qbfutures Documentation

Release 0.1

Western X

February 25, 2016

Contents

This Python package is a implementation of a concurrent.futures.Executor for PipelineFX's Qube. The API is compatible with the standard `Executor` and provides extensions for working with Qube.

CHAPTER 1

Installation

The contained “types” directory must be added to the Qube server’s `"worker_template_path"`, and the python package must be importable.

CHAPTER 2

Overview

Basic usage is exactly the same:

```
>>> executor = qbfutures.Executor():
>>> future = executor.submit(my_function, 1, 2, key="value")
>>> future.result()
"Awesome results!"
```

An extended submit function, *Executor.submit_ext*, allows you to provide more information to Qube about how to handle the job. Anything that would normally be set into a `qb.Job` object is viable and will be passed through:

```
>>> future = executor.submit(my_function, name="Job Name", groups="farm")
```

Keyword arguments can also be passed to *Executor.map*:

```
>>> results_iter = executor.map(my_function, range(10), cpus=10)
```

Finally, keyword arguments to the *Executor* constructor will be used as defaults on all submitted jobs:

```
>>> executor = Executor(cpus=4, groups='farm')
>>> # Submit some jobs, and they will take on the cpus and group above.
```

2.1 Batch Mode

Often, logical jobs will be spread into multiple chunks of work. If those are processed individually via *Executor.submit* they will be queued as individual jobs. A batch mode has been added to the API to facilitate grouping multiple function calls into a single Qube job:

```
>>> with Executor().batch(name="A set of functions", cpus=4) as batch:
...     f1 = batch.submit(a_function, 'input')
...     f2 = batch.submit_ext(another_function, name='work name')
...     map_iter = batch.map(mapping_function, range(10))
...
>>> f1.results()
>>> f2.results()
>>> list(map_iter)
```

While batch methods will return a *Future*, they will not be in a valid state until the batch has been submitted. They will not have job or work IDs, and iterating over a *Batch.map* result is undefined.

Since jobs submited via a batch are individual work items, extra keyword arguments to either *Batch.submit_ext* or *Batch.map* will be passed through to the `qb.Work`.

2.2 Maya

A *maya.Executor* subclass exists for use with Maya, which will bootstrap the Maya process, and optionally open a file to work on and set the workspace. It also provides convenience functions for cloning the current environment, and creating a temporary copy of the current file for the other processes to work on.

```
>>> executor = qbfutures.maya.Executor(clone_environ=True, cpus=4)
>>> executor.create_tempfile()
>>> with executor.batch("Get Node Types") as batch:
...     for node in cmds.ls(sl=True):
...         future = batch.submit(cmds.nodeType, node)
...         future.node = node
...
>>> for future in as_completed(batch.futures):
...     print future.job_id, future.work_id, future.node, future.result()
```

CHAPTER 3

API Reference

3.1 Executor

class qbfutures.**Executor**(***kwargs*)

An object which provides methods to execute functions asynchonously on Qube.

Any keyword arguments passed to the constructor are used as a template for every job submitted to Qube.

submit(*func*, **args*, ***kwargs*)

Schedules the given callable to be executed as `func(*args, **kwargs)`.

Returns The *Future* linked to the submitted job.

submit_ext(*func*, *args=None*, *kwargs=None*, ***extra*)

Extended submission with more control over Qube job.

Parameters

- **func** – The function to call.
- **args** (*list*) – The positional arguments to call with.
- **kwargs** (*dict*) – The keyword arguments to call with.
- ****extra** – Values to pass through to the `qb.Job`.

Returns The *Future* linked to the submitted job.

map(*func*, **iterables*, ***extra*)

Equivalent to `map(func, *iterables)` except `func` is executed asynchronously on Qube.

Parameters **timeout** – The number of seconds to wait for results, or `None`.

Any other keyword arguments will be passed through to the `qb.Job`:

```
>>> for result in Executor().map(my_function, range(10), cpus=4):
...     print result
```

batch(*name=None*, ***kwargs*)

Start a batch process.

Parameters

- **name** (*str*) – The name of the Qube job.
- ****kwargs** – Other parameters for the Qube job.

Returns The *Batch* to use to schedule jobs in a batch.

::

```
>>> with Executor().batch() as batch:
...     f1 = batch.submit(first_function)
...     f2 = batch.submit(second_function)
...
>>> print f1.results()
```

3.2 Future

class qbfutures.**Future**(*job_id*, *work_id*)
A Future representing a unit of work on Qube.

job_id = None
The Qube job ID.

work_id = None
The index of this work item into the job's agenda.

status()
Get the current status for this particular work item.

3.3 Batch

class qbfutures.core.**Batch**(*executor*, *job*)
Pseudo-executor that submits callables into a single Qube job.

Be careful not to use any of the resulting futures until the jobs have been submitted, either by using the Batch as a context manager, or calling *commit()*.

submit(*func*, **args*, ***kwargs*)
Same as *Executor.submit*

submit_ext(*func*, *args=None*, *kwargs=None*, ***extra*)
Same as *Executor.submit_ext*, except extra keyword arguments are passed to the qb.Work.

map(*func*, **iterables*, ***extra*)
Same as *Executor.map*, except extra keyword arguments are passed to the qb.Work.

commit()
Perform the actual job submittion. Called automatically if used as a context manager.

3.4 Maya

class qbfutures.maya.**Executor**(*clone_environ=None*, *create_tempfile=False*, *filename=None*, *workspace=None*, *version=None*, ***kwargs*)
An executor that is tailored to the Maya environment.

Parameters

- **clone_environ** (*bool*) – Convenience for *clone_environ()*.
- **create_tempfile** (*bool*) – Convenience for *create_tempfile()*.
- **filename** (*str*) – File to open once bootstrapped.

- **workspace** (*str*) – Workspace to set once bootstrapped.
- **version** (*int*) – Version of maya to use.

create_tempfile()
Save the current file in a temporary location for Qube processes to use.

clone_environ()
Set the jobs to use the same environment that we are currently in.

Sets the current filename, workspace, and version.

CHAPTER 4

Installation

This package depends upon `concurrent.futures`, which is included with Python 3. For Python 2, the futures package provides a backport.

Qube must also have access to the custom jobtype; either the `qbfutures` type must be copied to where your jobtypes are stored, or the `types` directory must be added to the `worker_template_path` within the `qb.conf` for your workers.

CHAPTER 5

Special Considerations

Unlike when using `threading`, callables and their arguments must be serialized (via `pickle`) to be passed to the Qube workers. This places some restrictions upon what can be used. A non-exhaustive list of rules include:

- callables (functions or classes) must be within the global scope of a module;
- aallables must be uniquely named within that module;
- a callable's module must have a `__name__` that is not importable;
- lambdas are not permissable (since they cannot be pickled).

5.1 Within `__main__`

Many of our tools are called via the `-m` switch of the python interpreter. In that case, callables within the main module are not unpickleable since their module is named `__main__` and then the callable will not be found. As such, you may also pass a string in the form `package.module:function` to specify a callable:

```
>>> executor.submit('awesome_package.the_tool:qube_handler', *args)
```

5.2 Lambda Workaround

Even though lambdas are not pickleable, we can also acheive the same effect as lambdas by calling `eval` and passing in a string of Python source code, and a dictionary for the scope to run that code in:

```
>>> executor.submit(eval, 'a + b', dict(a=1, b=2)).result()
3
```

5.3 Recursion

Jobs are free to schedule additional jobs, but sometimes this can run away from us and take over all of the worker resources. Therefore, a very conservative recursion limit has been setup; by default recusion will only be allowed to 4 levels, and the 5th recursive job will fail to schedule.

A `QBLVL` variable has been placed into the execution environment to track how deep the current recursion is, with the first job assuming a value of 1.

The recusion limit may be increased by setting a `QBFUTURES_RECURSION_LIMIT` variable in the environment.

5.4 Shutdown

For various reasons, we have to forcibly shutdown the Python process. We do our best to clean up as much as we can (by triggering `atexit`, clearing all modules, and running the garbage collector), but we cannot guarantee that your destructors will be called in the same way as a normal process.

Be mindful to cleanup all resources.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

B

C

E

F

J

M

S

W