
trio-util

Release 0.3.0

GROOVE X, Inc.

Oct 27, 2020

CONTENTS:

1	nursery utilities	1
2	value wrappers	3
3	repeated events	5
4	generators	7
5	iterators	9
6	exceptions	11
7	miscellaneous	13
	Index	15

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)
```

await `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))
```

await `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.

VALUE WRAPPERS

AsyncValue can wrap any type, offering the ability to wait for a specific value or transition. *AsyncBool* is just an *AsyncValue* that defaults to `False`.

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 match by equality
>>> await a.wait_value(7)
>>> ...
>>> # wait for value match by predicate
>>> await a.wait_value(lambda v: v > 10)
>>> ...
>>> # wait for transition by equality
>>> await a.wait_transition(14)
>>> ...
>>> # wait for transition by predicate (default: any transition)
>>> 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.

value

The wrapped value

await **wait_value** (*value_or_predicate*, *, *held_for*=0)

Wait until given predicate *f*(*value*) is `True`.

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

If a non-callable is provided, it's equivalent to a predicate matching the given value.

If *held_for* > 0, the predicate must match for that duration from the time of the call. “held” means that the predicate is continuously true.

returns value which satisfied the predicate (when *held_for* > 0, it's the most recent value)

await **wait_transition** (*value_or_predicate*=<function *_ANY_TRANSITION*>)

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

The predicate is tested whenever the *value* property changes. The default predicate responds to any value change.

If a non-callable is provided, it's equivalent to a predicate matching the given value.

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.

Other than the constructor value defaulting to False, this class is the same as AsyncValue.

Sometimes you want to wait on a condition involving multiple async values. This can be achieved without resorting to polling by employing the *compose_values()* context manager.

async with trio_util.compose_values(**value_map) **as** composed

Async context manager providing a composite of multiple AsyncValues

The composite object itself is an AsyncValue, with the *value* of each underlying object accessible as attributes on the composite *value*.

compose_values() expects named AsyncValue instances to be provided as keyword arguments. The attributes of the composite value will correspond to the given names.

It's mostly an implementation detail, but the composite value type is a namedtuple. Users should not write to the composite *value* attribute since it is exclusively managed by the context.

Synopsis:

```
>>> async_x, async_y = AsyncValue(-1), AsyncValue(10)
>>>
>>> async with compose_values(x=async_x, y=async_y) as async_xy:
>>>     result = await async_xy.wait_value(lambda val: val.x < 0 < val.y)
>>>
>>> result
CompositeValue(x=-1, y=10)
```


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).

GENERATORS

async for elapsed, delta in 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*.

@trio_util.trio_async_generator

async generator pattern which supports Trio nurseries and cancel scopes

Decorator which adapts an async generator using Trio constructs for safe use. (Normally, it's not allowed to yield from a nursery or cancel scope when implementing async generators.)

Though the wrapped function is written as a normal async generator, usage of the wrapper is different: the wrapper is an async context manager providing the async generator to be iterated.

Synopsis:

```
>>> @trio_async_generator  
>>> async def my_generator():  
>>>     # yield values, possibly from a nursery or cancel scope  
>>>     # ...  
>>>  
>>>  
>>> async with my_generator() as agen:  
>>>     async for value in agen:  
>>>         print(value)
```

Implementation: “The idea is that instead of pushing and popping the generator from the stack of the task that’s consuming it, you instead run the generator code as a second task that feeds the consumer task values.” See <https://github.com/python-trio/trio/issues/638#issuecomment-431954073>

ISSUE: pylint is confused by this implementation, and every use will trigger not-async-context-manager

ITERATORS

await `trio_util.azip(*iterables)`

async version of `izip` with parallel iteration

await `trio_util.azip_longest(*iterables, fillvalue=None)`

async version of `izip_longest` with parallel iteration

EXCEPTIONS

with `trio_util.multi_error_defer_to` (*privileged_types: *Type[BaseException]*, *propagate_multi_error=True, strict=True*)

Defer a `trio.MultiError` exception to a single, privileged exception

In the scope of this context manager, a raised `MultiError` will be coalesced into a single exception with the highest privilege if the following criteria is met:

1. every exception in the `MultiError` is an instance of one of the given privileged types

additionally, by default with `strict=True`:

2. there is a single candidate at the highest privilege after grouping the exceptions by `repr()`. For example, this test fails if both `ValueError('foo')` and `ValueError('bar')` are the most privileged.

If the criteria are not met, by default the original `MultiError` is propagated. Use `propagate_multi_error=False` to instead raise a `RuntimeError` in these cases.

Examples:

```
multi_error_defer_to(trio.Cancelled, MyException)
    MultiError([Cancelled(), MyException()]) -> Cancelled()
    MultiError([Cancelled(), MyException(),
                MultiError([Cancelled(), Cancelled()])]) -> Cancelled()
    MultiError([Cancelled(), MyException(), ValueError()]) -> *no change*
    MultiError([MyException('foo'), MyException('foo')]) -> MyException('foo')
    MultiError([MyException('foo'), MyException('bar')]) -> *no change*

multi_error_defer_to(MyImportantException, trio.Cancelled, MyBaseException)
    # where isinstance(MyDerivedException, MyBaseException)
    # and isinstance(MyImportantException, MyBaseException)
    MultiError([Cancelled(), MyDerivedException()]) -> Cancelled()
    MultiError([MyImportantException(), Cancelled()]) -> MyImportantException()
```

Parameters

- **privileged_types** – exception types from highest priority to lowest
- **propagate_multi_error** – if false, raise a `RuntimeError` where a `MultiError` would otherwise be leaked
- **strict** – propagate `MultiError` if there are multiple output exceptions to chose from (i.e. multiple exceptions objects with differing `repr()` are instances of the privileged type). When combined with `propagate_multi_error=False`, this case will raise a `RuntimeError`.

with `trio_util.defer_to_cancelled` (*args: *Type[Exception]*)

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.

Equivalent to `multi_error_defer_to(trio.Cancelled, *args)`.

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)
        ...
```


MISCELLANEOUS

```
class trio_util.TaskStats(* , slow_task_threshold=0.01, current_time=<function current_time>)  
    Bases: trio.abc.Instrument
```

Trio scheduler Instrument which logs various task stats at termination.

Includes max task wait time, slow task steps, and highest task schedule rate.

INDEX

A

`AsyncBool` (*class in trio_util*), 4
`AsyncValue` (*class in trio_util*), 3
`azip()` (*in module trio_util*), 9
`azip_longest()` (*in module trio_util*), 9

C

`compose_values()` (*in module trio_util*), 4

D

`defer_to_cancelled()` (*in module trio_util*), 11

M

`MailboxRepeatedEvent` (*class in trio_util*), 5
`multi_error_defer_to()` (*in module trio_util*), 11

P

`periodic()` (*in module trio_util*), 7

S

`set()` (*trio_util.MailboxRepeatedEvent method*), 6
`set()` (*trio_util.UnqueuedRepeatedEvent method*), 5

T

`TaskStats` (*class in trio_util*), 13
`trio_async_generator()` (*in module trio_util*), 7

U

`UnqueuedRepeatedEvent` (*class in trio_util*), 5

V

`value()` (*trio_util.AsyncValue property*), 3

W

`wait_all()` (*in module trio_util*), 1
`wait_any()` (*in module trio_util*), 1
`wait_transition()` (*trio_util.AsyncValue method*),
3
`wait_value()` (*trio_util.AsyncValue method*), 3