

Java加强

ZCX

一、异常

1. 异常代表程序出现的问题

2. Java的异常体系：

- Java.lang.Throwable代表所有的问题
- 接下来分为**Error**(代表系统级别的严重错误) 和 **Exception**(程序可能出现的问题，程序员用它和他的子类来封装程序可能出现的问题)
- **Exception**分为**RuntimeException**(运行时异常)和***其他异常***(编译时异常)

- ```
public static void show(){
 //运行时异常
 int[] arr = {10,20,30};
 System.out.println(arr[3]);数组超限
 String name = null;
 System.out.println(name.length()); 空指针异常
 System.out.println(10/0); 算术异常
 运行时异常，编译时不会报错
}
```

- ```
public static void show2(){  
    //编译时异常  
    String str = "2024-07-09 11:12:13";  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
    Date date = sdf.parse(str); //parse标红了，是编译时异常，提醒程序员这里容易出错  
    System.out.println(date);  
}
```

3. 异常的基本处理：

- **抛出异常(throws)**

- ```
public static void main(String[] args) throws ParseException {
 show2(); 调用时也要抛出异常
}

public static void show2() throws ParseException {
 //编译时异常
 String str = "2024-07-09 11:12:13";
```

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
Date date = sdf.parse(str); parse()抛出异常
System.out.println(date);
都抛完就不会报错了
```

- **捕获异常(try...catch)**

- ```
public static void main(String[] args) {
    try {
        show2();
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
try...catch拦截了异常，然后处理(如打印异常)
```

- 如故意写的错误：输出会打印异常信息

```
public static void show2() throws ParseException {
    //编译时异常
    String str = "2024-07-09 11:12:13";
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss"); 这里故意把-写成/
    Date date = sdf.parse(str);
    System.out.println(date);
}
java.text.ParseException: Unparseable date: "2024-07-09 11:12:13"
at java.base/java.text.DateFormat.parse(DateFormat.java:399)
at com.cxz.exception.demo1.show2(demo1.java:28)
at com.cxz.exception.demo1.main(demo1.java:11) 异常信息
```

4. 异常的作用

- 异常是用来定位程序bug的关键信息
- 可以作为方法内部的一种特殊返回值，以便通知上层调用者，方法的执行问题

- ```
public class demo2 {
 public static void main(String[] args) {
 try { 尝试捕获，如果没问题就正常执行，有问题就执行catch那段
 System.out.println(div(10, 0));
 } catch (Exception e) {
 e.printStackTrace(); // 打印异常信息
 }
 }
 //求两个数的除的结果，并返回这个结果
 public static int div(int a,int b){
 if(b==0){ 除数为0异常，抛异常怎么做：
 throw new RuntimeException("除数不能为0");
 }
 }
}
```

```

//通过throw new Exception对象，抛出一个异常
//因为如果除法计算有问题，那么就需要告诉程序员有问题
//但是程序必须要有int返回值了，还想要停止运行除法
//那么就设置除零异常就return -1;但是如果有的正常除法结果也是-1怎么办
//所以最终就可以用throw new Exception对象，把异常抛出去，返回异常结果
}
return a/b;
}
}

```

## 5. 自定义异常

- java无法为这个世界上全部的问题都提供异常类，所以要自定义异常类
- 自定义编译时异常，**extends Exception**(继承Exception类)

```

public class demo3 {
 public static void main(String[] args) throws AgelllegalException {
 show(0); 如果直接调用show()方法，就需要在main旁边加上throws AgelllegalException
 }
 //公司的系统，只要收到了年龄小于1岁或大于200岁，就是一个非法异常
 public static void show(int age) throws AgelllegalException { 定义方法，如果年龄小于1岁
 或大于200岁，就抛出异常
 注意，这个是throws，因为是编译时异常，不抛编译不通过
 if(age<1||age>200){
 throw new AgelllegalException("非法年龄"); 抛出异常，这个是throw
 }else{
 System.out.println("年龄合法");
 }
 }
}

public class AgelllegalException extends Exception { 自己定义的异常类，继承Exception
 类
 public AgelllegalException() { //无参构造器
 }
 public AgelllegalException(String age) { //有参构造器
 super(age); 调用父类构造器
 }
}

```

- ```

public static void main(String[] args) {
    try {
        show(0);
    } catch (AgelllegalException e) {

```

```
e.printStackTrace();
```

```
}
```

}使用try...catch处理异常，就不需要在main旁边写throws AgeIllegalException了

- 自定义运行时异常，extends RuntimeException(继承RuntimeException类)

```
public class demo3 {
    public static void main(String[] args) {
        try {
            show(0);
        } catch (AgeIllegalRuntimeException e) {
            e.printStackTrace();
        }
    }

    public static void show(int age) { 这个其实隐含了throws，因为反正是运行时异常，加不加没必要了
        if(age<1||age>200){
            throw new AgeIllegalRuntimeException("非法年龄");
        }else{
            System.out.println("年龄合法");
        }
    }

    public class AgeIllegalRuntimeException extends RuntimeException {
        自定义运行时异常类，继承RuntimeException类
        public AgeIllegalRuntimeException() { //无参构造器
        }
        public AgeIllegalRuntimeException(String age) { //有参构造器
            super(age);
        }
    }
}
```

- 一般定义都是运行时异常，因为编译时异常，必须处理，一层一层往上抛异常，每一层都要处理这个异常，不处理编译不通过，太过于侵入别人的程序。故最好定义运行时异常

6. 异常处理方案

- 一种是底层的异常层层抛出，最外层捕获异常，记录下异常信息，并响应适合用户观看的信息进行提示
- 第二种是最外层捕获异常后，尝试重新修复，在catch中进行修复处理，比如让try...catch整体进行循环，让用户重新输入直到输入正确的数据才停止

二、泛型

1. **泛型**：定义类、接口、方法时，同时声明了一个或多个类型变量(如< E >),称为泛型类、泛型接口、泛型方法。

- 作用：泛型提供了在编译阶段约束能够操作的数据类型，并自动进行检查的能力。

- ```
public class demo1 {
 public static void main(String[] args) {
 ArrayList < String > list = new ArrayList<>(); 定义ArrayList集合，泛型指定为String,这样集合中只能存放String类型数据
 list.add("hello");
 list.add("world");
 list.add("java");
 System.out.println(list);
 }
}
```

- 泛型的本质：把具体的数据类型作为参数传递给类型变量E(在各个方法中，使用E指代数据类型)

2. **泛型类**：定义类时，同时声明类型变量，称为泛型类

- **修饰符 class 类名<类型变量,类型变量...> { 定义泛型类}**
- 类型变量建议用大写字母，常用E,T,K,V

- ```
public class myarraylist < E >{  
} 定义泛型类，泛型类中定义的成员变量、方法，都可以使用类型变量E  
  
public class demo2 {  
    public static void main(String[] args) {  
        myarraylist< String > list = new myarraylist<>(); 创建对象，指定泛型类型为String  
    }  
}
```

3. **泛型接口**：定义接口时，同时声明类型变量，称为泛型接口

- **修饰符 interface 接口名<类型变量,类型变量...> { 抽象方法定义}**

- ```
public interface data < T > { 这是一个泛型接口，T的类型替代了一堆对象
 void add(T t); 抽象方法，参数类型为T
}

public class student { // 学生类，里面存放学生数据，这是学生对象
}

public class studentdata implements data < student >{ 学生数据操作实现类，实现studentdata接口，数据类型是student
 @Override
 public void add(student s) {
 } 抽象方法实现，参数类型为student，把student赋给了T
}
```

```

}
public class demo3 {
public static void main(String[] args) {
studentdata sd = new studentdata(); 创建对象
sd.add(new student()); 调用抽象方法，参数类型为student
}
}

```

#### 4. 泛型方法、通配符、上下限

- **泛型方法**：定义方法时，同时声明类型变量，称为泛型方法
- **修饰符 <类型变量,类型变量...> 返回值类型 方法名(参数列表) { 方法体}**

```

public class demo4 {
public static void main(String[] args)
{
String[] array = {"1","2","3"}; 创建一个字符串数组
printarray(array); 调用方法，参数类型为String
student[] students = new student[3]; 创建一个学生对象数组
students[0] = new student();
students[1] = new student();
students[2] = new student();
printarray(students); 调用方法，参数类型为student
}
public static < E > void printarray(E[] array){ 定义泛型方法，参数类型为E，E的类型替代了
student对象和String
}
}

```

- **通配符**： "?",可以在使用泛型的时候代表任意类型

```

->public class myarraylist < E >{
} 定义泛型类，数组

```

```

public class demo5 {
public static void main(String[] args) {
myarraylist< xiaomi> list = new myarraylist< xiaomi>(); 创建对象，指定泛型类型为xiaomi
go(list);
myarraylist< huawei> list2 = new myarraylist< huawei>(); 创建对象，指定泛型类型为huawei
go(list2); list2标红了
}
public static void go(myarraylist< xiaomi > list){
} 使用myarraylist泛型类，参数类型为xiaomi，就是由于使用泛型的时候定义的是xiaomi而不是
huawei，就导致xiaomi类型的对象能传进去，而hauwei的传不进去
}如果是myarraylist< car >，那么xiaomi和huawei都传不进去

```

所以此时需要添加一个通配符，即myarraylist<?>，才能都传进去

```
class car{
}
class xiaomi extends car{
}
class huawei extends car{
}
```

就算xiaomi和huawei都是继承于car，但对于泛型来说在编译时，就是完全不同的对象，就是泛型的不变性，想要变，就变为？

但是，？表示一切类型的对象都能传进去，比如跟car没关的dog类也能传进去，但是我本身设计的那个方法不想这么做

- 上下限：<? extends E> 泛型上限，表示E及其子类，<? super E> 泛型下限，表示E及其父类
- ```
public static void go(myarraylist<? extends car> list){
} 使用myarraylist泛型类，参数类型为car及其子类
```

```
public static void go2(myarraylist<? super car> list){
} 使用myarraylist泛型类，参数类型为car及其父类
```

5.泛型支持的类型

- 泛型和集合不支持基本数据类型，只支持引用数据类型
- 因为泛型是类型擦除的实现机制，在编译时实现的，这意味着在运行时所有的泛型数据类型会被替换为Object(即对象数据类型)
- 包装类：把基本数据类型的数据包装成对象的类型

基本数据类型	对应的包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

- 如何使用：(手动装箱，手动拆箱)
Integer it1 = Integer.valueOf(100);
Integer it2 = Integer.valueOf(100);
把100转换成Integer对象

Integer.valueOf()方法，里面把-128~127之间的数据缓存起来，所以，如果100在-128~127之间，那么Integer.valueOf()方法就会从缓存中取，不会重新创建对象,也就是说，it1和it2是同一个对象，但是如果it1和it2存的是130，则两个对象是不同的，因为超出了缓存范围

- 如何使用：(自动装箱和自动拆箱)

Integer it1 = 100;

Integer it2 = 100;

把基本数据类型数据转换成Integer对象(自动装箱)，其他注意事项跟上面的是一样的

int i1 = it1;

int i2 = it2;

把Integer对象转换成基本数据类型数据(自动拆箱)

- 包装类具备的其他功能：

- 可以把基本类型数据转换为字符串类型

- ```
int j = 23;
String[] rs = new String[]{Integer.toString(j)};
System.out.println(rs[0]);
```

输出为23  
如果是System.out.println(rs[0]+1);  
输出为231

- 可以把字符串类型的数值转换成数值本身对应的真实数据类型

- ```
String str = "98";
int i=Integer.parseInt(str); 把字符串类型的数值转换成数值本身对应真实的数据类型
System.out.println(i);
```

输出为98

```
System.out.println(i+2);
```

输出为100

```
int k = Integer.valueOf(str);
System.out.println(k);
```

输出为98

三、集合框架

1.集合：集合是一种容器，用来装数据的，类似于数组，但集合大小可变

2.集合体系结构

- **Collection**：单列集合，每个元素只包含一个值
- **Map**：双列集合，每个元素包含两个值(键值对)

3.collection集合

- collection< E>表示接口，规定了整体特点，然后分为List< E>接口和Set< E>接口。
- List< E>接口：又分为ArrayList< E>实现类和LinkedList< E>实现类

- Set< E>接口： 又分为HashSet< E>实现类和TreeSet< E>实现类。其中HashSet< E>实现类又有一个继承类LinkedHashSet< E>实现类

- **collection集合特点**

- List系列集合： 添加的元素是有序、可重复、有索引

- ```
List< String > list = new ArrayList<>();
```

**创建ArrayList对象**  

```
list.add("hello");
```

**添加元素**  

```
list.add("hello");
```

  

```
list.add("world");
```

  

```
list.add("java");
```

  

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

**输出结果为[hello, hello, world, java]**  

```
System.out.println(list.get(0));
```

**获取指定索引的元素,即输出hello**

- Set系列集合： 添加的元素是无序、不重复、无索引。

- LinkedHashSet< E>集合： 添加的元素是有序、不重复、无索引
    - HashSet< E>集合： 添加的元素是无序、不重复、无索引

- ```
Set< String> s = new HashSet<>();
```



```
s.add("hello");
```



```
s.add("hello");
```

添加重复元素，不会重复添加

```
s.add("world");
```



```
s.add("java");
```



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

输出结果为[world, java, hello]是无序的，并且没有get方法，没有索引

- TreeSet< E>集合： 添加的元素是按照默认大小升序排序、不重复、无索引

- **collection常用功能**

功能	描述
add(E e)	添加元素(布尔返回值)
remove(Object o)	删除元素(布尔返回值)
contains(Object o)	判断集合中是否包含某个元素(布尔返回值)
size()	获取集合中元素的个数
isEmpty()	判断集合是否为空(布尔返回值)
clear()	清空集合
toArray()	将集合转换为数组

- ```
Collection< String > list = new ArrayList<>();
```

  

```
list.add("hello");
```

**添加元素**  

```
list.add("world");
```

```
list.add("java");
System.out.println(list);
list.remove("world"); 删除元素
System.out.println(list);
System.out.println(list.size()); 获取集合中元素的个数
System.out.println(list.isEmpty()); 判断集合是否为空
System.out.println(list.contains("hello")); 判断集合中是否包含某个元素
list.clear(); 清空集合
String[] arr = list.toArray(new String[0]); 将集合转换为字符串数组
System.out.println(Arrays.toString(arr));
Object[] Ob = list.toArray(); 将集合转换为对象数组
```

- **collection集合遍历方式**

- **迭代器遍历**：迭代器是用来遍历集合的专用方式(数组没有迭代器)，在JAVA中迭代器的代表是Iterator

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class demo1 {
 public static void main(String[] args) {
 Collection c = new ArrayList<>();
 c.add(new String("AA"));
 c.add(new String("BB"));
 c.add(new String("CC"));
 c.add(new String("DD"));
 c.add(new String("EE"));
 c.add(new String("FF"));
 c.add(new String("GG"));
 c.add(new String("HH"));
 c.add(new String("II"));
 c.add(new String("JJ"));
 System.out.println(c);
 Iterator< String> it = c.iterator(); 迭代器
 while (it.hasNext()) { //判断当前位置有没有数据
 String s = it.next(); //取当前位置数据赋给s，并把指针移到下一位
 System.out.println(s);
 }
 }
}
```

- **增强for循环**：格式：**for(数据类型 变量名 : 集合名){}**
- 增强for循环可以用来遍历数组和集合

- 增强for遍历集合，本质就是迭代器遍历集合的简化写法

```
Collection c = new ArrayList<>();
for(String s:c){ 增强for循环
System.out.println(s);
}
```

- **lambda表达式：格式：(参数列表)->{方法体}**

```
c.forEach(new Consumer< String >() { ** //一开始的需要匿名内部类**
 @Override
 public void accept(String s) {
 System.out.println(s);
 }
});
c.forEach(s-> System.out.println(s)); //lambda简化后的
c.forEach(System.out::println); //方法引用,进一步简化了
```

- **认识并发修改异常问题：**遍历集合的同时又存在增删集合元素的行为时可能出现业务异常

```
public class demo2 {
 public static void main(String[] args) {
 ArrayList< String > list = new ArrayList<>();
 list.add("CCCC ");
 list.add("BBAA");
 list.add("CCAA");
 list.add("DDAA");
 list.add("EEAA");
 for(int i=0;i<list.size();i++){ 遍历集合,并删除含有AA的字符串
 String name = list.get(i);
 if(name.contains("AA")){
 list.remove(name);
 }
 }
 System.out.println(list);
 } 输出结果为[CCCC, CCAA, EEAA]发现没有删干净，因为每次删除一个元素，集合后边的
 } 元素都会马上前移，就导致了部分元素直接跳过判断了
}
```

```
for(int i=0;i<list.size();i++){ 一种解决方法
 String name = list.get(i);
 if(name.contains("AA")){
 list.remove(name);
 i--;加个i--操作，就可以了
 }
}
```

```

}
System.out.println(list);
• for(int i=list.size()-1;i>=0;i--){ 另一种解决方法，倒着遍历
String name = list.get(i);
if(name.contains("AA")){
list.remove(name);
}
}
System.out.println(list);

```

### • 三种遍历的区别

- Iterator< String > iterator = list.iterator(); 迭代器遍历知道并发删除异常问题，故可以使用迭代器里面的remove方法解决这个问题

这种方法可适用于没有索引的集合

```

while(iterator.hasNext()){
String name = iterator.next();
if(name.contains("AA")){
iterator.remove();
}不可以用list.remove(name)，会报错
}
System.out.println(list);

```

- 增强for循环，和lambda表达式都无法解决并发修改异常问题
- 因为增强for循环底层是迭代器，发现了修改数据就会报错，lambda表达式底层是增强for循环，也不能解决。因为明面上无法调用迭代器的remove方法，所以无法解决
- 总之要想遍历并删除就用迭代器。
- 增强for循环和lambda表达式只适合单纯做遍历
- for循环适合做有索引的遍历和修改
- 

## 4.List集合

- List集合特有方法:

| 方法名                           | 描述        |
|-------------------------------|-----------|
| void add(int index,E element) | 在指定位置插入元素 |
| E get(int index)              | 获取指定位置的元素 |
| E remove(int index)           | 删除指定位置的元素 |
| E set(int index,E element)    | 替换指定位置的元素 |

- ```

public static void main(String[] args) {
    List< String> list = new ArrayList<>();
    list.add("CCCC "); 添加元素
    list.add("BBAA");
    list.add("CCAA");
    list.add("DDAA");
    System.out.println(list);
    System.out.println(list.get(2)); 获取指定位置的元素
    list.add(2,"AAAA"); 在指定位置插入元素
    System.out.println(list);
    list.remove(3); 删除指定位置的元素
    System.out.println(list);
    list.set(2,"BBBB"); 替换指定位置的元素
    System.out.println(list);
}

```

- **List集合适合的遍历方式:**for循环、迭代器、增强for循环、lambda表达式
- **ArrayList的底层原理:**ArrayList底层是基于数组存储数据的
- **数组的特点:** 根据索引查询速度快, 增删效率低
- **LinkedList的底层原理:** LinkedList底层是基于双链表存储数据的
- **链表:** 是由一个一个结点组成的, 结点在内存不连续, 每个结点包含数据值和指向下一个结点的地址。增删相对快, 查询难, 因为无论查询哪个节点都要从头开始查
- **双链表:** 每个结点有数据值和两个指针, 一个指针指向下一个结点, 一个指针指向前一个结点。对首位元素进行增删快, 对中间元素进行增删慢
- **LinkedList方法:**

方法名	描述
void addFirst(E element)	在链表开头添加元素
void addLast(E element)	在链表末尾添加元素
E getFirst()	获取链表开头的元素
E getLast()	获取链表末尾的元素
E removeFirst()	删除链表开头的元素并返回该元素
E removeLast()	删除链表末尾的元素并返回该元素

- **LinkedList应用场景**
 - 设计队列, FIFO
 - 每次添加元素, 添加到链表末尾, 删除元素(即获取返回的删除的元素), 删除链表头的元素, 这样是从后往前加入的元素, 然后先进先出, 从头开始获取元素

- 设计栈：后进先出
- 每次添加元素从头开始添加，然后取元素也从头取，这样就是后进先出