day07【排序算法、异常、多线程基础】

今日内容

- 算法
  - 冒泡排序
  - 选择排序
  - 普通查找
  - 二分查找
- 异常
  - 异常的介绍
  - 抛出异常
  - 声明异常
  - 捕获异常
  - 获取异常信息
  - finally代码块
  - 异常注意事项
  - 自定义异常
- 多线程
  - 并发与并行
  - 进程与线程
  - 线程调度
  - Thread概述
  - 创建线程的2种基本方式
  - Runnable使用优势
  - 匿名内部类创建线程对象

教学目标

- [ ] 能够理解冒泡排序的执行原理
- [ ] 能够理解选择排序的执行原理
- [ ] 能够理解二分查找的执行原理
- [ ] 能够辨别程序中异常和错误的区别
- [ ] 能够说出异常的分类
- [ ] 列举出常见的三个运行期异常
- [ ] 能够使用try...catch关键字处理异常
- [ ] 能够使用throws关键字处理异常
- [ ] 能够自定义并使用异常类
- [ ] 说出进程和线程的概念
- [ ] 能够理解并发与并行的区别
- [ ] 能够描述Java中多线程运行原理
- [ ] 能够使用继承类的方式创建多线程
- [ ] 能够使用实现接口的方式创建多线程
- [ ] 能够说出实现接口方式的好处

第一章 常见算法

知识点--冒泡排序

目标

- 理解冒泡排序的执行原理

路径

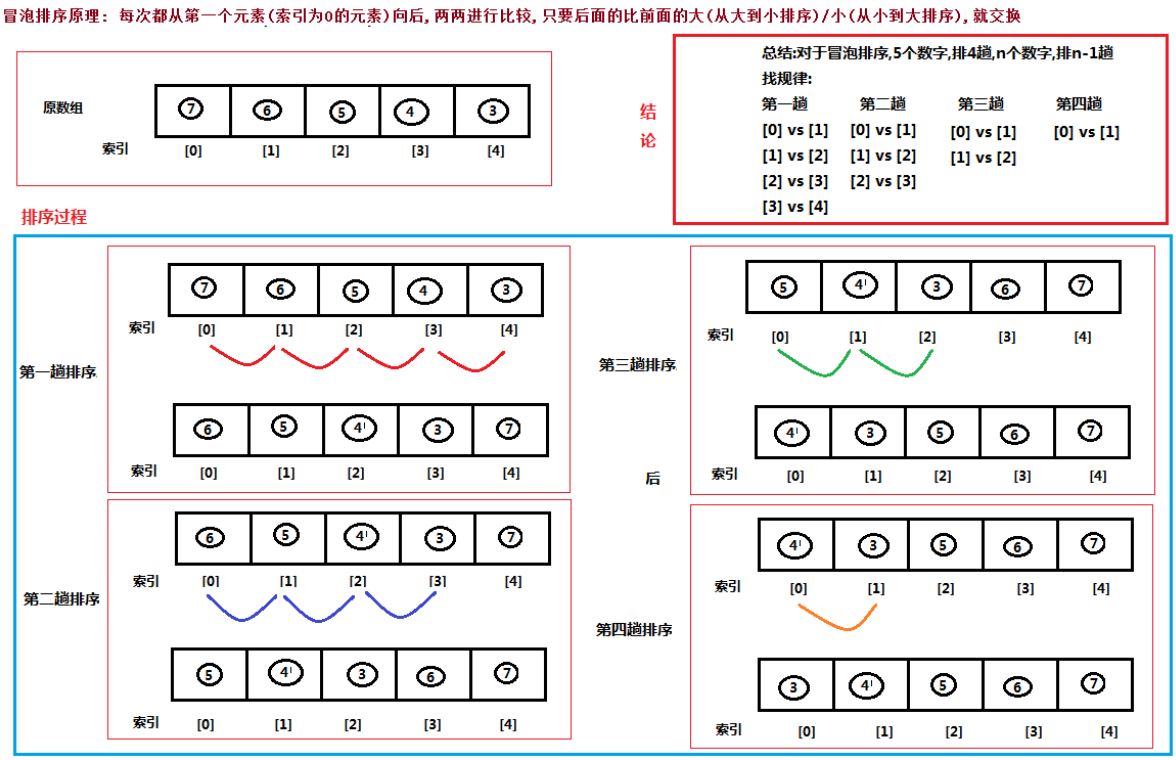
- 冒泡排序原理
- 冒泡排序图解
- 演示冒泡排序

讲解

1.1.1冒泡排序原理

- 对要进行排序的数组中相邻的数据进行两两比较，将较大的数据放在后面
- 每一轮比较完毕，最大值在最后面，下一轮比较就少一个数据参与
- 每轮比较都从第一个元素(索引为0的元素)开始
- 依次执行，直至所有数据按要求完成排序
- 如果有n个数据进行排序，总共需要比较n-1轮

1.1.2冒泡排序图解



1.1.3演示冒泡排序

需求:在数组中存储，“7, 6, 5, 4, 3”五个数据，并使用冒泡排序进行排序

//测试类代码

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

```
        //定义要进行排序的数组
        int[] arr = new int[]{7, 6, 5, 4, 3};
        //定义外循环，控制比较的轮数。
        //总的轮数比长度少1次，外循环的长度需要-1
        for (int i = 0; i < arr.length - 1; i++) {
            //定义内循环，控制每轮比较的顺序
            //j和j+1位置比较，j+1最大为最大索引，索引长度需要-1。
            //结束范围随着轮数i的增加而减少，长度需要 -i
            for (int j = 0; j < arr.length - 1 - i; j++) {
                //将较大值放到后面
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
        System.out.println(Arrays.toString(arr));
    }
}
```

小结

```
```

知识点--选择排序

目标

- 理解选择排序的执行原理

路径

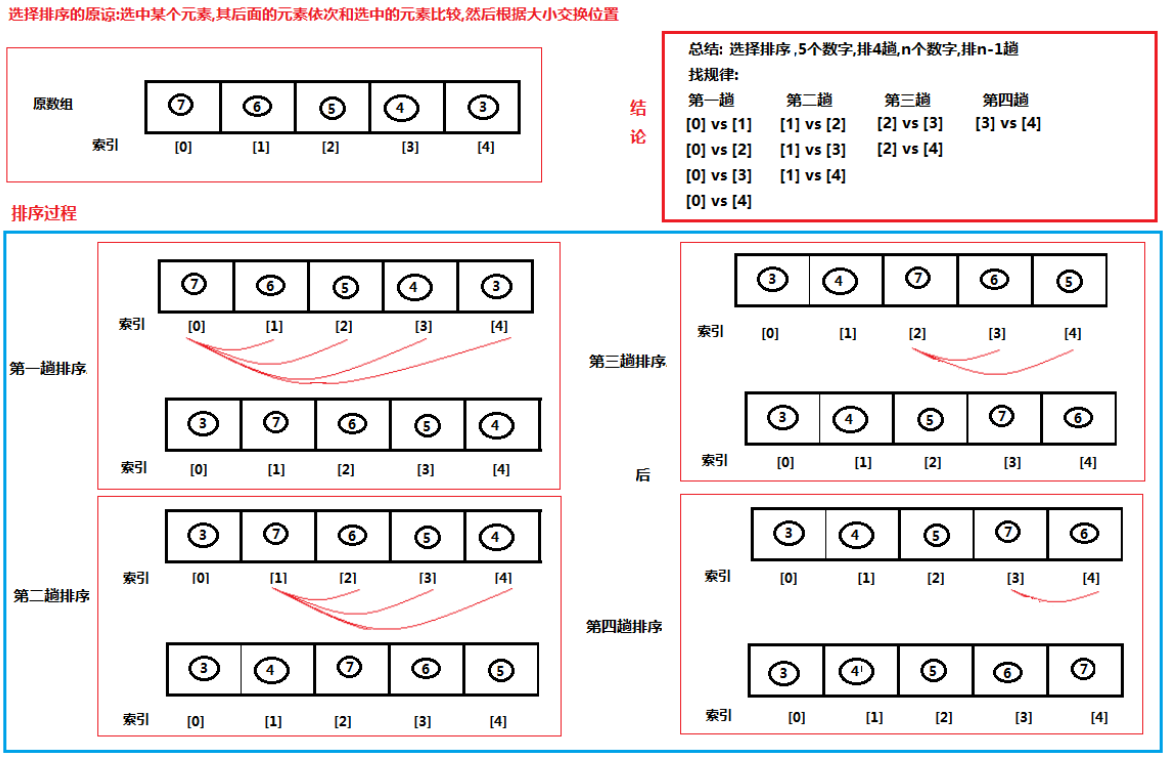
- 选择排序原理
- 选择排序图解
- 演示选择排序

讲解

1.2.1选择排序原理

- 对要进行排序的数组中，使某个元素依次和后面的元素逐个比较，将较大的数据放在后面
- 每一轮比较完毕，最小值在最前面，下一轮比较就少一个数据参与
- 每轮比较都从下一个(轮数+1)元素开始
- 依次执行，直至所有数据按要求完成排序
- 如果有n个数据进行排序，总共需要比较n-1轮

1.2.2选择排序图解



1.2.3演示选择排序

需求:在数组中存储，“7, 6, 5, 4, 3”五个数据，并使用选择进行排序

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //定义要进行排序的数组
        int[] arr = new int[]{7, 6, 5, 4, 3};
        // 定义外循环，控制比较的轮数。
        // 总的轮数比长度少1次，外循环的长度需要-1
        for (int i = 0; i < arr.length - 1; i++) {
            //定义内循环，控制比较的顺序
            //使用i代表每轮最前面的数值，用j代表每轮变动的数值，变动的数值从每轮开始的数值下一
位开始，所以j=i+1
            for (int j = i + 1; j < arr.length; j++) {
                //将较小的值放到前面
                if (arr[i] > arr[j]) {
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }
        System.out.println(Arrays.toString(arr));
    }
}
```

小结

知识点--普通查找

目标

- 理解普通查找执行原理

路径

- 普通查找原理
- 演示普通查找

讲解

1.3.1普通查找原理

- 遍历数组，获取每一个元素，
- 判断当前遍历的元素是否和要查找的元素相同，
- 如果相同就返回该元素的索引，结束循环。
- 循环结束视为没有找到，就返回一个负数作为标识(一般是-1)

1.3.2演示普通查找

需求:定义数组存储“7, 6, 5, 4, 3”五个数据，使用普通查找找到7的位置。

```
public class Test {
    public static void main(String[] args) {
        // 定义数组
        int[] arr = new int[]{7, 6, 5, 4, 3};
        //定义记录索引的变量
        int index = -1;
        //使用键盘录入要查找的数值
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入要查找的数值");
        int num = sc.nextInt();
        //遍历数组，查找与要找的数值相同的索引位置
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == num) {
                index = i;
                break;
            }
        }
        System.out.println("查找的位置为:"+index);
    }
}
```

小结

```

```

知识点--二分查找

目标

- 理解二分查找的执行原理

路径

- 二分查找
- 二分查找图解
- 演示二分查找

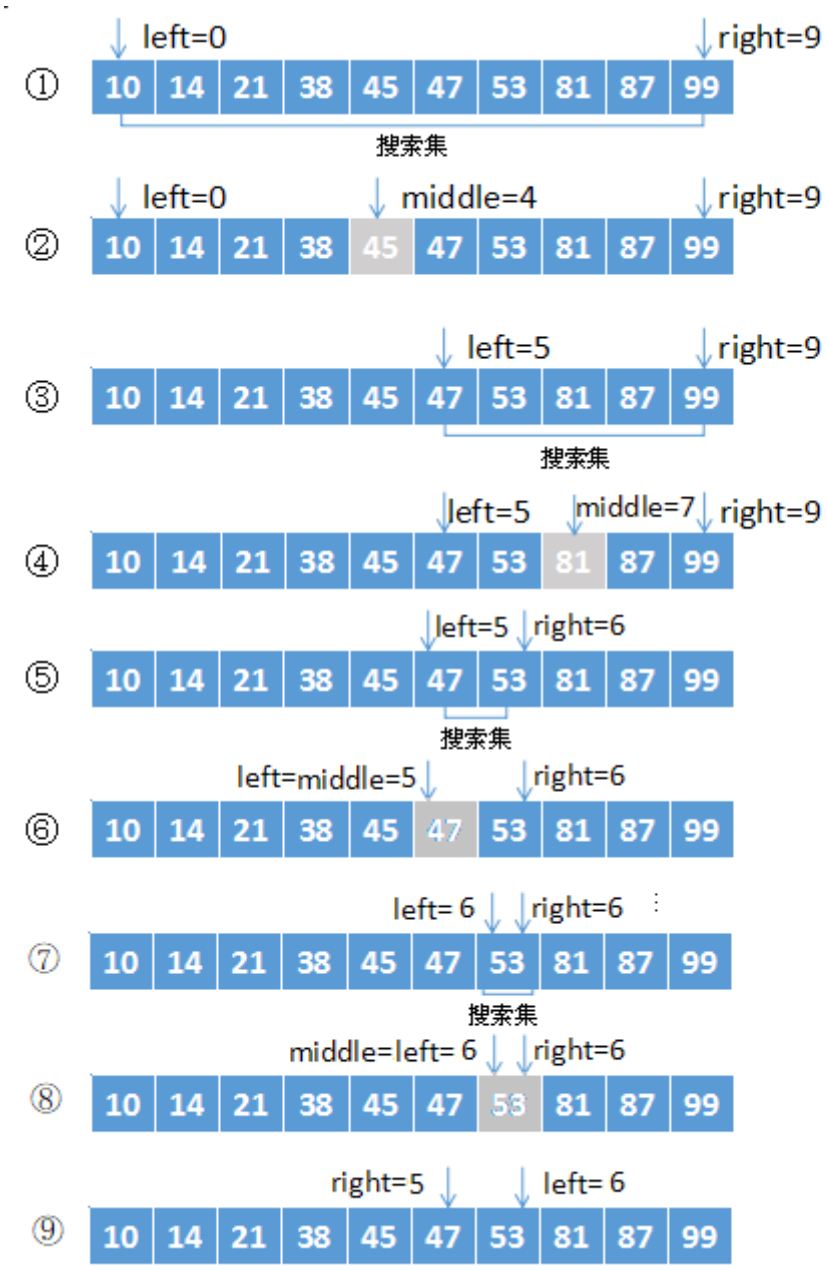
讲解

1.4.1二分查找原理

- 前提:**二分查找对数组是有要求的,数组必须已经排好序**
- 使用循环定义最小和最大索引位置，并判断最小索引小于等于最大索引时，执行以下操作
- 根据最小索引和最大索引获取中间索引
- 根据中间索引获取值，和要查找的元素比对，如果相同就返回索引
- 如果不相同，就比较中间元素和要查找的元素的值
    - 如果中间元素大于查找元素，说明查找位置在中间元素左侧，修改最大索引为中间索引-1
    - 如果中间元素小于查找元素，说明查找位置在中间元素右侧，修改最小索引为中间索引+1
- 从第1步开始重复执行，直到找到内容，或判断条件为false。

1.4.2二分查找图解

假设有一个给定有序数组(10,14,21,38,45,47,53,81,87,99),要查找50出现的索引



1.4.3演示二分查找

需求：将“1, 4, 16, 22, 25, 44, 55, 67, 88, 100”存入数组，并查找制定数值的位置

//二分查找代码

```
public class Test {
    public static void main(String[] args) {
        //  有序数组
        int[] arr = {1, 4, 16, 22, 25, 44, 55, 67, 88, 100};
        //  要查找的数值
        int findNum = 66;
        //  左索引
        int left = 0;
        //  右索引
        int right = arr.length - 1;
        //  中间索引
        int middle = (left + right) / 2;
        //  循环比较(左索引<=右索引)
        while (left <= right) {
            //  比较中间索引与要查找的数值关系（==  >  <）
```

```
            if (arr[middle] == findNum) {
                //  中间索引==查找的数值  结束循环，返回索引
                System.out.println("索引是：" + middle);
                System.exit(0);
                // return;
            } else if (arr[middle] > findNum) {
                //  中间索引>查找的数值  右索引=中间索引-1
                right = middle - 1;
            } else {
                //  中间索引<查找的数值  左索引=中间索引+1
                left = middle + 1;
            }
            // 修改中间索引
            middle = (left + right) / 2;
        }
        System.out.println("无索引" + -1);
    }
}
```

小结

```
```

第二章 异常

知识点--异常介绍

目标

- 了解异常体系和分类，理解异常的产生过程

路径

- 概述
- 异常体系
- 异常分类
- 演示异常的产生过程
- 异常中的关键字

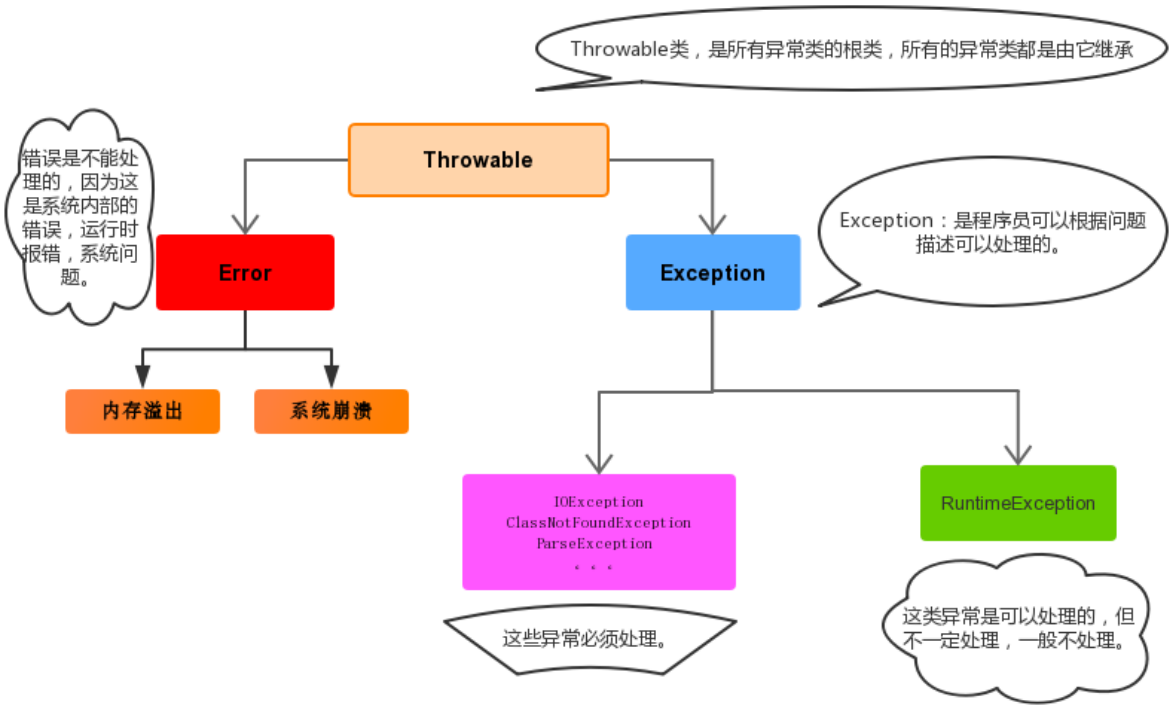
讲解

2.1.1概述

- 异常指程序在执行过程中,出现的非正常情况
- java中默认将异常抛给jvm处理，而jvm处理的方式就是中断运行、
- 在Java等面向对象的编程语言中，将异常问题，用指定的类来表示，并提供一定的处理办法。

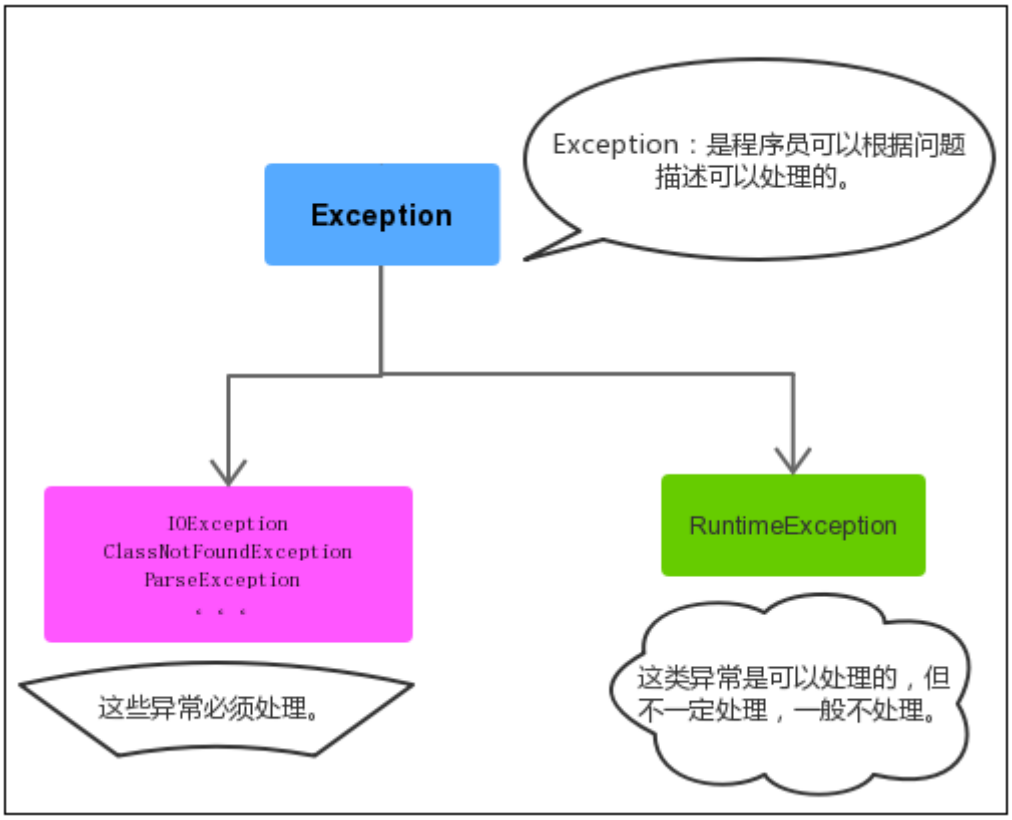
2.1.2异常体系

- `Throwable` 类是 Java 语言中所有**错误**或**异常**的超类，分类Error和Exception两个方向
- `java.lang.Error` 合理的应用程序不应该试图捕获的严重问题
  - 表示严重错误，无法处理的错误，只能事先避免，好比绝症。
  - 一旦出现，程序终止。
- `java.lang.Exception`，合理的应用程序想要捕获的条件
  - 表示异常，程序员可以修改代码解决，是必须要处理的。好比感冒、阑尾炎。
  - 一旦出现，由开发者决定如何处理
    - 给别人处理：将代码丢给调用者处理。
    - 自己处理：将问题截获，进行相应处理。



2.1.3异常(Exception)分类

- **编译时期异常**:checked异常。编译时期,检查异常,如果有异常没处理,则编译失败。(如日期格式化异常)
- **运行时期异常**:runtime异常。编译时期,运行异常不会被编译器检测(不报错)，运行时期,检查异常.如果有异常没处理,则运行失败。(如数学异常)



2.1.4演示异常的产生过程

需求：通过演示ArrayIndexOfBoundsException(数组索引越界异常)异常，分析异常的产生与传递过程

测试类代码

```
public class Test {
    public static void main(String[] args) {
        //定义数组
        int[] arr = new int[]{1, 2, 3};
        //使用获取数组指定位置元素的方法
        int ele = getElement(arr, 3);
    }

    public static int getElement(int[] arr, int index) {
        int num = arr[index];
        return num;
    }
}
```

上述程序执行过程图解：

2.1.5异常中关键字

- try:尝试　代码块，内部包含可能存在问题的代码
- catch:捕捉　与try结合使用，捕捉try代码块中可能存在的某个异常
- finally:最终 与try结合使用，存放try中异常未被捕捉，程序跳转时，一定要执行的代码。
- throw:抛出　手动抛出异常类的实例化对象
- throws:　声明方法中存在的一个或多个异常类型

小结

知识点--抛出异常(throw关键字)

目标

掌握抛出异常的使用

路径

- 概述
- throw的应用了解
- 演示throw关键字的使用

讲解

2.2.1概述

- 作用:在方法内部，抛出一个描述问题的异常对象。
- 处理方式：根据抛出的异常类型(编译异常/运行异常)特性，决定对于该异常的处理方式。
- 格式：throw new 异常类名(参数);

2.2.2throw的应用了解

- Objects类中的静态方法`public static <T> T requireNonNull(T obj)`:查看指定引用对象不是null，如果是null。

源码：

```
public static <T> T requireNonNull(T obj) {
    if (obj == null){
        //对为null的情况执行抛出空指针异常对象，并默认交由调用者处理。
        throw new NullPointerException();
    }
    return obj;
}
```

2.2.3演示throw关键字的使用

需求：演示抛出除数为0异常

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        // System.out.println(10 / 0);//java.lang.ArithmeticException: / by zero
        method(10, 0);
    }

    public static void method(int num1, int num2) {
        if (num2 == 0) {
            throw new ArithmeticException("by:" + num2);
        }
        int num3 = num1 / num2;// java.lang.ArithmeticException: / by zero
    }
}
```

小结

```

```

知识点--声明异常(throws关键字)

目标

掌握声明异常的使用

路径

- 概述
- 演示throws关键字的使用

讲解

2.3.1概述

处理异常的方式，在方法上声明方法中存在的一个或多个问题标识，提醒调用者处理这些异常。

处理方式：

如果是运行时异常,默认被声明抛出(也可以显示声明抛出)。

如果是编译时异常，且没有以捕获方式处理，就必须通过throws声明，否则报错。

当异常出现，将这个异常对象传递到调用者处，并结束当前方法的执行。

声明异常格式：修饰符 返回值类型 方法名(参数) throws 异常类名1,异常类名2...{ }

2.3.2演示声明异常throws

需求：通过被除数为0异常和日期解析异常，演示thorws的使用

//测试类代码

```
public class Test {
    public static void main(String[] args)throws ParseException {
        //运行时异常抛出异常,可以不抛出
        method1();
        //编译时异常抛出异常，调用者需要明确继续抛出还是捕获
        method2();
    }

    private static void method2() throws ParseException {
        String time = "2021-04-01";
        DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
        Date d = df.parse(time);
    }

    public static void method1() throws ArithmeticException {
        System.out.println(10 / 0);
    }
}
```

小结

```

```

知识点--捕获异常(try...catch关键字)

目标

掌握捕获异常的使用

路径

- 概述
- 多个异常捕获常见方式
- 演示try...cathch关键字的使用

讲解

2.4.1概述

作用:处理异常的方式，在方法内，使用特殊格式检查可能出问题的代码，捕捉时机产生的异常，从而避免提交给虚拟机，并给出异常出现后的相应操作。
处理方式:

如果是运行时异常,可以使用捕获的方式处理异常

如果是编译时异常，且没有以声明方式处理，就必须使用捕获方式，否则报错

当异常出现，并被捕获，程序会继续向下执行，不会影响程序运行。

捕获异常格式:

```
捕获异常格式1:
try{
    编写可能会出现异常的代码
}catch(异常类型  e){
    处理异常的代码/记录日志/打印异常信息/继续抛出异常
}
...
```

```
捕获异常格式2:
try{
    编写可能会出现异常的代码
}catch(异常类型1|异常类型2|...e){
    处理异常的代码/记录日志/打印异常信息/继续抛出异常
}
```

2.4.2多个异常捕获常见方式

```
多个异常分别try，分别捕获(处理)。
多个异常一次try，分别捕获(处理)。
多个异常一次try，一次捕获(处理)。
```

2.4.3演示try...cathch关键字的使用

需求：通过被除数为0异常和日期解析异常，演示try...catch的使用及常见异常捕获方式

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        //捕获运行时异常
        method1();
        //捕获编译时异常
        method2();
        // 多个异常分别处理时常见方式
        method3();
    }

    public static void method3() {
        // 多个异常分别try，分别捕获(处理)。
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException e) {
```

```
            System.out.println("您输入的除数为0");
        }
        String time = "2020年01月01日";
        Date parse = null;
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        try {
            parse = sdf.parse(time);
        } catch (ParseException p) {
            System.out.println("您的解析格式或提供的数据有误");
        }
        // 多个异常一次try，分别捕获(处理)。

        try {
            System.out.println(10 / 0);
            parse = sdf.parse(time);
        } catch (ArithmeticException e) {
            System.out.println("您输入的除数为0");
        } catch (ParseException p) {
            System.out.println("您的解析格式或提供的数据有误");
        }
        // 多个异常一次try，一次捕获(处理)。
        try {
            System.out.println(10 / 0);
            parse = sdf.parse(time);
        } catch (ArithmeticException | ParseException e) {
            System.out.println("出问题了");
        }
        try {
            System.out.println(10 / 0);
            parse = sdf.parse(time);
        } catch (Exception e) {
            System.out.println("出问题了");
        }
    }

    public static void method2() {
        String time = "2020年01月01日";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date parse = null;
        try {
            parse = sdf.parse(time);
        } catch (ParseException p) {
            System.out.println("您的解析格式或提供的数据有误");
        }
        System.out.println(parse);
        System.out.println("结束");
    }

    public static void method1() {
        System.out.println("开始");
        try {
            System.out.println(10 / 0);
            System.out.println("over1");
        } catch (ArithmeticException e) {
            System.out.println("您输入的除数为0");
        }
        System.out.println("over");
    }
```

```
}
```

小结

```

```

知识点--获取异常信息

目标

掌握获取异常信息的操作

路径

- 获取异常信息方法
- 演示获取异常信息

讲解

2.5.1获取异常信息方法

- `public String getMessage()`:获取异常的描述信息,原因(提示给用户的时候,就提示错误原因。
- `public String toString()`:获取异常百度的类型和异常描述信息(不用)。
- `public void printStackTrace()`:打印异常的跟踪栈信息并输出到控制台。

> 包含了异常的类型,异常的原因,还包括异常出现的位置,在开发和调试阶段,都得使用printStackTrace。开发中可以用catch将编译期异常转换成运行期异常处理。

2.5.2演示获取异常信息

需求：通过被除数为0异常演示获取异常信息的使用

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        System.out.println("开始");
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException a) {
            System.out.println(a.getMessage());
            System.out.println(a.toString());
            a.printStackTrace();
        }
        System.out.println("结束");
    }
}
```

小结

知识点--finally代码块

目标

掌握finnally的使用

路径

- 概述
- 演示finally代码块

讲解

2.6.1概述

作用:在方法内，负责在try中代码出现异常时，一定会执行的代码，一般用于一些在出现问题后一定要关闭的资源处理。

处理方式:try中的代码执行后，系统会确保final在jvm或该方法结束前，执行final中的代码。

格式:try...[catch...catch...]finally... catch可以没有

2.6.2运行效果常见情况

不try直接抛给虚拟机，程序结束，不执行finally中内容
加上try，不加catch，finally能执行，但是程序不再继续运行
加上try，并catch该异常，finally能执行，程序继续运行

2.6.3演示finally代码块

需求：通过被除数为0异常，演示finally的不同格式使用效果

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        System.out.println("开始");
        // 不try直接抛给虚拟机，程序结束，不执行finally中内容
        // System.out.println(10 / 0);
        // 加上try，不加catch，finally能执行，但是程序不再继续运行
        // try {
        //     System.out.println(10 / 0);
        // } finally {
        //     System.out.println("try中的代码执行了");
        // }
        // 加上try，并catch该异常，finally能执行，程序继续运行
        try {
            System.out.println(10 / 0);
        } catch (ArithmeticException a) {
            System.out.println("异常被捕获了");
        } finally {
            System.out.println("try中的代码执行了");
        }
        System.out.println("结束");
    }
}
//结果
```

小结

知识点--异常注意事项

目标

理解异常注意事项

路径

- 异常处理注意事项介绍
- 演示常处理注意事项

讲解

2.7.1异常处理注意事项介绍

1. try/catch/finally都不可以单独使用
2. 运行时异常被抛出可以不处理，因为方法默认抛出运行时异常。
3. 在try/catch后追加finally代码块，try中的代码执行后，无论是否报异常，finally中一定会被执行，通常用于资源回收
4. 方法重写时的注意事项

- 父类的方法抛出异常，子类覆盖(重写)父类方法时，只能抛出相同的异常或该异常子集。
- 父类的方法未抛出的异常，子类覆盖(重写)父类方法时，只能处理，不能抛出。

5. try...catch捕获多个异常时，前边的类不能是后边类的父类或同类

2.7.2演示常处理注意事项

需求：演示异常处理常见注意事项

//子父类异常-父类代码

```
public class Fu {
    //方法声明编译时异常
    public void method() throws Exception{
    }
    //方法未声明编译时异常
    public void method2() {
    }
}
```

//子父类异常-子类代码

```
public class Zi extends Fu {
    //方法声明编译时异常
    @Override
    public void method() throws ParseException {
        String time = "2020-12-12";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date parse = sdf.parse(time);
    }

    //方法未声明编译时异常
    @Override
    public void method2() {
        String time = "2020-12-12";
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        try {
            Date parse = sdf.parse(time);
        } catch (ParseException p) {

        }
    }
}
```

//测试类代码

```
public class Test {
    public static void main(String[] args) {
        // 1. try/catch/finally都不可以单独使用
        /*
        try{}
        System.out.println();
        catch(){}
        System.out.println();
        finally{}
        */
```

```
        // 运行时异常被抛出可以不处理，因为方法默认抛出运行时异常。
        // System.out.println(10 / 0);
        // 在try/catch后追加finally代码块，try中的代码执行后，无论是否报异常，finally中一定会被执行，通常用于资源回收
        /*
        try {
            System.out.println(5 / 0);
        } catch (ArithmeticException a) {
            System.out.println("捕获被除数为0异常");
        } finally {
            System.out.println("finally执行了");
        }
        */
        // 5. try...catch捕获多个异常时，前边的类不能是后边类的父类或同类
        try {
            System.out.println(5 / 0);
        }
        //前面的异常不能大于或等于后面的异常
       /*
        catch (Exception a) {
            System.out.println("捕获被除数为0异常");
        }
        */
        /*
        catch (ArithmeticException a) {
            System.out.println("捕获被除数为0异常");
        } */
        catch (ArithmeticException a) {
            System.out.println("捕获被除数为0异常");
        }
    }
}
```

小结

```

```

知识点--自定义异常

目标

- 理解异常类的定义和使用

路径

- 自定义异常概述
- 演示自定义异常使用

讲解

2.8.1概述

在开发中根据自己业务的异常情况来定义异常类表示某种异常问题.

Java中异常类具备异常发生中断程序的功能，但一些异常情况是java没有定义的,需要根据业务自行定义(例:年龄负数问题)。

自定义异常类分类:

- 自定义编译期异常: 自定义类 并继承于`java.lang.Exception`。
- 自定义运行时期异常:自定义类 并继承于`java.lang.RuntimeException`。

2.8.2演示自定义异常使用

需求：按照如下要求完成案例

```
模拟注册操作，如果用户名已存在，则抛出异常并提示：亲，该用户名已经被注册。
分析：
    - 首先定义一个注册异常类RegisterException
    - 模拟登陆操作，使用数组模拟数据库中存储的数据，并提供当前注册账号是否存在方法用于判断。
```

// 业务逻辑异常代码

```
public class RegisterException extends Exception {

    public RegisterException(String message){
        super(message);
    }
}
```

//测试类代码

```
public class Test {

    public static void main(String[] args) {

        //模拟数据库中存在的用户名
        String[] useNames = {"zhangsan", "lisi", "wangwu"};
        //使用键盘录入模拟输入用户名的操作
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入您想要申请的账户名...");
        String useName = sc.nextLine();
        //做验证
        // try {
        //     boolean flag = checkUseName(useName, useNames);
        // } catch (RegisterException r) {
        //     r.printStackTrace();
        //     //弹出一个对话框
        // }
        try {
            if (checkUseName(useName, useNames)) {
                System.out.println("恭喜您，您注册的账号未被是申请，您可以继续录入您的专属
密码...");
            }
        } catch (RegisterException r) {
            // r.printStackTrace();
            //弹出一个对话框
```

```
            System.out.println("抱歉，您输入的账户名已被他人注册，请您重新输入您要申请的账
号");
        }

    }

    public static boolean checkUseName(String useName, String[] useNames) throws
RegisterException {

        for (int i = 0; i < useNames.length; i++) {
            String sysUseName = useNames[i];
            //一旦发现用户名在数组用已经存在了，抛出一个用户名已存在异常
            if (sysUseName.equals(useName)) {
                throw new RegisterException("已被注册的用户名:" + useName);
            }
        }
        return true;
    }
}
```

小结

```
```

第三章 多线程

知识点--线程概述

目标

- 理解多线程常见基本概念

路径

- 并行与并发
- 进程与线程
- 线程调度

讲解

3.1.1概述

截至目前学习的程序都是按照顺序结构，自上而下执行单线执行，即单线程。

实际开发中经常出现多重操作，例如让两组循环语句同时执行。

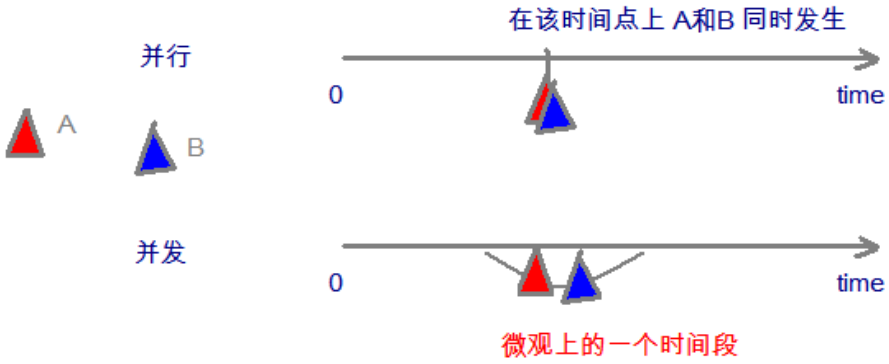
3.1.2并行与并发

前提：在操作系统中，安装了多个程序

在电脑中，多个CPU的电脑被对应多个处理器，被称为多核处理器，核越多，并行处理的程序就越多。

- **并行**：指两个或多个事件在**同一时刻**发生（同时执行）。
    - 微观上的同时运行，多个 CPU 系统中，执行多个程序，每一时刻可以同时执行多个线程。
- **并发**：指两个或多个事件在**同一个时间段内**发生(交替执行)。

- 宏观上的同时运行，单个CPU系统中，执行多个程序，每一时刻只能执行一个线程，感觉是同时运行



3.1.3进程与线程

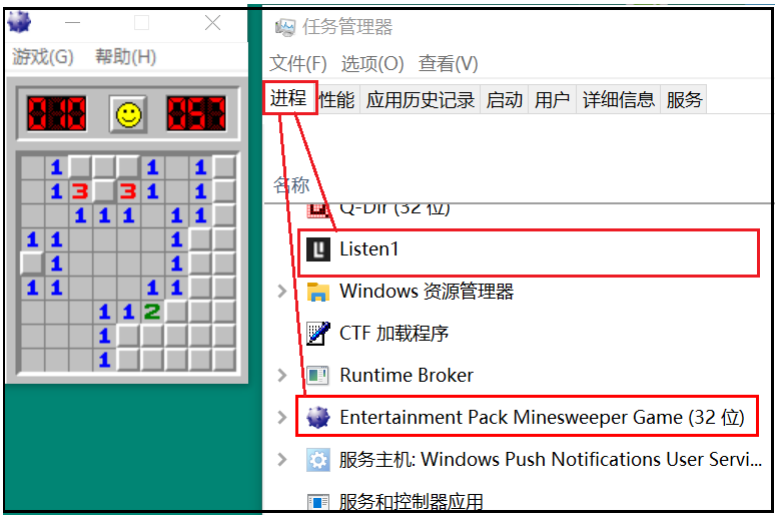
进程：进程是程序在内存中的一次执行过程，是系统运行程序的基本单位

- 系统运行一个程序即是一个进程从创建、运行到消亡的过程。一个应用程序可以同时运行多个进程；

线程：线程是进程中的一个执行单元，负责当前进程中程序的具体执行。

- 一个进程中至少有一个线程，如果有多个线程，该应用被称为多线程应用。

进程示例



线程示例

进程与线程的区别

- 进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。
- 线程：堆空间是共享的，栈空间是独立的，线程消耗的资源比进程小的多。

> 了解知识点
>
> 1:因为一个进程中的多个线程是并发运行的，那么从微观角度看也是有先后顺序的，哪个线程执行完全取决于 CPU 的调度，程序员是干涉不了的。而这也就造成的多线程的随机性。
>
> 2:Java 程序的进程里面至少包含两个线程，主进程也就是 main()方法线程，另外一个是垃圾回收机制线程。每当使用 java 命令执行一个类时，实际上都会启动一个 JVM，每一个 JVM 实际上就是在操作系统中启动了一个线程，java 本身具备了垃圾的收集机制，所以在 Java 运行时至少会启动两个线程。
>
> 3:由于创建一个线程的开销比创建一个进程的开销小的多，那么我们在开发多任务运行的时候，通常考虑创建多线程，而不是创建多进程。

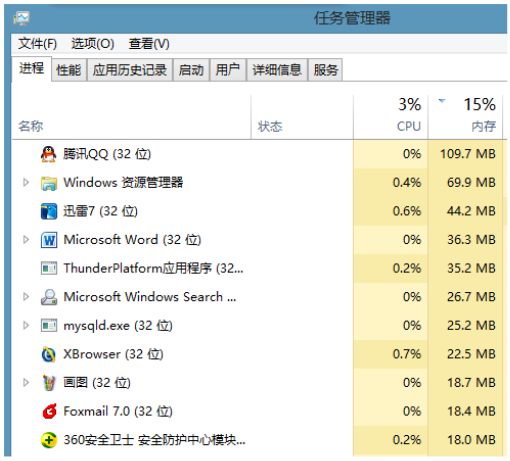
3.1.4线程调度

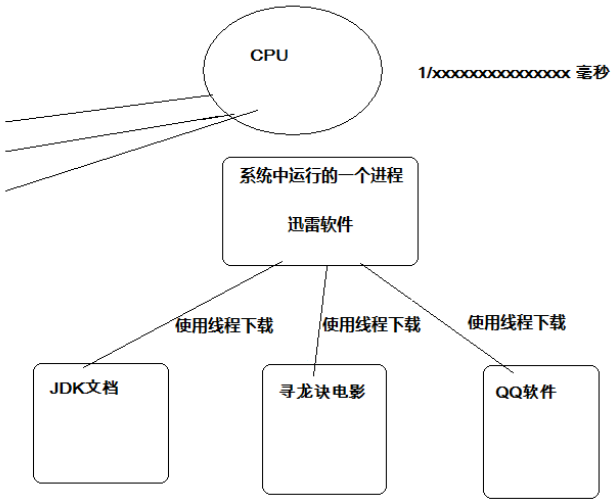
线程从宏观角度上理解线程是并行运行的，但是从微观角度上分析却是串行运行的，即一个线程一个线程的去运行。

- 线程调度:当一个CPU时，以某种顺序执行多个线程，我们把这种情况称之为线程调度。
- 随机性:进程中的多个线程，本质上是依赖于并发式运行，具体哪个线程什么时候被CPU调用执行，程序员并不能直接完全决定。
- 分时调度

  所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。
- 抢占式调度

  优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java使用的为抢占式调度。





小结

知识点--Thread类介绍

目标

- 掌握多线程基本使用

路径

- Thread类概述
- 构造方法
- 常用方法

讲解

3.2.1Thread概述

`java.lang.Thread`类代表**线程**，所有的线程对象都必须是Thread类或其子类的实例.

每个线程的作用是完成一定的任务，实际上就是执行一段程序流即一段顺序执行的代码.

Java使用线程执行体来代表这段程序流，在Tread线程中，使用run()方法代表线程执行体.

3.2.2构造方法

```
public Thread():分配一个新的线程对象。
public Thread(String name):分配一个指定名字的新的线程对象。
public Thread(Runnable target):分配一个带有指定目标新的线程对象。
public Thread(Runnable target,String name):分配一个带有指定目标新的线程对象并指定名字。
```

3.2.3常用方法

```
public String getName():获取当前线程名称。
public void start():导致此线程开始执行; Java虚拟机调用此线程的run方法。
public void run():此线程要执行的任务在此处定义代码。
public static void sleep(long millis):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）。
public static Thread currentThread():返回对当前正在执行的线程对象的引用。
```

小结

```
```

知识点--自定义线程类创建线程

目标

- 掌握自定义线程的使用过程

路径

- 自定义线程类步骤
- 演示自定义线程创建线程

讲解

3.3.1自定义线程步骤

```
定义Thread类的子类，并重写该类的run()方法。
创建Thread子类的实例，即创建了线程对象
调用线程对象的start()方法来启动该线程
```

3.3.2演示自定义线程创建线程

需求：通过继承Thread类方式创建线程对象

//测试类代码

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        //创建线程对象
        MyThread mt = new MyThread("gogogo:");
        //启动线程
        mt.start();

        //主线程中也开始打印100个数
        for (int i = 0; i < 100; i++) {
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() + ":i=" + i);
        }
    }
}
```

自定义线程类：

```
public class MyThread extends Thread {
```

```
    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {

            }
            System.out.println(getName() + "i=" + i);
        }
    }
}
```

小结

知识点--实现Runable接口创建线程

目标

- 掌握Runable创建线程的使用过程

路径

- 实现Runable创建线程步骤
- 演示实现Runable创建线程
- Runnable使用优势

讲解

3.4.1实现Runable创建线程步骤

定义Runnable接口的实现类，并重写该接口的run()方法。
创建Runnable实现类的实例，作为Thread的构造参数创建Thread对象。
调用Thread线程对象的start()方法来启动线程。

3.4.2演示实现Runable创建线程

需求：通过实现Runnable方式部类创建对象

//测试类代码

```
public class MyRunnable  implements Runnable {
    int i=10;
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
```

```
                Thread.sleep(1000);
            } catch (InterruptedException ie) {

            }
            System.out.println(Thread.currentThread().getName() + "i=" + i);
        }
    }
}
```

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        //创建实现类对象
        MyRunnable mr = new MyRunnable();
        //创建线程对象，并将Runnable实现类对象作为参数
        Thread t = new Thread(mr, "gogogo:");
        t.start();

        //主线程中也开始打印100个数
        for (int i = 0; i < 100; i++) {
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() + ":i=" + i);
        }
    }
}
```

3.4.3使用Runnable的优势

- Runnable适合多个相同的程序代码的线程去共享同一个资源。
- Runnable可以避免java中的单继承的局限性。
- Runnable方式增加程序的健壮性，实现解耦操作，代码既可以被多个线程使用又保持了与线程的独立性。
- 线程池只能放入实现Runable或Callable类线程，不能直接Thread或其子类。

小结

```

```

知识点--匿名内部类创建线程

目标

- 掌握匿名内部类创建线程使用过程

路径

- 匿名内部类创建线程步骤
- 演示匿名内部类创建线程

讲解

3.5.1匿名内部类创建线程步骤

使用线程的匿名内部类方式，可以方便的实现每个线程执行不同的线程任务操作。

- 使用匿名内部类的方式实现Runnable接口，
- 重写Runnable接口中的run方法
- 将匿名内部类创建的接口对象传入Thread对象
- 调用start方法执行线程

3.5.2演示匿名内部类创建线程

需求：通过匿名内部类创建多线程对象执行操作

//测试类代码

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
/*        //用匿名内部类创建一个Runnable的实现类对象
        Runnable r = new Runnable() {
            //重写方法

            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {

                    }
                    System.out.println(Thread.currentThread().getName() + "i=" +
i);
                }
            }
        };
        //创建线程对象
        Thread t = new Thread(r, "gogogo:");*/
        Thread t = new Thread(new Runnable() {
            //重写方法

            @Override
            public void run() {
                for (int i = 0; i < 100; i++) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {

                    }
                    System.out.println(Thread.currentThread().getName() + "i=" +
i);
                }
            }
        }, "gogogo:");
        //启动线程
        t.start();


        //主线程中也开始打印100个数
        for (int i = 0; i < 100; i++) {
```

```
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() + ":i=" + i);
        }
    }
}
```

小结

```
```