aiotools Documentation

Release 1.8.0.dev5+g1c2a6e2

Joongi Kim

Aug 26, 2023

CONTENTS

aiotools is a set of idiomatic utilities to reduce asyncio boiler-plates.

CHAPTER

ONE

ASYNC CONTEXT MANAGER

Provides an implementation of asynchronous context manager and its applications.

class AbstractAsyncContextManager

An abstract base class for asynchronous context managers.

AsyncContextManager

alias of `contextlib._AsyncGeneratorContextManager`

async_ctx_manager(*func*)

A helper function to ease use of `AsyncContextManager`.

actxmgr(*func*)

An alias of `async_ctx_manager()`.

class aclosing(*thing: aiotools.context.T_AClosable*)

An analogy to `contextlib.closing()` for async generators.

The motivation has been proposed by:

- https://github.com/njsmith/async_generator
- https://vorpus.org/blog/some-thoughts-on-asynchronous-api-design-in-a-post-asyncawait-world/#cleanup-in-generators-and-async-generators
- https://www.python.org/dev/peps/pep-0533/

class closing_async(*thing: aiotools.context.T_AsyncClosable*)

An analogy to `contextlib.closing()` for objects defining the `close()` method as an async function.

New in version 1.5.6.

class AsyncContextGroup(*context_managers: Optional[Iterable[*contextlib.AbstractAsyncContextManager*]] = None*)

Merges a group of context managers into a single context manager. Internally it uses `asyncio.gather()` to execute them with overlapping, to reduce the execution time via asynchrony.

Upon entering, you can get values produced by the entering steps from the passed context managers (those `yield`-ed) using an `as` clause of the `async with` statement.

After exits, you can check if the context managers have finished successfully by ensuring that the return values of `exit_states()` method are `None`.

Note: You cannot return values in context managers because they are generators.

If an exception is raised before the `yield` statement of an async context manager, it is stored at the corresponding manager index in the as-clause variable. Similarly, if an exception is raised after the `yield` statement of an async context manager, it is stored at the corresponding manager index in the `exit_states()` return value.

Any exceptions in a specific context manager does not interrupt others; this semantic is same to `asyncio.gather()`'s when `return_exceptions=True`. This means that, it is user's responsibility to check if the returned context values are exceptions or the intended ones inside the context body after entering.

Parameters **`context_managers`** – An iterable of async context managers. If this is `None`, you may add async context managers one by one using the *`add()`* method.

Example:

```
@aiotools.actxmgr
async def ctx(v):
  yield v + 10

g = aiotools.actxgroup([ctx(1), ctx(2)])

async with g as values:
    assert values[0] == 11
    assert values[1] == 12

rets = g.exit_states()
assert rets[0] is None  # successful shutdown
assert rets[1] is None
```

`add`(*cm*)

TODO: fill description

`exit_states`()

TODO: fill description

`class` **`actxgroup`**

An alias of *`AsyncContextGroup`*.

CHAPTER

TWO

ASYNC DEFERRED FUNCTION TOOLS

Provides a Golang-like `defer()` API using decorators, which allows grouping resource initialization and cleanup in one place without extra indentations.

Example:

```
async def init(x):
    ...

async def cleanup(x):
    ...

@aiotools.adefer
async def do(defer):  # <-- be aware of defer argument!
    x = SomeResource()
    await init(x)
    defer(cleanup(x))
    ...
    ...
```

This is equivalent to:

```
async def do():
    x = SomeResource()
    await init(x)
    try:
        ...
        ...
    finally:
        await cleanup(x)
```

Note that *`aiotools.context.AsyncContextGroup`* or `contextlib.AsyncExitStack` serves well for the same purpose, but for simple cleanups, this defer API makes your codes simple because it steps aside the main execution context without extra indentations.

Warning: Any exception in the deferred functions is raised transparently, and may block execution of the remaining deferred functions. This behavior may be changed in the future versions, though.

`defer`(*func*)

A synchronous version of the defer API. It can only defer normal functions.

`adefer`(*func*)

An asynchronous version of the defer API. It can defer coroutine functions, coroutines, and normal functions.

CHAPTER

THREE

ASYNC FORK

This module implements a simple `os.fork()`-like interface, but in an asynchronous way with full support for PID file descriptors on Python 3.9 or higher and the Linux kernel 5.4 or higher.

It internally synchronizes the beginning and readiness status of child processes so that the users may assume that the child process is completely interruptible after `afork()` returns.

class AbstractChildProcess

The abstract interface to control and monitor a forked child process.

abstract send_signal(*signum: int*) → None

Send a UNIX signal to the child process. If the child process is already terminated, it will log a warning message and return.

abstract async wait() → int

Wait until the child process terminates or reclaim the child process' exit code if already terminated. If there are other coroutines that has waited the same process, it may return 255 and log a warning message.

PosixChildProcess(*pid: int*) → None

A POSIX-compatible version of `AbstractChildProcess`.

PidfdChildProcess(*pid: int*, *pidfd: int*) → None

A PID file descriptor-based version of `AbstractChildProcess`.

async afork(*child_func: Callable[[], int]*) → *aiotools.fork.AbstractChildProcess*

Fork the current process and execute the given function in the child. The return value of the function will become the exit code of the child process.

Parameters **child_func** – A function that represents the main function of the child and returns an integer as its exit code. Note that the function must set up a new event loop if it wants to run asyncio codes.

CHAPTER FOUR

ASYNC FUNCTION TOOLS

`apartial`(*coro*, **args*, ***kwargs*)

Wraps a coroutine function with pre-defined arguments (including keyword arguments). It is an asynchronous version of `functools.partial()`.

`lru_cache`(*maxsize: int = 128*, *typed: bool = False*, *expire_after: Optional[float] = None*)

A simple LRU cache just like `functools.lru_cache()`, but it works for coroutines. This is not as heavily optimized as `functools.lru_cache()` which uses an internal C implementation, as it targets async operations that take a long time.

It follows the same API that the standard functools provides. The wrapped function has `cache_clear()` method to flush the cache manually, but leaves `cache_info()` for statistics unimplemented.

Note that calling the coroutine multiple times with the same arguments before the first call returns may incur duplicate executions.

This function is not thread-safe.

Parameters

- **`maxsize`** – The maximum number of cached entries.
- **`typed`** – Cache keys in different types separately (e.g., `3` and `3.0` will be different keys).
- **`expire_after`** – Re-calculate the value if the configured time has passed even when the cache is hit. When re-calculation happens the expiration timer is also reset.

CHAPTER

FIVE

ASYNC ITERTOOLS

`async aiter`(*obj*, *sentinel=<object object>*)

Analogous to the builtin `iter()`.

CHAPTER

SIX

MULTI-PROCESS SERVER

Based on *Async Context Manager*, this module provides an automated lifecycle management for multi-process servers with explicit initialization steps and graceful shutdown steps.

server(*func*)

A decorator wrapper for *AsyncServerContextManager*.

Usage example:

```
@aiotools.server
async def myserver(loop, pidx, args):
    await do_init(args)
    stop_sig = yield
    if stop_sig == signal.SIGINT:
        await do_graceful_shutdown()
    else:
        await do_forced_shutdown()

aiotools.start_server(myserver, ...)
```

class AsyncServerContextManager(*func: Callable[[...], Any], args, kwargs*)

A modified version of `contextlib.asynccontextmanager()`.

The implementation detail is mostly taken from the `contextlib` standard library, with a minor change to inject `self.yield_return` into the wrapped async generator.

yield_return: Optional[signal.Signals]

exception InterruptedBySignal

A new `BaseException` that represents interruption by an arbitrary UNIX signal.

Since this is a `BaseException` instead of `Exception`, it behaves like `KeyboardInterrupt` and `SystemExit` exceptions (i.e., bypassing except clauses catching the `Exception` type only)

The first argument of this exception is the signal number received.

class ServerMainContextManager(*func, args, kwargs*)

A modified version of `contextlib.contextmanager()`.

The implementation detail is mostly taken from the `contextlib` standard library, with a minor change to inject `self.yield_return` into the wrapped generator.

yield_return: Optional[signal.Signals]

main(*func*)

A decorator wrapper for *ServerMainContextManager*

Usage example:

```
@aiotools.main
def mymain():
    server_args = do_init()
    stop_sig = yield server_args
    if stop_sig == signal.SIGINT:
        do_graceful_shutdown()
    else:
        do_forced_shutdown()

aiotools.start_server(..., main_ctxmgr=mymain, ...)
```

start_server(*worker_actxmgr: typing.Callable[[asyncio.events.AbstractEventLoop, int, typing.Sequence[typing.Any]], aiotools.server.AsyncServerContextManager], main_ctxmgr: typing.Optional[typing.Callable[[], aiotools.server.ServerMainContextManager]] = None, extra_procs: typing.Iterable[typing.Callable] = (), stop_signals: typing.Iterable[signal.Signals] = (<Signals.SIGINT: 2>, <Signals.SIGTERM: 15>), num_workers: int = 1, args: typing.Iterable[typing.Any] = (), wait_timeout: typing.Optional[float] = None*) → None

Starts a multi-process server where each process has their own individual asyncio event loop. Their lifecycles are automantically managed – if the main program receives one of the signals specified in `stop_signals` it will initiate the shutdown routines on each worker that stops the event loop gracefully.

Parameters

- **`worker_actxmgr`** – An asynchronous context manager that dicates the initialization and shutdown steps of each worker. It should accept the following three arguments:
  - **loop**: the asyncio event loop created and set by aiotools
  - **pidx**: the 0-based index of the worker (use this for per-worker logging)
  - **args**: a concatenated tuple of values yielded by **main_ctxmgr** and the user-defined arguments in **args**.

  aiotools automatically installs an interruption handler that calls `loop.stop()` to the given event loop, regardless of using either threading or multiprocessing.
- **`main_ctxmgr`** – An optional context manager that performs global initialization and shutdown steps of the whole program. It may yield one or more values to be passed to worker processes along with **args** passed to this function. There is no arguments passed to those functions since you can directly access `sys.argv` to parse command line arguments and/or read user configurations.
- **`extra_procs`** – An iterable of functions that consist of extra processes whose lifecycles are synchronized with other workers. They should set up their own signal handlers.

  It should accept the following three arguments:
  - **intr_event**: Always `None`, kept for legacy
  - **pidx**: same to **worker_actxmgr** argument
  - **args**: same to **worker_actxmgr** argument
- **`stop_signals`** – A list of UNIX signals that the main program to recognize as termination signals.

- `num_workers` – The number of children workers.
- `args` – The user-defined arguments passed to workers and extra processes. If **main_ctxmgr** yields one or more values, they are *prepended* to this user arguments when passed to workers and extra processes.
- `wait_timeout` – The timeout in seconds before forcibly killing all remaining child processes after sending initial stop signals.

Returns None

Changed in version 0.3.2: The name of argument **num_proc** is changed to **num_workers**. Even if **num_workers** is 1, a child is created instead of doing everything at the main thread.

New in version 0.3.2: The argument `extra_procs` and `main_ctxmgr`.

New in version 0.4.0: Now supports use of threading instead of multiprocessing via **use_threading** option.

Changed in version 0.8.0: Now **worker_actxmgr** must be an instance of *`AsyncServerContextManager`* or async generators decorated by `@aiotools.server`.

Now **main_ctxmgr** must be an instance of *`ServerMainContextManager`* or plain generators decorated by `@aiotools.main`.

The usage is same to asynchronous context managers, but optionally you can distinguish the received stop signal by retrieving the return value of the `yield` statement.

In **extra_procs** in non-threaded mode, stop signals are converted into either one of `KeyboardInterrupt`, `SystemExit`, or *`InterruptedBySignal`* exception.

New in version 0.8.4: **start_method** argument can be set to change the subprocess spawning implementation.

Deprecated since version 1.2.0: The **start_method** and **use_threading** arguments, in favor of our new `afork()` function which provides better synchronization and pid-fd support.

Changed in version 1.2.0: The **extra_procs** will be always separate processes since **use_threading** is deprecated and thus **intr_event** arguments are now always `None`.

New in version 1.5.5: The **wait_timeout** argument.

CHAPTER

SEVEN

TASK GROUP

`current_taskgroup`

A `contextvars.ContextVar` that has the reference to the current innermost *TaskGroup* instance. Available only in Python 3.7 or later.

`current_ptaskgroup`

A `contextvars.ContextVar` that has the reference to the current innermost *PersistentTaskGroup* instance. Available only in Python 3.7 or later.

Warning: This is set only when *PersistentTaskGroup* is used with the `async with` statement.

class TaskGroup(*, *name=None*)

Provides a guard against a group of tasks spawend via its *create_task()* method instead of the vanilla fire-and-forgetting `asyncio.create_task()`.

See the motivation and rationale in the trio's documentation.

In Python 3.11 or later, this wraps `asyncio.TaskGroup` with a small extension to set the current taskgroup in a context variable.

create_task(*coro*, *, *name=None*)

Spawns a new task inside the taskgroup and returns the reference to the task. Setting the name of tasks is supported in Python 3.8 or later only and ignored in older versions.

get_name()

Returns the name set when creating the instance.

New in version 1.0.0.

Changed in version 1.5.0: Fixed edge-case bugs by referring the Python 3.11 stdlib's `asyncio.TaskGroup` implementation, including abrupt cancellation before all nested spawned tasks start without context switches and propagation of the source exception when the context manager (parent task) is getting cancelled but continued. All existing codes should run without any issues, but it is recommended to test thoroughly.

class PersistentTaskGroup(*, *name=None*, *exception_handler=None*)

Provides an abstraction of long-running task groups for server applications. The main use case is to implement a dispatcher of async event handlers, to group RPC/API request handlers, etc. with safe and graceful shutdown. Here "long-running" means that all tasks should keep going even when sibling tasks fail with unhandled errors and such errors must be reported immediately. Here "safety" means that all spawned tasks should be reclaimed before exit or shutdown.

When used as an async context manager, it works similarly to `asyncio.gather()` with `return_exceptions=True` option. It exits the context scope when all tasks finish, just like `asyncio.`

`TaskGroup`, but it does NOT abort when there are unhandled exceptions from child tasks; just keeps sibling tasks running and reporting errors as they occur (see below).

When *not* used as an async context maanger (e.g., used as attributes of long-lived objects), it persists running until *`shutdown()`* is called explicitly. Note that it is the user's responsibility to call *`shutdown()`* because *`PersistentTaskGroup`* does not provide the `__del__()` method.

Regardless how it is executed, it lets all spawned tasks run to their completion and calls the exception handler to report any unhandled exceptions immediately. If there are exceptions occurred again in the exception handlers, then it uses `loop.call_exception_handler()` as the last resort.

exception_handler should be an asynchronous function that accepts the exception type, exception object, and the traceback, just like `__aexit__()` dunder method. The default handler just prints out the exception log using `traceback.print_exc()`. Note that the handler is invoked within the exception handling context and thus `sys.exc_info()` is also available.

Since the exception handling and reporting takes places immediately, it eliminates potential arbitrary report delay due to other tasks or the execution method. This resolves a critical debugging pain when only termination of the application displays accumulated errors, as sometimes we don't want to terminate but just inspect what is happening.

`create_task`(*coro*, *, *name=None*)

Spawns a new task inside the taskgroup and returns the reference to a `future` describing the task result. Setting the name of tasks is supported in Python 3.8 or later only and ignored in older versions.

You may `await` the retuned future to take the task's return value or get notified with the exception from it, while the exception handler is still invoked. Since it is just a *secondary* future, you cannot cancel the task explicitly using it. To cancel the task(s), use *`shutdown()`* or exit the task group context.

Warning: In Python 3.6, `await`-ing the returned future hangs indefinitely. We do not fix this issue because Python 3.6 is now EoL (end-of-life) as of December 2021.

`get_name`()

Returns the name set when creating the instance.

`async shutdown`()

Triggers immediate shutdown of this taskgroup, cancelling all unfinished tasks and waiting for their completion.

`classmethod all_ptaskgroups`()

Returns a sequence of all currently existing non-exited persistent task groups.

New in version 1.5.0.

New in version 1.4.0.

Changed in version 1.5.0: Rewrote the overall implementation referring the Python 3.11 stdlib's `asyncio.TaskGroup` implementation and adapting it to the semantics for "persistency". All existing codes should run without any issues, but it is recommended to test thoroughly.

Changed in version 1.6.1: It no longer raises `BaseExceptionGroup` or `ExceptionGroup` upon exit or *`shutdown()`*, because it no longer stores the history of unhnadled exceptions from subtasks to prevent memory leaks for long-running persistent task groups. The users must register explicit exception handlers or task done callbacks to report or process such unhandled exceptions.

`exception TaskGroupError`

Represents a collection of errors raised inside a task group. Callers may iterate over the errors using the `__errors__` attribute.

In Python 3.11 or later, this is a mere wrapper of underlying `BaseExceptionGroup`. This allows existing user codes to run without modification while users can take advantage of the new `except*` syntax and `ExceptionGroup` methods if they use Python 3.11 or later. Note that if none of the passed exceptions passed is a `BaseException`, it automatically becomes `ExceptionGroup`.

CHAPTER

EIGHT

TASK GROUP UTILITIES

A set of helper utilities to utilize taskgroups in better ways.

`async as_completed_safe(`*coros*, *timeout=None*`)`

This is a safer version of `asyncio.as_completed()` which uses *`PersistentTaskGroup`* as an underlying coroutine lifecycle keeper.

Upon a timeout, it raises `asyncio.TimeoutError` immediately and cancels all remaining tasks or coroutines.

This requires Python 3.11 or higher to work properly with timeouts.

New in version 1.6.

CHAPTER NINE

TIMERS

Provides a simple implementation of timers run inside asyncio event loops.

class TimerDelayPolicy(*value*)

An enumeration of supported policies for when the timer function takes longer on each tick than the given timer interval.

`CANCEL = 1`

`DEFAULT = 0`

class VirtualClock

Provide a virtual clock for an asyncio event loop which makes timing-based tests deterministic and instantly completed.

patch_loop()

Override some methods of the current event loop so that sleep instantly returns while proceeding the virtual clock.

virtual_time() → float

Return the current virtual time.

create_timer(*cb: Callable[[float], Union[Generator[Any, None, None], Coroutine[Any, Any, None]]], interval: float*, *delay_policy:* aiotools.timer.TimerDelayPolicy *= TimerDelayPolicy.DEFAULT*, *loop: Optional[asyncio.events.AbstractEventLoop] = None*) → _asyncio.Task

Schedule a timer with the given callable and the interval in seconds. The interval value is also passed to the callable. If the callable takes longer than the timer interval, all accumulated callable's tasks will be cancelled when the timer is cancelled.

Parameters `cb` – TODO - fill argument descriptions

Returns You can stop the timer by cancelling the returned task.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX