ErwanAliasr1 README.md: Updating documentation 245f2a4 Mar 8, 2018

11 contributors

300 lines (237 sloc) | 15.6 KB

ceph-container

Ceph-related Docker files.

Core Components:

- `ceph/daemon-base`: Base container image containing Ceph core components.
- `ceph/daemon`: All-in-one container containing all Ceph daemons.

See README files in subdirectories for instructions on using containers.

Demo

- `ceph/demo`: Demonstration cluster for testing and learning. This container runs all the major Ceph and Ganesha components installed, bootstrapped, and executed for you to play with. (not intended for use in building a production cluster)

Contributing to ceph-container

1. Become familiar with the project structure.
2. Follow the appropriate contribution workflow

Project structure

The primary deliverables of this project are two container images: `daemon-base`, and `daemon`. As such, the project structure is influenced by what files and configurations go into each image. The main source base (including configuration files and build specifications) of the project is located in `src/` and is further specified in `src/daemon-base` and `src/daemon`.

Because this project supports several different Ceph versions and many OS distros, the structure also allows individual Ceph versions, individual distros, and combinations of Ceph-version-and-distro (we will call these **flavors**) to override the base source, configuration, and specification files by specifying their own versions of the files or new files entirely. In addition to overrides the `flavor-blacklist.txt` file can be configured prevent a file from overriding another for specific subsets of flavors.

Mentally modeling the end result of overrides and blacklists for any given flavor is difficult. Similarly, programmatically selecting the correct files for each flavor build is also difficult. In order to effectively work with this project structure, we introduce the concept of **staging**.

The concept of staging

Special tooling has been built to collect all source files with appropriate overrides and without blacklisted items into a unique staging directory for each flavor (each flavor is also specified by a target architecture to support ARM platforms). From a staging directory, containers can be built directly from the `<staging>/daemon-base/` and `<staging>/daemon/` image directories. Additionally, developers can inspect a staging directory's files to view exactly what will be (or has been) built into the container images. Additionally, in order to maintain a core source base that is as reusable as possible for all flavors, staging also supports a very basic form of templating. Some tooling has been developed to make working with staging as easy as possible.

Staging override order

It is key that staging is deterministic; therefore, a clear definition of override priority is needed. More specific files will override (overwrite) less specific files when staging. Note here that `FILE` may be a file or a directory containing further files.

```
# Most specific
ceph-releases/<ceph release>/<os distro>/<os release>/{daemon-base,daemon}/FILE
ceph-releases/<ceph release>/<os distro>/<os release>/FILE
ceph-releases/<ceph release>/<os distro>/{daemon-base,daemon}/FILE
ceph-releases/<ceph release>/<os distro>/FILE
ceph-releases/<ceph release>/{daemon-base,daemon}/FILE
ceph-releases/<ceph release>/FILE
ceph-releases/ALL/<os distro>/<os release>/{daemon-base,daemon}/FILE
ceph-releases/ALL/<os distro>/<os release>/FILE
ceph-releases/ALL/<os distro>/{daemon-base,daemon}/FILE
ceph-releases/ALL/<os distro>/FILE
ceph-releases/ALL/{daemon-base,daemon}/FILE
ceph-releases/ALL/FILE
src/{daemon-base,daemon}/FILE
src/FILE
# Least specific
```

Basic templating in staging

Variable file replacements

In any source file, a special variable in the form `__VAR_NAME__` (two leading and trailing underscores with capital letters and underscores between) can be placed. Once all files are staged, the `__VAR_NAME__` variable will be replaced with the raw contents of the file with the named `__VAR_NAME__`. Trailing whitespace is stripped from the variable file before insertion. `__VAR_NAME__` files are allowed to be empty, but they are not allowed to be nonexistent if a file declares them. A `__VAR_NAME__` definition file may contain nested `__OTHER_VAR_NAME__` variables.

If the `__DO_STUFF__` file is supposed to contain actions that need done it generally needs to return true. As an example, `echo 'first' && __DO_STUFF__ && echo 'last'` will print 'first' and 'last' correctly only if `__DO_STUFF__` returns true. A take-no-action override needs to have the content `/bin/true`, as an empty file will cause an error.

Variables can be nested meaning `__DO_STUFF__` can contain some code including a reference to another variable like `__ANOTHER_VARIABLE__`.

Environment variable replacements

In any source file, a special variable in the form `STAGE_REPLACE_WITH_ENV_VAR` (capital letters with underscores) can placed. Once all files are staged, the `STAGE_REPLACE_WITH_ENV_VAR` variable will be replaced with the raw contents of the environment variable named `ENV_VAR`.

Environment variable replacements **cannot** be nested inside of other environment variable replacements. `__VAR__` file definitions, however, may specify environment variable replacements.

A typical usage is to use `STAGE_REPLACE_WITH_ARCH` when you need to specify the building architecture.

Staging development aids

To practically aid developers, helpful tools have been built for staging:

- To create all default staging directories: `make stage`

- To create specific staging directory(-ies): `make FLAVORS_TO_BUILD=<flavors> stage`
- Find the source of a staged file: `cd <staging dir> ; ./find-src <file-path>`
- List of staged files and their sources: `<staging dir>/files-sources`
- List of all possible buildable flavors: `make show.flavors`
- Show flavors affected by branch changes: `make flavors.modified`
- Stage log: `stage.log`

Building images

`make` is used for ceph-container builds. See `make help` for all make options.

Specifying flavors for make

The `make` tooling allows the environment variable `FLAVORS_TO_BUILD` to be optionally set by the user to define which flavors to operate on. Flavor specifications follow a strict format that declares what ceph-container source to use, what architecture to build for, and what container image to use as the base for the build. See `make help` for a full description.

Building a single flavor

Once the flavor is selected, just specify its name in the **FLAVORS_TO_BUILD** and call the **build** target like in :

```
make FLAVORS_TO_BUILD=luminous,amd64,centos,7,_,centos,7 build
```

Building multiple flavors

Proceed as per the single one and separate them by a space like in :

```
make FLAVORS_TO_BUILD="luminous,amd64,centos,7,_,centos,7 kraken,amd64,centos,7,_,centos,7"  build
```

If you want to build all possible images, just call

```
make build.all
```

If you want to build them in parallel :

```
make build.parallel
```

Building images from staging

It is also possible to build container images directly from staging in the event that `make build` is not appealing. Simply stage the flavor(s) to be built, and then execute the desired build command for `daemon-base` and `daemon`.

```
# Example
cd <staging>
docker build -t <daemon base tag> daemon-base/
docker build -t <daemon tag> daemon/
```

Where does (should) source code live?

`src/`

The `src/` directory contains the core code that is applicable for all flavors -- all Ceph versions and all distros -- and is the "least specific" definition of source files for this project. This includes the ceph-container-specific code that is built into the container images as well as build configuration files. `src/` should be viewed as the base set of functionality all flavors must implement. To maximize reuse and keep the core source base in one place, care should be taken to try to make the core source applicable to all flavors whenever possible. Note the existence of the "CEPH_VERSION" environment variable (and other similar variables) built into the containers. ceph-container source code can use container environment variables to execute code paths conditionally, making an override of source files for specific Ceph versions unnecessary.

Where possible `src/` provides a specification of a "sane default" that may need to be overridden for certain flavors. For example, the `src/daemon/__DAEMON_PACKAGES__` file defines the daemon packages which should be installed in the base image. A distro is not required to install these packages via its package manager or to use this list (though it is recommended to gain as much reuse as possible), as each distro may have a different preferred method of installation. This list should, however, serve as a a guide to what must be installed.

`ceph-releases/ALL/`

The `ceph-releases/All/` directory contains code that is generic to all Ceph releases. It is more specific than `src/`. Source that is shared between distros (but is not part of ceph-container's core functionality) can be placed in this directory.

Distro-specific source is placed in `ceph-releases/ALL/<distro>` and is yet more specific. As an example, since all Ubuntu releases use `apt` as their package manager, the `ceph-releases/ALL/ubuntu` dir is able to provide definitions for package installation using `apt` for all Ubuntu-based flavors.

`ceph-releases/<ceph release>`

A `ceph-releases/<ceph release>` directory is more specific than `ceph-releases/ALL` and contains source that is specific to a particular Ceph release but is generic for all distros.

A `ceph-releases/<ceph release>/<distro>` directory is the most specific source directory and contains source that is specific to a particular Ceph-release-and-distro combination.

Contribution workflow

The goal when adding contributions should be to make modifications or additions to the least specific files in order to reuse as much code as possible. Only specify specific changes when they absolutely apply only to the specific flavor(s) and not to others.

General suggestions

- Make use of already-defined `__VAR__` files when possible to maximize reuse.
- Make changes in the least specific (see override order) directory as makes sense for the flavor to maximize reuse across flavors.

Fixing a bug

1. Stage the flavor on which the bug was found (`make FLAVORS_TO_BUILD=<bugged flavor> stage`).
2. Use the `find-src` script or `files-sources` list to locate the bugged file's source location.
3. Edit the source location to fix the bug.
4. Build test versions of the images (`make FLAVORS_TO_BUILD=<bugged flavor> build`).
5. Test the images you built in your environment.
6. Make a PR of your changes.

Adding a feature

1. Determine the scope of the feature
    - To which Ceph versions should the feature be added?
    - To which distros should the feature be added?
    - As a general guideline, new features should usually be added to all Ceph versions and distros.

2. Add relevant changes to files in the project structure such that they will be added to only the Ceph versions and distros in scope for the feature.
3. Build test versions of the images (`make FLAVORS_TO_BUILD=<in-scope-flavors> build`).
4. Test the images in your environment.
5. Make a PR of your changes.

Adding a Ceph release

Ideally, adding a new Ceph release is fairly easy. In the best case, all that needs done is adding flavors for the new Ceph version to the Makefile. At minimum, `ALL_BUILDABLE_FLAVORS` must be updated in the Makefile. If distro source is properly configured to support multiple Ceph releases and there are no special updates required, they are likely to work with just this minimal change.

Note the `$CEPH_VERSION` variable usually used in `__DOCKERFILE_INSTALL__` is substituted from the first field of the flavor name.

In this example, `luminous,amd64,centos,7,_,centos,7` , `$CEPH_VERSION` will be set to **luminous**.

Adding a new flavor name like `mimic,amd64,centos,7,_,centos,7` is enough to create a new **mimic** Ceph release.

In the worst case, trying to make as few modifications as possible:

1. Add flavors for new Ceph versions to the Makefile.
   - At minimum: `ALL_BUILDABLE_FLAVORS` .
2. Edit `src/` files to support the new version if necessary, making sure not to break previous versions. Keep container environment variables in mind here.
3. Edit `ceph-releases/ALL/<distro>` files to support the new version if necessary, making sure not to break previous versions.
4. Add `ceph-releases/<new ceph version>` files to support the new version if necessary.
5. Build test versions of the images (`make FLAVORS_TO_BUILD=<new release flavors> build`).
6. Test the images in your environment.
7. Make a PR of your changes.

Adding a distro build

1. Add flavors for the new distro to the Makefile.
   - At minimum: `ALL_BUILDABLE_FLAVORS` .
2. Add a `ceph-releases/ALL/<new distro>` directory for the new distro
   - Make sure to install all the required packages for `daemon-base` (see `src/daemon-base/__CEPH_BASE_PACKAGES__)` and for `daemon` (see `src/daemon/__DAEMON_PACKAGES__`).
   - Make sure to specify all `__VAR__` files without sane defaults for the container builds.
   - Make sure to override any `__VAR__` file sane defaults that do not apply to the new distro.
   - Refer to other distros for inspiration.
3. If necessary, add a `ceph-release/<ceph release>/<new distro>` directory for the new distro. Try to avoid this as much as possible and focus on code reuse from the less specific dirs.
4. Build test versions of the images (`make FLAVORS_TO_BUILD=<new distro flavors> build`).
5. Test the images in your environment.
6. Make a PR of your changes.

CI

We use Travis to run several tests on each pull request:

- we build both `daemon-base` and `daemon` images
- we run all the Ceph processes in a container based on the images we just built
- we execute a validation script at the end to make sure Ceph is healthy

For each PR, we try to detect which Ceph release is being impacted. Since we can only produce a single CI build with Travis, ideally this change will only be on a single release and distro. If we have multiple Ceph release and distro, we can only test one, since we have to build `base` and `daemon`. By default, we just pick up the first line that comes from the changes.

You can check the files in `travis-builds` to learn more about the entire process.

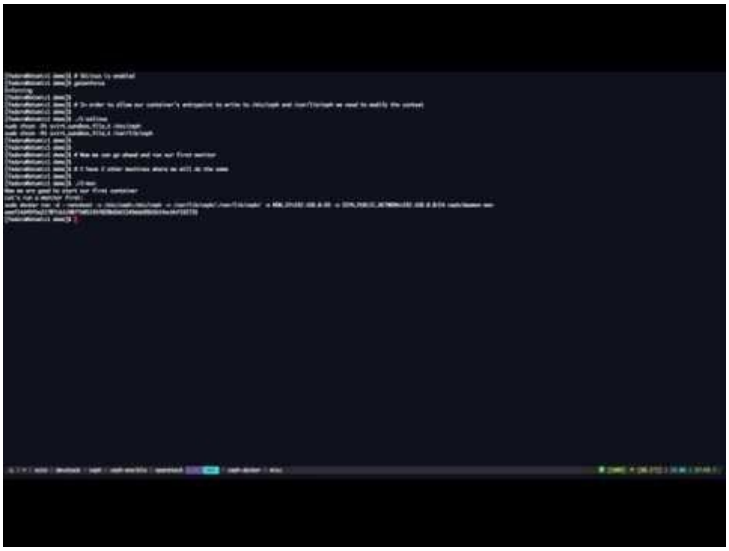
If you don't want to run a build for a particular commit, because all you are changing is the README for example, add `[ci skip]` to the git commit message. Commits that have `[ci skip]` anywhere in the commit messages are ignored by Travis CI.

We are also transitioning to have builds in Jenkins, this is still a work in progress and will start taking precedence once it is solid enough. Be sure to check the links and updates provided on pull requests.

Video demonstration

Manually

A recorded video on how to deploy your Ceph cluster entirely in Docker containers is available here:



With Ansible