trio-util

Release 0.8.0-dev

GROOVE X, Inc.

Apr 05, 2022

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 set:

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

or wait until both events are set:

```
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.

It may be tempting to use *wait_any()* where you have some long-running, async code that should be interrupted by an event. In the context of a function scope, you would likely structure it by defining a local function for the long-running code:

```
async def foo():
    # NOT recommended
    async def _my_endless_task():
        while True:
            await trio.sleep(5)
            if some_condition:
                counter += 1  # bug (missing `nonlocal` definition)
            elif other_condition:
                return  # maybe-bug: you intended to exit `foo()`?
```

(continues on next page)

(continued from previous page)

```
    counter = 0
    # run the endless async loop until some_event triggers
    await wait_any(_my_endless_task, some_event.wait)
    if counter > 10: ...
```

By writing the long-running code as a separate function rather than inline, we're interfering with the natural flow of control (can't write to local vars or use `continue` / `break` / `return` in the context of `foo()`). For this we have something better: *`move_on_when()`*.

In the spirit of Trio's `move_on_after()` cancel scope utility, *`move_on_when()`* represents a block of async code that can be interrupted by any async event. By letting the primary "task" be written inline, the code has access to everything wonderful about `foo()`'s local scope:

```
async def foo():
    counter = 0
    async with move_on_when(some_event.wait):
        while True:
            await trio.sleep(5)
            if some_condition:
                counter += 1
            elif other_condition:
                return
    if counter > 10: ...
```

async with trio_util.**move_on_when**(*fn*, **args*, ***kwargs*) **as cancel_scope**

Async context manager that exits if async fn(*args, **kwargs) returns.

The context manager yields a trio.CancelScope.

Synopsis:

```
async with move_on_when(my_event.wait) as cancel_scope:
    cancel_scope.shield = True
    await ...

# by this point either the body exited, my_event was triggered, or both
```

`run_and_cancelling()`, on the other hand, is a context manager that runs a background task that is not able to interrupt your block of code. At the end of the block, the background task is cancelled if necessary.

async with trio_util.**run_and_cancelling**(*fn*, **args*, ***kwargs*)

Async context manager that runs async fn(*args, **kwargs) and cancels it at block exit.

Synopsis:

```
async with run_and_cancelling(my_background_fn, my_arg=10):
    await ...
    # now the block exits, and my_background_fn is cancelled if still running
```

CHAPTER

TWO

VALUE WRAPPERS

AsyncValue can wrap any type, offering the ability to wait for a specific value or transition. It supports various broadcast and "pubsub" patterns, composition and transformation of values, and synchronizing values with eventual consistency.

Here's a quick example based on this real use case posted to one of Trio's forums:

> I noticed how hard [writing state machines in Trio] becomes, especially when there are requirements like e.g. "when in state paused longer than X toggle to stopped"...

AsyncValue together with Trio's cancellation make it easy:

```
current_state = AsyncValue(States.INIT)
...

async def monitor_paused_too_long():
    while True:
        await current_state.wait_value(States.PAUSED)
        with trio.move_on_after(X):
            await current_state.wait_transition()  # any transition out of PAUSED
            continue
        current_state.value = States.STOPPED
```

(Note that the `while` loop and `wait_value()` combination can be replaced with `async for _ in current_state.eventual_values(States.PAUSED): ...`, but the code above is best for an introduction.)

How does AsyncValue work?

If you wanted to be notified of specific value changes, one way to implement things would be to relay *every* value change to listeners and have them implement the filtering locally. But *AsyncValue* does *not* take this approach because it can be fraught with issues like poor performance, queues backing up when there is an unresponsive listener, etc. Rather, listeners pass a predicate representing the values or transitions they're interested in, and the `value` property setter evaluates these *synchronously* and then broadcasts matches as events back to the listener. This is simple for listeners while being efficient and ensuring that important value changes aren't lost.

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
```

(continues on next page)

(continued from previous page)

```
>>> ...
>>> # 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)
>>> ...
>>> # values via iteration (with eventual consistency)
>>> async for value a.eventual_values(lambda v: v > 10):
>>>     ...
>>> # wait for any transition
>>> await a.wait_transition()
>>> ...
>>> # wait for transition by equality
>>> await a.wait_transition(14)
>>> ...
>>> # wait for transition by predicate
>>> await a.wait_transition(lambda v, old: v > 10 and old < 0)
>>> ...
>>> # repeated transitions via iteration (misses values while blocked)
>>> async for value, _ in a.transitions(lambda v, old: v > 10 and old < 0):
>>>     ...
```

When using any of the wait methods or iterators in this API, note that the value may have changed again before the caller receives control. For clarity, the specific value that triggered the wakeup is always returned.

Comparison of eventual_values() and transitions() iterators:

```
eventual_values()                          transitions()
=================                          =============
• high level & safe                        • low level & requires care
• evaluated on each value change           • evaluated on each value change
    and when starting loop                     if caller not blocked in the body
• eventually iterates latest value         • latest value missed if caller is blocked
• can miss rapid value changes             • can miss rapid value changes
• condition uses new value only            • condition uses new and/or old value
• mainly used for sync'ing state           • mainly used for triggering work
• not subject to user races                • races possible on init and each iteration
                                               (especially if value changes␣
↪infrequently)
```

Performance note: assignment to the *value* property typically has O(N) complexity, where N is the number of actively waiting tasks. Shared predicates are grouped when possible, reducing N to the number of active predicates.

value
: The wrapped value

await wait_value(*value_or_predicate*, *, *held_for=0.0*)
: 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)

async for value in eventual_values(*value_or_predicate=<function _ANY_VALUE>*, *held_for=0.0*)

Yield values matching the predicate with eventual consistency

The initial value will be yielded immediately if it matches the predicate. Subsequent yields occur whenever the *value* property changes to a value matching the predicate. Note that rapid changes as well as multiple changes while the caller body is blocked will not all be reflected, but eventual consistency is ensured.

The default predicate will match any value.

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.

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.

Note that unlike *wait_value()*, it is easy to have race conditions when using *wait_transition()*. Always consider whether it's possible for the initially desired transition to have already occurred due to task scheduling order, etc.

returns (value, old_value) which satisfied the predicate

async for value, old_value in transitions(*value_or_predicate=<function _ANY_TRANSITION>*)

Yield (value, old_value) for transitions matching the predicate

Transitions that happen during the body of the loop are discarded.

The iteration:

```
>>> async for value, old_value in async_value.transitions(...)
>>>     ...
```

is equivalent to:

```
>>> while True:
>>>     value, old_value = await async_value.wait_transition(...)
>>>     ...
```

Unlike the *eventual_values()* iterator, use of the *transitions()* is prone to races when entering the loop. Always consider whether it's possible for the desired transition to have already occurred due to task scheduling order, etc.

with open_transform(*function*) **as output**

Yield a derived AsyncValue with the given transform applied

Synopsis:

```
>>> x = AsyncValue(1)
>>> with x.open_transform(lambda val: val * 2) as y:
>>>     assert y.value == 2
>>>     x.value = 10
>>>     assert y.value == 20
```

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.

with trio_util.**compose_values**(*, *_transform_=None*, ***value_map*) **as composed**

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)
>>>
>>> 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)
```

The *_transform_* parameter specifies an optional function to transform the final value. This is equivalent but more efficient than chaining a single open_transform() to the default compose_values() output. For example:

```
>>> with compose_values(x=async_x, y=async_y,
>>>                     _transform_=lambda val: val.x * val.y) as x_mul_y:
>>>     ...
```

is equivalent to:

```
>>> with compose_values(x=async_x, y=async_y) as async_xy, \
>>>         async_xy.open_transform(lambda val: val.x * val.y) as x_mul_y:
>>>     ...
```

Performance note: predicates on the output AsyncValue will be evaluated on every assignment to the *value* properties of the input AsyncValues. So if two inputs are being composed, each updated 10 times per second, the output predicates will be evaluated 20 times per second.

CHAPTER

THREE

REPEATED EVENTS

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

RepeatedEvent can be triggered repeatedly in a relatively safe manner while having multiple listeners.

class trio_util.**RepeatedEvent**

A repeated event that supports multiple listeners.

RepeatedEvent supports both "unqueued" and "eventual consistency" uses:

- unqueued - drop events while processing the previous one
- eventual consistency - some events may be missed while processing the previous one, but receiving the latest event is ensured

set()

Trigger an event

await wait()

Wait for the next event

async for ... in unqueued_events()

Unqueued event iterator

The listener will miss an event if it's blocked processing the previous one. This is effectively the same as the following manual loop:

```
>>> while True:
>>>     await event.wait()
>>>     # do work...
```

Typical usage:

```
>>> event = RepeatedEvent()

A task listens for events:

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

Another task triggers events:

>>> event.set()     # trigger event
>>> trio.sleep(0)   # listener will enter loop body
>>> event.set()     # listener misses this event since it's still in the loop␣
↪body
```

(continues on next page)

(continued from previous page)

```
>>> trio.sleep(2)
>>> event.set()     # listener will enter loop body again
```

async for ... in events(*, *repeat_last=False*)

Event iterator with eventual consistency

Use this iterator to coordinate some work whenever a collection or other stateful object is mutated. Although you may miss intermediate states, you're ensured to eventually receive an event to process the most recent state. (https://en.wikipedia.org/wiki/Eventual_consistency)

Parameters **repeat_last** – if true, repeat the last position in the event stream. If no event has been set yet it still yields immediately, representing the "start" position.

Typical usage:

```
>>> my_list = []
>>> repeated_event = RepeatedEvent()

Whenever your collection is mutated, call the `set()` method.

>>> my_list.append('hello')
>>> repeated_event.set()

The listener to continually process the latest state is:

>>> async for _ in repeated_event.events():
>>>     await persist_to_storage(my_list)

If you'd like to persist the initial state of the list (before any
set() is called), use the `repeat_last=True` option.
```

CHAPTER

FOUR

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

Normally, it's not allowed to yield from a Trio nursery or cancel scope when implementing async generators. This decorator makes it possible to do so, adapting a generator for safe use.

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

CHAPTER

FIVE

ITERATORS

When working with an asynchronous iterator, you may want to cancel iteration or raise an error when a single iteration takes too long. *iter_move_on_after()* and *iter_fail_after()* can wrap an iterator to provide this.

await trio_util.**iter_move_on_after**(*timeout*, *ait*)
async iterator adapter that stops if an iteration exceeds timeout

The timeout is a duration in seconds.

Synopsis:

```
async for v in iter_move_on_after(5, async_value.eventual_values()):
    ...
```

await trio_util.**iter_fail_after**(*timeout*, *ait*)
async iterator adapter that raises trio.TooSlowError if an iteration exceeds timeout

The timeout is a duration in seconds.

Synopsis:

```
async for v in iter_fail_after(5, async_value.eventual_values()):
    ...
```

azip() and *azip_longest()* are async equivalents of `zip()` and `itertools.zip_longest()`.

await trio_util.**azip**(**aiterables*)
async version of zip() with parallel iteration

await trio_util.**azip_longest**(**aiterables*, *fillvalue=None*)
async version of zip_longest() with parallel iteration

CHAPTER

SIX

EXCEPTIONS

with trio_util.**multi_error_defer_to**(**privileged_types*, *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*)

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)
        ...
```

CHAPTER

SEVEN

MISCELLANEOUS

class trio_util.**TaskStats**(*, *slow_task_threshold=0.01*, *high_rate_task_threshold=100*, *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