Cantal-Py Documentation

Release 0.2.5

Paul Colomiets

Jul 27, 2018

Contents

Contents:

CHAPTER 1

Tutorial

Cantal is a metrics collection system. It integrates with your application by providing some metrics. Here is a bare (but fully working) example:

```python
import cantal

ticks = cantal.Counter(group="main_loop", metric="ticks")

def main():
    cantal.start()

    while True:
        sleep(0.1)
        ticks.incr(1)

if __name__ == '__main__':
    main
```

The key things:

1. You declare metrics
2. Then you call `start()`
3. Afterwards you may freely adjust values of the metrics

Rules of thumb:

1. Create all metrics at module import time
2. Import all modules before calling `main()` or whatever init function
3. Writing metric is very cheap, so you don't need to aggregate them in user code
4. Creating metrics dynamically (after `start()`) is not supported, but you can create metrics in a loop or similar

See *Choosing Metrics* for more info.

CHAPTER 2

Choosing Metrics

CHAPTER 3

Reference

3.1 Basic Metrics

All metrics have a constructor which receives arbitrary keyword arguments. The values must be strings, and they are arbitrary key-value pairs to identify the metric in monitoring. Multiple keyword pairs are used to group metrics over multiple processes by different dimensions.

`class Counter`

The 64bit integer counter. Counter always grows. (However, it may wrap on overflow). In monitoring it's usually used for displaying per-second rates of some values. For example:

- Count of requests (=> requests per second)
- Duration of all requests (=> average request latency)
- Number of accepted connections
- Number of connection attempts to a DB

`incr`(*value=1*)

Increment the counter.

Hint: The operation is very cheap, so you don't need to aggregate multiple increments on your own.

`class Integer`

The 64bit signed integer value. It represents something that is useful on it's own (unlike `Counter`), without getting derivation of. Examples:

- Number of requests being processed
- Size of queue
- Number of connections in the connection pool (total, free, used)

`set`(*value*)

Update value of the metric

incr(*value=1*)
Increment value of metric

Hint: Updating the value is very cheap. You don't have to duplicate the value somewhere. For example, you may increment the counter on the start of the request and decrement afterwards. You may use *get()* method to get the value if you need to use it for applciation needs.

decr(*value=1*)
Decrement value of metric. Equivalent of `incr(-value)`

get()
Get value last written (or adjusted according to the counters)

class Float
This is similar to *Integer* but represents floating point value.

set(*value*)
Update value of the metric

class State
The class represents some internal state of the application. This stores some string state value and timestamp when it was last changed.

Parameters **size** – maximum size of the state data. Note that this number of bytes is reserved, so it shouldn't be too big. Truncation of data is perfectly okay. This should be *64*n - 8* for best performance.

Usege examples:

- Which resource the process is currently waits for
- Currently executing SQL query
- The start/process/shutting down application lifecycle

See *Fork* for more comprehensive state handling built on top of *State*.

context(*value*)
A context manager which sets state name to `value` and clears state on exit.

enter(*value*)
Enter the state with `value`. This also marks the timestamp when state is started. Better use context manager for most cases

exit()
Clear the state

3.2 Compound Utilities

class RequestTracker
The class embeds multiple counters so it's easy to track both incoming and outgoing requests.

Example:

```
http = RequestTracker('http')
sql = RequestTracker('http.sql')

def application(environ, start_response):
```

(continues on next page)

(continued from previous page)

```
    with http.request():
        do_something()
        with sql.request():
            value = sql_query()
        if value == None:
            http.errors.incr()
            start_response('500 Internal Server Error', [])
            return [b"Error"]
        do_something_else()
        start_response('200 OK', [])
        return [value.encode('utf-8')]
```

The counter group embeds the following primitive metrics:

- `requests` – the *Counter* of requests
- `total_duration` (aliased as `duration` in python attribute) – *Counter* for total duration of all requests (in milliseconds), this is later used to calculate average response time
- `errors` – *Counter* for number of errors
- `in_progress` – *Integer* of current requests in progress

You are free to use `req_tracker.errors.incr()` for all your custom errors which are not exceptions (i.e. non-200 HTTP response). Exceptions are tracked automatically.

This works for both synchronous and asynchronous processes. In synchonous ones the `in_progress` is likely to be `0` or `1` (but when summing over cluster you'll get some bigger values).

request()

Returns context manager that tracks requests.

The `requests` and `total_duration` are incremented *after* request.

The `errors` is automatically incremented if exception happened inside the context manager.

class Fork

The class to handle multiple states of the application. In the frontend it allows to draw chart of where application spends most of the time, and which states are reached more often.

Example:

```
track_request = Fork(['app', 'redis', 'sql'],
                     state="myapp.request_processing")

def process_request(req):
    with track_request.context():

        track_request.redis.enter()
        rdata = redis.get('something')

        track_request.sql.enter()
        sdata = postgres.query("SELECT ...")

        track_request.app.enter()
        return render_template(rdata, sdata)
```

context()

Enter the fork root state. The default state is named _ (single underscore). It's meant to enter some branch soon afterwards.

class Branch

Represents branch of a *Fork*. You shouldn't create it on it's own but use the attribute of a fork.

enter()

Enter the branch as part of this *Fork*.

3.3 Collection Classes

Usually you don't need to instantiate collection classes. They are handled internally.

class Collection

A collection of metrics when it's being populated with metrics.

class ActiveCollection

A collection of metrics when tracks metrics and can't have more metrics added.

3.4 Exceptions

class DuplicateValueException

Raised when you define two metrics with all the same key-value pairs.

CHAPTER 4

Integrations

This sections provides guides to integration of cantal metrics with various pythonic frameworks and tools.

Contents:

4.1 Aiohttp

4.1.1 Incoming(Server) Connections

Here is an example how to track number of incoming connections in HTTP server:

```
CONNECTIONS = cantal.Integer(group='http.server', metric='connections')

def adopt_aiohttp_server(Server):
   conn_made = Server.connection_made
   conn_lost = Server.connection_lost

   def connection_made(*a, **kw):
       CONNECTIONS.incr()
       return conn_made(*a, **kw)

   def connection_lost(*a, **kw):
       CONNECTIONS.decr()
       return conn_lost(*a, **kw)
   Server.connection_made = connection_made
   Server.connection_lost = connection_lost

from aiohttp.web_server import Server
adopt_aiohttp_server(Server)
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index