不可变集合
Stream流
    Stream流的获取
    Stream流的常用API
    Stream流综合案例
    收集Stream流
认识异常体系
    异常概述、体系
    常见运行时异常
    常见编译时异常
    异常的默认处理流程
    编译时异常的处理机制
    运行时异常的处理机制
    自定义异常
异常体系
    日志技术的概述
    日志技术体系
    Logback概述
    Logback快速入门
    Logback配置详解-输出位置、格式设置
    Logback配置详解-日志级别设置

不可变集合

- 概念

不可变集合，就是不可被修改的集合。
集合的数据项在创建的时候提供，并且在整个生命周期中都不可改变。否则报错。

- 作用

如果某个数据不能被修改，把它防御性地拷贝到不可变集合中是个很好的实践。
或者当集合对象被不可信的库调用时，不可变形式是安全的

- **如何创建不可变集合?**

  在List、Set、Map接口中，都存在of方法，可以创建一个不可变的集合 【since 1.9】
  这个集合不能添加，不能删除，不能修改。

```
List.of();
Set.of()
Map.of()
```

方法名称	说明
static <E> List<E> of(E...elements)	创建一个具有指定元素的List集合对象
static <E> Set<E> of(E...elements)	创建一个具有指定元素的Set集合对象
static <K , V> Map<K, V> of(E...elements)	创建一个具有指定元素的Map集合对象

```
    public static void main(String[] args) {
        //不可变集合，从jdk1.9开始,只可以访问不可以操作
        final List<Double> list = List.of(120.1, 22.4, 120.4, 111.2);
//        list.add(111.2);//java.lang.UnsupportedOperationException
//        list.set(1, 100.1);//java.lang.UnsupportedOperationException
        list.forEach(System.out::println);
    }
```

```
120.1
22.4
120.4
111.2
```

```
    @Test
    public void test1() {
        //不可变集合，不可以增删改，只可以查
        final Set<Double> set = Set.of(120.1, 22.4, 120.4, 111.2);
//        final Set<Double> set = Set.of(120.1, 22.4, 120.4, 111.2, 111.2);//当有重复值是也报错
java.lang.IllegalArgumentException: duplicate element: 111.2
//        set.clear();//java.lang.UnsupportedOperationException
        System.out.println(set);
        set.forEach(System.out::println);
    }
```

```
[120.1, 22.4, 111.2, 120.4]
120.1
22.4
111.2
120.4
```

Stream流

- 概念

在Java 8中，得益于Lambda所带来的函数式编程， 引入了一个全新的Stream流概念。

目的：用于简化集合和数组操作的API。

- 原生与Stream对比

```
    @Test
    public void test1() {
        final List<String> list = new ArrayList<>();
        Collections.addAll(list, "张三丰", "张无忌", "赵敏", "任盈盈", "小龙女", "张强");
        final ArrayList<String> zhangThree = new ArrayList<>();
        final ArrayList<String> zhangThree2 = new ArrayList<>();
        System.out.println(list);
        for (String s : list) {
            if (s.startsWith("张") && s.length() == 3) zhangThree.add(s);
        }
        System.out.println(zhangThree);

        System.out.println("-----------------------------------------------");

        list.stream().filter(s -> s.startsWith("张")).filter(s -> s.length() == 3).forEach(s ->
zhangThree2.add(s));
        System.out.println(zhangThree2);
    }
```

```
[张三丰, 张无忌, 赵敏, 任盈盈, 小龙女, 张强]
[张三丰, 张无忌]
-----------------------------------------------
[张三丰, 张无忌]
```

- **Stream流式思想的核心：**

通过数组/集合.stream(),得到集合或者数组的Stream流（就是一根传送带）

然后就用这个Stream流简化的API来方便的操作元素

- **Stream流的三类方法**

获取Stream流
　创建一条流水线，并把数据放到流水线上准备进行操作

中间方法
　流水线上的操作。一次操作完毕之后，还可以继续进行其他操作

终结方法
　一个Stream流只能有一个终结方法，是流水线上的最后一个操作foreach、count

- 小结

Stream流的作用是什么，结合了什么技术?
　简化集合、数组操作的API。结合了Lambda表达式。

Stream流的获取

- 集合获取Stream流

名称	说明
default Stream<E> stream()	获取当前集合对象的Stream流

- 数组获取Stream流方法

数组获取Stream流的方式

名称	说明
Arrays public static <T> Stream<T> stream(T[] array)	获取当前数组的Stream流
Stream public static<T> Stream<T> of(T... values)	获取当前数组/可变数据的Stream流

- 获取Stream流

```
    @Test
    public void test2() {
        //Collection集合
        final List<Object> list = new ArrayList<>();
        final Stream<Object> stream1 = list.stream();

        final Set<String> set = new HashSet<>();
        final Stream<String> stream2 = set.stream();

        //Map集合
```

```
        final HashMap<String, Integer> map = new HashMap<>();
        final Set<String> keySet = map.keySet();
        final Collection<Integer> values = map.values();
        final Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
        final Stream<String> stream3 = keySet.stream();
        final Stream<Integer> stream4 = values.stream();
        final Stream<Map.Entry<String, Integer>> stream5 = entrySet.stream();

        //数组
        final int[] ints = new int[4];
        final IntStream stream7 = Arrays.stream(ints);
        final Stream<int[]> stream6 = Stream.of(ints);
        stream6.forEach(System.out::println);
        stream7.forEach(System.out::println);
    }
```

```
[I@9e89d68
0
0
0
0
```

Stream流的常用API

- 中间方法

Stream流的常用API(中间操作方法)

名称	说明
`Stream<T> filter(Predicate<? super T> predicate)`	用于对流中的数据进行**过滤。**
`Stream<T> limit(long maxSize)`	获取前几个元素
`Stream<T> skip(long n)`	跳过前几个元素
`Stream<T> distinct()`	去除流中重复的元素。依赖(hashCode和equals方法)
`static <T> Stream<T> concat(Stream a, Stream b)`	**合并**a和b两个流为一个流

- 终结方法

Stream流的常见终结操作方法

名称	说明
`void forEach(Consumer action)`	对此流的每个元素执行遍历操作
`long count()`	返回此流中的元素数

```
    @Test
    public void test3() {
        final List<String> list = new ArrayList<>();
        Collections.addAll(list, "张三丰", "张无忌", "赵敏", "任盈盈", "小龙女", "张强");
        final Stream<String> stream = list.stream();//得到stream流

        // Stream<T> filter(Predicate<? super T> predicate);函数式接口
        //filter()内部是一个函数式接口，该接口中有一个叫test()的方法
        //filter()会遍历集合中的每一个元素，然后调用Predicate接口中的test方法，看是否满足条件，将不满足条件的干掉
        //filter()是一个中间方法
        stream.filter(new Predicate<String>() {
            @Override
            public boolean test(String s) {
                return s.startsWith("张");
            }
        }).forEach(System.out::println);//方法引用
    }
```

```
张三丰
张无忌
张强
```

```
    @Test
    public void test4() {
        final List<String> list = new ArrayList<>();
        Collections.addAll(list, "张三丰", "张无忌", "赵敏", "任盈盈", "小龙女", "张强", "张作霖", "张自忠", "张三
丰");
        //limit(n)取前几个
//        list.stream().filter(s->s.startsWith("张")).limit(3).forEach(System.out::println);//取前三个以张开头
的

        //skip(n)跳过前几个
//        list.stream().filter(s -> s.startsWith("张")).skip(3).forEach(System.out::println);//跳过三个以张开
头的

        //distinct()去重
        //去重，如果是自定义类型，则根据hashcode和equals
//        list.stream().filter(s -> s.startsWith("张")).distinct().forEach(System.out::println);

        //count()容器元素个数
```

```
//        System.out.println(list.stream().filter(s -> s.startsWith("张")).distinct().count());//获取过滤后元素个数

        //map()加工方法，第一个参数:原材料，第二个参数:返回值 第一种
        final ArrayList<Object> arrayList = new ArrayList<>();
/*        list.stream().map(new Function<String, Student>() {
            @Override
            public Student apply(String s) {
                return new Student(s);
            }
        }).forEach(student -> arrayList.add(student));
*/


        //map()加工方法，第一个参数:原材料，第二个参数:返回值 第二种
        //首先把姓张的过滤下来，然后去重，接着拿到前两个，下一步将原来字符串的元素加工成学生类型，通过遍历，将流上的学生对象添加到List集合中
//        list.stream().filter(s -> s.startsWith("张")).distinct().limit(2).map(s -> new Student(s)).
        //forEach(student -> arrayList.add(student));

        //map()加工方法，第一个参数:原材料，第二个参数:返回值 第二种
        //首先把姓张的过滤下来，然后去重，接着拿到前两个，下一步将原来字符串的元素加工成学生类型，通过遍历，将流上的学生对象添加到List集合中
        //使用构造器引用（前提：将接到的参数直接用于构造器的入参，初始化对象）
//        list.stream().filter(s -> s.startsWith("张")).distinct().limit(2).map(Student::new).forEach(student -> arrayList.add(student));
//        System.out.println(arrayList);


        //concat()合并流
        final Stream<String> stream1 = list.stream().filter(s -> !s.startsWith("张"));
        final Stream<String> stream2 = list.stream().skip(1).limit(4);

        //    public static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
//<T>表示这是一个泛型方法，Stream<T>表示的是返回值，入参的泛型表示，两个入参的泛型类型都要是返回值，或是返回值的子类；当两个入参的泛型类型不同时，可能要用Object
        final Stream<String> concat = Stream.concat(stream1, stream2);
        concat.forEach(s -> arrayList.add(s));
        System.out.println(arrayList);
    }
```

```
[赵敏, 任盈盈, 小龙女, 张无忌, 赵敏, 任盈盈, 小龙女]
```

- 注意
-

中间方法也称为非终结方法，调用完成后返回新的Stream流可以继续使用，支持链式编程。

在Stream流中无法直接修改集合、数组中的数据。想要访问集合中的元素，还是要用集合中的方法；流只是用来过滤加工数据

终结操作方法，调用完成后流就无法继续使用了，原因是不会返回Stream了

Stream流综合案例

```
public class StreamDemo2 {
    private static double allMoney;
    private static double allMoney2;

    public static void main(String[] args) {
        List<Employee> one = new ArrayList<>();
        one.add(new Employee("猪八戒", '男', 30000, 25000, null));
        one.add(new Employee("孙悟空", '男', 25000, 1000, "顶撞上司"));
        one.add(new Employee("沙僧", '男', 20000, 20000, null));
        one.add(new Employee("小白龙", '男', 20000, 25000, null));

        List<Employee> two = new ArrayList<>();
        two.add(new Employee("武松", '男', 15000, 9000, null));
        two.add(new Employee("李逵", '男', 20000, 10000, null));
        two.add(new Employee("西门庆", '男', 50000, 100000, "被打"));
        two.add(new Employee("潘金莲", '女', 3500, 1000, "被打"));
        two.add(new Employee("武大郎", '女', 20000, 0, "下毒"));

        /*
         * 把拿到最高工资的封装成优秀员工对象
         * */
        final Optional<Topperformer> optional = one.stream().max((e1, e2) -> Double.compare(e1.getSalary() + e1.getBonus(),
                                                                 e2.getBonus() + e2.getSalary()))
                .map(employee -> new Topperformer(employee.getName(), employee.getBonus() +
employee.getSalary()));
        final Topperformer topperformer = optional.get();
        System.out.println(topperformer);



        /*
         * 开发一部，去掉一高一低两个工资，求平均工资
         * */
        one.stream().sorted((e1, e2) -> Double.compare(e1.getSalary() + e1.getBonus(), e2.getSalary() +
e2.getBonus()))
                .skip(1).limit(one.size() - 2).forEach(employee -> {
            allMoney += employee.getBonus() + employee.getSalary();
        });
        System.out.println("开发一部平均工资为:" + allMoney / (one.size() - 2));



        /*
         * 开发一和二部，去掉一高一低两个工资，求平均工资
         * */
        final Stream<Employee> oneStream = one.stream();
        final Stream<Employee> towStream = two.stream();
        final Stream<Employee> concat = Stream.concat(oneStream, towStream);
        concat.sorted((e1, e2) -> Double.compare(e1.getSalary() + e1.getBonus(), e2.getSalary() +
e2.getBonus()))
                .skip(1).limit(one.size() + two.size() - 2).forEach(employee -> {
```

```
            allMoney2 += employee.getBonus() + employee.getSalary();
        });
        final BigDecimal money = BigDecimal.valueOf(allMoney2);
        final BigDecimal of = BigDecimal.valueOf(one.size() + two.size() - 2);
        System.out.println("开发部平均工资为:" + money.divide(of, 2, RoundingMode.HALF_UP));
    }
}
```

```
Topperformer{name='猪八戒', money=55000.0}
开发一部平均工资为:42500.0
开发部平均工资为:34285.71
```

收集Stream流

- 概念

收集Stream流的含义：就是把Stream流操作后的结果数据转回到集合或者数组中去

Stream流：方便操作集合/数组的手段

集合/数组：才是开发中的目的

Stream流的收集方法

名称	说明
R collect(Collector collector)	开始收集Stream流，指定收集器

Collectors工具类提供了具体的收集方式

名称	说明
public static <T> Collector toList()	把元素收集到List集合中
public static <T> Collector toSet()	把元素收集到Set集合中
public static Collector toMap(Function keyMapper , Function valueMapper)	把元素收集到Map集合中

```
    @Test
    public void test1() {
        final ArrayList<String> list = new ArrayList<>();
        Collections.addAll(list, "房山", "张丹", "张三", "健康卡", "健康卡");
        final List<String> list1 = list.stream().skip(2).collect(Collectors.toList());
        System.out.println(list1);

        //注意注意注意：流只能使用一次，用过了foreach count collect toArray等方法后，流就没了
        //重新获取流
        final Set<String> set = list.stream().collect(Collectors.toSet());
        System.out.println("set集合：" + set);//去重


//        final Object[] objects = list.stream().toArray();//这里的类型时Object,如果确定一定都是某个子类类型可以使用下面的方法
/*        final String[] toArray = list.stream().toArray(new IntFunction<String[]>() {
            @Override
            public String[] apply(int value) {
                return new String[value];
            }
        });*/



//        final String[] toArray = list.stream().toArray(v -> new String[v]);Lambda表达式简化
        final String[] toArray = list.stream().toArray(String[]::new);//构造器引用
        System.out.println("数组：" + Arrays.toString(toArray));
    }
```

```
[张三, 健康卡, 健康卡]
set集合：[张丹, 张三, 健康卡, 房山]
数组：[房山, 张丹, 张三, 健康卡, 健康卡]
```

认识异常体系

异常概述、体系

- **什么是异常?**

异常是程序在“编译”或者“执行”的过程中可能出现的问题

　注意：语法错误不算在异常体系中。

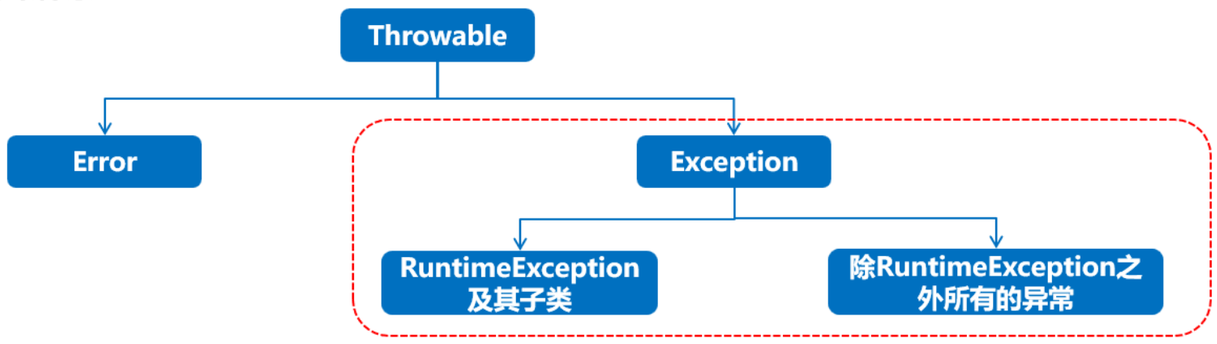
比如:数组索引越界、空指针异常、 日期格式化异常，等...

- **为什么要学习异常**?

异常一旦出现了，如果没有提前处理，程序就会退出JVM虚拟机而终止.

研究异常并且避免异常，然后提前处理异常，体现的是程序的安全, 健壮性。

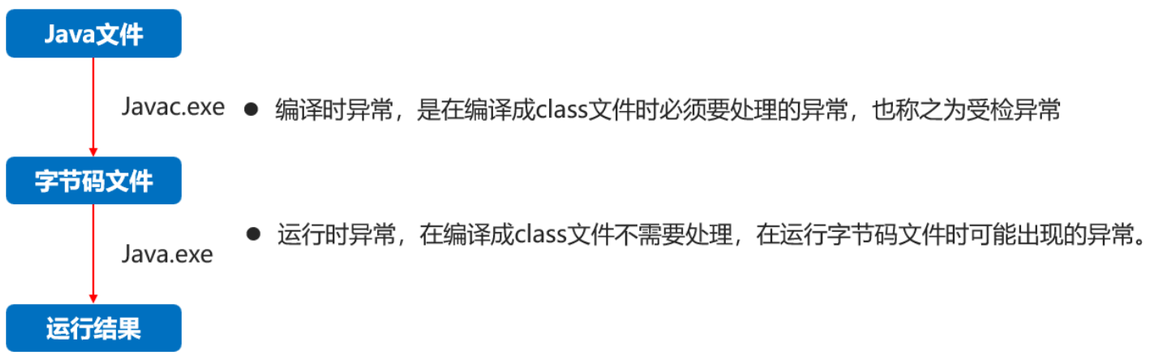
异常体系



Error：
　系统级别问题(操作系统出现问题)、JVM退出等，无法通过代码控制。

Exception：
　java.lang包下，称为异常类，它表示程序本身可以处理的问题
　运行时异常：RuntimeException及其子类，编译阶段不会报错。 (空指针异常，数组索引越界异常)
　编译时异常：除RuntimeException之外所有的异常，编译期必须处理的，否则程序不能通过编译。 (日期格式化异常)

编译时异常和运行时异常



- 编译时异常，是在编译成class文件时必须要处理的异常，也称之为受检异常
- 运行时异常，在编译成class文件不需要处理，在运行字节码文件时可能出现的异常。

简单来说：
　编译时异常 就是在编译的时候出现的异常 Idea会帮我们同步编译;如果用记事本，则会在执行javac.exe的时候报错
　运行时异常 就是在运行时出现的异常。 就是当jvm加载字节码文件时发现的异常

- 小结

异常是什么?
　异常（Exception）是代码在编译或者执行的过程中可能出现的错误；

异常分为几类?
　编译时异常：没有继承RuntimeExcpetion的异常，编译阶段就会出错。
　运行时异常：继承自RuntimeException的异常或其子类，编译阶段不报错，运行可能报错

学习异常的目的?
　避免异常的出现，同时处理可能出现的异常，让代码更稳健

常见运行时异常

直接继承自RuntimeException或者其子类，编译阶段不会报错，运行时可能出现的错误。

- **运行时异常示例**

数组索引越界异常: ArrayIndexOutOfBoundsException

```
    int[] ints=new int[2];
    System.out.println(ints[2]);
```

空指针异常 : NullPointerException，直接输出没有问题，但是调用空指针的变量的功能就会报错。

```
    String a=null;
    System.out.println(a);
```

数学操作异常：ArithmeticException

```
    int a=11/0;
```

类型转换异常：ClassCastException

```
    Object o = 12;
    String s= (String) o;
```

```
    String a="123mm";
    Integer.valueOf(a);
```

```
java.lang.ArrayIndexOutOfBoundsException: 3

    at exception_demo.RuntimeExceptionDemo1.test1(RuntimeExceptionDemo1.java:19) <25 个内部行>
```

异常栈信息，从下往上看

最下面的方法最先调用

```
public class RuntimeExceptionDemo1 {
    @Test
    public void test1() {
        System.out.println("程序开始----------------");
        final int[] ints = new int[3];
        System.out.println(ints[2]);
        /*
         * java.lang.ArrayIndexOutOfBoundsException: 3 【数组下标越界异常】运行时异常，都是RuntimeException的子类或孙子类
         *
         * class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException
         *
         * class IndexOutOfBoundsException extends RuntimeException
         * */
        //System.out.println(ints[3]);//运行时异常，如果出现并且没有try catch/throws jvm停止运行



        Object o = 12;
        Integer integer = (Integer) o;//只要有继承关系，就可以强转
        //String与基本数据类型及其包装类都不可以互相强转

        /*
         * java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
         *
         *class ClassCastException extends RuntimeException
         * */

        //String string = (String) o;//任何类都是Object的子类 【类型转换异常】 程序终止 运行时异常，是RuntimeException的子类


        /*
         * java.lang.ArithmeticException: / by zero
         * public class ArithmeticException extends RuntimeException
         * */

//        int a = 12 / 0;//【数学操作异常】 程序终止 运行时异常，是RuntimeException的直接子类



        String s1 = "123";
        System.out.println(Integer.valueOf(s1));
        String s2 = "12aaa";
        /*
         * java.lang.NumberFormatException: For input string: "12aaa"
         * class NumberFormatException extends IllegalArgumentException
         * class IllegalArgumentException extends RuntimeException
         * */
//        Integer.valueOf(s2);//【数字转换异常】程序终止 运行时异常，是RuntimeException的间接子类



        String a1 = null;
        System.out.println(a1);//直接输出没有问题，但调用功能时会出错
        /*
         * java.lang.NullPointerException
         * class NullPointerException extends RuntimeException
         *
         * */
        System.out.println(a1.length());//【空指针异常】 运行时异常 程序终止 是RuntimeException的直接子类

        System.out.println("程序结束----------------");

    }

}
```

运行时异常：一般是程序员业务没有考虑好或者是编程逻辑不严谨引起的程序错误，
自己的水平有问题

常见编译时异常

- **编译时异常**
-

没有直接或间接继承RuntimeException，编译阶就报错，必须处理，否则代码不通过

- **编译时异常的作用是什么**
-

是担心程序员的技术不行，在编译阶段就爆出一个错误，目的在于提醒不要出错!（提醒程序员此处容易写错）

即时出错了也会打出异常栈信息

编译时异常是可遇不可求

- 编译时异常示例

```
    @Test
    //使用大的Exception 可以外抛全部的异常
    public void test1() throws Exception {
        String date = "2015-01-12 10:23:21";
        final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        final Date parse = sdf.parse(date);//此处提示有未处理的异常，此时我们自信往外抛
        System.out.println(parse);
    }
```

```
Mon Jan 12 10:23:21 CST 2015
```

异常的默认处理流程

```
    public static void main(String[] args) {//3 抛给main方法，main抛给jvm，然后从异常点干掉程序
        System.out.println("程序开始.....");
        divides(1, 0);//2异常会从方法中出现的点这里抛出给调用者
        System.out.println("程序结束.....");
    }

    private static void divides(int i, int j) {
        System.out.println(i);
        System.out.println(j);
        int c = i / j;//1 默认会在出现异常的代码那里自动的创建一个异常对象：ArithmeticException。
        System.out.println(c);
    }
```

```
程序开始.....
1
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at exception_demo.ExceptionDemo.divides(ExceptionDemo.java:22)
    at exception_demo.ExceptionDemo.main(ExceptionDemo.java:15)
```

1.默认会在出现异常的代码那里自动的创建一个异常对象：ArithmeticException

2.异常会从方法中出现的点这里抛出给调用者，调用者最终抛出给JVM虚拟机

3.虚拟机接收到异常对象后，先在控制台直接输出异常栈信息数据

4.直接从当前执行的异常点干掉当前程序。

5.后续代码没有机会执行了，因为程序已经死亡

- 小结

默认的异常处理机制并不好，一旦真的出现异常，程序立即死亡!

编译时异常的处理机制

编译时异常是编译阶段就出错的，所以必须处理，否则代码根本无法通过

- **编译时异常的处理形式有三种：**

1.出现异常直接抛出去给调用者，调用者也继续抛出去

2.出现异常自己捕获处理，不麻烦别人

3.前两者结合，出现异常直接抛出去给调用者，调用者捕获处理

- **抛出异常格式：**

```
格式一：
    方法 throws 异常1 ，异常2 ，异常3 ..{
    }

规范做法：
    方法 throws Exception{

    }
    //代表可以抛出一切异常
```

- 方式一：外抛throws

```
    public static void main(String[] args) throws FileNotFoundException, ParseException {
        System.out.println("程序开始.....");
        format("1231321313");//main方法也不处理，把异常抛给jvm
        System.out.println("程序结束.....");
    }

    private static void format(String s) throws ParseException, FileNotFoundException {
        final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        final Date date = simpleDateFormat.parse(s);//把异常抛给方法的调用者
        System.out.println(date);
        final FileInputStream is = new FileInputStream("D://aafdsa/a.txt");
        System.out.println(is);
    }
```

- 方式二 try catch

监视捕获异常，用在方法内部，可以将方法内部出现的异常直接捕获处理。

这种方式还可以，发生异常的方法自己独立完成异常的处理，程序可以继续往下执行

- 格式：

```
try{
// 监视可能出现异常的代码!
}catch(异常类型1 变量){
// 处理异常 一般是打印异常栈信息
}catch(异常类型2 变量){
// 处理异常 一般是打印异常栈信息
}

建议格式:
try{
// 可能出现异常的代码!
}catch (Exception e){
e.printStackTrace(); // 直接打印异常栈信息
}

 //Exception可以捕获处理一切异常类型!
```

```
    public static void main(String[] args) throws Exception {
        System.out.println("程序开始.....");
        format("1231321313");
        System.out.println("程序结束.....");
    }

    private static void format(String s) throws Exception {
        try {
            final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            final Date date = simpleDateFormat.parse(s);//当此处出现异常时，下面的代码不会在执行；但还会跑try范围之外
的代码，直到跑完
            System.out.println(date);
            final FileInputStream is = new FileInputStream("D://aafdsa/a.txt");
            System.out.println(is);
        } catch (ParseException e) {//只会抓到一处异常，看谁先出异常
            e.printStackTrace();
        } catch (FileNotFoundException e) {//只会抓到一处异常，看谁先出异常
            e.printStackTrace();
        }
    }
```

```
程序开始.....
java.text.ParseException: Unparseable date: "1231321313"
    at java.text.DateFormat.parse(DateFormat.java:366)
    at exception_demo.ExceptionDemo.format(ExceptionDemo.java:28)
    at exception_demo.ExceptionDemo.main(ExceptionDemo.java:21)
程序结束.....
```

```
    public static void main(String[] args) throws Exception {
        System.out.println("程序开始.....");
        format("1231321313");
        System.out.println("程序结束.....");
    }

    private static void format(String s) {

        final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date date = null;
        try {
            date = simpleDateFormat.parse(s);//当此处出现异常时，下面的代码不会在执行；但还会跑try范围之外的代码，直到
跑完
        } catch (ParseException e) {
            e.printStackTrace();
        }
        System.out.println(date);
        FileInputStream is = null;
        try {
            is = new FileInputStream("D://aafdsa/a.txt");//这里也会跑，然后报错
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        System.out.println(is);

    }
```

```
程序开始.....
java.text.ParseException: Unparseable date: "1231321313"
    at java.text.DateFormat.parse(DateFormat.java:366)
    at exception_demo.ExceptionDemo.format(ExceptionDemo.java:30)
    at exception_demo.ExceptionDemo.main(ExceptionDemo.java:21)
java.io.FileNotFoundException: D:\aafdsa\a.txt (系统找不到指定的路径。)
    at java.io.FileInputStream.open0(Native Method)
    at java.io.FileInputStream.open(FileInputStream.java:195)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)
    at java.io.FileInputStream.<init>(FileInputStream.java:93)
    at exception_demo.ExceptionDemo.format(ExceptionDemo.java:37)
    at exception_demo.ExceptionDemo.main(ExceptionDemo.java:21)
null
null
程序结束.....
```

- **异常处理方式3 —— 前两者结合**

方法直接将异通过throws抛出去给调用者
调用者收到异常后直接捕获处理

```
    public static void main(String[] args) {
        System.out.println("程序开始.....");

        try {
            format("1231321313");//由调用者经行捕获，由此可以得知方法执行情况；也可以通过返回值来判断
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("程序结束.....");
    }

    private static void format(String s) throws Exception {//方法直接将异常抛给调用者

        final SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date date = null;
        date = simpleDateFormat.parse(s);//当此处出现异常时，下面的代码不会在执行；但还会跑try范围之外的代码，直到跑完

        System.out.println(date);
        FileInputStream is = null;
        is = new FileInputStream("D://aafdsa/a.txt");//这里也会跑，然后报错
        System.out.println(is);
    }
```

- 小结

在开发中按照规范来说第三种方式是最好的：底层的异常抛出去给最外层，最外层集中捕获处理。

实际应用中，只要代码能够编译通过，并且功能能完成，那么每一种异常处理方式似乎也都是可以的。

运行时异常的处理机制

运行时异常编译阶段不会出错，是运行时才可能出错的，所以编译阶段不处理也可以

按照规范建议还是处理：建议在最外层调用处集中捕获处理即可

因为编写出现异常的方法默认外抛RuntimeException

```
    public static void main(String[] args) {
        System.out.println("程序开始.....");
        chu(1, 0);//如果没有进行异常捕获，出现异常时，从这里开始就挂了；不会往下执行
        System.out.println("程序结束.....");
    }

    private static void chu(int a, int b) throws RuntimeException {//运行时异常默认是外抛RuntimeException,写不写
都有
        int c = a / b;//出现异常，外抛
        System.out.println(c);
    }
```

```
程序开始.....
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at exception_demo.ExceptionDemo.chu(ExceptionDemo.java:26)
    at exception_demo.ExceptionDemo.main(ExceptionDemo.java:21)
```

- 异常处理时代码稳健案例

```
    private static void guess() {
        final Scanner sc = new Scanner(System.in);
        while (true) {
            try {
                System.out.print("请输入数据：");
                final String data = sc.nextLine();
                final Double value = Double.valueOf(data);
                if (value > 0) {
                    System.out.println("成功输入：" + value);
                    break;
                } else {
                    System.out.println("请输入正数！");
                }
            } catch (Exception e) {
                System.out.println("请输入合法数据");
            }
        }
    }
```

```
请输入数据：.
请输入合法数据
请输入数据：.0
请输入正数!
请输入数据：0.1
成功输入：0.1
```

自定义异常

- **自定义异常的必要性**

Java无法为这个世界上全部的问题提供异常类。
如果企业想通过异常的方式来管理自己的某个业务问题，就需要自定义异常类了

- **自定义异常的好处：**

可以使用异常的机制管理业务问题，如提醒程序员注意

同时一旦出现bug，可以用异常的形式清晰的指出出错的地方

- **自定义异常的分类**

1、自定义编译时异常

- 定义一个异常类继承Exception.
- 重写构造器。
- 在出现异常的地方用throw new 自定义对象抛出，

  作用：编译时异常是编译阶段就报错，提醒更加强烈，一定需要处理！！

2、自定义运行时异常

- 定义一个异常类继承RuntimeException.
- 重写构造器
- 在出现异常的地方用throw new 自定义对象抛出!

  作用：提醒不强烈，编译阶段不报错！！运行时才可能出现！！

```
class AgeIllegalException extends Exception {
    public AgeIllegalException() {
    }

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

```
package exception_demo;


public class ExceptionDemo2 {

    public static void main(String[] args) {
        try {
            checkAge(-12);//编译时异常,此时选择捕获
        } catch (AgeIllegalException e) {
            e.printStackTrace();
        }

        System.out.println("程序结束！");
    }


    /*
     throw:在方法内部创建一个异常对象，从此点抛出一个编译时异常;此时应该在方法申明上外抛给调用者

     throws:在方法申明上外抛方法内部的没有处理异常给调用者
     * */
    private static void checkAge(int age) throws AgeIllegalException {
        if (age < 0 || age > 120) {
            throw new AgeIllegalException(age + " is ilegal!");//从此点抛出一个编译时异常
        }
        System.out.println("Age is legal!Please go on!");
    }
}
```

```
exception_demo.AgeIllegalException: -12 is ilegal!
    at exception_demo.ExceptionDemo2.checkAge(ExceptionDemo2.java:31)
    at exception_demo.ExceptionDemo2.main(ExceptionDemo2.java:15)
程序结束!
```

```
/*
 * 当该异常不会经常被触发，或影响不是很大，就定义成运行时异常
 * */
class AgeIllegalException extends RuntimeException {
    public AgeIllegalException() {
    }

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

```
public class ExceptionDemo2 {

    public static void main(String[] args) {
        try {
            checkAge(-12);//运行时异常,此时选择捕获;可以不捕获运行时异常
        } catch (AgeIllegalException e) {
            e.printStackTrace();
        }

        System.out.println("程序结束！");
    }


    /*
```

```
     *throw:在方法内部创建一个异常对象，从此点抛出一个编译时异常;此时应该在方法申明上外抛给调用者
     *
     * throws:在方法申明上外抛方法内部的没有处理异常给调用者
     * */
    private static void checkAge(int age) {
        if (age < 0 || age > 120) {
            throw new AgeIllegalException(age + " is ilegal!");//从此点抛出一个编译时异常
        }
        System.out.println("Age is legal!Please go on!");
    }

}
```

异常体系

日志技术的概述

程序中的日志： 程序中的日志可以用来记录程序运行过程中的信息，并可以进行永久存储

- **输出语句的弊端**

信息只能展示在控制台
不能将其记录到其他的位置（文件，数据库）
想取消记录的信息需要修改代码才可以完成

- **日志技术具备的优势**

可以将系统执行的信息选择性的记录到指定的位置（控制台、文件中、数据库中）。
可以随时以开关的形式控制是否记录日志，无需修改源代码

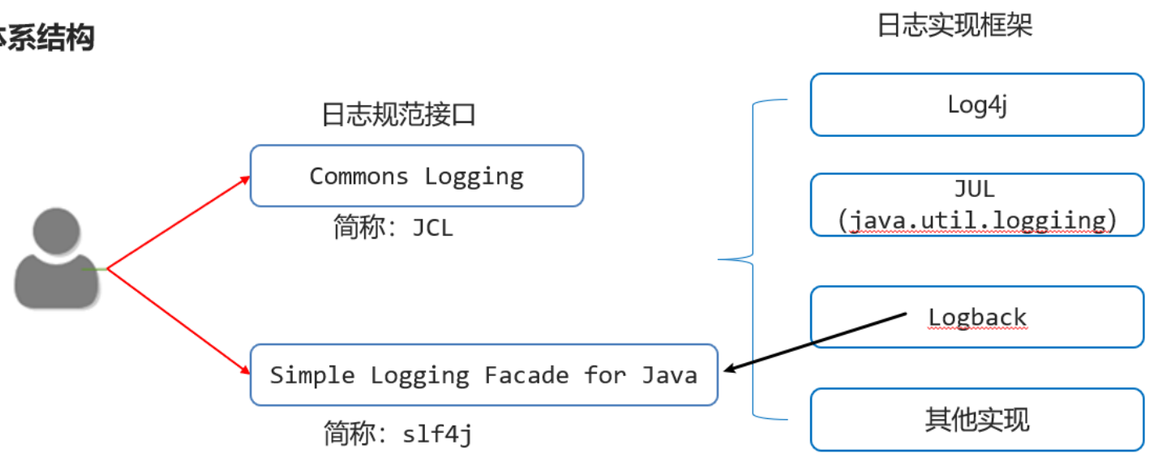
---

1. 日志技术的具体优势

	输出语句	日志技术
输出位置	只能是控制台	可以将日志信息写入到文件或者数据库中
取消日志	需要修改代码，灵活性比较差	不需要修改代码，灵活性比较好
多线程	性能较差	性能较好

日志技术体系

体系结构



日志规范：一些接口，提供给日志的实现框架设计的标准。
日志框架：牛人或者第三方公司已经做好的日志记录实现代码，后来者直接可以拿去使用。
因为对Commons Logging的接口不满意，有人就搞了SLF4J。因为对Log4j的性能不满意，有人就搞了Logback。
logback实现了Simple Logging Facade for java

Logback概述

- 概述

Logback是由log4j创始人设计的另一个开源日志组件，性能比log4j要好

> 官网地址：https://logback.qos.ch/index.html

Logback是基于slf4j的日志规范实现的框架。logback是Simple Logging Facade for java的实现类

- **Logback**主要分为四个技术模块

logback-core： logback-core 模块为其他两个模块奠定了基础，必须有。
logback-classic：它是log4j的一个改良版本，同时它完整实现了slf4j API。
logback-access 模块与 Tomcat 和 Jetty 等 Servlet 容器集成，以提供 HTTP 访问日志功能
slf4j-api：日志规范 java并没有提供slf4j的接口文件

Logback快速入门

1. 在项目下新建文件夹lib，导入Logback的相关jar包到该文件夹下，并添加到项目依赖库中去。(code、classic、slf4japi)到maven找
2. 将Logback的核心配置文件logback.xml直接拷贝到src目录下（必须是src下）
3. 在代码中获取日志的对象

```
public static final Logger LOGGER = LoggerFactory.getLogger("类对象");
```

4. 使用日志对象LOGGER调用其方法输出日志信息

- 快速入门

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class LogbackDemo {
    private static final Logger LOGGER = LoggerFactory.getLogger("LogbackDemo.class");

    public static void main(String[] args) {
        while (true) {
            try {
                LOGGER.info("程序即将执行.....");
                int a = 10;
                LOGGER.trace("a=" + a);
                int b = 0;
                LOGGER.trace("b=" + b);
                LOGGER.debug("a / b=" + a / b);
            } catch (Exception e) {
                e.printStackTrace();
                LOGGER.error(e + "");
            }
        }
    }
}
```

```
2022-03-13 21:43:05.283 [INFO ] LogbackDemo.class [main] : 程序即将执行.....
2022-03-13 21:43:05.283 [TRACE] LogbackDemo.class [main] : a=10
2022-03-13 21:43:05.283 [TRACE] LogbackDemo.class [main] : b=0
2022-03-13 21:43:05.283 [ERROR] LogbackDemo.class [main] : java.lang.ArithmeticException: / by zero
```

- **logback的配置文件**

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!--
        CONSOLE ：配置输出到控制台的日志格式
    -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <!--输出流对象 默认 System.out 改为 System.err所有用log输出的东西都死红色的-->
        <target>System.out</target>
        <encoder>
            <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度
                %msg：日志消息，%n是换行符-->
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%-5level] %c [%thread] : %msg%n</pattern>
        </encoder>
    </appender>

    <!-- FILE配置输出到文件的日志格式 -->
    <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
            <charset>utf-8</charset>
        </encoder>
        <!--日志输出路径-->
        <file>C:/code/itheima-data.log</file>
        <!--指定日志文件拆分和压缩规则-->
        <rollingPolicy
                class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
            <!--通过指定压缩文件名称，来确定分割文件方式-->
            <fileNamePattern>D:\learn\黑马程序员\java基础\day18、日志框架、阶段项目\\data-%d{yyyy-
MMdd}.log%i.gz</fileNamePattern>
            <!--文件拆分大小-->
            <maxFileSize>1MB</maxFileSize>
        </rollingPolicy>
    </appender>

    <!--
    level:用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF
    ， 默认debug
    <root>可以包含零个或多个<appender-ref>元素，标识这个输出位置将会被本日志级别控制。
    -->
    <root level="ALL"><!-- 这里的ALL表示打印一切级别的日志-->
        <appender-ref ref="CONSOLE"/><!--  将信息打印到控制台  -->
        <appender-ref ref="FILE"/><!--  将信息打印到文件  -->
    </root>
</configuration>
```

Logback配置详解-输出位置、格式设置

- 核心配置文件

Logback日志系统的特性都是通过核心配置文件logback.xml控制的

- **Logback**日志输出位置、格式设置：

通过logback.xml 中的<append>标签可以设置输出位置和日志信息的详细格式。
通常可以设置2个日志输出位置：一个是控制台、一个是系统文件中
输出到控制台的配置标志
<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
输出到系统文件的配置标志
<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">

Logback配置详解-日志级别设置

- 如果系统上线后只想记录一些错误的日志信息或者不想记录日志了，怎么办

> 可以通过设置日志的输出级别来控制哪些日志信息输出或者不输出。

- 日志级别

级别程度依次是：TRACE< DEBUG< INFO<WARN<ERROR ；默认级别是debug（忽略大小写），对应其方法。
作用：用于控制系统中哪些日志级别是可以输出的，只输出级别 不低于 设定级别的日志信息。
DEBUG级别：只打INFO,WARN,ERRO,不大TRACE

ALL 和 OFF分别是打开全部日志信息，及关闭全部日志信息

具体在<root level="INFO">标签的level属性中设置日志级别

```
    <root level="INFO">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE" />
    </root>
```

- 小结

Logback的日志级别是什么样的?
级别程度依次是：TRACE< DEBUG< INFO<WARN<ERROR
默认级别是debug（忽略大小写），只输出不低于当前级别的日志
ALL 和 OFF分别是打开全部日志和关闭全部日志