Python

Jun 08, 2018

Contents

1 Event loop 3

2 Event system 5

3 Daemon thread 7

4 Handle WorkerExit 9

5 Exceptions 11

6 Functions 13

7 Classes 17

A library which can help you create threaded APP. It adds event queue, parent, children to each thread.

The document would mention "thread" object multiple times, but it actually refers to *Worker* instead of builtin threading.Thread.

CHAPTER 1

Event loop

This library implements event loop for each thread. Each thread has its own event queue. With the event loop, we can pause/resume/stop the thread by sending specific events to event queue. For example:

```
from worker import create_worker, wait_forever

@create_worker
def worker():
    print("thread created")
    wait_forever()
```

In the previous code:

1. A thread is created
2. The thread prints "thread created"
3. The thread enters the event loop

The event loop does following stuff:

1. Events are processed.
2. Listeners get called. *Note: you should avoid re-enter the event loop inside a listener*
3. If there is a "STOP_THREAD" event, `WorkerExit` would be raised. *Keep this in mind and carefully add "breakpoints" in your application.*

CHAPTER 2

Event system

When you stop a thread by calling *Worker.stop()*, the thread wouldn't stop immediately:

```
from worker import Worker

thread = Worker().start()
thread.stop()
print(thread.is_running()) # true
```

When `stop` is called, an "STOP_THREAD" event is put in thread's event queue, after the thread processing the event, the thread would exit the event loop by raising *WorkerExit*.

To wait until the thread exits:

```
from worker import Worker, wait_thread

thread = Worker().start()
wait_thread(thread.stop())
print(thread.is_running()) # false
```

CHAPTER 3

Daemon thread

A daemon thread is a thread which won't prevent process to exit. This is dangerous that the daemon thread would be terminated without any cleanup.

In this library, there is no "real" daemon thread. However, we *do* have a `daemon` argument when creating threads, but it works in a different way:

1. When a thread is created, it has a `parent` attribute pointing to the creator thread (the parent thread).
2. When the parent thread exits, it would broadcast a "STOP_THREAD" event to its children and wait until all child threads are stopped.
3. However, if the child thread is marked as `daemon=True`, the parent thread will not wait it. Since the daemon child thread had received the "STOP_THREAD" event, it would eventually stop. But the parent thread doesn't know when.

CHAPTER 4

Handle WorkerExit

If you want to cleanup something:

```
from worker import create_worker, wait_forever

@create_worker
def server_thread():
    server = Server() # some kinds of multiprocess server
    server.run()
    try:
        wait_forever()
    finally:
        server.terminate() # the server would be correctly terminated when
                           # the event loop raises WorkerExit

# ... do something ...

server_thread.stop()
```

It would look better if the cleanup is wrapped in a contextmanager:

```
from contextlib import contextmanager
from worker import create_worker, wait_forever

@contextmanager
def open_server():
    server = Server()
    server.run()
    try:
        yield server
    finally:
        server.terminate()

@create_worker
def server_thread():
```

(continues on next page)

(continued from previous page)

```
    with open_server() as server:
        wait_forever()

# ... do something ...

server_thread.stop()
```

CHAPTER 5

Exceptions

exception WorkerExit
Raise this error to exit the thread.

CHAPTER 6

Functions

current()
Get current thread.

Return type *Worker*

is_main(*thread=None*)
Check if the thread is the main thread.

Parameters **thread** (Worker) – Use the current thread if not set.

Return type bool

sleep(*timeout*)
Use this function to replace `time.sleep()`, to enter the event loop.

This function is a shortcut of `current().wait_timeout(timeout)`.

Parameters **timeout** (*float*) – time to wait.

Following functions have an optional callback as the first argument. They are allowed to be used as a decorator. Take *create_worker()* for example:

```
def my_task():
    ...
my_thread = create_worker(my_task, daemon=True)
# my_thread is running

# v.s.

@create_worker(daemon=True)
def my_thread():
    ...
# my_thread is running
```

create_worker(*callback*, **args*, *parent=None*, *daemon=None*, *print_traceback=True*, ***kwargs*)
Create and start a *Worker*.

callback, parent, daemon, and print_traceback are sent to *Worker*, other arguments are sent to *Worker.start()*.

Return type *Worker*

async_(*callback*, **args*, ***kwargs*)
Create and start an *Async* task.

Parameters **callback** (*callable*) – The task that would be sent to *Async*.

Return type *Async*

Other arguments are sent to Async.start().

await_(*callback*, **args*, ***kwargs*)
This is just a shortcut of async_(...).get(), which is used to put blocking function into a new thread and enter the event loop.

Following functions are just shortcuts that would be bound to the current thread when called:

listen(**args*, ***kwargs*)
A shortcut function to current().listen().

Note: Listeners created by listen shortcut would have permanent=False, so that the listener wouldn't be added multiple time when the thread is restarted.

---

unlisten(**args*, ***kwargs*)
A shortcut function to current().unlisten().

later(**args*, ***kwargs*)
A shortcut function to current().later().

update(**args*, ***kwargs*)
A shortcut function to current().update().

wait_timeout(**args*, ***kwargs*)
A shortcut function to current().wait_timeout().

wait_forever(**args*, ***kwargs*)
A shortcut function to current().wait_forever().

wait_thread(**args*, ***kwargs*)
A shortcut function to current().wait_thread().

wait_event(**args*, ***kwargs*)
A shortcut function to current().wait_event().

wait_until(**args*, ***kwargs*)
A shortcut function to current().wait_until().

With these shortcuts, we can write code without referencing to threads:

```
from worker import listen, wait_forever, create_worker

@create_worker
def printer():
    # this function runs in a new thread
    @listen("PRINT") # the listener is registered on printer thread
    def _(event):
        print(event.data)
    wait_forever() # printer's event loop
```

(continues on next page)

(continued from previous page)

```
printer.fire("PRINT", "foo")
printer.fire("PRINT", "bar")
printer.stop().join()
```

CHAPTER 7

Classes

class Worker(*task=None*, *parent=None*, *daemon=None*, *print_traceback=True*)

The main Worker class.

Parameters

- **task** (*Callable*) – The function to call when the thread starts. If this is not provided, use *Worker.wait_forever()* as the default.
- **parent** (*Worker or bool*) – The parent thread.

  If parent is None (the default), it uses the current thread as the parent, unless the current thread is the main thread.

  If parent is False. The thread is parent-less.
- **daemon** (*bool*) – Create a daemon thread. See also *is_daemon()*.
- **print_traceback** – If True, print error traceback when the thread is crashed (`task` raises an error).

fire(*event*, **args*, ***kwargs*)

Put an event to the event queue.

Parameters **event** – If `event` is not an instance of *Event*, it would be converted into an *Event* object:

```
event = Event(event, *args, **kwargs)
```

is_daemon()

Return true if the thread is a daemon thread.

If `daemon` flag is not None, return the flag value.

Otherwise, return `parent.is_daemon()`.

If there is no parent thread, return False.

is_running()

Return True if the thread is live.

join()
Join the thread.

join() is a little different with *wait_thread()*:

- *join()* uses native `threading.Thread.join()`, it doesn't enter the event loop.
- *wait_thread()* enters the event loop and waits for the `WAIT_THREAD_PENDING_DONE` event. It also has a return value: `(thread_err, thread_ret)`.

later(*callback*, **args*, *timeout=0*, ***kwargs*)
Schedule a task on this thread.

Parameters

- **callback** (*callable*) – The task that would be executed.
- **timeout** (*float*) – In seconds. Wait some time before executing the task.

Returns If `timeout` is used, this method returns a daemon *Worker*, that would first `sleep(timeout)` before executing the task. Otherwise return None.

Return type *Worker* or None

Other arguments are sent to the callback.

The scheduled task would be executed inside the event loop i.e. inside the event listener, so you should avoid blocking in the task.

If a *Worker* is returned, you can *Worker.stop()* the worker to cancel the task before the task is executed.

listen(*callback*, **args*, ***kwargs*)
Register a listener. See *Listener* for argument details.

If `callback` is not provided, this method becomes a decorator, so you can use it like:

pause()
Pause the thread.

resume()
Resume the thread.

start(**args*, ***kwargs*)
Start the thread. The arguments are passed into the `worker`.

stop()
Stop the thread.

unlisten(*callback*)
Unlisten a callback

update()
Process all events inside the event queue. This allows you to create a break point without waiting.

Use this to hook the event loop into other frameworks. For example, tkinter:

```
from tkinter import Tk
from worker import update

root = Tk()

def worker_update():
    update()
```

(continues on next page)

(continued from previous page)

```
    root.after(100, worker_update)

worker_update()
root.mainloop()
```

wait(*param*, **args*, ***kwargs*)
A shortcut method of several `wait_*` methods.

The method is chosen according to the type of the first argument.

- str - *wait_event()*.
- *Async* - Just do *Async.get()*.
- *Worker* - *wait_thread()*.
- callable - *wait_until()*.
- others - *wait_timeout()*.

All `wait_*` methods enter the event loop.

wait_event(*name*, *timeout=None*, *target=None*)
Wait for specific event.

Parameters

- **name** (*str*) – Event name.
- **timeout** (*number*) – In seconds. If provided, return None when time's up.
- **target** (Worker) – If provided, it must match `event.target`.

Returns Event data.

wait_forever()
Create an infinite event loop.

wait_thread(*thread*)
Wait thread to end. Return `(thread_error, thread_result)` tuple.

wait_timeout(*timeout*)
Wait for timeout.

Parameters **timeout** (*float*) – In seconds. The time to wait.

wait_until(*condition*, *timeout=None*)
Wait until `condition(event)` returns True.

Parameters

- **condition** (*callable*) – A callback function, which receives an *Event* object and should return `bool`.
- **timeout** (*number*) – In seconds. If provided, return None when time's up.

Returns Event data.

class Async(*task*)
Bases: *worker.Worker*

Async class. Create asynchronous (threaded) task.

Parameters **task** (*Callable*) – The worker target.

This class would initiate a parent-less, daemon thread without printing traceback.

get()
Get the result.

If the task failed, this method raises an error. If the task is not completed, enter the event loop.

class Defer
Defer object. Handy in cross-thread communication. For example, update tkinter GUI in the main thread:

```
from tkinter import *
from worker import current, update, create_worker, Defer, is_main

main_thread = current()
root = Tk()

def hook():
    root.after(100, hook)
    update()

@create_worker
def worker():
    i = 0
    def update_some_gui(on_finished=None):
        print("gui", is_main())
        def remove_button():
            button.destroy()
            on_finished("OK")
        button = Button(
            root,
            text="Click me to fulfill defer {}".format(i),
            command=remove_button
        )
        button.pack()
    while True:
        defer = Defer()
        print("worker", is_main())
        main_thread.later(update_some_gui, 0, on_finished=defer.resolve)
        defer.get()
        i += 1

hook()
root.mainloop()
worker.stop()
```

get()
Enter the event loop and wait util the defer is fulfilled.

If the defer is resolved, return the result. If the defer is rejected, raise the result.

reject(*err*)
Reject with `err`

resolve(*value*)
Resolve with `value`

class Channel
Channel class. Broadcast events to multiple threads.

pub(**args*, ***kwargs*)
Publish an event to the channel. See *`Event`* for the arguments.

Events published to the channel are broadcasted to all subscriber threads.

sub(*thread=None*)
Subscribe `thread` to the channel.

Parameters **thread** (`Worker`) – The subscriber thread. Use current thread if not provided.

unsub(*thread=None*)
Unsubscribe to channel.

Parameters **thread** (`Worker`) – The subscriber thread. Use current thread if not provided.

class Event(*name*, *data=None*, *, *bubble=False*, *broadcast=False*, *target=None*)
Event data class.

Parameters

- **name** (*str*) – Event name.
- **data** – Event data.
- **bubble** (*bool*) – If true then the event would be bubbled up through parent.
- **broadcast** (*bool*) – If true then the event would be broadcasted to all child threads.
- **target** (`Worker`) – Event target. If none then set to the thread calling *Worker.fire*.

class Listener(*callback*, *event_name*, *, *target=None*, *priority=0*, *once=False*, *permanent=True*)
Listener data class.

Parameters

- **callback** (*callable*) – The listener callback.
- **event_name** (*str*) – The event name.
- **target** (`Worker`) – Only match specific `event.target`.
- **priority** (*int*) – The listener are ordered in priority. The higher is called first.
- **once** (*bool*) – If True then remove the listener once the listener is called.
- **permanent** (*bool*) – If False then remove the listener once the thread is stopped. Listeners created by *listen()* shortcut are non-permanent listeners.

Index

A

Async (class in worker), 19
async_() (in module worker), 14
await_() (in module worker), 14

C

Channel (class in worker), 20
create_worker() (in module worker), 13
current() (in module worker), 13

D

Defer (class in worker), 20

E

Event (class in worker), 21

F

fire() (Worker method), 17

G

get() (Async method), 19
get() (Defer method), 20

I

is_daemon() (Worker method), 17
is_main() (in module worker), 13
is_running() (Worker method), 17

J

join() (Worker method), 17

L

later() (in module worker), 14
later() (Worker method), 18
listen() (in module worker), 14
listen() (Worker method), 18
Listener (class in worker), 21

P

pause() (Worker method), 18
pub() (Channel method), 20

R

reject() (Defer method), 20
resolve() (Defer method), 20
resume() (Worker method), 18

S

sleep() (in module worker), 13
start() (Worker method), 18
stop() (Worker method), 18
sub() (Channel method), 20

U

unlisten() (in module worker), 14
unlisten() (Worker method), 18
unsub() (Channel method), 21
update() (in module worker), 14
update() (Worker method), 18

W

wait() (Worker method), 19
wait_event() (in module worker), 14
wait_event() (Worker method), 19
wait_forever() (in module worker), 14
wait_forever() (Worker method), 19
wait_thread() (in module worker), 14
wait_thread() (Worker method), 19
wait_timeout() (in module worker), 14
wait_timeout() (Worker method), 19
wait_until() (in module worker), 14
wait_until() (Worker method), 19
Worker (class in worker), 17
WorkerExit, 11