CEPH-MGR PLUGIN AUTHOR GUIDE

CREATING A PLUGIN

In pybind/mgr/, create a python module. Within your module, create a class that inherits from `MgrModule`.

The most important methods to override are:

- a `serve` member function for server-type modules. This function should block forever.
- a `notify` member function if your module needs to take action when new cluster data is available.
- a `handle_command` member function if your module exposes CLI commands.

INSTALLING A PLUGIN

Once your module is present in the location set by the `mgr module path` configuration setting, you can enable it via the `ceph mgr module enable` command:

```
ceph mgr module enable mymodule
```

Note that the MgrModule interface is not stable, so any modules maintained outside of the Ceph tree are liable to break when run against any newer or older versions of Ceph.

LOGGING

`MgrModule` instances have a `log` property which is a logger instance that sends log messages into the Ceph logging layer where they will be recorded in the mgr daemon's log file.

Use it the same way you would any other python logger. The python log levels debug, info, warn, err are mapped into the Ceph severities 20, 4, 1 and 0 respectively.

EXPOSING COMMANDS

Set the `COMMANDS` class attribute of your plugin to a list of dicts like this:

```
COMMANDS = [
    {
        "cmd": "foobar name=myarg,type=CephString",
        "desc": "Do something awesome",
        "perm": "rw"
    }
]
```

The `cmd` part of each entry is parsed in the same way as internal Ceph mon and admin socket commands (see mon/MonCommands.h in the Ceph source for examples)

CONFIG SETTINGS

Modules have access to a simple key/value store (keys and values are byte strings) for storing configuration. Don't use this for storing large amounts of data.

Config values are stored using the mon's config-key commands.

Hints for using these:

- Reads are fast: ceph-mgr keeps a local in-memory copy
- Don't set things by hand with "ceph config-key", the mgr doesn't update at runtime (only set things from within modules).

- Writes block until the value is persisted, but reads from another thread will see the new value immediately.

Any config settings you want to expose to users from your module will need corresponding hooks in `COMMANDS` to expose a setter.

ACCESSING CLUSTER DATA

Modules have access to the in-memory copies of the Ceph cluster's state that the mgr maintains. Accessor functions as exposed as members of MgrModule.

Calls that access the cluster or daemon state are generally going from Python into native C++ routines. There is some overhead to this, but much less than for example calling into a REST API or calling into an SQL database.

There are no consistency rules about access to cluster structures or daemon metadata. For example, an OSD might exist in OSDMap but have no metadata, or vice versa. On a healthy cluster these will be very rare transient states, but plugins should be written to cope with the possibility.

Note that these accessors must not be called in the modules `__init__` function. This will result in a circular locking exception.

`MgrModule.`**`get`**(*data_name*)

Called by the plugin to fetch named cluster-wide objects from ceph-mgr.

Parameters: **data_name** (*str*) – Valid things to fetch are osd_crush_map_text, osd_map, osd_map_tree, osd_map_crush, config, mon_map, fs_map, osd_metadata, pg_summary, df, osd_stats, health, mon_status.

Note:

All these structures have their own JSON representations: experiment or look at the C++ `dump()` methods to learn about them.

`MgrModule.`**`get_server`**(*hostname*)

Called by the plugin to fetch metadata about a particular hostname from ceph-mgr.

This is information that ceph-mgr has gleaned from the daemon metadata reported by daemons running on a particular server.

Parameters: **hostname** – a hostame

`MgrModule.`**`list_servers`**()

Like `get_server`, but gives information about all servers (i.e. all unique hostnames that have been mentioned in daemon metadata)

Returns: a list of infomration about all servers
Return type: list

`MgrModule.`**`get_metadata`**(*svc_type*, *svc_id*)

Fetch the daemon metadata for a particular service.

Parameters:
- **svc_type** (*str*) – service type (e.g., 'mds', 'osd', 'mon')
- **svc_id** (*str*) – service id. convert OSD integer IDs to strings when calling this

Return type: dict

`MgrModule.`**`get_counter`**(*svc_type*, *svc_name*, *path*)

Called by the plugin to fetch the latest performance counter data for a particular counter on a particular service.

Parameters:
- **svc_type** (*str*) –
- **svc_name** (*str*) –
- **path** (*str*) – a period-separated concatenation of the subsystem and the counter name, for example "mds.inodes".

Returns: A list of two-tuples of (timestamp, value) is returned. This may be empty if no data is available.

WHAT IF THE MONS ARE DOWN?

The manager daemon gets much of its state (such as the cluster maps) from the monitor. If the monitor cluster is inaccessible, whichever manager was active will continue to run, with the latest state it saw still in memory.

However, if you are creating a module that shows the cluster state to the user then you may well not want to mislead them by showing them that out of date state.

To check if the manager daemon currently has a connection to the monitor cluster, use this function:

`MgrModule.`**`have_mon_connection`**`()`

Check whether this ceph-mgr daemon has an open connection to a monitor. If it doesn't, then it's likely that the information we have about the cluster is out of date, and/or the monitor cluster is down.

REPORTING IF YOUR MODULE CANNOT RUN

If your module cannot be run for any reason (such as a missing dependency), then you can report that by implementing the `can_run` function.

static `MgrModule.`**`can_run`**`()`

Implement this function to report whether the module's dependencies are met. For example, if the module needs to import a particular dependency to work, then use a try/except around the import at file scope, and then report here if the import failed.

This will be called in a blocking way from the C++ code, so do not do any I/O that could block in this function.

:return a 2-tuple consisting of a boolean and explanatory string

Note that this will only work properly if your module can always be imported: if you are importing a dependency that may be absent, then do it in a try/except block so that your module can be loaded far enough to use `can_run` even if the dependency is absent.

SENDING COMMANDS

A non-blocking facility is provided for sending monitor commands to the cluster.

`MgrModule.`**`send_command`**`(`*`*args`*`, `*`**kwargs`*`)`

Called by the plugin to send a command to the mon cluster.

Parameters:
- **result** (*CommandResult*) – an instance of the `CommandResult` class, defined in the same module as MgrModule. This acts as a completion and stores the output of the command. Use `CommandResult.wait()` if you want to block on completion.
- **svc_type** (*str*) –
- **svc_id** (*str*) –
- **command** (*str*) – a JSON-serialized command. This uses the same format as the ceph command line, which is a dictionary of command arguments, with the extra `prefix` key containing the command name itself. Consult MonCommands.h for available commands and their expected arguments.
- **tag** (*str*) – used for nonblocking operation: when a command completes, the `notify()` callback on the MgrModule instance is triggered, with notify_type set to "command", and notify_id set to the tag of the command.

IMPLEMENTING STANDBY MODE

For some modules, it is useful to run on standby manager daemons as well as on the active daemon. For example, an HTTP server can usefully serve HTTP redirect responses from the standby managers so that the user can point his browser at any of the manager daemons without having to worry about which one is active.

Standby manager daemons look for a subclass of `StandbyModule` in each module. If the class is not found then the module is not used at all on standby daemons. If the class is found, then its `serve` method is called. Implementations of `StandbyModule` must inherit from `mgr_module.MgrStandbyModule`.

The interface of `MgrStandbyModule` is much restricted compared to `MgrModule` – none of the Ceph cluster state is available to the module. `serve` and `shutdown` methods are used in the same way as a normal module class. The `get_active_uri` method enables the standby module to discover the address of its active peer in order to make redirects. See the `MgrStandbyModule` definition in the Ceph source code for the full list of methods.

For an example of how to use this interface, look at the source code of the `dashboard` module.

LOGGING

Use your module's `log` attribute as your logger. This is a logger configured to output via the ceph logging framework, to the local ceph-mgr log files.

Python log severities are mapped to ceph severities as follows:

- DEBUG is 20
- INFO is 4
- WARN is 1
- ERR is 0

SHUTTING DOWN CLEANLY

If a module implements the `serve()` method, it should also implement the `shutdown()` method to shutdown cleanly: misbehaving modules may otherwise prevent clean shutdown of ceph-mgr.

IS SOMETHING MISSING?

The ceph-mgr python interface is not set in stone. If you have a need that is not satisfied by the current interface, please bring it up on the ceph-devel mailing list. While it is desired to avoid bloating the interface, it is not generally very hard to expose existing data to the Python code when there is a good reason.