much memory from the host as possible. Usually this is desirable (and a big advantage over virtual machines, which have an inflexible way of allocating memory). But sometimes applications can go out of control, allocate too much memory, and bring a machine grinding to a halt as it starts swapping. It's annoying, and it's happened to us many times in the past. We want a way of limiting a container's memory consumption to prevent this.

PROBLEM

You want to be able to limit the memory consumption of a container.

SOLUTION

Use the `-m/--memory` parameter to `docker run`.

If you're running Ubuntu, chances are that you don't have the memory-limiting capability enabled by default. To check, run `docker info`. If one of the lines in the output is a warning about `No swap limit support`, there's unfortunately some setup work you need to do. Be aware that making these changes can have performance implications on your machine for all applications—see the Ubuntu installation documentation for more information (http://docs.docker.com/engine/installation/ubuntulinux/#adjust-memory-and-swap-accounting).

In short, you need to indicate to the kernel at boot that you want these limits to be available. To do this, you'll need to alter /etc/default/grub as follows. If `GRUB_CMDLINE_LINUX` already has values in it, add the new ones at the end:

```
-GRUB_CMDLINE_LINUX=""
+GRUB_CMDLINE_LINUX="cgroup_enable=memory swapaccount=1"
```

You now need to run `sudo update-grub` and restart your computer. Running `docker info` should no longer give you the warning, and you're now ready to proceed with the main attraction.

First, let's crudely demonstrate that the memory limit does work by using a limit of 4 MB, the lowest possible.

Listing 15.5 Setting the lowest-possible memory limit for a container

```
$ docker run -it -m 4m ubuntu:14.04 bash
 root@cffc126297e2:/# \
python3 -c 'open("/dev/zero").read(10*1024*1024)'
 Killed
 root@e9f13cacd42f:/# \
A=$(dd if=/dev/zero bs=1M count=10 | base64)
 $
 $ echo $?
 137
```

Runs the container with a limit of 4 MB memory

Tries to load about 10 MB into memory

The process consumed too much memory and so was killed.

Tries to load 10 MB of memory directly into bash

Bash was killed, so the container exited.

Checks the exit code

The exit code is non-zero, indicating the container exited with an error.

There's a gotcha with this kind of constraint. To demonstrate this, we'll use the jess/stress image, which contains `stress`, a tool designed for testing the limits of a system.

> **TIP** Jess/stress is a helpful image for testing any resource limits you impose on your container. Try out the previous techniques with this image if you want to experiment more.

If you run the following command, you might be surprised to see that it doesn't exit immediately:

```
docker run -m 100m jess/stress --vm 1 --vm-bytes 150M --vm-hang 0
```

You've asked Docker to limit the container to 100 MB, and you've instructed `stress` to take up 150 MB. You can verify that `stress` is operating as expected by running this command:

```
docker top <container_id> -eo pid,size,args
```

The size column is in KB and shows that your container is indeed taking about 150 MB of memory, raising the question of why it hasn't been killed. It turns out that Docker double-reserves memory—half for physical memory and half to swap. If you try the following command, the container will terminate immediately:

```
docker run -m 100m jess/stress --vm 1 --vm-bytes 250M --vm-hang 0
```

This double reservation is just a default and can be controlled with the `--memory-swap` argument, which specifies the total virtual memory size (memory + swap). For example, to completely eliminate swap usage, you should set `--memory` and `--memory-swap` to be the same size. You can see more examples in the Docker `run` reference at https://docs.docker.com/engine/reference/run/#user-memory-constraints.

DISCUSSION

Memory limits are one of the hottest topics of any operations (or DevOps) team running a Docker platform. Misconfigured or poorly configured containers run out of assigned (or reserved) memory all the time (I'm looking at *you* Java developers!), requiring the writing of FAQs and runbooks to direct users to when they cry foul.

Being aware of what's going on here is a great help to supporting such platforms and giving users the context of what's going on.

15.3 Sysadmin use cases for Docker

In this section we're going to take a look at some of the surprising uses to which Docker can be put. Although it may seem strange at first glance, Docker can be used to make your cron job management easier and can be used as a form of backup tool.

> **TIP** A cron job is a timed, regular command that's run by a daemon included as a service with almost all Linux systems. Each user can specify their own schedule of commands to be run. It's heavily used by sysadmins to run periodic tasks, such as cleaning up log files or running backups.

This is by no means an exhaustive list of potential uses, but it should give you a taste of Docker's flexibility and some insight into how its features can be used in unexpected ways.

TECHNIQUE 107 Using Docker to run cron jobs

If you've ever had to manage cron jobs across multiple hosts, you may have come across the operational headache of having to deploy the same software to multiple places and ensuring the crontab itself has the correct invocation of the program you want to run.

Although there are other solutions to this problem (such as using Chef, Puppet, Ansible, or some other configuration management tool to manage the deployment of software across hosts), one option can be to use a Docker registry to store the correct invocation.

This isn't always the best solution to the problem outlined, but it's a striking illustration of the benefits of having an isolated and portable store of your applications' runtime configurations, and one that comes for free if you already use Docker.

PROBLEM

You want your cron jobs to be centrally managed and auto-updated.

SOLUTION

Pull and run your cron job scripts as Docker containers.

If you have a large estate of machines that need to run jobs regularly, you typically will use crontabs and configure them by hand (yes, that still happens), or you'll use a configuration management tool such as Puppet or Chef. Updating their recipes will ensure that when a machine's config management controller next runs, the changes are applied to the crontab, ready for the run following that.

TIP A *crontab* file is a special file maintained by a user that specifies the times scripts should be run. Typically these will be maintenance tasks, like compressing and archiving log files, but they could be business-critical applications, such as a credit card payment settler.

In this technique, we'll show you how to replace this scheme with Docker images delivered from a registry with 'docker pull'.

In the normal case, shown in figure 15.2, the maintainer updates the configuration management tool, which is then delivered to the servers when the agent is run. Meanwhile, the cron jobs are running with the old and new code while the systems update.

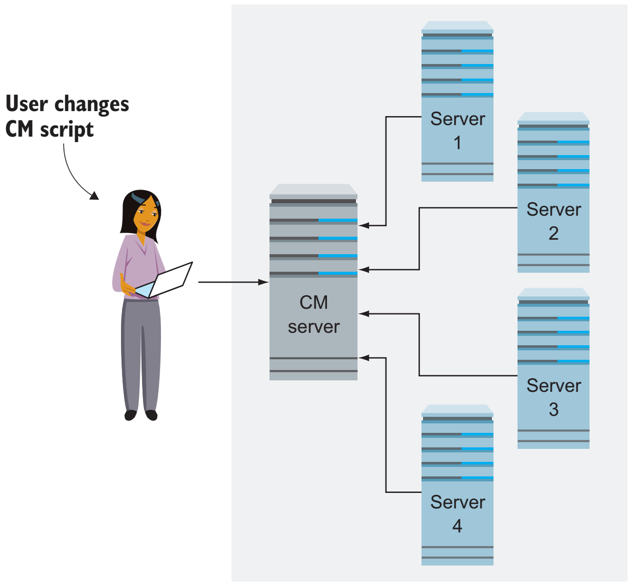

Figure 15.2 Each server updates cron scripts during a CM agent-scheduled run

In the Docker scenario, illustrated in figure 15.3, the servers pull the latest version of the code before the cron jobs run.

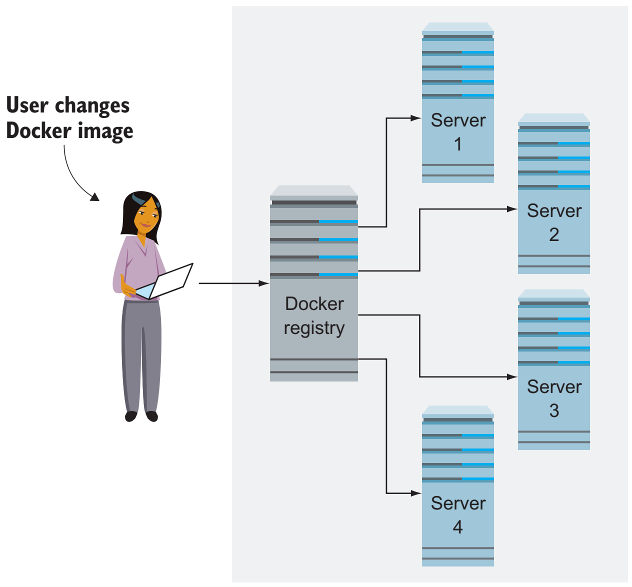

Figure 15.3 Each server pulls the latest image on every cron job run

At this point you may be wondering why it's worth bothering with this, if you already have a solution that works. Here are some advantages of using Docker as the delivery mechanism:

- Whenever a job is run, the job will update itself to the latest version from the central location.
- Your crontab files become much simpler, because the script and the code are encapsulated in a Docker image.
- For larger or more complex changes, only the deltas of the Docker image need be pulled, speeding up delivery and updates.
- You don't have to maintain the code or binaries on the machine itself.
- You can combine Docker with other techniques, such as logging output to the syslog, to simplify and centralize the management of these administration services.

For this example we're going to use the log_cleaner image we created in technique 49. You'll no doubt recall that this image encapsulated a script that cleaned up log files on a server and took a parameter for the number of days of log files to clean up. A crontab that uses Docker as a delivery mechanism would look something like the following listing.

Listing 15.6 Log cleaner crontab entry

```
0 0 * * * \
 IMG=dockerinpractice/log_cleaner && \
docker pull $IMG && \
 docker run -v /var/log/myapplogs:/log_dir $IMG 1
```

Runs this at midnight every day

First pulls the latest version of the image

Runs the log cleaner over a day's worth of log files

TIP If you're not familiar with cron, you may want to know that to edit your crontab you can run `crontab -e`. Each line specifies a command to be run at a time specified by the five items at the start of the line. Find out more by looking at the crontab man page.

If there's a failure, the standard cron mechanism of sending an email should kick into effect. If you don't rely on this, add a command with an `or` operator. In the following example, we assume your bespoke alerting command is `my_alert_command`.

Listing 15.7 Log cleaner crontab entry with alerting on error

```
0 0 * * * \
(IMG=dockerinpractice/log_cleaner && \
docker pull $IMG && \
docker run -v /var/log/myapplogs:/log_dir $IMG 1) \
|| my_alert_command 'log_cleaner failed'
```

TIP An `or` operator (in this case, the double pipe: `||`) ensures that one of the commands on either side will be run. If the first command fails (in this case, either of the two commands within the parentheses after the cron specification `0 0 * * *` joined by the `and` operator, `&&`), then the second will be run.

The `||` operator ensures that if any part of the log-cleaning job run failed, the alert command gets run.

DISCUSSION

We really like this technique for its simplicity and use of battle-tested technologies to solve a problem in an original way.

Cron has been around for decades (since the late 1970s, according to Wikipedia) and its augmentation by Docker image is a technique we use at home to manage regular jobs in a simple way.

TECHNIQUE 108 The "save game" approach to backups

If you've ever run a transactional system, you'll know that when things go wrong, the ability to infer the state of the system at the time of the problem is essential for a root-cause analysis.

Usually this is done through a combination of means:

- Analysis of application logs
- Database forensics (determining the state of data at a given point in time)
- Build history analysis (working out what code and config was running on the service at a given point in time)
- Live system analysis (for example, did anyone log onto the box and change something?)

For such critical systems, it can pay to take the simple but effective approach of backing up the Docker service containers. Although your database is likely to be separate from your Docker infrastructure, the state of config, code, and logs can be stored in a registry with a couple of simple commands.

PROBLEM

You want to keep backups of Docker containers.

SOLUTION

Commit the containers while running, and push the resulting image as a dedicated Docker repository.

Following Docker best practices and taking advantage of some Docker features can help you avoid the need to store container backups. As one example, using a logging driver as described in technique 102 instead of logging to the container filesystem means logs don't need to be retrieved from the container backups.

But sometimes reality dictates that you can't do everything the way you'd like, and you really need to see what a container looked like. The following commands show the entire process of committing and pushing a backup container.

Listing 15.8 Committing and pushing a backup container

Generates a timestamp to the granularity of a second

Generates a tag that points to your registry URL with a tag that includes the hostname and date

```
DATE=$(date +%Y%m%d_%H%M%S)
TAG="your_log_registry:5000/live_pmt_svr_backup:$(hostname -s)_${DATE}"
docker commit -m="$DATE" -a="Backup Admin" live_pmt_svr $TAG
docker push $TAG
```

Commits the container with the date as a message and "Backup Admin" as the author

Pushes the container to a registry

WARNING This technique will pause the container while it runs, effectively taking it out of service. Your service should either tolerate outages, or you should have other nodes running at the time that can service requests in a load-balanced fashion.

If this is done in a staggered rotation across all your hosts, you'll have an effective backup system and a means to restore the state for support engineers with as little ambiguity as possible. Figure 15.4 illustrates a simplified view of such a setup.

The backups only push the differences between the base image and the state of the container at the time it's backed up, and the backups are staggered to ensure that the service stays up on at least one host. The registry server only stores one copy of the base image and the diffs at each commit point, saving disk space.

DISCUSSION

You can take this technique one step further by combining this technique with a so-called "Phoenix deployment" model. Phoenix deployment is a model for deployment that emphasizes replacing as much of the system as possible rather than upgrading a deployment in-place. It's a central principle of many Docker tools.

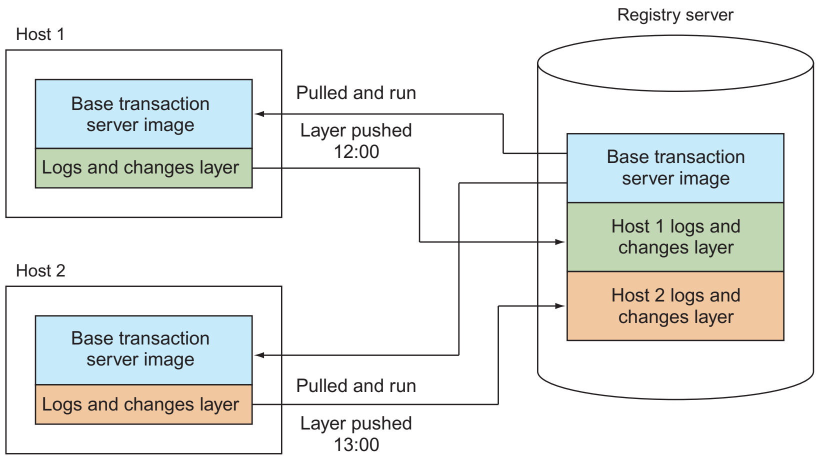
Host 1
Base transaction server image
Logs and changes layer
Pulled and run
Layer pushed 12:00
Registry server
Base transaction server image
Host 1 logs and changes layer
Host 2 logs and changes layer
Host 2
Base transaction server image
Logs and changes layer
Pulled and run
Layer pushed 13:00

Figure 15.4 Two-host backup of a service

In this case, rather than committing the container and letting it continue on afterward, you can do the following:

1. Pull a fresh copy of the latest image from your registry
2. Stop the running container
3. Start up a new container
4. Commit, tag, and push the old container to the registry

Combining these approaches gives you even more certainty that the live system hasn't drifted from the source image. One of us uses this approach to manage a live system on a home server.

Summary

- You can direct logging from your containers to your host's syslog daemon.
- Docker log output can be captured to a host-level service.
- cAdvisor can be used to monitor the performance of your containers.
- Container usage of CPU, core, and memory can be limited and controlled.
- Docker has some surprising uses, such as being a cron delivery tool and a backup system.