trio-util

Release 0.1.1

Groove X, Inc.

Jun 04, 2020

CONTENTS:

CHAPTER

ONE

NURSERY UTILITIES

The following utilities are intended to avoid nursery boilerplate in some simple cases.

wait_any() and *wait_all()* are used to simultaneously run async functions which either have side effects and don't return a value, or signal merely by exiting. For example, given two `trio.Event` objects `a` and `b`, we can wait until either event is true:

```
await wait_any(a.wait, b.wait)
```

or wait until both events are true:

```
await wait_all(a.wait, b.wait)
```

async trio_util.**wait_any**(**args*)

Wait until any of the given async functions are completed.

Equivalent to creating a new nursery and calling *start_soon()* on each async function, where the first function to return will cause the nursery to be cancelled.

If the function invocations require arguments, use partial():

```
await wait_any(partial(foo, 'hello'),
               partial(bar, debug=True))
```

async trio_util.**wait_all**(**args*)

Wait until all of the given async functions are completed.

Equivalent to creating a new nursery and calling *start_soon()* on each async function.

NOTE: Be careful when using this with a function that returns when some non-permanent condition is satisfied (e.g. *AsyncBool.wait_value*). While waiting for the other async function to complete, the state which satisfied the condition may change.

CHAPTER

TWO

VALUE WRAPPERS

These value wrappers offer the ability to wait for a specific value or transition. `AsyncValue` can wrap any type, while `AsyncBool` offers a simplified API for the common case of bool values.

class `trio_util.`**`AsyncValue`**(*value*)

Value wrapper offering the ability to wait for a value or transition.

Synopsis:

```
>>> a = AsyncValue(0)  # NOTE: can wrap any type (enum, tuple, ...)
>>> ...
>>> a.value = 5   # access underlying value
>>> ...
>>> # wait for value predicate
>>> await a.wait_value(lambda v: v > 10)
>>> ...
>>> # wait for transition predicate (default: any)
>>> await a.wait_transition(lambda v, old: v > 10 and old < 0)
```

When using *wait_value()* and *wait_transition()*, note that the value may have changed again before the caller receives control.

Performance note: assignment to the *value* property has O(N) complexity, where N is the number of active wait predicates.

property value

The wrapped value

async wait_value(*predicate*)

Wait until given predicate f(value) is True.

The predicate is tested immediately and, if false, whenever the *value* property changes.

returns value which satisfied the predicate

async wait_transition(*predicate=None*)

Wait until given predicate f(value, old_value) is True.

The predicate is tested whenever the *value* property changes. The default is *None*, which responds to any value change.

returns (value, old_value) which satisfied the predicate

class `trio_util.`**`AsyncBool`**(*value=False*)

Boolean wrapper offering the ability to wait for a value or transition.

Synopsis:

```
>>> a = AsyncBool()
>>> ...
>>> a.value = True   # access underlying value
>>> ...
>>> await a.wait_value(False)  # wait for a specific value
>>> ...
>>> await a.wait_transition()  # wait for a transition (default: any)
```

When using *wait_value()* and *wait_transition()*, note that the value may have changed again before the caller receives control.

property value

The wrapped value

async wait_value(*value*)

Wait until given value.

async wait_transition(*value=None*)

Wait until transition to given value (default None which means any).

CHAPTER

THREE

COLLECTIONS

`AsyncDictionary` has many uses, such as multiplexing a networking connection among tasks.

class `trio_util.`**`AsyncDictionary`**(**args*, ***kwargs*)

Bases: `collections.abc.MutableMapping`, `typing.Generic`

MutableMapping with waitable get and pop.

TODO: exception support using outcome package

`async get_wait`(*key: KT*) → VT

Return value of given key, blocking until populated.

`async pop_wait`(*key: KT*) → VT

Remove key and return its value, blocking until populated.

`is_waiting`(*key: KT*) → bool

Return True if there is a task waiting for key.

CHAPTER

FOUR

REPEATED EVENTS

`trio.Event` does not offer a clear() method, so it can't be triggered multiple times. It's for your own good.

The following are event classes which can be triggered repeatedly in a relatively safe manner. This is achieved by allowing only one listener and automatically clearing the event after it's received.

class `trio_util.`**`UnqueuedRepeatedEvent`**

An unqueued repeated event.

A repeated event may be triggered multiple times, and has only one listener. A call to set() is ignored if the listener is still processing the previous event (or there is no listener), and won't be queued.

```
>>> event = UnqueuedRepeatedEvent()
```

One task runs a listening loop, waiting for set():

```
>>> async for _ in event:
>>>     # do blocking work
>>>     await trio.sleep(1)
```

Another task triggers events:

```
>>> event.set()  # trigger event
>>> trio.sleep(0)  # event processing starts
>>> event.set()  # ignored, because previous event is still being processed
>>> trio.sleep(2)
>>> event.set()  # trigger event
```

set()

Trigger event if there is a waiting listener.

class `trio_util.`**`MailboxRepeatedEvent`**

A repeated event which queues up to one set() call.

A repeated event may be triggered multiple times, and has only one listener. Up to one set() call may be queued if the listener is still processing the previous event.

It's often used for signaling that an out-of-band data location has changed (hence the name "mailbox"). If the data is a collection type (list, etc.) it's safe to use this class, but other types of data are subject to overrun.

AVOID USING THIS CLASS FOR NON-COLLECTION DATA because data changes can be lost if the listener does not keep up. Consider using a queue instead (see trio.open_memory_channel) so that there is back pressure ensuring that the data is received.

```
>>> event = MailboxRepeatedEvent()
>>> data = None
```

One task runs a listening loop, waiting for set():

```
>>> async for _ in event:
>>>     # process data
>>>     print(data)
>>>     await trio.sleep(1)
```

Another task triggers iteration:

```
>>> data = 'foo'
>>> event.set()  # trigger event
>>> trio.sleep(0)  # event processing starts
>>> data = 'bar'
>>> event.set()  # trigger event (queued since previous event is being processed)
```

set()

Trigger event

Up to one event may be queued if there is no waiting listener (i.e. no listener, or the listener is still processing the previous event).

CHAPTER

FIVE

GENERATORS

`trio_util.`**`periodic`**(*period*)

Yield *(elapsed_time, delta_time)* with an interval of *period* seconds.

For example, to loop indefinitely with a period of 1 second, accounting for the time taken in the loop itself:

```
async for _ in periodic(1):
    ...
```

In the case of overrun, the next iteration begins immediately.

On the first iteration, *delta_time* will be *None*.

CHAPTER

SIX

ITERATORS

`trio_util.`**`azip`**(**aiterables*)

async version of izip with parallel iteration

`trio_util.`**`azip_longest`**(**aiterables*, *fillvalue=None*)

async version of izip_longest with parallel iteration

CHAPTER

SEVEN

MISCELLANEOUS

class trio_util.**TaskStats**(*current_time=<function current_time>*)

Bases: trio.abc.Instrument

Trio scheduler Instrument which logs various task stats at termination.

Includes max task wait time, slowest task step, and highest task schedule rate.

trio_util.**defer_to_cancelled**(**args*)

Context manager which defers MultiError exceptions to Cancelled.

In the scope of this context manager, any raised trio.MultiError exception which is a combination of the given exception types and trio.Cancelled will have the exception types filtered, leaving only a Cancelled exception.

The intended use is where routine exceptions (e.g. which are part of an API) might occur simultaneously with Cancelled (e.g. when using move_on_after()). Without properly catching and filtering the resulting MultiError, an unhandled exception will occur. Often what is desired in this case is for the Cancelled exception alone to propagate to the cancel scope.

Parameters **args** – One or more exception types which will defer to trio.Cancelled. By default, all exception types will be filtered.

Example:

```
# If MultiError([Cancelled, Obstacle]) occurs, propagate only Cancelled
# to the parent cancel scope.
with defer_to_cancelled(Obstacle):
    try:
        # async call which may raise exception as part of API
        await advance(speed)
    except Obstacle:
        # handle API exception (unless Cancelled raised simultaneously)
        ...
```

TODO: Support consolidation of simultaneous user API exceptions (i.e. MultiError without Cancelled). This would work by prioritized list of exceptions to defer to. E.g. given:

```
[Cancelled, WheelObstruction, RangeObstruction]
```

then:

```
Cancelled + RangeObstruction => Cancelled
WheelObstruction + RangeObstruction => WheelObstruction
```

INDEX