9. 多线程与并发

9.1 进程与线程

- 什么是进程：程序是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念。而进程是程序在处理机上的一次执行过程，它是一个动态的概念。进程是一个具有一定独立功能的程序，一个实体，每一个进程都有它自己的地址空间
- 进程的状态：进程执行时间的间断性，决定了进程可能具有多种状态。事实上，运行中的进程具有以下三种基本状态
  - 就绪状态（Ready）
  - 运行状态（Running）
  - 阻塞状态（Blocked）
- 线程：线程实际上是在进程基础上的进一步划分，一个进程启动之后，里面的若干程序又可以划分成若干个线程。线程是进程中的一个执行路径，共享一个内存空间，线程之间可以自由切换，并发执行，一个进程最少有一个线程（单线程程序）
- 并行：就是两个任务同时运行（多个CPU）
- 并发：是指两个任务同时请求运行，而处理器一次只能接受一个任务，就会把两个任务安排轮流执行，由于CPU时间片运行时间较短，就会感觉两个任务在同时执行

9.2. 线程的基本使用

- 线程实现的两种方式：
  - 继承Thread类

```
1 /**
2  * 实现线程的第一种方式：继承Thread类
```

```
 */
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(i);
        }
    }
}

// 启动方法
MyThread myThread = new MyThread();
myThread.start();// 启动线程
```

- 实现Runnable接口

```
/**
 * 实现线程的第二种方式: 实现Runnable接口
 */
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() +
"-" + i);
        }
    }
}

// 启动方法
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

9.3. 线程休眠

```
public static void sleep(long millis) throws InterruptedException
// 使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行），释放CPU的时间片，具体取决于系统定时器和调度程序的精确和准确性。线程不会丢失任何监视器的所有权
参数:
millis-以毫秒为单位的睡眠时间长度
异常:
IllegalArgumentException-如果millis值为负数
InterruptedException-如果任何线程中断当前线程。当抛出此异常时，当前线程的中断状态将被清除

public static void sleep(long millis, int nanos) throws InterruptedException
// 毫秒，纳秒

static Thread currentThread()
// 返回对当前正在执行的线程对象的引用
```

9.4. join与中断线程

- join

```
public final void join() throws InterruptedException
// 等待这个线程死亡。调用此方法的行为方式与调用完全相同
join(0);
异常InterruptedException-如果任何线程中断当前线程。当抛出此异常时，当前线程的中断状态将被清除

// 示例
/**
 * @author xiao儿
 * @date 2019/9/5 8:24
 * @Description JoinAndInterrupt
 *
 * join方法:
 * 加入线程，让调用的线程先执行指定时间，或执行完毕
 */
public class JoinAndInterrupt {
    public static void main(String[] args) {
        MyRunnable1 myRunnable1 = new MyRunnable1();
        Thread thread = new Thread(myRunnable1);
        thread.start();

        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + "-" + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (i == 20) {
                try {
                    thread.join();// 让thread执行完毕
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

class MyRunnable1 implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + "-" + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- 中断

```java
public void interrupt()
// 中断这个线程。除非当前线程中断自身，这是始终允许的

public static boolean interrupted()
// 测试当前线程是否中断。该方法可以清除线程的中断状态。换句话说，如果这个方法连续调用两次，那么第二次调用将返回false（除非线程再次中断，在第一个调用已经清除其中断状态之后，在第二个调用之前已经检查过）。忽略线程中断，因为线程在中断时不存在将被该方法返回false所反映

// 示例
/**
 * @author xiao儿
 * @date 2019/9/5 8:24
 * @Description JoinAndInterrupt
 * <p>
 * join方法:
 * 加入线程，让调用的线程先执行指定时间，或执行完毕
 */
public class JoinAndInterrupt {
    public static void main(String[] args) {
        MyRunnable1 myRunnable1 = new MyRunnable1();
        Thread thread = new Thread(myRunnable1);
        thread.start();

        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + "-" + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (i == 20) {
                thread.interrupt();// 中断线程，只是做了一个中断标记
            }
        }
    }
}

class MyRunnable1 implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            if (Thread.interrupted()) {// 测试中断状态，此方法会把中断状态清除
                break;
            }
            System.out.println(Thread.currentThread().getName() + "-" + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();// 抛出异常时会清除掉中断状态
                Thread.currentThread().interrupt();
            }
        }
```

```
    }
}
```

- 自定义标记中断线程

```
package day09_多线程与并发;

/**
 * @author xiao儿
 * @date 2019/9/5 8:24
 * @Description JoinAndInterrupt
 * <p>
 * join方法:
 * 加入线程，让调用的线程先执行指定时间，或执行完毕
 *
 * 中断线程:
 * （1）：使用interrupt方法来中断线程，设置一个中断状态（标记）
 * （2）：使用自定义标记中断的方式（推荐使用）
 */
public class JoinAndInterrupt {
    public static void main(String[] args) {
        MyRunnable2 myRunnable2 = new MyRunnable2();
        Thread thread1 = new Thread(myRunnable2);
        thread1.start();

        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + "-" +
i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (i == 20) {
                myRunnable2.flag = false;
            }
        }
    }
}

class MyRunnable2 implements Runnable {
    public boolean flag = true;

    public MyRunnable2() {
        flag = true;
    }

    @Override
    public void run() {
        int i = 0;
        while (flag) {
            System.out.println(Thread.currentThread().getName() + "-" +
(i++));
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
    }
}
```

9.5. 守护线程与yield

- 守护线程

```
public final void setDaemon(boolean on)
// 将此线程标记为daemon线程或用户线程。当运行的唯一线程都是守护进行线程时，Java虚拟机将推出
public final boolean isDaemon()
// 测试这个线程是否是守护线程

// 示例
/**
 * @author xiao儿
 * @date 2019/9/5 9:46
 * @Description DaemonAndYield
 */
public class DaemonAndYield {
    public static void main(String[] args) {
        MyRunnable3 myRunnable3 = new MyRunnable3();
        Thread thread = new Thread(myRunnable3);
        // 线程可以分为守护线程和用户线程，当进程中没有用户线程时，Java虚拟机会退出
        thread.setDaemon(true);// 把线程设置为守护线程
        thread.start();

        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + "-" + i);
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class MyRunnable3 implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName() + "-" + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- yield（不太使用）

```
public static void yield()
// 暂停当前正在执行的线程对象，并执行其他线程
```

9.6. 其他方法与优先级

- 其他方法

```
long getID();// 返回该线程的标识符
String getName();// 返回该线程的名称
void setName(String name);// 改变线程名称，使之与参数name相同
boolean isAlive();// 测试线程是否处于活动状态
```

- 优先级

```
void setPriority(int newPriority);// 更改线程的优先级
static int MAX_PRIORITY;// 线程可以具有的最高优先级
static int MIN_PRIORITY;// 线程可以具有的最低优先级
static int NORM_PRIORITY;// 分配给线程的默认优先级
```

9.7. 线程同步

- 多线程共享数据：在多线程的操作中，多个线程有可能同时处理同一个资源，这就是多线程中的共享数据

```
/**
 * @author xiao儿
 * @date 2019/9/5 10:27
 * @Description SharingData
 *
 * 多线程共享数据时，会发生线程不安全的情况
 * 多线程共享数据必须使用同步
 */
public class SharingData {
    public static void main(String[] args) {
        MyRunnable4 myRunnable4 = new MyRunnable4();
        Thread thread = new Thread(myRunnable4);
        Thread thread1 = new Thread(myRunnable4);
        thread.start();
        thread1.start();
    }
}

class MyRunnable4 implements Runnable {
    private int ticket = 10;// 售票

    @Override
    public void run() {
        for (int i = 0; i < 300; i++) {
            if (ticket > 0) {
                ticket--;
                try {
```

```
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("您购买的票剩余: " + ticket + "张");
            }
        }
    }
}
```

- 线程同步：解决数据共享问题，必须使用同步，所谓同步就是指多个线程在同一时间段内只能有一个线程执行指定代码，其他线程要等待此线程完成之后才可以继续执行
  - 线程同步的方法：
    - 同步代码块：

```
synchronized(要同步的对象) {
    要同步的操作;
}

// 示例
@Override
public void run() {
    for (int i = 0; i < 300; i++) {
        synchronized (object) {
            if (ticket > 0) {
                ticket--;
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

 System.out.println(Thread.currentThread().getName() + "--" + "您购买的票剩余: " + ticket + "张");
            }
        }
    }
}
```

    - 同步方法

```
public synchronized void method() {
    要同步的操作;
}

// 示例
// 同步方法: 同步的对象时当前对象
private synchronized void method() {
    if (ticket > 0) {
        ticket--;
        try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() +
"--" + "您购买的票剩余: " + ticket + "张");
    }
}
```

- Lock(ReentrantLock)

```
// 互斥锁
ReentrantLock reentrantLock = new ReentrantLock();

// Lock来实现同步
private void method2() {
    reentrantLock.lock();// 锁
    try {
        if (ticket > 0) {
            ticket--;
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()
+ "--" + "您购买的票剩余: " + ticket + "张");
        }
    } finally {
        reentrantLock.unlock();// 释放锁
    }
}
```

- 同步准则：当编写synchronized块时，有几个简单的准则可以遵循：
  - 使代码块保持简短。把不随线程变化的预处理和后处理移除synchronized块
  - 不要阻塞。如：InputStream.read()
  - 在持有锁的时候，不要对其他对象调用方法

9.8. 死锁

- 过多的同步有可能出现死锁，死锁的操作一般是在程序运行的时候才会出现
- 多线程中要进行资源的共享，就需要同步，但同步过多，就可能造成死锁
- 示例：

```
/**
 * @author xiao儿
 * @date 2019/9/5 18:14
```

```
 * @Description DeadThreadDemo
 *
 * 线程死锁：在一个同步方法中调用了另一个对象的同步方法，可能会产生死锁
 */
public class DeadThreadDemo {
    public static void main(String[] args) {
        new DeadThread();
    }
}

// 顾客
class Customer {
    public synchronized void say(Waiter w) {
        System.out.println("顾客说：先吃饭再买单！");
        w.doService();
    }

    public synchronized void doServide() {
        System.out.println("同意了，买完单再吃饭！");
    }
}

// 服务员
class Waiter {
    public synchronized void say(Customer c) {
        System.out.println("服务员说：先买单再吃饭！");
        c.doServide();
    }

    public void doService() {
        System.out.println("同意了，吃完饭再买单！");
    }
}

// 死锁线程
class DeadThread implements Runnable {
    Customer c = new Customer();
    Waiter w = new Waiter();

    public DeadThread() {
        new Thread(this).start();
        w.say(c);
    }

    @Override
    public void run() {
        c.say(w);
    }
}
```

9.9. 生产者与消费者应用案例

```
/**
 * @author xiao儿
 * @date 2019/9/6 9:17
 * @Description Food
 */
```

```java
public class Food {
    private String name;
    private String description;
    private boolean flag = true;// true表示可以生产，false表示可以消费

    public Food() {
    }

    public Food(String name, String description) {
        this.name = name;
        this.description = description;
    }

    /**
     * 生产产品
     *
     * @param name
     * @param description
     */
    public synchronized void set(String name, String description) {
        // 不能生产
        if (!flag) {
            try {
                this.wait();// 线程进入等待状态，释放监视器的所有权（对象锁）
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.setName(name);
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.setDescription(description);
        flag = false;
        this.notify();// 唤醒等待的线程（随机的其中一个）
    }

    /**
     * 消费产品
     *
     * @return
     */
    public synchronized void get() {
        // 不能消费
        if (flag) {
            try {
                this.wait();// 线程进入等待状态，释放监视器的所有权（对象锁）
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
64          System.out.println(this.getName() + "->" + this.getDescription());
65          flag = true;
66          this.notify();// 唤醒等待的线程（随机的其中一个）
67      }
68
69      public String getName() {
70          return name;
71      }
72
73      public void setName(String name) {
74          this.name = name;
75      }
76
77      public String getDescription() {
78          return description;
79      }
80
81      public void setDescription(String description) {
82          this.description = description;
83      }
84
85      @Override
86      public String toString() {
87          return "Food{" +
88                  "name='" + name + '\'' +
89                  ", description='" + description + '\'' +
90                  '}';
91      }
92  }
93
94  /**
95   * @author xiao儿
96   * @date 2019/9/6 9:26
97   * @Description Producer
98   */
99  public class Producer implements Runnable {
100     private Food food;
101
102     public Producer(Food food) {
103         this.food = food;
104     }
105
106     @Override
107     public void run() {
108         for (int i = 0; i < 20; i++) {
109             if (i % 2 == 0) {
110                 food.set("锅包肉", "酸甜口味，爽");
111             } else {
112                 food.set("佛跳墙", "大补，滋阴补阳");
113             }
114         }
115     }
116 }
117
118 /**
119  * @author xiao儿
120  * @date 2019/9/6 9:29
121  * @Description Consumer
```

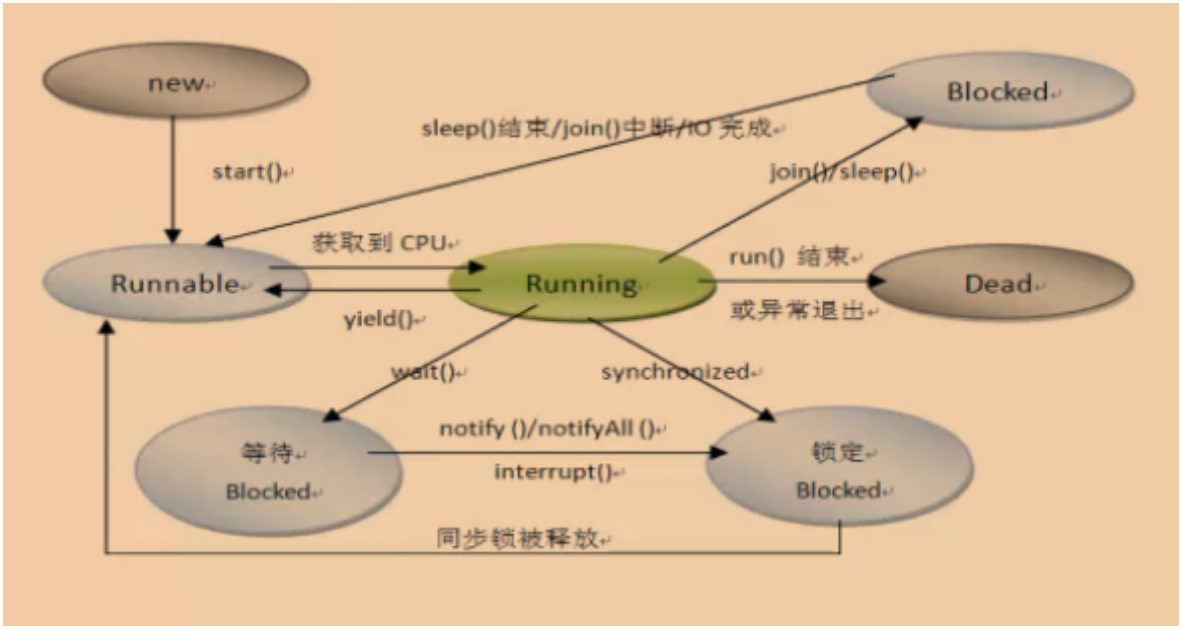
```
 */
public class Consumer implements Runnable {
    private Food food;

    public Consumer(Food food) {
        this.food = food;
    }

    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {
            food.get();
        }
    }
}

/**
 * @author xiao儿
 * @date 2019/9/6 9:17
 * @Description ProducerAndConsumer
 *
 * 两个线程协同工作，先生产，再消费
 *
 * 面试题：
 * sleep和wait的区别？
 * sleep：让线程进入休眠状态，让出CPU的时间片，不释放对象监视器的所有权（对象锁）
 * wait：让线程进入等待状态，让出CPU的时间片，并释放对象监视器的所有权（对象锁），等待其他线程通过notify方法来唤醒
 */
public class ProducerAndConsumer {
    public static void main(String[] args) {
        Food food = new Food();
        Producer producer = new Producer(food);
        Consumer consumer = new Consumer(food);
        Thread threadProducer = new Thread(producer);
        Thread threadConsumer = new Thread(consumer);
        threadProducer.start();
        threadConsumer.start();
    }
}
```

9.10. 线程生命周期



9.11. 线程池

- 定义：线程池是预先创建线程的一种技术。线程池在还没有任务到来之前，创建一定数量的线程，放入空闲队列中，然后对这些资源进行复用。减少频繁的创建和销毁对象
  - JDK1.5版本以上提供了现成的线程池
  - Java里面线程池的顶级接口是Executor，是一个执行线程的工具
  - 线程池接口是ExecutorService
- 常用的方法：

```
java.util.concurrent包：并发编程中很常用的使用工具类
Executor接口：执行已提交的Runnable任务的对象
ExecutorService接口：Executor提供了管理终止的方法，以及可为跟踪一个或多个异步任务执行状况而生成Future的方法
Executors类：此包中所定义的Executor、ExecutorService等的工厂和实用方法
```

- Executors类

```
newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行

newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小

newCacheThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小

newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求
```

- 线程池的大小一旦达到最大值就会保持不变
- 如果某个线程因为执行异而结束，那么线程池会补充一个新线程