Day_1

多线程的创建，方式一：继承于 Thread 类

1. 创建一个继承于 Thread 类的子类
2. 重写 Thread 类的 run() --> 将此线程执行的操作声明在 run()中
3. 创建 Thread 类的子类的对象
4. 通过此对象调用 start()

创建多线程的方式二：实现 Runnable 接口

1. 创建一个实现了 Runnable 接口的类
2. 实现类去实现 Runnable 中的抽象方法：run()
3. 创建实现类的对象
4. 将此对象作为参数传递到 Thread 类的构造器中，创建 Thread 类的对象
5. 通过 Thread 类的对象调用 start()

比较创建线程的两种方式。

开发中：优先选择：实现 Runnable 接口的方式

原因：1. 实现的方式没有类的单继承性的局限性

2. 实现的方式更适合来处理多个线程有共享数据的情况。

联系：public class Thread implements Runnable

相同点：两种方式都需要重写 run(),将线程要执行的逻辑声明在 run()中。

测试 Thread 中的常用方法：

1. start():启动当前线程；调用当前线程的 run()
2. run(): 通常需要重写 Thread 类中的此方法，将创建的线程要执行的操作声明在此方法中
3. currentThread():静态方法，返回执行当前代码的线程
4. getName():获取当前线程的名字
5. setName():设置当前线程的名字
6. yield():释放当前 cpu 的执行权
7. join():在线程 a 中调用线程 b 的 join(),此时线程 a 就进入阻塞状态，直到线程 b 完全执行完以后，线程 a 才结束阻塞状态。
8. stop():已过时。当执行此方法时，强制结束当前线程。
9. sleep(long millitime):让当前线程“睡眠”指定的 millitime 毫秒。在指定的 millitime 毫秒时间内，当前线程是阻塞状态。
10. isAlive():判断当前线程是否存活

线程的优先级：

1.

MAX_PRIORITY：10

MIN _PRIORITY：1

NORM_PRIORITY：5 -->默认优先级

2.如何获取和设置当前线程的优先级：

getPriority():获取线程的优先级

setPriority(int p):设置线程的优先级

说明：高优先级的线程要抢占低优先级线程 cpu 的执行权。但是只是从概率上讲，高优先级的线程高概率的情况下被执行。并不意味着只有当高优先级的线程执行完以后，低优先级的线程才执行。

Day_2

例子：创建三个窗口卖票，总票数为 100 张.使用实现 Runnable 接口的方式

1. 问题：卖票过程中，出现了重票、错票 -->出现了线程的安全问题

2.问题出现的原因：当某个线程操作车票的过程中，尚未操作完成时，其他线程参与进来，也操作车票。

3.如何解决：当一个线程 a 在操作 ticket 的时候，其他线程不能参与进来。直到线程 a 操作完 ticket 时，其他线程才可以开始操作 ticket。这种情况即使线程 a 出现了阻塞，也不能被改变。

4.在 Java 中，我们通过同步机制，来解决线程的安全问题。

方式一：同步代码块

```
synchronized(同步监视器){
    //需要被同步的代码
}
```

说明：1.操作共享数据的代码，即为需要被同步的代码。

-->不能包含代码多了，也不能包含代码少了。

2.共享数据：多个线程共同操作的变量。比如：ticket 就是共享数据。

3.同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。

要求：多个线程必须要共用同一把锁。

补充：在实现 Runnable 接口创建多线程的方式中，我们可以考虑使用 this 充当同步监视器

方式二：同步方法。

如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明同步的。

5.同步的方式，解决了线程的安全问题。---好处操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低。 ---局限性

说明：在继承 Thread 类创建多线程的方式中，慎用 this 充当同步监视器，考虑使用当前类充当同步监视器。

使用同步方法解决实现 Runnable 接口的线程安全问题

关于同步方法的总结：

1. 同步方法仍然涉及到同步监视器，只是不需要我们显式的声明。
2. 非静态的同步方法，同步监视器是：this

   静态的同步方法，同步监视器是：当前类本身

使用同步机制将单例模式中的懒汉式改写为线程安全的

```
public class BankTest {
```

```
}
class Bank
    private Bank(){}
    private static Bank instance = null;
    public static Bank getInstance(){
        //方式一：效率稍差
//          synchronized (Bank.class) {
//              if(instance == null){
//                  instance = new Bank();
//              }
//              return instance;
//          }
        //方式二：效率更高
        if(instance == null){

            synchronized (Bank.class) {
                if(instance == null){
                    instance = new Bank();
                }
            }
        }
        return instance;
    }
}
```

解决线程安全问题的方式三：Lock 锁　--- JDK5.0 新增

1. 面试题：synchronized 与 Lock 的异同？
相同：二者都可以解决线程安全问题
不同：synchronized 机制在执行完相应的同步代码以后，自动的释放同步监视器
Lock 需要手动的启动同步（lock()），同时结束同步也需要手动的实现（unlock()）

2.优先使用顺序：
Lock > 同步代码块（已经进入了方法体，分配了相应资源） > 同步方法（在方法体之外）

面试题：如何解决线程安全问题？有几种方式

演示线程的死锁问题

1.死锁的理解：不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁

2.说明：

1）出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续

* 2）我们使用同步时，要避免出现死锁。

线程通信的例子：使用两个线程打印 1-100。线程 1, 线程 2 交替打印

涉及到的三个方法：

wait():一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。

notify():一旦执行此方法，就会唤醒被 wait 的一个线程。如果有多个线程被 wait，就唤醒优先级高的那个。

notifyAll():一旦执行此方法，就会唤醒所有被 wait 的线程。

说明：

1.wait()，notify()，notifyAll()三个方法必须使用在同步代码块或同步方法中。

2.wait()，notify()，notifyAll()三个方法的调用者必须是同步代码块或同步方法中的同步监视器。否则，会出现 IllegalMonitorStateException 异常

3.wait()，notify()，notifyAll()三个方法是定义在 java.lang.Object 类中。

面试题：sleep() 和 wait()的异同？

1.相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。

2.不同点：1）两个方法声明的位置不同：Thread 类中声明 sleep() , Object 类中声明 wait()

2）调用的要求不同：sleep()可以在任何需要的场景下调用。 wait()必须使用在同步代码块或同步方法中

3)关于是否释放同步监视器:如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。

创建线程的方式三：实现 Callable 接口。 --- JDK 5.0 新增

如何理解实现 Callable 接口的方式创建多线程比实现 Runnable 接口创建多线程方式强大？

1. call()可以有返回值的。

2. call()可以抛出异常，被外面的操作捕获，获取异常的信息

3. Callable 是支持泛型的

创建线程的方式四：使用线程池

好处：

1.提高响应速度（减少了创建新线程的时间）

2.降低资源消耗（重复利用线程池中线程，不需要每次都创建）

3.便于线程管理

corePoolSize：核心池的大小

maximumPoolSize：最大线程数

keepAliveTime：线程没有任务时最多保持多长时间后会终止

面试题：创建多线程有几种方式？四种！

Day_3

String 的使用

String:字符串，使用一对""引起来表示。

1.String 声明为 final 的，不可被继承

2.String 实现了 Serializable 接口：表示字符串是支持序列化的。

实现了 Comparable 接口：表示 String 可以比较大小

3.String 内部定义了 final char[] value 用于存储字符串数据

4.String:代表不可变的字符序列。简称：不可变性。

体现：

1.当对字符串重新赋值时，需要重写指定内存区域赋值，不能使用原有的 value 进行赋值。

2. 当对现有的字符串进行连接操作时，也需要重新指定内存区域赋值，不能使用原有的 value 进行赋值。

3. 当调用 String 的 replace()方法修改指定字符或字符串时，也需要重新指定内存区域赋值，不能使用原有的 value 进行赋值。

5.通过字面量的方式（区别于 new）给一个字符串赋值，此时的字符串值声明在字符串常量池中。

6.字符串常量池中是不会存储相同内容的字符串的。

String 的实例化方式：

方式一：通过字面量定义的方式

方式二：通过 new + 构造器的方式

面试题：String s = new String("abc");方式创建对象，在内存中创建了几个对象？

两个:一个是堆空间中 new 结构，另一个是 char[]对应的常量池中的数据："abc"

****

```
public void test3(){
    String s1 = "javaEE";
    String s2 = "hadoop";

    String s3 = "javaEEhadoop";
    String s4 = "javaEE" + "hadoop";
    String s5 = s1 + "hadoop";
    String s6 = "javaEE" + s2;
    String s7 = s1 + s2;

    System.out.println(s3 == s4);//true
    System.out.println(s3 == s5);//false
    System.out.println(s3 == s6);//false
    System.out.println(s3 == s7);//false
    System.out.println(s5 == s6);//false
    System.out.println(s5 == s7);//false
    System.out.println(s6 == s7);//false

    String s8 = s6.intern();//返回值得到的 s8 使用的常量值中已经存在的“javaEEhadoop”
    System.out.println(s3 == s8);//true
```

结论：

1.常量与常量的拼接结果在常量池。且常量池中不会存在相同内容的常量。

2.只要其中有一个是变量，结果就在堆中。

3.如果拼接的结果调用 intern()方法，返回值就在常量池中

涉及到 String 类与其他结构之间的转换

String 与基本数据类型、包装类之间的转换。

String --> 基本数据类型、包装类：调用包装类的静态方法：parseXxx(str)

基本数据类型、包装类 --> String:调用 String 重载的 valueOf(xxx)

String 与 char[]之间的转换

String --> char[]:调用 String 的 toCharArray()

char[] --> String:调用 String 的构造器

String 与 byte[]之间的转换

编码：String --> byte[]:调用 String 的 getBytes()

解码：byte[] --> String:调用 String 的构造器

编码：字符串 -->字节 (看得懂 --->看不懂的二进制数据)

解码：编码的逆过程，字节 --> 字符串 （看不懂的二进制数据 ---> 看得懂）

说明：解码时，要求解码使用的字符集必须与编码时使用的字符集一致，否则会出现乱码。

String 方法

int length()：返回字符串的长度： return value.length

char charAt(int index)： 返回某索引处的字符 return value[index]

boolean isEmpty()：判断是否是空字符串：return value.length == 0

String toLowerCase()：使用默认语言环境，将 String 中的所有字符转换为小写

String toUpperCase()：使用默认语言环境，将 String 中的所有字符转换为大写

String trim()：返回字符串的副本，忽略前导空白和尾部空白

boolean equals(Object obj)：比较字符串的内容是否相同

boolean equalsIgnoreCase(String anotherString)：与 equals 方法类似，忽略大小写

String concat(String str)：将指定字符串连接到此字符串的结尾。 等价于用“+”

int compareTo(String anotherString)：比较两个字符串的大小

String substring(int beginIndex)：返回一个新的字符串，它是此字符串的从 beginIndex 开始截取到最后的一个子字符串。

String substring(int beginIndex, int endIndex) ：返回一个新字符串，它是此字符串从 beginIndex 开始截取到 endIndex(不包含)的一个子字符串。

boolean endsWith(String suffix)：测试此字符串是否以指定的后缀结束

boolean startsWith(String prefix)：测试此字符串是否以指定的前缀开始

boolean startsWith(String prefix, int toffset)：测试此字符串从指定索引开始的子字符串是否以指定前缀开始

boolean contains(CharSequence s)：当且仅当此字符串包含指定的 char 值序列时，返回 true

int indexOf(String str)：返回指定子字符串在此字符串中第一次出现处的索引

int indexOf(String str, int fromIndex)：返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始
int lastIndexOf(String str)：返回指定子字符串在此字符串中最右边出现处的索引
int lastIndexOf(String str, int fromIndex)：返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索

注：indexOf 和 lastIndexOf 方法如果未找到都是返回-1

替换：
String replace(char oldChar, char newChar)：返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
String replace(CharSequence target, CharSequence replacement)：使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。
String replaceAll(String regex, String replacement)：使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
String replaceFirst(String regex, String replacement)：使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。
匹配:
boolean matches(String regex)：告知此字符串是否匹配给定的正则表达式。
切片：
String[] split(String regex)：根据给定正则表达式的匹配拆分此字符串。
String[] split(String regex, int limit)：根据匹配给定的正则表达式来拆分此字符串，最多不超过 limit 个，如果超过了，剩下的全部都放到最后一个元素中。

关于 StringBuffer 和 StringBuilder 的使用
String、StringBuffer、StringBuilder 三者的异同？
String:不可变的字符序列；底层使用 char[]存储
StringBuffer:可变的字符序列；线程安全的，效率低；底层使用 char[]存储
StringBuilder:可变的字符序列；jdk5.0 新增的，线程不安全的，效率高；底层使用 char[]存储
源码分析：
String str = new String();//char[] value = new char[0];
String str1 = new String("abc");//char[] value = new char[]{'a','b','c'};

StringBuffer sb1 = new StringBuffer();//char[] value = new char[16];底层创建了一个长度是 16 的数组。
System.out.println(sb1.length());//
sb1.append('a');//value[0] = 'a';
sb1.append('b');//value[1] = 'b';

StringBuffer sb2 = new StringBuffer("abc");//char[] value = new char["abc".length() + 16];

//问题 1. System.out.println(sb2.length());//3
//问题 2. 扩容问题:如果要添加的数据底层数组盛不下了，那就需要扩容底层的数组。

默认情况下，扩容为原来容量的2倍 +2，同时将原有数组中的元素复制到新的数组中。
指导意义：开发中建议大家使用：StringBuffer(int capacity) 或 StringBuilder(int capacity)

StringBuffer 的常用方法：
StringBuffer append(xxx)：提供了很多的 append()方法，用于进行字符串拼接
StringBuffer delete(int start,int end)：删除指定位置的内容
StringBuffer replace(int start, int end, String str)：把[start,end)位置替换为 str
StringBuffer insert(int offset, xxx)：在指定位置插入 xxx
StringBuffer reverse() ：把当前字符序列逆转
public int indexOf(String str)
public String substring(int start,int end):返回一个从 start 开始到 end 索引结束的左闭右开区间的子字符串
public int length()
public char charAt(int n)
public void setCharAt(int n ,char ch)

总结：
增：append(xxx)
删：delete(int start,int end)
改：setCharAt(int n ,char ch) / replace(int start, int end, String str)
查：charAt(int n)
插：insert(int offset, xxx)
长度：length();
*遍历：for() + charAt() / toString()

对比 String、StringBuffer、StringBuilder 三者的效率：
从高到低排列：StringBuilder > StringBuffer > String

JDK 8 之前日期和时间的 API 测试
java.util.Date 类
|---java.sql.Date 类

1.两个构造器的使用
>构造器一：Date()：创建一个对应当前时间的 Date 对象
>构造器二：创建指定毫秒数的 Date 对象
2.两个方法的使用
>toString():显示当前的年、月、日、时、分、秒
>getTime():获取当前 Date 对象对应的毫秒数。（时间戳）

3. java.sql.Date 对应着数据库中的日期类型的变量
>如何实例化
>如何将 java.util.Date 对象转换为 java.sql.Date 对象

```
public void test2(){
        //构造器一：Date()：创建一个对应当前时间的 Date 对象
        Date date1 = new Date();
        System.out.println(date1.toString());//Sat Feb 16 16:35:31 GMT+08:00 2019
        System.out.println(date1.getTime());//1550306204104

        //构造器二：创建指定毫秒数的 Date 对象
        Date date2 = new Date(155030620410L);
        System.out.println(date2.toString());

        //创建 java.sql.Date 对象
        java.sql.Date date3 = new java.sql.Date(35235325345L);
        System.out.println(date3);//1971-02-13

        //如何将 java.util.Date 对象转换为 java.sql.Date 对象
        //情况一：
//          Date date4 = new java.sql.Date(2343243242323L);
//          java.sql.Date date5 = (java.sql.Date) date4;
        //情况二：
        Date date6 = new Date();
        java.sql.Date date7 = new java.sql.Date(date6.getTime());
```

Day_4

jdk8 之前的日期时间的 API 测试

1. System 类中 currentTimeMillis();
2. java.util.Date 类和子类 java.sql.Date
3. SimpleDateFormat
4. Calendar

SimpleDateFormat 的使用：SimpleDateFormat 对日期 Date 类的格式化和解析

1.两个操作

1.1 格式化：日期 ---> 字符串

1.2 解析： 格式化的逆过程： 字符串 ---> 日期

2.SimpleDateFormat 的实例化

```
@Test
public void test(){
        //实例化 SimpleDateFormat：使用默认的构造器
        SimpleDateFormat sdf = new SimpleDateFormat();

        //格式化： 日期 ---> 字符串
        Date date = new Date();
        System.out.println(date);

        String format = sdf.format(date);
```

```
        System.out.println(format);

        //解析
        String str = "2020/11/5  下午 4:20 ";
        Date date1 = null;
        try {
            date1 = sdf.parse(str);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        System.out.println(date1);

//**************按照指定的方式格式化和解析：调用带参的构造器*********
//        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy.MMMM.dd
GGG hh:mm aaa");
//                    2020.十一月.05 公元 04:33 下午
        SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
        //格式化                        2020-11-05 04:34:35
        String format1 = sdf2.format(date);
        System.out.println(format1);
        //解析:要求字符串必须是符合 SimpleDateFormat 识别的格式（否则会抛异常）
        try {
            Date date2 = sdf2.parse("2020-11-05 04:34:35");
            System.out.println(date2);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
```

Calendar 日历类(抽象类)的使用

注意:

获取月份时：一月是 0，二月是 1，以此类推，12 月是 11

获取星期时：周日是 1，周二是 2 ， 。。。。周六是 7

```
@Test
public void testCalendar(){
//1.实例化
//方式一：创建其子类的对象(GregorianCalendar)的对象
//方式二：调用其静态方法 getInstance()
Calendar calendar = Calendar.getInstance();
System.out.println(calendar.getClass());//class java.util.GregorianCalendar

//2.常用方法
//get()
int day = calendar.get(Calendar.DAY_OF_MONTH);//对当前日期
System.out.println(day);//5
```

```
System.out.println(calendar.get(Calendar.DAY_OF_YEAR));//310

//set()
//calendar 可变
calendar.set(Calendar.DAY_OF_MONTH,22);
day = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(day);//22

//add()
calendar.add(Calendar.DAY_OF_MONTH,3);
day = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(day);//25
calendar.add(Calendar.DAY_OF_MONTH,-3);
day = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(day);//22

//getTime()  日历类  ---> Date
Date date = calendar.getTime();
System.out.println(date);//Sun Nov 22 21:21:39 GMT+08:00 2020

//setTime() Date    --->  日历类
Date date1 = new Date();
calendar.setTime(date1);
day = calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(day);//5
}
```

jdk8 中日期时间 API 的测试

```
public class JDK8DateTimeTest {

    @Test
    public void testDate(){
        //偏移量
        Date date1= new Date(2020 - 1900,9 - 1,8); //Tue Sep 08 00:00:00 GMT+08:00 2020
        System.out.println(date1); //Fri Oct 08 00:00:00 GMT+08:00 3920
    }


    /*
        LocalDate,LocalTime,LocalDateTime  的使用
    说明：LocalDateTime 相较于 LocalDate,LocalTime  使用频率更高
     */

    @Test
```

```
    public void test1(){
        //now():获取当前的日期、时间、日期+时间。
        LocalDate date = LocalDate.now();
        LocalTime time = LocalTime.now();
        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println(date);//2020-11-06
        System.out.println(time);//09:22:38.732068200
        System.out.println(localDateTime);//2020-11-06T09:22:38.732068200

        //of():设置指定的年、月、日、时、分、秒没有偏移量
        LocalDateTime localDateTime1 = LocalDateTime.of(2020, 10, 6, 13, 23, 43);
        System.out.println(localDateTime1);//2020-10-06T13:23:43

        //getXxx 2020-11-6 9:32
        System.out.println(localDateTime.getDayOfMonth());//6
        System.out.println(localDateTime.getDayOfWeek());//FRIDAY
        System.out.println(localDateTime.getMonth());//NOVEMBER
        System.out.println(localDateTime.getMonthValue());//11
        System.out.println(localDateTime.getMinute());//32

        //体现不可变性
        //withXxx：设置相关的属性
        LocalDateTime localDateTime2 = localDateTime.withDayOfMonth(22);//有返回值
        System.out.println(localDateTime2);//2020-11-22T09:34:46.426532800
        System.out.println(localDateTime);//2020-11-06T09:34:46.426532800
        LocalDateTime localDateTime3 = localDateTime.withHour(4);
        System.out.println(localDateTime3);//2020-11-06T04:36:27.147355800
        System.out.println(localDateTime);//2020-11-06T09:36:27.147355800

        //不可变性
        //plusXxx,minusXxx：加减时间
        LocalDateTime localDateTime4 = localDateTime.plusMonths(3);
        System.out.println(localDateTime4);//2021-02-06T09:38:15.183859500
        LocalDateTime localDateTime5 = localDateTime.minusDays(6);
        System.out.println(localDateTime5);//2020-10-31T09:39:51.500684700
    }
}
    /*
        Instant 的使用
        类似于 java.util.Date 类
     */
    @Test
    public void test2(){
        //now():获取本初子午线对应的标准时间
```

```
        Instant instant = Instant.now();
        System.out.println(instant); //2020-11-06T01:50:19.559106600Z
        //添加时间的偏移量
        OffsetDateTime offsetDateTime = instant.atOffset(ZoneOffset.ofHours(8));
        System.out.println(offsetDateTime);//2020-11-06T09:50:19.559106600+08:00
        //获取自 1970 年 1 月 1 日 0 时 0 分 0 秒（UTC）开始的毫秒数
        long milli = instant.toEpochMilli();
        System.out.println(milli);//1604627588988
        //通过给定的毫秒数获取 Instance 的实例    -->Date(Long millis)
        Instant instant1 = Instant.ofEpochMilli(1604627588988L);
        System.out.println(instant1);//2020-11-06T01:53:08.988Z
    }

    /*
        DateTimeFormatter：格式化或解析日期、时间
        类似于 SimpleDateFormat
     */
    @Test
    public void test3(){
    //方式一：预定义的标准格式。如：
ISO_LOCAL_DATE_TIME;ISO_LOCAL_DATE;ISO_LOCAL_TIME
        DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
            //格式化：日期 --> 字符串  format()
        LocalDateTime localDateTime = LocalDateTime.now();
        String str1 = formatter.format(localDateTime);
        System.out.println(localDateTime); //2020-11-06T10:09:40.534201900
        System.out.println(str1);//2020-11-06T10:09:40.5342019
            //解析：字符串 --> 日期  parse()
        TemporalAccessor parse = formatter.parse("2020-11-06T10:02:27.7492053");
        System.out.println(parse);//{},ISO resolved to 2020-11-06T10:02:27.749205300

//方式二：本地化相关的格式。如：ofLocalizedDateTime(FormatStyle.LONG)FormatStyle.LONG
FormatStyle.MEDIUM FormatStyle.SHORT:适用于 LocalDateTime

DateTimeFormatter formatter1 = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);
        String str2 = formatter1.format(localDateTime);
        System.out.println(str2);//2020/11/6 上午 10:09

FormatStyle.FULL FormatStyle.LONG FormatStyle.MEDIUM FormatStyle.SHORT: 适 用 于
LocalDate

DateTimeFormatter formatter2 = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
        LocalDate now = LocalDate.now();
            //格式化
```

```
        String str3 = formatter2.format(now);
        System.out.println(str3);//2020 年 11 月 6 日星期五

        //方式三：自定义的格式。如：ofPattern(“yyyy-MM-dd hh:mm:ss”)
DateTimeFormatter formatter3 = DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss");
        //格式化
        String str4 = formatter3.format(localDateTime);
        System.out.println(str4);//2020-11-06 10:15:49
        //解析
        TemporalAccessor parse1 = formatter3.parse("2020-11-06 10:15:49");
        System.out.println(parse1);          //{SecondOfMinute=49,MilliOfSecond=0,
MinuteOfHour=15, NanoOfSecond=0, HourOfAmPm=10, MicroOfSecond=0},ISO resolved to
2020-11-06
    }
}

一、说明：java 中的对象，正常情况下，只能进行比较： == 或 ！=，不能使用 > 或 < 的
 *       但是在开发的常见中，需要对多个对象进行排序，言外之意，需要比较对象的大小。
 *       如何实现？使用两个接口中的任何一个：Comparable 和 Comparator
 *
 * 二、Comparable 接口与 Comparactor 的使用的对比
 *    Comparable 接口：先继承 Comparable 接口，再重写 compareTo()方法
 *    Comparactor：先创建 Comparactor 对象，再重写重写 compare(Object o1,Object o2)方法
 * Comparable 接口的方式一旦一定，保证 Comparable 接口实现类的对象在任何位置可以比
较大小
 *    Comparactor 接口属于临时性的比较

public class CompareTest {
    /* Comparable 接口的使用举例  ：自然排序
        1.像 String、包装类等实现了 Comparable 接口，重写了 compareTo()方法，给出了
比较两个对象大小的方法
        2.像 String、包装类重写 compareTo()方法以后，进行了从小到大的排列
        3.重写 compareTo()的规则：
          如果当前对象 this 大于形参对象 obj，则返回正整数，
          如果当前对象 this 小于形参对象 obj，则返回负整数，
          如果当前对象 this 等于形参对象 obj，则返回零。
        4.对于自定义类老说，如果需要排序，我们可以让自定义类实现 Comparable 接口，
重写 compareTo()方法，
            在 compareTo()指明如何排序

    @Test
    public void test1(){
        String[] arr = new String[]{"AA","CC","MM","KK","GG","JJ","DD"};
        Arrays.sort(arr);
```

```
        System.out.println(Arrays.toString(arr));
    }

    @Test
    public void test2(){
        Goods[] arr = new Goods[5];
        arr[0] = new Goods("lenovoMouse",34);
        arr[1] = new Goods("dellMouse",43);
        arr[2] = new Goods("xiaomiMouse",11);
        arr[3] = new Goods("huaweiMouse",65);
        arr[4] = new Goods("microsoftMouse",43);
        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr));
        //[Goods{name='xiaomiMouse', price=11.0}, Goods{name='lenovoMouse', price=34.0},
        // Goods{name='dellMouse', price=43.0}, Goods{name='microsoftMouse',price=43.0},
        // Goods{name='huaweiMouse', price=65.0}]
    }
    /*
    Comparator 接口的使用：定制排序
    1.背景：
      当元素的类型没有实现 java.lang.Comparable 接口而又不方便修改代码，或者实现了
java.lang.Comparable 接口的排序规则不适合当前的操作，那么可以考虑使用 Comparator 的
对象来排序
    2.重写 compare(Object o1,Object o2)方法，比较 o1 和 o2 的大小：
      如果方法返回正整数，则表示 o1 大于 o2；
      如果返回 0，表示相等；
      返回负整数，表示 o1 小于 o2。
     */
    @Test
    public void test3(){
        String[] arr = new String[]{"AA","CC","MM","KK","GG","JJ","DD"};
        Arrays.sort(arr,new Comparator(){
            public int compare(Object o1, Object o2) {
                if(o1 instanceof String && o2 instanceof String){
                    String s1 = (String) o1;
                    String s2 = (String) o2;
                    return -s1.compareTo(s2);
                }
                throw new RuntimeException("输入的数据类型不一致！");
            }
        });
        System.out.println(Arrays.toString(arr));
    }
```

```
    @Test
    public void test4(){
        Goods[] arr = new Goods[5];
        arr[0] = new Goods("lenovoMouse",34);
        arr[1] = new Goods("dellMouse",43);
        arr[2] = new Goods("xiaomiMouse",11);
        arr[3] = new Goods("huaweiMouse",65);
        arr[4] = new Goods("microsoftMouse",43);
        Arrays.sort(arr,new Comparator(){
            //按照产品名称从低到高排序，再按照价格从高到低排序 (小的先放 大的后放)
             public int compare(Object o1, Object o2) {
                if(o1 instanceof Goods && o2 instanceof Goods){
                    Goods goods1 = (Goods) o1;
                    Goods goods2 = (Goods) o2;
                    if(goods1.getName().equals(goods2.getName())){
                        return -Double.compare(goods1.getPrice(),goods2.getPrice());
                    }else{
                        return goods1.getName().compareTo(goods2.getName());
                    }
                }
                throw new RuntimeException("输入的数据类型不一致！");
            }
        });
        System.out.println(Arrays.toString(arr));
        //[Goods{name='dellMouse', price=43.0}, Goods{name='huaweiMouse', price=65.0},
        // Goods{name='lenovoMouse', price=34.0},
        //     Goods{name='microsoftMouse',     price=43.0},     Goods{name='xiaomiMouse',
price=11.0}]
    }
}
```

其他常用类的使用

```
 * 1.System
 *   native long currentTimeMillis()：该方法的作用是返回当前的计算机时间，
 *   void exit(int status)：该方法的作用是退出程序。
 *   void gc()：该方法的作用是请求系统进行垃圾回收。
 *   String getProperty(String key)：该方法的作用是获得系统中属性名为 key 的属性对应的
值。

 * 2.Math
 *   abs 绝对值
 *   acos,asin,atan,cos,sin,tan 三角函数
 *   sqrt 平方根
 *   pow(double a,doble b) a 的 b 次幂
```

```
*    log 自然对数
*    exp e 为底指数
*    max(double a,double b)
*    min(double a,double b)
*    random() 返回 0.0 到 1.0 的随机数
*    long round(double a) double 型数据 a 转换为 long 型（四舍五入）
*    toDegrees(double angrad) 弧度—>角度
*    toRadians(double angdeg) 角度—>弧度
*
* 3.BigInteger 和 BigDecimal
*    BigInteger 可以表示不可变的任意精度的整数
*    构造器：BigInteger(String val)：根据字符串构建 BigInteger 对象
*    BigDecimal 类支持不可变的、任意精度的有符号十进制定点数。
*    构造器：public BigDecimal(double val)、public BigDecimal(String val)
```

Day_5

一、枚举类的使用

```
*    1. 枚举类的理解：类的对象只有有限个，确定的。我们称此类为枚举类。
*    2. 当需要定义一组常量时，强烈建议使用枚举类
*    3. 如果枚举类中只有一个对象，则可以作为单例模式的实现方式。
```

*** 二、如何定义枚举类**

*** 方式一**：jdk5.0 之前，自定义枚举类

```
//自定义枚举类
class Season{
    //声明 Season 对象的属性:private final 修饰
    private final String seasonName;
    private final String seasonDesc;
    //2.私有化类的构造器,并给对象属性赋值
    private Season(String seasonName,String seasonDesc){
        this.seasonDesc = seasonDesc;
        this.seasonName = seasonName;
    }
    //3.提供当前枚举类的多个对象：public static final 的
    public static final Season SPRING = new Season("春天","春暖花开");
    public static final Season SUMMER = new Season("夏天","夏日炎炎");
    public static final Season AUTUMN = new Season("球天","秋高气爽");
    public static final Season WINTER = new Season("冬天","冰天雪地");
    //4.其他诉求 1：获取枚举类对象的属性
    public String getSeasonName() {
        return seasonName;
    }
    public String getSeasonDesc() {
        return seasonDesc;
```

```
    }
    //4.其他诉求 2：提供 toString()
    public String toString() {
        return seasonName + "到了,到处" + seasonDesc ;
    }
}
 * 方式二：jdk5.0 时，可以使用 enum 关键字定义枚举类
public class SeasonTest1 {
    public static void main(String[] args) {
        System.out.println(Season1.SPRING);//SPRING
        System.out.println(Season1.SUMMER);//SUMMER
        System.out.println(Season1.AUTUMN);//AUTUMN
        Season1 winter = Season1.WINTER;
        System.out.println(winter);//WINTER
        winter.show();//这是一个季节！---> 大约在冬季
    }
}
interface Info{
    void show();
}
enum Season1 implements Info{
    //1.提供当前枚举类的对象，多个对象之间用","隔开，末尾对象用";"结尾。
    SPRING("春天","春暖花开"){
        public void show() {
            System.out.println("春天在哪里？");
        }
    },
    SUMMER("夏天","夏日炎炎"){
        public void show() {
            System.out.println("宁夏");
        }
    },
    AUTUMN("秋天","秋高气爽"){
        public void show() {
            System.out.println("秋天不回来");
        }
    },
    WINTER("冬天","冰天雪地"){
        public void show() {
            System.out.println("大约在冬季");
        }
    };
    //2.声明 Season 对象的属性：private final 修饰
    private final String seasonName;
```

```
    private final String seasonDesc;
    //3.私有化类的构造器,并给对象属性赋值
    private Season1(String seasonName, String seasonDesc){
        this.seasonDesc = seasonDesc;
        this.seasonName = seasonName;
    }
    //4.其他诉求 1：获取枚举类对象的属性
    public String getSeasonName() {
        return seasonName;
    }
    public String getSeasonDesc() {
        return seasonDesc;
    }
//  public void show() {
//      System.out.println("这是一个季节！");
//  }
//  //4.其他诉求 2：提供 toString()
//  public String toString() {
//      return seasonName + "到了,到处" + seasonDesc ;
//  }
}
```

*** 三、Enum 类中的常用方法**

* values()方法：返回枚举类型的对象数组。该方法可以很方便地遍历所有的枚举值。

* valueOf(String str)：可以把一个字符串转为对应的枚举类对象。要求字符串必须是枚举类对象的“名字”。如不是，会有运行时异常：IllegalArgumentException。

* toString()：返回当前枚举类对象常量的名称

```
public class SeasonTest {
    public static void main(String[] args) {
        System.out.println(Season1.SPRING);//春天到了,到处春暖花开
        System.out.println(Season1.SUMMER);//夏天到了,到处夏日炎炎
        System.out.println(Season1.AUTUMN);//球天到了,到处秋高气爽
        //toString():返回当前枚举类对象常量的名称
        System.out.println(Season1.WINTER.toString());//冬天到了,到处冰天雪地

        //values():返回枚举类型的对象数组
        Season1[] values = Season1.values();
        for (int i = 0; i < values.length; i++) {
            System.out.println(values[i]);
        }//SPRING SUMMER AUTUMN WINTER

        //valueOf(String str)：返回枚举类中对象名是 objName 的”对象“
        Season1 winter = Season1.valueOf("WINTER");//winter 是一个对象
        //没有则抛异常：IllegalArgumentException。
//      Season1 winter1 = Season1.valueOf("WINTER1");
```

```
        System.out.println(winter);//WINTER
    }
}
```

*** 四、使用 enum 关键字定义的枚举类实现接口的情况**

*　　情况一：实现接口，在 enum 类中实现抽象方法

*　　情况二：让枚举类的对象分别实现接口中的抽象方法，使得每个对象的方法都不同

注解的使用

*　1. 理解 Annotation：

*　① jdk5.0 新增的功能

*　② Annotation 其实就是代码里的特殊标记, 这些标记可以在编译, 类加载, 运行时被读取, 并执行相应的处理。通过使用 Annotation, 程序员可以在不改变原有逻辑的情况下, 在源文件中嵌入一些补充信息。

*　③ 在 JavaSE 中，注解的使用目的比较简单，例如标记过时的功能，忽略警告等。在 JavaEE/Android 中注解占据了更重要的角色，例如用来配置应用程序的任何切面，代替 JavaEE 旧版中所遗留的繁冗代码和 XML 配置等。

*　2. Annotation 的使用示例

*　示例一：生成文档相关的注解

*　示例二：在编译时进行格式检查(JDK 内置的三个基本注解)

*　　@Override: 限定重写父类方法, 该注解只能用于方法

*　　@Deprecated: 用于表示所修饰的元素(类, 方法等)已过时。通常是因为所修饰的结构危险或存在更好的选择

*　　@SuppressWarnings: 抑制编译器警告

*　示例三：跟踪代码依赖性，实现替代配置文件功能

*

*　3. 如何自定义注解：参照@SuppressWarnings 定义

*　　① 注解声明为：@interface

*　　② 内部定义成员，通常使用 value 表示

*　　③ 可以指定成员的默认值，使用 default 定义

*　　④ 如果自定义注解没有成员，表明是一个标识作用

```
\*              public @interface MyAnnotation {
\*                  String value() default "hello";
\*              }
```

*　　如果注解有成员，在使用注解时，需要指明成员的值。

*　　自定义注解必须配上注解的信息处理流程(使用反射)才有意义。

*　　自定义注解一般都会指明两个元注解：Retention、Target

*

*　4.jdk 提供的 4 种元注解

*　　元注解：对现有的注解进行解释说明的注解

*　　Retention:指定该 Annotation 的生命周期:SOURCE\CLASS(默认)\RUNTIME 只有声明为 RUNTIME 声明周期的注解，才能通过反射获取。

*　　Target: 用于指定被修饰的 Annotation 能用于修饰哪些程序元素

*　　*******以下出现频率低**********

```
 *            Documented：表示所修饰的注解在被 javadoc 解析时，保留下来
 *            Inherited：被它修饰的 Annotation 将具有继承性。
 *
 *   5.通过反射获取注解信息  --->到反射内容时系统讲解
 *
 *   6.jdk 8 中注解的新特性：可重复注解、类型注解
 *      6.1 可重复注解:
       ① 在 MyAnnotation 上声明一个 Repeatable，成员值为 MyAnnotations.class
 *     ② MyAnnotation 的 Target 和 Retention 等元注解与 MyAnnotations 相同。(注解要相同)
 *      6.2 类型注解：
 *          ElementType.TYPE_PARAMETER 表示该注解能写在类型变量的声明语句中（如：泛型声明）。
 *          ElementType.TYPE_USE 表示该注解能写在使用类型的任何语句中
 ----------------------------------------------------------------------------------------
@Repeatable(MyAnnotations.class)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE,TYPE_PARAMETER,TYPE_USE})
public @interface MyAnnotation {
    String value() default "hello";
}

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE})
public @interface MyAnnotations {
    MyAnnotation[] value();
}

public class AnnotationTest {
    public static void main(String[] args) {
        Person p = new Student();
        p.walk();
    }
    @Test   //使用反射进行获取、调用来获取注解信息
    public void testGetAnnotation(){
        Class clazz = Student.class;
        Annotation[] annotations = clazz.getAnnotations();
        for (int i = 0; i < annotations.length; i++) {
            System.out.println(annotations[i]);
        }
    }
```

```
}
//jdk 8 之前的写法：
//@MyAnnotations({@MyAnnotation(value = "hi"),@MyAnnotation(value = "abc")})
@MyAnnotation(value = "hi")
@MyAnnotation(value = "abc")
class Person{
	private String name;
	private int age;

	public Person() {
	}
	@SuppressWarnings("unused")
	public Person(String name, int age) {
		this.name = name;
		this.age = age;
	}
	public void walk(){
		System.out.println("人走路");
	}
	@SuppressWarnings("unused")
	public void eat(){
		System.out.println("人吃饭");
	}
}

interface Info1{
	@SuppressWarnings("unused")
	void show();
}

class Student extends Person implements Info1{
	@Override
	public void walk() {
		System.out.println("学生走路");
	}

	@Override
	public void show() {

	}
}
class Generic<@MyAnnotation T>{
	public void show() throws @MyAnnotation RuntimeException{
		ArrayList<@MyAnnotation String> list = new ArrayList<>();
```

```
        int num = (@MyAnnotation int) 10L;
    }
}
```

一、集合框架的概述

```
 *    1.集合、数组都是对多个数据进行存储操作的结构，简称 Java 容器
 *       说明：此时的存储，主要指的是内存层面的存储，不涉及到持久化的存储(.txt,.jpg,.avi,数据库中)
 *
 *    2.1 数组在存储多个数据方面的特点：
 *       > 一旦初始化以后，其长度就确定了。
 *       > 数组一旦定义好，其元素的类型也就确定了。我们也就只能操作指定类型的数据。比如：String[] arr；int[] arr1；Object[] arr2 多态保证对象元素可以不同
 *    2.2 数组在存储多个数据方面的缺点：
 *       > 一旦初始化后，其长度就不可修改。
 *       > 数组中提供的方法非常有限，对于添加、删除、插入数据等操作，非常不便。同时效率不高。
 *       > 获取数组中实际元素的个数的需求，数组没有现成的属性或方法可用。
 *       > 数组存储数据的特点：有序、可重复。对于无序、不可重复的需求，不能满足。
 *
 * 二、集合框架
 *       |---Collection 接口：单列集合，用来存储一个一个的对象
 *            |---List 接口：存储有序的，可重复的数据。    ---> ”动态“数组
 *                 |---ArrayList、LinkedList、Vector
 *
 *            |---Set 接口：存储无序的，不可重复的数据     ---> 高中数学中的"集合”
 *                 |---HashSet、LinkedHashSet、TreeSet
 *
 *             |---Map 接口：双列集合，用来存储一对(key-value)一对的数据   --->高中数学中的"函数"
 *                 |---HashMap、LinkedHashMap、TreeMap、Hashtable、Properties
 *
 * 三、Collection 接口中的方法的使用
 *
public class CollectionTest {

    @Test
    public void test1(){
        Collection coll = new ArrayList();
        //add(Object e):将元素 e 添加到集合 coll 中
        coll.add("AA");
        coll.add("BB");
        coll.add(123);//自动装箱
        coll.add(new Date());
```

```
        //size():获取添加的元素的个数
        System.out.println(coll.size());//4
        //addAll():将 coll1 集合中的元素添加到当前的集合中
        Collection coll1 = new ArrayList();
        coll1.add(456);
        coll1.add("CC");
        coll.addAll(coll1);
        System.out.println(coll.size());//6
        System.out.println(coll);//[AA, BB, 123, Sat Nov 07 11:13:55 GMT+08:00 2020, 456,
CC]
        //clear():清空集合元素
        coll.clear();
        //isEmpty():判断当前集合是否为空
        System.out.println(coll.isEmpty());//false --> true
    }
}
```

Day_6

Collection 接口中声明的方法的测试

```
 *  结论：判断身份都是 equals
 *  向 Collection 接口的实现类的对象中添加数据 obj 时，要求 obj 所在类要重写 equals()
public class CollectionTest {
    @Test
    public void test1(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("Tom"));
        coll.add(false);
        coll.add(new Person("Jerry",20));
//      Person p = new Person("Jerry",20)
//      coll.add(p);
        //1.contains(Object obj):当前集合中是否包含 obj
    我们判断时会去调用 obj 对象所在类的 equals 方法，要求 obj 所在类要重写 equals 方法
        boolean contains = coll.contains(123);
        System.out.println(contains);//true
        System.out.println(coll.contains(new String("Tom")));//true
    System.out.println(coll.contains(new Person("Jerry", 20)));//false---> true(重写 equals 方法)

   //2.containsAll(Collection coll1):判断形参 coll1 中的所有元素是否都存在于当前集合中
        Collection coll1 = Arrays.asList(123,456);
        System.out.println(coll.containsAll(coll1));//true
    }
    @Test
```

```
public void test2(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new Person("Jerry",20));
        coll.add(new String("Tom"));
        coll.add(false);

        //3.remove(Object obj):从当前集合中移除 obj 元素
        coll.remove(123);//返回 boolean
        System.out.println(coll);//[456, Person{name='Jerry', age=20}, Tom, false]
        coll.remove(new Person("Jerry",20));
        System.out.println(coll);//[456, Tom, false]

    //4.removeAll(Collection coll1):差集：从当前集合中移除 coll1 中所有的元素(交集元素)
        Collection coll1 = Arrays.asList(123,456);
        coll.removeAll(coll1);
        System.out.println(coll);//[Tom, false]
    }
    @Test
    public void test3(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new Person("Jerry",20));
        coll.add(new String("Tom"));
        coll.add(false);

//5.retainAll(Collection coll1)：交集，获取当前集合和 coll1 集合的交集，并返回给当前集合
        Collection coll1 = Arrays.asList(123,456,789);
        coll.retainAll(coll1);
        System.out.println(coll);//[123, 456]

        //6.equals(Object obj):要想返回 true，需要当前集合和形参集合的元素都相同
        Collection coll2 = new ArrayList();
        coll2.add(456); //交换后  true -> false
        coll2.add(123);
        coll2.add(new Person("Jerry",20));
        coll2.add(new String("Tom"));
        coll2.add(false);
        System.out.println(coll.equals(coll2));//true
    }
    @Test
    public void test4(){
```

```
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new Person("Jerry",20));
        coll.add(new String("Tom"));
        coll.add(false);

        //7.hashCode():返回当前对象的哈希值
        System.out.println(coll.hashCode());//1177854711

        //8.集合 ---> 数组 ：toArray()
        Object[] arr = coll.toArray();
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);//123.456.Person{name='Jerry', age=20}.Tom.false
        }
        //拓展：数组 ---> 集合 :调用 Arrays 类的静态方法 asList()
        List<String> list = Arrays.asList(new String[]{"AA", "BB", "CC"});
        System.out.println(list);//[AA, BB, CC]
        List<int[]> arr1 = Arrays.asList(new int[]{123, 456});
        System.out.println(arr1);//[[I@11531931]
        List arr2 = Arrays.asList(123, 456);
        System.out.println(arr2);//[123,456]
        List arr3 = Arrays.asList(new Integer[]{123, 456});
        System.out.println(arr3);//[123,456]

//9.iterator()：返回 Iterator 接口的实例，用于遍历集合元素。放在 InteratorTest.java 中测试
    }
}
```

* **集合元素的遍历操作**：使用迭代器 Iterator 接口
 * 1.内部的方法：hasNext() 和 next()
 * 2.集合对象每次调用 iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前。
 * 3.内部定义了 remove()方法，可以在遍历的时候，删除集合中的元素。此方法不同于集合直接调用 remove()
 *
 * @author skoud
 * @create 2020-11-07-15:20
 */

```
public class IteratorTest {
    @Test
    public void test() {
        Collection coll = new ArrayList();
        coll.add(123);
```

```
        coll.add(456);
        coll.add(new Person("Jerry", 20));
        coll.add(new String("Tom"));
        coll.add(false);
        Iterator iterator = coll.iterator();
        //方式一：
//      System.out.println(iterator.next());
//      System.out.println(iterator.next());
//      System.out.println(iterator.next());
//      System.out.println(iterator.next());
//      System.out.println(iterator.next());
        //报异常
//      System.out.println(iterator.next());
        //方式二:不推荐
//      for (int i = 0; i < coll.size(); i++) {
//          System.out.println(iterator.next());
//      }
        //方式三：推荐
        while (iterator.hasNext()) {
System.out.println(iterator.next());//123、456、Person{name='Jerry', age=20}、Tom、false
        }
    }
    @Test
    public void test2(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new Person("Jerry", 20));
        coll.add(new String("Tom"));
        coll.add(false);
        //错误方式一：
//      Iterator iterator = coll.iterator();
//      while (iterator.next() != null){
//          System.out.println(iterator.next());//456 Tom 异常 跳着输出
//      }
        //错误方式二：
        //集合对象每次调用 iterator()方法都得到一个全新的迭代器对象，默认游标都在集合的第一个元素之前。
        while ((coll.iterator().hasNext())){
            System.out.println(coll.iterator().next());//123 死循环
        }
    }
    @Test
    public void test3(){
```

```
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new Person("Jerry", 20));
        coll.add(new String("Tom"));
        coll.add(false);
        //删除集合中"Tom"
        Iterator iterator = coll.iterator();
        while(iterator.hasNext()){
            Object obj = iterator.next();
            if("Tom".equals(obj)){
                iterator.remove();
            }
        }
        iterator = coll.iterator();//重新调用
        while(iterator.hasNext()){
            System.out.println(iterator.next()); //123、456、Person{name='Jerry', age=20}、false
        }
    }
}
```

jdk5.0 新增了 foreach 循环，用于遍历集合、数组

```
public class ForTest {
    @Test
    public void test1(){
        Collection coll = new ArrayList();
        coll.add(123);
        coll.add(456);
        coll.add(new String("Tom"));
        coll.add(new Person("Jerry",20));
        coll.add(false);

        //for(集合中元素类型 局部变量 : 集合对象)
        for(Object obj : coll){
            System.out.println(obj);//123、456、Tom、Person{name='Jerry', age=20}、false
        }
    }
    @Test
    public void test2(){
        int[] arr = new int[]{1,2,3,4,5,6};
        //for(集合元素的类型 局部变量 :  数组对象)
        for(int i : arr){
            System.out.println(i); //1、2、3、4、5、6
        }
```

```
    }
    @Test
    public void test3() {
        String[] arr = new String[]{"MM","MM","MM"};
        //方式一：普通 for 赋值
//        for (int i = 0; i < arr.length; i++) {
//            arr[i] = "GG";
//        }
        //方式二：增强 for 循环
        for(String s : arr){
            s = "GG";
        }
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]); //MM、MM、MM
        }
    }
}
```

1.list 接口框架

```
 *   |---Collection 接口：单列集合，用来存储一个一个的对象
 *      |---List 接口：存储有序的，可重复的数据。  ---> ”动态“数组，替换原有的数组
 *           |---ArrayList：作为 List 接口的主要实现类，线程不安全的，效率高；底层使用 Object[] elementData 存储。
 *           |---LinkedList：对于频繁的插入、删除操作使用此类比 ArrayList 高 ；底层使用双向链表存储
 *           |---Vector：作为 List 接口的古老实现类，线程安全的，效率低；底层使用 Object[] 存储。
 *
 *   2.ArrayList 的源码分析：
 *      2.1 jdk7  的情况下
 *          ArrayList list = new ArrayList(); //底层创建了长度是 10 的 Object[]数组
 *          list.add(123);//elementData[0] = new Integer(123);
 *          ...
 *          list.add(11);//如果此次的添加导致底层 elementData 数组容量不够，则扩容。
 *          默认情况下，扩容为原来容量的 1.5 倍，同时需要将原有数组中的数据复制到新的数组中。

 *          结论：建议开发中使用带参的构造器：ArrayList list = new ArrayList(int capacity);
 *
 *      2.2 jdk 8 中 ArrayList 的变化：
 *          ArrayList list = new ArrayList();//底层 Object[] elementData 初始化为{},并没有创建长度为 10 的数组
 *          list.add(123);//第一次调用 add()时，底层才创建了长度为 10 的数组，并将数据 123 添加到 elementData 中
```

```
*          。。。
*          后续的添加和扩容操作与 jdk7 无异。
*
*      2.3 小结：jdk7 中的 ArrayList 的创建类似于单例的饿汉式，而 jdk8 中的 ArrayList 的对象的创建类似于单例的懒汉式延迟了数组的创建，节省内存。
*
*   3.LinkedList 的源码分析：
*         LinkedList list = new LinkedList();//内部声明了 Node 类型的 first 和 last 属性，默认值为 null
*         list.add(123)；//将 123 封装到 Node 中，创建了 Node 对象
*
*         其中，Node 定义为：体现了 LinkedList 的双向链表的说法
*          private static class Node<E> {
*             E item;
*             Node<E> next;
*             Node<E> prev;
*
*             Node(Node<E> prev, E element, Node<E> next) {
*                 this.item = element;
*                 this.next = next;
*                 this.prev = prev;
*             }
*       }
*   4. Vector 的源码分析：jdk7 和 jdk8 中通过 Vector()构造器创建对象时，底层都创建了长度为 10 的数组，在扩容方面，默认扩容为原来数组长度的 2 倍。
*
*   面试题：ArrayList、LinkedList、Vector 三者的异同？
*   同：三个类都实现了 List 接口，存储数据的特点相同：存储有序的，可重复的数据。
*   不同：见上
*
*   5. List 接口中的常用方法

public class ListTest {
    /*
       void add(int index, Object ele):在 index 位置插入 ele 元素
       boolean addAll(int index, Collection eles):从 index 位置开始将 eles 中的所有元素添加进来
       Object get(int index):获取指定 index 位置的元素
       int indexOf(Object obj):返回 obj 在集合中首次出现的位置
       int lastIndexOf(Object obj):返回 obj 在当前集合中末次出现的位置
       Object remove(int index):移除指定 index 位置的元素，并返回此元素
       Object set(int index, Object ele):设置指定 index 位置的元素为 ele
       List subList(int fromIndex, int toIndex):返回从 fromIndex 到 toIndex 位置的子集合
```

```
总结：常用方法
增：add(Object obj)
删：remove(Object obj)、remove(int index)
改：set(int index,Object ele)
查：get(int index)
插：add(int index,Object ele)
长度：size()
遍历：① Iterator 迭代器方式
      ② 增强 for 循环
      ③ 普通的循环
 */
@Test
public void test2(){
    ArrayList list = new ArrayList();
    list.add(123);
    list.add(456);
    list.add("AA");

    //方式一：terator 迭代器方式
    Iterator iterator = list.iterator();
    while (iterator.hasNext()){
        System.out.println(iterator.next());
    }
    //方式二：增强 for 循环
    for(Object obj : list){
        System.out.println(obj);
    }
    //方式三：普通的循环
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
@Test
public void test1(){
    ArrayList list = new ArrayList();
    list.add(123);
    list.add(456);
    list.add("AA");
    list.add(new Person("Tom",12));
    list.add(456);
    System.out.println(list);//[123, 456, AA, Person{name='Tom', age=12}, 456]

    //void add(int index, Object ele):在 index 位置插入 ele 元素
    list.add(1,"BB");
```

```
        System.out.println(list);//[123, BB, 456, AA, Person{name='Tom', age=12}, 456]
        //boolean addAll(int index, Collection eles):从 index 位置开始将 eles 中的所有元素添加进来
        List list1 = Arrays.asList(1, 2, 3);
        list.addAll(list1);
        System.out.println(list.size());//9
        //Object get(int index):获取指定 index 位置的元素
        System.out.println(list.get(1));//BB
        //int indexOf(Object obj):返回 obj 在集合中首次出现的位置
        System.out.println(list.indexOf(456));//2  没有则返回-1
        //int lastIndexOf(Object obj):返回 obj 在当前集合中末次出现的位置
        System.out.println(list.lastIndexOf(456));//5
        //Object remove(int index):移除指定 index 位置的元素，并返回此元素  可以按 Obj 删
        Object obj = list.remove(0);
        System.out.println(obj);//123
        System.out.println(list);//[BB, 456, AA, Person{name='Tom', age=12}, 456, 1, 2, 3]
        //Object set(int index, Object ele):设置指定 index 位置的元素为 ele
        list.set(1,"CC");
        System.out.println(list);//[BB, CC, AA, Person{name='Tom', age=12}, 456, 1, 2, 3]
        //List subList(int fromIndex, int toIndex):返回从 fromIndex 到 toIndex 位置的左闭右开子集合
        System.out.println(list.subList(3, 5));//[Person{name='Tom', age=12}, 456]
        System.out.println(list);//没有变化
    }
}
```

*** 1.Set 接口的框架：**

```
*    |---Collection 接口：单列集合，用来存储一个一个的对象
*         |---Set 接口：存储无序的，不可重复的数据     ---> 高中数学中的"集合”
*              |---HashSet：作为 Set 接口的主要实现类；线程不安全的；可以存储 null 值
*                   |---LinkedHashSet：作为 HashSet 的子类，遍历其内部数据时，可以按照添加的顺序遍历，对于频繁的遍历操作，LinkedHashSet 效率高于 HashSet
*              |---TreeSet：可以按照添加对象的指定属性，进行排序。
*
*   1.Set 接口中没有额外定义新的方法，使用的都是 Collection 中声明的方法。
*   2.要求：向 Set（主要指：HashSet、LinkedHashSet）中添加的数据，其所在的类一定要重写 hashCode()和 equals()。
*     要求：重写的 hashCode()和 equals()尽可能保持一致性：相等的对象必须具有相等的散列码
*      重写两个方法的小技巧：对象中用作 equals()方法比较的 Field，都应该用来计算 hashCode 值。
```

```
public class SetTest {
```

/*

一、Set：储存无序的、不可重复的数据

以 HashSet 为例说明：

1.无序性：不等于随机性，存储的数据在底层数组中并非按照数组索引的顺序添加，而是根据数据的哈希值确定的

2.不可重复性：保证添加的元素按照 equals()判断时，不能返回 true。即：相同的元素只能添加一个。

二、添加元素的过程：以 HashSet 为例：

我们向 HashSet 中添加元素 a，首先调用元素 a 所在类的 hasCode()方法，计算元素 a 的哈希值，此哈希值接着通过某种算法计算出在 HashSet 底层数组中的存放位置(即为：索引位置)，判断数组此位置上是否已经有元素：

如果此位置上没有其他元素，则元素 a 添加成功。 --->情况 1

如果此位置上有其他元素 b(或者以链表存在的多个元素)，则比较元素 a 与元素 b 的哈希值：

如果 hash 值不相同，则元素 a 添加成功。 --->情况 2

如果 hash 值相同，进而需要调用元素 a 所在类的 equals()方法：

如果 equals()返回 true，元素 a 添加失败

如果 equals()返回 false，元素 a 添加成功。 --->情况 3

对于添加成功的情况 2 和情况 3 而言，元素 a 与已经存在指定索引位置上数据以链表的方式存储。

jdk 7 ：元素 a 放到数组中，指向原来的元素

jdk 8 ：原来的元素在数组中，指向元素 a

总结：“7 上 8 下”

HashSet 底层：数组+链表的结构。

```
@Test
public void test() {
    Set set = new HashSet();//jdk7 中创建长度 16 的数组
    set.add(456);
    set.add(123);
    set.add(123);
    set.add("AA");
    set.add("CC");
    set.add(new User("Tom", 12));
    set.add(new User("Tom", 12));
    set.add(129);

    Iterator iterator = set.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());//AA、CC、129、User{name='Tom', age=12}
        // 456、123、User{name='Tom', age=12}
        //打开 hashCode 则 User 变成一个
```

```
        }
    }
    //LinkedHashSet 的使用
    //LinkedHashSet 作为 HashSet 的子类，在添加数据的同时，每个数据还维护了两个引用，记录此数据前一个数据和后一个数据
    //优点：对于频繁的遍历操作，LinkedHashSet 效率高于 HashSet
    @Test
    public void test2() {
        Set set = new LinkedHashSet();
        set.add(456);
        set.add(123);
        set.add(123);
        set.add("AA");
        set.add("CC");
        set.add(new User("Tom", 12));
        set.add(new User("Tom", 12));
        set.add(129);

        Iterator iterator = set.iterator();
        while (iterator.hasNext()) {
    System.out.println(iterator.next());//456、123、AA、CC、User{name='Tom', age=12}、129
        }
    }
}

public class TreeSetTest {
    /*
      1.向 TreeSet 中添加的数据，要求是相同类的对象
      2.两种排序方式：自然排序（实现 Comparable 接口）和 定制排序(Comparator)
      3.自然排序中，比较两个对象是否相同的标准为：compareTo()返回 0，不再是 equals()。
      4.定制排序中，比较两个对象是否相同的标准为：compare()返回 0，不再是 equals()。
     */
    @Test
    public void test() {
        TreeSet set = new TreeSet();

        //失败：不能添加不同类的对象
//        set.add(123);
//        set.add(456);
//        set.add("AA");
//        set.add(new User("Tom",12));

        //举例一：
//        set.add(34);
```

```
//            set.add(-34);
//            set.add(43);
//            set.add(11);
//            set.add(8);
//            Iterator iterator = set.iterator();
//            while (iterator.hasNext()){
//                System.out.println(iterator.next());//未写 compareTo 之前：-34、8、11、34、43
            //举例二：自然排序
            set.add(new User("Tom", 12));
            set.add(new User("Jerry", 32));
            set.add(new User("Jim", 2));
            set.add(new User("Mike", 65));
            set.add(new User("Jack", 33));
            set.add(new User("Jack", 56));//根据 compareTo()定义(有二级排序)，有此对象
            Iterator iterator = set.iterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());//写完 compareTo 之后（按姓名排序）：
                //User{name='Jack',  age=56}  User{name='Jack',  age=33}  User{name='Jerry',
age=32}
               //User{name='Jim',age=2} User{name='Mike', age=65} User{name='Tom', age=12}
            }
        }
        //按照姓名从小到大排序
    @Override
    public int compareTo(Object o) {
        if(o instanceof User){
            User user = (User) o;
//            return this.name.compareTo(user.name);
            int compare = this.name.compareTo(user.name);
            if(compare != 0){
                return compare;
            }else{
                return Integer.compare(this.age, user.age);
            }
        }else {
            throw new RuntimeException("输入的类型不匹配！");
        }
    }
}


    @Test
    public void test2() {
        Comparator com = new Comparator() {
```

```
                //按照年龄从小到大排序
                public int    (Object o1, Object o2) {
                    if (o1 instanceof User && o2 instanceof User) {
                        User u1 = (User) o1;
                        User u2 = (User) o2;
                        return Integer.compare(u1.getAge(), u2.getAge());
                    } else {
                        throw new RuntimeException("输入的数据类型不匹配！");
                    }
                }
            };
            TreeSet set = new TreeSet(com);//定制排序
            set.add(new User("Tom", 12));
            set.add(new User("Jerry", 32));
            set.add(new User("Jim", 2));
            set.add(new User("Mike", 65));
            set.add(new User("Jack", 33));
            set.add(new User("Mary", 33));//根据 compare 定义，输出没有此对象
            set.add(new User("Jack", 56));
            Iterator iterator = set.iterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
                //User{name='Jim', age=2}User{name='Tom', age=12} User{name='Jerry', age=32}
                //  User{name='Jack',  age=33}  User{name='Jack',  age=56}  User{name='Mike',
age=65}
            }
        }
}

public class ListExer {
            区分 List 中 remove(int index)和 remove(Object obj)
        @Test
        public void testListRemove() {
            List list = new ArrayList();
            list.add(1);
            list.add(2);
            list.add(3);
            updateList(list);
            System.out.println(list);//[1,2]
        private static void updateList(List list) {
            list.remove(2);//删除索引 2 位置元素
            list.remove(new Integer(2));//删除元素 2
        }
}
```

Day_7

一、Map 的实现类的结构

* |----Map:双列数据：存储 key-value 对的数据 --->类似于高中的函数：y=f(x)
* |----HashMap:作为 Map 的主要实现类；线程不安全的，效率高；存储 null 的 key 和 value
* |----LinkedHashMap:保证在遍历 map 元素时，可以按照添加的顺序实现遍历。
* 原因：在原有的 HashMap 底层结构的基础上，添加了一堆指针，指向前一个和后一个元素
* 对于频繁的遍历操作，此类执行效率高于 HashMap
* |----TreeMap:保证按照添加的 key-value 对进行排序，实现排序遍历。此时考虑 key 的自然排序或定制排序
* 底层使用红黑树
* |----Hashtable:作为古老的实现类；线程安全的，效率低；不能存储 null 的 key 和 value
* |----Properties:常用来处理配置文件。key 和 value 都是 String 类型
*
* HashMap 的底层: 数组 + 链表 (jdk7 及之前)
* 数组 + 链表 + 红黑树 (jdk8)
*
* 面试题：
* 1.HashMap 的底层实现原理？
* 2.HashMap 和 Hashtable 的异同？
* 3.CurrentHashMap 与 Hashtable 的异同

二、Map 结构的理解：

* Map 中的 key：无序的、不可重复的，使用 Set 存储所有的 key ---> key 所在的类要重写 equals 和 hashCode 方法（以 HashMap 为例）
* Map 中的 value：无序的，可重复的，使用 Collection 存储所有的 value ---> value 所在类要重写 equals
* 一个键值对：key-value 构成了一个 Entry 对象。
* Map 中的 entry：无序的、不可重复的，使用 Set 存储所有的 entry
*

三、 HashMap 的底层实现原理？（以 jdk7 为例说明）

* HashMap map = new HashMap();
* 在实例化以后，底层创建了长度为 16 的一维数组 Entry[] table。
* ...可能已经执行多次 put...
* map.put(key1,value1):
* 首先，调用 key1 的所在类的 hashCode()方法计算 key1 哈希值，此哈希值经过某种算法计算以后，得到在 Entry 数组中存放位置
* 如果此位置上数据为空，此时的 key1-value1 添加成功 ---情况 1
* 如果此位置上的数据不为空，(意味着此位置上存在一个或多个数据(以链表形式存在))，比较 key1 和已经存在的一个或多个数据的哈希值：
* 如果 key1 的哈希值与已经存在的数据的哈希值都不相同，此时 key1-value1 添加成功。 ----情况 2

* 如果 key1 的哈希值与已经存在的某一个数据（key2-value2）的哈希值相同，继续比较：调用 key1 所在类的 equals 方法，比较：
* 如果 equals()返回 false：此时 key1-value1 添加成功。 ---情况 3
* 如果 equals()返回 true：使用 value1 替换 value2.（修改）
*
* 补充：关于情况 2 和情况 3：此时 key1-value1 和原来的数据以链表的方式存储。
* 在不断的添加过程中，会涉及到扩容问题，当超出临界值(且要存放的位置非空)时，扩容。默认的扩容方式：扩容为原来容量的 2 倍，并将原有的数据复制过来。
*
* jdk8 相较于 jdk7 在底层实现方面的不同：
* 1。 new HashMap():底层没有创建一个长度为 16 的数组
* 2. jdk8 底层的数组是：Node[],而非 Entry[]
* 3. 首次调用 put()方法时，底层创建长度为 16 的数组
* 4. jdk7 底层结构只有：数组+链表。jdk8 中底层结构：数组+链表+红黑树。
* 当数组的某一个索引位置上的元素以链表形式存在的数据个数 >8 且当前数组的长度 >64 时，
* 此时索引位置上的所有数据改为使用红黑树存储。
*
* DEFAULT_INITIAL_CAPACITY : HashMap 的默认容量，16
* DEFAULT_LOAD_FACTOR：HashMap 的默认加载因子：0.75
* threshold：扩容的临界值，=容量*填充因子：16 * 0.75 => 12
* TREEIFY_THRESHOLD：Bucket 中链表长度大于该默认值，转化为红黑树:8
* MIN_TREEIFY_CAPACITY：桶中的 Node 被树化时最小的 hash 表容量:64
*

四、LinkedHashMap 的底层实现原理（了解）

* 源码中：
* static class Entry<K,V> extends HashMap.Node<K,V> {
* Entry<K,V> before, after;//能够记录添加的元素的先后顺序
* Entry(int hash, K key, V value, Node<K,V> next) {
* super(hash, key, value, next);
* }
* }
*

五、Map 中定义的方法：

* 添加、删除、修改操作：
* Object put(Object key,Object value)：将指定 key-value 添加到(或修改)当前 map 对象中
* void putAll(Map m):将 m 中的所有 key-value 对存放到当前 map 中
* Object remove(Object key)：移除指定 key 的 key-value 对，并返回 value
* void clear()：清空当前 map 中的所有数据

* 元素查询的操作：
* Object get(Object key)：获取指定 key 对应的 value
* boolean containsKey(Object key)：是否包含指定的 key
* boolean containsValue(Object value)：是否包含指定的 value

```
*     int size()：返回 map 中 key-value 对的个数
*     boolean isEmpty()：判断当前 map 是否为空
*     boolean equals(Object obj)：判断当前 map 和参数对象 obj 是否相等

*      元视图操作的方法：
*     Set keySet()：返回所有 key 构成的 Set 集合
*     Collection values()：返回所有 value 构成的 Collection 集合
*     Set entrySet()：返回所有 key-value 对构成的 Set 集合
*
* 总结：常用方法：
* 添加：put(Object key,Object value)
* 删除：remove(Object key)
* 修改：put(Object key,Object value)
* 查询：get(Object key)
* 长度：size()
* 遍历：keySet() / values() / entrySet()
public class MapTest {
    /*
    元视图操作的方法：
        Set keySet()：返回所有 key 构成的 Set 集合
        Collection values()：返回所有 value 构成的 Collection 集合
        Set entrySet()：返回所有 key-value 对构成的 Set 集合
     */
    @Test
    public void test5(){
        Map map = new HashMap();
        map.put("AA",123);
        map.put(45,1234);
        map.put("BB",56);
        //遍历不可用迭代器 iterator
        //遍历所有的 key 集
        Set set = map.keySet();
        Iterator iterator = set.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.next());//AA BB 45
        }
        //遍历所有的 value 集
        Collection values = map.values();
        for(Object obj : values){
            System.out.println(obj);//123 56 1234
        }
        //遍历所有的 Key-value 集
        //方式一：entrySet()
        Set entrySet = map.entrySet();
```

```
        Iterator iterator1 = entrySet.iterator();
        while (iterator1.hasNext()){
            Object obj = iterator1.next();
            //entrySet 集合中的元素都是 entry
            Map.Entry entry = (Map.Entry) obj;
            System.out.println(entry.getKey() + "---->" + entry.getValue());
            //AA---->123 BB---->56 45---->1234
        }
//      方式二:
        Set keySet = map.keySet();
        Iterator iterator2 = keySet.iterator();
        while (iterator2.hasNext()){
            Object key = iterator2.next();
            Object value = map.get(key);
            System.out.println(key + "===" + value);
            //AA===123 BB===56 45===1234
        }
    }

    /*
    元素查询的操作：
        Object get(Object key)：获取指定 key 对应的 value
        boolean containsKey(Object key)：是否包含指定的 key
        boolean containsValue(Object value)：是否包含指定的 value
        int size()：返回 map 中 key-value 对的个数
        boolean isEmpty()：判断当前 map 是否为空
        boolean equals(Object obj)：判断当前 map 和参数对象 obj 是否相等
     */
    @Test
    public void test4(){
        Map map = new HashMap();
        map.put("AA",123);
        map.put(45,123);
        map.put("BB",56);
        //Object get(Object key)
        System.out.println(map.get(45));//123,不存在则返回 null
        //boolean containsKey(Object key)
        boolean isExist = map.containsKey("BB");
        System.out.println(isExist);//true
        isExist = map.containsValue(123);
        System.out.println(isExist);//true
        map.clear();
        System.out.println(map.isEmpty());//true,clear 后 map 依然存在
    }
```

```
    /*
     添加、删除、修改操作：
        Object put(Object key,Object value)：将指定 key-value 添加到(或修改)当前 map 对象
中
        void putAll(Map m):将 m 中的所有 key-value 对存放到当前 map 中
        Object remove(Object key)：移除指定 key 的 key-value 对，并返回 value
        void clear()：清空当前 map 中的所有数据
     */
    @Test
    public void test3(){
        Map map = new HashMap();
        //添加
        map.put("AA",123);
        map.put(45,123);
        map.put("BB",56);
        //修改
        map.put("AA",87);
        System.out.println(map);//{AA=87, BB=56, 45=123}
        Map map1 = new HashMap();
        map1.put("CC",123);
        map1.put("DD",123);
        map.putAll(map1);
        System.out.println(map);//{AA=87, BB=56, CC=123, DD=123, 45=123}
        //remove(Object key)
        Object value = map.remove("CC");
        System.out.println(value);//123,没有则返回 null
        System.out.println(map);//{AA=87, BB=56, DD=123, 45=123}
        //clear()
        map.clear();
        System.out.println(map.size());//0
        System.out.println(map);//{}
    }

    @Test
    public void test2(){
        HashMap map = new HashMap();
        map = new LinkedHashMap();
        map.put(123,"AA");
        map.put(345,"BB");
        map.put(12,"CC");
       System.out.println(map);//{345=BB, 123=AA, 12=CC} ---> {123=AA, 345=BB, 12=CC}
    }
```

```
    @Test
    public void test(){
        Map map = new HashMap();
        map.put(null,null);
//        Map map1 = new Hashtable();
//        map1.put(2,null);//报错
    }
}

public class TreeMapTest {
    //向 TreeMap 中添加 key-value，要求 key 必须是用一个类创建的对象
    //因为要按照 key 进行排序：自然排序、定制排序
    //自然排序
    @Test
    public void test() {
        TreeMap map = new TreeMap();
        User u1 = new User("Tom", 23);
        User u2 = new User("Jerry", 32);
        User u3 = new User("Jack", 20);
        User u4 = new User("Rose", 18);

        map.put(u1, 98);
        map.put(u2, 89);
        map.put(u3, 76);
        map.put(u4, 100);

        Set entrySet = map.entrySet();
        Iterator iterator = entrySet.iterator();
        while (iterator.hasNext()) {
            Object obj = iterator.next();
            //entrySet 集合中的元素都是 entry
            Map.Entry entry = (Map.Entry) obj;
            System.out.println(entry.getKey() + "---->" + entry.getValue());
            //User{name='Jack', age=20}---->76
            //User{name='Jerry', age=32}---->89
            //User{name='Rose', age=18}---->100
            //User{name='Tom', age=23}---->98
        }
    }
    //定制排序
    @Test
    public void test2(){
        TreeMap map = new TreeMap(new Comparator() {
            @Override
```

```
                public int compare(Object o1, Object o2) {
                    if(o1 instanceof User && o2 instanceof User){
                        User u1 = (User) o1;
                        User u2 = (User) o2;
                        return Integer.compare(u1.getAge(),u2.getAge());
                    }
                    throw new RuntimeException("输入的数据类型不一致！");
                }
            });
            User u1 = new User("Tom", 23);
            User u2 = new User("Jerry", 32);
            User u3 = new User("Jack", 20);
            User u4 = new User("Rose", 18);

            map.put(u1, 98);
            map.put(u2, 89);
            map.put(u3, 76);
            map.put(u4, 100);

            Set entrySet = map.entrySet();
            Iterator iterator = entrySet.iterator();
            while (iterator.hasNext()) {
                Object obj = iterator.next();
                //entrySet 集合中的元素都是 entry
                Map.Entry entry = (Map.Entry) obj;
                System.out.println(entry.getKey() + "---->" + entry.getValue());
                //User{name='Rose', age=18}---->100
                //User{name='Jack', age=20}---->76
                //User{name='Tom', age=23}---->98
                //User{name='Jerry', age=32}---->89
            }
        }
    }

    public class PropertiesTest {
        //Properties：常用来处理配置文件。key 和 value 都是 String 类型
        public static void main(String[] args) {
            FileInputStream fis = null;
            try {
                Properties pros = new Properties();
                fis = new FileInputStream("jdbc.properties");
                pros.load(fis);//加载流对应的文件
                String name = pros.getProperty("name");
                String passworld = pros.getProperty("password");
```

```
            System.out.println("name = " + name + ",password = " + passworld);//name = Tom,password = abc123
        }catch (IOException e){
            e.printStackTrace();
        }finally {
            if(fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

总结：

List 类似于数组，对于 Set 和 Map，常用都是使用 HashXxx，LinkedHashXxx 可以按存储顺序遍历（要重写 equal 和 hashCode 方法），TreeXxx 可以按照自己的意愿进行排序遍历（要重写 compare 或 compareTo 方法 ）。

List 特点：可重复，存储有序

Set 特点：不可重复，存储无序

Map 特点：key-value 形式储存

纵向看：key->类似 Set

Value->类似 Collection(无序，可重复)

横向看：类似 Set

Vector、Hashtable、Properties 都是线程安全的古老类。

Collections:操作 Collection、Map 的工具类

```
 *
 *  面试题：Collection 和 Collections 的区别？
 *
 * @author skoud
 * @create 2020-11-14-21:43
 */
public class CollectionsTest {
    /*
    reverse(List)：反转 List 中元素的顺序
    shuffle(List)：对 List 集合元素进行随机排序
    sort(List)：根据元素的自然顺序对指定 List 集合元素按升序排序
    sort(List，Comparator)：根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
    swap(List，int， int)：将指定 list 集合中的 i 处元素和 j 处元素进行交换

    Object max(Collection)：根据元素的自然顺序，返回给定集合中的最大元素
```

Object max(Collection，Comparator)：根据 Comparator 指定的顺序，返回给定集合中的最大元素

Object min(Collection)

Object min(Collection，Comparator)

int frequency(Collection，Object)：返回指定集合中指定元素的出现次数

void copy(List dest,List src)：将 src 中的内容复制到 dest 中

boolean replaceAll(List list，Object oldVal，Object newVal)：使用新值替换 List 对象的所有旧值

*/

```
@Test
public void test1(){
    List list = new ArrayList();
    list.add(123);
    list.add(43);
    list.add(765);
    list.add(-97);
    list.add(0);
    //报异常 IndexOutOfBoundsException: Source does not fit in dest
//  List dest = new ArrayList();
//  Collections.copy(dest,list);
    List dest = Arrays.asList(new Object[list.size()]);
    Collections.copy(dest,list);
    System.out.println(dest);//[123, 43, 765, -97, 0]

    /*
    Collections 类中提供了多个 synchronizedXxx()方法，
    该方法可使将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题
     */
    //返回的 list1 即为线程安全的 list
    List list1 = Collections.synchronizedList(list);
}
@Test
public void test(){
    List list = new ArrayList();
    list.add(123);
    list.add(43);
    list.add(765);
    list.add(-97);
    list.add(0);
    System.out.println(list);//[123, 43, 765, -97, 0]
//  Collections.reverse(list);
//  Collections.shuffle(list);
//  Collections.sort(list);
```

```
//          Collections.swap(list,1,2);
            list.add(765);
            System.out.println(Collections.frequency(list, 765));//2
            System.out.println(list);//[0, -97, 765, 43, 123]、[123, 43, -97, 0, 765]、
                        // [-97, 0, 43, 123, 765]、[123, 765, 43, -97, 0]
        }
}
```