# day15

## Collections工具类

1. Collections : 类, 来自于java.util包, 是对于单列集合封装的一个工具类, 类中有很多方法功能可以对单列集合(List, Set)进行操作
2. Collections工具类在API中没有给出构造, 证明不能实例化对象, 类中的所有成员和方法都是static修饰的
3. Collections中常用方法:
4. sort(List list) : 将参数提供的list集合按照默认的升序进行排列(从小到大),返回值类型void
5. binarySearch(List<T> list, T key ): 需要list为升序集合; 查找key值在list集合中出现的索引位置, 如果找了索引位返回大于等于0整数结果; 否则返回负数
6. [frequency](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collections.html" \l "frequency(java.util.Collection, java.lang.Object))([Collection](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)<?> c, [Object](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/lang/Object.html" \o "java.lang 中的类) o) : 查找元素o在集合c中出现的次数, 返回int类型
7. max([Collection](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)  c): 获取到集合中的最大值
8. min([Collection](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collection.html" \o "java.util 中的接口)  c): 获取到集合中的最小值
9. replaceAll(List<T> list, T old, T newObj ): 将参数list集合中的所有old元素替换成newObj元素
10. reverse(List list) : 将参数list集合进行逆序元素排列
11. [shuffle](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/Collections.html" \l "shuffle(java.util.List))([List](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/../../java/util/List.html" \o "java.util 中的接口)<?> list) : 将参数list集合中的元素进行随机的混乱排序, 可以使用在麻将, 扑克牌的洗牌动作中

代码

|  |
| --- |
| package com.ujiuiye.map;  import java.util.ArrayList;  import java.util.Collections;  public class Demo03\_Collections {  public static void main(String[] args) {  ArrayList<Integer> list = new ArrayList<>();  list.add(12);  list.add(25);  list.add(3);  list.add(-8);  list.add(12);    // 1. sort(List list) : 将参数提供的list集合按照默认的升序进行排列(从小到大)  Collections.sort(list);  System.out.println(list);//[-8, 3, 12, 12, 25]    // 2)binarySearch(List<T> list, T key ): 需要list集合默认升序集合; 查找key值在list集合中出现的索引位置, 如果找了索引位返回大于等于0整数结果; 否则返回负数  System.out.println(Collections.binarySearch(list, 12));// 2  System.out.println(Collections.binarySearch(list, 10));// -3    // 3)frequency(Collection<?> c, Object o) : 查找元素o在集合c中出现的次数, 返回int类型  System.out.println(Collections.frequency(list, 12));// 2    // 4)max(Collection  c): 获取到集合中的最大值  // 5)min(Collection  c): 获取到集合中的最小值  System.out.println(Collections.max(list));// 25  System.out.println(Collections.min(list));// -8    // 6)replaceAll(List<T> list, T old, T newObj): 将参数list集合中的所有old元素替换成newObj元素  Collections.replaceAll(list, 12, 888);  System.out.println(list);//[-8, 3, 888, 888, 25]    // 7)reverse(List list) : 将参数list集合进行逆序元素排列  Collections.reverse(list);  System.out.println(list);// [25, 888, 888, 3, -8]  }  } |

## 异常

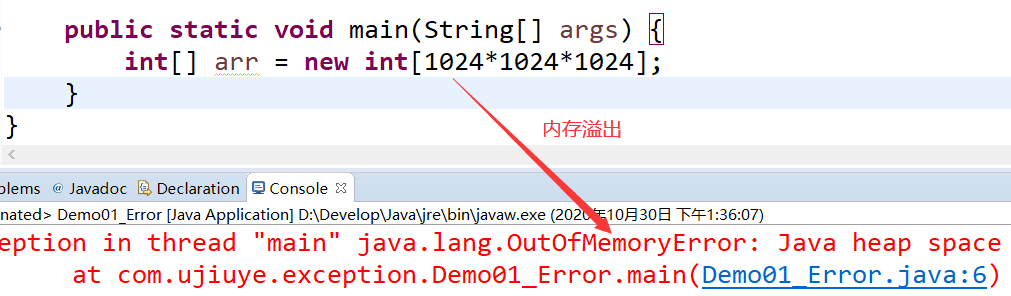
### 异常的概述

1. 异常(Exception) : 在Java代码的编译和运行过程中, 发生的一些不正常的, 不符合情况的, 错误, 都统称为异常
2. Java中所有异常都封装成异常类, 当有某一种异常发生, 创建出对应异常对象, 将异常发生原因, 发生位置封装到这个异常对象中, 在代码中, 获取到这个异常对象, 就可以了解异常的问题原因
3. 代码中发生异常 : 可以跳转代码; 可以处理掉异常代码继续运行; 或者也可以停止代码

### 异常的体系结构

1. Throwable : 所有的错误Error和异常Exception的父类, 只有在Throwable体系中, 才可以进行异常的处理

-- Error : 所有错误的父类, 表示代码中发生了比较严重的问题, 通常不进行处理, 只能修改代码, 目前不会关注



--Exception : 所有异常的父类, 异常表示代码中发生的不太严重的问题, 可以解决, 可以处理, 甚至可以避免

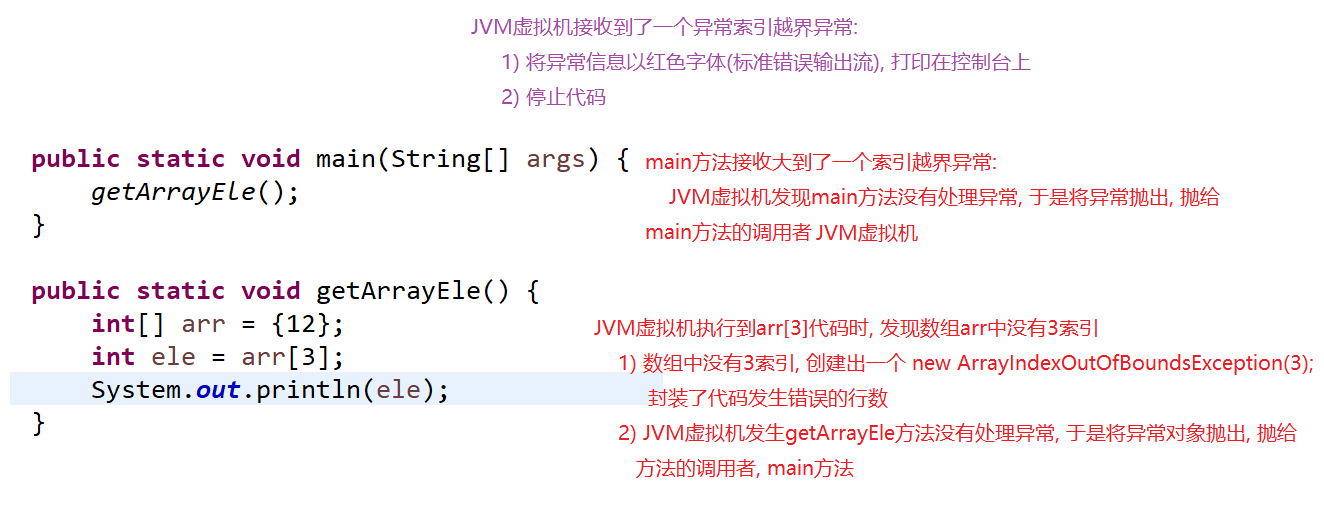
-- a : 运行时异常 RuntimeException: 在代码运行时, 才会发生的异常, 运行时异常发生不需要在代码中进行任何的处理

--b : 编译时异常 : 除了运行时异常之外的所有异常, 都是编译时异常; 如果代码中发生了编译时异常, 代码会自动提示进行异常的处理

### JVM虚拟机默认处理异常

如果代码中没有任何对于异常的处理机制, 那么如果发生异常,由JVM虚拟机进行默认异常处理;

从代码发生问题的行数起, 将异常信息抛出给方法的调用者, 一层一层往上抛出, 直到抛给main, main抛给JVM虚拟机, 最终虚拟机打印异常信息, 停止代码



### 异常的手动处理方式

1. 异常声明 :

如果方法中出现编译时异常, 可以在方法上声明可能会发生的这些异常的类型, 异常声明目的就是为了告诉方法的调用者, 这个方法可能会有异常问题发生, 调用时, 需要进行预处理

1. 异常捕获 :

如果方法中出现了异常, 可以使用一些语法结构, 将异常进行捕获, 捕获成功之后, 可以对这个异常进行处理, 异常处理完毕, 代码继续运行, 不停止; try...catch系列语句

### throw抛出异常

1. throw : 关键字, 表示异常的抛出
2. throw关键字的使用场景:

如果代码中有一些不符合实际情况场景, 可以在这些场景中使用throw关键字, 创建出一个异常, 然后抛出这个异常, 抛出给当前方法的调用者, throw关键字可以做到让代码跳转或者停止

1. throw关键字的使用语法:

throw new 异常类(写异常发生的具体原因);

// 异常类: 可以是Exception和Exception 的任意一个子类

1. throw关键字使用注意:
2. throw关键字每次只能抛出一个异常, 如果有多个异常分别设计在多个场景中, 每一场景抛出一个异常
3. throw关键字需要设计在方法中
4. 实际开发中throw关键字的使用:

实际开发中, 需要验证前端传递来的数据是否符合实际的要求 : 例如 : 密码只能有数字和字母大小写组成, 长度5-15位之间; 如果前台传递的数据不符合实际要求, 代码没有必须继续运行, 可以抛出一个错误信息 : 提示密码设定不符合规则

String regex = “[1-9a-zA-Z]{5,15}”;

if(!password.matches(regex)){

// 代码不用继续

throw new Exception(“密码设计不符合规则, 需要密码中只能有数字和字母大小写”);

}

// price 需要支付金额

double price = 1500;

// have 目前余额

double have = 1200;

if(price > have){// 余额不够支付,交易失败,代码停止报错

throw new Exception(“余额不足”);

}

代码

|  |
| --- |
| package com.ujiuye.exception;  public class Demo03\_Throw {  public static void main(String[] args) {  // int[] arr = {1,2,3,4};  int[] arr = null;  getArrayEle(arr,6);  }    // 定义出一个方法 : 求int[]数组中执行索引位置元素的结果  public static void getArrayEle(int[] arr, int index) {  // 1. 提供参数数组arr不能为null  if(arr == null) {// 不合符实际情况场景  throw new NullPointerException("数组不能为null");  }    // 2.提供index索引位置需要在arr数组中存在  if(index < 0 || index > arr.length-1) {  throw new ArrayIndexOutOfBoundsException(index + "索引在数组arr中不存在");  }    int ele = arr[index];  System.out.println(ele);  }  } |

### throws异常声明

1. throws : 关键字, 表示异常的声明, 是异常手动处理方式之一
2. throws关键字的使用场景 :

如果方法中有编译时期异常, 可以在方法上使用throws关键字, 将可能会发生的所有的编译时期异常都声明一下;

目的 : 告知方法的调用者, 此方法可能会有异常, 使用需要谨慎, 对可能会发生的异常做出预处理

1. throws关键字使用语法结构:

修饰符 返回值类型 方法名(参数列表) throws 异常类型1, 异常类型2...{

方法体;

}

代码

|  |
| --- |
| package com.ujiuye.exception;  import java.io.FileNotFoundException;  public class Demo04\_Throws {  public static void main(String[] args) throws FileNotFoundException, Exception{  String file = "123.txt";  // main方法调用了带有异常功能 : checkFile  checkFile(file);  }    // 定义出一个方法功能 : 问是否可以匹配文件 abc.txt  // 目前针对于checkFile方法功能 : 异常已经算作处理了  public static void checkFile(String file) throws FileNotFoundException,Exception{  if(file == null) {  // NullPointerException 运行时期异常, 发生不需要进行处理  // throw new NullPointerException("提供的文件不能null");  throw new Exception("提供的文件不能null");  }    if(!"abc.txt".equals(file)) {  //Unhandled exception type FileNotFoundException  throw new FileNotFoundException("提供文件不是目标文件abc.txt");  }    System.out.println(file);  }  } |

### throw和throws关键字的比较

1. 功能不同:

a : throw, 表示抛出异常

b : throws , 表示声明异常

1. 使用的代码位置不同:

a : throw, 使用在方法中

b : throws, 在方法之上, 在参数列表之后使用

1. 对于异常处理个数不同:

a : throw, 每次只能抛出一个异常

b : throws , 每次可以声明多个异常, 多个异常之间使用逗号进行分隔

1. 执行:

a : throw , 代码执行到throw关键字, 一定有一个异常会发生

b : throws , 代码执行到throws关键字, 不一定真的有异常发生

### try...catch语句捕获异常

1. try...catch语法结构:

try{

// 可能会发生异常的代码;

}catch(异常类型 变量名){

// 对于异常处理方案;

}

1. 解释:

a : try : 关键字, 用于检测try大括号中的代码是否有异常

b : catch : 关键字, 表示捕获异常

catch小括号中写出需要捕获的异常的类型

catch大括号中就是对于异常的处理方法

1. try...catch语句执行机制:
2. 先运行try代码块中的逻辑, try检测代码是否出现异常; 如果没有异常发生, 代码一切正常执行, catch也不需要捕获和运行了
3. 如果try代码块中检测出了异常, 从当前发生异常行开始, 代码直接跳转到catch语句中, try中后面代码不再运行,将异常信息抛出给catch, catch将接收到异常与小括号中预计要捕获的异常进行匹配

a : catch接收到的异常与预计要捕获的异常匹配, 捕获异常成功, 执行catch大括号中的逻辑处理异常, 异常处理完毕, 代码不停止, 接着整个try...catch语句之后继续运行

b : catch接收到的异常与预计要捕获的异常没有匹配, 没有捕获住异常, 最终异常就会抛给JVM虚拟机, 就是执行JVM默认异常处理机制

代码

|  |
| --- |
| package com.ujiuye.exception;  public class Demo05\_TryCatch {  public static void main(String[] args) {  try {// 可能会发生问题的代码  int i = 10 / 0; // 第7行发生数学运算问题 : new ArithmeticException("/ by 0");  System.out.println(i);    }catch(ArithmeticException ex){// ArithmeticException ex = new ArithmeticException("/ by 0");  int i = 0;  System.out.println("将i值设置为默认的0");  }  System.out.println("game over");  }  } |

### try...catch...catch多catch语句捕获异常

1. 如果try代码块中, 可能会发生多种异常, 这种情况可以使用多个catch语句, 每一个catch语句捕获住一个异常
2. 多catch语法结构:

try{

// 可能会发生异常的代码;

}catch(异常类型1 变量名1){

// 异常1的处理方式;

}catch(异常类型2 变量名2){

// 异常2的处理方式;

}...

1. 多catch语句执行机制:
2. 先运行try代码块中的逻辑, try检测代码是否出现异常; 如果没有异常发生, 代码一切正常执行, catch也不需要捕获和运行了
3. 如果try代码块中发生异常, 从当前发生异常代码行, try之后的代码就不再运行,直接跳转到catch语句中, catch接收到了一种异常; 从上到下依次与catch语句中异常类型进行匹配, 如果匹配了某一个异常类型成功, 执行对应异常的处理方案; 整个try...catch语法结构结束, 代码继续运行;
4. 如果多个catch语句都无法与接收到的异常匹配, 那么异常只能一层一层往上抛出, 直到抛给JVM, 使用默认异常处理机制

1. 多catch语句使用的注意事项:

如果捕获的多个异常之间具有子父类的继承关系

1. 先捕获子类异常, 后捕获父类异常; 防止多态发生导致代码报错
2. 也可以直接捕获一个父类异常, 利用多态

代码

|  |
| --- |
| package com.ujiuye.exception;  import java.io.FileNotFoundException;  public class Demo06\_多catch语句处理异常 {  public static void main(String[] args) {  String file = "cc.txt";  try {  checkFile(file);  }catch(FileNotFoundException ex) {  file = "abc.txt";  System.out.println("给file类型设置了默认文件匹配 : abc.txt");  }catch(Exception ex) {  file = "123.txt";  System.out.println("给file类型设置了默认文件匹配 : 123.txt");  }    /\* 有问题的捕获方式  \* try {  checkFile(file);  }catch(Exception ex) {// Exception ex = new FileNotFoundException("提供文件不是目标文件abc.txt");  file = "123.txt";  System.out.println("给file类型设置了默认文件匹配 : 123.txt");  }catch(FileNotFoundException ex) { // 无法抵达的代码,报错  file = "abc.txt";  System.out.println("给file类型设置了默认文件匹配 : abc.txt");  }\*/    try {  checkFile(file);  }catch(Exception ex) {// Exception ex = new FileNotFoundException("提供文件不是目标文件abc.txt");  file = "123.txt";  System.out.println("给file类型设置了默认文件匹配 : 123.txt");  }    System.out.println("game over");  }    // 定义出一个方法功能 : 问是否可以匹配文件 abc.txt  // 目前针对于checkFile方法功能 : 异常已经算作处理了  public static void checkFile(String file) throws FileNotFoundException,Exception{  if(file == null) {  // NullPointerException 运行时期异常, 发生不需要进行处理  // throw new NullPointerException("提供的文件不能null");  throw new Exception("提供的文件不能null");  }    if(!"abc.txt".equals(file)) {  //Unhandled exception type FileNotFoundException  throw new FileNotFoundException("提供文件不是目标文件abc.txt");  }  System.out.println(file);  }  } |

### try...catch...finally语句处理异常

1. 语法结构:

try{

// 可能会发生异常的代码;

}catch(异常类型1 变量名1){

// 异常1的处理方式;

}catch(异常类型2 变量名2){

// 异常2的处理方式;

}...finally{

// 一定会执行语句

}

1. 解释 :

finally : 关键字, 表示一定会执行的

1. 如果代码中有一定需要执行的代码, 可以设计在finally中, 通常finally中进行资源关闭

1. 语法执行机制:
2. try代码块中没有异常, 在整个语法结构结束之前, 运行finally中代码逻辑
3. try代码中发生异常, 异常被某个catch语句捕获成功, 在整个语法结构结束之前, 运行finally中代码逻辑
4. try代码中发生异常, 但是catch代码块没有捕获成功, 在整个语法结构结束之前, 运行finally中代码逻辑

代码

|  |
| --- |
| package com.ujiuye.exception;  public class Demo07\_TryCatchFinally {  public static void main(String[] args) {  System.out.println(useI());// 20  System.out.println("game over");  }    public static int useI() {  try {  int i = 10 / 0;  return i;  }catch(NullPointerException e) {  int i = 10;  System.out.println("0不能做除数, 设置默认值为10");  return i;  }finally {// 注意 : 尽量不要在finally中写完return语句, 代码太过固定  int i = 20;  System.out.println("我一定要运行");  // return i;  }  }  } |

### try...finally语句

1. 语法结构:

try{

// 代码逻辑;

}finally{

// 一定会执行的代码;

}

1. 运行机制:
2. try代码块中没有异常, 语法结束之前, 执行finally
3. try代码块中有异常, 语法结束之前, 执行finally

注意 : 这种语法因为没有catch语句, 因此代码发生异常, 无法捕获, 只能依靠JVM默认处理机制

1. 语法使用场景 : 代码中有必须要执行的逻辑, 设计到finally中; 但是对于代码中的异常不处理

代码

|  |
| --- |
| package com.ujiuye.exception;  public class Demo08\_TryFinally {  public static void main(String[] args) {  try {  int[] arr = {12};  System.out.println(arr[0]);  }finally {  System.out.println("我必须执行");  }  System.out.println("game over");  }  } |

### Throwable中方法

1. Throwable中构造:

a : new Throwable(String message)

当创建出一个异常对象时, 需要将异常发生的具体原因描述清楚

1. Throwable中对于异常信息追踪方法功能:
2. getMessage() : 获取到异常中封装的异常信息, 返回String类型
3. toString() : 获取到封装的异常信息, 以及异常类型
4. [printStackTrace](mk:@MSITStore:C:\\Users\\96243\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Throwable.html" \l "printStackTrace())() : 获取到封装的异常信息,以及异常类型, 以及异常具体发生的代码行数, 最具体的异常信息追踪, 本身自带输出打印功能, 并且是以标准错误流红色字体打印

代码

|  |
| --- |
| package com.ujiuye.exception;  public class Demo09\_Throwable中方法 {  public static void main(String[] args) {  try {  fun();  } catch (Exception e) {// = new Exception("数组不能为null");  String s = e.getMessage();  System.out.println(s);    String s1 = e.toString();  System.out.println(s1);    e.printStackTrace();  }  System.out.println("game over");  }    public static void fun() throws Exception {  int[] arr = null;    if(arr == null) {  throw new Exception("数组不能为null");  }  }  } |

### 自定义异常

1. 自己定义出一个异常类, 用于表示某一种异常情况
2. 需要自定义异常的原因, 就是JDK提供的所有异常类, 除了追踪异常信息之外,没有任何其他方法功能, 因此有异常发生, 没有办法 通过这个异常对象中的功能解决问题; 因此自定义出一个异常类, 可以在自定义类中封装很多解决异常问题的方式
3. 自定义异常类实现步骤
4. 定义出一个类, 使用Exception为类名的结尾, Exception就是异常类典型的结尾标志
5. 可以继承Exception为父类, 那么自定义异常类就是编译时期异常类;可以选择继承Runtime为父类, 那么自定义异常类就是运行时异常
6. 创建出一个可以封装异常信息的构造方法

public 自定义异常类(String message){

super(message);

}

代码

|  |
| --- |
| package com.ujiuye.exception;  public class MyException extends Exception {  public MyException() {    }  // 调用父类封装异常信息的方法即可  public MyException(String message) {  super(message);  }  // 在自定义异常类中可以随意封装方法, 这是JDK提供的异常类中所没有的  public void use() {  System.out.println("我可以解决问题");  }  } |

|  |
| --- |
| package com.ujiuye.exception;  public class Demo10\_自定义异常 {  public static void main(String[] args) {  try {  fun();  } catch (MyException e) {  e.printStackTrace();  e.use();  }  }    public static void fun() throws MyException {  if(true) {  throw new MyException("自定义异常类");  }  }  } |