

java并发

什么是多线程中的上下文切换？

CPU从一个线程切换到另一个线程时，需要保存当前线程的上下文状态，包括程序计数器、寄存器、栈指针等，以便下次继续执行时该线程能够恢复到正确的执行状态

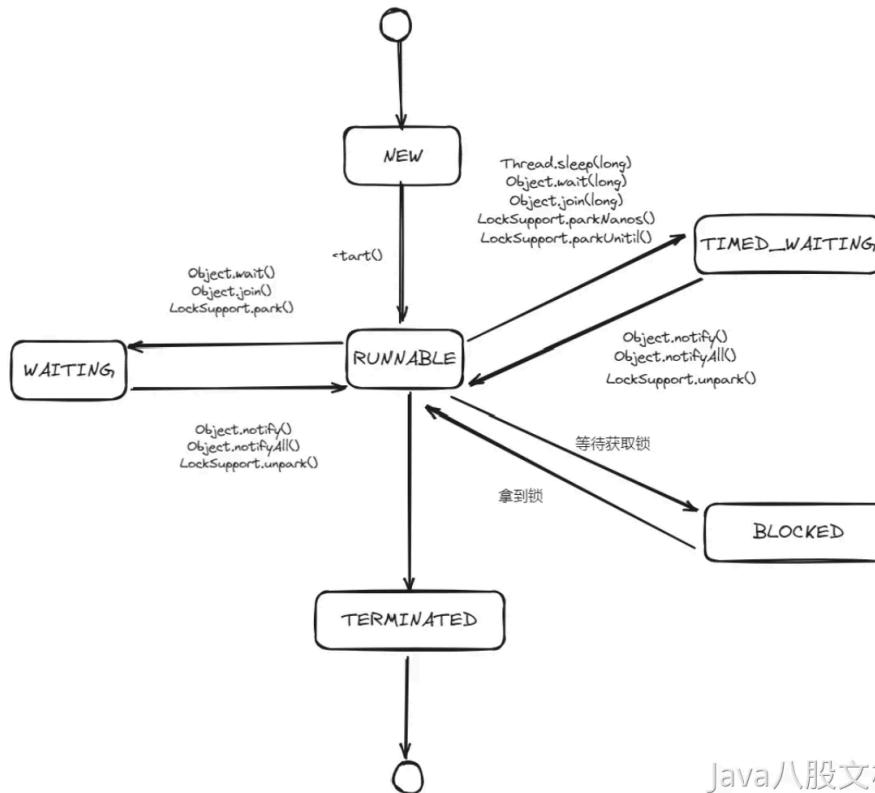
过多的上下文切换会降低系统的运行效率，开销比直接使用单线程大

如何减少上下文切换？

- 减少线程数：通过线程池来管理，减少线程的创建和销毁
- 使用无锁并发编程：无锁并发编程可以避免线程因等待锁而进入阻塞状态，从而减少上下文切换的发生
- 使用CAS：CAS可以避免线程的阻塞和唤醒，减少上下文切换
- 使用协程：协程是一种用户态线程，其切换不需要操作系统的参与，因此可以避免上下文切换。（避免的是操作系统级别的上下文切换，但是仍然需要在JVM层面做一些保存和恢复线程的状态，但是也成本低得多）
- 合理使用锁：需要避免过多地使用同步块或同步方法，尽量缩小同步块或同步方法的范围，从而减少线程的等待时间，避免上下文切换的发生

线程的几种状态、状态之间的流转

- 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法
- 运行(RUNNABLE)：Java线程中将就绪（READY）和运行中（RUNNING）两种状态笼统的称为“运行”
 - 就绪（READY）：线程对象创建后，其他线程（比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中并分配CPU使用权
 - 运行中（RUNNING）：就绪(READY)的线程获得了CPU时间片，开始执行程序代码
- 阻塞(BLOCKED)：表示线程阻塞于锁
- 等待(WAITING)：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）
- 超时等待(TIMED_WAITING)：该状态不同于WAITING，它可以在指定的时间后自行返回
- 终止(TERMINATED)：表示该线程已经执行完毕



Java八股文档_By Hollis

WAITING和TIMED_WAITING的区别？

- WAITING是等待状态，在Java中，调用wait方法时，线程会进入到WAITING状态
- TIMED_WAITING是超时等待状态，当线程执行sleep方法时，线程会进入TIMED_WAIT状态
- 处于WAITING和TIMED_WAIT的线程，都是会让出CPU的，这时候其他线程就可以获得CPU时间片开始执行
- 在对象的锁释放上面并不一样，如果加了锁，sleep方法不会释放对象上的锁，而wait方法是会释放锁的

为什么线程没有RUNNING状态？

因为CPU时间片很短，一般也就是10~20ms，大概1s内，同一个线程可能一部分时间处于READY状态，一部分时间又处于RUNNING状态

所以这个状态不准，只需要知道线程是不是可执行的，只要获得时间片，就能立即执行

守护线程和普通线程的区别？

- 用户线程：一般用于执行用户级任务
- 守护线程：即后台线程，最经典的就是GC。在守护线程中产生的新线程也是Daemon的

JVM会在所有用户线程都执行完毕后再退出，而不会等守护线程执行完

JDK21中的虚拟线程

虚拟线程就是协程

JAVA的线程模型，主要采用的是基于轻量级进程实现的一对一的线程模型（每一个java线程对应一个操作系统中的轻量级进程）

线程的创建、析构以及同步等动作都需要进行系统调用，在用户态和内核态之间来回切换

虚拟线程不再是一个线程都一对一对应一个操作系统的线程了，而是会将多个虚拟线程映射到少量操作系统线程中，通过有效的调度来避免那些上下文切换

虚拟线程和平台线程的区别

- 虚拟线程总是守护线程，且无法被更改。需要注意的是，当所有启动的非守护线程都终止时，JVM将终止。这意味着JVM不会等待虚拟线程完成后才退出
- 虚拟线程不能更改优先级，即使使用 `setPriority()`，虚拟线程始终拥有normal的优先级
- 虚拟线程是不支持`stop()`、`suspend()`或`resume()`等方法

如何使用

- 通过 `Thread.startVirtualThread()` 可运行一个虚拟线程

```
1 Thread.startVirtualThread(() -> {  
2     System.out.println("虚拟线程执行中...");  
3 });
```

- 通过 `Thread.Builder` 也可以创建虚拟线程

- `Thread.ofPlatform()` 创建平台线程
 - `Thread.ofVirtual()` 创建虚拟线程

```
1 Thread.Builder platformBuilder = Thread.ofPlatform().name("平台线程");  
2 Thread.Builder virtualBuilder = Thread.ofVirtual().name("虚拟线程");  
3  
4 Thread t1 = platformBuilder.start(() -> {...});  
5 Thread t2 = virtualBuilder.start(() -> {...});
```

- 线程池创建虚拟线程，可以通过 `Executors.newVirtualThreadPerTaskExecutor()` 来创建虚拟线程（不建议，因为线程池的设计是为了避免创建新的操作系统线程的开销）

```
1 try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
2     IntStream.range(0, 10000).forEach(i -> {  
3         executor.submit(() -> {  
4             Thread.sleep(Duration.ofSeconds(1));  
5             return i;  
6         });  
7     });  
8 }
```

创建线程的几种方式

- 继承**Thread**类创建线程
- 实现**Runnable**接口创建线程
- 通过**Callable**和**FutureTask**创建线程
- 通过**线程池**创建线程

Runnable和Callable的区别

- 都可以用来创建新的线程，实现**Runnable**接口需要**run**方法，实现**Callable**接口需要实现**call**方法
- Runnable**的**run**无返回值，**Callable**的**call**有返回值，类型为**Object**
- Callable**中可以抛出受检异常，**Runnable**不可以
- Callable**和**Runnable**都可以应用于**executors**，而**Thread**只支持**Runnable**

Future

Future是一个接口，代表了一个异步执行的结果。接口中的方法用来检查执行是否已经完成，等待完成和得到执行结果。

当执行完成后，只能通过 `get()` 方法得到结果，它会阻塞直到结果准备好了。

如果想取消，那么调用 `cancel()` 方法

FutureTask是**Future**的一个实现，它实现了一个可以提交给**Executor**执行的任务，并且可以用来检查任务的执行状态和获取任务的执行结果

```
1 // FutureTask和Callable的实例
2 import java.util.concurrent.Callable;
3 import java.util.concurrent.ExecutionException;
4 import java.util.concurrent.FutureTask;
5
6 public class FutureAndCallableExample {
7     public static void main(String[] args) throws InterruptedException,
8         ExecutionException {
9         Callable<String> callable = () -> {
10             System.out.println("Entered Callable");
11             Thread.sleep(2000);
12             return "Hello from Callable";
13         };
14
15         FutureTask<String> futureTask = new FutureTask<>(callable);
16         Thread thread = new Thread(futureTask);
17         thread.start();
18
19         System.out.println("Do something else while callable is getting
20             executed");
21         System.out.println("Retrieved: " + futureTask.get());
22     }
23 }
```

```
1 // 线程池和Callable的实例
2 import java.util.concurrent.Callable;
3 import java.util.concurrent.ExecutionException;
4 import java.util.concurrent.ExecutorService;
5 import java.util.concurrent.Executors;
6 import java.util.concurrent.Future;
7
8 public class FutureAndCallableExample {
9     public static void main(String[] args) throws InterruptedException,
10         ExecutionException {
11         ExecutorService executor = Executors.newSingleThreadExecutor();
12         Callable<String> callable = () -> {
13             System.out.println("Entered Callable");
14             Thread.sleep(2000);
15             return "Hello from Callable";
16         };
17
18         System.out.println("Submitting Callable");
19         Future<String> future = executor.submit(callable);
20
21         System.out.println("Do something else while callable is getting
```

```
    executed");
21         System.out.println("Retrieved: " + future.get());
22
23     executor.shutdown();
24 }
25 }
```

run/start, wait/sleep, notify/notifyAll的区别

run和start的区别

- start方法是启动一个线程的入口
- run方法直接调用，就会在单线程中直接运行run方法，起不到多线程的效果

sleep和wait的区别

- sleep可以在任何地方使用，而wait只能在同步方法或者同步代码块中使用
- wait会释放对象锁，而sleep不会
- wait的线程会进入 WAITING 状态，直到被唤醒；sleep的线程会进入 TIMED_WAIT 状态，等到指定时间后会再尝试获取CPU时间
- wait释放锁，因为java锁的目标是对象，所以针对的目标都是对象，所以定义在Object类中。而sleep不需要释放锁，针对的目标是线程，所以定义在Thread类中

notify和notifyAll的区别

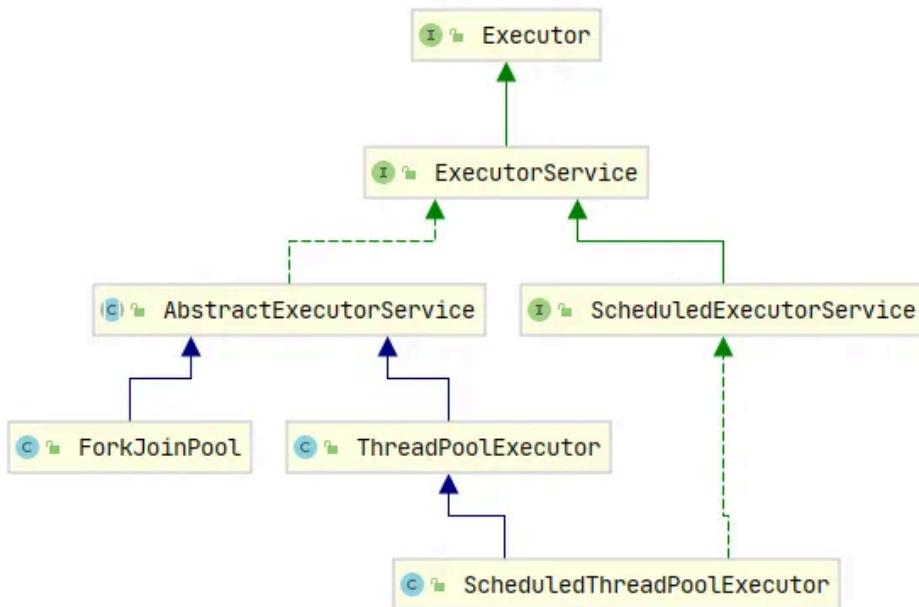
一个线程进入 WAITING 状态后，必须等待其他线程 `notify()` / `notifyAll()` 才会从等待队列中被移除

使用 `notifyAll()` 可以唤醒所有处于 WAITING 状态的线程，使其重新进入锁的竞争队列中，而 `notify()` 只能唤醒一个

虽然 `notifyAll()` 可以唤醒所有的线程，让他们都竞争锁，但是最终只有一个可以获得锁并执行

但是唤醒了也只是进入竞争队列，并不代表可以立刻获得CPU时间片开始运行，因为wait方法被调用的时候，线程就已经释放了对象锁

线程池如何实现



Java八股文档_By Hollis

Executors

Executors创建出来的线程池都实现了ExecutorService接口，常用方法如下

- newFixedThreadPool(int threads): 创建固定数目线程的线程池
- newCachedThreadPool(): 创建一个可缓存的线程池
 - 调用execute 将重用以前构造的线程（如果线程可用）。如果没有可用的线程，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程
- newSingleThreadExecutor(): 创建一个单线程化的Executor
- newScheduledThreadPool(int corePoolSize): 创建一个支持定时以及周期性的任务执行线程池，多数情况可以替代Timer

```

1   public ThreadPoolExecutor(int corePoolSize,
2                             int maximumPoolSize,
3                             long keepAliveTime,
4                             TimeUnit unit,
5                             BlockingQueue<Runnable> workQueue,
6                             ThreadFactory threadFactory,
7                             RejectedExecutionHandler handler) {
8     if (corePoolSize < 0 ||
9         maximumPoolSize <= 0 ||
10        maximumPoolSize < corePoolSize ||
11        keepAliveTime < 0)
  
```

```
12         throw new IllegalArgumentException();
13     if (workQueue == null || threadFactory == null || handler == null)
14         throw new NullPointerException();
15     this.acc = System.getSecurityManager() == null ?
16         null :
17         AccessController.getContext();
18     this.corePoolSize = corePoolSize;
19     this.maximumPoolSize = maximumPoolSize;
20     this.workQueue = workQueue;
21     this.keepAliveTime = unit.toNanos(keepAliveTime);
22     this.threadFactory = threadFactory;
23     this.handler = handler;
24 }
```

- acc: 获取调用上下文
- corePoolSize: 核心线程数量，常驻的线程数量
- maximumPoolSize: 最大的线程数量，常驻+临时线程数量
- workQueue: 多余任务等待队列
- keepAliveTime: 非核心线程空闲时间，超过这个时间，就消灭
- threadFactory: 创建线程的工厂，这个地方可以统一处理创建的线程的属性
- handler: 线程池拒绝策略

源码分析 ::

execute方法 ::

用于往线程池中添加一个任务

```
1   public void execute(Runnable command) {
2       if (command == null)
3           throw new NullPointerException();
4       /*
5        * Proceed in 3 steps:
6        *
7        * 1. If fewer than corePoolSize threads are running, try to
8        * start a new thread with the given command as its first
9        * task.  The call to addWorker atomically checks runState and
10       * workerCount, and so prevents false alarms that would add
11       * threads when it shouldn't, by returning false.
12       *
13       * 2. If a task can be successfully queued, then we still need
```

```

14         * to double-check whether we should have added a thread
15         * (because existing ones died since last checking) or that
16         * the pool shut down since entry into this method. So we
17         * recheck state and if necessary roll back the enqueueing if
18         * stopped, or start a new thread if there are none.
19         *
20         * 3. If we cannot queue task, then we try to add a new
21         * thread. If it fails, we know we are shut down or saturated
22         * and so reject the task.
23         */
24         int c = ctl.get();
25         // 判断当前线程数是否小于核心线程数, 如果小于, 则执行addWorker方法创建新的线程
26         执行任务
27         if (workerCountOf(c) < corePoolSize) {
28             if (addWorker(command, true))
29                 return;
30             c = ctl.get();
31         }
32         // 判断线程池是否在运行, 如果在, 任务队列是否允许插入
33         if (isRunning(c) && workQueue.offer(command)) {
34             int recheck = ctl.get();
35             // 插入成功后再次验证线程池能否运行, 如果不行, 则移除插入的任务, 并抛出拒绝
36             if (!isRunning(recheck) && remove(command))
37                 reject(command);
38             // 如果在运行, 但是没有线程了, 就启用一个线程
39             else if (workerCountOf(recheck) == 0)
40                 addWorker(null, false);
41         }
42         // 如果添加非核心线程失败, 就直接拒绝
43         else if (!addWorker(command, false))
44             reject(command);

```

addWorker方法 ::

```

private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (; ; ) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            !(rs == SHUTDOWN &&
              firstTask == null &&
              !workQueue.isEmpty())))
            return false;

        for (; ; ) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY || wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get(); // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
}

```

```

boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // Recheck while holding lock.
            // Back out on ThreadFactory failure or if
            // shut down before lock acquired.
            int rs = runStateOf(ctl.get());

            if (rs < SHUTDOWN || (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive()) // precheck that t is startable
                    throw new IllegalThreadStateException();
                workers.add(w);
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                workerAdded = true;
            }
        } finally {
            mainLock.unlock();
        }
        if (workerAdded) {
            t.start();
            workerStarted = true;
        }
    }
} finally {
    if (!workerStarted)
        addWorkerFailed(w);
}
return workerStarted;
}

```

1. 做是否能够添加工作线程条件的过滤

判断线程池的状态，如果线程池的状态 \geq SHUTDOWN，则不处理提交的任务

2. 自旋，更新创建工作线程数量

通过参数core判断要穿件的线程是否为核心线程

3. 获取线程池主锁

线程池的工作线程通过Worker类实现，通过ReentrantLock锁来保证线程安全

4. 添加线程到workers中（线程池中）

5. 启动新的线程

workers底层就是一个hashSet

```
1 private final HashSet<Worker> workers = new HashSet<Worker>();
```

runWorker方法 ::


```

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        while (task != null || (task = getTask()) != null) {
            w.lock();
            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted. This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}

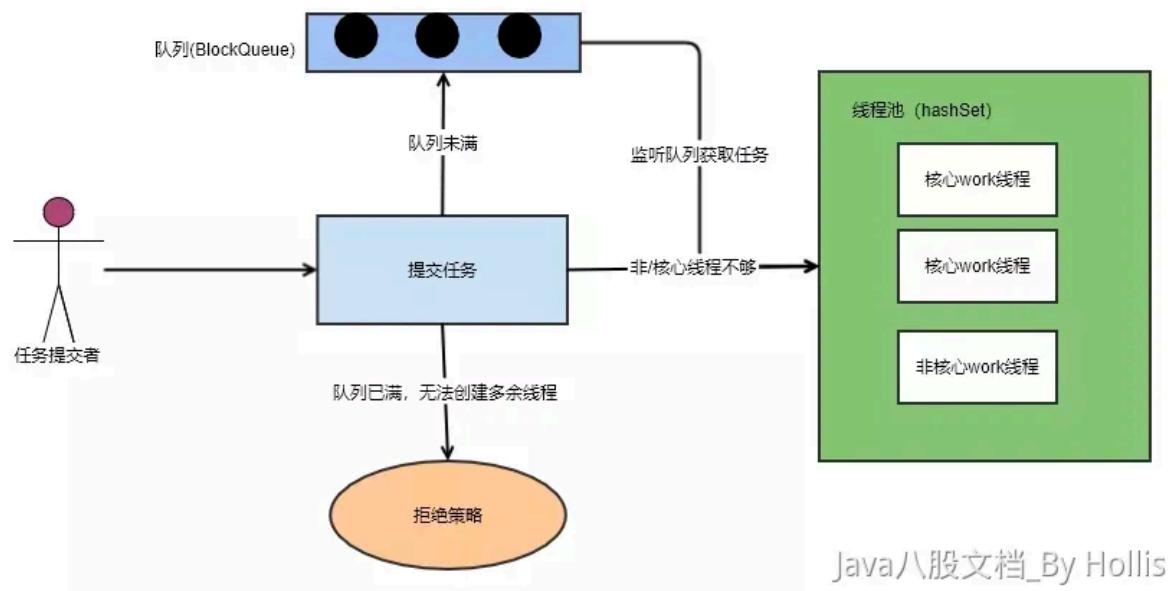
```

- 是否第一次执行任务，或者从队列中可以获取到任务
- 获取到任务后执行前置钩子
- 执行任务
- 执行任务后置钩子

`beforeExecute`、`afterExecute` 允许我们自己继承线程池，做任务执行的前后处理

总结：

1. 线程池的本质是一个`hashSet`，多余的任务会放在阻塞队列
2. 只有当阻塞队列满的时候，才会触发非核心线程的创建
3. 线程池提供了两个钩子 `beforeExecute` 和 `afterExecute`，可以让我们自己继承线程池来实现



Java八股文档_By Hollis

线程数设定多少比较合适。

影响线程数的因素

- CPU核数
 - 多核处理器：理想情况下每个核心运行一个线程最高效
 - 超线程技术：利用特殊的硬件指令，让两个逻辑内核模拟成两个物理芯片，让单个处理器拥有线程级并行计算能力
- 应用类型

- CPU密集型：对于CPU密集型的任务（如计算密集型任务），线程数最好设置为核心数的1到1.5倍，因为这些任务主要消耗CPU资源
- IO密集型：如果任务涉及大量的等待或阻塞（如数据库操作、文件操作、网络操作等），则可以配置更多的线程，比如2倍，因为线程在等待时CPU可以切换去处理其他任务
- JVM和系统资源
 - 内存限制：每个线程会占用一定的内存（如栈空间），如果创建过多的线程，可能会消耗大量内存，甚至导致内存溢出
 - 操作系统限制：操作系统对进程可创建的线程数通常有限制，过多的线程可能导致系统性能下降
- RT要求：如果系统对响应时间有严格要求，可能需要更多线程来减少处理时延
- 任务特性：不同的任务可能对线程数的需求不同，长时任务与短时任务，同步任务与异步任务都需要考虑不同的线程配置

参考公式：

- 如果是CPU密集型，线程池大小设置为N+1
- 如果是IO密集型，线程池大小设置为2N+1

如果一个应用不能明确定位出是CPU密集型还是IO密集型

$$\text{线程数} = \text{CPU核心数} \times \text{目标CPU利用率} \times (1 + \frac{\text{等待时间}}{\text{计算时间}})$$

- 等待时间是指线程在执行过程中花费在外部等待操作完成的时间，通常包括IO，数据库操作，网络请求等，或者其他资源的同步等待。
在等待时间内，线程通常不占用CPU资源
- 计算时间是指线程实际进行处理的时间，即线程在CPU上执行操作的时间。计算时间通常指的是CPU密集型操作，如数学计算、数据处理等。
- I/O密集型任务：对于I/O密集型任务，等待时间通常远大于计算时间，这意味着可以分配更多的线程。当一个线程在等待时（如等待网络响应），CPU可以切换到另一个线程进行计算，从而提高CPU利用率。
- CPU密集型任务：对于CPU密集型任务，计算时间通常占主导地位。在这种情况下，增加线程数可能不会提高性能，因为大部分时间都在进行CPU计算，线程之间的上下文切换可能导致性能下降。

不建议直接套公式，因为设置线程池大小还需要考虑JVM，机器资源等，有时候我们看到的CPU核数也不一定是真的

可以在刚上线的时候，先根据公式大致的设置一个数值，然后再根据你自己的实际业务情况，以及不断的压测结果，再不断调整，最终达到一个相对合理的值。

ThreadLocal是如何实现的？

ThreadLocal通过为每一个线程创建一份共享变量的副本来保证各个线程之间的变量的访问和修改互不影响

内部存放的值是线程内共享，线程间互斥

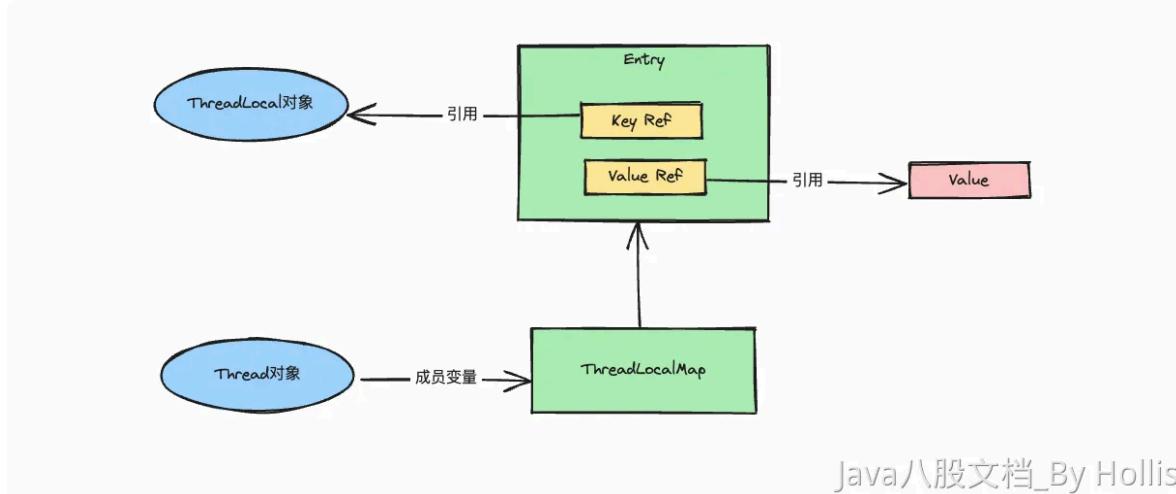
有4个方法

- initialValue: 返回此线程局部变量的初始值
- get: 返回此线程局部变量的当前线程副本中的值，如果这是线程第一次调用，会创建并初始化此副本
- set: 将此线程局部变量的当前线程副本中的值设置为指定值
- remove: 移除此线程局部变量的值

实现原理

ThreadLocal用于保存线程的独有变量的数据结构是一个内部类：**ThreadLocalMap**

key就是当前的ThreadLocal对象，value就是我们想要保存的值



Thread类对象中维护了ThreadLocalMap成员变量，而ThreadLocalMap维护了以ThreadLocal为key，需要存储的数据为value的Entry数组

Thread类中内部维护了两个变量

- threadLocals
- inheritableThreadLocals

初始值都为null，类型为ThreadLocal.ThreadLocalMap

在静态内部类ThreadLocalMap维护一个数据结构类型为Entry的数组

```
1 static class Entry extends WeakReference<ThreadLocal<?>> {
2     /** The value associated with this ThreadLocal. */
3     Object value;
4
5     Entry(ThreadLocal<?> k, Object v) {
6         super(k);
7         value = v;
8     }
9 }
```

Entry结构实际上是继承了一个ThreadLocal类型的弱引用并将其作为key，value为Object类型

ThreadLocalMap内部的变量

```
1 // 默认的数组初始化容量
2 private static final int INITIAL_CAPACITY = 16;
3 // Entry数组，大小必须为2的幂
4 private Entry[] table;
5 // 数组内部元素个数
6 private int size = 0;
7 // 数组扩容阈值，默认为0，创建了ThreadLocalMap对象后会被重新设置
8 private int threshold;
```

构造方法如下

```
1 /**
2  * Construct a new map initially containing (firstKey, firstValue).
3  * ThreadLocalMaps are constructed lazily, so we only create
4  * one when we have at least one entry to put in it.
5 */
6 ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
7     // 初始化Entry数组，大小 16
8     table = new Entry[INITIAL_CAPACITY];
9     // 用第一个键的哈希值对初始大小取模得到索引，和HashMap的位运算代替取模原理一样
10    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
11    // 将Entry对象存入数组指定位置
12    table[i] = new Entry(firstKey, firstValue);
13    size = 1;
14    // 初始化扩容阈值，第一次设置为10
15 }
```

```
16     setThreshold(INITIAL_CAPACITY);  
    }
```

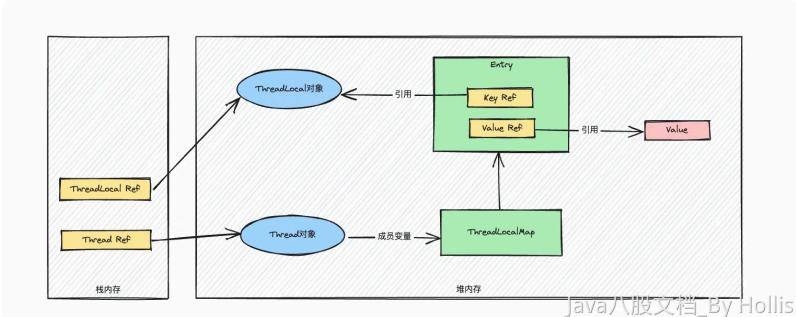
该构造方法是懒加载的，只有当我们创建一个Entry对象并需要放入到Entry数组的时候才会去初始化Entry数组。

ThreadLocal的使用场景

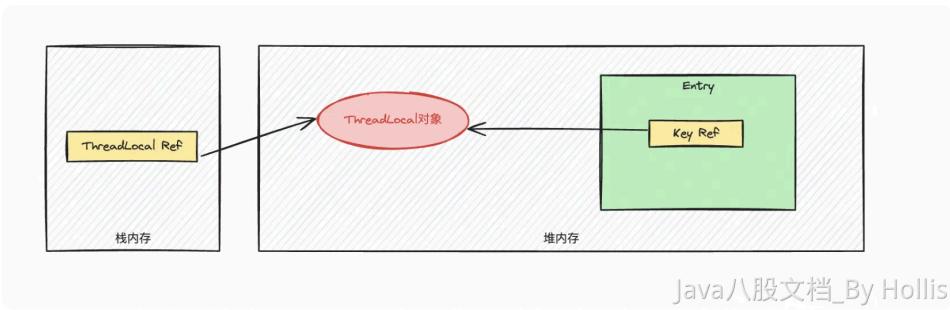
1. 解决并发问题
 - a. 解决simpleDateFormat的线程不安全问题
 - b. 数据库session保证每个线程有自己的会话实例
2. 在线程中传递数据
 - a. 用户身份信息获取
 - b. traceId存储
 - c. PageHelper分页，分页参数信息，页码和页大小都存储在threadLocal中

ThreadLocal内存泄露问题

导致内存泄露的部分其实就是堆上存储的ThreadLocalMap中的K-V部分



Java八股文档_By Hollis



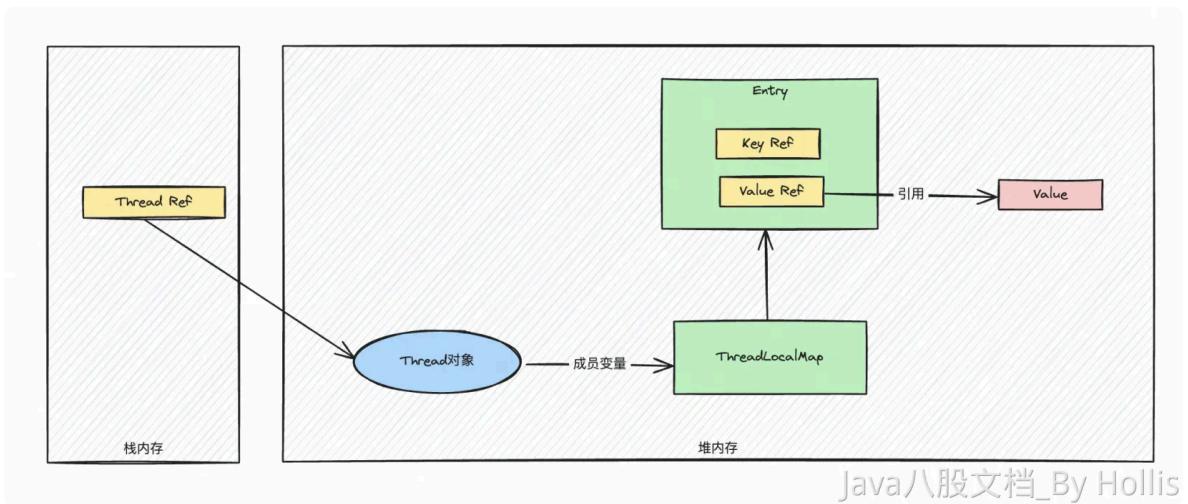
Java八股文档_By Hollis

ThreadLocalMap的key就是ThreadLocal对象，它有两个引用源

- 栈上的ThreadLocal对它的引用

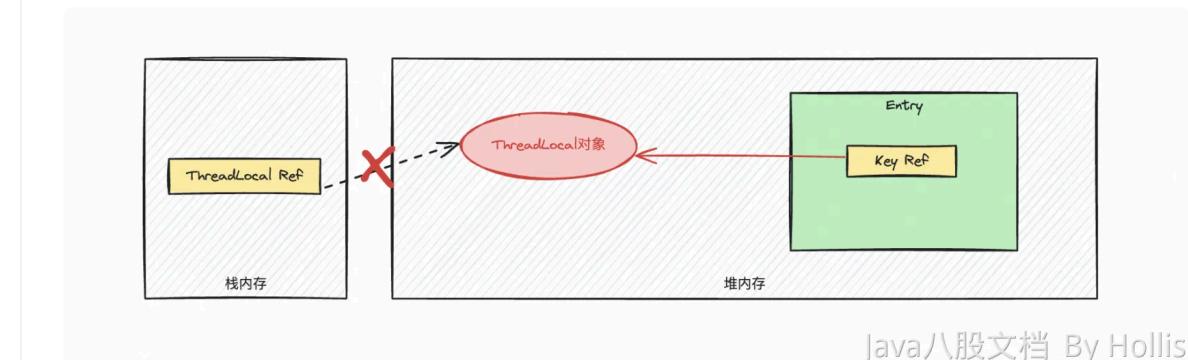
- ThreadLocalMap中的key对它的引用

对于value来说，就一条，就是从Thread对应过来的

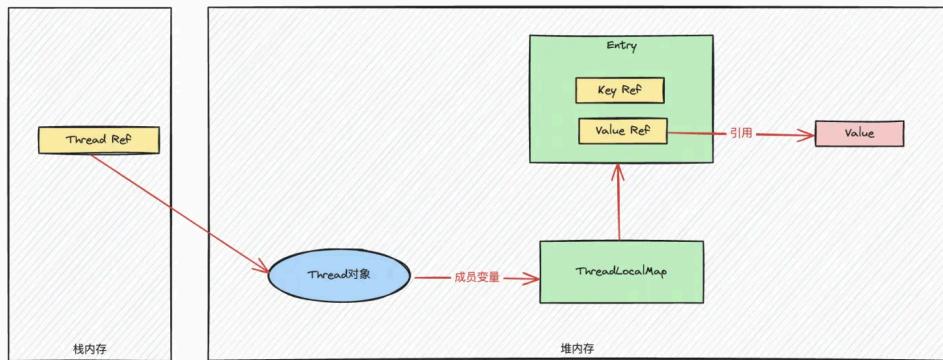


会存在以下两种情况

- 栈上的ThreadLocal Ref引用不再使用了，即方法结束后这个对象引用就不再用了，那么，ThreadLocal对象因为还有一条引用链在，所以就会导致他无法被回收，久而久之可能就会对导致OOM



- Thread对象如果一直在被使用，比如线程池中被重复使用，那么从这条引用链就一直在，那么就会导致ThreadLocalMap无法被回收



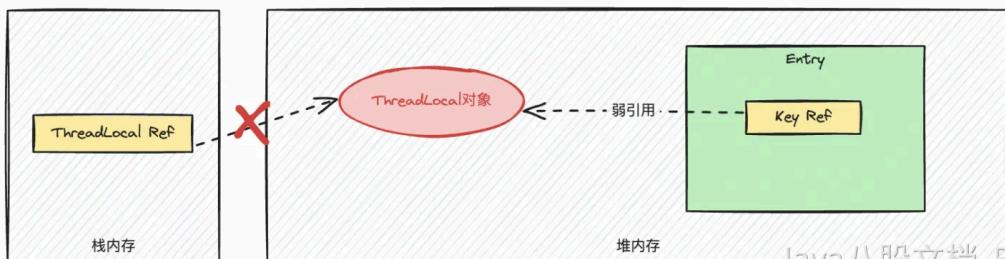
Java八股文档_By Hollis

解决方法

弱引用解决内存泄露

栈上的ThreadLocal Ref引用不在使用了，即方法结束后这个对象引用就不再用了，那么，ThreadLocal对象因为还有一条引用链在，所以就会导致他无法被回收，久而久之可能就会对导致OOM。

为了解决这个问题ThreadLocalMap使用弱引用，当ThreadLocal对应只有一个弱引用时，将在下一次GC时回收掉

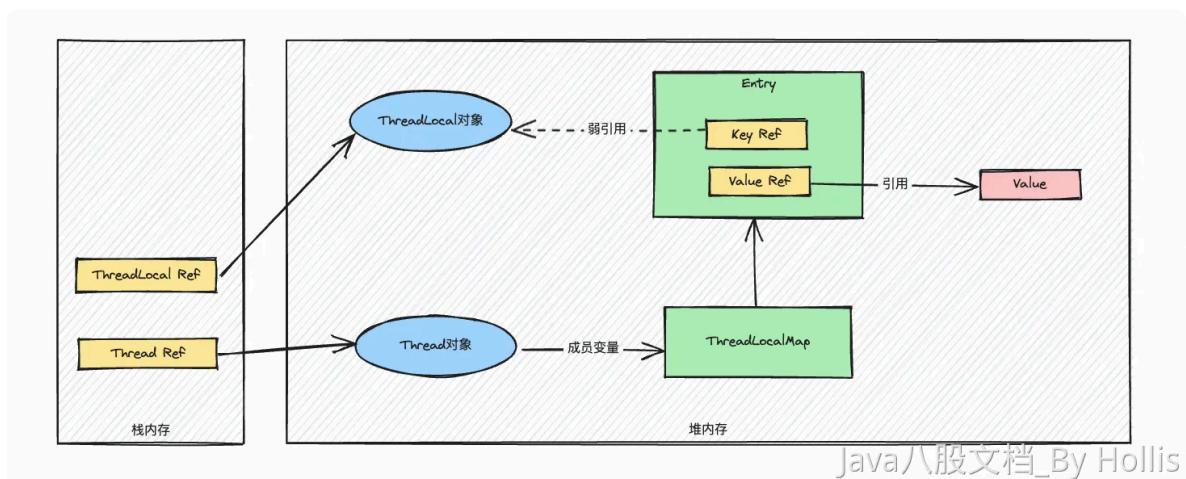


Java八股文档_By Hollis

- 强引用：Java中默认的引用类型，一个对象如果具有强引用那么只要这种引用还存在就不会被回收。比如`String str = new String("Hello ThreadLocal");`，其中str就是一个强引用，当然，一旦强引用出了其作用域，那么强引用随着方法弹出线程栈，那么它所指向的对象将在合适的时机被JVM垃圾收集器回收。
- 软引用：如果一个对象具有软引用，在JVM发生内存溢出之前（即内存充足够使用），是不会GC这个对象的；只有到JVM内存不足的时候才会调用垃圾回收期回收掉这个对象。软引用和一个引用队列联合使用，如果软引用所引用的对象被回收之后，该引用就会加入到与之关联的引用队列中

- 弱引用：如果一个对象只具有弱引用，那么这个对象就会被垃圾回收器回收掉（被弱引用所引用的对象只能生存到下一次GC之前，当发生GC时候，无论当前内存是否足够，弱引用所引用的对象都会被回收掉）。弱引用也是和一个引用队列联合使用，如果弱引用的对象被垃圾回收期回收掉，JVM会将这个引用加入到与之关联的引用队列中。弱引用的对象可以通过弱引用的get方法得到，当引用的对象被回收掉之后，再调用get方法就会返回null
- 虚引用：虚引用是所有引用中最弱的一种引用，其存在就是为了将关联虚引用的对象在被GC掉之后收到一个通知

手动清理ThreadLocal



value那条引用是强引用，也就是说，它的生命周期是和Thread一样的，只要Thread还在（比如线程池中线程被复用），这个对象就无法回收

ThreadLocalMap底层用数组来保存元素，使用“线性探测法”来解决hash冲突的，在每次调用ThreadLocal的get、set、remove等方法的时候，内部会实际调用ThreadLocalMap的get、set、remove等操作

ThreadLocalMap的每次get、set、remove，都会清理key为null，但是value还存在的Entry，具体实现在 `expungeStaleEntry` 函数

所以，当一个ThreadLocal用完之后，手动调用一下remove，就可以在下一次GC时，把Entry清理掉

线程同步的方式

线程同步指多个线程之间按照顺序访问同一个共享资源，避免因为并发冲突导致的问题

- `synchronized`: Java中最基本的线程同步机制，可以修饰代码块或方法，保证同一时间只有一个线程访问该代码块或方法，其他线程需要等待锁的释放

```
1 public synchronized void synchronizedMethod() {  
2     // 同步的代码块  
3 }
```

- ReentrantLock：与synchronized类似，但是更灵活，支持公平锁、可中断锁、多个条件变量等功能

```
1 import java.util.concurrent.locks.ReentrantLock;  
2  
3 private final ReentrantLock lock = new ReentrantLock();  
4  
5 public void someMethod() {  
6     lock.lock();  
7     try {  
8         // 同步的代码块  
9     } finally {  
10        lock.unlock();  
11    }  
12 }
```

- Semaphore：允许多个线程同时访问共享资源，但是限制访问的线程数量。可以用于控制并发访问的线程数量，避免系统资源被过度占用

```
1 import java.util.concurrent.Semaphore;  
2  
3 private final Semaphore semaphore = new Semaphore(2);  
4  
5 public void accessResource() throws InterruptedException {  
6     semaphore.acquire(); // 获取许可  
7     try {  
8         // 访问共享资源  
9     } finally {  
10        semaphore.release(); // 释放许可  
11    }  
12 }
```

- CountDownLatch：允许一个或者多个线程等待其他线程执行完毕后再执行，可以用于线程之间的协调和通信

```
1 import java.util.concurrent.CountDownLatch;  
2  
3 private final CountDownLatch latch = new CountDownLatch(3);
```

```
4
5  public void doWork() throws InterruptedException {
6      // 每个线程完成工作后调用 latch.countDown()
7      latch.countDown();
8      // 主线程等待所有工作线程完成
9      latch.await();
10 }
```

- CyclicBarrier: 允许多个线程在一个栅栏处等待，直至所有线程都到达栅栏位置之后，才会继续执行

```
1 import java.util.concurrent.CyclicBarrier;
2
3 private final CyclicBarrier barrier = new CyclicBarrier(3);
4
5 public void doWork() throws InterruptedException, BrokenBarrierException {
6     // 每个线程执行到 barrier.await() 处等待
7     barrier.await();
8     // 所有线程到达屏障点后同时继续执行
9 }
```

- Phaser: 与CyclicBarrier类似，但是支持更灵活的栅栏操作，可以动态地注册和注销参与者，并可以控制各个参与者的到达和离开

```
1 import java.util.concurrent.Phaser;
2
3 private final Phaser phaser = new Phaser(3);
4
5 public void doWork() {
6     // 线程执行到某个阶段，调用 phaser.arriveAndAwaitAdvance()
7     phaser.arriveAndAwaitAdvance();
8     // 所有线程到达阶段后同时继续执行下一阶段
9 }
```

CountDownLatch、CyclicBarrier、Semaphore的区别？ ..

- CountDownLatch: 是一个计数器，允许一个或者多个线程等待其他线程完成操作。它通常用来实现一个线程等待其他多个线程完成操作之后再继续执行的操作
- CyclicBarrier: 是一个同步屏障，允许多个线程相互等待，直到到达某个公共屏障点，才能继续执行。它通常用来实现多个线程在同一个屏障处等待，然后再一起继续执行的操作

- Semaphore：是一个计数信号量，它允许多个线程同时访问共享资源，并通过计数器来控制访问数量。它通常用来实现一个线程需要等待获取一个许可证才能访问共享资源，或者需要释放一个许可证才能完成操作的操作

死锁是什么，如何解决？

两个或者两个以上的进程（或者线程）在执行过程中，由于竞争资源或者彼此通信而造成的一种阻塞现象。若无外力作用，将无法推进。这些永远在互相等待的进程成为死锁进程

产生死锁的4个必要条件

- **互斥条件**：一个资源每次只能被一个进程使用
- **占有且等待**：一个进程因请求资源而阻塞，对已获得的资源保持不放
- **不可强行占有**：进程已获得的资源，在未使用完成之前，不能强行剥夺
- **循环等待条件**：若干进程之间形成一种头尾相接的循环等待资源关系

如何解除死锁

- 破坏不可抢占：设置优先级，使优先级高的可以抢占资源
- 破坏循环等待：保证多个进程（线程）的执行循序相等即可避免循环等待

java内存模型 (JMM)

java内存模型屏蔽了各种硬件和操作系统的访问差异，保证了java程序在各种平台下对内存的访问都能保证效果的统一

java内存模型规定了所有的共享变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。

不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主内存之间进行数据同步

JMM作用于工作内存和主内存之间的数据同步过程。规定了如何做数据同步以及什么时候做数据同步

JMM是一种规范，目的是解决由于多线程通过共享内存通信时，存在的本地内存数据不一致，编译器会对代码指令重新排序、处理器会对代码乱序执行等带来的问题

多级缓存与一致性问题

- 单线程：cpu核心的缓存只会被一个线程访问。缓存独占，不会出现访问冲突等问题
- 单核cpu，多线程：进程中的多个线程会同时访问进程中的共享内存，CPU将某块内存加载到缓存后，不同线程在访问相同的物理地址时，都会映射到相同的缓存位置，这样即使发生线程的切换，缓存仍然不会失效，并且由于任何时刻只能有一个线程在执行，所以不会出现缓存访问冲突
- 多核cpu，多线程：每个核心只要有一个L1缓存。多个线程访问进程中的某个共享内存，且这多个线程分别在不同的核心上执行，则每个核心都会在各自的cache中保留一份共享内存的缓存。由于多核是可以并行的，坑会出下多个线程同时写各自缓存的情况，那么就有可能导致不同

在多核cpu中，每个核自己的缓存中，关于同一个数据的缓存内容可能不一致 --> 缓存一致性问题

CPU时间片与原子性

因为线程是CPU调度的基本单位，cpu有时间片的概念，会根据不同的调度算法进行线程的调度。所以在多线程场景下就会发生原子性问题。

例如线程在执行一个 读改写 的操作，在执行完 读改 之后，时间片耗尽，就会被要求放弃CPU，并重新进行调度，这种情况写 读改写 就不是一个原子操作

在单线程中，就算不是原子操作也没问题，只需要在下次被调度时，继续执行完就行了，但是多线程场景，是会存在多个线程对同一个共享资源操作的情况

再例如 `i++` 一共有3个步骤，`load`、`add`、`save`。共享变量会被多个线程同时操作，这样 读改写 就不是原子的，如果*i=1*，可能期望结果是3，实际结果是2

并发编程中的原子性和数据库中的原子性一样吗？ --> 不一样

原子性的概念：一个操作中是不可中断的，要么全部执行完成，要么都不执行

数据库事务中，保证原子性是通过事务的提交和回滚，但在并发编程中，是不涉及回滚的。所以并发编程中的原子性，强调的是一个操作的不可分割性

针对并发编程，原子性的定义不和事务中的原子性完全一样，应该定义为：

一段代码或者一个变量的操作，在没有执行完成之前，不能被其他线程执行

指令重排与有序问题

由于处理器优化和指令重排，CPU可能对输入代码进行乱序执行，比如 `Load -> add -> save` 有可能被优化成 `load -> save -> add`。这就是有序性问题

计算机内存模型

为了保证共享内存的正确性（可见性、有序性、原子性），内存模型定义了共享内存系统中多线程程序读写操作行为的规范。通过规范来保证指令执行的正确性，保证了并发场景下的一致性、原子性和有序性

内存模型解决并发问题主要采用两种方式：

- 限制处理器优化
- 使用内存屏障

JMM实现原理

java提供了一系列和并发处理相关的关键字，比如volatile、synchronized、final、concurrent包等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字

原子性

在Java中，为了保证原子性，提供了两个高级的字节码指令 `monitorenter` 和 `monitorexit`。可以通过这两个指令来保证原子性

可见性

Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值的这种依赖主内存作为传递媒介的方式来实现的

`volatile` 关键字提供了一个功能，那就是被其修饰的变量在被修改后，会被立即同步到主内存，被其修饰的变量在每次使用前都从主内存刷新。因为 `volatile` 可以保证多线程操作时变量的可见性

除了volatile，synchronized和final两个关键字也可以，只不过实现方式不同

有序性

在Java中，可以使用 `synchronized` 和 `volatile` 来保证多线程之间操作的有序性。实现方式有所区别：

- `volatile` 关键字会禁止指令重排。
- `synchronized` 关键字保证同一时刻只允许一条线程操作。

JMM和MESI

总线嗅探与总线风暴

synchronized

- 同步方法时隐式的，同步方法的常量池有一个 ACC_SYNCHRONIZED 标志，当某个线程要开始访问的时候，会检查是否有这个标识

如果有，则需要先获得监视器锁 -> 执行方法 -> 执行完释放监视器锁（如果发生了异常，且方法内部没有处理该异常，就在异常被抛到方法外面之前监视器锁会被自动释放）

这个时候如果有其他线程来请求执行方法，会因为无法获取监视器锁而被阻断。

- 同步代码块使用 monitoreenter 和 monitorexit 两个指令实现。

- monitoreenter 指令为加锁

- monitorexit 指令为释放锁

- 每个对象维护着一个被锁次数的计数器，未被锁定计数器为0，当一个线程获得锁（执行 monitoreenter）后，计数器会自增1，当同一个线程再次获得该对象的锁，计数器再自增。当同一个线程释放锁（执行 monitorexit），计数器自减。当计数器为0时，锁被释放，其他线程便可以获取到锁

synchronization 锁特点

- 互斥性**：同一个时间点，只有一个线程可以获得锁，获得锁的线程才可以处理被synchronized修饰的代码片段
- 阻塞性**：只有获得锁的线程才可以执行被synchronized修饰的代码片段，未获得锁的线程只能阻塞，等待锁被释放
- 可重入性**：如果一个线程已经获得锁，在锁未被释放之前，再次请求锁的时候，是必然可以获得锁的

用法

- 同步方法

```
1 //同步方法，对象锁
2 public synchronized void doSth(){
3     System.out.println("Hello World");
4 }
5
6 //同步方法，类锁
7 public synchronized static void doSth(){
8     System.out.println("Hello World");
9 }
```

- 同步代码块

```
1 //同步代码块，类锁
2 public void doSth1(){
3     synchronized (Demo.class){
4         System.out.println("Hello World");
5     }
6 }
7
8 //同步代码块，对象锁
9 public void doSth1(){
10    synchronized (this){
11        System.out.println("Hello World");
12    }
13 }
```

Monitor

java提供了同步机制、互斥锁机制来解决线程安全的问题，这种机制的保障源于监视锁Monitor，每个对象都拥有自己的监视锁。当尝试获得对象的锁的时候，其实是对该对象的Monitor进行操作

什么是Monitor

对应的所有方法都被"互斥"的执行，好比一个Monitor只有一个运行许可，任一线程进入任一方法都需要获得这个许可，离开时再把许可归还

提供signal机制：允许正持有许可的线程暂时放弃许可，等待某个条件变量，当条件成立后，当前进程可以通知正在等待这个条件变量的线程，让它重新去获的运行许可

Monitor代码实现

JVM中Monitor基于C++，由ObjectMonitor实现，主要数据结构

```
1 ObjectMonitor() {
2     _header      = NULL;
3     _count       = 0;
4     _waiters     = 0,
5     _recursions  = 0;
6     _object      = NULL;
7     _owner       = NULL;
8     _WaitSet     = NULL;
9     _WaitSetLock = 0 ;
10    _Responsible = NULL ;
11    _succ        = NULL ;
12    _cxq         = NULL ;
13    FreeNext     = NULL ;
14    _EntryList   = NULL ;
15    _SpinFreq    = 0 ;
16    _SpinClock   = 0 ;
17    OwnerIsThread = 0 ;
18 }
```

其中的几个关键属性

- `_owner`: 指向持有ObjectMonitor对象的线程
- `_WaitSet`: 存放处于 wait 状态的线程队列
- `_EntryList`: 存放处于等待锁 block 状态的线程队列
- `_recursions`: 锁的重入次数
- `_count`: 用来记录该线程获取锁的次数

当多个线程同时访问一段同步代码时，首先会进入 `_EntryList` 队列中，当某个线程获取到对象的 monitor 后，进入 `_owner` 区域并把 monitor 中的 `_owner` 变量设置为当前线程，同时 monitor 的计数器 `_count` +1，即获得对象锁

若持有 monitor 的线程调用 `wait()` 方法，将释放当前持有的 monitor，`_owner` 变量恢复为 null，`_count` -1，同时该线程进入 `_WaitSet` 集合等待被唤醒。若当前线程执行完毕也将释放 monitor 锁并复位变量的值，以便其他线程进入获取 monitor 锁

具体代码实现：[Monitor代码实现](#)

在JDK1.6之前，`synchronized`的实现才会直接调用ObjectMonitor的 `enter` 和 `exit` 方法，这种锁被称为重量级锁

为什么这种方式操作锁很重呢？

- Java的线程是需要映射到操作系统原生线程上的，如果需要阻塞或者唤醒一个线程就需要操作系统的帮忙，这时就需要用户态陷入核心态（状态转换需要花费很多的处理器时间）。对于一些简单的同步块，状态转换消耗的时间可能比用户代码执行的时间还长

synchronized是如何保证原子性、可见性、有序性的？

原子性问题：线程1获得CPU时间片开始执行，但是执行过程中，CPU时间片耗尽，需要让出CPU，线程2获得了时间片开始执行，但是对于线程1来说，它的操作并没有执行完，也没有全都不执行

如何保证原子性：

通过 `monitorenter` 和 `monitorexit` 两个字节码指令来实现

当线程执行到 `monitorenter` 的时候要先获得锁，才能执行后面的方法。当线程执行到 `monitorexit` 的时候则要释放锁。在未释放之前，其他线程是无法再次获得锁的。

注意：线程1执行 `monitorenter` 指令的时候，会对Monitor进行加锁，加锁后其他线程无法获得锁，除非线程1主动解锁。即使在执行过程中，线程1CPU时间片用完，但是由于没有主动解锁，下一个时间片还是只能被线程1获取到，直到所有代码执行完。保证了原子性

如何保证有序性：

有序性即程序执行的顺序按照代码的先后顺序来执行

CPU可能对输入的代码进行乱序执行，比如load->add->save可能被优化成load->save->add。这就可能导致有序性问题

最好的办法就是：直接禁止指令重排和处理器优化。**单靠synchronized是做不到的**。这里需要扩展一下有序性的概念。

深入JAVA虚拟机关于有序性原句：**如果在本线程内观察，所有操作都是天然有序的。如果在一个线程中观察另一个线程，所有操作都是无序的。**

这里其实和as-if-serial语义有关：不管怎么重排序，单线程程序的执行结果都不能被改变。编译器和处理器无论如何优化，都必须遵守这个语义。

as-if-serial语义保证了单线程中，指令重排是有一定限制的，而只要编译器和处理器遵守了这个语义，就可以认为单线程程序是按照顺序来执行的。实际上还是有指令重排，只不过我们无须关心这种重排的干扰。

由于synchronized修饰的代码，同一时间只能被同一线程来访问，也就是单线程执行的，所以可以保证有序性

如何保证可见性 .

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，另一个线程能够立刻看到修改的值

JMM限制了线程对变量的所有操作都必须在工作内存中操作主内存的副本拷贝，而不能直接读写主内存。

不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递需要自己的工作内存和主内存之间进行数据同步。所以就可能出现线程1修改了某个变量的值，但是线程2不可见。

为了保证可见性，有一条规则是：对一个变量解锁之前，必须先把此变量同步回主内存中。这样解锁后，后续的线程就可以访问到被修改后的值。

所以被synchronized锁住的对象，在解锁后，其值就具有可见性

synchronized的锁升级过程 .

- JDK1.6及之前的班嗯，synchronized锁是通过对象内部的监视器锁（对象锁）来实现
 - 当一个线程请求对象锁时，如果该对象没有被锁住，则会获取锁并继续执行
 - 如果该对象已经被锁住，则会进入阻塞状态，直到锁被释放
- 这种锁被称为"重量级锁"，因为锁的获取和释放都需要在操作系统上来进行线程的阻塞和唤醒，会带来很大的开销
- JDK1.6之后，引入了"偏向锁"（JDK15中，废弃了偏向锁（<https://openjdk.org/jeps/374>））、"轻量级锁"、"重量级锁"三种锁状态

JAVA中，锁的状态分为4种，mark word的低两位表示锁的状态

- 无锁状态：01
- 偏向锁状态：01
- 轻量级锁状态：00
- 重量级锁状态：10

Hotspot的实现

锁状态	25位	31位	1位	4bit	1bit 偏向锁位	2bit 锁标志位
无锁态 (new)	unused	hashCode (如果有调用)	unused	分代年龄	0	0 1
锁状态	54位	2位	1位	4bit	1bit 偏向锁位	2bit 锁标志位
偏向锁	当前线程指针 JavaThread*	Epoch	unused	分代年龄	1	0 1
锁状态	62位				2bit 锁标志位	
轻量级锁 自旋锁 无锁	指向线程栈中Lock Record的指针				0	0
重量级锁	指向互斥量（重量级锁）的指针				1	0
GC标记信息	CMS过程用到的标记信息				Java八股文档	By Hollis

无锁

当一个线程第一次访问这个对象的同步块时，JVM会在对象头中设置该线程的Thread ID，并将对象头的状态位设置为“偏向锁”(表示对象当前偏向第一个访问它的线程)。

偏向锁

在偏向锁模式下，锁会偏向于第一个获取它的线程，JVM会在对象头中记录该线程的ID作为偏向锁的持有者，并将对象头中的Mark Word中的一部分作为偏向锁标识

这种情况下，如果其他线程访问该对象，会先检查该对象的偏向锁标识，如果和自己的线程ID相同，则直接获取锁。如果不同，则该对象的锁状态会升级到重量级锁状态。

触发条件：首次进入synchronized块自动开启，假设JVM启动参数没有禁用偏向锁。

轻量级锁

当有另一个线程尝试获取一个偏向锁，偏向锁会升级为轻量级锁

轻量级锁状态，JVM为对象头中的Mark Word预留了一部分空间，用于存储指向线程栈中锁记录的指针

当一个线程尝试获取轻量级锁时，JVM做法：

- 将对象头中的Mark Word复制到线程栈中的锁记录（Lock Record）：每个Java对象头部都有一个Mark Word，它用于存储对象自身的运行数据（如哈希码、锁状态信息、代年龄等）。当线程尝试获取轻量级锁时，JVM会在当前线程的栈帧中创建一个锁记录空间，然后将对象头中的Mark Word复制到这个锁记录中。这个复制的Mark Word被称为“Displaced Mark Word”

- 尝试通过CAS操作更新对象头的Mark Word: JVM尝试使用CAS操作，将对象头的Mark Word更新为指向锁记录的指针。如果这个更新操作成功，那么这个线程就成功获取了这个对象的轻量级锁

如果替换成功，则该线程获取锁成功；如果失败，则表示已经有其他线程获取到了锁，则该锁会升级为重量级锁

触发条件：当有另一个线程尝试获取偏向锁时，偏向锁会自动升级为轻量级锁

为什么需要将对象头中的Mark Word复制到线程栈中？

主要原因是**为了保留对象的原始信息**，复制Mark Word到线程栈中是为了在锁释放时能够恢复对象头的原始状态。因为锁的获取与释放是成对出现的，所以在锁释放时，JVM需要这份复制的原始Mark Word来恢复对象头，确保对象状态的正确性

重量级锁

当轻量级锁的CAS操作失败时，即出现了实际的竞争，锁会进一步升级为重量级锁

当锁状态升级为重量级锁，JVM会将对象头中的Mark Word修改为指向一个重量级锁结构 `Monitor`，该结构包含一个 `Entry Set`，用于管理那些尝试获取锁但暂时无法获得的线程

`Entry Set` 是重量级锁 `Monitor` 的一部分，用来存放那些**获取锁但是未成功的线程**。这些线程处于 `BLOCKED` 状态。当持有锁的线程释放锁时，JVM会从 `Entry Set` 中选择一个线程唤醒，并将线程状态设置为 `READY`，然后等待该线程重新获取该对象的锁

触发条件：当轻量级锁的CAS操作失败，轻量级锁升级为重量级锁

锁能降级吗？

锁降级如果是指锁从重量级状态回退到轻量级或偏向锁状态的过程，那么可以明确的说，当前的 HotSpot虚拟机实现是不支持的

锁一旦升级为重量级锁，那么就会保持在这个状态，直到锁被完全释放

但是，如果说，一旦一个锁从偏向、到轻量级锁、再到重量级锁加锁之后，后面的所有加锁都是以重量级锁的方式加锁了，这么说也不对！

因为有一种特殊的降级，就是重量级锁的Monitor对象不再被任何线程持有的过程。这一过程可以在STW暂停期间进行，这时所有Java线程都停在安全点（SafePoint），这个过程会做以下事情：

- 锁状态检查：在STW期间，JVM会检查所有的Monitor对象
- 确定降级对象：JVM识别出那些没有被任何线程持有的Monitor对象。通常是通过检查Monitor对象的锁计数器或者所有权信息来实现

- 降级操作：对于那些确定未被使用的Monitor对象，JVM会进行所谓的deflation操作，即清理这些对象的状态，使其不再占用系统资源。在某些情况下，这可能涉及到重置Monitor状态，释放与其相关的系统资源等。

以上，仅限于HotSpot虚拟机，像JRockit虚拟机就支持从重量级锁降级到轻量级锁

JRockit锁降级说明：当最后一个争用线程释放重量级锁时，锁通常仍然保持为重量级。即使没有争用，获取重量级锁的代价也比获取轻量级锁（thin lock）更高。如果JRockit认为锁会从变轻中受益，它可能会再次将其“压缩”为轻量级锁当最后一个争用线程释放重量级锁时，锁通常仍然保持为重量级。即使没有争用，获取重量级锁的代价也比获取轻量级锁（thin lock）更高。如果JRockit认为锁会从变轻中受益，它可能会再次将其“压缩”为轻量级锁

synchronized的重量级锁很慢，为什么还需要？

synchronized本身是一个重量级锁，是为了让并发不那么高的情况下，不要那么重才引入的偏向锁和轻量级锁

重量级锁本身是有必要的，因为并发抢锁时，只有阻塞才能保证线程对共享资源的安全访问，能避免数据竞争的不一致性问题

同时在锁竞争非常激烈的情况下，轻量级锁的自旋会导致CPU资源浪费，重量级锁时通过阻塞挂起来避免这些资源浪费。对于任务较重的操作，重量级锁能提高系统稳定性，避免性能因频繁自旋而下降