# Lambda表达式

Lambda表达式为Java添加了确实的函数式变成特性，使我们能将函数当做一等公民看待。

在将函数作为一等公民的语言中，Lambda表达式的类型是函数。但在Java中，Lambda表达式是对象，他们必须依附于一类特别的对象类型----函数式接口。

Java Lambda表达式是一种匿名函数，它是没有声明的方法，即没有访问修饰符，返回值声明和名字。

作用：1.传递行为，而不仅仅是值

2.提升抽象层次

3.API重用性更好

4.更加灵活

基本语法：(argument)->{body}

如：(arg1,arg2…)->{body}

(type1 arg1 ,type2 arg2)->{body}

(int a ,int b)->{return a+b ;}

一个Lambda表达式可以有零个或多个参数

参数的类型可以明确声明，也可以根据上下文来推断，如(int a)与(a)效果相同

所有参数需包含在圆括号内，参数之间使用逗号相隔

空圆括号代表参数为空

当只有一个参数，且其类型可推导时，圆括号()可省略。如:b->return b\*b ;

Lambda表达式主体可以包含零条或多条语句

如果Lambda表达式的主题只有一条语句，花括号{}可省略。匿名函数的返回类型与该主题表达式一致。

如果Labbda表达式的主题包含多条语句，则这些语句必须包含该{}中，匿名函数的返回与普通方法返回值类型一样，若没有返回值则为空。

# 函数式接口

@FunctionalInterface注解官方文档说明：

该注解时一个信息，用于声明一个接口是函数式接口，从概念上看，一个函数式接口只有一个精确的abstract方法，有非抽象默认方法的实现。如果一个接口声明一个抽象方法用于重写Object中的public方法，这并不能使接口的抽象方法增加，这是因为任何实现该接口的类将会直接或间接继承了Object类。

注意：一个函数式接口的实例可以通过lambda 表达式，方法引用或者构造方法引用来创建。

如果一个被该注解注释的类型，编译器将会生成一个错误信息，除了：

* 该类型是一个接口类型，并且不是注解类型，枚举类型或者class
* 一个注解类型满足函数式接口的要求

然而，编译器对满足函数式接口要求的接口，看作一个函数式接口，不管是否该接口类型被@FunctionalInterface标注。

总结：1.如果一个接口只有一个抽象方法，该接口就是一个函数式接口

2.如果我们在某个接口上声明了FunctionalInterface注解，那么编译器就会按照函数式接口的定义来要求接口。

3.如果某个接口只有一个抽象方法，但我们并没有给该接口声明FunctionalInterface注解，但编译器依旧把该接口看作函数式接口。

Java.util.function包中的接口分为四类：分别是Consumer(消费型接口)，Supplier(供给型接口)，Predicate(谓词型接口)以及Function(功能型接口)。

每种基本接口还包含若干相关的接口。如Consumer接口，用于处理基本数据类型的IntConsumer，LongConsumer和DoubleConsumer接口。

/\*\*

\*函数式接口

\*/

@FunctionalInterface

**public** **interface** MyInterface

{

**void** log();

String toString();

}

**class** Main

{

**public** **void** test(MyInterface myinterface)

{

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

myinterface.log();

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

}

**public** **static** **void** main(String[] args)

{

Main main = **new** Main();

//jdk7之前的写法

main.test(**new** MyInterface() {

@Override

**public** **void** log() {

System.***out***.println("jdk7之前的写法");

}

});

//jdk8的写法

main.test(()->{System.***out***.println("jdk8的写法");});

}

}

从上面中定义的接口可以看出，对于toString方法的定义，编译器并没有报错，这是因为MyInterface直接继承了Object类，从而继承了toString方法，这里我么只是对toString方法进行了重写，默认的实现还是Object的实现。

## Function接口

官方API说明：代表一个function，接受一个参数，从而产出一个结果。

Function<T,R>这里T：function接受的类型。R：function返回的类型。

**public** **class** Demo3

{

**public** **void** compute(**int** a ,Function<Integer ,Integer> function)

{

System.***out***.println(function.apply(a));

}

**public** **static** **void** main(String[] args)

{

Demo3 demo = **new** Demo3();

demo.compute(2, param->{**return** param+param ;});

demo.compute(8, param->{**return** param\*param ;});

}

}

从这个demo中，我们可以看出在每次调用compute方法时都会传递一个Function函数，这里我们传递的是行为

**public** **class** Demo4

{

**public** **int** testCompose(**int** a ,Function<Integer ,Integer> function1 , Function<Integer, Integer> function2 )

{

**return** function1.compose(function2).apply(a);

}

**public** **int** testAndEnd(**int** a ,Function<Integer ,Integer> function1 , Function<Integer, Integer> function2)

{

**return** function1.andThen(function2).apply(a);

}

**public** **static** **void** main(String[] args)

{

Demo4 demo = **new** Demo4();

System.***out***.println(demo.testCompose(2, param->param+2, param->param\*param)); //6

System.***out***.println(demo.testAndEnd(2, param->param+2, param->param\*param));//16

}

}

对于Function<T,V>的compose(Function<T,R> before)：

**default** <V> Function<V, R> compose(Function<? **super** V, ? **extends** T> before) {

Objects.*requireNonNull*(before);

**return** (V v) -> apply(before.apply(v));

}

从源码我们可以看出，先调用before的apply方法，然后将方法执行的接口传递给自身的apply方法。所以在testCompose方法中，先执行param->param\*param，然会将执行的结果（这里是4）传递给funtion1的param->param+2，从而得到结果6.

对于andThen方法：源码

**default** <V> Function<T, V> andThen(Function<? **super** R, ? **extends** V> after) {

Objects.*requireNonNull*(after);

**return** (T t) -> after.apply(apply(t));

}

该方法先调用自身的apply方法，然后执行after的apply。这与compose方法正好相反。

对于compose和andThen方法都返回Function<T,V>类型，所以对多个Function<T,V>进行串联起来，就好像一条链一样。

## BiFunction接口

Function<T,V>接口只能接受一个输入，从而得到一个输出结果，如果我们希望有多个输入时，则Function接口就无法满足我们的需求。

BiFunction<T,U,R>接受两个输入，T第一个输入，U第二个输入，R返回结果。

**public** **class** Demo5

{

**public** **int** compute(**int** param1 ,**int** param2 ,BiFunction<Integer,Integer,Integer> bifunction)

{

**return** bifunction.apply(param1, param2);

}

**public** **int** testAndThen(**int** param1 ,**int** param2 ,BiFunction<Integer, Integer, Integer> bifunction , Function<Integer,Integer> function)

{

**return** function.apply(bifunction.apply(param1, param2));

}

**public** **static** **void** main(String[] args)

{

Demo5 demo = **new** Demo5();

**int** num1 = 10 ;

**int** num2 = 20 ;

System.***out***.println(num1+"+"+num2+"="+demo.compute(num1,num2,(param1,param2)->param1+param2));

System.***out***.println(num1+"-"+num2+"="+demo.compute(num1,num2,(param3,param4)->param3-param4));

System.***out***.println(num1+"\*"+num2+"="+demo.compute(num1,num2,(param5,param6)->param5\*param6));

System.***out***.println(num1+"/"+num2+"="+demo.compute(num1,num2,(param7,param8)->param7/param8));

System.***out***.println(demo.testAndThen(num1, num2, (n1,n2)->n1+n2,n3->n3\*n3));

}

}

对于BiFunction<T,U,R>并没有compose方法，并且andThen方法接受的参数类型为Function<T,R>类型。

这是因为BiFunction<T,U,R>返回值的个数为一个，而BiFunction中的apply方法接受两个参数。

### 案例

**public** **class** Demo6

{

**private** List<Person> persons = **new** ArrayList<>();

**public** Demo6()

{

persons.add(**new** Person("li",20));

persons.add(**new** Person("sai",30));

persons.add(**new** Person("hao",44));

}

/\*\*

\*根据usernane查询相应的person对象

\*/

**public** List<Person> getPersonByUsername(String username ,List<Person> allPerson)

{

**return** persons.stream().filter(person->person.getUsername().equals(username))

.collect(Collectors.*toList*());

}

/\*\*

\*查询所有年龄大于age的person对象

\*/

**public** List<Person> getPersonByage(**int** age , List<Person> allPersons)

{

BiFunction<Integer ,List<Person>,List<Person>> biFunction = (ageOfPerson,personList)->

{

**return** personList.stream().filter(person->person.getAge()>=ageOfPerson)

.collect(Collectors.*toList*());

};

**return** biFunction.apply(25, allPersons);

}

/\*\*

\* 动态条件查询person对象，则方法参数传递的是行为，而具体的动作是由用户来决定的

\*/

**public** List<Person> getPersonByCondition(Object condition ,List<Person> allPersons,

BiFunction<Object,List<Person>,List<Person>> biFunction)

{

**return** biFunction.apply(condition, allPersons);

}

**public** **static** **void** main(String[] args)

{

Demo6 demo = **new** Demo6();

List<Person> list = demo.getPersonByUsername("hao", demo.persons);

System.***out***.println(list.get(0).toString());

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

BiFunction<Object,List<Person>,List<Person>> biFunction = (condition,allPersons)->{

**if**(condition **instanceof** Integer)

{

**int** age = (Integer)condition ;

**return** allPersons.stream().filter(person->person.getAge()>=age)

.collect(Collectors.*toList*());

}

**if**(condition **instanceof** String)

{

String username = (String)condition ;

**return** allPersons.stream().filter(person->person.getUsername().equals(username))

.collect(Collectors.*toList*());

}

**return** **null** ;

};

List<Person> persons = demo.getPersonByCondition(20, demo.persons, biFunction) ;

persons.forEach(person->System.***out***.println(person.toString()));

}

}

## Predicate接口

表示一个判断，boolean值的function，只有一个参数。

**public** **static** **void** main(String[] args)

{

Predicate<String> predicate = str->str.length()>5 ;

System.***out***.println(predicate.test("lisaihao"));

}

根据提供的test方法，根据提供的参数来返回true还是false。

### 案例

**import** java.util.\*;

**import** java.util.function.\*;

**public** **class** Demo8

{

**private** List<Integer> nums = **new** ArrayList<>();

**public** Demo8()

{

**for**(**int** i = 1 ;i<=10 ;i++)

{

nums.add(i);

}

}

**public** **void** testPredateTest(List<Integer> list ,Predicate<Integer> predicate)

{

**for**(Integer i : list)

{

**if**(predicate.test(i))

System.***out***.println(i);

}

}

**public** **void** testAnd(List<Integer> list ,Predicate<Integer> condition1 ,Predicate<Integer> condition2)

{

**for**(Integer i : list)

{

**if**(condition1.and(condition2).test(i))

{

System.***out***.println(i);

}

}

}

**public** **void** testOr(List<Integer> list ,Predicate<Integer> condition1 ,Predicate<Integer> condition2)

{

**for**(Integer i : list)

{

**if**(condition1.or(condition2).test(i))

{

System.***out***.println(i);

}

}

}

**public** **static** **void** main(String[] args)

{

Demo8 demo = **new** Demo8();

//打印偶数

demo.testPredateTest(demo.nums, num->num%2==0);

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

//打印大于5的数

demo.testPredateTest(demo.nums, num->num>=5);

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

//打印所有

demo.testPredateTest(demo.nums, num->**true**);

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

//什么都不打印

demo.testPredateTest(demo.nums, num->**false**);

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

//打印大于5，并为偶数的数字

demo.testAnd(demo.nums, num->num>=5,num->num%2==0);

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

//打印大于5，或为偶数的数字

demo.testOr(demo.nums, num->num>=5,num->num%2==0);

}

}

## Supplier接口

表示一个结果的供应者，并不要求在每次调用时返回一个新的或者与以前有区别的结果

**package** code.lsh.list;

**public** **class** Person

{

**private** String username ;

**private** **int** age ;

**public** Person() {}

**public** Person(String username , **int** age)

{

**this**.username = username ;

**this**.age = age ;

}

@Override

**public** String toString() {

**return** "Person [username=" + username + ", age=" + age + "]";

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

使用该接口可以代替工厂方法；

**public** **class** Demo1

{

**public** **static** **void** main(String[] args)

{

Supplier<Person> supplier = ()->**new** Person();

Person person = supplier.get();

System.***out***.println(person);

}

}

## BinaryOperator接口

API说明：

代表有着两个相同类型运算对象的操作，并且返回相同类型的运算结果，该接口是BiFunction接口的特殊形式。

操作数和返回结果具有相同的类型。

接口的定义**public** **interface** BinaryOperator<T> **extends** BiFunction<T,T,T>，泛型的类型头为T。

**public** **class** Demo1

{

**public** **int** compute(**int** num1 ,**int** num2 ,BinaryOperator<Integer> bo)

{

**return** bo.apply(num1, num2) ;

}

**public** BinaryOperator<String> testMinBy(String str1 ,String str2 ,Comparator<String> comparator)

{

**return** BinaryOperator.*minBy*(comparator);

}

**public** **static** **void** main(String[] args)

{

Demo1 demo = **new** Demo1();

BinaryOperator< Integer> bo = (num1,num2)->num1+num2 ;

System.***out***.println(demo.compute(10,20,bo));

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

demo.testMinBy("hao", "sai",(str1,str2)->str1.length()-str2.length());

}

}

## Optional类

Java.util.Optional<T>类，该类的作用从代码中删除NullPointer’Exception异常。Optional类设计用来在返回值可以合法为null时与用户通信。如果根据某些条件过滤值流后恰好没有元素存留，就会出现返回值为null的情况。

API说明：

Optional是一个容器对象，可能包含一个非空的值，也可能不包含

该类是基于值的，对一些敏感操作，如：引用相等（==），哈希码的同一，或者同步，则Optional实现将会产生不可预料的错误。我们应该避免使用。

Value-based Classes：1.final和不可变的（即使包含一个可变对象的引用）对象。

2.实现了equals，hashCode和toString方法，这些实现仅仅是通过实例状态计算出来的，而不是其他对象或变量计算出来的。

3.不是使用敏感的操作，如在实例之间引用相等（==），哈希码，或者实例固有锁的同步。

4.仅仅依靠于equals()方法来考虑相等，而不是依靠==

5.没有可访问的构造方法，仅仅通过工厂方法来创建实例，对于返回实例的标识不做任何承诺（返回的对象可能不相同）

6.在相等时，可以自由交换，这就意味着在任何计算或者方法调用中x.equals(y)返回true，或者生产一个不可见的改变行为，x实例和y实例可以交换。

Optional实例：1.被声明为final且是不可变的（但可能包含对可变对象的引用）。

2.不提供公共构造方法，因此必须采用工厂方法加以实例化。

3.具有equals，hashCode与toString的实现，这些实现是通过实例的状态计算出来的。

工厂方法：

创建一个空的Optional

**public** **static**<T> Optional<T> empty() {

@SuppressWarnings("unchecked")

Optional<T> t = (Optional<T>) ***EMPTY***;

**return** t;

}

创建一个非空的Optional对象

**public** **static** <T> Optional<T> of(T value) {

**return** **new** Optional<>(value);

}

//创建一个为空的Optional对象，或者非空的Optional对象

**public** **static** <T> Optional<T> ofNullable(T value) {

**return** value == **null** ? *empty*() : *of*(value);

}

注意：对于Optional<T>是无法作为方法参数或者成员变量的，但可以作为方法的返回值。

Optional实例是不可变的，但它包装的对象却不一定是不可变的。如果创建一个包含可变对象实例的Optional，则仍然可以对实例进行修改。

AtomicInteger counter = new AtomicInteger();  
Optional<AtomicInteger> opt = Optional.*ofNullable*(counter);  
System.*out*.println(opt);  
  
counter.incrementAndGet();  
System.*out*.println(opt);  
  
opt.get().incrementAndGet();  
System.*out*.println(opt);  
  
opt = Optional.*ofNullable*(new AtomicInteger());  
System.*out*.println(opt);

不可以将不可变性与final混为一谈。

### 案例

**public** **static** **void** main(String[] args)

{

Optional<String> optional = Optional.*of*("hello");

/\*//该写法在java8中并不推荐

if(optional.isPresent())

{

System.out.println(optional.get());

}\*/

optional.ifPresent(str->System.***out***.println(str));

//如果Optional容器中没有值，则打印world，否则打印容器中的值

System.***out***.println(optional.orElse("world"));

//如果如果Optional容器中没有值，则Supplier中的get方法

System.***out***.println(optional.orElseGet(()->"hello world"));

}

对于集合，如果集合不为空，则将集合遍历打印：这里采用员工和公司多对一的案例：

**public** **class** Employee

{

**private** String name ;

**public** Employee(String name)

{

**this**.name = name ;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**public** **class** Company

{

**private** String cName ;

**private** List<Employee> employees ;

**public** String getcName() {

**return** cName;

}

**public** **void** setcName(String cName) {

**this**.cName = cName;

}

**public** List<Employee> getEmployees() {

**return** employees;

}

**public** **void** setEmployees(List<Employee> employees) {

**this**.employees = employees;

}

}

**public** **static** **void** main(String[] args)

{

Employee employee1 = **new** Employee("zhangsan");

Employee employee2 = **new** Employee("lisai");

List<Employee> employees = Arrays.*asList*(employee1,employee2);

Company company = **new** Company();

company.setcName("company1");

// company.setEmployees(employees);

Optional<Company> optional = Optional.*ofNullable*(company);

System.***out***.println(optional.map(theCompany->theCompany.getEmployees()).orElse(Collections.*emptyList*()));

}

如果公司一个的员工集合不为空，则打印集合，否则就创建一个空的集合，打印[]

如果T的为String，orElse总会创建一个字符串，则orElseGet使用Supplier，仅在Optinal为空时才创建。对于字符串orElse和orElseGet的区别可以忽略不计。

如果T是一个复杂对象-，那么传入的Supplier的orElseGet方法能确保仅在需要时才创建对象。

Optional<ComplexObject> val = values.steam.findFirst();  
val.orElse(new ComplexObject());  
val.orElseGet(ComplexObjext::new)

这里Optinal就是采用了Supplier的延迟执行或惰性执行的一种应用。

#### flatMap和map方法

Optinal中的flatMap方法的签名如下：

public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {  
 Objects.*requireNonNull*(mapper);  
 if (!isPresent())  
 return *empty*();  
 else {  
 return Objects.*requireNonNull*(mapper.apply(value));  
 }  
}

public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {  
 Objects.*requireNonNull*(mapper);  
 if (!isPresent())  
 return *empty*();  
 else {  
 return Optional.*ofNullable*(mapper.apply(value));  
 }  
}

该方法的签名与map方法类似，二者的Function参数应用于每个元素并生成一个结果，返回类型为Optional<U>。具体而言，如果参数T存在，Optional.flatMap方法将函数应用于T并返回Optional，它包装包含的值；如果T不存在，方法将返回一个空的Optional。

class Manager{  
 private String name ;  
 public Manager(String name){  
 this.name = name ;  
 }  
 public String getName(){  
 return name ;  
 }  
}  
class Department{  
 private Manager boss;  
 public Optional<Manager> getBoss(){  
 return Optional.*ofNullable*(boss);  
 }  
 public void setBoss(Manager boss){  
 this.boss = boss ;  
 }  
}  
  
class Company{  
 private Department department;  
 public Optional<Department> getDepartment(){  
 return Optional.*ofNullable*(department);  
 }  
 public void setDepartment(Department department){  
 this.department = department;  
 }  
}

Manager manager = new Manager("lisaihao");  
Department d = new Department();  
d.setBoss(manager);  
System.*out*.println(d.getBoss());  
System.*out*.println("Name:"+d.getBoss().orElse(new Manager("zhuyuanhao")).getName());

使用Optional中的map方法：

System.*out*.println("Name:"+d.getBoss().map(Manager::getName));

**执行结果：**

Optional[code.lsh.java8.Manager@76ed5528]

Name:lisaihao

Name:Optional[lisaihao]

Name:Optional[lisaihao]

Company co = new Company();  
co.setDepartment(d);  
System.*out*.println("Company Dept:"+co.getDepartment());  
System.*out*.println("Company Dept Manager:"+co.getDepartment().  
 map(Department::getBoss));

**运行结果：**

Company Dept:Optional[code.lsh.java8.Department@3941a79c]

Company Dept Manager:Optional[Optional[code.lsh.java8.Manager@76ed5528]]

从运行结果可以看出：Optional中嵌套Optional。我们可以使用flatMap方法可以将结构“展平”

Optional<String> result = co.getDepartment().flatMap(Department::getBoss).map(Manager::getName);  
System.*out*.println(result);  
System.*out*.println("-------------------------------------------------");  
Optional<Company> company = Optional.*of*(co);  
System.*out*.println(  
 company.flatMap(Company::getDepartment)  
 .flatMap(Department::getBoss)  
 .map(Manager::getName)  
);

**运行结果：**

Optional[lisaihao]

-------------------------------------------------

Optional[lisaihao]

class Employee{  
 private int id ;  
 public Employee(int id ){  
 this.id = id;  
 }  
 public int getId(){  
 return id;  
 }  
 public void setId(int id){  
 this.id = id;  
 }  
}

public class Demo3  
{  
 private static List<Employee> *allEmployee* = new ArrayList<>();  
 private static Random *random* = new SecureRandom();  
 static{  
 for(int i = 0 ;i<10;i++){  
 *allEmployee*.add(new Employee(*random*.nextInt(10)));  
 }  
 }  
  
 */\*\*  
 \*根据id查询Employee  
 \*/* public Optional<Employee> findEmployeeById(int id){  
 if(*allEmployee*.contains(id)){  
 return Optional.*of*(*allEmployee*.get(id));  
 }  
 return Optional.*empty*();  
 }  
  
 */\*\*  
 \*通过ids获取Employee集合  
 \*/* public List<Employee> findEmployeesByIds(List<Integer> ids){  
 return ids.stream()  
 .map(this::findEmployeeById) //Stream<Optional<Employee>>  
 .filter(Optional::isPresent) //删除空的Optional  
 .map(Optional::get) //Stream<Employee>  
 .collect(Collectors.*toList*()); //List<Employee>  
 }  
}

使用Optional.map方法：

public List<Employee> findEmployeesByIds(List<Integer> ids){  
 return ids.stream()  
 .map(this::findEmployeeById) //Stream<Optional<Employee>>  
 .flatMap(optional-> {  
 return optional.map(Stream::*of*) //Optional<Stream<Employee>>  
 .orElseGet(Stream::*empty*); //Stream<Employee>  
 }) //Stream<Employee>  
 .collect(Collectors.*toList*());  
}

如果findEmployeeById方法返回一个空的Optional，optional.map(String::of)方法同样将返回一个空的Optional，而orElseGet(Stream::empty)方法将返回一个空的流，因此得到Stream<Employee>元素与空流的组合，Stream.flatMap方法的真正用途就在于此。仅对非空流而言，Stream.flatMap方法将所有内容简化为Stream<Employee>，因此collect方法可以将非空流作为员工列表返回。

# 方法引用

通俗的讲，方法引用时Lambda表达式的语法糖。

**方法引用可以分为4中：**

1.类名::方法名

案例：如根据学生的分数进行升序排序。

**public** **class** Student

{

**private** String name ;

**private** **int** score ;

**public** Student(String name ,**int** score)

{

**this**.name = name ;

**this**.score = score ;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getScore() {

**return** score;

}

**public** **void** setScore(**int** score) {

**this**.score = score;

}

/\*\*

\* 根据分数进行排序

\*/

**public** **static** **int** compareStudentByScore(Student student1 , Student student2)

{

**return** student1.getScore() - student2.getScore() ;

}

}

**public** **static** **void** main(String[] args)

{

Student s1 = **new** Student("zhangshan",66);

Student s2 = **new** Student("wangwu",45);

Student s3 = **new** Student("lisi",87);

Student s4 = **new** Student("mazi",98);

List<Student> students = Arrays.*asList*(s1,s2,s3,s4) ;

/\*//方法一

//调用list接口中的sort方法进行排序，该方法在集合本身上进行比较，并不会产生新的集合

students.sort((student1 ,student2)->Student.compareStudentByScore(student1, student2));

students.forEach(stu->System.out.println(stu.getScore()));

System.out.println("-------------------------------------");\*/

//方法二：调用方法引用

students.sort(Student::*compareStudentByScore*);

students.forEach(stu->System.***out***.println(stu.getScore()));

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

}

2.引用名::实例方法名

**public** **class** StudentComparator

{

/\*\*

\* 根据分数排序

\*/

**public** **int** compareStudentByScore(Student student1 , Student student2)

{

**return** student1.getScore() - student2.getScore() ;

}

/\*\*

\* 根据姓名排序

\*/

**public** **int** comparatorStudentByName(Student s1 ,Student s2)

{

**return** s1.getName().charAt(0)-s2.getName().charAt(0);

}

}

**public** **static** **void** main(String[] args)

{

Student s1 = **new** Student("zhangshan",66);

Student s2 = **new** Student("wangwu",45);

Student s3 = **new** Student("lisi",87);

Student s4 = **new** Student("mazi",98);

List<Student> students = Arrays.*asList*(s1,s2,s3,s4) ;

StudentComparator sc = **new** StudentComparator();

//方法一

students.sort((student1,student2)->sc.compareStudentByScore(student1, student2));

students.forEach(stu->System.***out***.println(stu.getScore()));

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

//方法二，方法引用

students.sort(sc::compareStudentByScore);

students.forEach(stu->System.***out***.println(stu.getScore()));

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

}

3.类名::实例方法名

修改Student类中comparatorStudentByScore方法为：

**public** **int** compareStudentByScore(Student student)

{

**return** **this**.getScore() - student.getScore();

}

**public** **class** Demo1

{

**public** **static** **void** main(String[] args)

{

Student s1 = **new** Student("zhangshan",66);

Student s2 = **new** Student("wangwu",45);

Student s3 = **new** Student("lisi",87);

Student s4 = **new** Student("mazi",98);

List<Student> students = Arrays.*asList*(s1,s2,s3,s4) ;

//方法引用 类名::实例方法名

students.sort(Student::compareStudentByScore);

students.forEach(stu->System.***out***.println(stu.getScore()));

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

}

}

我们定义实例方法comparatorStudentByScore直接收一个参数，比较的方式为当前实例（Student实例）与传入的Student进行比较。在students.sort(Student::comparatorS

tudentByScore)中，实际上是由传入comparator方法中的第一个参数调用comparatorStudent

ByScore方法，而传入的第二个参数，传入到comparatorStudentByScore方法中了。

**总结：**lambda表达式的第一个参数被作为这个实例方法的调用这，而其他参数被作为方法的参数。

**4.构造方法引用, 类名::new**

**public** **class** Demo2

{

**public** **void** test(Supplier<String> supplier)

{

System.***out***.println(supplier.get());

}

**public** String test2(String str ,Function<String,String> function)

{

**return** function.apply(str);

}

**public** **static** **void** main(String[] args)

{

Demo2 demo = **new** Demo2();

demo.test(String::**new**);

System.***out***.println(demo.test2("hao",String::**new**));

}

}

# 接口

## 接口的默认方法

在jdk8中打破了传统我们对java接口的认知，我们在接口中定义方法，但方法必须使用default关键字修饰。

这里我们都一个特殊情况进行讨论：

**public** **interface** MyInterface1 {

**default** **void** myMethod()

{

System.***out***.println("MyInterface1");

}

}

**public** **interface** MyInterface2 {

**default** **void** myMethod()

{

System.***out***.println("MyInterface1");

}

}

这里我们定义连个接口MyInterface1和MyInterface2，并且定义了相同的方法，如果同时实现这两个接口，会发生什么情况？编译器会让我们重写这两个接口都有的方法。

**public** **class** Main **implements** MyInterface1,MyInterface2

{

@Override

**public** **void** myMethod()

{

}

}

如果我们希望使用MyInterface1的方法，我们可以：

**public** **class** Main **implements** MyInterface1,MyInterface2

{

@Override

**public** **void** myMethod()

{

MyInterface1.**super**.myMethod();

}

}

如果有一个Main2类实现了MyInterface1类，并且Main类继承了Main2类和实现了MyInterfact2接口。

**public** **class** Main2 **implements** MyInterface1{}

Java之所以不支持多继承是为了避免所谓的钻石问题。

## 接口的静态方法

接口方法声明为static

在Java8中，我们可以随时为接口添加静态方法，步骤如下：

1. 为方法添加static关键字。
2. 提供一种无法被重写的实现，此时，静态方法类似于默认方法，包含了Javadoc的Defautl Method(默认标签)中。
3. 通过接口名访问方法。类不需要通过实现接口来使用静态方法。

我们可以分析java.util.Comparator接口中定义的comparing方法就是一种实现的静态方法，如：

public static <T, U> Comparator<T> comparing(  
 Function<? super T, ? extends U> keyExtractor,  
 Comparator<? super U> keyComparator)  
{  
 Objects.*requireNonNull*(keyExtractor);  
 Objects.*requireNonNull*(keyComparator);  
 return (Comparator<T> & Serializable)  
 (c1, c2) -> keyComparator.compare(keyExtractor.apply(c1),  
 keyExtractor.apply(c2));  
}

public static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor) {  
 Objects.*requireNonNull*(keyExtractor);  
 return (Comparator<T> & Serializable)  
 (c1, c2) -> Integer.*compare*(keyExtractor.applyAsInt(c1), keyExtractor.applyAsInt(c2));  
}

注意：1.静态方法必须有一个实现。2.无法重写静态方法。3.通过接口名调用静态方法。4.无法实现接口以使用静态方法。

# 流（Stream）

流是一种元素的序列，它不存储元素，也不会修改原始源。Java的函数式编程通常涉及从某些数据源生成流，通过一系列成为**流水线**的中间操作传递元素，并利用终止表达式完成这一过程。

## Stream接口

API说明：

支持并行或串行的聚和操作的元素序列。

IntStream,LongStream和DoubleStrem都是原生类型特化的类型的Stream，并且它们遵循Stream的特征和限制。它们和Stream是平行的。

如果执行一个流的运算，则流运算会被组合到一个流管道中。一个流管道包含一个源（可以是数组，集合，I/O通道），包含零个或多个中间操作（中间操作会将流转化成另一个流，如filter），和终止操作（会产生一个结果，或其他影响，例如count()会forEach(Consumer)，这是因为Consumer会接受一个流中元素的引用，因此我们在accept()方法中通过引用进行修改对象），流是惰性的，在源数据的操作只有在终止操作被触发时时才会执行，仅当需要时源数据元素才被消费。

流和集合虽然有相似性，但是它们的目标是不相同的。集合主要关注于元素的高效管理和访问，与此相反，流并没有提供一个方式用于访问和操作流中的元素。它主要以声明的方式来描述源和源中的运算操作，源的操作可以在源上以聚合的方式来进行（集合主要关注与元素的管理和访问，而流主要关注与元素的运算）。如果流提供的操作并不能满足我们的需要，我们可以是还用iterator()和spliteratior()方法来控制流的遍历。

一个流应该仅仅被调用一次（调用一个中间操作，或终止操作），

流操作分类：1.惰性求值 2.及早求值。

流不存储值，通过管道的方式获取值

本质是函数式的，对流的操作会生成一个结果，不过并不会修改底层的数据源，集合可以作为流的底层数据源。

很多流操作（过滤，映射，排序）都可以延迟实现。

Stream声明：**public** **interface** Stream<T> **extends** BaseStream<T, Stream<T>>

Stream类的接受T类型，继承BaseStream，接受T和Stream<T>类型。

**public** **interface** BaseStream<T, S **extends** BaseStream<T, S>>

入门案例：

**public** **static** **void** main(String[] args)

{

List<Integer> list = Arrays.*asList*(1,2,3,4,5);

System.***out***.println(list.stream().map(i->i\*2).reduce(0, Integer::*sum*));

}

Stream<String> stream = Stream.*of*("hello","world","hello world");

String[] array = stream.toArray(length->**new** String[length]);

Arrays.*asList*(array).forEach(System.***out***::println);

流的串行和惰性操作：

**public** **class** Demo2

{

**public** **static** **void** main(String[] args)

{

List<String> list = Arrays.*asList*("li","sai","hao","sai hao");

Stream<String> stream = list.stream();

stream.mapToInt(item->{

**int** length = item.length();

System.***out***.println(item);

**return** length ;

}).filter(length->length==3).findFirst().ifPresent(System.***out***::println);;

}

}

这里findFirst()方法是一个终止操作。

### 流的创建

* 利用Stream.of方法创建流

public static<T> Stream<T> of(T t) {  
 return StreamSupport.*stream*(new Streams.StreamBuilderImpl<>(t), false);  
}

传入单个元素T t，返回只包含一个元素的单例顺序流。

* 利用Stream.iterate方法

public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f) {  
 略  
}

iterate方法返回一个无限顺序的有序流（infinite sequential ordered stream），它有迭代应用到初始元素种子的函数f产生。

Stream.*iterate*(BigDecimal.*ONE*,n->n.add(BigDecimal.*ONE*))  
 .limit(10).forEach(System.*out*::println);

由于生成的流是无界的，所以通过中间操作limit加以限制。

* 利用Stream.generate方法

public static<T> Stream<T> generate(Supplier<T> s) {  
 Objects.*requireNonNull*(s);  
 return StreamSupport.*stream*(  
 new StreamSpliterators.InfiniteSupplyingSpliterator.OfRef<>(Long.*MAX\_VALUE*, s), false);  
}

通过多次调用Supplier产生一个顺序的无序流（sequential,unordered stream）

Stream.*generate*(Math::*random*).limit(10).forEach(System.*out*::println);

### API方法

* flatMap

<R> [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)<R> flatMap([Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html),? extends [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)<? extends R>> mapper)

**public** **class** Demo2

{

**public** **static** **void** main(String[] args)

{

List<String> list = Arrays.*asList*("li sai","sai hao","hao li","sai hao");

list.stream().map(item->item.split(" ")).flatMap(Arrays::*stream*).distinct().forEach(System.***out***::println);

}

}

List<String> list = Arrays.*asList*("li","sai","hao");

List<String> list2 = Arrays.*asList*("li","sai","hao");

list.stream().flatMap(item->list.stream().map(item2->item+" "+item2)).collect(Collectors.*toList*()).forEach(System.***out***::println);;

* IntStream.reduce 归约操作

int sum = IntStream.*rangeClosed*(1,10).reduce((x,y)->x+y).orElse(0);  
System.*out*.println("sum="+sum);

二元运算符返回的值在下一次作为迭代时作为累加器，而每次传入的流中的值作为被加数。

我们还可以处理流中的每一个数字，然后在求和：

int sum = IntStream.*rangeClosed*(1,10).reduce((x,y)->x+y\*2).orElse(0);  
System.*out*.println("sum="+sum);

重载的方法：第一个参数表示在进行操作是的第一个参数（具有累加器初始值的reduce方法）。

T reduce(T identity, BinaryOperator<T> accumulator);

int sum = IntStream.*rangeClosed*(1,10).reduce(0,(x,y)->x+y\*2);  
System.*out*.println("sum="+sum);

二院操作符的标识值：在示例中将第一个参数称为累加器的初始值（initial value），不过方法签名将其称为标识值（identity value）。关键字identity表示应该为二元运算符提供一个值，以便与其他值结合时返回另一个值。加法操作的标识值为0，乘法操作的标识值为1，字符串拼接操作的标识值为空字符串。

在Java的标准库中引入一些新的方法使归约操作变得特别简单：如Integer，Long和Double提供的sum方法。

int sum = Stream.*of*(1,2,3,4,5,6,7,8,9,10).reduce(0,Integer::*sum*);  
System.*out*.println("sum="+sum);

求流中的最大值：

int max = Stream.*of*(1,2,3,4,5,6,7,8,9,10).reduce(Integer.*MIN\_VALUE*,Integer::*max*);  
System.*out*.println("max="+max);

* peek方法对流进行调试

int sum = IntStream.*rangeClosed*(1,10)  
 .peek(n->System.*out*.printf("original:%d%n",n))  
 .map(n->n\*2)  
 .peek(n->System.*out*.printf("double:%d%n",n))  
 .filter(n->n%3==0)  
 .peek(n->System.*out*.printf("filter:%d%n",n))  
 .sum();  
 System.*out*.println("sum="+sum);

* collect

## BaseStream

BaseStream是所有流的父接口，是元素的序列，支持串行和并行的聚合操作。BaseStream继承了AutoCloseable接口，调用close()方法时，在管道中所有的关闭处理器都会被调用。

方法介绍：

**boolean** isParallel()：如果一个终止操作被执行返回一个流是否以并发的方式执行。如果在终止操作之后调用该方法可能会发生不可预料的事情。

S sequential()：放回一个等价的串行流，或者返回它本身，它本身已经是串行的，或者底层流

被修改为串行流。返回类型为S，即类型为Stream，因为没一个流操作都会产生一个新的流对象，而S就是代表新生成的流对象。

S onClose(Runnable closeHandler)：返回一个有关闭处理器的等价流，当流中的close方法被调用时，关闭处理器才被调用。它们被执行的顺序与它们添加的顺序一样（在调用该方法时，会返回一个新的流，新的流对象可以继续调用该方法）。所有的处理器都可以被执行，即使前一个处理器抛出异常。如果任意一个处理器抛出异常，第一个异常将传递给调用端，其他异常都被作为被压制的异常（除非剩余的异常与第一个异常是相同的，因此异常不会被压制）。

**public** **static** **void** main(String[] args)

{

List <String> list = Arrays.*asList*("li","sai","hao");

**try**(Stream<String> stream = list.stream())

{

stream.onClose(()->{

System.***out***.println("aaa");

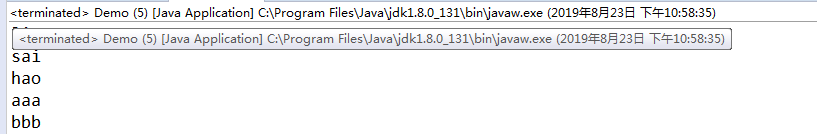
}).onClose(()->{

System.***out***.println("bbb");

}).forEach(System.***out***::println);

}

}



List <String> list = Arrays.*asList*("li","sai","hao");

**try**(Stream<String> stream = list.stream())

{

stream.onClose(()->{

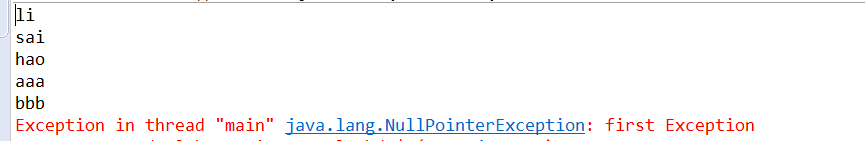
System.***out***.println("aaa");

**throw** **new** NullPointerException("first Exception");

}).onClose(()->{

System.***out***.println("bbb");

}).forEach(System.***out***::println);



发生异常并不影响关闭处理器的执行。

List <String> list = Arrays.*asList*("li","sai","hao");

**try**(Stream<String> stream = list.stream())

{

stream.onClose(()->{

System.***out***.println("aaa");

**throw** **new** NullPointerException("first Exception");

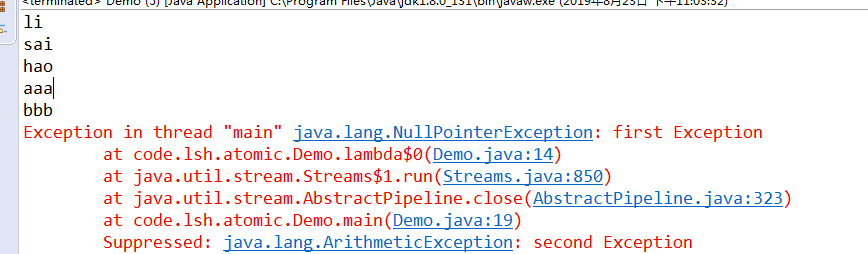
}).onClose(()->{

System.***out***.println("bbb");

**throw** **new** ArithmeticException("second Exception");

}).forEach(System.***out***::println);

}



第二个关闭处理器抛出的异常被压制（Suppressed）。

List <String> list = Arrays.*asList*("li","sai","hao");

NullPointerException npe = **new** NullPointerException("My Exception");

**try**(Stream<String> stream = list.stream())

{

stream.onClose(()->{

System.***out***.println("aaa");

**throw** npe;

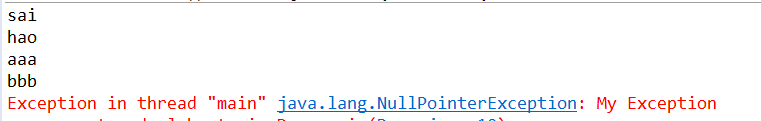
}).onClose(()->{

System.***out***.println("bbb");

**throw** npe;

}).forEach(System.***out***::println);

}



对于同一异常并不会被压制。

## 分割迭代器（Spliterator）

ApI说明：

一个对象用于对源中的元素进行遍历和分区。通过Spliterator所覆盖的元素的源可以是数组，集合，IO通道，或者生成器函数。

一个分割迭代器可以一个一个的遍历元素（tryAdvance()），也可以以块的方式遍历(forEachRemaining())。

一个分割迭代器可以分区它自身的元素（使用trySplit()方法）作为另一个分割迭代器，尽可能使用并行的方式来操作。使用Spliterator操作不能进行分割，或者不高度平衡或高效的行为，这是不可能从并行中受益。遍历和分割都消耗元素，每一个分割器只对单个块的运算有用的。

一个分割迭代器从ORDERED（有顺序的），DISTINCT（不同的），SORTED（排序的），SIZED（确定大小的），NONNULL（不为空的），IMMUTABLE（不可变的），CONCURRENT（并发的）和SUBSIZED（分成子的）报告它自身结构，源和元素的特征集合characteristics()。这些特性可以被Spliterator客户端所使用，用于控制，具体化和简化操作。例如，Collection的分隔符将会报告一个SIZED，Set集合分隔符将会报告一个DISTINCT，SortedSet分隔符将会报告一个SORTED，特性质都是使用简单的位操作来标识的。一些特性质会额外限制方法行为，例如如果ORDERED，遍历方法必须遵循它们在文档中定义的顺序，在未来新的特征值或许被定义，因此实现者不应该将没有列出的值赋予意义。

一个没有报告IMMUTABLE或CONCURRENT的分割迭代器，我们期待有文档策略的考量：当分割迭代器绑定到元素的源上时；并且在绑定之后要对元素源结构上的检测，延迟绑定分割器到元素源上，在第一次遍历，第一次分割，或第一次调用，而不是在分割器创建时。对非延迟绑定器在创建时或者调用分割器任意方法时绑定到元素源上。在绑定之前修改了源，则在Spliterator遍历时会反映出来，

## Collector（收集器）接口

一个可变的汇聚操作，它的作用是将输入元素累计到一个可变的结果容器中，可选的，在所有的输入元素已经被处理之后，将累计的结果转化为最终的表示。汇聚操作既支持串行也支持并行执行。

Collectors提供了Collector常用的汇聚操作，Colllectors实际上是一个工厂。

一个Collectors通过4个函数所指定的，这4个函数会协同工作，累计entries到一个可变的结果容器中，并且可选地将结果执行一个最终的转换：1.创建一个新的结果容器（supplier()）

2.将新的数据元素合并到结果容器中（accumulator()）

3. 将两个结果容器合并一个（combiner()）

4．对容器执行一个可选的最终的转换（finisher()）

Collectors有一组characteristics集合，比如Collector.Characteristics.CONCURRENT，提供一个hints可以被汇聚所使用，用于提高执行的性能。

使用collector的串行汇聚操作的实现，将使用supplier函数创建一个单一的结果容器，并且每一个输入元素都会调用accumulator函数一次。一种并行实行，将输入进行分区，没一个分区都会创建一个结果容器。累计每一个分区的内容到该分区的结果容器中，然后使用combiner函数将这些子结果合并成最终结果。

为了确保串行和并行执行产生等价的结果，collector函数必须满足同一性和结合性的约束。

同一性表明，对于任何部分累计结果，与一个空的结果容器结合必须生产出一个等价的结果。就是说，对于部分累计结果a，is the result of any series of accumulator and combiner invocations，a必须等价于combiner.apply(a,supplier.get())。

结合性约束，分割计算必须生产等价的结果。因此，对于任何输入元素t1和t2，运算的结果r1和r2，在下面的运算必须相等：

串行： A a1=supplier.get();

accumulator.accept(a1,t1);

accumulator.accept(a1,t2); //这里a1表示每一次累计的中间结果，t1,t2下一次

R r1=finisher.apply(a1); //累计元素

并行： A a2 = supplier.get();

accumulator.accept(a2, t1);

A a3 = supplier.get();

accumulator.accept(a3, t2);

R r2 = finisher.apply(combiner.apply(a2, a3));

基于Collector实现的汇聚操作库，比如Stream.collect(Collector)，必须遵守如下的约定：

* 传递给accumulator函数，combiner函数和finisher函数的第一个参数必须是先前调用supplier，accumulator或combiner函数的结果。
* 实现不应该对supplier，accumulator，或combiner函数做任何操作，除了再一次传递给accumulator，combiner，或者finisher函数，或者返回给调用者。
* 如果传递给combiner或finisher函数的结果，并且相同的对象没有从该函数中返回，则再也不会被使用。
* 如果一个结果传递给combiner或finisher函数，则它再也不会被传递给accumulator函数了。
* 对于一个非并发的收集器，supplier，acuumulator或combiner函数返回的任何结果必须serially thread-confined（当前线程），它能够使收集在并行的情况下发生，而Collecor不需要实现任何其他同步方法。

汇聚实现必须管理输入，确保输入已经被分区了，分区的处理是相互的隔离，在accumulaton完成后combining才会发生。

* 对于并发的收集器，可以自由实现并发汇聚。在accumulator函数被多个线程并发调用时，而并发汇聚只有一个。使用相同的并发修饰结果容器，而不是在accumulation中保持结果的隔离。如果收集器的characteristics是Collector.Characteristics.UNORDERED或者最初的数据是无序的，则并发汇聚应该被调用。

### 案例

**public** **class** Student

{

**private** String name ;

**private** **int** score ;

**private** **int** age ;

**public** Student(String name, **int** score, **int** age) {

**this**.name = name;

**this**.score = score;

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getScore() {

**return** score;

}

**public** **void** setScore(**int** score) {

**this**.score = score;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", score=" + score + ", age=" + age + "]";

}

}

**public** **static** **void** main(String[] args)

{

Student s1 = **new** Student("zhangsan", 100, 20);

Student s2 = **new** Student("lisi", 90, 20);

Student s3 = **new** Student("wangwu", 90, 21);

Student s4 = **new** Student("zhangsan", 80, 25);

List<Student> list = Arrays.*asList*(s1,s2,s3,s4);

//根据学生姓名进行分组

Map<String, List<Student>> collect = list.stream().collect(Collectors.

*groupingBy*(Student::getName));

//根据分数进行分组

Map<Integer, List<Student>> collect2 = list.stream().collect(Collectors.

*groupingBy*(Student::getScore));

//根据学生姓名分组并返回同一个组中的个数

Map<String, Long> collect3 = list.stream().collect(Collectors.

*groupingBy*(Student::getName,Collectors.*counting*()));

//获取同一组中的分数平均值

Map<String, Double> collect4 = list.stream().collect(Collectors.

*groupingBy*(Student::getName,Collectors.*averagingDouble*(Student::getScore)));

//根据分数大于等于90进行分区

Map<Boolean, List<Student>> collect5 = list.stream().collect(Collectors.

*partitioningBy*(stu->stu.getScore()>=90));

//获取分数最大的学生

list.stream().collect(Collectors.*maxBy*(Comparator.*comparingInt*(Student::getScore))).

ifPresent(System.***out***::println);

//获取分数最小的学生

list.stream().collect(Collectors.*minBy*(Comparator.*comparingInt*(Student::getScore))).

ifPresent(System.***out***::println);

//拼接学生的姓名

String collect6 = list.stream().map(Student::getName).collect(Collectors.*joining*());

//拼接学生的姓名，并以空格分割

String collect7 = list.stream().map(Student::getName).collect(Collectors.*joining*(" "));

//拼接学生的姓名，并以空格分割，添加前缀后后缀

String collect8 = list.stream().map(Student::getName).collect(Collectors.*joining*(" ","<strong>","</strong>"));

//先根据学生的分数分组，在根据学生名分组

Map<Integer, Map<String, List<Student>>> collect9 = list.stream().collect(Collectors.

*groupingBy*(Student::getScore,Collectors.*groupingBy*(Student::getName)));

//分数大于85进行分区

Map<Boolean, List<Student>> collect10 = list.stream().collect(Collectors.

*partitioningBy*(stu->stu.getScore()>85));

//分数大于85进行分区，然后在大于90的分区

Map<Boolean, Map<Boolean, List<Student>>> collect11 = list.stream().collect(Collectors.*partitioningBy*(stu->stu.getScore()>85,

Collectors.*partitioningBy*(stu->stu.getAge()>90)));

//分数大于80的个数

Map<Boolean, Long> collect12 = list.stream().collect(Collectors.*partitioningBy*(stu->stu.getScore()>85,

Collectors.*counting*()));

}

### 自定义收集器

**import** java.util.\*;

**import** java.util.function.BiConsumer;

**import** java.util.function.BinaryOperator;

**import** java.util.function.Function;

**import** java.util.function.Supplier;

**import** java.util.stream.\*;

**public** **class** MySetCollector<T> **implements** Collector<T,Set<T>,Set<T>>

{

@Override

**public** Supplier<Set<T>> supplier()

{

System.***out***.println("supplier invoked");

**return** HashSet<T>::**new**;

}

@Override

**public** BiConsumer<Set<T>, T> accumulator()

{

System.***out***.println("accumulator invoked");

**return** Set::add;

}

@Override

**public** BinaryOperator<Set<T>> combiner()

{

System.***out***.println("combiner invoked");

**return** (set1,set2)->{set1.addAll(set2);**return** set1;};

}

@Override

**public** Function<Set<T>, Set<T>> finisher()

{

System.***out***.println("finisher invoked");

**return** Function.*identity*();

}

@Override

**public** Set<Characteristics> characteristics()

{

System.***out***.println("characteristics invoked");

**return** Collections.*unmodifiableSet*(EnumSet.*of*(Characteristics.***IDENTITY\_FINISH***));

}

**public** **static** **void** main(String[] args)

{

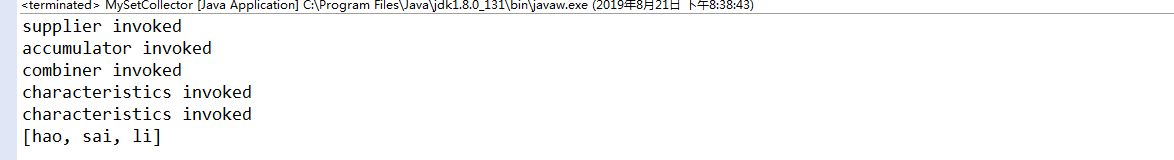
List<String> list = Arrays.*asList*("li","sai","hao");

Set<String> set = list.stream().collect(**new** MySetCollector<String>());

System.***out***.println(set);

}

}



这里finisher()方法并没有执行，并且characteristics执行了两次。

分析源码：ReferencePipeline类实现了Stream接口

@Override

@SuppressWarnings("unchecked")

**public** **final** <R, A> R collect(Collector<? **super** P\_OUT, A, R> collector) {

A container;

**if** (isParallel()

&& (collector.characteristics().contains(Collector.Characteristics.***CONCURRENT***))

&& (!isOrdered() || collector.characteristics().contains(Collector.Characteristics.***UNORDERED***))) {

container = collector.supplier().get();

BiConsumer<A, ? **super** P\_OUT> accumulator = collector.accumulator();

forEach(u -> accumulator.accept(container, u));

}

**else** {

container = evaluate(ReduceOps.*makeRef*(collector));

}

**return** collector.characteristics().contains(Collector.Characteristics.***IDENTITY\_FINISH***)

? (R) container

: collector.finisher().apply(container);

}

该方法的return语句中会调用collector的characteristics()方法。如果characteristics返回的集合中包含了IDENTITY\_FINISH，则会强制的将collect方法中间结果类型容器转换为最终的结果类型，并不会调用finisher方法，否则调用finisher()方法。

ReduceOps类的makeRef方法：

**public** **static** <T, I> TerminalOp<T, I>

makeRef(Collector<? **super** T, I, ?> collector) {

Supplier<I> supplier = Objects.*requireNonNull*(collector).supplier();

BiConsumer<I, ? **super** T> accumulator = collector.accumulator();

BinaryOperator<I> combiner = collector.combiner();

**class** ReducingSink **extends** Box<I>

**implements** AccumulatingSink<T, I, ReducingSink> {

@Override

**public** **void** begin(**long** size) {

state = supplier.get();

}

@Override

**public** **void** accept(T t) {

accumulator.accept(state, t);

}

@Override

**public** **void** combine(ReducingSink other) {

state = combiner.apply(state, other.state);

}

}

**return** **new** ReduceOp<T, I, ReducingSink>(StreamShape.***REFERENCE***) {

@Override

**public** ReducingSink makeSink() {

**return** **new** ReducingSink();

}

@Override

**public** **int** getOpFlags() {

**return** collector.characteristics().contains(Collector.Characteristics.***UNORDERED***)

? StreamOpFlag.***NOT\_ORDERED***

: 0;

}

};

}

该方法首先调用了collector的supplier方法，在调用accumulator方法，然后在调用combiner方法。

在方法getOpFlags()方法中会调用collector的characteristics()方法一次。

#### 深入理解Characteristics

##### Characteristics.IDENTITY\_FINISH

Characteristics.IDENTITY\_FINISH：收集器中间结果容器类型与返回的结果类型相同。

**import** java.util.\*;

**import** java.util.function.\*;

**import** java.util.stream.\*;

**public** **class** MyCollector<T> **implements** Collector<T,Set<T>,Map<T,T>>

{

@Override

**public** Supplier<Set<T>> supplier()

{

System.***out***.println("supplier invoked");

**return** HashSet<T>::**new**;

}

@Override

**public** BiConsumer<Set<T>, T> accumulator()

{

System.***out***.println("accumulator invoked");

**return** Set::add;

}

@Override

**public** BinaryOperator<Set<T>> combiner() {

System.***out***.println("combiner invoked");

**return** (set1,set2)->{set1.addAll(set2);**return** set1;};

}

@Override

**public** Function<Set<T>, Map<T, T>> finisher()

{

System.***out***.println("finisher invoked");

**return** set->{

Map<T,T> map = **new** HashMap<>();

set.stream().forEach(item->map.put(item, item));

**return** map;

};

}

@Override

**public** Set<Characteristics> characteristics() {

System.***out***.println("characteristics invoked");

**return** Collections.*unmodifiableSet*(EnumSet.*of*(Characteristics.***UNORDERED***));

}

**public** **static** **void** main(String[] args) {

List<String> list = Arrays.*asList*("li","sai","hao","a","aa","b","cc");

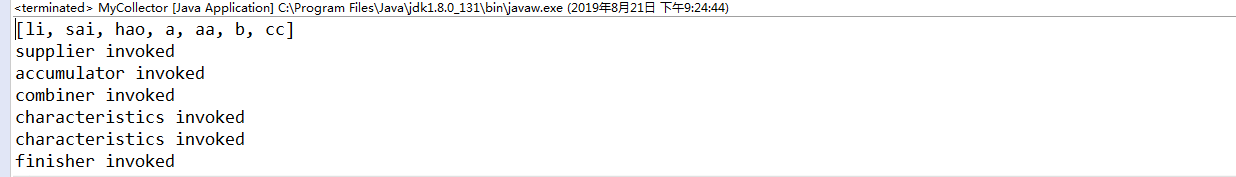
System.***out***.println(list);

Map<String,String> map = list.stream().collect(**new** MyCollector<String>());

System.***out***.println(map);

}

}



该类为自定的收集器，该收集器将一个集合转换为Map，该类的生成的中间容器类型为Set，返回的结果类型为Map。

修改characteristics()方法：

@Override

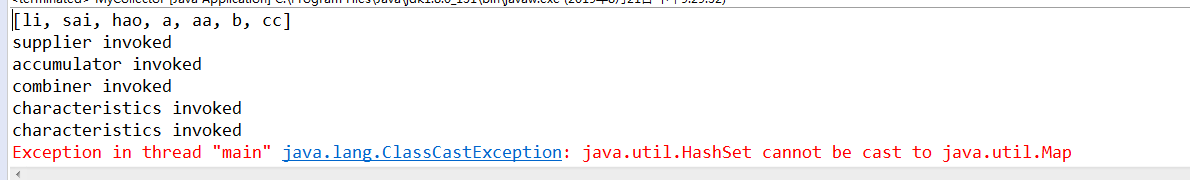
**public** Set<Characteristics> characteristics() {

System.***out***.println("characteristics invoked");

**return** Collections.*unmodifiableSet*(EnumSet.*of*(Characteristics.***UNORDERED***,Characteristics.***IDENTITY\_FINISH***));

}

运行结果：



我们在characteristics()方法返回结果中添加了Characteristics.IDENTITY\_FINISH类型，该类型表示finisher方法可以被省略，如果设置了，则一个未检查的类型转换必须成功，而这里我们定义了，则将中间容器类型Set强制转换为Map类型，从而发生类型转换异常。

##### Characteristics.CONCURRENT

Characteristics.CONCURRENT：表明这个收集器是并行的，表示对个线程可以并发操作相同的结果容器。

**import** java.util.\*;

**import** java.util.function.\*;

**import** java.util.stream.\*;

**public** **class** MyCollector<T> **implements** Collector<T,Set<T>,Map<T,T>>

{

@Override

**public** Supplier<Set<T>> supplier()

{

System.***out***.println("supplier invoked");

**return** HashSet<T>::**new**;

}

@Override

**public** BiConsumer<Set<T>, T> accumulator()

{

System.***out***.println("accumulator invoked");

**return** (set,item)->{

System.***out***.println("accumulator："+set+Thread.*currentThread*().getName());

set.add(item);

};

}

@Override

**public** BinaryOperator<Set<T>> combiner() {

System.***out***.println("combiner invoked");

**return** (set1,set2)->{set1.addAll(set2);**return** set1;};

}

@Override

**public** Function<Set<T>, Map<T, T>> finisher()

{

System.***out***.println("finisher invoked");

**return** set->{

Map<T,T> map = **new** HashMap<>();

set.stream().forEach(item->map.put(item, item));

**return** map;

};

}

@Override

**public** Set<Characteristics> characteristics() {

System.***out***.println("characteristics invoked");

**return** Collections.*unmodifiableSet*(EnumSet.*of*(Characteristics.***UNORDERED***,Characteristics.***CONCURRENT***));

}

**public** **static** **void** main(String[] args) {

**for**(**int** i = 0 ;i<10000;i++)

{

List<String> list = Arrays.*asList*("li","sai","hao","a","aa","b","cc","dd","bb","f","ff");

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

Map<String,String> map = list.parallelStream().collect(**new** MyCollector<String>());

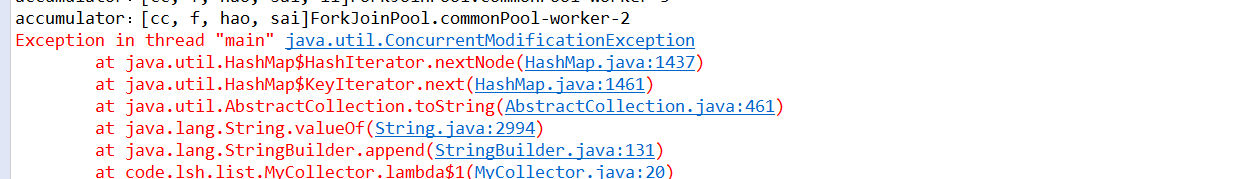
System.***out***.println(map);

}

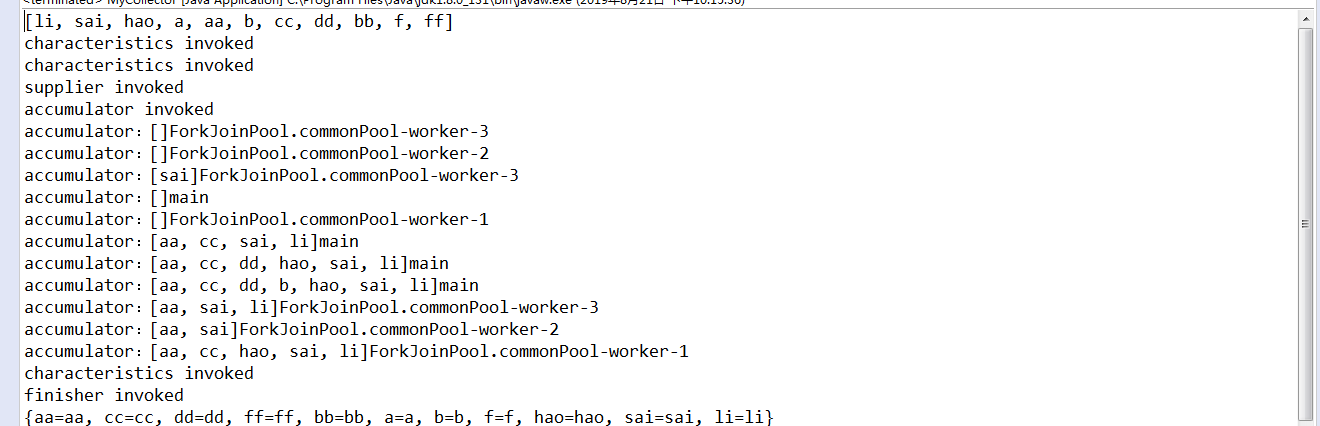
}

}

这里循环执行1000次：发生了异常。

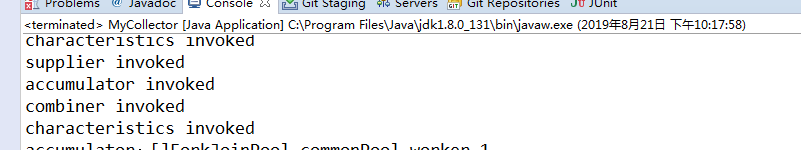


ConcurrentModificationException表明了一个线程对容器进行修改，而其他线程对集合进行遍历，因此发生了异常了。这表明了多个线程对同一结果容器进行操作。



combiner()方法并没有被执行，表明了多个线程操作了相同的结果容器。

修改characteristics()方法：去掉Characteristics.CONCURRENT。



combiner()方法执行，表明了多个线程操作不同的结果容器，最后这些容器会在combiner()方法中进行合并。

### Collectors方法介绍

对于Collectors静态工厂类来说，其实现一共分为两种情况：1.通过CollectorImpl来实现。

2.通过reducing方法来实现；reducing方法本身有通过CollectorImpl实现的

在Collectors中，如果方法返回的类型为收集器类型，而方法参数有接受一个收集器，通常将方法参数的收集器命名为下游（downstream）收集器。

Collectors定义了一个静态内部类（CollecorImpl），该类实现了Collector接口。

**public** **static** <T>

Collector<T, ?, List<T>> toList() {

**return** **new** CollectorImpl<>((Supplier<List<T>>) ArrayList::**new**, List::add,

(left, right) -> { left.addAll(right); **return** left; },

***CH\_ID***);

}

该方法将流中的元素保存到List集合中，该方法直接返回CollectorImpl对象，CH\_ID：表示了不会调用finisher方法。

**public** **static** <T, C **extends** Collection<T>>

Collector<T, ?, C> toCollection(Supplier<C> collectionFactory) {

**return** **new** CollectorImpl<>(collectionFactory, Collection<T>::add,

(r1, r2) -> { r1.addAll(r2); **return** r1; },

***CH\_ID***);

}

toList是toCollection的一个具体表现，toCollection方法必须传入一个结果容器类型，可以是List，ArrayList，LinkedList，Set，HashSet，TreeSet等。

**public** **static** <T>

Collector<T, ?, Set<T>> toSet() {

**return** **new** CollectorImpl<>((Supplier<Set<T>>) HashSet::**new**, Set::add,

(left, right) -> { left.addAll(right); **return** left; },

***CH\_UNORDERED\_ID***);

}

CH\_UNORDER\_ID：表示无序，并且中间结果容器和返回结果类型一样，不需要调用finisher方法。

**public** **static** Collector<CharSequence, ?, String> joining() {

**return** **new** CollectorImpl<CharSequence, StringBuilder, String>(

StringBuilder::**new**, StringBuilder::append,

(r1, r2) -> { r1.append(r2); **return** r1; },

StringBuilder::toString, ***CH\_NOID***);

}

该方法将流中的元素拼接成一个字符串，流中的元素必须是字符串（第一个参数为实现了CharSequence接口），而中间结果类型为StringBuilder而返回类型为String，所以会调用finisher函数。

**public** **static** <T, U, A, R>

Collector<T, ?, R> mapping(Function<? **super** T, ? **extends** U> mapper,

Collector<? **super** U, A, R> downstream) {

BiConsumer<A, ? **super** U> downstreamAccumulator = downstream.accumulator();

**return** **new** CollectorImpl<>(downstream.supplier(),

(r, t) -> downstreamAccumulator.accept(r, mapper.apply(t)),

downstream.combiner(), downstream.finisher(),

downstream.characteristics());

}

适配一个收集器，在accumulation之前，调用mapping函数将每一个输入元素接受T类型映射成接受U类型。

**public** **static**<T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream,

Function<R,RR> finisher) {

Set<Collector.Characteristics> characteristics = downstream.characteristics();

**if** (characteristics.contains(Collector.Characteristics.***IDENTITY\_FINISH***)) {

**if** (characteristics.size() == 1)

characteristics = Collectors.***CH\_NOID***;

**else** {

characteristics = EnumSet.*copyOf*(characteristics);

characteristics.remove(Collector.Characteristics.***IDENTITY\_FINISH***);

characteristics = Collections.*unmodifiableSet*(characteristics);

}

}

**return** **new** CollectorImpl<>(downstream.supplier(),

downstream.accumulator(),

downstream.combiner(),

downstream.finisher().andThen(finisher),

characteristics);

}

该方法首先将Characteristics.***IDENTITY\_FINISH***移除，这是因为执行finisher函数。该方法return语句中，第四个参数，首先downstream的finisher函数，后调用了Function的andThen方法传入我们代用该方法出入的finisher函数。 ，

**public** **static** <T> Collector<T, ?, Integer>

summingInt(ToIntFunction<? **super** T> mapper) {

**return** **new** CollectorImpl<>(

() -> **new** **int**[1],

(a, t) -> { a[0] += mapper.applyAsInt(t); },

(a, b) -> { a[0] += b[0]; **return** a; },

a -> a[0], ***CH\_NOID***);

}

这里使用数据类型，而不使用数字代替数组主要是因为数字本身是值类型，而数组为引用类型

案例：计算流中的数值元素的总和：

List<Integer> list = Arrays.*asList*(1,2,3,4,5) ;

**int** sum = list.stream().collect(Collectors.*summingInt*(i->i));

System.***out***.println(sum);

分组方法：

**public** **static** <T, K> Collector<T, ?, Map<K, List<T>>>

groupingBy(Function<? **super** T, ? **extends** K> classifier) {

**return** *groupingBy*(classifier, *toList*());

}

该方法直接返回groupingBy的重载方法，方法参数类型为Function类型，接受一个T类型和T类型之上的元素，返回一个K类型和K类型之下的元素。该方法返回的收集器对象中返回类型为Map<K,List<T>>，即key的类型为分类器(classifier)类型，而value值是与key关联的T。

**public** **static** <T, K, A, D>

Collector<T, ?, Map<K, D>> groupingBy(Function<? **super** T, ? **extends** K> classifier,

Collector<? **super** T, A, D> downstream) {

**return** *groupingBy*(classifier, HashMap::**new**, downstream);

}

该方法接受两个方法参数，一个是分类器Function，另一个是收集器downstream。收集器中元素类型为T和T之上的类型，中间容器类型为A，返回结果类型为D，所以方法返回的收集器中返回类型为Map<K,D>。

**public** **static** <T, K, D, A, M **extends** Map<K, D>>

Collector<T, ?, M> groupingBy(Function<? **super** T, ? **extends** K> classifier,

Supplier<M> mapFactory,

Collector<? **super** T, A, D> downstream) {

Supplier<A> downstreamSupplier = downstream.supplier();

BiConsumer<A, ? **super** T> downstreamAccumulator = downstream.accumulator();

//作为方法返回类型的参数。使用了分类器返回的结果类型作为K，使用下游收集器中间容器类型作为方法返回收集器的中//间容器类型。

BiConsumer<Map<K, A>, T> accumulator = (m, t) -> {

K key = Objects.*requireNonNull*(classifier.apply(t), "element cannot be mapped to a null key");

A container = m.computeIfAbsent(key, k -> downstreamSupplier.get());

downstreamAccumulator.accept(container, t);

};

BinaryOperator<Map<K, A>> merger = Collectors.<K, A, Map<K, A>>*mapMerger*(downstream.combiner());

@SuppressWarnings("unchecked")

Supplier<Map<K, A>> mangledFactory = (Supplier<Map<K, A>>) mapFactory;

**if** (downstream.characteristics().contains(Collector.Characteristics.***IDENTITY\_FINISH***)) {

**return** **new** CollectorImpl<>(mangledFactory, accumulator, merger, ***CH\_ID***);

}

**else** {

@SuppressWarnings("unchecked")

Function<A, A> downstreamFinisher = (Function<A, A>) downstream.finisher();

Function<Map<K, A>, M> finisher = intermediate -> {

intermediate.replaceAll((k, v) -> downstreamFinisher.apply(v));

@SuppressWarnings("unchecked")

M castResult = (M) intermediate;

**return** castResult;

};

**return** **new** CollectorImpl<>(mangledFactory, accumulator, merger, finisher, ***CH\_NOID***);

}

}

返回一个收集器，对T类型的输入元素实现了层叠的“group by”操作。根据分类器（classifier）函数实现元素的分组，使用下游收集器，对给定的key有关联的value执行汇聚操作。通过收集器的supplier工厂函数生成一个Map集合。

# Comparator接口

**public** **class** Demo4

{

**public** **static** **void** main(String[] args)

{

List<String> list = Arrays.*asList*("li","sai","hao");

Collections.*sort*(list,(item1,item2)->item1.length()-item2.length());

//根据字符串的长度降序排列

Collections.*sort*(list,Comparator.*comparingInt*(String::length).reversed());

//这里为哈编译器无法推断item类型，

Collections.*sort*(list,Comparator.*comparingInt*((String item)->item.length()).

reversed());

//先根据字符串大小升序排列，对于长度相等的字符串根据ASCLL码升序排序 list.sort(Comparator.*comparingInt*(String::length).thenComparing(String.***CASE\_INSENSITIVE\_ORDER***));

//先根据字符串大小升序排列，对于长度相等的字符串根据ASCLL码降序排序

list.sort(Comparator.*comparingInt*(String::length).thenComparing(String::toLowerCase,

Comparator.*reverseOrder*()));

}

}

## 问题

**Lambda类型推断为问题：**

**public** **static** **void** main(String[] args)

{

List<String> list = Arrays.*asList*("li","sai","hao");

//1

Collections.*sort*(list,(item1,item2)->item1.length()-item2.length());

//2

Collections.*sort*(list,Comparator.*comparingInt*(String::length).reversed());

//3

Collections.*sort*(list,Comparator.*comparingInt*((String item)->item.length()).reversed());

//4

Collections.*sort*(list,Comparator.*comparingInt*( item->item.length()));

}

在3中如果不显示指定item类型，则编译器为我们认为item为Object类型。

在3中我们必须指定item类型，而如果我们将reversed()方法移除，则不需要指定item类型，这是为什么？

# 文件I/O

文件处理：使用BufferedReader或Files类定义的静态方法lines，以流的形式返回文件内容。

public static void main(String[] args)throws Exception{  
 String path = "F:\\hao\\word.txt";  
 Files.*lines*(Paths.*get*(path))  
 .filter(s->s.length()>10)  
 .sorted(Comparator.*comparingInt*(String::length).reversed())  
 .limit(8)  
 .forEach(System.*out*::println);  
}

流形式检索文件：使用Files的list方法

Files.*list*(Paths.*get*("F:\\hao"))  
 .forEach(System.*out*::println);

文件系统遍历：使用Files的walk方法。

Files.*walk*(Paths.*get*("F:\\hao"))  
 .forEach(System.*out*::println);

# 日期API

Java.time包的开发遵守了JSR 310规范（Date-Time API），并支持ISO 8601标准，且对闰年和个别地区实行的夏时制规则做了相应的调整。

时区ID包括两种形式： 1.相对于UTC/格林尼治标准时间的固定偏移量 如： -05：00

2.地理区域，如America/Chicago

3.相对于祖鲁时间的偏移量。

获取可用时区ID：

Set<String> regionNames = ZoneId.*getAvailableZoneIds*();  
regionNames.stream().forEach(System.*out*::println);

## Date-Time API的基本类

Instant，Duration，Period，LocalDate，LocalTime，LocalDatetime，ZonedDateTime等类定义的工厂方法。

Instant类：对时间轴上的单一瞬时点建模，可以用于记录应用程序中的事件时间

System.*out*.println(Instant.*now*());  
System.*out*.println(LocalDate.*now*());  
System.*out*.println(LocalTime.*now*());  
System.*out*.println(LocalDateTime.*now*());  
System.*out*.println(ZonedDateTime.*now*());

我们也可以通过静态工厂方法of用于生成新的值。

LocalTime类定义的of方法包括多种重载形式，根据可用的小时，分，秒以及纳秒值获取当前日期。

LocalDateTime类定义的of方法同样包括了多种重载方法，根据可用的年，月，日，小时，分，秒以及纳秒值获取当前日期和时间。

**注意：**of方法的月份字段经过重载，以接受Month枚举或起始值为1的整数。

LocalDate案例：

**public** **static** **void** main(String[] args)

{

LocalDate localDate = LocalDate.*now*();

System.***out***.println(localDate);

**int** year = localDate.getYear();

**int** month = localDate.getMonthValue();

**int** day = localDate.getDayOfMonth();

LocalDate ld = LocalDate.*of*(2019, 8, 16);

System.***out***.println(ld);

MonthDay md1 = MonthDay.*of*(month, day);

MonthDay md2 = MonthDay.*from*(ld);

**if**(md1.equals(md2))

System.***out***.println("equals");

**else**

System.***out***.println("not equals");

}

Date-Time API中各种方法所有的前缀：

|  |  |  |
| --- | --- | --- |
| 方法 | 类型 | 用途 |
| of | 静态工厂 | 创建实例 |
| from | 静态工厂 | 将输入参数转换为目标类的实例 |
| parse | 静态工厂 | 解析输入字符串 |
| format | 实例 | 生成格式化输出 |
| get | 实例 | 返回对象状态的一部分 |
| is | 实例 | 查询对象状态 |
| with | 实例 | 通过修改现有对象的某个元素来创建新对象 |
| plus,minus | 实例 | 分别通过现有对象的增减来创建新对象 |
| to | 实例 | 将对象转换为另一种类型 |
| at | 实例 | 将对象与另一个对象合并 |

LocalDateTime添加时区：

LocalDateTime dateTime = LocalDateTime.*of*(2019,Month.*SEPTEMBER*,23,20,32,56);  
ZonedDateTime zdt = dateTime.atZone(ZoneId.*of*("Asia/Shanghai"));  
System.*out*.println(zdt);

Month枚举：

System.*out*.println("Days is Feb in a leap year:"+Month.*FEBRUARY*.length(true));  
System.*out*.println("Days of year for first day of Sept(leap year):"+  
Month.*SEPTEMBER*.firstDayOfYear(true));  
System.*out*.println("Month.of(1):"+Month.*of*(1));  
System.*out*.println("Adding two months:"+Month.*JANUARY*.plus(2));  
System.*out*.println("Subtracting a month:"+Month.*SEPTEMBER*.minus(1));

LocalDateTime dateTime = LocalDateTime.*now*();  
DateTimeFormatter formatter = DateTimeFormatter.*ofPattern*("yyyy-MM-dd HH:mm:ss");  
ZonedDateTime zdt = dateTime.atZone(ZoneId.*of*("Asia/Shanghai"));  
System.*out*.println(dateTime.format(formatter));  
//增加三天  
LocalDateTime dateTime1 = dateTime.plusDays(3);  
System.*out*.println(dateTime1.format(formatter));  
//增加三周  
LocalDateTime dateTime2 = dateTime.plusWeeks(3);  
System.*out*.println(dateTime2.format(formatter));  
//增加三个月  
LocalDateTime dateTime3 = dateTime.plusMonths(3);  
System.*out*.println(dateTime3.format(formatter));  
//增加三年  
LocalDateTime dateTime4 = dateTime.plusYears(3);  
System.*out*.println(dateTime4.format(formatter));

LocalDateTime.plus方法和minus方法：

public LocalDateTime plus(long amountToAdd, TemporalUnit unit) {

public LocalDateTime plus(TemporalAmount amountToAdd) {

LocalDateTime dateTime5 = dateTime.plus(2, ChronoUnit.*DAYS*);  
System.*out*.println(dateTime5.format(formatter));  
LocalDateTime dateTime6 = dateTime.plus(2,ChronoUnit.*SECONDS*);  
System.*out*.println(dateTime6.format(formatter));

对于传入TemporalAmount的方法而言，参数通常是Period或Duration，但也可以是任何实现TemporalAmount接口类型。

//增加两年3个月零4天  
Period perioad = Period.*of*(2,3,4);  
LocalDateTime dateTime7 = dateTime6.plus(perioad);  
System.*out*.println(dateTime7.format(formatter));