CONFIGURATION MANAGEMENT SYSTEM

The configuration management system exists to provide every daemon with the proper configuration information. The configuration can be viewed as a set of key-value pairs.

How can the configuration be set? Well, there are several sources:

- the ceph configuration file, usually named ceph.conf
- command line arguments::

  –debug-ms=1 –debug-pg=10 etc.

- arguments injected at runtime using “injectargs” or “config set”

THE CONFIGURATION FILE

Most configuration settings originate in the Ceph configuration file.

How do we find the configuration file? Well, in order, we check:

- the default locations
- the environment variable CEPH_CONF
- the command line argument -c

Each stanza of the configuration file describes the key-value pairs that will be in effect for a particular subset of the daemons. The “global” stanza applies to everything. The “mon”, “osd”, and “mds” stanzas specify settings to take effect for all monitors, all OSDs, and all mds servers, respectively. A stanza of the form mon.$name, osd.$name, or mds.$name gives settings for the monitor, OSD, or MDS of that name, respectively. Configuration values that appear later in the file win over earlier ones.

A sample configuration file can be found in src/sample.ceph.conf.

METAVARIABLES

The configuration system allows any configuration value to be substituted into another value using the `$varname` syntax, similar to how bash shell expansion works.

A few additional special metavariables are also defined:

- $host: expands to the current hostname
- $type: expands to one of “mds”, “osd”, “mon”, or “client”
- $id: expands to the daemon identifier. For `osd.0`, this would be `0`; for `mds.a`, it would be `a`; for `client.admin`, it would be `admin`.
- $num: same as $id
- $name: expands to $type.$id

READING CONFIGURATION VALUES

There are two ways for Ceph code to get configuration values. One way is to read it directly from a variable named “g_conf,” or equivalently, “g_ceph_ctx->_conf.” The other is to register an observer that will be called every time the relevant configuration values changes. This observer will be called soon after the initial configuration is read, and every time after that when one of the relevant values changes. Each observer tracks a set of keys and is invoked only when one of the relevant keys changes.

The interface to implement is found in common/config_obs.h.

The observer method should be preferred in new code because

- It is more flexible, allowing the code to do whatever reinitialization needs to be done to implement the new configuration value.
- It is the only way to create a std::string configuration variable that can be changed by injectargs.
- Even for int-valued configuration options, changing the values in one thread while another thread is reading them can lead to subtle and impossible-to-diagnose bugs.

For these reasons, reading directly from g_conf should be considered deprecated and not done in new code. Do not ever alter g_conf.

CHANGING CONFIGURATION VALUES

Configuration values can be changed by calling g_conf->set_val. After changing the configuration, you should call g_conf->apply_changes to re-run all the affected configuration observers. For convenience, you can call g_conf->set_val_or_die to make a configuration change which you think should never fail.

Injectargs, parse_argv, and parse_env are three other functions which modify the configuration. Just like with set_val, you should call apply_changes after calling these functions to make sure your changes get applied.

DEFINING CONFIG OPTIONS

New-style config options are defined in common/options.cc. All new config options should go here (and not into legacy_config_opts.h).

LEVELS

The Option constructor takes a "level" value:

- *LEVEL_BASIC* is for basic config options that a normal operator is likely to adjust.
- *LEVEL_ADVANCED* is for options that an operator *can* adjust, but should not touch unless they understand what they are doing. Adjusting advanced options poorly can lead to problems (performance or even data loss) if done incorrectly.
- *LEVEL_DEV* is for options in place for use by developers only, either for testing purposes, or to describe constants that no user should adjust but we prefer not to compile into the code.

DESCRIPTION AND LONG DESCRIPTION

Short description of the option. Sentence fragment. e.g.:

```
.set_description("Default checksum algorithm to use")
```

The long description is complete sentences, perhaps even multiple paragraphs, and may include other detailed information or notes.:

```
.set_long_description("crc32c, xxhash32, and xxhash64 are available.  The _16 and _8 variants
```

DEFAULT VALUES

There is a default value for every config option. In some cases, there may also be a *daemon default* that only applies to code that declares itself as a daemon (in thise case, the regular default only applies to non-daemons).

SAFETY

If an option can be safely changed at runtime:

```
.set_safe()
```

SERVICE

Service is a component name, like "common", "osd", "rgw", "mds", etc. It may be a list of components, like:

```
.add_service("mon mds osd mgr")
```

For example, the rocksdb options affect both the osd and mon.

TAGS

Tags identify options across services that relate in some way. Example include;

- network – options affecting network configuration
- mkfs – options that only matter at mkfs time

ENUMS

For options with a defined set of allowed values:

```
.set_enum_allowed({"none", "crc32c", "crc32c_16", "crc32c_8", "xxhash32", "xxhash64"})
```