Pex Harness for Apps Developers

This page, along with this step-by-step tutorial is intended to help Apps Developers get started writing pipelines using the LSST harness classes provided in the pex_harness package.

Pipeline Harness Components

The Pipeline Harness is a set of classes for assembling pipelines out of reusable scientific processing modules. Its purpose is to separate application concerns from middleware concerns. As an apps developer that's all you need/want to know, but if you are interested in understanding how it all works under the covers, take a look at the links at the bottom of the page.

A. Stages

For an apps developer, the first point of contact with the harness is the ?Stage class. A `Stage` acts as a modular encapsulation for an algorithm. Its interaction with the rest of the harness is through three methods:

- preprocess
- process
- postprocess

In order to implement your own algorithm, you must define a class that derives from `Stage`, and overrides at least one of the above methods.

```
from lsst.pex.harness.Stage import Stage

#Define a class that derives from Stage
class MyAlgorithmStage(Stage):
    ...
    #implements the process method
    def process(self):
       #Algorithm code goes here!
```

But choosing which method(s) to override requires a more knowledge about the harness, in particular Slices.

B. Pipelines and Slices

The harness is designed to help apps developers run their code across multiple CPUs and threads. A Pipeline will have at least two process threads (currently implemented as separate processes). The first is the master pipeline controller thread, and the second is a slave worker thread. You can define additional worker threads, but there will always be only one master thread. In LSST the the controller thread is termed the `Pipeline` and worker threads are called `Slices`.

The `Pipeline` orchestrates the execution of `Stages` on your `Slice`(s). As an apps developer, what you need to take away from this is that every `Stage`'s **preprocess** and **postprocess** methods are run **in serial** on the Pipeline, and nowhere else.

A `Slice` is a single worker thread, and it is responsible for running the **process** method of every `Stage` in your pipeline. Slices of a pipeline need to operate on data-parallel chunks of data. If that parallelism changes between two stages, the stages need to be part of different pipelines, with an Event between them.

C. Events

All you need to understand about events is that they are sent using a reliable queued event system (currently ActiveMQ), and that they are nothing more than a `PropertySet` object. Take special care that Event names match throughout your pipeline. There's nothing more embarrassing than having a pipeline idle waiting for an Event with a typo in its name.

D. Clipboard

This is an important concept to master. On each `Slice`, stages communicate using the `Clipboard`. Stages can pull data from the clipboard, and push data to the clipboard. A Stage may choose to provide data to a subsequent stage with the command:

```
self.outputQueue.addDataset(self.activeClipboard)
```

which pushes the clipboard to the "outputQueue". And you need not concern yourself with what that is or how it works. Just know that its contents will be available to subsequent stages. Stages are not limited to pushing out one clipboard.

Later stages may retrieve clipboards from their "inputQueue" with the command:

```
self.activeClipboard = self.inputQueue.getNextDataset()
```

There may be more than one clipboard in the "inputQueue" and you can inspect them all with a loop like this:

```
for i in xrange(self.inputQueue.size()):
    clipboard = self.inputQueue.getNextDataset()
    ...
```

II. Configuring With Policies

Pipelines are defined by creating a policy file that lists all the stages, along with other information about how to structure the pipeline.

You can find out more about acceptable pipeline configurations: DC3PipelinePolicyDesign

Additionally you can provide policy files for each of your stages. When authoring Stages, you should provide Policy Dictionary files that describe acceptable policy files for that Stage.

III. Interslice Communication

Although this is not encouraged as the principal form of communication, the harness allows for communication between separate slices of the same pipeline. To enable interslice communication you must add the following to your pipeline policy file:

- Enable Interslice in the pipeline

  ```
  shareDataOn: true
  ```
- For each stage that will share data between slices, you must instruct it to do so:

```
...
appStage: {
    stageName: ...
    ...
    shareData: true
    ...
}
...
```

Once interslice communication has been enabled, you can tell stages what data to share. For example:

```
class MyStage(Stage)
    ...
    def process(self):
        ...
        # Create a PropertySet object
        propertySet = lsst.daf.Base.PropertySet()

        # Add it to the clipboard under the name "myKey"
        self.activeClipboard.put("myKey", propertySet)

        # Mark it as a shared item
        self.activeClipboard.setShared("myKey", True)

        # Push out the clipboard
       self.outputQueue.addDataset(self.activeClipboard)
```

There are also mechanisms for setting up topologies to allow slices to share information with each other, rather than only with the master Pipeline process. These are not yet fully documented.

IV. Predefined Stages

The good guys at middleware have defined a small set of `Stage` classes to help make writing pipelines a great deal simpler. All you have to do, is configure the `Stage` by using a policy file.

A. InputStage

source: ?InputStage.py
description: perform data retrieval. *see twiki*
policy reference: *see twiki*

B. OutputStage

source: ?OutputStage.py
description: perform data persistence. *see twiki*
policy reference: *see twiki*

C. EventStage

source: ?EventStage.py
description: publish custom events to communicate across pipelines.
policy reference: *see twiki*

D. SymLinkStage

source: ?SymLinkStage.py
description: Create symbolic links for files outside run directory*see twiki*
policy reference: *see twiki*

V. Pipeline Execution

1. The Pipeline and the Slice objects are created and configured according to a given Policy. As part of the configuration, the Pipeline notes the events that it needs to receive and when. The Pipeline initializes a stage position counter, *N*, that indicates which Stage is to be executed next, setting it to first stage (N=1).
2. The Pipeline object transfers the Clipboard from the output side of stage *N-1* to the input side of stage *N*. For the first stage (when the *N=1*), this means initializing the Clipboard to an empty state.
3. The Pipeline object receives any events it is configured to listen for prior to the execution of the stage *N*. If so configured, it waits until that event has been received. Upon receipt, it places any data contained in it on the Clipboard for stage *N* in the Pipeline process and also re-sends the event (under a different name) to each of the Slice processes, which also place the event data on their own Clipboards.
4. The Pipeline object calls the `preprocess` method on its instance of the Stage *N*.
5. After `preprocess` is complete, the Pipeline object inspects any Clipboard output by the Stage for shareable data. It then sends an MPI scatter message out to all of the worker processes telling them to execute the parallel portion of stage N; the message is followed by a serialization of any shareable data. *(Additional info on interslice communication here.)*
6. The Slice object in each worker process receives the message. The Slice object transfers the Clipboard from the output side of stage *N-1* to the input side of stage *N*. It then unserializes any shareable data items it receives from the master process and places them on the input clipboard for the stage *N* (with the shareable tag turned off). It then calls the `process` method on its instance of the Stage *N* object.
7. After the `process` method completes, each Slice object inspects any Clipboard output by the Stage for shareable data. It sends an MPI gather message back to the master process indicating that the parallel processing is done; the message is followed by a serialization of any shareable data. *(Additional info on interslice communication here.)*
8. When the Pipeline object has received the messages from all the worker nodes, it aggregates any received shareable data items and places them back on the master Clipboard and posts it to the input clipboard of stage N. It then executes the `postprocess` step stage *N*.
9. When the `postprocess` is done, the Pipeline object can repeat this sequence beginning with step 2 for stage *N+1*, and so on, until the entire chain of stages is complete.

Further Reading

Many of these pages were written for DC2 so they are NOT the cutting edge of development. All the same I have found them to be invaluable source of knowledge if you want to take a peek under the hood.

1. Pipeline Framework in DC2
2. Testing Pipelines in DC2
3. Stage File Organization in DC2
4. Data Mapping in DC3