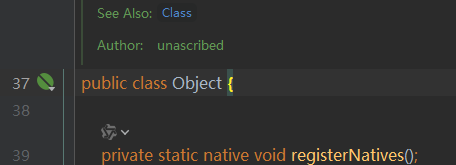
# Object类

## 基本概念

在Java中，Object 类是**所有类的超类**（根类），即所有类（包括数组）都直接或间接继承自 Object。如果一个类没有显式继承其他类，它会**默认继承**Object。Object 类定义在 java.lang 包中，是Java类层次结构的基石。

## 源码



## 常用的方法

### boolean equals(Object obj)

指示某个对象是否与此对象相等（判断对象的值是否相等）。

### Class getClass()

返回此对象的运行时类，返回动态时的某个类。

### int hashCode()

返回对象的哈希码值

### void notify()

唤醒在此对象监视器上等待的单个线程。

### void notifyAll()

唤醒在此对象监视器上等待的所有线程。

### String toString()

返回该对象的字符串表示。

### void wait()

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。

### protected void finalize()

垃圾回收（GC）时调用，用于资源清理（不推荐使用，已废弃）。

问题：

1、执行时机不确定，可能引发内存泄漏。

2、Java 9+ 已标记为 @Deprecated，建议使用 AutoCloseable 或 Cleaner 替代。

## ****为什么需要Object类****

### 1、统一类型

所有对象都可向上转型为 Object，方便通用编程（如集合类 List<Object>）。

### 2、提供默认行为

如 toString()、equals() 等方法的默认实现。

### 3、多线程支持

wait()、notify() 用于线程间通信。

## 注意事项

1、重写 equals() 必须重写 hashCode()，否则违反哈希表契约。

2、clone() 慎用：深拷贝需手动实现。

3、避免 finalize()：使用 try-with-resources 或 Cleaner 替代。

## 总结

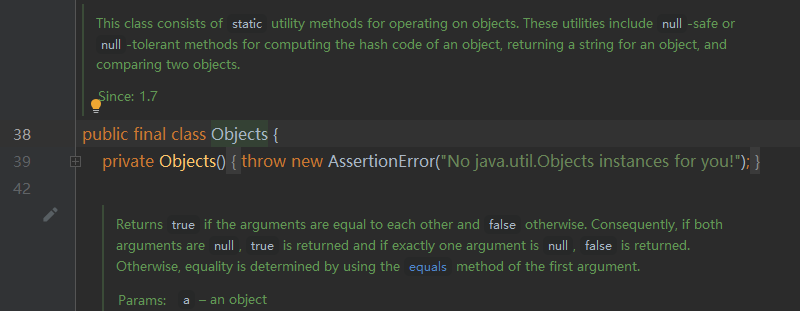


# Objects类

## 基本概念

java.util.Objects 是 Java 7 引入的一个工具类，提供了一系列静态方法用于操作对象。这些方法主要用于处理对象的比较、判空、哈希计算等常见操作，能够简化代码并减少空指针异常（NullPointerException）的风险。

## 源码



## 作用

### 1、简化对象操作

提供常用的工具方法，避免重复编写样板代码。

### 2、增强空安全性

在方法中内置空值检查，减少空指针异常。

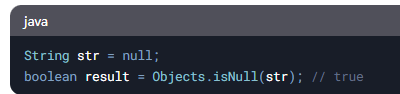
### 3、提高代码可读性

使用标准化的工具方法，使代码更简洁易懂。

## 方法

### public static boolean isNull(Object obj)

判断对象是否为 null



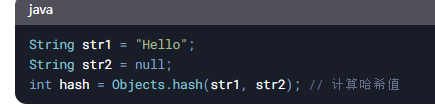
### public static <T> T requireNonNull(T obj)

检查对象是否为 null，如果是 null，则抛出 NullPointerException。



### public static int hash(Object... values)

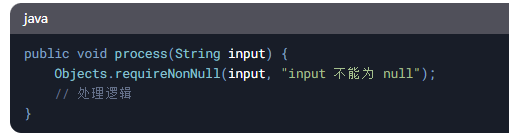
为多个对象生成哈希值，避免手动处理 null



## 使用场景

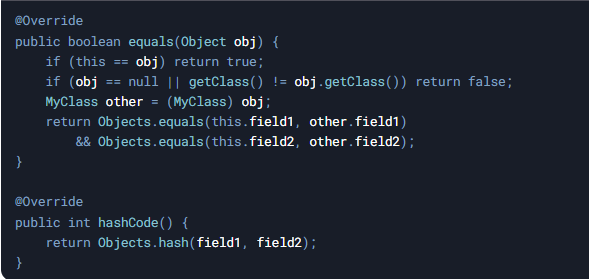
### 1、空值检查

在方法中检查参数是否为 null，避免空指针异常



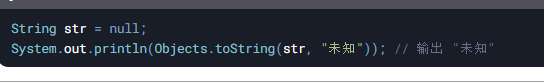
### 2、对象比较

在实现 equals 和 hashCode 方法时，使用 Objects.equals 和 Objects.hash 简化代码。



### 3、日志与调试

在日志或调试信息中安全地输出对象的字符串表示。



## 优点

### 1、****减少空指针异常****

内置空值检查，避免手动判空

### ****2、代码简洁****

提供标准化的工具方法，减少样板代码

### **3、**提高可读性****

使用工具方法使代码更清晰易懂

# System类

## 方法

### public static native long currentTimeMillis();

获取当前系统时间的时间戳（以毫秒为单位）

# Type接口

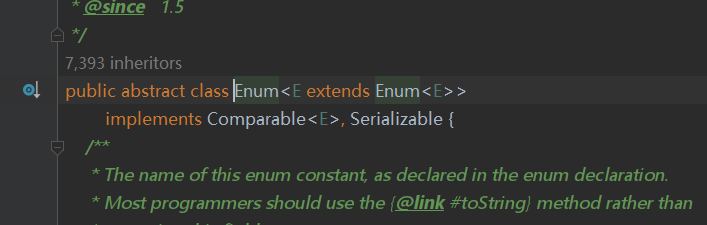
## 基本概念

# Enum类

## 基本概念

在Java中，Enum是一个抽象类，它是所有枚举类型的根类。

## 源码



## 方法

### name()

返回枚举常量的名称，即声明时的标识符。

### ordinal()

返回枚举常量在枚举声明中的位置索引，从0开始计数。

### toString()

返回枚举常量的字符串表示，通常是 name() 方法的返回值。

### compareTo()

用于比较两个枚举常量的顺序。它通过比较两个枚举常量的 ordinal() 值来确定它们的顺序关系。

### public static <T extends Enum<T>> T valueOf(Class<T> enumType,String name)

根据枚举常量的名称获取对应的枚举常量

## 特性

### 1、类型安全

enum 提供了类型安全性，只有定义好的常量可以被使用，防止了意外的值。

### 2、自动生成方法

每个枚举值都是该枚举类的一个实例，Java 会自动为每个枚举值生成 toString() 方法。

### 3、实现接口

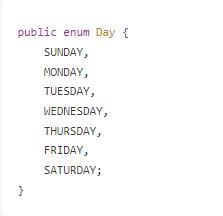
枚举可以实现接口，但不能继承其他类（因为所有的枚举隐式地继承了 java.lang.Enum）。

### 4、构造函数

可以给枚举定义构造函数并传递参数，使得枚举更加灵活。

## 定义枚举

在 Java 中，使用 enum 关键字定义枚举类型。例如定义一个代表星期几的枚举类。



## 枚举的高级用法

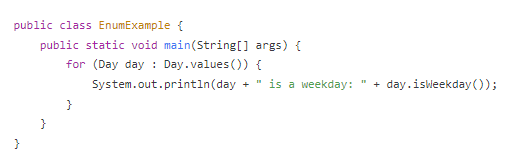
### 带参数的枚举

你可以为枚举的每个常量定义属性。例如，我们可以为星期几的枚举添加一个表示工作日的属性：



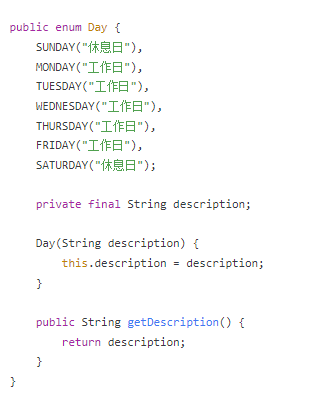
### 使用枚举的方法

使用上述包含布尔属性的 Day 枚举，可以编写如下代码：



### 添加方法和字段

枚举不仅可以有字段和构造函数，还可以添加方法。例如，返回枚举的描述信息：



## 使用场景

### 1、状态管理

例如，订单状态（待处理、已发货、已完成等）。

### 2、配置选项

例如，应用程序的不同模式（开发、测试、生产）。

### 3、固定集合

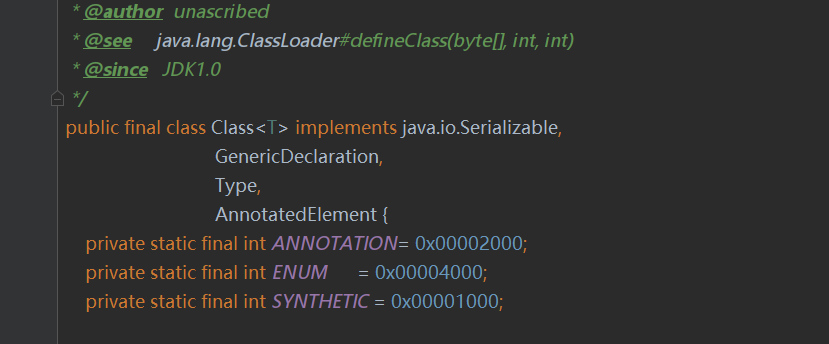
如颜色、方向、星期等。

# Class类

## 基本概念

在 Java 中，Class 类是反射机制的核心部分，代表了一个类或者接口的运行时对象。通过 Class 类，Java 程序可以动态地获取类的信息、创建对象、调用方法、访问属性等。

## 源码



## 方法

### 获取类的信息

#### String getName()

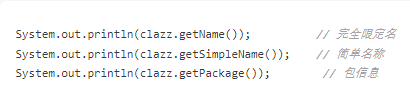
返回类的完全限定名

#### String getSimpleName()

返回类的简单名称

#### Package getPackage()

返回类所属包的信息。



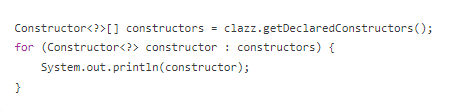
### 获取构造方法

#### Constructor<?>[] getConstructors()

获取 public 构造方法

#### Constructor<?>[] getDeclaredConstructors()

获取所有构造方法（包括私有）



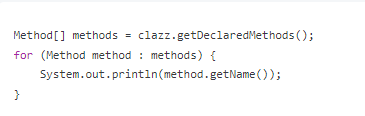
### 获取方法

#### public Method[] getMethods()

获取public方法。

#### public Method[] getDeclaredMethods()

获取所有方法（包括私有）。



#### public Method getDeclaredMethod(String name, Class<?>... parameterTypes)

与指定名称和参数匹配的此类方法的对象

### 获取字段

#### public Field[] getFields()

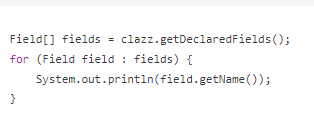
获取public字段

#### public Field getDeclaredField(String name)

根据名称获取指定的字段

#### public Field[] getDeclaredFields()

获取所有字段（包括私有）



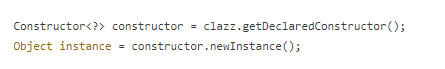
### 创建对象

#### public T newInstance()

通过 Class 对象可以创建类的实例。

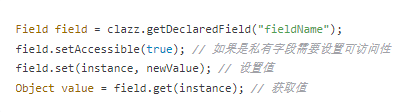


**注意**：newInstance() 方法已经被弃用，推荐使用 getDeclaredConstructor().newInstance()。



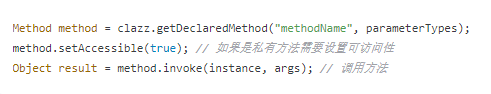
### 访问属性

通过Field 对象可以访问和修改类的属性。



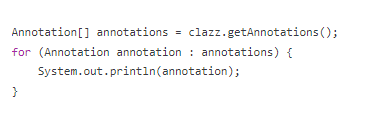
### 调用方法

通过 Method 对象可以调用类的方法



### 注解处理

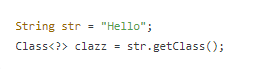
使用 Class 对象可以获取类上的注解



## 获取Class对象的方法

### 1、使用 对象.getClass()

每个对象都有一个 getClass() 方法，可以通过实例对象获取其类的 Class 对象。



### 2、使用 Class.forName()

可以通过类名字符串获取 Class 对象。



### 3、使用.class语法

可以直接使用类名加 .class 获取 Class 对象。



### 4、通过 ClassLoader.loadClass("全限定类名")

特点：

1、通过类加载器加载类，不会初始化静态块（与Class.forName()的区别）。

2、适用于需要控制类加载过程的场景（如热部署）。

示例：



**与**Class.forName()**的区别**



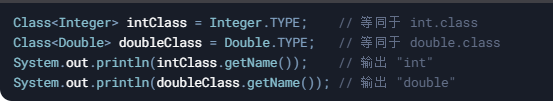
### ****5、通过包装类的****TYPE****字段（基本类型）****

特点：

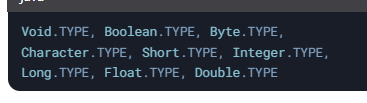
1、获取基本数据类型的Class对象。

2、Java为每个基本类型提供了包装类的TYPE字段。

示例：



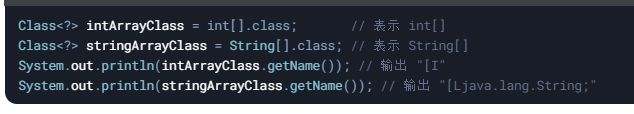
支持的TYPE字段：



### 6、其他特殊场景

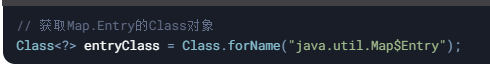
#### （1）****数组类型的Class对象****

通过元素类型[].class获取



#### **（2）**通过反射获取嵌套类****

使用$符号表示嵌套类：



### 注意事项

## 总结

Class 类是 Java 反射的重要组成部分，提供了操作类及其成员的功能。通过反射，开发者可以实现动态类型检查、动态方法调用和对象创建等功能，极大地增强了 Java 程序的灵活性。然而使用反射也会带来性能开销和安全性问题，因此应谨慎使用。在实际开发中，反射常用于框架设计、依赖注入、序列化等场景。

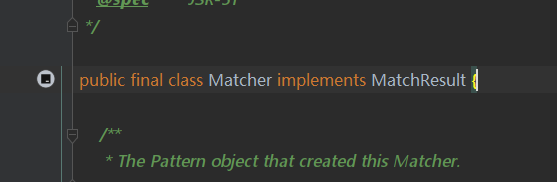
# ClassLoader类

# Matcher类

## 基本概念

Matcher 是 Java 正则表达式 API 中的核心类，用于对输入字符串执行匹配操作。它是通过 Pattern.matcher() 方法创建的，提供了丰富的匹配和查找功能。

## 源码



## 方法

### public boolean find()

查找匹配



### public boolean matches()

完全匹配



### public boolean lookingAt()

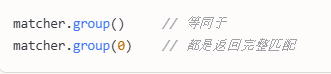
从开头匹配



### public String group()

返回**上一次匹配成功的完整字符串**

等价关系



使用示例：



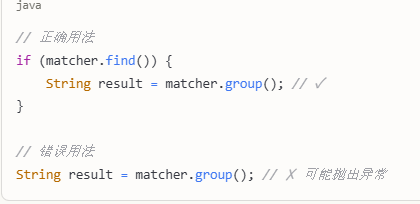
调用前提

**必须先调用匹配方法：**

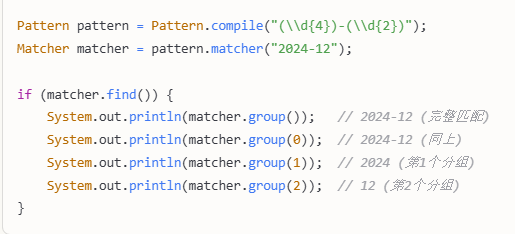
find() - 查找匹配

matches() - 完全匹配

lookingAt() - 开头匹配



与分组的关系：



核心要点：

1、返回**完整的匹配结果**

**2、必须在成功匹配后调用。**

**3、**group()就是group(0) 的简写

4、如果没有匹配成功就调用会抛出异常

简单说：group() 就是"给我刚才找到的那个完整匹配"。

### public String group(int group)

## 使用流程



# RandomStringUtils类

## 方法

### public static String randomAlphabetic(int count)

# DigestUtils类

# Date类

## 基本概念

Date类用于表示日期和时间，它位于java.util包中。很多与日期时间有关的类都能够转换为long类型整数。

示例代码

Date now = new Date();

long time = System.currentTimeMillis();

Date date = new Date(time);

## 构造函数

### （1）Date()

无参的构造函数

### （2）Date(long date)

传递一个长整型数字

## 比较Date对象

### after方法

用于测试日期对象是否在指定日期之后。

方法声明：

public boolean after(Date when)

when:是要比较的日期对象。

Date date = new Date();

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

//解析日期

Date date1 = sdf.parse("2011-05-12 15:16:00");

if(date.after(date1)){

System.out.println("之后");

}else{

System.out.println("之前");

}

### before方法

用于测试日期对象是否在指定日期之前。

方法声明：public boolean before(Date when)

## 更改Date对象

getTime()和setTime()方法，分别用于设置和获取Date对象的毫秒数。

## 日期的格式化输出

使用String.format()方法通过特殊转换符作为参数可以实现对日期和时间的格式化。

示例：

Date now = new Date();

String time = String.format("%td",now);

System.out.println(time);

## 常见的日期时间转换符

%te 一个月中的某一天 %td 一个月中的第几天

%ta 指定语言环境中的星期几简称

%tb 指定语言环境中的月份简称 %tm 月份

%tc 包含全部日期和时间信息

%tY 4位年份

%td 获取当月的日期

## 时间格式化转换符

%tH 2位数字的24时值的小时

%tI 2位数字的12时值的小时

%tM 2位数字的分钟（00-59）

%tS 2位数字的秒数 （00-60）

%tL 3位数字的毫秒数（000-999）

%tp 语言环境中的上午或下午标记

## 常见的日期时间格式组合

%tF "年-月-日"格式（4位年份）

%tD "月/日/年"（2位年份）

%tc 全部日期和时间格式

%tr "时：分：秒PM(AM)"(12时制)

%tT "时：分：秒"格式（24时值）

%tR "时：分"格式（24时值）

# Calendar类

## 从date对象中获取月份

Calendar cal = Calendar.getInstance();

cal.setTime(new Date());

int month = cal.get(Calendar.MONTH);

System.out.println(month);

# Integer类

## 基本概念

Integer类是Java中的一个包装类，它封装了一个int基本数据类型的值。Integer类提供了对int值的各种操作和方法。Integer类位于java.lang包中，并且它是Number类的子类。它是不可变的，即一旦创建了一个Integer对象，其值不能更改。

## 构造方法

### Integer(int value)

创建一个表示指定int值的Integer对象。

### Integer(String s)

创建一个表示由指定字符串所解析的int值的Integer对象。字符串必须是一个有效的整数表示。

## 静态常量

### Integer.MAX\_VALUE

表示int类型的最大值（2^31 - 1）。

### Integer.MIN\_VALUE

表示int类型的最小值（-2^31）。

### Integer.SIZE

int类型的位数（32位）。

### Integer.BYTES

int类型的字节数（4字节）。

## 静态方法

### Integer.parseInt(String s)

将字符串解析为int类型。

### Integer.valueOf(String s)

将字符串转换为Integer对象。

### Integer.valueOf(int i)

将int值转换为Integer对象。

### Integer.toString(int i)

将int值转换为字符串。

### Integer.compare(int x, int y)

比较两个int值。

## 实例方法

### int intValue()

返回Integer对象的int值。

### boolean equals(Object obj)

比较Integer对象是否与另一个对象相等。

### int compareTo(Integer anotherInteger)

比较当前Integer对象与指定Integer对象的大小。

## 常用功能

### 自动装箱和拆箱

Java提供了自动装箱和拆箱功能，即int类型可以自动转换为Integer对象，反之亦然。例如：

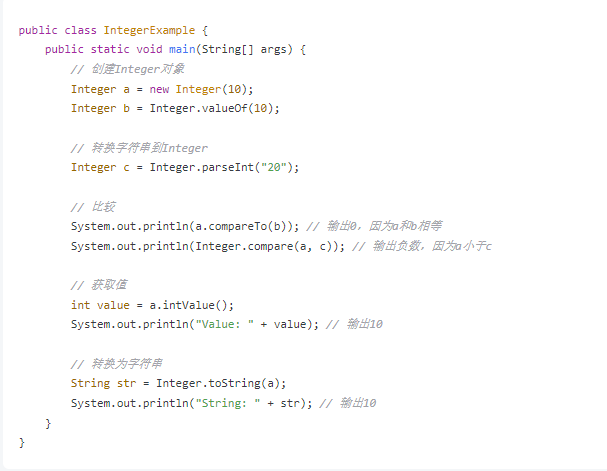
Integer a = 5; // 自动装箱

int b = a; // 自动拆箱

### 缓存机制

Integer类在-128到127范围内的Integer对象会被缓存。这个缓存机制优化了内存使用和性能。

## 示例代码

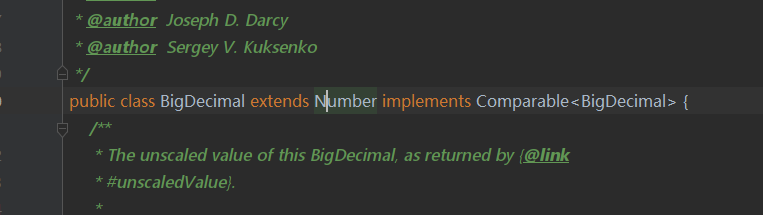


# BigDecimal类

## 基本概念

BigDecimal 类是 Java 中用于高精度计算的类，它提供了任意精度的浮点数表示和运算。在实际开发中，特别是涉及到财务计算或者需要精确控制小数点位数的场景下，使用 BigDecimal 可以避免浮点数运算带来的精度问题。

## 源码



## 主要特点和用途

### 1、任意精度

BigDecimal 可以表示任意精度的有符号十进制数。它不受标准浮点数精度的限制，因此适合于需要高精度计算的场景。

### 2、精确的小数计算

在金融领域或其他需要精确小数计算的场景中，使用 BigDecimal 可以确保计算结果的精度和准确性。

### 3、不可变性

BigDecimal 对象一旦创建，其值不能被修改。任何数学运算都会返回一个新的 BigDecimal 对象。

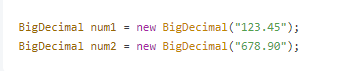
### 4、舍入模式

BigDecimal 支持多种舍入模式，例如四舍五入、向上取整、向下取整等，可以根据需要指定舍入规则。

## 构造方法

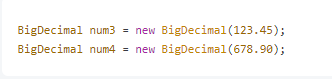
### BigDecimal(String val)

通过字符串构造 BigDecimal 对象。



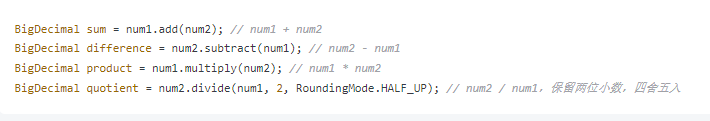
### BigDecimal(double val)

通过 double 类型构造 BigDecimal 对象。注意，这种方式不完全精确，建议优先使用字符串构造方法。



## 基本数学运算

BigDecimal 提供了 add(), subtract(), multiply() 和 divide() 方法进行基本的数学运算。这些方法返回一个新的 BigDecimal 对象，不会修改原对象。



### public BigDecimal add(BigDecimal augend)

加法运算

### public BigDecimal subtract(BigDecimal subtrahend)

减法运算

### public BigDecimal multiply(BigDecimal multiplicand)

乘法运算

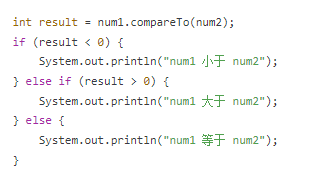
### public BigDecimal divide(BigDecimal divisor)

除法运算

## 比较方法

### compareTo(BigDecimal val)

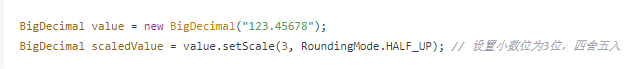
比较当前 BigDecimal 对象与指定对象的大小关系，返回一个整数，表示两者的比较结果。



## 数值操作方法

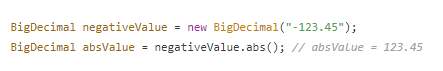
### setScale(int newScale, RoundingMode roundingMode)

设置小数位数并进行舍入，返回一个新的 BigDecimal 对象。



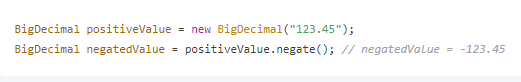
### abs()

返回当前 BigDecimal 对象的绝对值。



### negate()

返回当前 BigDecimal 对象的相反数。



## 转换方法

### intValue()

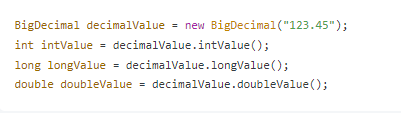
将BigDecimal 对象转换为int

### longValue()

将BigDecimal 对象转换为long

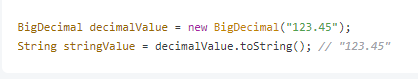
### doubleValue()

将BigDecimal 对象转换为double



### toString()

将 BigDecimal 对象转换为字符串表示形式。



# Math类

包含了所有用于数学运算的函数方法，这些方法都是静态的，所以每个方法只要使用"Math.数学方法"就可以调用。

## 主要方法

（1）double abs(double a)：返回double值的绝对值

（2）float abs(float a)：返回float值的绝对值

（3）int abs(int a)：返回Int值的绝对值

（4）long abs(long a)：返回long值的绝对值

（5）double cbrt(double a)：返回double值的立方根。

（6）double **max(double a,double b)**：返回两个double值中较大的一个。

（7）float max(float a,float b)：返回两个float值中较大的一个。

（8）long max(long a,long b)：返回两个long值中较大的一个。

（9）int max(int a ,int b)：返回两个int值中较大的一个。

（10）double **random()**：返回带正号的double值，该值大于等于0.0且小于1.0。返回一个随机数。

（11）long **round**(double a)：返回最接近参数的long。

（12）double **ceil**(double a)：返回大于等于a的最小整数。

（13）double **floor**(double a)：返回小于等于a的最大整数。

### public static int toIntExact(long value)

用于将长整型（long）数据类型的值转换为整型（int）数据类型，并且在转换过程中会检查是否发生溢出。

# Random类

## 基本概念

java.util.Random 是 Java 中用于生成伪随机数的类。它提供了多种方法生成随机数，包括整数、浮点数、布尔值等。Random 类广泛应用于模拟、游戏、测试等场景。

## 源码



## 方法

### nextInt()

生成一个随机的 int 值（范围是所有可能的 int 值）

### nextInt(int bound)

生成一个 [0, bound) 范围内的随机整数。

### nextLong()

生成一个随机的 long 值。

### nextFloat()

生成一个 [0.0, 1.0) 范围内的随机浮点数。

### nextDouble()

生成一个 [0.0, 1.0) 范围内的随机双精度浮点数。

### nextBoolean()

生成一个随机的 boolean 值（true 或 false）

### nextBytes(byte[] bytes)

生成随机字节并填充到指定的字节数组中。

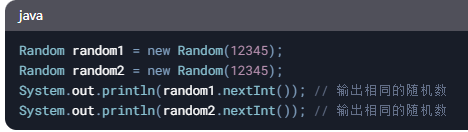
### synchronized public void setSeed(long seed)

设置随机数生成器的种子。

## 注意事项

### 1、种子的作用

如果使用相同的种子创建 Random 对象，生成的随机数序列是相同的。



### 2、线程安全性

1、Random 类是线程安全的，但多线程环境下可能会存在性能问题。

2、如果需要高性能的多线程随机数生成，可以使用 ThreadLocalRandom 或 java.util.concurrent.ThreadLocalRandom。

### 3、随机数的分布

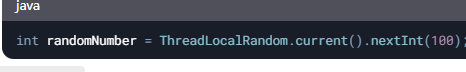
1、Random 生成的随机数是均匀分布的。

2、如果需要其他分布（如正态分布），可以使用 java.util.random.RandomGenerator（Java 17 引入）。

## Random****类的替代方案****

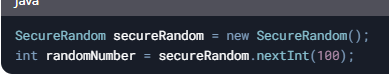
### 1、ThreadLocalRandom

适用于多线程环境，性能优于 Random。



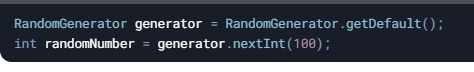
### 2、SecureRandom

提供加密强度更高的随机数生成器，适用于安全敏感的场景。



### 3、java.util.random.RandomGenerator（Java 17+）

提供了更灵活的随机数生成器接口，支持多种随机数算法。



## （1）通过Math类生成随机数（一般常用这个）

random()方法默认生成大于等于0、小于1.0的double型随机数。

m+(int)(Math.random()\*n) 返回大于等于m小于m+n之间的随机数。

## （2）使用Random类生成随机数

Random random = new Random();

在Random类中提供了获取各种数据类型随机数的方法。

nextInt() 返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。（不是0到1的数）

nextInt(int n) 返回一个伪随机数，它是取自此随机数生成器序列的、在 0（包括）和指定值（不包括）之间均匀分布的 int 值。

# SimpleDateFormat类

# DateTimeFormatter类

## 基本概念

DateTimeFormatter 类是用于格式化和解析日期时间对象的类。它提供了一组预定义的模式（pattern）和方法，使得我们可以灵活地处理日期时间的显示和解析。DateTimeFormatter 类是线程安全的，因此可以在多个线程中共享使用。它是通过调用 ofPattern()、ofLocalizedDateTime() 或 ofLocalizedDate() 等静态方法来创建实例的。

## 示例

### 1、创建自定义的日期时间格式



上述代码创建了一个自定义的格式化器，用于将日期时间对象格式化为形如 "yyyy-MM-dd HH:mm:ss" 的字符串。

### 2、使用预定义的日期时间格式



# NumberFormat

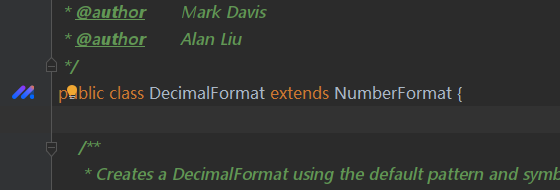
# DecimalFormat

## 基本概念

DecimalFormat是NumberFormat的一个子类，用于格式化十进制数字，通过该类可以为要输出的数字加上单位或控制数字的精度。一般情况下可以实例化Decimal对象时传递数字格式，也可以通过DecimalFormat类的applyPattern()方法来实现数字格式化。

DecimalFormat 类是 Java 中用于格式化数字的一个非常有用的类，它位于 java.text 包中。这个类可以让你以指定的格式来显示数字，特别是在控制小数点后的位数、千位分隔符、货币符号等方面非常有用。它非常适合用于格式化浮动数值，比如货币、百分比等

## 源码



## 常见的模式字符

### 0和#（数字）

0 表示数字的占位符。如果一个数字在该位置上没有值，则用 0 填充。

# 也表示占位符，但如果该位置上没有数字，就不显示任何字符。

例如：

模式 "0000" 格式化数字 45 为 "0045"。

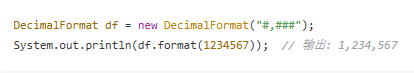
模式 "#.##" 格式化数字 123.4 为 "123.4"，格式化 123.456 为 "123.46"（保留两位小数）

### .(小数点)

小数点用于分隔整数部分和小数部分。例如，"#.##" 只保留最多两位小数

### ,(千位分隔符)

使用逗号 , 可以在整数部分添加千位分隔符。例如，使用模式 "#,###" 会把数字 1234567 格式化为 1,234,567



E 代表分隔科学计数法中的尾数和指数。

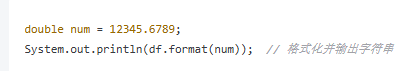
% 代表百分数

\u2030 代表千分数

## 方法

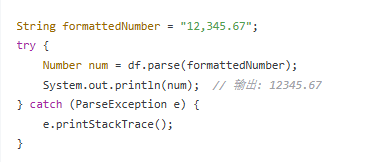
### public final String format(double number)

格式化数字，返回格式化后的字符串。



### public Number parse(String source)

将格式化的字符串转换回数字（Number 类型）



# ResourceBundle类

这个类用于读取属性文件.properties的内容。

## 常用的方法

ResourceBundle.getBundle("")：

//返回当前指定的资源包

ResourceBundle.getBundle("").getString("");

//返回当前资源包中指定字符串的值

实例：

创建config.properties属性文件

name=jinshihui

age=24

String str = ResourceBundle.getBundle("config").getString("name");//jinshihui

String str1 = ResourceBundle.getBundle("config").getString("age");//24

# Arrays类

## 基本概念

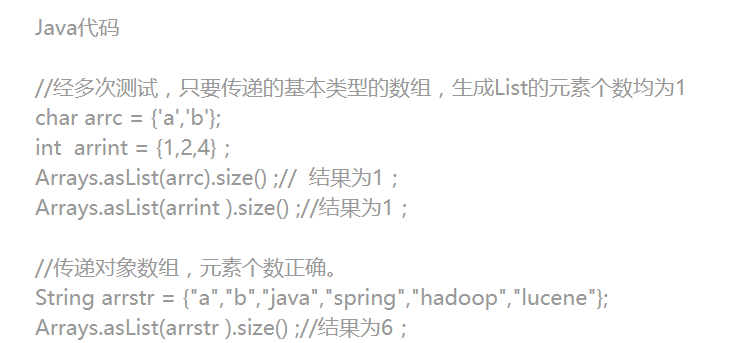
[Java](http://lib.csdn.net/base/java).util.Arrays类能方便地操作数组，它提供的所有方法都是静态的。具有以下功能：

1. 给数组赋值，通过fill方法。
2. 给数组排序，通过sort方法，升序排序。
3. 比较数组，通过equals方法比较数组中元素的值是否相等。
4. 查找数组元素，通过binarySearch方法能对排序好的数组进行二分查找法操作。

## 方法

### static List asList()

将数组转换为List集合，这个数组类型必须是引用类型，不能是8种基本数据类型。这个放过比较重要。



### int binarySearch(Object[] a,Object key)

使用二分搜索法来搜索指定数组，以获得指定对象。

参数：a,要搜索的数组，key，要搜索的值。

返回：如果它包含在数组中，则返回搜索键的索引；否则返回 (-(插入点) - 1)。

示例：

**int**[] a= {1,2,3,4};

**int** num = Arrays.*binarySearch*(a, 3);

System.***out***.println(num);//2

### T[] copyOf(T[] original,int newLength)

//复制指定的数组，截取或用 null 填充（如有必要），以使副本具有指定的长度。

参数：original，要复制的数组，newlength，要返回的副本的长度。

返回：原数组的副本，截取或用 null 填充以获得指定的长度

示例：

**int**[] a= {1,2,3,4};

**int**[] num = Arrays.*copyOf*(a, 3);

System.***out***.println(Arrays.*toString*(num));

### boolean[] copyOfRange(boolean[] original,int from,int to)

//将指定数组的指定范围复制到一个新数组。该范围的初始索引 (from) 必须位于 0 和 original.length（包括）之间

参数：original,要复制的数组，from 要复制的范围的初始索引（包括），to，要复制的最后索引（不包括））（此索引可能位于数组范围之外）

返回：包含取自原数组指定范围的新数组，截取或用 false 元素填充以获得所需长度。

示例：

**int**[] a = {1,2,3,4};

**int**[] b = Arrays.*copyOfRange*(a, 0,3);

System.***out***.println(Arrays.*toString*(b));

### boolean deepEquals(Object[] a,Object[]b)

//如果两个指定数组彼此是深层相等 的，则返回 true。与 [equals(Object[],Object[])](mk:@MSITStore:C:\Users\jinshihui\Desktop\我的编程知识总结\Java\JDK_API_1_6_zh_CN.CHM::/java/util/Arrays.html#equals(java.lang.Object[], java.lang.Object[])) 方法不同，此方法适用于任意深度的嵌套数组。

### boolean equals(Object[] a,Object[]b)

//如果两个指定的 Objects 数组彼此相等，则返回 true。如果两个数组包含相同数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。

说明：

1、deepEquals用于判定两个指定数组彼此是否深层相等，此方法适用于任意深度的嵌套数组。

 2、equals用于判定两个数组是否相等，如果两个数组以相同顺序包含相同元素，则返回true，否则返回false。

3、如果两个数组使用equals返回true，则使用deepEquals也返回true，也就是说在比较的两个数组均为一维数组的前提下，equals和deepEquals的比较结果没有差别。

4、如果要比较多维数组，则需要使用deepEquals方法。

### public static <T> Stream<T> stream(T[] array)

stream() 方法用于将数组转换为一个 Stream 对象。Stream 是 JDK 8 中引入的一个新特性，它提供了一种流式处理数据的方式。stream() 方法接受一个数组作为参数，并返回一个 Stream 对象。该 Stream 对象可以用于对数组中的元素进行各种操作，比如过滤、映射、排序等。

示例：



上面的代码中，我们创建了一个包含三个字符串的数组，并使用 Arrays.stream() 方法将该数组转换为一个 Stream 对象。最终得到的 stream 可以用于对数组中的元素进行各种操作。通过 Stream 接口提供的丰富的方法，我们可以对数组中的元素进行过滤、映射、排序等操作，甚至可以进行并行处理以提高性能。

# List.of() vs Arrays.asList()比较

## Arrays.asList()

### 基本概念

Arrays.asList() 方法是由 java.util.Arrays 类提供的工具。它将数组转换为由原始数组支持的固定大小列表。

### 示例



### 关键特性

1、固定大小列表： 列表大小是固定的，不能添加或删除元素，但可以修改现有元素。

2、由数组支持： 对列表的修改会反映到数组中，反之亦然。

3、允许空元素： 列表中可以包含 null 值。

### 潜在问题

尝试添加或删除元素将抛出 UnsupportedOperationException。



允许修改元素，且会影响原始数组：



## List.of()

### 基本概念

List.of() 是Java 9中引入的静态工厂方法，用于创建不可修改的列表。

### 示例



### 关键特性

1、不可修改列表： 列表无法被修改，任何修改操作都会导致异常。

2、不由数组支持： 它是一个独立的列表，修改不会影响任何底层数组。

3、不允许空元素： 插入 null 会抛出 NullPointerException。

### 潜在问题

任何修改尝试都会抛出 UnsupportedOperationException。



尝试包含 null 元素：



### 并列比较



## 什么时候使用各方法

### 使用 Arrays.asList()

你需要一个由数组支持的固定大小列表。

你可能需要修改现有元素。

你需要在列表中包含 null 元素。

你希望列表中的更改反映到原始数组中。

### 使用 List.of()

你需要一个不可修改的列表。

你想防止对列表的修改。

你不需要 null 元素。

你处理的是不可修改的数据集。

## 如何影响你的集合

了解这些方法有助于你做出明智的选择：

性能： 由于线程安全和减少防御性复制，不可修改的列表可能具有更高的性能。

安全性： 防止在多线程环境中意外修改。

清晰性： 明确声明不可修改性使代码更易于理解和维护。

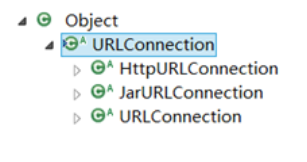
## 总结

通过深入分析 Arrays.asList()和List.of()的特点和差异，我们可以看出，尽管它们都是用于将数组转换为列表的工具，但它们在可变性、空值处理、以及与底层数组的关系等方面有着截然不同的设计理念。Arrays.asList()适用于需要一个固定大小、可以修改元素但无法添加或删除元素的场景，且允许null值的存在；而List.of()更强调不可变性，适合需要不可修改的数据集合，并且对null元素敏感。在选择这两种方法时，开发者不仅要考虑代码的可读性，还要注重数据的安全性与性能优化。通过合理选择合适的方法，我们可以编写出更加健壮、高效的代码，从而有效减少错误，提高应用程序的稳定性和可维护性。

# URLConnection类

这是一个抽象类，表示指向URL指定资源的活动连接，这个类在java.net包下，URLConnection类本身依赖于Socket类实现网络连接。一般认为，URLConnection类提供了比Socket类更易于使用、更高级的网络连接抽象。

URLConnection继承树：



URLConnection类声明为抽象类，除了connect()方法，其他方法都已经实现，URLConnection的三个实现子类都位于sun.net.www包下。

## 构造方法

protected URLConnection(URL url)：

//构造一个指定的URL连接。

connect()方法由子类实现本地与服务器的连接方式。一般使用TCP socket，但也可以使用其他机制来建立。

abstract  void connect()：

//打开到此 URL 引用的资源的通信链接（如果尚未建立这样的连接）

当派生URLConnection子类时，通常会覆盖URLConnection的其他方法，使其返回有意义的值。

## 普通方法

### String getContentType()

//返回 Content-type 头字段的值。即数据的MIME内容类型。若类型不可用，则返回null。除了HTTP协议，极少协议会使用MIME首部。若内容类型是文本。则Content-type首部可能会包含一个标识内容编码方式的字符集。

例：Content-type:text/html; charset=UTF-8

### int getContentLength()

//返回 Content-length 头字段的值，该方法获取内容的字节数。许多服务器只有在传输二进制文件才发送Content-length首部，在传输文本文件时并不发送。若没有该首部，则返回-1。若需要知道读取的具体字节数，或需要预先知道创建足够大的缓冲区来保存数据时，可以使用该方法。

### String getContentEncoding()

//返回 Content-encoding 头字段的值，获取内容的编码方式。若内容无编码，则该方法返回null. 注意：Content-encoding(内容编码)与字符编码不同。字符编码方式由Content-type首部或稳定内容的信息确定，它指出如何使用字节指定字符。内容编码方式则指出字节如何编码其他字节。

### String getHeaderField(String name)

//返回指定头字段的值

### Map<String,List<String>> getHeaderFields()

//返回头字段不可修改的Map。

## 配置连接

### protected  URL url

该字段指定URLConnection连接的URL。构造函数会在创建URLConnection时设置，此后就不能再改变。

### protected  boolean connected

当连接打开时，该字段为true，若连接关闭，则为false。没有直接读取或改变connected值的方法。任何导致URLConnection连接的方法都会将此变量设为true，包括connect()，getInputStream()，和getOutputStream()。任何导致URLConnection断开的方法都会将此字段设为false。java.net.URLConnection没有断开方法，但一些子类，如java.net.HttpURLConnection中存在这样的方法。

### protected  boolean allowUserInteraction

该字段表示是否允许与用户交互，默认为false，即不允许交互。该值只能在URLConnection连接前设置。例：Web浏览器可能需要访问用户名和口令。

以上字段定义了客户端如何向服务器作出请求。这些字段只能在URLConnection连接前修改，否则抛出IllegalStateException异常。

## 关于超时

Java 1.5增加了如下4个方法，允许查询和修改连接的超时值。即底层socket在抛出SocketTimeoutException异常前等待远端响应时间。

### int getReadTimeout()

//获取超时值，单位为毫秒

### void setReadTimeout(int timeout)

//设置连接的超时值，单位为毫秒。

### int getConnectTimeout()

//获取连接超时的时间

### void setConnectTimeout(int timeout)

//设置连接超时的时间

这四个方法将0解释为永不超时。如果超时值为负数。两个设置方法都会抛出IllegalArgumentException异常。

## 配置HTTP首部

客户端向服务器发送请求，不仅会发送请求行，还发送首部。Web服务器可以根据此信息向不同客户端提供不同的页面，获取和设置cookie，通过口令设置用户等等。通过在客户端发送和服务器响应的首部中放置不同的字段，就可以完成这些工作。

每个URLConnection的具体子类都在首部中设置一些默认的键值对（实际上只有HttpURLConnection这样做，因为HTTP是唯一以这种方式使用首部的主要协议。）通过以下方法可以设置URLConnection在打开连接增加HTTP首部信息。

### void setRequestProperty(String key, String value)

//此方法只在连接打开之前，否则将会抛出IllegalStateException异常。但该方法仅支持对一个首部设置一个值。HTTP允许一个属性有多个值。各个值之间用分号隔开。例：cookie:username=sjq; session=hdfu23asdjf901j;

### void addRequestProperty(String key, String value)

//该方法用于添加某个指定首部字段的值。

### String getRequestProperty(String key)

//获取首部字段的值

### Map<String,List<String>> getRequestProperties()

//用于获取URLConneciton所拥有HTTP首部。

# HttpURLConnection类

# BASE64Encoder

这个类是用于base64加密的。

代码如下：

String str = "jinshihui";

BASE64Encoder encoder = new BASE64Encoder();

str = encoder.encode(str.getBytes("UTF-8"));

System.out.println(str);

# BASE64Decoder

这个类是用于base64解码的。

代码如下：

BASE64Decoder decoder = new BASE64Decoder();

byte[] bytes = decoder.decodeBuffer(str);

System.out.println(new String(bytes,"UTF-8"));

说明：从JDK 1.8开始，就提供了java.util.Base64.Decoder和java.util.Base64.Encoder的JDK公共API，可代替sun.misc.BASE64Decoder和sun.misc.BASE64Encoder的JDK内部API，代码如下：

//这是base64位编码

String str1= "jinshihui";

Encoder encoder2 = java.util.Base64.getEncoder();

String str2 = encoder2.encodeToString(str1.getBytes("UTF-8"));

System.out.println(str2);

//这是base64位解码

Decoder decoder2 = java.util.Base64.getDecoder();

byte[] bytes2 = decoder2.decode(str2);

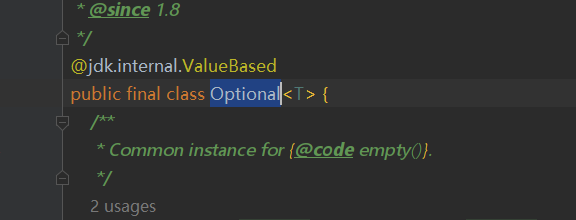
System.out.println(new String(bytes2,"UTF-8"));

# Optional类

## 基本概念

Optional类实际上是个容器，它可以保存类型T的值，或者仅仅保存null。Optional 类的引入很好的解决空指针异常。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。尽量避免在程序中直接调用Optional对象的get()和isPresent()方法，避免使用Optional类型声明实体类的属性。

## 源码



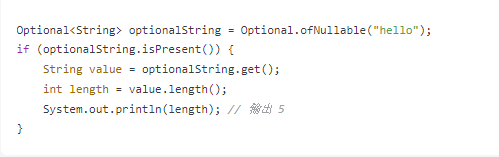
## 方法

### public T get()

用于获取 Optional 对象中的值。它会返回被包装在 Optional 对象中的值，如果该值存在的话。

示例：

假设有一个 Optional<String> 对象，表示一个字符串值，我们想要获取这个字符串的长度。可以使用 get() 方法实现：



在上述示例中，我们创建了一个 Optional<String> 对象，该对象包含了字符串 "hello"。然后通过调用 isPresent() 方法来检查 Optional 对象是否有值。如果有值，则通过 get() 方法获取实际的值，并进行相应的操作。

注意：

1、如果 Optional 对象为空（即没有值），调用 get() 方法会抛出 NoSuchElementException 异常。因此在使用 get() 方法之前应该先使用 isPresent() 方法检查 Optional 对象是否有值。

2、避免频繁地使用 get() 方法来获取值，因为这样可能会违反 Optional 的设计初衷，即显式地处理有可能为空的情况。

总之，get() 方法是用于从 Optional 对象中获取实际值的方法。但是在调用前需要确保 Optional 对象非空，以避免抛出异常。

### public static <T> Optional<T> ofNullable(T value)

若 t 不为 null,创建 Optional 实例，否则创建空实例。

### public static <T> Optional<T> of(T value)

创建一个 Optional 实例

### ofNullable()和of()方法的区别

1、ofNullable() 方法允许传入 null 值，如果传入的值为 null，则返回一个空的 Optional 对象。例如：

Optional<String> str1 = Optional.ofNullable(null); // 返回一个空的 Optional 对象

Optional<String> str2 = Optional.ofNullable("hello"); // 返回一个包含字符串 "hello" 的 Optional 对象

2、of() 方法要求传入的值不能为空，如果传入的值为 null，则会抛出 NullPointerException 异常。

例如：

Optional<String> str1 = Optional.of(null); //抛出NullPointerException 异常。

Optional<String> str2 = Optional.of("hello"); //返回一个包含字符串 "hello" 的 Optional 对象。

因此当你不确定传入的参数是否为空时，可以使用 ofNullable() 方法来创建 Optional 对象；当你确定传入的参数不为空时，可以使用 of() 方法来创建 Optional 对象。需要注意的是无论是使用 ofNullable() 还是 of() 方法创建的 Optional 对象都可能为空。因此在使用 Optional 对象时仍然需要进行空指针判断。

### public static<T> Optional<T> empty()

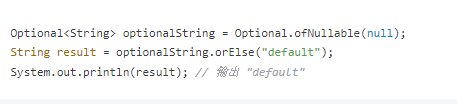
创建一个空的Optional实例

### public T orElse(T other)

用于获取 Optional 对象中的值，如果该值存在，则返回该值；如果值不存在，则返回一个默认值。

示例：

假设有一个 Optional<String> 对象，表示一个可能存在的字符串值，我们想要获取这个字符串值，如果不存在，则返回一个默认的字符串 "default"。可以使用 orElse() 方法实现。



在上述示例中，我们创建了一个 Optional<String> 对象，该对象为空（没有值）。然后通过使用 orElse() 方法，并传入默认值 "default"，来获取 optionalString 中的值。由于 optionalString 是空的，所以 orElse() 方法会返回默认值 "default"。需要注意的是，orElse() 方法的默认值参数 defaultValue 可以是任意类型，不一定需要与 Optional 对象中的值类型一致。

总之，orElse() 方法是用于获取 Optional 对象中的值的方法。如果值存在，则返回该值；如果值不存在，则返回一个默认值，这样可以方便地处理可能为空的情况。

### public T orElseGet(Supplier<? extends T> supplier)

如果存在值，则返回该值，否则返回由供应函数生成的结果，可以提供更灵活的默认值生成方式。

### orElse和orElseGet方法的区别

#### 1、调用时机

（1）orElse 方法无论 Optional 对象是否为空，都会创建默认值。

（2）orElseGet 方法只有在 Optional 对象为空时才会调用 Supplier 生成默认值。

#### 2、效率考虑

1、如果默认值是一个常量或者已经计算好的值，两者效率差别不大。

2、如果默认值的计算成本较高或者需要在需要时才生成，推荐使用 orElseGet 方法，避免不必要的计算开销。

#### 3、Supplier使用

（1）orElseGet 更适合使用 lambda 表达式或方法引用来生成默认值，能够延迟计算。

（2）orElse 更适合提供直接可用的默认值。

总结：

orElse 和 orElseGet 方法都是处理 Optional 对象中空值情况的方法，可以根据需要选择合适的方法来提供默认值。一般来说，推荐在需要时动态生成默认值时使用 orElseGet 方法，这样可以避免不必要的计算，提高效率。

### public boolean isPresent()

如果存在值返回true，否则返回false。

### public boolean isEmpty()

如果值不存在返回true，否则返回false。

### public void ifPresent(Consumer<? super T> consumer)

当 Optional 实例包含一个非空的值时（即值存在），则执行指定的操作。这个操作由传入的 Consumer 函数式接口决定，该接口定义了一个接收泛型类型 T 的参数的方法。

示例：



在这个例子中：

Optional.of("Alice") 创建了一个包含字符串 "Alice" 的 Optional 对象。

ifPresent(name -> System.out.println("Hello, " + name)) 表示如果 optionalName 中包含值，则执行 Consumer 中定义的操作，这里是打印 "Hello, Alice"

注意事项：

1、空值处理：如果 Optional 对象中的值为 null 或者 empty，则 ifPresent 方法不会执行任何操作，因为没有值可供消费。

2、替代方法：如果需要在值不存在时执行某些操作，可以考虑使用 orElse 或者 orElseGet 方法来提供一个默认值或者生成值的方法。

总结：

ifPresent(Consumer<? super T> consumer) 方法是 Optional 类中用来处理存在值时的操作的一种优雅方式。它允许你在避免显式的 null 检查的同时，以函数式编程的风格来执行特定的操作，从而使代码更加清晰和简洁。

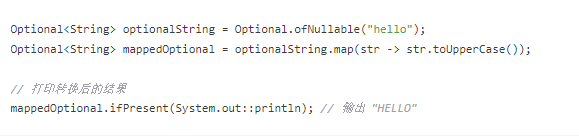
### public void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)

### public <U> Optional<U> map(Function<? super T, ? extends U> mapper)

  如果有值对其处理，并返回处理后的Optional，否则返回Optional.empty()

示例：

假设有一个 Optional<String> 对象，表示一个字符串值，我们想要将这个字符串转换为大写形式。可以使用 map() 方法实现：



在上述示例中，我们创建了一个 Optional<String> 对象，该对象包含了字符串 "hello"。然后，我们调用 map() 方法，并传入一个 Lambda 表达式，将字符串转换为大写形式。最后，通过使用 ifPresent() 方法检查并打印转换后的结果。

总之，map() 方法提供了一种便捷的方式来对 Optional 对象中的值进行转换操作，并将转换结果封装在新的 Optional 对象中。

### public <U> Optional<U> flatMap(Function<? super T, ? extends Optional<? extends U>> mapper)

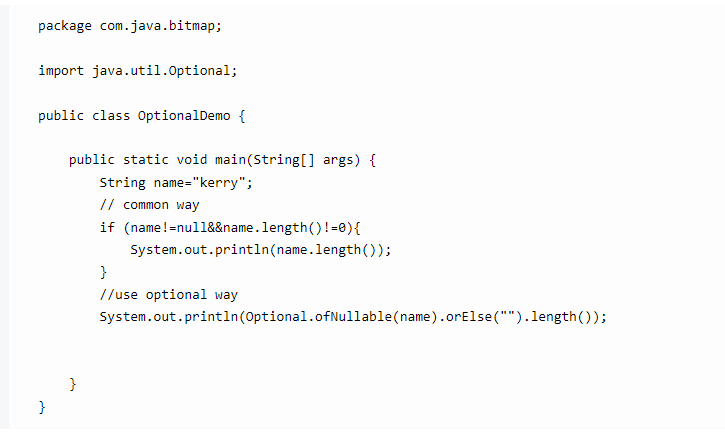
flatMap() 方法用于对包含在 Optional 对象中的值进行操作，并返回一个新的 Optional 对象。

## 代码示例

### 1、在函数方法中

## 使用场景

### 1、判断是否为空



### 2、遍历List

# Collectors类

## 基本概念

Collectors 类是 Java 8 引入的一个工具类，提供了一组静态方法，用于在流式操作中对元素进行收集和汇总。

## 作用

Collectors 类的主要作用是将流中的元素收集为不同类型的结果，例如集合（List、Set、Map等）、字符串、统计信息（如总数、平均值、最大值等）等。它提供了丰富的功能来处理和操作流的元素，使得收集和汇总数据变得更加简单和高效。

## 方法

### public static <T, K, U> Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,Function<? super T, ? extends U> valueMapper)

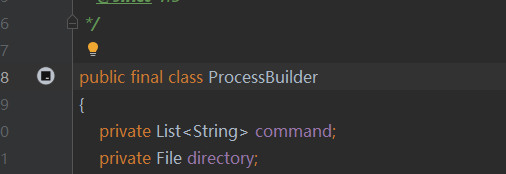
将流中的元素收集到一个 Map 集合中，其中 keyMapper 和 valueMapper 是函数，用于从元素中提取键和值。

# ProcessBuilder类

## 基本概念

ProcessBuilder类是Java中用于创建和管理操作系统进程的一个类。它提供了一个简单的接口来启动和管理子进程，包括设置进程的环境变量、工作目录、输入输出流等。ProcessBuilder 是 java.lang.Process的一个增强型 API，使得与系统进程交互变得更加灵活和方便。

## 源码



## 主要功能

### 1、创建和配置进程

您可以设置要启动的命令及其参数、工作目录和环境变量等。

### 2、获取输入输出流

可以访问子进程的标准输入、标准输出和错误流，以便与子进程进行交互。

### 3、控制进程

启动、等待进程完成、销毁进程等操作。

## 主要方法

### 构造函数

#### ProcessBuilder()

创建一个新的 ProcessBuilder 实例。

#### ProcessBuilder(List<String> command)

用指定的命令初始化一个新的 ProcessBuilder 实例。

### 配置方法

#### command(String... command)

设置要执行的命令及其参数。

#### directory(File directory)

设置子进程的工作目录。

#### environment()

获取子进程的环境变量。

#### redirectErrorStream(boolean redirectErrorStream)

设置是否将标准错误流合并到标准输出流。

#### redirectOutput(Redirect redirect)

设置标准输出流的重定向。

#### redirectError(Redirect redirect)

设置标准错误流的重定向。

### 启动方法

#### start()

启动进程，并返回 Process 对象。

### 其他方法

#### inheritIO()

将当前进程的输入、输出和错误流重定向到当前 Java 进程的相应流。

### 示例代码

# StringUtils类

## 方法

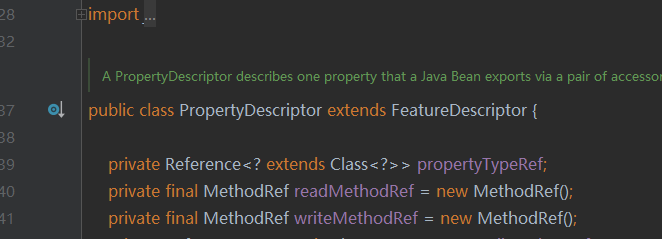
### public static String arrayToCommaDelimitedString(@Nullable Object[] arr)

# PropertyDescriptor类

## 基本概念

PropertyDescriptor 类是 Java 标准库（java.beans 包）中的一个重要类，主要用于描述 Java Bean 中的属性。它提供了对 Java Bean 属性的元数据访问，包括属性的名称、读写方法（getter 和 setter）等。PropertyDescriptor 是 Java 反射机制的一部分，常用于动态访问和操作 Java Bean 的属性。

## 源码



## 作用

### 1、****描述Java Bean属性****

PropertyDescriptor 用于描述 Java Bean 中的一个属性，包括属性的名称、类型、读方法（getter）和写方法（setter）。

### ****2、动态访问属性****

通过 PropertyDescriptor，可以动态获取或设置 Java Bean 的属性值，而不需要直接调用 getter 和 setter 方法。

### ****3、内省（Introspection）****

PropertyDescriptor 是 Java 内省机制的核心类之一。内省是指通过反射机制获取 Java Bean 的属性和方法信息。

### **4、**工具和框架的支持****

许多工具和框架（如 Spring、Hibernate）使用 PropertyDescriptor 来实现动态绑定、数据映射和依赖注入等功能。

## 方法

# Introspector类

# ReflectionUtils类

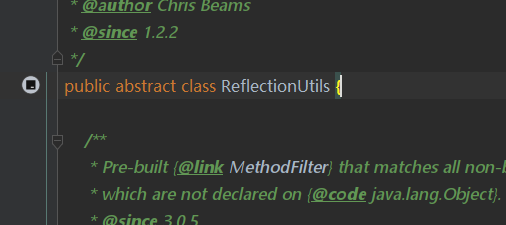
## 基本概念

ReflectionUtils 是 Spring框架内部大量使用的工具类，也是开发自定义框架和工具时的重要帮手。

## 说明

这个类在spring-core-5.3.31.jar包中

## 源码



## 核心作用

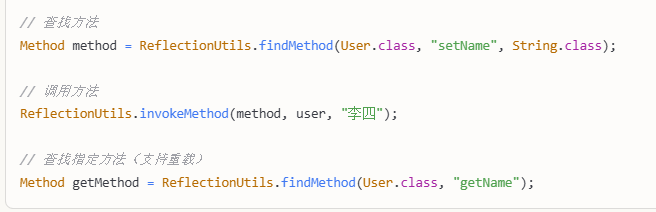
ReflectionUtils 是 Spring 框架提供的反射工具类，封装了 Java 反射 API 的常用操作，简化反射编程并提供更好的异常处理。

## 功能分类

### 字段操作



### 方法操作



## 方法

### findField()

查找字段（支持继承链）

### makeAccessible()

设置可访问性

### getField()

获取字段值

### setField()

设置字段值

### doWithFields()

遍历字段并执行回调

### findMethod()

查找方法（支持参数类型匹配）

### invokeMethod()

调用方法

### doWithMethods()

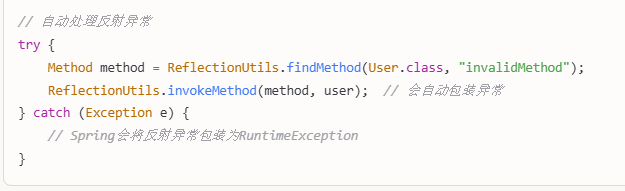
遍历方法并执行回调

### getAllDeclaredMethods()

获取所有声明的方法

## 异常处理

自动处理反射异常



## 核心优势

### 1、简化API

封装了复杂的反射操作，提供简洁的方法

### 2、异常处理

自动处理检查异常，转换为运行时异常

### 3、继承支持

自动查找父类中的字段和方法

### 4、回调机制

提供doWithFields/doWithMethods 批量处理

### 5、空值安全

内置空值检查，避免NullPointerException

## 注意事项

1、反射操作性能相对较低，避免在高频调用中使用

2、破坏了封装性，需要谨慎使用

3、主要用于框架开发，业务代码中应优先考虑其他设计模式

# DocumentBuilderFactory类

# DocumentBuilder