tf.train.Coordinator

Contents

Class **Coordinator**

Defined in `tensorflow/python/training/coordinator.py`.

See the guides: Reading data > Reading from files, Threading and Queues > Manual Thread Management, Training > Coordinator and QueueRunner

A coordinator for threads.

This class implements a simple mechanism to coordinate the termination of a set of threads.

Usage:

```
# Create a coordinator.
coord = Coordinator()
# Start a number of threads, passing the coordinator to each of them.
...start thread 1...(coord, ...)
...start thread N...(coord, ...)
# Wait for all the threads to terminate.
coord.join(threads)
```

Any of the threads can call `coord.request_stop()` to ask for all the threads to stop. To cooperate with the requests, each thread must check for `coord.should_stop()` on a regular basis. `coord.should_stop()` returns `True` as soon as `coord.request_stop()` has been called.

A typical thread running with a coordinator will do something like:

```
while not coord.should_stop():
  ...do some work...
```

Exception handling:

A thread can report an exception to the coordinator as part of the `request_stop()` call. The exception will be re-raised from the `coord.join()` call.

Thread code:

```
try:
  while not coord.should_stop():
    ...do some work...
except Exception as e:
  coord.request_stop(e)
```

Main code:

```
try:
  ...
  coord = Coordinator()
  # Start a number of threads, passing the coordinator to each of them.
  ...start thread 1...(coord, ...)
  ...start thread N...(coord, ...)
  # Wait for all the threads to terminate.
  coord.join(threads)
except Exception as e:
  ...exception that was passed to coord.request_stop()
```

To simplify the thread implementation, the Coordinator provides a context handler **`stop_on_exception()`** that automatically requests a stop if an exception is raised. Using the context handler the thread code above can be written as:

```
with coord.stop_on_exception():
  while not coord.should_stop():
    ...do some work...
```

Grace period for stopping:

After a thread has called **`coord.request_stop()`** the other threads have a fixed time to stop, this is called the 'stop grace period' and defaults to 2 minutes. If any of the threads is still alive after the grace period expires **`coord.join()`** raises a RuntimeError reporting the laggards.

```
try:
  ...
  coord = Coordinator()
  # Start a number of threads, passing the coordinator to each of them.
  ...start thread 1...(coord, ...)
  ...start thread N...(coord, ...)
  # Wait for all the threads to terminate, give them 10s grace period
  coord.join(threads, stop_grace_period_secs=10)
except RuntimeError:
  ...one of the threads took more than 10s to stop after request_stop()
  ...was called.
except Exception:
  ...exception that was passed to coord.request_stop()
```

Properties

`joined`

Methods

`__init__`

```
__init__(clean_stop_exception_types=None)
```

Create a new Coordinator.

Args:

- `clean_stop_exception_types`: Optional tuple of Exception types that should cause a clean stop of the coordinator. If an exception of one of these types is reported to `request_stop(ex)` the coordinator will behave as if `request_stop(None)` was called. Defaults to `(tf.errors.OutOfRangeError,)` which is used by input queues to signal the end of input. When feeding training data from a Python iterator it is common to add `StopIteration` to this list.

clear_stop

```
clear_stop()
```

Clears the stop flag.

After this is called, calls to `should_stop()` will return `False`.

join

```
join(
    threads=None,
    stop_grace_period_secs=120,
    ignore_live_threads=False
)
```

Wait for threads to terminate.

This call blocks until a set of threads have terminated. The set of thread is the union of the threads passed in the `threads` argument and the list of threads that registered with the coordinator by calling `Coordinator.register_thread()`.

After the threads stop, if an `exc_info` was passed to `request_stop`, that exception is re-raised.

Grace period handling: When `request_stop()` is called, threads are given 'stop_grace_period_secs' seconds to terminate. If any of them is still alive after that period expires, a `RuntimeError` is raised. Note that if an `exc_info` was passed to `request_stop()` then it is raised instead of that `RuntimeError`.

Args:

- `threads`: List of `threading.Threads`. The started threads to join in addition to the registered threads.
- `stop_grace_period_secs`: Number of seconds given to threads to stop after `request_stop()` has been called.
- `ignore_live_threads`: If `False`, raises an error if any of the threads are still alive after `stop_grace_period_secs`.

Raises:

- `RuntimeError`: If any thread is still alive after `request_stop()` is called and the grace period expires.

raise_requested_exception

```
raise_requested_exception()
```

If an exception has been passed to `request_stop`, this raises it.

register_thread

```
register_thread(thread)
```

Register a thread to join.

Args:

- `thread`: A Python thread to join.

request_stop

```
request_stop(ex=None)
```

Request that the threads stop.

After this is called, calls to `should_stop()` will return `True`.

> ★ **Note:** If an exception is being passed in, in must be in the context of handling the exception (i.e. `try: ... except Exception as ex: ...`) and not a newly created one.

Args:

- `ex`: Optional `Exception`, or Python `exc_info` tuple as returned by `sys.exc_info()`. If this is the first call to `request_stop()` the corresponding exception is recorded and re-raised from `join()`.

should_stop

```
should_stop()
```

Check if stop was requested.

Returns:

True if a stop was requested.

stop_on_exception

```
stop_on_exception(
    *args,
    **kwds
)
```

Context manager to request stop when an Exception is raised.

Code that uses a coordinator must catch exceptions and pass them to the `request_stop()` method to stop the other threads managed by the coordinator.

This context handler simplifies the exception handling. Use it as follows:

```
with coord.stop_on_exception():
  # Any exception raised in the body of the with
  # clause is reported to the coordinator before terminating
  # the execution of the body.
  ...body...
```

This is completely equivalent to the slightly longer code:

```
try:
  ...body...
exception Exception as ex:
  coord.request_stop(ex)
```

Yields:

nothing.

`wait_for_stop`

```
wait_for_stop(timeout=None)
```

Wait till the Coordinator is told to stop.

Args:

- `timeout`: Float. Sleep for up to that many seconds waiting for should_stop() to become True.

Returns:

True if the Coordinator is told stop, False if the timeout expired.
