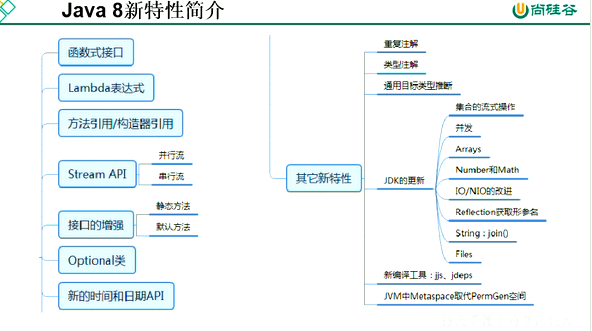
# Java8新特性



# 一、Lambda表达式

## 1.接口的默认方法

### 1.1接口的默认方法

在Java8之前Java中接口里面的方法默认都是public abstract的抽象方法并且没有方法体

### 1.2 static方法

1、使用static修饰的接口中的方法并且必须有主体；

2、接口的static方法只能被接口本身调用：接口名.方法名(…)；

3、接口的static方法不能够被子接口继承；

4、接口的static方法不能够被实现类覆写及直接调用；

### 1.3 default方法

1、使用default修饰的接口中的方法并且必须有主题；

2、接口的default方法只能被实现类；

3、接口的default方法能够被子接口继承；

4、接口的default方法能够被实现类覆写及直接调用；

## 2.函数式接口

### 2.1什么是函数式接口

函数式接口（Functional Interface）就是有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。函数式接口可以被隐式转换为lambda表达式。

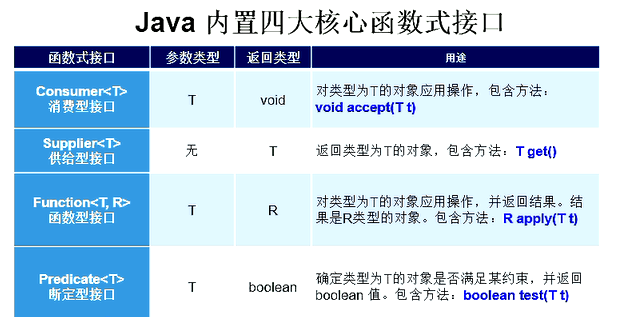
### 2.2函数式接口API

JDK1.8之前已有的函数式接口：



JDK1.8新增加的函数式接口：

java.util.function此包中包含了很多类，用来支持Java的函数式编程





### 2.3函数式接口注解

@FunctionalInterface

我们在函数式接口的上面加上此注解后，里面就只能够有一个抽象方法了，当然不加此注解的且只有一个抽象方法的接口也是函数式接口，只是没有限定提示而已。

## 3.Lambda表达式

### 3.1什么是lambda表达式

简单来说：可以看成是对匿名内部类的简写，使用Lambda表达式时，接口必须是函数式接口

Lambda表达式的本质：作为函数式接口的实例

### 3.2Lambda表达式的语法

**基本语法：**

<函数式接口> <变量名> = (参数1,参数2….) ->{

//方法体

}

**特点说明**：

1、=右边的类型会根据左边的函数式接口类型自动判断；

2、如果形参列表为空，只需保留()；

3、如果形参只有一个，()可以省略，只需要参数的名称即可；

4、如果执行语句只有一句，且无返回值，{}可以省略，若有返回值，则若想省去{}，则必须同时省略return，且执行语句也保证只有1句；

5、形参列表的数据类型会自动推断；

6、lambda不会生成一个单独的内部类文件；

7、lambda只能访问final修饰的局部变量；

例如

**有参数返回：**

**public class** Test {  
 **public static void** main(String[] args) {  
 *//用匿名内部类实现接口* MyInterface mil = **new** MyInterface(){  
 @Override  
 **public int** sum(**int** num1, **int** num2) {  
 **return** num1 + num2;  
 }  
 };  
 *//使用Lambda表达式的写法：常规写法* MyInterface mil2 = (**int** num1, **int** num2)-> {  
 **return** num1 + num2;  
 };  
 */\*\*  
 \* 使用Lambda表达式的写法：简写  
 \* 1.形参列表的数据类型会自动推断  
 \* 2.如果执行语句只有一句，且无返回值，{}可以省略，  
 \* 若有返回值，则若想省去{}，则必须同时省略return，且执行语句也保证只有1句  
 \*/* MyInterface mil3 = (num1, num2)-> num1 + num2;  
 System.***out***.println(mil.sum(1,1));  
 }  
}

**无参数返回：**

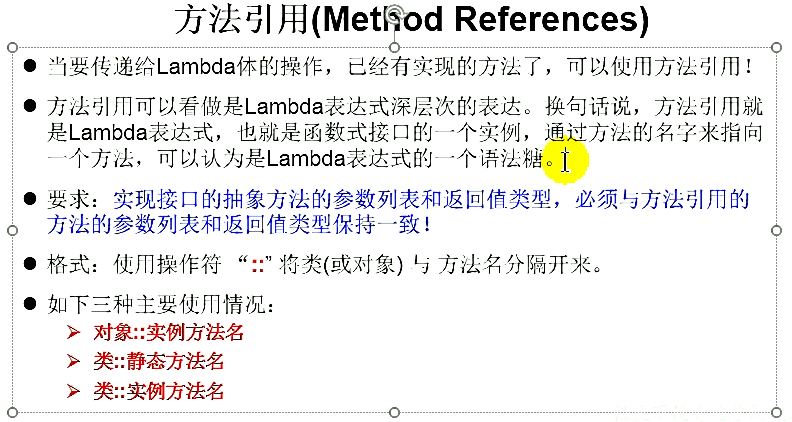
**public class** Test {  
 **public static void** main(String[] args) {  
 *//用匿名内部类实现接口* MyInterface mi1 = **new** MyInterface() {  
 @Override  
 **public void** test() {  
 System.***out***.println(666);  
 }  
 };  
 *//使用Lambda表达式的写法：常规写法* MyInterface mi2 = () -> {  
 System.***out***.println(666);  
 };  
 *//使用Lambda表达式的写法：简写* MyInterface mi3 = () -> {};  
 }  
}

### 3.3Lambda作用域

可以直接访问标记了final的外层局部变量，或者实例字段以及静态变量；

访问局部变量必须是final修饰的，编译时会自动把局部变量编译为final修饰；

## 4.方法引用



### 4.1构造方法引用

**public class** Person {  
 **private** String **firstName**;  
 **private** String **lastName**;  
 **public** Person(String firstName, String lastName) {  
 **this**.**firstName** = firstName;  
 **this**.**lastName** = lastName;  
 }  
 @Override  
 **public** String toString() {  
 **return "Person{"** + **"firstName='"** + **firstName** + **"', lastName='"** + **lastName** + **'\''** +**'}'**;  
 }  
}

**public interface** PersonFactory {  
 Person createPerson(String firstName,String lastName);  
}

**public class** Test {  
 **public static void** main(String[] args) {  
 *//使用匿名内部类* PersonFactory factory = **new** PersonFactory() {  
 @Override  
 **public** Person createPerson(String firstName, String lastName) {  
 **return new** Person(firstName,lastName);  
 }  
 };  
 *//使用Lambda表达式，常规写法* PersonFactory factory2 = (firstName,lastName)->**new** Person(firstName,lastName);  
 *//使用Lambda表达式，简写；  
 // 构造方法引用：要求接口参数列表和构造方法参数列表一致* PersonFactory factory3 = Person::**new**;  
 Person person = factory.createPerson(**"张三"**,**"李四"**);  
 System.***out***.println(person.toString());  
 }  
}

### 4.2静态方法引用

**public interface** ParseIntereface {  
 **int** parse(String str);  
}

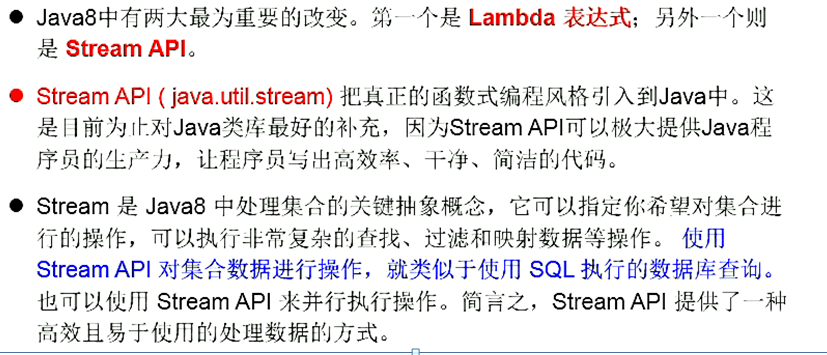
**public class** Test {  
 **public static void** main(String[] args) {  
 *//使用匿名内部类* ParseIntereface p1 = **new** ParseIntereface() {  
 @Override  
 **public int** parse(String str) {  
 **return** Integer.*parseInt*(str);  
 }  
 };  
 *//使用Lambda表达式：常规* ParseIntereface p2 = str -> Integer.*parseInt*(str);  
 *//简写：静态方法引用* ParseIntereface p3 = Integer::*parseInt*;  
 }  
}

### 4.3实例方法引用

**public class** Test {  
 *//使用匿名内部类* String **str** = **"Hello World"**;  
 Function<String,Boolean> **function1** = **new** Function<String, Boolean>() {  
 @Override  
 **public** Boolean apply(String s) {  
 **return str**.endsWith(s);  
 }  
 };  
 *//Lambda常规写法* Function<String,Boolean> **function2** = s -> **str**.endsWith(s);  
 *//Lambda简写，实例方法引用* Function<String,Boolean> **function3** = **str**::endsWith;  
}

# 二、Stream API

## 1.1 Stream API说明





## 1.2 Stream概念

**Stream是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。**

**“集合讲的数据，Stream讲的是计算”**

**注意：**

1. **Stream 不会存储元素。**
2. **Stream不会改变原对象。相反会返回一个持有结果的新Stream**
3. **Stream 操作延迟执行。这意味着他们会等到需要结果的时候才执行**

## 1.3 Stream的操作步骤



### 1.3.1创建Stream

（1）通过集合：Java8中的Collection接口被扩展，提供了两个获取流的方法：

**default** Stream<E> stream()：返回一个顺序流

**default** Stream<E> parallelStream()：返回一个并行流

（2）通过数组：Java8中的Arrays的静态方法stream()：

**static** <T> Stream<T> stream(T[] array)

（3）通过Stream类中的静态方法of()，可以接收任意数量的参数

**static**<T> Stream<T> of(T... values)

（4）创建无限流

迭代：

**static**<T> Stream<T> iterate(**final** T seed, **final** UnaryOperator<T> f)

生成：

**static**<T> Stream<T> generate(Supplier<T> s)

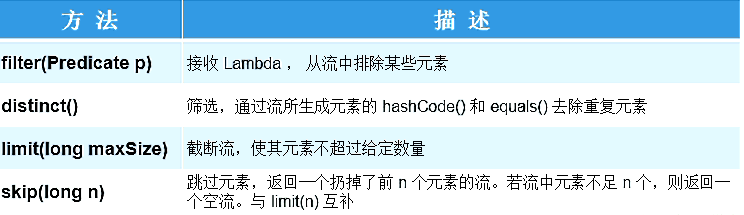
例如：

**public class** Test {  
 **public static void** main(String[] args) {  
 *//获取Stream的方式：  
 //1.通过集合* List<String> list = **new** ArrayList<String>();  
 Stream<String> stringStream1 = list.stream();*//顺序流* Stream<String> stringStream2 = list.parallelStream();*//并行流  
 //2.通过数组* String[] str = **new** String[10];  
 Stream<String> stringStream3 = Arrays.*stream*(str);  
 *//3.通过Stream类中的静态方法of()* Stream<String> stream = Stream.*of*(**"a"**,**"b"**,**"c"**);  
 *//4.创建无限流  
 //遍历前10个偶数* Stream.*iterate*(0,t->t+2).limit(10).forEach(System.***out***::println);  
 *//生成  
 //生成10个随机数*Stream.*generate*(Math::*random*).limit(10).forEach(System.***out***::println);  
 }  
}

### 1.3.2 中间操作

**“惰性求值”**：是指多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行任何操作，而在终止操作时一次性全部处理。

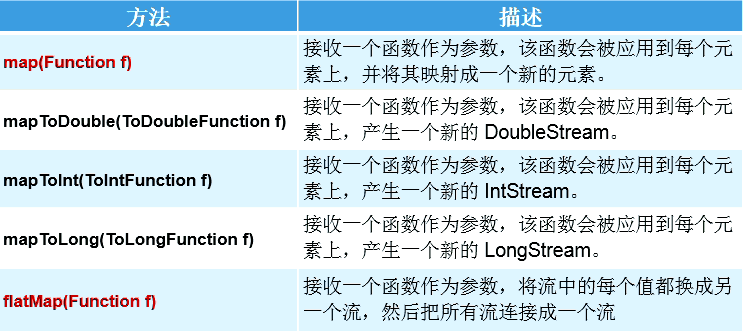
#### 筛选和切片



例如：

*/\*\*  
 \* 2、中间操作--筛选和切片  
 \*/***public void** operationStream1(){  
 List<Integer> list = **new** ArrayList<Integer>();  
 list.add(1);  
 list.add(2);  
 list.add(2);  
 list.add(3);  
 list.add(3);  
 list.add(4);  
 list.add(4);  
 list.add(5);  
 *//1.筛选和切片  
 //filter(Predicate p) 接收Lambda，从流中排除某些元素* Stream<Integer> stream = list.stream();  
 stream.filter(i -> i>3).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*"**);  
 *//limit(n) 截断流，使其元素不超过给定数量* list.stream().limit(4).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*"**);  
 *//skip(n) 返回扔掉前n个元素的流，不足n个，返回空的流* list.stream().skip(4).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*"**);  
 *//distinct() 筛选，通过流所生成元素的hashCode()和equals()去除重复元素* list.stream().distinct().forEach(System.***out***::println);  
}

#### 映射

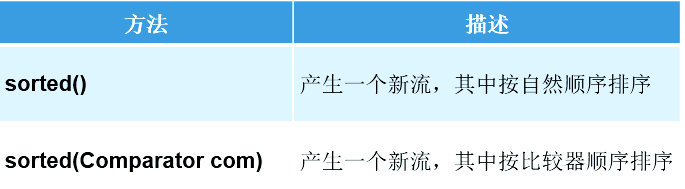


例如：

*//将字符串中的多个字符构成的集合转换为对应是Stream实例***public static** Stream<Character> fromStringToStream(String str){  
 ArrayList<Character> list = **new** ArrayList<>();  
 **for** (Character c:str.toCharArray()){  
 list.add(c);  
 }  
 **return** list.stream();  
}

*/\*\*  
 \*2、中间操作--映射  
 \*/***public void** operationStream2(){  
 List<String> list = Arrays.*asList*(**"aa"**,**"bb"**,**"cc"**);  
 *//map(Function f) 接收一个函数参数，该函数会被应用到每个元素上，并将其映射成一个新的元素* list.stream().map(str -> str.toUpperCase()).forEach(System.***out***::println);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 List<Emp> emps = **new** Emp().getList();  
 Stream<Integer> stream = emps.stream().map(Emp::getAge);  
 stream.forEach(System.***out***::println);  
 *//练习  
 //Stream<Stream<Character>> streamStream = list.stream().map(str -> StreamTest.fromStringToStream(str));* Stream<Stream<Character>> streamStream = list.stream().map(StreamTest::*fromStringToStream*);  
 streamStream.forEach(s->{  
 s.forEach(System.***out***::println);  
 });  
 *//flatMap(Function f) 类似list.addALL()方法；map和flatMap的区别理解为list.add()和list.addALL()方法区别* Stream<Character> characterStream = list.stream().flatMap(StreamTest::*fromStringToStream*);  
 characterStream.forEach(System.***out***::println);  
}

#### 排序



例如：

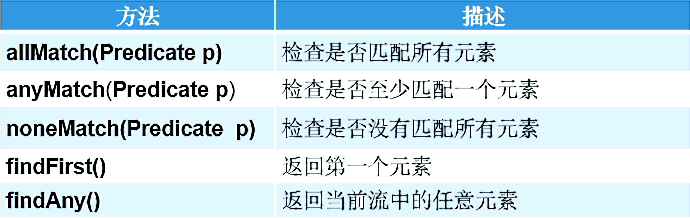
*/\*\*  
 \*2、中间操作--排序  
 \*/***public void** operationStream3(){  
 List<String> list = Arrays.*asList*(**"1"**, **"4"**, **"3"**, **"2"**);  
 *//sorted() 自然排序，升序* list.stream().sorted().forEach(System.***out***::println);  
 *//sorted(Comparator c) 自定义排序* List<Emp> emps = **new** Emp().getList();  
 emps.stream().sorted((e1,e2)->Integer.*compare*(e1.getAge(),e2.getAge()))  
 .forEach(System.***out***::println);  
}

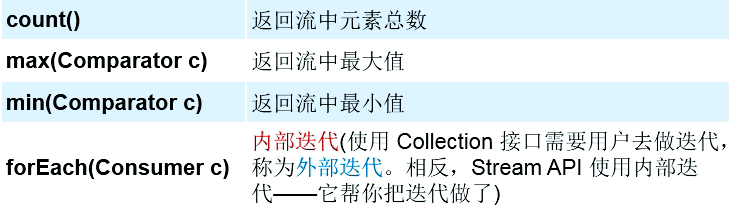
### 1.3.3终止操作

（1）终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List，Integer，甚至是void。

（2）流进行了终止操作后，不能再次使用。

#### 1. 匹配与查找

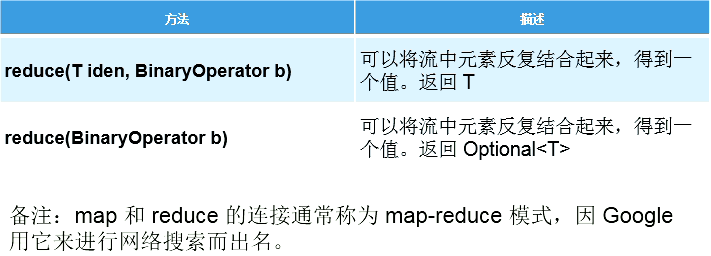




例如：

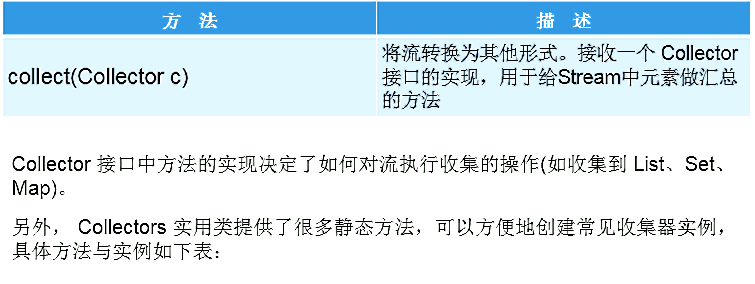
*/\*\*  
 \* 3.终止操作-匹配与查找  
 \*/***public void** ctrlStream1(){  
 List<Emp> emps = **new** Emp().getList();  
 *//boolean allMatch(Predicate p)-检查是否匹配所有元素  
 //年龄18，16,20* **boolean** b = emps.stream().allMatch(e -> e.getAge() > 18);  
 System.***out***.println(b);  
 *//boolean anyMatch(Predicate p)-检查是否至少匹配一个元素* **boolean** b1 = emps.stream().anyMatch(e -> e.getAge() > 20);  
 System.***out***.println(b1);  
 *//boolean noneMatch(Predicate p)-检查是否没有匹配元素* **boolean** b2 = emps.stream().noneMatch(e -> e.getAge() > 20);  
 System.***out***.println(b2);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 *//count()-返回流中元素的总数量* **long** count = emps.stream().filter(e -> e.getAge() > 18).count();  
 *//max()-返回流中的最大值* Stream<Integer> integerStream = emps.stream().map(e -> e.getAge());  
 Optional<Integer> max = integerStream.max(Integer::compareTo);  
 System.***out***.println(max);  
}

#### 2. 规约



*/\*\*  
 \* 3.终止操作-规约  
 \*/***public void** ctrlStream2(){  
 *//T reduce(T identity, BinaryOperator b)-可以将流中的值反复结合起来，得到一个值。返回T  
 //计算1-10自然数的和* List<Integer> list = Arrays.*asList*(1,2,3,4,5,6,7,8,9,10);  
 *//第一个参数是初始值，0开始和是55,10开始是65* Integer sum = list.stream().reduce(0, Integer::*sum*);  
 System.***out***.println(sum);  
 *// Optional<T> reduce(BinaryOperator b);-可以将流中的值反复结合起来，得到一个值。返回Optional* List<Emp> emps = **new** Emp().getList();  
 Optional<Integer> ageSum = emps.stream().map(e -> e.getAge()).reduce(Integer::*sum*);  
 System.***out***.println(ageSum);  
}

#### 收集

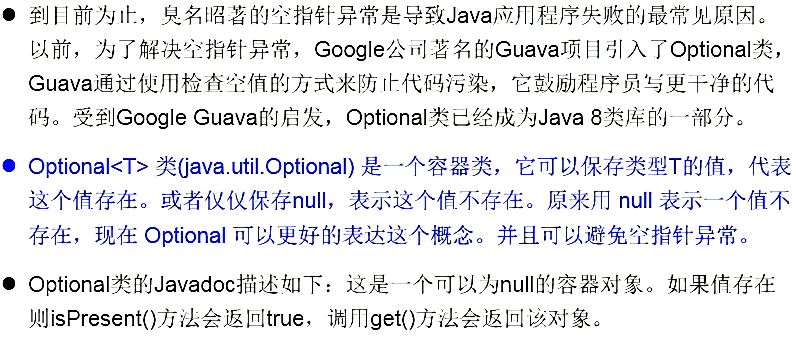


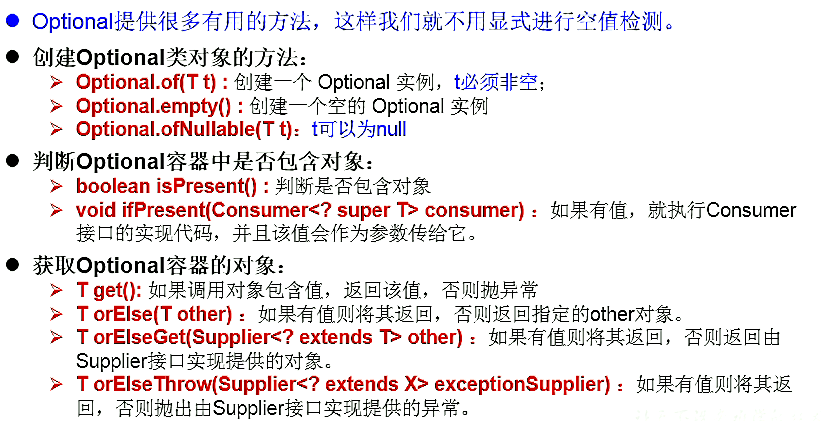
**Collectors类：**

****

*/\*\*  
 \* 3.终止操作-收集  
 \*/***public void** ctrlStream3(){  
 *//R collect(Collector c) 将流转换为其他形式。接收一个Collector接口的实现，用于给Stream中元素做汇总的方法* List<Emp> emps = **new** Emp().getList();  
 List<Emp> empList = emps.stream().filter(e -> e.getAge() > 16).collect(Collectors.*toList*());  
 empList.forEach(System.***out***::println);  
}

# 三、 Optional类





# 四、时间日期类API

JDK1.8之后提供了一套全新的时间日期API 这套全新的API在 java.time 包下；

这三个日期API里面采用静态方法 now()

## 获取当前的日期时间:

年月日：LocalDate

LocalDateTime now = LocalDateTime.now();

System.out.println(now);

时分秒：LocalTime

LocalDate now1 = LocalDate.now();

System.out.println(now1);

年月日时分秒：LocalDateTime

LocalTime now2 = LocalTime.now();

System.out.println(now2);

指定年月日时分秒：

指定年月日，时分秒 使用静态的 of()：

public class MyTest2 {

public static void main(String[] args) {

LocalDateTime of = LocalDateTime.of(2010, 10, 10, 17, 20, 30);

LocalDate of1 = LocalDate.of(2019, 10, 10);

LocalTime of2 = LocalTime.of(18, 20, 30);

}

}

## 与获取相关的方法:

get系类的方法

now.getYear()：获取年

now.getMonth()：获得月份, 返回一个 Month 枚举值

now.getMonthValue()：获得月份(1-12)

now.getDayOfMonth()：获得月份天数(1-31)

now.getHour()：获取小时

now.getMinute()：获取分钟

now.getSecond()：获取秒

now.getDayOfWeek()：获取星期几

public static void main(String[] args) {

//注：ISO - 8601 日历系统是国际标准化组织制定的现代公民的日期和时间的表示法

LocalDateTime now = LocalDateTime.now();

//与获取相关的方法

int year = now.getYear();

Month month = now.getMonth();

int monthValue = now.getMonthValue();

System.out.println(year);

System.out.println(month.getValue());

System.out.println(monthValue);

System.out.println(now.getDayOfMonth());

System.out.println(now.getHour());

System.out.println(now.getMinute());

System.out.println(now.getSecond());

DayOfWeek dayOfWeek = now.getDayOfWeek();//获取星期几

System.out.println(dayOfWeek.getValue());

}

## DateTimeFormatter类:解析和格式化日期或时间的类:

public static void main(String[] args) {

LocalDateTime now = LocalDateTime.now();

// System.out.println(now);

//DateTimeFormatter格式化日期的类

DateTimeFormatter f = DateTimeFormatter.ofPattern("yyyy年MM月dd日 HH时mm分ss秒");

//格式化日期的方法

String time = now.format(f);

System.out.println(time);

}

通过静态方法ofPattern(“yyyy-MM-dd”)来获取对象；

.format()方法：

把一个日期对象的默认格式 格式化成指定的格式

把一个日期字符串转成日期对象

使用日期类中的parse方法传入一个日期字符串, 传入对应的日期格式化类

DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("yyyy-MM-dd");

String time = "2018-12-12";

LocalDate parse = LocalDate.parse(time, dateFormat);

System.out.println(parse);

程序：

public class MyTest2 {

public static void main(String[] args) {

// DateTimeFormatter:解析和格式化日期或时间的类

DateTimeFormatter f = DateTimeFormatter.ofPattern("yyyy年MM月dd日");

//方式1：把日期格式化成字符串

LocalDate now = LocalDate.now();

String format = now.format(f);

System.out.println(format);

//方式2

String format1 = f.format(now);

System.out.println(format1);

//5. 把一个日期字符串转成日期对象

// 使用日期类中的parse方法传入一个日期字符串, 传入对应的日期格式化类

DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("yyyy-MM-dd");

String time = "2018-12-12";

LocalDate parse = LocalDate.parse(time, dateFormat);

System.out.println(parse);

}

}

## 日期类型互相转换：

public static void main(String[] args) {

//与转换相关的方法

LocalDateTime now = LocalDateTime.now();

//转换成 年月日的这个日期

LocalDate localDate = now.toLocalDate();

//转换成 时分秒的这个日期

LocalTime localTime = now.toLocalTime();

}

静态方法paser() ：

将一个日期字符串解析成日期对象, 注意字符串日期的写法的格式要正确, 否则解析失败

例如://LocalDateTime parse = LocalDateTime.parse(“2007-12-03T10:15:30”);

public static void main(String[] args) {

//5. 解析的静态方法parse("2007-12-03T10:15:30");

//paser() 将一个日期字符串解析成日期对象, 注意字符串日期的写法的格式要正确, 否则解析失败

//例如:

//LocalDateTime parse = LocalDateTime.parse("2007-12-03T10:15:30");

String str="2007-12-03";

DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

LocalDate parse = LocalDate.parse(str, dateTimeFormatter);

System.out.println(parse);

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

String str2 = "2007-12-03 15:20:30";

LocalDateTime parse1 = LocalDateTime.parse(str2, DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

System.out.println(parse1);

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

String str3="15:20:30";

LocalTime parse2 = LocalTime.parse(str3, DateTimeFormatter.ofPattern("HH:mm:ss"));

System.out.println(parse2);

}

## 判断的方法：

isAfter()：判断一个日期是否在指定日期之后

isBefore()：判断一个日期是否在指定日期之前

isLeapYear()：判断是否是闰年注意是LocalDate类中的方法

public static void main(String[] args) {

LocalDate now = LocalDate.now();

LocalDate of = LocalDate.of(2010, 2, 19);

boolean after = of.isAfter(now);//判断一个日期是否在指定日期之后

boolean after1 = now.isAfter(of);//判断一个日期是否在指定日期之后

System.out.println(after);

System.out.println(after1);

boolean before = of.isBefore(now);//判断一个日期是否在指定日期之前

System.out.println(before);

boolean before1 = now.isBefore(of);//判断一个日期是否在指定日期之前

System.out.println(before1);

boolean leapYear = of.isLeapYear();//判断是否是闰年注意是LocalDate类中的方法

System.out.println(leapYear);

boolean leapYear1 = now.isLeapYear();//判断是否是闰年注意是LocalDate类中的方法

System.out.println(leapYear1);

}

## 添加年月日时分秒的方法：

plus系列的方法都会返回一个新的LocalDateTime的对象

LocalDateTime localDateTime = now.plusYears(1);

System.out.println(localDateTime);

LocalDateTime localDateTime1 = now.plusDays(2);

System.out.println(localDateTime1);

public static void main(String[] args) {

//年月日，时分秒 加上对应的时间量

LocalDateTime now = LocalDateTime.now();

System.out.println(now);

//注意添加完相应的时间量后，返回的是一个新对象

LocalDateTime localDateTime = now.plusYears(1);

System.out.println(localDateTime);

LocalDateTime localDateTime1 = now.plusDays(2);

System.out.println(localDateTime1);

}

## 减去年月日时分秒的方法：

minus 系列的方法注意都会返回一个新的LocalDateTime的对象

LocalDateTime localDateTime = now.minusYears(2);

System.out.println(localDateTime);

LocalDateTime localDateTime1 = now.minusMonths(3);

System.out.println(localDateTime1);

public static void main(String[] args) {

//给年月日时分秒减去相应的时间量

LocalDateTime now = LocalDateTime.now();

//注意：减去相应的时间量后，返回的是一个新的日期对象

LocalDateTime localDateTime = now.minusYears(2);

System.out.println(localDateTime);

////注意：减去相应的时间量后，返回的是一个新的日期对象

LocalDateTime localDateTime1 = now.minusMonths(3);

System.out.println(localDateTime1);

}

## 指定年月日时分秒的方法：

with系列的方法注意都会返回一个新的LocalDateTime的对象

LocalDate localDate = now.withYear(2020);

System.out.println(localDate);

LocalDate localDate1 = now.withMonth(12);

System.out.println(localDate1);

public static void main(String[] args) {

// 8. 指定年月日时分秒的方法 with系列的方法 注意都会返回一个新的LocalDateTime的对象

LocalDate of = LocalDate.of(2011, 10, 10);

LocalDate now = LocalDate.now();

System.out.println(now);

//指定完对应的日期后，返回的是一个新的日期对象

LocalDate localDate = now.withYear(2020);

//指定完对应的日期后，返回的是一个新的日期对象

LocalDate localDate1 = now.withMonth(12);

System.out.println(localDate);

System.out.println(localDate1);

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

LocalDate localDate2 = now.withDayOfMonth(20);

System.out.println(localDate2);

TemporalAdjuster temporalAdjuster = TemporalAdjusters.firstDayOfNextMonth();

temporalAdjuster = TemporalAdjusters.lastDayOfMonth();

temporalAdjuster = TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY);

temporalAdjuster = TemporalAdjusters.previous(DayOfWeek.MONDAY);

LocalDate with = now.with(temporalAdjuster);

LocalDate with1 = now.with(ChronoField.DAY\_OF\_YEAR, 100);

System.out.println(with1);

System.out.println(with);

}

## Instant 时间戳类：

Instant 时间戳类从1970 -01 - 01 00:00:00 截止到当前时间的毫秒值

获取对象的方法：

Instant now = Instant.now();

System.out.println(now.toString());

注意默认获取出来的是当前的美国时间和我们相差八个小时

我们在东八区 所以可以加8个小时 就是我们的北京时间

Instant中设置偏移量的方法:

atOffset() 设置偏移量

返回的是一个偏移后的时间

OffsetDateTime offsetDateTime = now.atOffset(ZoneOffset.ofHours(8));

System.out.println(offsetDateTime);

## 设置时区方法：

atZone()方法的参数是要一个时区的编号可以通过时区编号类获取出来

ZonedDateTime zonedDateTime = now.atZone(ZoneId.systemDefault());

System.out.println(zonedDateTime);

public static void main(String[] args) {

// Instant 时间戳类从1970 -01 - 01 00:00:00 截止到当前时间的毫秒值

Instant now = Instant.now();

System.out.println(now.toString());

long l = now.toEpochMilli(); //获取当前时间的毫秒

long l1 = System.currentTimeMillis();

System.out.println(l);

System.out.println(l1);

//设置偏移量,返回的是一个偏移后的时间

OffsetDateTime offsetDateTime = now.atOffset(ZoneOffset.ofHours(8));

System.out.println(offsetDateTime);

//设置时区

ZonedDateTime zonedDateTime = now.atZone(ZoneId.systemDefault());

System.out.println(zonedDateTime);

}

get系列的方法：

getEpochSecond()：获取从1970-01-01 00:00:00到当前时间的秒值

getNano()：是把获取到的当前时间的秒数 换算成纳秒

public class MyTest3 {

public static void main(String[] args) {

//4. get系列的方法

Instant now = Instant.now();

//获取当前的毫秒值

long l = now.toEpochMilli();

//获取当前的秒值

long epochSecond = now.getEpochSecond();

System.out.println(l);

System.out.println(now);

// 1 毫秒(ms) = 1000000 纳秒(ns)

// 1 毫秒(ms) = 1000000 纳秒(ns)

System.out.println(epochSecond \* 1000);

System.out.println();

int nano = now.getNano();

System.out.println(nano);

}

}

ofEpochSecond() 方法：给计算机元年增加秒数

Instant instant = Instant.ofEpochMilli(0);

public class MyTest4 {

public static void main(String[] args) {

// 5. ofEpochSecond() 方法 给计算机元年增加秒数

Date date = new Date(0);

System.out.println(date);

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

Instant instant = Instant.ofEpochMilli(0);//给计算机元年增加毫秒数

Instant instant1 = Instant.ofEpochSecond(60 \* 60);

System.out.println(instant);

System.out.println(instant1);

}

}

## Duration类与Period:

### Duration:用于计算两个“时间”间隔的类

public class MyTest5 {

public static void main(String[] args) {

Instant start = Instant.now();

for (int i = 0; i < 100000; i++) {

System.out.println("aa");

}

Instant end = Instant.now();

Duration between = Duration.between(start, end);

//long seconds = between.getSeconds();

//System.out.println(seconds);

long l = between.toMillis(); //两个时间戳间隔的毫秒

System.out.println(l);

}

}

### Period:用于计算两个“日期”间隔的类

public class MyTest6 {

public static void main(String[] args) {

LocalDate of = LocalDate.of(1995, 5, 5);

LocalDate now = LocalDate.now();

//##Period:

//用于计算两个“日期”间隔的类

Period between = Period.between(of, now);

int years = between.getYears();

int months = between.getMonths();

int days = between.getDays();

System.out.println(years);

System.out.println(months);

System.out.println(days);

}

}

## ZoneID 世界时区类：

1.获取世界各个地方的时区的集合 的方法getAvailableZoneIds()

使用ZoneID中的静态方法getAvailableZoneIds();来获取

例如:Set availableZoneIds = ZoneId.getAvailableZoneIds();

2.获取系统默认时区的ID

ZoneId zoneId = ZoneId.systemDefault(); //Asia/Shanghai

3.获取带有时区的日期时间对象

//创建日期对象

LocalDateTime now = LocalDateTime.now();

//获取不同国家的日期时间根据各个地区的时区ID名创建对象

ZoneId timeID = ZoneId.of("Asia/Shanghai");

//根据时区ID获取带有时区的日期时间对象

ZonedDateTime time = now.atZone(timeID);

System.out.println(time);

//方式2 通过时区ID 获取日期对象

LocalDateTime now2 = LocalDateTime.now(ZoneId.of("Asia/Shanghai"));

System.out.println(now2);