discord.py

v: stable

discord.ext.tasks – asyncio.Task helpers

New in version 1.1.0.

One of the most common operations when making a bot is having a loop run in the background at a specified interval. This pattern is very common but has a lot of things you need to look out for:

- How do I handle `asyncio.CancelledError`?
- What do I do if the internet goes out?
- What is the maximum number of seconds I can sleep anyway?

The goal of this discord.py extension is to abstract all these worries away from you.

Recipes

A simple background task in a `Cog`:

```python
from discord.ext import tasks, commands

class MyCog(commands.Cog):
    def __init__(self):
        self.index = 0
        self.printer.start()

    def cog_unload(self):
        self.printer.cancel()

    @tasks.loop(seconds=5.0)
    async def printer(self):
        print(self.index)
        self.index += 1
```

Adding an exception to handle during reconnect:

```python
import asyncpg
from discord.ext import tasks, commands

class MyCog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
```

v: stable

```
        self.data = []
        self.batch_update.add_exception_type(asyncpg.PostgresConnectionError)
        self.batch_update.start()

    def cog_unload(self):
        self.batch_update.cancel()

    @tasks.loop(minutes=5.0)
    async def batch_update(self):
        async with self.bot.pool.acquire() as con:
            # batch update here...
            pass
```

Looping a certain amount of times before exiting:

```
from discord.ext import tasks
import discord

@tasks.loop(seconds=5.0, count=5)
async def slow_count():
    print(slow_count.current_loop)

@slow_count.after_loop
async def after_slow_count():
    print('done!')

class MyClient(discord.Client):
    async def setup_hook(self):
        slow_count.start()
```

Waiting until the bot is ready before the loop starts:

```
from discord.ext import tasks, commands

class MyCog(commands.Cog):
    def __init__(self, bot):
        self.index = 0
        self.bot = bot
        self.printer.start()

    def cog_unload(self):
        self.printer.cancel()

    @tasks.loop(seconds=5.0)
    async def printer(self):
        print(self.index)
        self.index += 1
```

v: stable

```python
    @printer.before_loop
    async def before_printer(self):
        print('waiting...')
        await self.bot.wait_until_ready()
```

Doing something during cancellation:

```python
from discord.ext import tasks, commands
import asyncio

class MyCog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
        self._batch = []
        self.lock = asyncio.Lock()
        self.bulker.start()

    async def cog_unload(self):
        self.bulker.cancel()

    async def do_bulk(self):
        # bulk insert data here
        ...

    @tasks.loop(seconds=10.0)
    async def bulker(self):
        async with self.lock:
            await self.do_bulk()

    @bulker.after_loop
    async def on_bulker_cancel(self):
        if self.bulker.is_being_cancelled() and len(self._batch) != 0:
            # if we're cancelled and we have some data left...
            # let's insert it to our database
            await self.do_bulk()
```

Doing something at a specific time each day:

```python
import datetime
from discord.ext import commands, tasks

utc = datetime.timezone.utc

# If no tzinfo is given then UTC is assumed.
time = datetime.time(hour=8, minute=30, tzinfo=utc)
```

v: stable

```python
class MyCog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
        self.my_task.start()

    def cog_unload(self):
        self.my_task.cancel()

    @tasks.loop(time=time)
    async def my_task(self):
        print("My task is running!")
```

Doing something at multiple specific times each day:

```python
import datetime
from discord.ext import commands, tasks

utc = datetime.timezone.utc

# If no tzinfo is given then UTC is assumed.
times = [
    datetime.time(hour=8, tzinfo=utc),
    datetime.time(hour=12, minute=30, tzinfo=utc),
    datetime.time(hour=16, minute=40, second=30, tzinfo=utc)
]

class MyCog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
        self.my_task.start()

    def cog_unload(self):
        self.my_task.cancel()

    @tasks.loop(time=times)
    async def my_task(self):
        print("My task is running!")
```

API Reference

class `discord.ext.tasks.`**`Loop`**

v: stable

Attributes

- current_loop
- hours

Methods

- async __call__

A background task helper that abstracts the loop and reconnection logic for you.

The main interface to create this is through `loop()` .

@**after_loop**

A decorator that registers a coroutine to be called after the loop finishes running.

The coroutine must take no arguments (except `self` in a class context).

Note

This coroutine is called even during cancellation. If it is desirable to tell apart whether something was cancelled or not, check to see whether `is_being_cancelled()` is `True` or not.

Parameters:

coro (coroutine) – The coroutine to register after the loop finishes.

Raises:

`TypeError` – The function was not a coroutine.

@**before_loop**

A decorator that registers a coroutine to be called before the loop starts running.

This is useful if you want to wait for some bot state before the loop starts, such as `discord.Client.wait_until_ready()` .

The coroutine must take no arguments (except `self` in a class context).

Changed in version 2.0: Calling `stop()` in this coroutine will stop the loop v: stable
initial iteration is run.

Parameters:

coro (coroutine) – The coroutine to register before the loop runs.

Raises:

TypeError – The function was not a coroutine.

`@error`

A decorator that registers a coroutine to be called if the task encounters an unhandled exception.

The coroutine must take only one argument the exception raised (except `self` in a class context).

By default this logs to the library logger however it could be overridden to have a different implementation.

New in version 1.4.

Changed in version 2.0: Instead of writing to `sys.stderr`, the library's logger is used.

Parameters:

coro (coroutine) – The coroutine to register in the event of an unhandled exception.

Raises:

TypeError – The function was not a coroutine.

property **seconds**

Read-only value for the number of seconds between each iteration. `None` if an explicit `time` value was passed instead.

New in version 2.0.

Type:

Optional[`float`]

property **minutes**

Read-only value for the number of minutes between each iteration. `None` if an explicit `time` value was passed instead.

New in version 2.0.

Type:

Optional[`float`]

property **hours**

Read-only value for the number of hours between each iteration. `None` if an explicit `time` value was passed instead.

v: stable

New in version 2.0.

Type:

Optional[`float`]

property **`time`**

Read-only list for the exact times this loop runs at. `None` if relative times were passed instead.

New in version 2.0.

Type:

Optional[List[`datetime.time`]]

property **`current_loop`**

The current iteration of the loop.

Type:

`int`

property **`next_iteration`**

When the next iteration of the loop will occur.

New in version 1.3.

Type:

Optional[`datetime.datetime`]

await **`__call__`**(**args*, ***kwargs*)

This function is a *coroutine*.

Calls the internal callback that the task holds.

New in version 1.6.

Parameters:

- ***args** – The arguments to use.
- ****kwargs** – The keyword arguments to use.

`start`(**args*, ***kwargs*)

Starts the internal task in the event loop.

Parameters:

- ***args** – The arguments to use.
- ****kwargs** – The keyword arguments to use.

Raises:

`RuntimeError` – A task has already been launched and is running. v: stable

Returns:

The task that has been created.

Return type:

`asyncio.Task`

`stop()`

Gracefully stops the task from running.

Unlike `cancel()`, this allows the task to finish its current iteration before gracefully exiting.

> **Note**
>
> If the internal function raises an error that can be handled before finishing then it will retry until it succeeds.
>
> If this is undesirable, either remove the error handling before stopping via `clear_exception_types()` or use `cancel()` instead.

Changed in version 2.0: Calling this method in `before_loop()` will stop the loop before the initial iteration is run.

New in version 1.2.

`cancel()`

Cancels the internal task, if it is running.

`restart(*args, **kwargs)`

A convenience method to restart the internal task.

> **Note**
>
> Due to the way this function works, the task is not returned like `start()`.

Parameters:

- ***args** – The arguments to use.
- ****kwargs** – The keyword arguments to use.

`add_exception_type(*exceptions)`

Adds exception types to be handled during the reconnect logic.

By default the exception types handled are those handled by `discord.Client.connect()`, which includes a lot of internet disconnection errors.

v: stable

This function is useful if you're interacting with a 3rd party library that raises its own set of exceptions.

Parameters:

***exceptions** (Type[`BaseException`]) – An argument list of exception classes to handle.

Raises:

`TypeError` – An exception passed is either not a class or not inherited from `BaseException` .

`clear_exception_types` ( )

Removes all exception types that are handled.

> **Note**
>
> This operation obviously cannot be undone!

`remove_exception_type` (* *exceptions*)

Removes exception types from being handled during the reconnect logic.

Parameters:

***exceptions** (Type[`BaseException`]) – An argument list of exception classes to handle.

Returns:

Whether all exceptions were successfully removed.

Return type:

`bool`

`get_task` ( )

Optional[`asyncio.Task`]: Fetches the internal task or `None` if there isn't one running.

`is_being_cancelled` ( )

Whether the task is being cancelled.

`failed` ( )

`bool` : Whether the internal task has failed.

New in version 1.2.

`is_running` ( )

`bool` : Check if the task is currently running.

New in version 1.4.

`change_interval` (* , *seconds* = *0* , *minutes* = *0* , *hours* = *0* , *time* = *...*)

Changes the interval for the sleep time.

New in version 1.2.

Parameters:

- **seconds** (`float`) – The number of seconds between every iteration.
- **minutes** (`float`) – The number of minutes between every iteration.
- **hours** (`float`) – The number of hours between every iteration.
- **time** (Union[`datetime.time` , Sequence[`datetime.time`]]) –
  The exact times to run this loop at. Either a non-empty list or a single value of `datetime.time` should be passed. This cannot be used in conjunction with the relative time parameters.
  New in version 2.0.

  > **Note**
  >
  > Duplicate times will be ignored, and only run once.

Raises:

- `ValueError` – An invalid value was given.
- `TypeError` – An invalid value for the `time` parameter was passed, or the `time` parameter was passed in conjunction with relative time parameters.

@discord.ext.tasks.**loop**(***, *seconds=...*, *minutes=...*, *hours=...*, *time=...*, *count=None*, *reconnect=True*)

A decorator that schedules a task in the background for you with optional reconnect logic. The decorator returns a `Loop` .

Parameters:

- **seconds** (`float`) – The number of seconds between every iteration.
- **minutes** (`float`) – The number of minutes between every iteration.
- **hours** (`float`) – The number of hours between every iteration.
- **time** (Union[`datetime.time` , Sequence[`datetime.time`]]) –
  The exact times to run this loop at. Either a non-empty list or a single value of `datetime.time` should be passed. Timezones are supported. If no timezone is given for the times, it is assumed to represent UTC time.
  This cannot be used in conjunction with the relative time parameters.

  > **Note**
  >
  > Duplicate times will be ignored, and only run once.

  v: stable

  New in version 2.0.

- **count** (Optional[`int`]) – The number of loops to do. `None` if it should be an infinite

- **count** (Optional[`int`]) – The number of loops to do, `None` if it should be an infinite loop.
- **reconnect** (`bool`) – Whether to handle errors and restart the task using an exponential back-off algorithm similar to the one used in `discord.Client.connect()` .

Raises:

- `ValueError` – An invalid value was given.
- `TypeError` – The function was not a coroutine, an invalid value for the `time` parameter was passed, or `time` parameter was passed in conjunction with relative time parameters.