第 3 章 多线程

可以负责任地说，多线程的开发能力可以不用，但不能没有，因为它既是衡量 Java 工程师能力的主要标准之一，又是应用程序性能的托底保障。而且多线程唯一可以确定的就是它的运行结果无法确定，这也正是它最为有趣的地方。

本章结合笔者实际工作经验，讲解了 Tread 和 Runnable 的区别、常用 Thread API、线程关键字、线程池、CAS 原子操作、锁与 AQS 等内容。

3.1 正确认识多线程

知道多线程的工程师不少，但真正了解它的却不多。笔者并非妄自菲薄认为自己懂得比别人多，而是从自身实际开发经验出发，与读者分享一些自身对多线程的认识，也同时发现自身的不足。

3.1.1 混乱的生命周期

关于线程生命周期的问题，很多资料都不统一，例如，在线程生命周期的状态上就有不同的认识。

1. 有说四种的：NEW、RUNNABLE、BOLCKED、TERMINATED；
2. 有说五种的：NEW、RUNNABLE、RUNNING、BOLCKED、DEAD；
3. 有说六种的：NEW、RUNNABLE、WAITING、TIMED_WAITING、BOLCKED、TERMINATED。

笔者认为，不管四种、五种还是六种，都只是“纸面”上的猜测。“Talk is cheap， show me the code”，只有通过代码来看才是最直观准确的。

代码清单 3-1 StandardThreadState.java

```
public class StandardThreadState implements Runnable {
    ......

    public static void main(String[] args) {
        StandardThreadState state = new StandardThreadState();
        Thread thread1 = new Thread(state);
        Thread thread2 = new Thread(state);
        System.out.println(thread1.getState());// NEW
        thread1.start();
        System.out.println(thread1.getState());// RUNNABLE
        thread2.start();
        try {
            TimeUnit.MILLISECONDS.sleep(100);
            System.out.println(thread1.getState());// TIMED_WAITING
```

```
            System.out.println(thread2.getState());// BLOCKED
            TimeUnit.MILLISECONDS.sleep(2000);
            System.out.println(thread1.getState());// TERMINATED
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

执行上述代码后，打印结果如下：

```
NEW
RUNNABLE
TIMED_WAITING
BLOCKED
TERMINATED
```

结果显示有五种。把代码稍加改动，然后再运行试试。如代码清单 3-2 所示。

代码清单 3-2 MoreThreadState.java

```
public class MoreThreadState implements Runnable {
    private synchronized void sync() {
        try {
            TimeUnit.SECONDS.sleep(2);
            wait();
    ......
}
```

再次执行，打印结果如下：

```
NEW
RUNNABLE
TIMED_WAITING
BLOCKED
WAITING
TIMED_WAITING
```

把两种结果合并一下，那么线程的生命周期就有六种：NEW、RUNNABLE、TIMED_WAITING、BLOCKED、WAITING、TERMINATED，并没有所谓的 RUNNING、READY 和 DEAD 这几种状态。

事实上，线程所有的状态都已在 Thread 类的 State 枚举中定义过了，根据代码运行结果得到的线程生命周期状态如图 3-1 所示。

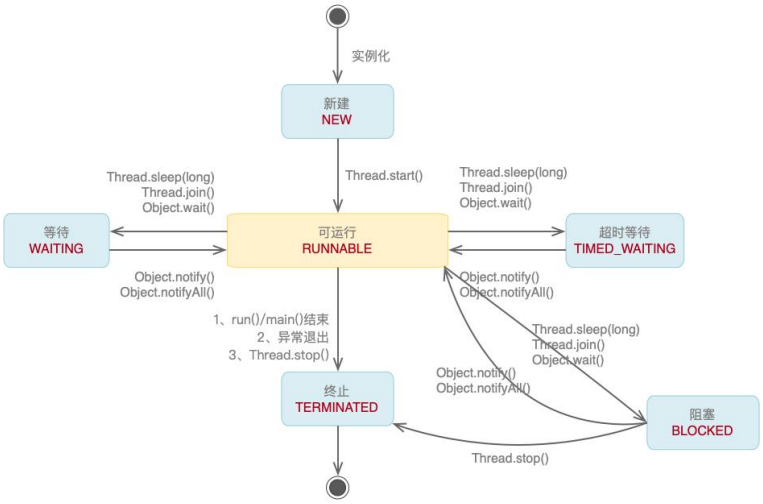

图 3-1 线程生命周期状态图

这个线程生命周期状态图就正确了吗？不一定，后续还会回过头来审视它。

3.1.2 Thread 与 Runnable

在多线程里用得最多的就是 Thread 类和 Runnable 接口了，但其实它们还有两个误区。

误区一：“有两种创建线程的方式，或者继承 Thread 类并重写 run()方法，或者实现 Runnable 接口”。虽然 Thread 也实现了 Runnable 接口，但只有 Thread 类才能真正创建并通过 start()方法启动线程，而 Runnable 接口是没有这个能力的。

准确地说，定义线程所要执行的 run()方法有两种：

1. 继承并重写 Thread 的 run()方法；
2. 实现 Runnable 接口的 run()方法，并将 Runnable 作为 Thread 或其子类的构造器参数。

误区二：“Thread 类和 Runnable 接口的 run()方法效果是一样的”，真的吗？

代码清单 3-3 SubThread.java

```
public class SubThread extends Thread {
    private int i = 1;

    @Override
    public void run() {
        while (i <= 10) {
            System.out.println("当前线程: " + Thread.currentThread() + " - " + i++);
        }
    }

    public static void main(String[] args) {
        SubThread subThread1 = new SubThread();
```

```
        SubThread subThread2 = new SubThread();
        subThread1.start();
        subThread2.start();
    }
}
```

代码清单 3-4 ExtractMethodRunnable.java

```
public class ExtractMethodRunnable implements Runnable {
    private int i = 1;

    @Override
    public void run() {
        while (i <= 10) {
            System.out.println("当前线程: " + Thread.currentThread() + " - " + i++);
        }
    }

    public static void main(String[] args) {
        ExtractMethodRunnable extractMethod = new ExtractMethodRunnable();
        Thread thread1 = new Thread(extractMethod);
        Thread thread2 = new Thread(extractMethod);
        thread1.start();
        thread2.start();
    }
}
```

1. 代码清单 3-3 中的两个子线程 subThread1 和 subThread2 分别单独从 1 打印到 10，因为 Thread 类的 run()方法是不能共享的，也就是说 A 线程不能把 B 线程的 run()方法当作自己的执行单元；

2. 而代码清单 3-4 利用 Runnable 接口让两个线程共同完成从 1 到 10 的打印。

这说明 Runnable 才是真正的多线程，用它封装功能代码，比直接用 Thread 类更好。

3.2 常见的 Thread API

线程生命周期中各种状态之间的互相转换，是通过一系列的方法来完成的，俗称 Thread API。这些方法有的由 Object 类提供，有的由 Thread 类实现；有的是实例方法，有的是静态方法。

3.2.1 等待（wait）与通知（notify / notifyAll）

Java 从一开始就非常重视多线程并给其提供了良好的支持，这种重视直接反映在顶级类 Object 的两个方法之上：wait(int)和 notify() / notifyAll()。先看看代码清单 3-5 的执行结果。

代码清单 3-5 WaitNotify.java

```
public class WaitNotify {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("t1  等待");
            synchronized ("锁") {
                System.out.println("t1  开始");
                try {
                    "锁".wait();// t1 进入等待队列并释放锁
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("t1  结束");
            }
        });
        Thread t2 = new Thread(() -> {
            System.out.println("t2  等待");
            synchronized ("锁") {
                System.out.println("t2  开始");
                try {
                    "锁".notifyAll();// 通知其他所有线程（t1 和 t3）进入同步队列
                } catch (Exception e) {
                    e.printStackTrace();
                }
                System.out.println("t2  结束");
            }
        });
        Thread t3 = new Thread(() -> {
            System.out.println("t3  等待");
            synchronized ("锁") {
                System.out.println("t3  开始");
                try {
                    TimeUnit.MILLISECONDS.sleep(100);
                    "锁".wait();// t3 进入等待队列并释放锁
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                System.out.println("t3  结束");
```

```
            }
        });
        t1.start();
        t2.start();
        t3.start();
    }
}
```

反复执行代码清单 3-5 会发现这么一个规律：

1. 如果 t2 抢到锁并在 t3 之前执行，那么一定会有线程处于等待状态，如图 3-2 所示；

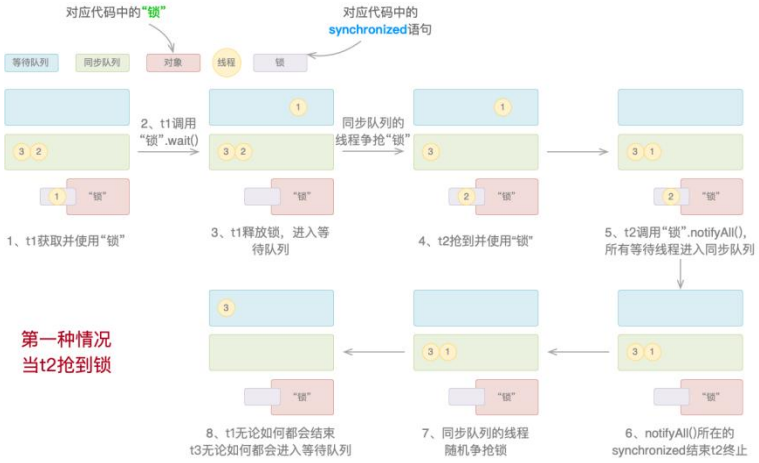

图 3-2 当 t2 先抢到锁时

2. 如果 t3 抢到锁并在 t2 之前执行，那么所有线程就都可以结束，如图 3-3 所示。

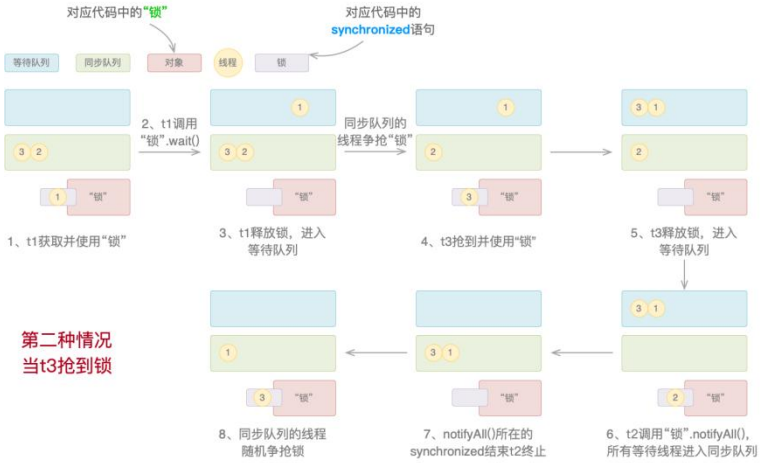

图 3-3 当 t3 先抢到锁时

这里面涉及到两个概念：线程的等待队列和同步队列。

1. 线程等待队列：存放处于 WAITING 或 TIMED_WAITING 状态的线程，在其中排队等待被唤醒或自动醒来（之后再进到同步队列）；

2. 线程同步队列：存放处于 RUNNABLE 可运行状态的线程，在其中排队等待执行。

如果 t2 先抢到锁，在线程任务执行完成后立即调用 notifyAll()方法。因为此时 t1 处于等待队列，而 t3 还未抢到锁，所以执行完 notifyAll()方法后 t1 和 t3 共同处于同步队列。当 t2 终止时，t1 和 t3 争抢锁，不管它们之中哪个抢到锁，t3 都会进入等待队列，而 t1 一定会执行完。

如果 t3 先抢到锁，在线程任务执行完成后立即调用 wait()方法。故此时 t1 和 t3 都处于等待队列。当唯一一个处于同步队列的线程 t2 调用 notifyAll()方法后，t1 和 t3 全部进入同步队列并争抢锁，不管它们之中哪个先抢到，t1 和 t3 都会执行完。

理解了代码清单 3-5 也就理解了 wait()和 notify() / notifyAll()的机制。不管再增加多少个线程，本质上都是一样。

顺便说一句 wait(int)和 wait()的区别：一个可自动醒来，一个要 notify() / notifyAll()唤醒。

3.2.2 插队（join）与谦让（yield）

join(int)和 yield()都是 Thread 类的方法，只不过 join(int)是实例方法，yield()是静态方法。

join(int)的意思如果用大白话说就是强行插队，如图 3-4 所示。

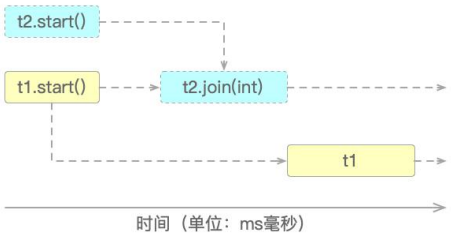

图 3-4 执行 join(int)方法的线程可以强行插队

上图显示，t1 和 t2 同时启动，而当 t2 执行 join(int)后，就会强行让**其他线程**等待自己执行完之后再开始执行。正如代码清单 3-6 所示。

代码清单 3-6 JoinThread.java

```
public class JoinThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("thread1");
        });
        Thread t2 = new Thread(() -> {
            ......
        });
        t1.start();
        t2.start();
        try {
            t2.join(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

执行上述代码后，打印结果如下：

```
thread2
thread1
flag =
```

为什么 flag 没有值呢？原因如图 3-5 和图 3-6 所示那样。

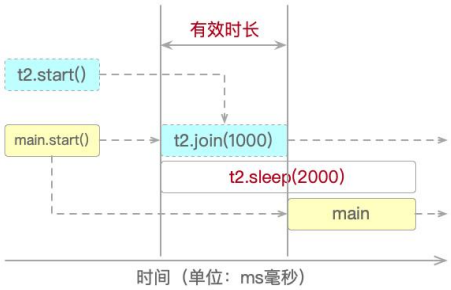

图 3-5 join(int)时间 < sleep()时间

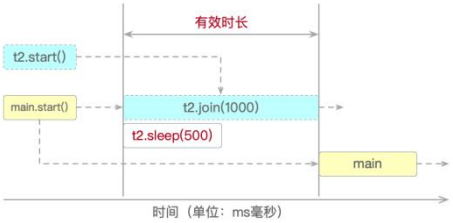

图 3-6 join(int)时间 > sleep()时间

1. 当 join(int)时间 < sleep()时间，t2 执行 join(int)时间后 main 马上接着执行，不等 sleep()结束，相当于 sleep()被截断了；

2. 当 join(int)时间 > sleep()时间，t2 执行时间以 join(int)为准，之后 main 接着执行；

3. 当 join(0)时，则 main 要无限期等待，直到 t2 执行结束。

在代码清单 3-6 中有一点务必要清楚：join()方法并不会使所有线程暂停，而是使调用 join()方法的线程暂停，具体来说就是 main 主线程调用了 join()方法，所以连同它里面的 t1 线程也被一并暂停了，而并不是 t2 让 t1.start()暂停了。所以，上面所说的“其他线程”指的其实是 main 线程，这里的 t1 只是个龙套而已。

至于 Thread.yield()，则是一种非强制的方法，它会告诉线程调度器自愿放弃当前的 CPU 资源，但也仅仅只是“告诉”，执不执行就无法预知了，它后续可能依然会参与到对 CPU 资源的争夺中，且调用 Thread.yield()方法的线程并不会释放锁。它的作用如代码清单 3-7 所示。

代码清单 3-7 YieldThread.java

```
public class YieldThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("t1  开始执行");
            Thread.yield();
            System.out.println("t1  执行结束");
        });
        Thread t2 = new Thread(() -> {
            System.out.println("t2  开始执行");
            try {
                TimeUnit.MILLISECONDS.sleep(1000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println("t2  执行结束");
        });
        t1.start();
        t2.start();
    }
}
```

执行结果可能是下面的任意一种：

```
t1  开始执行
t2  开始执行
t1  执行结束
t2  执行结束
```

```
t1  开始执行
t1  执行结束
t2  开始执行
t2  执行结束
```

```
t2  开始执行
t1  开始执行
t1  执行结束
t2  执行结束
```

```
t2  开始执行
t2  执行结束
t1  开始执行
t1  执行结束
```

从结果可以看出，Thread.yield()的“谦让”也只是“说说而已”。

3.2.3 休眠（sleep）与打断（interrupt）

自 JDK 1.5 引入 TimeUnit.[X].sleep()之后（[X]表示具体时间单位），Thread.sleep()就不再被建议使用了。因为 TimeUnit 更加强大且优雅。例如，如果想休眠 5 天 4 小时 3 分 2 秒 1 毫秒，用 Thread.sleep()该怎么办？——难办！但用 TimeUnit.[X].sleep()就很好解决。

```
TimeUnit.DAYS.sleep(5);
TimeUnit.HOURS.sleep(4);
TimeUnit.MINUTES.sleep(3);
TimeUnit.SECONDS.sleep(2);
TimeUnit.MILLISECONDS.sleep(1);
```

是不是直观清晰得多？现在用 interrupt()验证线程生命周期的状态转换。因为代码占用篇幅较多故作简略，完整代码在 cn.javabook.chapter03.threadapi.SleepInterrupt。

SleepInterrupt 执行后，打印结果如下所示：

```
Thread-0 - 0
Thread-0 - 1
Exception in thread "Thread-0" ...: sleep interrupted
    ...
Caused by: java.lang.InterruptedException: sleep interrupted
    at java.base/java.lang.Thread.sleep(Native Method)
    ... 1 more
t1 线程状态：TIMED_WAITING
t1 打断状态：false
Exception in thread "Thread-1" ...: java.lang.InterruptedException
    ...
Caused by: java.lang.InterruptedException
    at java.base/java.lang.Object.wait(Native Method)
    ... 1 more
t2 线程状态：WAITING
t2 打断状态：false
t3 线程状态：RUNNABLE
t3 打断状态：true
```

可以清楚地看到 t1、t2 和 t3 的不同线程状态和打断状态。至于原因的分析，就留给读者吧。

3.2.4 常用方法的比较

现在稍微总结一下前面已介绍过的 Thread API，如表 3-1 所示。

表 3-1 常见 Thread API 的比较

常见方法	作用	锁	方法类型	所属对象
wait(int)	等待指定时间或一直等待下去	立即释放	实例方法	Object
notify()	随机唤醒某个线程	不立即释放	实例方法	Object
notifyAll()	唤醒全部线程	不立即释放	实例方法	Object
join(int)	强制某个线程插队执行	不释放锁	实例方法	Thread
yield()	使当前线程让出 CPU 资源，但有可能会立即又抢回	不释放锁	静态方法	Thread
sleep(int)	是某个线程休眠指定的时间	不释放锁	静态方法	Thread TimeUnit

interrupt()	打断处于 wait、sleep 的线程或调用 interrupt 的线程	不立即释放	实例方法	Thread

在搞清楚了线程生命周期及其状态转换的方法之后，再回过头来重新审视一下图 3-1。可以把它调整为图 3-7 所示的内容。

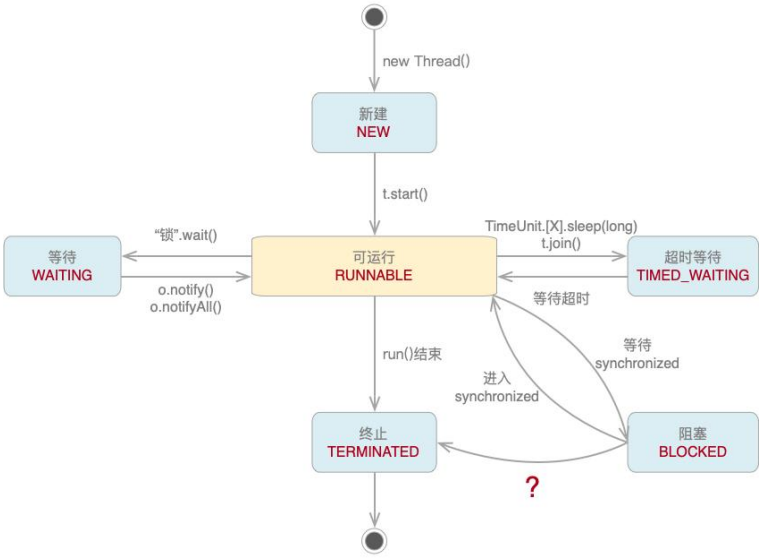

图 3-7 调整后的线程生命周期状态图

图 3-7 去掉了图 3-1 中全部未验证过的内容，更加精简。限于篇幅、经验与时间，难免出现漏误。但只要能起到抛砖引玉的作用，那么它也还算有一点点价值了。

3.3 线程关键字

Java 用于修饰线程的关键字有两个：volatile 和 synchronized。相对于 synchronized，volatile 更加轻量。volatile 用于修饰变量，而 synchronized 则主要用于修饰方法和代码块。

3.3.1 Java 内存模型 JMM

因为计算机的缓存结构直接决定了 Java 内存模型（Java Memory Model，JMM）及依附于其上的多线程机制，所以对计算机的内存结构、JMM 机制和它们之间的关系稍做了解就很有必要，如图 3-8 和图 3-9 所示。

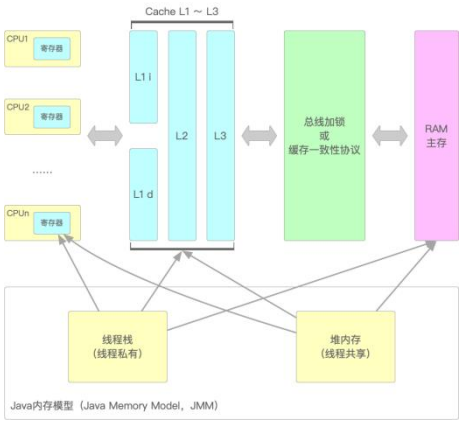

图 3-8 计算机缓存结构和 JMM 的关系

从图 3-8 可以知道：

1. 由于 CPU 和 RAM 速度的不对等，因此在它们之间增加了高速缓存。目前多数的计算机都是三级缓存结构，最靠近 CPU 的缓存称为 L1（L1 又分为 L1 instruction 和 L1 data），然后是 L2，L3；

2. 缓存解决了速度不匹配的旧问题，但带来了数据不一致的新问题。例如在 RAM 中 i=1，但由于 CPU 执行了 i++的操作，因此到了 Cache 中 i=2，这样就导致 RAM 和 Cache 的数据出现了不一致现象，现在主流的解决方案就是缓存一致性协议；

3. JMM 是用来解决 JVM 与不同计算机平台之间的协作问题的。它定义了线程、缓存和 RAM 之间的抽象关系。

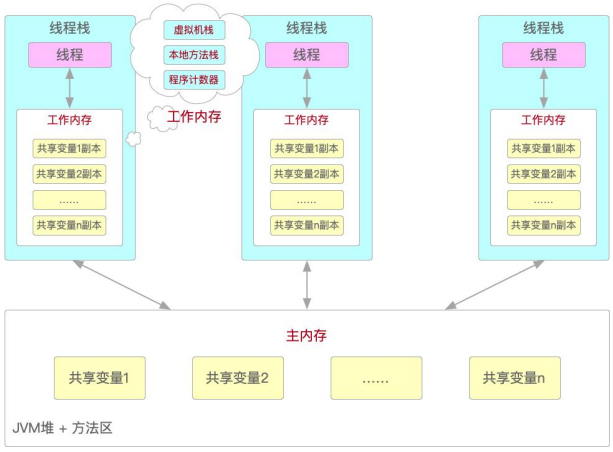

图 3-9 JMM 机制概述

从图 3-9 可以知道：

1. JVM 堆和栈的关系，有些类似于缓存和缓冲之间的关系。JVM 堆中存放共享数据、对象实例和数组等引用类型；而栈中存放线程局部变量、操作数、动态链接等数据；

2. 当线程工作时，因为无法直接操作主内存及其他线程工作内存中的数据，所以会把主内存中的数据拷贝一份到自己的工作内存中，之后再将操作完毕的数据保存到主内存。

通过以上两张图，就把计算机缓存结构和 JMM 的关系，以及 JMM 中线程的工作机制大致讲清楚了。

3.3.2 volatile：你是我的眼

正如数据库的 CAP 定理无法同时兼顾事务一致性、可用性和分区容错性三者一样，volatile 也是三选二，即在并发编程的原子性、有序性、可见性中，只保证有序性和可见性，而不保证原子性。

所谓有序性，对应的是 CPU 的指令重排（这个了解就行）。所谓可见性，是当主内存中存在共享变量 i = 0 时，线程 1、线程 2 和线程 3 会分别将 i 拷贝到自己的工作内存中。显然三个线程执行后 i 的值将完全不同。如果将变量 i 用 volatile 关键字修饰，那么当某个线程修改 i 的值后，其他的线程就立即可以知道 i 的最新值，这就是可见性的意义：变量值的变化对所有线程都“可见”。代码清单 3-8 很好地演示了这个特性。

代码清单 3-8 VolatileVisibility.java

```
public class VolatileVisibility {
    static int init = 0;
    public static void main(String[] args) {
        new Thread(() -> {
            int local = init;
            while (local < 5) {
                if (init != local) {
                    System.out.println("Reader 线程读取 init 值为： " + init);
                    local = init;
                }
            }
        }, "Reader").start();
        new Thread(() -> {
            int local = init;
            while (local < 5) {
                System.out.println("Writer 线程将 init 赋值为： " + ++local);
                init = local;
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "Writer").start();
    }
}
```

执行上述代码后，打印结果如下：

```
Writer 线程将 init 赋值为： 1
```

```
Writer 线程将 init 赋值为：2
Writer 线程将 init 赋值为：3
Writer 线程将 init 赋值为：4
Writer 线程将 init 赋值为：5
```

结果显示，Reader 完全没有反应，而且 Writer 线程终止以后，主线程依旧不能结束。这是因为 Reader 无法读取到 init 的变化而陷入了 local < 5 的死循环。

如果以 volatile 关键字修饰变量 init，就可以实现预期效果。但它需要具备两个条件：

1. 对变量的写操作不依赖于当前值，不会出现 x++或 x = x + y 这样的“自我赋值”语句；
2. 同一条语句中不会出现多个变量：if(x > y) { // dosomething; } 因为它不是线程安全的。

3.3.2 synchronized：幽灵捕手

synchronized 关键字最经典的例子就是解决头疼的“超卖”、“吞票”和“幽灵票”现象。所谓“超卖”指的是售出数量超出原定总量；“吞票”就是不管售票数量是否正确，都会出现缺少某张票号的情况；“幽灵票”则是出现了根本不存在的票号，而且这些问题一般都会同时出现。如代码清单 3-9 所示。

代码清单 3-9 SynchronizedThread.java

```
public class SynchronizedThread implements Runnable {
    private static int tickets = 10;

    @Override
    public void run() {
        while (true) {
            if(tickets > 0) {
                System.out.println(Thread.currentThread().getName() + " 正在出售 " +
(--tickets) + " 号车票");
                try {
                    TimeUnit.MILLISECONDS.sleep(100);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            } else {
                System.out.println(Thread.currentThread().getName() + " 车票已售完");
                break;
            }
        }
    }

    public static void main(String[] args) {
        SynchronizedThread window = new SynchronizedThread();
```

```
        Thread thread1 = new Thread(window, "一号窗口");
        Thread thread2 = new Thread(window, "二号窗口");
        Thread thread3 = new Thread(window, "三号窗口");
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

执行上述代码后，打印结果如下（每次结果都可能会略有不同）：

```
三号窗口 正在出售 7 号车票
一号窗口 正在出售 9 号车票
二号窗口 正在出售 8 号车票
三号窗口 正在出售 6 号车票
二号窗口 正在出售 6 号车票
一号窗口 正在出售 6 号车票
三号窗口 正在出售 4 号车票
二号窗口 正在出售 3 号车票
一号窗口 正在出售 3 号车票
一号窗口 正在出售 1 号车票
三号窗口 正在出售 2 号车票
二号窗口 正在出售 1 号车票
二号窗口 正在出售 0 号车票
一号窗口 正在出售 -1 号车票
三号窗口 正在出售 0 号车票
三号窗口 车票已售完
二号窗口 车票已售完
一号窗口 车票已售完
```

结果显示：

1. “超卖”：总共只有 10 张票但却卖了 15 张，0 号、1 号、3 号、6 号被卖了多次；
2. “吞票”：5 号车票一直没有出现，仿佛被吞掉了；
3. “幽灵票”：居然出现了不应该出现的-1 号车票。

可以想像，如果这是高铁售票系统，会出什么样的乱子。

因为这并非某个变量的可见性问题，而是方法或者代码块被多个线程“共享”的问题。将代码清单 3-9 稍加修改，让核心部分不被线程共享就行了。

```
public class SynchronizedThread implements Runnable {
    ......
    @Override
    public void run() {
```

```
            synchronized ("锁") {
                while (true) {
                    ......
                }
            }
        }
        ......
}
```

现在，无论这段代码再运行多少次都是正常的——“幽灵”再也不会出现了。

最后，对 volatile 和 synchronized 的特性稍作总结，如表 3-2 所示。

表 3-2 volatile 和 synchronized 的比较

比较项	volatile	synchronized
作用范围	实例变量、静态变量	方法、代码块
并发特性保障	保障可见性、有序性	保障可见性、有序性、原子性
是否阻塞	不会造成阻塞	会造成阻塞
性能量级	轻量级，开销较小	重量级，需加锁、解锁额外操作，可能会影响效率
适用条件	适用于状态标记量的设置，且无其他变量参与	需要持有锁，和对象锁或类锁无关

3.4 线程池

老司机们都知道，对燃油车来说，真正费油的并不是行驶旅程的长短，而是长期的怠速、频繁刹车和猛踩油门等动作。因为在换挡的过程中，发动机要额外多做一些工作，当然就要多费一些油。同样，Java 线程抽象的生命周期包括：

1. T1：创建（启动）；
2. T2：运行（行驶）；
3. T3：销毁（制动）。

如果不作控制，那么 T1 + T3 的开销将远大于 T2。

3.4.1 快速了解线程池

线程池的难点之一就是搞清楚 corePoolSize、workQueue、maximumPoolSize 这些参数之间的关系。

假如一个工地有若干个工作小队（等同于线程），这些工作小队需要加入项目组（线程池）才能有活干，长期不干活的就要被清理出项目组，如代码清单 3-10 所示。

代码清单 3-10 ManagerGroup.java

```
public class ManagerGroup {
    private static ExecutorService projectGroup = new ThreadPoolExecutor(
            5, // corePoolSize：核心小队数量
            20, // maximumPoolSize：最多能容纳多少个小队
            30, // keepAliveTime：多久没活干就请出项目组
            TimeUnit.SECONDS, // unit：时间单位
            new ArrayBlockingQueue<Runnable>(5),// workQueue：有多少小队后就不接收
            new ThreadPoolExecutor.CallerRunsPolicy() // handler：拒绝小队加入时的处理
    );

    // 增加包工头
    public static void addTask(Manager manager) {
        projectGroup.execute(manager);
    }

    public static void main(String[] args) {
        Manager manager = new Manager();
        Worker worker = new Worker();
        manager.setWorker(worker);
        ManagerGroup.addTask(manager);
    }
}
```

当 workQueue 使用的是有界阻塞队列 ArrayBlockingQueue 时，上面代码中各参数的关系如图 3-10 所示。

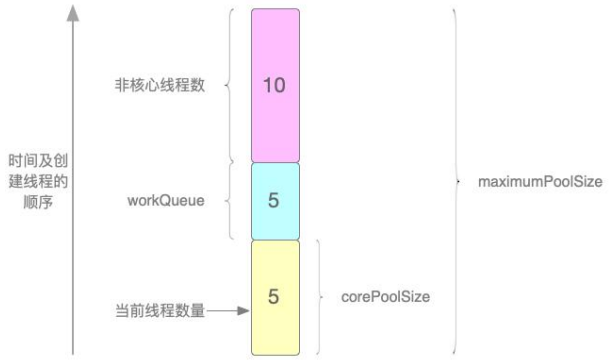

图 3-10 线程池参数之间的关系

1. 当 corePoolSize = 5，workQueue = 5，maximumPoolSize = 10 时，先创建 5 个核心线程，核心线程数满了再把新线程丢进 workQueue，等待队列满时会比较最大线程数 maximumPoolSize，此时 5 + 5 = 10 == 10，发现已不能继续创建线程，执行拒绝策略；

2. 当 corePoolSize = 5，workQueue = 5，maximumPoolSize = 8 时，先创建 5 个核心线程，核心线程数满了再把新线程丢进 workQueue，等待队列满时会比较最大线程数 maximumPoolSize，此时 5 + 5 = 10 > 8，发现已不能继续创建线程，执行拒绝策略；

3. 当 corePoolSize = 5，workQueue = 5，maximumPoolSize = 20 时，先创建 5 个核心线程，

核心线程数满了再把新线程丢进 workQueue，等待队列满时会比较最大线程数 maximumPoolSize，此时 5 + 5 = 10 < 20，可以继续创建 10 个新的非核心线程执行任务，超过了 20 再执行拒绝策略；

4. keepAliveTime 指的是 corePoolSize < 当前线程数 < maximumPoolSize 时，那些非核心线程空闲多久会被“优化”。

3.4.2 池中的队列

线程池中用于缓存任务的 workQueue 阻塞队列有这么几种：同步队列 SynchronousQueue、有界阻塞队列 ArrayBlockingQueue、无界阻塞队列 LinkedBlockingQueue 和有优先级的无界阻塞队列 PriorityBlockingQueue，它们之间的区别如图 3-11 所示。

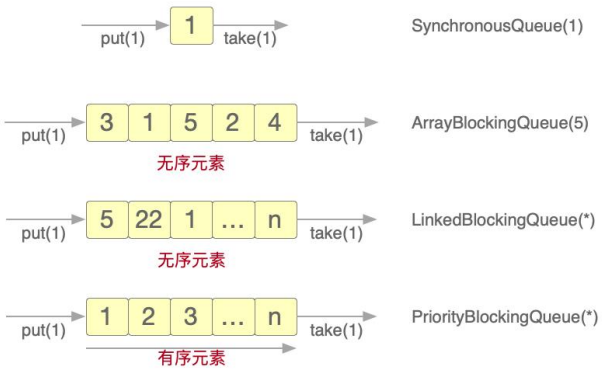

图 3-11 几种常用线程池之间的区别

1. SynchronousQueue 只能缓存一个元素，数据只能进一个再出一个，反之亦然；

2. ArrayBlockingQueue 相当于升级版的 SynchronousQueue，数据量比 SynchronousQueue 多，但也是进多少出多少，反之亦然；

3. LinkedBlockingQueue 则来者不拒，其元素数量的唯一限制就是内存的大小；

4. PriorityBlockingQueue 和 LinkedBlockingQueue 类似，但是执行 take()操作时，它的元素是有序的。

在实际开发中，因为 SynchronousQueue、LinkedBlockingQueue 和 PriorityBlockingQueue 这几种要么容量太小，要么又太大，倒是 ArrayBlockingQueue 有这种限且可以指定数量的队列应用较多，就以它为例子来说明。如代码清单 3-11 所示。

代码清单 3-11 SichuanCuisine.java

```
public class SichuanCuisine {
    public static BlockingQueue<String> queue = new ArrayBlockingQueue<>(5);

    static class Producer implements Runnable {
        @Override
        public void run() {
            try {
                queue.put("川菜");
```

```
                System.out.println("厨师做好" + Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    static class Consumer implements Runnable {
        @Override
        public void run() {
            try {
                String food = queue.take();
                System.out.println("客人消费" + fThread.currentThread().getName() + ood);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new Thread(new Consumer(), "第 " + i + " 道").start();
        }
        for (int i = 0; i < 5; i++) {
            new Thread(new Producer(), "第 " + i + " 道").start();
        }
    }
}
```

上述代码执行后的打印结果很清楚地说明了 ArrayBlockingQueue 的作用。

3.5 CAS 原子操作

CAS 是 Compare And Swap（比较与交换）的缩写，它用于实现多线程同步，允许算法执行“读-修改-写”操作，而无需担心被其他线程修改。用直白的话来说就是它的操作过程足够细，以至于细到线程都奈何不了它。

原子指令是指不会被线程调度机制打断的指令。这种操作一旦开始，就会一直运行到结束，中间没有任何线程可以打断或唤醒它，要么全部完成，要么全部失败。

整个 java.util.concurrent 工具包都是建立在 CAS 之上的，尤其是 Java 中大多数锁的实现类 AbstractQueuedSynchronizer（AQS），也是以 CAS 为基础的，它提供了一系列的独占锁、共享锁、可重入锁、自旋锁等线程控制手段。CAS 操作都是通过 sun 包下 Unsafe 类完成的，而 Unsafe 类中的方法都是 native 方法，由本地实现，和操作系统、CPU 都有关系。

CAS 有一个通用的实现范式：

1. 首先声明变量为 volatile；
2. 然后使用 CAS 的原子条件来更新；
3. 同时配合 volatile 实现线程之间的同步。

下面以 AtomicInteger 类为例来实现 CAS，如代码清单 3-12 所示。

代码清单 3-12 AtomicIntegerTest1.java

```
public class AtomicIntegerTest1 {
    public static volatile AtomicInteger atomic = new AtomicInteger(0);
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 0; i < 10; i++) {
            Runnable runnable = new Runnable() {
                @Override
                public void run() {
                    atomic.getAndIncrement();
                }
            };
            executor.submit(runnable);
        }
        TimeUnit.MILLISECONDS.sleep(100);
        executor.shutdown();
        System.out.println(atomic.get());
    }
}
```

虽然有 3 个线程轮流给变量 atomic 做累加，但执行后打印出来的结果就是 10，丝毫不受线程的影响。

3.6 锁与 AQS

在 Java 面试中，有一类高频出现的问题是：Java 有几种锁？每种锁的机制是什么？

这个问题笔者也曾问过，但大部分求职者除了说出“死锁”之外，还有什么其他锁就不清楚了。笔者认为，按使用场景来分，Java 有五大类 11 种锁，如图 3-12 所示。

图 3-12 Java 中的锁

当然，不同的人对锁的理解不同，可能分类和数量会不太一样，但这不影响本质。因为与锁及其底层实现技术 AQS（AbstractQueueSynchronizer，抽象队列同步器）相关的内容十分庞大且复杂。笔者结合自身经验，尝试将复杂的部分用简单的话来讲述，希冀能给读者继续更深入地学习了解锁和 AQS 提供一种有益的参考。

3.6.1 锁

与锁相关的类继承结构如图 3-13 所示。

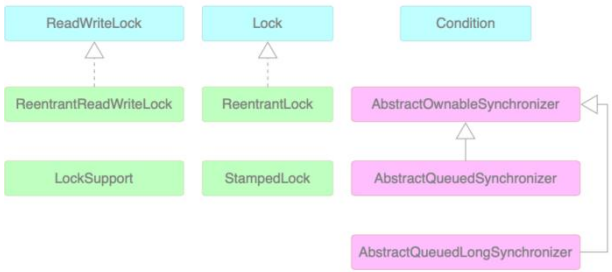

图 3-13 与锁相关的类继承结构

因为内容较多，这里只讲重点。

1. 悲观锁：包括 synchronized 关键字和 Lock 类，它确保写操作必定会成功；

2. 乐观锁：包括 CAS 算法和原子类，适合读操作多的场景，且容许数据偶有不一致的现象出现。

悲观锁与乐观锁的调用方式如代码清单 3-13 所示。

代码清单 3-13 PessimisticAndOptimisticLock.java

```
public class PessimisticAndOptimisticLock {
    public synchronized void callPessimisticLock() {
        // TODO 操作共享资源
    }
    private ReentrantLock lock = new ReentrantLock();
    public void operateResource() {
```

```
            lock.lock();
            // TODO 操作共享资源
            lock.unlock();
        }
        private AtomicInteger atomic = new AtomicInteger();
        public void optimistic() {
            atomic.getAndIncrement();
        }
}
```

在代码层面，自旋锁其实就是经常看到的 for(;;)语句。Java 的另一种循环语句 while...do 底层调用的也是 for(;;)。关于自旋锁有如下事实：

1. 如果锁被占用的时间很短，自旋锁的效果就较好，反之很差；
2. 自旋等待的时间如果超过了限定次数就会挂起；
3. 限定的默认自旋次数 10 次，可通过 JVM 参数-XX:PreBlockSpin 更改；
4. 可通过 JVM 参数-XX:+UseSpinning 开启是否使用自旋锁。

再接下来是可重入锁和非可重入锁。

1. 可重入锁：又叫递归锁，意思是同一个线程在外层方法获取到锁的时候，再进入到内层方法时就自动获取到锁（ReentrantLock 和 synchronized 都是可重入锁）。
2. 非可重入锁：进入内层方法时，需要将外层锁释放，但由于线程已在方法中，无法释放，因此极有可能会造成死锁。

开发中最常见的其实就是悲观锁、乐观锁和自旋锁（尤其在 JDK 源码中大量应用），理解清楚了这几个，其他的知道就行。

3.6.2 AQS

AQS（AbstractQueuedSynchronizer，抽象队列同步器）是用来实现锁及其他同步功能组件的 Java 底层技术，java.util.concurrent 包下大部分类的实现都离不开它。

通过继承 AQS：

1. ReentrantLock 和 Semaphore 类的内部类实现了公平锁和非公平锁；
2. CountDownLatch 类的内部类实现了共享锁；
3. ReentrantReadWriteLock 类的内部类实现了排他锁和共享锁。

AQS 主要实现两大功能：独占（Exclusive，有时也叫排他）和共享（Share）。AQS 在其内部维护一个 FIFO（First In First Out，先进先出）的线程阻塞队列，以及一个用 volatile 关键字修饰的状态变量 state。该 FIFO 队列有一个独特的名称：CLH（它是三个作者名称的首字母，即 Craig，Landin，Hagersten）。

用一句简单的话来概括 AQS：它基于 volatile 变量、CAS 和自旋这些工具，来改变线程的状态，成功则获取锁，失败则进入 CLH 线程阻塞队列。

一般子类自定义实现 AQS 时，要么是独占，要么是共享。也就是要么实现 tryAcquire()和 tryRelease()系列方法，要么实现 tryAcquireShared()和 tryReleaseShared()系列方法。

CLH 队列由多个 node 节点组成，而且大量使用“CAS 自旋 volatile 变量”这种经典代码，

如代码清单 3-14 所示。

代码清单 3-14 AbstractQueuedSynchronizer.java 部分源码和注释

```
private Node enq(final Node node) {
    for (;;) {// 自旋
        Node t = tail;// 将队尾指针给当前节点
        if (t == null) {
            // 如果尾节点为 null，说明队列还没有任何节点，那么头节点也就是尾节点
            if (compareAndSetHead(new Node())) {
                tail = head;
            }
        } else {
            node.prev = t;// 否则尾节点成为当前待加入节点的前继节点
            if (compareAndSetTail(t, node)) {// 将当前节点设置为尾节点
                // 尾节点的后续节点为当前节点
                t.next = node;
                return t;
            }
        }
    }
}
```

另外，在 cn.javabook.chapter03.lockaqs 源码包中有笔者给出关键注释的 AQS 类，感兴趣的读者可以阅读。

整个 AQS 的流程如图 3-14 所示。

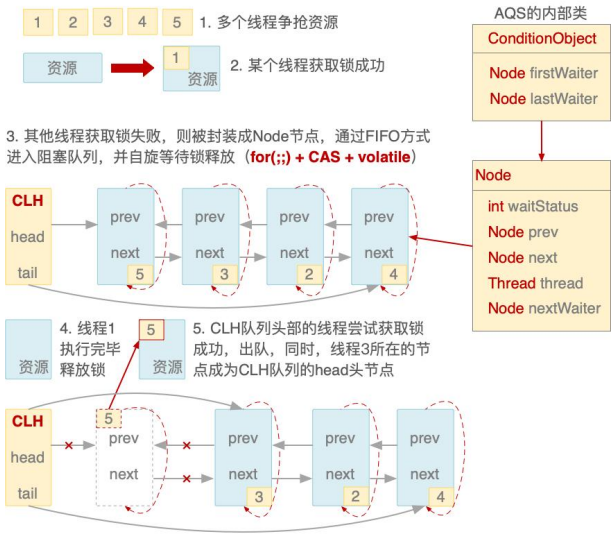

图 3-14 AQS 流程图

1. 当某个线程争抢资源成功时，其他线程就被封装成 Node 节点，通过 FIFO 方式进入 CLH 队列，并且不断“自旋”等待锁的释放，这就是自旋锁 for(;;)所起的作用，如上图中第

1 步至第 3 步所展示的那样；

2. 当抢到资源的线程执行完毕，CLH 队列头部的线程会尝试获取锁，这里采取的是公平锁策略。同时头节点后的下一个节点成为新的头节点，如上图中第 4 步和第 5 步所展示的那样。

AQS 的整体流程其实很简单，其核心就是 acquire(int) / release(int)和 tryAcquire(int) / tryRelease(int)这两组方法，而且它们总是成对出现。

3.6.3 叫号器

学以致用，下面就来看一看 AQS 好玩的地方。假如李星云的餐厅需要开发一个排队就餐程序，当人多排队的时候，每次只发放并允许进入三个号码，这其实就是一个叫号器。也就是说，它是一把可以指定资源数量的共享锁，而这正是 AQS 的两大功能之一。

代码清单 3-15 AqsShareLockTest.java

```
public class AqsShareLockTest {
    public static void main(String[] args) throws InterruptedException {
        AqsShareLock.count = 3;
        final Lock lock = new AqsShareLock();

        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        lock.lock();
                        System.out.println("持有 " + Thread.currentThread().getName() + " 的客人可以进餐厅就餐");
                        TimeUnit.MILLISECONDS.sleep(2000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        lock.unlock();
                    }
                }
            }).start();
        }
    }
}
```

运行程序之后，可以看到每隔一段时就连续释放出 3 个号，因为是 10 人进餐，所以最后 1 次就只有 1 个号码出现——这完全符合预期。

3.7 本章小节

笔者首先通过实际代码验证的方式，重新阐述线程周期的六种状态及其转换的方法。同时澄清了 Thread 和 Runnable 之间的误区。

接着，笔者列举了六种多线程常用的 API 方法，其中既有 Thread 类的静态方法也有 Object 类的实例方法，熟练掌握这些方法是后续学习多线程的基石。

多线程的两大关键字 volatile 和 synchronized 各司其职，一个负责变量，一个负责方法或代码块。volatile 只保证有序性和可见性，且使用 volatile 是有条件限制的。synchronized 作为重量级锁可以全面保证并发编程的三大特性（可见行、有序性、原子性）。

为了提升线程使用效率，Java 推出了线程池。对于线程池中阻塞队列的 corePoolSize、maximumPoolSize 和 workQueue 数量及其关系问题，笔者给出了解释。最后，笔者通过一个通用的代码清单将四大线程池一并做了演示。

Java 为了弥补 volatile 不保证原子性的缺陷，增加原子类和 CAS 操作，它是 JUC 大多数工具类和 AQS 的实现基础。

笔者将 Java 中的锁分为 5 大类 11 种，在开发中出现较多的是悲观锁和乐观锁、自旋锁、独占锁和共享锁。AQS 是基于 CLH 线程阻塞队列，通过“volatile 变量 + CAS + 自旋”的方式来改变线程状态的，最后以一个叫号器的例子演示了 AQS 的用法。