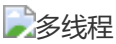
大纲图

 多线程

什么是线程和进程?

进程

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

比如：当我们启动 main 函数时其实就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。

线程

- 线程是一个比进程更小的执行单位
- 一个进程在其执行的过程中可以产生**多个线程**
- 与进程不同的是同类的多个线程共享进程的**堆**和**方法区**资源，但每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程

并发和并行的区别?

并发：同一时间段，多个任务都在执行 (单位时间内不一定同时执行)；

并行：单位时间内，多个任务同时执行。

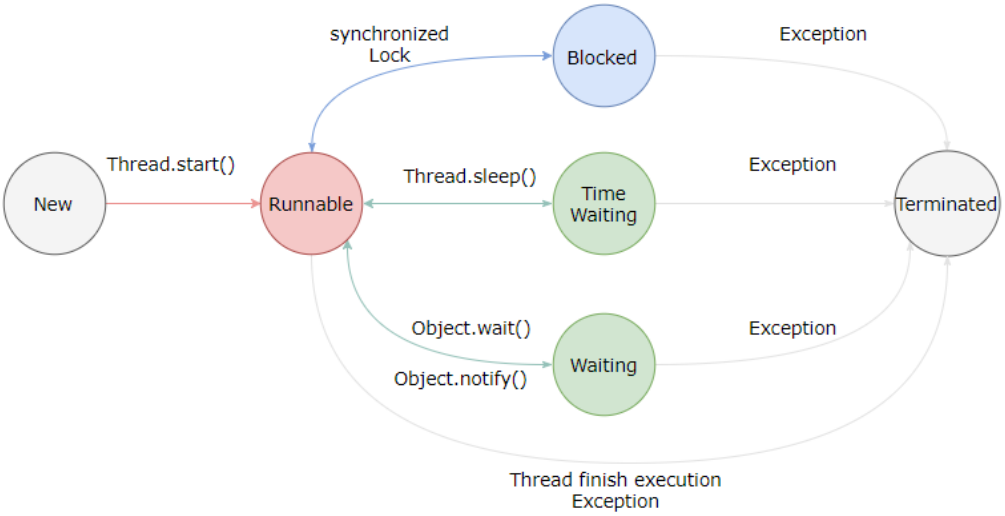
为什么使用多线程?

- **从计算机底层来说：** 线程可以比作是轻量级的进程，是程序执行的最小单位,线程间的切换和调度的成本远远小于进程。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
- **从当代互联网发展趋势来说：**现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。

使用多线程可能会带来什么问题?

可能会带来**内存泄漏**、**上下文切换**、**死锁**有受限于硬件和软件的资源闲置问题。

说说线程的生命周期?



线程创建之后它将处于`New`（新建）状态，调用`start()`方法后开始运行，线程这时候处于`READY`（可运行） 状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于`RUNNING`（运行） 状态。

当线程执行`wait()`方法之后，线程进入`WAITING`（等待） 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而`TIME_WAITING`(超时等待) 状态相当于在等待状态的基础上增加了超时限制，比如通过`sleep（long millis）`方法或`wait（long millis）`方法可以将 Java 线程置于`TIMED WAITING`状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到`BLOCKED`（阻塞）状态。线程在执行 Runnable 的`run()`方法之后将会进入到`TERMINATED`（终止） 状态。

什么是上下文切换?

多线程编程中一般**线程的个数都大于 CPU 核心的个数**，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配**时间片并轮转**的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

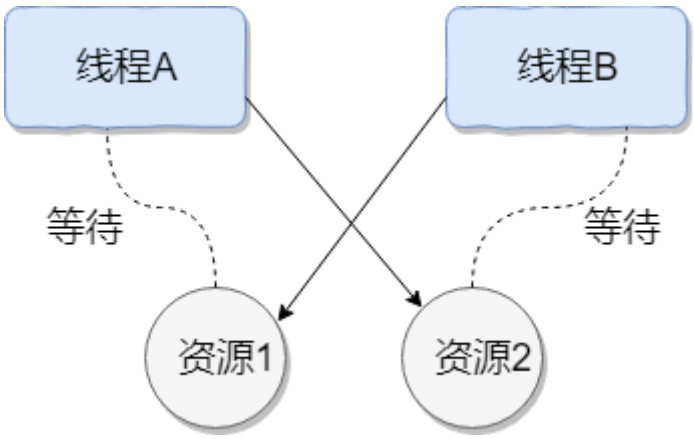
实际上就是**任务从保存到再加载的过程就是一次上下文切换。**

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

什么是死锁？如何避免死锁?

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



学过操作系统的朋友都知道产生死锁必须具备以下四个条件：

- 互斥条件：该资源任意一个时刻只由一个线程占用。(同一时刻，这个碗是我的，你不能碰)
- 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。（我拿着这个碗一直不放）
- 不剥夺条件:线程已获得的资源在未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。（我碗中的饭没吃完，你不能抢，释放权是我自己的，我想什么时候放就什么时候放）
- 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。（我拿了A碗，你拿了B碗，但是我还想要你的B碗，你还想我的A碗）

```
public class DeadLockDemo {
    private static Object resource1 = new Object();//资源 1
    private static Object resource2 = new Object();//资源 2

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get resource1");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource2");
                synchronized (resource2) {
                    System.out.println(Thread.currentThread() + "get
resource2");
                }
            }
        }, "线程 1").start();

        new Thread(() -> {
            synchronized (resource2) {
                System.out.println(Thread.currentThread() + "get resource2");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread() + "waiting get
resource1");
                synchronized (resource1) {
```

```
                    System.out.println(Thread.currentThread() + "get
resource1");
                }
            }
        }, "线程 2").start();
    }
}
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 `synchronized (resource1)` 获得 resource1 的监视器锁，然后通过 `Thread.sleep(1000);`让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

如何找到死锁

通过jdk常用的命令jsp和jstack，jsp查看java程序的id，jstack查看方法的栈信息等。

说说Sleep和Wait方法的区别

- 两者最主要的区别在于：**sleep 方法没有释放锁，而 wait 方法释放了锁 。**
- 两者都可以暂停线程的执行。
- Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。
- wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用wait(long timeout)超时后线程会自动苏醒。

Synchronzed

使用方式

- **修饰实例方法，作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁**
- **修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁 。**
- **修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。**

单例

```
public class Singleton {

    private volatile static Singleton uniqueInstance; // 第一步

    private Singleton() { // 第二步，私有
    }

    public static Singleton getUniqueInstance() {
       //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) { // 双重校验
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
```

```
        }
        return uniqueInstance;
    }
}
```

注意：**uniqueInstance 采用 volatile 关键字修饰也是很有必要**

uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

Synchronized 和 ReenTrantLock 的对比

1. **两者都是可重入锁**:两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。
2. **Synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API**:synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。
3. **ReenTrantLock 比 Synchronized 增加了一些高级功能**
    1. **等待可中断**：过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
    2. **可实现公平锁**
    3. **可实现选择性通知（锁可以绑定多个条件）**：线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”
    4. **性能已不是选择标准**：在jdk1.6之前synchronized 关键字吞吐量随线程数的增加，下降得非常严重。1.6之后，**synchronized 和 ReenTrantLock 的性能基本是持平了。**

底层原理

synchronized 关键字底层原理属于 JVM 层面。

1. **synchronized 同步语句块的情况**

    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。

2. **synchronized 修饰方法的的情况**

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 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。

1.6版本的优化

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。

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！

   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。

   升级过程：

   1. 访问Mark Word中偏向锁的标识是否设置成1，锁标识位是否为01，确认偏向状态
   2. 如果为可偏向状态，则判断当前线程ID是否为偏向线程
   3. 如果偏向线程未当前线程，则通过cas操作竞争锁，如果竞争成功则操作Mark Word中线程ID设置为当前线程ID
   4. 如果cas偏向锁获取失败，则挂起当前偏向锁线程，偏向锁升级为轻量级锁。

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。**

   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！

   升级过程：

   1. 线程由偏向锁升级为轻量级锁时，会先把锁的对象头MarkWord复制一份到线程的栈帧中，建立一个名为锁记录空间（Lock Record），用于存储当前Mark Word的拷贝。
   2. 虚拟机使用cas操作尝试将对象的Mark Word指向Lock Record的指针，并将Lock record里的owner指针指对象的Mark Word。
   3. 如果cas操作成功，则该线程拥有了对象的轻量级锁。第二个线程cas自旋锁等待锁线程释放锁。
   4. 如果多个线程竞争锁，轻量级锁要膨胀为重量级锁，Mark Word中存储的就是指向重量级锁（互斥量）的指针。其他等待线程进入阻塞状态。

3. **自旋锁和自适应自旋**

   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。**

   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。

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. **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。

6. 总结升级过程：

   1. 检测Mark Word里面是不是当前线程的ID，如果是，表示当前线程处于偏向锁
   2. 如果不是，则使用CAS将当前线程的ID替换Mard Word，如果成功则表示当前线程获得偏向锁，置偏向标志位1
   3. 如果失败，则说明发生竞争，撤销偏向锁，进而升级为轻量级锁。
   4. 当前线程使用CAS将对象头的Mark Word替换为锁记录指针，如果成功，当前线程获得锁
   5. 如果失败，表示其他线程竞争锁，当前线程便尝试使用自旋来获取锁。
   6. 如果自旋成功则依然处于轻量级状态。
   7. 如果自旋失败，则升级为重量级锁。

Volatile

Volatile的特性

1. **可见性**

   volatile的可见性是指当一个变量被volatile修饰后，这个变量就对所有线程均可见。白话点就是说当一个线程修改了一个volatile修饰的变量后，其他线程可以立刻得知这个变量的修改，拿到最这个变量最新的值。

```
public class VolatileVisibleDemo {

//    private  boolean isReady = true;
    private volatile boolean isReady = true;

    void m() {
        System.out.println(Thread.currentThread().getName() + " m
start...");
        while (isReady) {
        }
        System.out.println(Thread.currentThread().getName() + " m end...");
    }
```

```
    public static void main(String[] args) {
        VolatileVisibleDemo demo = new VolatileVisibleDemo();
        new Thread(() -> demo.m(), "t1").start();
        try {TimeUnit.SECONDS.sleep(1);} catch (InterruptedException e)
{e.printStackTrace();}
        demo.isReady = false; // 刚才一秒过后开始执行
    }
}
```

分析：一开始isReady为true，m方法中的while会一直循环，而主线程开启开线程之后会延迟1s将isReady赋值为false，若不加volatile修饰，则程序一直在运行，若加了volatile修饰，则程序最后会输出t1 m end...

2. **有序性**

有序性是指程序代码的执行是按照代码的实现顺序来按序执行的；volatile的有序性特性则是指禁止JVM指令重排优化。

```
public class Singleton {
    private static Singleton instance = null; // valotile
    //private static volatile Singleton instance = null;
    private Singleton() { } // 私有

    public static Singleton getInstance() { // 双重校验
        //第一次判断
        if(instance == null) {
            synchronized (Singleton.class) { // 加锁
                if(instance == null) {
                    //初始化，并非原子操作
                    instance = new Singleton(); // 这一行代码展开其实分三步走
                }
            }
        }
        return instance;
    }
}
```

上面的代码是一个很常见的单例模式实现方式，但是上述代码在多线程环境下是有问题的。为什么呢，问题出在instance对象的初始化上，因为`instance = new Singleton();`这个初始化操作并不是原子的，在JVM上会对应下面的几条指令：

```
memory =allocate();    //1. 分配对象的内存空间
ctorInstance(memory);  //2. 初始化对象
instance =memory;     //3. 设置instance指向刚分配的内存地址
```

上面三个指令中，步骤2依赖步骤1，但是步骤3不依赖步骤2，所以JVM可能针对他们进行指令重拍序优化，重排后的指令如下：

```
memory =allocate();    //1. 分配对象的内存空间
instance =memory;     //3. 设置instance指向刚分配的内存地址
ctorInstance(memory);  //2. 初始化对象
```

这样优化之后，内存的初始化被放到了instance分配内存地址的后面，这样的话当线程1执行步骤3这段赋值指令后，刚好有另外一个线程2进入getInstance方法判断instance不为null，这个时候线程2拿到的instance对应的内存其实还未初始化，这个时候拿去使用就会导致出错。

所以我们在用这种方式实现单例模式时，会使用volatile关键字修饰instance变量，这是因为volatile关键字除了可以保证变量可见性之外，还具有防止指令重排序的作用。当用volatile修饰instance之后，JVM执行时就不会对上面提到的初始化指令进行重排序优化，这样也就不会出现多线程安全问题了。

3. **不能保证原子性**

   volatile关键字能保证变量的可见性和代码的有序性，但是不能保证变量的原子性，下面我再举一个volatile与原子性的例子：

```
public class VolatileAtomicDemo {
    public static volatile int count = 0;

    public static void increase() {
        count++;
    }

    public static void main(String[] args) {
        Thread[] threads = new Thread[20];
        for(int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(() -> {
                for(int j = 0; j < 1000; j++) {
                    increase();
                }
            });
            threads[i].start();
        }
        //等待所有累加线程结束
        while (Thread.activeCount() > 1) {
            Thread.yield();
        }
        System.out.println(count);
    }
}
```

上面这段代码创建了20个线程，每个线程对变量count进行1000次自增操作，如果这段代码并发正常的话，结果应该是20000，但实际运行过程中经常会出现小于20000的结果，因为count++这个自增操作不是原子操作。看图

内存屏障

Java的Volatile的特征是任何读都能读到最新值，本质上是JVM通过内存屏障来实现的；为了实现volatile内存语义，JMM会分别限制重排序类型。下面是JMM针对编译器制定的volatile重排序规则表：

是否能重排序	第二个操作		
第一个操作	普通读/写	volatile读	volatile写
普通读/写			no
volatile读	no	no	no
volatile写		no	no

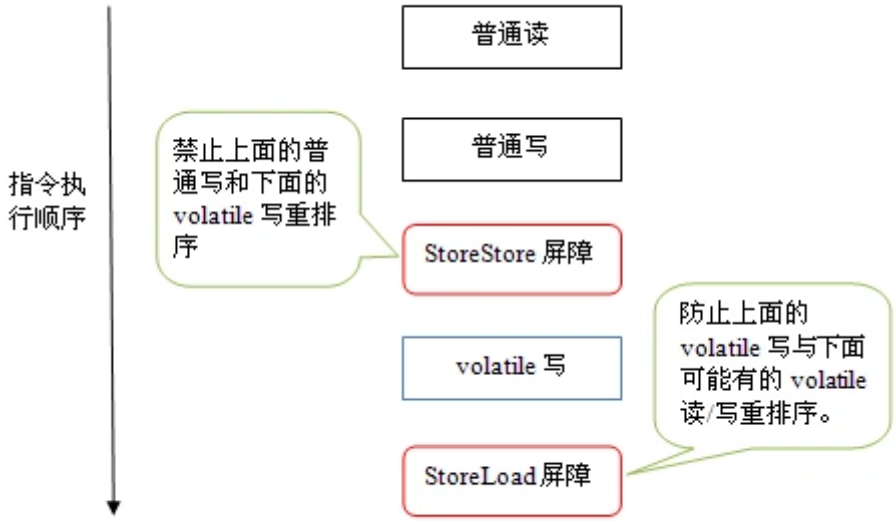
从上表我们可以看出：

- 当第二个操作是volatile写时，不管第一个操作是什么，都不能重排序。这个规则确保volatile写之前的操作不会被编译器重排序到volatile写之后。
- 当第一个操作是volatile读时，不管第二个操作是什么，都不能重排序。这个规则确保volatile读之后的操作不会被编译器重排序到volatile读之前。
- 当第一个操作是volatile写，第二个操作是volatile读时，不能重排序。

为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，JMM采取保守策略。下面是基于保守策略的JMM内存屏障插入策略：

- 在每个volatile写操作的前面插入一个StoreStore屏障。
- 在每个volatile写操作的后面插入一个StoreLoad屏障。
- 在每个volatile读操作的后面插入一个LoadLoad屏障。
- 在每个volatile读操作的后面插入一个LoadStore屏障。

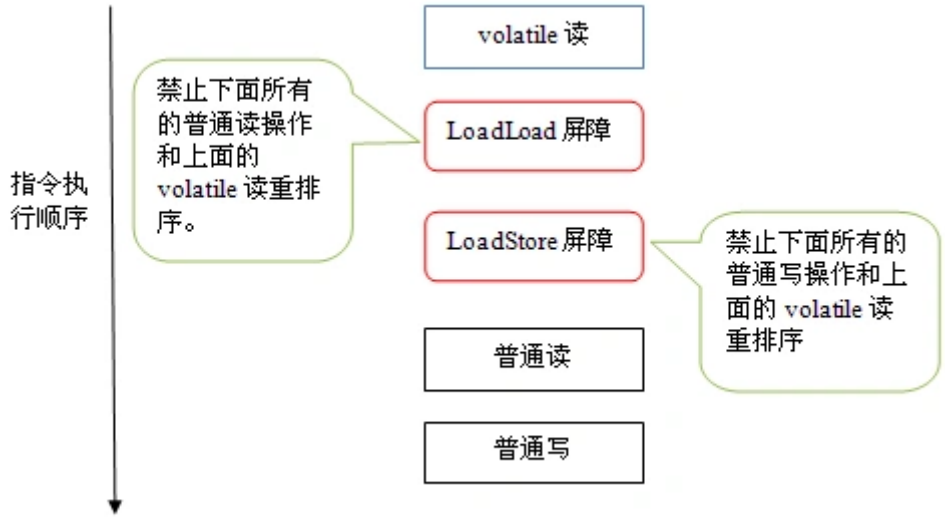
volatile写插入内存指令图：



上图中的StoreStore屏障可以保证在volatile写之前，其前面的所有普通写操作已经对任意处理器可见了。这是因为StoreStore屏障将保障上面所有的普通写在volatile写之前刷新到主内存。

这里比较有意思的是volatile写后面的StoreLoad屏障。这个屏障的作用是避免volatile写与后面可能有的volatile读/写操作重排序。因为编译器常常无法准确判断在一个volatile写的后面，是否需要插入一个StoreLoad屏障（比如，一个volatile写之后方法立即return）。为了保证能正确实现volatile的内存语义，JMM在这里采取了保守策略：在每个volatile写的后面或在每个volatile读的前面插入一个StoreLoad屏障。从整体执行效率的角度考虑，JMM选择了在每个volatile写的后面插入一个StoreLoad屏障。因为volatile写-读内存语义的常见使用模式是：一个写线程写volatile变量，多个读线程读同一个volatile变量。当读线程的数量大大超过写线程时，选择在volatile写之后插入StoreLoad屏障将带来可观的执行效率的提升。从这里我们可以看到JMM在实现上的一个特点：首先确保正确性，然后再去追求执行效率。

volatile读插入内存指令图：



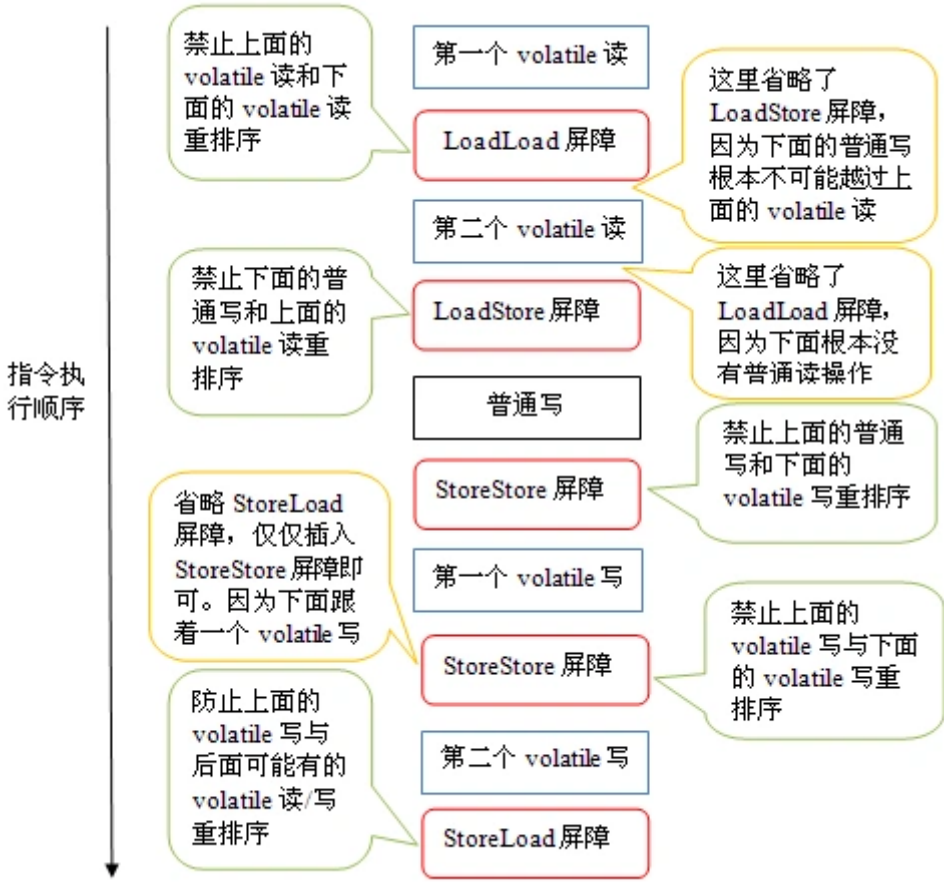
上图中的LoadLoad屏障用来禁止处理器把上面的volatile读与下面的普通读重排序。LoadStore屏障用来禁止处理器把上面的volatile读与下面的普通写重排序。

上述volatile写和volatile读的内存屏障插入策略非常保守。在实际执行时，只要不改变volatile写-读的内存语义，编译器可以根据具体情况省略不必要的屏障。下面我们通过具体的示例代码来说明：

```
class VolatileBarrierExample {
    int a;
    volatile int v1 = 1;
    volatile int v2 = 2;

    void readAndWrite() {
        int i = v1;           //第一个volatile读
        int j = v2;           // 第二个volatile读
        a = i + j;            //普通写
        v1 = i + 1;          // 第一个volatile写
        v2 = j * 2;          //第二个 volatile写
    }
}
```

针对readAndWrite()方法，编译器在生成字节码时可以做如下的优化：



注意，最后的StoreLoad屏障不能省略。因为第二个volatile写之后，方法立即return。此时编译器可能无法准确断定后面是否会有volatile读或写，为了安全起见，编译器常常会在这里插入一个StoreLoad屏障。

volatile汇编

```
0x0000000011214bb49: mov    %rdi,%rax
  0x0000000011214bb4c: dec    %eax
  0x0000000011214bb4e: mov    %eax,0x10(%rsi)
  0x0000000011214bb51: lock addl $0x0,(%rsp)     ;*putfield v1
                                                 ; -
com.earnfish.VolatileBarrierExample::readAndWrite@21 (line 35)

  0x0000000011214bb56: imul   %edi,%ebx
  0x0000000011214bb59: mov    %ebx,0x14(%rsi)
  0x0000000011214bb5c: lock addl $0x0,(%rsp)     ;*putfield v2
                                                 ; -
com.earnfish.VolatileBarrierExample::readAndWrite@28 (line 36)
```

```
v1 = i - 1;           // 第一个volatile写
v2 = j * i;           // 第二个volatile写
```

可见其本质是通过一个lock指令来实现的。那么lock是什么意思呢?

它的作用是使得本CPU的Cache写入了内存，该写入动作也会引起别的CPU invalidate其Cache。所以通过这样一个空操作，可让前面volatile变量的修改对其他CPU立即可见。

- 锁住内存
- 任何读必须在写完成之后再执行
- 使其它线程这个值的栈缓存失效

CAS

我们在读Concurrent包下的类的源码时，发现无论是 ReenterLock内部的AQS，还是各种Atomic开头的原子类，内部都应用到了`CAS`

```
public class Test {

    public AtomicInteger i;

    public void add() {
        i.getAndIncrement();
    }
}
```

我们来看`getAndIncrement`的内部：

```
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

再深入到`getAndAddInt`():

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}
```

现在重点来了，`compareAndSwapInt（var1, var2, var5, var5 + var4）`其实换成`compareAndSwapInt（obj, offset, expect, update）`比较清楚，意思就是如果`obj`内的`value`和`expect`相等，就证明没有其他线程改变过这个变量，那么就更新它为`update`，如果这一步的`CAS`没有成功，那就采用自旋的方式继续进行`CAS`操作，取出乍一看这也是两个步骤了啊，其实在`JNI`里是借助于一个`CPU`指令完成的。所以还是原子操作。

CAS底层

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe,
jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

p是取出的对象，addr是p中offset处的地址，最后调用了`Atomic::cmpxchg(x, addr, e)`，其中参数x是即将更新的值，参数e是原内存的值。代码中能看到cmpxchg有基于各个平台的实现。

ABA问题

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。

目前在JDK的atomic包里提供了一个类 `AtomicStampedReference` 来解决ABA问题。

```
public class ABADemo {
    static AtomicInteger atomicInteger = new AtomicInteger(100);
    static AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(100, 1);

    public static void main(String[] args) {
        System.out.println("=====ABA的问题产生=====");
        new Thread(() -> {
            atomicInteger.compareAndSet(100, 101);
            atomicInteger.compareAndSet(101, 100);
        }, "t1").start();

        new Thread(() -> {
            // 保证线程1完成一次ABA问题
            try { TimeUnit.SECONDS.sleep(1); } catch (InterruptedException e) {
e.printStackTrace(); }
            System.out.println(atomicInteger.compareAndSet(100, 2020) + " " +
atomicInteger.get());
        }, "t2").start();
        try { TimeUnit.SECONDS.sleep(2); } catch (InterruptedException e) {
e.printStackTrace(); }

        System.out.println("=====解决ABA的问题=====");
        new Thread(() -> {
            int stamp = atomicStampedReference.getStamp(); // 第一次获取版本号
            System.out.println(Thread.currentThread().getName() + " 第1次版本号" +
stamp);
            try { TimeUnit.SECONDS.sleep(2); } catch (InterruptedException e) {
e.printStackTrace(); }
            atomicStampedReference.compareAndSet(100, 101,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1);
            System.out.println(Thread.currentThread().getName() + "\t第2次版本号"
+ atomicStampedReference.getStamp());
            atomicStampedReference.compareAndSet(101, 100,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1);
            System.out.println(Thread.currentThread().getName() + "\t第3次版本号"
+ atomicStampedReference.getStamp());
        }, "t3").start();

        new Thread(() -> {
            int stamp = atomicStampedReference.getStamp();
            System.out.println(Thread.currentThread().getName() + "\t第1次版本号"
+ stamp);
            try { TimeUnit.SECONDS.sleep(4); } catch (InterruptedException e) {
e.printStackTrace(); }
            boolean result = atomicStampedReference.compareAndSet(100, 2020,
stamp, stamp + 1);
```

```
            System.out.println(Thread.currentThread().getName() + "\t修改是否成功"
+ result + "\t当前最新实际版本号：" + atomicStampedReference.getStamp());
            System.out.println(Thread.currentThread().getName() + "\t当前最新实际
值：" + atomicStampedReference.getReference());
        }, "t4").start();

    }
}
```

ThreadLocal

如果想实现每一个线程都有自己的专属本地变量该如何解决呢？ JDK中提供的`ThreadLocal`类正是为了解决这样的问题。 **`ThreadLocal`类主要解决的就是让每个线程绑定自己的值，可以将`ThreadLocal`类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。**如果你创建了一个`ThreadLocal`变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是`ThreadLocal`变量名的由来。他们可以使用`get()`和`set()`方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。**

原理

```
public class Thread implements Runnable {
 ......
//与此线程有关的ThreadLocal值。由ThreadLocal类维护
ThreadLocal.ThreadLocalMap threadLocals = null;

//与此线程有关的InheritableThreadLocal值。由InheritableThreadLocal类维护
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
 ......
}
```

从上面`Thread`类 源代码可以看出`Thread` 类中有一个 `threadLocals` 和 一个 `inheritableThreadLocals` 变量，它们都是 `ThreadLocalMap` 类型的变量,我们可以把 `ThreadLocalMap` 理解为`ThreadLocal` 类实现的定制化的 `HashMap`。默认情况下这两个变量都是 null，只有当前线程调用 `ThreadLocal` 类的 `set`或`get`方法时才创建它们，实际上调用这两个方法的时候，我们调用的是`ThreadLocalMap`类对应的 `get()`、`set()`方法。

`ThreadLocal`类的`set()`方法

```
    public void set(T value) {
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
    }
    ThreadLocalMap getMap(Thread t) {
        return t.threadLocals;
    }
```

最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，ThreadLocal可以理解为只是ThreadLocalMap的封装，传递了变量值。

每个Thread中都具备一个ThreadLocalMap，而ThreadLocalMap可以存储以ThreadLocal为key的键值对。 比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话，会使用 `Thread` 内部都是使用仅有那个`ThreadLocalMap` 存放数据的，`ThreadLocalMap` 的 key 就是 `ThreadLocal` 对象，value 就是 `ThreadLocal` 对象调用 `set` 方法设置的值。`ThreadLocal` 是 map结构是为了让每个线程可以关联多个 `ThreadLocal` 变量。这也就解释了ThreadLocal声明的变量为什么在每一个线程都有自己的专属本地变量。

内存泄露

`ThreadLocalMap` 中使用的 key 为 `ThreadLocal` 的弱引用,而 value 是强引用。所以，如果 `ThreadLocal` 没有被外部强引用的情况下，在垃圾回收的时候会 key 会被清理掉，而 value 不会被清理掉。这样一来，`ThreadLocalMap` 中就会出现key为null的Entry。假如我们不做任何措施的话，value 永远无法被GC 回收，这个时候就可能会产生内存泄露。ThreadLocalMap实现中已经考虑了这种情况，在调用 `set()`、`get()`、`remove()` 方法的时候，会清理掉 key 为 null 的记录。使用完 `ThreadLocal` 方法后 最好手动调用 `remove()` 方法

并发集合容器

为什么说ArrayList线程不安全?

看add方法的源码

```
public boolean add(E e) {

    /**
     * 添加一个元素时，做了如下两步操作
     * 1.判断列表的capacity容量是否足够，是否需要扩容
     * 2.真正将元素放在列表的元素数组里面
     */
    ensureCapacityInternal(size + 1);  // Increments modCount!! // 可能因为该操作，导致下一步发生数组越界
    elementData[size++] = e; // 可能null值
    return true;
}
```

数组越界

1. 列表大小为9，即size=9
2. 线程A开始进入add方法，这时它获取到size的值为9，调用ensureCapacityInternal方法进行容量判断。
3. 线程B此时也进入add方法，它获取到size的值也为9，也开始调用ensureCapacityInternal方法。
4. 线程A发现需求大小为10，而elementData的大小就为10，可以容纳。于是它不再扩容，返回。
5. 线程B也发现需求大小为10，也可以容纳，返回。
6. 线程A开始进行设置值操作， elementData[size++] = e 操作。此时size变为10。
7. 线程B也开始进行设置值操作，它尝试设置elementData[10] = e，而elementData没有进行过扩容，它的下标最大为9。于是此时会报出一个数组越界的异常ArrayIndexOutOfBoundsException.

null值情况

elementData[size++] = e不是一个原子操作:

1. elementData[size] = e;
2. size = size + 1;

逻辑:

1. 列表大小为0，即size=0
2. 线程A开始添加一个元素，值为A。此时它执行第一条操作，将A放在了elementData下标为0的位置上。
3. 接着线程B刚好也要开始添加一个值为B的元素，且走到了第一步操作。此时线程B获取到size的值依然为0，于是它将B也放在了elementData下标为0的位置上。
4. 线程A开始将size的值增加为1
5. 线程B开始将size的值增加为2

这样线程AB执行完毕后，理想中情况为size为2，elementData下标0的位置为A，下标1的位置为B。而实际情况变成了size为2，elementData下标为0的位置变成了B，下标1的位置上什么都没有。并且后续除非使用set方法修改此位置的值，否则将一直为null，因为size为2，添加元素时会从下标为2的位置上开始。

解决非安全集合的并发都有哪些?

ArrayList->Vector->SynchronizedList->CopyOnWriteArrayList

ArraySet->SynchronizedSet->CopyOnWriteArraySet

HashMap->SynchronizedMap->ConcurrentHashMap

并发同步容器

AQS原理

AQS 使用一个 int 成员变量来表示同步状态，通过内置的 FIFO 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。

```
private volatile int state;//共享变量，使用volatile修饰保证线程可见性
```

状态信息通过 protected 类型的`getState`，`setState`，`compareAndSetState`进行操作

```
//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);
}
```

AQS 定义两种资源共享方式

1. **Exclusive**（独占）只有一个线程能执行，如 ReentrantLock。又可分为公平锁和非公平锁,ReentrantLock 同时支持两种锁。

   总结：公平锁和非公平锁只有两处不同：

   1. 非公平锁在调用 lock 后，首先就会调用 CAS 进行一次抢锁，如果这个时候恰巧锁没有被占用，那么直接就获取到锁返回了。
   2. 非公平锁在 CAS 失败后，和公平锁一样都会进入到 tryAcquire 方法，在 tryAcquire 方法中，如果发现锁这个时候被释放了（state == 0），非公平锁会直接 CAS 抢锁，但是公平锁会判断等待队列是否有线程处于等待状态，如果有则不去抢锁，乖乖排到后面。

2. **Share**（共享）多个线程可同时执行，如 Semaphore/CountDownLatch。Semaphore、CyclicBarrier、ReadWriteLock 。

AQS 使用了模板方法模式，自定义同步器时需要重写下面几个 AQS 提供的模板方法：

```
isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。
```

CountDownLatch

CountDownLatch是共享锁的一种实现,它默认构造 AQS 的 state 值为 count。当线程使用countDown方法时,其实使用了`tryReleaseShared`方法以CAS的操作来减少state,直至state为0就代表所有的线程都调用了countDown方法。当调用await方法的时候，如果state不为0，就代表仍然有线程没有调用countDown方法，那么就把已经调用过countDown的线程都放入阻塞队列Park,并自旋CAS判断state == 0，直至最后一个线程调用了countDown，使得state == 0，于是阻塞的线程便判断成功，全部往下执行。

三种用法：

1. 某一线程在开始运行前等待 n 个线程执行完毕。将 CountDownLatch 的计数器初始化为 n ：`new CountDownLatch(n)`，每当一个任务线程执行完毕，就将计数器减 1 `countdownlatch.countDown()`，当计数器的值变为 0 时，在`CountDownLatch上 await()` 的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。
2. 实现多个线程开始执行任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的 `CountDownLatch` 对象，将其计数器初始化为 1 ： `new CountDownLatch(1)`，多个线程在开始执行任务前首先 `coundownlatch.await()`，当主线程调用 countDown() 时，计数器变为 0，多个线程同时被唤醒。
3. 死锁检测：一个非常方便的使用场景是，你可以使用 n 个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁。

```
public class CountDownLatchDemo {

    public static void main(String[] args) throws InterruptedException {
        countDownLatchTest();
//        general();
    }

    public static void general() {
        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + " 上完自习，离开教师");
            }, "Thread --> " + i).start();
        }
        while (Thread.activeCount() > 2) {
            try { TimeUnit.SECONDS.sleep(2); } catch (InterruptedException e) { e.printStackTrace(); }
            System.out.println(Thread.currentThread().getName() + " ====班长最后走人");
```

```
        }
    }

    public static void countDownLatchTest() throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(6);
        for (int i = 0; i < 6; i++) {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + " 上完自习，
离开教师");
                countDownLatch.countDown();
            }, "Thread --> " + i).start();
        }
        countDownLatch.await();
        System.out.println(Thread.currentThread().getName() + " ====班长最后走人");
    }
}
```

CyclicBarrier

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。

```
public class CyclicBarrierDemo {
    public static void main(String[] args) {
        cyclicBarrierTest();
    }

    public static void cyclicBarrierTest() {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(7, () -> {
            System.out.println("====召唤神龙====");
        });
        for (int i = 0; i < 7; i++) {
            final int tempInt = i;
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName() + " 收集到第"
+ tempInt + "颗龙珠");
                try {
                    cyclicBarrier.await();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }, "" + i).start();
        }
    }
}
```

当调用 `CyclicBarrier` 对象调用 `await()` 方法时，实际上调用的是`dowait(false, 0L)`方法。`await()` 方法就像树立起一个栅栏的行为一样，将线程挡住了，当拦住的线程数量达到 parties 的值时，栅栏才会打开，线程才得以通过执行。

```
    // 当线程数量或者请求数量达到 count 时 await 之后的方法才会被执行。上面的示例中 count
的值就为 5。
    private int count;
    /**
     * Main barrier code, covering the various policies.
     */
    private int dowait(boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
               TimeoutException {
        final ReentrantLock lock = this.lock;
        // 锁住
        lock.lock();
        try {
            final Generation g = generation;

            if (g.broken)
                throw new BrokenBarrierException();

            // 如果线程中断了，抛出异常
            if (Thread.interrupted()) {
                breakBarrier();
                throw new InterruptedException();
            }
            // cout减1
            int index = --count;
            // 当 count 数量减为 0 之后说明最后一个线程已经到达栅栏了，也就是达到了可以执行
await 方法之后的条件
            if (index == 0) {  // tripped
                boolean ranAction = false;
                try {
                    final Runnable command = barrierCommand;
                    if (command != null)
                        command.run();
                    ranAction = true;
                    // 将 count 重置为 parties 属性的初始化值
                    // 唤醒之前等待的线程
                    // 下一波执行开始
                    nextGeneration();
                    return 0;
                } finally {
                    if (!ranAction)
                        breakBarrier();
                }
            }

            // loop until tripped, broken, interrupted, or timed out
            for (;;) {
                try {
                    if (!timed)
                        trip.await();
                    else if (nanos > 0L)
                        nanos = trip.awaitNanos(nanos);
                } catch (InterruptedException ie) {
                    if (g == generation && ! g.broken) {
                        breakBarrier();
                        throw ie;
                    } else {
                        // We're about to finish waiting even if we had not
```

```
                        // been interrupted, so this interrupt is deemed to
                        // "belong" to subsequent execution.
                        Thread.currentThread().interrupt();
                    }
                }

                if (g.broken)
                    throw new BrokenBarrierException();

                if (g != generation)
                    return index;

                if (timed && nanos <= 0L) {
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

总结：`CyclicBarrier` 内部通过一个 count 变量作为计数器，count 的初始值为 parties 属性的初始化值，每当一个线程到了栅栏这里了，那么就将计数器减一。如果 count 值为 0 了，表示这是这一代最后一个线程到达栅栏，就尝试执行我们构造方法中输入的任务。

Semaphore

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。

```
public class SemaphoreDemo {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(3);// 模拟三个停车位
        for (int i = 0; i < 6; i++) { // 模拟6部汽车
            new Thread(() -> {
                try {
                    semaphore.acquire();
                    System.out.println(Thread.currentThread().getName() + " 抢到
车位");
                    // 停车3s
                    try { TimeUnit.SECONDS.sleep(3); } catch
(InterruptedException e) { e.printStackTrace(); }
                    System.out.println(Thread.currentThread().getName() + " 停车
3s后离开车位");
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release();
                }
            }, "Car " + i).start();
        }
    }
}
```

阻塞队列

- ArrayBlockingQueue:由数组结构组成的有界阻塞队列.
- LinkedBlockingQueue:由链表结构组成的有界(但大小默认值Integer>MAX_VALUE)阻塞队列.
- PriorityBlockingQueue:支持优先级排序的无界阻塞队列.
- DelayQueue:使用优先级队列实现的延迟无界阻塞队列.
- SynchronousQueue:不存储元素的阻塞队列,也即是单个元素的队列.
- LinkedTransferQueue:由链表结构组成的无界阻塞队列.
- LinkedBlockingDuque:由链表结构组成的双向阻塞队列.
- 抛出异常方法：add remove
- 不抛异常：offer poll
- 阻塞 put take
- 带时间 offer poll

生产者消费者

synchronized版本的生产者和消费者，比较繁琐

```
public class ProdConsumerSynchronized {

    private final LinkedList<String> lists = new LinkedList<>();

    public synchronized void put(String s) {
        while (lists.size() != 0) { // 用while怕有存在虚拟唤醒线程
            // 满了， 不生产了
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        lists.add(s);
        System.out.println(Thread.currentThread().getName() + " " +
lists.peekFirst());
        this.notifyAll(); // 这里可是通知所有被挂起的线程，包括其他的生产者线程
    }

    public synchronized void get() {
        while (lists.size() == 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(Thread.currentThread().getName() + " " +
lists.removeFirst());
        this.notifyAll(); // 通知所有被wait挂起的线程  用notify可能就死锁了。
    }

    public static void main(String[] args) {
        ProdConsumerSynchronized prodConsumerSynchronized = new
ProdConsumerSynchronized();

        // 启动消费者线程
        for (int i = 0; i < 5; i++) {
```

```
            new Thread(prodConsumerSynchronized::get, "ConsA" + i).start();
        }

        // 启动生产者线程
        for (int i = 0; i < 5; i++) {
            int tempI = i;
            new Thread(() -> {
                prodConsumerSynchronized.put("" + tempI);
            }, "ProdA" + i).start();
        }
    }
}
```

ReentrantLock

```
public class ProdConsumerReentrantLock {

    private LinkedList<String> lists = new LinkedList<>();

    private Lock lock = new ReentrantLock();

    private Condition prod = lock.newCondition();

    private Condition cons = lock.newCondition();

    public void put(String s) {
        lock.lock();
        try {
            // 1. 判断
            while (lists.size() != 0) {
                // 等待不能生产
                prod.await();
            }
            // 2.干活
            lists.add(s);
            System.out.println(Thread.currentThread().getName() + " " +
lists.peekFirst());
            // 3. 通知
            cons.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void get() {
        lock.lock();
        try {
            // 1. 判断
            while (lists.size() == 0) {
                // 等待不能消费
                cons.await();
            }
            // 2.干活
            System.out.println(Thread.currentThread().getName() + " " +
lists.removeFirst());
```

```
            // 3. 通知
            prod.signalAll();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        ProdConsumerReentrantLock prodConsumerReentrantLock = new
ProdConsumerReentrantLock();
        for (int i = 0; i < 5; i++) {
            int tempI = i;
            new Thread(() -> {
                prodConsumerReentrantLock.put(tempI + "");
            }, "ProdA" + i).start();
        }
        for (int i = 0; i < 5; i++) {
            new Thread(prodConsumerReentrantLock::get, "ConsA" + i).start();
        }
    }
}
```

BlockingQueue

```
public class ProdConsumerBlockingQueue {

    private volatile boolean flag = true;

    private AtomicInteger atomicInteger = new AtomicInteger();

    BlockingQueue<String> blockingQueue = null;

    public ProdConsumerBlockingQueue(BlockingQueue<String> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }

    public void myProd() throws Exception {
        String data = null;
        boolean retValue;
        while (flag) {
            data = atomicInteger.incrementAndGet() + "";
            retValue = blockingQueue.offer(data, 2, TimeUnit.SECONDS);
            if (retValue) {
                System.out.println(Thread.currentThread().getName() + " 插入队列"
+ data + " 成功");
            } else {
                System.out.println(Thread.currentThread().getName() + " 插入队列"
+ data + " 失败");
            }
            TimeUnit.SECONDS.sleep(1);
        }
        System.out.println(Thread.currentThread().getName() + " 大老板叫停了，
flag=false，生产结束");
    }
```

```
    public void myConsumer() throws Exception {
        String result = null;
        while (flag) {
            result = blockingQueue.poll(2, TimeUnit.SECONDS);
            if (null == result || result.equalsIgnoreCase("")) {
                flag = false;
                System.out.println(Thread.currentThread().getName() + " 超过2s没有
取到蛋糕，消费退出");
                return;
            }
            System.out.println(Thread.currentThread().getName() + " 消费队列" +
result + "成功");
        }
    }

    public void stop() {
        flag = false;
    }

    public static void main(String[] args) {
        ProdConsumerBlockingQueue prodConsumerBlockingQueue = new
ProdConsumerBlockingQueue(new ArrayBlockingQueue<>(10));
        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + " 生产线程启动");
            try {
                prodConsumerBlockingQueue.myProd();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }, "Prod").start();

        new Thread(() -> {
            System.out.println(Thread.currentThread().getName() + " 消费线程启动");
            try {
                prodConsumerBlockingQueue.myConsumer();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }, "Consumer").start();

        try { TimeUnit.SECONDS.sleep(5); } catch (InterruptedException e) {
e.printStackTrace(); }
        System.out.println("5s后main叫停，线程结束");
        prodConsumerBlockingQueue.stop();
    }
}
```

线程池

线程池的好处

- **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。

FixedThreadPool

`FixedThreadPool` 被称为可重用固定线程数的线程池。通过 Executors 类中的相关源代码来看一下相关实现：

```
    /**
     * 创建一个可重用固定数量线程的线程池
     */
    public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
        return new ThreadPoolExecutor(nThreads, nThreads,
                                      0L, TimeUnit.MILLISECONDS,
                                      new LinkedBlockingQueue<Runnable>(),
                                      threadFactory);
    }
```

从上面源代码可以看出新创建的 `FixedThreadPool` 的 `corePoolSize` 和 `maximumPoolSize` 都被设置为 nThreads，这个 nThreads 参数是我们使用的时候自己传递的。

1. 如果当前运行的线程数小于 corePoolSize， 如果再来新任务的话，就创建新的线程来执行任务；
2. 当前运行的线程数等于 corePoolSize 后， 如果再来新任务的话，会将任务加入 `LinkedBlockingQueue`；
3. 线程池中的线程执行完 手头的任务后，会在循环中反复从 `LinkedBlockingQueue` 中获取任务来执行；

不推荐使用

`FixedThreadPool` 使用无界队列 `LinkedBlockingQueue`（队列的容量为 Intger.MAX_VALUE）作为线程池的工作队列会对线程池带来如下影响：

1. 当线程池中的线程数达到 `corePoolSize` 后，新任务将在无界队列中等待，因此线程池中的线程数不会超过 corePoolSize；
2. 由于使用无界队列时 `maximumPoolSize` 将是一个无效参数，因为不可能存在任务队列满的情况。所以，通过创建 `FixedThreadPool` 的源码可以看出创建的 `FixedThreadPool` 的 `corePoolSize` 和 `maximumPoolSize` 被设置为同一个值。
3. 由于 1 和 2，使用无界队列时 `keepAliveTime` 将是一个无效参数；
4. 运行中的 `FixedThreadPool`（未执行 `shutdown()` 或 `shutdownNow()`）不会拒绝任务，在任务比较多的时候会导致 OOM（内存溢出）。

SingleThreadExecutor

```
    /**
     *返回只有一个线程的线程池
     */
    public static ExecutorService newSingleThreadExecutor(ThreadFactory
threadFactory) {
        return new FinalizableDelegatedExecutorService
            (new ThreadPoolExecutor(1, 1,
                                    0L, TimeUnit.MILLISECONDS,
                                    new LinkedBlockingQueue<Runnable>(),
                                    threadFactory));
    }
```

和上面一个差不多，只不过core和max都被设置为1

CachedThreadPool

```
    /**
     * 创建一个线程池，根据需要创建新线程，但会在先前构建的线程可用时重用它。
     */
    public static ExecutorService newCachedThreadPool(ThreadFactory
threadFactory) {
        return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                      60L, TimeUnit.SECONDS,
                                      new SynchronousQueue<Runnable>(),
                                      threadFactory);
    }
```

`CachedThreadPool` 的 `corePoolSize` 被设置为空（0），`maximumPoolSize` 被设置为 Integer.MAX.VALUE，即它是无界的，这也就意味着如果主线程提交任务的速度高于 `maximumPool` 中线程处理任务的速度时，`CachedThreadPool` 会不断创建新的线程。极端情况下，这样会导致耗尽 cpu 和内存资源。

1. 首先执行 `SynchronousQueue.offer(Runnable task)` 提交任务到任务队列。如果当前 `maximumPool` 中有闲线程正在执行 `SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)`，那么主线程执行 offer 操作与空闲线程执行的 `poll` 操作配对成功，主线程把任务交给空闲线程执行，`execute()` 方法执行完成，否则执行下面的步骤 2；
2. 当初始 `maximumPool` 为空，或者 `maximumPool` 中没有空闲线程时，将没有线程执行 `SynchronousQueue.poll(keepAliveTime,TimeUnit.NANOSECONDS)`。这种情况下，步骤 1 将失败，此时 `CachedThreadPool` 会创建新线程执行任务，execute 方法执行完成；

ScheduledThreadPoolExecutor省略，基本不会用

ThreadPoolExecutor（重点）

```
    /**
     * 用给定的初始参数创建一个新的ThreadPoolExecutor。
     */
    public ThreadPoolExecutor(int corePoolSize,//线程池的核心线程数量
                              int maximumPoolSize,//线程池的最大线程数
                              long keepAliveTime,//当线程数大于核心线程数时，多余的空闲
线程存活的最长时间
                              TimeUnit unit,//时间单位
                              BlockingQueue<Runnable> workQueue,//任务队列，用来储存
等待执行任务的队列
                              ThreadFactory threadFactory,//线程工厂，用来创建线程，
一般默认即可
                              RejectedExecutionHandler handler//拒绝策略，当提交的任
务过多而不能及时处理时，我们可以定制策略来处理任务
                               ) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler == null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
```

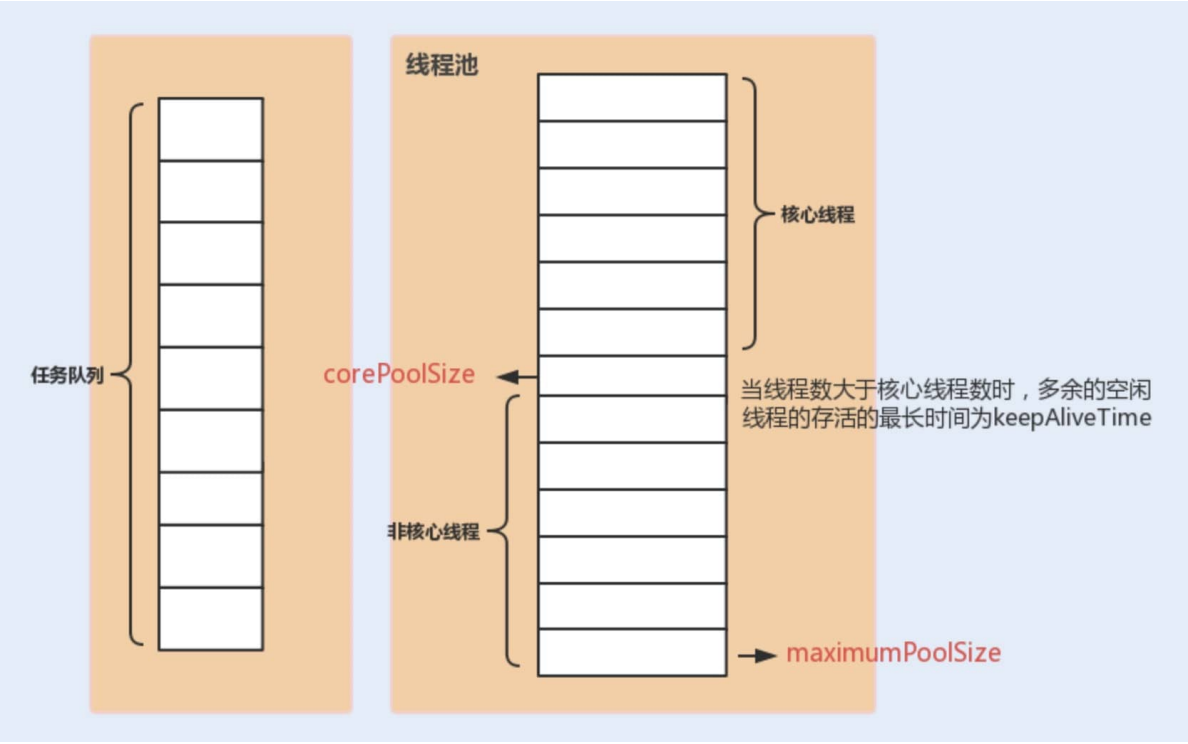
```
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

`ThreadPoolExecutor` **3 个最重要的参数:**

- `corePoolSize` **:** 核心线程数线程数定义了最小可以同时运行的线程数量。
- `maximumPoolSize` **:** 当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- `workQueue`**:** 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，信任就会被存放在队列中。

`ThreadPoolExecutor` 其他常见参数:

- `keepAliveTime`:当线程池中的线程数量大于 `corePoolSize` 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 `keepAliveTime`才会被回收销毁;
- `unit` : `keepAliveTime` 参数的时间单位。
- `threadFactory` :executor 创建新线程的时候会用到。
- `handler` :饱和策略。关于饱和策略下面单独介绍一下.



如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，`ThreadPoolTaskExecutor` 定义一些策略:

- `ThreadPoolExecutor.AbortPolicy`: 抛出 `RejectedExecutionException`来拒绝新任务的处理。
- `ThreadPoolExecutor.CallerRunsPolicy`: 调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy`: 不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy`: 此策略将丢弃最早的未处理的任务请求。

> Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 饱和策略的话来配置线程池的时候默认使用的是 `ThreadPoolExecutor.AbortPolicy`。在默认情况下，`ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 来拒绝新来的任务 ，这代表你将丢失对这个任务的处理。对于可伸缩的应用程序，建议使用 `ThreadPoolExecutor.CallerRunsPolicy`。当最大池被填满时，此策略为我们提供可伸缩队列。（这个直接查看 `ThreadPoolExecutor` 的构造函数源码就可以看出，比较简单的原因，这里就不贴代码了。）

> Executors 返回线程池对象的弊端如下：
>
> - `FixedThreadPool` 和 `SingleThreadExecutor` ： 允许请求的队列长度为 Integer.MAX_VALUE,可能堆积大量的请求，从而导致 OOM。
> - **CachedThreadPool 和 ScheduledThreadPool** ： 允许创建的线程数量为 Integer.MAX_VALUE ， 可能会创建大量线程，从而导致 OOM。

```
public class ThreadPoolExecutorDemo {
    public static void main(String[] args) {
        ExecutorService threadpools = new ThreadPoolExecutor(
                3,
                5,
                1l,
                TimeUnit.SECONDS,
                new LinkedBlockingDeque<>(3),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy());
//new ThreadPoolExecutor.AbortPolicy();
//new ThreadPoolExecutor.CallerRunsPolicy();
//new ThreadPoolExecutor.DiscardOldestPolicy();
//new ThreadPoolExecutor.DiscardPolicy();
        try {
            for (int i = 0; i < 8; i++) {
                threadpools.execute(() -> {
                    System.out.println(Thread.currentThread().getName() + "\t办理业务");
                });
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            threadpools.shutdown();
        }
    }
}
```

Java锁机制

公平锁/非公平锁

公平锁指多个线程按照申请锁的顺序来获取锁。非公平锁指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象（很长时间都没获取到锁-非洲人...），ReentrantLock，了解一下。

可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁，典型的synchronized，了解一下

```
synchronized void setA() throws Exception {
    Thread.sleep(1000);
    setB(); // 因为获取了setA()的锁，此时调用setB()将会自动获取setB()的锁，如果不自动获取的话方法B将不会执行
}
synchronized void setB() throws Exception {
    Thread.sleep(1000);
}
```

独享锁/共享锁

- 独享锁：是指该锁一次只能被一个线程所持有。
- 共享锁：是该锁可被多个线程所持有。

互斥锁/读写锁

上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是其具体的实现

乐观锁/悲观锁

1. **乐观锁与悲观锁不是指具体的什么类型的锁，而是指看待兵法同步的角度。**
2. **悲观锁认为对于同一个人数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观的认为，不加锁的并发操作一定会出现问题。**
3. **乐观锁则认为对于同一个数据的并发操作，是不会发生修改的。在更新数据的时候，会采用尝试更新，不断重新的方式更新数据。乐观的认为，不加锁的并发操作时没有事情的。**
4. **悲观锁适合写操作非常多的场景，乐观锁适合读操作非常多的场景，不加锁带来大量的性能提升。**
5. **悲观锁在Java中的使用，就是利用各种锁。乐观锁在Java中的使用，是无锁编程，常常采用的是CAS算法，典型的例子就是原子类，通过CAS自旋实现原子类操作的更新。重量级锁是悲观锁的一种，自旋锁、轻量级锁与偏向锁属于乐观锁**

分段锁

1. **分段锁其实是一种锁的设计，并不是具体的一种锁，对于ConcurrentHashMap而言，其并发的实现就是通过分段锁的形式来哦实现高效的并发操作。**
2. **以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap（JDK7与JDK8中HashMap的实现）的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是ReentrantLock（Segment继承了ReentrantLock）**
3. **当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。但是，在统计size的时候，可就是获取hashmap全局信息的时候，就需要获取所有的分段锁才能统计。**
4. **分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。**

偏向锁/轻量级锁/重量级锁

1. **这三种锁是锁的状态，并且是针对Synchronized。在Java5通过引入锁升级的机制来实现高效Synchronized。这三种锁的状态是通过对象监视器在对象头中的字段来表明的。偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。**
2. **偏向锁的适用场景：始终只有一个线程在执行代码块，在它没有执行完释放锁之前，没有其它线程去执行同步快，在锁无竞争的情况下使用，一旦有了竞争就升级为轻量级锁，升级为轻量级锁的时候需要撤销偏向锁，撤销偏向锁的时候会导致stop the word操作；在有锁竞争时，偏向锁会多做很多额外操作，尤其是撤销偏向锁的时候会导致进入安全点，安全点会导致stw，导致性能下降，这种情况下应当禁用。**
3. **轻量级锁是指当锁是偏向锁的时候，被另一个线程锁访问，偏向锁就会升级为轻量级锁，其他线程会通过自选的形式尝试获取锁，不会阻塞，提高性能。**
4. **重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让其他申请的线程进入阻塞，性能降低。**

自旋锁

1. **在Java中，自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。**
2. **自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。**
3. **自旋锁尽可能的减少线程的阻塞，适用于锁的竞争不激烈，且占用锁时间非常短的代码来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗。**
4. **但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适用使用自旋锁了，因为自旋锁在获取锁前一直都是占用cpu做无用功，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要cpu的线程又不能获取到cpu，造成cpu的浪费。**

Java锁总结

Java锁机制可归为Sychornized锁和Lock锁两类。Synchronized是基于JVM来保证数据同步的，而Lock则是硬件层面，依赖特殊的CPU指令来实现数据同步的。

- Synchronized是一个非公平、悲观、独享、互斥、可重入的重量级锁。
- ReentrantLock是一个默认非公平但可实现公平的、悲观、独享、互斥、可重入、重量级锁。
- ReentrantReadWriteLock是一个默认非公平但可实现公平的、悲观、写独享、读共享、读写、可重入、重量级锁。