2019
版

治愈系 Java 工程师面试指导课程

[Part4・多线程并发编程和 JUC]

[本教程梳理了 java 多线程和并发编程的复习脉络，并且涵盖了多线程和并发的高频面试题；建议大家将目录结构打开，对照目录结构做一个复习的、提纲准备面试。]

1 线程的状态

1.1 线程的创建方式

1.1.1 继承 Thread 类

Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方 法就是通过 Thread 类的 start()实例方法。start()方法是一个 native 方法，它将启动一个新线 程，并执行 run()方法。

在 Java 5.0 提供了 java.util.concurrent （简称 JUC ）包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

1.1.2 实现 Runnable 接口

如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口。

1.1.3 ExecutorService、Callable<Class>、Future 有返回值线程

有返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须 Runnable 接口。执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务 返回的 Object 了，再结合线程池接口 ExecutorService 就可以实现传说中有返回结果的多线程 了。

1.1.3.1 使用 Callable 接口创建线程

Java 5.0 在 java.util.concurrent 提供了一个新的创建执行线程的方式：Callable 接口，Callable 接口类似于 Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的。

- Callable 接口和 Runnable 接口的区别
  - ✓ Runnable 不会返回结果
  - ✓ Runnalbe 无法抛出经过检查的异常。
  - ✓ Callable 需要依赖 FutureTask 实现类的支持，用来接收运算结果（FutureTask 也可以用作闭锁）

```
/*
 * 一、创建执行线程的方式三：实现 Callable 接口。 相较于实现 Runnable 接口的方式，方法可以有返回值，并且可以抛出异常。
 *
 * 二、执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。 FutureTask 是 Future 接口的实现类
 */
public class TestCallable {
```

```
    public static void main(String[] args) {
        ThreadDemo td = new ThreadDemo();

        //1.执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。
        FutureTask<Integer> result = new FutureTask<>(td);

        new Thread(result).start();

        //2.接收线程运算后的结果
        try {
            Integer sum = result.get();  //FutureTask 可用于 闭锁
            System.out.println(sum);
            System.out.println("------------------------------------");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }

}

class ThreadDemo implements Callable<Integer>{

    @Override
    public Integer call() throws Exception {
        int sum = 0;

        for (int i = 0; i <= 100000; i++) {
            sum += i;
        }

        return sum;
    }

}
```


1.1.4 基于线程池的方式

线程和数据库连接这些资源都是非常宝贵的资源。那么每次需要的时候创建，不需要的时候销 毁，是非常浪费资源的。那么我们就可以使用缓存的策略，也就是使用线程池。

1.1.5 实现 Runnable 接口和 Callable 接口的区别

如果想让线程池执行任务的话需要实现的 Runnable 接口或 Callable 接口。 Runnable 接口或 Callable 接口实现类都可

以被 ThreadPoolExecutor 或 ScheduledThreadPoolExecutor 执行。两者的区别在于 Runnable 接口不会返回结果但

是 Callable 接口可以返回结果。

备注： 工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。

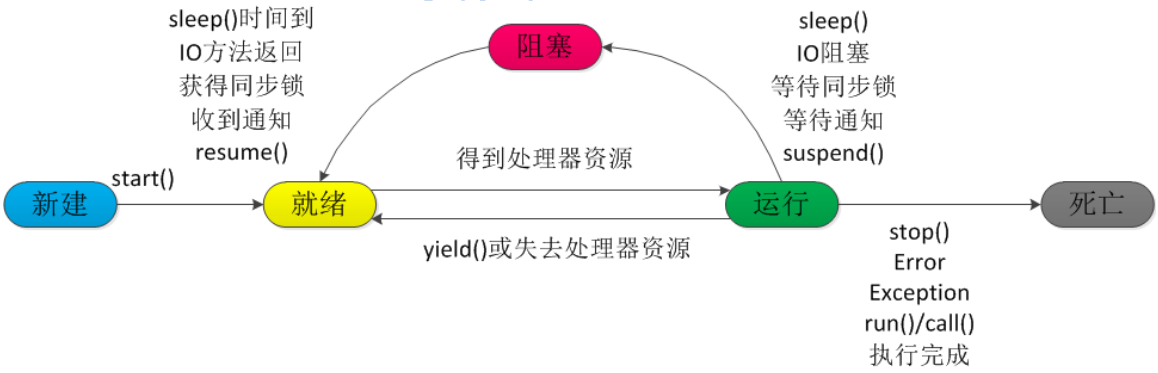
（ Executors.callable（Runnable task） 或 Executors.callable（Runnable task，Object resule） ）。

1.1.6 start 与 run 区别

1. start（）方法来启动线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕， 可以直接继续执行下面的代码。
2. 通过调用 Thread 类的 start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运 行。
3. 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运 行 run 函数当中的代码。 Run 方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

1.2 线程生命周期(状态)

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。 在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞 (Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自 运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换



1.2.1 新建状态（NEW）

当程序使用 new 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配 内存，并初始化其成员变量的值

1.2.2 就绪状态（RUNNABLE）：

当线程对象调用了 start()方法之后，该线程处于就绪状态。Java 虚拟机会为其创建方法调用栈和 程序计数器，等待调度运行。

1.2.3 运行状态（RUNNING）：

如果处于就绪状态的线程获得了 CPU，开始执行 run()方法的线程执行体，则该线程处于运行状 态。

1.2.4 阻塞状态（BLOCKED）：

阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。 直到线程进入可运行(runnable)状态，才有机会再次获得 cpu timeslice 转到运行(running)状 态。

阻塞的情况分三种：

等待阻塞（o.wait->等待对列）：

运行(running)的线程执行 o.wait()方法，JVM 会把该线程放入等待队列(waitting queue) 中。

同步阻塞(lock->锁池)

运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线 程放入锁池(lock pool)中。

其他阻塞(sleep/join)

运行(running)的线程执行 Thread.sleep(long ms)或 t.join()方法，或者发出了 I/O 请求时， JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入可运行(runnable)状态。

1.2.5 线程死亡（DEAD）

线程会以下面四种方式结束，结束后就是死亡状态。

1.2.5.1 正常结束

线程正常结束 run()或 call()方法执行完成。

1.2.5.2 异常退出

线程抛出一个未捕获的 Exception 或 Error。

1.2.5.3 Stop 方法终止

3.直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。

程序中可以直接使用 thread.stop()来强行终止线程，但是 stop 方法是很危险的，就象突然关 闭计算机电源，

而不是按正常程序关机一样，可能会 产生不可预料的结果，不安全主要是： thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子 线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用 thread.stop()后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈 现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因 此，并不推荐使用 stop 方法来终止线程。

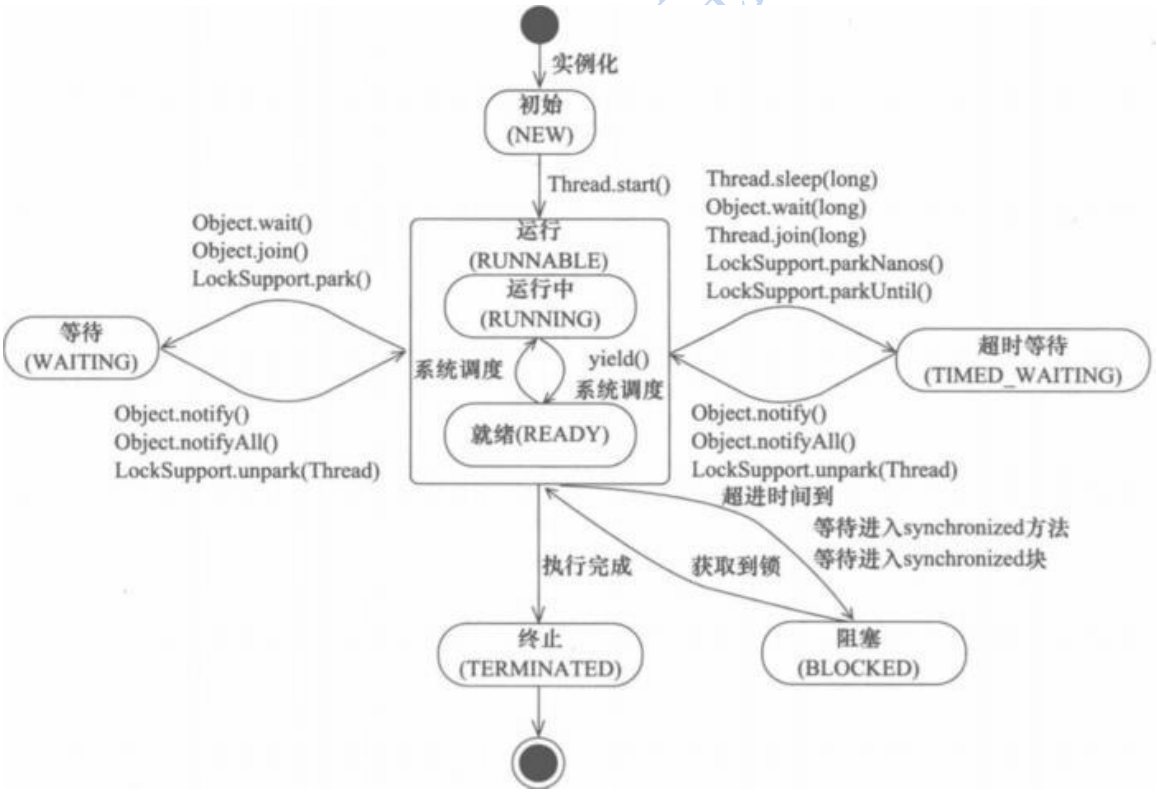
1.2.5.4 Interrupt 方法结束线程

使用 interrupt()方法来中断线程有两种情况：

1. 线程处于阻塞状态：如使用了 sleep,同步锁的 wait,socket 中的 receiver,accept 等方法时， 会使线程处于阻塞状态。当调用线程的 interrupt()方法时，会抛出 InterruptException 异常。 阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后 break 跳出循环状态，从而让 我们有机会结束这个线程的执行。通常很多人认为只要调用 interrupt 方法线程就会结束，实 际上是错的， 一定要先捕获 InterruptedException 异常之后通过 break 来跳出循环，才能正 常结束 run 方法。
2. 线程未处于阻塞状态：使用 isInterrupted()判断线程的中断标志来退出循环。当使用 interrupt()方法时，中断标志就会置 true，和使用自定义的标志来控制循环是一样的道理。

2 线程的常用方法

线程相关的基本方法有 wait，notify，notifyAll，sleep，join，yield 等。



2.1 线程等待（wait）

调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意的 是调用 wait()方法后，会释放对象的锁。因此，wait 方法一般用在同步方法或同步代码块中。

2.2 线程睡眠（sleep）

sleep 导致当前线程休眠，与 wait 方法不同的是 sleep 不会释放当前占有的锁,sleep(long)会导致 线程进入 TIMED-WATING 状态，而 wait()方法会导致当前线程进入 WATING 状态

2.3 sleep 与 wait 区别

1. 对于 sleep()方法，我们首先要知道该方法是属于 Thread 类中的。而 wait()方法，则是属于 Object 类中的
2. sleep()方法导致了程序暂停执行指定的时间，让出 cpu 该其他线程，但是他的监控状态依然 保持者，当指定的时间到了又会自动恢复运行状态。
3. 在调用 sleep()方法的过程中，线程不会释放对象锁。
4. 而当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此 对象调用 notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

2.4 线程让步（yield）

yield 会使当前线程让出 CPU 执行时间片，与其他线程一起重新竞争 CPU 时间片。一般情况下， 优先级高的线程有更大的可能性成功竞争得到 CPU 时间片，但这又不是绝对的，有的操作系统对 线程优先级并不敏感。

2.5 线程中断（interrupt）

中断一个线程，其本意是给这个线程一个通知信号，会影响这个线程内部的一个中断标识位。这 个线程本身并不会因此而改变状态(如阻塞，终止等)。

1. 调用 interrupt()方法并不会中断一个正在运行的线程。也就是说处于 Running 状态的线 程并不会因为被中断而被终止，仅仅改变了内部维护的中断标识位而已。
2. 若调用 sleep()而使线程处于 TIMED-WATING 状态，这时调用 interrupt()方法，会抛出 InterruptedException,从而使线程提前结束 TIMED-WATING 状态。
3. 许多声明抛出 InterruptedException 的方法(如 Thread.sleep(long mills 方法))，抛出异 常前，都会清除中断标识位，所以抛出异常后，调用 isInterrupted()方法将会返回 false。
4. 中断状态是线程固有的一个标识位，可以通过此标识位安全的终止线程。比如 ,你想终止 一个线程 thread 的时候,可以调用 thread.interrupt()方法,在线程的 run 方法内部可以 根据 thread.isInterrupted()的值来优雅的终止线程。

2.6 Join 等待其他线程终止

join() 方法，等待其他线程终止，在当前线程中调用一个线程的 join() 方法，则当前线程转为阻塞 状态，回到另一个线程结束，当前线程再由阻塞状态变为就绪状态，等待 cpu 的宠幸。

为什么要用 join()方法?

很多情况下，主线程生成并启动了子线程，需要用到子线程返回的结果，也就是需要主线程需要 在子线程结束后再结束，这时候就要用到 join() 方法。

```
System.out.println(Thread.currentThread().getName() +"线程运行开始!"); Thread6 thread1 = new Thread6();
        thread1.setName("线程  B");
        thread1.join();
        System.out.println("这时  thread1 执行完毕之后才能执行主线程");
```

2.7 线程唤醒（notify）

Object 类中的 notify() 方法，唤醒在此对象监视器上等待的单个线程，如果所有线程都在此对象 上等待，则会选择唤醒其中一个线程，选择是任意的，并在对实现做出决定时发生，线程通过调 用其中一个 wait() 方法，在对象的监视器上等待，直到当前的线程放弃此对象上的锁定，才能继 续执行被唤醒的线程，被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞 争。类似的方法还有 notifyAll() ，唤醒再次监视器上等待的所有线程。

2.8 其他方法

1. sleep()：强迫一个线程睡眠N毫秒。
2. isAlive()： 判断一个线程是否存活。
3. join()： 等待线程终止。
4. activeCount()： 程序中活跃的线程数。
5. enumerate()： 枚举程序中的线程。
6. currentThread()： 得到当前线程。
7. isDaemon()： 一个线程是否为守护线程。
8. setDaemon()： 设置一个线程为守护线程。(用户线程和守护线程的区别在于，是否等待主线 程依赖于主线程结束而结束)
9. setName()： 为线程设置一个名称。
10. wait()： 强迫一个线程等待。
11. notify()： 通知一个线程继续运行。
12. setPriority()： 设置一个线程的优先级。
13. getPriority()：获得一个线程的优先级。

3 Java 多线程常见问题

3.1 上下文切换

多线程并不一定是要在多核处理器才支持的，就算是单核也是可以支持多线程的。 CPU 通过给每个线程分配一定的时间片，由于时间非常短通常是几十毫秒，所以 CPU 可以不停的切换线程执行任务从而达到了多线程的效果。但是由于在线程切换的时候需要保存本次执行的信息，在该线程被 CPU 剥夺时间片后又再次运行恢复上次所保存的信息的过程就称为上下文切换。

上下文切换是非常耗效率的。
通常有以下解决方案:

- 采用无锁编程，比如将数据按照 Hash(id) 进行取模分段，每个线程处理各自分段的数据，从而避免使用锁。
- 采用 CAS(compare and swap) 算法，如 Atomic 包就是采用 CAS 算法
- 合理的创建线程，避免创建了一些线程但其中大部分都是处于 waiting 状态，因为每当从 waiting 状态切换到 running 状态都是一次上下文切换。

3.2 死锁

死锁的场景一般是：线程 A 和线程 B 都在互相等待对方释放锁，或者是其中某个线程在释放锁的时候出现异常如死循环之类的。这时就会导致系统不可用。

3.2.1 产生死锁四个条件

- 互斥条件：该资源任意一个时刻只由一个线程占用。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件:线程已获得的资源在末使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
- 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。

3.2.2 死锁的解决方案

- 尽量一个线程只获取一个锁。
- 一个线程只占用一个资源。
- 尝试使用定时锁，至少能保证锁最终会被释放。

3.3 资源限制

当在带宽有限的情况下一个线程下载某个资源需要 1M/S,当开 10 个线程时速度并不会乘 10 倍，反而还会增加时间，毕竟上下文切换比较耗时。
如果是受限于资源的话可以采用集群来处理任务，不同的机器来处理不同的数据，就类似于开始提到的无锁编程。

4 Java 多线程三大核心

4.1 原子性

Java 的原子性就和数据库事务的原子性差不多，一个操作中要么全部执行成功或者失败。
JMM 只是保证了基本的原子性，但类似于 i++ 之类的操作，看似是原子操作，其实里面涉及到:

- 获取 i 的值。
- 自增。

- 再赋值给 i。

这三步操作，所以想要实现 i++ 这样的原子操作就需要用到 synchronized 或者是 lock 进行加锁处理。
如果是基础类的自增操作可以使用 AtomicInteger 这样的原子类来实现(其本质是利用了 CPU 级别的 的 CAS 指令来完成的)。
其中用的最多的方法就是: incrementAndGet() 以原子的方式自增。 源码如下:

```
public final long incrementAndGet() {
        for (;;) {
            long current = get();
            long next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }
```

首先是获得当前的值，然后自增 +1。接着则是最核心的 compareAndSet() 来进行原子更新。

```
public final boolean compareAndSet(long expect, long update) {
        return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
    }
```

其逻辑就是判断当前的值是否被更新过，是否等于 current，如果等于就说明没有更新过然后将当前的值更新为 next，如果不等于则返回 false 进入循环，直到更新成功为止。
还有其中的 get() 方法也很关键，返回的是当前的值，当前值用了 volatile 关键词修饰，保证了内存可见性。

```
 private volatile int value;
```

4.1.1 CAS 算法

CAS (Compare-And-Swap) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令【比如 i++】，用于管理对共享数据的并发访问。CAS 是一种无锁的非阻塞算法的实现。CAS 包含了 3 个操作数：需要读写的内存值 V、进行比较的值 A、拟写入的新值 B；当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

CAS 操作是抱着乐观的态度进行的(乐观锁)，它总是认为自己可以成功完成操作。当多个线程同时 使用 CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂 起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理， CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

4.1.2 ABA 问题

CAS 会导致“ABA 问题”。CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时 刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操 作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过 程就是没有问题的。
部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修 改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本 号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问 题，因为版本号只会增加不会减少。

描述: 第一个线程取到了变量 x 的值 A，然后巴拉巴拉干别的事，总之就是只拿到了变量 x 的值 A。这段时间内第二个线程也取到了变量 x 的值 A，然后把变量 x 的值改为 B，然后巴拉巴拉干别的事，最后又把变量 x 的值变为 A （相当于还原了）。在这之后第一个线程终于进行了变量 x 的操作，但是此时变量 x 的值还是 A，所以 compareAndSet 操作是成功。 - 例子描述(可能不太合适，但好理解): 年初，现金为零，然后通过正常劳动赚了三百万，之后正常消费了（比如买房子）三百万。年末，虽然现金零收入（可能变成其他形式了），但是赚了钱是事实，还是得交税的！

4.1.1 原子类

基本类型
使用原子的方式更新基本类型
AtomicInteger：整形原子类
AtomicLong：长整型原子类
AtomicBoolean ：布尔型原子类
数组类型
使用原子的方式更新数组里的某个元素
AtomicIntegerArray：整形数组原子类
AtomicLongArray：长整形数组原子类
AtomicReferenceArray ：引用类型数组原子类
引用类型
AtomicReference：引用类型原子类
AtomicStampedRerence：原子更新引用类型里的字段原子类
AtomicMarkableReference ：原子更新带有标记位的引用类型
对象的属性修改类型
AtomicIntegerFieldUpdater:原子更新整形字段的更新器
AtomicLongFieldUpdater：原子更新长整形字段的更新器
AtomicStampedReference ：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。
CAS 的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法
是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个 volatile 变
量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。
关于 Atomic 原子类这部分更多内容可以查看我的这篇文章：并发编程面试必备：JUC 中的

4.1.1.1 Atomic 原子变量

原子变量类的小工具包，支持在单个变量上解除锁的线程安全编程。事实上，此包中的类可将 volatile 值、字段和数组元素的概念扩展到那些也提供原子条件更新操作的类。

类 AtomicBoolean、AtomicInteger、AtomicLong 和 AtomicReference 的实例各自提供对相应类型单个变量的访

问和更新。每个类也为该类型提供适当的实用工具方法。

AtomicIntegerArray、AtomicLongArray 和 AtomicReferenceArray 类进一步扩展了原子操作，对这些类型的数组提供了支持。这些类在为其数组元素提供 volatile 访问语义方面也引人注目，这对于普通数组来说是不受支持的。

他们的实现原理相同， 区别在与运算对象类型的不同。令人兴奋地，还可以通过 AtomicReference<V>将一个对象的所 有操作转化成原子操作。我们知道，在多线程程序中，诸如++i 或 i++等运算不具有原子性，是不安全的线程操作之一。 通常我们会使用 synchronized 将该操作变成一个原子操作，但 JVM 为此类操作特意提供了一些 同步类，使得使用更方便，且使程序运行效率变得更高。通过相关资料显示，通常 AtomicInteger 的性能是 ReentantLock 的好几倍。

核心方法：

boolean compareAndSet(expectedValue, updateValue)

java.util.concurrent.atomic 包下提供了一些原子操作的常用类:

AtomicBoolean 、AtomicInteger 、AtomicLong 、 AtomicReference

AtomicIntegerArray 、AtomicLongArray

AtomicMarkableReference

AtomicReferenceArray

AtomicStampedReference

4.1.1.2 AtomicInteger

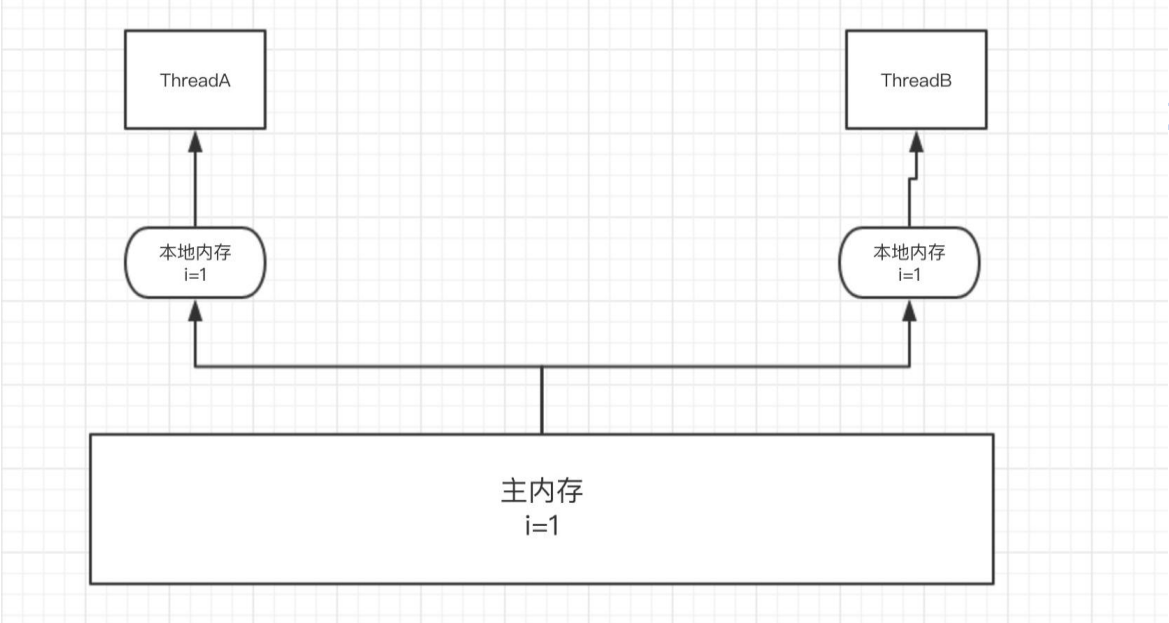
首先说明，此处 AtomicInteger，一个提供原子操作的 Integer 的类，常见的还有 AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference 等

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private volatile int value;

    public final int get() {
        return value;
    }

    public final int getAndIncrement() {
        for (; ; ) {  //CAS 自旋，一直尝试，直达成功
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }

    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
}
```

getAndIncrement 采用了 CAS 操作，每次从内存中读取数据然后将此数据和+1 后的结果进行 CAS 操作，如果成功就返回结果，否则重试直到成功为止。而 compareAndSet 利用 JNI 来完成 CPU 指令的操作。

4.2 可见性

现代计算机中，由于 CPU 直接从主内存中读取数据的效率不高，所以都会对应的 CPU 高速缓存，先将主内存中的数据读取到缓存中，线程修改数据之后首先更新到缓存，之后才会更新到主内存。如果此时还没有将数据更新到主内存其他的线程此时来读取就是修改之前的数据。



如上图所示。

volatile 关键字就是用于保证内存可见性，当线程 A 更新了 volatile 修饰的变量时，它会立即刷新到主线程，并且将其余缓存中该变量的值清空，导致其余线程只能去主内存读取最新值。

使用 volatile 关键词修饰的变量每次读取都会得到最新的数据，不管哪个线程对这个变量的修改都会立即刷新到主内存。

synchronized 和加锁也能能保证可见性，实现原理就是在释放锁之前其余线程是访问不到这个共享变量的。但是和 volatile 相比开销较大。

4.3 顺序性

以下这段代码:

```
int a = 100 ; //1
int b = 200 ; //2
int c = a + b ; //3
```

正常情况下的执行顺序应该是 1>>2>>3。但是有时 JVM 为了提高整体的效率会进行指令重排导致执行的顺序可能是 2>>1>>3。但是 JVM 也不能是什么都进行重排，是在保证最终结果和代码顺序执行结果一致的情况下才可能进行重排。

重排在单线程中不会出现问题，但在多线程中会出现数据不一致的问题。

Java 中可以使用 volatile 来保证顺序性，synchronized 和 lock 也可以来保证有序性，和保证原子性的方式一样，通过同一段时间只能一个线程访问来实现的。

除了通过 volatile 关键字显式的保证顺序之外， JVM 还通过 happen-before 原则来隐式的保证顺序性。

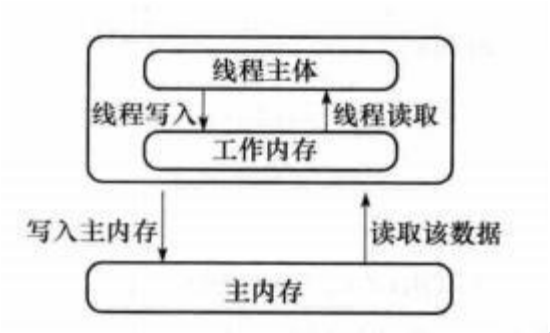
其中有一条就是适用于 volatile 关键字的，针对于 volatile 关键字的写操作肯定是在读操作之前，也就是说读取的值肯定是最新的。

4.4 volatile 关键字

Java 语言提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他 线程。volatile 变量具备两种特性，volatile 变量不会被缓存在寄存器或者对其他处理器不可见的 地方，因此在读取 volatile 类型的变量时总会返回最新写入的值。 Volatile 的作用有两个：禁止重排序和变量可见性

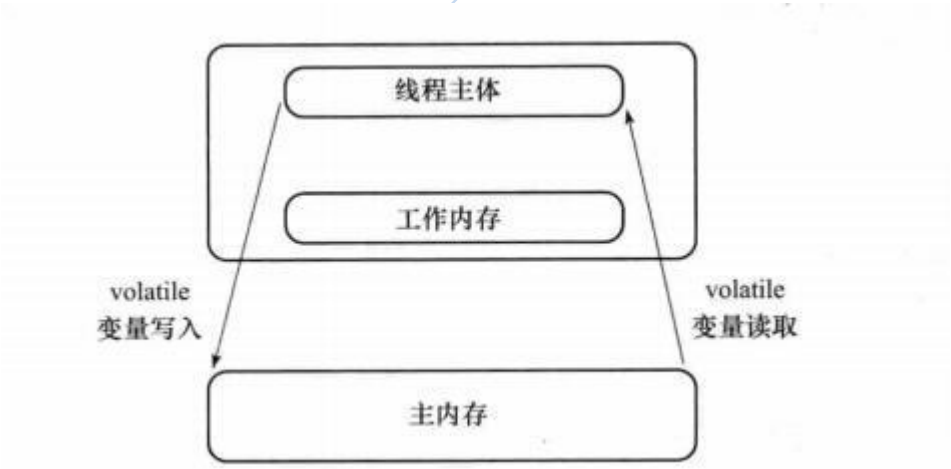
4.4.1 Java 内存模型

在 JDK1.2 之前，Java 的内存模型实现总是从**主存**（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数据的不一致**。



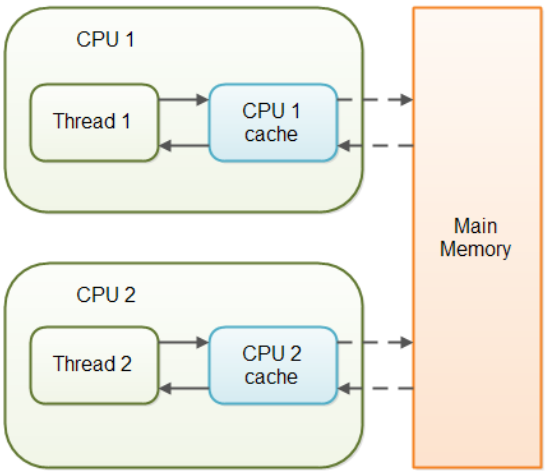
要解决这个问题，就需要把变量声明为 **volatile**，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。

说白了， **volatile** 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。



4.4.2 禁止重排序

volatile 禁止了指令重排。
比 sychronized 更轻量级的同步锁
在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 sychronized 关键字更轻量级的同步机制。volatile 适合这种场景：一个变量被多个线程共享，线程直接给这个变量赋值。



当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到 CPU 缓存中。如果计算机有多个 CPU，每个线程可能在不同的 CPU 上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

4.4.3 适用场景

值得说明的是对 volatile 变量的单次读/写操作可以保证原子性的，如 long 和 double 类型变量， 但是并不能保证 i++这种操作的原子性，因为本质上 i++是读、写两次操作。在某些场景下可以 代替 Synchronized。但是,volatile 的不能完全取代 Synchronized 的位置，只有在一些特殊的场景下，才能适用 volatile。总的来说，必须同时满足下面两个条件才能保证在并发环境的线程安全：
（1）对变量的写操作不依赖于当前值（比如 i++），或者说是单纯的变量赋值（boolean flag = true）。
（2）该变量没有包含在具有其他变量的不变式中，也就是说，不同的 volatile 变量之间，不 能互相依赖。***只有在状态真正独立于程序内其他内容时才能使用 volatile。***

4.4.4 变量可见性

保证该变量对所有线程可见，这里的可见性指的是当一个线程修改了变量的值，那么新的 值对于其他线程是可以立即获取的。

内存可见性（Memory Visibility）是指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。

可见性错误是指当读操作与写操作在不同的线程中执行时，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。

我们可以通过同步来保证对象被安全地发布。除此之外我们也可以使用一种更加轻量级的 volatile 变量。Java 提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。可以将 volatile 看做一个轻量级的锁，但是又与锁有些不同：

对于多线程，不是一种互斥关系

不能保证变量状态的“原子性操作”

```
/*
 * 一、volatile 关键字：当多个线程进行操作共享数据时，可以保证内存中的数据可见。
 *                    相较于 synchronized 是一种较为轻量级的同步策略。
 *
 * 注意：
 * 1. volatile 不具备“互斥性”
 * 2. volatile 不能保证变量的“原子性”
 */
public class TestVolatile {

    public static void main(String[] args) {
        ThreadDemo td = new ThreadDemo();
        new Thread(td).start();

        while(true){
            if(td.isFlag()){
                System.out.println("------------------");
                break;
            }
        }

    }

}

class ThreadDemo implements Runnable {

    //如果不加volatile关键字，那么主线程永远不会结束，因为flag是不可见的
    private volatile boolean flag = false;

    @Override
    public void run() {

        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
        }
```

```
        flag = true;

        System.out.println("flag=" + isFlag());

    }

    public boolean isFlag() {
        return flag;
    }

    public void setFlag(boolean flag) {
        this.flag = flag;
    }

}
```

4.4.5 volatile 双重检查锁的单例模式

可以用 volatile 实现一个双重检查锁的单例模式：

```
public class Singleton {
    private static volatile Singleton singleton;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }

}
```

这里的 volatile 关键字主要是为了防止指令重排。 如果不用 volatile，singleton = new Singleton();，这段代码其实是分为三步：

- 分配内存空间。(1)
- 初始化对象。(2)
- 将 singleton 对象指向分配的内存地址。(3)

加上 volatile 是为了让以上的三步操作顺序执行，反之有可能第三步在第二步之前被执行就有可能导致某个线程拿到的单例对象还没有初始化，以致于使用报错。

4.4.6 volatile 控制停止线程的标记

```
private volatile boolean flag ;
private void run(){
    new Thread(new Runnable() {
        @Override
        public void run() {
            while (flag) {
                doSomeThing();
            }
        }
    });
}

private void stop(){
    flag = false ;
}
```

这里如果没有用 volatile 来修饰 flag ，就有可能其中一个线程调用了 stop()方法修改了 flag 的值并不会立即刷新到主内存中，导致这个循环并不会立即停止。
这里主要利用的是 volatile 的内存可见性。

- 总结一下:

volatile 关键字只能保证可见性，顺序性，**不能保证原子性**。

5 Java 多线程的锁

5.1 锁的分类

5.1.1 同一进程

5.1.1.1 重入锁

使用 ReentrantLock 获取锁的时候会判断当前线程是否为获取锁的线程，如果是则将同步的状态 +1 ,释放锁的时候则将状态 -1。只有将同步状态的次数置为 0 的时候才会最终释放锁。

5.1.1.2 读写锁

使用 ReentrantReadWriteLock ,同时维护一对锁：读锁和写锁。当写线程访问时则其他所有锁都将阻塞，读线程访问时则不会。通过读写锁的分离可以很大程度的提高并发量和吞吐量。

5.1.2 不同进程

不同进程中的锁机制即分布式锁，通常有以下几种解决方案

5.1.2.1 基于数据库

可以创建一张表，将其中的某个字段设置为唯一索引，当多个请求过来的时候只有新建记录成功的请求才算获取到锁，当使用完毕删除这条记录的时候即释放锁。
存在的问题:

- 数据库单点问题，挂了怎么办?
- 不是重入锁，同一进程无法在释放锁之前再次获得锁，因为数据库中已经存在了一条记录了。
- 锁是非阻塞的，一旦 `insert` 失败则会立即返回，并不会进入阻塞队列只能下一次再次获取。
- 锁没有失效时间，如果那个进程解锁失败那就没有请求可以再次获取锁了。

解决方案:

- 数据库切换为主从，不存在单点。
- 在表中加入一个同步状态字段，每次获取锁的是加 `1` ，释放锁的时候`-1`，当状态为 `0` 的时候就删除这条记录，即释放锁。
- 非阻塞的情况可以用 `while` 循环来实现，循环的时候记录时间，达到 `X` 秒记为超时，`break`。
- 可以开启一个定时任务每隔一段时间扫描找出多少 `X` 秒都没有被删除的记录，主动删除这条记录。

5.1.2.2 基于 Redis

使用 `setNX(key) setEX(timeout)` 命令，只有在该 `key` 不存在的时候创建这个 `key`，就相当于获取了锁。由于有超时时间,所以过了规定时间会自动删除,这样也可以避免死锁。使用 redis 实现分布式锁的核心就是 setnx+getset 方式

```
加锁：setnx(lock, 时间戳+超时时间)
解决并发：
while(jedis.setnx(lock, now+超时时间)==0）{
    if(now>jedis.get(lock) && now>jedis.getset(lock, now+超时时间)){
        break;
    }else{
        Thread.sleep(300);
    }
}
执行业务代码;
jedis.del(lock);
释放锁：jedis.del(lock);
```

如果在公司里落地生产环境用分布式锁的时候，一定是会用开源类库的，比如 Redis 分布式锁，一般就是用 Redisson 框架就好了，非常的简便易用。

```
RLock rlock = redisson.getLock(“mylock”);
rlock.lock();
```

```
….
Rlock.unlock();
```

5.1.2.3 基于 ZK

临时节点

5.2 Synchronized 加锁

5.2.1 synchronized 关键字的理解【整体介绍】

synchronized 关键字底层原理属于 JVM 层面。

当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程 同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。 Java 中可 以使用 synchronized 关键字来取得一个对象的同步锁。

synchronized 关键字解决的是多个线程之间访问资源的同步性，synchronized 关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，synchronized 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

5.2.2 synchronized 关键字最主要的三种使用方式

synchronized 它可以把任意一个非 NULL 的对象当作锁。他属于独占式的悲观锁，同时属于可重 入锁。

一个对象里面如果有多个 synchronized 方法，某一个时刻内，只要一个线程去调用其中的一个 synchronized 方法了，其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些 synchronized 方法

1. 作用于方法时，锁住的是对象的实例(this)； synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列， 当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。
2. 当作用于静态方法时，锁住的是 Class 实例，又因为 Class 的相关数据存储在永久带 PermGen （jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁， 会锁所有调用该方法的线程；因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁
3. 同步块，锁的是 () 中的对象。。

5.2.3 synchronized 关键字原理

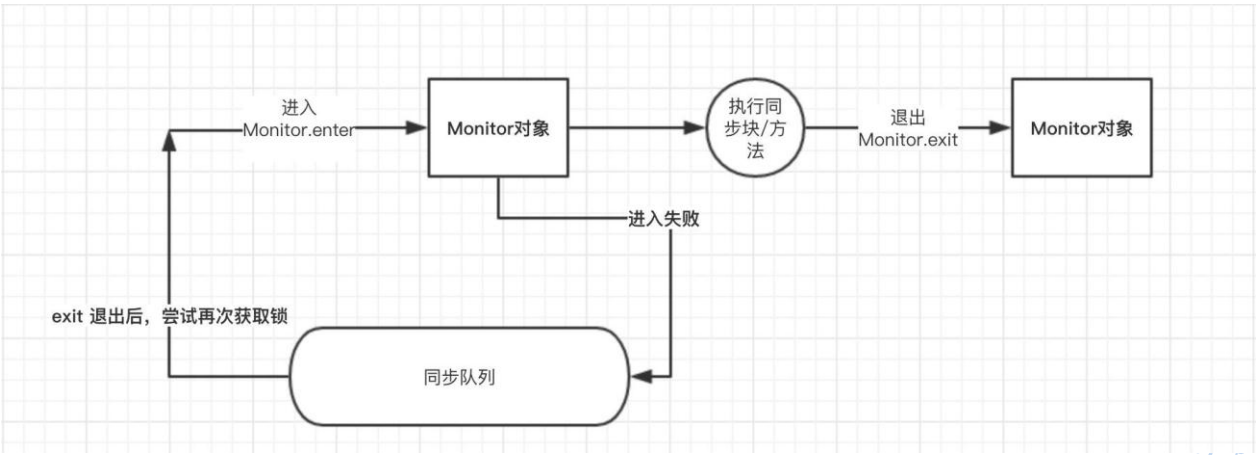
5.2.3.1 synchronized 同步语句块的情况

```
public class SynchronizedDemo {

    public void method() {

        synchronized (this) {

            System.out.println("synchronized 代码块");

        }

    }

}
```

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 `javac SynchronizedDemo.java` 命令生成编译后的 .class 文件，然后执行 `javap -c -s -v -l SynchronizedDemo.class`。

```
public void method();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String Method 1 start
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
    Exception table:
       from    to  target type
           4    14    17   any
          17    20    17   any
    LineNumberTable:
      line 5: 0
      line 6: 4
      line 7: 12
      line 8: 22
    StackMapTable: number_of_entries = 2
      frame_type = 255 /* full_frame */
        offset_delta = 17
        locals = [ class test/SynchronizedDemo, class java/lang/Object ]
        stack = [ class java/lang/Throwable ]
      frame_type = 250 /* chop */
        offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"
```

从上面我们可以看出：



synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor 对象存在于每个 Java 对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么 Java 中任意对象可以作为锁的原因) 的持有权.当计数器为 0 则可以成功获取，获取后将锁计数器设为 1 也就是加 1。相应的在执行 monitorexit 指令后，将锁计数器设为 0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

5.2.3.2 synchronized 修饰方法的的情况

```
public class SynchronizedDemo2 {
        public synchronized void method() {
            System.out.println("synchronized 方法");
        }
}
```

```
{
  public test.SynchronizedDemo2();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 3: 0

  public synchronized void method();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String synchronized 鏂规硶
         5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 5: 0
        line 6: 8
}
SourceFile: "SynchronizedDemo2.java"
```

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的却是 ACC*SYNCHRONIZED* *标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC*SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

在 Java 早期版本中，synchronized 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

5.2.4 JDK1.6 之后的底层优化

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

5.2.4.1 偏向锁

引入偏向锁的目的和引入轻量级锁的目的很像，他们都是为了没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。但是不同是：轻量级锁在无竞争的情况下使用 CAS 操作去代替使用互斥量。而偏向锁在无竞争的情况下会把整个同步都消除掉。

偏向锁的“偏”就是偏心的偏，它的意思是会偏向于第一个获得它的线程，如果在接下来的执行中，该锁没有被其他线程获取，那么持有偏向锁的线程就不需要进行同步！关于偏向锁的原理可以查看《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》第二版的 13 章第三节锁优化。

但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。

5.2.4.2 轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6 之后加入的)。**轻量级锁不是为了代替重量级锁，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗，因为使用轻量级锁时，不需要申请互斥量。另外，轻量级锁的加锁和解锁都用到了 CAS 操作。** 关于轻量级锁的加锁和解锁的原理可以查看《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》第二版的 13 章第三节锁优化。

轻量级锁能够提升程序同步性能的依据是“对于绝大部分锁，在整个同步周期内都是不存在竞争的”，这是一个经验数据。如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥操作的开销。但如果存在锁竞争，除了互斥量开销外，还会额外发生 CAS 操作，因此在有锁竞争的情况下，轻量级锁比传统的重量级锁更慢！如果锁竞争激烈，那么轻量级将很快膨胀为重量级锁！

5.2.4.3 自旋锁和自适应自旋

百度百科对自旋锁的解释

> 何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。

互斥同步对性能最大的影响就是阻塞的实现，因为挂起线程/恢复线程的操作都需要转入内核态中完成（用户态转换到内核态会耗费时间）。

一般线程持有锁的时间都不是太长，所以仅仅为了这一点时间去挂起线程/恢复线程是得不偿失的。 所以，虚拟机的开发团队就这样去考虑：“我们能不能让后面来的请求获取锁的线程等待一会而不被挂起呢？看看持有锁的线程是否很快就会释放锁”。**为了让一个线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就叫做自旋。**

自旋锁在 JDK1.6 之前其实就已经引入了，不过是默认关闭的，需要通过`--XX:+UseSpinning` 参数来开启。JDK1.6 及 1.6 之后，就改为默认开启的了。需要注意的是：自旋等待不能完全替代阻塞，因为它还是要占用处理器时间。如果锁被占用的时间短，那么效果当然就很好了！反之，相反！自旋等待的时间必须要有限度。如果自旋超过了限定次数任然没有获得锁，就应该挂起线程。**自旋次数的默认值是 10 次，用户可以修改`--XX:PreBlockSpin`来更改。**

另外,**在 JDK1.6 中引入了自适应的自旋锁。自适应的自旋锁带来的改进就是：自旋的时间不在固定了，而是和前一次同一个锁上的自旋时间以及锁的拥有者的状态来决定，虚拟机变得越来越“聪明”了。**

5.2.4.4 锁消除

锁消除理解起来很简单，它指的就是虚拟机即使编译器在运行时，如果检测到那些共享数据不可能存在竞争，那么就执行锁消除。锁消除可以节省毫无意义的请求锁的时间。

5.2.4.5 锁粗化

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小，——直在共享数据的实际作用域才进行同步，这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待线程也能尽快拿到锁。

大部分情况下，上面的原则都是没有问题的，但是如果一系列的连续操作都对同一个对象反复加锁和解锁，那么会带来很多不必要的性能消耗。

5.2.5 synchronized 关键字和 volatile 关键字的区别

- **volatile 关键字**是线程同步的**轻量级实现，所以 volatile 性能肯定比 synchronized 关键字要好**。但是 **volatile 关键字只能用于变量而 synchronized 关键字可以修饰方法以及代码块**。synchronized 关键字在 JavaSE1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后

执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些。**

- **多线程访问 volatile 关键字不会发生阻塞，而 synchronized 关键字可能会发生阻塞**
- **volatile 关键字能保证数据的可见性，但不能保证数据的原子性。synchronized 关键字两者都能保证。**
- **volatile 关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。**

5.3 Lock 同步锁

用于解决多线程安全问题的方式：

- synchronized:隐式锁
  - ✓同步代码块
  - ✓同步方法
- Lock 显示锁
  - ✓ 同步锁 Lock

需要通过 lock() 方法手动上锁，通过 unlock() 方法手动进行释放锁

在 Java 5.0 之前，协调共享对象的访问时可以使用的机制只有 synchronized 和 volatile 。Java 5.0 后增加了一些新的机制，但并不是一种替代内置锁的方法，而是当内置锁不适用时，作为一种可选择的高级功能。

ReentrantLock 实现了 Lock 接口，并提供了与 synchronized 相同的互斥性和内存可见性。但相较于 synchronized 提供了更高的处理锁的灵活性。提供了更高的处理锁的灵活性。

5.3.1 Lock 接口的主要方法

1. void lock(): 执行此方法时, 如果锁处于空闲状态, 当前线程将获取到锁. 相反, 如果锁已经 被其他线程持有, 将禁用当前线程, 直到当前线程获取到锁.
2. boolean tryLock()：如果锁可用， 则获取锁， 并立即返回 true, 否则返回 false. 该方法和 lock()的区别在于, tryLock()只是"试图"获取锁， 如果锁不可用， 不会导致当前线程被禁用, 当前线程仍然继续往下执行代码. 而 lock()方法则是一定要获取到锁， 如果锁不可用, 就一 直等待, 在未获得锁之前,当前线程并不继续向下执行.
3. void unlock()：执行此方法时，当前线程将释放持有的锁. 锁只能由持有者释放, 如果线程 并不持有锁, 却执行该方法, 可能导致异常的发生.
4. Condition newCondition()：条件对象，获取等待通知组件。该组件和当前的锁绑定， 当前线程只有获取了锁，才能调用该组件的 await()方法，而调用后，当前线程将缩放锁。
5. getHoldCount() ：查询当前线程保持此锁的次数，也就是执行此线程执行 lock 方法的次 数。
6. getQueueLength（）：返回正等待获取此锁的线程估计数，比如启动 10 个线程，1 个 线程获得锁，此时返回的是 9
7. getWaitQueueLength：（Condition condition）返回等待与此锁相关的给定条件的线 程估计数。比如 10 个线程，用同一个 condition 对象，并且此时这 10 个线程都执行了 condition 对象的 await 方法，那么此时执行此方法返回 10
8. hasWaiters(Condition condition) ： 查 询 是 否 有 线 程 等 待 与 此 锁 有 关 的 给 定 条 件 (condition)，对于指定 contidion 对象，有多少线程执行了 condition.await 方法
9. hasQueuedThread(Thread thread)：查询给定线程是否等待获取此锁
10. hasQueuedThreads()：是否有线程等待此锁
11. isFair()：该锁是否公平锁
12. isHeldByCurrentThread()： 当前线程是否保持锁锁定，线程的执行 lock 方法的前后分 别是 false 和 true

13. isLock()：此锁是否有任意线程占用
14. lockInterruptibly（）：如果当前线程未被中断，获取锁
15. tryLock（）：尝试获得锁，仅在调用时锁未被线程占用，获得锁
16. tryLock(long timeout TimeUnit unit)：如果锁在给定等待时间内没有被另一个线程保持， 则获取该锁。

5.3.1.1 tryLock 和 lock 和 lockInterruptibly 的区别

1. tryLock 能获得锁就返回 true，不能就立即返回 false，tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2. lock 能获得锁就返回 true，不能的话一直等待获得锁
3. lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程， lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

5.3.2 AQS 框架

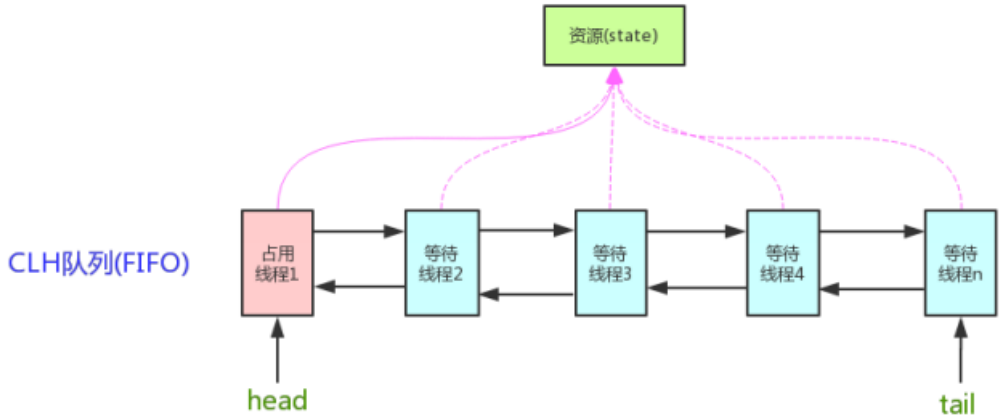
AbstractQueuedSynchronizer 类如其名，抽象的队列式的同步器，AQS 定义了一套多线程访问 共享资源的同步器框架，许多同步类实现都依赖于它，如常用的 ReentrantLock/Semaphore/CountDownLatch。

5.3.2.1 AQS 的原理

AQS 核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制 AQS 是用 CLH 队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。

AQS(AbstractQueuedSynchronizer)原理图：



AQS 维护了一个 volatile int state（代表共享资源）和一个 FIFO 线程等待队列（多线程争用资源被阻塞时会进入此队列）来完成获取资源线程的排队工作。AQS 使用 CAS 对该同步状态进行原子操作实现对其值的修改。这里 volatile 是核心关键词,实现了可见性。state 的访问方式有三种:

getState()
setState()
compareAndSetState()

```
//返回同步状态的当前值
protected final int getState() {
return state;
}
//  设置同步状态的值
protected final void setState(int newState) {
state = newState;
}
//原子地（CAS 操作）将同步状态值设置为给定值 update 如果当前同步状态的值等于 expect（期望值）
protected final boolean compareAndSetState(int expect, int update) {
return unsafe.compareAndSwapInt(this, stateOffset, expect, update);
}
```

5.3.2.2 AQS 定义两种资源共享方式

Exclusive（独占）：只有一个线程能执行，如 ReentrantLock。
Share（共享）：多个线程可同时执行，如 Semaphore/CountDownLatch。Semaphore、CountDownLatCh、 CyclicBarrier、ReentrantReadWriteLock 也就是读写锁允许多个线程同时对某一资源进行读。
不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在顶层实现好了。

5.3.3 ReentrantLock

使用 synchronized 来做同步处理时，锁的获取和释放都是隐式的，实现的原理是通过编译后加上不同的机器指令来实现。

而 ReentrantLock 就是一个普通的类，它是基于 AQS(AbstractQueuedSynchronizer)来实现的。
是一个**重入锁**：一个线程获得了锁之后仍然可以**反复**的加锁，不会出现自己阻塞自己的情况。
AQS 是 Java 并发包里实现锁、同步的一个重要的基础框架。

ReentantLock 继承接口 Lock 并实现了接口中定义的方法，他是一种可重入锁，除了能完 成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等 避免多线程死锁的方法。

5.3.4 ReadWriteLock 读写锁

为了提高性能，Java 提供了读写锁，在读的地方使用读锁，在写的地方使用写锁，灵活控制，如 果没有写锁的情况下，读是无阻塞的,在一定程度上提高了程序的执行效率。读写锁分为读锁和写 锁，多个读锁不互斥，读锁与写锁互斥，这是由 jvm 自己控制的，你只要上好相应的锁即可。

5.3.4.1 读锁

如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁

5.3.4.2 写锁

如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！ Java 中读写锁有个接口 java.util.concurrent.locks.ReadWriteLock，也有具体的实现 ReentrantReadWriteLock。

5.3.4.3 ReadWriteLock 读写锁

ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

ReadWriteLock 读取操作通常不会改变共享资源，但执行写入操作时，必须独占方式来获取锁。对于读取操作占多数的数据结构。

ReadWriteLock 能提供比独占锁更高的并发性。而对于只读的数据结构，其中包含的不变性可以完全不需要考虑加锁操作。

读写需要互斥、写写需要互斥、读读不需要互斥

```
public class TestReadWriteLock {

    public static void main(String[] args) {
        ReadWriteLockDemo rw = new ReadWriteLockDemo();

        new Thread(new Runnable() {

            @Override
            public void run() {
                rw.set((int)(Math.random() * 101));
            }
        }, "Write:").start();


        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {

                @Override
                public void run() {
                    rw.get();
                }
            }).start();
        }
    }
```

```
}

class ReadWriteLockDemo{

    private int number = 0;

    private ReadWriteLock lock = new ReentrantReadWriteLock();

    //读
    public void get(){
        lock.readLock().lock(); //上锁

        try{
            System.out.println(Thread.currentThread().getName() + " : " + number);
        }finally{
            lock.readLock().unlock(); //释放锁
        }
    }

    //写
    public void set(int number){
        lock.writeLock().lock();

        try{
            System.out.println(Thread.currentThread().getName());
            this.number = number;
        }finally{
            lock.writeLock().unlock();
        }
    }
}
```

5.4 synchronized 和 Lock 的比较

5.4.1 两者的共同点

1. 都是用来协调多线程对共享对象、变量的访问
2. 都是可重入锁，同一线程可以多次获得同一个锁
3. 都保证了可见性和互斥性

5.4.2 两者的不同点

1. ReentrantLock 显示的获得、释放锁，synchronized 隐式获得释放锁
2. ReentrantLock 相比 synchronized 的优势是可响应、可轮回、可中断（Lock 可以让等待锁的线程响应中断）、公平锁、多个锁，synchronized 等待的线程会一直等待下去，不能够响应中断
3. ReentrantLock 是 API 级别的，使用的同步非阻塞，采用的是乐观并发策略，synchronized 是 JVM 级别的是同步阻塞，使用的是悲观并发策略
4. ReentrantLock 可以实现公平锁 Lock lock=new ReentrantLock(true);//true 公平锁 false 非公平锁
   - 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
   - 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
5. ReentrantLock 通过 Condition 可以绑定多个条件
6. Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现。
7. synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生； 而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象， 因此使用 Lock 时需要在 finally 块中释放锁。
10. 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。
11. Lock 可以提高多个线程进行读操作的效率，既就是实现读写锁等。

6 线程池

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量，超出数量的线程排队等候，等其它线程执行完毕， 再从队列中取出任务来执行。他的**主要特点为：线程复用；控制最大并发数；管理线程**。

6.1 为什么要用线程池？

线程池提供了一个线程队列，队列中保存着所有等待状态的线程，避免了创建与销毁的额外开销，提高了响应的速度。线程池还维护一些基本统计信息，例如已完成任务的数量。使用线程池有很多好处：

- 降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。 当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

6.2 线程池体系结构

- 线程池的体系架构

```
*    java.util.concurrent.Executor : 负责线程的使用与调度的根接口
*        |--ExecutorService 子接口: 线程池的主要接口
*            |--ThreadPoolExecutor 线程池的实现类
*            |--ScheduledExecutorService 子接口：负责线程的调度
*                |--ScheduledThreadPoolExecutor ：继承 ThreadPoolExecutor， 实现 ScheduledExecutorService
*
```

6.2.1 ThreadPoolExecutor 的构造方法

```
public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
Executors.defaultThreadFactory(), defaultHandler);
}
```

1. corePoolSize：指定了线程池中的线程数量。
2. maximumPoolSize：指定了线程池中的最大线程数量。
3. keepAliveTime：当前线程池数量超过 corePoolSize 时，多余的空闲线程的存活时间，即多 次时间内会被销毁。
4. unit：keepAliveTime 的单位。
5. workQueue：任务队列，被提交但尚未被执行的任务。
6. threadFactory：线程工厂，用于创建线程，一般用默认的即可。
7. handler：拒绝策略，当任务太多来不及处理，如何拒绝任务。

6.2.1.1 拒绝策略

线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。

JDK 内置的拒绝策略如下：

- AbortPolicy ：直接抛出异常，阻止系统正常运行。
- CallerRunsPolicy ：只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的 任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。 3. DiscardOldestPolicy ： 丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再 次提交当前任务。
- DiscardPolicy ：该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢 失，这是最好的一种方案。

以上内置拒绝策略均实现了 RejectedExecutionHandler 接口，若以上策略仍无法满足实际 需要，完全可以自己扩展 RejectedExecutionHandler 接口。

6.3 四种线程池创建方式

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。

为了便于跨大量上下文使用，此类提供了很多可调整的参数和扩展钩子 (hook)。但是，强烈建议程序员使用较为方便的工具类 Executors 工厂方法 ：

- ✓ ExecutorService newFixedThreadPool()：创建固定大小的线程池，可以进行自动线程回收
- ✓ ExecutorService newCachedThreadPool()：缓存线程池，线程数量不固定可以根据需求自动的更改数量。
- ✓ ExecutorService newSingleThreadExecutor()：创建单个线程池。线程池中只有一个线程
- ✓ ScheduledExecutorService newScheduledThreadPool()：创建固定大小的线程，可以延迟或定时的执行任务。

6.3.1 new FixedThreadPool

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大 多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务， 则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何 线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之 前，池中的线程将一直存在。

6.3.2 new CachedThreadPool

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行 很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造 的线程（如果线程可用）。如果现有线程没有可用的，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。因此，长时间保持空闲的线程池不会使用任何资 源。

6.3.1 new SingleThreadExecutor

Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程）,这个线程 池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去！

6.3.2 new ScheduledThreadPool

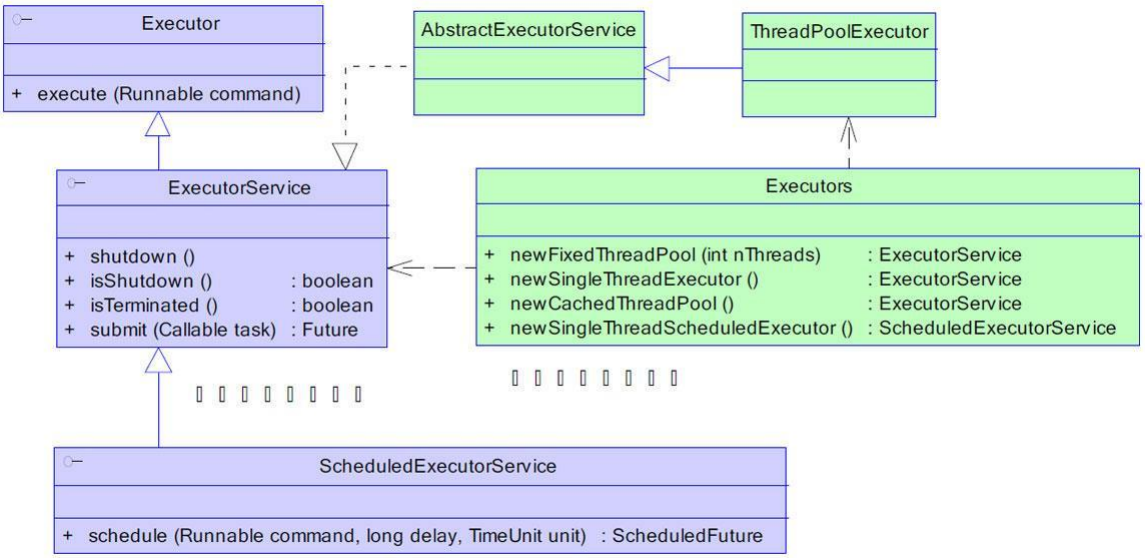
创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

6.4 线程池的组成

一般的线程池主要分为以下 4 个组成部分：

1. 线程池管理器：用于创建并管理线程池
2. 工作线程：线程池中的线程
3. 任务接口：每个任务必须实现的接口，用于工作线程调度其运行
4. 任务队列：用于存放待处理的任务，提供一种缓冲机制

Java 中的线程池是通过 Executor 框架实现的 ，该框架中用到了 Executor， Executors， ExecutorService，ThreadPoolExecutor ，Callable 和 Future、FutureTask 这几个类。



6.5 Java 线程池工作过程

1. 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面 有任务，线程池也不会马上执行它们。
2. 当调用 execute() 方法添加一个任务时，线程池会做如下判断：

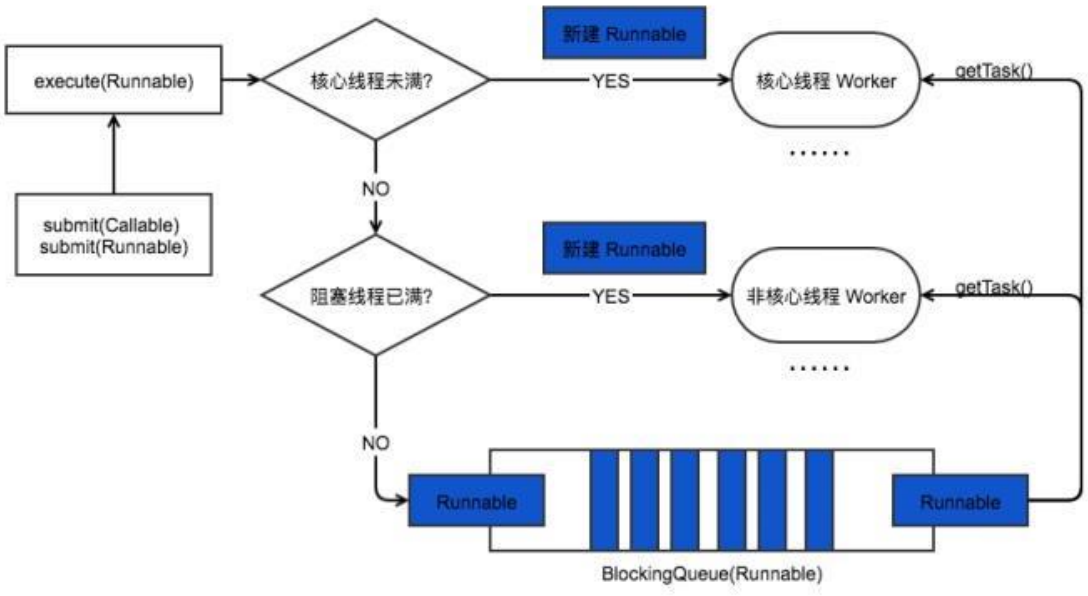
a) 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；

b) 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；

c) 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要 创建非核心线程立刻运行这个任务；

d) 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池 会抛出异常 RejectExecutionException。

3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断，如果当前运 行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它 最终会收缩到 corePoolSize 的大小。



7 常见的线程同步工具类

7.1 CountDownLatch 闭锁

CountDownLatch 是 java.util.concurrent 包下一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。利用它可以实现**类似计数器**的功能。比如有 一个任务 A，它要等待其他 4 个任务执行完毕之后才能执行，此时就可以利用 CountDownLatch 来实现这种功能了。

CountDownLatch 一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。以下的应用场景是可以使用闭锁来实现的

- 闭锁可以延迟线程的进度直到其到达终止状态
- 闭锁可以用来确保某些活动直到其他活动都完成才继续执行：
- 确保某个计算在其需要的所有资源都被初始化之后才继续执行;
- 确保某个服务在其依赖的所有其他服务都已经启动之后才启动;
- 等待直到某个操作所有参与者都准备就绪再继续执行。

```
/*
 * CountDownLatch ：闭锁，在完成某些运算时，只有其他所有线程的运算全部完成，当前运算才继续执行
 */
public class TestCountDownLatch {

    public static void main(String[] args) {
        final CountDownLatch latch = new CountDownLatch(50);
        LatchDemo ld = new LatchDemo(latch);

        long start = System.currentTimeMillis();

        for (int i = 0; i < 50; i++) {
            new Thread(ld).start();
        }

        try {
            latch.await();
        } catch (InterruptedException e) {
        }

        long end = System.currentTimeMillis();

        System.out.println("耗费时间为: " + (end - start));
    }
```

```
}

class LatchDemo implements Runnable {

    private CountDownLatch latch;

    public LatchDemo(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {

        try {
            for (int i = 0; i < 50000; i++) {
                if (i % 2 == 0) {
                    System.out.println(i);
                }
            }
        } finally {
            latch.countDown();
        }

    }

}
```

7.2 CyclicBarrier 循环栅栏

CyclicBarrier(循环栅栏)： CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。

CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier（回环栅栏-等待至 barrier 状态再全部同时执行）字面意思回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环 是因为当所有等待线程都被释放以后， CyclicBarrier 可以被重用。我们暂且把这个状态就叫做 barrier，当调用 await()方法之后，线程就处于 barrier 了。

CyclicBarrier 中最重要的方法就是 await 方法，它有 2 个重载版本：

1. public int await()：用来挂起当前线程，直至所有线程都到达 barrier 状态再同时执行后续任 务；
2. public int await(long timeout, TimeUnit unit)：让这些线程等待至一定的时间，如果还有 线程没有到达 barrier 状态就直接让到达 barrier 的线程执行后续任务。具体使用如下，另外 CyclicBarrier 是可以重用的

```
public class CyclicBarrierTest {
    public static void main(String[] args) {
        int N = 4;
        CyclicBarrier barrier = new CyclicBarrier(N);
        for (int i = 0; i < N; i++) {
            new Writer(barrier).start();
        }
    }

    static class Writer extends Thread {
        private CyclicBarrier cyclicBarrier;

        public Writer(CyclicBarrier cyclicBarrier) {
            this.cyclicBarrier = cyclicBarrier;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(5000);       //以睡眠来模拟线程需要预定写入数据操作
                System.out.println("线程" + Thread.currentThread().getName() + "写入数据完毕，等待其他线程写入完毕");
                cyclicBarrier.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
            System.out.println("所有线程写入完毕，继续处理其他任务，比如数据操作");
        }
    }
}
```

7.3 Semaphore 信号量

Semaphore(信号量)-允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

Semaphore 可以控制同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release()释放一个许可。

Semaphore 类中比较重要的几个方法：

1. public void acquire(): 用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许 可。
2. public void acquire(int permits):获取 permits 个许可
3. public void release() { } :释放许可。注意，在释放许可之前，必须先获获得许可。
4. public void release(int permits) { }:释放 permits 个许可

上面 4 个方法都会被阻塞，如果想立即得到执行结果，可以使用下面几个方法

Semaphore 是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信 号，做完自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore 可以用来 构建一些对象池，资源池之类的，比如数据库连接池

1. public boolean tryAcquire():尝试获取一个许可，若获取成功，则立即返回 true，若获取失 败，则立即返回 false
2. public boolean tryAcquire(long timeout, TimeUnit unit):尝试获取一个许可，若在指定的 时间内获取成功，则立即返回 true，否则则立即返回 false
3. public boolean tryAcquire(int permits):尝试获取 permits 个许可，若获取成功，则立即返 回 true，若获取失败，则立即返回 false
4. public boolean tryAcquire(int permits, long timeout, TimeUnit unit): 尝试获取 permits 个许可，若在指定的时间内获取成功，则立即返回 true，否则则立即返回 false
5. 还可以通过 availablePermits()方法得到可用的许可数目。

例子：若一个工厂有 5 台机器，但是有 8 个工人，一台机器同时只能被一个工人使用，只有使用完了，其他工人才能继续使用。那么我们就可以通过 Semaphore 来实现：

```
public class SemaphoreTest {
    public static void main(String[] args) {
        int N = 8;
        Semaphore semaphore = new Semaphore(5); //机器数目
        for (int i = 0; i < N; i++) {
            new Worker(i, semaphore).start();
        }
    }

    static class Worker extends Thread {
        private int num;
        private Semaphore semaphore;

        public Worker(int num, Semaphore semaphore) {
            this.num = num;
            this.semaphore = semaphore;
        }

        @Override
        public void run() {
            try {
                semaphore.acquire();
                System.out.println("工人" + this.num + "占用一个机器在生产...");
                Thread.sleep(2000);
                System.out.println("工人" + this.num + "释放出机器");
                semaphore.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

7.3.1 实现互斥锁（计数器为 1）

我们也可以创建计数为 1 的 Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量， 表示两种互斥状态。

它的用法如下：

```
// 创建一个计数阈值为 5 的信号量对象 // 只能 5 个线程同时访问
Semaphore semp = new Semaphore(5);
try {     // 申请许可
    semp.acquire();
    try {
        // 业务逻辑
    } catch (Exception e) {
    } finally {
    // 释放许可
    semp.release(); }
} catch (InterruptedException e) {

}
```

7.3.2 Semaphore 与 ReentrantLock

Semaphore 基本能完成 ReentrantLock 的所有工作，使用方法也与之类似，通过 acquire()与 release()方法来获得和释放临界资源。经实测，Semaphone.acquire()方法默认为可响应中断锁，与 ReentrantLock.lockInterruptibly()作用效果一致，也就是说在等待临界资源的过程中可以被 Thread.interrupt()方法中断。

此外，Semaphore 也实现了可轮询的锁请求与定时锁的功能，除了方法名 tryAcquire 与 tryLock 不同，其使用方法与 ReentrantLock 几乎一致。Semaphore 也提供了公平与非公平锁的机制，也 可在构造函数中进行设定。Semaphore 的锁释放操作也由手动进行，因此与 ReentrantLock 一样，为避免线程因抛出异常而无法正常释放锁的情况发生，释放锁的操作也必须在 finally 代码块中完成。

7.4 Condition 控制线程通信

Condition 接口描述了可能会与锁有关联的条件变量。这些变量在用法上与使用 Object.wait 访问的隐式监视器类似，但提供了更强大的功能。需要特别指出的是，单个 Lock 可能与多个 Condition 对象关联。为了避免兼容性问题，Condition 方法的名称与对应的 Object 版本中的不同。

在 Condition 对象中，与 wait、notify 和 notifyAll 方法对应的分别是 await、signal 和 signalAll。Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。