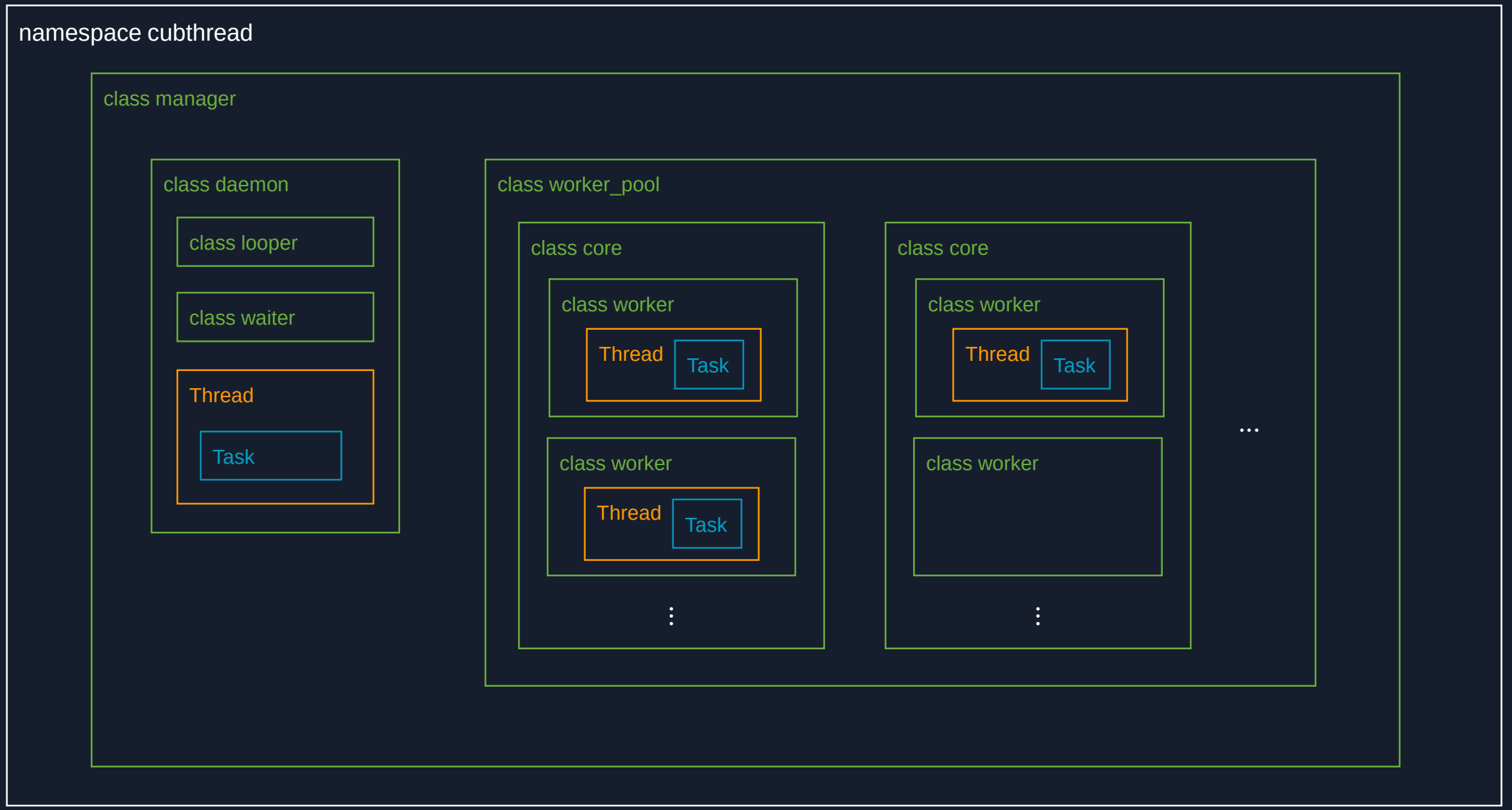
Thread

cubthread

yahong


namespace cubthread
class manager
class daemon
class looper
class waiter
Thread
Task
class worker_pool
class core
class worker
Thread
Task
class worker
Thread
Task
class core
class worker
Thread
Task
class worker
...

Task

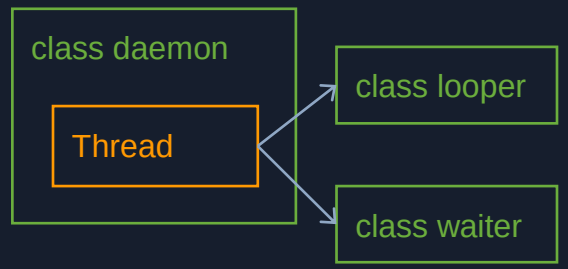
Daemon

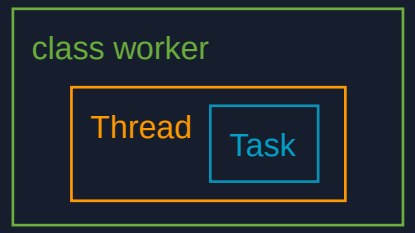
Lopper

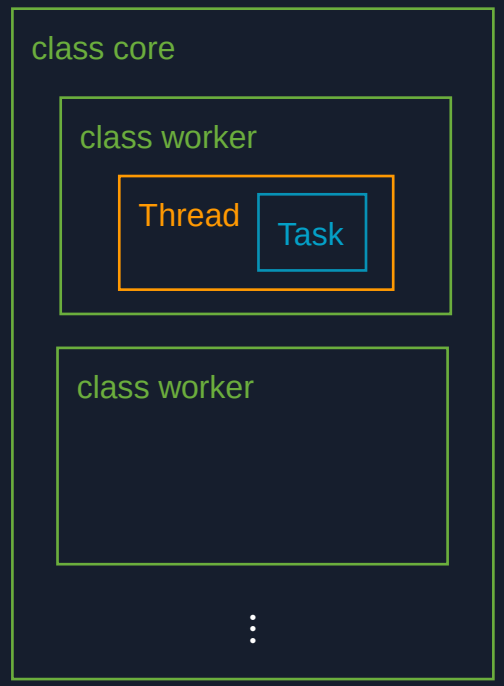
Waiter

Worker

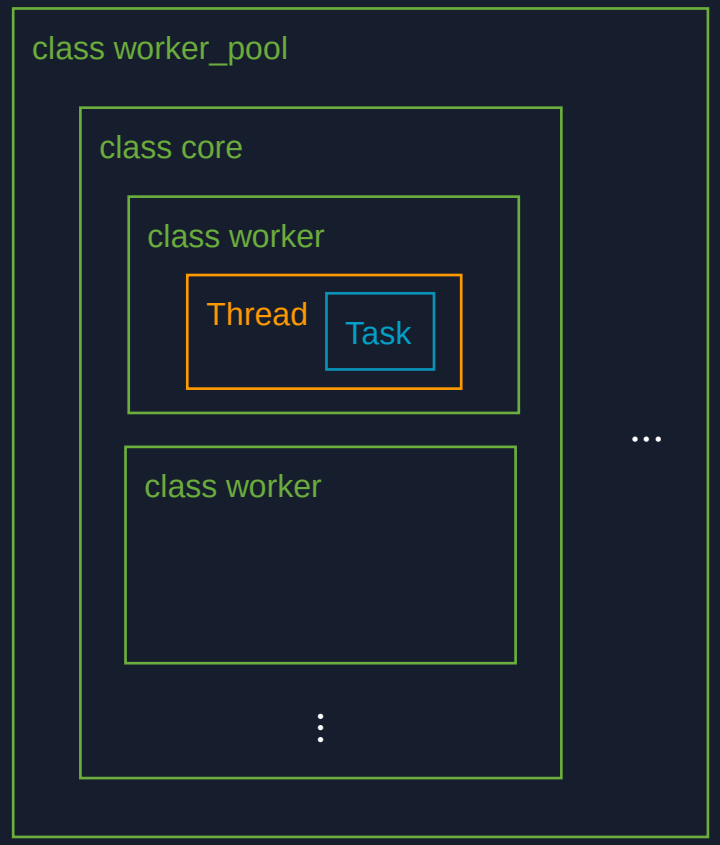
Core



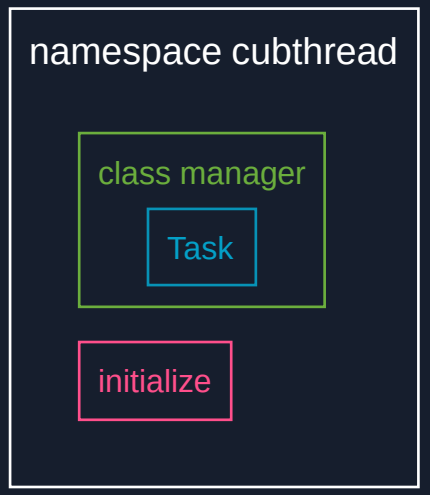




Worker Pool

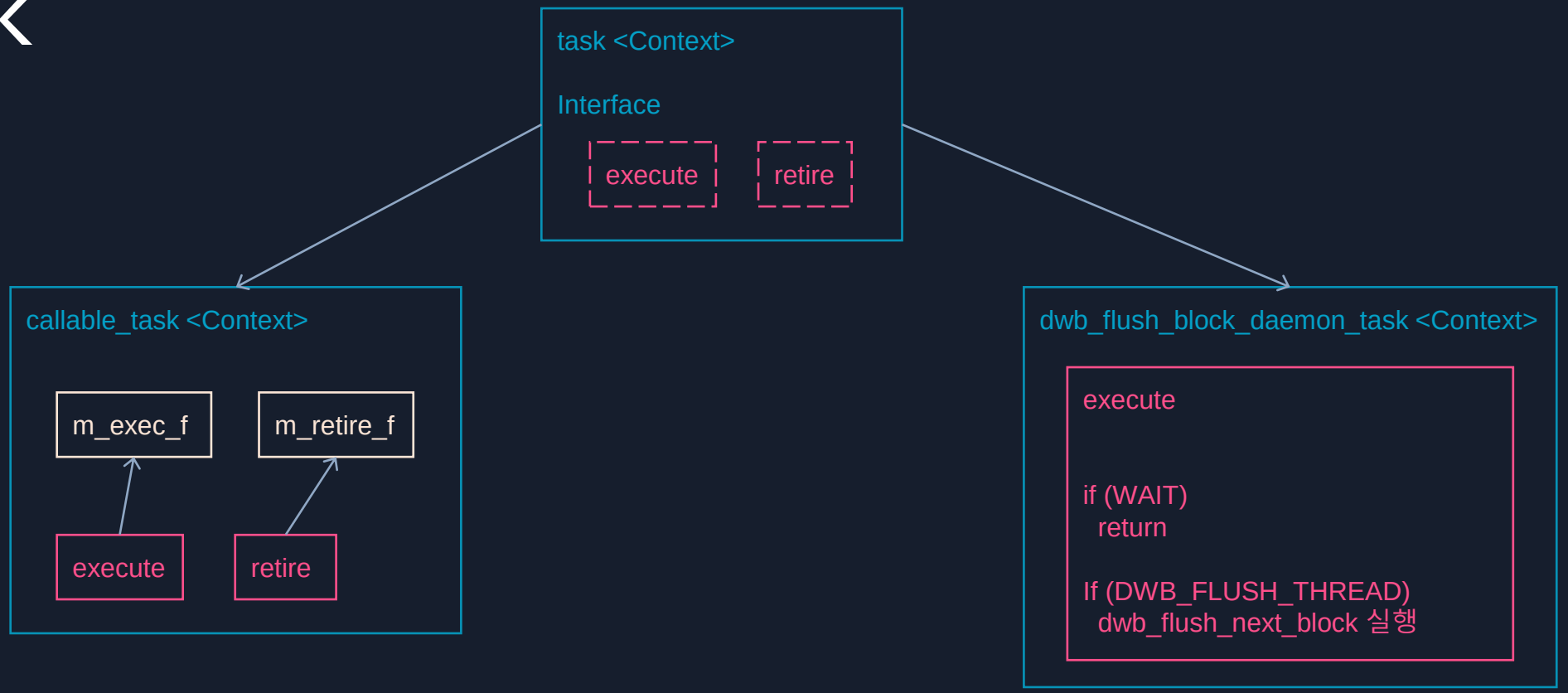


Manager



Task

Task



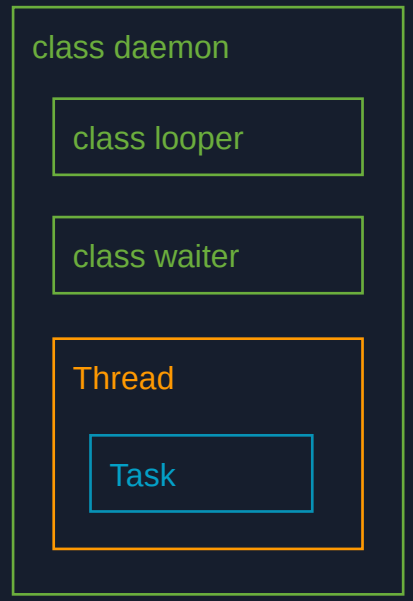
Task는 Thread에서 이루어지는 작업의 가장 작은 단위입니다.

Context를 가지는 Task와 가지지 않는 Task 모두 존재하며, Context는 공유 캐시로 사용됩니다.

Task

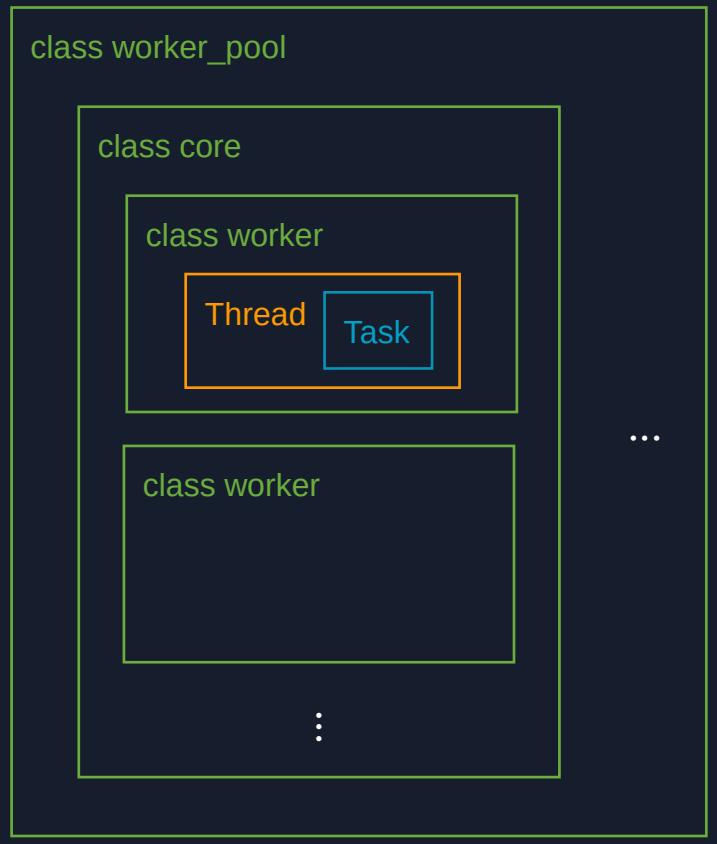
Task에 적합한 처리 방식에 따라 크게 두 가지 방법으로 관리
실행 방식으로는 세 가지로 분리

Task 주기적 실행

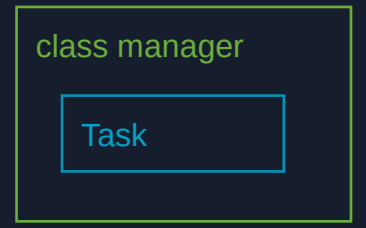


Task 일회성 실행

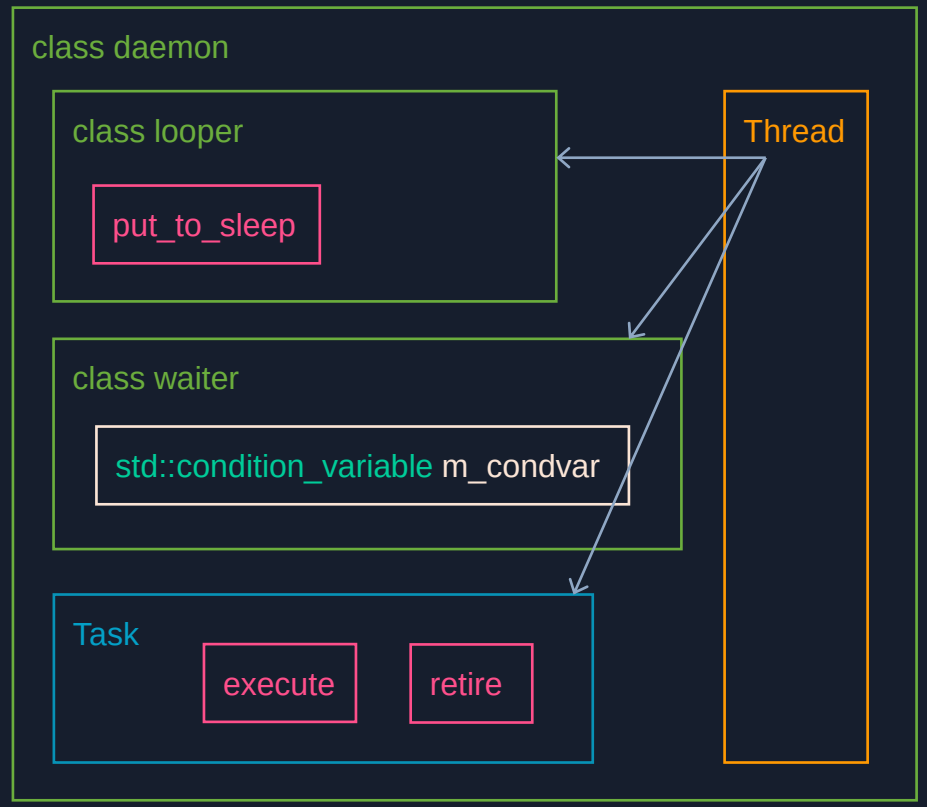
Worker Pool

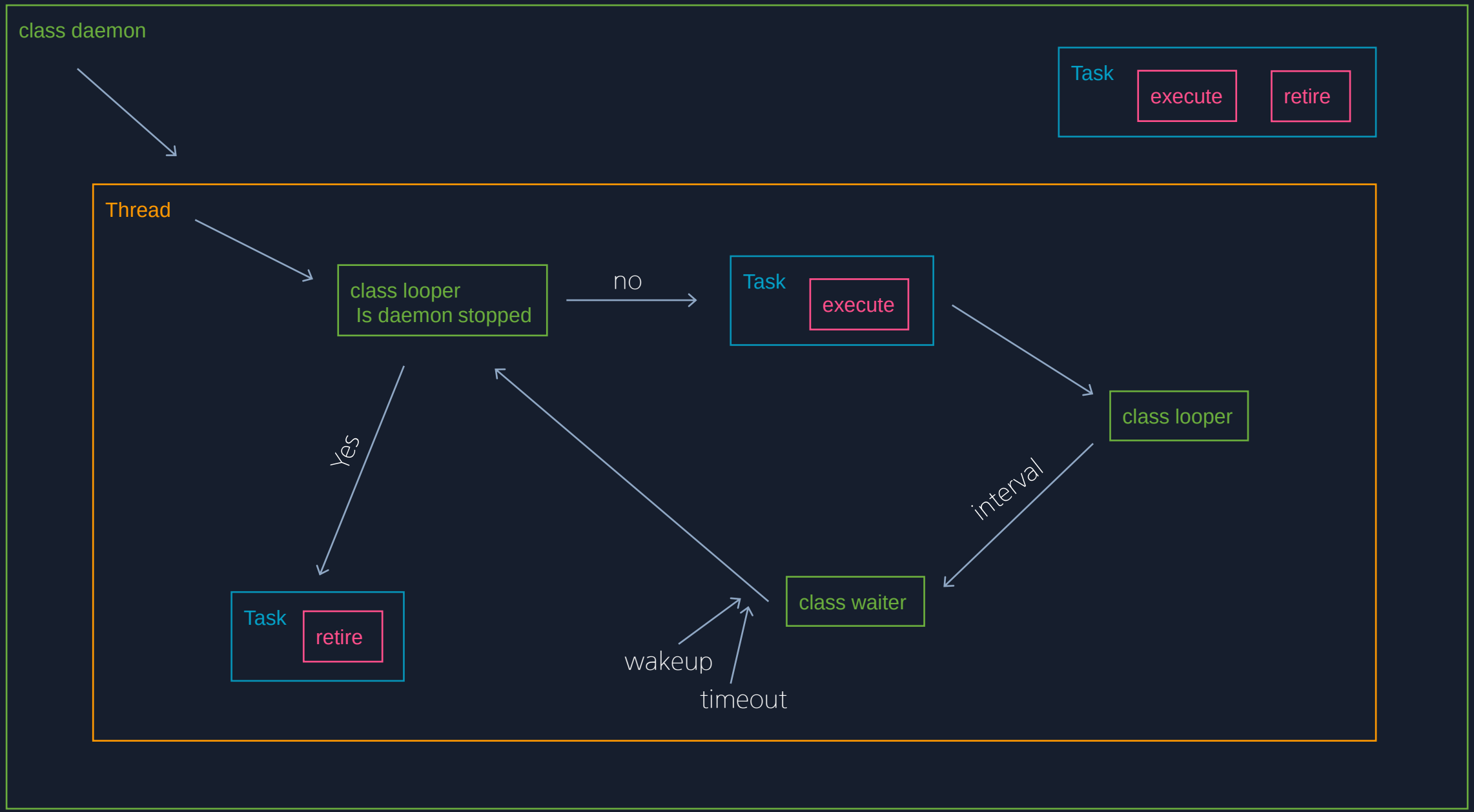


Manager



Daemon


class daemon
class looper
put_to_sleep
Thread
class waiter
std::condition_variable m_condvar
Task
execute
retire


class daemon
Task
execute
retire
Thread
class looper
Is daemon stopped
no
Task
execute
class looper
Yes
interval
Task
retire
class waiter
wakeup
timeout

Looper

Daemon

Looper

class looper

- m_stop
- put_to_sleep
- type
  - INF_WAITS
  - FIXED_WAITS
  - INCREASING_WAITS
  - CUSTOM_WAITS

Looper의 type

INF_WAITS 무기한 대기

FIXED_WAITS 고정된 기간만큼 대기

INCREASING_WAITS 최대 3개의 Interval을 받아서 순회 후 무기한 대기

Interval: 1 | 10 | 100

이렇게 Interval 배열을 넘겨주면 Interval이 차례대로

1ms, 10ms, 100ms, 무기한 대기

가 됩니다

CUSTOM_WAITS 위에 3 종류는 생성자로 Interval 혹은 Interval 배열을 받지만 이것은 Interval을 계산할 함수를 받습니다.

```
std::function<void (bool &, delta_time &)>
```

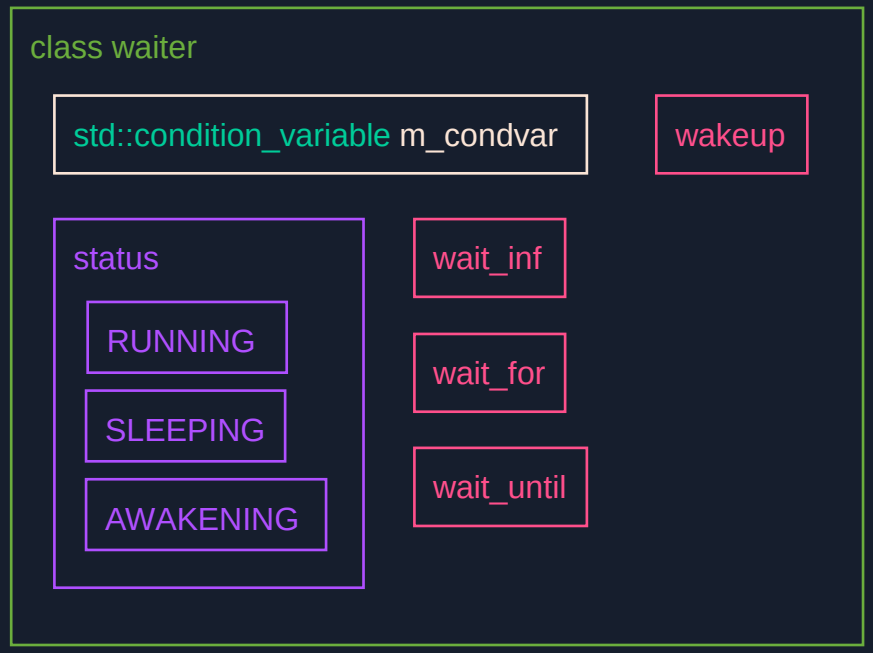
Daemon의 상태 관리는 Looper에 의존하고 있습니다. Daemon의 정지 여부도 Looper가 관리합니다.

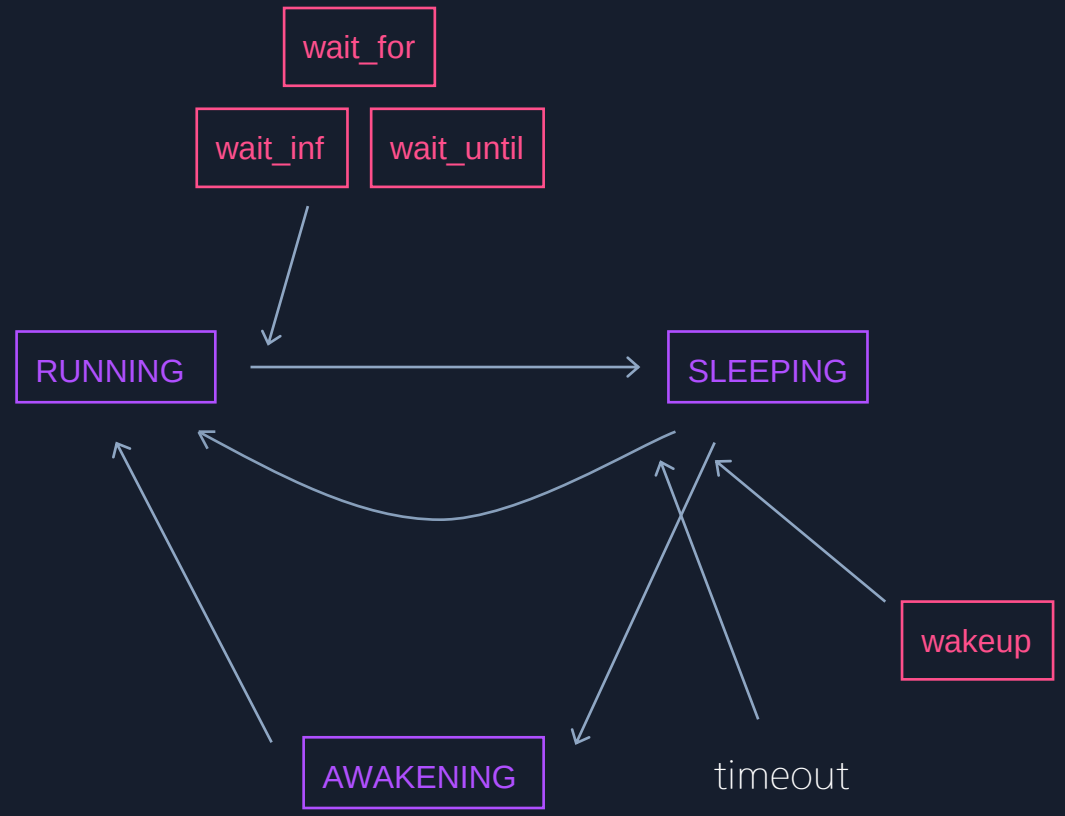
Looper는 Daemon의 다음 Task까지의 대기 시간을 계산하고, Daemon의 Waiter로 넘겨줘 대기를 처리합니다.

Waiter

Daemon

Waiter





Waiter는 조건 변수 m_condvar를 사용하여 해당 Context 혹은 Thread를 blocking할 수 있습니다.

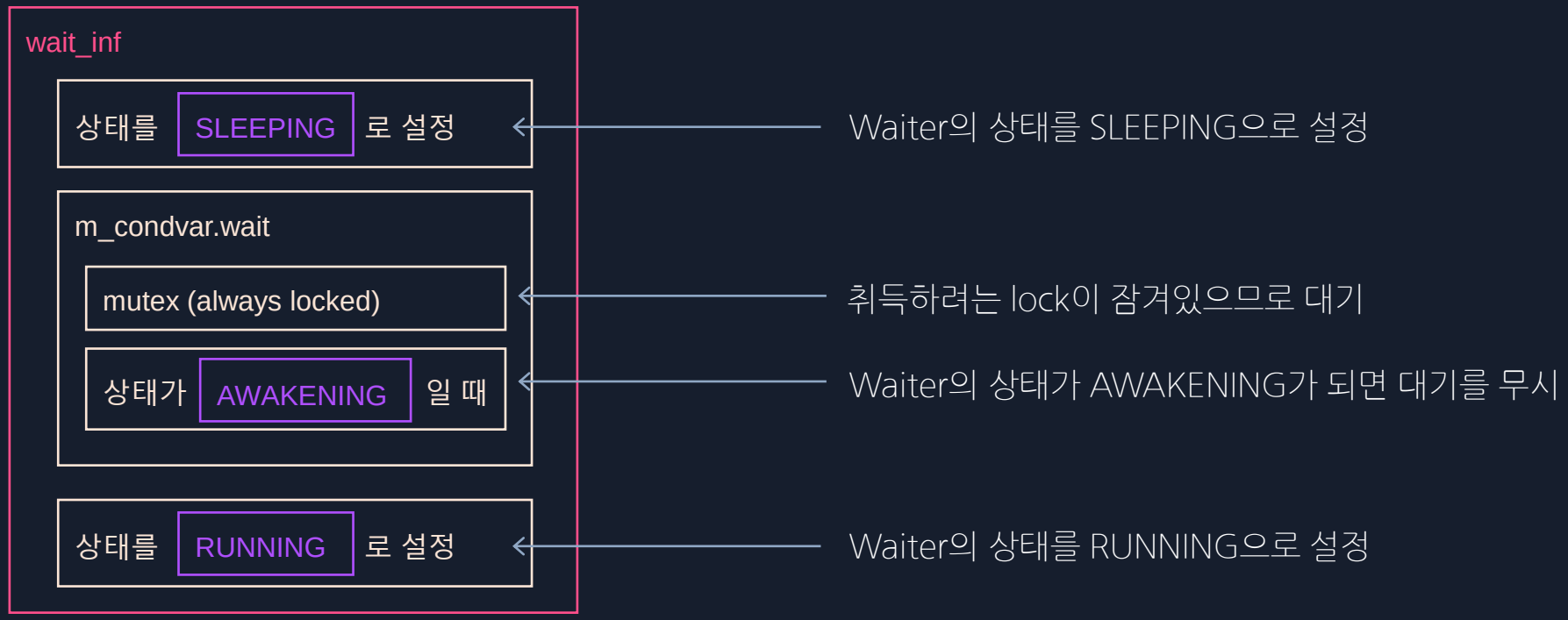
Context에서 대기가 필요한 부분을 효과적으로 처리합니다.

wakeup과 timeout의 차이점은 상태 변화 과정에 있습니다.

timeout은 SLEEPING에서 바로 RUNNING으로 변하는데 반해,

wakeup은 AWAKENING을 거쳐서 RUNNING으로 변합니다.

Waiter wait_inf



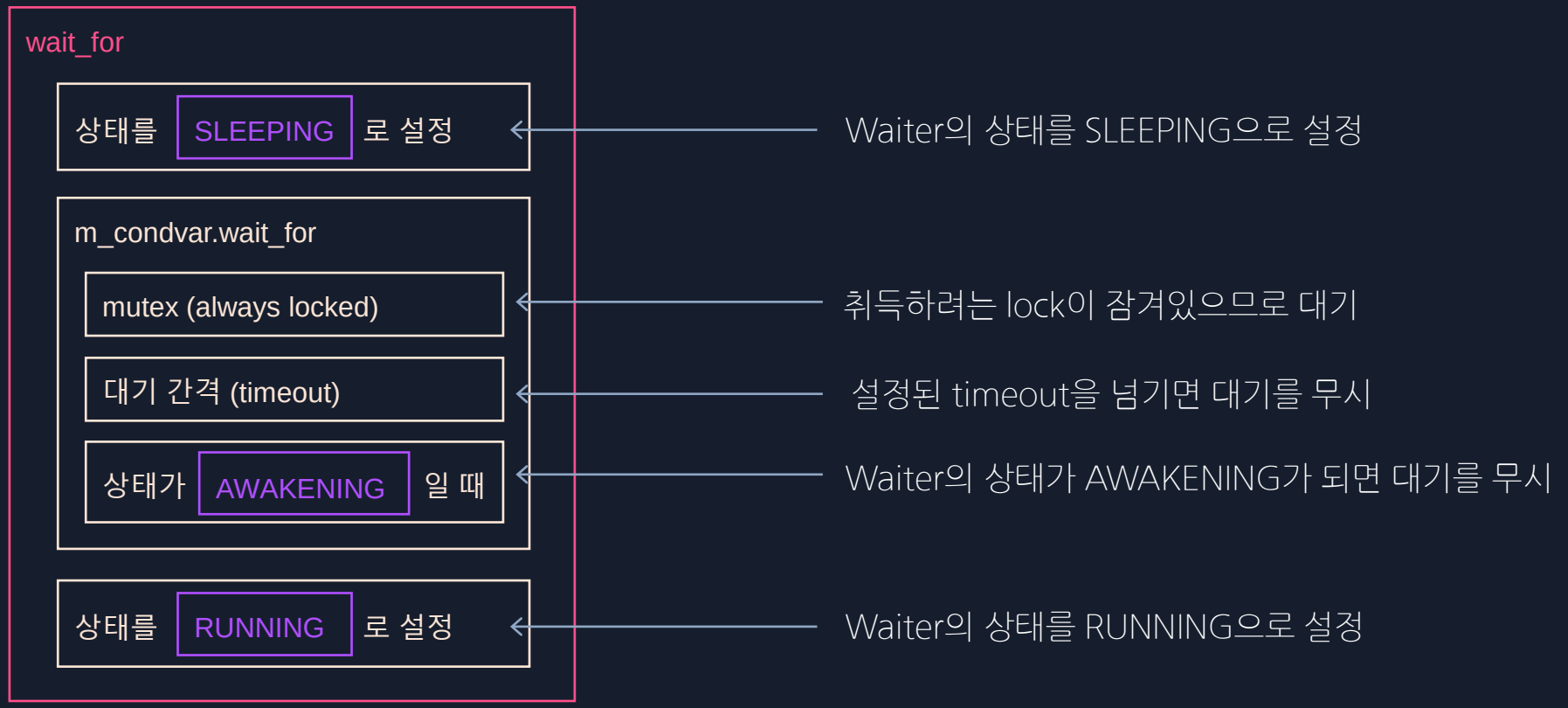
wait_inf가 실행되면 조건 변수 m_condvar.wait로 lock을 넘겨줘 코드가 실행되고 있는 주체의 Context를 blocking한다.

이때 조건으로 “상태가 AWAKENING일 때” 가 존재하는데, waiter의 내부상태가 wakeup의 호출로 AWAKENING이 되면 Context를 깨웠을 때, 다시 blocking이 일어나지 않는다.

따라서 wait_inf로 대기에 들어가게 되면, wakeup이외에는 해당 Context가 깨어나지 않는다.

때문에 wait_inf는 반드시 SLEEPING -> AWAKENING -> RUNNING 의 과정으로 깨어난다.

Waiter wait_for

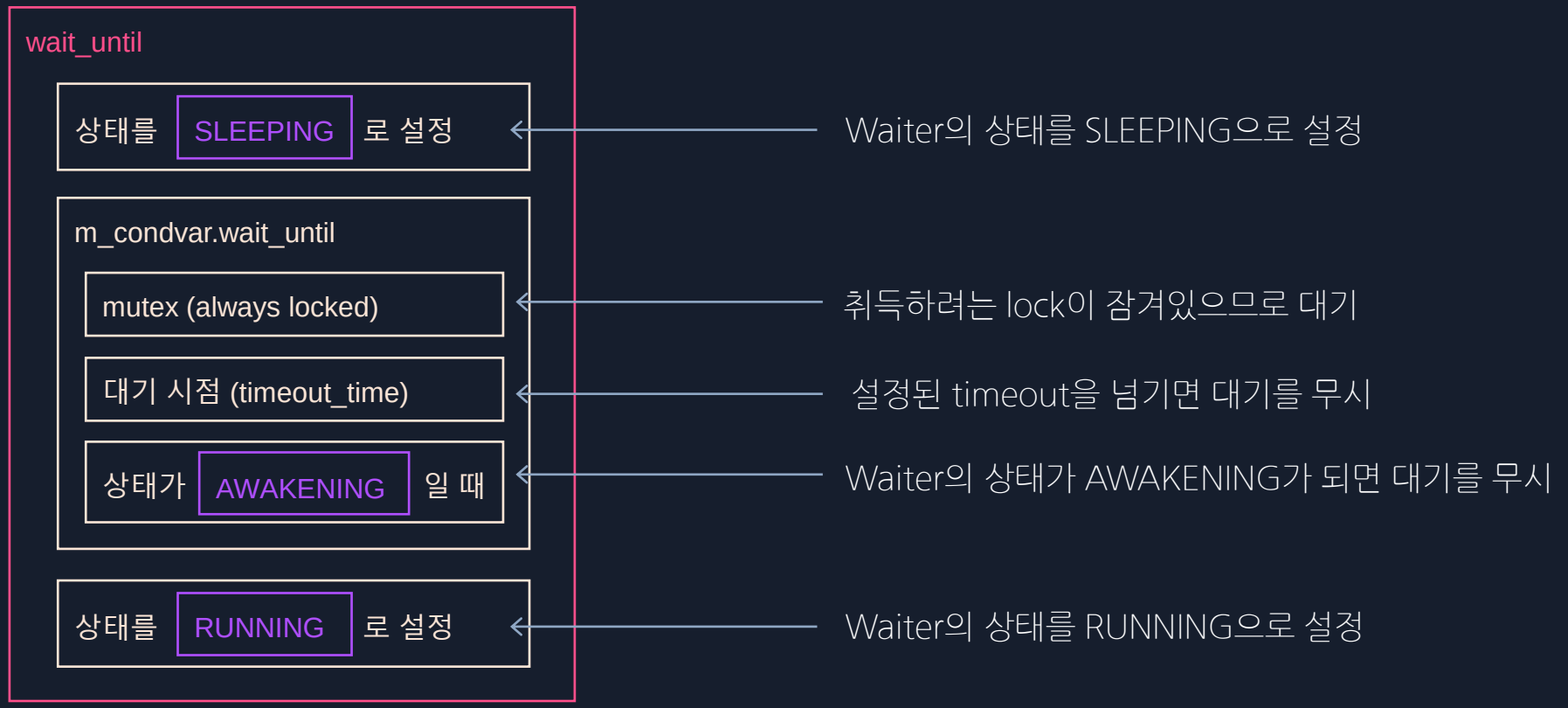


wait_inf 와 동일하지만, m_condvar.wait_for을 사용하여 지정된 기간 동안 대기하도록 timeout을 설정할 수 있다.

timeout에 설정된 시간이 지나면 자동으로 대기를 끝내고 상태를 RUNNING으로 설정한다.

따라서 timeout으로 대기를 마친 경우에는 AWAKENING을 거치지 않고 SLEEPING -> RUNNGIN 로 깨어난다.

Waiter wait_until

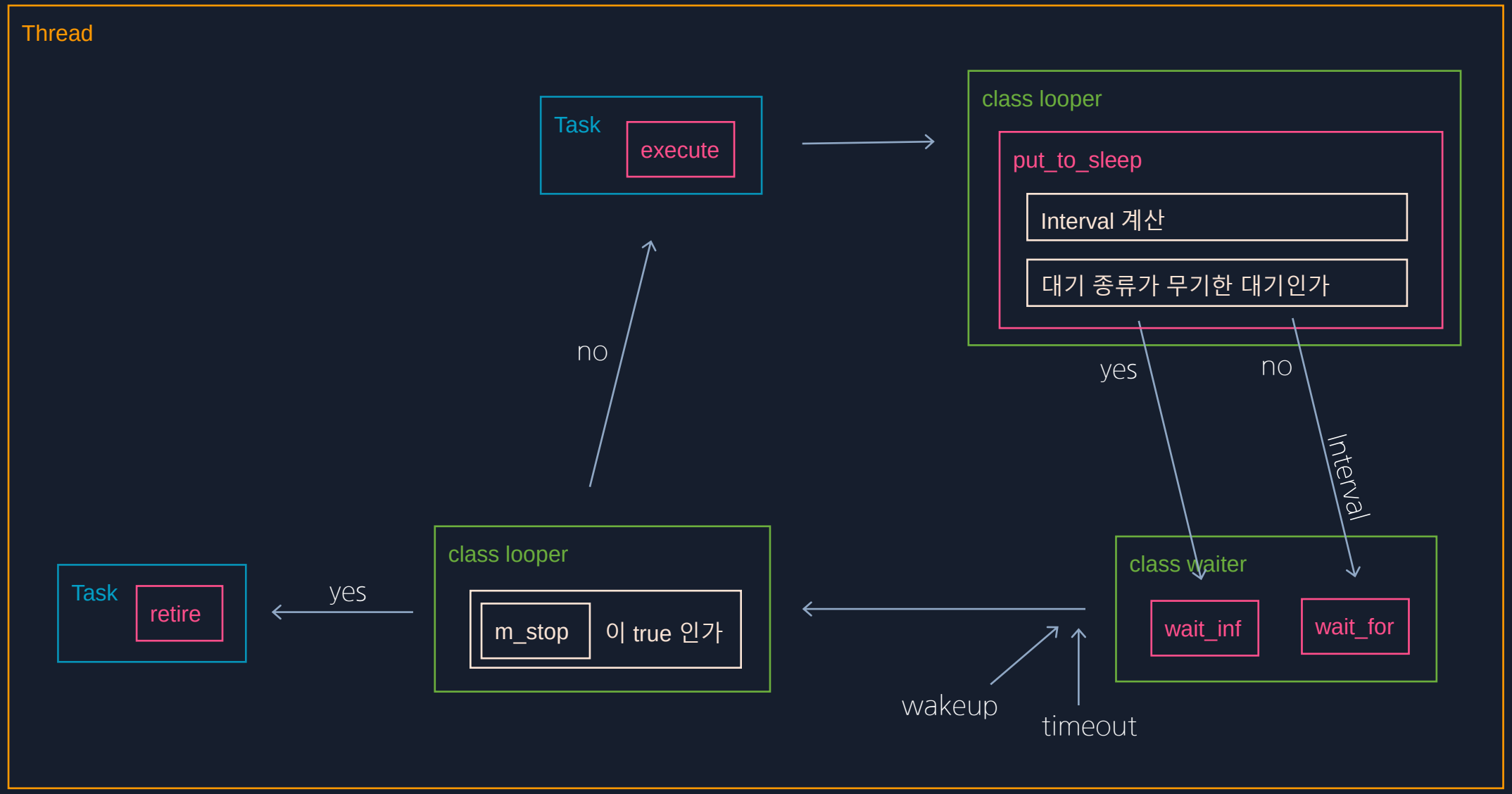


wait_for 과 동일하지만 timeout 설정에서 차이를 가집니다.
어느 시간 동안 대기하는 wait_for 과 다르게 wait_until 은 어느 시점까지 대기합니다.

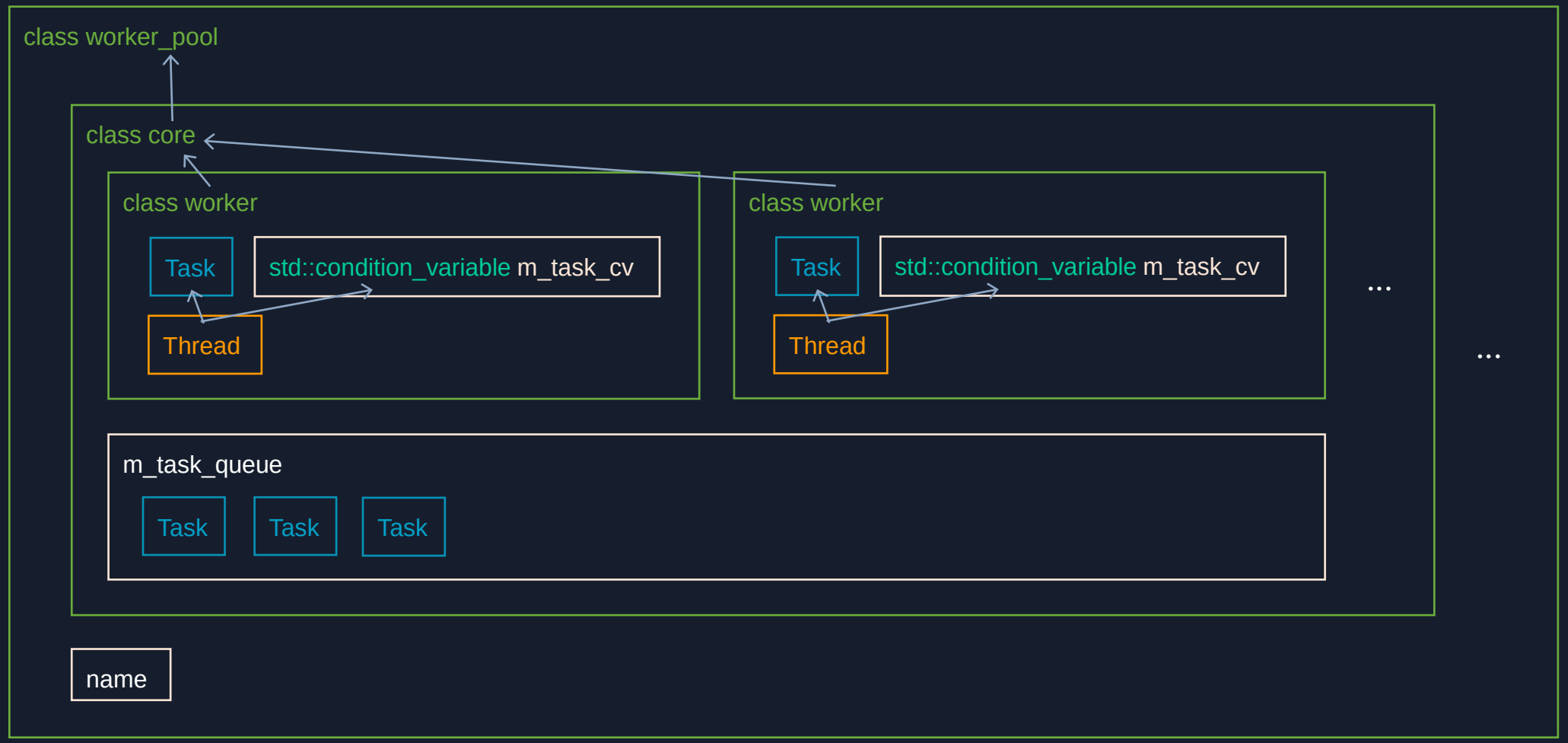
wait_until의 경우 사용되고 있지는 않은 것 같습니다.

Looper → Waiter

Daemon


Thread
Task
execute
class looper
put_to_sleep
Interval 계산
대기 종류가 무기한 대기인가
yes
no
Interval
class waiter
wait_inf
wait_for
wakeup
timeout
class looper
m_stop
이 true 인가
yes
Task
retire
no

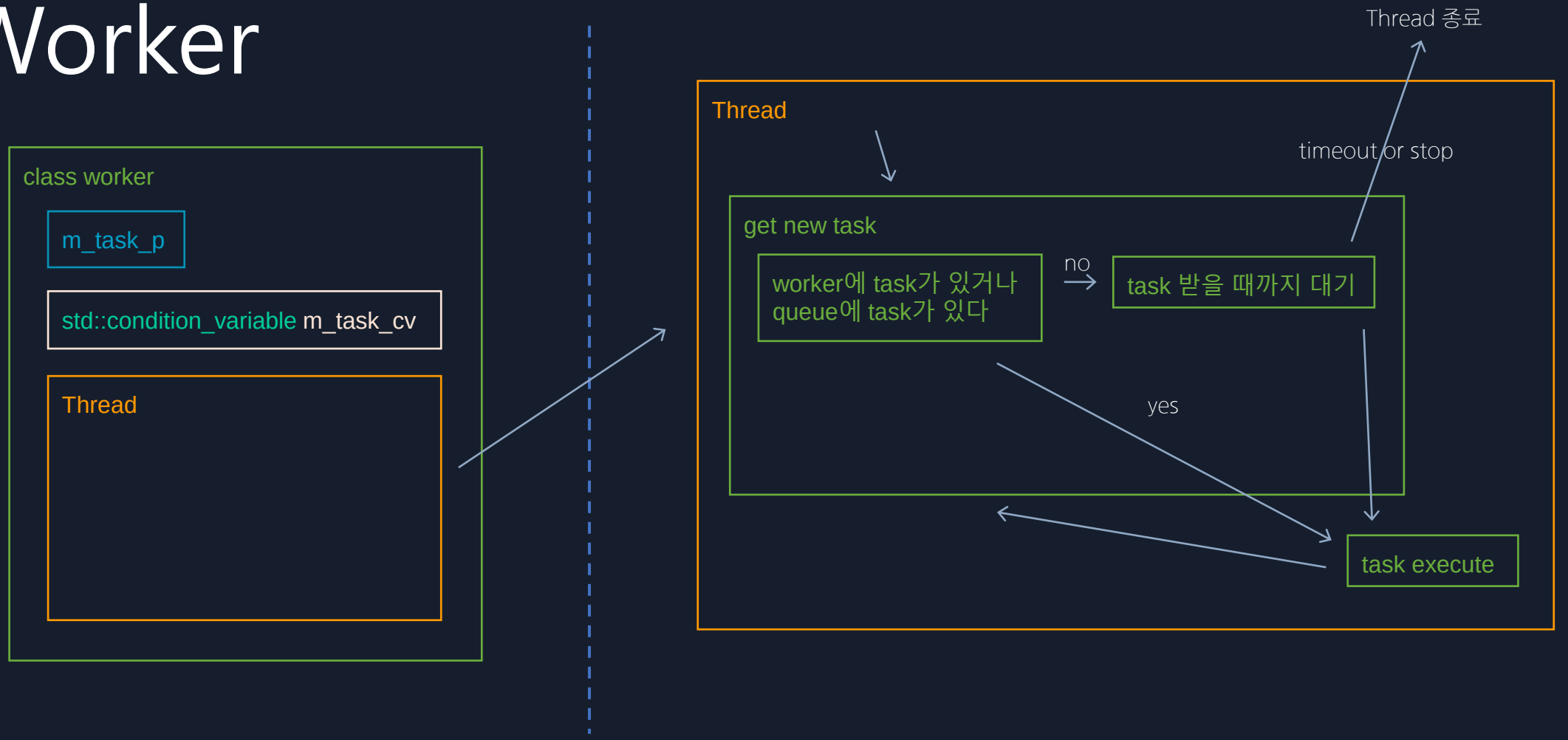
Worker Pool


class worker_pool
class core
class worker
Task
std::condition_variable m_task_cv
Thread
class worker
Task
std::condition_variable m_task_cv
Thread
...
...
m_task_queue
Task
Task
Task
name

Worker

Worker Pool

Worker

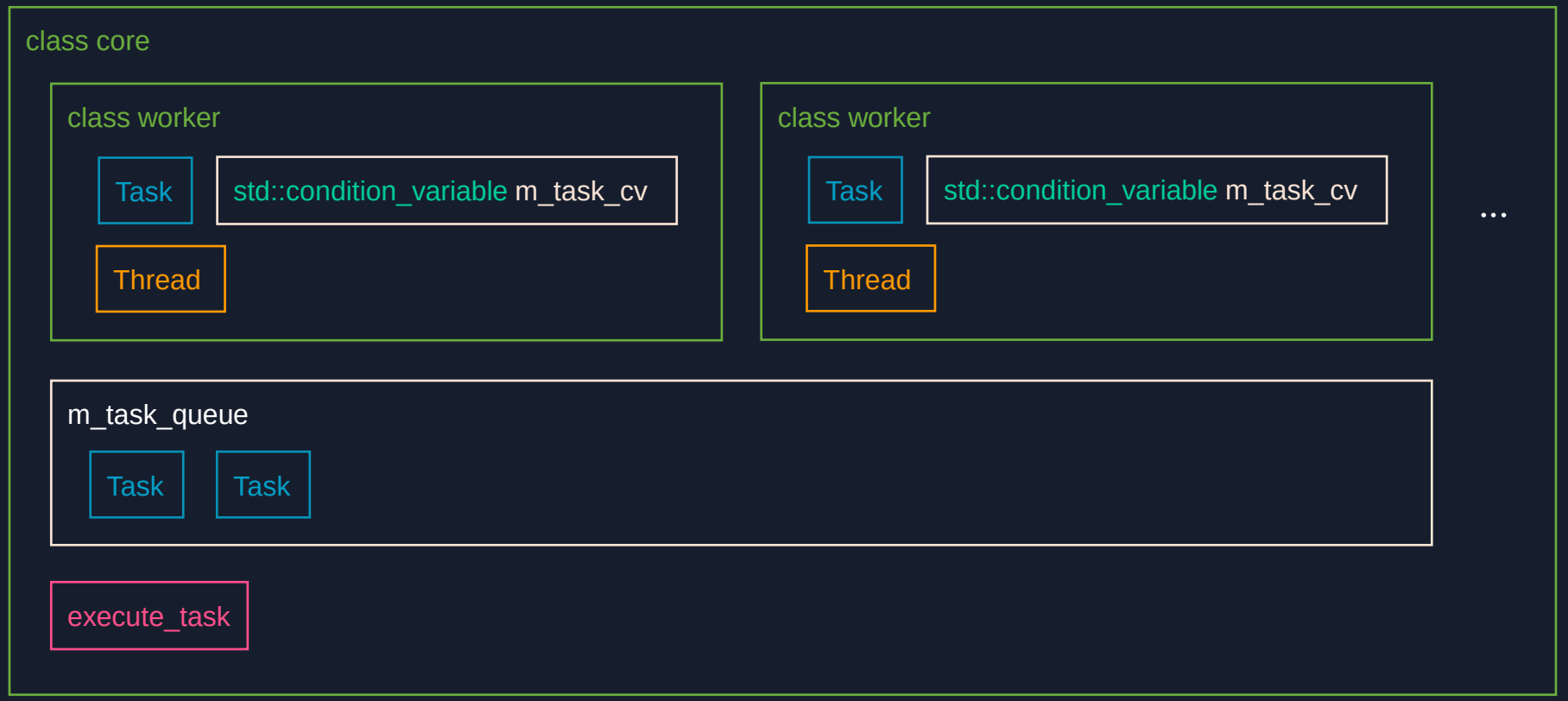


worker는 worker pool 에서 작업을 담당하는 실질적인 부분입니다.
내부에 thread를 가지고 있고, 해당 thread로 task를 받아 처리합니다.
thread가 종료된 worker는 다시 작업을 할당받을 때까지 쉬게 됩니다.

Core

Worker Pool

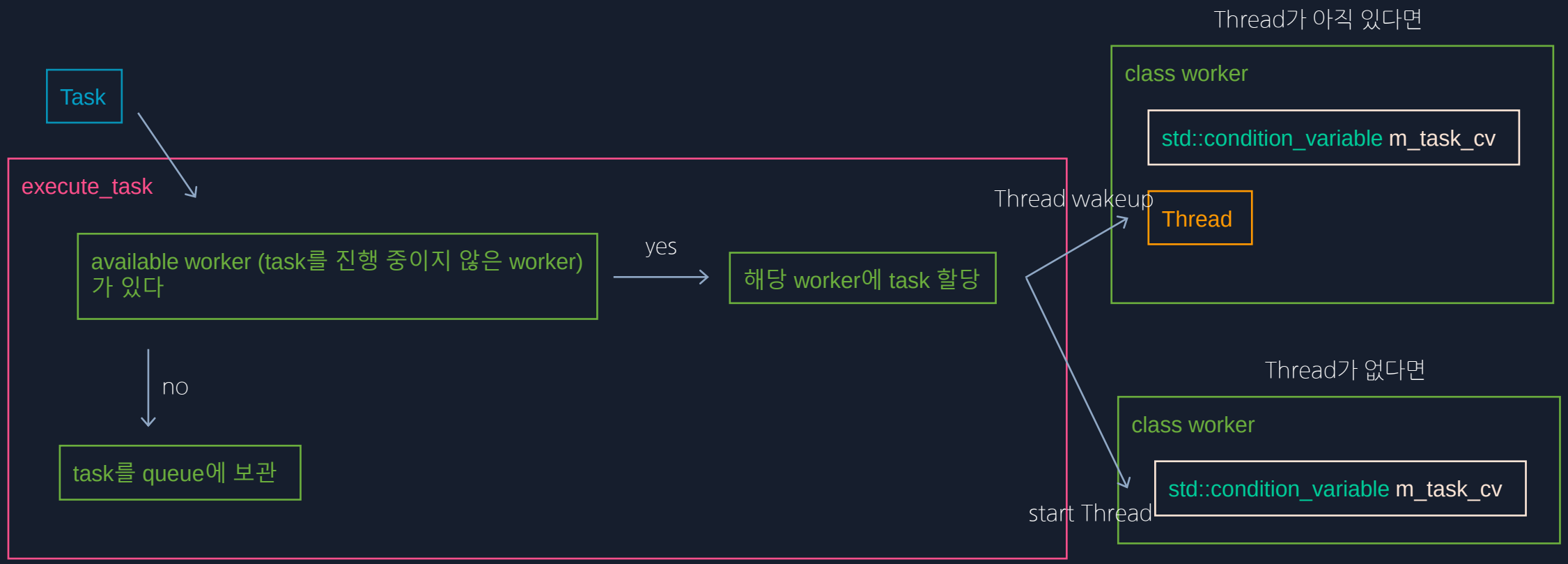
Core



execute_task 를 통해서 worker pool에서 core로 task를 할당합니다.

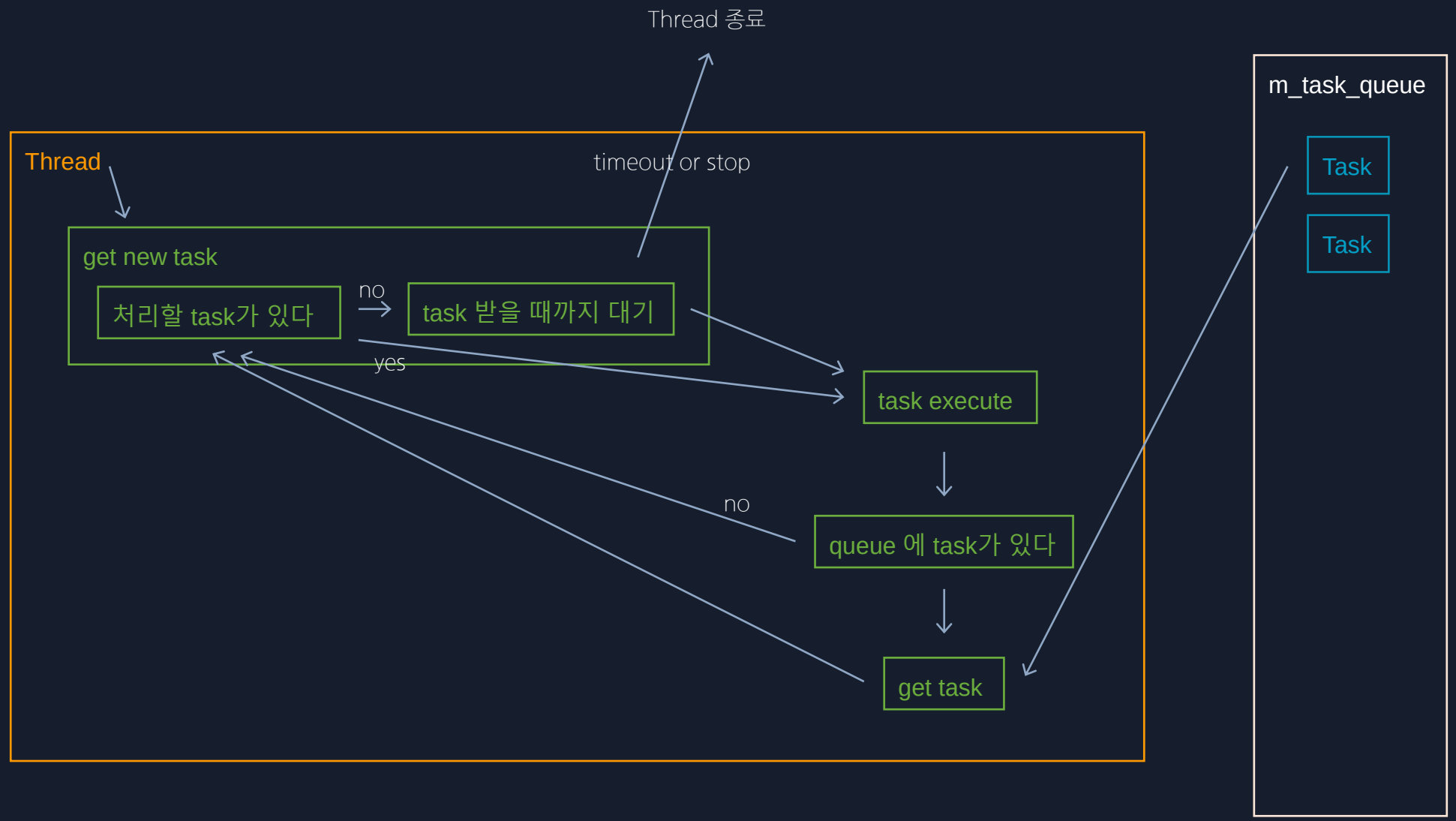
task를 받은 core는 내부에서 task를 관리합니다.

빈 worker가 없다면 task를 queue에 가지고 있거나, 빈 worker가 있다면 task를 넘겨줍니다.



execute_task 를 통해서 위와 같은 흐름으로 task를 관리합니다.

queue에 보관된 task는 이후 core에서 처리하지 않고, 작업이 끝난 thread가 core의 queue를 열람해 처리합니다.

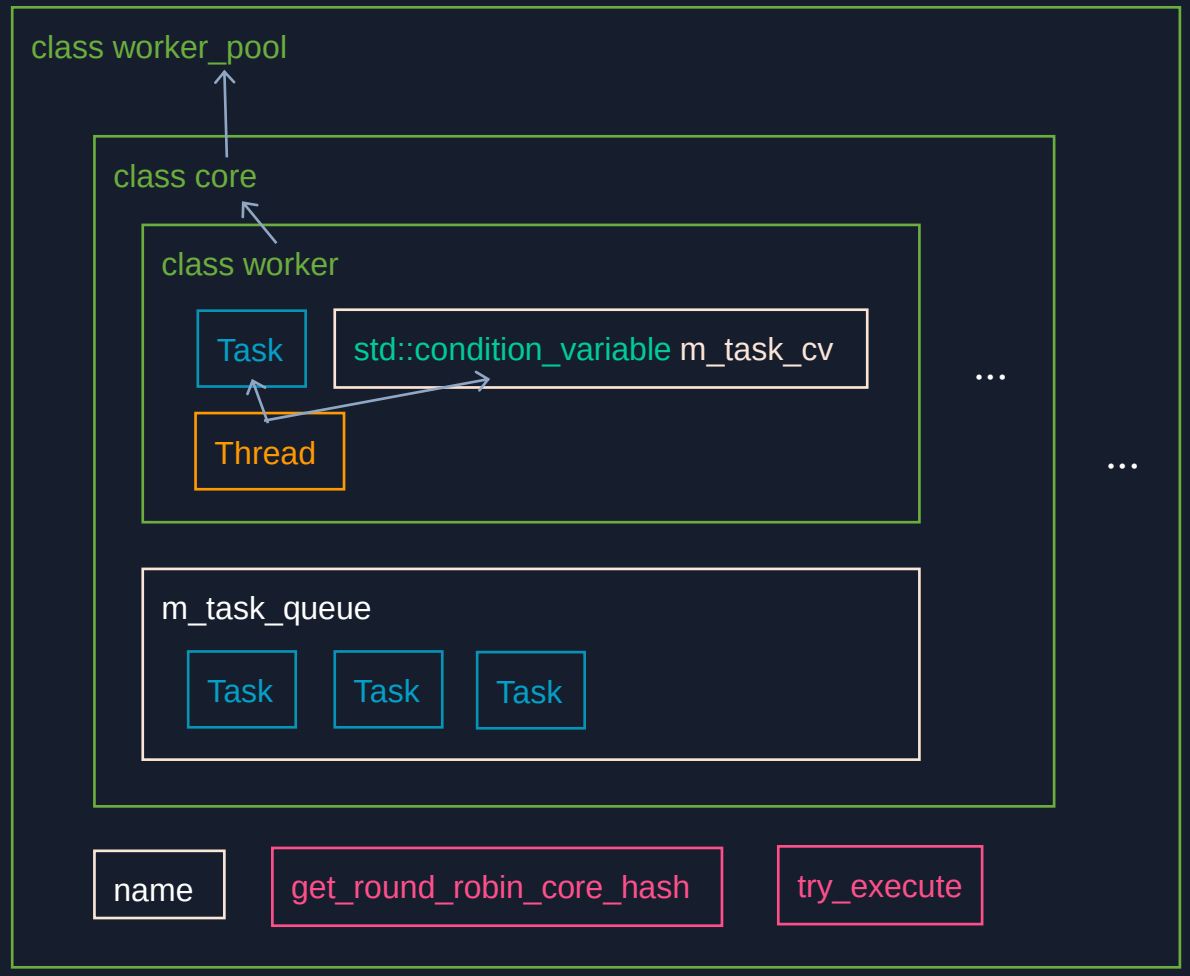


worker가 queue에서 task를 가져오는 과정입니다

Worker Pool

Worker Pool

Worker Pool



Worker Pool은 core에 task를 할당할 때 라운드 로빈 방식을 사용합니다.

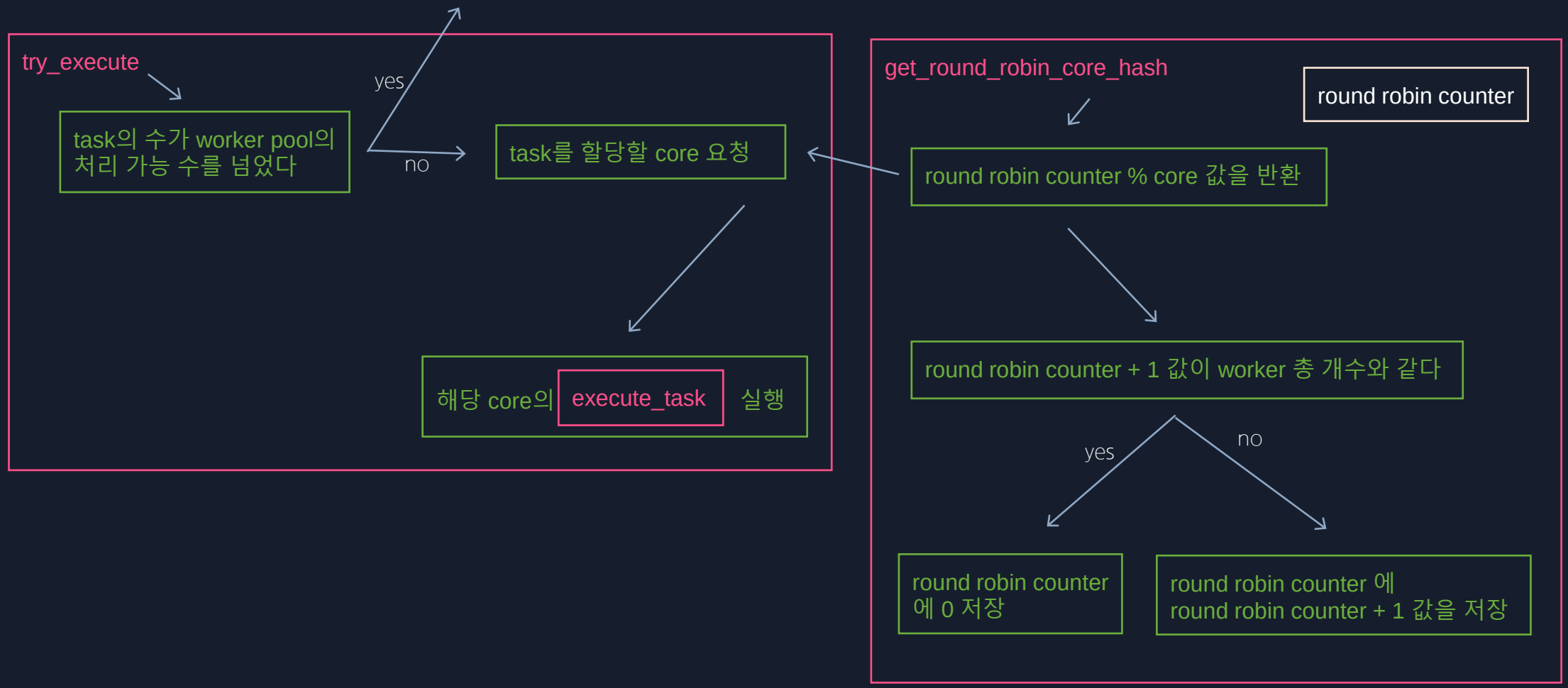
라운드 로빈이란,

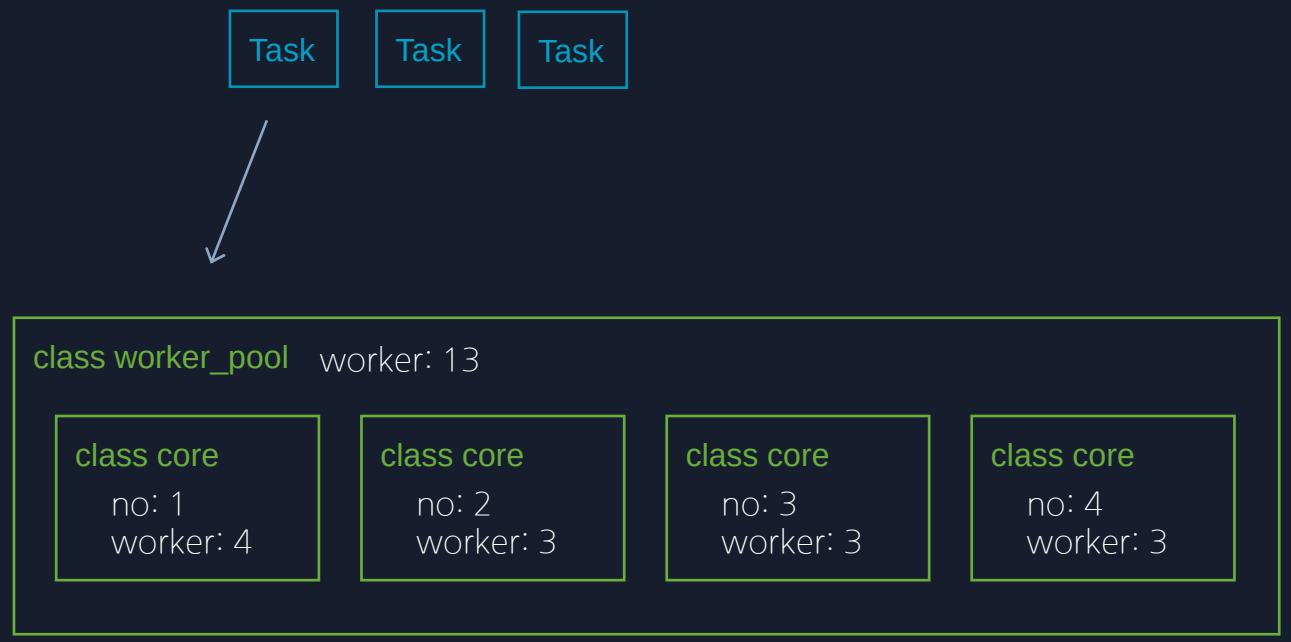
자원에 대해 우선순위를 두지 않고 순서대로 할당하는 방식을 말합니다.

때문에 각 core들은 task를 균등하게 할당받아 처리할 수 있습니다.

Worker Pool 마다 최대 처리할 수 있는 task 의 개수가 정해져있습니다.

각 core의 queue에 들어있는 task의 개수를 포함하여 최대 처리 가능 task 수가 넘어가면 worker pool은 task를 받지 않습니다.

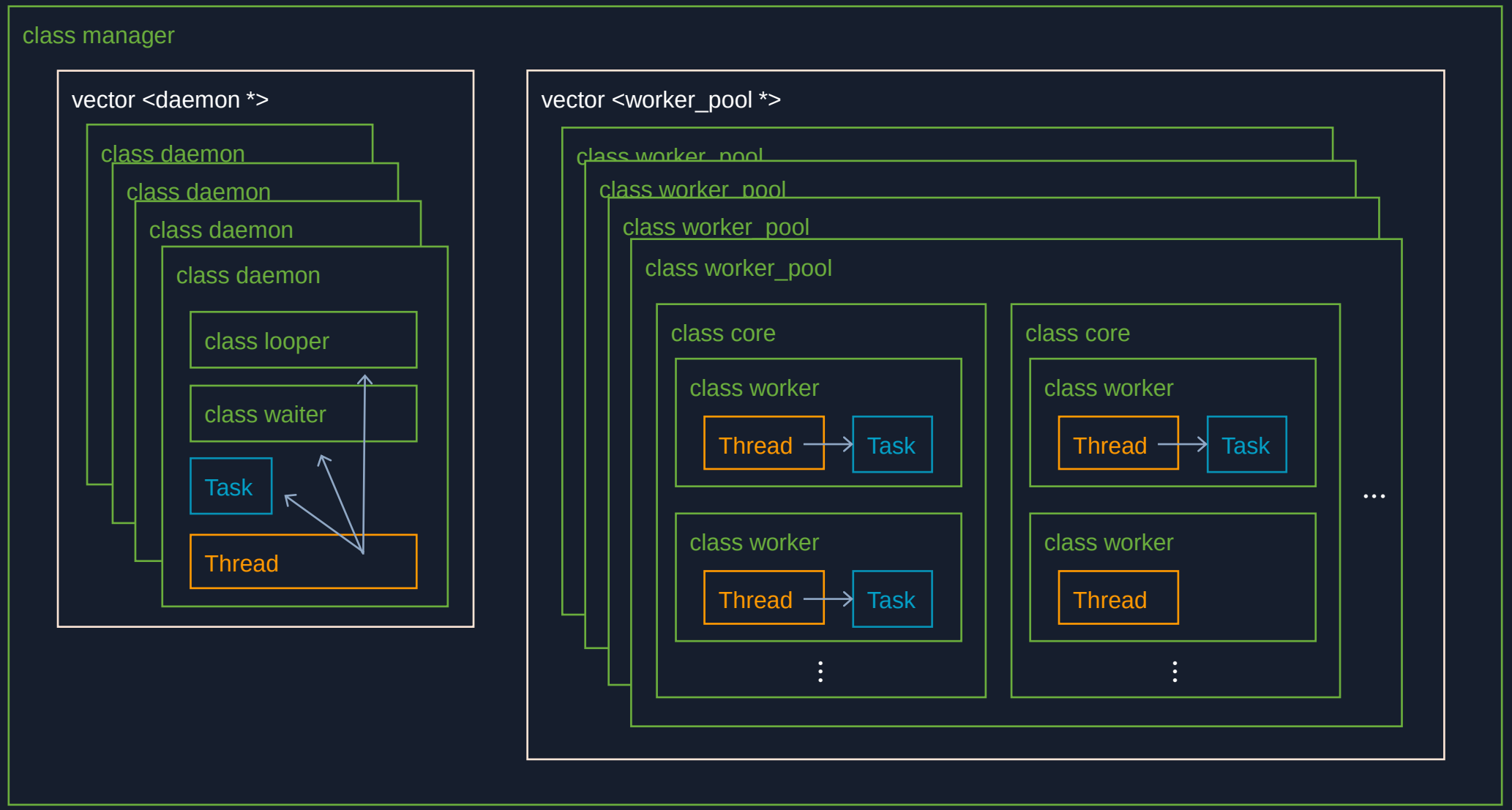

try_execute
task의 수가 worker pool의
처리 가능 수를 넘었다
yes
no
task를 할당할 core 요청
해당 core의
execute_task
실행
get_round_robin_core_hash
round robin counter
round robin counter % core 값을 반환
round robin counter + 1 값이 worker 총 개수와 같다
yes
no
round robin counter
에 0 저장
round robin counter 에
round robin counter + 1 값을 저장



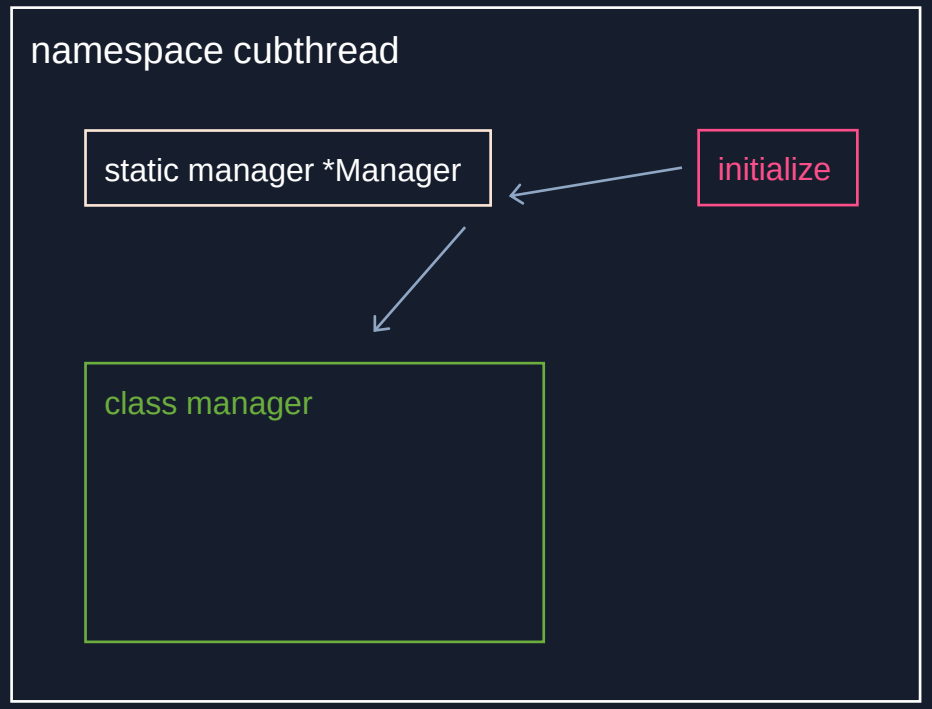
worker 의 총 개수가 core 수의 배수가 아니라면 위처럼 몇 개의 core들은 추가적인 worker를 갖게 됩니다.

Core 1	Core 2	Core 3	Core 4
1	2	3	4
5	6	7	8
9	10	11	12
13			
14	15	16	17
18	19	20	21
22	23	24	25
26			
27	28	29	30

Manager

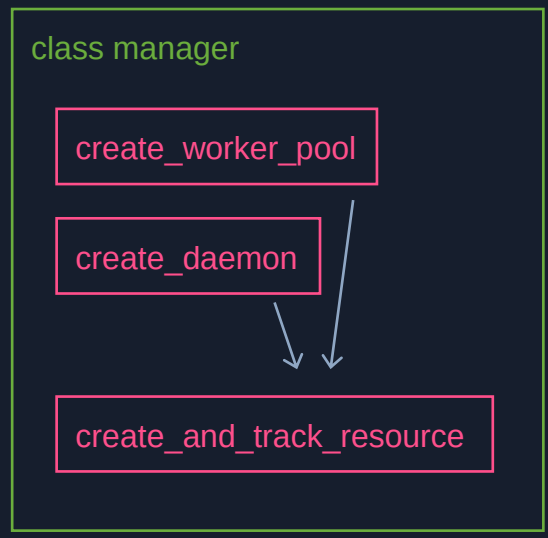

class manager
vector <daemon *>
class daemon
class daemon
class daemon
class daemon
class looper
class waiter
Task
Thread
vector <worker_pool *>
class worker_pool
class worker_pool
class worker_pool
class worker_pool
class core
class worker
Thread
Task
class worker
Thread
Task
class core
class worker
Thread
Task
class worker
Thread
...

Manager



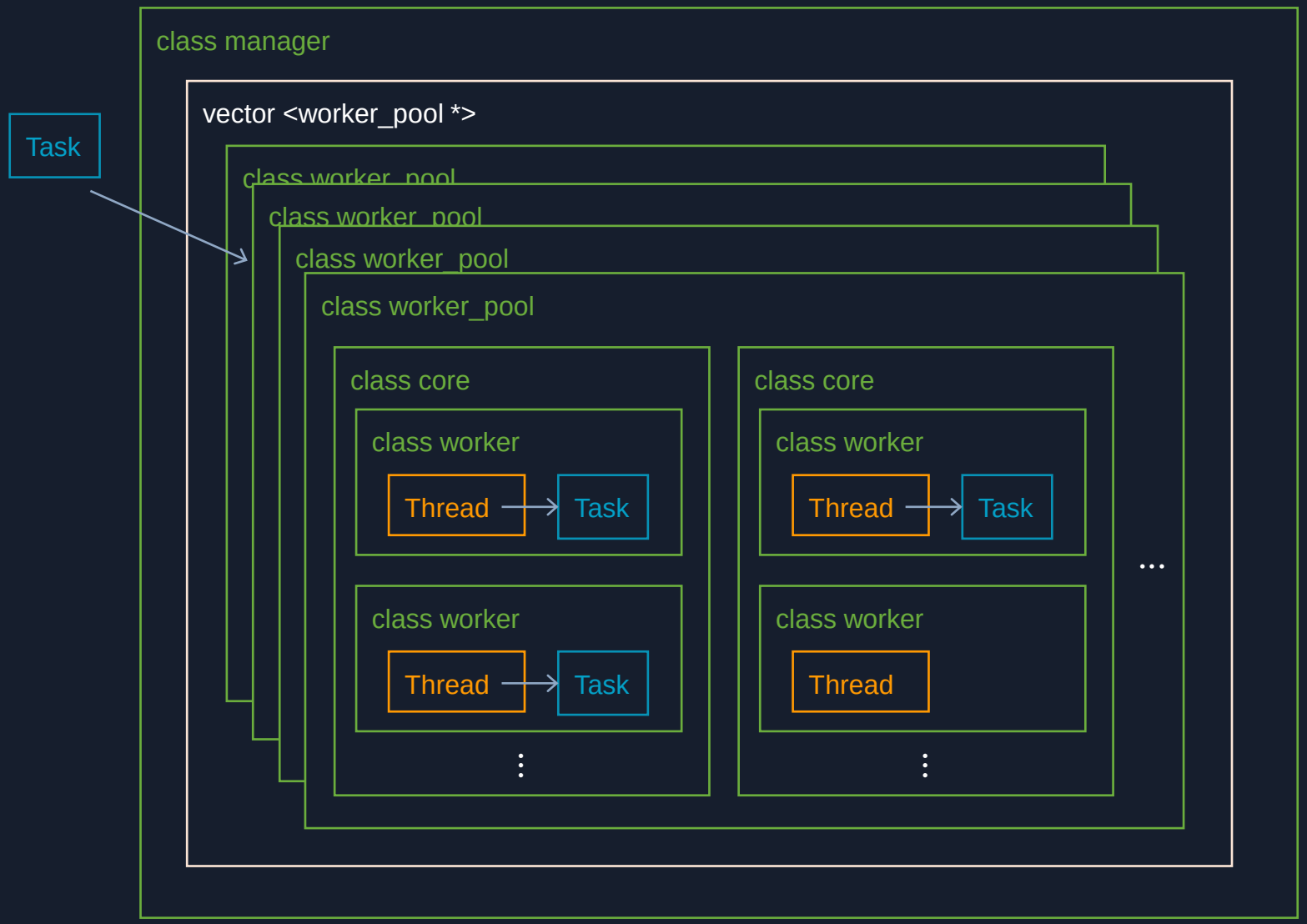
cubthread 초기화 시점이 되는 cubthread::initialize 가 호출될 때 Manage 도 할당됩니다.

cubthread는 server에만 존재하며 각각 존재합니다.

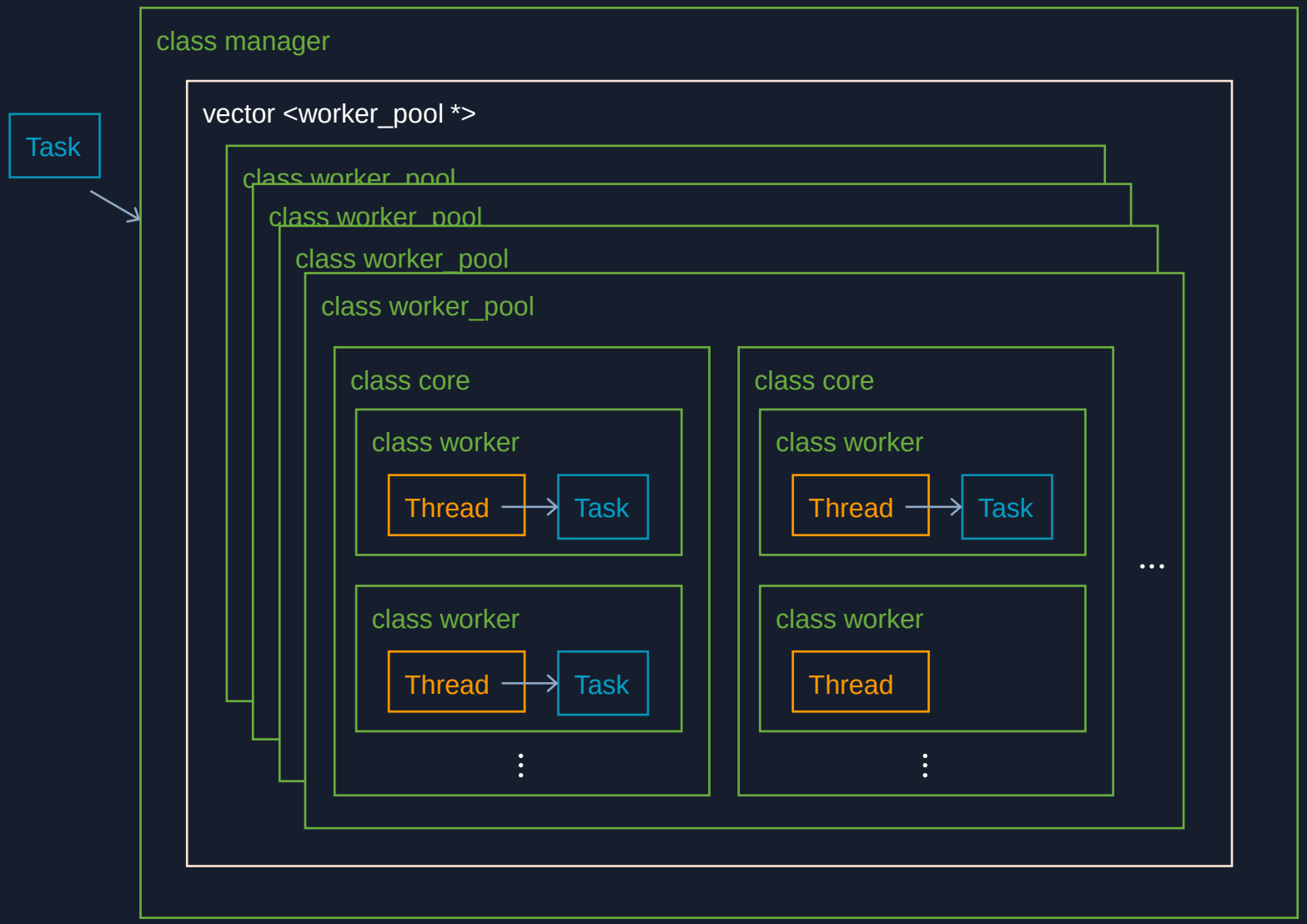


위의 두 함수가 호출되면 최종적으로 create_and_track_resource 를 호출하여 자원이 추가됩니다.

추가될 수 있는 자원은 현재로서는 daemon과 worker pool 이 있습니다.



일반적으로 manager에 task의 실행을 요청할 때, worker pool을 지목해 해당 worker pool에 task를 넘겨줍니다.



하지만, worker pool 을 지목하지 않고 그냥 manager에 task를 넘겨줄 수 있습니다.

이 경우, 새 thread 생성 없이 해당 manager 자체에서 task를 실행합니다.

Thank you