目 录

第一章 线程概述 1
1.1 线程相关概念 1
1.1.1 进程 1
1.1.2 线程 1
1.1.3 主线程和子线程 1
1.1.4 串行、并发和并行 1
1.2 线程的创建与启动 2
1.3 线程的常用方法 3
1.3.1 currentThread() 方法 3
1.3.2 setName()/getName() 5
1.3.3 isAlive() 5
1.3.4 sleep() 5
1.3.5 getId() 6
1.3.6 yield() 6
1.3.7 setPriority() 6
1.3.8 interrupt() 6
1.3.9 setDaemon() 6
1.4 线程的生命周期 6
1.5 多线程编程的优势与存在的风险 7
1.5.1 优势 7
1.5.2 风险 7

第二章 线程安全问题 8
2.1 原子性 8
2.2 可见性 8
2.3 有序性 8
2.3.1 重排序 8
2.3.2 指令重排序 9
2.3.3 貌似串行语义 9
2.4 Java 内存模型 10

第三章 线程同步 11
3.1 线程同步机制简介 11
3.2 锁 11
3.2.1 锁的作用 11
3.2.2 锁相关的概念 11
3.3 内部锁：synchronized 关键字 12
3.3.1 synchronized 同步代码块 12
3.3.2 synchronized 修饰实例方法 14
3.3.3 synchronized 修饰类方法 15
3.3.4 线程异常 17
3.3.5 死锁 19
3.4 轻量级同步机制 volative 关键字 20
3.4.1 volative 的作用 20
3.4.2 volative 的非原子特性 20
3.4.3 常用的原子类进行自增自减操作 21
3.5 CAS 21
3.5.1 ABA 问题 23
3.6 原子变量类 23
3.6.1 AtomicLong 23
3.6.2 AtomicIntegerArray 26
3.6.3 AtomicIntegerFieldUpdater 27
3.6.4 AtomicReference 29

第四章 线程间通信 33
4.1 等待/通知机制 33
4.1.1 什么是等待/通知机制 33
4.1.2 等待/通知机制的实现 33

第一章　线程概述

1.1　线程相关概念

1.1.1　进程

进程 (Process)：计算机中的程序关于某数据集合上的一次运行活动，是操作系统进行资源分配与调度的基本单位。

可以简单的理解为正在操作系统中运行的一个程序。

1.1.2　线程

线程 (Thread) 是进程的一个执行单元。

一个线程就是进程中一个单一顺序的控制流，一个线程就是进程的一个执行分支。

进程是线程的容器，一个进程至少有一个线程。一个进程中也可以有多个线程。

操作系统中是以进程为单位分配资源，如虚拟存储空间，文件描述符等。每个线程都有各自的线程栈，自己的寄存器环境，自己的线程本地存储。

1.1.3　主线程和子线程

JVM 启动时会创建一个主线程，该主线程负责执行 main() 方法，主线程就是运行 main 方法的线程。

Java 中的线程不是孤立的，线程之间存在一些联系。如果在 A 线程中创建了 B 线程，称 B 线程为 A 线程的子线程，相应的 A 线程就是 B 线程的父线程。

1.1.4　串行、并发和并行

1.1.4.1　串行

串行 (Sequential)：对于多个任务，在执行完一个任务后再去做下一个任务。缺点：耗时。

1.1.4.2　并发

并发 (Concurrent)：对于多个任务，先做其中一个任务，在该任务进入等待状态时（例如等待用户的输入）就开始做下一个任务。

1.1.4.3　并行

并行 (Parallel)：多个任务同时开始，总耗时取决于所需时间最长的那个任务。

1.1.4.4　对比

并发可以提高事物的处理效率，即一段时间内可以处理或者完成更多的事情。

并行是一种更为严格的，理想的并发。

从硬件角度来说，如果是单核 CPU，一个处理器只能执行一个线程的情况下，处理器可以使用时间片轮转技术，可以让 CPU 在各个线程间进行切换。对于用户来说，感觉是多个线程在同时执行。

如果是多核 CPU，可以为不同的线程分配不同的 CPU 内核。

1.2 线程的创建与启动

在 Java 中，创建一个线程，就是创建一个 `Thread` 类（子类）的对象（实例）。

`Thread` 类有两个常用的构造方法： `Thread()` 与 `Thread(Runnable)` ，对应的创建线程的两种方式：

- 定义 `Thread` 类的子类
- 定义一个 `Runnable` 接口的实现类

这两种创建线程得到方式没有本质的区别。

创建线程方式一

```java
//1.定义类继承Thread
public class MyThread extends Thread{
    //2.重写父类中的run方法

    //run()方法体中的代码就是子线程要执行的任务
    @Override
    public void run() {
//        super.run();
        System.out.println("这是子线程打印的内容！");
    }
}

public class Test {
    public static void main(String[] args)  {
        System.out.println("Java虚拟机启动main线程，该线程执行main方法");
        //3.创建子线程对象
        MyThread myThread = new MyThread();

        //4.启动线程
        //调用线程的start()方法来启动线程
        //启动线程的实质就是请求JVM运行相应的线程
        //这个线程具体在什么时候运行由线程调度器(Scheduler)决定
        //注意：start()方法的调用结束并不意味着子线程开始运行，只是通知JVM该线程准备好了
        //      新开启的线程会执行run()方法
        //      如果开启了多个线程，start()调用得到顺序不一定就是线程启动的顺序
        //      多线程运行结果与代码顺序或调用顺序无关
        myThread.start();
        try{
            Thread.sleep(1000);//线程休眠，单位是毫秒
        }catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

创建线程方式二

```java
/**
```

```
 * 当线程类已经有父类，就不能用继承Thread类的形式来创建线程
 * 可以使用实现Runnable接口的形式来实现
 * 1.定义类实现Runnable接口
 */
public class MyRunnable implements Runnable{
    //2.重写Runnable接口中的抽象方法run
    //run()就是子线程要执行的代码
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("sub thread-->" + i);
        }
    }
}

public class test {
    public static void main(String[] args) {
        //3.创建Runnable接口的实现类对象
        MyRunnable myRunnable = new MyRunnable();
        //4.创建线程对象
        //Thread的一个构造方法参数类型为实现Runnable接口的对象
        //使用该构造方法创建一个线程对象
        Thread thread = new Thread(myRunnable);
        //5.开启线程
        thread.start();
        //main线程
        for (int i = 0; i < 100; i++) {
            System.out.println("main thread-->" + i);
        }

        //有时候调用Thread(Runnable)构造方法时，实参也会传递匿名内部类对象
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    System.out.println("匿名内部内子线程 thread-->" + i);
                }
            }
        });
        thread1.start();
    }
}
```

1.3 线程的常用方法

1.3.1 currentThread() 方法

`Thread.currentThread()`（类方法）：可以获得当前线程

Java 中的任何一段代码都是执行在某个线程当中的，执行当前代码的线程就是当前线程。

同一段代码可能被不同的线程执行，因此当前线程是相对的。

该方法的返回值是在代码实际运行时的线程对象。

```
public class SubThread extends Thread{
    public SubThread(){
        System.out.println("构造方法打印当前线程名称: " + Thread.currentThread().
            getName());
    }
    @Override
    public void run() {
        System.out.println("run方法打印当前线程名称: " + Thread.currentThread().
            getName());
    }
}

public class Test01CurrentThread {
    public static void main(String[] args) {
        System.out.println("main方法打印当前线程名称: " + Thread.currentThread().
            getName());
        //创建子线程,调用SubThread构造方法
        //在main线程中调用构造方法，所以构造方法中的当前线程就是main线程
        SubThread subThread = new SubThread();
        subThread.start();//启动子线程，子线程会调用run方法，所以run()当前线程就是
            Thread-0子线程。
        subThread.run();//在main方法中直接调用run()方法，没有开启新的线程，所以run
            ()方法所在的当前线程就是main方法
    }
}
```

当前线程就是调用该方法的线程。

复杂一点的代码

```
public class SubThread1 extends Thread {
    public SubThread1() {
        System.out.println("构造方法中，Thread.currentThread().getName():" +
            Thread.currentThread().getName());
        System.out.println("构造方法中: this.getName(): " + this.getName());
    }

    @Override
    public void run() {
        System.out.println("run方法中，Thread.currentThread().getName():" + Thread
            .currentThread().getName());
        System.out.println("run方法中: this.getName(): " + this.getName());

    }
}

public class Test02CurrentThread {
    public static void main(String[] args) throws InterruptedException {

```

```
18          //创建子线程对象
19          SubThread1 subThread1 = new SubThread1();
20          subThread1.setName("st1");//设置线程名称
21          subThread1.start();
22          Thread.sleep(500);//main线程睡眠500毫秒
23          Thread thread = new Thread(subThread1);
24          thread.setName("st2");
25          thread.start();
26      }
27  }
```

运行结果

```
1  构造方法中，Thread.currentThread().getName():main //new该对象是由main调用的，来到
       SubThread1的构造方法，Thread.currentThread()获得当前线程
2  构造方法中: this.getName(): Thread-0 //this.getName()中this代表当前对象，在构造方
       法中，当前对象是new出来的对象，new出来的SubThread1对象的线程就是thread-0
3  run方法中，Thread.currentThread().getName():st1 //启动线程后，就会执行run方法，当
       前线程和this是同一线程，都是st1
4  run方法中: this.getName(): st1
5
6  //new Thread(subThread1)创建新的线程，Thread是父类，不会执行子类的构造方法
7  //thread.start()启动线程执行run方法，现在是st2线程在执行run方法，所以currentThread
       ()得到的是st2线程
8  run方法中，Thread.currentThread().getName():st2
9  //创建Thread实例时，给它传递了一个Runnable接口实现的对象subThread1,run方法中的this
       指的就是subThread1
10 run方法中: this.getName(): st1
```

1.3.2 `setName()/getName()`

这两个方法是实例方法.

```
1  Thread thread = new Thread();
2
3  thread.setName("t1");//设置线程名称
4  thread.getName();
```

通过设置线程名称有助于程序调试，提高程序的可读性，建议为每个线程都设置一个能够体现线程功能的名称。

1.3.3 `isAlive()`

`thread.isAlive()` 判断当前进程是否处于活动状态。
活动状态：线程已启动并且尚未终止。

1.3.4 `sleep()`

`Thread.sleep(millis)`：让当前线程休眠指定的毫秒数
当前线程指的是 `Thread.currentThread()` 返回的线程

1.3.5 `getId()`

`thread.getId()`：获得并返回线程的唯一标识。

某个编号的线程运行结束后，该编号可能被后续创建的线程使用。

重启 JVM 后，同一个线程的编号可能不一样。

1.3.6 `yield()`

`Thread.yield()`：放弃当前的 CPU 资源。

1.3.7 `setPriority()`

`thread.setPriority(num)`：设置线程的优先级

Java 线程的优先级的取值范围是 1-10, 超出该范围就会抛出异常 `IllegalArgumentException`。

在操作系统中，优先级较高的线程获得的 CPU 资源越多。

线程优先级本质上是只是给线程调度器一个提示信息，以便于调度器决定调度哪些线程，并不能保证优先级高的线程线运行。

Java 优先级设之不当或者滥用可能导致某些线程永远无法得到运行，即产生了线程饥饿。

线程优先级并不是设置的越高越好，一般情况下使用普通的优先级即可，即在开发时不必设置线程的优先级，采用默认值即可。

线程的优先级具有继承性，在 A 线程中创建了 B 线程，则 B 线程的优先级与 A 线程一致。

1.3.8 `interrupt()`

`thread.interrupt()`：中断线程

调用该方法仅仅是在当前线程打一个停止标志，并不是真正的停止线程。

`thread.isInterrupted()` 返回线程的中断标志

可以使用中断标志来结束线程，即

```
\\在run方法中
if(this.isInterrupted()){
    return;
}
```

1.3.9 `setDaemon()`

`thread.setDaemon(true)`：将某个线程设置为守护线程, 设置守护线程的代码应在线程的启动前。

Java 中的线程分为用户线程和守护线程，守护线程是为其他线程提供服务的线程，如垃圾回收器就是一个典型的守护线程。

守护线程不能单独运行，当 JVM 中没有其他用户线程，只有守护线程时，守护线程自动销毁，JVM 就会退出。

1.4 线程的生命周期

可以通过 `getState()` 方法来获得，线程状态是 `Thread.State` 枚举类型定义的。有以下几种

- `NEW`：新建状态，创建了线程对象，在调用 `start` 启动之前的状态。

- `RUNNABLE`：可运行状态，是一个复合状态，包括 `READY` 和 `RUNNING` 两个状态。`READY` 状态表示该线程可以被线程调度器进行调度使它处于 `RUNNING` 状态。`RUNNING` 状态表示线程正在执行，`Thread.yield()` 方法可以把线程由 `RUNNING` 状态转换为 `READY` 状态。
- `BLOCKED`：阻塞状态，线程发起阻塞的 I/O 操作，或者申请由其他线程占用的独占资源，线程会转为 `BLOCKED`，处于阻塞状态的线程不会占用 CPU 资源，当阻塞 I/O 操作执行完或者线程获得了其申请的资源，线程可以转化为 `RUNNABLE`。
- `WAITING`:等待状态,线程执行了 `object.wait()` 方法或 `thread.join()` 方法,会把线程转换为 `WAITING` 等待状态，执行 `object.notify()` 方法，或者加入的线程执行完毕，当前的线程会转换为 `RUNNABLE`。
- `TIME_WAITING`:与 `WAITING` 类似,都是等待状态,也可调用 `Thread.sleep()` 方法将其转换为 `TIME_WAITING` 状态，区别在于处于该状态的线程不会无限的等待。如果线程没有在指定的时间范围内完成期望操作，该线程就会自动转换为 `RUNNABLE`。
- `TERMINATED`：终止状态，线程结束处于终止状态。

1.5 多线程编程的优势与存在的风险

1.5.1 优势

- 提高系统的吞吐率（Throughout），多线程编程编程可以使一个进程有多个并发（concurrent, 即同时进行）的操作。
- 提高响应性（Responsiveness）。例如 Web 服务器会采用一些专门的线程负责用户的请求处理，缩短了用户的等待时间。
- 充分利用多核（Multicore）处理器资源。通过多线程充分的利用 CPU 资源。

1.5.2 风险

- 线程安全 (Thread safe) 问题，多个线程共享数据时，如果没有采取正确的并发访问控制措施，就会产生数据一致性问题。如读取脏数据（过期的数据），如丢失数据更新。
- 线程活性 (Thread liveness) 问题。由于程序自身的缺陷或资源的稀缺性导致线程一直处于非 `RUNNABLE` 状态。常见的活性故障有：
  - 死锁 (Deadlock)：类似于鹬蚌相争。
  - 锁死 (Lockout)：类似于睡美人故事中的王子挂了
  - 活锁 (Livelock)：类似于小猫咬自己的尾巴。
  - 饥饿 (Starvation)：类似于健壮的雏鸟总是从母鸟嘴中抢到食物，弱小的雏鸟总是挨饿。
- 上下文切换 (Context Switch)：处理器从执行一个线程切换到执行另外一个线程需要性能消耗
- 可靠性：可能会有一个线程导致 JVM 意外终止，其他线程也无法执行。

第二章　线程安全问题

非线程安全主要指多个线程对同一个对象的实例变量进行操作时，会出现值被更改，值不同步的情况。
线程安全问题表现为三个方面：原子性，可见性和有序性。

2.1　原子性

原子 (Atomic) 就是不可分割的意思，原子操作得到不可分割有两层含义

- 访问（读、写）某个共享变量的操作从其他线程来看，该操作要么已经执行完毕，要么尚未发生，即其他线程看不到当前操作的中间结果。
- 访问同一组共享变量的原子操作是不能够交错的。

Java 有两种方式实现原子性：一是使用锁，另一种是使用处理器的 CAS(Compare and Swap) 指令。
锁具有排他性，可以保证共享变量在某一时刻只能被一个线程访问。
CAS 指令直接在硬件（处理器和内存）上实现原子操作，看做是硬件锁。
Java 中提供了一个线程安全`AtomicInteger`类，保证了操作的原子性。

2.2　可见性

在多线程环境中，一个线程对某个共享变量进行更新后，后续的其他线程可能无法立即读取到这个更新的结果。

如果一个线程对共享变量更新后，后续访问该变量的其他线程可以读到更新的结果，称这个线程对共享变量的更新对其他线程可见。否则称这个线程对共享变量的更新对其他线程不可见。

多线程程序因为可见性问题可能会导致其他线程读取到旧数据。

2.3　有序性

有序性（Ordering）是指在什么情况下一个处理器上运行的一个线程所执行的内存访问操作在另一个处理器运行的其他线程看来是乱序（Out of Order）的。

乱序是指内存访问操作的顺序看起来发生了变化。

2.3.1　重排序

在多核处理器环境下，编写的顺序结构这种操作执行得到顺序可能是没有保障的：

- 编译器可能会改变两个操作的先后顺序。
- 处理器也可能不会按照目标代码的顺序执行。

一个处理器上执行的多个操作，在其他处理器看来它的顺序与目标代码指定的顺序可能不一样，这种现象称为重排序。

重排序是对内存访问有关操作的一种优化，可以在不影响单线程程序正确的情况下提升程序的性能。但是可能会对多线程程序的正确性产生影响，可能导致线程安全问题。

一组概念：

- 程序顺序：处理器上运行的目标代码所指定的内存访问顺序。

- 执行顺序：内存访问操作在处理器上实际执行的顺序。
- 感知顺序：给定处理器所感知到的该处理器及其他处理器的内存访问操作顺序。

可以把重排序指令分为两种：

- 指令重排序：由 JIT 编译器处理器引起的，指程序顺序与执行顺序不一样。
- 存储子系统重排序是由高速缓存，写缓冲器引起的。感知顺序与执行顺序不一致。

2.3.2 指令重排序

在源码顺序与程序顺序不一致或程序顺序与执行顺序不一致的情况下，我们就说发生了指令重排序（Instruction Reorder）。

指令重排是一种动作，确实对指令顺序做了调整，重排序了对象指令。javac 编译器一般不会执行指令重排序，而 JIT 编译器可能执行指令重排序。

处理器也可能执行指令重排序，使得执行顺序与程序顺序不一致。

2.3.2.1 存储子系统重排序

存储子系统是指写缓冲器与高速缓存。

写缓冲器（Store buffer,Write buffer）用来提高写高速缓存操作的操作效率的。

即使处理器严格按照程序执行顺序执行两个内存访问操作，在存储子系统的作用下，其他处理器对这两个操作的感知顺序也可能与程序顺序不一致。即这两个操作顺序看起来像是发生了变化，这种现象称为存储子系统重排序。

存储子系统重排序并没有真正的对指令执行顺序进行调整，而是造成一种指令执行顺序被调整的假相。

存储子系统重排序对象是内存操作的结果。

从处理器角度来看，读内存就是从指定的 RAM 地址中加载数据到寄存器，称为 Load 操作。写内存就是把数据存储到指定的地址表示的 RAM 存储单元 VS，称为 Store 操作。内存重排序有以下四种可能：

- LoadLoad 重排序。一个处理器上先后执行两个读操作 L1 和 L2，其他处理器对这两个内存操作的感知顺序可能是 L2，L1（顺序反了）。
- StoreStore 重排序。
- LoadStore 重排序。
- StoreLoad 重排序。

高速缓存 (Cache) 是 CPU 中为了匹配与主内存处理速度不匹配而设计的一个高速缓存。

2.3.3 貌似串行语义

JIT 编译器，处理器，存储子系统是按照一定的规则对指令，内存操作的结果进行重排序。给单线程程序造成一种假象–指令是按照源码的顺序执行的，这种假象称为貌似串行语义。并不能保证多线程环境程序的正确性。

为了保证貌似串行语义，有数据依赖关系的语句不会被重排序。

2.4 Java 内存模型

每个线程都有独立的栈空间，在每个栈空间中每个方法都有独立的一块空间。

每个线程都可以访问堆内存。

计算机的凑不直接从主内存读取数据，CPU 读取数据时，先把主内存的数据读入 cache 中，再把 Cache 的数据读到 Register 寄存器中（寄存器在 CPU 内）。

JVM 中的共享数据可能会被分配到寄存器，每个 CPU 都有自己的寄存器，一个 CPU 不能读取其他 CPU 的寄存器内容。如果两个线程分别运行在不同的处理器上，而共享的数据被分配到寄存器上，就会产生可见性问题。即使 JVM 中的共享数据分配到主内存中，也不能保证数据的可见性。也可能分配到 Cache 中，也可能分配到主内存中。

一个处理器上运行的线程对数据的更新可能只是更新到处理器的写缓冲器上，还未到达 Cache 缓存，更不用说到达主内存，一个处理器不能读取到另一个处理器写缓冲器上的内容，会产生另一个处理器上的线程无法看到该处理器对共享数据的更新。

一个处理器的 Cache 不能读取另一个处理器的 Cache，但是一个处理器可以通过缓存一致性协议来读取其他处理器缓存中的数据，并将读到的数据更新到该处理的 Cache 中，该过程称为缓存同步。缓存同步使得一个处理器上运行的线程可以读取到另外一个处理器上运行的的线程对共享数据的所做的更新，保障了可见性。为了保证可见性，必须使一个处理器对共享数据的更新最终被写入高处理器的 Cache 中，这个过程称为冲刷处理器缓存。

规定：每个线程之间的共享数据都存储在主内存中，每个线程都有一个私有的本地（工作）内存。线程的工作内存是抽象的概念，不是真实存在的，它涵盖写缓冲器、寄存器还有其他硬件的优化。每个线程从主内存中把数据读取到本地工作内存中，在工作内存中保存共享数据的副本。线程在自己的工作内存中处理数据，仅对当前线程课件，对其他线程是不可见的。

第三章　线程同步

3.1　线程同步机制简介

线程同步机制是一套用于协调线程之间的数据访问机制。该机制可以保障线程安全。

Java 平台提供的线程同步机制包括：锁，`volatile` 关键字，`final` 关键字，`static` 关键字以及相关的 API，如 `Object.wait()`，`Object.notify()` 等。

3.2　锁

线程安全问题是多个线程并发访问共享数据。

将多个线程对共享数据的并发访问转换为串行访问，即一个共享数据一次只能被一个线程访问。锁就是利用这种思路来保证线程安全的。

锁 (Lock) 可以理解为对共享数据进行保护的一个许可证，对于同一个许可证保护的共享数据来说，任何线程想要访问这个共享数据，必须先持有该许可证。一个线程只有在持有许可证的情况下才能对这些共享数据进行访问，并且一个许可证一次只能被一个线程持有。许可证线程在结束对共享数据的访问后必须释放其持有的许可证。

一个线程在访问共享数据前必须先获得锁，获得锁的线程称为锁的持有线程，一个锁一次只能被一个线程持有。锁的持有线程在获得锁之后和释放锁之前这段时间所执行的代码称为临界区（Critical Section）。

锁具有排他性：一个锁一次只能被一个线程持有。这种锁称为互斥锁，排它锁。

JVM 把锁分为内部锁和显示锁。内部锁通过 `synchronized` 关键字实现；显示锁通过 `java.concurrent.locks.Lock` 接口的实现类实现的.

3.2.1　锁的作用

锁可以实现对共享数据的安全访问。保障线程的原子性、可见性、与有序性。

锁是通过互斥保障原子性。

可见性的保障是通过写线程冲刷处理器的缓存和读线程刷新处理器缓存两个动作实现的。在 Java 平台中，锁的获得隐含着刷新处理器缓存的动作。锁的释放隐含着冲刷处理器缓存的动作。

锁能够保障有序性。写线程在临界区执行的操作在读线程所执行的临界区看来像是完全按照源码顺序执行的。

使用锁保障线程的安全性，必须满足以下条件：

- 这些线程在访问共享数据时必须使用同一个锁。
- 即使是读取共享数据的线程也需要使用同步锁。

3.2.2　锁相关的概念

- 可重入性：一个线程持有该锁的时候能再次（多次）申请该锁。如果一个线程持有一个锁的时候还能够继续成功申请该锁，称该锁是可重入的，否则就称该锁为不可重入的。
- 锁的争用与调度：Java 平台中内部锁属于非公平锁，显式 Lock 锁既支持公平锁又支持非公平锁。
- 锁的粒度：一个锁可以保护的共享数据的数量大小称为锁的粒度。锁保护的共享数据量大，称该锁的粒度粗，否则就称该锁的粒度细。锁的粒度过粗会导致线程在申请锁的时候进行不必要的等待。锁的粒度过细会增加锁调度的开销。

3.3 内部锁：synchronized 关键字

Java 中的每个对象都有一个与之关联的内部锁 (Intrinsic lock)，这种锁也称为监视器 (Monitor)。这种内部锁是一种排它锁，可以保障原子性、可见性、有序性。

内部锁是通过 synchronized 关键字实现的。 synchronized 关键字可以修饰代码块，修饰方法。

修饰代码块语法：

```
synchronized( 对象锁 ){
    同步代码块，可以在同步代码块中访问共享数据
}
```

修饰实例方法就称为同步实例方法。

修饰类方法就称为同步类方法。

3.3.1 synchronized 同步代码块

```
/**
 * synchronized 同步代码块
 */
public class Test01 {
    public static void main(String[] args) {
        //创建两个线程分别调用mm方法
        //先创建一个Test01对象，通过对象名调用mm()
        Test01 test01 = new Test01();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test01.mm();//使用的锁对象this就是test01对象
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test01.mm();//使用的锁对象this也是test01对象
            }
        }).start();
    }
    /**
     * 打印100行字符串
     */
    public  void mm(){
        synchronized (this) {//经常使用当前对象作为锁对象
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() + "--->" + (i
                    + 1));
            }
        }
    }
}
```

对于不同的锁则不能实现同步

```
/**
 * synchronized 同步代码块
 * 如果线程的锁不同，则不能实现同步。
 * 想要同步必须使用同一个锁对象
 */
public class Test02 {
    public static void main(String[] args) {
        //创建两个线程分别调用mm方法
        //先创建一个Test01对象，通过对象名调用mm()
        Test02 test01 = new Test02();
        Test02 test02 = new Test02();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test01.mm();//使用的锁对象this就是test01对象
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test02.mm();//使用的锁对象this就是test02对象
            }
        }).start();
    }
    /**
     * 打印100行字符串
     */
    public  void mm(){
        synchronized (this) {//经常使用当前对象作为锁对象
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() + "--->" + (i
                    + 1));
            }
        }
    }
}
```

```
/**
 * synchronized 同步代码块
 * 锁对象也可以是常量
 */
public class Test03 {
    public static void main(String[] args) {
        //创建两个线程分别调用 mm 方法
        //先创建一个 Test01 对象，通过对象名调用 mm()
        Test03 test01 = new Test03();
        Test03 test02 = new Test03();
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                test01.mm();//使用的锁对象是常量对象 OBJ
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test02.mm();//使用的锁对象是常量对象 OBJ
            }
        }).start();
    }
    public static final Object OBJ = new Object();//定义一个常量
    /**
     * 打印 100 行字符串
     */
    public  void mm(){
        synchronized (OBJ) {//可以使用常量对象作为锁对象
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() + "--->"
                 + (i + 1));
            }
        }
    }
}
```

不管是实例方法还是类方法，只要是同一个锁对象都能实现线程同步。

3.3.2 synchronized 修饰实例方法

```
/**
 * synchronized 同步实例方法
 * 把整个方法体作为同步代码块
 * 默认的锁对象是 this 对象
 */
public class Test05 {
    public static void main(String[] args) {
        //创建两个线程分别调用 mm 方法
        //先创建一个 Test01 对象，通过对象名调用 mm()
        Test05 test01 = new Test05();
        new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                test01.mm();//使用的锁对象 this 就是 test01 对象
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test01.mm1();//使用的锁对象 this 也是 test01 对象
            }
        }).start();
    }

    /**
     * 打印 100 行字符串
     */
    public void mm() {
        synchronized (this) {//经常使用当前对象作为锁对象
            System.out.println(" 修饰代码块");
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() + "--->"
                + (i + 1));
            }
        }
    }
    //使用 synchronized 修饰实例方法，同步实例方法，默认 this 作为锁对象
    public synchronized void mm1() {
        System.out.println(" 修饰方法");
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + "--->"
            + (i + 1));
        }
    }
}
```

3.3.3 synchronized 修饰类方法

```
/**
 * synchronized 同步类方法
 * 把整个方法体作为同步代码块
 * 默认的锁对象是当前类的运行时类对象，又称之为类锁
 */
public class Test06 {
```

```
public static void main(String[] args) {
    //创建两个线程分别调用 mm 方法
    //先创建一个 Test01 对象，通过对象名调用 mm()
    Test06 test01 = new Test06();
    Test06 test02 = new Test06();
    new Thread(new Runnable() {
        @Override
        public void run() {
            test01.mm();//使用当前类的运行时类作为锁对象
        }
    }).start();
    new Thread(new Runnable() {
        @Override
        public void run() {
            test02.sm1();//使用当前类的运行时类作为锁对象
        }
    }).start();
    new Thread(new Runnable() {
        @Override
        public void run() {
            sm();
        }
    }).start();
}

public static final Object OBJ = new Object();//定义一个常量

/**
 * 打印 100 行字符串
 */
public void mm() {
    synchronized (Test06.class) {//可以使用使用当前类的运行时类作为锁对象
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() +
            "--->" + (i + 1));
        }
    }
}

public static void sm() {
    synchronized (Test06.class) {//可以使用当前类的运行时类作为锁对象。
        System.out.println(" 静态方法");
        for (int i = 0; i < 100; i++) {
```

```
                System.out.println(Thread.currentThread().getName() +
                "--->" + (i + 1));
            }
        }
    }

    public synchronized static void sm1() {
//         synchronized (OBJ) {//可以使用常量对象作为锁对象
        System.out.println(" 修饰静态方法");
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() +
            "--->" + (i + 1));
//            }
        }
    }
}
```

3.3.4 线程异常

同步时，若线程出现异常，则会自动释放锁。

```
/**
 * synchronized 同步类方法
 * 同步时，某个线程出现异常，则该线程会释放锁
 */
public class Test08 {
    public static void main(String[] args) {
        //创建两个线程分别调用 mm 方法
        //先创建一个 Test01 对象，通过对象名调用 mm()
        Test08 test01 = new Test08();
        Test08 test02 = new Test08();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test01.mm();//使用当前类的运行时类作为锁对象
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                test02.sm1();//使用当前类的运行时类作为锁对象
            }
        }).start();
        new Thread(new Runnable() {
```

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

    public static final Object OBJ = new Object();//定义一个常量

    /**
     * 打印 100 行字符串
     */
    public void mm() {
        synchronized (Test08.class) {//可以使用使用当前类的运行时类作为锁对象
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() +
                "--->" + (i + 1));
                if (i == 20) {
                    System.out.println(Integer.parseInt("a"));//产生异常的语句
                }
            }

        }
    }

    public static void sm() {
        synchronized (Test08.class) {//可以使用当前类的运行时类作为锁对象。
            System.out.println(" 静态方法");
            for (int i = 0; i < 100; i++) {
                System.out.println(Thread.currentThread().getName() +
                "--->" + (i + 1));
            }
        }
    }

    public synchronized static void sm1() {
//        synchronized (OBJ) {//可以使用常量对象作为锁对象
        System.out.println(" 修饰静态方法");
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() +
             "--->" + (i + 1));
//            }
        }
```

```
    }
}
```

3.3.5 死锁

```
/**
 * @program: DataStructures
 * @description
 *  死锁演示
 *      多线程中，同步可能需要多个锁，如果获得锁的顺序不一致，可能会导致死锁
 * @author: 戛剑生
 * @creat: 2021-03-05 08:35:48
 **/
public class Test09 {
    public static void main(String[] args) {
        SubThread subThread1 = new SubThread();
        subThread1.setName("a");
        subThread1.start();
        SubThread subThread2 = new SubThread();
        subThread2.setName("b");
        subThread2.start();
    }
    static class SubThread extends Thread{
        private static final Object LOCK1 = new Object();
        private static final Object LOCK2 = new Object();
        @Override
        public void run() {
            if ("a".equals(Thread.currentThread().getName())){
                synchronized (LOCK1){
                    System.out.println
                        ("a 线程获得 LOCK1 锁，还需要获得 LOCK2 锁！");
                    synchronized (LOCK2){
                        System.out.println
                            ("a 线程获得 LOCK1 锁后获得 LOCK2 锁！");
                    }
                }
            }

            if ("b".equals(Thread.currentThread().getName())){
                synchronized (LOCK2){
                    System.out.println
                        ("b 线程获得 LOCK2 锁，还需要获得 LOCK1 锁！");
                    synchronized (LOCK1){
                        System.out.println
```

```
                        ("b 线程获得 LOCK2 锁后获得 LOCK1 锁! ");
                }
            }
        }
    }
}
```

上述代码中，如果线程 a 启动，同时线程 b 启动，线程 a 拿到 LOCK1，线程 b 拿到 LOCK2，线程 a 若要执行完临界区代码，则需拿到 LOCK2，而线程 b 若要执行完临界区代码，则需拿到 LOCK1，由于 LOCK2 被线程 b 占用，线程 a 则需等待，无法继续执行，则无法释放锁 LOCK1，同理线程 b 也无法释放锁 LOCK2，造成死锁。

如何避免死锁？

当需要获得多个锁时，所有线程锁的获得顺序保持一致即可。

3.4 轻量级同步机制 volative 关键字

3.4.1 volative 的作用

该关键字的作用是使变量在多个线程之间可见。

解决变量的可见性： volative 关键字强制线程从公共内存中读取变量的值而不是从工作内存中读取。

3.4.1.1 volative 与 synchronize 关键字比较

1. volative 关键字是线程同步的一个轻量级实现，性能好于 synchronize 。

2. volative 只能修饰变量， synchronize 可以修饰方法、代码块，随着 JDK 新版本的发布，synchronize 的执行效率也有较大的提升。在开发中使用 synchronize 的比率还是很大的。

3. 多线程访问 volative 变量不会发生阻塞，而 synchronize 可能会阻塞。

4. volative 能保证数据的可见性，但不能保证原子性。而 synchronize 可以保证原子性，也可以保证可见性。

5. volative 解决的是变量在多个线程的可见性， synchronize 解决多个线程访问公共资源的同步性。

3.4.2 volative 的非原子特性

volative 增加了实例变量在多个线程的可见性，但是不具备原子性。

```
public class Test03 {
    public static void main(String[] args) {
//        SubThread subThread = new SubThread();
        for (int i = 0; i < 10; i++) {
            new SubThread().start();
        }
    }
    static class SubThread extends Thread{
        //volatile 关键字仅仅是表示所有线程从主内存中读取 count 变量的值
        //某个线程未运行结束，其他线程可能会抢到 CPU 执行权，无法保证原子性
```

```
        public  volatile static int count;
        public static void addCount(){
            for (int i = 0; i < 1000; i++) {
                count++;
            }
            System.out.println(Thread.currentThread().getName()
            + "count" + count);
        }
        //必须使用 synchronized 保证原子性
        public synchronized static void addCount1(){
            for (int i = 0; i < 1000; i++) {
                count++;
            }
            System.out.println(Thread.currentThread().getName()
            + "count" + count);
        }
        @Override
        public void run() {
            addCount1();
        }
    }
}
```

3.4.3 常用的原子类进行自增自减操作

`i++` 操作不是原子操作，除了使用 `synchronize` 进行同步外，也可以使用 `AtomicInteger` 和 `AtomicLong` 原子类进行实现。

```
//使用AtomicInteger对象
static private AtomicInteger count = new AtomicInteger();
count.getAndIncrement();//相当于++count;
count.incrementAndGet();//相当于count++;
```

3.5 CAS

CAS(Compare And Swap) 是由硬件实现的。
CAS 可以将 read-modify-write 这类操作转换为原子操作。
`i++` 自增操作包括三个子操作:

- 读取 `i` 变量值
- 对 `i` 的值 +1
- 把 `i` 加 1 之后的值保存到主内存

CAS 原理：在把数据更新到主内存时，再次读取主内存变量的值 `value`，如果现在变量的值 `value` 与期望的值（操作起始时读取的值 `expectedValue`）一样就更新。

```
public class CASTest {
    public static void main(String[] args) {
        CASCounter casCounter = new CASCounter();
        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable(){
                @Override
                public void run() {
                    System.out.println(Thread.currentThread().getName()
                        + "count" + casCounter.incrementAndGet());
                }
            }).start();

        }
        //System.out.println(casCounter.getValue());
    }
}

class CASCounter {
    //使用 volatile 修饰 value 值，使线程可见
    volatile private long value;

    public long getValue() {
        return value;
    }

    //定义一个 compare and swap 方法

    /**
     *
     * @param expectValue   是自增前主内存的值
     * @param newValue  自增后的值
     * @return
     */
    private boolean compareAndSwap(long expectValue, long newValue) {
        //如果 value 的值与期望的 expectedValue 值一样
        //就把当前的 value 字段替换为 newValue 值
        synchronized (this) {
            if (value == expectValue){
                this.value = newValue;
                return true;
            }
            return false;
        }
```

```
    }

    //定义自增的方法
    public long incrementAndGet(){
        long oldValue;
        long newValue;
        do {
            oldValue = value;
            newValue = oldValue + 1;
        }while (!compareAndSwap(oldValue,newValue));
        return newValue;
    }
}
```

CAS 实现原子操作背后有一个假设：共享变量的当前值与当前线程提供的期望值相同，就认为这个变量没有被其他线程修改过。实际上该假设不一定总是成立。

3.5.1 ABA 问题

ABA 问题是 CAS 机制中出现的一个问题：一个线程把数据 A 变为 B 然后又重新变为 A，此时另外一个线程读取时，发现仍然是 A，就误以为该数据没有改变过。

规避方法：为共享变量引入一个修订号（时间戳），每次修改共享变量时，相应的修订号就会增加 1。通过对修订号就可以准确的判断变量是否被其他线程修改过。`AtomicStampedReference` 类就是基于这种思想实现的。

3.6 原子变量类

原子变量类是基于 CAS 实现的，当对共享变量进行 read-modify-write 更新操作时，通过原子变量类可以保障操作的原子性与可见性。对变量进行 read-modify-write 更新操作是指当前的操作不是一个简单的操作，而是变量的新值依赖变量的旧值。如自增自减操作。`volatile` 只能保证可见性，无法保证原子性，原子变量类内部就是借助一个 `volatile` 变量，并且保证了该变量的 read-modify-write 操作的原子性，有时把原子变量看作增强的 `volatile` 变量。

原子变量类有 12 个：

- 基础数据型：`AtomicInteger`，`AtomicLong`，`AtomicBoolean`
- 数组型：`AtomicIntegerArray`，`AtomicLongArray`，`AtomicReferenceArray`
- 字段更新器：`AtomicIntegerFieldUpdater`，`AtomicLongFieldUpdater`，`AtomicReferenceFieldUpdater`
- 引用型：`AtomicReference`，`AtomicStampedReference`，`AtomicMarkableReference`

3.6.1 `AtomicLong`

代码如下：

```
import java.util.concurrent.atomic.AtomicLong;

/**
 * @program: DataStructures
 * @description 使用原子变量类定义一个计数器
 * 该计数器在整个程序中都可以使用，所有的地方都使用这一个计数器
 * 这个计数器就可以设计为单例
 * @author: 戛剑生
 * @creat: 2021-03-05 14:38:30
 **/
public class Indicator {
    //构造方法私有化
    private Indicator() {
    }

    //定义一个私有的本类静态的对象
    private static final Indicator INSTANCE = new Indicator();

    //提供一个公共静态的方法返回该类唯一实例
    public static Indicator getInstance() {
        return INSTANCE;
    }

    //使用源自变量类保存请求总数
    private final AtomicLong requestCount = new AtomicLong(0);//记录请求总数
    private final AtomicLong successCount = new AtomicLong(0);//记录处理成功总数
    private final AtomicLong failCount = new AtomicLong(0);//记录处理失败总数

    //有新的请求
    public void newRequestReceive() {
        requestCount.incrementAndGet();
    }

    //处理成功的
    public void requestProcessSuccess() {
        successCount.incrementAndGet();
    }

    //处理失败的
    public void requestProcessFail() {
        failCount.incrementAndGet();
    }
```

```
    //查看总数，成功数，失败数
    public long getRequestCount() {
        return requestCount.get();
    }

    public long getSuccessCount() {
        return successCount.get();
    }

    public long getFailCount() {
        return failCount.get();
    }
}

import java.util.Random;

/**
 * @program: DataStructures
 * @description
 *      模拟服务器的请求总数，处理成功数，处理失败数
 * @author: 戛剑生
 * @creat: 2021-03-05 14:35:15
 **/
public class Test {
    public static void main(String[] args) {
        //通过线程模拟请求
        //在实际应用中可以在 ServletFilter 中调用 Indicator 计数器相关方法
        for (int i = 0; i < 10000; i++) {
            int finalI = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
                    //每个线程就是一个请求
                    //请求总数要加 1
                    Indicator.getInstance().newRequestReceive();
                int num = new Random().nextInt( );
                if (num % 2 == 0){//用随机数模拟请求成功或失败，
                    //偶数表示成功
                    Indicator.getInstance().requestProcessSuccess();
                }else{
                    Indicator.getInstance().requestProcessFail();
                }
                }
```

```
            }).start();
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
        System.out.println(" 发起的总请求数: " +
            Indicator.getInstance().getRequestCount());
        System.out.println(" 发起的成功请求数: " +
            Indicator.getInstance().getSuccessCount());
        System.out.println(" 发起的失败请求数: " +
            Indicator.getInstance().getFailCount());
    }
}
```

3.6.2 AtomicIntegerArray

原子更新数组。

代码如下：

```
import java.util.concurrent.atomic.AtomicIntegerArray;

/**
 * @program: DataStructures
 * @description
 *      在多线程中使用 AtomicIntegerArray 原子数组
 * @author: 戛剑生
 * @creat: 2021-03-05 15:33:14
 **/
public class Test02 {
    //定义原子数组
    static AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(10);
    public static void main(String[] args) {
        //定义线程数组
        Thread[] threads = new Thread[10];
        //给线程数组的每个线程赋值
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new AddThread();
        }
        //开启子线程
        for (Thread thread:
             threads) {
            thread.start();
```

```
        }

        //在所有子线程执行完后查看原子数组中各个元素的值
        //把所有的子线程合并到当前的主线程中
        for (Thread thread:
                threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(atomicIntegerArray);
    }
    //定义一个线程类，在线程类中修改原子数组
    static class AddThread extends Thread{

        @Override
        public void run() {
            //把原子数组的每个元素自增 1000 次
            for (int i = 0; i < 1000; i++) {
                for (int j = 0; j < atomicIntegerArray.length(); j++) {
                    atomicIntegerArray.getAndIncrement(j);
                }
            }

        }
    }
}
```

3.6.3 `AtomicIntegerFieldUpdater`

`AtomicIntegerFieldUpdater` 可以对原子整数字段进行更新。要求：

- 字符必须使用 `volatile` 修饰, 使得线程之间可见。
- 只能是实例变量不能是静态变量，也不能使用 `final` 修饰。

代码实现

```
public class User {
    int id;
    volatile int age;

    public User(int id, int age) {
        this.id = id;
```

```
        this.age = age;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", age=" + age +
                '}';
    }
}

public class SubThread extends Thread{
    private User user;//要更新的 User 对象
    //创建一个 AtomicIntegerFieldUpdater 更新器
    //AtomicIntegerFieldUpdater 是抽象类，不能直接创建对象
    private AtomicIntegerFieldUpdater<User> updater =
            AtomicIntegerFieldUpdater.newUpdater(User.class,"age");

    public SubThread(User user) {
        this.user = user;
    }

    @Override
    public void run() {
        //在子线程中对 user 对象的 age 字段自增
        for (int i = 0; i < 10; i++) {
            System.out.println(updater.getAndIncrement(user));
        }
    }
}

public class Test {
    public static void main(String[] args) {
        User user = new User(12, 10);
        //开启十个线程
        for (int i = 0; i < 10; i++) {
            new SubThread(user).start();
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        System.out.println(user);
    }
}
```

3.6.4 AtomicReference

可以原子读写一个引用类型的数据。

```
import java.util.concurrent.atomic.AtomicReference;

public class Test01 {
    public static void main(String[] args) {
        //创建 100 个子线程修改字符串
        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    if (atomicReference.compareAndSet("abcd", "def")) {
                        System.out.println(Thread.currentThread().getName()
                            + " 把字符串更改为 def");
                    }
                }
            }).start();
        }
        //创建 100 个子线程修改字符串
        for (int i = 0; i < 100; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    if (atomicReference.compareAndSet("def", "abcd")) {
                        System.out.println(Thread.currentThread().getName()
                            + " 把字符串更改为 abcd");
                    }
                }
            }).start();
        }
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(atomicReference.get());
    }
```

```
    //创建一个 AtomicReference 对象
    static AtomicReference<String> atomicReference =
        new AtomicReference<>("abcd");
```

ABA 问题

```
package com.atWSN.atomics.atomicReference;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

/**
 * @program: DataStructures
 * @description 演示 AtomicReference 的 ABA 问题
 * @author: 戛剑生
 * @creat: 2021-03-05 16:38:25
 **/
public class Test02 {
    private static AtomicReference<String> atomicReference
            = new AtomicReference<>("abc");

    public static void main(String[] args) {
        //创建第一个线程, 先把 abc 字符串改为 def 再把字符串还原为 abc
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                atomicReference.compareAndSet("abc", "def");
                System.out.println(Thread.currentThread().getName() +
                        "---->" + atomicReference.get());
                atomicReference.compareAndSet("def", "abc");
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(1l);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(atomicReference.compareAndSet
                        ("abc", "xyz"));
            }
```

```
        });
        t1.setName("t1");
        t2.setName("t2");
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(atomicReference.get());
    }
}
```

使用时间戳解决 ABA 问题

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicStampedReference;

/**
 * @program: DataStructures
 * @description 使用 AtomicStampedReference 解决 ABA 问题
 * 在 AtomicStampedReference 有一个整数标记值 stamp
 * 每次执行 CAS 操作时，都要对比它的版本，即比较 stamp 的值
 * @author: 戛剑生
 * @creat: 2021-03-05 17:03:16
 **/
public class Test03 {
    //    private static AtomicReference<String> atomicReference
//            = new AtomicReference<>("abc");
    //定义 AtomicStampedReference 引用操作"abc" 字符串，指定初始化版本号为 0
    private static AtomicStampedReference<String> atomicStampedReference
            = new AtomicStampedReference<>("abc", 0);

    public static void main(String[] args) {
//创建第一个线程，先把 abc 字符串改为 def 再把字符串还原为 abc
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                //每次修改，版本号加 1
                atomicStampedReference.compareAndSet
                        ("abc", "def",
```

```
                            atomicStampedReference.getStamp(),
                    atomicStampedReference.getStamp() + 1);
            System.out.println(Thread.currentThread().getName()
                    + "--" + atomicStampedReference.getReference());
            atomicStampedReference.compareAndSet("def",
                    "abc", atomicStampedReference.getStamp(),
                    atomicStampedReference.getStamp() + 1);
        }
    });
    Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
            int stamp = atomicStampedReference.getStamp();//获得版本号
            try {
                TimeUnit.SECONDS.sleep(1l);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(atomicStampedReference.
                    compareAndSet("abc",
                            "xyz",stamp,stamp+1));
        }
    });
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(atomicStampedReference.getReference());
}
```

第四章　线程间通信

4.1　等待/通知机制

4.1.1　什么是等待/通知机制

在单线程编程中，要执行的操作需要满足一定条件才能执行，可以把这个操作放在`if`语句块中。

在多线程编程中，可能某一线程 A 条件没有满足，只是暂时的，稍后其他线程 B 可能会更新该条件使得 A 线程的条件得到满足。可以将 A 线程暂停，直到它的条件得到满足后，再将 A 线程唤醒。伪代码：

```
atomics{    //原子操作
    while(条件不成立){
        等待;
    }

    当前线程被唤醒条件满足后，继续执行下面的操作
}
```

4.1.2　等待/通知机制的实现

4.1.2.1　`wait()`方法

`Object`类中的`wait()`方法可以使执行当前代码的线程等待，暂停执行，直到接到通知或被中断为止。

注意：

- `wait()`方法只能在同步代码块中由锁对象调用。
- 调用`wait()`方法后，当前线程会暂停并立即释放锁。

```
//在调用wait()方法前获得对象的内部锁
synchronized(锁对象){
    while(条件){//该条件不成立
        //通过锁对象调用wait()方法暂停线程
        锁对象.wait()
    }
    //线程的条件满足了继续向下执行
}
```

```
/**
 * @program: DataStructures
 * @description
 *      wait() 会使线程等待
 *      需要放在同步代码块中通过锁对象调用
 * @author: 戛剑生
 * @creat: 2021-03-05 21:07:28
 **/
public class Test02 {
    public static void main(String[] args) {
```

```
        try {
            String test  = "aaa";
            System.out.println(" 同步前");
            synchronized (test){
                System.out.println(" 同步代码块开始！");
                //需要通过锁对象调用
                //不是锁对象调用会产生 ILegalMonitorStateException 异常
                test.wait();    // 调用 wait 方法后当前线程就会等待同时释放锁对象
                //当前线程需要被唤醒
                //如果没有唤醒就会一直等待
                System.out.println("wait 后边的代码！");
            }
            System.out.println(" 同步代码块后边的代码");
        }catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("main 后边的其他代码……");
    }
```

4.1.2.2 `notify()` 方法

`Object` 类中的 `notify()` 方法可以唤醒线程，该方法也必须在同步代码块中由锁对象调用，没有使用锁对象调用 `wait()` / `notify()` 会抛出 `ILegalMonitorStateException` 异常。如果有多个等待的线程，`notify()` 需要等当前同步代码块执行完后才会释放锁对象，所以一般将 `noyify()` 方法放在同步代码块的最后。它的伪代码如下

```
synchronized(锁对象){
    //执行修改保护条件的代码
    //唤醒其他线程
    锁对象.notify();
}
```

```
/**
 * @program: DataStructures
 * @description 需要通过 notify() 唤醒等待的线程
 * @author: 戛剑生
 * @creat: 2021-03-05 21:18:43
 **/
public class Test03 {
    public static void main(String[] args) {
        final String LOCK = "lock";//定义一个字符串作为锁对象
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (LOCK) {
```

```
                System.out.println(Thread.currentThread().getName()
                    + " 开始等待" + System.currentTimeMillis());
                try {
                    LOCK.wait();//线程等待，会释放锁对象
                                //当前线程转入 WAITING 等待状态
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()
                    + " 结束等待" + System.currentTimeMillis());
            }
        }
    });
    t1.setName("t1");

    //定义第二个线程负责唤醒第一个线程
    Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
            //notify() 方法也需要在同步代码块中由锁对象调用
            synchronized (LOCK) {
                System.out.println(Thread.currentThread().getName()
                    + " 开始唤醒" + System.currentTimeMillis());

                LOCK.notify();//唤醒在 LOCK 锁对象上等待的某一个线程
                //
                System.out.println(Thread.currentThread().getName()
                    + " 结束唤醒" + System.currentTimeMillis());
            }
        }
    });
    t2.setName("t2");

    t1.start();//开启 t1 线程，t1 线程等待
    try {
        Thread.sleep(3000);//为了确保等待，让主线程睡 3s
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    t2.start();//t1 线程开启 3 秒后，再开启 t2 线程唤醒 t1 线程
}
```

`notify()` 不会立即释放锁对象

```
import java.util.ArrayList;
import java.util.List;

/**
 * @program: DataStructures
 * @description notify() 不会立即释放锁对象
 * @author: 戛剑生
 * @creat: 2021-03-05 21:18:43
 **/
public class Test04 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
//        final String list = "lock";//定义一个字符串作为锁对象
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (list) {
                    System.out.println(Thread.currentThread().getName()
                        + " 开始运行" + System.currentTimeMillis());
                    try {
                        if (list.size() != 5) {
                            list.wait();
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()
                        + " 结束运行" + System.currentTimeMillis());
                }
            }
        });
        t1.setName("t1");

        //定义第二个线程负责唤醒第一个线程
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                //notify() 方法也需要在同步代码块中由锁对象调用
                synchronized (list) {
                    System.out.println(Thread.currentThread().getName()
                        + " 运行开始" + System.currentTimeMillis());
                    for (int i = 0; i < 10; i++) {
                        System.out.println(Thread.currentThread().getName()
```

```
                        + " 添加第" + (i + 1) + " 个数据");
                    list.add("data -- " + (i + 1));
                    if (list.size() == 5) {
                        list.notify();
                        System.out.println
                            (Thread.currentThread().getName()
                                + " 已发出唤醒通知");
                    }
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                }

                //
                System.out.println(Thread.currentThread().getName()
                    + " 运行结束" + System.currentTimeMillis());
            }
        }
    });
    t2.setName("t2");

    t1.start();//开启 t1 线程, t1 线程等待
    try {
        Thread.sleep(3000);//为了确保等待, 让主线程睡 3s
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    t2.start();//t1 线程开启 3 秒后, 再开启 t2 线程唤醒 t1 线程
}
}
```

4.1.3 `interrupt()` 方法会中断 `wait()`

当线程处于 `wait()` 等待状态时，调用线程对象的 `interrupt()` 方法会中断线程的等待状态。`wait()` 被中断会产生 `InterruptException`。

```
package com.atWSN.thread.wait;

/**
 * @program: DataStructures
 * @description
```

```
 *      中断线程会唤醒线程的等待
 * @author: 戛剑生
 * @creat: 2021-03-06 08:26:07
 **/
public class Test05 {
    public static void main(String[] args) {
        SubThread subThread = new SubThread();
        subThread.setName("t");
        subThread.start();

        try {
            Thread.sleep(2000);//主线程睡眠 2 秒，确保子线程处于 wait() 状态
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        subThread.interrupt();
    }

    static private final Object OBJ = new Object();

    static class SubThread extends Thread {
        @Override
        public void run() {
            synchronized (OBJ) {
                try {
                    System.out.println(Thread.currentThread().getName() + " 开始等待");
                    OBJ.wait();
                    System.out.println(Thread.currentThread().getName() + " 结束等待");
                } catch (InterruptedException e) {
//                    e.printStackTrace();
                    System.out.println("wait 等待被中断了");
                    System.out.println(e.getMessage());
                }
            }
        }
    }
```