consign

发行版本 *1.0.2*

SpectrePrediction

2022 年 10 月 09 日

Contents

CHAPTER 1

consign

1.1 consign package

1.1.1 consign Package

Functions

asleep(secs)	协程的 sleep 函数，休眠一定时间
chain_reaction(func)	chain_reaction 使得嵌套协程函数可以以链式的方式运行.
coroutine([debug, create_callback, ...])	consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行
repeated_call(debug)	repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用
wait(task, *[, time_out])	wait 阻塞等待一个 Task 任务完成，并在期间参与工作

asleep

`consign.`**`asleep`**(*secs: float*)

协程的 sleep 函数，休眠一定时间

`asleep` 与 `sleep` 不同，`asleep` 是非阻塞的、协程的

`asleep` 通过 **不断的切换控制权**和 **判断时间**来实现休眠

> **警告：** `asleep` 无法保证执行顺序 (即使是在单/主线程下)
>
> 因为控制权无法保证谁先获取
>
> 举例 1 和 2 函数抛出了委托 sleep，同时 sleep 5 秒
>
> 在 5 秒后，无法保证是 1 先于 2 获取到执行的控制权
>
> `asleep` 休眠的时长是 **不精确的**，受到当前任务量和阻塞时间的影响
>
> 因为 `asleep` 在转移控制权的过程中，队列里依旧存在其余任务
>
> 你无法保证下一个任务是不是阻塞的
>
> 如果存在任务是阻塞的，那么他就有可能对 `asleep` 照成影响
>
> ---
>
> **小技巧：** 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率
>
> ---
>
> 例子
>
> 举例：1 函数在第一次抛出 `asleep` 5 秒，但 2 函数抛出一个需要 **阻塞执行 7 秒**的 **非协程**任务
>
> 假设在单/主线程下执行
>
> 那么 2 函数的任务阻塞时间大于 >5 秒，且没有控制权的转移 (非协程)
>
> 因而，1 函数无法在 sleep 5 秒后得到立即执行，

`asleep` 为了方便使用，已经被 `chain_reaction` 和 `repeated_call` 修饰

二者分别使得 `asleep` 以顺序方式执行、`asleep` 以有参的形式可以再被无参调用执行

参数

`secs` (*`float`*) –休眠的秒数

chain_reaction

consign.**chain_reaction**(*func*)

chain_reaction 使得嵌套协程函数可以以链式的方式运行.

> **警告:** chain_reaction 接收的参数 func 必须是被 `coroutine` 修饰的委托函数.

被 `chain_reaction` 修饰的函数会被添加上对应的标记

他会重新构造一个精心设计的回调函数来完成嵌套协程到外层协程控制权的切换

`Worker` 会通过标记确认一个协程是否是被 `chain_reaction` 修饰，随后通过标记传递必须的信息以供回调时使用

参见:

之所以会需要标记传递信息，原因在于现在的 `chain_reaction` 的主动权在函数手上

而旧版在 Worker 手上，相比之下新版更加自由和随心所欲，但旧版方便，却不好控制

`chain_reaction` 是否修饰的区别在于:

正常的被修饰的协程函数在被 yield 调用是非阻塞的，直接返回 Task 类

被修饰的函数在被 yield 调用时会等待函数执行完毕，并直接返回值而非 Task 类

简单的说，他能让协程函数中的协程函数以普通顺序的方式运行，但同时能够享受控制权切换的效果

例子

使用 chain_reaction

```
from consign import coroutine, chain_reaction

@chain_reaction
@coroutine
def get_value():
    yield ...
    return 10

@coroutine
def my_test():
    value = yield get_value
    print(value)
    # value is 10
```

(续下页)

(接上页)

```
my_test()
CoroutineWorker().loop_work(forever=True)
```

不使用 chain_reaction

```
from consign import coroutine

@coroutine
def get_value():
    yield ...
    return 10

@coroutine
def my_test():
    value = yield get_value
    print(value)
    # value is Task类
    # 并且此时get_value并没完成

my_test()
CoroutineWorker().loop_work(forever=True)
```

chain_reaction 是比轮询更优雅的，但并非在所有情况下

通常而言，如果对时间有更高可控性的可以使用 chain_reaction

而更高性能还是建议使用 wait

参数

func (*function*) –必须是被 coroutine 修饰的委托函数

抛出

AssertionError –当传入参数 func 不是被 coroutine 修饰的委托函数时抛出

coroutine

consign.**coroutine**(*debug: bool = False*, *, *create_callback=None*, *complete_callback=None*)

consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行

他的作用很简单，就是包裹函数，无论是普通函数还是生成器函数，使得 consign 得以运行他们

参见:

普通函数的话，coroutine 会生成一个包装好的生成器函数替代

随后如同对待生成器函数一般使用

`coroutine` 修饰时在程序创建之初会创建 `ConsignOrder` 并记录在 order 属性

在每次调用时创建 `Task` ，同时每次调用的返回值被 `Task` 替代，你可以从其中获取更多信息

被 `coroutine` 修饰的函数变成协程后，是非阻塞的，状态、返回值等都可以在 `Task` 中获取

`coroutine` 支持传入一些参数，其中比较特别是 debug 参数

debug 参数允许你在不修改代码的情况下，以代码原本的阻塞逻辑运行

`coroutine` 为修饰器而设计，但你也可以把它当作函数使用，但这也会使得 `ConsignOrder` 被不停重复创建而效率低下

yield 是 **控制权转移**的关键，consign 在单线程下依旧以 **并发**的形式运行，关键就在于 yield。

被 `coroutine` 修饰的函数最好是生成器函数，且 **尽可能的切分函数控制权**提高控制权转换的频率

一般

```
def do_something_x():
    io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = do_something_1()
    b = do_something_2()
    return a, b
```

还行

```
def do_something_x():
    io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = yield do_something_1()
    b = yield do_something_2()
    return a, b
```

nice!

```
@coroutine
def do_something_x():
    yield io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = yield do_something_1()
    b = yield do_something_2()
    wait(a) or chain_reaction(do_something_x)
    return a, b
```

参数

- **debug** (*bool*) –如果为 True，coroutine 允许你在不修改代码的情况下，以代码原本的阻塞逻辑运行

  如果是非生成器函数直接运行并返回结果

  如果是生成器函数会以一个模拟普通函数的函数 代替运行 (阻塞并不断的 next) 并返回结果

- **create_callback** (*function*) –创建协程函数时执行的回调函数，create_callback 在协程函数执行前执行

  create_callback 需要一个参数用于接收此次运行时的 Task

- **complete_callback** (*function*) –完成协程函数时执行的回调函数，complete_callback 在协程函数获取到返回值后执行，但此时协程函数状态并非完成

  complete_callback 需要一个参数用于接收此次运行时的 Task

返回

repeated_call

consign.**repeated_call**(*debug*)

repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用

```
def test(something: any):
    return something

test("hi")
```

当其被修饰时，相当于

```
@repeated_call
def test(something: any):
    return something

# 此时返回的是一个替代函数，test并没有被真正执行
test("hi")
# 完整的执行过程，等价于没修饰前的test("hi")
test("hi")()
```

repeated_call 并不是必须的，与之相似功能的还有 functools 中的 partial 以及 class 中的 __call__ 方法

> **警告：** 一般而言，他的使用是为了 consign 中的一项规则：yield 一个 function 时，function 应当满足无参的条件
>
> 但值得注意的是，这个条件可能在 **未来被移除**，有违简单易用这个初衷，他是我为了自己旧版项目做的兼容

repeated_call 并不是必须的！因为 consign 只在乎你 yield 的是否是一个无参的可调用对象如果你本身是无参的，可以直接传递函数名

通常情况下，你可以直接在 yield 前调用 function，**这其实意味着你 yield 的是调用函数后的返回值**。

参数

debug (*bool*) –如果为 True，那么返回原函数

抛出

AssertionError –当传入参数 func 不是 callable 时抛出

wait

consign.**wait**(*task*, *, *time_out=0.1*)

wait 阻塞等待一个 Task 任务完成，并在期间参与工作

wait 是 Supervisor 类的上层封装

简单的讲，他的作用就是轮询等待一个 Task

当这一个 Task 的状态转变为结束后 返回此 Task 的 value (也就是对应协程函数的返回值)

期间他会将自身代入成为一个 Worker 进行 loop_work

某种程度上看，wait 相当于特别一点的 loop_work ，主要在于他的 WorkArea 继承机制

由于继承机制，往往你需要明确你的 wait 工作在哪个区域下？为哪个区域而工作？

wait 一个 Task 时，他会继承 Task 中的 WorkArea 并生成对应的 CoroutineWorker 进行 loop_work

wait 不对 约定的结束信号 (是自定义的一种情况，使用者基本无需关心) 做处理碰到结束信号时他会重新提交回原 WorkArea

wait 轮询的时间是 **不精确的**，受到当前任务量和阻塞时间的影响

因为 wait 的轮询任务在转移控制权的过程中，队列里依旧存在其余任务

你无法保证下一个任务是不是阻塞的

如果存在任务是阻塞的，那么他就有可能对 wait 照成影响

小技巧: 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率

---

wait 的原理是：wait 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 wait 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

参数

- **task** (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回
- **time_out** –最短轮询的间隔，他一般用在多线程中，当对应 WorkArea 的 queue 为空时，time_out 才会触发

> **警告:** 尽可能不要设置 time_out 为 None
>
> 这或许会导致 wait 被阻塞没法正常退出

返回

目标 Task 中的 value

Classes

AsyncWorker([work_area_name, work_area])	AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker
ContextVar	
CoroutineWorker([work_area_name, work_area])	CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石
Supervisor(task)	wait 的底层实现
WorkArea([name])	WorkArea 是 consign 的基石，负责规划 Worker 和协程

AsyncWorker

class consign.**AsyncWorker**(*work_area_name: str = 'DEFAULT_WORK_AREA'*, *, *work_area=None*)

基类：*CoroutineWorker*

AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker

AsyncWorker 可以将 耗时的事情 (任意阻塞函数) 分配给许多线程去处理，得以实现异步

参见:

当然，如果你使用 chain_reaction ，那么他依旧还是顺序执行的

所以如果性能有更高要求，建议灵活使用 wait

AsyncWorker 的使用方法通常是 init_thread 创建线程，很少且不建议使用 loop_work

如果你使用 AsyncWorker ，那么 loop_work(forever=False) **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性 (官方的描述: 由于多线程或者多进程的上下文，这个数字是不可靠的。请注意，这可能会在 Unix 平台上引起 NotImplementedError) 以及框架的获取提交导致的

小技巧: 通常在多线程下使用 loop_work(forever=True) 或者重复 while True 使用 loop_work(forever=False)

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

所以如果 AsyncWorker 不使用 init_thread 直接使用 loop_work 其实和 CoroutineWorker 效果是 **相同的**

都是使用 **当前进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

AsyncWorker 创建的线程默认是 **守护线程**，如果你的程序没有阻塞一路到 exit 的话，可能导致协程任务异常被终止

所以在 **非 cmd** 下，最好使用 while True 或者 loop_work(forever=True) 去阻塞，或者设置参数选择生成非守护线程。

例子

```
some_coroutine()

aw = AsyncWorker()
using_thread_num = 5
# name_iter 传入为None的时候，默认使用uuid去生成名字
aw.init_thread(using_thread_num, name_iter=(f"DEFAULT_WORK_AREA_{i}" for i in␣
↪range(using_thread_num)))

# 如果程序主进程没有其他内容，那么需要你使用阻塞，可以使用while或者Worker的loop_
↪work
some_loop()
```

参数

- **work_area_name** (*str*) –WorkArea 的名字，AsyncWorker 会通过名字获取对应的 WorkArea
- **work_area** (WorkArea) –work_area 是显式参数，需要显式调用

  可以直接指定 WorkArea ，如果直接指定，会跳过 work_area_name 的查找过程

抛出

AssertionError –当传入参数 work_area_name 无法被找到时抛出

Attributes Summary

thread_list	在获取 thread_list 时，同时检测并删除 __thread_list 中已经关闭的线程

Methods Summary

clear_dead_thread()	一般不建议外部调用
create_thread([name, daemon])	创建一个持续运行的线程，用于对协程函数进行执行
init_thread(create_num[, name_iter, daemon])	可以一次初始化多个线程
submit_thread_stop_flag()	向对应 WorkArea 提交约定的结束信号

Attributes Documentation

thread_list

在获取 thread_list 时，同时检测并删除 __thread_list 中已经关闭的线程

> **警告：** 注意这个处理会同步修改到 __thread_list
>
> 返回的 List 并不是副本而是某种’引用’

返回

存放线程的列表

Methods Documentation

clear_dead_thread()

一般不建议外部调用

遍历列表并判断线程是否存活

如果线程不存活，那么从列表中删除此线程

返回

None

create_thread(*name=None*, *daemon=True*)

创建一个持续运行的线程，用于对协程函数进行执行

参数

- **name** (*str*) –创建的线程名，如果为 None，则使用 uuid 创建
- **daemon** (*bool*) –是否守护线程

返回

创建的线程对象

init_thread(*create_num*, *name_iter=None*, *daemon=True*)

可以一次初始化多个线程

是对 `create_thread` 的上层封装

参数

- **create_num** (*int*) –创建对应线程的数量
- **name_iter** –生成器或者容器，如果没有默认使用传递 None 即使用 uuid 命名

> **警告：** 没有对 `name_iter` 长度做校验但 `name_iter` 长度应该跟 create_num 相同，避免出乎意料的结果

- **daemon** (*bool*) –创建的是否是守护线程

返回

submit_thread_stop_flag()

向对应 `WorkArea` 提交约定的结束信号

> **警告：** 约定的结束信号不一定能立马结束，因为队列中可能还有其他内容

返回

CoroutineWorker

class consign.**CoroutineWorker**(*work_area_name: str = 'DEFAULT_WORK_AREA'*, *, *work_area=None*)

基类：`object`

CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石

`CoroutineWorker` 可以将 控制权切换以实现并发的效果 (通过 yield 实现控制权切换)

`CoroutineWorker` 的使用方法通常是 `loop_work` 阻塞并完成当前队列中的全部内容

如果你使用 `CoroutineWorker` 在多线程/进程环境下，那么 `loop_work(forever=False)` **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性以及框架的获取提交导致的

小技巧： 通常在多线程下使用 `loop_work(forever=True)` 或者重复 `while True` 使用 `loop_work(forever=False)`

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

使用的是 **当前进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

例子

```
some_coroutine()
```

(续下页)

(接上页)

```
cw = CoroutineWorker()
cw.loop_work()
```

参数

- **work_area_name** (*str*) –WorkArea 的名字，CoroutineWorker 会通过名字获取对应的 WorkArea
- **work_area** (WorkArea) –work_area 是显式参数，需要显式调用

  可以直接指定 WorkArea ，如果直接指定，会跳过 work_area_name 的查找过程

抛出

AssertionError –当传入参数 work_area_name 无法被找到时抛出

Methods Summary

loop_work(*[, time_out, forever])	loop_work 阻塞并完成当前队列中的全部内容或者收到停止信号
qsize()	获得 WorkArea 队列中的大概数量
submit_work(*args)	提交任务到 WorkArea 队列
work_once([time_out])	CoroutineWorker 获取一次 WorkArea 队列中的内容并执行一次

Methods Documentation

loop_work (*, *time_out=0.1*, *forever=True*)

loop_work 阻塞并完成当前队列中的全部内容或者收到停止信号

> **警告：** 在多线程/进程环境下
>
> forever 参数为 False 时需要考虑 队列 QSIZE 的不确定性 (由于多线程或者多进程的上下文，这个数字是不可靠的。)
>
> **小技巧：** 通常在多线程下使用 loop_work(forever=True) 或者重复 while True 使用 loop_work(forever=False)

参数

- **time_out** –等待 WorkArea 队列获取内容的等待超时时长

- **forever** –是否永久阻塞

  如果 `forever` 为 False，那么 `loop_work` 会考虑队列的 `qsize` 是否 <=0

  如果队列的 `qsize` <=0，那么退出循环

  如果你需要指定某个任务完成并阻塞，那么建议使用 `wait` 来确保完成优于直接使用 `forever` =True

返回

None

抛出

ValueError –当参数 `time_out` 不正确时抛出

qsize()

获得 `WorkArea` 队列中的大概数量

> **警告:** 由于多线程或者多进程的上下文，这个数字是不可靠的。
>
> 请注意，这可能会在 Unix 平台上引起 NotImplementedError ，如 macOS ，因为其上没有实现 sem_getvalue() 。

返回

获得 `WorkArea` 队列中的大概数量

抛出

NotImplementedError –能会在 Unix 平台上引起 NotImplementedError ，如 macOS ，因为其上没有实现 sem_getvalue()

submit_work(**args*)

提交任务到 `WorkArea` 队列

不建议外部调用，如果调用需要明白自己正在做什么

参数

args –按照规定 `WorkArea` 队列中是只应当 put 一个定义好的 tuple

其格式为 (yield_value, generator, receipts)

这里因为不向外调用，所以使用可变参偷了个懒，就无需再手动创建 tuple

返回

None

抛出

Full –当 `WorkArea` 队列存放满时触发

work_once(*time_out=None*)

CoroutineWorker 获取一次 WorkArea 队列中的内容并执行一次

任意一种情况返回都会视作一次 Work

参数

time_out –等待 WorkArea 队列获取内容的等待超时时长

返回

返回 bool 类型

False 意味着收到了约定的结束信号

True 表示正常结束 Work

抛出

- **Empty** –当 WorkArea 队列为空时阻塞时长超过参数 time_out 时抛出
- **ValueError** –当参数 time_out 不正确时抛出

Supervisor

class consign.**Supervisor**(*task*)

基类：object

wait 的底层实现

wait 是 Supervisor 类的上层封装，wait 向外暴露，但 Supervisor 不被直接暴露

Supervisor 能够阻塞等待一个 Task 任务完成，并在期间参与工作

Supervisor 的原理是：Supervisor 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 Supervisor 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

更多详情请查看 wait 函数

参数

task (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回

Methods Summary

polling_func()	轮询任务，用于判断 Task 的状态
run_until_complete([time_out])	阻塞、工作直到轮询任务完成

Methods Documentation

polling_func()

轮询任务，用于判断 Task 的状态

他是一个 coroutine 修饰的协程函数

Supervisor 会自动继承 Task 中的 WorkArea 并在此提交

每次执行控制权切换前判断 Task 的状态是否是已完成，完成会告知 Supervisor

返回

None

run_until_complete(*time_out=0.1*)

阻塞、工作直到轮询任务完成

参数

time_out –其实就是 CoroutineWorker 中 work_once 的参数

等待 WorkArea 队列获取内容的等待超时时长

也是最短轮询的间隔，他一般用在多线程中，当对应 WorkArea 的 queue 为空时，time_out 才会触发

返回

Task 类中的 value，也就是对应协程的返回值

WorkArea

class consign.**WorkArea**(*name: str = 'DEFAULT_WORK_AREA', *args, **kwargs*)

基类：object

WorkArea 是 consign 的基石，负责规划 Worker 和协程

WorkArea 分离所有 Worker 的工作区域，这可以有助于你分配资源以及更好的使用 Worker

WorkArea 被设计成线程安全的，借助 ContextVar 在每个线程中都拥有独立的值，其本质是 threading.local

WorkArea 默认在导入此库时，创建默认的 DEFAULT_WORK_AREA 变量作为默认工作地址，并置于 builtins 中：

1. 这意味着你可以通过直接访问 DEFAULT_WORK_AREA.get() 获取当前的工作区域

当然并不推荐，一旦你这样使用，你应当明白你想要做些什么，以及会造成那些影响

2. DEFAULT_WORK_AREA 是全局的统一的，依托 ContextVar 可以实现在不同线程中不同的值

不同线程间修改不会出现相互影响

WorkArea 作为上下文管理器其实现的本质就是对 DEFAULT_WORK_AREA 的修改和恢复

WorkArea 会修改 DEFAULT_WORK_AREA 的值来影响“Worker“的运行

WorkArea 本质是一个上下文管理器，但同时支持修饰器写法

修饰器代码来源于 ContextDecorator, 写法相当于语法糖, 他们是相等的:

使用修饰器

```
@cm()
def f():
    # Do stuff
```

使用上下文管理器

```
def f():
    with cm():
        # Do stuff
```

WorkArea 的 name 参数非常重要:

你可以注意到 WorkArea 被 same_name_singleton 修饰，same_name_singleton 是一个单例的修饰器

(新版本，为了适配多进程，修饰器无法被序列化，所以改用元类进行单例，效果不变)

WorkArea 通过 name 区分，同名 (str 相同，但大小写区分) 的情况下只有一个 WorkArea 实例

但值得注意的是:

即使是同一个单例 WorkArea ，也并不意味着 WorkArea 成员变量 old_work_area_queue 存放的值是相同的

old_work_area_queue 的本质是被 ContextVar 包裹的变量

为了防止多线程中共用一个 WorkArea 导致的 LifoQueue 顺序冲突

old_work_area_queue 会在每个单独的线程中创建一个 LifoQueue ，由线程共享，以此保证线程安全

参数

name (*str*) –WorkArea 的名字，同名 WorkArea 是同一个实例，默认是名字使用 DEFAULT_WORK_AREA

小技巧： 理论上能 hash 的值都能传入 name

为了减少意料之外的事情发生，建议输入是 str

Methods Summary

__call__(func)	适配修饰器写法的实现
as_default()	返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

Methods Documentation

__call__ (*func*)

适配修饰器写法的实现

参数

func (*function*) –被包裹的函数中的全部协程函数将会运行于此 WorkArea 的作用范围

static as_default ()

返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

as_default 将会去 builtins 中寻找 DEFAULT_WORK_AREA 变量并返回其中保存的 WorkArea

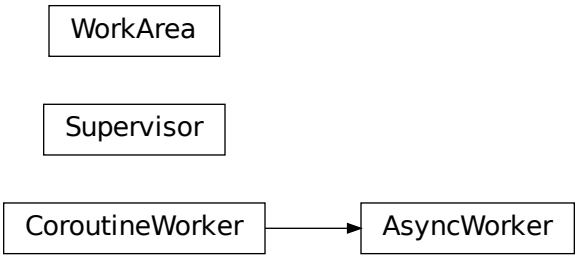
DEFAULT_WORK_AREA 变量一般会在导入库时被创建

如果没能找到，as_default 会创建 DEFAULT_WORK_AREA 包装并置入 builtins 并返回

返回

返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

Class Inheritance Diagram



1.1.2 Subpackages

consign.decorator package

consign.decorator Package

Functions

chain_reaction(func)	chain_reaction 使得嵌套协程函数可以以链式的方式运行.
coroutine([debug, create_callback, ...])	consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行
repeated_call(debug)	repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用

chain_reaction

`consign.decorator.`**`chain_reaction`**(*func*)

chain_reaction 使得嵌套协程函数可以以链式的方式运行.

> **警告：** chain_reaction 接收的参数 func 必须是被 `coroutine` 修饰的委托函数.

被 `chain_reaction` 修饰的函数会被添加上对应的标记

他会重新构造一个精心设计的回调函数来完成嵌套协程到外层协程控制权的切换

`Worker` 会通过标记确认一个协程是否是被 `chain_reaction` 修饰，随后通过标记传递必须的信息以供回调时使用

参见:

之所以会需要标记传递信息，原因在于现在的 `chain_reaction` 的主动权在函数手上

而旧版在 Worker 手上，相比之下新版更加自由和随心所欲，但旧版方便，却不好控制

`chain_reaction` 是否修饰的区别在于：

正常的被修饰的协程函数在被 yield 调用是非阻塞的，直接返回 Task 类

被修饰的函数在被 yield 调用时会等待函数执行完毕，并直接返回值而非 Task 类

简单的说，他能让协程函数中的协程函数以普通顺序的方式运行，但同时能够享受控制权切换的效果

例子

使用 chain_reaction

```
from consign import coroutine, chain_reaction

@chain_reaction
@coroutine
def get_value():
    yield ...
    return 10

@coroutine
def my_test():
    value = yield get_value
    print(value)
    # value is 10

my_test()
CoroutineWorker().loop_work(forever=True)
```

不使用 chain_reaction

```
from consign import coroutine

@coroutine
def get_value():
    yield ...
    return 10

@coroutine
def my_test():
    value = yield get_value
    print(value)
    # value is Task类
    # 并且此时get_value并没完成

my_test()
CoroutineWorker().loop_work(forever=True)
```

chain_reaction 是比轮询更优雅的，但并非在所有情况下

通常而言，如果对时间有更高可控性的可以使用 chain_reaction

而更高性能还是建议使用 wait

参数

func (*function*) –必须是被 coroutine 修饰的委托函数

抛出

AssertionError –当传入参数 func 不是被 coroutine 修饰的委托函数时抛出

coroutine

consign.decorator.**coroutine**(*debug: bool = False*, *, *create_callback=None*, *complete_callback=None*)

consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行

他的作用很简单，就是包裹函数，无论是普通函数还是生成器函数，使得 consign 得以运行他们

参见:

普通函数的话，coroutine 会生成一个包装好的生成器函数替代

随后如同对待生成器函数一般使用

coroutine 修饰时在程序创建之初会创建 ConsignOrder 并记录在 order 属性

在每次调用时创建 Task ，同时每次调用的返回值被 Task 替代，你可以从其中获取更多信息

被 coroutine 修饰的函数变成协程后，是非阻塞的，状态、返回值等都可以在 Task 中获取

`coroutine` 支持传入一些参数，其中比较特别是 debug 参数

debug 参数允许你在不修改代码的情况下，以代码原本的阻塞逻辑运行

`coroutine` 为修饰器而设计，但你也可以把它当作函数使用，但这也会使得 `ConsignOrder` 被不停重复创建而效率低下

yield 是 **控制权转移**的关键，consign 在单线程下依旧以 **并发**的形式运行，关键就在于 yield。

被 `coroutine` 修饰的函数最好是生成器函数，且 **尽可能的切分函数控制权**提高控制权转换的频率

一般

```
def do_something_x():
    io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = do_something_1()
    b = do_something_2()
    return a, b
```

还行

```
def do_something_x():
    io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = yield do_something_1()
    b = yield do_something_2()
    return a, b
```

nice!

```
@coroutine
def do_something_x():
    yield io.sleep(long long time)
    return something

@coroutine
```

(续下页)

(接上页)

```
def my_function():
    a = yield do_something_1()
    b = yield do_something_2()
    wait(a) or chain_reaction(do_something_x)
    return a, b
```

参数

- **debug** (*bool*) –如果为 True，coroutine 允许你在不修改代码的情况下，以代码原本的阻塞逻辑运行

  如果是非生成器函数直接运行并返回结果

  如果是生成器函数会以一个模拟普通函数的函数 代替运行并返回结果

- **create_callback** (*function*) –创建协程函数时执行的回调函数，create_callback 在协程函数执行前执行

  create_callback 需要一个参数用于接收此次运行时的 Task

- **complete_callback** (*function*) –完成协程函数时执行的回调函数，complete_callback 在协程函数获取到返回值后执行，但此时协程函数状态并非完成

  complete_callback 需要一个参数用于接收此次运行时的 Task

返回

repeated_call

consign.decorator.**repeated_call**(*debug*)

repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用

```
def test(something: any):
    return something

test("hi")
```

当其被修饰时，相当于

```
@repeated_call
def test(something: any):
    return something

# 此时返回的是一个替代函数，test并没有被真正执行
```

(续下页)

(接上页)

```
test("hi")
# 完整的执行过程，等价于没修饰前的test("hi")
test("hi")()
```

repeated_call 并不是必须的，与之相似功能的还有 functools 中的 partial 以及 class 中的 __call__ 方法

> **警告：** 一般而言，他的使用是为了 consign 中的一项规则：yield 一个 function 时，function 应当满足无参的条件
>
> 但值得注意的是，这个条件可能在 **未来被移除**，有违简单易用这个初衷，他是我为了自己旧版项目做的兼容

repeated_call 并不是必须的！因为 consign 只在乎你 yield 的是否是一个无参的可调用对象如果你本身是无参的，可以直接传递函数名

通常情况下，你可以直接在 yield 前调用 function，**这其实意味着你 yield 的是调用函数后的返回值**。

参数

debug (*bool*) –如果为 True，那么返回原函数

抛出

AssertionError –当传入参数 func 不是 callable 时抛出

Submodules

consign.decorator.chainreactiondecorator Module

Functions

chain_reaction(func)	chain_reaction 使得嵌套协程函数可以以链式的方式运行.
wraps(wrapped[, assigned, updated])	Decorator factory to apply update_wrapper() to a wrapper function

chain_reaction

consign.decorator.chainreactiondecorator.**chain_reaction**(*func*)

chain_reaction 使得嵌套协程函数可以以链式的方式运行.

> **警告:** chain_reaction 接收的参数 func 必须是被 `coroutine` 修饰的委托函数.

被 `chain_reaction` 修饰的函数会被添加上对应的标记

他会重新构造一个精心设计的回调函数来完成嵌套协程到外层协程控制权的切换

`Worker` 会通过标记确认一个协程是否是被 `chain_reaction` 修饰，随后通过标记传递必须的信息以供回调时使用

参见:

之所以会需要标记传递信息，原因在于现在的 `chain_reaction` 的主动权在函数手上

而旧版在 Worker 手上，相比之下新版更加自由和随心所欲，但旧版方便，却不好控制

`chain_reaction` 是否修饰的区别在于:

正常的被修饰的协程函数在被 yield 调用是非阻塞的，直接返回 Task 类

被修饰的函数在被 yield 调用时会等待函数执行完毕，并直接返回值而非 Task 类

简单的说，他能让协程函数中的协程函数以普通顺序的方式运行，但同时能够享受控制权切换的效果

例子

使用 chain_reaction

```
from consign import coroutine, chain_reaction

@chain_reaction
@coroutine
def get_value():
    yield ...
    return 10

@coroutine
def my_test():
    value = yield get_value
    print(value)
    # value is 10
```

(续下页)

(接上页)

```
my_test()
CoroutineWorker().loop_work(forever=True)
```

不使用 chain_reaction

```
from consign import coroutine

@coroutine
def get_value():
    yield ...
    return 10

@coroutine
def my_test():
    value = yield get_value
    print(value)
    # value is Task类
    # 并且此时get_value并没完成

my_test()
CoroutineWorker().loop_work(forever=True)
```

chain_reaction 是比轮询更优雅的，但并非在所有情况下

通常而言，如果对时间有更高可控性的可以使用 chain_reaction

而更高性能还是建议使用 wait

参数

func (*function*) –必须是被 coroutine 修饰的委托函数

抛出

AssertionError –当传入参数 func 不是被 coroutine 修饰的委托函数时抛出

consign.decorator.consigndecorator Module

Functions

`coroutine`([debug, create_callback, ...])	consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行
`isgeneratorfunction`(object)	Return true if the object is a user-defined generator function.
`wraps`(wrapped[, assigned, updated])	Decorator factory to apply update_wrapper() to a wrapper function

coroutine

`consign.decorator.consigndecorator.`**`coroutine`**(*debug: bool = False*, *, *create_callback=None*, *complete_callback=None*)

consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行

他的作用很简单，就是包裹函数，无论是普通函数还是生成器函数，使得 consign 得以运行他们

参见:

普通函数的话，`coroutine` 会生成一个包装好的生成器函数替代

随后如同对待生成器函数一般使用

`coroutine` 修饰时在程序创建之初会创建 `ConsignOrder` 并记录在 order 属性

在每次调用时创建 `Task` ，同时每次调用的返回值被 `Task` 替代，你可以从其中获取更多信息

被 `coroutine` 修饰的函数变成协程后，是非阻塞的，状态、返回值等都可以在 `Task` 中获取

`coroutine` 支持传入一些参数，其中比较特别是 debug 参数

debug 参数允许你在不修改代码的情况下，以代码原本的阻塞逻辑运行

`coroutine` 为修饰器而设计，但你也可以把它当作函数使用，但这也会使得 `ConsignOrder` 被不停重复创建而效率低下

yield 是 **控制权转移**的关键，consign 在单线程下依旧以 **并发**的形式运行，关键就在于 yield。

被 `coroutine` 修饰的函数最好是生成器函数，且 **尽可能的切分函数控制权**提高控制权转换的频率

一般

```
def do_something_x():
    io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = do_something_1()
    b = do_something_2()
    return a, b
```

还行

```
def do_something_x():
    io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = yield do_something_1()
    b = yield do_something_2()
    return a, b
```

nice!

```
@coroutine
def do_something_x():
    yield io.sleep(long long time)
    return something

@coroutine
def my_function():
    a = yield do_something_1()
    b = yield do_something_2()
    wait(a) or chain_reaction(do_something_x)
    return a, b
```

参数

- **debug** (*bool*) –如果为 True，`coroutine` 允许你在不修改代码的情况下，以代码原本的阻塞逻辑运行

如果是非生成器函数直接运行并返回结果

如果是生成器函数会以一个模拟普通函数的函数 代替运行并返回结果

- **create_callback** (*function*) –创建协程函数时执行的回调函数，`create_callback` 在协程函数执行前执行

  `create_callback` 需要一个参数用于接收此次运行时的 `Task`

- **complete_callback** (*function*) –完成协程函数时执行的回调函数，`complete_callback` 在协程函数获取到返回值后执行，但此时协程函数状态并非完成

  `complete_callback` 需要一个参数用于接收此次运行时的 `Task`

返回

consign.decorator.consignorder Module

Classes

ConsignOrder(consignor_func[, ...])	

ConsignOrder

class consign.decorator.consignorder.**ConsignOrder**(*consignor_func*, *create_callback=None*, *complete_callback=None*)

基类：`object`

Methods Summary

all_info()	

Methods Documentation

all_info()

Class Inheritance Diagram

consign.decorator.consigntask Module

Classes

Enum(value)	Generic enumeration.
Task(order)	
TaskResult(value)	An enumeration.
TaskState(value)	An enumeration.

Task

class consign.decorator.consigntask.**Task**(*order*)

基类：object

Methods Summary

all_info()	
complete()	
create()	
create_receipts()	不建议自行调用，它会在被修饰函数启动时创建:return:
submit(something)	

Methods Documentation

all_info()

complete()

create()

create_receipts()

不建议自行调用，它会在被修饰函数启动时创建:return:

submit(*something*)

TaskResult

class consign.decorator.consigntask.**TaskResult**(*value*)

基类：Enum

An enumeration.

Attributes Summary

NoGet

Attributes Documentation

NoGet = 'NoGet'

TaskState

class consign.decorator.consigntask.**TaskState**(*value*)

基类：Enum

An enumeration.

Attributes Summary

NoStart
RunCompleteCallBack
RunCreateCallBack
TaskDone
TaskRunning

Attributes Documentation

NoStart = 'NoStart'

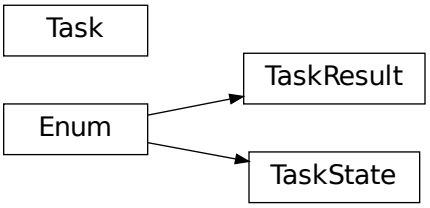
RunCompleteCallBack = 'RunCompleteCallBack'

RunCreateCallBack = 'RunCreateCallBack'

TaskDone = 'TaskDone'

TaskRunning = 'TaskRunning'

Class Inheritance Diagram



consign.decorator.repeatedcalldecorator Module

Functions

repeated_call(debug)	repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用
wraps(wrapped[, assigned, updated])	Decorator factory to apply update_wrapper() to a wrapper function

repeated_call

consign.decorator.repeatedcalldecorator.**repeated_call**(*debug*)

repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用

```
def test(something: any):
    return something

test("hi")
```

当其被修饰时，相当于

```
@repeated_call
def test(something: any):
    return something
```

(续下页)

(接上页)

```
# 此时返回的是一个替代函数，test并没有被真正执行
test("hi")
# 完整的执行过程，等价于没修饰前的test("hi")
test("hi")()
```

repeated_call 并不是必须的，与之相似功能的还有 functools 中的 partial 以及 class 中的 __call__ 方法

> **警告：** 一般而言，他的使用是为了 consign 中的一项规则：yield 一个 function 时，function 应当满足无参的条件
>
> 但值得注意的是，这个条件可能在 **未来被移除**，有违简单易用这个初衷，他是我为了自己旧版项目做的兼容

repeated_call 并不是必须的！因为 consign 只在乎你 yield 的是否是一个无参的可调用对象如果你本身是无参的，可以直接传递函数名

通常情况下，你可以直接在 yield 前调用 function，**这其实意味着你 yield 的是调用函数后的返回值**。

参数

debug (*bool*) –如果为 True，那么返回原函数

抛出

AssertionError –当传入参数 func 不是 callable 时抛出

consign.utils package

consign.utils Package

Functions

asleep(secs)	协程的 sleep 函数，休眠一定时间

asleep

consign.utils.**asleep**(*secs: float*)

协程的 sleep 函数，休眠一定时间

asleep 与 sleep 不同，asleep 是非阻塞的、协程的

asleep 通过 **不断的切换控制权**和 **判断时间**来实现休眠

警告: `asleep` 无法保证执行顺序

因为控制权无法保证谁先获取

举例 1 和 2 函数抛出了委托 sleep，同时 sleep 5 秒

在 5 秒后，无法保证是 1 先于 2 获取到执行的控制权

`asleep` 休眠的时长是 **不精确的**，受到当前任务量和阻塞时间的影响

因为 `asleep` 在转移控制权的过程中，队列里依旧存在其余任务

你无法保证下一个任务是不是阻塞的

如果存在任务是阻塞的，那么他就有可能对 `asleep` 照成影响

小技巧: 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率

例子

举例：1 函数在第一次抛出 `asleep` 5 秒，但 2 函数抛出一个需要 **阻塞执行 7 秒**的 **非协程**任务

假设在单/主线程下执行

那么 2 函数的任务阻塞时间大于 >5 秒，且没有控制权的转移

因而，1 函数无法在 sleep 5 秒后得到立即执行，

`asleep` 为了方便使用，已经被 `chain_reaction` 和 `repeated_call` 修饰

二者分别使得 `asleep` 以顺序方式执行、`asleep` 以有参的形式可以再被无参调用执行

参数

secs (*float*) –休眠的秒数

Submodules

consign.utils.sleep Module

Functions

asleep(secs)	协程的 sleep 函数，休眠一定时间
chain_reaction(func)	chain_reaction 使得嵌套协程函数可以以链式的方式运行.
coroutine([debug, create_callback, ...])	consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行
monotonic()	Monotonic clock, cannot go backward.
repeated_call(debug)	repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用

asleep

consign.utils.sleep.**asleep**(*secs: float*)

协程的 sleep 函数，休眠一定时间

asleep 与 sleep 不同，asleep 是非阻塞的、协程的

asleep 通过 **不断的切换控制权**和 **判断时间**来实现休眠

> **警告：** asleep 无法保证执行顺序
>
> 因为控制权无法保证谁先获取
>
> 举例 1 和 2 函数抛出了委托 sleep，同时 sleep 5 秒
>
> 在 5 秒后，无法保证是 1 先于 2 获取到执行的控制权
>
> asleep 休眠的时长是 **不精确的**，受到当前任务量和阻塞时间的影响
>
> 因为 asleep 在转移控制权的过程中，队列里依旧存在其余任务
>
> 你无法保证下一个任务是不是阻塞的
>
> 如果存在任务是阻塞的，那么他就有可能对 asleep 照成影响
>
> ---
>
> **小技巧：** 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率
>
> ---

> 例子
>
> 举例：1 函数在第一次抛出 `asleep` 5 秒，但 2 函数抛出一个需要 **阻塞执行 7 秒**的 **非协程**任务
>
> 假设在单/主线程下执行
>
> 那么 2 函数的任务阻塞时间大于 >5 秒，且没有控制权的转移
>
> 因而，1 函数无法在 sleep 5 秒后得到立即执行，

`asleep` 为了方便使用，已经被 `chain_reaction` 和 `repeated_call` 修饰

二者分别使得 `asleep` 以顺序方式执行、`asleep` 以有参的形式可以再被无参调用执行

参数

secs (*float*) –休眠的秒数

consign.workarea package

consign.workarea Package

Classes

`ContextVar`	
`WorkArea`([name])	WorkArea 是 consign 的基石，负责规划 Worker 和协程

WorkArea

class `consign.workarea.`**`WorkArea`**(*name: str = 'DEFAULT_WORK_AREA', *args, **kwargs*)

基类：`object`

WorkArea 是 consign 的基石，负责规划 Worker 和协程

`WorkArea` 分离所有 `Worker` 的工作区域，这可以有助于你分配资源以及更好的使用 `Worker`

`WorkArea` 被设计成线程安全的，借助 `ContextVar` 在每个线程中都拥有独立的值，其本质是 `threading.local`

`WorkArea` 默认在导入此库时，创建默认的 `DEFAULT_WORK_AREA` 变量作为默认工作地址，并置于 `builtins` 中：

1. 这意味着你可以通过直接访问 `DEFAULT_WORK_AREA.get()` 获取当前的工作区域

当然并不推荐，一旦你这样使用，你应当明白你想要做些什么，以及会造成那些影响

2. DEFAULT_WORK_AREA 是全局的统一的，依托 ContextVar 可以实现在不同线程中不同的值

不同线程间修改不会出现相互影响

WorkArea 作为上下文管理器其实现的本质就是对 DEFAULT_WORK_AREA 的修改和恢复

WorkArea 会修改 DEFAULT_WORK_AREA 的值来影响“Worker“的运行

WorkArea 本质是一个上下文管理器，但同时支持修饰器写法

修饰器代码来源于 ContextDecorator, 写法相当于语法糖, 他们是相等的:

使用修饰器

```
@cm()
def f():
    # Do stuff
```

使用上下文管理器

```
def f():
    with cm():
        # Do stuff
```

WorkArea 的 name 参数非常重要:

你可以注意到 WorkArea 被 same_name_singleton 修饰，same_name_singleton 是一个单例的修饰器

(新版本，为了适配多进程，修饰器无法被序列化，所以改用元类进行单例，效果不变)

WorkArea 通过 name 区分，同名的情况下只有一个 WorkArea 实例

但值得注意的是:

即使是同一个单例 WorkArea ，也并不意味着 WorkArea 成员变量 old_work_area_queue 存放的值是相同的

old_work_area_queue 的本质是被 ContextVar 包裹的变量

为了防止多线程中共用一个 WorkArea 导致的 LifoQueue 顺序冲突

old_work_area_queue 会在每个单独的线程中创建一个 LifoQueue ，由线程共享，以此保证线程安全

参数

name (*str*) –WorkArea 的名字，同名 WorkArea 是同一个实例，默认是名字使用 DEFAULT_WORK_AREA

小技巧： 理论上能 hash 的值都能传入 name

为了减少意料之外的事情发生，建议输入是 str

Methods Summary

__call__(func)	适配修饰器写法的实现
as_default()	返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

Methods Documentation

__call__ (*func*)

适配修饰器写法的实现

参数

func (*function*) –被包裹的函数中的全部协程函数将会运行于此 WorkArea 的作用范围

static as_default ()

返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

as_default 将会去 builtins 中寻找 DEFAULT_WORK_AREA 变量并返回其中保存的 WorkArea

DEFAULT_WORK_AREA 变量一般会在导入库时被创建

如果没能找到，as_default 会创建 DEFAULT_WORK_AREA 包装并置入 builtins 并返回

返回

返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

Class Inheritance Diagram

Submodules

consign.workarea.workarea Module

Classes

`ContextVar`	
`LifoQueue`([maxsize])	Variant of Queue that retrieves most recently added entries first.
`Queue`([maxsize])	Create a queue object with a given maximum size.
`SameNameSingleton`(*args, **kwargs)	
`WorkArea`([name])	WorkArea 是 consign 的基石，负责规划 Worker 和协程

SameNameSingleton

class consign.workarea.workarea.**SameNameSingleton**(**args*, ***kwargs*)

基类：`type`

Methods Summary

`__call__`([name])	Call self as a function.

Methods Documentation

__call__(*name: str = 'DEFAULT_WORK_AREA', *args, **kwargs*)

Call self as a function.

WorkArea

class consign.workarea.workarea.**WorkArea**(*name: str = 'DEFAULT_WORK_AREA', *args, **kwargs*)

基类：object

WorkArea 是 consign 的基石，负责规划 Worker 和协程

WorkArea 分离所有 Worker 的工作区域，这可以有助于你分配资源以及更好的使用 Worker

WorkArea 被设计成线程安全的，借助 ContextVar 在每个线程中都拥有独立的值，其本质是 threading.local

WorkArea 默认在导入此库时，创建默认的 DEFAULT_WORK_AREA 变量作为默认工作地址，并置于 builtins 中：

1. 这意味着你可以通过直接访问 DEFAULT_WORK_AREA.get() 获取当前的工作区域

当然并不推荐，一旦你这样使用，你应当明白你想要做些什么，以及会造成那些影响

2. DEFAULT_WORK_AREA 是全局的统一的，依托 ContextVar 可以实现在不同线程中不同的值

不同线程间修改不会出现相互影响

WorkArea 作为上下文管理器其实现的本质就是对 DEFAULT_WORK_AREA 的修改和恢复

WorkArea 会修改 DEFAULT_WORK_AREA 的值来影响“Worker“的运行

WorkArea 本质是一个上下文管理器，但同时支持修饰器写法

修饰器代码来源于 ContextDecorator, 写法相当于语法糖, 他们是相等的：

使用修饰器

```
@cm()
def f():
    # Do stuff
```

使用上下文管理器

```
def f():
    with cm():
        # Do stuff
```

WorkArea 的 name 参数非常重要:

你可以注意到 WorkArea 被 same_name_singleton 修饰，same_name_singleton 是一个单例的修饰器

(新版本，为了适配多进程，修饰器无法被序列化，所以改用元类进行单例，效果不变)

WorkArea 通过 name 区分，同名的情况下只有一个 WorkArea 实例

但值得注意的是:

即使是同一个单例 WorkArea ，也并不意味着 WorkArea 成员变量 old_work_area_queue 存放的值是相同的

old_work_area_queue 的本质是被 ContextVar 包裹的变量

为了防止多线程中共用一个 WorkArea 导致的 LifoQueue 顺序冲突

old_work_area_queue 会在每个单独的线程中创建一个 LifoQueue ，由线程共享，以此保证线程安全

参数

name (*str*) –WorkArea 的名字，同名 WorkArea 是同一个实例，默认是名字使用 DEFAULT_WORK_AREA

小技巧: 理论上能 hash 的值都能传入 name

为了减少意料之外的事情发生，建议输入是 str

Methods Summary

__call__(func)	适配修饰器写法的实现
as_default()	返回名为 DEFAULT_WORK_AREA 默认的 WorkArea

Methods Documentation

__call__(*func*)

适配修饰器写法的实现

参数

func (*function*) –被包裹的函数中的全部协程函数将会运行于此 `WorkArea` 的作用范围

static as_default()

返回名为 `DEFAULT_WORK_AREA` 默认的 `WorkArea`

`as_default` 将会去 `builtins` 中寻找 `DEFAULT_WORK_AREA` 变量并返回其中保存的 `WorkArea`

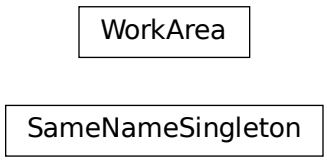
`DEFAULT_WORK_AREA` 变量一般会在导入库时被创建

如果没能找到，`as_default` 会创建 `DEFAULT_WORK_AREA` 包装并置入 `builtins` 并返回

返回

返回名为 `DEFAULT_WORK_AREA` 默认的 `WorkArea`

Class Inheritance Diagram



consign.worker package

consign.worker Package

Functions

wait(task, *[, time_out])	wait 阻塞等待一个 Task 任务完成，并在期间参与工作

wait

consign.worker.**wait**(*task*, *, *time_out=0.1*)

wait 阻塞等待一个 Task 任务完成，并在期间参与工作

wait 是 Supervisor 类的上层封装

简单的讲，他的作用就是轮询等待一个 Task

当这一个 Task 的状态转变为结束后 返回此 Task 的 value

期间他会将自身代入成为一个 Worker 进行 loop_work

某种程度上看，wait 相当于特别一点的 loop_work ，主要在于他的 WorkArea 继承机制

由于继承机制，往往你需要明确你的 wait 工作在哪个区域下？为哪个区域而工作？

wait 一个 Task 时，他会继承 Task 中的 WorkArea 并生成对应的 CoroutineWorker 进行 loop_work

wait 不对 约定的结束信号做处理碰到结束信号时他会重新提交回原 WorkArea

wait 轮询的时间是 **不精确的**，受到当前任务量和阻塞时间的影响

因为 wait 的轮询任务在转移控制权的过程中，队列里依旧存在其余任务

你无法保证下一个任务是不是阻塞的

如果存在任务是阻塞的，那么他就有可能对 wait 照成影响

小技巧： 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率

---

wait 的原理是：wait 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 wait 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

参数

- **task** (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回
- **time_out** –最短轮询的间隔，他一般用在多线程中，当对应 WorkArea 的 queue 为空时，time_out 才会触发

> **警告：** 尽可能不要设置 time_out 为 None
>
> 这或许会导致 wait 被阻塞没法正常退出

返回

目标 `Task` 中的 `value`

Classes

AsyncWorker([work_area_name, work_area])	AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker
CoroutineWorker([work_area_name, work_area])	CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石
Supervisor(task)	`wait` 的底层实现

AsyncWorker

class consign.worker.**AsyncWorker**(*work_area_name: str = 'DEFAULT_WORK_AREA'*, *, *work_area=None*)

基类：*CoroutineWorker*

AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker

`AsyncWorker` 可以将 耗时的事情分配给许多线程去处理，得以实现异步

参见:

当然，如果你使用 `chain_reaction` ，那么他依旧还是顺序执行的

所以如果性能有更高要求，建议灵活使用 `wait`

`AsyncWorker` 的使用方法通常是 `init_thread` 创建线程，很少且不建议使用 `loop_work`

如果你使用 `AsyncWorker` ，那么 `loop_work(forever=False)` **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性以及框架的获取提交导致的

小技巧: 通常在多线程下使用 `loop_work(forever=True)` 或者重复 `while True` 使用 `loop_work(forever=False)`

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

所以如果 `AsyncWorker` 不使用 `init_thread` 直接使用 `loop_work` 其实和 `CoroutineWorker` 效果是 **相同的**

都是使用 **当前进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

`AsyncWorker` 创建的线程默认是 **守护线程**，如果你的程序没有阻塞一路到 exit 的话，可能导致协程任务异常被终止

所以在 **非 cmd** 下，最好使用 `while True` 或者 `loop_work(forever=True)` 去阻塞，或者设置参数选择生成非守护线程。

例子

```
some_coroutine()

aw = AsyncWorker()
using_thread_num = 5
# name_iter 传入为None的时候，默认使用uuid去生成名字
aw.init_thread(using_thread_num, name_iter=(f"DEFAULT_WORK_AREA_{i}" for i in␣
↪range(using_thread_num)))

# 如果程序主进程没有其他内容，那么需要你使用阻塞，可以使用while或者Worker的loop_
↪work
some_loop()
```

参数

- **work_area_name** (*str*) –`WorkArea` 的名字，`AsyncWorker` 会通过名字获取对应的 `WorkArea`
- **work_area** (`WorkArea`) –work_area 是显式参数，需要显式调用

  可以直接指定 `WorkArea`，如果直接指定，会跳过 `work_area_name` 的查找过程

抛出

AssertionError –当传入参数 `work_area_name` 无法被找到时抛出

Attributes Summary

thread_list	在获取 thread_list 时，同时检测并删除 __thread_list 中已经关闭的线程

Methods Summary

clear_dead_thread()	一般不建议外部调用
create_thread([name, daemon])	创建一个持续运行的线程，用于对协程函数进行执行
init_thread(create_num[, name_iter, daemon])	可以一次初始化多个线程
submit_thread_stop_flag()	向对应 `WorkArea` 提交约定的结束信号

Attributes Documentation

thread_list

在获取 thread_list 时，同时检测并删除 __thread_list 中已经关闭的线程

> **警告：** 注意这个处理会同步修改到 __thread_list
>
> 返回的 List 并不是副本而是某种’引用’

返回

存放线程的列表

Methods Documentation

clear_dead_thread()

一般不建议外部调用

遍历列表并判断线程是否存活

如果线程不存活，那么从列表中删除此线程

返回

None

create_thread(*name=None*, *daemon=True*)

创建一个持续运行的线程，用于对协程函数进行执行

参数

- **name** (*str*) –创建的线程名，如果为 None，则使用 uuid 创建
- **daemon** (*bool*) –是否守护线程

返回

创建的线程对象

init_thread(*create_num*, *name_iter=None*, *daemon=True*)

可以一次初始化多个线程

是对 `create_thread` 的上层封装

参数

- **create_num** (*int*) –创建对应线程的数量
- **name_iter** –生成器或者容器，如果没有默认使用传递 None 即使用 uuid 命名

> **警告：** 没有对 `name_iter` 长度做校验但 `name_iter` 长度应该跟 create_num 相同，避免出乎意料的结果

- **daemon** (*bool*) –创建的是否是守护线程

返回

submit_thread_stop_flag()

向对应 `WorkArea` 提交约定的结束信号

> **警告：** 约定的结束信号不一定能立马结束，因为队列中可能还有其他内容

返回

CoroutineWorker

class consign.worker.**CoroutineWorker**(*work_area_name: str = 'DEFAULT_WORK_AREA'*, *, *work_area=None*)

基类：`object`

CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石

`CoroutineWorker` 可以将 控制权切换以实现并发的效果

`CoroutineWorker` 的使用方法通常是 `loop_work` 阻塞并完成当前队列中的全部内容

如果你使用 `CoroutineWorker` 在多线程/进程环境下，那么 `loop_work(forever=False)` **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性以及框架的获取提交导致的

小技巧： 通常在多线程下使用 `loop_work(forever=True)` 或者重复 `while True` 使用 `loop_work(forever=False)`

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

使用的是 **当前进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

例子

```
some_coroutine()
```

(续下页)

(接上页)

```
cw = CoroutineWorker()
cw.loop_work()
```

参数

- **work_area_name** (*str*) –WorkArea 的名字，CoroutineWorker 会通过名字获取对应的 WorkArea
- **work_area** (WorkArea) –work_area 是显式参数，需要显式调用

  可以直接指定 WorkArea ，如果直接指定，会跳过 work_area_name 的查找过程

抛出

AssertionError –当传入参数 work_area_name 无法被找到时抛出

Methods Summary

loop_work(*[, time_out, forever])	loop_work 阻塞并完成当前队列中的全部内容或者收到停止信号
qsize()	获得 WorkArea 队列中的大概数量
submit_work(*args)	提交任务到 WorkArea 队列
work_once([time_out])	CoroutineWorker 获取一次 WorkArea 队列中的内容并执行一次

Methods Documentation

loop_work (*, *time_out=0.1*, *forever=True*)

loop_work 阻塞并完成当前队列中的全部内容或者收到停止信号

> **警告：** 在多线程/进程环境下
>
> forever 参数为 False 时需要考虑 队列 QSIZE 的不确定性
>
> **小技巧：** 通常在多线程下使用 loop_work(forever=True) 或者重复 while True 使用 loop_work(forever=False)

参数

- **time_out** –等待 WorkArea 队列获取内容的等待超时时长

- **forever** –是否永久阻塞

  如果 `forever` 为 False，那么 `loop_work` 会考虑队列的 `qsize` 是否 <=0

  如果队列的 `qsize` <=0，那么退出循环

  如果你需要指定某个任务完成并阻塞，那么建议使用 `wait` 来确保完成优于直接使用 `forever` =True

返回

None

抛出

ValueError –当参数 `time_out` 不正确时抛出

qsize()

获得 `WorkArea` 队列中的大概数量

> **警告：** 由于多线程或者多进程的上下文，这个数字是不可靠的。
>
> 请注意，这可能会在 Unix 平台上引起 NotImplementedError ，如 macOS ，因为其上没有实现 sem_getvalue() 。

返回

获得 `WorkArea` 队列中的大概数量

抛出

NotImplementedError –能会在 Unix 平台上引起 NotImplementedError ，如 macOS ，因为其上没有实现 sem_getvalue()

submit_work(**args*)

提交任务到 `WorkArea` 队列

不建议外部调用，如果调用需要明白自己正在做什么

参数

args –按照规定 `WorkArea` 队列中是只应当 put 一个定义好的 tuple

其格式为 (yield_value, generator, receipts)

这里因为不向外调用，所以使用可变参偷了个懒，就无需再手动创建 tuple

返回

None

抛出

Full –当 `WorkArea` 队列存放满时触发

work_once(*time_out=None*)

CoroutineWorker 获取一次 WorkArea 队列中的内容并执行一次

任意一种情况返回都会视作一次 Work

参数

time_out –等待 WorkArea 队列获取内容的等待超时时长

返回

返回 bool 类型

False 意味着收到了约定的结束信号

True 表示正常结束 Work

抛出

- **Empty** –当 WorkArea 队列为空时阻塞时长超过参数 time_out 时抛出
- **ValueError** –当参数 time_out 不正确时抛出

Supervisor

class consign.worker.**Supervisor**(*task*)

基类：object

wait 的底层实现

wait 是 Supervisor 类的上层封装，wait 向外暴露，但 Supervisor 不被直接暴露

Supervisor 能够阻塞等待一个 Task 任务完成，并在期间参与工作

Supervisor 的原理是：Supervisor 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 Supervisor 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

更多详情请查看 wait 函数

参数

task (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回

Methods Summary

polling_func()	轮询任务，用于判断 `Task` 的状态
run_until_complete([time_out])	阻塞、工作直到轮询任务完成

Methods Documentation

polling_func()

轮询任务，用于判断 `Task` 的状态

他是一个 `coroutine` 修饰的协程函数

`Supervisor` 会自动继承 `Task` 中的 `WorkArea` 并在此提交

每次执行控制权切换前判断 `Task` 的状态是否是已完成，完成会告知 `Supervisor`

返回

None

run_until_complete(*time_out=0.1*)

阻塞、工作直到轮询任务完成

参数

time_out –其实就是 CoroutineWorker 中 work_once 的参数

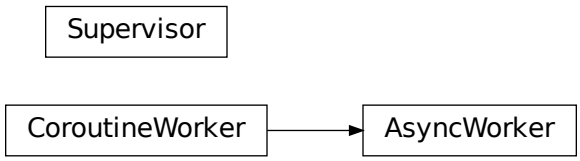
等待 `WorkArea` 队列获取内容的等待超时时长

也是最短轮询的间隔，他一般用在多线程中，当对应 `WorkArea` 的 `queue` 为空时，`time_out` 才会触发

返回

`Task` 类中的 value，也就是对应协程的返回值

Class Inheritance Diagram



Submodules

consign.worker.asyncworker Module

Classes

AsyncWorker([work_area_name, work_area])	AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker
CoroutineWorker([work_area_name, work_area])	CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石
Process	DummyProcess 的别名
repeat(object [,times])	for the specified number of times.

AsyncWorker

class consign.worker.asyncworker.**AsyncWorker**(*work_area_name: str = 'DEFAULT_WORK_AREA'*, *, *work_area=None*)

基类：*CoroutineWorker*

AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker

AsyncWorker 可以将 耗时的事情分配给许多线程去处理，得以实现异步

参见:

当然，如果你使用 chain_reaction ，那么他依旧还是顺序执行的

所以如果性能有更高要求，建议灵活使用 wait

AsyncWorker 的使用方法通常是 init_thread 创建线程，很少且不建议使用 loop_work

如果你使用 AsyncWorker ，那么 loop_work(forever=False) **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性以及框架的获取提交导致的

小技巧: 通常在多线程下使用 loop_work(forever=True) 或者重复 while True 使用 loop_work(forever=False)

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

所以如果 AsyncWorker 不使用 init_thread 直接使用 loop_work 其实和 CoroutineWorker 效果是 **相同的**

都是使用 **当前进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

AsyncWorker 创建的线程默认是 **守护线程**，如果你的程序没有阻塞一路到 exit 的话，可能导致协程任务异常被终止

所以在 **非 cmd** 下，最好使用 while True 或者 loop_work(forever=True) 去阻塞，或者设置参数选择生成非守护线程。

例子

```
some_coroutine()

aw = AsyncWorker()
using_thread_num = 5
# name_iter 传入为None的时候，默认使用uuid去生成名字
aw.init_thread(using_thread_num, name_iter=(f"DEFAULT_WORK_AREA_{i}" for i in␣
↪range(using_thread_num)))

# 如果程序主进程没有其他内容，那么需要你使用阻塞，可以使用while或者Worker的loop_
↪work
some_loop()
```

参数

- **work_area_name** (*str*) –WorkArea 的名字，AsyncWorker 会通过名字获取对应的 WorkArea
- **work_area** (WorkArea) –work_area 是显式参数，需要显式调用

  可以直接指定 WorkArea ，如果直接指定，会跳过 work_area_name 的查找过程

抛出

AssertionError –当传入参数 work_area_name 无法被找到时抛出

Attributes Summary

thread_list	在获取 thread_list 时，同时检测并删除 __thread_list 中已经关闭的线程

Methods Summary

clear_dead_thread()	一般不建议外部调用
create_thread([name, daemon])	创建一个持续运行的线程，用于对协程函数进行执行
init_thread(create_num[, name_iter, daemon])	可以一次初始化多个线程
submit_thread_stop_flag()	向对应 WorkArea 提交约定的结束信号

Attributes Documentation

thread_list

在获取 thread_list 时，同时检测并删除 __thread_list 中已经关闭的线程

> **警告：** 注意这个处理会同步修改到 __thread_list
>
> 返回的 List 并不是副本而是某种’引用’

返回

存放线程的列表

Methods Documentation

clear_dead_thread()

一般不建议外部调用

遍历列表并判断线程是否存活

如果线程不存活，那么从列表中删除此线程

返回

None

create_thread(*name=None*, *daemon=True*)

创建一个持续运行的线程，用于对协程函数进行执行

参数

- **name** (*str*) –创建的线程名，如果为 None，则使用 uuid 创建
- **daemon** (*bool*) –是否守护线程

返回

创建的线程对象

init_thread(*create_num*, *name_iter=None*, *daemon=True*)

可以一次初始化多个线程

是对 `create_thread` 的上层封装

参数

- **create_num** (*int*) –创建对应线程的数量
- **name_iter** –生成器或者容器，如果没有默认使用传递 None 即使用 uuid 命名

> **警告：** 没有对 `name_iter` 长度做校验但 `name_iter` 长度应该跟 create_num 相同，避免出乎意料的结果

- **daemon** (*bool*) –创建的是否是守护线程

返回

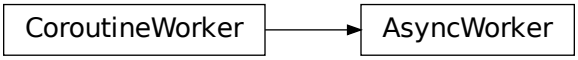
submit_thread_stop_flag()

向对应 `WorkArea` 提交约定的结束信号

> **警告：** 约定的结束信号不一定能立马结束，因为队列中可能还有其他内容

返回

Class Inheritance Diagram



consign.worker.coroutineworker Module

Classes

`AutoCallback`(callback, exceptions)	
`AutoReceipts`(receipts)	
`CheckChainReaction`(result)	
`CoroutineWorker`([work_area_name, work_area])	CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石
`Empty`	Exception raised by Queue.get(block=0)/get_nowait().

CoroutineWorker

class consign.worker.coroutineworker.**CoroutineWorker**(*work_area_name: str = 'DEFAULT_WORK_AREA'*, *, *work_area=None*)

基类：object

CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石

CoroutineWorker 可以将 控制权切换以实现并发的效果

CoroutineWorker 的使用方法通常是 loop_work 阻塞并完成当前队列中的全部内容

如果你使用 CoroutineWorker 在多线程/进程环境下，那么 loop_work(forever=False) **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性以及框架的获取提交导致的

小技巧: 通常在多线程下使用 loop_work(forever=True) 或者重复 while True 使用 loop_work(forever=False)

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

使用的是 **当前进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

例子

```
some_coroutine()

cw = CoroutineWorker()
cw.loop_work()
```

参数

- **work_area_name** (*str*) –WorkArea 的名字，CoroutineWorker 会通过名字获取对应的 WorkArea
- **work_area** (WorkArea) –work_area 是显式参数，需要显式调用

  可以直接指定 WorkArea ，如果直接指定，会跳过 work_area_name 的查找过程

抛出

AssertionError –当传入参数 work_area_name 无法被找到时抛出

Methods Summary

loop_work(*[, time_out, forever])	loop_work 阻塞并完成当前队列中的全部内容或者收到停止信号
qsize()	获得 WorkArea 队列中的大概数量
submit_work(*args)	提交任务到 WorkArea 队列
work_once([time_out])	CoroutineWorker 获取一次 WorkArea 队列中的内容并执行一次

Methods Documentation

loop_work(*, *time_out=0.1*, *forever=True*)

loop_work 阻塞并完成当前队列中的全部内容或者收到停止信号

> **警告：** 在多线程/进程环境下
>
> forever 参数为 False 时需要考虑 队列 QSIZE 的不确定性
>
> **小技巧：** 通常在多线程下使用 loop_work(forever=True) 或者重复 while True 使用 loop_work(forever=False)

参数

- **time_out** –等待 WorkArea 队列获取内容的等待超时时长
- **forever** –是否永久阻塞

  如果 forever 为 False，那么 loop_work 会考虑队列的 qsize 是否 <=0

  如果队列的 qsize <=0，那么退出循环

  如果你需要指定某个任务完成并阻塞，那么建议使用 wait 来确保完成优于直接使用 forever =True

返回

None

抛出

ValueError –当参数 time_out 不正确时抛出

qsize()

获得 WorkArea 队列中的大概数量

> **警告：** 由于多线程或者多进程的上下文，这个数字是不可靠的。
>
> 请注意，这可能会在 Unix 平台上引起 NotImplementedError ，如 macOS ，因为其上没有实现 sem_getvalue() 。

返回

获得 `WorkArea` 队列中的大概数量

抛出

`NotImplementedError` –能会在 Unix 平台上引起 NotImplementedError ，如 macOS ，因为其上没有实现 sem_getvalue()

`submit_work`(**args*)

提交任务到 `WorkArea` 队列

不建议外部调用，如果调用需要明白自己正在做什么

参数

`args` –按照规定 `WorkArea` 队列中是只应当 put 一个定义好的 tuple

其格式为 (yield_value, generator, receipts)

这里因为不向外调用，所以使用可变参偷了个懒，就无需再手动创建 tuple

返回

None

抛出

`Full` –当 `WorkArea` 队列存放满时触发

`work_once`(*time_out=None*)

`CoroutineWorker` 获取一次 `WorkArea` 队列中的内容并执行一次

任意一种情况返回都会视作一次 Work

参数

`time_out` –等待 `WorkArea` 队列获取内容的等待超时时长

返回

返回 bool 类型

False 意味着收到了约定的结束信号

True 表示正常结束 Work

抛出

- **`Empty`** –当 `WorkArea` 队列为空时阻塞时长超过参数 `time_out` 时抛出
- **`ValueError`** –当参数 `time_out` 不正确时抛出

Class Inheritance Diagram

consign.worker.iterationcontext Module

Functions

except_and_pass_func(*args, **kwargs)	

except_and_pass_func

`consign.worker.iterationcontext.`**`except_and_pass_func`**(**args*, ***kwargs*)

Classes

AutoCallback(callback, exceptions)	
AutoClosing(thing[, exceptions])	
AutoReceipts(receipts)	
CheckChainReaction(result)	
NoChainReactionError	

AutoCallback

class consign.worker.iterationcontext.**AutoCallback**(*callback*, *exceptions*)

基类：object

AutoClosing

class consign.worker.iterationcontext.**AutoClosing**(*thing*, *exceptions=<class 'StopIteration'>*)

基类：*AutoCallback*

Methods Summary

thing_close(exc_type, exc_val, exc_tb)	

Methods Documentation

thing_close(*exc_type*, *exc_val*, *exc_tb*)

AutoReceipts

class consign.worker.iterationcontext.**AutoReceipts**(*receipts*)

基类：*AutoCallback*

Methods Summary

send_receipts(exc_type, exc_val, exc_tb)	

Methods Documentation

send_receipts(*exc_type*, *exc_val*, *exc_tb*)

CheckChainReaction

class consign.worker.iterationcontext.**CheckChainReaction**(*result*)

基类：*AutoCallback*

Methods Summary

raise_error_func(*args, **kwargs)	
set_chain_reaction(parent_generator, ...)	

Methods Documentation

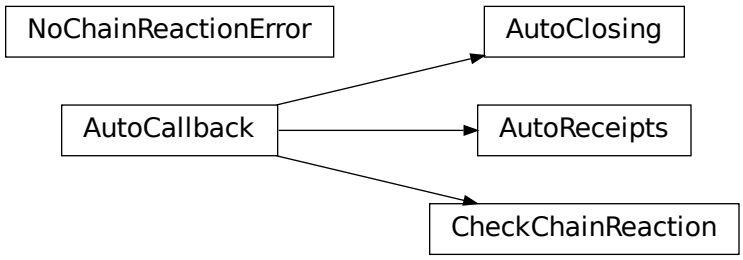
raise_error_func(**args*, ***kwargs*)

set_chain_reaction(*parent_generator*, *parent_receipts*)

NoChainReactionError

exception consign.worker.iterationcontext.**NoChainReactionError**

Class Inheritance Diagram



consign.worker.wait Module

Functions

`coroutine`([debug, create_callback, ...])	consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行
`pass_func`()	
`wait`(task, *[, time_out])	wait 阻塞等待一个 Task 任务完成，并在期间参与工作

pass_func

`consign.worker.wait.`**`pass_func`**`()`

wait

`consign.worker.wait.`**`wait`**(*task*, *, *time_out=0.1*)

wait 阻塞等待一个 Task 任务完成，并在期间参与工作

`wait` 是 `Supervisor` 类的上层封装

简单的讲，他的作用就是轮询等待一个 `Task`

当这一个 `Task` 的状态转变为结束后 返回此 Task 的 value

期间他会将自身代入成为一个 `Worker` 进行 `loop_work`

某种程度上看，`wait` 相当于特别一点的 `loop_work` ，主要在于他的 `WorkArea` 继承机制

由于继承机制，往往你需要明确你的 wait 工作在哪个区域下？为哪个区域而工作？

`wait` 一个 `Task` 时，他会继承 `Task` 中的 `WorkArea` 并生成对应的 `CoroutineWorker` 进行 `loop_work`

`wait` 不对 约定的结束信号做处理碰到结束信号时他会重新提交回原 `WorkArea`

`wait` 轮询的时间是 **不精确的**，受到当前任务量和阻塞时间的影响

因为 `wait` 的轮询任务在转移控制权的过程中，队列里依旧存在其余任务

你无法保证下一个任务是不是阻塞的

如果存在任务是阻塞的，那么他就有可能对 `wait` 照成影响

小技巧： 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率

wait 的原理是：wait 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 wait 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

参数

- **task** (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回
- **time_out** –最短轮询的间隔，他一般用在多线程中，当对应 WorkArea 的 queue 为空时，time_out 才会触发

> **警告：** 尽可能不要设置 time_out 为 None
>
> 这或许会导致 wait 被阻塞没法正常退出

返回

目标 Task 中的 value

Classes

AutoCallback(callback, exceptions)	
CoroutineWorker([work_area_name, work_area])	CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石
Empty	Exception raised by Queue.get(block=0)/get_nowait().
Supervisor(task)	wait 的底层实现
Task(order)	
TaskResult(value)	An enumeration.
TaskState(value)	An enumeration.
WorkArea([name])	WorkArea 是 consign 的基石，负责规划 Worker 和协程

Supervisor

class consign.worker.wait.**Supervisor**(*task*)

基类：object

wait 的底层实现

wait 是 Supervisor 类的上层封装，wait 向外暴露，但 Supervisor 不被直接暴露

Supervisor 能够阻塞等待一个 Task 任务完成，并在期间参与工作

Supervisor 的原理是：Supervisor 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 Supervisor 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

更多详情请查看 wait 函数

参数

task (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回

Methods Summary

polling_func()	轮询任务，用于判断 Task 的状态
run_until_complete([time_out])	阻塞、工作直到轮询任务完成

Methods Documentation

polling_func()

轮询任务，用于判断 Task 的状态

他是一个 coroutine 修饰的协程函数

Supervisor 会自动继承 Task 中的 WorkArea 并在此提交

每次执行控制权切换前判断 Task 的状态是否是已完成，完成会告知 Supervisor

返回

None

run_until_complete(*time_out=0.1*)

阻塞、工作直到轮询任务完成

参数

time_out –其实就是 CoroutineWorker 中 work_once 的参数

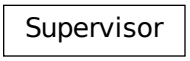
等待 WorkArea 队列获取内容的等待超时时长

也是最短轮询的间隔，他一般用在多线程中，当对应 `WorkArea` 的 `queue` 为空时，`time_out` 才会触发

返回

`Task` 类中的 value，也就是对应协程的返回值

Class Inheritance Diagram



CHAPTER 2

API

consign

2.1 consign

Modules

consign.decorator

consign.utils

consign.workarea

consign.worker

2.1.1 consign.decorator

Modules

consign.decorator.chainreactiondecorator	
consign.decorator.consigndecorator	
consign.decorator.consignorder	
consign.decorator.consigntask	
consign.decorator.repeatedcalldecorator	

consign.decorator.chainreactiondecorator

Functions

chain_reaction(func)	chain_reaction 使得嵌套协程函数可以以链式的方式运行.

consign.decorator.consigndecorator

Functions

coroutine([debug, create_callback, ...])	consign 的核心，coroutine 使得被修饰函数可以以协程的方式被执行

consign.decorator.consignorder

Classes

ConsignOrder(consignor_func[, ...])

consign.decorator.consigntask

Classes

Task(order)	
TaskResult(value)	An enumeration.
TaskState(value)	An enumeration.

consign.decorator.repeatedcalldecorator

Functions

repeated_call(debug)	repeated_call 函数/修饰器可以将一个函数修饰成闭包，使得函数以有参形式调用后可以再一次被无参调用

2.1.2 consign.utils

Modules

consign.utils.sleep

consign.utils.sleep

Functions

asleep(secs)	协程的 sleep 函数，休眠一定时间

2.1.3 consign.workarea

Classes

WorkArea([name])	WorkArea 是 consign 的基石，负责规划 Worker 和协程

2.1.4 consign.worker

Modules

consign.worker.asyncworker	
consign.worker.coroutineworker	
consign.worker.iterationcontext	
consign.worker.wait(task, *[, time_out])	wait 阻塞等待一个 Task 任务完成，并在期间参与工作

consign.worker.asyncworker

Classes

AsyncWorker([work_area_name, work_area])	AsyncWorker 异步协程 Worker，AsyncWorker 能够创建线程以应对耗时的阻塞 IO，基于 CoroutineWorker

consign.worker.coroutineworker

Classes

CoroutineWorker([work_area_name, work_area])	CoroutineWorker 协程 Worker，CoroutineWorker 能够以并发的方式执行特定的协程函数，是所有 Worker 的基石

consign.worker.iterationcontext

Functions

except_and_pass_func(*args, **kwargs)

Classes

AutoCallback(callback, exceptions)
AutoClosing(thing[, exceptions])
AutoReceipts(receipts)
CheckChainReaction(result)

Exceptions

NoChainReactionError

consign.worker.wait

`consign.worker.`**`wait`**(*task*, *, *time_out=0.1*)

wait 阻塞等待一个 Task 任务完成，并在期间参与工作

`wait` 是 `Supervisor` 类的上层封装

简单的讲，他的作用就是轮询等待一个 `Task`

当这一个 `Task` 的状态转变为结束后 返回此 Task 的 value

期间他会将自身代入成为一个 `Worker` 进行 `loop_work`

某种程度上看，`wait` 相当于特别一点的 `loop_work` ，主要在于他的 `WorkArea` 继承机制

由于继承机制，往往你需要明确你的 wait 工作在哪个区域下？为哪个区域而工作？

`wait` 一个 `Task` 时，他会继承 `Task` 中的 `WorkArea` 并生成对应的 `CoroutineWorker` 进行 `loop_work`

`wait` 不对 约定的结束信号做处理碰到结束信号时他会重新提交回原 `WorkArea`

`wait` 轮询的时间是 **不精确的**，受到当前任务量和阻塞时间的影响

因为 `wait` 的轮询任务在转移控制权的过程中，队列里依旧存在其余任务

你无法保证下一个任务是不是阻塞的

如果存在任务是阻塞的，那么他就有可能对 `wait` 照成影响

小技巧: 你可以通过创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**提高控制权转换的频率

---

wait 的原理是：wait 向此 Task 类中的 WorkArea 提交一个 轮询任务

轮询任务的作用就是每次执行控制权切换前判断 Task 的状态是否是已完成

当 wait 在接收到 Task 完成信号后，会在执行完当前手头内容后退出阻塞并返回返回值

参数

- **task** (Task) –需要等待的 Task 类, 如果传入的对象并非 Task 类，原样返回
- **time_out** –最短轮询的间隔，他一般用在多线程中，当对应 WorkArea 的 queue 为空时，time_out 才会触发

> **警告:** 尽可能不要设置 time_out 为 None
>
> 这或许会导致 wait 被阻塞没法正常退出

返回

目标 Task 中的 value

Generator-based Coroutines, Easy to use, Using the yield syntax

consign 是基于 generator 的协程框架，易于使用，使用 yield 语法，同时允许普通函数和生成器函数

consign 可以使函数得以以协程的方式运行，以更低的代码侵入性，获得更高的效率

> **警告:** 由于依托 generator 的缘故，consign 中的异步在 **Win 环境**下 **无法使用多进程**，目前未测试其他系统。
>
> 因为目前 **没有库支持反序列化** generator 。
>
> 而需要序列化好像是因为 Win 中没有 os.fork ，只能去模拟，所以 **不知道 Unix 系统能不能使用多进程**。
>
> 你是 **可以在多进程的程序中使用 consign** ，但存在一定限制（并不大），建议使用 **组合**替代 **继承**可以避免大部分问题。

CHAPTER 3

文档介绍

文档使用 *Sphinx* 框架，*furo* 主题，左侧侧边栏是关键的部分

回到此页

需要回到 此页面 (index.html) 可以通过点击左上角的 `consign 文档字样`

搜索框

左侧是搜索框可以直接查询函数

注意的是函数结果可能有多个，同名的函数却在不同长短的路径下。

这不影响，因为他们都是相同的内容，只是路径不同。

建议选择路径最全的 `包名 + 模块名 + 文件名 + 类`，他们通常可以通过点击 `源代码`跳转到 GitHub 中的源代码

consign

左侧侧边栏的 consign 通过点击可以查看全部函数列表。

> **注意:** 直接查看的函数 (即没有完整路径的函数) 点击 `源代码` 几乎无法跳转到源代码
>
> 这是由于自动生成文档导致的
>
> 解决办法是通过找到 consign 中靠后的 `consign.xxx package` 中具体的 `xxx Module` 中的函数
>
> 点击正确的函数后提示的 `源代码` 才能正确跳转 GitHub 源代码，多多担待
>
> ---
>
> **小技巧:** 或者通过 `API` 中找到对应函数并跳转也是可以的

API

左侧侧边栏的 api 是 consign 全模块的概览图。

你可以快速访问模块下具体有那些文件，以及获得每个文件中的函数列表以及他们的一句话描述。

如果需要查看函数详细，可以点击函数名跳转到具体函数页面

其他

参见:

存在部分不导出的函数，即无法通过直接从 consing 库中得到的函数，他们通常是其他函数的基石

只能在靠后的 `consign.xxx package` 中具体的 `xxx Module` 中，或者 `API` 中查看

我写了注释但不知道为什么 autodoc 不对他们起作用，或许是配置不正确?

如果你对他们的实现感兴趣，最好直接看 GitHub 源代码，我都留了注释

CHAPTER 4

快速开始

consign 几乎没有需要安装的依赖，在 GitHub 中开箱即用即可

参见:

现经测试, 在 WIN10、PYTHON==3.6 (理论上 python>=3.4 版本兼容) 运行良好，期待完成更多测试

CHAPTER 5

例子

最简单的 (处于 cmd 模式下)

CoroutineWorker

```
>>> from consign import coroutine, asleep, CoroutineWorker
>>> import threading
>>> @coroutine
... def my_test(name: str):
...     print(f"{name} start in {threading.currentThread()}")
...     result = yield asleep(3)
...     print(f"{name} end in {threading.currentThread()} result is {result}")
...     return name
...
>>> test_task1, test_task2 = my_test("task1"), my_test("task2")
>>> test_task1
{
   'task_state': <TaskState.NoStart: 'NoStart'>,
   'value': <TaskResult.NoGet: 'NoGet'>,
   'work_area': <'DEFAULT_WORK_AREA' Work at 0x25940db46a0 and in <_
↪MainThread(MainThread, started 31352)>>,
   'order': {
      'consignor_func': <function my_test at 0x000002593EEF2EA0>,
      'create_area': <'DEFAULT_WORK_AREA' Work at 0x25940db46a0 and in <_
↪MainThread(MainThread, started 31352)>>,
```

(续下页)

(接上页)

```
        'create_callback': None,
        'complete_callback': None
    }
}
>>> CoroutineWorker().loop_work(forever=False)
task1 start in <_MainThread(MainThread, started 24332)>
task2 start in <_MainThread(MainThread, started 24332)>
task1 end in <_MainThread(MainThread, started 24332)> result is None
task2 end in <_MainThread(MainThread, started 24332)> result is None
>>> test_task1
{
    'task_state': <TaskState.TaskDone: 'TaskDone'>,
    'value': 'task1',
    'work_area': <'DEFAULT_WORK_AREA' Work at 0x23fbe9f46d8 and in <_
↪MainThread(MainThread, started 24332)>>,
    'order': {
        'consignor_func': <function my_test at 0x0000023FBCB42EA0>,
        'create_area': <'DEFAULT_WORK_AREA' Work at 0x23fbe9f46d8 and in <_
↪MainThread(MainThread, started 24332)>>,
        'create_callback': None,
        'complete_callback': None
    }
}
```

AsyncWorker

```
>>> from consign import coroutine, asleep, AsyncWorker
>>> import threading, time
>>> @coroutine
... def my_test(name: str):
...     print(f"{name} start in {threading.currentThread()}")
...     result = yield time.sleep(3)
...     print(f"{name} end in {threading.currentThread()} result is {result}")
...     return name
...
>>> aw = AsyncWorker()
>>> aw.init_thread(5)
>>> test_task1, test_task2 = my_test("task1"), my_test("task2")
task1 start in <DummyProcess(consign_98cfdee9-aea0-47d7-b501-0b5bfd277a39, started␣
↪daemon 22920)>
task2 start in <DummyProcess(consign_184134fe-2fbe-4161-8458-d186aba693e8, started␣
↪daemon 31020)>
```

(续下页)

(接上页)

```
task1 end in <DummyProcess(consign_98cfdee9-aea0-47d7-b501-0b5bfd277a39, started␣
↪daemon 22920)> result is None
task2 end in <DummyProcess(consign_184134fe-2fbe-4161-8458-d186aba693e8, started␣
↪daemon 31020)> result is None
>>> test_task1
{
   'task_state': <TaskState.TaskDone: 'TaskDone'>,
   'value': 'task1',
   'work_area': <'DEFAULT_WORK_AREA' Work at 0x1b7034046d8 and in <_
↪MainThread(MainThread, started 17700)>>,
   'order': {
      'consignor_func': <function my_test at 0x000001B701572EA0>,
      'create_area': <'DEFAULT_WORK_AREA' Work at 0x1b7034046d8 and in <_
↪MainThread(MainThread, started 17700)>>,
      'create_callback': None,
      'complete_callback': None
   }
}
```

细节部分解释

coroutine

@coroutine 是修饰器，也是 consign 的关键，**他修饰的函数会以委托协程的方式运行**。

如果函数是一个 **生成器函数**，那么当函数运行到 yield 时会 **被切换控制权**。

如果函数是 **非生成器函数**，那么 @coroutine 会将其 **包装成生成器函数**并继续下去，但同时也失去了函数内控制权切换的能力

@coroutine 会生成一些信息 `ConsignOrder` ，以及每次运行时也会生成 `Task`

所有被修饰的函数都会变成 **非阻塞的**，他们的返回值 **被 Task 类替代**，Task 中存放着包括返回值在内的全部信息

yield

yield 是 **控制权转移的关键**，consign 在单线程下依旧以并发的形式运行，关键就在于 yield。

consign 允许 yield **任意内容**，但比较特殊的是 function。

如果是 **非 function 内容**，返回的值是本身，同时切换控制权，这没什么好说的。

如果 yield 的是一个 function, **则需要 function 是一个无参函数**

小技巧： 你可以通过很多方法实现将一个有参函数再次无参形式调用：

consign 中的 `repeated_call` ，functools 中的 `partial` 以及 class 中的 `__call__` 方法

满足条件的 function 会在 yield 后由 consign 执行

大部分情况下，你可以直接在 yield 前调用 function，**这其实意味着你 yield 的是调用函数后的返回值**。

小技巧: 特殊的情况一般是留给需要 yield 一个同样被 @coroutine 修饰的协程函数时使用的

他可以和 链式反应 (chain_reaction) 和 等待 (wait) 一同使用达到事半功倍的效果

> **警告:** 注意：yield function 这个条件可能在未来删除，他其实是我 为了兼容旧版框架留下的 (旧版的 consign 并不成熟，所以没有开源，但我存在项目还在使用他)

无需在此留下太多的注意力，好吧我承认这是设计上的失误，曾经的

worker

`AsyncWorker` 继承自 `CoroutineWorker` ，他们有些许不同

如果你使用 `AsyncWorker` ，那么 `loop_work(forever=False)` **在切换很快时是不可信的**

这是由于队列 QSIZE 不确定性以及框架的获取提交导致的

小技巧: 通常在多线程下使用 `loop_work(forever=True)` 或者重复 `while True` 使用 `loop_work(forever=False)`

---

loop_work 会把当前线程 **阻塞**并作为一个 **临时的 worker** 去工作

所以如果 `AsyncWorker` 不使用 `init_thread` 直接使用 `loop_work` 其实和 `CoroutineWorker` 效果是 **相同的**

都是使用 **主进程**去工作，是 **单线程**的，所以任何阻塞的函数都会导致阻塞。

小技巧: 你可以从代码中注意到

`AsyncWorker` 提前于函数执行前使用了 `init_thread(5)` ，这意味着 `AsyncWorker` 创建了 5 个线程

他们会一同去处理所有的协程，所以我无需使用 `loop_work` ，协程函数就能被瓜分执行

如果你不使用 `init_thread` ，那么 `AsyncWorker` 会很失望的

（当然！你可以使用 create_thread 创建单个线程，他其实是 init_thread 的一部分）

`AsyncWorker` 创建的线程默认是 **守护线程**，如果你的程序没有阻塞一路到 exit 的话，可能导致协程任务异常被终止

所以在 **非 cmd** 下，最好使用 `while True` 或者 `loop_work(forever=True)` 去阻塞，或者设置参数选择生成非守护线程。

生成线程的名字如果没有传参，那么默认是 uuid 的

所以会出现 *consign_184134fe-2fbe-4161-8458-d186aba693e8* 这种名字，具体需要去看文档。

sleep

sleep 是有点不同的，**asleep 是 consign 实现的**，他其实 **也是被修饰的协程函数**。

他其实是 **在不停的切换控制权**通过 consign 来实现并发的，所以他是 **非阻塞**的。

这也是 consign 能在单线程下并发的原因：控制权切换

同时也是大部分协程的缩影。而 `time.sleep` 是 **阻塞**的。

但为什么后面使用了 `time.sleep` 呢？因为我们通过这个 sleep 来模拟一个阻塞的，非协程的 io 函数。

你可以看到，`AsyncWorker` 通过多线程完成了这一切, 使得 `time.sleep` 的效果和 `asleep` 相同。

> **警告：** 如果使用 `CoroutineWorker` 或者 `AsyncWorker` 创建的线程都被阻塞了，那么依旧会产生阻塞的效果。
>
> 当然，你可以创建 **充足的线程**以应对这个问题，同时 **尽可能的切分函数控制权**也可以充分利用 consign

task

Task 是什么？他是 **每次协程函数运行时**产生的 **类**，同时也是协程函数的 **替代返回值**。

Task 里存放则很多信息，包括协程此次任务的完成情况，函数的返回值。

> **警告：** 他很重要，是协程的运行的基础，部分信息被用在处理过程中，所以如果你要 **修改**，**最好知道他是要做什么的**

Task 被打印成一个字典，但 Task 其实并非是字典而是类，所以如果需要获取其值，通过 **类成员**而非 **下标**或者 **key** 去访问

当然 Task 也提供以字典方式访问的功能，但并不划算

一个看不见的恶魔

哈哈！开个玩笑，但看不见是真的，他很重要，他就是 `WorkArea` 。

WorkArea 详情可以去看文档，简要的来说，你看起来没有设置过 `WorkArea` ，其实使用的就是默认的 `WorkArea` ，他的变量名是 `DEFAULT_WORK_AREA` 。

备注： 一旦你导入了 consign，`DEFAULT_WORK_AREA` 就会被创建并指定一个 `WorkArea`

所有没有被 上下文管理器 (with) 包裹的协程其实都是运行在名为 DEFAULT_WORK_AREA 的 WorkArea 上。

没有指定名字的 Worker 其实是为默认也就是 DEFAULT_WORK_AREA 的 WorkArea 工作的

你可以通过直接访问 `DEFAULT_WORK_AREA` 这个变量，但一般不建议

`DEFAULT_WORK_AREA` 同样被创建在 `builtins` 中，所以他无需从 consign 包名下使用，即使例如 PyCharm 提示变量不存在

所有的 `Worker` 同理，他们需要传入 `WorkArea` 名或者类，这表明这个 Worker 是为哪个 WorkArea 在服务。

不同的 `WorkArea` 会相互隔离，但注意的是，**同名的 WorkArea 视作同一个 WorkArea** ，同名 WorkArea 是**单例**的。

他其实不可怕，他能方便你去管理 Worker 和分配协程函数，他的使用方法在文档中，感兴趣就去看看吧

consign 更适合和一些特别的情况发生化学反应，我们来看看更多例子，并开始慢慢了解

CHAPTER 6

更多例子

让我们来看看比较常见的写法

```
from consign import coroutine, wait, AsyncWorker
import time

DEBUG = False

@coroutine(DEBUG)
def my_io_read(path: str):
    print(f"Let me start reading {path}", threading.currentThread())
    yield time.sleep(3)
    print(f"reading {path} over", threading.currentThread())
    return f"<{path} read data>"

@coroutine(DEBUG)
def preprocess(path: str):
    print(f"preprocess {path} somethings", threading.currentThread())
    yield time.sleep(1.5)
    print(f"preprocess {path} is over", threading.currentThread())
    return f"<preprocess {path}>"

@coroutine(DEBUG)
def my_test(path: str):
    print(f"my_test start and path is {path}", threading.currentThread())
```

(续下页)

(接上页)

```
    data_task = yield my_io_read(path)
    some_preprocess_task = yield preprocess(path)
    data, some_preprocess = wait(data_task), wait(some_preprocess_task)
    print(f"my_test over {some_preprocess} and data is {data}", threading.
↪currentThread())


aw = AsyncWorker()
using_thread_num = 5
aw.init_thread(using_thread_num, name_iter=(f"DEFAULT_WORK_AREA_{i}" for i in␣
↪range(using_thread_num)))

my_test("test/task1.jpg")
my_test("test/task2.jpg")

aw.loop_work(forever=True)
```

输出的结果是:

```
my_test start and path is test/task1.jpg <_MainThread(MainThread, started 29856)>
my_test start and path is test/task2.jpg <DummyProcess(consign_DEFAULT_WORK_AREA_1,␣
↪started daemon 35364)>
Let me start reading test/task1.jpg <_MainThread(MainThread, started 29856)>
Let me start reading test/task2.jpg <DummyProcess(consign_DEFAULT_WORK_AREA_1,␣
↪started daemon 35364)>
preprocess test/task1.jpg somethings <DummyProcess(consign_DEFAULT_WORK_AREA_3,␣
↪started daemon 36668)>
preprocess test/task2.jpg somethings <DummyProcess(consign_DEFAULT_WORK_AREA_4,␣
↪started daemon 37204)>
preprocess test/task1.jpg is over <DummyProcess(consign_DEFAULT_WORK_AREA_3, started␣
↪daemon 36668)>
preprocess test/task2.jpg is over <DummyProcess(consign_DEFAULT_WORK_AREA_4, started␣
↪daemon 37204)>
reading test/task1.jpg over <DummyProcess(consign_DEFAULT_WORK_AREA_0, started daemon␣
↪37332)>
reading test/task2.jpg over <_MainThread(MainThread, started 29856)>
my_test over <preprocess test/task2.jpg> and data is <test/task2.jpg read data>
↪<DummyProcess(consign_DEFAULT_WORK_AREA_0, started daemon 37332)>
my_test over <preprocess test/task1.jpg> and data is <test/task1.jpg read data>
↪<DummyProcess(consign_DEFAULT_WORK_AREA_4, started daemon 37204)>
```

值得注意的点:

DEBUG

DEBUG 是全局标志，他会在程序被创建时决定了 `@coroutine` 是否以协程的方式运行

如果 DEBUG=True，那么所有被传递参数的 `@coroutine` 修饰器，都会退化成普通函数的线性运行方式而无需改动代码

参见:

由于无需改动代码，事实上函数类型没变，只是 `@coroutine` 会返回一个替代的函数

如果原修饰函数是普通函数，那么替代函数直接返回结果

如果原修饰函数是生成器函数，这个替代函数内部会阻塞并不断的 next，直到生成器函数运行完毕后返回结果

DEBUG 仅在程序初次加载时起作用，后续修改变量无效

wait

wait 的作用是等待一个 Task 完成，期间他会将自身代入成为一个 `Worker` 进行 `loop_work`

wait 很特殊，他可以理解成一个特殊的 `loop_work` ，为什么特殊主要在于他的 `WorkArea` 继承机制

他内部组合了 `CoroutineWorker` , 简单说几个特性:

1. wait 一个 Task 时，他会继承 Task 中的 `WorkArea` 并生成对应的 `CoroutineWorker` 进行 `loop_work`
2. wait 不对 约定的结束信号做处理，碰到结束信号时他会重新提交回原 `WorkArea`
3. wait 轮询的时间是不精确的，受到当前任务量和阻塞时间的影响

通常 wait 很好理解，但是当使用 `WorkArea` 时，由于继承机制，往往你需要明确你的 wait 工作在哪个区域下？为哪个区域而工作？

当然，这也是 wait 的特性之一，用的好的话是可以产生奇妙的化学反应，比如实现协程分配者

更多例子还在完善中…

Python 模块索引

C

索引

符号

A

C

E

I

L

N

P

Q

R

S

T

W

◆

模块