法务Java后端技术培训

基础篇V1.0

目录

一、 引言 11

二、 Java8特性 12

1. Lambda表达式 12

1.1. 为什么使用Lambda表达式 12

1.2. Lambda表达式的基础语法 12

1.2.1. 语法格式一：无参数，且无返回值 12

1.2.2. 语法格式二：有一个参数，并且无返回值 15

1.2.3. 语法格式三：若只有一个参数，参数的小括号可以省略不写 17

1.2.4. 语法格式四：有两个以上的参数，有返回值，并且Lambda体中有多条语句 17

1.2.5. 语法格式五：当Lambda体中只有一条语句时，return和大括号都可以省略不写 18

1.2.6. 语法格式六：Lambda表达式中的参数列表的数据类型可以省略不写 18

1.3. Lambda表达式实例（获取薪水不低于5000元的员工信息） 18

2. 方法引用和构造器引用 22

2.1. 为何要使用方法引用与构造器引用？ 22

2.2. 方法引用 22

2.2.1. 方法引用是什么？ 22

2.2.2. 方法引用的语法格式 23

2.2.2.1. 第一种语法格式：对象::实例方法名 23

2.2.2.2. 第二种语法格式：类::静态方法名 26

2.2.2.3. 第三种语法格式：类::实例方法名 26

2.2.3. 方法引用实例 27

2.2.3.1. 第一种语法格式：对象::实例方法名 27

2.2.3.2. 第二种语法格式：类::静态方法名 28

2.2.3.3. 第三种语法格式：类::实例方法名 29

2.3. 构造器引用 30

2.3.1. 构造器引用实例 32

2.3.1.1. 不带参的构造器引用 34

2.3.1.2. 带参的构造器引用 35

2.4. 数组引用 37

2.4.1. 数组引用实例 37

3. 函数式接口 38

3.1. 什么是函数式接口 38

3.2. Java 8中内置的四大核心函数式接口 39

3.2.1. 消费型接口 40

3.2.2. 供给型接口 40

3.2.3. 函数型接口 41

3.2.4. 断言型接口 42

3.2.5. 其他接口 43

3.3. 函数式接口实例 44

4. Stream API 45

4.1. 什么是Stream？ 45

4.2. Stream使用的三个步骤 46

4.3. 如何创建Stream？ 46

4.3.1. 通过Collection接口创建 46

4.3.2. 通过数组创建 47

4.3.3. 通过Stream类中的静态方法of()创建 48

4.3.4. 通过静态方法Stream.iterate()或者Stream.generate()创建无限流 48

4.3.4.1. 迭代 48

4.3.4.2. 生成 49

4.4. 中间操作 50

4.4.1. 筛选与切片 50

4.4.1.1. filter(Predicate p) 52

4.4.1.2. limit(long maxSize) 53

4.4.1.3. skip(long n) 54

4.4.1.4. distinct() 55

4.4.2. 映射 58

4.4.2.1. map(Function f) 58

4.4.2.2. flatMap(Function f) 59

4.4.3. 排序 61

4.4.3.1. sorted() 62

4.4.3.2. sorted(Comparator comp) 63

4.5. 终止操作 64

4.5.1. 查找与匹配 64

4.5.1.1. allMatch(Predicate p) 65

4.5.1.2. anyMatch(Predicate p) 66

4.5.1.3. noneMatch(Predicate p) 67

4.5.1.4. findFirst() 68

4.5.1.5. findAny() 69

4.5.1.6. count() 70

4.5.1.7. max(Comparator c) 71

4.5.1.8. min(Comparator c) 72

4.5.2. 归约 72

4.5.2.1. reduce(T identity，Binaryoperator b) 73

4.5.2.2. reduce(Binaryoperator b) 73

4.5.3. 收集 74

4.5.3.1. toList() 74

4.5.3.2. toSet() 75

4.5.3.3. toCollection(…) 76

4.5.3.4. counting() 77

4.5.3.5. averagingDouble(…) 78

4.5.3.6. summingDouble(…) 79

4.5.3.7. maxBy(…) 79

4.5.3.8. minBy(…) 80

4.5.3.9. groupingBy(…) 80

4.5.3.10. partitioningBy(…) 81

4.5.3.11. summarizingDouble(…) 82

4.5.3.12. joining(…) 83

4.6. Stream API实例 84

5. 并行流与串行流 85

5.1. Fork/Join框架 85

5.1.1. 了解Fork/Join框架 85

5.1.2. Fork/Join框架与传统线程池的区别 86

5.1.3. 小结 87

5.2. 并行流 88

5.2.1. Fork/Join框架的简单使用 88

6. 重复注解与类型注解 88

6.1. 重复注解 88

6.1.1. 重复注解是什么？ 88

6.1.2. 如何定义重复注解？ 88

6.2. 类型注解 89

7. 接口中的默认方法和静态方法 89

7.1. 默认方法 89

7.2. 静态方法 92

8. 传统日期时间格式化的多线程安全问题 92

9. 全新的日期时间API 94

9.1. 日期、时间和日期时间 94

9.2. 常用方法 94

9.3. 时间戳 96

9.4. 间隔 97

9.5. 时间校正器 100

9.6. 时区 101

三、 MySQL数据库 102

1. 单表查询 102

1.1. 查询所有字段 102

1.2. 查询指定字段 103

1.2.1. 查询单个字段 103

1.2.2. 查询多个字段 103

1.3. 查询结果不重复 103

1.4. 对查询结果排序 103

1.4.1. 升序排序（asc）,默认的排序方式，加与不加都可以 103

1.4.2. 降序排序（desc）,如果选择降序排序，必须末尾加关键字desc 103

1.5. 分组查询 103

1.6. 使用 LIMIT限制查询结果的数量 104

2. 条件查询 104

2.1. 查询指定记录 104

2.2. 带in关键字的查询 104

2.3. 带between and的范围查询 105

2.4. 查询空值 105

2.5. 带and的多条件查询 105

2.6. 带or的多条件查询 105

3. 多表联查（内连接） 106

4. 多表联查（外连接） 106

4.1. 左连接查询 left join 106

4.2. 右连接 right join 107

5. And和OR的优先级 107

6. 模糊查询 108

6.1. 通配符的分类 108

6.1.1. 百分号通配符'%'，匹配任意长度的字符，甚至包括零字符 108

6.1.2. 下划线通配符'\_',一次只能匹配任意一个字符 109

6.2. MySQL内置函数检索(locate,position,instr) 109

6.2.1. LOCATE()函数 109

6.2.1.1. LOCATE(substr,str) 109

6.2.1.2. LOCATE（substr, str, [pos]） 109

6.2.2. POSITION()方法 110

6.2.3. INSTR()方法 110

6.3. MySQL基于regexp、rlike的正则匹配查询 110

6.3.1. regexp中的 OR:| 111

6.3.2. REGEXP中的正则匹配:[] 111

6.3.3. 字符类匹配(posix) 111

6.3.4. [:<:]和[:>:] 112

7. 分组函数 112

7.1. 简单使用 112

7.2. 特点 112

7.2.1. 参数支持类型 112

7.2.2. 是否忽略NULL值 112

7.3. 可以和distinct搭配使用实现去重的运算 112

7.4. count函数的其他介绍 113

7.4.1. 统计总行数 113

7.4.2. 额外加上列全1，统计1的个数，即为统计总行数（也可以是其他数值或字符） 113

7.5. 和分组函数一同查询的字段有限制 113

7.6. datediff(日期1，日期2) 计算两个日期相差的天数 113

8. 分组查询 113

8.1. 创建分组 114

8.1.1. GROUP BY单独使用 114

8.1.2. GROUP BY与聚合函数 114

8.1.3. GROUP BY与GROUP\_CONCAT 115

8.1.4. 多字段分组 115

8.2. 使用HAVING过滤分组 115

8.3. GROUP BY与WITH ROLLUP 115

8.4. GROUP BY与ORDER BY 116

9. Union用法 116

四、 SpringBoot事务 116

1. 定义 116

2. 特性和实现 117

2.1. 事物的四个基本性质(ACID) 117

2.1.1. 原子性(Atomicity) 117

2.1.2. 一致性(Consistency) 117

2.1.3. 隔离性(Isolation) 117

2.1.3.1. 读未提交(Read uncommitted) 117

2.1.3.2. 读提交(Read ucommitted) 118

2.1.3.3. 重复读(Repeatable read) 118

2.1.3.4. 序列化(Serializable) 119

2.1.4. 持久性(Durability) 119

2.2. 事务的实现 119

2.2.1. 编程式事务 119

2.2.2. 声明式事务 119

3. 声明式事务的使用技巧 119

4. 声明式事务的常用配置 121

5. 声明式事务的隔离级别 122

6. 声明式事务的传播行为 122

五、 Exception异常 123

1. 异常的划分 123

2. 异常的语法 124

3. 异常的常用方法 125

4. 异常处理的原则 126

5. 自定义异常 126

6. Throw、Throws 127

7. Try-Catch-Finally 128

8. RuntimeException运行时异常 129

9. 异常在子父类中的体现 129

10. finally与return的执行顺序 130

11. 异常的注意事项 132

12. 常见的异常 133

六、 @Validated注解 134

1. 简述 134

七、 @RequestMapping 135

1. @Controller和@RestController 135

2. @Autowired 136

3. @RequestMapping 136

3.1. @RequestMapping 136

3.2. @GetMapping 136

3.3. @PostMapping 137

3.4. @PutMapping和@PatchMapping 137

3.5. @DeleteMapping 137

4. @ResponseBody 137

5. @PathVariable 138

6. @RequestParam 138

八、 Redis缓存 138

1. Redis数据存储结构 138

1.1. 数据结构 139

1.1.1. 字符串 139

1.1.2. 哈希 139

1.1.3. 列表 139

1.1.4. 集合 139

1.1.5. 有序集合 139

1.2. 存储结构 140

1.2.1. 字符串的存储结构 140

1.2.2. 哈希的存储结构 140

1.2.3. 列表的存储结构 140

1.2.4. 集合的存储结构 141

1.2.5. 有序列表的存储结构 141

2. Redis缓存生命周期 141

2.1. 为什么要设置key生存时间 141

2.2. 设置key的生存时间 142

2.3. 访问key的生存时间 142

2.4. 清除生存时间 142

3. Redis缓存穿透 142

4. Redis缓存击穿 145

4.1. 使用互斥锁 145

4.1.1. 单机环境 145

4.1.2. 分布式环境 146

4.2. 热点数据永不过期 146

4.3. 定时刷新 148

5. Redis缓存雪崩 148

5.1. 加锁排队 148

5.2. 设置过期时间随机值 149

5.3. 设置过期标志更新缓存 149

九、 SpringBoot入门 150

1. 创建demo项目（2.3.6-Release） 150

2. MySQL数据源配置 153

2.1. 导入依赖 153

2.2. 连接数据库 153

2.3. 测试 154

3. Redis缓存集成配置 157

4. 单表CURD操作 163

5. 销售订单作业练习 164

十、 其他 165

# 引言

本文主要介绍了法务产品Java后端培训基础知识，只要包含：Java8特性、MySQL数据库、Redis缓存、SpringBoot入门等，以帮助读者快速掌握产品的主要技术栈。

# Java8特性

在Spring5之后，框架设计上用到了很多Java8的特性，所以有必要把Java8常用的特性进行学习掌握。

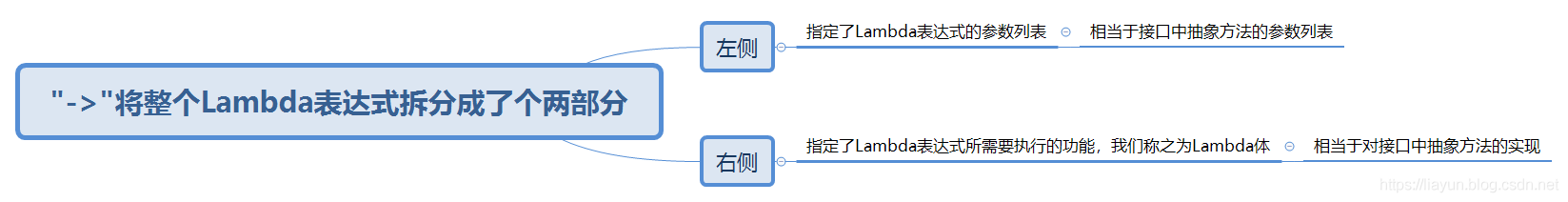
## Lambda表达式

### 为什么使用Lambda表达式

Lambda表达式可以大大简化匿名内部类的代码量，可以更加清晰地展示主要代码，以此增强整体代码的可读性。

### Lambda表达式的基础语法

在Java8中引入了一个新的操作符，即"->"，该操作符被称为箭头操作符或者Lambda操作符，箭头操作符将整个Lambda表达式拆分成了个两部分，如图1-1所示。

图 1‑1 Lambda表达式的基础语法

#### 语法格式一：无参数，且无返回值

第一种语法格式说的就是接口中的那个抽象方法无参，并且无返回值。例如Runnable接口，Runnable接口的定义如下：

package java.lang;  
  
*/\*\*  
 \* The <code>Runnable</code> interface should be implemented by any  
 \* class whose instances are intended to be executed by a thread. The  
 \* class must define a method of no arguments called <code>run</code>.  
 \* <p>  
 \* This interface is designed to provide a common protocol for objects that  
 \* wish to execute code while they are active. For example,  
 \* <code>Runnable</code> is implemented by class <code>Thread</code>.  
 \* Being active simply means that a thread has been started and has not  
 \* yet been stopped.  
 \* <p>  
 \* In addition, <code>Runnable</code> provides the means for a class to be  
 \* active while not subclassing <code>Thread</code>. A class that implements  
 \* <code>Runnable</code> can run without subclassing <code>Thread</code>  
 \* by instantiating a <code>Thread</code> instance and passing itself in  
 \* as the target. In most cases, the <code>Runnable</code> interface should  
 \* be used if you are only planning to override the <code>run()</code>  
 \* method and no other <code>Thread</code> methods.  
 \* This is important because classes should not be subclassed  
 \* unless the programmer intends on modifying or enhancing the fundamental  
 \* behavior of the class.  
 \*  
 \** ***@author*** *Arthur van Hoff  
 \** ***@see*** *java.lang.Thread  
 \** ***@see*** *java.util.concurrent.Callable  
 \** ***@since*** *JDK1.0  
 \*/*@FunctionalInterface  
public interface Runnable {  
 */\*\*  
 \* When an object implementing interface <code>Runnable</code> is used  
 \* to create a thread, starting the thread causes the object's  
 \* <code>run</code> method to be called in that separately executing  
 \* thread.  
 \* <p>  
 \* The general contract of the method <code>run</code> is that it may  
 \* take any action whatsoever.  
 \*  
 \** ***@see*** *java.lang.Thread#run()  
 \*/* public abstract void run();  
}

这里就以Runnable接口为例来讲述这种语法格式。原来我们是使用匿名内部类的方式来实现该接口的，就像下面这样：

Runnable r = new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("Hello World!" + num);  
 }  
};  
r.run();

现在，匿名内部类可以使用Lambda表达式来充当了，即使用Lambda表达式实现Runnable接口：

Runnable r1 = () -> System.*out*.println("Hello Lambda!" + num);  
r1.run();

#### 语法格式二：有一个参数，并且无返回值

第二种语法格式说的就是接口中的那个抽象方法有一个参数，并且无返回值。例如Consumer接口，Consumer接口的定义如下：

package java.util.function;  
  
import java.util.Objects;  
  
*/\*\*  
 \* Represents an operation that accepts a single input argument and returns no  
 \* result. Unlike most other functional interfaces, {****@code*** *Consumer} is expected  
 \* to operate via side-effects.  
 \*  
 \* <p>This is a <a href="package-summary.html">functional interface</a>  
 \* whose functional method is {****@link*** *#accept(Object)}.  
 \*  
 \** ***@param*** <*T*> *the type of the input to the operation  
 \*  
 \** ***@since*** *1.8  
 \*/*@FunctionalInterface  
public interface Consumer<T> {  
  
 */\*\*  
 \* Performs this operation on the given argument.  
 \*  
 \** ***@param*** *t the input argument  
 \*/* void accept(T t);  
  
 */\*\*  
 \* Returns a composed {****@code*** *Consumer} that performs, in sequence, this  
 \* operation followed by the {****@code*** *after} operation. If performing either  
 \* operation throws an exception, it is relayed to the caller of the  
 \* composed operation. If performing this operation throws an exception,  
 \* the {****@code*** *after} operation will not be performed.  
 \*  
 \** ***@param*** *after the operation to perform after this operation  
 \** ***@return*** *a composed {****@code*** *Consumer} that performs in sequence this  
 \* operation followed by the {****@code*** *after} operation  
 \** ***@throws*** *NullPointerException if {****@code*** *after} is null  
 \*/* default Consumer<T> andThen(Consumer<? super T> after) {  
 Objects.*requireNonNull*(after);  
 return (T t) -> { accept(t); after.accept(t); };  
 }  
}

这里就以Consumer接口为例来讲述这种语法格式。我们可以使用Lambda表达式对Consumer接口中有一个参数的accept方法进行实现：

@Test  
public void test2() {  
 Consumer<String> con = (x) -> System.*out*.println(x);  
 con.accept("我是李宇航！");  
}

#### 语法格式三：若只有一个参数，参数的小括号可以省略不写

第三种语法格式说的就是接口中的那个抽象方法如果只有一个参数，那么参数的小括号可以省略不写了，但习惯上还是要写上。还是以上面的Consumer接口为例来讲述这种语法格式：

@Test  
public void test2() {  
// Consumer<String> con = (x) -> System.out.println(x);  
 Consumer<String> con = x -> System.*out*.println(x);  
 con.accept("我是李宇航！");  
}

#### 语法格式四：有两个以上的参数，有返回值，并且Lambda体中有多条语句

第四种语法格式说的就是接口中的那个抽象方法有两个以上的参数，有返回值，并且Lambda体中有多条语句，那么此时Lambda体必须使用大括号括起来。这里就以Comparator接口为例来讲述这种语法格式：

@Test  
public void test3() {  
 Comparator<Integer> com = (x, y) -> {  
 System.*out*.println("函数式接口");  
 return Integer.*compare*(x, y);  
 };  
}

#### 语法格式五：当Lambda体中只有一条语句时，return和大括号都可以省略不写

第五种语法格式说的就是接口中的那个抽象方法有两个以上的参数，有返回值，并且Lambda体中只有一条语句，那么此时return和大括号都可以省略不写。这里同样以Comparator接口为例来讲述这种语法格式：

@Test  
public void test4() {  
 Comparator<Integer> com = (x, y) -> Integer.*compare*(x, y);  
}

#### 语法格式六：Lambda表达式中的参数列表的数据类型可以省略不写

最后这种语法格式说的就是Lambda表达式中的参数列表的数据类型可以省略不写，因为JVM的编译器可以通过上下文来推断出数据类型，这个过程我们称之为"类型推断"。

### Lambda表达式实例（获取薪水不低于5000元的员工信息）

创建Employee实体类：

package Lamda;  
  
public class Employee {  
 private int id;  
 private String name;  
 private int age;  
 private double salary;  
  
 public Employee() {  
 }  
  
 public Employee(int id, String name, int age, double salary) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.salary = salary;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(double salary) {  
 this.salary = salary;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", age=" + age +  
 ", salary=" + salary +  
 '}';  
 }  
}

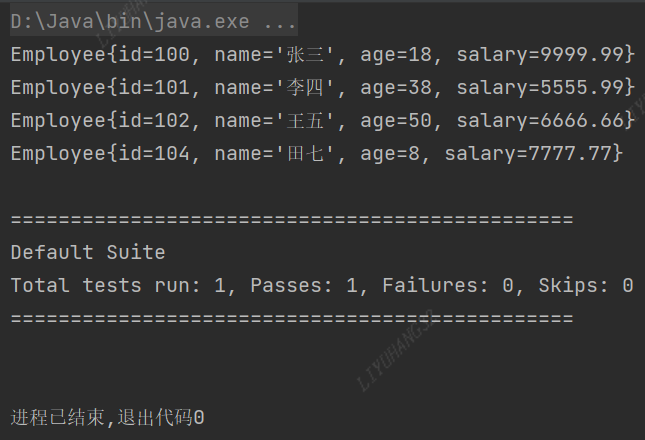
创建MyPredicate接口：

package Lamda;  
  
public interface MyPredicate<T>{  
 public boolean test(T t);  
}

实现方法（LambdaDemo类）：

package Lamda;  
  
import org.testng.annotations.Test;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class Lamdademo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99),  
 new Employee(101, "李四", 38, 5555.99),  
 new Employee(102, "王五", 50, 6666.66),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(104, "田七", 8, 7777.77)  
 );  
  
 public List<Employee> filterEmployee(List<Employee> list, MyPredicate mp) {  
 List<Employee> emps = new ArrayList<Employee>();  
 for (Employee : list)  
 if (mp.test(employee))  
 emps.add(employee);  
 return emps;  
 }  
  
 @Test  
 /\*-----使用Lamda表达式-----\*/  
 public void test4() {  
 List<Employee> list = filterEmployee(employees, (MyPredicate<Employee>) e -> e.getSalary() >= 5000);  
 list.forEach(System.*out*::println);  
 }  
}

运行结果如图1-2所示。

图 1‑2 运行结果

## 方法引用和构造器引用

### 为何要使用方法引用与构造器引用？

如果不使用Lambda表达式进行程序编写的话，那么大可不必关注方法引用和构造器引用，但是如果使用Lambda表达式，再配合方法引用和构造器引用之后，那么可以使Lambda编写匿名内部类的代码变得更加简洁。在不影响性能的前提下简洁的代码可以增强代码的可读性（当然是在阅读者知晓对方语法的前提下）

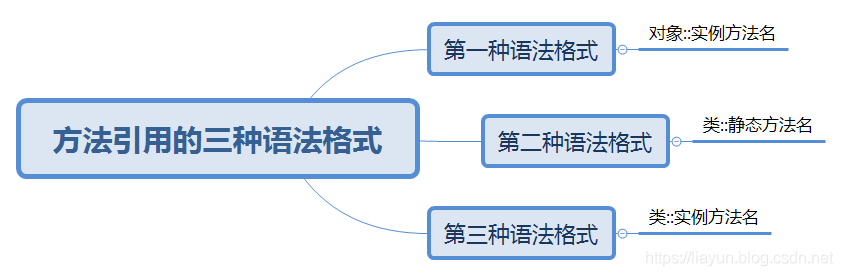
### 方法引用

#### 方法引用是什么？

当要传递给Lambda体的操作，已经有实现的方法了，那么这时便可以使用方法引用了。也可以理解为方法引用是Lambda表达式的另外一种表现形式。不过有一点需要我们注意：Lambda体中调用方法的参数列表与返回值类型，要与函数式接口中抽象方法的参数列表与返回值类型保持一致！

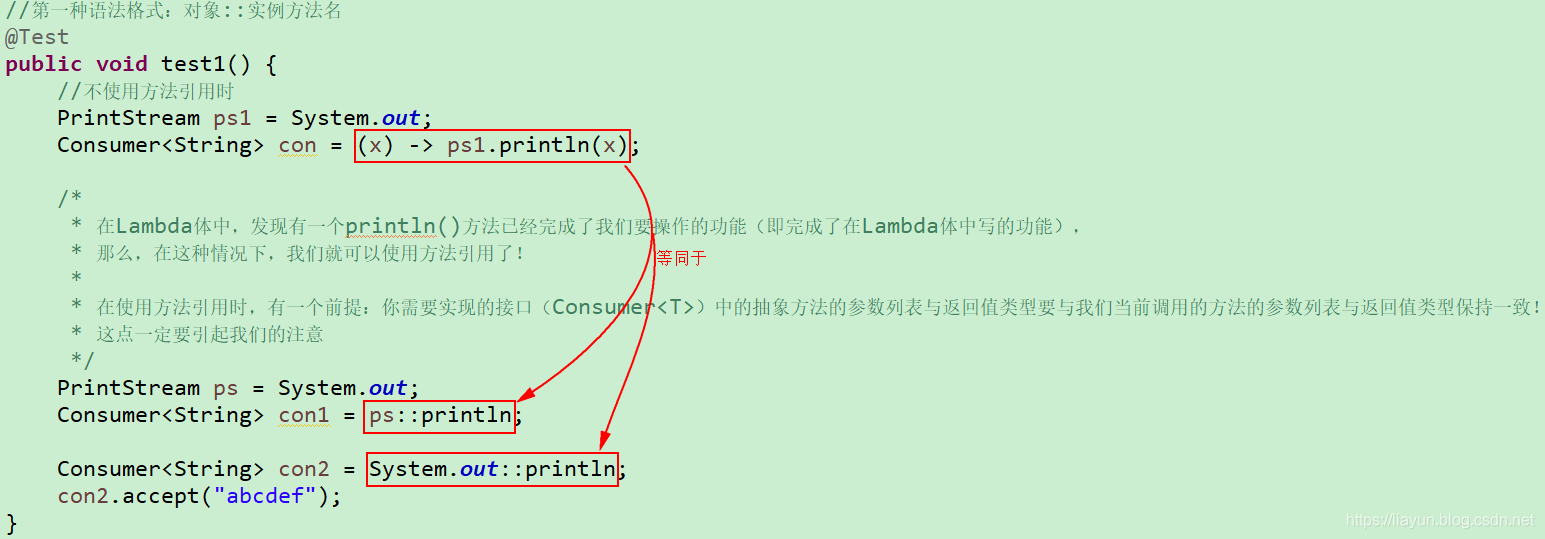
#### 方法引用的语法格式

方法引用使用操作符::将方法名和对象或类的名字分隔开来。方法引用主要有三种语法格式，它们分别是：对象::实例方法名、类::静态方法名、类::实例方法名（如图2-1所示），下面会分别一一介绍这三种语法格式。

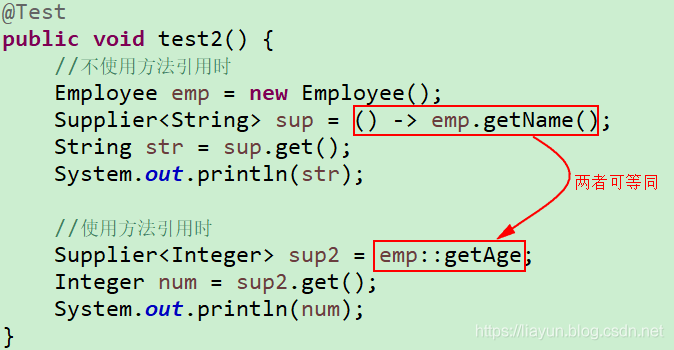
图 2‑1方法引用的语法格式

##### 第一种语法格式：对象::实例方法名

观察如下的Java程序（如图2-2所示），你会看到在Lambda体中有一个println()方法已经完成了我们要操作的功能（即完成了在Lambda体中所写的功能），那么，在这种情况下我们就可以使用方法引用了！

图 2‑2 第一种语法格式：对象::实例方法名(1)

在使用方法引用时，有一个前提：你需要实现的接口（例如Consumer<T>）中的抽象方法的参数列表与返回值类型要与我们当前调用的方法的参数列表与返回值类型保持一致！这点一定要引起我们的注意。

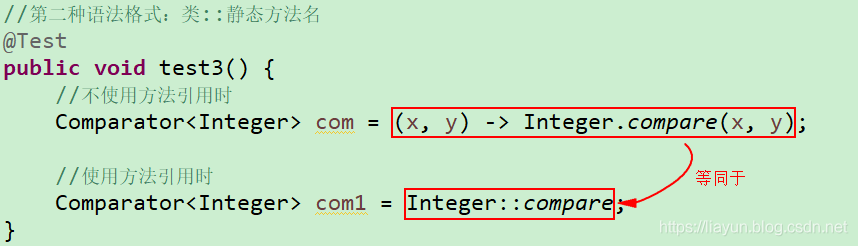
图 2‑3 第一种语法格式：对象::实例方法名(2)

其中，Employee实体类的内容如下：

package Methods;  
  
public class Employee {  
 private int id;  
 private String name;  
 private int age;  
 private double salary;  
  
 public Employee() {  
 }  
  
 public Employee(int id, String name, int age, double salary) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.salary = salary;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(double salary) {  
 this.salary = salary;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", age=" + age +  
 ", salary=" + salary +  
 '}';  
 }  
}

##### 第二种语法格式：类::静态方法名

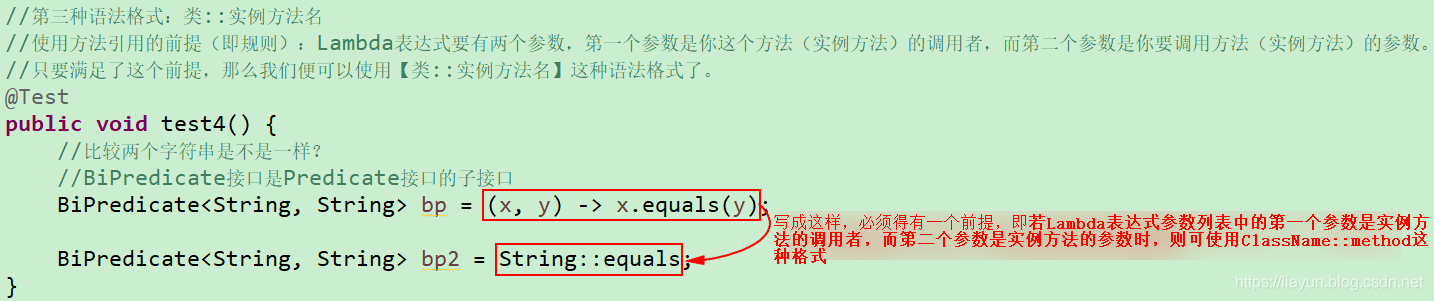
使用方法引用的这种格式（如图2-4所示），可以引用类的静态方法。

图 2‑4 第二种语法格式：类::静态方法名

你可以查看一下Integer.compare()方法的参数列表与返回值类型和Comparator接口中compare()方法的参数列表与返回值类型是不是一致的？显然是一致的。

##### 第三种语法格式：类::实例方法名

使用方法引用的这种格式得有一个前提，也可以说是规则，即若Lambda表达式参数列表中的第一个参数是你这个方法（实例方法）的调用者，而第二个参数是你要调用方法（实例方法）的参数时，则可使用ClassName::method这种格式（如图2-5所示）。

图 2‑5 第三种语法格式：类::实例方法名

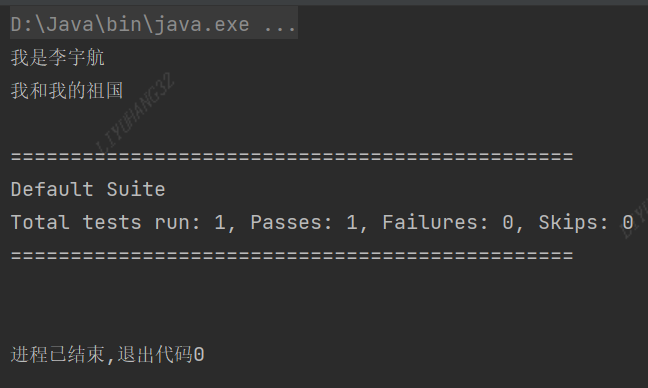
#### 方法引用实例

##### 第一种语法格式：对象::实例方法名

创建MethodsDemo类：

package Methods;  
  
import org.testng.annotations.Test;  
  
import java.io.PrintStream;  
import java.util.function.Consumer;  
  
public class MethodsDemo {  
 @Test  
 public void test() {  
 //不使用方法引用时  
 PrintStream ps1 = System.*out*;  
 Consumer<String> con = (method) -> System.*out*.println(method);  
 con.accept("我是李宇航");  
 //使用方法引用时  
 Consumer<String> con1 = ps1::println;  
 con1.accept("我和我的祖国");  
 }  
}

运行结果如图2-6所示。

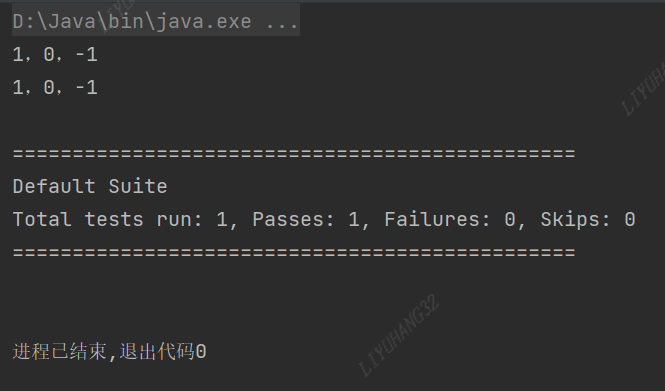
图 2‑6 第一种语法格式：对象::实例方法名

##### 第二种语法格式：类::静态方法名

创建MethodsDemo类：

package Methods;  
  
import org.testng.annotations.Test;  
  
import java.util.Comparator;  
  
public class MethodsDemo {  
 @Test  
 public void test() {  
 ////不使用方法引用时  
 Comparator<Integer> co1 = (x, y) -> Integer.*compare*(x, y);  
 Comparator<Integer> co2 = (x, y) -> Integer.*compare*(x, y);  
 Comparator<Integer> co3 = (x, y) -> Integer.*compare*(x, y);  
 System.*out*.println(co1.compare(50, 20) + "，" + co2.compare(50, 50) + "，" + co3.compare(20, 50));  
 //使用方法引用时  
 Comparator<Integer> com1 = Integer::*compare*;  
 Comparator<Integer> com2 = Integer::*compare*;  
 Comparator<Integer> com3 = Integer::*compare*;  
 System.*out*.println(com1.compare(50, 20) + "，" + com2.compare(50, 50) + "，" + com3.compare(20, 50));  
 }  
}

运行结果如图2-7所示。

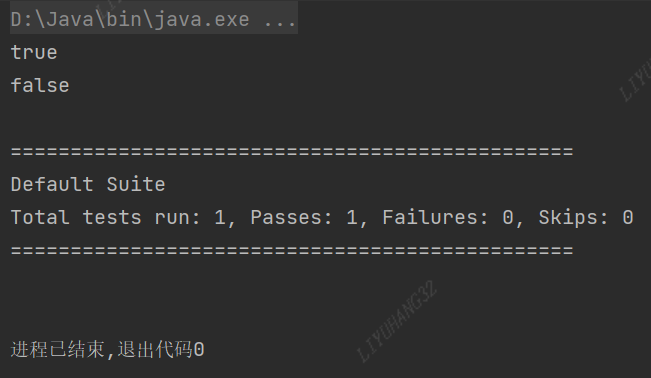
图 2‑7 第二种语法格式：类::静态方法名

##### 第三种语法格式：类::实例方法名

创建MethodsDemo类：

package Methods;  
  
import org.testng.annotations.Test;  
import java.util.function.BiPredicate;  
  
public class MethodsDemo {  
 @Test  
 public void test() {  
 //不使用方法引用时  
 BiPredicate<Integer,Integer> bp=(x,y)->x.equals(y);  
 System.*out*.println(bp.test(5,5));  
 //使用方法引用时  
 BiPredicate<String,String> bp1=String::equals;  
 System.*out*.println(bp1.test("李宇航","我和我的祖国"));  
 }  
}

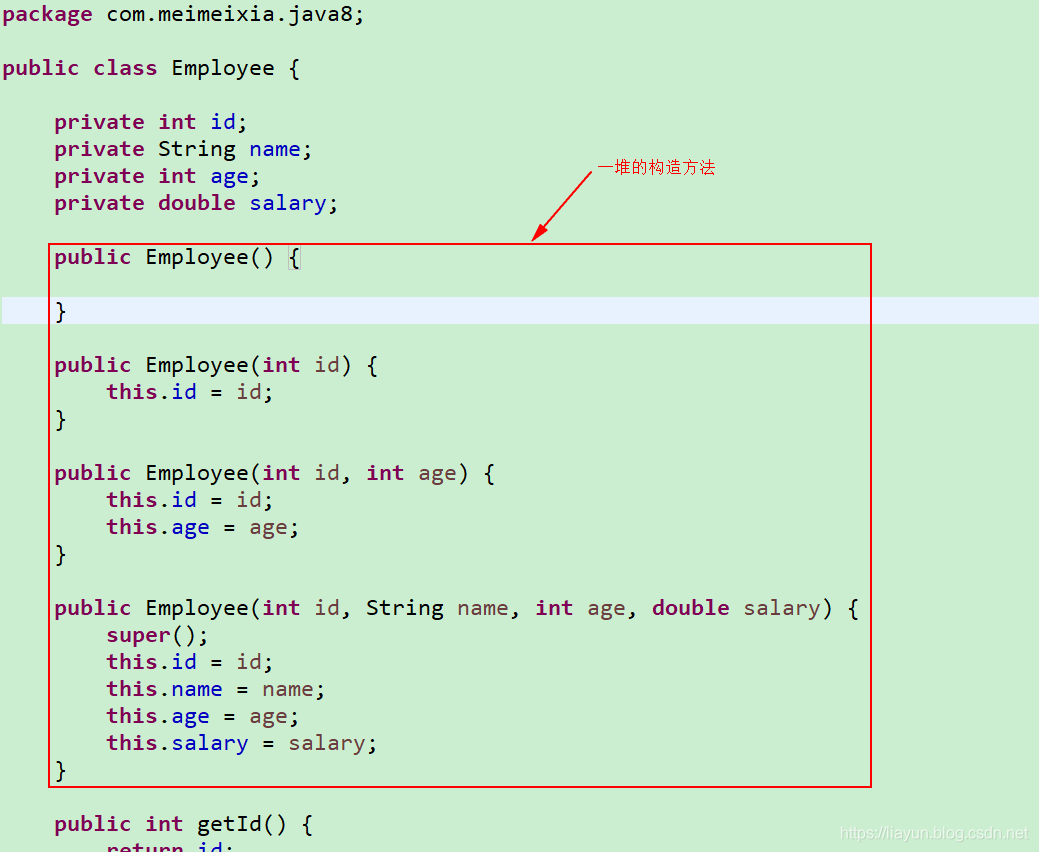
运行结果如图2-7所示。

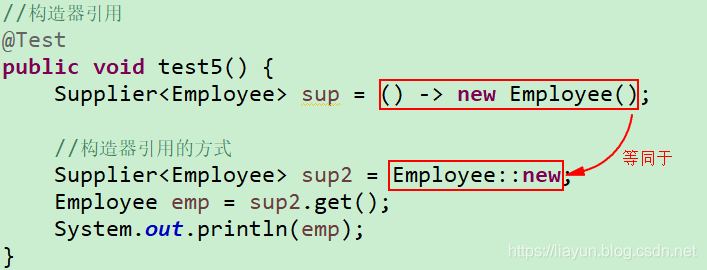
图 2‑8 第三种语法格式：类::实例方法名

### 构造器引用

构造器引用的语法格式是ClassName::new。它与函数式接口相结合，自动与函数式接口中方法兼容，可以把构造器引用赋值给定义的方法，但构造器的参数列表要与接口中抽象方法的参数列表保持一致！

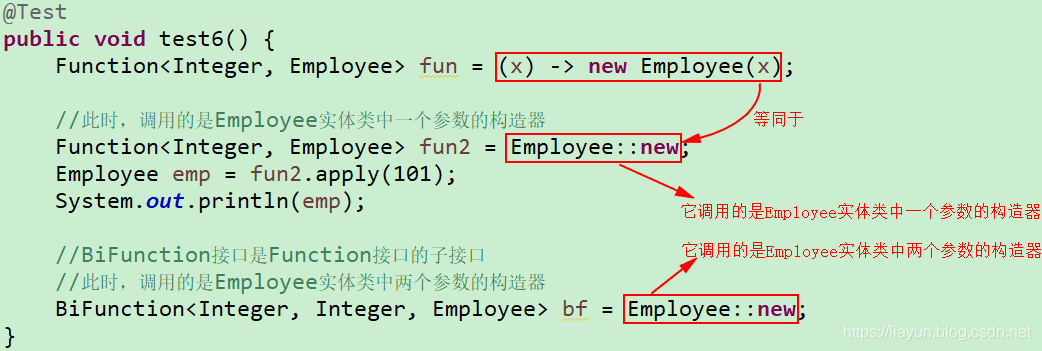
为了演示构造器引用，我们可以在Employee实体类中多写几个构造方法，如图2-9所示，然后，使用一下构造器引用的方式，如图2-10所示。

图 2‑9 Employee实体类

图 2‑10 构造器引用

不知你有没有想过，Employee实体类中有好几个构造器，那么现在调用的是哪个构造器呢？道理跟方法引用一样，由于构造器的参数列表要与接口中抽象方法的参数列表保持一致，所以此时调用的是无参的构造器。

现在，我要调用Employee实体类中一个参数的构造器，那该怎么办呢？如图2-11所示。

图 2‑11 带参构造器引用

#### 构造器引用实例

创建Employee实体类：

package Methods;  
  
public class Employee {  
 private int id;  
 private String name;  
 private int age;  
 private double salary;  
  
 public Employee() {  
 }  
  
 public Employee(int id) {  
 this.id = id;  
 }  
  
 public Employee(int id, int age) {  
 this.id = id;  
 this.age = age;  
 }  
  
 public Employee(int id, String name, int age, double salary) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.salary = salary;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 public double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(double salary) {  
 this.salary = salary;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", age=" + age +  
 ", salary=" + salary +  
 '}';  
 }  
}

##### 不带参的构造器引用

package Methods;  
  
import org.testng.annotations.Test;  
  
import java.util.function.Supplier;  
  
public class MethodsDemo {  
 //不带参的构造器引用  
 @Test  
 public void test(){  
 //不使用构造器引用  
 Supplier<Employee> sup=()->new Employee();  
 Employee emp1=sup.get();  
 System.*out*.println(emp1);  
 //使用构造器引用  
 Supplier<Employee> sup1=Employee::new;  
 Employee emp2=sup.get();  
 System.*out*.println(emp2);  
 }  
}

运行结果如图2-12所示。

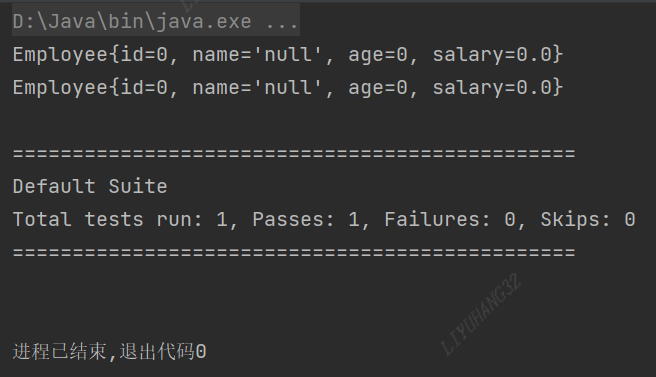
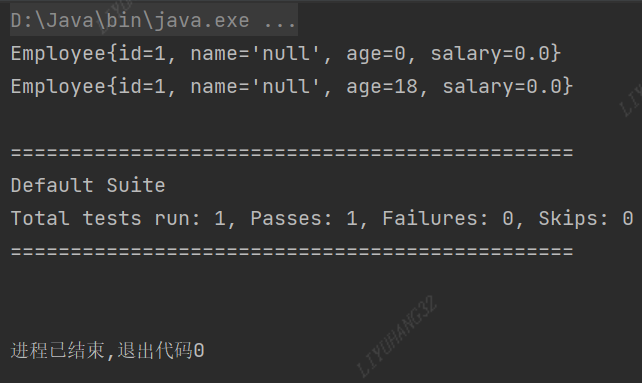


图 2‑12 运行结果

##### 带参的构造器引用

package Methods;  
  
import org.testng.annotations.Test;  
  
import java.util.function.BiFunction;  
import java.util.function.Function;  
  
public class MethodsDemo {  
 //带参的构造器引用  
 @Test  
 public void test() {  
 //不使用构造器引用  
 Function<Integer, Employee> fun = (x) -> new Employee(x);  
 Employee emp1 = fun.apply(1);  
 System.*out*.println(emp1);  
 //使用构造器引用  
 BiFunction<Integer, Integer, Employee> bf = Employee::new;  
 Employee emp2 = bf.apply(1, 18);  
 System.*out*.println(emp2);  
 }  
}

运行结果如图2-13所示。

图 2‑13 运行结果

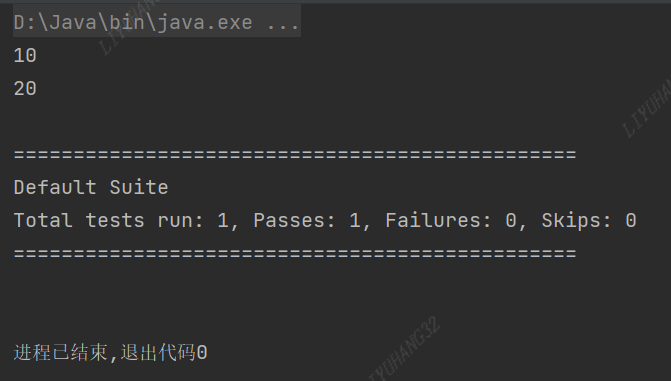
### 数组引用

数组引用的语法格式是Type[]::new。下面我就通过一个案例简单使用一下数组引用。

#### 数组引用实例

package Methods;  
  
import org.omg.CORBA.PUBLIC\_MEMBER;  
import org.testng.annotations.Test;  
  
import java.util.function.\*;  
  
public class MethodsDemo {  
 @Test  
 public void test() {  
 //不使用数组引用  
 Function<Integer, String[]> fun = (x) -> new String[x];  
 String[] strs1 = fun.apply(10);  
 System.*out*.println(strs1.length);  
 //使用数组引用  
 Function<Integer, String[]> fun2 = String[]::new;  
 String[] strs2 = fun.apply(20);  
 System.*out*.println(strs2.length);  
 }  
}

运行结果如图2-14所示。

图 2‑14 运行结果

## 函数式接口

### 什么是函数式接口

Lambda表达式可以很简洁的代替匿名内部类的代码编写，而匿名内部类往往是实现某一接口的一个抽象方法。所以，在使用Lambda表达式时，我们最应该关注的应该是接口的抽象方法，并且这个接口还必须只有一个抽象方法。我们称这种只有一个抽象方法的接口为"函数式接口"，而Lambda表达式恰好需要一个函数式接口的支持。

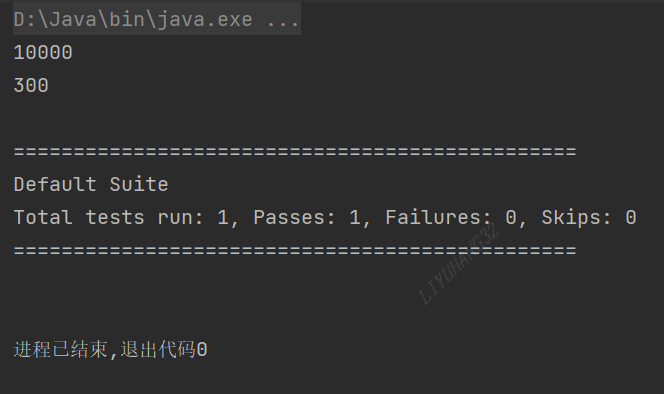
知道了函数式接口是什么之后，我们就可以通过Lambda表达式来创建这种接口的对象了（若Lambda表达式抛出一个受检异常，那么该异常需要在目标接口的抽象方法上进行声明）。我们还可以在任意函数式接口上使用@FunctionalInterface注解，这样做可以检查它是否是一个函数式接口，同时javadoc也会包含一条声明，说明这个接口是一个函数式接口。下面我就用一个例子来说明一下函数式接口的用法，即简单使用一下Lambda表达式来对一个数进行运算（不管啥运算）。

首先，创建一个函数式接口，例如MyFun.java：

package FunctionalInterface;  
@java.lang.FunctionalInterface  
public interface MyFun {  
 public Integer getValue(Integer num);  
}

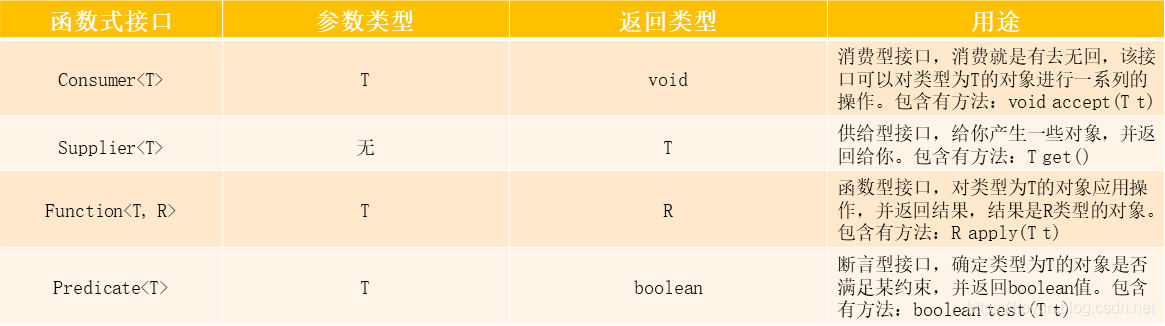
然后，编写一个对一个数进行运算的方法，运行结果如图3-1所示：

package FunctionalInterface;  
  
import org.testng.annotations.Test;  
  
public class FunctionalInterface {  
 public Integer operation(Integer num,MyFun mf){  
 return mf.getValue(num);  
 }  
 @Test  
 public void test(){  
 Integer num = operation(100,(x)->x\*x);  
 System.*out*.println(num);  
 System.*out*.println(operation(100,(y)->y+200));  
 }  
}

图 3‑1 运行结果

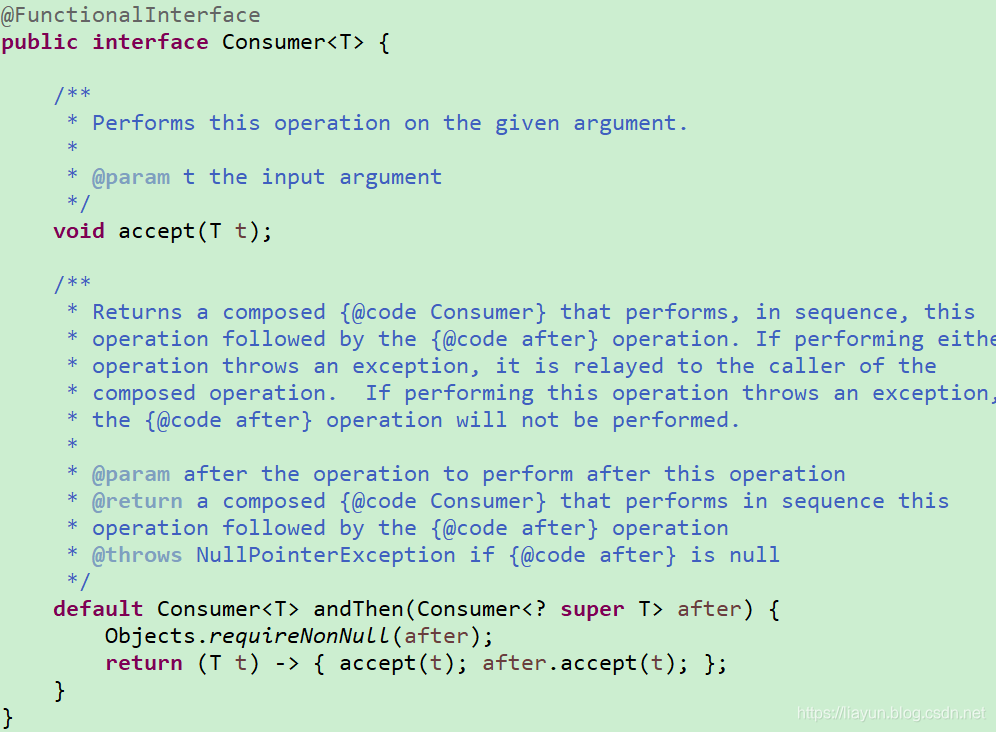
### Java 8中内置的四大核心函数式接口

Java 8中内置了一些函数式接口来供开发者们调用，这样就不需要开发者们重复定义函数式接口了。Java 8中提供了四大核心函数式接口，它们分别是消费型接口、供给型接口、函数型接口、断言型接口，如图3-2所示。

图 3‑2 Java 8中内置的四大核心函数式接口

#### 消费型接口

Java 8中消费型接口的定义如图3-3所示。

图 3‑3 消费型接口

#### 供给型接口

Java 8中供给型接口的定义如图3-4所示。

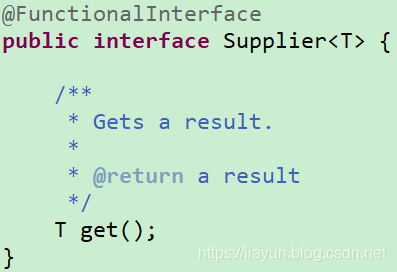


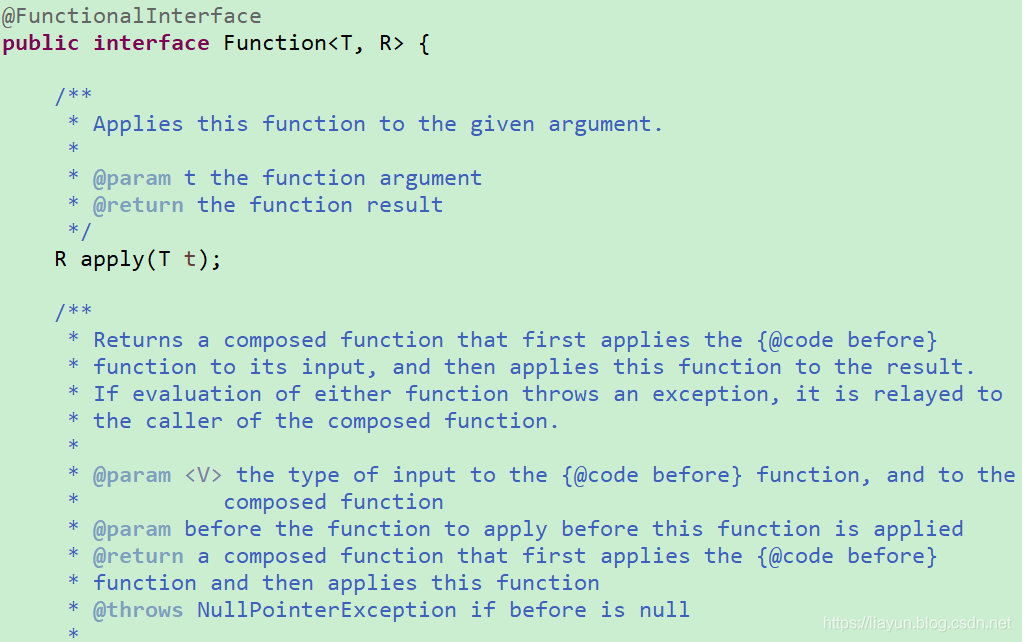
图 3‑4 供给型接口

供给型接口主要就是给你产生一些对象，并返回给你。例如，产生一些指定个数的整数，并放入集合中。

@Test  
public void test() {  
 List<Integer> numList = getNumList(10, () -> (int) (Math.*random*() \* 100));  
 for (Integer num : numList) {  
 System.*out*.println(num);  
 }  
}  
  
//需求：产生一些指定个数的整数，并放入集合中  
public List<Integer> getNumList(int num, Supplier<Integer> sup) {  
 List<Integer> list = new ArrayList<Integer>();  
 for (int i = 0; i < num; i++) {  
 Integer n = sup.get();  
 list.add(n);  
 }  
 return list;  
}

#### 函数型接口

Java8中函数型接口的定义如图3-5所示。

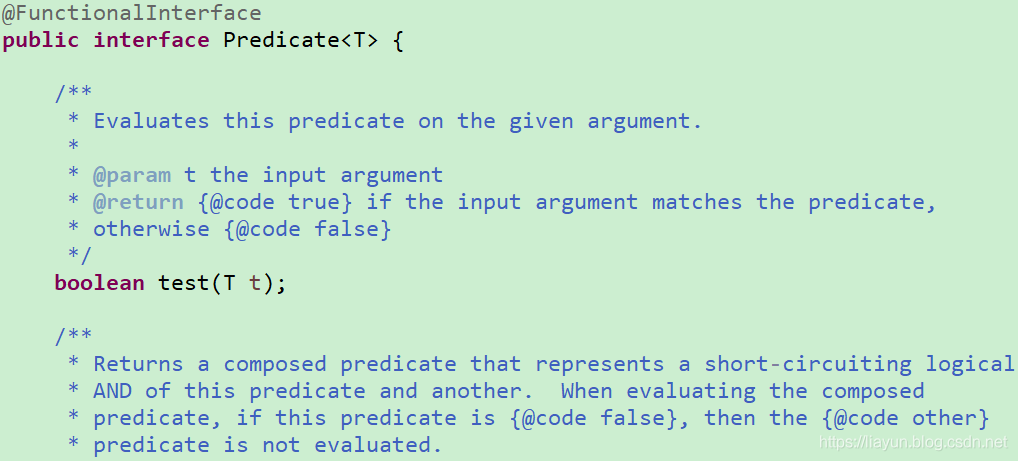
图 3‑5 函数型接口

函数型接口可以用于处理字符串。

@Test  
 public void test6() {  
 String newStr = strHandler("\t\t\t 我是美美侠啊 ", (str) -> str.trim());  
 System.*out*.println(newStr);  
  
 String subStr = strHandler("我是美美侠啊", (str) -> str.substring(2, 5));  
 System.*out*.println(subStr);  
 }  
  
 //Function<T, R>：函数型接口  
//需求：专门用于处理字符串  
 public String strHandler(String str, Function<String, String> fun) {  
 return fun.apply(str);  
 }

#### 断言型接口

Java 8中断言型接口的定义如图3-6所示。

图 3‑6 断言型接口

断言型接口可以用于做一些判断操作，例如处理字符串时，将满足指定条件的字符串放入集合中去。

@Test  
 public void test7() {  
 List<String> list = Arrays.*asList*("Hello", "meimeixia", "Lambda", "www", "ok");  
 List<String> strList = filterStr(list, (s) -> s.length() > 3);  
 for (String str : strList) {  
 System.*out*.println(str);  
 }  
 }  
  
 //Predicate<T>：断言型接口，用于做一些判断操作  
//需求：还是处理字符串，将满足指定条件的字符串放入集合中去  
 public List<String> filterStr(List<String> list, Predicate<String> pre) {  
 List<String> strList = new ArrayList<String>();  
 for (String str : list) {  
 if (pre.test(str)) {  
 strList.add(str);  
 }  
 }  
 return strList;  
 }

#### 其他接口

除了Java 8中内置的四大核心函数式接口，还有一些其他的接口，如图3-7所示。

图 3‑7 其他接口

### 函数式接口实例

声明一个带两个泛型的函数式接口，泛型类型为<T，R>，其中T为参数，R为返回值，在接口中声明对应抽象方法。在TestLambda类中声明一个方法，使用接口作为参数，计算两个long型参数的和，再计算两个long型参数的乘积。

首先，要创建函数式接口:

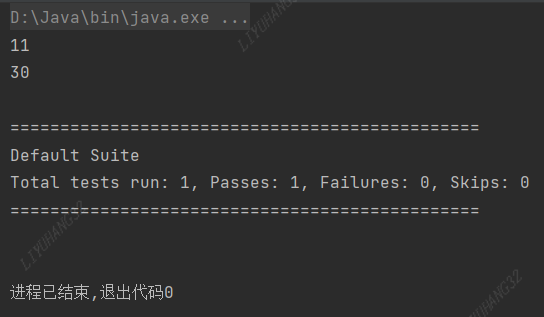
package FunctionalInterface;  
@FunctionalInterface  
public interface MF<T,R> {  
 public R getVAlue(T t1,T t2);  
}

然后，在TestLambda类中编写一个对两个Long型数据进行处理的方法，接着再对该方法进行测试。

public Long mf(Long l1,Long l2,MF<Long,Long> mf){  
 return mf.getVAlue(l1,l2);  
}

@Test  
public void test(){  
 System.*out*.println(mf(5L,6L,(x,y)->(x+y)));  
 System.*out*.println(mf(5L,6L,(x,y)->(x\*y)));  
}

运行结果如图3-8所示。

图 3‑8 运行结果

## Stream API

### 什么是Stream？

Java 8中有两大最为重要的改变。第一个是Lambda表达式；另外一个则是Stream API（位于包java.util.stream.\*下）。Stream是Java 8中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。使用Stream API对集合数据进行操作，就类似于使用SQL执行的数据库查询。也可以使用Stream API来并行执行操作。简而言之，Stream API提供了一种高效且易于使用的处理数据的方式。

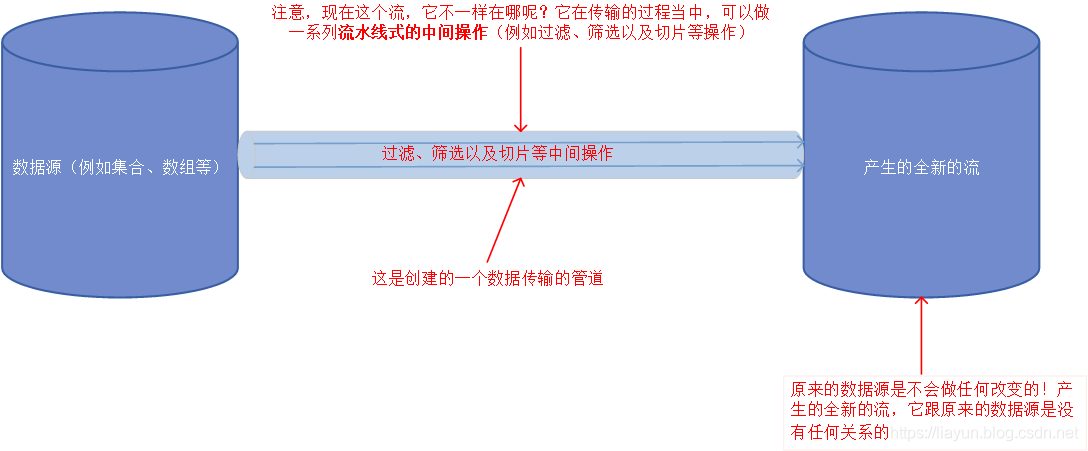
那Stream（流）到底是什么呢？Stream（流）是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。“集合讲的是数据，流讲的是计算！” 我们尤其注意以下几点：

Stream自己不会储存元素；

Stream不会改变源对象，相反，他们会返回一个持有结果的新Sream；

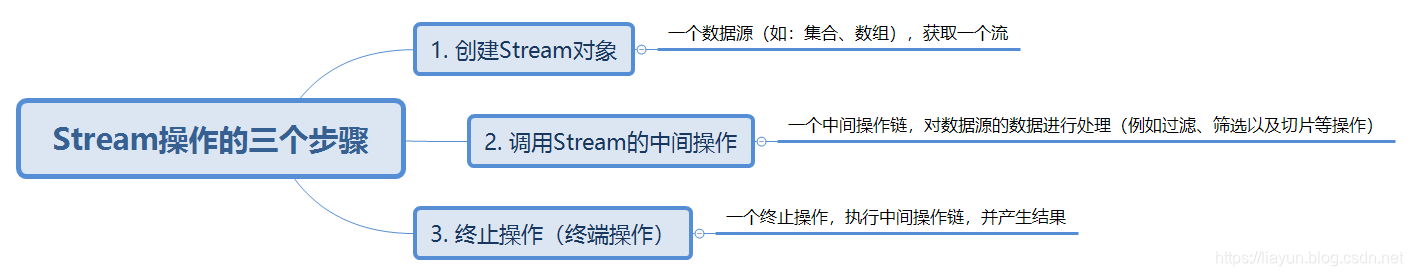
Stream操作时延迟执行的，这意味着它们会等到需要结果的时候才执行。

如果要用一个图来形象地描述Stream（流），那么可能是下面这样的，如图4-1所示。

图 4‑1 Stream

### Stream使用的三个步骤

Stream的操作一般可以分为以下三个步骤：创建Stream对象、调用Stream的中间操作、终止操作（终端操作）。如图4-2所示。

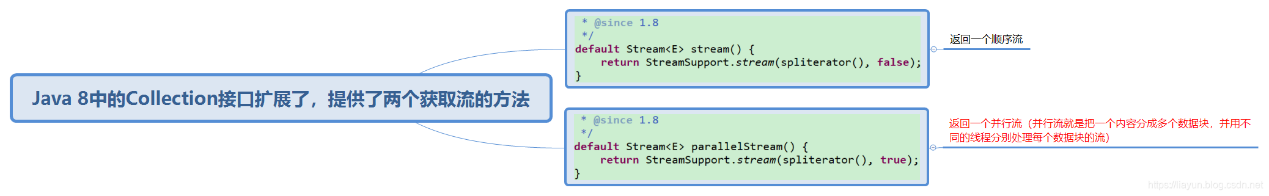
图 4‑2 Stream使用的三个步骤

### 如何创建Stream？

创建Stream（流）一共有四种方式.

#### 通过Collection接口创建

通过Collection系列集合提供的stream()方法或者paralleStream()方法获取流对象，如图4-3所示。

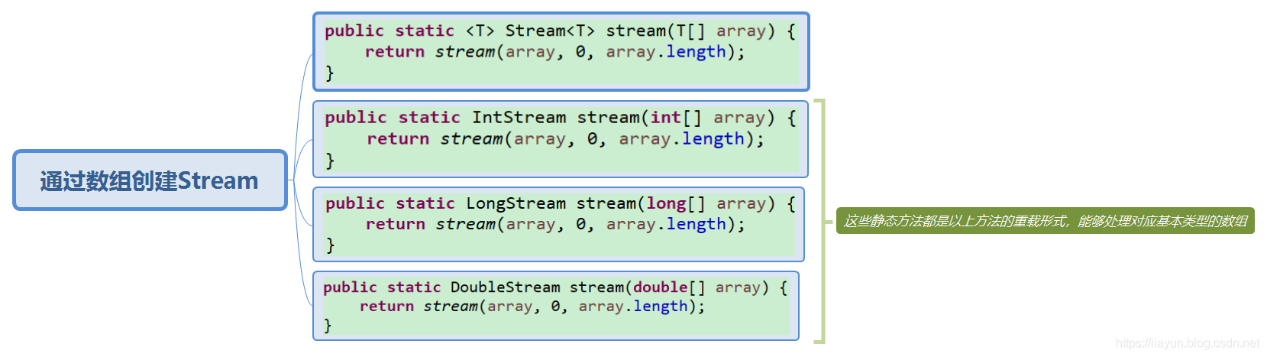
图 4‑3 通过Collection接口创建

例如，以下代码就是通过这种方式来创建流的：

package StreamAPI;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.stream.Stream;  
  
public class StreamAPIDemo {  
 //1. 可以通过Collection系列集合提供的stream()方法（获取的是串行流）或者paralleStream()方法（获取的是并行流）获取流对象  
 List<String> list = new ArrayList<String>();  
 //流也可以带泛型  
 Stream<String> stream1 = list.stream();  
}

#### 通过数组创建

Java 8中的Arrays类的stream()静态方法可以获取数组流，而且它有许多重载的方法可以返回不同基本数据类型数组的Steam对象，如图4-4所示。

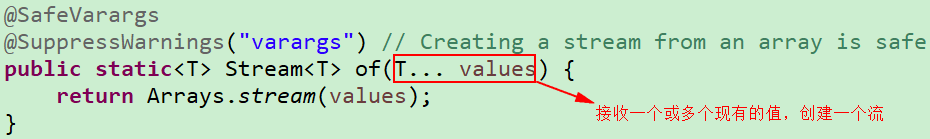
图 4‑4 通过数组创建

例如，以下代码就是通过这种方式来获取数组流的：

package StreamAPI;  
  
import java.util.Arrays;   
import java.util.stream.Stream;  
  
public class StreamAPIDemo {  
 //2. 通过Arrays类中的静态方法stream()获取数组流  
 Employee[] emps = new Employee[10];  
 Stream<Employee> stream2 = Arrays.*stream*(emps);  
}

#### 通过Stream类中的静态方法of()创建

可以使用Stream类中的静态方法of()，通过显示值来创建一个流，该方法可以接收任意数量的参数，如图4-5所示。

图 4‑5 通过Stream类中的静态方法of()创建

例如，以下代码就是通过这种方式来获取流的：

package StreamAPI;  
import java.util.stream.Stream;  
  
public class StreamAPIDemo {  
 //3. 通过Stream类中的静态方法of()获取流  
 Stream<String> stream3 = Stream.*of*("aa", "bb", "cc");  
}

#### 通过静态方法Stream.iterate()或者Stream.generate()创建无限流

可以使用静态方法Stream.iterate()或者Stream.generate()来创建无限流。创建无限流也有两种方式，一种是通过迭代来创建，即使用Stream类中的静态方法iterate()来创建，另一种是通过生成来创建，即使用Stream类中的静态方法generate()来创建。

##### 迭代

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.stream.Stream;  
  
public class StreamAPIDemo {  
 @Test  
 public void test() {  
 Stream<Integer> stream = Stream.*iterate*(0, x -> (x + 2));  
 stream.forEach(System.*out*::println);  
 }  
}

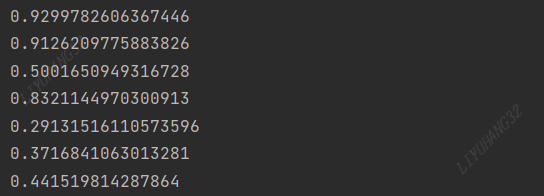
创建出来的无限流如图4-6所示。

图 4‑6 无限流（迭代）

##### 生成

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.stream.Stream;  
  
public class StreamAPIDemo {  
 @Test  
 public void test(){  
 Stream.*generate*(()->Math.*random*()).forEach(System.*out*::println);  
 }  
}

创建出来的无限流如图4-7所示。

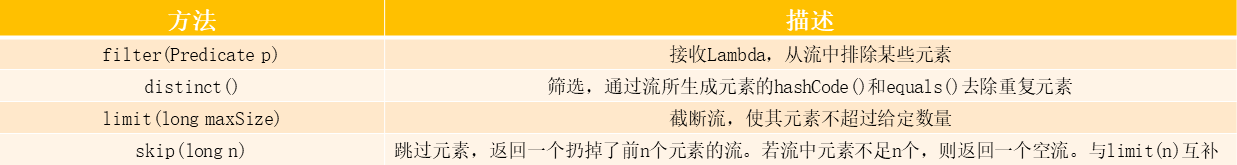
图 4‑7 无限流（生成）

### 中间操作

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的处理！而是在终止操作时一次性全部处理，这称为"惰性求值"。

#### 筛选与切片

筛选与切片的方法和描述如图4-8所示。

图 4‑8 筛选与切片

下面通过Employee实体类的一些实例进行展示。

Employee实体类的内容如下：

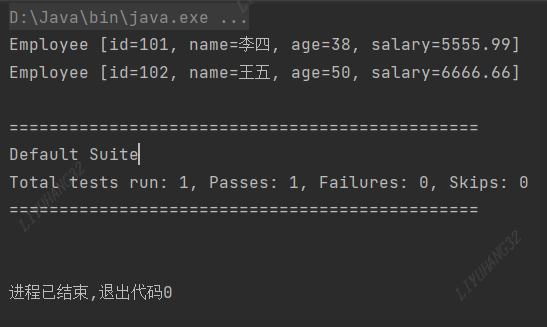
package StreamAPI;  
  
public class Employee {  
  
 private Integer id;  
 private String name;  
 private Integer age;  
 private Double salary;  
  
 public Employee() {  
 super();  
 }  
  
 public Employee(Integer id) {  
 super();  
 this.id = id;  
 }  
  
 public Employee(Integer id, Integer age) {  
 super();  
 this.id = id;  
 this.age = age;  
 }  
  
 public Employee(Integer id, String name, Integer age, Double salary) {  
 super();  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.salary = salary;  
 }  
  
 public Integer getId() {  
 return id;  
 }  
  
 public void setId(Integer id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Integer getAge() {  
 return age;  
 }  
  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
  
 public Double getSalary() {  
 return salary;  
 }  
  
 public void setSalary(Double salary) {  
 this.salary = salary;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee [id=" + id + ", name=" + name + ", age=" + age + ", salary=" + salary + "]";  
 }  
}

##### filter(Predicate p)

获取公司中年龄大于35的雇员的实现类的内容如下：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Stream;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99),  
 new Employee(101, "李四", 38, 5555.99),  
 new Employee(102, "王五", 50, 6666.66),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(104, "田七", 8, 7777.77)  
 );  
  
 //演示filter(Predicate p)  
 @Test  
 public void test() {  
 Stream<Employee> steam = employees.stream()  
 .filter((e) -> {  
// System.out.println("Stream API的中间操作");  
 return e.getAge() > 35;  
 });  
 /\*  
 \* 我为了让大家看结果，我就得来一个终止操作，不然的话，以上中间操作是没有任何结果的！  
 \* 因为中间操作没有执行，所以是没有任何结果的（中间操作它是不会执行任何操作的）。  
 \* 只有当你执行终止操作以后，所有的中间操作才一次性全部处理（一次性执行全部内容，这个过程我们就称为"惰性求值"，或者称为"延迟加载"）  
 \*/  
 steam.forEach(System.*out*::println);  
 }  
}

运行结果如图4-9所示。

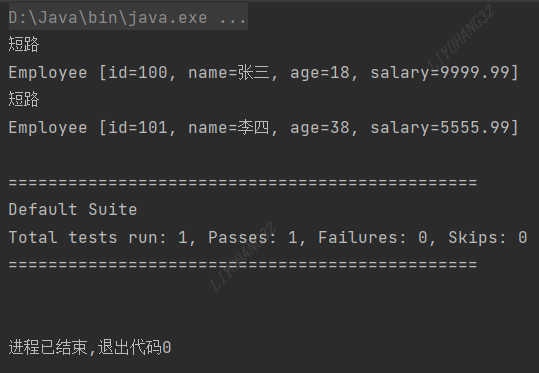
图 4‑9 filter(Predicate p)

##### limit(long maxSize)

获取公司中两个薪水大于5000元的雇员的实现类的内容如下：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99),  
 new Employee(101, "李四", 38, 5555.99),  
 new Employee(102, "王五", 50, 6666.66),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(104, "田七", 8, 7777.77)  
 );  
  
 @Test  
 public void test() {  
 employees.stream()  
 .filter(e -> {  
 System.*out*.println("短路");  
 return e.getSalary() > 5000;  
 })  
 .limit(2)  
 .forEach(System.*out*::println);  
 }  
}

运行结果如图4-19所示。

图 4‑10 limit(long maxSize)

从上图可以看出，迭代操作只执行了两次，也就是说它只要找到满足条件的数据以后，之后的迭代操作就不再继续执行了，这个过程就叫做短路！在某种情况下，它可以用于提高效率。实际上，它跟我们之前学的&&和||有点类似。

##### skip(long n)

获取公司中薪水大于5000元的雇员（跳过两个）的实现类的内容如下：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99),  
 new Employee(101, "李四", 38, 5555.99),  
 new Employee(102, "王五", 50, 6666.66),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(104, "田七", 8, 7777.77)  
 );  
  
 @Test  
 public void test() {  
 employees.stream()  
 .filter(e -> (e.getSalary() > 5000))  
 .skip(2)  
 .forEach(System.*out*::println);  
 }  
}

运行结果如图4-11所示。

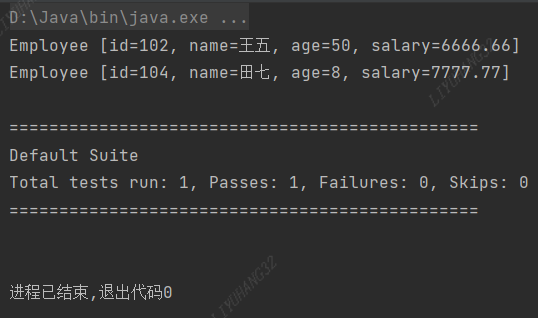


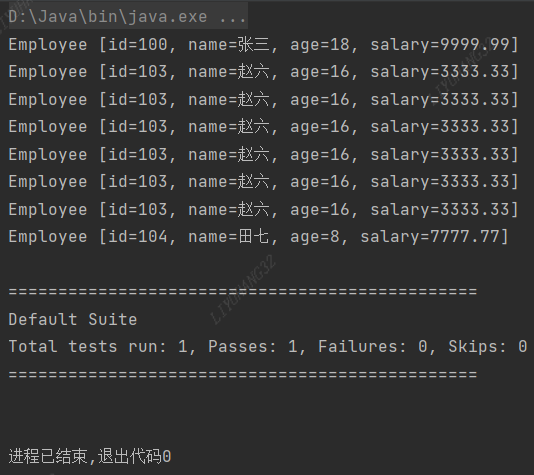
图 4‑11 skip(long n)

##### distinct()

为了更好地配合该案例的演示，首先得在StreamAPIDemo类的employees集合中添加几个重复的元素，获取公司中年龄小于35的雇员的实现类的内容如下：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99),  
 new Employee(101, "李四", 38, 5555.99),  
 new Employee(102, "王五", 50, 6666.66),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(104, "田七", 8, 7777.77)  
 );  
 @Test  
 public void test(){  
 employees.stream()  
 .filter(e->(e.getAge())<35)  
 .distinct()  
 .forEach(System.*out*::println);  
 }  
}

运行结果如图4-12所示。

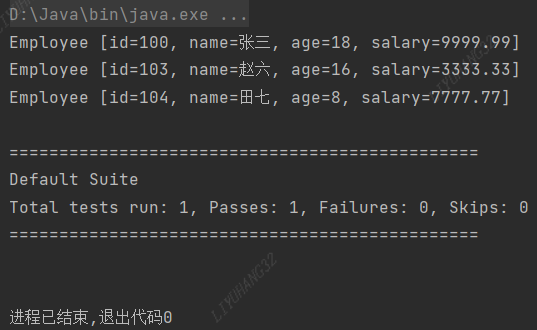
图 4‑12 错误的distinct()

并没有去除重复元素，为什么没去掉呢？因为它是通过流所生成元素的hashCode()和equals()来去除重复元素。

所以，要想去除重复元素，必须得重写Employee实体类中的hashCode()和equals()这两个方法。

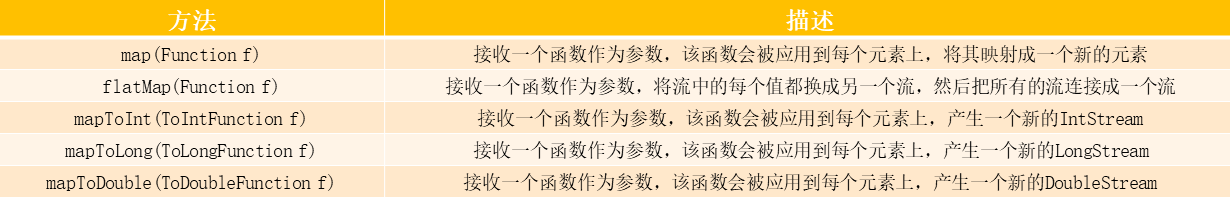
@Override  
public boolean equals(Object o) {  
 if (this == o) return true;  
 if (!(o instanceof Employee)) return false;  
 Employee employee = (Employee) o;  
 return getId().equals(employee.getId()) && getName().equals(employee.getName()) && getAge().equals(employee.getAge()) && getSalary().equals(employee.getSalary()) && getStatus() == employee.getStatus();  
}  
  
@Override  
public int hashCode() {  
 return Objects.hash(getId(), getName(), getAge(), getSalary(), getStatus());  
}

重写完Employee实体类中的hashCode()和equals()这两个方法后，再次运行以上test()方法，运行结果如图4-13所示。

图 4‑13 正确的distinct()

#### 映射

映射的方法和描述如图4-14所示。

图 4‑14 映射

##### map(Function f)

将一个集合中的所有元素（英文字母）都给转成大写,运行结果如图4-15所示：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 @Test  
 public void test(){  
 List<String> list= Arrays.*asList*("aaa","bbb","ccc","ddd","eee");  
 list.stream()  
 .map(str->str.toUpperCase())  
 .forEach(System.*out*::println);  
 }  
}

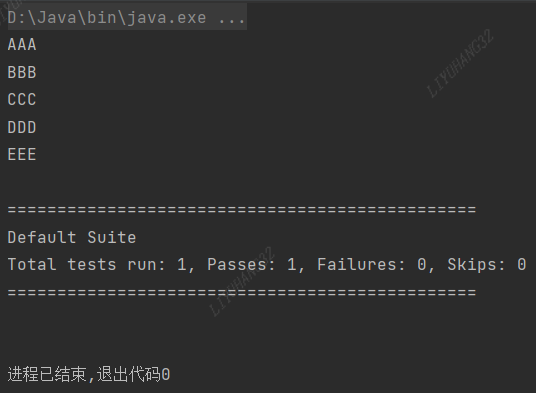


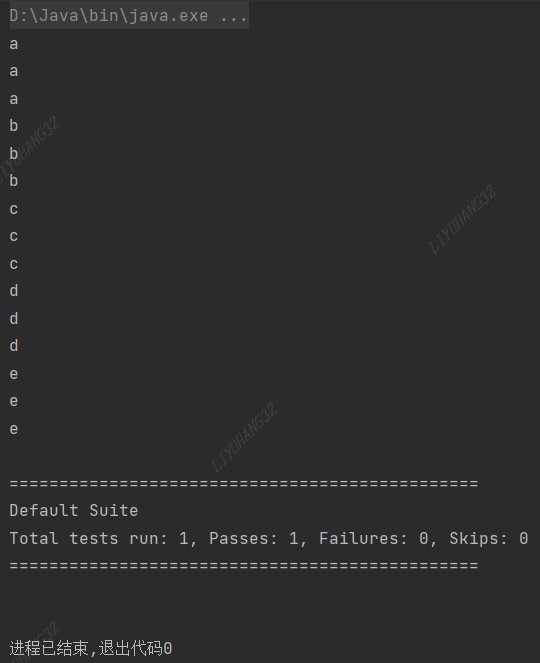
图 4‑15 运行结果

从中我们可以得出一个结论：依次从数据源中提取出来的每个元素都将应用到map方法中的(str) -> str.toUpperCase()这个函数上，然后就产生一个新流，得到一个结果。

##### flatMap(Function f)

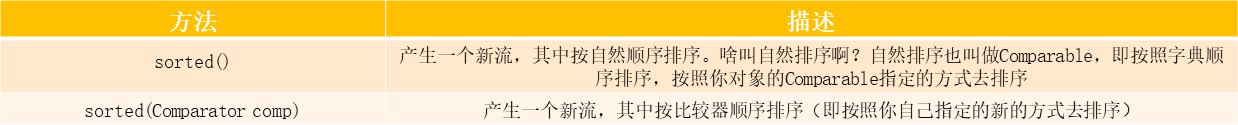
在讲解flatMap(Function f)之前，我们可以在StreamAPIDemo类中先编写一个方法，用于解析字符串，并把字符串中的一个一个的字符给单独提取出来，放到集合中，运行结果如图4-16所示

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.stream.Stream;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 //在讲解flatMap(Function f)之前，我们可以在类中先编写一个方法，用于解析字符串，并把字符串中的一个一个的字符给单独提取出来，放到集合中  
 public static Stream<Character> filterCharacter(String str) {  
 List<Character> list = new ArrayList<Character>();  
 for (Character ch : str.toCharArray()) {  
 list.add(ch);  
 }  
 return list.stream();  
 }  
 @Test  
 public void test(){  
 List<String> list= Arrays.*asList*("aaa","bbb","ccc","ddd","eee");  
 Stream<Character> sm=list.stream()  
 .flatMap(StreamAPIDemo::*filterCharacter*);  
 sm.forEach(System.*out*::println);  
 }  
}

图 4‑16 运行结果

#### 排序

排序的方法和描述如图4-17所示。

图 4‑17 排序

##### sorted()

将一个集合中的所有元素（英文字母）按照自然顺序排序运行结果如图4-18所示：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 //演示sorted()，自然排序  
 @Test  
 public void test() {  
 List<String> list = Arrays.*asList*("aaa", "bbb", "ccc", "ddd", "eee");  
 list.stream()  
 .sorted()  
 .forEach(System.*out*::println);  
 }  
  
}

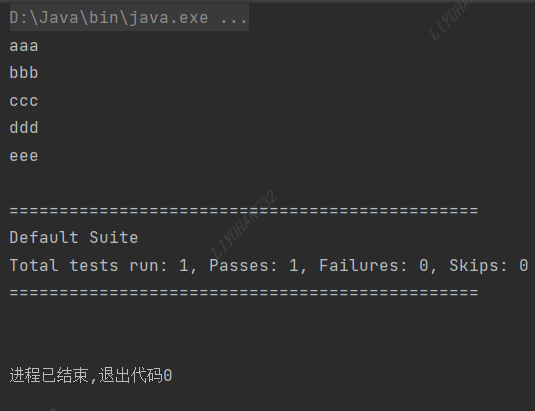


图 4‑18 运行结果

##### sorted(Comparator comp)

通过定制排序比较两个Employee（先按年龄比，年龄相同再按姓名比）运行结果如图4-19所示：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99),  
 new Employee(101, "李四", 38, 5555.99),  
 new Employee(102, "王五", 50, 6666.66),  
 new Employee(103, "赵六", 16, 3333.33),  
/\*  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
 new Employee(103, "赵六", 16, 3333.33),  
\*/  
 new Employee(104, "田七", 8, 7777.77)  
 );  
 //演示sorted()，自然排序  
 @Test  
 public void test8() {  
 employees.stream()  
 .sorted((e1,e2)->{  
 if(e1.getAge()==e2.getAge())  
 return e1.getName().compareTo(e2.getName());  
 else  
 return e1.getAge().compareTo(e2.getAge());  
 }).forEach(System.*out*::println);  
 }  
}

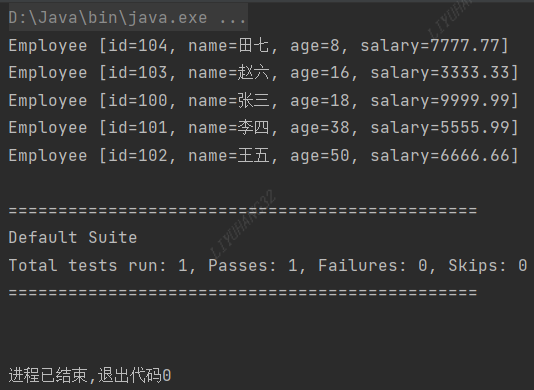


图 4‑19 运行结果

### 终止操作

终止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。

#### 查找与匹配

查找与匹配的方法和描述如图4-20所示。

图 4‑20 查找与匹配

在Employee实体类中添加private Status status；

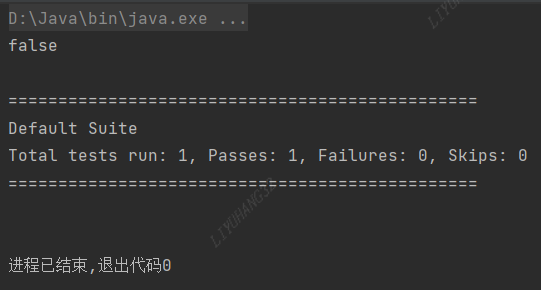
创建枚举类型Status;

package StreamAPI;  
  
import java.util.stream.Stream;  
  
public enum Status {  
 *FREE*("空闲"),*BUSY*("忙碌"),*VOCATION*("休假"),*LEFT*("离职");  
 private String desc;  
  
 Status(String desc) {  
 this.desc = desc;  
 }  
  
 @Override  
 public String toString() {  
 return "Status{" +  
 "desc='" + desc + '\'' +  
 '}';  
 }  
}

##### allMatch(Predicate p)

检查公司中所有员工是否处于忙碌状态，运行结果如图4-21所示：

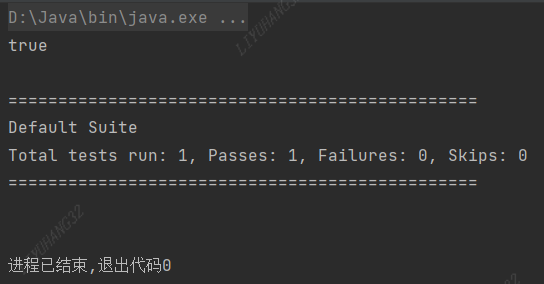
package StreamAPI;  
  
import org.testng.annotations.Test;  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99, Status.*FREE*),  
 new Employee(101, "李四", 38, 5555.99, Status.*BUSY*),  
 new Employee(102, "王五", 50, 6666.66, Status.*VOCATION*),  
 new Employee(103, "赵六", 16, 3333.33, Status.*FREE*),  
 new Employee(104, "田七", 8, 7777.77, Status.*BUSY*)  
 );  
 @Test  
 public void test(){  
 boolean b1 = employees.stream()  
 .allMatch(e->e.getStatus().equals(Status.*FREE*));  
 System.*out*.println(b1);  
 }  
}

图 4‑21 allMatch(Predicate p)

##### anyMatch(Predicate p)

检查公司中是否有员工处于忙碌状态,运行结果如图4-22所示：

package StreamAPI;  
  
import org.testng.annotations.Test;  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99, Status.*FREE*),  
 new Employee(101, "李四", 38, 5555.99, Status.*BUSY*),  
 new Employee(102, "王五", 50, 6666.66, Status.*VOCATION*),  
 new Employee(103, "赵六", 16, 3333.33, Status.*FREE*),  
 new Employee(104, "田七", 8, 7777.77, Status.*BUSY*)  
 );  
 @Test  
 public void test(){  
 boolean b1 = employees.stream()  
 .anyMatch(e->e.getStatus().equals(Status.*FREE*));  
 System.*out*.println(b1);  
 }  
   
}

图 4‑22 运行结果

##### noneMatch(Predicate p)

检查公司中是否所有员工都没有处于忙碌状态，运行结果如图4-23所示：

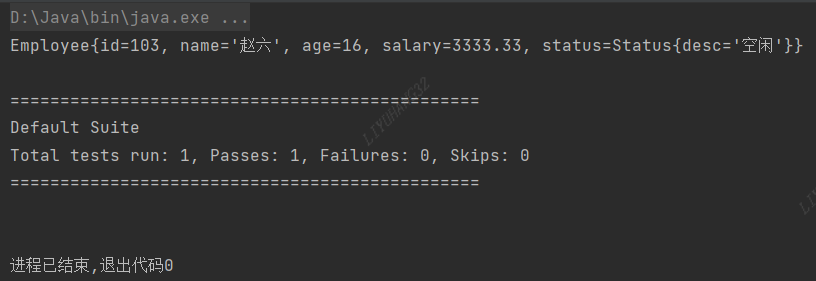
package StreamAPI;  
  
import org.testng.annotations.Test;  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99, Status.*FREE*),  
 new Employee(101, "李四", 38, 5555.99, Status.*BUSY*),  
 new Employee(102, "王五", 50, 6666.66, Status.*VOCATION*),  
 new Employee(103, "赵六", 16, 3333.33, Status.*FREE*),  
 new Employee(104, "田七", 8, 7777.77, Status.*BUSY*)  
 );  
 @Test  
 public void test(){  
 boolean b1 = employees.stream()  
 .noneMatch(e->e.getStatus().equals(Status.*FREE*));  
 System.*out*.println(b1);  
 }  
}

图 4‑23 运行结果

##### findFirst()

获取公司中薪资最低的员工信息，运行结果如图4-24所示：

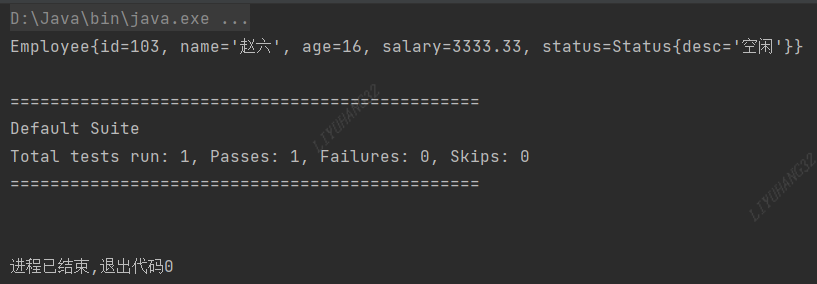
package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Optional;  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99, Status.*FREE*),  
 new Employee(101, "李四", 38, 5555.99, Status.*BUSY*),  
 new Employee(102, "王五", 50, 6666.66, Status.*VOCATION*),  
 new Employee(103, "赵六", 16, 3333.33, Status.*FREE*),  
 new Employee(104, "田七", 8, 7777.77, Status.*BUSY*)  
 );  
 @Test  
 public void test(){  
 Optional<Employee> op=employees.stream()  
 .sorted((e1,e2)-> Double.*compare*(e1.getSalary(), e2.getSalary()))  
 .findFirst();  
 System.*out*.println(op.get());  
 }  
}

图 4‑24运行结果

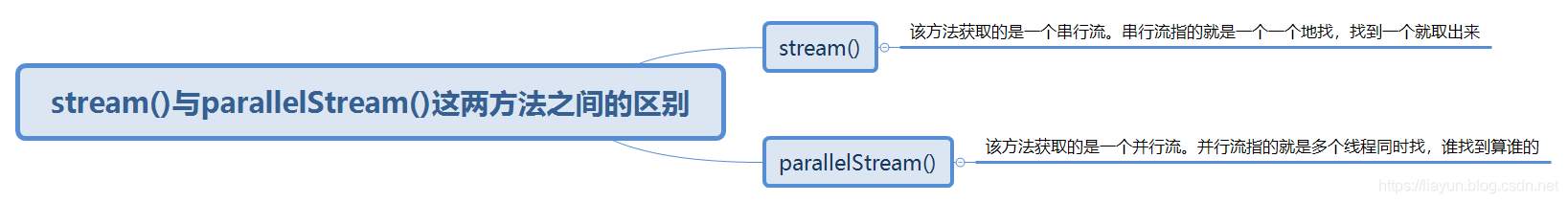
##### findAny()

获取公司中处于空闲状态的任意一个员工的信息，运行结果如图4-25所示：

package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Optional;  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99, Status.*FREE*),  
 new Employee(101, "李四", 38, 5555.99, Status.*BUSY*),  
 new Employee(102, "王五", 50, 6666.66, Status.*VOCATION*),  
 new Employee(103, "赵六", 16, 3333.33, Status.*FREE*),  
 new Employee(104, "田七", 8, 7777.77, Status.*BUSY*)  
 );  
 @Test  
 public void test(){  
 Optional<Employee> op=employees.parallelStream()  
 .filter(e->e.getStatus().equals(Status.*FREE*))  
 .findAny();  
 System.*out*.println(op.get());  
 }  
}

图 4‑25 运行结果

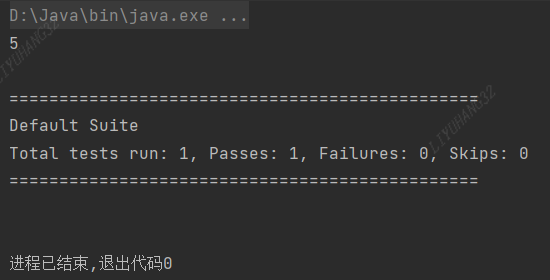
这里，一定要注意stream()与parallelStream()这两方法之间的区别。下一讲我就会介绍到串行流与并行流，这里稍微了解一下即可，如图4-26所示。

图 4‑26 串行流与并行流

##### count()

获取公司员工的总数，运行结果如图4-27所示：

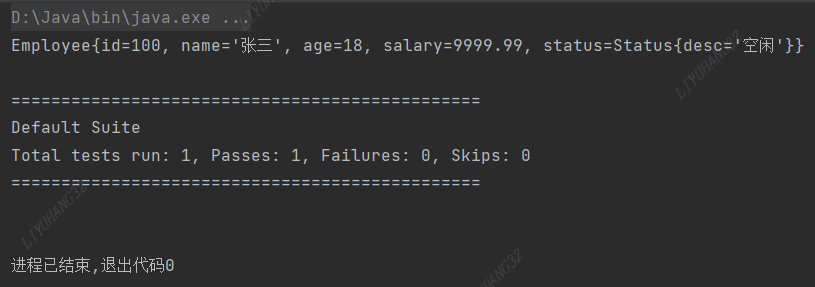
package StreamAPI;  
  
import org.testng.annotations.Test;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class StreamAPIDemo {  
 List<Employee> employees = Arrays.*asList*(  
 new Employee(100, "张三", 18, 9999.99, Status.*FREE*),  
 new Employee(101, "李四", 38, 5555.99, Status.*BUSY*),  
 new Employee(102, "王五", 50, 6666.66, Status.*VOCATION*),  
 new Employee(103, "赵六", 16, 3333.33, Status.*FREE*),  
 new Employee(104, "田七", 8, 7777.77, Status.*BUSY*)  
 );  
 @Test  
 public void test(){  
 long count= employees.stream()  
 .count();  
 System.*out*.println(count);  
 }  
}

图 4‑27 运行结果

##### max(Comparator c)

获取公司中薪资最高的员工信息，运行结果如图4-28所示：

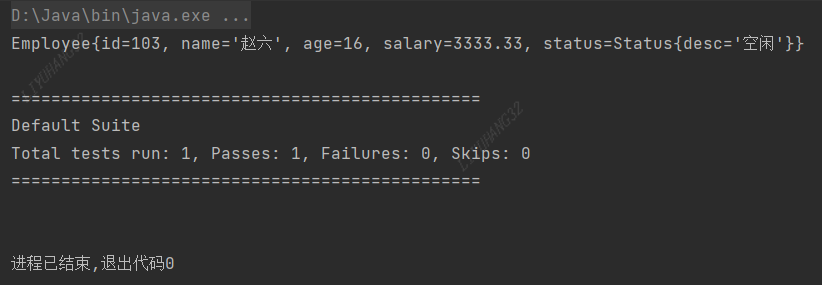
@Test  
public void test(){  
 Optional<Employee> op = employees.stream()  
 .max((e1,e2)-> Double.*compare*(e1.getSalary(),e2.getSalary()));  
 System.*out*.println(op.get());  
}

图 4‑28 运行结果

##### min(Comparator c)

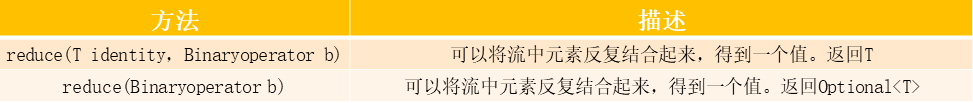
获取公司中的最小的工资是多少，运行结果如图4-29所示

@Test  
public void test(){  
 Optional<Employee> op = employees.stream()  
 .min((e1,e2)->Double.*compare*(e1.getSalary(),e2.getSalary()));  
 System.*out*.println(op.get());  
}

图 4‑29 运行结果

#### 归约

归约的方法和描述如图4-30所示。

图 4‑30 归约

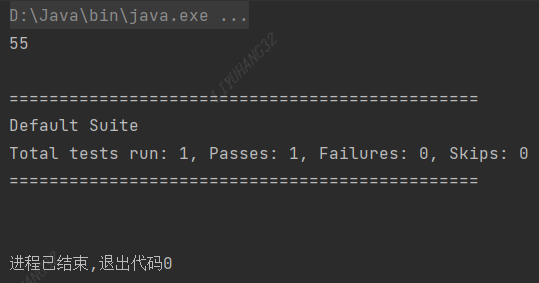
##### reduce(T identity，Binaryoperator b)

第一个参数：identity（起始值），例如0；

第二个参数：（Binaryoperator） b，二元运算。

计算1到10之间的和，运行结果如图4-31所示：

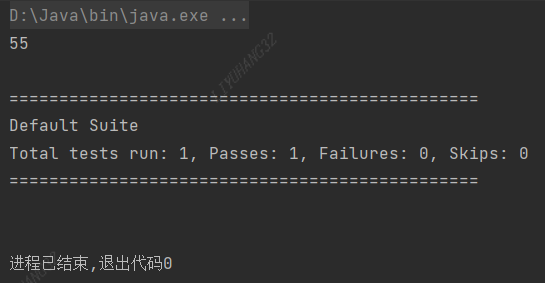
@Test  
public void test(){  
 List<Integer> list= Arrays.*asList*(1,2,3,4,5,6,7,8,9,10);  
 Integer sum=list.stream()  
 .reduce(0,(x,y)->(x+y));  
 System.*out*.println(sum);  
}

图 4‑31 运行结果

##### reduce(Binaryoperator b)

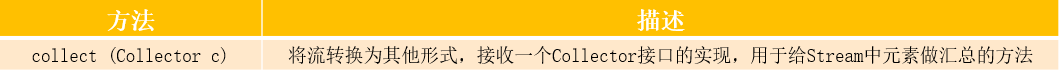
计算1到10之间的和，运行结果如图4-32所示：

@Test  
public void test(){  
 List<Integer> list= Arrays.*asList*(1,2,3,4,5,6,7,8,9,10);  
 Optional<Integer> op= list.stream()  
 .reduce(Integer::*sum*);  
 System.*out*.println(op.get());  
}

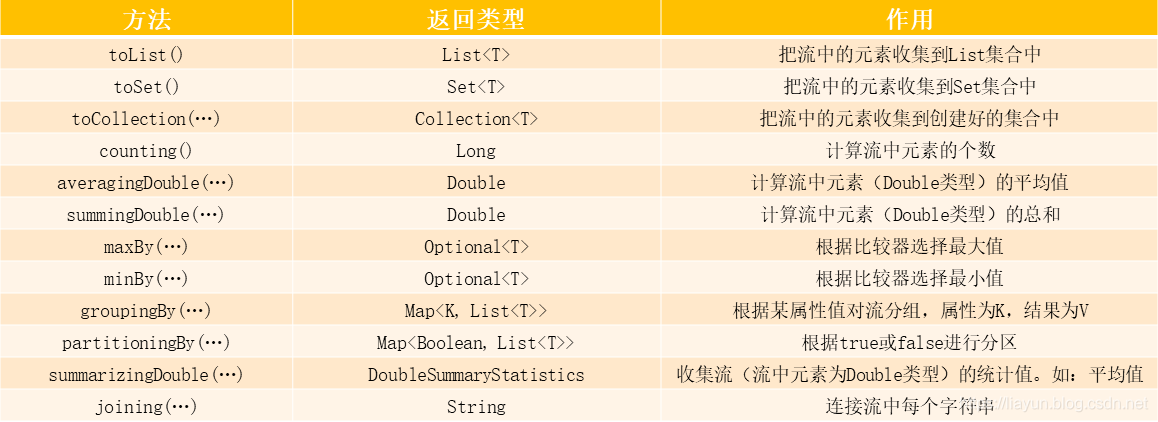
图 4‑32 运行结果

#### 收集

收集的方法和描述如图4-33所示。

图 4‑33 收集

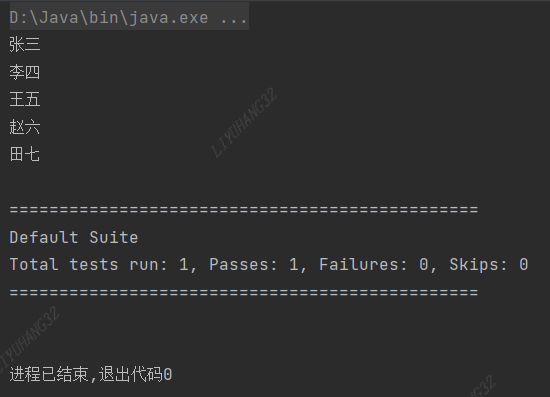
Collector接口中方法的实现决定了如何对流执行收集操作（如收集到List、Set以及Map中），但是Collectors实现类中提供了许多静态方法，可以方便的创建常见的收集器实例。Collectors实现类中常用的静态方法如图4-34所示。

图 4‑34 Collectors实现类中常用的静态方法

##### toList()

把当前公司中所有员工的名字给提取出来，并把这些名字放到一个List集合中，运行结果如图4-35所示：

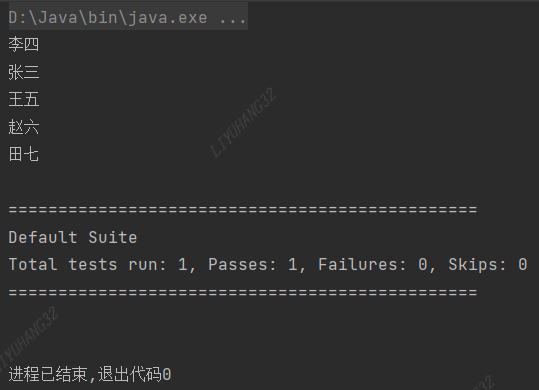
@Test  
public void test(){  
 List<String> list=employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*toList*());  
 list.forEach(System.*out*::println);  
}

图 4‑35 运行结果

##### toSet()

我们想把当前公司中所有员工的名字给提取出来，并把这些名字放到一个Set集合中，运行结果如图4-36所示：

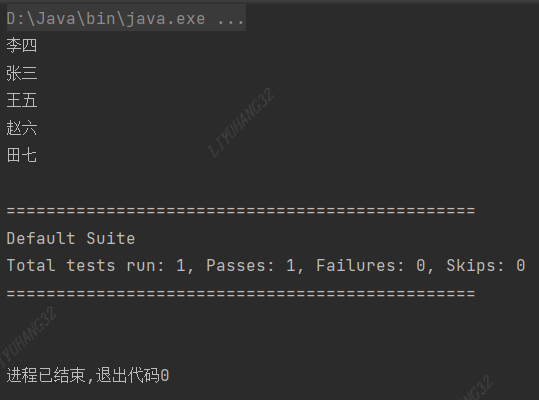
@Test  
public void test(){  
 Set<String> set=employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*toSet*());  
 set.forEach(System.*out*::println);  
}

图 4‑36 运行结果

##### toCollection(…)

把当前公司中所有员工的名字给提取出来，并把这些名字放到一个创建好的集合中，运行结果如图4-37所示：

@Test  
public void test(){  
 HashSet<String> hs= employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*toCollection*(HashSet::new));  
 hs.forEach(System.*out*::println);  
}

图 4‑37 运行结果

##### counting()

收集公司中员工的总人数 ，运行结果如图4-38所示：

@Test  
public void test(){  
 Long count= employees.stream()  
 .collect(Collectors.*counting*());  
 System.*out*.println(count);  
}

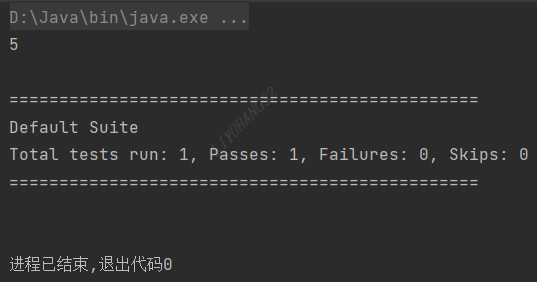
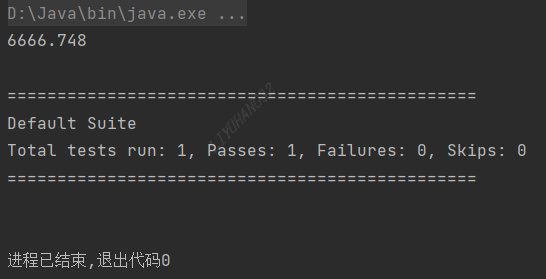


图 4‑38 运行结果

##### averagingDouble(…)

收集公司中所有员工工资的平均值，运行结果如图4-39所示：

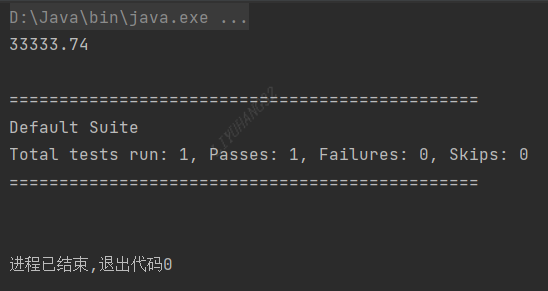
@Test  
public void test(){  
 Double avg= employees.stream()  
 .collect(Collectors.*averagingDouble*(Employee::getSalary));  
 System.*out*.println(avg);  
}

图 4‑39运行结果

##### summingDouble(…)

收集公司中所有员工工资的总和，运行结果如图4-40所示：

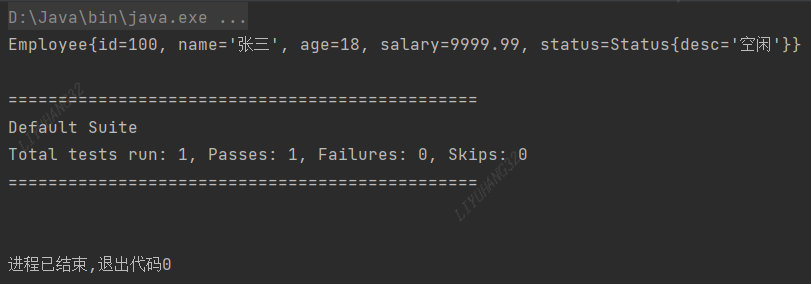
@Test  
public void test(){  
 Double sum=employees.stream()  
 .collect(Collectors.*summingDouble*(Employee::getSalary));  
 System.*out*.println(sum);  
}

图 4‑40 运行结果

##### maxBy(…)

收集公司中最高工资的那个员工的信息，运行结果如图4-41所示：

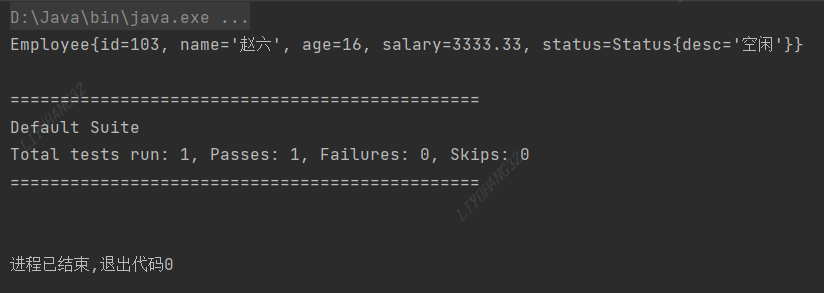
@Test  
public void test(){  
 Optional<Employee> max= employees.stream()  
 .collect(Collectors.*maxBy*((e1,e2)->Double.*compare*(e1.getSalary(),e2.getSalary())));  
 System.*out*.println(max.get());  
}

图 4‑41 运行结果

##### minBy(…)

收集公司中的最低工资运行结果如图4-42所示：

@Test  
public void test(){  
 Optional<Employee> max= employees.stream()  
 .collect(Collectors.*minBy*((e1,e2)->Double.*compare*(e1.getSalary(),e2.getSalary())));  
 System.*out*.println(max.get());  
}

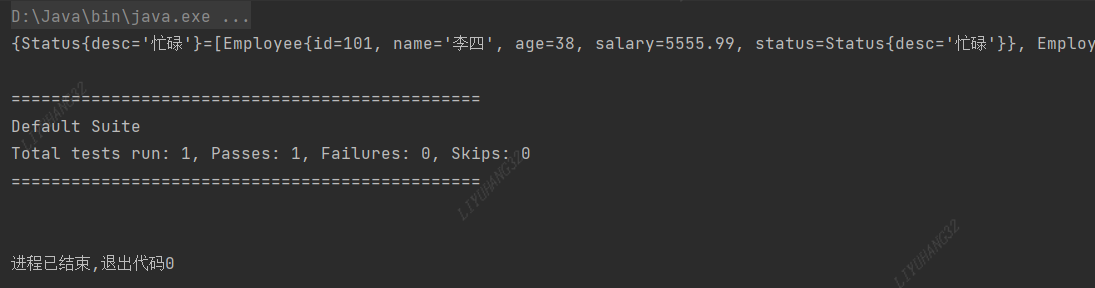
图 4‑42 运行结果

##### groupingBy(…)

###### 单级分组

将公司中所有员工按照状态来分组运行结果如图4=43所示：

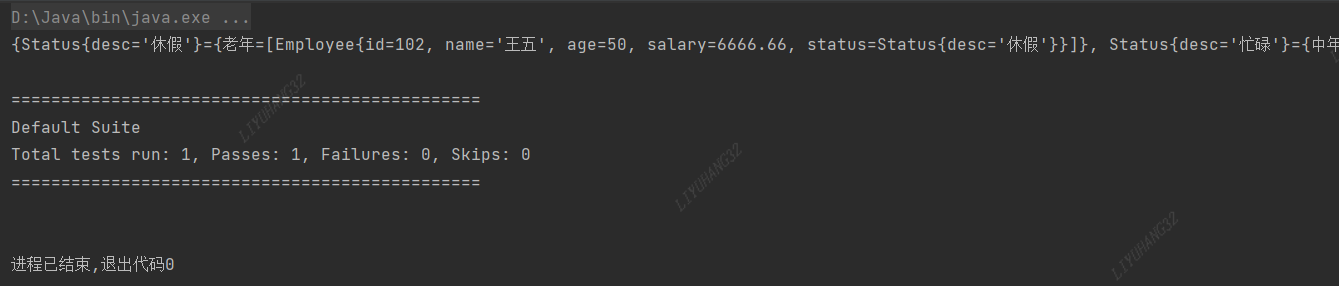
@Test  
public void test(){  
 Map<Status,List<Employee>> map= employees.stream()  
 .collect(Collectors.*groupingBy*(Employee::getStatus));  
 System.*out*.println(map);  
}

图 4‑43 运行结果

###### 多级分组

先按照状态分组，按照状态分组完之后，再按照年龄段来分组，比如说35岁以内的就叫青年，50岁以内的就叫中年，剩下的就叫老年，运行结果如图4-44所示：

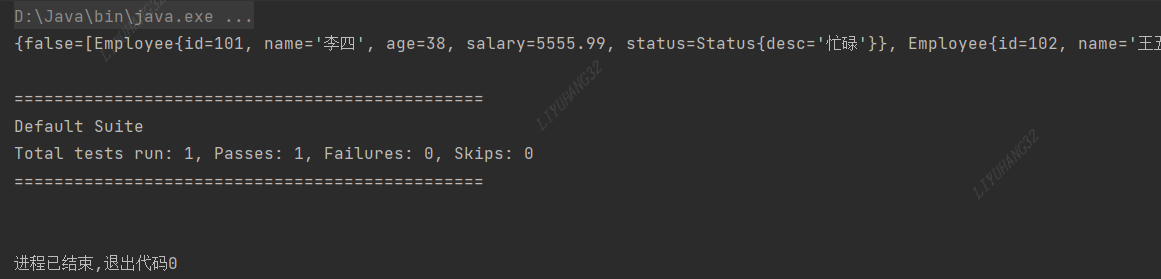
@Test  
public void test(){  
 Map<Status,Map<String,List<Employee>>> map= employees.stream()  
 .collect(Collectors.*groupingBy*(Employee::getStatus,Collectors.*groupingBy*(e->{  
 if (e.getAge()<35)  
 return "青年";  
 else if (e.getAge() < 50)  
 return "中年";  
 else  
 return "老年";  
 })));  
 System.*out*.println(map);  
}

图 4‑44 运行结果

##### partitioningBy(…)

公司中所有员工进行分区（分片），工资大于8000的一个区，剩下的一个区运行结果如图4-45所示：

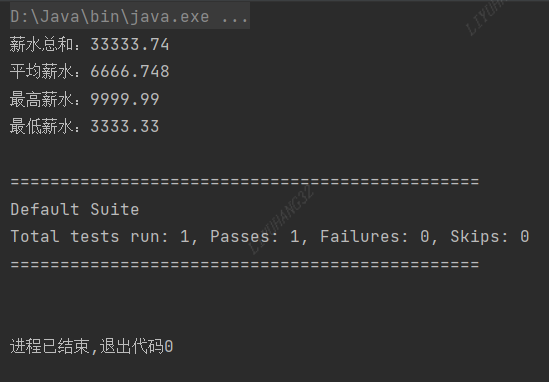
@Test  
public void test(){  
 Map<Boolean,List<Employee>> map=employees.stream()  
 .collect(Collectors.*partitioningBy*(e->e.getSalary()>8000));  
 System.*out*.println(map);  
}

图 4‑45 运行结果

##### summarizingDouble(…)

收集公司中所有员工工资的总和、平均值，还要收集公司中的最高与最低工资，运行结果如图4-46所示：

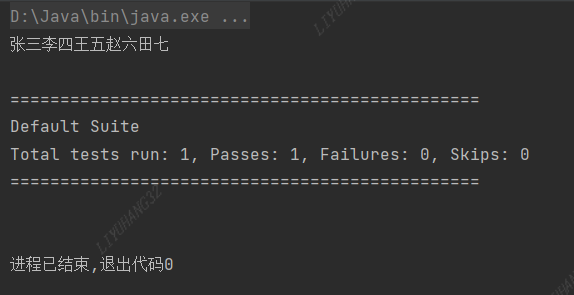
@Test  
public void test(){  
 DoubleSummaryStatistics dss= employees.stream()  
 .collect(Collectors.*summarizingDouble*(Employee::getSalary));  
 System.*out*.println("薪水总和："+dss.getSum());  
 System.*out*.println("平均薪水："+dss.getAverage());  
 System.*out*.println("最高薪水："+dss.getMax());  
 System.*out*.println("最低薪水："+dss.getMin());  
}

图 4‑46 运行结果

##### joining(…)

提取出公司中所有员工的名字，然后拼接成一串字符串，运行结果如图4-47所示：

@Test  
public void test(){  
 String str= employees.stream()  
 .map(Employee::getName)  
 .collect(Collectors.*joining*());  
 System.*out*.println(str);  
}

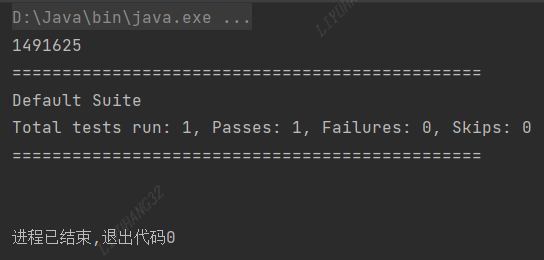
图 4‑47 运行结果

### Stream API实例

给定一个数字列表，如何返回一个由每个数的平方构成的列表呢？即给定[1, 2, 3, 4, 5]，应该返回[1, 4, 9, 16, 25]：

@Test  
public void test(){  
 Integer[] num=new Integer[] {1,2,3,4,5};  
 Arrays.*stream*(num)  
 .map(x->x\*x)  
 .forEach(System.*out*::print);  
}

运行结果如图4-48所示：

图 4‑48 运行结果

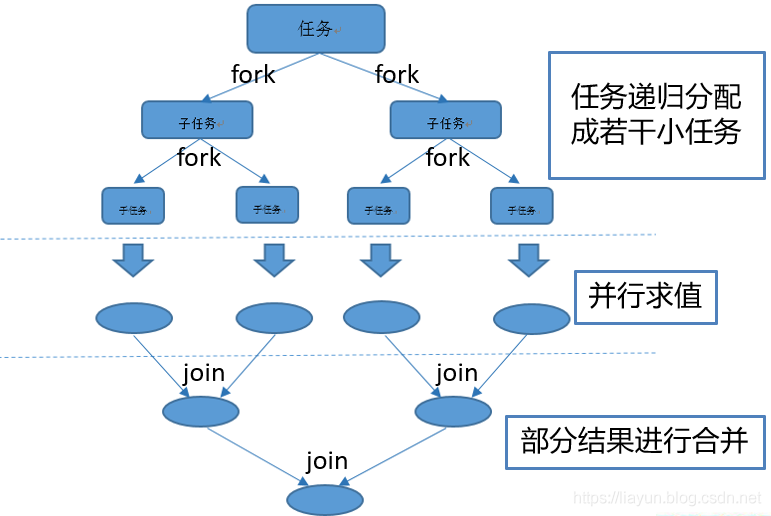
## 并行流与串行流

Java 8的速度变快了，之所以我们说Java 8的速度变快了，很重要的一点就是有了这个并行流，并行流起到了一个很关键的作用，它使得我们的程序很容易就能切换成一个多线程，而且更能利用CPU的资源。

### Fork/Join框架

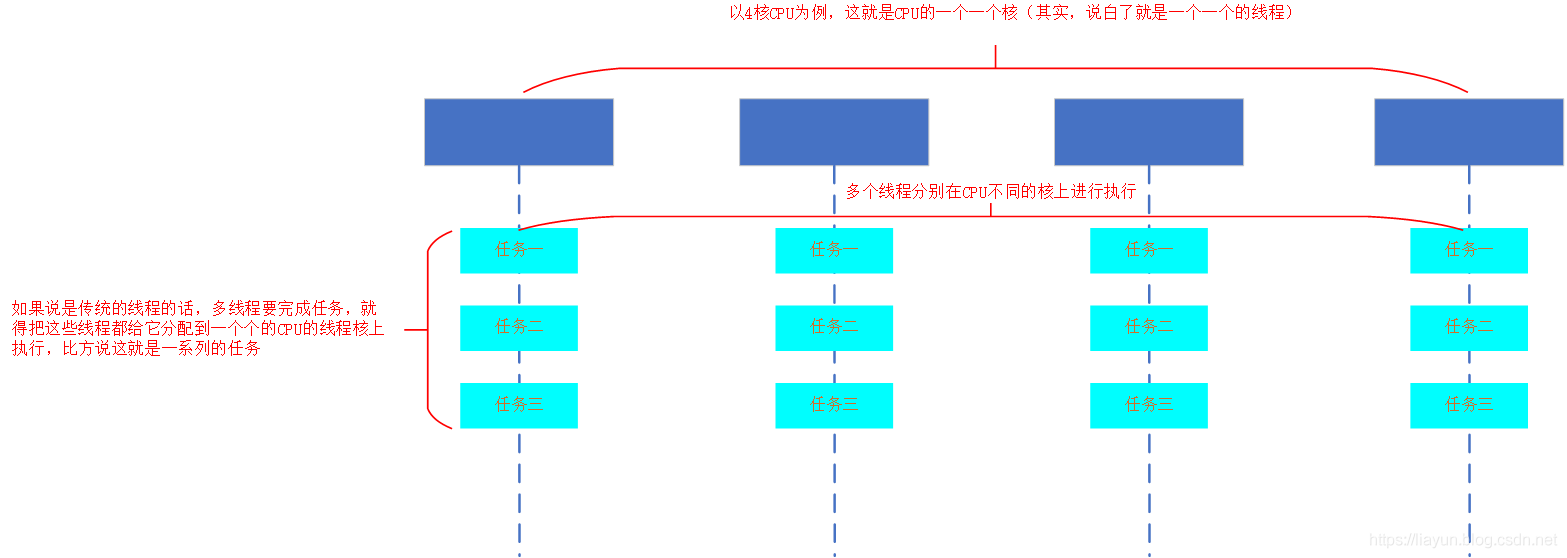
#### 了解Fork/Join框架

在讲解并行流以前，我们首先得提另外一个框架，这个框架就是Fork/Join框架，那什么是Fork/Join框架呢？所谓的Fork/Join框架就是在必要的情况下，将一个大任务进行拆分（fork）成若干个小任务（拆到不可再拆时为止），再将一个个的小任务运算的结果进行汇总（join），如图5-1所示。

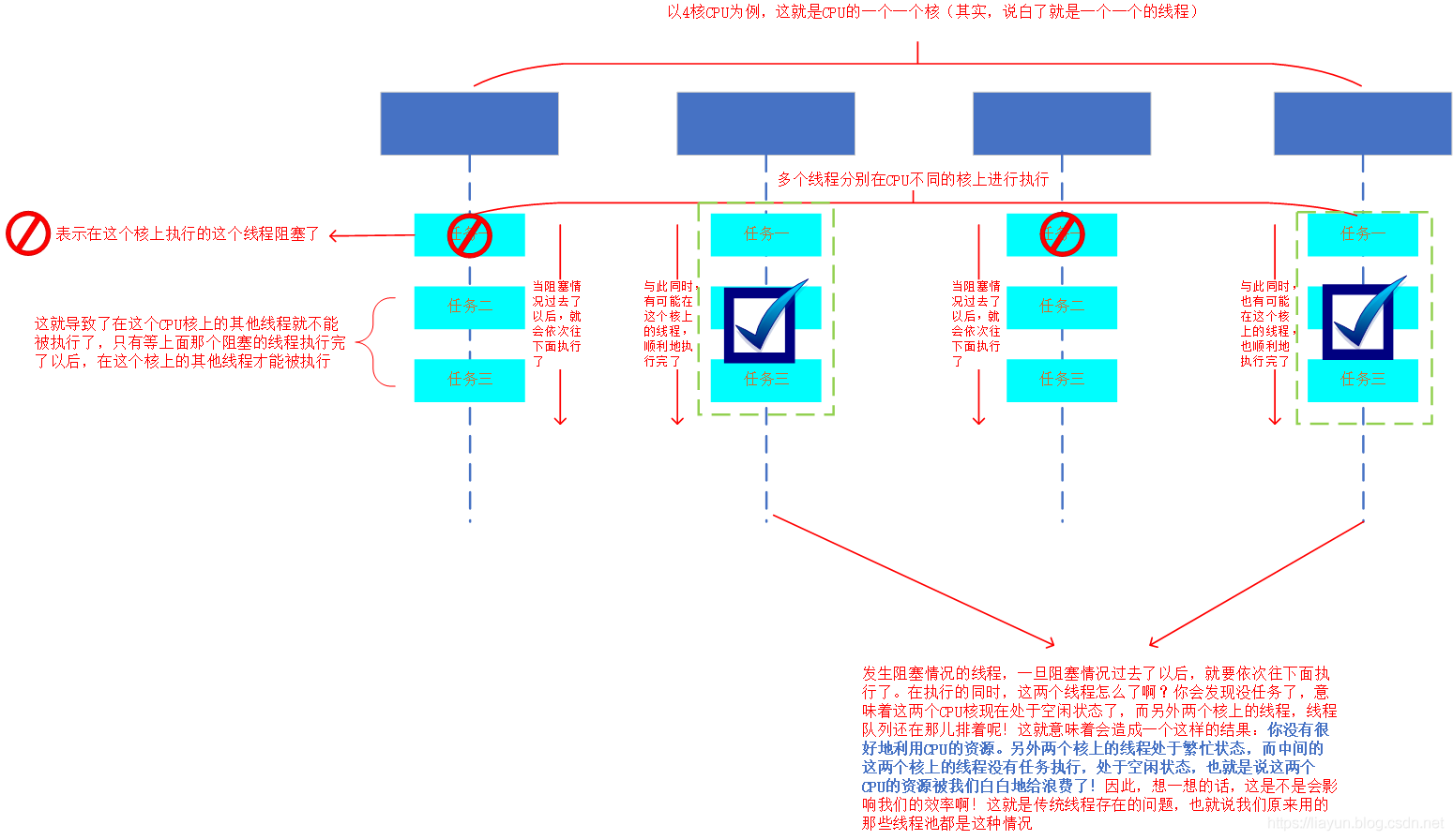
图 5‑1 Fork/Join框架

#### Fork/Join框架与传统线程池的区别

就拿多线程来说，所谓的多线程就是把我们的任务分别分配到CPU不同的核上（也就是CPU不同的线程上）进行执行，比如说，以4核CPU为例如图5-2所示。

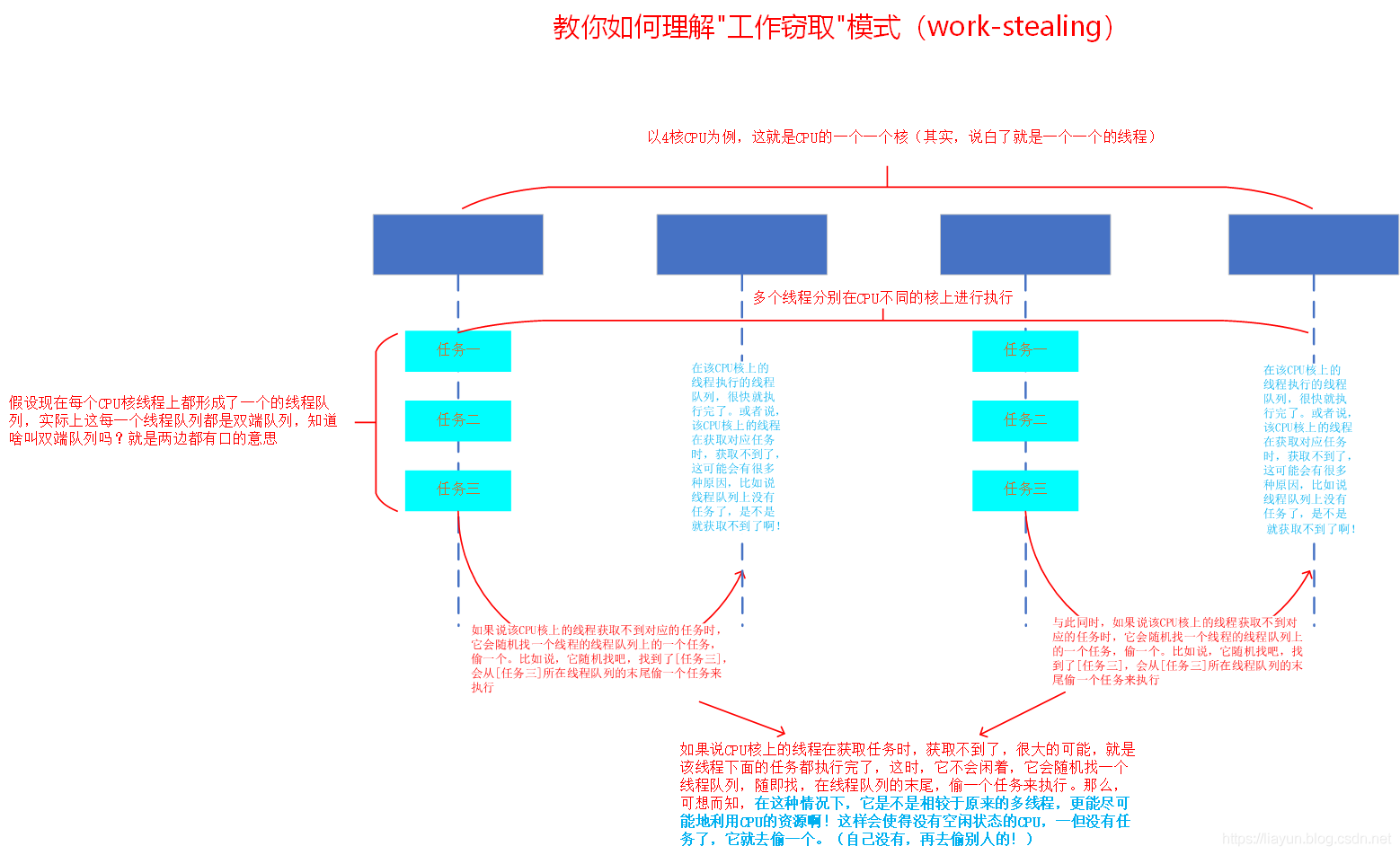
图 5‑2 多线程任务分配

如果说是传统的线程的话，那么会存在一个什么问题呢？在使用传统线程的情况下，每一个任务都有可能阻塞。为什么会发生阻塞这种情况呢？每个线程执行的时候，什么时候执行，这是由CPU时间片给它分配的执行权决定的，当这个时间片段用完了以后，CPU会强制剥夺它的执行权，然后交给其他的线程去执行，这时就有可能出现阻塞的情况。阻塞的情况有很多种，如果说一旦某一个线程发生了阻塞的情况的话，这会造成一个什么样的结果呢？

图 5‑3 传统线程任务分配

那么，现在的这个Fork/Join框架，它到底是怎么工作的呢？Fork/Join框架就是把一个大任务拆分成若干个小任务，然后它会把这些若干个小任务都压入到对应的线程当中（也就是说，它会把一个个的小任务压入到每个CPU的每个线程中，默认CPU有几核，它就利用几个线程），然后形成一个个的线程队列，形成一个个的线程队列以后，它会有什么特点呢？

这时，我们就不得不说说Fork/Join框架的核心了，即"工作窃取"模式（work-stealing），也就是说Fork/Join框架的执行采用的是"工作窃取"模式（work-stealing）。什么叫做"工作窃取"模式（work-stealing）呢？当执行新的任务时，它可以将其拆分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中，如图5-4所示。

图 5‑4 工作窃取

#### 小结

相对于一般的线程池实现，Fork/Join框架的优势体现在对其中包含的任务的处理方式上。在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在Fork/Join框架实现中，如果某个子问题由于等待另外一个子问题的完成而无法继续运行，那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行，这种方式减少了线程的等待时间，提高了性能。

### 并行流

并行流就是把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。

#### Fork/Join框架的简单使用

以计算1到100亿之间的和所用的时间为例：

@Test  
public void test(){  
 Instant start = Instant.*now*();//使用Java8中的新特性计算时间，得到一个时间戳  
 LongStream.*rangeClosed*(0, 10000000000L)  
 .parallel()  
 .reduce(0, Long::*sum*);  
 Instant end = Instant.*now*();  
 System.*out*.println(Duration.*between*(start,end).toMillis());  
}

## 重复注解与类型注解

### 重复注解

#### 重复注解是什么？

重复注解表示可以在同一处位置（方法、类以及类变量等等）多次使用同一个注解，效果如下所示：

public class AnnotationDemo {  
 @MyAnnotation("Hello")  
 @MyAnnotation("World")  
 public void test(){  
 System.*out*.println("李宇航");  
 }  
}

#### 如何定义重复注解？

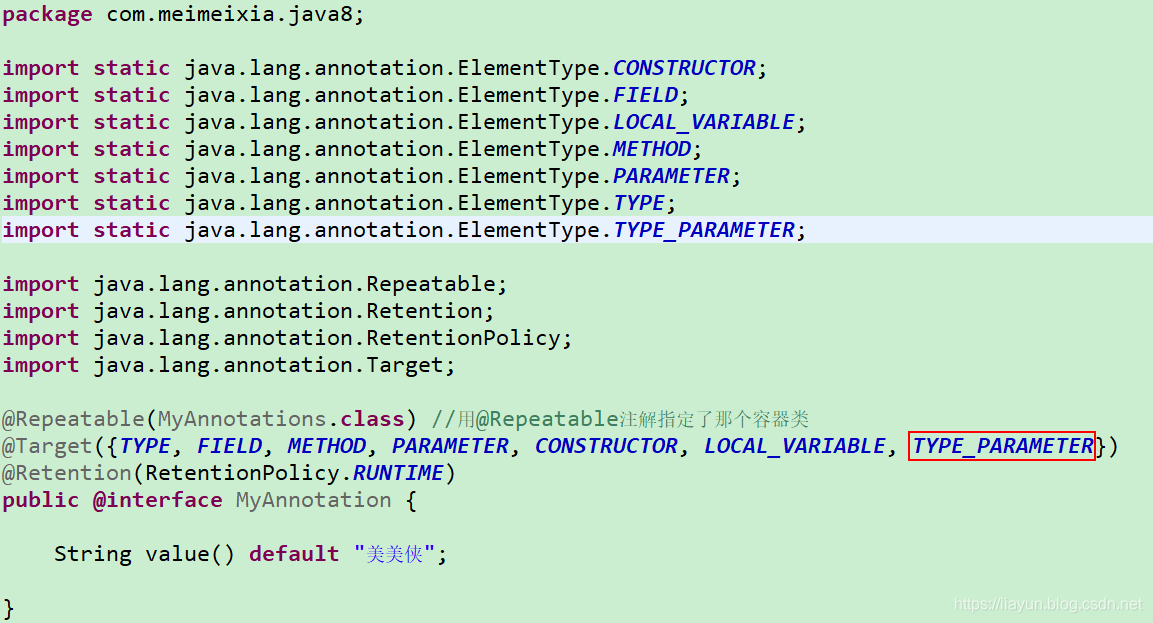
在Java 8中提供了一个@Repeatable注解，对用户来说，它可以隐藏掉注解容器，所以，定义重复注解只需要在原有的注解基础上添加@Repeatable这个元注解标签，如图6-1所示。



图 6‑1 @Repeatable注解

### 类型注解

在注解类中的@Target元注解中添加变量"TYPE\_PARAMETER"，可以使得注解能够定义在类型上如图6-2所示。

图 6‑2 类型注解

## 接口中的默认方法和静态方法

### 默认方法

我们说接口中以前只能有两个成员，一个是全局静态常量，另外一个是抽象方法。而现在在Java 8中，它不仅可以有这些，而且Java 8允许接口中包含具有具体实现的方法，该方法称为"默认方法"，默认方法使用default关键字修饰，效果如下所示：

public interface MyFun {  
 default String getName(){  
 return "李宇航";  
 }  
}

不禁我们就要想了接口中的默认方法有什么特点呢？接口中的默认方法有一个原则，即"类优先"原则。啥是"类优先"原则呢？若一个接口中定义了一个默认方法，而另一个父类或接口中又定义了一个同名的方法时，则：

1、选择父类中的方法：如果一个父类提供了具体的实现，那么接口中具有相同名称和参数的默认方法会被忽略；

2、接口冲突：如果一个父接口提供一个默认方法，而另一个接口也提供了一个具有相同名称和参数列表的方法（不管方法是否是默认方法），那么必须覆盖该方法来解决冲突。

具体描述如下，运行结果如图7-1所示：

MyFun接口：

public interface MyFun {  
 default String getName(){  
 return "李宇航";  
 }  
}

父类MyClass:

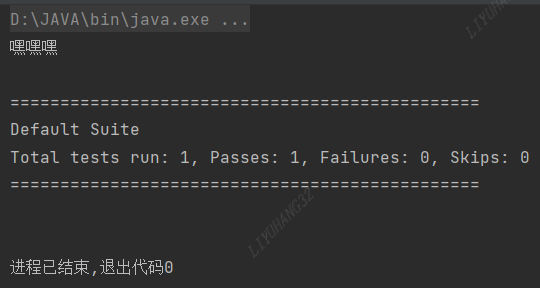
public class MyClass {  
 public String getName() {  
 return "嘿嘿嘿";  
 }  
}

子类SubClass，继承MyClass，实现MyFun：

public class SubClass extends MyClass implements MyFun{}

实现类：

@Test  
public void test() {  
 SubClass sc =new SubClass();  
 System.*out*.println(sc.getName());  
}

图 7‑1 运行结果

从以上运行结果可知，若一个接口中定义了一个默认方法，而另一个父类中又定义了一个同名的方法时，那么会选择父类中的方法，接口中具有相同名称和参数的默认方法会被忽略掉。

若一个接口中定义了一个默认方法，而另一个接口中又定义了一个具有相同名称和参数列表的方法（不管方法是否是默认方法），则多继承时会出现接口冲突的情况，此时，必须覆盖该方法来解决冲突。例如，有下面这样的两个接口和一个实现类：

接口（MyFun）：

public interface MyFun {  
 default String getName(){  
 return "李宇航";  
 }  
}

接口（MyInterface）

public interface MyInterface {  
 default String getName() {  
 return "呵呵呵";  
 }  
}

实现类（SubClass），如果把该实现类写成下面这样，那么你会发现编译报错，其实就是相同的默认方法有两个，不知道用哪个，这个时候你就必须实现getName()默认方法了：

public class SubClass implements MyFun,MyInterface{}

所以，我们还得必须实现getName()默认方法：

public class SubClass implements MyFun,MyInterface{  
 @Override  
 public String getName(){  
 return MyFun.super.getName();  
// return MyInterface.super.getName();  
 }  
}

### 静态方法

在Java 8中，接口中还允许添加静态方法。例如：

//在Java 8中，接口中不仅可以有默认方法，还可以有静态方法  
public static void show() {  
 System.*out*.println("接口中的静态方法");  
}

如何调用以上静态方法呢？和之前一样，可以通过父接口.方法这种格式调用：

MyInterface.show();

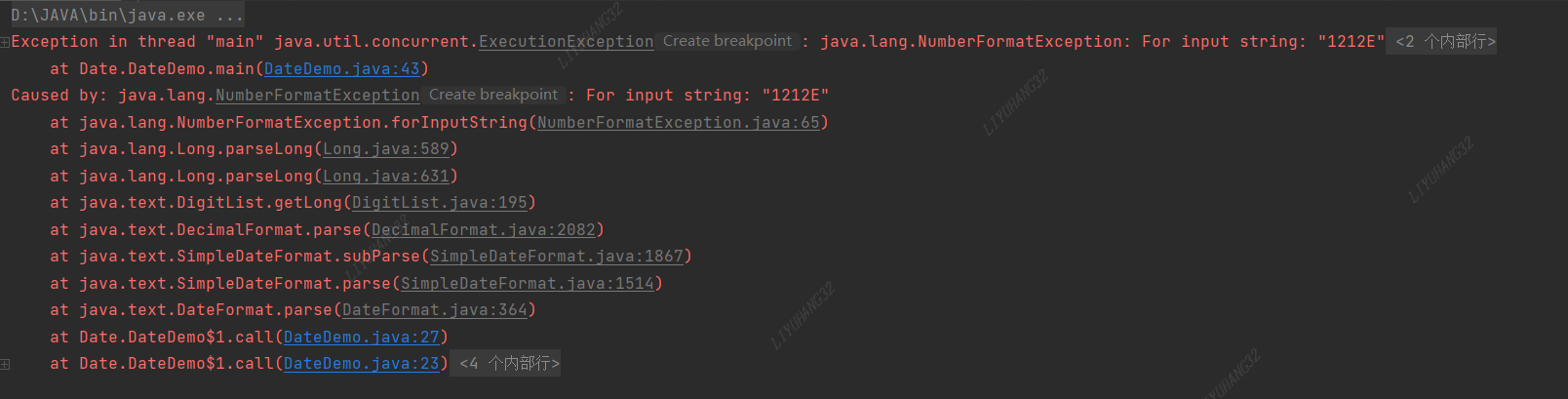
## 传统日期时间格式化的多线程安全问题

回顾一下以前的那套日期时间API，你就能发现它是线程不安全的，是可变的。这里就以传统日期时间格式化为例，看看它存在什么多线程安全问题？

如果我们想要使用SimpleDateFormat类来对一个时间或者日期进行格式化，并且还要使用多线程来操作，即使用多线程同时对一个时间或者日期进行格式化，那么该怎么办呢？我们可以创建一个线程池，然后分10次去访问定义好的一个任务（该任务就是专门用于格式化一个时间或者日期的），都来解析某个时间或者日期：

package Date;  
  
import java.text.SimpleDateFormat;  
import java.util.ArrayList;  
import java.util.Date;  
import java.util.List;  
import java.util.concurrent.Callable;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.Future;  
  
*/\*\*  
 \* 这里就以传统日期时间格式化为例，看看它存在什么多线程安全问题？  
 \** ***@author*** *liayun  
 \*  
 \*/*public class DateDemo {  
  
 public static void main(String[] args) throws Exception {  
 SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");  
  
 //定义好如下一个任务（task），该任务就是专门用于格式化一个时间或者日期的  
 Callable<Date> task = new Callable<Date>() {  
  
 @Override  
 public Date call() throws Exception {  
 return sdf.parse("20191207");  
 }  
  
 };  
  
 //创建一个长度为10的线程池  
 ExecutorService pool = Executors.*newFixedThreadPool*(10);  
  
 List<Future<Date>> results = new ArrayList<Future<Date>>();  
  
 //分10次去访问以上定义好的任务（task），然后它就会返回一个结果（叫Future），结果我都给它放在上面的集合里面  
 for (int i = 0; i < 10; i++) {  
 results.add(pool.submit(task));  
 }  
  
 for (Future<Date> future : results) {  
 System.*out*.println(future.get());  
 }  
 }  
  
}

运行以上程序，会发现报了如下错误，日期已经格式化不下去了。这说明已经存在多线程安全问题了，也就是说SimpleDateFormat类或者传统的时间日期API均存在多线程安全问题，如图8-1所示。

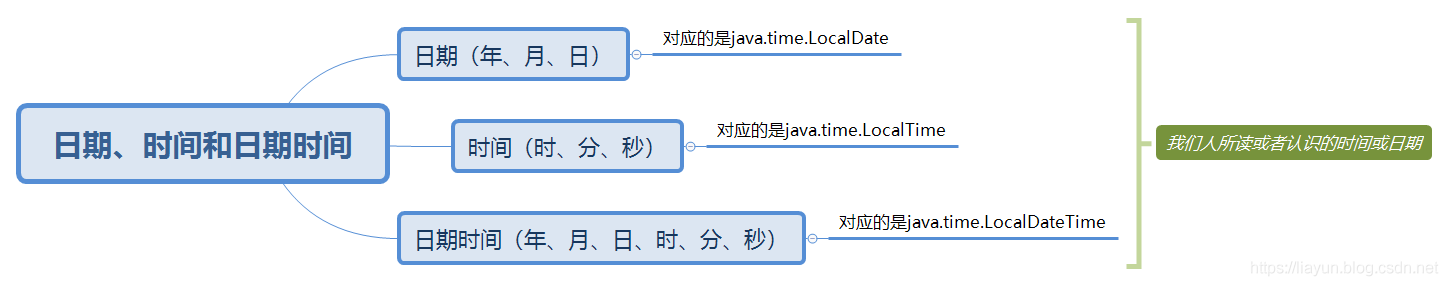
图 8‑1 SimpleDateFormat类或者传统的时间日期API均存在多线程安全问题

而现在，使用Java 8中这套全新的日期时间API之后，就没有什么多线程安全问题了，因为你不管做什么样的改变，它都会给你产生一个全新的实例，所以说它是线程安全的，这样就解决了多线程的安全问题（下章）。

## 全新的日期时间API

### 日期、时间和日期时间

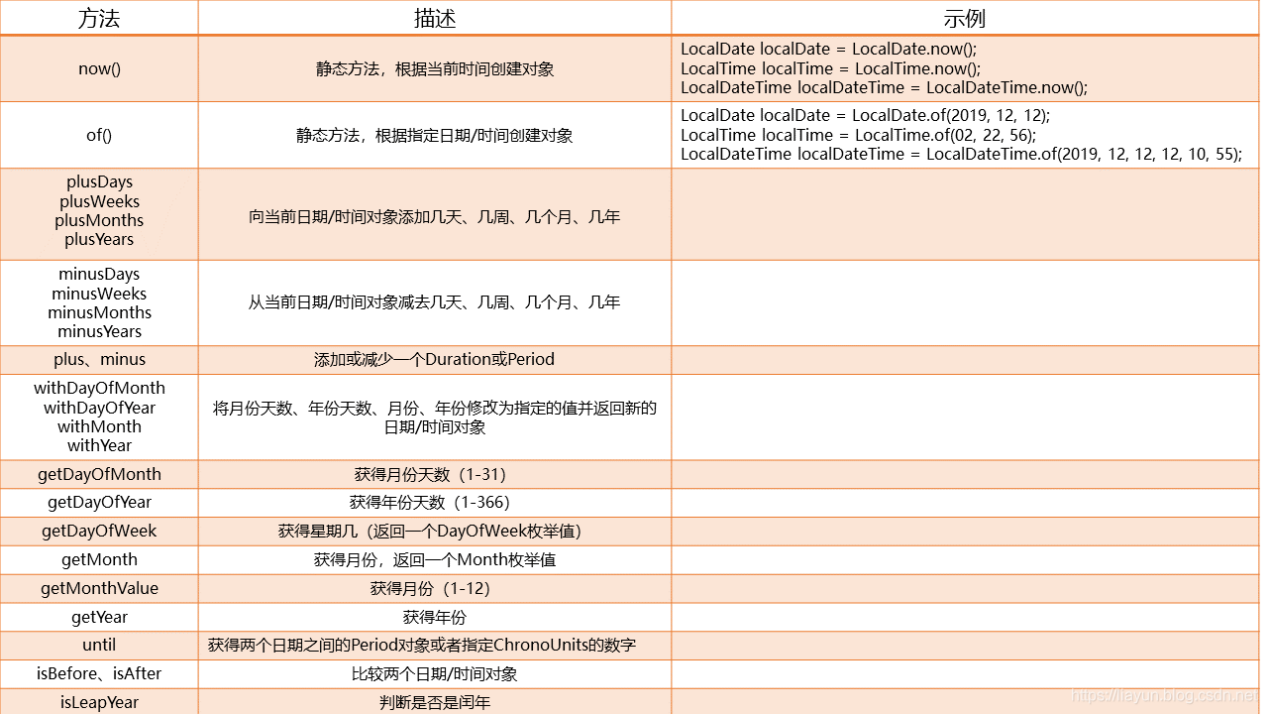
日期、时间和日期时间的描述如图9-1所示。

图 9‑1

LocalDate、LocalTime以及LocalDateTime类的实例是不可变的对象，也就是说不管做什么样的改变，都将产生一个全新的实例，所以才说它是线程安全的，能够解决多线程的安全问题。它们分别表示使用ISO-8601日历系统的日期、时间以及日期和时间，它们提供了简单的日期或时间，并不包含当前的时间信息，也不包含与时区相关的信息。温馨提示：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。

### 常用方法

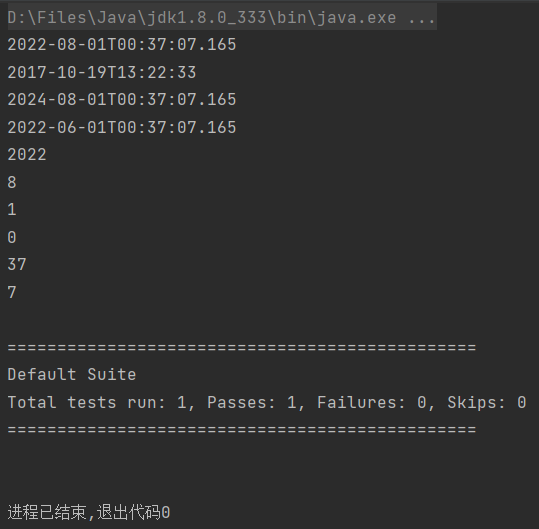
这三个类的方法都差不多了多少，常用的方法如图9-2所示。

图 9‑2 日期、时间和日期时间的常用方法

下面我就以LocalDateTime为例来演示一下上表所列出的一些方法：

@Test  
public void test() {  
 LocalDateTime ldt = LocalDateTime.*now*();//获取当前系统日期时间  
 System.*out*.println(ldt);  
  
 LocalDateTime ldt2 = LocalDateTime.*of*(2017, 10, 19, 13, 22, 33);//构造一个指定日期时间（哪年哪月哪日 几时几分几秒）的对象  
 System.*out*.println(ldt2);  
  
 //注意，不管做什么样的改变，它都将产生一个全新的实例  
 LocalDateTime ldt3 = ldt.plusYears(2);//将当前日期时间加上2年  
 System.*out*.println(ldt3);  
  
 LocalDateTime ldt4 = ldt.minusMonths(2);//将当前日期时间减去2个月  
 System.*out*.println(ldt4);  
  
 System.*out*.println(ldt.getYear()); //输出当前日期时间的年份  
 System.*out*.println(ldt.getMonthValue()); //输出当前日期时间的月份  
 System.*out*.println(ldt.getDayOfMonth()); //输出当前日期时间的月份天数（即几号，）  
 System.*out*.println(ldt.getHour()); //输出当前日期时间的时  
 System.*out*.println(ldt.getMinute()); //输出当前日期时间的分  
 System.*out*.println(ldt.getSecond()); //输出当前日期时间的秒  
}

运行以上测试方法，你便能看到如图9-2所示的内容。

图 9‑3 运行结果

### 时间戳

时间戳对应的是java.time.Instant，它专门用于"时间戳"的运算。什么是"时间戳"？指的是从Unix元年（传统的设定为UTC时区1970年1月1日午夜时分）开始到某个时间之间的毫秒值。有一点需要知道，时间戳是给计算机读的，它不是给我们人读的。下面，我会用一个案例来演示一下时间戳：

@Test  
public void test() {  
 Instant ins1 = Instant.*now*();//默认获取的是以UTC时区（世界协调时间，也叫格林威治时间）为基础的（当前时间的）时间戳  
 System.*out*.println(ins1);//因为中国在东八区，所以这句代码输出的时间跟我的电脑时间是不同的  
  
 //做一个偏移量的运算  
 OffsetDateTime odt = ins1.atOffset(ZoneOffset.*ofHours*(8));//既然中国在东八区，故要偏移8个小时，这样子获取到的时间才是自己电脑的时间  
 System.*out*.println(odt);//带偏移量的时间和日期  
  
 System.*out*.println(ins1.toEpochMilli());//转成对应的毫秒值，如果是当前时间的时间戳，结果跟System.currentTimeMillis()是一样的  
  
 //对时间戳做一个稍微的改变  
 Instant ins2 = Instant.*ofEpochSecond*(60);  
 System.*out*.println(ins2);//此时，得到的是1970-01-01T00:01:00Z，也即Unix元年过去1分钟之后的时间戳  
}

运行以上测试方法，你便能看到如图9-4所示的内容。

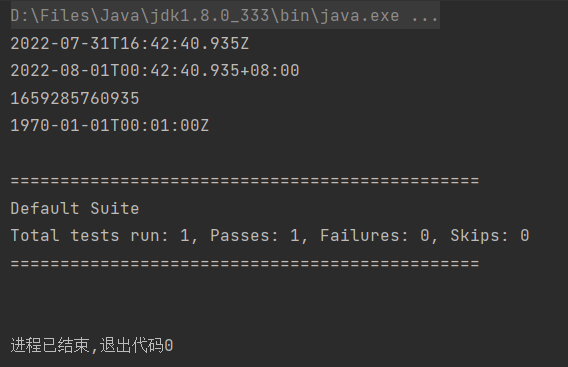
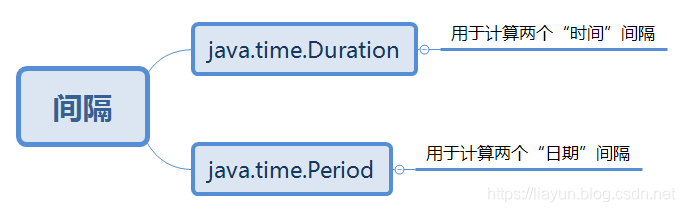


图 9‑4 运行结果

### 间隔

表示间隔的有两个类，它们分别是：java.time.Duration——用于计算两个“时间”间隔、java.time.Period——用于计算两个“日期”间隔。如图9-5所示。

图 9‑5 两个间隔

使用Duration类不仅可以计算两个时间之间的间隔，还能计算两个时间戳之间的间隔：

@Test  
public void test() {  
 Instant ins1 = Instant.*now*();  
 try {  
 Thread.*sleep*(3000);//中间睡一会  
 } catch (InterruptedException e) {  
  
 }  
 Instant ins2 = Instant.*now*();  
  
 //计算两个时间戳之间的间隔  
 Duration duration = Duration.*between*(ins1, ins2);  
 // System.out.println(duration);//PT3S  
 System.*out*.println(duration.toMillis());//获取毫秒  
  
 System.*out*.println("---------------------------------");  
  
 LocalTime lt1 = LocalTime.*now*();  
 try {  
 Thread.*sleep*(3000);//中间睡一会  
 } catch (InterruptedException e) {  
  
 }  
 LocalTime lt2 = LocalTime.*now*();  
  
 //计算两个时间之间的间隔  
 Duration duration2 = Duration.*between*(lt1, lt2);  
 // System.out.println(duration2);//可能会输出PT3S或者输出PT3.001S，至于多出来的0.001秒其实就是除去线程睡眠时间之外执行计算时间间隔那句代码消耗的时间  
 System.*out*.println(duration2.toMillis());//可能会输出3000或者输出3001  
}

运行以上测试方法，你便能看到如图9-6所示的内容。

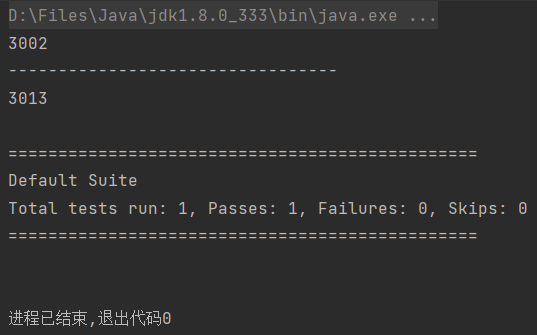


图 9‑6 运行结果

使用Period类可以计算两个日期之间的间隔，示例代码如下：

@Test  
public void test() {  
 //起始时间指定为2019年1月1日  
 LocalDate ld1 = LocalDate.*of*(2019, 1, 1);  
 //终止时间指定为当前时刻  
 LocalDate ld2 = LocalDate.*now*();  
  
 Period period = Period.*between*(ld1, ld2);  
 System.*out*.println(period);//可能会输出P3Y7M，Y代表年，M代表月，D代表日，说明起始时间和当前时刻之间的日期间隔是0年11个月零17天  
  
 System.*out*.println(period.getYears());  
 System.*out*.println(period.getMonths());  
 System.*out*.println(period.getDays());  
}

运行以上测试方法，你便能看到如图9-7所示的内容。

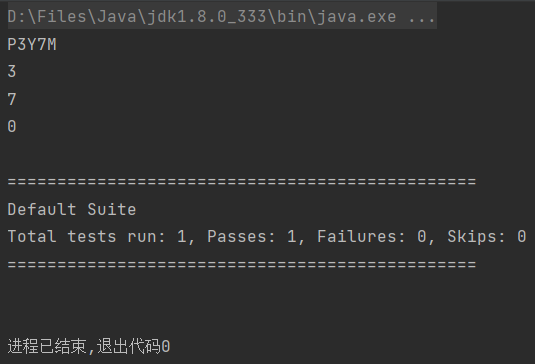


图 9‑7 运行结果

### 时间校正器

有时我们可能需要执行一些特殊的操作，例如将日期调整到"下一个周日"、"下一个工作日"或者"下一个结婚纪念日"等操作。执行这些特殊的操作，就得使用到TemporalAdjuster，它是一个接口，也被称为时间校正器。除此之外，还得知道一个TemporalAdjusters类，该类通过静态方法提供了大量的常用TemporalAdjuster的实现。将日期调整到"下一个周日"的代码如下：

@Test  
public void test() {  
 LocalDateTime ldt = LocalDateTime.*now*();  
 System.*out*.println(ldt);  
  
 LocalDateTime ldt2 = ldt.withDayOfMonth(10);//将当前系统日期时间中的月份天数指定为10  
 System.*out*.println(ldt2);  
  
 //通过时间校正器，就可以指定一些特殊的操作了，比如说下一个周日、下一个工作日、下一个结婚纪念日  
 //下一个周日  
 LocalDateTime ldt3 = ldt.with(TemporalAdjusters.*next*(DayOfWeek.*SUNDAY*));  
 System.*out*.println(ldt3);  
}

运行以上测试方法，你便能看到如图9-8所示的内容。

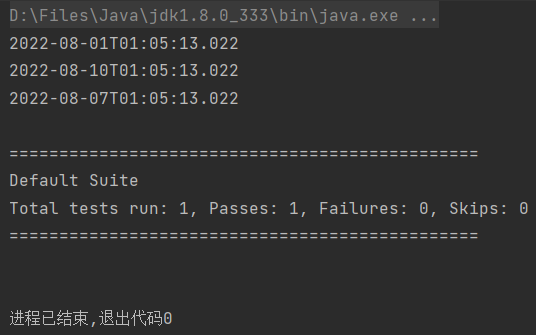
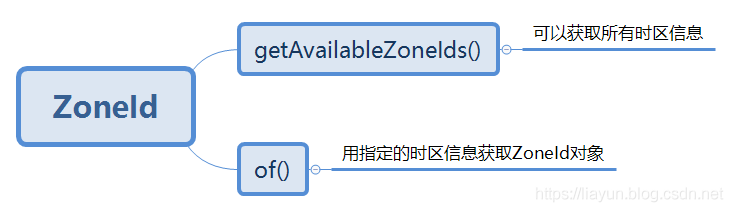


图 9‑8 运行结果

### 时区

Java 8中加入了对时区的支持，带时区的时间为分别为：ZonedDate、ZonedTime以及ZonedDateTime。其中每个时区都对应着地区ID，地区ID都为"{区域}/{城市}"的格式，例如Asia/Shanghai。

还有一个类，我们也得知道，用的还比较多，它就是ZoneId，该类包含了所有的时区信息，而且它还有两个比较常用的方法，如图9-9所示。

图 9‑9 时区方法

我们可以通过java.time.ZoneId类去查看在Java 8中支持的时区有哪些，如：

@Test  
public void test() {  
 //在Java 8中，支持多少时区  
 Set<String> set = ZoneId.*getAvailableZoneIds*();  
 set.forEach(System.*out*::println);  
}

运行以上测试方法，你便能看到如图9-10所示的内容。

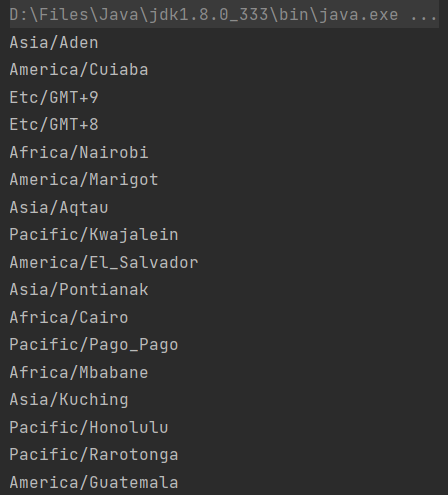


图 9‑10 运行结果

# MySQL数据库

## 单表查询

### 查询所有字段

select查询记录最简单的形式是从一个表中检索所有记录,实现的方法是使用星号(\*)通配符指定查找所有列的名称。语法格式如下：select \* from 表名;

### 查询指定字段

#### 查询单个字段

查询表中的某一个字段，语法格式为：select 列名 from 表名;

#### 查询多个字段

使用select声明，可以获取多个字段下的数据，只需要在关键字select后面指定要查找的字段的名称，不同字段名称之间用逗号（,）分隔开，最后一个字段后面不需要加逗号，语法格式如下；select 字段名1,字段名2,…,字段名n from 表名;

### 查询结果不重复

在select语句中，可以使用distinct关键字指示MySQL消除重复的记录值：

select distinct 字段名 from 表名;

### 对查询结果排序

#### 升序排序（asc）,默认的排序方式，加与不加都可以

select 字段名1 from 表名 order by 字段名2 asc;

#### 降序排序（desc）,如果选择降序排序，必须末尾加关键字desc

select 字段名1 from 表名 order by 字段名2 desc;

### 分组查询

分组查询是对数据按照某个或多个字段进行分组, MySQL中使用 group by关键字对数据进行分组,基本语法形式为：

group by 字段名 having <条件表达式>;

字段值为进行分组时所依据的列名称;"HAVING<条件表达式>"指定满足表达式限定条件的结果将被显示。

### 使用 LIMIT限制查询结果的数量

SELECT返回所有匹配的行,有可能是表中所有的行,如仅仅需要返回第一行或者前几行,使用 LIMIT关键字,基本语法格式如下：

limit 位置偏移量,行数;

## 条件查询

### 查询指定记录

数据库中包含大量的数据，根据特殊要求，可能只需要查询表中的指定数据，即对数据进行过滤。在select语句中，通过where子句可以对数据进行过滤，语法格式为：

select 字段名1,字段名2,…,字段名n

from 表名

where c查询条件;

在where子句中，MySQL提供了一系列条件判断符，查询结果如表2-1所示。

表 2‑1 条件判断符

|  |  |
| --- | --- |
| 操作符 | 说明 |
| =  <>,!=  <  <=  >  >= | 相等  不等于  小于  小于或者等于  大于  大于或者等于 |
| between | 位于两指之间 |

### 带in关键字的查询

in操作符用来查询满足指定范围内的条件的记录,使用in操作符,将所有检索条件用括号括起来,检索条件之间用逗号分隔开,只要满足条件范围内的一个值即为匹配项：

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m in(条件1,条件2);

### 带between and的范围查询

between and用来查询某个范围内的值，该操作符需要两个参数，及范围的开始值和结束值，如果字段值满足指定的那位查询条件，则这些记录被退回：select 字段名1,字段名2,…,字段名n from 表名 where 字段名m between 条件1 and 条件2;

### 查询空值

数据表创建的时候,设计者可以指定某列中是否可以包含空值(NULL)。空值不同于0,也不同于空字符串。空值一般表示数据未知、不适用或将在以后添加数据。在 select语句中使用is NULL子句,可以查询某字段内容为空记录。

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m is NULL;

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m is not NULL;

### 带and的多条件查询

使用select查询时,可以增加查询的限制条件,这样可以使查询的结果更加精确。 MySQL在 where子句中使用and操作符限定只有满足所有查询条件的记录才会被返回。可以使用and连接两个甚至多个查询条件,多个条件表达式之间用and分开：

select 字段名1,字段名2,…,字段名n from 表名 where 字段名a=条件1 and 字段名b=条件2;

### 带or的多条件查询

与and相反,在 where声明中使用or操作符,表示只需要满足其中一个条件的记录即可返回。or也可以连接两个甚至多个查询条件,多个条件表达式之间用or分开：

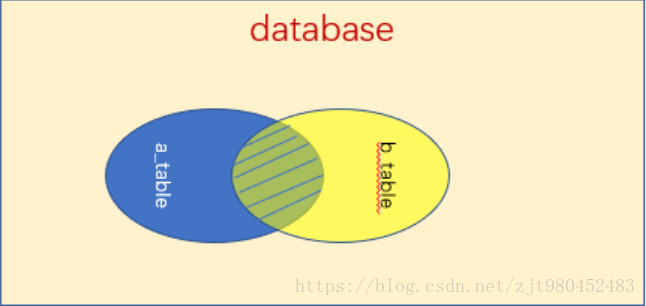
select 字段名1,字段名2,…,字段名n from 表名 where 字段名a=条件1 or 字段名b=条件2;

## 多表联查（内连接）

关键字：inner join on

语句：select \* from a\_table inner join b\_table on a.a\_id = b.b\_id;

说明：组合两个表中的记录，返回关联字段相符的记录，也就是返回两个表的交集（阴影）部分，如图3-1所示。

图 3‑1 内连接

## 多表联查（外连接）

### 左连接查询 left join

关键字：left join on / left outer join on

语句：select \* from a\_table a left join b\_table b on a.a\_id = b.b\_id;

说明： left join 是left outer join的简写，它的全称是左外连接，是外连接中的一种。 左(外)连接，左表(a\_table)的记录将会全部表示出来，而右表(b\_table)只会显示符合搜索条件的记录。右表记录不足的地方均为NULL，如图4-1所示。

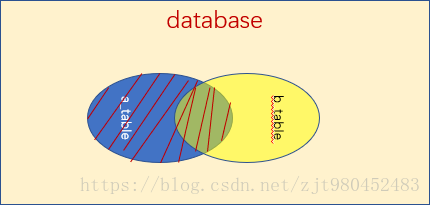


图 4‑1 左连接

### 右连接 right join

关键字：right join on / right outer join on

语句：select \* from a\_table a right outer join b\_table b on a.a\_id = b.b\_id;

说明：right join是right outer join的简写，它的全称是右外连接，是外连接中的一种。与左(外)连接相反，右(外)连接，左表(a\_table)只会显示符合搜索条件的记录，而右表(b\_table)的记录将会全部表示出来。左表记录不足的地方均为NULL，如图4-2所示。

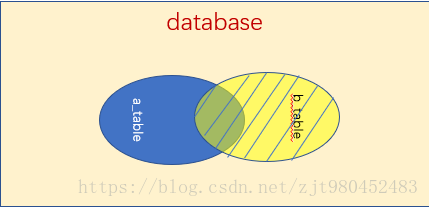


图 4‑2 右连接

## And和OR的优先级

在一般的编程语言中and与or的优先级是相同的,在运算的时候按照顺序从左到右开始运算,如Java、Python等,但是在MySQL中，对and和or的优先级做了一下调整.使得and的运算优先级高于or的优先级。

在实际执行的过程中,同时遇到and和or运算符的时候,会优先执行and运算符,然后再执行or运算.如下:

select \* from t1 where 条件1 and 条件2 or 条件3;等价于select \* from t1 where (条件1 and 条件2) or 条件3;

select \* from t1 where 条件1 and 条件2 or 条件3 and 条件4;等价于select \* from t1 where (条件1 and 条件2) or (条件3 and 条件4);

## 模糊查询

### 通配符的分类

1、%" 百分号通配符: 表示任何字符出现任意次数 (可以是0次)。

2、"\_" 下划线通配符:表示只能匹配单个字符,不能多也不能少,就是一个字符。当然，也可以like "陈\_\_\_\_"，数量不限。

like操作符:like作用是指示MySQL后面的搜索模式是利用通配符而不是直接相等匹配进行比较；但如果like后面没出现通配符，则在SQL执行优化时将 like 默认为 “=”执行。

注意: 如果在使用like操作符时，后面没有使用通用匹配符（%或\_），那么效果是和“=”一致的。在SQL执行优化时查询优化器将 like 默认为 “=”执行,select \* from movies where movie\_name like '唐伯虎';只能匹配movie\_name=“唐伯虎”的结果,而不能匹配像“唐伯虎点秋香”或“唐伯虎点香烟”这样的结果。

#### 百分号通配符'%'，匹配任意长度的字符，甚至包括零字符

如表6-1所示：

表 6‑1 百分号通配符

|  |  |
| --- | --- |
| 操作符 | 说明 |
| s% | 查找以s开头的字符串 |
| %s | 查找以s结尾的字符串 |
| %s% | 查找含有s的字符串 |
| s1%s2 | 查找以s1开头且以s2结尾的字符串 |

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m like 's%’;

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m like ‘%s’;

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m like ‘%s%';

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m like 's1%s2';

#### 下划线通配符'\_',一次只能匹配任意一个字符

另一个非常有用的通配符是下划线通配符'',该通配符的用法和'%'相同,区别是'%'可以匹配多个字符,而'\_'只能匹配任意单个字符,如果要匹配多个字符,则需要使用相同个数的'\_'：

select 字段名1,字段名2,…,字段名n from 表名 where 字段名m like '\_s';

### MySQL内置函数检索(locate,position,instr)

话接上文，通过内置函数locate,position,instr进行匹配，相当于Java中的str.contains()方法，返回的是匹配内容在字符串中的位置，效率和可用性上都优于通配符匹配。

SELECT \* from app\_info where INSTR(`appName`, '%') > 0;

SELECT \* from app\_info where LOCATE('%', `appName`) > 0;

SELECT \* from app\_info where POSITION( '%' IN `appName`) > 0;

MySQL中的角标从左往右是从1开始的，不像Java最左边第一位角标是0，因此在MySQL中角标为0时说明不存在。

#### LOCATE()函数

##### LOCATE(substr,str)

返回 substr 在 str 中第一次出现的位置。如果 substr 在 str 中不存在，返回值为 0，如果substr 在 str 中存在，返回值为：substr 在 str中第一次出现的位置。

##### LOCATE（substr, str, [pos]）

从位置pos开始的字符串str中第一次出现子字符串substr的位置。 如果substr不在str中，则返回0。 如果substr或str为NULL，则返回NULL。

#### POSITION()方法

语法：POSITION(substr IN substr)。

这个方法可以理解为locate(substr，str)方法的别名，因为它和locate(substr，str)方法的作用是一样的。

#### INSTR()方法

语法: INSTR(str,substr)

返回字符串str中第一次出现子字符串substr的位置。INSTR()与LOCATE()的双参数形式相同，只是参数的顺序相反。

### MySQL基于regexp、rlike的正则匹配查询

MySQL中的regexp和rlike关键字属于同义词，功能相同。本文以regexp为准。

REGEXP 不支持通配符"%、\_"，支持正则匹配规则，是一种更细力度且优雅的匹配方式，包含的参数类型如表6-1所示。

表 6‑2 参数类型

|  |  |
| --- | --- |
| 参数类型 | 作用 |
| (^) | 匹配字符串的开始位置，如“^a”表示以字母a开头的字符串。 |
| ($) | 匹配字符串的结束位置，如“X^”表示以字母X结尾的字符串。 |
| (.) | 这个字符就是英文下的点，它匹配任何一个字符，包括回车、换行等。 |
| (\*) | 星号匹配0个或多个字符，在它之前必须有内容。如：select \* from table where name regexp 'ba\*'（可以命中“baaa”）。 |
| (+) | 加号匹配1个或多个字符，在它之前也必须有内容。加号跟星号的用法类似，只是星号允许出现0次，加号则必须至少出现一次。 |
| (?) | 问号匹配0次或1次。 |
| {n} | 匹配指定n个。 |
| {n,} | 匹配不少于n个。 |
| {n,m} | 匹配n-m个。 |

#### regexp中的 OR:|

功能：可以搜索多个字符串之一，相当于 or。

#### REGEXP中的正则匹配:[]

功能：匹配[]符号中几个字符之一，支持解析正则表达式。

#### 字符类匹配(posix)

mysql中有一些特殊含义的符号，可以代表不同类型的匹配，如表6-3所示。

表 6‑3 字符类匹配类型

|  |  |
| --- | --- |
| 字符类 | 作用 |
| [:alnum:] | 匹配字面和数字字符 (等同于[A~Za~z0~9]) 。 |
| [:alpha:] | 匹配字母字符。(等同于[A~Za~z])。 |
| [:blank:] | 匹配空格或制表符(同[\\\t])。 |
| [:cntrl:] | 匹配控制字符(ASCII0到37和127)。 |
| [:digit:] | 匹配十进制数字。(等同于[0-9])。 |
| [:graph:] | 匹配ASCII码值范围33~126的字符。与[:print:]相似，但不包括空格字符。 |
| [:print:] | 任何可打印字符。 |
| [:lower:] | 匹配小写字母，等同于[a-z]。 |
| [:upper:] | 匹配大写字母，等同于[A-Z]。 |
| [:space:] | 匹配空白字符（同[\\f\\n\\r\\t\\v]）。 |
| [:xdigit:] | 匹配十六进制数字。等同于[0-9A-Fa-f]。 |

这种字符类需要主要的外层要加一层[]。

#### [:<:]和[:>:]

上面的字符类中有两个比较特殊的，这两个是关于位置的，[:<:]匹配词的开始,[:>:]匹配词的结束，它们和 ^、$ 不同。

后者是匹配整个整体的开头和结束，而前者是匹配一个单词的开始和结束。

## 分组函数

功能：用作统计使用，又称聚合函数或统计函数或组函数。

分类：sum求和、avg求平均、max最大值、min最小值、count计算个数。

### 简单使用

语法：sum（1个参数）

例：

SELECT SUM(salary) FROM employees;

SELECT ROUND(AVG(salary),2),MAX(salary),MIN(salary) FROM employees;

### 特点

#### 参数支持类型

sum、avg：数值型

min、max、count：任何类型

#### 是否忽略NULL值

sum、avg、min、max：忽略NULL值

count：计算非空值的个数，忽略NULL值

### 可以和distinct搭配使用实现去重的运算

例：

select sum(distinct salary),sum(salary) from employees;

SELECT COUNT(DISTINCT salary),COUNT(salary) FROM employees;

### count函数的其他介绍

#### 统计总行数

SELECT COUNT(\*) FROM employees; #任何一行中只要有一个不是NULL就加上

#### 额外加上列全1，统计1的个数，即为统计总行数（也可以是其他数值或字符）

SELECT COUNT(1) FROM employees;

效率问题：

MYISAM存储引擎下，count（\*）效率最高

INNODB存储引擎下，count（\*）和count（1）效率差不多，比count（字段）效率高一些

一般使用count（\*）来统计行数

### 和分组函数一同查询的字段有限制

和分组函数一同查询的字段要求是group by后的字段

如：SELECT AVG(salary),employee\_id FROM employees; #无意义

### datediff(日期1，日期2) 计算两个日期相差的天数

SELECT DATEDIFF(MAX(hiredate),MIN(hiredate)) FROM employees;

## 分组查询

分组查询是对数据按照某个或多个字段进行分组，在MYSQL中使用GROUP BY关键字对数据进行分组

GROUP BY关键字可以将查询结果按照某个字段或多个字段进行分组。字段中值相等的为一组

分组的核心是：在查询SQL中指定分组的列名，然后根据该列的值进行分组，值相等的为一组

分组查询的基本的语法格式如下：

GROUP BY 字段名 [HAVING 条件表达式]

参数：

1、字段名：是指按照该字段的值进行分组(分组是所依据的列名称)

2、HAVING条件表达式：用来限制分组后的显示，符合条件表达式的结果将被显示

### 创建分组

1、对数据进行分组，一般有两种使用场景：

⑴单独使用GROUP BY关键字，

⑵将GROUP BY关键字与聚合函数一起使用(常用)

2、对于GROUP BY子句的使用，需要注意以下几点：

⑴GROUP BY子句可以包含任意数目的列，使其可以对分组进行嵌套，为数据分组提供更加细致的控制

⑵GROUP BY子句列出的每个列都必须是检索列或有效的表达式，但不能是聚合函数。若在SELECT语句中使用表达式，则必须在GROUP BY子句中指定相同的表达式

若用于分组的列中包含有NULL值，则NULL将作为一个单独的分组返回；若该列中存在多个NULL值，则将这些NULL值所在的行分为一组

#### GROUP BY单独使用

单独使用 GROUP BY关键字时，查询结果会只显示每个分组的第一条记录。

#### GROUP BY与聚合函数

1、GROUP BY关键字通常与集合函数一起使用。集合函数包括COUNT()函数、SUM()函数、AVG()函数、MAX()函数和MIN()函数

2、如果GROUP BY不与聚合函数一起使用，那么查询结果就是字段取值的分组情况

⑴字段中取值相同的记录为一组，但是只显示该组的第一条记录(跟前面GROUP BY单独使用一样)

3、常用的聚合函数有：

⑴COUNT()函数：用于统计记录的条数

⑵SUM()函数：用于计算字段的值的总和

⑶AVG()函数：用于计算字段的值的平均值

⑷MAX()函数：用于查询字段的最大值

⑸MIN()函数：用于查询字段的最小值

#### GROUP BY与GROUP\_CONCAT

GROUP BY关键字可以和GROUP\_CONCAT()函数一起使用。GROUP\_CONCAT()函数会把每个分组的字段值都显示出来

#### 多字段分组

1、使用GROUP BY可以对多个字段进行分组，GROUP BY关键字后面跟需要分组的字段

2、MYSQL根据多字段的值来进行层次分组，分组层次从左到右

⑴即先按第一个字段分组，然后在第一个字段值相同的记录中，再根据第二个字段的值进行分组...一次类推。

### 使用HAVING过滤分组

1、GROUP BY可以和HAVING一起限定显示记录所需满足的条件：只有满足条件的分组才会被显示

2、HAVING关键字是对分组结果进行过滤。WHERE关键字是对表数据进行过滤

⑴两者同时存在时：肯定是先计算WHERE，WHERE排除的记录肯定是不会出现在分组内的。

### GROUP BY与WITH ROLLUP

WITH POLLUP关键字用来在所有记录的最后加上一条记录，这条记录是上面所有记录的总和，即统计记录数量。

### GROUP BY与ORDER BY

1、某些情况下需要对分组进行排序

⑴一般情况下ORDER BY是用来对查询结果进行排序的。当其与GROUP BY一起使用时，可以对分组结果进行排序

2、需要注意：当使用ROLLUP时，就不能同时使用ORDER BY子句进行结果排序了

⑴即：ROLLUP和ORDER BY是互相排斥的

## Union用法

关键字：union /union all

语句：(select colum1,colum2...columN from tableA ) union (select colum1,colum2...columN from tableB )或 (select colum1,colum2...columN from tableA ) union all (select colum1,colum2...columN from tableB )；

union语句注意事项：

1.通过union连接的SQL它们分别单独取出的列数必须相同；

2.不要求合并的表列名称相同时，以第一个sql 表列名为准；

3.使用union 时，完全相等的行，将会被合并，由于合并比较耗时，一般不直接使用 union 进行合并，而是通常采用union all 进行合并；

4.被union 连接的sql 子句，单个子句中不用写order by ，因为不会有排序的效果。但可以对最终的结果集进行排序；

# Spring事务

## 定义

事务(Transaction)是指业务逻辑上对数据库进行的一系列持久化操作，要么全部成功，要不全部失败

## 特性和实现

### 事物的四个基本性质(ACID)

#### 原子性(Atomicity)

事务的原子性指事务是事务是一个不可分割的工作单位这组操作要么全部执行，要么全部不发生。就拿银行转账来说，，原子性指的是从A账户扣款和转账到B账户两件事一起发生，不能只发生其中一样。

#### 一致性(Consistency)

在事务开始的前后，被操作的数据的完整性都处于一致性的状态。

还是银行转账的例子，一致性指的是在转账的前后A和B两个账户的总金额是不变的，始终保持一致。

#### 隔离性(Isolation)

多个事务并发时，事务之间是隔离的，每个事物都有自己的完整数据空间，不会影响到其他事务的运行效果。

数据库事务的隔离级别有4种，由低到高分别为Read uncommitted、Read committed、Repeatable read、Serializable。而且，在事务的并发操作中可能会出现脏读，不可重复读，幻读。值得一提的是：大多数数据库默认的事务隔离级别是Read committed，比如Sql Server , Oracle。Mysql的默认隔离级别是Repeatable read。下面通过实事例一一阐述他们的概念与联系。

##### 读未提交(Read uncommitted)

读未提交，顾名思义，就是一个事务可以读取另一个未提交事务的数据。

事例：老板要给程序员发工资，程序员的工资是3.6万/月。但是发工资时老板不小心按错了数字，按成3.9万/月，该钱已经打到程序员的户口，但是事务还没有提交，就在这时，程序员去查看自己这个月的工资，发现比往常多了3千元，以为涨工资了非常高兴。但是老板及时发现了不对，马上回滚差点就提交了的事务，将数字改成3.6万再提交。

分析：实际程序员这个月的工资还是3.6万，但是程序员看到的是3.9万。他看到的是老板还没提交事务时的数据。这就是脏读。

那怎么解决脏读呢？Read committed！读提交，能解决脏读问题。

##### 读提交(Read ucommitted)

读提交，顾名思义，就是一个事务要等另一个事务提交后才能读取数据。

事例：程序员拿着信用卡去享受生活（卡里当然是只有3.6万），当他埋单时（程序员事务开启），收费系统事先检测到他的卡里有3.6万，就在这个时候！！程序员的妻子要把钱全部转出充当家用，并提交。当收费系统准备扣款时，再检测卡里的金额，发现已经没钱了（第二次检测金额当然要等待妻子转出金额事务提交完）。程序员就会很郁闷，明明卡里是有钱的…

分析：这就是读提交，若有事务对数据进行更新（UPDATE）操作时，读操作事务要等待这个更新操作事务提交后才能读取数据，可以解决脏读问题。但在这个事例中，出现了一个事务范围内两个相同的查询却返回了不同数据，这就是不可重复读。

那怎么解决可能的不可重复读问题？Repeatable read ！

##### 重复读(Repeatable read)

重复读，就是在开始读取数据（事务开启）时，不再允许修改操作

事例：程序员拿着信用卡去享受生活（卡里当然是只有3.6万），当他埋单时（事务开启，不允许其他事务的UPDATE修改操作），收费系统事先检测到他的卡里有3.6万。这个时候他的妻子不能转出金额了。接下来收费系统就可以扣款了。

分析：重复读可以解决不可重复读问题。写到这里，应该明白的一点就是，不可重复读对应的是修改，即UPDATE操作。但是可能还会有幻读问题。因为幻读问题对应的是插入INSERT操作，而不是UPDATE操作。

什么时候会出现幻读？

事例：程序员某一天去消费，花了2千元，然后他的妻子去查看他今天的消费记录（全表扫描FTS，妻子事务开启），看到确实是花了2千元，就在这个时候，程序员花了1万买了一部电脑，即新增INSERT了一条消费记录，并提交。当妻子打印程序员的消费记录清单时（妻子事务提交），发现花了1.2万元，似乎出现了幻觉，这就是幻读。

那怎么解决幻读问题？Serializable！

##### 序列化(Serializable)

Serializable 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

#### 持久性(Durability)

事务提交后，该事务该事务对数据所做的修改将持久地保存在数据库，并不会回滚。

事物的原子性、一致性和持久性是通过数据库的redo/undo日志文件实现的。redo log处理系统故障，undo log处理事务回滚。如果在事务提交之后出现数据库崩溃（断电）的情况，在恢复供电时，数据库会根据重写日志对数据进行前滚。

### 事务的实现

在Spring中，事务有两种实现方式，分别是编程式事务和声明式事务。

#### 编程式事务

编程式事务管理使用Transaction Template或者直接使用底层的Platform TransactionManager。对于编程式事务，Spring推荐使用Transaction Template。

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

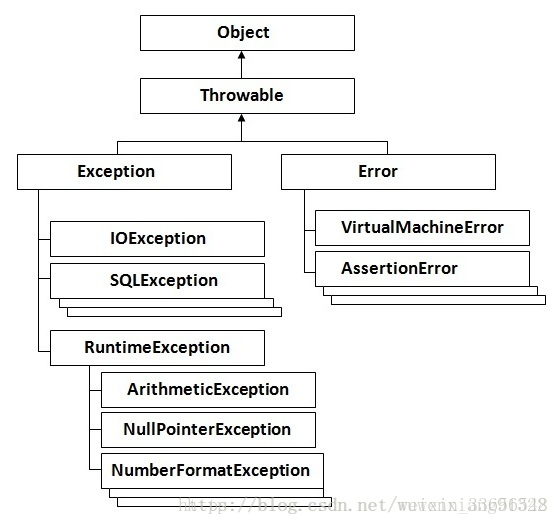
#### 声明式事务

声明式事务基于AOP，其本质是对方法前后进行拦截，在方法开始前创建或加入一个事务，再根据目标配方法执行的结果决定提交还是回滚事务。只需在类、方法上加上@Transactional注解就可以使用事务，没有入侵性，简单粗暴。

## 声明式事务的使用技巧

1、@Transactional可以用作于接口、接口方法、类、类方法上，当作用到类时，该类下所有public方法都将具有该类型的事务属性，同时，也可以在方法级别使用该注解来覆盖类级别的定义。Spring的建议是在具体的实现类和类方法使用@Transactional注解，而不是使用在接口上。因为注解不能继承，不能被基于接口的代理类所识别，注解失效。

2、声明式事务管理默认只对非检查型异常unchecked Exception进行回滚，也就是对RuntimeException异常以及它的子类进行回滚操作，如图2-1所示。

图 3‑1 声明式事务管理

如果需要让checked Exception也进行回滚，需加上@Transactional(rollbackFor=Exception.class)，如果需要让unchecked Exception不进行回滚，需加上@Transactional(not RollbackFor=Exception.class)

3、在Springboot使用声明式事务需要在Application启动类加入@EnableTransactionManagerment注解，相当于Spring的自动扫描。

## 声明式事务的常用配置

声明式事务的常用配置如表4-1所示。

表 4‑1 声明式事务的常用配置

|  |  |
| --- | --- |
| 参数名称 | 功能描述 |
| readOnly | 该属性用于设置当前事务是否为只读事务，设置为true表示只读，false则表示可读写，默认值为false。例如：@Transactional(readOnly=true) |
| rollbackFor | 该属性用于设置需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，则进行事务回滚。例如：指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class}) |
| rollbackForClassName | 该属性用于设置需要进行回滚的异常类名称数组，当方法中抛出指定异常名称数组中的异常时，则进行事务回滚。例如：指定单一异常类名称@Transactional(rollbackForClassName=”RuntimeException”)指定多个异常类名称：@Transactional(rollbackForClassName={“RuntimeException”,”Exception”}) |
| noRollbackFor | 该属性用于设置不需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，不进行事务回滚。例如：指定单一异常类：@Transactional(noRollbackFor=RuntimeException.class)指定多个异常类：@Transactional(noRollbackFor={RuntimeException.class, Exception.class}) |
| noRollbackForClassName | 该属性用于设置不需要进行回滚的异常类名称数组，当方法中抛出指定异常名称数组中的异常时，不进行事务回滚。例如：指定单一异常类名称：@Transactional(noRollbackForClassName=”RuntimeException”)指定多个异常类名称：@Transactional(noRollbackForClassName={“RuntimeException”,”Exception”}) |
| propagation | 该属性用于设置事务的传播行为。例如：@Transactional(propagation=Propagation.NOT\_SUPPORTED,readOnly=true) |
| isolation | 该属性用于设置底层数据库的事务隔离级别，事务隔离级别用于处理多事务并发的情况，通常使用数据库的默认隔离级别即可，基本不需要进行设置 |
| timeout | 该属性用于设置事务的超时秒数，默认值为-1表示永不超时 |

## 声明式事务的隔离级别

隔离级别是指若干个并发的事务之间的隔离程度。TransactionDefinition 接口中定义了五个表示隔离级别的常量：

TransactionDefinition.ISOLATION\_DEFAULT：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是TransactionDefinition.ISOLATION\_READ\_COMMITTED。

TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读，不可重复读和幻读，因此很少使用该隔离级别。比如PostgreSQL实际上并没有此级别。

TransactionDefinition.ISOLATION\_READ\_COMMITTED：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值。

TransactionDefinition.ISOLATION\_REPEATABLE\_READ：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。该级别可以防止脏读和不可重复读。

TransactionDefinition.ISOLATION\_SERIALIZABLE：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

## **声明式事务的传播行为**

所谓事务的传播行为是指，如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。在TransactionDefinition定义中包括了如下几个表示传播行为的常量：

TransactionDefinition.PROPAGATION\_REQUIRED：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。

TransactionDefinition.PROPAGATION\_REQUIRES\_NEW：创建一个新的事务，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_SUPPORTS：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。

TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED：以非事务方式运行，如果当前存在事务，则把当前事务挂起。

TransactionDefinition.PROPAGATION\_NEVER：以非事务方式运行，如果当前存在事务，则抛出异常。

TransactionDefinition.PROPAGATION\_MANDATORY：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。

TransactionDefinition.PROPAGATION\_NESTED：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。

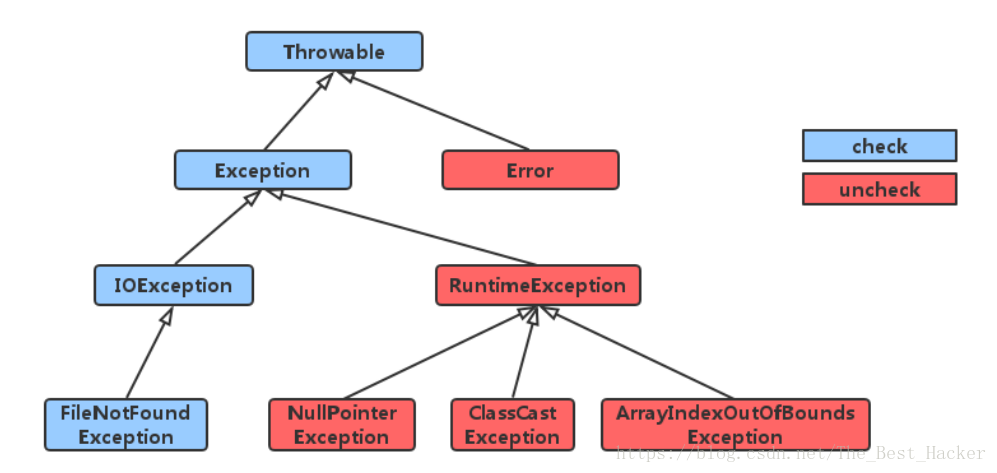
# Exception异常

## 异常的划分

按继承体系划分：分为两个子类，一个是错误(Error)，另一个是异常(Exception)。

错误属于比较严重的，异常属于可以恢复的错误。如图1-1所示所示。

按检查和未检查划分：分为检查异常（蓝色方框）和未检查异常（红素方框），如图1-1所示。

图 1‑1 异常的划分

区别：检查异常需要配合try-catch或throws关键字一起使用，否则会引起编译错误。在编译时，如果没有处理（没有throw,也没有try），编译失败。该异常被标识，代表着可以被处理。未检查异常用不用关键字都可以。在编译时不需要处理，编译器不检查。该异常发生时，建议不处理，让程序停止，需要对代码进行修正。

## 异常的语法

1、积极处理

try {

可能引发异常的语句;

} catch (Exception) {

捕捉到的异常信息;

} finally {无论上面的catch是否能捕捉到真正的异常，都会执行的语句（这步操作根据需求添加）

一定会执行的语句;

}

简单的理解：try中的语句就是可能会引起异常，而异常会导致程序终止，如果catch可以捕捉到上述的异常，那么就可以使程序正常运行。不管是否捕捉到对应的异常，finally中的语句都是要执行的。

特殊的4种情况，finally语句块不会被执行：

1. 在finally语句块中发生了异常。
2. 在前面的代码中使用了System.exit()退出程序。
3. 程序所有的线程死亡。
4. 关闭CPU。

当然一个try后面可以跟上多个catch语句这些catch中的内容是并列的或者继承关系。如果是继承关系，需注意父类不能写在子类的前面

2、消极处理

给出throws。

如果语句中存在异常，没有try-catch关键字，那么异常发生时，会将异常传递给方法的调用者，这时候需要在方法声明上抛出异常。如果方法中出现的是检查异常，则必须给出throws，如果是未检查异常，则可以给，也可以不给。

举例：

public class Demo1 {  
 /\*  
 处理方式一：抛出异常给jvm虚拟机，让虚拟机处理这个异常（虚拟机的处理方法是调用printStackTrace方法，打印异常信息）  
 public static void main(String[] args) throws Exception {  
 int x=div(12,0);  
 System.out.println(x);  
   
   
 }\*/  
 public static void main(String[] args) {  
 int x= 0;  
 /\*处理方式二：用try...catch捕获异常\*/  
 try {  
 x = *div*(12,0);  
 System.*out*.println(x);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 }  
 public static int div(int a,int b) throws Exception{  
 return a/b;  
 }  
}

## 异常的常用方法

getMessage()：返回的是异常的信息；

getStackTrace()：打印异常的跟踪信息，打印堆栈中的异常出现位置异常（名称、信息）。

在JVM默认的异常处理机制，就是调用printStackTrace方法，打印异常的堆栈跟踪信息

创建异常类：

1. class MyException extends Excepion{//检查异常}
2. class MyException extends RuntimeException{//未检查异常}

## 异常处理的原则

1. 函数内容如果抛出需要检测的异常，那么函数上必须声明，否则必须在函数内用try-catch捕捉，否则编译失败。
2. 如果调用到了声明异常的函数，要么try-catch，要么throws，否则编译失败。
3. 什么时候用catch，什么时候用throws？

功能内容可以解决，用catch；解决不了，用throws告诉调用者，由调用者解决。

1. 一个功能如果抛出了多个异常，那么调用时，必须有对应的多个catch进行针对性地处理。

## 自定义异常

在项目中会出现特有的问题，而这些问题并未被Java所描述并封装为对象，所以对于这些特有的问题可以按照Java封装的思想，将特有的问题进行自定义的异常处理。

如何定义异常信息？

因为父类中已经将异常信息的操作都完成了，所以子类只需要在构造时，将异常信息通过super()传递给父类，就可以直接地通过getMessage()方法获取自定义得到异常信息。

自定义异常类必须继承Exception，原因：因为异常类和异常对象都被抛出，他们都有可抛性。只有Throwable体系中的类和对象才可以被throws和throw操作。

步骤：

创建自定义异常类

在方法中通过throw关键字抛出异常

如果在当前抛出的异常方法中处理异常，可以使用try-catch语句块捕获并处理，否则在方法的声明处通过throws关键字指明要抛出给方法调用者的异常，继续下一操作

在出现异常方法的调用者中捕获并处理异常。

举例：

public class MyException extends Exception {  
 private int value;  
 MyException(){  
 super();  
 }  
 MyException(String msg,int value){  
 super(msg);  
 this.value=value;  
 }  
 public int getValue(){  
 return value;  
 }  
}  
  
public class Demo2 {  
 public int div(int a,int b) throws MyException {  
 if(b<0){  
 throw new MyException("出现了除数是负数的情况",b);  
 }  
 return a/b;  
 }  
  
 public static void main(String[] args) {  
 Demo2 d=new Demo2();  
 try {  
 int x=d.div(12,-4);  
 } catch (MyException e) {  
 System.*out*.println(e.toString());  
 System.*out*.println("错误的负数是："+e.getValue());  
 }  
 }  
}

## Throw、Throws

如果某个方法可能发生异常，但又不想立刻处理，则可以使用throws或者throw关键字在方法中抛出异常，但最终要有能够处理该异常的代码。

1. throws通常被应用在声明方法时。多个异常可使用逗号分隔。
2. RuntimeException、或是他们的子类，可以不使用throws关键字抛出异常，编译仍能通过，但在运行时会被系统抛出。
3. throw关键字用于方法体中，并且抛出一个异常对象。程序在执行到throw语句时立即终止，后面语句不执行。通过throw抛出异常后，若想在上一级代码中捕获并处理异常，则需要在抛出的异常方法中使用throws在方法声明中指明要抛出的异常；如果要捕捉throw抛出的异常，必须使用try-catch语句块。
4. throw通常用来抛出用户自定义异常。

## Try-Catch-Finally

格式一：

try-catch-finally

格式二：

try-finally

格式三：

try-catch

举例：

*/\*\*  
 \* try{  
 \*  
 \* }catch(Exception e){  
 \*  
 \* }finally{  
 \*  
 \* }  
 \*/*public class Demo3 {  
 //编译可以通过  
 public void method1(){  
 try {  
 throw new Exception();  
 }catch (Exception e){  
  
 }  
 }  
  
 //编译不能通过，异常未被声明  
 public void method2(){  
 throw new Exception();  
 }  
  
 //编译可以通过  
 public void method3() throws Exception{  
 try {  
 throw new Exception();  
 }catch (Exception e){  
 try {  
 throw e;  
 }catch (){  
  
 }  
 }  
 }  
  
 //编译不能通过  
 public void method4(){  
 try{  
 throw new Exception();  
 }finally{  
  
 }  
 }  
 //结论：没有catch块，异常就不能被处理，只能声明（抛出去）。否则编译就不能通过。  
  
}

## RuntimeException运行时异常

RuntimeException及其子类，如果在函数内抛出异常，函数上可以不用声明，编译一样会通过。如果函数上声明了该异常，调用者可以不进行处理，编译同样通过。之所以不用在函数声明是因为不需要让调用者处理，当该异常发生时，希望程序停止，因为在运行时出现了无法继续运算的情况，希望停止程序，对代码进行修正。

## 异常在子父类中的体现

1. 子类在覆盖父类时，如果父类的方法抛出异常，那么子类的覆盖方法只能抛出父类的异常或者该异常的子类。
2. 如果父类抛出异常多个，那么子类在覆盖方法，只能抛出父类异常的子集。
3. 如果子类发生父类没有的异常，那么子类就只能在内部处理该异常，不能抛出。
4. 如果父类或者接口的方法中没有异常抛出，那么子类在覆盖方法时也不可以抛出异常。如果子类方法发生异常，就必须要进行try-catch处理，不能抛出异常

## finally与return的执行顺序

*/\*\*  
 \* 异常中的return  
 \*/*public class Demo5 {  
 public static void main(String[] args) {  
 /\*int x1=method1();  
 System.out.println(x1);  
 输出：  
 finally  
 1  
 \*/  
 /\*int x2 = method2();  
 System.out.println(x2);  
 输出：  
 finally  
 2\*/  
   
 /\*int x3 = method3();  
 System.out.println(x3);  
 输出：  
 finally  
 0\*/  
   
 /\*int x4 = method4();  
 System.out.println(x4);  
 输出：  
 finally  
 0  
 结论：如果try...catch中的return返回的是一个变量，且函数是从其中一个返回时，  
 后面finally中语句即使有对返回值的操作，也不影响返回值的值。  
 \*/  
 }  
 public static int method1(){  
 try{  
  
 return 1;  
 }finally{  
 System.*out*.println("finally");  
 }  
 }  
  
 public static int method2(){  
 try{  
 int a=8/0;  
 return 1;  
 }catch (Exception e){  
 return 2;  
 }finally {  
 System.*out*.println("finally");  
 }  
 }  
  
 public static int method3(){  
 try{  
 int a=8/0;  
 return 1;  
 }catch (Exception e){  
 return 2;  
 }finally {  
 System.*out*.println("finally");  
 return 0;  
 }  
 }  
  
 public static int method4(){  
 int result=0;  
 try{  
 return result;  
 }finally {  
 System.*out*.println("finally");  
 result= 1;  
 }  
 }  
  
 //错误：缺少返回语句(如果不抛出异常，那么这个方法中就没有返回值)  
 public static int method5(){  
 int result=0;  
 try{  
 int re=10/0;  
 }catch (Exception e){  
 result=10;  
 return result;  
 }finally {  
 System.*out*.println("finally");  
 }  
  
 }  
}

## 异常的注意事项

1. 方法重写——子类方法不能比父类抛出更多的异常，子类重写的方法可以不抛出。
2. 将检查异常包装为未检查异常（建议）。
3. 不要吞异常——即catch块后面什么也不写，这样即使将来产生异常，也不知道是什么异常。
4. 如果try、catch、finally中都有return以finally中的为准，如果try中有return，即使finally对return后的值做了改动，也不会影响到返回后的结果。
5. 在调用方法时，方法的类型不为void，则在try块和catch块中，都需要加入return（前提：不存在finally中的return）。
6. 当调用的方法出现检查异常或自定义异常，则在调用时，如果使用try-catch（积极处理），则可以正常使用，不用一级一级地向上抛出。如果为未检查异常，则不需要加以处理。
7. 如果catch捕获的异常不完整，则编译不会通过。
8. 异常产生后，如果不做任何处理，程序就会被终止。
9. 一个方法被覆盖时，覆盖他的方法必须抛出相同的异常或异常的子类。如果父类抛出多个子类，则覆盖方法必须抛出那些异常的子类，不能抛出新的异常
10. 与有返回值的方法连用时要注意：
11. public class Exception5 {  
     public static int test(){  
     try {  
     Scanner scanner = new Scanner(System.*in*);  
     int n = scanner.nextInt();  
     int r = 1/n;  
     return r;  
     } catch (Exception e ) {  
     System.*out*.println(e.getMessage());  
     return -1; // 解决方法1: 在catch也写一个return返回结果  
    // throw e; // 解决方法2: 把异常重新抛出  
     }  
     }  
      
     public static void main(String[] args) {  
     int r = *test*();  
     System.*out*.println(r);  
     }  
    }

## 常见的异常

数组下标越界--->ArrayIndexOutOfBoundsException

访问null的对象的方法或属性(空指针异常)--->NullPointerException

类型转换异常--->ClassCastException

未找到相应类异常--->ClassNotFoundException

遍历集合的同时remove--->ConcurrentModificationException（并发修改异常）

算术异常（除零）--->ArithmeticException

在堆内存耗尽时--->OutOfMemoryError（new的太多了）

在栈内存耗尽时--->StackOverflowError（应用程序递归太深而发生堆栈溢出）

数组中包含不兼容的值抛出的异常--->ArrayStoreException

操作数据库异常类--->SQLException

字段未找到异常--->NoSuchFieldException

方法未找到抛出的异常--->NoSuchMethodException

字符串转化为数字异常--->NumberFormatException

数组元素为负数抛出的异常--->NegativeArraySizeException

字符串索引超出范围--->StringIndexOutBoundsException

输入输出异常--->IOExceotion

不允许访问某类异常--->IllegalAccessException

文件已结束异常--->EOFException

文件未找到异常--->FileNotFoundException

# @Validated注解

## 简述

@Validation是一套帮助我们继续对传输的参数进行数据校验的注解，通过配置Validation可以很轻松的完成对数据的约束。

@Validated作用在类、方法和参数上

@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface Validated {  
 Class<?>[] value() default {};  
}

所有参数注释含义见图1-1。

图 1‑1 参数注解含义

# @RequestMapping

## @Controller和@RestController

@Controller 用来响应页面，表示当前的类为控制器。

@RestController 是@ResponseBody和@Controller的结合。表明当前类是控制器且返回的是一组数据，不是页面。

@Controller 将当前修饰的类注入SpringBoot IOC容器，使得从该类所在的项目跑起来的过程中，这个类就被实例化。当然也有语义化的作用，即代表该类是充当Controller的作用。

@responseBody注解的作用是将controller的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到response对象的body区，通常用来返回JSON数据或者是XML数据。

## @Autowired

这个注解的作用是将其他的类，接口引入，类似于之前的类的初始化等，用这个注解，类中或接口的方法就可以直接调用了。

## @RequestMapping

### @RequestMapping

@RequestMapping注解的主要用途是将Web请求与请求处理类中的方法进行映射。Spring MVC和Spring WebFlux都通过RquestMappingHandlerMapping和RequestMappingHndlerAdapter两个类来提供对@RequestMapping注解的支持。

@RequestMapping注解对请求处理类中的请求处理方法进行标注；@RequestMapping注解拥有以下的六个配置属性：

1. value:映射的请求URL或者其别名。
2. method:兼容HTTP的方法名。
3. params:根据HTTP参数的存在、缺省或值对请求进行过滤。
4. header:根据HTTP Header的存在、缺省或值对请求进行过滤。
5. consume:设定在HTTP请求正文中允许使用的媒体类型。
6. product:在HTTP响应体中允许使用的媒体类型。

提示：在使用@RequestMapping之前，请求处理类还需要使用@Controller或@RestController进行标记。

### @GetMapping

@GetMapping注解用于处理HTTP GET请求，并将请求映射到具体的处理方法中。具体来说，@GetMapping是一个组合注解，它相当于是@RequestMapping(method=RequestMethod.GET)的快捷方式。作用：对应查询，表明是一个查询URL映射。

### @PostMapping

@PostMapping注解用于处理HTTP POST请求，并将请求映射到具体的处理方法中。@PostMapping与@GetMapping一样，也是一个组合注解，它相当于是@RequestMapping(method=HttpMethod.POST)的快捷方式。作用：对应增加，表明是一个增加URL映射。

### @PutMapping和@PatchMapping

@PutMapping注解用于处理HTTP PUT请求，并将请求映射到具体的处理方法中，@PutMapping是一个组合注解，相当于是@RequestMapping(method=HttpMethod.PUT)的快捷方式。

@PatchMapping注解用于处理HTTP PATCH请求，并将请求映射到对应的处理方法中。@PatchMapping相当于是@RequestMapping(method=HttpMethod.PATCH)的快捷方式。

作用：对应更新，表明是一个更新URL映射。

两者都是更新，@PutMapping为全局更新，@PatchMapping是对put方式的一种补充，put是对整体的更新，patch是对局部的更新。

### @DeleteMapping

@DeleteMapping注解用于处理HTTP DELETE请求，并将请求映射到删除方法中。@DeleteMapping是一个组合注解，它相当于是@RequestMapping(method=HttpMethod.DELETE)的快捷方式。作用：对应删除，表明是一个删除URL映射。

## @ResponseBody

@ResponseBody会自动将控制器中方法的返回值写入到HTTP响应中。特别的，@ResponseBody注解只能用在被@Controller注解标记的类中。如果在被@RestController标记的类中，则方法不需要使用@ResponseBody注解进行标注。

## @PathVariable

@PathVariable注解是将方法中的参数绑定到请求URI中的模板变量上。可以通过@RequestMapping注解来指定URI的模板变量，然后使用@PathVariable注解将方法中的参数绑定到模板变量上。特别地，@PathVariable注解允许我们使用value或name属性来给参数取一个别名。

模板变量名需要使用“{ }”进行包裹，如果方法的参数名与URI模板变量名一致，则在@PathVariable中就可以省略别名的定义。

提示：如果参数是一个非必须的，可选的项，则可以在@PathVariable中设置require = false。

## @RequestParam

@RequestParam注解用于将方法的参数与Web请求的传递的参数进行绑定。使用@RequestParam可以轻松的访问HTTP请求参数的值。

该注解的其他属性配置与@PathVariable的配置相同，特别的，如果传递的参数为空，还可以通过defaultValue设置一个默认值。

# Redis缓存

Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。

它是基于高性能的Key-Value、并提供多种语言的 API的非关系型数据库。不过与传统数据库不同的是 redis 的数据是存在内存中的，所以存写速度非常快。

## Redis数据存储结构

Redis支持五种数据类型：String（字符串），Hash（哈希），List（列表），Set（集合）及zset(sortedset：有序集合)。

### 数据结构

#### 字符串

String是Redis最基本的数据类型，结构为一个key对应一个value。String类型是二进制安全的，意味着可以包含任何数据，比如jpg图片或者序列化的对象。String类型的最大能存储512M。

#### 哈希

Redis的哈希是field和value之间的映射，即键值对的集合，所以特别适合用于存储对象。Redis 中每个 hash 最多可以存储 232 - 1 键值对（40多亿）。

#### 列表

Redis列表是简单的字符串列表，按照插入顺序排序。支持添加一个元素到列表头部（左边）或者尾部（右边）的操作。一个列表最多可以包含 232- 1 ，即超过40亿个元素。

#### 集合

Redis集合是String类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

集合中最大的成员数为 232- 1 ，即每个集合最多可存储40多亿个成员。

集合的一大特点就是不能有重复元素，如果插入重复元素，Redis会忽略该操作

#### 有序集合

Redis 有序集合和集合一样也是String类型元素的集合，且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。Redis正是通过分数来为集合中的成员进行从小到大的排序。有序集合的成员是唯一的，但分数(score)却可以重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

集合中最大的成员数为 232- 1 ，即每个集合最多可存储40多亿个成员。

需要注意的是，Redis有序集合是默认升序的。

### 存储结构

#### 字符串的存储结构

Redis的所有的key都采用字符串保存，而值可以是字符串，列表，哈希，集合和有序集合对象的其中一种。字符串存储的逻辑类型即REDIS\_STRING，其物理实现（enconding）可以为 REDIS\_ENCODING\_INT、 REDIS\_ENCODING\_EMBSTR或REDIS\_ENCODING\_RAW。首先，如果可以使用REDIS\_ENCODING\_EMBSTR编码，Redis首选REDIS\_ENCODING\_EMBSTR保存；其次，如果可以转换，Redis会尝试将一个字符串转化为Long，保存为REDIS\_ENCODING\_INT，如“26”、“180”等；最后，Redis会保存为REDIS\_ENCODING\_RAW，如“liyuhang”、“Redis”等。

#### 哈希的存储结构

REDIS\_HASH可以有两种encoding方式： REDIS\_ENCODING\_ZIPLIST 和 REDIS\_ENCODING\_HT

Hash表默认的编码格式为REDIS\_ENCODING\_ZIPLIST，在收到来自用户的插入数据的命令时：

（1）调用hashTypeTryConversion函数检查键/值的长度大于配置的hash\_max\_ziplist\_value（默认64）

（2）调用hashTypeSet判断节点数量大于配置的hash\_max\_ziplist\_entries（默认512）

以上任意条件满足则将Hash表的数据结构从REDIS\_ENCODING\_ZIPLIST转为REDIS\_ENCODING\_HT。

#### 列表的存储结构

REDIS\_SET有两种encoding方式，REDIS\_ENCODING\_ZIPLIST和REDIS\_ENCODING\_LINKEDLIST。

列表的默认编码格式为REDIS\_ENCODING\_ZIPLIST，当满足以下条件时，编码格式转换为REDIS\_ENCODING\_LINKEDLIST：

（1）元素大小大于list-max-ziplist-value（默认64）

（2）元素个数大于配置的list-max-ziplist-entries（默认512）

#### 集合的存储结构

REDIS\_SET有两种encoding方式： REDIS\_ENCODING\_INTSET 和 REDIS\_ENCODING\_HT。

集合的元素类型和数量决定了encoding方式，默认采用REDIS\_ENCODING\_INTSET ，当满足以下条件时，转换为REDIS\_ENCODING\_HT：

（1）元素类型不是整数

（2）元素个数超过配置的set-max-intset-entries（默认512）

#### 有序列表的存储结构

REDIS\_ZSET有两种encoding方式： REDIS\_ENCODING\_ZIPLIST（同上）和 REDIS\_ENCODING\_SKIPLIST。

由于有序集合每一个元素包括：<member，score>两个属性，为了保证对member和score都有很好的查询性能，REDIS\_ENCODING\_SKIPLIST同时采用字典和有序集两种数据结构来保存数据元素。字典和有序集通过指针指向同一个数据节点来避免数据冗余。

字典中使用member作为key，score作为value，从而保证在O(1)时间对member的查找跳跃表基于score做排序，从而保证在 O(logN) 时间内完成通过score对memer的查询。

有序集合默认也是采用REDIS\_ENCODING\_ZIPLIST的实现，当满足以下条件时，转换为REDIS\_ENCODING\_SKIPLIST：

（1）数据元素个数超过配置zset\_max\_ziplist\_entries 的值（默认值为 128 ）

（2）新添加元素的 member 的长度大于配置的zset\_max\_ziplist\_value 的值（默认值为 64 ）

## Redis缓存生命周期

### 为什么要设置key生存时间

设置key的生存时间，可以用于以下使用场景：

在登录网站后，将用户session存储在内存，设置一个过期时间，超过这个时间后，用户必须重新登录（例如aws控制台的session过期时间为12个小时）。

使用redis队列时，通常设置一个过期时间，这样即使队列的消费者应用出bug，队列内的消息也不会积压。

### 设置key的生存时间

在set key是指定生存时间

set完key后再指定生存时间。使用expire命令

### 访问key的生存时间

使用TTL key可以访问key的生存时间。

时间复杂度：

O(1)

返回值：

当 key 不存在时，返回 -2 。

当 key 存在但没有设置剩余生存时间时，返回 -1 。

否则，以秒为单位，返回 key 的剩余生存时间。

### 清除生存时间

已经设置生存时间的key，如果想清除掉生存时间，将其变成永久存在的key，可以使用persist命令。

返回值：

1：if the timeout was removed.

0：if key does not exist or does not have an associated timeout.

## Redis缓存穿透

缓存穿透是指用户恶意的发起大量请求去查询一个缓存(Redis)和数据库(DB)中都没有的数据，出于容错考虑从数据库(DB)查不到数据则不写入缓存(Redis)这将导致每个请求都要到数据库(DB)中查询，失去了缓存的意义，从而导致数据库因压力过大挂掉如图3-1所示。

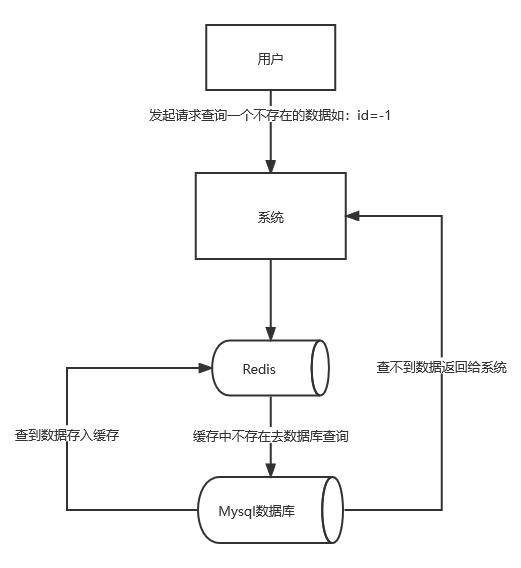


图 3‑1 缓存穿透

解决方案一：对空数据进行缓存

对空数据进行缓存是一个简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

这种办法虽然能阻挡大量穿透请求，但是这个空值不具有任何实际业务性，而且如果大量发送获取不存在数据的穿透请求的话（例如恶意攻击），则会浪费缓存空间，如果这种null值过量的话，还会淘汰掉本身缓存存在的数据，这就会使我们的缓存命中率下降，伪代码如下。

//缓存穿透伪代码  
public object GetProductListNew() {  
 int cacheTime = 30;  
 String cacheKey = "product\_list";  
  
 String cacheValue = CacheHelper.Get(cacheKey);  
 if (cacheValue != null) {  
 return cacheValue;  
 }  
  
 cacheValue = CacheHelper.Get(cacheKey);  
 if (cacheValue != null) {  
 return cacheValue;  
 } else {  
 //数据库查询不到，为空  
 cacheValue = GetProductListFromDB();  
 if (cacheValue == null) {  
 //如果发现为空，设置个默认值，也缓存起来  
 cacheValue = string.Empty;  
 }  
 //product\_list,空值,过期时间  
 CacheHelper.Add(cacheKey, cacheValue, cacheTime);  
 return cacheValue;  
 }  
}

解决方案二：布隆过滤器

这种技术在缓存之前再加一层屏障，里面存储目前数据库中存在的所有key。当业务系统有查询请求的时候，首先去BloomFilter中查询该key是否存在。若不存在，则说明数据库中也不存在该数据，因此缓存都不要查了，直接返回null。若存在，则继续执行后续的流程，先前往缓存中查询，缓存中没有的话再前往数据库中的查询，伪代码如下。

///解决方案二：布隆过滤器  
String get(String key) {  
 String value = redis.get(key);  
 if (value == null) {  
 if(!bloomfilter.mightContain(key)){  
 //不存在则返回  
 return null;  
 }else{  
 //可能存在则查数据库  
 value = db.get(key);  
 redis.set(key, value);  
 }  
 }  
 return value；  
}

## Redis缓存击穿

缓存击穿是指一个Key非常热点，在某些时间点被超高并发地访问，当这个Key在失效的瞬间持续的大并发穿破了缓存(Redis)直接请求到数据库(DB)从而导致数据库出现问题。

### 使用互斥锁

这种解决方案思路比较简单，就是只让一个线程查询数据库，其它线程等待查询数据库的线程执行完毕并且重新将数据加入缓存，其它线程从缓存获取数据就可以了。

如果是单机系统，可以用synchronized或者lock来处理。分布式系统可以用redis的setnx操作。

#### 单机环境

单机环境中实现的原理大概就是当缓存数据过期后，大量请求进来，只有首个线程可以访问数据库其它线程暂停等主线程查询到数据并释放了锁，其它线程就可以直接在缓存中读取数据了。

public String get(key){  
 //从缓存中获取数据  
 String value = redis.get(key);  
 if(value == null){ //缓存中数据不存在  
 if(reenLock.tryLock()){ //获取锁  
 //从数据库获取数据  
 value=db.get(key);  
 //更新缓存数据  
 if(value!=null){  
 redis.set(key, value, expire\_secs);  
 }  
 //释放锁  
 reenLock.unlock();  
 }else{ //获取锁失败  
 //暂停100ms再重新取获取数据  
 Thread.*sleep*(100);  
 value = redis.get(key);  
 }  
 }  
}

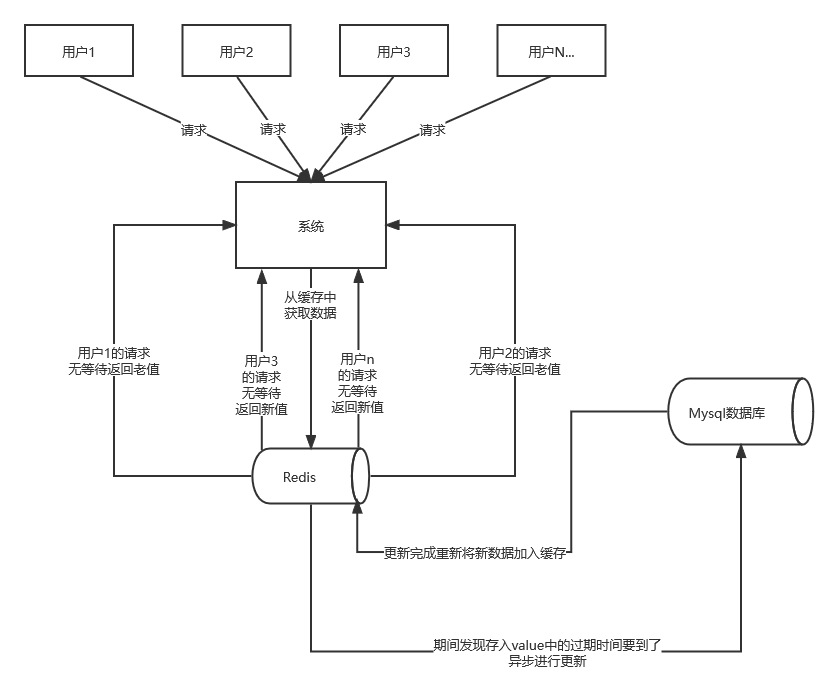
#### 分布式环境

在缓存失效的时候先判断拿出来的值为空，不是立即去查库，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的setnx），去set一个mutex key，当操作返回成功时，再进行查库操作并回设缓存。否则，就重试整个get缓存的方法。

public String get(key){  
 //从缓存中获取数据  
 String value = redis.get(key);  
 if(value == null){ //缓存中数据不存在  
 if(reenLock.tryLock()){ //获取锁  
 //从数据库获取数据  
 value=db.get(key);  
 //更新缓存数据  
 if(value!=null){  
 redis.set(key, value, expire\_secs);  
 }  
 //释放锁  
 reenLock.unlock();  
 }else{ //获取锁失败  
 //暂停100ms再重新取获取数据  
 Thread.*sleep*(100);  
 value = redis.get(key);  
 }  
 }  
}

### 热点数据永不过期

要值得注意的是，这里说到的永不过期并不是将热点数据存在时间设置为无限制。而是将过期时间存在key对应的value里，如果发现要过期了，通过一个后台的异步线程进行缓存的重建，如图4-1所示。

图 4‑1 缓存击穿

从实战看，这种方法对于性能非常友好，唯一不足的就是重建缓存时候，其余线程(非重建缓存的线程)可能访问的是老数据，但是对于一般的互联网功能来说这个还是可以忍受。

public String get(Sting key){  
 V v = redis.get(key);  
 String value = v.getValue();  
 long timeout = v.getTimeout();  
 if (v.timeout <= System.*currentTimeMillis*()){  
 // 异步更新后台异常执行  
 threadPool.execute(new Runnable(){  
 public void run(){  
 String keyMutex = "mutex:" + key;  
 if(redis.setnx(keyMutex, "1")){  
 //3 min timeout to avoid mutex holder crash  
 redis.expire(keyMutex, 3 \* 60);  
 String dbValue = db.get(key);  
 redis.set(key, dbValue);  
 redis.delete(keyMutex);  
 }  
 }  
 });  
 }  
 return value;  
}

### 定时刷新

后台定义一个job(定时任务)专门主动更新缓存数据，比如一个缓存中的数据过期时间是30分钟,那么job每隔29分钟定时刷新数据(将从数据库中查到的数据更新到缓存中)。

这种方案比较容易理解，但会增加系统复杂度。比较适合那些 key 相对固定。缓存粒度较大的业务，key比较分散的则不太适合，实现起来也比较复杂。

## Redis缓存雪崩

缓存雪崩是指缓存服务器重启或缓存(Redis)中的数据在同一时间内大批量过期，因查询数据量太大引起数据库压力过大甚至宕机。

### 加锁排队

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

注意：在高并发场景下尽可能不用！

public object GetProductListNew() {  
 int cacheTime = 30;  
 String cacheKey = "product\_list";  
 String lockKey = cacheKey;  
  
 String cacheValue = CacheHelper.get(cacheKey);  
 if (cacheValue != null) {  
 return cacheValue;  
 } else {  
 synchronized(lockKey) {  
 cacheValue = CacheHelper.get(cacheKey);  
 if (cacheValue != null) {  
 return cacheValue;  
 } else {  
 //这里一般是sql查询数据  
 cacheValue = GetProductListFromDB();  
 CacheHelper.Add(cacheKey, cacheValue, cacheTime);  
 }  
 }  
 return cacheValue;  
 }  
}

### 设置过期时间随机值

避免缓存设置相近的有效期，为有效期增加随机值(1-5分钟)使失效时间均匀分布。这样每一个缓存的过期时间的重复率就会降低，很难引发集体失效的事件。  
redis.set(key, value, random);

### 设置过期标志更新缓存

缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际key的缓存；

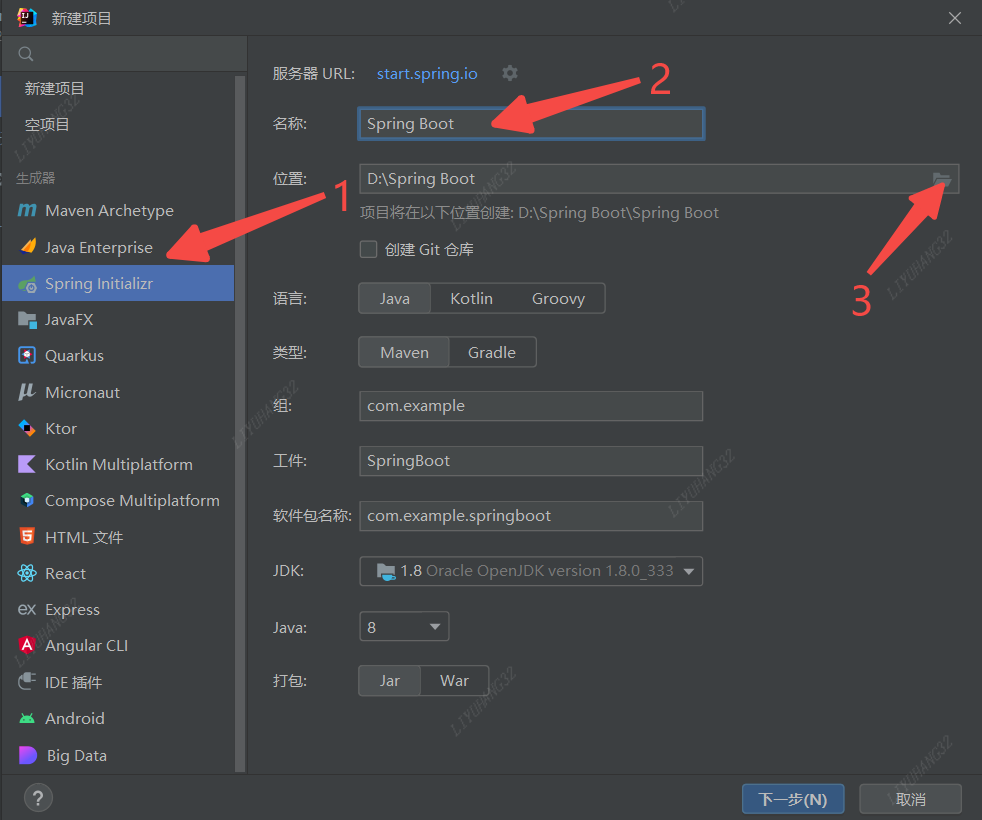
缓存数据：它的过期时间比缓存标记的时间延长1倍，例：标记缓存时间30分钟，数据缓存设置为60分钟。这样，当缓存标记key过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

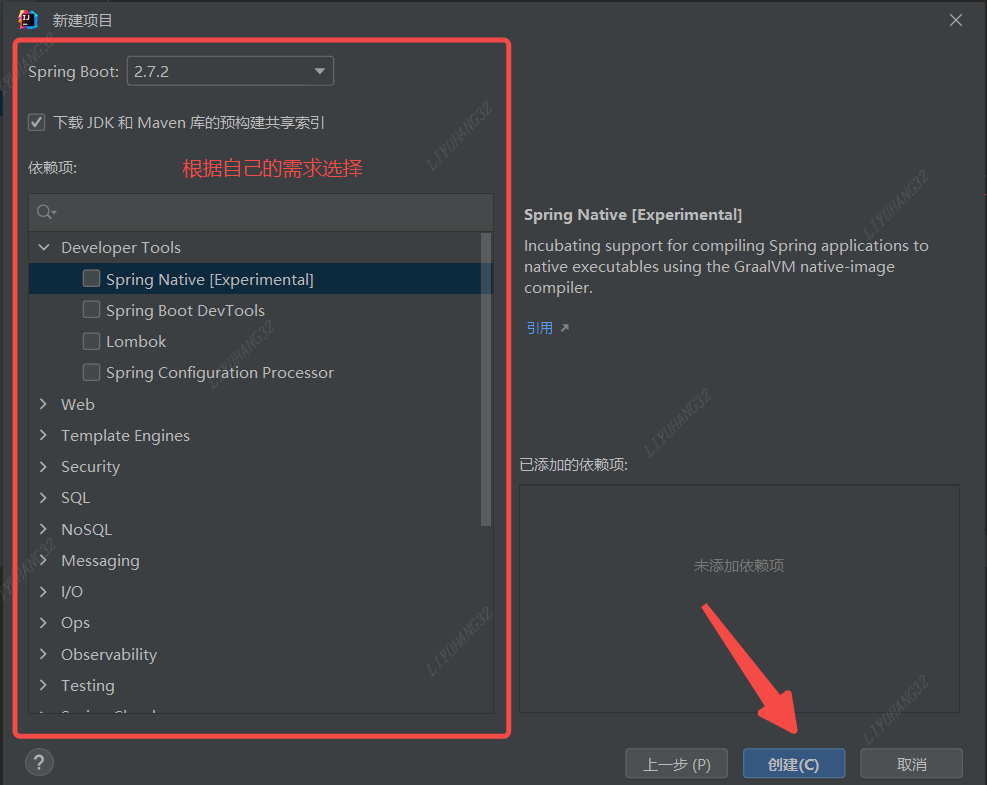
public object GetProductListNew() {  
 int cacheTime = 30;  
 String cacheKey = "product\_list";  
 //缓存标记  
 String cacheSign = cacheKey + "\_sign";  
  
 String sign = CacheHelper.Get(cacheSign);  
 //获取缓存值  
 String cacheValue = CacheHelper.Get(cacheKey);  
 if (sign != null) {  
 return cacheValue; //未过期，直接返回  
 } else {  
 CacheHelper.Add(cacheSign, "1", cacheTime);  
 ThreadPool.QueueUserWorkItem((arg) -> {  
 //这里一般是 sql查询数据  
 cacheValue = GetProductListFromDB();  
 //日期设缓存时间的2倍，用于脏读  
 CacheHelper.Add(cacheKey, cacheValue, cacheTime \* 2);  
 });  
 return cacheValue;  
 }  
}

# SpringBoot入门

## 创建demo项目（2.3.6-Release）

创建Spring Boot项目的步骤如图1-1、图1-2所示。

图 1‑1

图 1‑2

创建好的项目如图1-3所示。

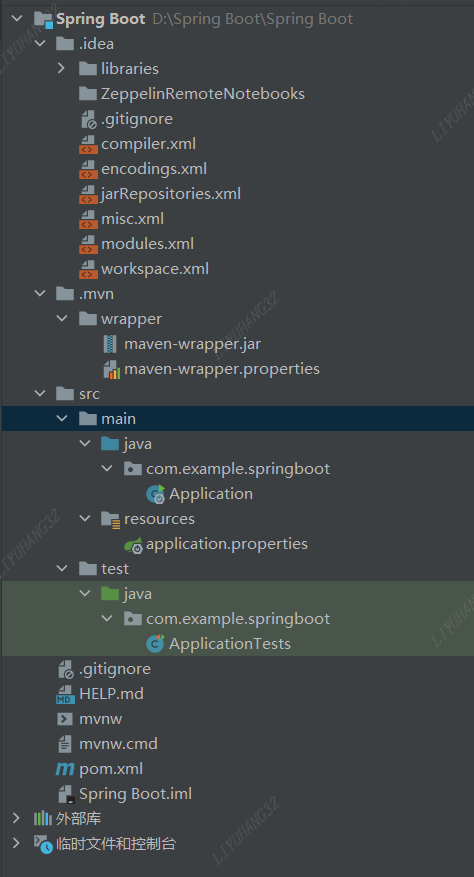
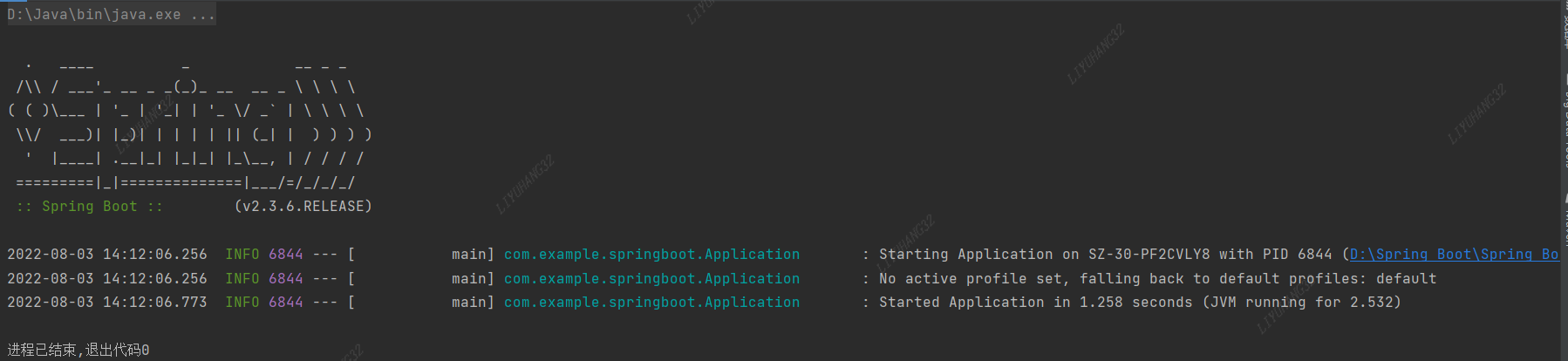


图 1‑3 创建好的Spring Boot项目

运行结果如图1-4所示。

图 1‑4 运行结果

## MySQL数据源配置

### 导入依赖

一般在新建SpringBoot项目时，勾选了MySQL以及JDBC依赖，可以直接使用，无须再次导入依赖。

1.在pom文件中导入MySQL依赖

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.29</version>  
</dependency>

2.在pom文件中导入JDBC依赖

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
</dependency>

3.在pom文件中导入mybatis-plus依赖

<dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-boot-starter</artifactId>  
 <version>3.3.1</version>  
</dependency>

### 连接数据库

在application.yml中进行连接数据库的简单配置

spring.datasource.url=jdbc:mysql://localhost:3306/KYH?serverTimezone=GMT%2B8  
spring.datasource.username=root  
spring.datasource.password=A15114506582a  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.jpa.properties.hibernate.hbm2ddl.auto=update  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect  
spring.jpa.show-sql= true

### 测试

使用mybatis-plus进行映射

采用了Lombok简化代码（添加Lombok依赖）

1.创建Students实体类

package com.example.springboot.domain;  
  
import lombok.Data;  
  
@Data  
public class Students {  
 private Integer sid;  
 private String sname;  
 private String sex;  
  
}

2.在Mapper包下创建UserMapper继承BaseMapper

package com.example.springboot.dao;  
  
import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
import com.example.springboot.domain.Students;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface StudentsMapper extends BaseMapper<Students> {}

3.创建student接口

package com.example.springboot.service;  
  
import com.baomidou.mybatisplus.extension.service.IService;  
import com.example.springboot.domain.Students;  
  
import java.util.List;  
  
public interface StudentsServiceInterface extends IService<Students> {  
 List<Students> select(Students students);  
}

4.创建service类，实现抽象方法

package com.example.springboot.service;  
  
  
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;  
import com.example.springboot.dao.StudentsMapper;  
import com.example.springboot.domain.Students;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
@Service  
public class StudentsService extends ServiceImpl<StudentsMapper, Students> implements StudentsServiceInterface {  
 @Autowired  
 private StudentsMapper studentsMapper;  
  
 @Override  
 public List<Students> select(Students students) {  
 return studentsMapper.selectList(null);  
 }  
}

5.创建controller类

package com.example.springboot.controller;  
  
import com.example.springboot.domain.Students;  
import com.example.springboot.service.StudentsService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class StudentsController {  
 @Autowired  
 private StudentsService studentsService;  
 @GetMapping("test")  
 public List<Students> select(Students students){  
 return studentsService.select(students);  
 }  
}

运行结果如图2-1所示。

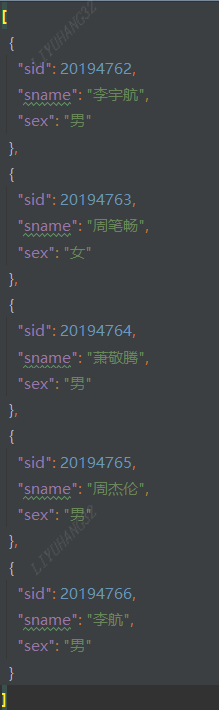


图 2‑1 运行结果

## Redis缓存集成配置

导入依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-cache</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.baomidou</groupId>  
 <artifactId>mybatis-plus-boot-starter</artifactId>  
 <version>3.3.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.29</version>  
 </dependency>  
  
 <dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.10.1</version>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
</dependencies>

application.properties

########################################################  
###datasource 配置 MySQL 数据源；  
########################################################  
spring.datasource.url = jdbc:mysql://localhost:3306/lyh  
spring.datasource.username = root  
spring.datasource.password = A15114506582a  
spring.datasource.driverClassName = com.mysql.cj.jdbc.Driver  
spring.datasource.max-active=20  
spring.datasource.max-idle=8  
spring.datasource.min-idle=8  
spring.datasource.initial-size=10  
########################################################  
###REDIS (RedisProperties) redis 基本配置；  
########################################################  
# database name  
spring.redis.database=0  
# server password  
#spring.redis.password=  
#connection port  
spring.redis.port=6379  
# pool settings ...  
spring.redis.pool.max-idle=8  
spring.redis.pool.min-idle=0  
spring.redis.pool.max-active=8  
spring.redis.pool.max-wait=-1  
# name of Redis server  
#spring.redis.sentinel.master=  
# comma-separated list of host:port pairs  
#spring.redis.sentinel.nodes=

cinfig类

package com.example.redis.Config;  
import com.fasterxml.jackson.annotation.JsonAutoDetect;  
import com.fasterxml.jackson.annotation.PropertyAccessor;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import com.fasterxml.jackson.databind.jsontype.impl.LaissezFaireSubTypeValidator;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.redis.connection.RedisConnectionFactory;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;  
import org.springframework.data.redis.serializer.StringRedisSerializer;  
  
import java.time.Duration;  
  
@Configuration  
public class RedisCacheConfig{  
 @Bean  
 public RedisTemplate<String,Object> redisTemplate(RedisConnectionFactory factory){  
 RedisTemplate<String,Object> redisTemplate = new RedisTemplate<String,Object>();  
 redisTemplate.setConnectionFactory(factory);  
 //Json序列化配置  
 Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer(Object.class);  
 ObjectMapper objectMapper = new ObjectMapper();  
 objectMapper.setVisibility(PropertyAccessor.*ALL*, JsonAutoDetect.Visibility.*ANY*);  
 objectMapper.activateDefaultTyping(LaissezFaireSubTypeValidator.*instance*, ObjectMapper.DefaultTyping.*NON\_FINAL*);  
 //String的序列化  
 StringRedisSerializer stringRedisSerializer=new StringRedisSerializer();  
 //key采用String的序列化方式  
 redisTemplate.setKeySerializer(stringRedisSerializer);  
 //hashkey采用String的序列化方式  
 redisTemplate.setHashKeySerializer(stringRedisSerializer);  
 //value采用jackson的序列化方式  
 redisTemplate.setValueSerializer(jackson2JsonRedisSerializer);  
 //hashvalue采用jackson的序列化方式  
 redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer);  
 return redisTemplate;  
 }  
}

实体类：

package com.example.redis.Domain;  
  
import com.baomidou.mybatisplus.annotation.TableId;  
import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
  
import java.io.Serializable;  
  
@NoArgsConstructor  
@AllArgsConstructor  
@Data  
public class Students implements Serializable {  
  
 @TableId(value ="sid")  
 private Long sid;  
 @TableId(value ="sname")  
 private String sname;  
 @TableId(value ="sex")  
 private String sex;  
}

Mapper：

package com.example.redis.DAO;  
  
import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
import com.example.redis.Domain.Students;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface StudentsMapper extends BaseMapper<Students> {}

Service：

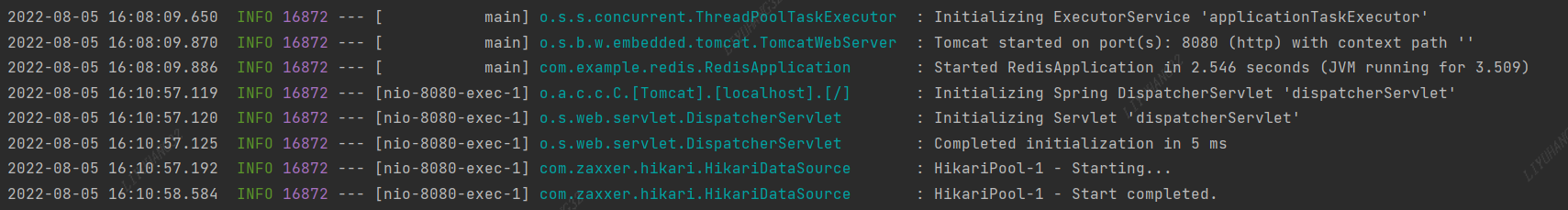
package com.example.redis.Service;  
  
import com.baomidou.mybatisplus.extension.service.IService;  
import com.example.redis.Domain.Students;  
  
import java.util.List;  
  
public interface StudentsServiceInterface extends IService<Students> {  
 List<Students> select(Students students);  
 Students getCacheById(Long sid);  
}

package com.example.redis.Service;  
  
  
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;  
import com.example.redis.DAO.StudentsMapper;  
import com.example.redis.Domain.Students;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cache.annotation.CacheConfig;  
import org.springframework.cache.annotation.Cacheable;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
  
@Service  
@CacheConfig(cacheNames = "Students")  
public class StudentsService extends ServiceImpl<StudentsMapper, Students> implements StudentsServiceInterface {  
 @Autowired  
 private StudentsMapper studentsMapper;  
  
 @Override  
 public List<Students> select(Students students) {  
 return studentsMapper.selectList(null);  
 }  
 @Override  
 @Cacheable(key="#sid")  
 public Students getCacheById(Long sid){  
 return getById(sid);  
 }  
}

Controller

package com.example.redis.Controller;  
  
import com.example.redis.Domain.Students;  
import com.example.redis.Service.StudentsService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.util.List;  
  
@RestController  
public class StudentsController{  
 @Autowired  
 private StudentsService studentsService;  
 @GetMapping("test")  
 public List<Students> select(Students students){  
 return studentsService.select(students);  
 }  
 @GetMapping("{sid}")  
 public Students getById(@PathVariable Long sid){  
 return studentsService.getCacheById(sid);  
 }  
}

运行结果如图3-1至3-3所示。

图 3‑1 直接从数据库获取数据

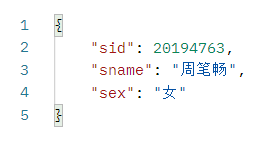
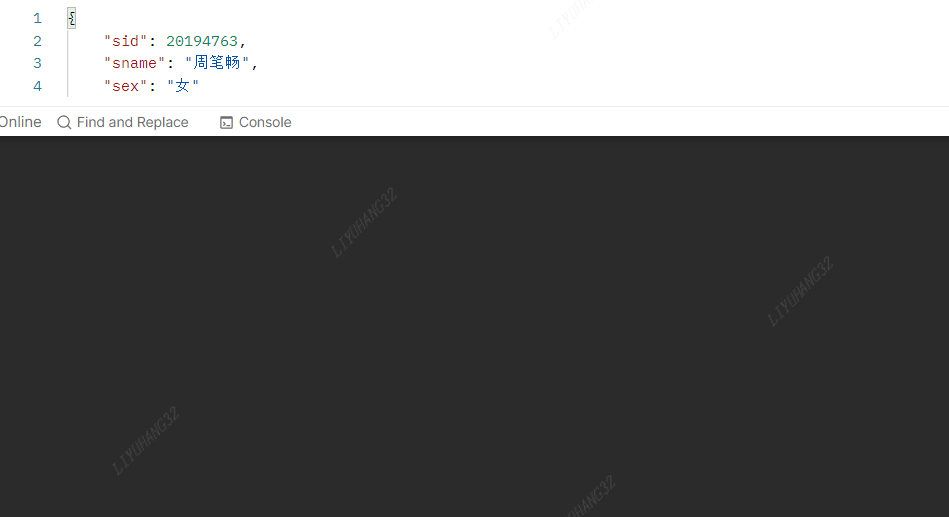


图 3‑2 运行结果

图 3‑3 从缓存中获取数据

## 单表CURD操作

Controller

package com.example.redis.Controller;  
  
import com.example.redis.Domain.Students;  
import com.example.redis.Service.StudentsService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.data.repository.query.ReturnedType;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
public class StudentsController{  
 @Autowired  
 private StudentsService studentsService;  
 //查询所有信息  
 @GetMapping()  
 public List<Students> select(){  
 return studentsService.list();  
 }  
 //根据id查询  
 @GetMapping("{id}")  
 public Students getById(@PathVariable Long id){  
 return studentsService.GetById(id);  
 }  
 //添加用户  
 @PostMapping("add")  
 public boolean add(@RequestBody Students students){  
 return studentsService.save(students);  
 }  
 //删除用户  
 @DeleteMapping("{id}")  
 public boolean delete(@PathVariable("id") Long id){  
 return studentsService.removeById(id);  
 }  
 //更新用户  
 @PutMapping("Update")  
 public boolean update(@RequestBody Students students){  
 return studentsService.updateById(students);  
 }  
}

## 销售订单作业练习

结构包含销售订单头、订单行、客户、商品等基础数据，具体界面后续补充。

# 其他

未完待续