目 录

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

```
//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

```
/**
```

```
 * 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());
                }
            }
        }
    }
```

4.1.4 `notify()` 与 `notifyAll()`

`notify()` 一次只能唤醒一个线程，如果有多个等待的线程，只能随机的唤醒其中的某一个；想要唤醒所有等待的线程，需要调用 `notifyAll()`

```
/**
 * @program: DataStructures
 * @description
 *      notifyAll 唤醒所有等待的线程
 * @author: 戛剑生
 * @creat: 2021-03-06 08:47:17
 **/
public class Test06 {
    public static void main(String[] args) {
        Object LOCK = new Object();
        SubThread t1 = new SubThread(LOCK);
        t1.setName("t1");
        SubThread t2 = new SubThread(LOCK);
        t2.setName("t2");
        SubThread t3 = new SubThread(LOCK);
        t3.setName("t3");
        SubThread t4 = new SubThread(LOCK);
        t4.setName("t4");

        t1.start();
        t2.start();
        t3.start();
        t4.start();

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //唤醒子线程
        synchronized (LOCK){
            LOCK.notify();//调用一次 notify 只能随机唤醒其中一个线程
            //其他等待的线程依然处于等待状态
            //对于处于等待状态的线程来说，错过了通知信号，这种现象也称为信号丢失
        }
        //唤醒全部子线程
        synchronized (LOCK){
            LOCK.notifyAll();//调用一次 notifyAll 唤醒其中所有线程
        }
    }
    static class SubThread extends Thread{
        private Object LOCK;
```

```
        public SubThread(Object LOCK) {
            this.LOCK = LOCK;
        }

        @Override
        public void run() {
            synchronized (LOCK){
                try {
                    System.out.println(Thread.currentThread().getName()
                        + " 开始等待！");
                    LOCK.wait();
                    System.out.println(Thread.currentThread().getName()
                        + " 结束等待！");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

            }
        }
    }
```

4.1.5 `wait(long)` 的使用

`wait(long)` 是带有 `long` 类型参数的 `wait()` 等待，如果在参数指定的时间内没有被唤醒。超时后会自动唤醒。

```
/**
 * @program: DataStructures
 * @description
 *      wait(long)
 * @author: 戛剑生
 * @creat: 2021-03-06 19:35:45
 **/
public class Test07 {
    public static void main(String[] args) {
        final Object LOCK = new Object();
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                synchronized (LOCK){
                    System.out.println(Thread.currentThread().getName()
                        + " 开始等待！");
                    try {
                        LOCK.wait(5000);//如果 5 秒内没有被唤醒，会自动唤醒
```

```
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName()
                        + " 结束等待！");
                }
            }
        });
        thread.setName("t1");
        thread.start();
    }
}
```

4.1.6 通知过早

线程 `wait()` 后,可以调用 `notify()` 唤醒线程,如果 `notify()` 唤醒的过早,在等待之前就调用了 `notify()`，可能会打乱程序正常的执行逻辑。

4.1.7 `wait()` 等待的条件发生了变化

在使用 `wait` / `notify()` 模式时，注意 `wait` 条件发生了变化，也可能会造成程序的逻辑混乱。

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

/**
 * @program: DataStructures
 * @description 定义一个集合
 * 定义一个线程向集合中添加数据，添加完数据后通知另外的线程从集合中取数据
 * 定义一个线程从集合中取数据，如果集合中没有数据就等待
 * @author: 戛剑生
 * @creat: 2021-03-06 20:24:49
 **/
public class Test10 {
    public static final Object LOCK = new Object();
    //定义一个 List 集合
    static List list = new ArrayList<>();

    public static void main(String[] args) {
        Thread t1 = new Subtract();
        t1.setName(" 取数据的线程 1");
        Thread t2 = new Add();
        t2.setName(" 添加数据的线程");
        //测 1: 先开启添加数据的线程，再开启一个取数据的线程
```

```
        //大多数情况下会正常取数据
//        t2.start();
//        t1.start();
        //测 2：先开启取数据的线程，再开启添加数据的线程
        //取数据的线程会先等待，等到添加数据之后，再取数据
//        t1.start();
//        t2.start();
        //测 3：开启两个取数据的线程，再开启添加数据的线程
        Thread t3 = new Subtract();
        t3.setName(" 取数据的线程 2");
        t1.start();
        t3.start();
        t2.start();
        /**
         * 某次的执行结果如下
         * 取数据的线程 1 开始等待!
         * 取数据的线程 2 开始等待!
         * 添加数据的线程添加数据!
         * 添加数据的线程完成添加数据!
         * 取数据的线程 2 结束等待!
         * 取数据的线程 2 从集合中取了 x 后，集合中剩余的数据数量：0
         * 取数据的线程 1 结束等待!
         * Exception in thread " 取数据的线程 1"
         * java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
         *
         * 出现异常的原因：向集合中添加了一次数据，但取了两次
         * 解决方案：等待的线程被唤醒后，再判断一次集合中是否有数据可取
         * 需要把 subtract 方法中的 if 改为 while
         */
    }

    //定义方法从集合中取数据
    public static void subtract() {
        synchronized (LOCK) {
//            if (list.isEmpty()) {
            while (list.isEmpty()) {
                System.out.println(Thread.currentThread().getName()
                    + " 开始等待! ");
                try {
                    LOCK.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
                System.out.println(Thread.currentThread().getName()
                    + " 结束等待！");
            }
            Object data = list.remove(0);
            System.out.println(Thread.currentThread().getName()
                + " 从集合中取了" + data + " 后，集合中剩余的数据数量："
                    + list.size());
        }
    }

    //向集合中添加数据
    //添加完 2 数据后，通知等待的线程取数据
    public static void add() {
        synchronized (LOCK) {
            System.out.println(Thread.currentThread().getName()
                + " 添加数据！");
            list.add("x");
            LOCK.notifyAll();
            System.out.println(Thread.currentThread().getName()
                + " 完成添加数据！");
        }
    }

    //定义线程类调用 subtract
    static class Subtract extends Thread {
        @Override
        public void run() {
            subtract();
        }
    }

    //定义线程类调用 add
    static class Add extends Thread {
        @Override
        public void run() {
            add();
        }
    }
```

4.1.8 生产者消费者模式

Java 中，负责生产数据的模块是生产者，负责使用数据的模块是消费者。生产者消费者解决数据的平衡问题，即先有数据，然后才能使用，没有数据时，消费者需要等待。

4.1.8.1 生产者-消费者：操作数据

操作数据的类

```
/**
 * @program: DataStructures
 * @description
 *      定义一个操作数据的类
 * @author:戛剑生
 * @creat: 2021-03-06 21:33:03
 **/
public class ValueOP {
    private String  value = "";
    //定义方法修改value字段的值
    public void setValue(){
        //如果value不是空串"", 就等待
        synchronized (this){
            while (!value .equals("")){
                try {
                    System.out.println("当前有数据, 无需生产");
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //如果是空串, 就设置value字段的值
            String value = System.currentTimeMillis()+" - "+System.nanoTime();
            this.value = value;
            System.out.println("set设置的值是: " + value);
            System.out.println("生产成功");
            System.out.println();
            this.notifyAll();
        }
    }
    //定义方法读取value字段的值
    public void getValue(){
        synchronized (this){
            //如果字符串是空串就等待
            while (value .equals("")){
                System.out.println("没有数据可以取" + Thread.currentThread().
                    getName() + "等待");
                System.out.println();
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            //不是空串, 读取字段的值
            System.out.println("get的值是: " + this.value);
            System.out.println("取数据结束");
```

```
            System.out.println();
            this.value = "";
            this.notifyAll();
        }
    }
}
```

生产者

```
/**
 * @program: DataStructures
 * @description
 *      生产者
 * @author:戛剑生
 * @creat: 2021-03-06 21:44:04
 **/
public class ProducerThread extends Thread{
    private ValueOP obj;

    public ProducerThread(ValueOP obj) {
        this.obj = obj;
    }

    @Override
    public void run() {
        while (true){
            System.out.println(Thread.currentThread().getName() + "正在生产");
            obj.setValue();
        }
    }
}
```

消费者

```
/**
 * @program: DataStructures
 * @description
 *      消费者
 * @author:戛剑生
 * @creat: 2021-03-06 21:44:04
 **/
public class ConsumerThread extends Thread{
    //消费者使用数据，就是使用ValueOP类的value字段值
    private ValueOP obj;

    public ConsumerThread(ValueOP obj) {
        this.obj = obj;
    }

    @Override
    public void run() {
```

```
        while (true){
            System.out.println(Thread.currentThread().getName() + "正在消费");
            obj.getValue();
        }
    }
}
```

多生产者多消费者模型

```
/**
 * @program: DataStructures
 * @description 测试多生产，多消费的情况
 * @author:戛剑生
 * @creat: 2021-03-06 21:32:37
 **/
public class Test02 {
    public static void main(String[] args) {
        ValueOP obj = new ValueOP();

        Thread producer1 = new ProducerThread(obj);
        Thread producer2 = new ProducerThread(obj);
        Thread producer3 = new ProducerThread(obj);
        Thread producer4 = new ProducerThread(obj);

        Thread consumer1 = new ConsumerThread(obj);
        Thread consumer2 = new ConsumerThread(obj);
        Thread consumer3 = new ConsumerThread(obj);
        Thread consumer4 = new ConsumerThread(obj);
        Thread consumer5 = new ConsumerThread(obj);
        Thread consumer6 = new ConsumerThread(obj);

        producer1.setName("生产者线程1");
        producer2.setName("生产者线程2");
        producer3.setName("生产者线程3");
        producer4.setName("生产者线程4");
        consumer1.setName("消费者线程1");
        consumer2.setName("消费者线程2");
        consumer3.setName("消费者线程3");
        consumer4.setName("消费者线程4");
        consumer5.setName("消费者线程5");
        consumer6.setName("消费者线程6");
        producer1.start();
        producer2.start();
        producer3.start();
        producer4.start();
        consumer1.start();
        consumer2.start();
        consumer3.start();
        consumer4.start();
        consumer5.start();
        consumer6.start();
```

```
43          //结果：生产与消费交替运行
44      }
45  }
```

4.1.8.2 操作集合

使生产者把数据存储到 `List` 集合中，消费者从 `List` 集合中取数据，使用 `List` 集合模拟栈。

操作栈的类

```
1   import java.util.ArrayList;
2   import java.util.List;
3   import java.util.Random;
4
5   /**
6    * @program: DataStructures
7    * @description
8    * @author:戛剑生
9    * @creat: 2021-03-07 08:08:29
10   **/
11  public class MyStack {
12      private List<String> list = new ArrayList<>();//定义一个集合模拟栈
13      private static final int capacity = 2;//定义集合的最大容量
14      private int size = 0;
15
16      //定义方法模拟入栈
17      public synchronized void push() {
18          //栈中的数据已满，就等待
19          while (list.size() >= capacity) {
20              System.out.println("数据栈容量已满，无法存入数据，请等待！");
21              System.out.println(Thread.currentThread().getName() + "开始等待！");
22              try {
23                  this.wait();
24              } catch (InterruptedException e) {
25                  e.printStackTrace();
26              }
27              System.out.println(Thread.currentThread().getName() + "结束等待！");
28          }
29          String data = "data:" + new Random().nextInt(101);
30          list.add(data);
31          size++;
32          System.out.println(Thread.currentThread().getName() + "添加了数据：" +
                data);
33          this.notifyAll();
34      }
35
36      public synchronized void pop() {
37          while (list.isEmpty()) {
38              System.out.println("当前数据栈为空，无法取出数据！进入等待");
39              System.out.println(Thread.currentThread().getName() + "开始等待");
40              try {
```

```
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "结束等待");
        }
        String data = list.remove(size - 1);
        size--;
        System.out.println(Thread.currentThread().getName() + "取出了数据: " +
            data);
        this.notifyAll();
    }
}
```

生产者

```
/**
 * @program: DataStructures
 * @description 生产者
 * @author:戛剑生
 * @creat: 2021-03-07 08:22:10
 **/
public class ProducerThread extends Thread{
    private MyStack myStack;
    public ProducerThread(MyStack myStack){
        this.myStack = myStack;
    }
    @Override
    public void run() {
        while (true){
            myStack.push();
        }
    }
}
```

消费者

```
/**
 * @program: DataStructures
 * @description 消费者
 * @author:戛剑生
 * @creat: 2021-03-07 08:22:10
 **/
public class ConsumerThread extends Thread{
    private MyStack myStack;
    public ConsumerThread(MyStack myStack){
        this.myStack = myStack;
    }
    @Override
    public void run() {
        while (true){
```

```
            myStack.pop();
        }
    }
}
```

多生产者多消费者模型测试

```
public class Test02 {
    public static void main(String[] args) {
        MyStack obj = new MyStack();

        Thread producer1 = new ProducerThread(obj);
        Thread producer2 = new ProducerThread(obj);
        Thread producer3 = new ProducerThread(obj);
        Thread producer4 = new ProducerThread(obj);

        Thread consumer1 = new ConsumerThread(obj);
        Thread consumer2 = new ConsumerThread(obj);
        Thread consumer3 = new ConsumerThread(obj);
        Thread consumer4 = new ConsumerThread(obj);
        Thread consumer5 = new ConsumerThread(obj);
        Thread consumer6 = new ConsumerThread(obj);

        producer1.setName("生产者线程1");
        producer2.setName("生产者线程2");
        producer3.setName("生产者线程3");
        producer4.setName("生产者线程4");
        consumer1.setName("消费者线程1");
        consumer2.setName("消费者线程2");
        consumer3.setName("消费者线程3");
        consumer4.setName("消费者线程4");
        consumer5.setName("消费者线程5");
        consumer6.setName("消费者线程6");
        producer1.start();
        producer2.start();
        producer3.start();
        producer4.start();
        consumer1.start();
        consumer2.start();
        consumer3.start();
        consumer4.start();
        consumer5.start();
        consumer6.start();
        //结果：生产与消费交替运行
    }
}
```

4.2 通过管道流实现线程间的通信

在 java.io 包中的 PipeStream 管道流用于在线程之间传送数据，一个线程发送数据到输出管道，另一个线程从输入管道中读取数据。

相关的类包括：字节流 PipedInputStream 和 PipedOutputStream，字符流 PipedReader 和 PipedWriter.

```
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;

/**
 * @program: DataStructures
 * @description: 使用PipedInputStream和PipedOutputStream管道字节流在线程之间传递数据
 * @author:戛剑生
 * @creat: 2021-03-07 08:58:15
 **/
public class Test01 {
    public static void main(String[] args) {
        //定义管道字节流
        PipedInputStream pipedInputStream = new PipedInputStream();
        PipedOutputStream pipedOutputStream = new PipedOutputStream();
        try{
            pipedInputStream.connect(pipedOutputStream);
            //创造线程向管道流中写入数据
            new Thread(new Runnable(){
                @Override
                public void run() {
                    writeData(pipedOutputStream);
                }
            }).start();
            //创造线程从管道流中读取数据
            new Thread(new Runnable(){
                @Override
                public void run() {
                    readData(pipedInputStream);
                }
            }).start();
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        try {
                pipedOutputStream.close();
                pipedInputStream.close();
            } catch (IOException e) {
```

```
                e.printStackTrace();
            }
    }

    //定义方法向管道流中写入数据
    //从线程中把数据写入管道，需要一个输出流管道用于接收
    public static void writeData(PipedOutputStream out) {
        //分别把0~100之间的数写入管道中
        try {
            for (int i = 0; i < 100; i++) {
                String data = "" + (i + 1);
                out.write(data.getBytes());//把字节数组写入到输出管道流中
            }

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                out.close();//关闭管道流
            } catch (IOException e1) {
                e1.printStackTrace();
            }
        }
    }

    //定义方法从管道流中读取数据
    public static void readData(PipedInputStream in) {
        try {
            byte[] bytes = new byte[1024 * 4];
            int len = 0;
            while (true) {
                len = in.read(bytes);
                if (len == -1){
                    break;
                }
                System.out.println(new String(bytes,0,len));
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                in.close();
            } catch (IOException e1) {
                e1.printStackTrace();
            }
        }
    }
}
```

4.3 ThreadLocal 的使用

除了控制资源的访问外，还可以通过增加资源来保证线程安全，ThreadLocal 主要是解决为每个线程绑定自己的值。

```java
/**
 * @program: DataStructures
 * @description: ThreadLocal的使用
 * @author:戛剑生
 * @creat: 2021-03-07 09:47:26
 **/
public class Test {
    //定义一个ThreadLocal对象
    static ThreadLocal threadLocal = new ThreadLocal();

    //定义线程类
    static class SubThread extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 20; i++) {
                //设置线程关联的值
                threadLocal.set(Thread.currentThread().getName() + "-" + (i + 1));
                //调用get方法读取关联的值
                System.out.println(Thread.currentThread().getName() + " value = "
                    + threadLocal.get());
            }
        }
    }

    public static void main(String[] args) {
        SubThread t1 = new SubThread();
        SubThread t2 = new SubThread();
        t1.setName("t1");
        t2.setName("t2");
        t1.start();
        t2.start();
    }
}
```

```java
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @program: DataStructures
 * @description: 在多线程环境中，把字符串转换为日期对象
 * 多个线程使用同一个 SimpleDateFormat 对象可能会产生线程安全问题
 * 为每一个线程指定自己的 SimpleDateFormat 对象，使用 ThreadLocal
 * @author: 戛剑生
```

```
 * @creat: 2021-03-07 09:56:11
 **/
public class Test02 {
    //定义 SimpleDateFormat 对象，该对象可以字符串转换为日期
    private static SimpleDateFormat sdf =
        new SimpleDateFormat("yyyy 年 MM 月 dd 日 HH:mm:ss");
    static ThreadLocal<SimpleDateFormat> threadLocal = new ThreadLocal<>();

    static class ParseDate implements Runnable {
        private int i = 0;

        public ParseDate(int i) {
            this.i = i;
        }

        @Override
        public void run() {
            try {
                String str = (i % 60 < 10 ? ("0" + i % 60) : ("" + i % 60));
                String text = "2021 年 03 月 07 日 09:56:" + str;//构建日期字符串
//                Date date = sdf.parse(text);//把字符串转换为日期
//                System.out.println(i + "-" + date);
                //先判断当前线程是否有 SimpleDateFormat 对象，
                //如果当前线程没有 SimpleDateFormat 对象就创建一个
                //如果有就直接使用
                if (threadLocal.get() == null) {
                    threadLocal.set
                        (new SimpleDateFormat("yyyy 年 MM 月 dd 日 HH:mm:ss"));
                }
                Date date =threadLocal.get().parse(text);
                System.out.println(i + "-" + date);
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        //创建 100 个线程
        for (int i = 0; i < 100; i++) {
            new Thread(new ParseDate(i)).start();
        }
    }
```

```
}
```

4.3.1 初始化 ThreadLocal 的值

定义 ThreadLocal 子类，在子类中重写 initialValue() 为其指定初始值

```
import java.util.Date;
import java.util.Random;

/**
 * @program: DataStructures
 * @description ThreadLocal 初始值: 定义一个子类
 *              在子类中重写 initialValue() 为其指定初始值
 * @author: 戛剑生
 * @creat: 2021-03-07 10:49:32
 **/
public class Test03 {
    //定义 ThreadLocal 的子类
    static class SubThreadLocal extends ThreadLocal<Date> {
        //重写 initialValue(), 设置初始值
        @Override
        protected Date initialValue() {
//            return new Date();//把当前日期设置为初始值
            return new Date(System.currentTimeMillis() - 1000 * 60 * 15);
            //把 15 分钟之前设置为初始值
        }
    }

    //    //定义 ThreadLocal 对象
//    static ThreadLocal threadLocal = new ThreadLocal();
    //定义 ThreadLocal 对象
    static SubThreadLocal threadLocal = new SubThreadLocal();

    //定义线程类
    static class SubThread extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                //第一次调用 ThreadLocal 的 get 方法会返回 null
                System.out.println(i + 1 + "---"
                    + Thread.currentThread().getName()
                        + " value = " + threadLocal.get());
                //如果没有初始值就设置: 这里设置一个日期
                if (threadLocal.get() == null) {
```

```
                threadLocal.set(new Date());
            }
            try {
                Thread.sleep(new Random().nextInt(2000) + 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) {
    SubThread t1 = new SubThread();
    SubThread t2 = new SubThread();
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();
}
```

第五章 Lock显示锁

在 JDK5 中增加了 `Lock` 锁接口。

有 `ReentrantLock` 实现类，该锁称为可重入锁，它功能要比 `synchronized` 多。

5.1 锁的可重入性

锁的可重入是指一个线程获得一个对象锁后，再次请求该对象锁时是可以获得该对象锁的。

```
/**
 * @program: DataStructures
 * @description : 演示锁的可重入性
 * @author:戛剑生
 * @creat: 2021-03-07 15:11:04
 **/
public class Test01 {
    public  synchronized void sm1(){
        System.out.println("同步方法1: ");
        //线程执行sm1方法，默认this作为锁对象，在sm1()方法中调用了sm2方法，当前线程还是持有this锁对象的
        //sm2同步方法默认的锁对象是this对象，要执行sm2必须先获得this对象，当前this对象被当前线程持有，可以再次获得this对象，这就是锁的可重入性。
        sm2();
    }

    private  synchronized void sm2() {
        System.out.println("同步方法2: ");
        sm3();
    }

    private  synchronized void sm3() {
        System.out.println("同步方法3: ");
    }

    public static void main(String[] args) {
        Test01 t = new Test01();
        new Thread(new Runnable(){
            @Override
            public void run() {
                t.sm1();
            }
        }).start();
    }
}
```

5.2 ReentrantLock

5.2.1 ReentrantLock 的基本使用

调用 lock 方法获得锁，调用 unlock() 方法释放锁

5.2.1.1 Lock 锁的基本使用

lock 的基本使用

```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description : Lock锁的基本使用
 * @author:戛剑生
 * @creat: 2021-03-07 15:25:38
 **/
public class Test02 {
    //定义显示锁
    static Lock lock = new ReentrantLock();

    //定义方法
    public static void sm() {
        //先获得锁
        lock.lock();
        //下边这部分就是同步代码块
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + "--" + (i + 1));
        }
        //释放锁
        lock.unlock();
    }

    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                sm();
            }
        };
        for (int i = 0; i < 3; i++) {
            new Thread(r).start();
        }
    }
}
```

5.2.1.2 使用 Lock 锁同步不同方法中的同步代码块

```
import java.util.Random;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description : 使用 Lock 锁同步不同方法中的同步代码块
 * @author: 戛剑生
 * @creat: 2021-03-07 15:25:38
 **/
public class Test03 {
    //定义显示锁
    static Lock lock = new ReentrantLock();

    //定义方法 1
    public static void sm() {
        //经常在 try 代码块中获得锁，在 finally 子句中释放锁
        try {
            //先获得锁
            lock.lock();
            System.out.println(Thread.currentThread().getName() +
                " -- method1 -- " + System.currentTimeMillis());
            Thread.sleep(new Random().nextInt(1000) + 1);
            System.out.println(Thread.currentThread().getName() +
                " -- method1 -- " + System.currentTimeMillis());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            //释放锁
            lock.unlock();
        }
    }

    //定义方法 2
    public static void sm1() {
        //经常在 try 代码块中获得锁，在 finally 子句中释放锁
        try {
            //先获得锁
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " -- method2 -- " + System.currentTimeMillis());
            Thread.sleep(new Random().nextInt(1000) + 1);
```

```
            System.out.println(Thread.currentThread().getName()
                + " -- method2 -- " + System.currentTimeMillis());
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            //释放锁
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                sm();
            }
        };
        Runnable r2 = new Runnable() {
            @Override
            public void run() {
                sm1();
            }
        };
        for (int i = 0; i < 3; i++) {
            new Thread(r1).start();
        }
        for (int i = 0; i < 3; i++) {
            new Thread(r2).start();
        }
    }
}
```

5.2.1.3 锁的可重入性

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: ReentrantLock 锁的可重入性
 * @author: 戛剑生
 * @creat: 2021-03-07 15:47:24
 **/
public class Test04 {
```

```
    static class SubThread extends Thread {
        //定义锁
        private static Lock lock = new ReentrantLock();
        public static int num = 0;

        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                try {
                    lock.lock();
                    //可重入锁是指可以反复获得该锁
                    lock.lock();
                    num++;
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                } finally {
                    lock.unlock();
                    //上边使用几次锁这里就要释放几次
                    lock.unlock();
                }
            }
        }
    }

    public static void main(String[] args) {
        SubThread t1 = new SubThread();
        t1.setName("t1");
        SubThread t2 = new SubThread();
        t2.setName("t2");

        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(SubThread.num);
    }
}
```

5.2.2 lockInterruptibly()

lockInterruptibly()作用：如果当前线程未被中断则获得锁，如果当前线程被中断则抛出异常。

```
package com.atWSN.thread.lock.reentrant;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 演示 lockInterruptibly() 方法
 * @author: 戛剑生
 * @creat: 2021-03-07 16:31:47
 **/
public class Test05 {
    public static void main(String[] args) {
        Service service = new Service();
        Runnable r = new Runnable() {
            @Override
            public void run() {
                service.serviceMethod();
            }
        };

        Thread t1 = new Thread(r);
        t1.setName("t1");
        t1.start();
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Thread t2 = new Thread(r);
        t2.setName("t2");
        t2.start();
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        t2.interrupt();//t2 中断

        /**
         * 程序运行结果:
```

```
         * t1 -- 获得锁!
         * null
         * t2 -- 释放锁!
         * Exception in thread "t2" java.lang.IllegalMonitorStateException
         *         at java.util.concurrent.locks.
         *       ReentrantLock$Sync.tryRelease(ReentrantLock.java:151)
         *         at java.util.concurrent.
         *      locks.AbstractQueuedSynchronizer.
         *          release(AbstractQueuedSynchronizer.java:1261)
         *         at java.util.concurrent.locks.ReentrantLock.
         *      unlock(ReentrantLock.java:457)
         *         at com.atWSN.thread.lock.reentrant.
         *      Test05$Service.serviceMethod(Test05.java:57)
         *         at com.atWSN.thread.lock.reentrant.Test05$1.
         *      run(Test05.java:18)
         *         at java.lang.Thread.run(Thread.java:748)
         * t1 -- 释放锁!
         */
    }

    static class Service {
        private Lock lock = new ReentrantLock();

        public void serviceMethod() {
            try {
//                lock.lock();//获得锁定，即使调用了线程的 interrupt() 方法
                            //也没有真正中断线程
                lock.lockInterruptibly();//如果线程被中断了，不会获得锁，会抛出异常
                System.out.println(Thread.currentThread().getName()
                    + " -- 获得锁! ");
                //模拟执行一段耗时的操作
                for (int i = 0; i < Integer.MAX_VALUE >>> 2; i++) {
                    new StringBuilder();
                }
            } catch (Exception e) {
                System.out.println(e.getMessage());
            } finally {
                System.out.println(Thread.currentThread().getName()
                    + " -- 释放锁! ");
                lock.unlock();
            }
        }
    }
```

```
}
```

对于 `synchronized` 内部锁来说，如果一个线程在等待锁，只有两个结果：要么该线程获得锁继续执行，要么就保持等待。

对于 `ReentrantLock()` 可重入锁来说，提供另外一种可能。在等待锁的过程中，程序可以根据需要取消对锁的请求。

```
import java.util.Random;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 通过 ReentrantLock 的 lockInterruptibly() 方法避免死锁问题
 * @author: 戛剑生
 * @creat: 2021-03-07 20:25:02
 **/
public class Test06 {
    static class IntLock implements Runnable {
        //创建两个 ReentrantLock 对象
        public static ReentrantLock lock1 = new ReentrantLock();
        public static ReentrantLock lock2 = new ReentrantLock();
        int lockNumber;//定义整数变量决定使用哪个锁

        public IntLock(int lockNumber) {
            this.lockNumber = lockNumber;
        }

        @Override
        public void run() {
            try {
                if (lockNumber % 2 == 1) {
                    lock1.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName()
                        + " 获得锁" + lockNumber % 2 + " 还需要获得锁"
                            + (lockNumber % 2 + 1));
                    Thread.sleep(new Random().nextInt(1000));
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName()
                        + " 同时获得锁" + lockNumber % 2
                            + " 和锁" + (lockNumber % 2 + 1));
                } else {
                    lock2.lockInterruptibly();
                    System.out.println(Thread.currentThread().getName()
                        + " 获得锁" + (lockNumber % 2 + 2)
```

```
                        + " 还需要获得锁" + (lockNumber % 2 + 1));
                Thread.sleep(new Random().nextInt(1000));
                lock1.lockInterruptibly();
                System.out.println(Thread.currentThread().getName()
                    + " 同时获得锁" + (lockNumber % 2 + 2)
                        + " 和锁" + (lockNumber % 2 + 1));
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            if (lock1.isHeldByCurrentThread()) {//锁被当前线程持有就会释放
                lock1.unlock();
            }
            if (lock2.isHeldByCurrentThread()) {//锁被当前线程持有就会释放
                lock2.unlock();
            }
            System.out.println(Thread.currentThread().getName()
                + " 线程退出！");
        }
    }
}

public static void main(String[] args) {
    IntLock intLock1 = new IntLock(11);
    IntLock intLock2 = new IntLock(22);
    Thread t1 = new Thread(intLock1);
    Thread t2 = new Thread(intLock2);
    t1.setName("t1");
    t2.setName("t2");
    t1.start();
    t2.start();

    //在 main 线程中等待 3000 秒，如果还有线程没有结束就中断该线程
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    //可以中断任何一个线程来解决死锁
    if (t2.isAlive()) {
        t2.interrupt();//t2 线程会放弃对锁 1 的申请，同时释放锁 2
    }
```

```
//        try {
//            Thread.sleep(3000);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//        if (t2.isAlive()) {
//            t2.interrupt();
//        }
    }
```

5.2.3 tryLock() 方法

5.2.3.1 tryLock(long time,TimeUnit unit)

tryLock(long time,TimeUnit unit)的作用在给定等待时长内锁没有被其他线程持有，并且当前线程也没有被中断，则获得该锁。

通过该方法获得锁的限时等待。

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: tryLock(long time, TimeUnit unit) 的基本使用
 * @author: 戛剑生
 * @creat: 2021-03-07 21:36:23
 **/
public class Test07 {
    static class TimeLock implements Runnable {
        private static ReentrantLock lock = new ReentrantLock();//定义一个锁对象

        @Override
        public void run() {
            try {
                if (lock.tryLock(4, TimeUnit.SECONDS)) {//获得锁返回 true
                    System.out.println(Thread.currentThread().getName()
                        + " 获得锁");
                    System.out.println(" 执行相应的任务");
                    Thread.sleep(3000);//一个线程 1 获得锁并执行耗时任务
                    //该任务需要 3 秒钟
                    // 另一个线程 2 尝试获得锁，线程 2 两秒内还没获得锁的话就会放弃
                } else {//没有获得锁
                    System.out.println(Thread.currentThread().getName()
                        + " 没有获得锁！");
                }
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (lock.isHeldByCurrentThread()) {
                    lock.unlock();
                    System.out.println(Thread.currentThread().getName()
                        + " 释放锁！");
                }
            }
        }
    }

    public static void main(String[] args) {
        TimeLock timeLock = new TimeLock();
        Thread t1 = new Thread(timeLock);
        t1.setName("t1");
        Thread t2 = new Thread(timeLock);
        t2.setName("t2");
        Thread t3 = new Thread(timeLock);
        t3.setName("t3");
        t1.start();
        t2.start();
        t3.start();

    }
```

5.2.3.2 `tryLock()` 方法

仅在调用时锁定未被其他线程持有的锁。如果调用该方法时，锁对象被其他线程持有，则放弃。

调用该方法尝试获得锁，如果该锁没有被其他线程占用则返回 `true` 表示锁定成功。如果锁被其他线程占用则返回 `false`，不等待。

```
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: tryLock() 的基本使用，锁对象没有被其他线程持有的情况下
 *               才会获得该锁定
 * @author: 戛剑生
 * @creat: 2021-03-07 21:36:23
 **/
public class Test08 {
    static class TimeLock implements Runnable {
        private static ReentrantLock lock = new ReentrantLock();//定义一个锁对象
```

```
    @Override
    public void run() {
        try {
            if (lock.tryLock()) {//获得锁返回 true
                System.out.println(Thread.currentThread().getName()
                    + " 获得锁");
                System.out.println(" 执行相应的任务");
                Thread.sleep(3000);//一个线程 1 获得锁并执行耗时任务
                //该任务需要 3 秒钟
                // 另一个线程 2 尝试获得锁，线程 2 两秒内还没获得锁的话就会放弃
            } else {//没有获得锁
                System.out.println(Thread.currentThread().getName()
                    + " 没有获得锁！");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            if (lock.isHeldByCurrentThread()) {
                lock.unlock();
                System.out.println(Thread.currentThread().getName()
                    + " 释放锁！");
            }
        }
    }
}

public static void main(String[] args) {
    TimeLock timeLock = new TimeLock();
    Thread t1 = new Thread(timeLock);
    t1.setName("t1");
    Thread t2 = new Thread(timeLock);
    t2.setName("t2");
    Thread t3 = new Thread(timeLock);
    t3.setName("t3");
    t1.start();
    t2.start();
    t3.start();

}
```

5.2.3.3 tryLock 不会造成死锁

```
import java.util.Random;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 使用 tyrLock() 可以避免死锁
 * @author: 戛剑生
 * @creat: 2021-03-08 14:46:57
 **/
public class Test09 {
    static class IntLock implements Runnable {
        private static ReentrantLock lock1 = new ReentrantLock();
        private static ReentrantLock lock2 = new ReentrantLock();
        private int lockNum;

        public IntLock(int lockNum) {
            this.lockNum = lockNum;
        }

        @Override
        public void run() {
            if (lockNum % 2 == 0) {//偶数先获得锁 1 再获得锁 2
                while (true) {
                    try {
                        if (lock1.tryLock()) {
                            System.out.println(Thread.currentThread().getName()
                                + " 获得锁 1，还未获得锁 2");
                            Thread.sleep(new Random().nextInt(10));
                            try {
                                if (lock2.tryLock()) {
                                    System.out.println(Thread.currentThread()
                                        .getName()
                                            + " 同时获得锁 1 获得锁 2");
                                    return;
                                }
                            } catch (Exception e) {
                                e.printStackTrace();
                            } finally {
                                if (lock2.isHeldByCurrentThread()) {
                                    lock2.unlock();
                                }
                            }
```

```
//                            break;
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        if (lock1.isHeldByCurrentThread()) {
                            lock1.unlock();
                        }
                    }
                }
            } else {//奇数先获得锁 2 再获得锁 1
                while (true) {
                    try {
                        if (lock2.tryLock()) {
                            System.out.println(Thread.currentThread().getName()
                                + " 获得锁 2，还未获得锁 1");
                            Thread.sleep(new Random().nextInt(10));
                            try {
                                if (lock1.tryLock()) {
                                    System.out.println(Thread.currentThread()
                                        .getName()
                                        + " 同时获得锁 1 获得锁 2");
                                    return;
                                }
                            } catch (Exception e) {
                                e.printStackTrace();
                            } finally {
                                if (lock1.isHeldByCurrentThread()) {
                                    lock1.unlock();
                                }
                            }
//                            break;
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        if (lock2.isHeldByCurrentThread()) {
                            lock2.unlock();
                        }
                    }
                }
            }
        }
```

```
    }

    public static void main(String[] args) {
        IntLock intLock1 = new IntLock(11);
        IntLock intLock2 = new IntLock(22);
        Thread t1 = new Thread(intLock1);
        Thread t2 = new Thread(intLock2);
        t1.setName("t1");
        t2.setName("t2");
        t1.start();
        t2.start();
    }
}
```

5.2.3.4 newCondition()

关键字 synchronized 与 wait() / notify() 两个方法一起使用可以实现等待/通知模式。

在 Lock 锁的 newCondition() 方法返回一个 Condition 对象, Condition 类也可以实现等待/通知模式。

使用 notify() 通知时，JVM 会随机唤醒某个等待的线程，使用 Condition 类可以选择性通知需要被唤醒的线程。 Condition 常用的两个方法： await() 会使当前线程等待，同时会释放锁。当其他线程调用 signal 时，线程会重新获得锁并继续执行。

signal() 用于唤醒一个等待的线程。

注意：在调用 Condition 的 await() / signal() 方法时，也要求线程持有相关的 Lock 锁，调用 await() 后线程会释放这个锁， signal() 调用后会从当前 Condition 对象的等待队列中，唤醒一个线程，唤醒的线程尝试获得锁，一旦获得锁成功就继续执行。

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: Condition 等待与通知
 * @author: 戛剑生
 * @creat: 2021-03-08 16:30:42
 **/
public class Test01 {
    //定义锁
    static Lock lock  = new ReentrantLock();
    //获得 Condition 对象
    static Condition condition = lock.newCondition();
    //定义线程子类
    static class SubThread extends Thread{
        @Override
```

```
        public void run() {
            try {
                lock.lock();//调用 await 方法前必须先获得锁。
                System.out.println(Thread.currentThread().getName()
                    + " 获得锁");
                condition.await();
                System.out.println(Thread.currentThread().getName()
                    + " 线程等待");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
                System.out.println(Thread.currentThread().getName()
                    +" 释放锁");
            }
        }
    }
    public static void main(String[] args) {
        SubThread t1 = new SubThread();
        t1.setName("t1");
        t1.start();//子线程启动后会转入等待状态

        try {
            Thread.sleep(3000);//主线程睡眠 3 秒
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        //唤醒子线程的等待
        try {
            lock.lock();
            condition.signal();//调用 signal 的线程也必须获得对应的锁
                               //此时是 main 线程调用的
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
```

5.2.3.5 多个 Condition 实现通知部分线程

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
```

```
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 多个 Condition 实现通知部分线程
 * @author: 戛剑生
 * @creat: 2021-03-08 16:49:17
 **/
public class Test02 {

    static class Service {
        private Lock lock = new ReentrantLock();
        private Condition conditionA = lock.newCondition();
        private Condition conditionB = lock.newCondition();

        //定义方法，使用 ConditionA 等待
        public void waitMethodA() {
            try {
                lock.lock();
                System.out.println(Thread.currentThread().getName()
                    + " 开始等待" + System.currentTimeMillis());
                conditionA.await();
                System.out.println(Thread.currentThread().getName()
                    + " 结束等待" + System.currentTimeMillis());
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

        //定义方法，使用 ConditionB 等待
        public void waitMethodB() {
            try {
                lock.lock();
                System.out.println(Thread.currentThread().getName()
                    + " 开始等待" + System.currentTimeMillis());
                conditionB.await();
                System.out.println(Thread.currentThread().getName()
                    + " 结束等待" + System.currentTimeMillis());
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
```

```
            lock.unlock();
        }
    }

    //定义方法唤醒 ConditionA 上的等待
    public void signalA() {
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 唤醒 A 的时间" + System.currentTimeMillis());
            conditionA.signal();
            System.out.println(Thread.currentThread().getName()
                + " 唤醒 A 的时间" + System.currentTimeMillis());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    //定义方法唤醒 ConditionB 上的等待
    public void signalB() {
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 唤醒 B 的时间" + System.currentTimeMillis());
            conditionB.signal();
            System.out.println(Thread.currentThread().getName()
                + " 唤醒 B 的时间" + System.currentTimeMillis());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}

public static void main(String[] args) {
    Service service = new Service();
    new Thread(new Runnable() {
        @Override
        public void run() {
            service.waitMethodA();
```

```
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                service.waitMethodB();
            }
        }).start();
        try {
            Thread.sleep(3000);//main 线程睡眠
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //唤醒 conditionA 的等待
        service.signalA();
        service.signalB();
    }
```

5.2.3.6 实现生产者/消费者设计模式，两个线程交替打印

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 实现生产者/消费者设计模式，两个线程交替打印
 * @author: 戛剑生
 * @creat: 2021-03-08 17:14:29
 **/
public class Test03 {
    static class MyService{
        private Lock lock=new ReentrantLock();//创建锁对象
        private Condition condition = lock.newCondition();//创建 condition 对象
        private boolean flag = true;//定义一个交替打印的标识

        //定义方法：只打印----横线
        public void printOne(){
            try {
                //锁定
                lock.lock();
                while (flag){//flag 为 true 进行等待
                    condition.await();
                }
```

```
            //flag 为 false 进行打印
            System.out.println(Thread.currentThread().getName()
                + "------------------");
            //打印完后把 flag 置为 true;
            flag = true;
            //通知另外的线程打印
            condition.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    //定义方法：只打印 **** 星号
    public void printTwo(){
        try {
            //锁定
            lock.lock();
            while (!flag){//flag 为 false 进行等待
                condition.await();//等待时会释放锁对象
            }
            //flag 为 true 进行打印
            System.out.println(Thread.currentThread().getName()
                + "***************");
            //打印完后把 flag 置为 false;
            flag = false;
            //通知另外的线程打印
            condition.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

}

public static void main(String[] args) {
    MyService myService = new MyService();
    new Thread(new Runnable() {
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
```

```
                    myService.printOne();
                }
            }
        }).start();
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    myService.printTwo();
                }
            }
        }).start();
    }
```

5.2.3.7 生产者/消费者设计模式-多对多

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 使用 Condition 实现生产者/消费者设计模式, 多对多
 *  即有多个线程打印-----, 多个线程打印 *****
 * @author: 戛剑生
 * @creat: 2021-03-08 17:14:29
 **/
public class Test04 {
    static class MyService{
        private Lock lock=new ReentrantLock();//创建锁对象
        private Condition condition = lock.newCondition();//创建 condition 对象
        private boolean flag = true;//定义一个交替打印的标识

        //定义方法: 只打印----横线
        public void printOne(){
            try {
                //锁定
                lock.lock();
                while (flag){//flag 为 true 进行等待
                    condition.await();
                }
                //flag 为 false 进行打印
                System.out.println(Thread.currentThread().getName()
                    + "------------------");
```

```
            //打印完后把 flag 置为 true;
            flag = true;
            //通知另外的线程打印
            condition.signal();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    //定义方法：只打印 **** 星号
    public void printTwo(){
        try {
            //锁定
            lock.lock();
            while (!flag){//flag 为 false 进行等待
                condition.await();//等待时会释放锁对象
            }
            //flag 为 true 进行打印
            System.out.println(Thread.currentThread().getName()
                + "***************");
            //打印完后把 flag 置为 false;
            flag = false;
            //通知另外的线程打印
            condition.signalAll();//唤醒所有的线程
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

}

public static void main(String[] args) {
    MyService myService = new MyService();
    for (int i = 0;i < 10;i++) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    myService.printOne();
                }
```

```
                }
            }).start();
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 100; i++) {
                        myService.printTwo();
                    }
                }
            }).start();
        }
    }
```

5.2.4 公平锁与非公平锁

大多数情况下，锁的申请都是非公平的，如果线程 1 与线程 2 都在请求 A 锁，当锁 A 可用时，系统只是会从阻塞队列中随机的选择一个线程，不能保证公平性。

公平锁会按照时间的先后顺序保证先到先得。公平锁的这一特点不会使线程饥饿。

`synchronized` 内部锁就是非公平的，`ReentrantLock()` 重入锁提供了一个构造方法：`ReentrantLock(boolean fair)`，在创建锁对象时，实参传递 `true` 可以把该锁设置为公平锁。公平锁看起来很公平，但是要实现公平锁必须要求系统维护一个有序队列。公平锁的实现成本相对较高，性能低。因此默认情况下锁是非公平的。不是特别的需求，一般不使用公平锁。

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: 公平锁与非公平锁
 * @author: 戛剑生
 * @creat: 2021-03-08 21:59:34
 **/
public class Test01 {
//    static Lock lock = new ReentrantLock();//定义一个锁，默认是非公平锁。
    static Lock lock = new ReentrantLock(true);//定义一个锁，公平锁。

    public static void main(String[] args) {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        lock.lock();
                        System.out.println(Thread.currentThread().getName()
```

```
                        + " 获得了锁对象");
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        }
    };

    for (int i = 0; i < 5; i++) {
        new Thread(runnable).start();
    }
    /**
     * 运行程序发现:
     *  1. 如果是非公平锁
     *      系统会倾向于让一个线程再次获得已经持有的锁
     *      这种策略是高效的但是非公平。
     *  2. 如果是公平锁，多个线程不会发生同一个线程连续获得锁的可能。
     *      保证了锁的公平，按照先到先得的顺序分配锁。
     */
}
```

5.2.5 几个常用的方法

5.2.5.1 `int getHoldCount()`

可以返回当前线程调用 `lock()` 方法的次数。

```
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: int getHoldCount() 可以返回当前线程调用 lock() 方法的次数
 * @author: 戛剑生
 * @creat: 2021-03-08 22:13:12
 **/
public class Test02 {
    static ReentrantLock lock = new ReentrantLock();//定义一个锁对象
    public static void m1(){
        try {
            lock.lock();
            //打印线程调用 lock() 的次数
            System.out.println(Thread.currentThread().getName()
                + "--hold count : " + lock.getHoldCount());
```

```
            //调用 m2 方法
            //ReentrantLock 是可重入锁
            //在 m2 方法中可再次获得该锁对象
            m2();
        }finally {
            lock.unlock();
        }
    }

    public static void m2(){
        try {
            lock.lock();
            //打印线程调用 lock() 的次数
            System.out.println(Thread.currentThread().getName()
                + "--hold count : " + lock.getHoldCount());
        }finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        //main 线程调用 m1
        m1();
        /**
         * 程序运行结果:
         * main--hold count : 1
         * main--hold count : 2
         */
    }
```

5.2.5.2 int getQueueLength()

返回正等待获得锁的线程预估数（可能会有误差）。

```
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description:int getQueueLength() 的使用
 * @author: 戛剑生
 * @creat: 2021-03-08 22:23:49
 **/
public class Test03 {
    static  private ReentrantLock lock = new ReentrantLock();
```

```
    public static void sm(){
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 获得锁，执行方法，估计等待获得锁的线程数"
                    + lock.getQueueLength());
            Thread.sleep(1000);//睡眠 1 秒，模拟执行时间
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                sm();
            }
        };

        for (int i = 0; i < 10; i++) {
            new Thread(r).start();
        }
    }
}
```

5.2.5.3 `int getWaitQueueLength(Condition condition)`

返回与 `Condition` 条件相关的等待的线程预估数。

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description:int getWaitQueueLength(Condition condition)
 * 返回与 Condition 条件相关的等待的线程预估数。
 * @author: 戛剑生
 * @creat: 2021-03-08 22:33:59
 **/
public class Test04 {
    static class Service {
        private ReentrantLock lock = new ReentrantLock();
```

```
    Condition condition = lock.newCondition();

    public void waitMethod() {
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 进入等待前，现在该 condition
                    条件上等待的线程预估数："
                     + lock.getWaitQueueLength(condition));
            condition.await();

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void notifyMethod() {
        try {
            lock.lock();
            condition.signalAll();//唤醒所有等待的线程
            System.out.println(" 唤醒所有等待后，condition
                条件上等待的线程预估数："
                    + lock.getWaitQueueLength(condition));
        } finally {
            lock.unlock();
        }
    }
}

public static void main(String[] args) {
    Service service = new Service();
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            service.waitMethod();
        }
    };
    for (int i = 0; i < 10; i++) {
        new Thread(runnable).start();
    }
    try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //1 秒后唤醒所有的等待
        service.notifyMethod();
    }
```

5.2.5.4 boolean hasQueuedThread(Thread thread)

查询参数指定的线程是否在等待获得锁。

5.2.5.5 boolean hasQueuedThreads()

查询是否还有线程在等待获得该锁。

```
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description boolean hasQueuedThread(Thread thread)
 * 查询指定的线程是否在等待获得锁
 * boolean hasQueuedThreads(): 查询是否有线程在等待获得锁
 * @author: 戛剑生
 * @creat: 2021-03-09 08:30:16
 **/
public class Test05 {
    static ReentrantLock lock = new ReentrantLock();

    public static void waitMethod() {
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 获得了锁。");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName()
                + " 释放了锁对象！");
            lock.unlock();
        }
    }

    public static void main(String[] args) {
```

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        waitMethod();
    }
};
Thread[] threads = new Thread[10];//定义线程数组
//给线程数组的元素赋值，每个线程都调用 waitMethod 方法并启动线程
for (int i = 0; i < threads.length; i++) {
    threads[i] = new Thread(r);
    threads[i].setName("t" + (i + 1));
    threads[i].start();
}
try {
    Thread.sleep(3000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
//判断数组中的每个线程是否在等待获得锁
System.out.println(threads[0].getName()
    +lock.hasQueuedThread(threads[0]));
System.out.println(threads[1].getName()
    +lock.hasQueuedThread(threads[1]));
System.out.println(threads[2].getName()
    +lock.hasQueuedThread(threads[2]));
System.out.println(threads[3].getName()
    +lock.hasQueuedThread(threads[3]));
System.out.println(threads[4].getName()
    +lock.hasQueuedThread(threads[4]));
System.out.println(threads[5].getName()
    +lock.hasQueuedThread(threads[5]));
System.out.println(threads[6].getName()
    +lock.hasQueuedThread(threads[6]));
System.out.println(threads[7].getName()
    +lock.hasQueuedThread(threads[7]));
System.out.println(threads[8].getName()
    +lock.hasQueuedThread(threads[8]));
System.out.println(threads[9].getName()
    +lock.hasQueuedThread(threads[9]));
try {
    Thread.sleep(2000);
} catch (InterruptedException e) {
    e.printStackTrace();
```

```
        }
        //再次判断是否还有线程在等待获得该锁
        System.out.println(lock.hasQueuedThreads());
    }
}
```

5.2.5.6 boolean hasWaiters(Condition condition)

查询是否有线程在等待指定的 Condition 条件。

```
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: boolean hasWaiters(Condition condition)
 * 查询是否有线程在等待指定的 condition 条件
 * @author: 戛剑生
 * @creat: 2021-03-09 08:57:18
 **/
public class Test06 {
    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    static void sm() {
        try {
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 获得锁定！");
            System.out.println(" 是否有线程在等待当前 condition 条件？"
                + lock.hasWaiters(condition) + " 等待数量"
                    + lock.getWaitQueueLength(condition));
            condition.await(new Random().nextInt(3),
                TimeUnit.SECONDS);//超时后自动唤醒
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName()
                + " 释放锁对象");
            lock.unlock();
        }
    }
```

```
    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                sm();
            }
        };
        for (int i = 0; i < 10; i++) {
            new Thread(r).start();
        }
    }
```

5.2.5.7 boolean isFair()

判断是否为公平锁。

5.2.5.8 boolean isHeldByCurrentThread()

判断当前线程是否持有锁。

```
import java.util.Random;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: boolean isHeldByCurrentThread()
 * 判断锁是否被当前线程持有
 * boolean isFair()
 * 判断是否为公平锁
 * @author: 戛剑生
 * @creat: 2021-03-09 09:14:56
 **/
public class Test07 {
    static class Service {
        private ReentrantLock lock = new ReentrantLock();

        //通过构造方法接收 boolean，判断当前锁是否公平
        public Service(boolean isFair) {
            this.lock = new ReentrantLock(isFair);
        }

        public void serviceMethod() {
            try {
                System.out.println(" 是否为公平锁？ " +
```

```
                    (lock.isFair() ? " 是" : " 不是")
                        + "--" + Thread.currentThread().getName()
                        + " 调用 lock 前是否持有锁? "
                        + (lock.isHeldByCurrentThread() ? " 持有" : " 未持有"));
                lock.lock();
                System.out.println(Thread.currentThread().getName()
                    + " 调用 lock 后是否持有锁? "
                        + (lock.isHeldByCurrentThread() ? " 持有" : " 未持有"));
            } finally {
                try {
                    lock.unlock();
                } catch (Exception e) {
                    System.out.println(e.getMessage());
                }
            }
        }
    }

    public static void main(String[] args) {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                int num = new Random().nextInt(10);
                System.out.println(num);
                Service service = new Service(num % 2 == 0);
                service.serviceMethod();
            }
        };
        for (int i = 0; i < 3; i++) {
            new Thread(r, "t" + (i + 1)).start();
        }
    }
```

5.2.5.9 boolean isLocked()

查询当前锁是否被线程持有。

```
import java.util.concurrent.locks.ReentrantLock;

/**
 * @program: DataStructures
 * @description: isLocked() 判断锁是否被当前线程持有
 * @author: 戛剑生
 * @creat: 2021-03-09 09:37:42
```

```
 **/
public class Test08 {
    static ReentrantLock lock = new ReentrantLock();

    static void sm() {
        try {
            System.out.println("lock 是否被" + Thread.currentThread().getName()
                + " 持有? " + (lock.isLocked() ? " 是" : " 否"));
            lock.lock();
            System.out.println(Thread.currentThread().getName()
                + " 获得锁! ");
            System.out.println(" 线程获得锁后 lock 是否被"
                + Thread.currentThread().getName() + " 持有? "
                    + (lock.isLocked() ? " 是" : " 否"));
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            try {
                lock.unlock();
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }

    public static void main(String[] args) {
        System.out.println("11-----" + lock.isLocked());
        new Thread(new Runnable(){
            @Override
            public void run() {
                sm();
            }
        }).start();
        try {
            Thread.sleep(3000);//确保子线程执行结束
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("22-----" + lock.isLocked());
    }
```

5.3 ReentrantReadWriteLock 读写锁

synchronized 内部锁与 ReentrantLock 锁都是独占锁（排它锁），同一时间只允许一个线程执行同步代码块中的代码，可以保证线程的安全性，但是执行效率低。

ReentrantReadWriteLock 读写锁可以理解为一种改进的排它锁，也可以称作共享/排它锁。允许多个线程同时读取共享数据，但一次只允许一个线程对共享数据进行更新。

读写锁通过读锁与写锁来完成操作。线程在读取共享数据前必须先持有读锁，该读锁可以同时被多个线程持有，即读锁是共享的。

线程在修改共享数据前必须先持有写锁，写锁是排他的，当一个线程持有写锁时，其他线程无法获得相应的锁。

读锁只是在读线程之间共享，任何一个线程持有读锁时，其他线程无法获得写锁，以保证线程在读取数据期间没有其他线程对数据进行更新。使得读线程能够读到数据的最新值，保证在读数据期间共享变量不被修改。提高了读取数据的并发性。

读写锁允许读读共享，读写互斥，写写互斥。

5.3.1 读锁

获得条件：写锁没有被任意线程持有。

排他性：对读线程是共享的，对写线程是排他的。

作用：允许多个线程可以同时读取共享数据，保证在读共享数据时，没有其他线程对共享数据进行修改。

5.3.2 写锁

获得条件：该写锁未被其他线程持有，并且相应的读锁也未被其他线程持有。

排他性：对读线程和写线程都是排他的。

作用：保证写线程以独占的方式修改共享数据。

5.3.3 ReadWriteLock 接口

该接口的方法：readLock() 返回读锁，writeLock() 返回写锁。注意：这两个方法返回的是同一个锁对象的两个不同的角色，不是分别获得两个不同的锁。

该接口的实现类：ReentrantReadWriteLock

5.3.4 读写锁的基本使用方法

```
//定义读写锁
ReadWriteLock rwLock = new ReentrantReadWriteLock();
//获得读锁
Lock rLock  = rwLock.readLock();
//获得写锁
Lock wLock  = rwLock.writeLock();
//读数据的方法
rLock.lock();//先申请读锁
try{
    //读取共享数据
```

```
}finally{
    rLock.unlock();//释放读锁
}

//写数据的方法
wLock.lock();//先申请写锁
try{
    //写入共享数据
}finally{
    wLock.unlock();//释放写锁
}
```

5.3.4.1 读读共享

ReadWriteLock 读写锁可以实现多个线程同时读取共享数据，即读读是共享的，可以提高程序读取数据的效率。

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * @program: DataStructures
 * @description: 演示 ReadWriteLock 演示读读共享即允许多多个线程同时获得读锁
 * @author: 戛剑生
 * @creat: 2021-03-09 10:19:03
 **/
public class Test01 {
    static class Service{
        //先定义读写锁
        ReadWriteLock rwLock = new ReentrantReadWriteLock();
        //定义方法读取数据
        public void readMethod(){
            try {
                rwLock.readLock().lock();//申请读锁。
                System.out.println(Thread.currentThread().getName()
                    + " 获得读锁，开始读取数据的时间"
                        + System.currentTimeMillis());
                TimeUnit.SECONDS.sleep(3);//模拟读取数据的用时
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                rwLock.readLock().unlock();//释放读锁
            }
```

```
        }
    }

    public static void main(String[] args) {
        Service service = new Service() ;
        for (int i = 0; i < 5; i++) {
            new Thread(new Runnable(){
                @Override
                public void run() {
                    service.readMethod();//在线程中调用 readMethod() 读取数据
                }
            }).start();
        }
        //程序运行后，多个线程几乎可以同时获得读锁执行 lock() 后面的代码。
    }
```

5.3.4.2 写写互斥

ReadWriteLock 读写锁中的写锁只允许有一个线程执行 lock() 后面的代码。

```
/**
 * @program: DataStructures
 * @description: 演示 ReadWriteLock 演示写写互斥即只允许一个线程同时获得写锁
 * @author: 戛剑生
 * @creat: 2021-03-09 10:30:01
 **/
public class Test02 {
    static class Service{
        //先定义读写锁
        ReadWriteLock rwLock = new ReentrantReadWriteLock();
        //定义方法修改数据
        public void write(){
            try {
                rwLock.writeLock().lock();//申请获得写锁
                System.out.println(Thread.currentThread().getName()
                    + " 获得写锁，开始修改数据的时间"
                        + System.currentTimeMillis());
                System.out.println(" 修改数据中");
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                System.out.println(Thread.currentThread().getName()
                    + " 写入数据完毕的时间" + System.currentTimeMillis());
```

```
                rwLock.writeLock().unlock();
            }
        }
    }

    public static void main(String[] args) {
        Service service = new Service();
        for (int i = 0; i < 5; i++) {
            new Thread(new Runnable(){
                @Override
                public void run() {
                    service.write();
                }
            }).start();
        }
        //从执行时间看，同一时间只有一个线程获得写锁，实现了写写互斥
    }
```

5.3.4.3 读写互斥

写锁时独占锁，是排它锁，读线程与写线程也是互斥的。

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * @program: DataStructures
 * @description: 演示 ReadWriteLock 中的读写互斥 一个线程获得读锁时，写线程等待
 * 一个线程获得写锁时，其他线程等待
 * @author: 戛剑生
 * @creat: 2021-03-09 11:19:17
 **/
public class Test03 {
    static class Service {
        //先定义读写锁
        ReadWriteLock rwLock = new ReentrantReadWriteLock();

        //定义方法修改数据
        public void write() {
            try {
                rwLock.writeLock().lock();//申请获得写锁
                System.out.println(Thread.currentThread().getName()
                    + " 获得写锁，开始修改数据的时间"
```

```
                    + System.currentTimeMillis());
            System.out.println(" 修改数据中");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName()
                + " 写入数据完毕的时间" + System.currentTimeMillis());
            rwLock.writeLock().unlock();
        }
    }

    //定义方法读取数据
    public void readMethod() {
        try {
            rwLock.readLock().lock();//申请读锁。
            System.out.println(Thread.currentThread().getName()
                + " 获得读锁, 开始读取数据的时间"
                    + System.currentTimeMillis());
            TimeUnit.SECONDS.sleep(3);//模拟读取数据的用时
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName()
                + " 数据读取结束的时间" + System.currentTimeMillis());
            rwLock.readLock().unlock();//释放读锁
        }
    }
}

public static void main(String[] args) {
    Service service = new Service();
    //定义一个线程读数据
    new Thread(new Runnable() {
        @Override
        public void run() {
            service.readMethod();
        }
    }).start();
    //定义一个线程写数据
    new Thread(new Runnable() {
        @Override
        public void run() {
```

```
                service.write();
            }
        }).start();
        //定义一个线程读数据
        new Thread(new Runnable() {
            @Override
            public void run() {
                service.readMethod();
            }
        }).start();
        //定义一个线程读数据
        new Thread(new Runnable() {
            @Override
            public void run() {
                service.readMethod();
            }
        }).start();
        //定义一个线程读数据
        new Thread(new Runnable() {
            @Override
            public void run() {
                service.readMethod();
            }
        }).start();
    }
```

第六章　线程管理

6.1　线程组

类似于在计算机中使用文件夹来管理文件，也可以使用线程组来管理线程。在线程组中来定义一组相似（相关）的线程，在线程组中也可以定义子线程组。

`Thread` 类有几个构造方法允许在创建线程时指定线程组，如果在创建线程时没有指定线程组则该线程属于父线程所在的线程组。

JVM 在创建 `main` 线程时会为它指定一个线程组，因此每个 Java 线程都有一个线程组与之关联。可以调用线程的 `getThreadGroup` 方法返回线程组。

线程组开始是处于安全的考虑，设计用来区分不同的 Applet, 然而 `ThreadGroup` 并未实现这一目标，在新开发的系统中，已经不常用线程组。现在一般 会将一组相关的线程存入一个数组或一个集合中。如果仅仅是用来区分线程时，可以使用线程名称来区分。大多情况下可以忽略线程组。

`main` 线程组的父线程组是 `system`，线程组是自己的父线程组。

6.1.1　设置守护线程组

守护线程是为其他线程提供服务的，当 JVM 中只有守护线程时，守护线程会自动销毁，JVM 会退出。

调用线程组 `setDaemon(true)` 可以把线程组设置为守护线程组，当守护线程组中没有任何活动线程时，线程组会自动销毁。

注意线程组的守护属性，不影响线程组中线程的守护属性，或者说守护线程组中的线程可以使非守护线程。

6.2　捕获线程的执行异常

在线程的 `run` 方法中，如果有受检异常必须进行捕获处理。如果想要获得 `run()` 方法中的运行时异常信息，可以通过回调 `UncaughtExceptionHandler` 接口获得哪个线程运行时出了异常。在 `Thread` 类中有关处理运行异常的方法有：

`getDefaultUncaughtExceptionHandler()`（类方法）可以获得全局的（默认的）`UncaughtExceptionHandler`

`getUncaughtExceptionHandler()`（实例方法）获得当前线程的 `UncaughtExceptionHandler`。

`setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)` 设置全局的 `UncaughtExceptionHandler`。

`setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)` 设置某个线程的的 `UncaughtExceptionHandler`。

当线程运行过程中出现异常，JVM 会调用 `Thread` 类的 `dispatchUncaughtException(Throwable e)` 方法，该方法会调用 `getUncaughtExceptionHandler().uncaughtException(this,e)`。如果想要获得线程中出现异常的信息，就需要设置线程的 `UncaughtExceptionHandler`。

```
/**
 * @program: DataStructures
 * @description: 演示设置线程的 UncaughtExceptionHandler 回调接口
 * @author: 戛剑生
 * @creat: 2021-03-09 15:43:25
```

```
 **/
public class Test01 {
    public static void main(String[] args) {
        //1. 设置线程全局的回调接口
        Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
            @Override
            public void uncaughtException(Thread t, Throwable e) {
                //t 参数接收发生异常的线程，e 就是该线程中的异常
                System.out.println(t.getName() + " 产生了异常" +e.getMessage());
            }
        });

        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getName() + " 开始运行");
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    //线程中的受检异常必须捕获处理
                    e.printStackTrace();
                }
                System.out.println(12/0);//产生算术异常
            }
        });
        t1.setName("t1");
        t1.start();

        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                String txt = null;
                System.out.println(txt.length());//产生空指针异常
            }
        });

        t2.setName("t2");
        t2.start();
        /**
         * 实际开发中，这种异常处理的方式比较常用
         * 尤其是异步执行的方法
         * 如果线程产生了异常，JVM 会调用 dispatchUncaughtException(Throwable e) 方法
         * 在该方法中，调用了 getUncaughtExceptionHandler().uncaughtException(this,e)
```

```
        * 如果当前线程设置了 UncaughtExceptionHandler() 回调接口就调用它自己的
        *       uncaughtException 方法
        *       如果没有设置则调用当前线程所在线程组的 UncaughtExceptionHandler() 方法
        *       如果线程组也没有设置该回调接口，直接把异常信息定向到 System.err 中
        */
    }
```

6.3 注入 Hook 钩子线程

很多软件包括 MySQL、Zookeeper 等都存在 Hook 线程的校验机制，目的是校验进程是否已启动，防止重复启动程序。

Hook 线程也称为钩子线程，当 JVM 退出的时候会执行 Hook 线程。经常在程序启动的时候创建一个.lock 文件，用.lock 文件校验程序是否启动，在程序退出（JVM 退出时）删除该.lock 文件，在 Hook 线程中除了防止重新启动进程外，还可以做资源释放，尽量避免在 Hook 线程中进行复杂的操作。

6.4 线程池

6.4.1 线程池的定义

可以以如下方式开启一个线程，当 `run` 方法结束后，线程对象会被垃圾回收器释放。

```
new Thread(new Runnable() {
    @Override
    public void run() {
        //线程要执行的任务
    }
}).start();
```

在实际的生产环境中，可能需要很多的线程来支撑整个应用，当线程数量非常多时，反而会耗尽 CPU 资源，如果不对线程进行控制和管理，反而会影响程序的性能。

线程开销主要包括：创建与启动线程的开销；线程销毁的开销；线程调度的开销；线程总数受限于 CPU 处理器数量。

线程池就是有效使用线程的一种常用的方式。线程池内部可以预先创建一定数量的工作线程。客户端代码直接将任务作为一个对象提交给线程池，线程池将这些任务缓存在工作队列中，线程池中的工作线程就不断地从队列中取出任务并执行。

6.4.2 JDK 对线程池的支持

JDK 提供了一套 Executor 框架，可以帮助开发人员有效的使用线程池。

6.4.3 线程池的基本使用

`Executors` 创建线程池的几个方法：`newCachedThreadPool()`、`newFixedThreadPool(int nThread)`、`new SingleThread`
返回值都为 `ExecutorService`

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * @program: DataStructures
 * @description 线程池的基本使用
 * @author: 戛剑生
 * @creat: 2021-03-09 20:13:23
 **/
public class Test01 {
    public static void main(String[] args) {
        //创建一个线程池
        //有 5 个线程
        ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);
        //向线程池中提交任务
        for (int i = 0; i < 20; i++) {
            fixedThreadPool.execute(new Runnable() {
                @Override
                public void run() {
                    System.out.println(Thread.currentThread().getId()
                        + " 编号的线程正在执行! 开始时间"
                            +System.currentTimeMillis() );
                    try {
                        Thread.sleep(3000);//模拟执行任务的时长
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }

    }
```

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/**
 * @program: DataStructures
 * @description 线程池的计划任务
 * @author: 戛剑生
 * @creat: 2021-03-09 20:23:24
 **/
```

```
public class Test02 {

    public static void main(String[] args) {
        //创建一个有调度功能的线程池
        //参数 1: Runnable 任务, 参数 2: 延迟时间, 参数 3: 时间单位
        ScheduledExecutorService scheduledExecutorService =
            Executors.newScheduledThreadPool(10);
        //在延迟两秒后执行任务
        scheduledExecutorService.schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId()
                    + "---" + System.currentTimeMillis());
            }
        },2, TimeUnit.SECONDS);
//        //以固定的频率执行任务
//        //在 3 秒后执行任务, 以后每隔 5 秒重新执行一次
//        scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
//            @Override
//            public void run() {
//                System.out.println(Thread.currentThread().getId()
//                      + "---以固定频率开启任务: " +System.currentTimeMillis());
//                try {
//                    TimeUnit.SECONDS.sleep(1);//睡眠模拟任务执行时间
//                    //如果任务执行时长超过了时间间隔
//                    //则任务完成后立即开启下一个任务。
//                } catch (InterruptedException e) {
//                    e.printStackTrace();
//                }
//            }
//        },3,5,TimeUnit.SECONDS);


        //在上次任务结束后在固定的延迟后再执行该任务
        scheduledExecutorService.scheduleWithFixedDelay(new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId()
                    + "---以固定频率开启任务: " +System.currentTimeMillis());
                try {
                    TimeUnit.SECONDS.sleep(1);//睡眠模拟任务执行时间
                    //则任务完成后等待固定时间间隔开启下一个任务。
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
            }
        },3,5,TimeUnit.SECONDS);//不管任务耗时多长
        // 总是在任务结束后的 5 秒内再次开启新的任务

    }
}
```

6.4.4 核心线程池的底层实现

查看 `Executors` 工具类中 `newCachedThreadPool()`， `newSingleThreadExecutor`， `newFciedThreadPool()` 原码：

```
//该线程池在极端情况下，每次提交新的任务都会创建新的线程执行
//适合用来执行大量耗时短且提交频繁的任务
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
        60L, TimeUnit.SECONDS,
            new SynchronousQueue<Runnable>());
}

//核心线程数等于最大线程数
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)
    return new ThreadPoolExecutor(nThreads, nThreads,
        0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(),
                                threadFactory);
}

//
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

`Excutors` 工具类中返回线程池的方法底层都使用了 `ThreadPoolExecutor` 线程池。这些方法都是 `ThreadPoolExecut` 程池的封装。

`ThreadPoolExecutor` 线程池的构造方法：

```
public ThreadPoolExecutor(int corePoolSize,//指定线程池中核心线程的数量
                          int maximumPoolSize,//指定线程池中最大线程数量
```

```
                long keepAliveTime,//当线程池中线程数量超过 corePoolSize 时
                  //多余的空闲线程存活时长，即空闲线程在多长时间内销毁
                TimeUnit unit,//是上一个参数的单位
                BlockingQueue<Runnable> workQueue,//任务队列
                ThreadFactory threadFactory,//线程工厂用于创建线程的
                RejectedExecutionHandler handler//拒绝策略
                        //当任务太多来不及处理时，如何拒绝

                ) {
```

说明：`workQueue`工作队列是指提交未执行的任务队列。它是 BlockingQueue 接口的对象，仅用于存取`Runnable`任务，根据队列功能分类，在`ThreadPoolExecutor`构造方法中可以使用以下几种阻塞队列：

- 直接提交队列：由`synchronousQueue`对象提供，该队列没有容量，提交给线程池的任务不会被真实的保存，总是将新的任务提交给线程执行，如果没有空闲线程，则尝试创建新的线程，如果线程数量已经达到`maximumPoolSize`规定的最大值则执行拒绝策略。
- 有界任务队列：由`ArrayBlockingQueue`对象实现，在创建`ArrayBlockingQueue`对象时，可以指定一个容量。当有任务需要执行时，如果线程池中线程数小于`corePoolSize`核心线程数，则创建新的线程。如果大于该数则加入等待队列。如果队列已满，则无法加入。在线程数小于`maximumPoolSize`指定的最大线程数的前提下创建新的线程来执行，如果线程数大于`maximumPoolSize`规定的最大值则执行拒绝策略。
- 无界任务队列：由`LinkedBlokingQueue`对象实现，与有界队列相比，除非系统资源耗尽否则无界队列不存在任务入队失败的情况。当有新的任务时，在系统线程数小于`corePoolSize`核心线程数，则创建新的线程来执行任务，当线程池中的线程数量大于`corePoolSize`核心线程数，则把任务加入阻塞队列。
- 优先任务队列：通过`PriorityBlockingQueue`实现的，是带有任务优先级的队列，是一个特殊的无界队列。不管是`ArrayBlockingQueue`还是`LinkedBlokingQueue`队列都是先进先出算法处理任务。在`PriorityBlockingQueue`队列中可以根据任务优先级顺序先后执行。

6.4.5 拒绝策略

`ThreadPoolExecutor`的最后一个参数指定了拒绝策略。当提交给线程池的任务数量超过实际的承载能力时，如何处理？

即线程池中的线程已经用完了，等待队列也满了，无法为新提交的任务服务，可以通过拒绝策略来处理这个问题。

JDK 提供了 4 种拒绝策略：

- `AbortPolicy`策略：会抛出一个异常
- `CallerRunsPolicy`策略，只要线程池没有关闭，会在调用者线程中运行当前被丢弃的任务。
- `DiscardOldestPolicy`策略：会将任务队列中最老的任务丢弃，最老的任务指的是即将要执行的任务，尝试再次提交新任务。
- `DiscardPolicy`策略：直接丢弃这个无法处理的任务。

`Executors` 工具类提供的静态方法返回的线程池默认的拒绝策略是 `AbortPolicy` 抛出异常。如果内置的拒绝策略无法满足实际需求，可以扩展 `RejectedExecutionHandler` 接口。

拒绝策略示例：

```
import java.util.Random;
import java.util.concurrent.*;

/**
 * @program: DataStructures
 * @description: 自定义拒绝策略
 * @author: 戛剑生
 * @creat: 2021-03-10 08:46:04
 **/
public class Test03 {

    public static void main(String[] args) {
        //定义一个任务
        Runnable r = new Runnable() {
            @Override
            public void run() {
                int num = new Random().nextInt(4);
                System.out.println(Thread.currentThread().getId()
                    + "--" + System.currentTimeMillis()
                        + " 开始睡眠！" + num + " 秒");
                try {
                    TimeUnit.SECONDS.sleep(num);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        //创建线程池，自定义拒绝策略
        ThreadPoolExecutor threadPoolExecutor =
                new ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS,
                    new LinkedBlockingQueue<>(10),
                        Executors.defaultThreadFactory(),
                            new RejectedExecutionHandler() {
                    @Override
                    public void rejectedExecution(Runnable r,
                        ThreadPoolExecutor executor) {
                        //r 就是请求的任务
                        //executor 就是当前线程池
                        System.out.println(r + "is discarding..");
```

```
                    //不做任何处理，直接丢弃任务
                    //实际开发可根据不同情况进行处理
                }
            });
        //向线程池提交若干任务
        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            threadPoolExecutor.submit(r);//向线程池提交任务
        }
    }
```

6.4.6 `ThreadFactory`

线程池中的线程从哪来？来自于`ThreadFactory`

`ThreadFactory`是一个接口，只有一个用来创建线程的方法：`Thread newThread(Runnable r);`，当线程池需要创建线程时就会调用该方法。

```
import java.util.Random;
import java.util.concurrent.*;

/**
 * @program: DataStructures
 * @description 自定义线程工厂
 * @author: 戛剑生
 * @creat: 2021-03-10 10:01:20
 **/
public class Test04 {
    public static void main(String[] args) {
        //定义任务
        Runnable r =   new Runnable() {
            @Override
            public void run() {
                int num = new Random().nextInt(5);
                System.out.println(Thread.currentThread().getId()
                    + "--" + System.currentTimeMillis() + " 开始睡眠" + num+
                        " 秒");
                try {
                    TimeUnit.SECONDS.sleep(num);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        //创建线程池，使用自定义线程工厂，采用默认的拒绝策略
        ExecutorService executorService =
```

```
            new ThreadPoolExecutor(5,5,0,TimeUnit.SECONDS,
                new SynchronousQueue<>(),
                    new ThreadFactory(){
                        @Override
                        public Thread newThread(Runnable r) {
                            //根据参数 r 接收的任务，创建一个线程
                            Thread t = new Thread(r);
                            t.setDaemon(true);//设置为守护线程
                                //当主线程运行结束后，线程池中的线程会自动退出
                            System.out.println(" 创建了线程" + t);
                            return t;
                        }
                    });
        //默认的拒绝策略是抛出异常
        //上述创建的线程池为直接提交队列
        //当给当前线程池提交的任务超过 5 个时，线程池默认抛出异常
        for (int i = 0; i < 6; i++) {
            executorService.submit(r);
        }
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //主线程睡眠超时，主线程结束，线程池中的线程会自动退出
    }
```

6.4.7 监控线程池

`ThreadPoolExecutor` 提供了一组方法用于监控线程池。

`int getActiveCount` 获得线程池中当前活动线程的数量。

`long getCompletedTaskCount` 返回线程池完成任务的数量。

`int getCorePoolsize` 返回线程池中核心线程的数量。

`int getLargestPoolSize` 返回线程池中曾经达到的线程的最大数。

`getMaxinumPoolSize` 返回线程池的最大容量。

`getPoolSize` 返回当前线程池的大小。

`BlockingQueue<Runnable> getQueue()`：返回线程池的阻塞队列。

`long getTaskCount()` 返回线程池收到的任务总数。

```
import java.util.concurrent.*;

/**
 * @program: DataStructures
 * @description: 监控线程池
```

```
 * @author: 戛剑生
 * @creat: 2021-03-10 10:29:21
 **/
public class Test05 {
    public static void main(String[] args) {
        //定义任务
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println(Thread.currentThread().getId()
                    + " 编号的线程开始执行。"+System.currentTimeMillis());

                try {
                    Thread.sleep(10000);//线程睡眠 10 秒模拟任务执行时长
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        //定义线程池
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2,5,0,
            TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(5),
                    Executors.defaultThreadFactory(),
                        new ThreadPoolExecutor.DiscardPolicy());

        //向线程池提交 30 个任务
        for (int i = 0; i < 30; i++) {
            threadPoolExecutor.submit(r);
            System.out.println(" 当前线程池核心线程数: "
                + threadPoolExecutor.getCorePoolSize()
                    +", 最大线程数" + threadPoolExecutor.getMaximumPoolSize()
                        + ", 当前线程池大小: " + threadPoolExecutor.getPoolSize()
                        +", 活动线程数大小: "+threadPoolExecutor.getActiveCount()
                        +", 共收到任务数: "+threadPoolExecutor.getTaskCount()
                        +", 完成任务数: "
                        +threadPoolExecutor.getCompletedTaskCount()
                        +", 等待任务数: " + threadPoolExecutor.getQueue().size());
            try {
                TimeUnit.MILLISECONDS.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }

        System.out.println("-----------------------------------");
        while (threadPoolExecutor.getActiveCount()>=0){
            System.out.println(" 当前线程池核心线程数: "
            + threadPoolExecutor.getCorePoolSize()+", 最大线程数"
            + threadPoolExecutor.getMaximumPoolSize()
            + ", 当前线程池大小: " + threadPoolExecutor.getPoolSize()
            +", 活动线程数大小: "+threadPoolExecutor.getActiveCount()
            +", 共收到任务数: "+threadPoolExecutor.getTaskCount()
            +", 完成任务数: "
            +threadPoolExecutor.getCompletedTaskCount()
            +", 等待任务数: " + threadPoolExecutor.getQueue().size());
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

6.4.8 扩展线程池

有时需要对线程池进行扩展，如监控每个任务开始和结束时间，或者自定义一些其他的增强功能。`ThreadPoolExecutor`线程池提供了两个方法：

- `protected void afterExecute(Runnable r,Throwable t)`
- `protected void beforeExecute(Thread t,Runnable r)`

在线程池执行某个任务前会调用`beforeExecute`方法,在任务结束后(或者任务异常退出)会执行`afterExecute`.

查看`ThreadPoolExecutor`源码，在该类中定义了一个内部类`Worker`，`ThreadPoolExecutor`线程池中的工作线程就是`Worker`类的实例。`Worker`实例在执行时也会调用`beforeExecute`、`afterExecute`方法。

```
import java.util.concurrent.*;

/**
 * @program: DataStructures
 * @description
 * @author: 戛剑生
 * @creat: 2021-03-10 14:15:23
 **/
public class Test06 {
    //定义一个任务类
    private static class MyTask implements Runnable{
        private String taskName;
```

```
    public MyTask(String taskName) {
        this.taskName = taskName;
    }

    public String getTaskName() {
        return taskName;
    }

    @Override
    public void run() {
        System.out.println(taskName
            + " 正在被线程"+Thread.currentThread().getId()+" 执行");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public static void main(String[] args) {
    //定义扩展线程池
    //可以定义线程池类继承 ThreadPoolExecutor
    //在子类中重写 beforeExecute()/afterExecute() 方法

    //也可以直接使用 ThreadPoolExecutor 内部类
    ExecutorService executorService
        = new ThreadPoolExecutor(5,5,0,TimeUnit.SECONDS,
            new LinkedBlockingQueue<>()){
        //在内部类中重写开始方法
        @Override
        protected void beforeExecute(Thread t, Runnable r) {
            System.out.println(t.getId()
                + " 线程准备执行任务"+((MyTask)r).getTaskName());
        }

        @Override
        protected void afterExecute(Runnable r, Throwable t) {
            System.out.println(((MyTask)r).getTaskName()
                + " 任务执行完毕");
        }
```

```
            @Override
            protected void terminated() {
                System.out.println(" 线程池退出！ ");
            }
        };

        for (int i = 0; i < 5; i++) {
            MyTask task = new MyTask(" 任务" + i);
            executorService.execute(task);
        }

        //关闭线程池
        //仅仅是说线程池不再接收新的任务，线程池中已接收的任务正常执行完毕
        executorService.shutdown();
    }
```

6.4.9 优化线程池数量

线程池大小对系统性能是有一定影响的，过大或者过小都会无法发挥最优的系统性能。线程池大小不需要做的非常精确，只需避免极大或极小的情况即可。一般来说，线程池大小需要考虑 CPU 的数量，内存大小等因素。在《Java Concurrency in Practice》书中给出一个估算线程池大小的公式：线程池大小 = CPU 的数量 * 目标 CPU 的使用率 *（1+ 等待时间/计算时间）。

6.4.10 线程池死锁

如果在线程池中执行的任务 A 在执行过程中又向线程池提交了任务 B，任务 B 添加到了线程池的等待队列中，如果任务 A 的结束需要等待任务 B 的执行结果，就有可能出现：线程池中所有的工作线程都处于等待任务处理结果，而这些任务在阻塞队列中等待执行，线程池中没有可以对阻塞队列中的任务进行处理的线程，这种等待会一直持续下去，从而造成死锁。

适合给线程池提交相互独立的任务，而不是彼此依赖的任务，对于彼此依赖的任务，可以考虑分别提交给不同的线程池来执行。

6.4.11 线程池中的异常处理

使用 `ThreadPoolExecutor` 进行 `submit` 提交任务时，有的任务抛出了异常，但线程池并没有进行提示，即线程池把任务中的异常给吃掉了，可以把 `submit` 提交改为 `execute` 执行，也可以对 `ThreadPoolExecutor` 线程池进行扩展，对提交的任务进行包装。

```
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * @program: DataStructures
 * @description: 线程池可能会吃掉程序中的异常
```

```
 * @author: 戛剑生
 * @creat: 2021-03-10 14:59:53
 **/
public class Test07 {
    //定义一个类实现 Runnable 接口用于实现两个数相除
    static class DivideTask implements Runnable {
        private int x;
        private int y;

        public DivideTask(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName()
                + " 计算: " + x + " / " + y + " = " + (x / y));
        }
    }

    public static void main(String[] args) {
        //创建线程池
        ThreadPoolExecutor executorService
            = new ThreadPoolExecutor(0,Integer.MAX_VALUE,0, TimeUnit.SECONDS,
                new SynchronousQueue<>());
        //向线程池添加计算两个数相除的任务
        for (int i = 0; i < 5; i++) {
//            executorService.submit(new DivideTask(10,i));
            executorService.execute(new DivideTask(10,i));
        }
        /**
         * 运行程序只有四条计算结果，实际上线程池提交了 5 个计算任务
         * 当 i==0 时，会产生算术异常，但线程池把该异常给吃掉了
         * 导致我们对该异常一无所知
         * 解决方法:
         * 1. 把 submit() 改为 execute()
         * 2. 对线程池进行扩展，对 submit 方法进行包装
         */
    }
```

2. 对线程池进行扩展，对 submit 方法进行包装

```
/**
```

```
 * @program: DataStructures
 * @description: 自定义线程池类，对 ThreadPoolExecutor 进行扩展
 * @author: 戛剑生
 * @creat: 2021-03-10 14:59:53
 **/
public class Test08 {
    //自定义线程池内，对任务进行包装
    //
    private static class TraceThreadPoolExecutor extends ThreadPoolExecutor{
        public TraceThreadPoolExecutor(int corePoolSize,
            int maximumPoolSize,
                long keepAliveTime, TimeUnit unit,
                    BlockingQueue<Runnable> workQueue) {
            super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
        }
        //定义方法，对执行的任务进行包装
        //接收两个参数：一是接收要执行的任务，二是 Exception 异常
        public Runnable wrap(Runnable r,Exception exception){
            return new Runnable() {
                @Override
                public void run() {
                    try {
                        r.run();
                    }catch (Exception e){
                        exception.printStackTrace();
                        throw e;
                    }
                }
            };
        }
        //重写 submit 方法

        @Override
        public Future<?> submit(Runnable task) {
            return super.submit(wrap(task,new Exception(" 客户跟踪异常")));
        }

        @Override
        public void execute(Runnable command) {
            super.execute(wrap(command,new Exception(" 客户跟踪异常")));
        }
    }
```

```
    //定义一个类实现 Runnable 接口用于实现两个数相除
    static class DivideTask implements Runnable {
        private int x;
        private int y;

        public DivideTask(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName()
                + " 计算: " + x + " / " + y + " = " + (x / y));
        }
    }

    public static void main(String[] args) {
//        //创建线程池
//        ThreadPoolExecutor executorService
    = new ThreadPoolExecutor(0,Integer.MAX_VALUE,0, TimeUnit.SECONDS,
//                new SynchronousQueue<>());
        //使用自定义线程池
        TraceThreadPoolExecutor traceThreadPoolExecutor
            = new TraceThreadPoolExecutor(0,Integer.MAX_VALUE,0,
                TimeUnit.SECONDS,new SynchronousQueue<>());
        //向线程池添加计算两个数相除的任务
        for (int i = 0; i < 5; i++) {
            traceThreadPoolExecutor.submit(new DivideTask(10,i));
//            traceThreadPoolExecutor.execute(new DivideTask(10,i));
        }
        /**
         * 运行程序只有四条计算结果，实际上线程池提交了 5 个计算任务
         * 当 i==0 时，会产生算术异常，但线程池把该异常给吃掉了
         * 导致我们对该异常一无所知
         * 解决方法:
         * 1. 把 submit() 改为 execute()
         * 2. 对线程池进行扩展，对 submit 方法进行包装
         */
    }
```

6.4.12 ForkJoinPool 线程池

分而治之是一个有效处理大数据的办法，著名的 MapReduce 就是采用这种分而治之的思想，简单点说，就说要处理 1000 个数据的能力，但是不具备处理 1000 个数据的能力，可以只处理 10 个数据，可以把这 1000 个数据分阶段处理 100 次，每次处理 10 个，把 100 次处理的结果合成，形成最后这 1000 个数据的处理结果。

把一个大任务调用 fork() 方法分解为若干个小的任务，把小任务的处理结果调用 join() 合并为大任务的处理结果。

系统对 ForkJoinPool 线程池进行了优化，提交的任务数量与线程的数量不一定是一对一的关系。在多数情况下，一个物理线程实际上需要处理多个逻辑任务：线程 A 把自己的任务执行完毕，线程 B 的任务队列中还有若干的任务等待执行，线程 A 会从线程 B 的等待队列中取任务帮助线程 B 完成；线程 A 在帮助 B 执行任务时，总是从线程 B 的等待队列底部开始取任务。

`ForkJoinPool` 线程池中最常用的方法是：

`<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)`，向线程池提交一个 `ForkJoinTask` 任务。`ForkJoinTask` 任务支持 `fork()` 分解与 `join()` 等待的任务。

`ForkJoinTask` 有两个重要的子类：`RecursiveAction`，`RecursiveTask`，区别在于 `RecursiveAction` 没有返回值，`RecursiveTask` 可以带有返回值。

```
import java.util.ArrayList;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.RecursiveTask;

/**
 * @program: DataStructures
 * @description: 演示 ForkJoinPool 线程池的使用
 * 使用线程池模拟数列求和
 * @author: 戛剑生
 * @creat: 2021-03-10 15:56:49
 **/
public class Test09 {
    //计算数列的和需要返回结果
    //可以定义一个任务继承 RecursiveTask
private static class CountTask extends RecursiveTask<Long>{
        private static final int THRESHOLD = 10000;//定义数据规模的阈值
        //即允许计算 10000 个数内的和，
        //超过该阈值数列就要分解

        private long start;//计算数列的起始值
        private long end;//计算数列的结束值

        //约定每次分解为 100 个小任务
        private static final int TASKNUM = 100;

        public CountTask(long start, long end) {
```

```
        this.start = start;
        this.end = end;
    }

    //重写 RecursiveTask 的 compute()
    @Override
    protected Long compute() {
        long sum = 0;
        //判断任务是否需要继续分解
        //当前数列 end 与 start 范围的数据超过阈值
        //就需要继续分解
        if (end - start < THRESHOLD){
            for (long i = start; i <= end; i++) {
                sum += i;
            }
        }else{
            //继续分解
            //约定每次分解为 100 个小任务，计算每个任务的计算量
            long step = (end - start + 1)/TASKNUM;
            //创建一个任务集合
            ArrayList<CountTask> subTaskList = new ArrayList<>();
            long pos = start;//任务的起始位置
            for (int i = 0; i < TASKNUM; i++) {
                long lastOne = pos + step;//每个任务的结束位置
                if (lastOne > end){
                    lastOne = end;
                }
                //创建子任务
                CountTask countTask = new CountTask(pos,lastOne);
                //把任务添加到集合中
                subTaskList.add(countTask);
                //调用 fork 提交子任务
                countTask.fork();
                //调整下个任务的起始位置
                pos += step +1;
            }
            //等待所有的任务结束后，合并计算结果
            for (CountTask task: subTaskList
                 ) {
                sum += task.join();
            }
        }
        return sum;
```

```
        }
    }

    public static void main(String[] args) {
        //创建 ForkJoinPool 线程池
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        //创建一个大的任务
        CountTask task = new CountTask(0l,200031);
        ForkJoinTask<Long> result = forkJoinPool.submit(task);
        try {
            Long res = result.get();//调用任务的 get() 方法返回结果
            System.out.println(" 计算数列结果为: "+res);
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
```

第七章　保障线程安全的设计技术

从面向对象设计的角度出发介绍几种保障线程安全的设计技术。这些技术可以使得我们在不必借助锁的情况下保障线程安全。避免锁可能导致的问题和开销。

7.1　Java 运行时存储空间

Java 运行时（Java Runtime）空间可以分为栈区、堆区与方法区（非堆空间）。

栈空间（Stack Space）为线程的执行准备的一段固定大小的存储空间，每个线程都有独立的线程栈空间，创建线程时就为线程分配栈空间，在线程栈中每调用一个方法就给方法分配一个栈帧。栈帧用于存储方法的局部变量，返回值等私有数据。即局部变量是存储在栈空间中的，基本类型变量也是存储在栈空间中，引用类型变量值也是存储在栈空间中，引用的对象存储在堆中，由于线程栈是相互独立的，一个线程不能访问另一个线程的空间，因此线程对局部变量以及只能通过当前线程的局部变量才能访问的对象进行的操作具有固定的线程安全性。

堆空间（Heap Space）用于存储对象，是在 JVM 启动时分配的一块可以动态扩容的内存空间，创建对象时，在堆空间给对象分配存储空间，实例变量就是存储在堆空间中的，堆空间是多个线程可以共享的空间，因此实例变量可以被多个线程共享。即多个线程同时操作实例变量就可能存在线程安全问题。

方法空间（非堆空间，Non-Heap Space）用于存储常量，类的元数据等，非堆空间也是在 JVM 启动时分配的一块可以动态扩容的内存空间。类的元数据包括静态变量，类有哪些方法以及这些方法的元数据（方法名，参数，返回值等），非堆空间也是多个线程可以共享的空间，访问非堆空间中的静态变量也可能存在线程安全问题。

堆空间和非堆空间是线程可以共享的空间，即实例变量与静态变量是县城可以共享的，可能存在线程安全问题，栈空间是线程私有的存储空间，局部变量存储在栈空间中，局部变量具有固有的线程安全性。

7.2　无状态对象

对象就是数据及对数据操作的封装，对象所包含的数据称为对象的状态。实例变量与静态变量称为状态变量，如果一个类的同一个实例被多个线程同时共享，并不会使这些线程存在共享的状态，那么该类的实例称为无状态对象，反之如果一个类被多个线程共享会使这些线程存在共享状态，那么该类的实例就称为有状态对象。实际无状态对象就是不包含实例变量的对象，也不包含任何静态变量的对象。

线程安全问题的前提是多个线程存在共享的数据，因此实现线程安全的一种办法就是避免在多个线程之间共享数据。使用无状态对象就是这种办法。

7.3　不可变对象

不可变对象指一经创建它的状态就保持不变的对象，不可变对象也具有固有的线程安全性。不可变对象可以被多个线程共享。当不可变对象现实实体的状态发生变化时，系统会创建一个新的不可变对象，就如 `String` 字符串对象。一个不可变对象需要满足以下条件：

- 类本身使用 `final` 修饰，防止通过创建子类来改变它的定义
- 所有的字段都是 `final` 修饰的， `final` 字段在创建对象时，必须显示初始化，不能被修改
- 如果字段引用了其他状态可变的对象（集合、数组）则这些字段必须是 `private` 私有的。

不可变对象主要的应用场景：

- 被建模对象的状态变化不频繁
- 同时对一组数据进行写操作，可以应用不可变对象，即可以保障原子性也可以避免锁的使用。
- 使用不可变对象作为安全可靠的 Map 键，HashMap 键值对的存储位置与键的哈希码有关，如果键的内部状态发生变化会导致键的哈希码不同，可能会影响键值对的存储位置，如果 HashMap 的键是一个不可变对象，则 hashCode() 方法的返回值恒定，存储位置是固定的。

7.4 线程特有对象

可以选择不共享非线程安全对象。对于非线程安全的对象，每个线程都创建一个该对象的实例，各个线程访问各自创建的实例，一个线程不能访问另外一个线程创建的实例。这种各个线程 创建各自的实例，一个实例只能被一个线程访问的对象就称为线程特有对象。线程特有对象既保障了对非线程安全对象的访问的线程安全，又避免了锁的开销，线程特有的对象也具有固有的线程安全性。

`ThreadLocal<T>` 类相当于线程访问其特有对象的代理，即各个线程通过 `ThreadLoad` 对象可以创建并访问各自的线程特有对象，泛型 `T` 指定了线程特有的对象类型，一个线程可以使用不同的 `ThreadLocal` 实例来创建并访问不同的线程特有对象。

`ThreadLocal` 实例为每个访问它的线程都关联了一个该线程特有对象。`ThreadLocal` 实例都有当前线程与特有实例之间的一个关联。

7.5 装饰器模式

装饰器模式可以用来实现线程安全，它的基本思想是为非线程安全的的对象创建一个相应的线程安全的外包装对象，客户端代码不直接访问访问非线程安全对象而是直接访问它的外包装对象。外包装对象与非线程安全的对象具有相同的接口。即外包装对象与非线程安全的对象使用方式相同。而外包装对象内部通常会借助锁，以线程安全的方式调用相应的非线程安全对象的方法。

在 `java.util.Collections` 工具类中提供了一组 `synchronizedXXX(xxx)` 方法,可以把不是线程安全的 `xxx` 集合转换为线程安全的集合，它就是采用了这种装饰器模式。这个方法的方绘值就是指定集合的外包装对象。

使用装饰器模式的一个好处：实现关注点分离。在这种设计中，实现同一组功能的对象的两个版本：非线程安全的对象与线程安全的对象，对于非线程安全的在设计时只关注要实现的功能，对于线程安全的版本只关注线程安全性。