

## DefaultThreadPool Implementation Explained

User

```
class DefaultThreadPool<JOB : Runnable> : ThreadPool<JOB> {

    companion object {
        private const val MAX_WORKER_NUMBERS = 10
        private const val DEFAULT_WORKER_NUMBERS = 5
        private const val MIN_WORKER_NUMBERS = 1
    }

    private val jobs = LinkedList<JOB>()
    private val workers = Collections.synchronizedList(ArrayList<Worker>())

    private var workerNum = DEFAULT_WORKER_NUMBERS
    private val threadNum = AtomicLong()

    constructor() {
        addWorkersInternal(DEFAULT_WORKER_NUMBERS)
    }

    constructor(num: Int) {
        workerNum = if (num > MAX_WORKER_NUMBERS) MAX_WORKER_NUMBERS else max(MIN_WORKER_NUMBERS, num)
        addWorkersInternal(workerNum)
    }

    private inner class Worker(var thread: Thread? = null) : Runnable {

        @Volatile
        private var isRunning = true

        override fun run() {
            while (isRunning) {
                var job: JOB
                synchronized(jobs) {
                    while (jobs.isEmpty()) {
                        try {
                            jobs.wait()
                        } catch (_: InterruptedException) {
                            Thread.currentThread().interrupt()
                            return
                        }
                    }
                }
                job = jobs.removeFirst()
            }
            try {
                job.run()
            } catch (_: Exception) {
            }
        }

        fun shutdown() {
            isRunning = false
            thread?.interrupt()
        }
    }

    private fun addWorkersInternal(num: Int) {
        repeat(num) {
            val worker = Worker()
            workers.add(worker)
            val thread = Thread(worker, "ThreadPool-Worker-${threadNum.incrementAndGet()}")
            worker.thread = thread
            thread.start()
        }
    }

    override fun execute(job: JOB) {
        synchronized(jobs) {
            jobs.addLast(job)
            jobs.notify()
        }
    }

    override fun shutdown() {
        workers.forEach { it.shutdown() }
    }
}
```

```

    override fun addWorkers(num: Int) {
        var n = num
        synchronized(jobs) {
            if (n + this.workerNum > MAX_WORKER_NUMBERS) {
                n = MAX_WORKER_NUMBERS - this.workerNum
            }
            addWorkersInternal(n)
            this.workerNum += n
        }
    }

    override fun removeWorker(num: Int) {
        synchronized(jobs) {
            if (num >= this.workerNum) {
                throw IllegalArgumentException("beyond workNum")
            }
            var count = 0
            while (count < num) {
                val worker = workers[count]
                if (workers.remove(worker)) {
                    worker.shutdown()
                    count++
                }
            }
            this.workerNum -= count
        }
    }

    override val jobSize: Int
        get() = jobs.size

    private fun Any.wait() =
        (this as java.lang.Object).wait()

    private fun Any.notify() =
        (this as java.lang.Object).notify()

```

} 解释一下这段代码

#### ChatGPT

这段代码定义了一个通用的 `DefaultThreadPool` 类，它实现了一个简单的线程池，用于管理和执行一组并发任务。以下是对代码的详细解释：

#### ### 类和泛型定义

```

```kotlin
class DefaultThreadPool<JOB : Runnable> : ThreadPool<JOB>
```

```

- 这个类使用泛型 `JOB`，并且要求 `JOB` 实现 `Runnable` 接口。它继承了 `ThreadPool<JOB>` 接口，这个接口定义了线程池的基本操作。

#### ### 伴生对象和常量

```

```kotlin
companion object {
    private const val MAX_WORKER_NUMBERS = 10
    private const val DEFAULT_WORKER_NUMBERS = 5
    private const val MIN_WORKER_NUMBERS = 1
}
```

```

- 伴生对象定义了线程池中工作的线程数的三个常量：`MAX\_WORKER\_NUMBERS`（最大工作线程数）、`DEFAULT\_WORKER\_NUMBERS`（默认工作线程数）和 `MIN\_WORKER\_NUMBERS`（最小工作线程数）。

#### ### 成员变量

```

```kotlin
private val jobs = LinkedList<JOB>()
private val workers = Collections.synchronizedList<ArrayList<Worker>>()
private var workerNum = DEFAULT_WORKER_NUMBERS
private val threadNum = AtomicLong()
```

```

- `jobs` 是一个 `LinkedList<JOB>`，用于存储待执行的任务。
- `workers` 是一个线程安全的 `ArrayList<Worker>`，用于存储工作线程。
- `workerNum` 保存当前工作的线程数，默认值为 `DEFAULT\_WORKER\_NUMBERS`。
- `threadNum` 是一个原子长整型，用于生成线程名称的计数器。

### ### 构造函数

```
``kotlin
constructor() {
    addWorkersInternal(DEFAULT_WORKER_NUMBERS)
}

constructor(num: Int) {
    workerNum = if (num > MAX_WORKER_NUMBERS) MAX_WORKER_NUMBERS else max(MIN_WORKER_NUMBERS, num)
    addWorkersInternal(workerNum)
}
``
```

- 无参构造函数会创建 `DEFAULT\_WORKER\_NUMBERS` 个工作线程。
- 带参构造函数允许用户指定初始的工作线程数，确保线程数在 `MIN\_WORKER\_NUMBERS` 和 `MAX\_WORKER\_NUMBERS` 之间。

### ### 内部类 `Worker`

```
``kotlin
private inner class Worker(var thread: Thread? = null) : Runnable {

    @Volatile
    private var isRunning = true

    override fun run() {
        while (isRunning) {
            var job: JOB
            synchronized(jobs) {
                while (jobs.isEmpty()) {
                    try {
                        jobs.wait()
                    } catch (_: InterruptedException) {
                        Thread.currentThread().interrupt()
                        return
                    }
                }
                job = jobs.removeFirst()
            }
            try {
                job.run()
            } catch (_: Exception) {
            }
        }
    }

    fun shutdown() {
        isRunning = false
        thread?.interrupt()
    }
}
``
```

- `Worker` 是 `ThreadPool` 的工作线程，负责从 `jobs` 队列中获取任务并执行。
- `isRunning` 是一个 `volatile` 变量，用于控制线程的生命周期。
- `run` 方法是线程的入口，它会循环从 `jobs` 队列中获取任务并执行，直到 `isRunning` 变为 `false`。
- `shutdown` 方法用于停止线程的运行。

### ### 添加工作线程的内部方法

```
``kotlin
private fun addWorkersInternal(num: Int) {
    repeat(num) {
        val worker = Worker()
        workers.add(worker)
        val thread = Thread(worker, "ThreadPool-Worker-${threadNum.incrementAndGet()}")
        worker.thread = thread
        thread.start()
    }
}
``
```

- 这个方法用于内部添加指定数量的工作线程，并启动它们。

### ### 接口方法实现

- `execute(job: JOB)`：接收一个任务 `job` 并将其添加到 `jobs` 队列中，然后通知等待线程有新任务可执行。
- `shutdown()`：调用每个工作线程的 `shutdown()` 方法以停止所有工作线程。

- ``addWorkers(num: Int)``: 动态增加线程池中的工作线程数。

- ``removeWorker(num: Int)``: 从线程池中移除指定数量的工作线程, 如果移除数量超过当前线程数, 则抛出异常。

- ``jobSize``: 返回当前待执行任务的数量。

### ``wait`` 和 ``notify`` 方法

```
``kotlin
private fun Any.wait() =
    (this as java.lang.Object).wait()

private fun Any.notify() =
    (this as java.lang.Object).notify()
...

```

- 这两个扩展方法使得 Kotlin 中的 ``Any`` 对象可以调用 Java 的 ``Object.wait()`` 和 ``Object.notify()`` 方法, 来实现线程同步。

### 总结

这个类实现了一个简单的线程池, 能够在多线程环境中安全地管理并发任务。用户可以动态调整工作线程的数量, 线程池会自动处理任务队列并在后台执行任务。

User

为什么这里的workers必须是线程安全的

ChatGPT

在 ``DefaultThreadPool`` 类中, ``workers`` 列表必须是线程安全的, 因为多个线程可能会同时访问和修改这个列表。如果不使用线程安全的集合, 可能会导致数据竞争、列表的状态不一致, 甚至程序崩溃。以下是一些具体的原因:

### 1. \*\*多个线程同时修改 ``workers`` 列表\*\*

- \*\*添加和移除工作线程\*\*: 线程池在运行过程中, 可能会动态增加或减少工作线程。例如, 当调用 ``addWorkers`` 或 ``removeWorker`` 方法时, 线程池需要修改 ``workers`` 列表。这些操作可能会与其他线程执行任务冲突, 导致竞争条件。

- \*\*线程安全的必要性\*\*: 如果 ``workers`` 列表不是线程安全的集合, 那么在没有同步机制的情况下, 可能会出现多个线程同时修改列表的情况, 这会导致数据不一致, 如一个线程添加元素的同时, 另一个线程尝试遍历或删除元素。

### 2. \*\*保证列表的一致性\*\*

- \*\*一致性问题\*\*: 例如, 在 ``removeWorker`` 方法中, 会遍历 ``workers`` 列表并移除指定数量的 ``Worker``。如果列表不是线程安全的, 而其他线程正在同时添加或移除元素, 那么在遍历过程中列表的结构可能会发生变化, 导致 ``ConcurrentModificationException`` 或其他意外行为。

- \*\*线程安全的列表\*\*: 使用 ``Collections.synchronizedList`` 来保证所有对 ``workers`` 列表的访问 (如添加、移除和遍历) 都是线程安全的。

### 3. \*\*线程之间的协调\*\*

- \*\*协调工作线程\*\*: ``workers`` 列表中保存了所有的工作线程对象。线程池需要通过这个列表来管理和协调这些工作线程的启动、关闭等操作。如果多个线程需要同时操作 ``workers`` 列表, 那么必须使用线程安全的集合来保证这些操作的原子性和正确性。

- \*\*避免数据竞争\*\*: 线程安全的集合通过内部的同步机制 (如锁) 来避免数据竞争, 从而保证在多线程环境下 ``workers`` 列表始终处于一致的状态。

### 4. \*\*防止死锁和其他并发问题\*\*

- \*\*死锁和竞态条件\*\*: 如果 ``workers`` 列表是非线程安全的集合, 那么多个线程同时访问这个列表可能导致死锁、竞态条件、数据丢失等问题。这会影响程序的性能, 还可能导致线程池无法正常工作。

通过使用线程安全的集合 (如 ``Collections.synchronizedList``), 可以确保 ``workers`` 列表在多线程环境下的操作是安全的, 从而避免上述潜在的问题。这对于线程池这样需要管理多个并发任务的组件来说至关重要。

User

那为什么, addWorkers和removeWorker中还需要synchronized块

ChatGPT

即使 ``workers`` 列表是线程安全的, ``addWorkers`` 和 ``removeWorker`` 方法中仍然需要使用 ``synchronized`` 块来保护关键代码块, 以确保线程池的状态 (例如, 工作线程的数量) 在修改时保持一致性。这里有几个原因解释为什么需要 ``synchronized`` 块:

### 1. \*\*保护多个共享资源的操作\*\*

- \*\*共享资源\*\*: 在 ``addWorkers`` 和 ``removeWorker`` 中, 除了 ``workers`` 列表之外, 还涉及到其他共享资源的修改, 如 ``workerNum`` (当前工作线程数)。这些资源与 ``workers`` 列表密切相关, 必须保证它们之间的操作是原子的。

- \*\*原子性操作\*\*: 使用 ``synchronized`` 块, 可以确保在一个线程添加或移除工作线程的过程中, 没有其他线程可以同时进入这个代码块, 从而避免在中途状态不一致的问题。

### 2. \*\*防止竞态条件\*\*

- \*\*竞态条件的风险\*\*: 即使 ``workers`` 列表本身是线程安全的, 在操作 ``workerNum``、遍历列表、添加或移除工作线程等多个步骤时, 如果没有 ``synchronized`` 块, 可能会引发竞态条件。例如, 一个线程可能在添加工作线程的过程中被中断, 导致 ``workerNum`` 和实际列表中的线程数不一致。

- **一致性**：通过 `synchronized` 块，可以确保这些操作以原子方式执行，避免了竞态条件，确保线程池在添加或移除线程时的状态是一致的。

### ### 3. 复合操作的原子性

- **复合操作**：`addWorkers` 和 `removeWorker` 涉及一系列操作，例如检查当前线程数、添加/移除线程、更新 `workerNum` 等。这些操作需要作为一个整体原子地执行。如果这些操作不在 `synchronized` 块中保护，可能会导致部分操作被其他线程看到，从而导致逻辑错误。
- **操作顺序和原子性**：例如，在 `addWorkers` 中，`workerNum` 的更新和线程的实际添加是相关联的操作。`synchronized` 块可以确保在整个操作过程中没有其他线程插入或干扰，从而保持逻辑的一致性。

### ### 4. 协调不同的线程操作

- **不同线程的协调**：多个线程可能会同时调用 `addWorkers` 或 `removeWorker`，如果没有 `synchronized` 块保护，会导致它们彼此干扰。比如，一个线程正在增加工作线程，而另一个线程同时试图减少工作线程，如果没有同步机制，结果可能是不可预测的。
- **防止不一致状态**：例如，在 `removeWorker` 方法中，如果没有 `synchronized` 块，可能会发生一个线程减少了工作线程数，但在执行过程中，另一个线程又添加了线程，这会导致 `workerNum` 和实际线程数不同步，造成混乱。

### ### 5. 确保操作的完整性

- **操作完整性**：`synchronized` 块确保在进行操作时，其他线程不能在中途干扰，保证了操作的完整性和线程池状态的准确性。
- **避免多次修改同一资源**：例如，在移除线程时，如果没有同步机制，多个线程可能同时移除同一个工作线程，导致不必要的错误或线程池的状态不正确。

### ### 总结

虽然 `workers` 列表本身是线程安全的，但 `addWorkers` 和 `removeWorker` 方法涉及到对多个共享资源的操作（如 `workerNum` 和 `workers` 列表），以及多个步骤的复合操作。这些操作需要以原子方式执行，以防止竞态条件、状态不一致和其他并发问题。`synchronized` 块提供了这种原子性，确保线程池在添加或移除工作线程时能够正确更新和维护其内部状态。

### User

那为什么获得的锁是 jobs？我用 workers 不行吗

### ChatGPT

在 `DefaultThreadPool` 的 `addWorkers` 和 `removeWorker` 方法中，选择使用 `jobs` 作为锁对象而不是 `workers`，主要是为了确保线程池状态的一致性，并避免死锁等问题。这里有几个关键点来解释为什么使用 `jobs` 作为锁对象，而不是 `workers`：

### ### 1. 锁的选择影响范围

- **jobs 是核心共享资源**：`jobs` 列表是线程池的核心数据结构，几乎所有涉及任务队列的操作（例如任务的添加、移除、等待等）都需要锁住 `jobs` 对象。这使得使用 `jobs` 作为锁对象可以覆盖更多操作，并保证与任务相关的状态一致性。
- **广泛的影响范围**：在线程池中，很多操作是围绕着 `jobs` 列表进行的。例如，`Worker` 线程在运行时也需要对 `jobs` 进行操作和同步。如果 `addWorkers` 和 `removeWorker` 使用不同的锁（如 `workers`），可能会导致不同步问题，影响线程池整体的状态一致性。

### ### 2. 避免死锁

- **死锁的风险**：假设在 `addWorkers` 和 `removeWorker` 中使用 `workers` 作为锁对象，而在其他地方使用 `jobs` 作为锁对象，这可能导致一个线程锁住了 `workers`，同时另一个线程锁住了 `jobs`，它们都等待对方释放锁，从而导致死锁。
- **统一的锁对象**：通过统一使用 `jobs` 作为锁对象，可以避免多个锁之间的竞争和死锁问题。这种统一的锁策略简化了同步逻辑，并减少了潜在的并发错误。

### ### 3. 保持一致性

- **任务队列和工作线程的同步**：`jobs` 列表不仅存储了任务，还决定了工作线程是否有任务可做。在管理工作线程（增加或减少线程）时，操作往往与任务队列的状态有关，因此使用 `jobs` 作为锁对象可以确保这些操作的同步性。
- **避免不一致的状态**：如果在 `addWorkers` 和 `removeWorker` 中使用 `workers` 作为锁对象，而其他与 `jobs` 相关的操作使用 `jobs` 作为锁对象，可能会导致线程池在操作之间看到不一致的状态。例如，任务被添加到 `jobs` 中时，还未唤醒足够的工作线程处理任务，因为两个操作没有被正确同步。

### ### 4. 实现逻辑的简化

- **逻辑上的简化**：选择一个关键对象（`jobs`）作为同步的锁，可以简化线程池的实现逻辑。程序员只需要记住大多数与线程池状态相关的操作都要同步在 `jobs` 上，这减少了代码中可能引入的同步错误。
- **一致的同步策略**：如果在一个方法中使用 `workers`，在另一个方法中使用 `jobs`，这会导致同步策略变得复杂和难以管理。统一使用 `jobs` 作为锁对象有助于保持代码的一致性和可维护性。

### ### 总结

虽然 `workers` 列表在增加和删除工作线程时会被修改，但 `jobs` 列表与线程池的整体状态更紧密相关。使用 `jobs` 作为锁对象，可以保证与任务队列和工作线程相关的操作在全局范围内的一致性。这样可以避免死锁，确保线程池的所有操作能够正确同步，同时保持代码的简洁性和易维护性。

使用 `jobs` 作为锁对象是一种设计选择，确保线程池在多线程环境下的正确行为。如果使用 `workers` 作为锁对象，可能会引入复杂的同步问题，导致线程池状态不一致或出现并发错误。

### User

但是这两个方法里根本没对 jobs 做任何修改。锁上又有什么作用

### ChatGPT

你指出的很正确，`addWorkers` 和 `removeWorker` 方法中确实没有对 `jobs` 列表直接进行任何修改。锁住 `jobs` 的目的并不是为了保护对 `jobs` 的直接修改，而是为了保护与 `jobs` 相关的操作和线程池的整体一致性。这种锁的选择可能是出于以下原因：

### ### 1. \*\*确保线程池状态的整体一致性\*\*

虽然 `addWorkers` 和 `removeWorker` 方法本身不直接修改 `jobs` 列表，但它们修改了工作线程的数量，这些线程会对 `jobs` 列表进行操作。锁住 `jobs` 确保了在调整工作线程数的过程中，没有线程在 `jobs` 上执行与任务处理相关的操作（例如，添加任务、移除任务或等待任务的到来）。

这种锁的选择可以保证：

- 在添加或移除线程的过程中，`jobs` 列表的状态不会因为其他线程的操作而变得不一致。
- 确保在添加或移除线程期间，工作线程的数量和任务队列的状态是同步的。

### ### 2. \*\*防止竞态条件\*\*

在多线程环境中，如果 `addWorkers` 或 `removeWorker` 方法在执行时，其他线程正处于等待任务或处理任务的状态，可能会发生竞态条件。例如：

- 当添加新的工作线程时，如果不锁住 `jobs`，新的工作线程可能会在 `jobs` 被其他线程修改的过程中启动，导致状态的不一致。
- 在减少工作线程时，如果不锁住 `jobs`，可能会导致在减少线程的过程中，其他线程正在从 `jobs` 列表中取任务或添加任务，进而引发状态不一致或线程池的错误行为。

### ### 3. \*\*协调与任务处理相关的操作\*\*

锁住 `jobs` 可以确保在添加或移除工作线程时，所有与任务处理相关的操作（包括任务的添加、取出、等待等）都能正确协调。这可以避免工作线程数的变化对任务队列处理带来的不确定性。

### ### 4. \*\*保持同步逻辑的一致性\*\*

虽然在 `addWorkers` 和 `removeWorker` 中没有直接修改 `jobs` 列表，但 `jobs` 是线程池中的一个关键共享资源。将 `jobs` 作为锁对象，可以确保在整个线程池的操作中使用一致的同步策略，避免因锁对象不一致而引发的同步问题。例如，在其他涉及 `jobs` 列表的操作（如任务的 `execute`）中，通常也会锁住 `jobs`，这样就可以确保这些操作与工作线程的调整之间的同步一致性。

### ### 5. \*\*可能的设计遗留或扩展性考虑\*\*

另一种可能性是，设计者使用 `jobs` 作为锁对象是出于未来扩展性或设计上的一致性考虑。即便当前 `addWorkers` 和 `removeWorker` 没有修改 `jobs`，但未来的修改或扩展可能会引入对 `jobs` 的直接操作。

### ### 总结

尽管在 `addWorkers` 和 `removeWorker` 中没有直接修改 `jobs` 列表，但锁住 `jobs` 可以确保与任务处理相关的所有操作的同步性和线程池状态的一致性。这样可以防止竞态条件，保持代码的一致性，并确保在多线程环境下线程池的稳定性和正确性。

### User

你刚刚说的第二点里。即使不锁jobs，新的线程启动的时候也不会启动jobs中的任务。因为线程的run方法里面也会锁住jobs

### ChatGPT

User is analyzing thread synchronization and locking mechanisms in a custom thread pool implementation.

### Tool

Model set context updated.

### ChatGPT

你说得很对。`Worker` 线程在运行时，会锁住 `jobs` 列表，以确保对任务队列的操作是线程安全的。这确实可以防止新的线程在启动时立即干扰其他正在处理 `jobs` 列表的线程。

在这种情况下，`addWorkers` 和 `removeWorker` 方法中锁住 `jobs` 的意义似乎减少了，因为 `Worker` 线程的关键部分已经通过 `jobs` 锁进行了保护。既然 `Worker` 线程在处理任务时已经有了适当的同步措施，为什么还要在这些方法中锁住 `jobs` 呢？

### ### 重新考虑 `synchronized` 块的使用

#### 1. \*\*锁住 `workers` 更合适\*\*

- 如果 `addWorkers` 和 `removeWorker` 的主要操作是对 `workers` 列表进行修改，那么直接锁住 `workers` 列表似乎更合理。这可以确保在添加或移除工作线程时，对 `workers` 列表的所有修改都能正确同步。
- 这样做也符合“锁住实际操作的共享资源”的原则。

#### 2. \*\*减少不必要的锁竞争\*\*

- 如果使用 `workers` 作为锁对象，避免了不必要的锁竞争，因为 `jobs` 锁主要用于任务队列的操作，`workers` 锁则用于线程管理，这样可以将锁的使用分开，减少竞争。

### ### 总结

在这种特定场景下，如果不需要对 `jobs` 列表进行操作，且 `Worker` 线程本身已经锁住了 `jobs` 列表，那么在 `addWorkers` 和 `removeWorker` 中锁住 `jobs` 可能是多余的。锁住 `workers` 列表可能更合理，因为这些方法的主要操作是对 `workers` 列表的修改。这样可以更精确地控制锁的使用范围，并减少锁竞争的可能性。