Java 程序设计进阶-网课

作者

2019

目录

第一章　线程上

课前问题：

1. 程序代码一般都是顺序执行的，如何让程序在运行过程中，实现多个代码段的同时运行？
2. 如何创建多个线程？创建线程的两种方式是什么？
3. 线程为什么要休眠？
4. Thread 类都有哪些方法？

1.1　线程的基本概念

线程：一个线程是一个程序内部的顺序控制流。
线程和进程

- 每个进程都有独立的代码和数据空间（进程上下文），进程切换开销大。
- 线程是轻量的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换的开销小。

多进程：在操作系统中，能同时运行多个任务（程序）。
多进程：在同一应用程序中，有多个顺序流同时执行。

线程的概念模型

1. 虚拟的 CPU，封装在 java.lang.Thread 类中。
2. CPU 所执行的代码，传递给 Thread 类。
3. CPU 所处理的数据，传递给 Thread 类。

线程体

1. Java 的线程是通过 java.lang.Thread 类实现的。
2. 每个线程都是通过某个特定 Thread 对象的方法 run() 来完成其操作，方法 run() 成为线程体。

构造线程的两种方法

1. 定义一个线程类，它继承 Thread 并重写其中的方法 run()；

2. 通过实现接口 Runnable 的类作为线程的目标对象,在初始化一个 Thread 类或者 Thread 子类的线程对象时，把目标对象传递这个线程实例，目标对象提供线程体 run().

   public Thread(ThreadGroup group, Runnable target, String name);

题目

1. Java 中的第一个线程都属于某个线程组；

2. 线程只能在其创建时设置所属的线程组；

3. 线程创建之后，不可以从一个线程组转移到另一个线程组；

4. 新建的线程默认情况下属于其父线程所属的线程组；

5. 线程组成：程序计数器、堆栈、栈指针，进程地址空间中的代码不属于线程。

6. Java 的线程模型：代码可以与其他线程共享，数据可以被多个线程共享，线程模型在 java.lang.Thread 类中被定义。

1.2 通过 Thread 类创建线程

1.2.1 Thread 类

1. Thread 直接继承了 Object 类，并实现了 Runnable 接口，位于 lang.lang。
2. 封装了线程对象需要的属性和方法。

1.2.2 创建线程

1. 从 Thread 类派生一个子类，并创建子类的对象；
2. 子类应该重写 Thread 类的 run() 方法，写入需要在新线程中执行的语句段；
3. 调用 start 方法来启动新线程，自动进入 run 方法。

在 main 中调用新的线程，注：

- main 方法调用 thread.start() 方法启动新线程后并不等待其 run() 方法返回就继续运行，线程的 run 方法在一边独自运行，不影响原来的 main 方法运行。

问题

1. 一个线程是一个 Thread 类的实例，线程从传递给纯种的 Runnable 实例 run() 方法开始执行；
2. 线程操作的数据来自 Runnable 实例，新建的线程调用 start() 方法不能立即进入运行状态，进入就绪状态，必须还要保证当前线程获取到资源以后调用 run() 才能开始执行；
3. 在线程 A 中执行线程 B 的 join() 方法，则线程 A 等待直到 B 执行完成；
4. 线程 A 通过调用 interrupt() 方法来中断其阻塞状态；
5. 若线程 A 调用方法 isAlive() 返回值为 true，则说明 A 正在执行中；
6. currentThread() 获取当前线程对象，而不是引用；
7. 程序的执行完毕与超级线程 (daemon threads) 无关。

1.3 线程的休眠

1.3.1 延长主线程

使用 Thread.sleep(1)，休眠 1ms; 在主线程休眠时，会执行子线程。

对于线程，休眠时会把执行权交给其他线程，如果在 main 线程中调用多个线程，会导致当前线程按照 start 顺序执行。

休眠原因：让其他线程得到执行的机会。

1.3.2 问题

1. Thread.sleep(cnt); 会导致抛出异常；

1.4 Thread 类详解

常用方法见表 1.1

1.5 通过 Runnable 接口创建线程

1.5.1 Runnable 使用及特点

Runnable 接口只有一个 run() 方法，实际上 Thread 也是实现了 Runnable 接口；

表 1.1: Thread 类常用 API 方法

名称	说明
public Thread()	构造一个新的线程对象，默认名为 Thread-n，n 是从 0 开始递增的整数
public Thread(Runnable target)	构造一个新的线程对象，以一个实现 Runnable 接口的类的对象为参数。默认名为 Thread-n，n 是从 0 开始递增的整数。
public Thread(String name)	构造一个新的线程对象，并同时指定线程名
public static Thread currentThread()	返回当前正在运行的线程对象
public static void yield()	使当前线程对象暂停，允许别的线程开始运行
public static void sleep(long millis)	使当前线程暂停运行指定毫秒数，但线程并不失去已获得的锁
public void start()	启动线程，JVM 将调用此线程的 run() 方法，结果是将同时运行两个线程，当前线程和执行 run 方法的线程
public void run()	Thread 的子类应该重写此方法，内容应为该线程应执行的任务
public final void stop()	停止线程运行，释放该线程占用的对象锁
public void interrupt()	中断此线程
public fianl void join()	如果此前启动了线程 A，调用 join 方法将等待线程 A 死亡才能继续执行当前线程
public final void join(long millis)	如果此前启动了线程 A，调用 join 方法将等待指定毫秒数或线程 A 死亡才能继续执行当前线程
public final void setPriority(int new Priority)	设置线程优先级
public final void setDaemon(Boolean on)	设置是否为后台进程，如果当前运行线程均为后台线程，则 JVM 停止运行，此方法必须在 start() 方法调用前使用
public final void checkAccess()	判断当前线程是否有权力修改调用此方法的线程
public void setName(String name)	更改本线程的名称为指定参数
public final boolean isAlive()	测试线程是否处于活动状态，如果线程被启动并且没有死亡则返回 true

- 便于多个线程共享资源

- Java 不支持多继承，如果已经继承了某个基类，便需要实现 Runnable 接口来生成多线程

- 以实现 Runnable 的对象为参数建立新的线程
- start 启动线程

两种方法比较：

- 使用 Runnable 接口

  可以将 CPU、代码和数据分开，形成清晰的模型，还可以从其他类继承

- 直接继承 Thread 类

  编写简单，直接继承，重写 run 方法，不能再从其他类继承

1.5.2 问题

1. java 的线程体由 Thread 类的 run() 方法定义，线程创建时已经确定了提供线程体的对象，java 中每一个线程都有自己的名字

1.6 线程内部的数据共享

用同一个实现了 Runnable 接口的对象作为参数创建多个线程；这些多个线程共享同一个对象中的相同的数据。

使用一个 Runnable 类型对象创建的多个新线程，这多个新线程就共享了这个对象的私有成员。

```
1  //MyThread.java
2  public class MyThread implements Runnable{
3
4      private int sleeptime;
5
6      public MyThread() {
7          sleeptime = (int) (Math.random() * 5000);
8      }
9
10     @Override
11     public void run() {
12         try {
13             System.out.println(Thread.currentThread().getName() + "
                   sleep time: " + sleeptime);
```

```
14              Thread.sleep(sleeptime);
15          } catch (Exception ex) {
16          } finally {
17              System.out.println(Thread.currentThread().getName() + "
                    finished");
18          }
19      }
20  }
21
22  //MyThreadTest.java
23  public class MyThreadTest {
24      public static void main(String[] args) {
25          MyThread mythread = new MyThread();
26          System.out.println("main thread start");
27          new Thread(mythread, "Thread 1").start();
28          new Thread(mythread, "Thread 2").start();
29          new Thread(mythread, "Thread 3").start();
30          System.out.println("main thread end");
31      }
32  }
33
34  //output
35  main thread start
36  main thread end
37  Thread 1 sleep time: 3750
38  Thread 2 sleep time: 3750
39  Thread 3 sleep time: 3750
40  Thread 2 finished
41  Thread 3 finished
42  Thread 1 finished
```

第二章　线程中

课前思考

1. 多线程是如何实现同步的？
2. 多线程中如何避免死锁问题？
3. 线程的生命周期是怎么样的？
4. 多个线程之间优先级如何控制？

学习目标

1. java 中多线程的同步控制方法
2. 掌握线程的生命周期
3. 理解多线程同步的锁机制和线程优先级

2.1　线程同步的思路

2.1.1　多线程的同步控制

- 有时线程之间彼此不独立，需要同步

  线程间的互斥：

  同时运行的几个线程需要共享一些数据

  共享的数据，在某个时刻只允许一个线程对其进行操作

  “生产者/消费者”问题

  假设一个线程负责往数据区写数据，另一个线程从同一数据区读数据，两个线程并行执行

  如果数据区已满，生产者要等消费者取走一些数据后才能再写

  当数据去为空，消费者要等生产者写入一些数据后再取

```
1  //Ticket.java
2  public class Ticket {
3      int number = 0;
4      int size;
5      boolean available = false;
6      public Ticket(int size) {
7          this.size = size;
8      }
9  }
10 //Producer.java
11 public class Producer extends Thread {
12     Ticket t = null;
13     public Producer(Ticket t) {
14         this.t = t;
15     }
16     public void run() {
17         while (t.number < t.size) {
18             System.out.println("Producer puts ticket " + (++t.
                   number));
19             t.available = true;
20         }
21     }
22 }
23 //Consumer.java
24 public class Consumer extends Thread {
25     Ticket t = null;
26     int i = 0;
27     public Consumer(Ticket t) {
28         this.t = t;
29     }
30     public void run() {
31         while (i < t.size) {
32             if (t.available == true && i <= t.number) {
33                 System.out.println("Consumer buys ticket " + (++i))
                       ;
```

```
34              }
35              if (i == t.number) {
36                  t.available = false;
37              }
38          }
39      }
40  }
41  //Machine.java
42  public class Machine {
43      public static void main(String[] args) {
44          Ticket t = new Ticket(10);
45          new Consumer(t).start();
46          new Producer(t).start();
47      }
48  }
49  //output
50  Producer puts ticket 1
51  Producer puts ticket 2
52  Consumer buys ticket 1
53  Producer puts ticket 3
54  Consumer buys ticket 2
55  Producer puts ticket 4
56  Consumer buys ticket 3
57  Producer puts ticket 5
58  Consumer buys ticket 4
59  Producer puts ticket 6
60  Producer puts ticket 7
61  Producer puts ticket 8
62  Consumer buys ticket 5
63  Producer puts ticket 9
64  Producer puts ticket 10
65  Consumer buys ticket 6
66  Consumer buys ticket 7
67  Consumer buys ticket 8
68  Consumer buys ticket 9
69  Consumer buys ticket 10
```

如果将 line 35-line37 改为下面代码，因为当消费者休眠时，CPU 执行生产者 t.available=true，但当消费者休眠完毕，将会设置 t.available=true，因此会导致程序一直运行。

```
1 if (i == t.number) {
2     try {
3         Thread.sleep(1);
4     } catch (InterruptedException e) {
5     }
6     t.available = false;
7 }
```

2.2 线程同步的实现方式-Synchronization

2.2.1 线程同步

- 互斥：许多线程在统一共享数据上操作而不互相干扰，同一时刻只能有一个线程访问该共享数据。因此有些方法或程序段在同一时刻只能被一个线程执行，称之为监视区。
- 协作：多个线程可以有条件地同时操作共享数据。执行监视区代码的线程在条件满足的情况下可以允许其他线程进入监视区。

synchronized

synchronized：线程同步关键字，实现互斥。

- 用于指定需要同步的代码段或方法，也就是监视区。
- 可实现与一个锁的交互
- 功能：首先判断对象锁是否存在，如果存在就获得锁，然后就可以执行紧随其后的代码段；如果对象的锁不存在（已被其他线程拿走），就进入等待状态，直到获得锁。
- 当被 synchronized 限定的代码段执行完，就释放锁。

Java 使用监视器机制

- 每个对象只有一个锁，利用多线程对锁的争夺实现线程间的互斥
- 当线程 A 获得一个对象的锁后，线程 B 必须等待线程 A 完成规定的操作，并释放锁后，才能获得该对象的锁，并执行线程 B 中的操作。

使用锁将生产和消费变为原子操作，解决上节的问题：

```
1  //生产者
2  synchronized (t) {
3      System.out.println("Producer puts ticket " + (++t.number));
4      t.available = true;
5  }
6  //消费者
7  synchronized (t) {
8      if (t.available == true && i <= t.number) {
9          System.out.println("Consumer buys ticket " + (++i));
10     }
11     if (i == t.number) {
12         try {
13             Thread.sleep(1);
14         } catch (InterruptedException e) {
15             e.printStackTrace();
16         }
17         t.available = false;
18     }
19 }
```

当线程执行到 synchronized 时，检查传入的实参对象，并申请得到该对象的锁，如果得不到，那么线程就被放到一个与该对象锁相对应的等待线程池中，直到该对象的锁被返回，池中等待线程才能重新去获得锁，然后执行。

对指定代码段同步控制，还可以定义整个方法，要在方法定义前加上 synchronized 即可。

同步与锁的要点

- 只能同步方法、代码块，不能同步变量
- 每个对象只有一个锁；
- 类可以同时拥有同步和非同步方法，非同步方法可以被多个线程自由访问而不受锁的限制
- 如果两个线程使用相同的实例来调用 synchronized 方法，那么一次只能有一个线程执行方法，另一个需要等待锁。
- 线程休眠时，所持的任何锁都不会释放。

- 线程可以获得多个锁，如在一个对象的同步方法里面调用另一个对象的同步方法，则获取了两个对象的同步锁。

- 同步损害并发性，应缩小同步范围。

- 使用同步代码块时，应该指定在哪个对象上同步。

使用 synchronized 修饰方法：

```
1  //Ticket.java
2  public synchronized void put() {
3      System.out.println("Producer puts ticket " + (++number));
4      available = true;
5  }
6  public synchronized void sell() {
7      if (available == true && i <= number) {
8          System.out.println("Consumer buys ticket " + (++i));
9      }
10     if (i == number) {
11         available = false;
12     }
13 }
14 //生产者
15 public void run() {
16     while (t.number < t.size) {
17         t.put();
18     }
19 }
20 //消费者
21 public void run() {
22     while (t.i < t.size) {
23         t.sell();
24     }
25 }
```

2.3 线程的等待和唤醒

2.3.1 线程的等待-wait() 方法

- 为了更有效地协调不同线程的工作，需要在线程间建立沟通渠道，通过线程间的“对话”来解决线程间的同步问题
- java.lang.Object 的方法

  wait()：如果当前状态不适合本线程执行，正在执行同步代码（synchronized）的某个线程 A 调用该方法（在对象 x 上），该线程暂停执行而进入对象 x 的等待池，并释放已获得的对象 x 的锁。线程 A 要一直等到其他线程在对象 x 上调用 notify 或 notifyAll 方法，才能够在重新获得对象 x 的锁后继续执行（从 wait 语句后继续执行）

2.3.2 线程的唤醒-notify() 和 notifyAll() 方法

- notify() 随机唤醒一个等待的线程，本线程继续执行

  线程被唤醒之后，还要等发出唤醒消息者释放监视者，这期间关键数据仍可能被改变

  被唤醒的线程开始执行时，一定要判断当前状态是否适合自己运行
- notifyAll() 唤醒所有等待的线程，本线程继续执行

```
1  //Ticket.java
2  public synchronized void put() {
3      if (available) {
4          try {
5              wait();
6          } catch (InterruptedException e) {}
7      }
8      System.out.println("Producer puts ticket " + (++number));
9      available = true;
10     notify();
11 }
12 public synchronized void sell() {
13     if (!available) {
14         try {
15             wait();
```

```
16          } catch (InterruptedException e) {}
17      }
18      System.out.println("Consumer buys ticket " + (number));
19      available = false;
20      notify();
21      if (number == size) {
22          number = size + 1;
23      }
24  }
25  //生产者
26  public void run() {
27      while (t.number < t.size) {
28          t.put();
29      }
30  }
31  //消费者
32  public void run() {
33      while (t.number < t.size) {
34          t.sell();
35      }
36  }
```

2.4 后台线程

后台线程

- 也叫守护线程，通常是为了辅助其他线程而运行的线程。
- 不妨碍程序终止
- 一个进程中只要还有一个前程线程在运行，这个线程就不会结束，如果一个进程中所有前台线程结束，那么后台线程就会结束。
- 垃圾回收是一个后台线程
- 启动 start() 之前，使用 setDaemon(true)，使它变为后台线程

```java
public class DaemonTest {
    public static void main(String[] args) {
        Thread t = new DaemonThread();
        t.setDaemon(true);
        t.start();
    }
}
class DaemonThread extends Thread {
    @Override
    public void run() {
        while (true){
            System.out.println("I am running.");
        }
    }
}
```

打印结果：运行很快就结束了，但有语句输出；但不设置后台会一直执行。

2.5 线程的生命周期与死锁

2.5.1 线程的生命周期

- 线程从产生到消亡的过程；
- 一个线程在任何时刻都处于某种线程状态（thread state）。

1. 诞生状态：线程刚被创建
2. 就绪状态：线程的 start 方法已被执行，线程已准备好运行
3. 运行状态：处理机分配给线程，线程正在运行
4. 阻塞状态：

   在线程发出输入输出请求且必须等待其返回

   遇到 synchronized 标记的方法而未获得锁

   为等候一个条件变量，线程调用 wait() 方法
5. 休眠状态：执行 sleep 方法进入休眠
6. 死亡状态：线程完成或退出

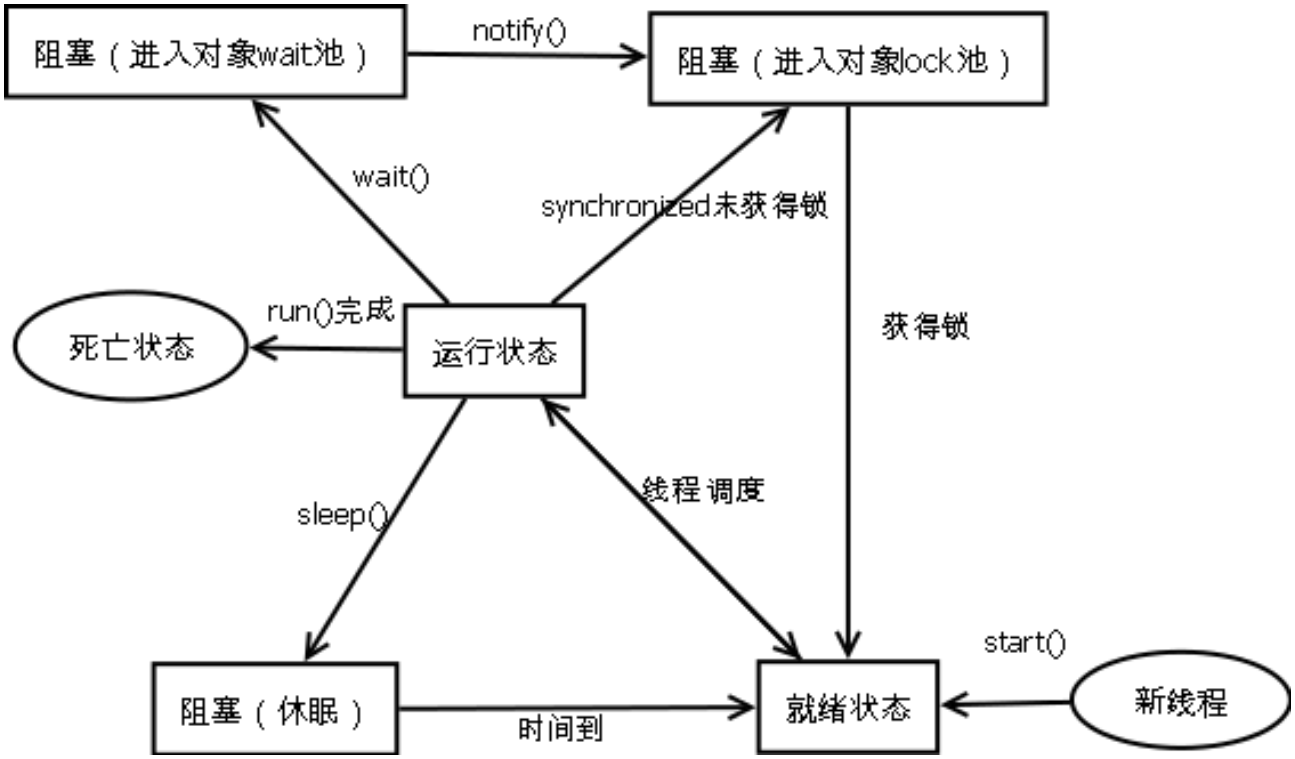

图 2.1: 线程状态生命周期图

2.5.2 死锁

线程在运行过程中，其中某个步骤往往需要满足一些条件才能继续进行下去，如果这个条件不能满足，线程将在此步骤出现阻塞。

如线程 A 等 B，B 等 C，C 又等... 最后回到 A，任何线程不能执行，造成死锁（deadlock）。

2.5.3 控制线程的生命

结束线程的生命：

- 可控制 run 方法中循环条件方式来结束一个线程
- 使用 stop 结束线程生命

  但如果一个线程正在操作共享数据段，操作过程没有完成就用 stop 结束的话，将会导致数据不完整。

2.6 线程的调度

2.6.1 线程的优先级

线程调度

1. 在但 CPU 的系统中，多个线程需要共享 CPU，在任何时间点上实际只能有一个线程在运行；
2. 控制多个线程在同一个 CPU 上以某种顺序运行称为线程调度；
3. Java 虚拟机支持一种非常简单的、确定的调度算法——固定优先级算法，基于线程优先级对其进行调度；
4. 每个 Java 线程都有一个优先级，范围 1-10，默认为 5；
5. 在线程 A 中创建 B，则 B 初始优先级和 A 相同；
6. 如果 A 为后台线程，B 也为后台线程；
7. 可在线程创建后任何时刻，通过 setPriority(int priority) 改变优先级;

基于优先级线程调度

1. 高优先级线程比低优先级先执行；
2. 对相同优先级，Java 处理是随机的；
3. 底层操作系统支持优先级可能小于 10，这会造成一些混乱，只能使用优先级作为作为粗略工具，最后可以通过 yield() 完成。

线程优先级队列
如果某线程正在运行，则只有出现以下情况之一，才会暂停执行

- 一个更高优先级线程处于就绪状态；
- 由于输入输出、调用 sleep、wait、yield 使其阻塞；
- 对于支持时间分片，时间分片执行完。

注：

1. JVM 本身不支持某个线程抢夺另一个正在执行的线程具有同优先级的执行权；
2. yield() 会使正运行线程放弃执行，同优先级线程有机会调度，低优先级仍会被忽略。

第三章　线程下

课前思考

- 线程安全描述的是谁的特性？
- 如何实现线程安全？
- 线程的锁如何优化？

3.1　线程安全与线程兼容与对立

3.1.1　线程安全

线程安全：当多个线程访问同一个对象时，如果不用考虑这些线程在运行时环境的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的。
Java 的线程安全：

- 不可变
- 绝对线程安全
- 相对线程安全
- 线程兼容和对立

3.1.2　不可变

- final 修饰：public final a = 100;
- java.lang.String: String s = ”string”;
- 枚举类型：public enum ColorRED,BLUE
- java.lang.Number 的子类，如 Long，Double
- BigInteger,BigDecimal（数值类型的高精度实现）

3.1.3 绝对线程安全

- 满足线程安全定义的；
- Java API 标注自己是线程安全的类绝大部分不是绝对线程安全的（java.util.Vector）

3.1.4 相对线程安全

通常意义上的线程安全，需要保证这个对象单独操作是线程安全的，调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就需要在调用时使用同步手段保证调用的正确性。（如：Vector、HashTable）

```
1  public class VectorSafe {
2  private static Vector<Integer> vector = new Vector<Integer>();
3      public static void main(String[] args) {
4          while (true) {
5              for (int i = 0; i < 10; i++) {
6                  vector.add(i);
7              }
8              Thread remove = new Thread(new Runnable() {
9              @Override
10             public void run() {
11                 for (int i = 0; i < vector.size(); i++) {
12                     vector.remove(i);
13                 }
14             }
15             });
16             Thread print = new Thread(new Runnable() {
17             @Override
18             public void run() {
19                 for (int i = 0; i < vector.size(); i++) {
20                     System.out.println(vector.get(i));
21                 }
22             }
23             });
24             remove.start();
25             print.start();
26             while (Thread.activeCount() > 20);
```

```
27          }
28      }
29  }
```

有时会出错，有时不会出错，出错时为数组下标越界错误。

改进方法 synchronized：

```
1  Thread remove = new Thread(new Runnable() {
2      @Override
3      public void run() {
4          synchronized (vector) {
5              for (int i = 0; i < vector.size(); i++) {
6                  vector.remove(i);
7              }
8          }
9      }
10 });
11 Thread print = new Thread(new Runnable() {
12     @Override
13     public void run() {
14         synchronized (vector) {
15             for (int i = 0; i < vector.size(); i++) {
16                 System.out.println(vector.get(i));
17             }
18         }
19     }
20 });
```

3.1.5 线程兼容和线程对立

线程兼容：对象本身不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全使用。

线程对立：无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。

- Java 中线程对立：Thread 类的 suspend() 和 resume() 方法可能导致死锁。

3.2 线程的安全实现-互斥同步

线程安全的实现方式：互斥同步、非阻塞同步、无同步方案。

3.2.1 互斥同步

同步的互斥实现方式：临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphone）。

第一种方式：Java 中使用 Synchronized：编译后，会在同步块前后形成 monitorenter 和 monitorexit 两个字节码。

1. synchronied 同步块对自己是可重入的，不会将自己锁死；
2. 同步块在已进入的线程执行完之前，会阻塞后面线程的进入。

第二种方式：重入锁 ReentrantLock(java.util.concurrent)。

- 相比 synchronized，重入锁可实现：等待可中断、公平锁、锁可以绑定多个条件。
- Synchronized 表现为原生语法层面的互斥锁，而 ReentrantLock 表现为 API 层面的互斥锁。

使用 ReentrantLock：

```
1  public class BufferInterruptibly {
2      private ReentrantLock lock = new ReentrantLock();
3      public void write() {
4          lock.lock();
5          try {
6              long start = System.currentTimeMillis();
7              System.out.println("开始往这个buff写数据");
8              for (;;) {
9                  if (System.currentTimeMillis() - start > Integer.
                       MAX_VALUE) {
10                     break;
11                 }
12             }
13             System.out.println("写数据完毕");
14         } finally {
15             lock.unlock();
16         }
```

```
17      }
18      public void read() throws InterruptedException {
19          lock.lockInterruptibly();
20          try {
21              System.out.println("从buff读数据");
22          }finally {
23              lock.unlock();
24          }
25      }
26  }
```

对于单核、多核，ReentrantLock 效率更高更稳定。

3.3 线程的安全实现-非阻塞同步

阻塞同步：互斥同步存在的问题是在进行线程阻塞和唤醒所带来的性能问题，这种同步称为阻塞同步（Blocking Synchronization）。
非阻塞同步：不同于悲观并发策略，而是使用基于冲突检测的乐观并发策略，就是先进行操作，如果没有其他线程征用共享数据，则操作成功；否则就是产生了冲突，采取不断重试直到成功为止的策略，这种策略不需要把线程挂起，称为非阻塞同步。
使用非阻塞同步条件：

- 使用硬件处理器指令进行不断重试策略；
  - 测试并设置（Test-and-Set）
  - 获取并增加（Fetch-and-Increment）
  - 交换（Swap）
  - 比较并交换（Compare-and-Swap，简称 CAS）
  - 加载链接，条件存储（Load-Linked,Store-conditional，简称 LL,SC）
    Java 实现类 AtomicInteger，AtomicDouble

```
1   class Counter {
2       private volatile int count = 0;
3       public synchronized void increment() {
4           count++;
5       }
```

```
6       public int getCount() {
7           return count;
8       }
9   }
10  class CounterNon {
11      private AtomicInteger count = new AtomicInteger();
12      public void increment() {
13          count.incrementAndGet();
14      }
15      public int getCount() {
16          return count.get();
17      }
18  }
```

3.4 线程的安全实现-无同步方案

3.4.1 无同步方案-可重入代码

也叫纯代码，相对线程安全来说，可以保证线程安全，可以在代码执行过程中中断它，转而去执行另一段代码，而在控制权返回后，原来的程序不会出现任何错误。

3.4.2 无同步方案-线程本地存储

如果一段代码中所需要的数据必须与其他代码共享，那就看看这些代码共享数据的代码是否能保证在同一个线程中执行，如果能保证，就可以把共享数据的可见范围限定在同一线程之内，这样无需同步也能保证线程之间也不会出现数据争用问题。

```
1   public class SequenceNumber {
2       private static ThreadLocal<Integer> seqNum = new ThreadLocal<
            Integer>(){
3           @Override
4           protected Integer initialValue() {
5               return 0;
6           }
7       };
8       public int getNextNum() {
```

```
9           seqNum.set(seqNum.get() + 1);
10          return seqNum.get();
11      }
12
13      public static void main(String[] args) {
14          SequenceNumber sn = new SequenceNumber();
15          TestClient t1 = new TestClient(sn);
16          TestClient t2 = new TestClient(sn);
17          TestClient t3 = new TestClient(sn);
18          t1.start();
19          t2.start();
20          t3.start();
21      }
22      private static class TestClient extends Thread {
23          private SequenceNumber sn;
24          public TestClient(SequenceNumber sn) {
25              this.sn = sn;
26          }
27          public void run() {
28              for (int i = 0; i < 3; i++) {
29                  System.out.println("thread[" + Thread.currentThread
                        ().getName() + "]sn[" + sn.getNextNum() + "]");
30              }
31          }
32      }
33  }
```

3.5 锁优化

锁优化方式：自旋锁、自适应锁、锁消除、锁粗化、偏向锁。

3.5.1 自旋锁

互斥同步存在的问题：挂起线程和恢复线程都需要转入内核态中完成，这些操作给系统的并发性能带来很大压力。

自旋锁：如果物理机器有一个以上的处理器能让两个或以上线程同时并行执行，那就可以让

后面请求锁的那个线程等待一会，但不放弃处理器的执行时间，看看持有锁的线程是否能很快就会释放锁。为了让线程等待，我们只有让线程执行一个忙循环（自旋），这项技术称为自旋锁，Java 默认自选次数 10 次。

3.5.2 自适应锁

自适应自旋：自适应意味着自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间及锁拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它允许自旋等待相对更长的一段时间。

3.5.3 锁消除

锁消除：JVM 即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。
判断依据：如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁无需进行。

3.5.4 锁粗化

锁粗化：代码中，同步块范围尽量小，只在共享数据的实际作用域中才进行同步，这样是为了使得同步操作的数量尽可能变小。
另一种情况，如果一系列的连续操作都对同一个对象反复加锁，甚至加锁是出现在循环体中，那即使没有现成争用，频繁的进行互斥同步也会导致不必要的性能消耗，此时只需要将同步块范围扩大，即：锁粗化。

3.5.5 偏向锁

偏向锁目的：消除数据无竞争情况下的同步原语，进一步提高程序运行的性能。偏向锁就是在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做。

3.6 小结

1. 线程安全、线程兼容与线程对立；
2. 线程安全的实现方式；
3. 锁优化。