day16_多线程基础

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握HashSet的基本使用
2.掌握TreeSet的基本使用
3.掌握Comparable和Comparator的使用
4.掌握Map集合的基本方法
5.掌握Map的遍历(keySet()和entrySet())
6.掌握哈希表的基本原理
7.掌握二叉树的遍历
```

1.2 教学难点:

```
1.数据结构-哈希表实现
对于哈希表的实现也是面试经常问的,但是需要结合数组和链表,相对复杂些,有精力的可以学习一下
2.数据结构-二叉树的原理及实现
二叉树也是相对复杂度数据结构,可以暂时不做深究
3.HashMap的去重和排序
4.TreeMap的去重和排序
```

二 教学目标

```
1.掌握线程的基本概念
2.掌握线程的生命周期
3.掌握常见线程的方法
4.掌握线程同步的实现
5.掌握synchronized的使用
6.了解多线程的原理的理解
7.了解对单例实现线程同步
```

三 教学导读

3.1. 为什么要使用线程?

在程序中完成某一个功能的时候,我们会将他描述成任务,这个任务需要在线程中完成.

3.2. 串行与并发

如果在程序中，有多个任务需要被处理，此时的处理方式可以有**串行**和**并发**:

- 串行（同步）： 所有的任务，按照一定的顺序，依次执行。如果前面的任务没有执行结束，后面的任务等待。
- 并发（异步）： 将多个任务同时执行，在一个时间段内，同时处理多个任务。

> 生活中， 其实有很多串行和并发的案例。 最常见的就是排队买饭。 小明到KFC吃饭， 发现有好几个窗口可以点餐。 选择了其中的一个窗口进行排队。 此时， KFC采用的模式就是串行加并发的模式。 每一个窗口之前， 有很多顾客在排队， 此时他们的任务是串行的， 前面的顾客没有处理完之后， 后面的顾客只能等待。 同时， 多个窗口之间的顾客是可以同时点餐的， 他们是并发的。

> 使用并发任务， 也可以在一定程度上提高效率。 例如: 小明下班回到家， 需要洗衣服、做饭、扫地。 假设， 洗衣服耗时10分钟， 做饭耗时10分钟， 洗衣服耗时10分钟， 那么这些任务如果都给小明一件件的做， 一共要耗时30分钟。 如果小明找两个帮手， 比如雇两个保姆， 他们三个人每人处理一件任务， 则共耗时10分钟。
>
> 在程序中， 有些任务是比较耗时的， 特别是涉及到非常大的文件的处理、或者网络文件的处理。 此时就需要用异步任务来处理， 否则就会阻塞主线程， 导致用户的交互卡顿。 合适的使用并发任务， 可以在一定程度上提高程序的执行效率。

3.3. 并发的原理

一个程序如果需要被执行， 必须的资源是CPU和内存。 在内存上开辟空间， 为程序中的变量进行数据的存储； 同时需要CPU处理程序中的逻辑。 现在处于一个硬件过剩的时代， 但是即便是硬件不发达的时代， 并发任务也是可以实现的。 以单核的CPU为例， 处理任务的核心只有一个， 那就意味着， 如果CPU在处理一个程序中的任务， 其他所有的程序都得暂停。 那么并发是怎么实现的呢?

其实所谓的并发， 并不是真正意义上的多个任务同时执行。 而是CPU快速的在不同的任务之间进行切换。 在某一个时间点处理任务A， 下一个时间点去处理任务B， 每一个任务都没有立即处理结束。 CPU快速的在不同的任务之间进行切换， 只是这个切换的速度非常快， 人类是识别不了的， 因此会给人一种“多个任务在同时执行”的假象。

因此， 所谓的并发， 其实就是CPU快速的在不同的任务之间进行切换的一种假象。

> 思考:

> 既然多个任务并发， 可以在一定程度上提高程序的执行效率， 那么并发数量是不是越高越好呢?
>
> 并不是！ 多个任务的并发， 其实就是CPU在不同的任务之间进行切换。 如果并发的数量过多， 会导致分配到每一个任务上的CPU时间片较短， 也并不见得会提高程序的执行效率。 而且， 每一个任务的载体（线程）也是需要消耗资源的， 过多的线程， 会导致其他资源的浪费。
>
> 例如: 上述案例中， 我们说到了小明雇保姆干活， 那么是不是保姆越多越好呢?
>
> 不一定！ 雇保姆需要花钱， 就类比于开辟线程执行并发的任务需要消耗资源一样。 那么在雇保姆的时候就得想， 你真的需要这么多保姆吗？ 家里有十件事情需要处理， 那么就一定需要雇十个保姆吗？ 没有必要!

3.4. 进程和线程

- 进程， 是对一个程序在运行过程中， 占用的各种资源的描述。
- 线程， 是进程中的一个最小的执行单元。 其实， 在操作系统中， 最小的任务执行单元并不是线程， 而是句柄。 只不过句柄过小， 操作起来非常的麻烦， 因此线程就是我们可控的最小的任务执行单元。

其实， 对于操作系统来说， 一个任务就是一个进程。 例如， 打开了QQ，就是一个QQ的进程； 再打开一个QQ， 就是一个新的QQ的进程； 打开了一个微信， 就是一个微信的进程。 在一个任务中， 有的时候是需要同时处理多件事情的， 例如打开一个QQ音乐， 需要同时播放声音和播放歌词。 那么这些进程中的子任务， 就是一个个的线程。

每一个进程至少要处理一件任务， 因此， 每一个进程中至少要包含一个线程。 如果一个进程中所有的线程都结束了， 那么这个进程也就结束了。

多个线程的同时执行， 是需要这些线程去争抢CPU资源， 而CPU资源的分配是以时间片为单位的。 即某一个线程抢到了0.01秒的CPU时间片， 在这个时间内， CPU处理这个线程的任务。 至于哪一个线程能够抢到CPU时间片， 则由操作系统进行资源调度。

3.5. 进程和线程的异同

相同点: 进程和线程都是为了处理多个任务并发而存在的。

不同点: 进程之间是资源不共享的， 一个进程中不能访问另外一个进程中的数据。 而线程之间是资源共享的， 多个线程可以共享同一个数据。 也正因为线程之间是资源共享的， 所以会出现临界资源的问题。

3.6. 进程和线程的关系

一个进程， 在开辟的时候， 会自动的创建一个线程， 来处理这个进程中的任务。 这个线程被称为是主线程。 在程序运行的过程中， 还可以开辟其他线程， 这些被开辟出来的其他线程， 都是子线程。

也就是说， 一个进程中， 是可以包含多个线程。 一个进程中的某一个线程崩溃了， 只要还有其他线程存在， 就不会影响整个进程的执行。 但是如果一个进程中， 所有的线程都执行结束了， 那么这个进程也就终止了。

3.7. 总结

- 程序:一个可执行的文件
- 进程:一个正在运行的程序.也可以理解成在内存中开辟了一块儿空间
- 线程:负责程序的运行,可以看做一条执行的通道或执行单元,所以我们通常将进程的工作理解成线程的工作
- 进程中可不可以没有线程? 必须有线程,至少有一个.
- 当有一个线程的时候我们称为单线程(唯一的线程就是主线程).

  当有一个以上的线程同时存在的时候我们称为多线程.
- 多线程的作用:为了实现同一时间干多件事情(并发执行).

四 教学内容

4.1. 线程的生命周期(会)

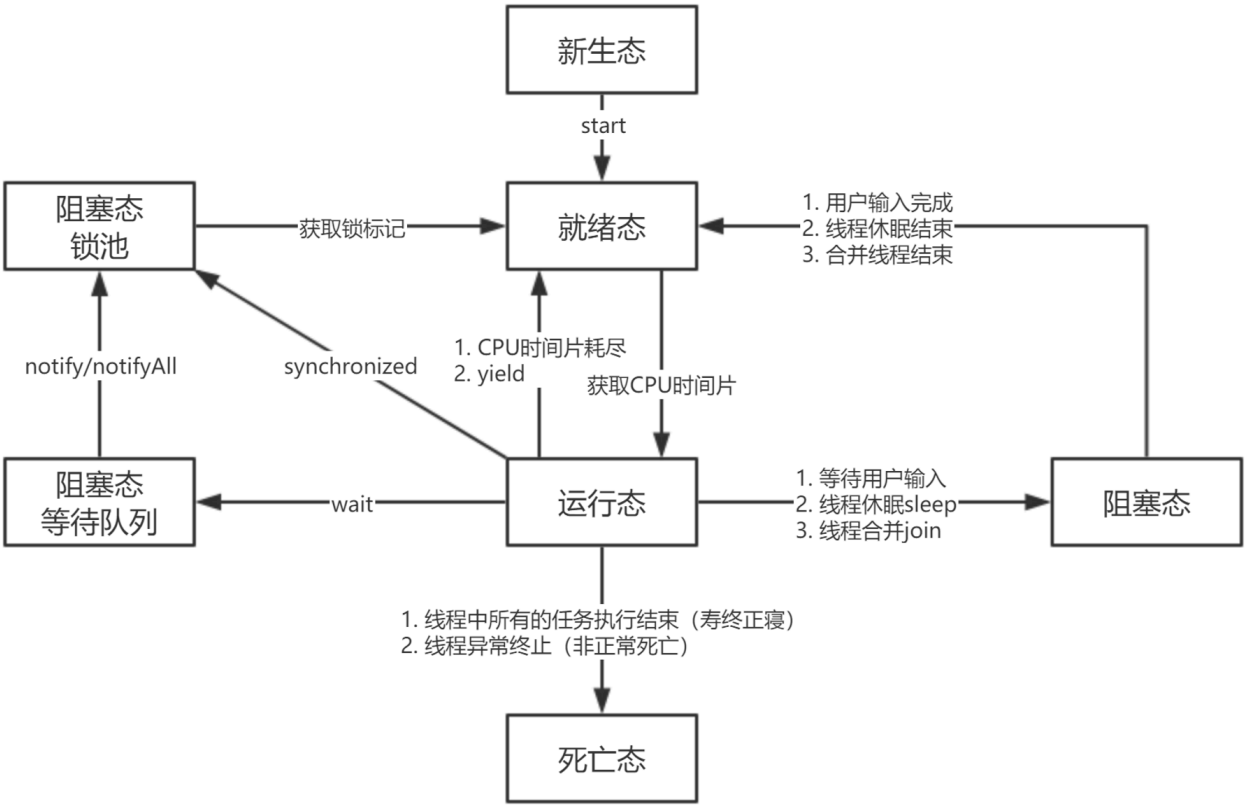
4.1.1. 线程的状态

线程的生命周期， 指的是一个线程对象， 从最开始的创建， 到最后的销毁， 中间所经历的过程。 在这个过程中， 线程对象处于不同的状态。

- New： 新生态， 一个线程对象刚被实例化完成的时候， 就处于这个状态。
- Runnable: 就绪态， 处于这个状态的线程， 可以参与CPU时间片的争抢。
- Run: 运行态， 某一个线程抢到了CPU时间片， 可以执行这个线程中的逻辑
- Block: 阻塞态， 线程由于种种原因， 暂时挂起， 处于阻塞（暂停）状态。 这个状态的线程， 不参与CPU时间片的争抢。

- Dead: 死亡态， 线程即将被销毁。

4.1.2. 线程的生命周期图



4.2. 理解多线程(会)

4.2.1. 对线程并发执行的说明

简单理解(cpu单核):从宏观上看,线程有并发执行,从微观上看,并没有,在线程完成任务时,实际工作的是cpu,我们将cpu工作描述为时间片(单次获取cpu的时间,一般在几十毫秒).cpu只有一个,本质上同一时间只能做一件事,因为cpu单次时间片很短,短到肉眼无法区分,所以当cpu在多个线程之间快速切换时,

宏观上给我们的感觉是多件事同时在执行.

注意:

1.cpu是随机的,线程之间本质上默认是抢cpu的状态,谁抢到了谁就获得了时间片,就工作,所以多个线程的工作也是默认随机的.

2.在使用多线程时,并不是线程数越多越好,本质上大家共同使用一个cpu,完成任务的时间并没有减少.要根据实际情况创建线程,多线程是为了实现同一时间完成多件事情的目的.比如我们用手机打开一个app时,需要滑动界面浏览,同时界面的图片需要下载,对于这两个功能最好同时进行,这时可以使用多线程.

4.2.2.多线程的实例演示

- 代码演示的是主线程和垃圾回收线程在同时工作时的状态
- 什么叫任务区?

我们将线程工作的地方称为任务区.

每一个线程都有一个任务区,任务区通过对应的方法产生作用.

- JVM默认是多线程吗?

至少要有两个线程:

主线程:任务区:main函数

垃圾回收线程:任务区:finalize函数

4.2.3. 示例代码

代码说明:

1.gc()方法:之前讲过,是垃圾回收器

原理:当执行gc时,会触发垃圾回收机制,开启垃圾回收线程,执行finalize方法

2.finalize()方法:垃圾回收线程的任务区

正常情况下,这个函数是由系统调用的,重写只是为了更好的观察多线程的发生

当Test对象被释放的时候,会自动的调用finalize方法

3.线程和任务的关系

任务区结束,线程随着任务的结束而结束,线程随着任务的开始而开始.当线程还在工作的时候,进程不能结束.

对于主线程来说:当main函数结束时,主任务区结束,主线程结束

对于垃圾回收线程:当finalize函数结束,垃圾回收任务结束,垃圾回收线程结束

4.通过运行程序,我们发现字符串main和字符串finalize的打印顺序是随机的(可以多运行几次)

说明:cpu的特性是多个线程之间是抢cpu的关系,cpu有随机性

```
public class Demo4 {
    //就是主线程的任务区
    public static void main(String[] args) {
        new Test();
        /*
         * 手动运行垃圾回收器
         */
        System.gc();
        System.out.println("main");
    }
}
```

```
12
13 class Test{
14     @Override
15     /*
16      * 重写finalize方法
17      */
18     protected void finalize() throws Throwable {
19         System.out.println("finalize");
20     }
21 }
```

4.3. 创建线程(会)

4.3.1 原因分析

默认情况下,主线程和垃圾回收线程都是由系统创建的,但是我们需要完成自己的功能,所以需要创建自己的线程

java将线程面向对象了,形成的类就是Thread,在Thread类内部执行任务的方法叫run()

4.3.2. 线程对象的实例化

在Java中， 使用Thread类来描述一个线程。 实例化一个线程， 其实就是一个Thread对象。

4.3.2.1. 直接使用Thread类创建线程对象

- 线程对象刚刚被实例化的时候， 线程处于新生态,还没有线程的功能。如果需要让这个线程执行他的任务， 需要调用 start() 方法， 使线程进入到就绪态， 争抢CPU时间片。
- 为什么通过调用start()方法开启线程,而不是通过手动调用run()?

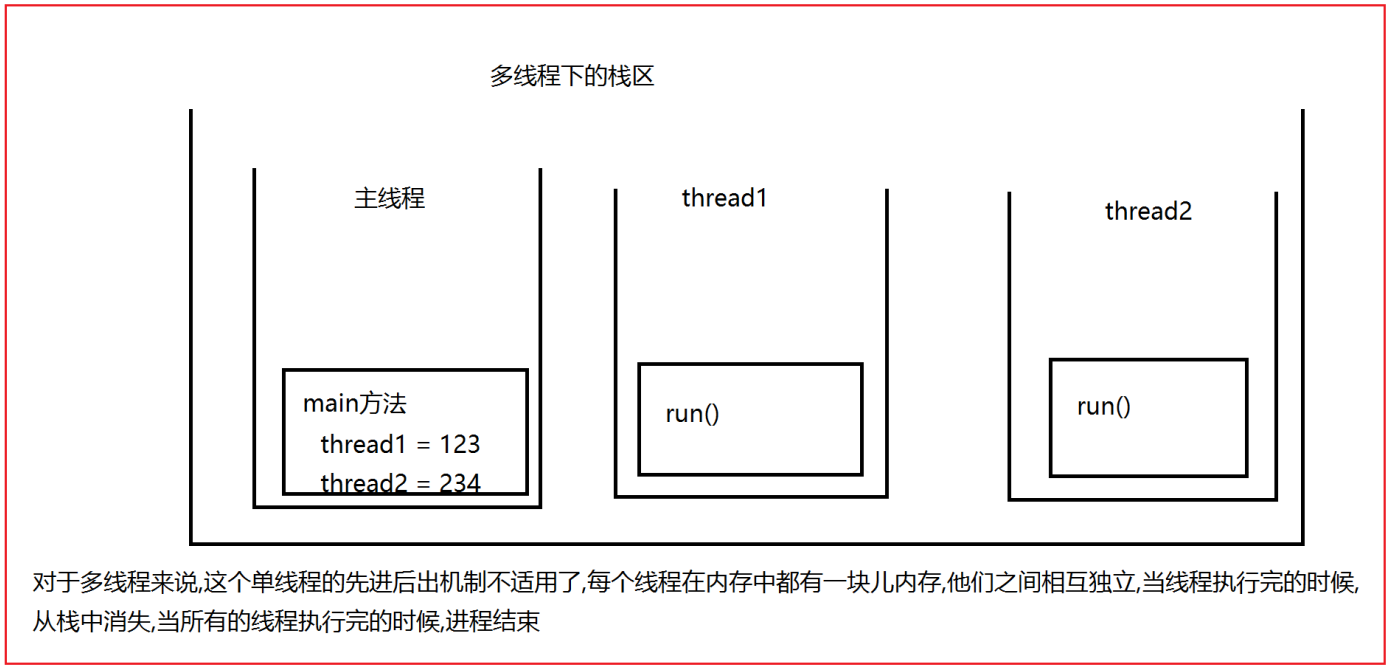
答:因为线程获取cpu是随机的,run是线程的任务区,代表功能.如果手动执行run,此时线程可能没有拿到cpu,无法工作,操作失败.通过start,让线程处于就绪状态,随时拥有抢cpu的能力,当抢到cpu后,再自动执行run,实现并发的任务.

- 为什么要使用Thread类的子类对象?

答:我们实现的实际功能,Thread类作为系统类不能提前知晓,所以无法将功能代码放入Thread的run方法里.如果想实现自己的功能,可以写Thread类的子类,重写run方法,这也是为什么Thread的run方法是一个空方法.

```
public class Demo5 {
    public static void main(String[] args) {//为了方便研究,先暂时不考虑垃圾回收线程.
        //创建自己的线程对象--还没有线程的功能
        Thread t1 = new Thread();
        Thread t2 = new Thread();
        //当执行start方法后,他才有了线程的功能--开启线程
        t1.start();
        t2.start();//有三个线程(主线程+两个子线程)
```

4.3.2.2. 多线程的内存展示



对于多线程来说,这个单线程的先进后出机制不适用了,每个线程在内存中都有一块儿内存,他们之间相互独立,当线程执行完的时候,从栈中消失,当所有的线程执行完的时候,进程结束

4.3.2.3. 继承Thread类

- 继承自Thread类， 做一个Thread的子类。 在子类中， 重写父类中的run方法， 在这个重写的方法中， 指定这个线程需要处理的任务。
- Thread.currentThread() : 可以用在任意的位置， 获取当前的线程。

  如果是Thread的子类， 可以在子类中， 使用this获取到当前的线程。
- 当我们手动调用run的时候,他失去了任务区的功能,变成了一个普通的方法. 当run作为一个普通方法时,内部对应的线程跟调用他的位置保持一致.
- 结果分析:

主线程和两个子线程之间是随机打印的,他们是抢cpu的关系.

- 通过创建Thread子类的方式实现功能,线程与任务绑定在了一起,操作不方法

我们可以将任务从线程中分离出来,哪个线程需要工作,就将任务交给谁,操作方便,灵活-使用Runnable接口

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Demo5 {
    public static void main(String[] args) {//为了方便研究,先暂时不考虑垃圾回收线程.
        MyThread t1 = new MyThread("bing");
        MyThread t2 = new MyThread("ying");
        t1.start();
        t2.start();

```

```
13          //手动调用run()方法
14          t1.run();
15
16          for (int i = 0; i <10; i++) {
17
System.out.println(Thread.currentThread().getName()+"
i:"+i);
18          }
19      }
20  }
21
22  class  MyThread extends  Thread{
23      String name1;
24
25      public MyThread(String name1) {
26          this.name1 = name1;
27      }
28
29      //任务区
30      //重写run方法,实现我们的功能.run就是我们的任务区
31      /*
32       * Thread.currentThread():获取的是当前的线程
33       * Thread.currentThread().getName():线程的名字
34       */
35      @Override
36      public void run() {
37          for (int i = 0; i <10; i++) {
38              System.out.println(this.name1+"
"+Thread.currentThread().getName()+"    i:"+i);
39          }
40      }
41  }
```

4.3.2.4. 使用Runnable接口

- 在Thread类的构造方法中， 有一个重载的构造方法， 参数是Runnable 接口。 因此， 可以通过Runnable接口的实现类对象进行Thread对象的实例化。
- 这里Thread内部默认有一个run,又通过runnable传入一个run,为什么优先调用的是传入的run?

  如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
    public static void main(String[] args) {
        // Runnable接口的匿名实现类
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                System.out,println("子线程处理的逻辑");
            }
        };
        // 实例化线程对象
        Thread thread = new Thread(runnable);
    }
}
```

4.3.2.5. 优缺点对比

- 继承的方式： 优点在于可读性比较强， 缺点在于不够灵活。 如果要定制一个线程， 就必须要继承自Thread类， 可能会影响原有的继承体系。
- 接口的方式： 优点在于灵活， 并且不会影响一个类的继承体系。 缺点在于可读性较差。

> 后面课程中， 用的比较多的方式是使用接口的方式。

课上练习

```
/*
 * 实现四个售票员售票
 * 将四个售票员看做四个线程
 * 数据:一份
 * 实现多线程的方式两种:
 * 第一种方式:通过创建Thread子类的方式实现功能----线程与任务绑定在了一起,操作不方法
 * 第二种:将任务从线程中分离出来,哪个线程需要工作,就将任务交给谁,操作方便,灵活
 */
public class Demo6 {
    public static void main(String[] args) {
        //创建4个线程代表4个售票员

//线程与任务不分离
//        SubThread s1 = new SubThread();
//        SubThread s2 = new SubThread();
//        SubThread s3 = new SubThread();
//        SubThread s4 = new SubThread();
//
```

```
//          //开启线程
//          s1.start();
//          s2.start();
//          s3.start();
//          s4.start();

          //线程与任务分离测试
          Ticket ticket = new Ticket();
          Thread t1 = new Thread(ticket);
          Thread t2 = new Thread(ticket);
          Thread t3 = new Thread(ticket);
          Thread t4 = new Thread(ticket);
          //开启线程
          t1.start();
          t2.start();
          t3.start();
          t4.start();

     }
}

//线程与任务不分离
class SubThread extends  Thread{
     static  int num = 20;//想让大家共用这个num
     @Override
     public void run() {
          for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName()+"
num:"+ --num);
          }
     }
}
```

```
50
51 //线程与任务分离
52 class Ticket implements  Runnable{
53     int num = 20;//想让大家共用这个num
54     @Override
55     public void run() {
56         for (int i = 0; i < 5; i++) {
57
   System.out.println(Thread.currentThread().getName()+"
   num:"+ --num);
58         }
59     }
60 }
```

4.3.3. 线程名字的设置

每一个线程， 都有一个名字。 如果在实例化线程的时候不去设定名字， 那么这个线程会拥有一个默认的名字。

- 设置线程的名字， 使用方法 setName(String name)

```
1  /**
2   * @Author 千锋大数据教学团队
3   * @Company 千锋好程序员大数据
4   * @Description
5   */
6  public class Program {
7      public static void main(String[] args) {
8          Thread thread = new Thread(() -> {
9              System.out.println("子线程的逻辑");
10         });
11         // 设置线程的名字
```

```
        thread.setName("子线程的名字");
    }
}
```

- Thread类对象， 在进行实例化的时候， 可以同时设置线程的名字。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
    public static void main(String[] args) {
        // 使用接口的方式进行线程的实例化
        Thread thread = new Thread(() -> {}, "线程的名字");
    }
}
```

- 如果使用继承Thread类的方式进行的实例化， 可以添加一个构造方法， 进行实例化对象的同时进行名称的设置。 在构造方法中， 使用super(String) 进行父类方法的调用。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class MyThread extends Thread {
    public MyThread(String name) {
        super(name);
```

```
    }

    @Override
    public void run() {
        System.out.println("子线程的逻辑");
    }
}
```

设置线程名字， 可以使用上述三种方式， 但是获取线程线程的名字， 只有一个方法， 就是 getName()

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
    public static void main(String[] args) {
        // 使用接口的方式进行线程的实例化
        Thread thread = new Thread(() -> {}, "线程的名字");

        System.out.println(thread.getName());
    }
}
```

4.3.4. 线程的礼让

线程礼让， 就是当前已经抢到CPU资源的正在运行的线程， 释放自己持有的CPU资源， 回到就绪状态， 重新参与CPU时间片的争抢。

```
/**
```

```
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Program {
    public static void main(String[] args) {
        // 使用接口的方式进行线程的实例化
        Runnable runnable = () -> {
            for (int i = 0; i < 10; i++) {

System.out.println(Thread.currentThread().getName() + "
: " + i);
                if (i == 5) {
                    Thread.yield();
                }
            }
        };
        // 实例化两个线程,  处理的逻辑完全相同
        Thread thread0 = new Thread(runnable, "t0");
        Thread thread1 = new Thread(runnable, "t1");

        thread0.start();
        thread1.start();
    }
}
```

4.4. 线程同步(会)

4.4.1. 临界资源问题

4.4.1.1. 临界资源

在一个进程中， 多个线程之间是可以资源共享的。 如果在一个进程中的一个资源同时被多个线程访问， 这个资源就是一个临界资源。

如果多个线程同时访问临界资源， 会对这个资源的值造成影响。

4.4.1.2. 临界资源问题

多个线程同时访问一个资源的情况下， 一个线程在操作这个资源的时候，将值取出进行运算， 在还没来得及进行修改这块空间的值之前， 值又被其他的线程取走了。 此时就会出现临界资源的问题， 造成这个资源的值出现不是我们预期的值。

4.4.1.3. 解决方案

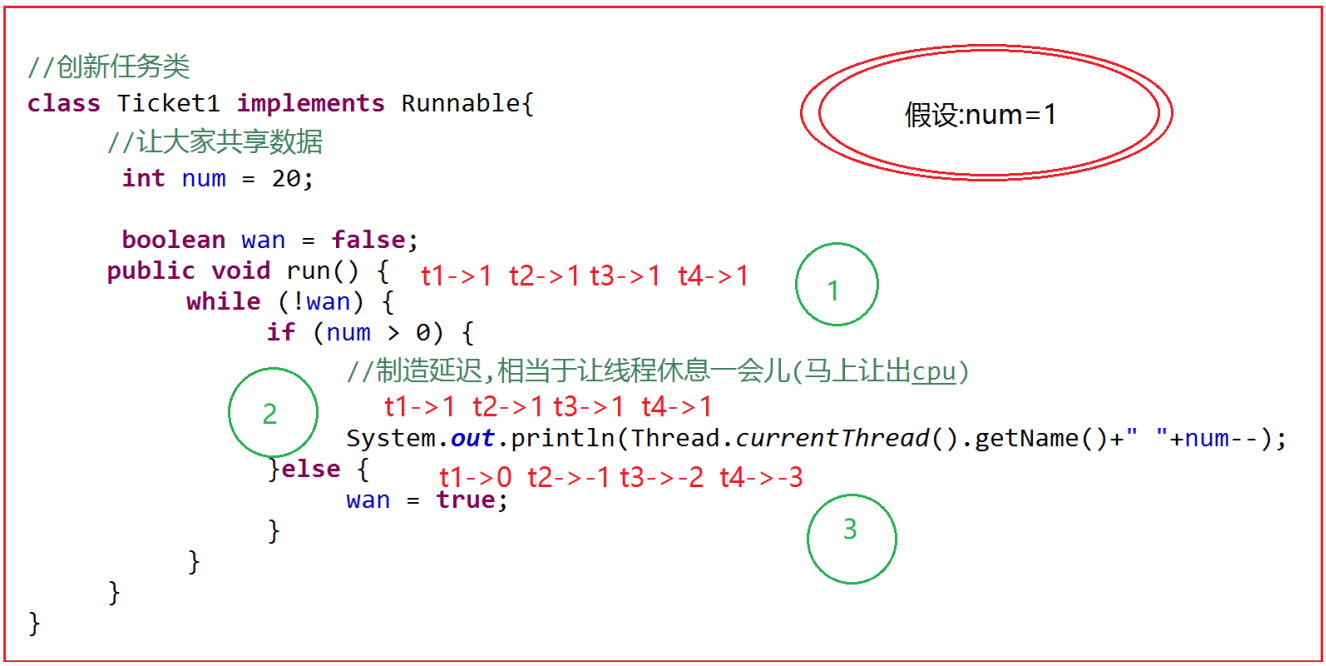
临界资源问题出现的原因就是多个线程在同时访问一个资源， 因此解决方案也很简单， 就是不让多个线程同时访问即可。

在一个线程操作一个资源的时候， 对这个资源进行“上锁”， 被锁住的资源， 其他的线程无法访问。

> 类似多个人去公共卫生间， 每一个人在进到卫生间的时候， 都会从里面进行反锁。 此时， 其他人如果也需要使用这个卫生间， 就得在门外等待。

4.4.2. 案例理解

- 分析: 在实现四个售票员售票的案例中,出现了一些不正常的问题,影响了线程安全.



对图示的说明：
作用:重现线程安全问题的出现
过程：
1.假设num=1,方便错误重新,t1,t2,t3,t4分别代表4个线程,4个线程公用一个num,在开始执行任务前,4个线程对应的num都是1
2.开始执行后,线程之间会抢cpu,假设t1抢到了cpu,马上执行任务,但是只要时间片用完,t1会立刻释放cpu,然后停止工作,并且停在当前位置,保持就绪状态,等待下次抢cpu.在某些情况下有可能出现t1,t2,t3,t4同时停留在2处,此时他们都值都是1.
3.因为在2处,已经在if判断内部,if判断失去了作用,此时假设t1又抢到了cpu,执行num--后,num值变成了0,t2抢到cpu,执行num--,num变成-1,t3抢到cpu,执行num--,num变成-2,t4抢到cpu,执行num--,num继续变成-3.

- 线程安全问题: 4个线程共用了一个数据,出现了-1,-2,-3等错误的数据
- 原因分析:

出现了临界资源问题
1.共用了一个数据
2.共享语句有多条,一个线程使用cpu,没有使用完,cpu被抢走,当再次抢到cpu的时候,直接执行后面的语句,造成了错误的发生.

- 解决:在代码中使用同步代码块儿或同步方法(同步锁)
- 解释:在某一段任务中,同一时间只允许一个线程执行任务,其他的线程即使抢到了cpu,也无法进入当前的任务区间,只有当当前的线程将任务执行完后,其他的线程才能有资格进入
- **示例代码**

```
package com.qf.test;

public class Demo7 {
    public static void main(String[] args) {
        //线程与任务分离测试
        Ticket1 ticket = new Ticket1(20);
        Thread t1 = new Thread(ticket);
        Thread t2 = new Thread(ticket);
        Thread t3 = new Thread(ticket);
        Thread t4 = new Thread(ticket);
        //开启线程
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}

class Ticket1 implements  Runnable{
    int num;//想让大家共用这个num
    boolean flag = true;//判断车票是否卖完

    //作为锁
    Object obj = new Object();
    public Ticket1(int num) {
        this.num = num;
    }
```

```
28
29      @Override
30      public void run() {
31          while (flag) {
32              //让当前的线程休息5000毫秒
33              try {
34                  Thread.sleep(5000);
35              } catch (InterruptedException e) {
36                  e.printStackTrace();
37              }
38              //不能用匿名对象直接充当锁
39              //synchronized (new Object())
40              //对象作为锁
41              synchronized (obj) {
42                  if (num > 0) {//由于同一时间只能有一个线程进
    入,所以我们成为线程互斥
43
    System.out.println(Thread.currentThread().getName() + "
    num:" + --num);
44                  } else {
45                      flag = !flag;
46                  }
47              }
48          }
49      }
50  }
51
```

4.4.3. 线程锁

- 线程锁， 就是用来“锁住”一个临界资源， 其他的线程无法访问。 在程序中， 可以分为**对象锁**和**类锁**

- 对作为锁的对象的要求: 1.必须是对象
  2.必须保证被多个线程共享
- 对象锁： 任何的普通对象或者this， 都可以被当做是一把锁来使用。但是需要注意， 必须要保证不同的线程看到的锁， 需要是同一把锁才能生效。 如果不同的线程看到的锁对象是不一样的， 此时这把锁将没有任何意义。

  注意: 不能直接使用匿名对象作为锁,因为这样每次都是在重新new,要保证锁是被大家共享.
- 类锁： 可以将一个类做成锁， 使用类.class (类的字节码文件对象)来作为锁。因为类的字节码文件的使用范围太大,所以一般我们不使用他作为锁,只有在静态方法中.

4.4.4. synchronized

如果在一个方法中， 所有的逻辑， 都需要放到同一个同步代码段中执行。这样的方法， 可以直接做成同步方法。

4.4.4.1 同步方法

- 非静态同步方法,使用的对象锁(this)

  是某个对象实例内，synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法
- 静态同步方法,使用的类锁(当前类的.class文件)

是某个类的范围，synchronized static aStaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用。

静态同步函数在进内存的时候不会创建对象，但是存在其所属类的字节码文件对象，属于class类型的对象，所以静态同步函数的锁是其所属类的字节码文件对象

4.4.4.2 同步代码块

synchronized关键字用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。

同步代码段， 是来解决临界资源问题最常见的方式。 将一段代码放入到同步代码段中， 将这段代码上锁。

第一个线程抢到了锁标记后， 可以对这个紧接资源上锁， 操作这个临界资源。 此时其他的线程再执行到synchronized的时候， 会进入到锁池， 直到持有锁的线程使用结束后， 对这个资源进行解锁。 此时， 处于锁池中的线程都可以抢这个锁标记， 哪一个线程抢到了， 就进入到就绪态， 没有抢到锁的线程， 依然处于锁池中。

- 同步代码块儿的构成:

```
synchronized(锁(对象)){
    同步的代码
}
```

- 同步代码块儿的特点:1.可以保证线程的安全 2.由于每次都要进行判断处理,所以降低了执行效率

4.4.4.3 比较同步代码块儿和同步函数

- 同步代码块儿使用更加的灵活,只给需要同步的部分代码同步即可,而同步函数是给这个函数内的所有代码同步.
- 由于处于同步的代码越少越好,所以最好使用同步代码块儿
- 什么时候使用同步代码块儿或者同步方法

```
1.多个线程共享一个数据
2.至少有两个线程
```

4.4.4.4 synchronized在继承中的使用

synchronized关键字是不能继承的，也就是说，基类的方法synchronized f(){}在继承类中并不自动是synchronized f(){}，而是变成了f(){}。继承类需要你显式的指定它的某个方法为synchronized方法；

4.4.4.5. 示例代码

```
/*
 * 实例:两个人同时向银行同一个账户存钱
 * 两个人---两个线程       一份数据
 */

public class Demo8 {
    public static void main(String[] args) {
        //1.创建任务类对象
        SaveMoney saveMoney = new SaveMoney();
        //2.创建两个线程当做两个人
        Thread t0 = new Thread(saveMoney);
        Thread t1 = new Thread(saveMoney);
        //3.开启线程
        t0.start();
```

```
15          t1.start();
16
17      }
18  }
19
20  class Bank {
21      int money;
22      //使用同步代码块儿
23  //  public void addMoney(int money) {
24  //    synchronized (Bank.class) {
25  //        this.money += money;
26  //        System.out.println(this.money);
27  //    }
28  //  }
29      //使用同步函数
30      //非静态的同步函数
31      //在synchronized后面默认有一个this
32  //  public synchronized void addMoney(int money) {
33  //      this.money += money;
34  //      System.out.println(this.money);
35  //  }
36      //静态的同步函数
37      //在synchronized后面默认有一个当前类的字节码文件对象-----
    Bank.class
38      public synchronized static void addMoney(int money)
    {
39  //      this.money += money;
40  //      System.out.println(this.money);
41      }
42
43      public synchronized void show() {
44
45      }
```

```
}

class SaveMoney implements Runnable{
    Bank bank = new Bank();
    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {

            bank.addMoney(100);
        }

    }
}
```

4.4.5. 线程应用(单例设计模式)

懒汉式单例， 在多线程的环境下， 会出现问题。 由于临界资源问题的存在， 单例对象可能会被实例化多次。

因此， 单例设计模式， 尤其是懒汉式单例， 需要针对多线程的环境进行处理。

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Demo10 {
    public static void main(String[] args) {
        Test1 test = new Test1();
        Thread t0 = new Thread(test);
        Thread t1 = new Thread(test);
```

```
        t0.start();
        t1.start();
    }
}

class Test1 implements Runnable{
    public void run() {
        SingleInstance2 singleInstance2 = SingleInstance2.getInstance();
    }
}
//饿汉式,由于公共方法中只有一行公共的代码,所以不会产生线程安全问题
class SingleInstance1{
    private static final SingleInstance1 s = new SingleInstance1();
    private SingleInstance1() {
    }
    public static SingleInstance1 getInstance() {
        return s;
    }
}

//懒汉式,
class SingleInstance2{
    private static  SingleInstance2 s = null;
    private SingleInstance2() {
    }
    public  static SingleInstance2 getInstance() {
        if (s == null) {//尽量减少线程安全代码的判断次数,提高效率

```

```
39              synchronized (SingleInstance2.class) {//使用同步代码块儿实现了线程安全
40                  if (s == null) {
41                      s = new  SingleInstance2();
42                  }
43              }
44          }
45          return s;
46      }
47  }
```