[**#**](https://pig4cloud.com/#%E4%BB%80%E4%B9%88%E6%98%AF%E6%B5%81%EF%BC%9F) **什么是流？**

先来看看Pig upms 中的使用

~~~

@Override

@Transactional(rollbackFor = Exception.class)

public Boolean saveUser(UserDTO userDto) {

SysUser sysUser = new SysUser();

BeanUtils.copyProperties(userDto, sysUser);

sysUser.setDelFlag(CommonConstants.STATUS\_NORMAL);

sysUser.setPassword(ENCODER.encode(userDto.getPassword()));

baseMapper.insert(sysUser);

List<SysUserRole> userRoleList = userDto.getRole()

.stream().map(roleId -> {

SysUserRole userRole = new SysUserRole();

userRole.setUserId(sysUser.getUserId());

userRole.setRoleId(roleId);

return userRole;

}).collect(Collectors.toList());

return sysUserRoleService.saveBatch(userRoleList);

}

~~~

流是Java8引入的全新概念，它用来处理集合中的数据，暂且可以把它理解为一种高级集合。 众所周知，集合操作非常麻烦，若要对集合进行筛选、投影，需要写大量的代码，而流是以声明的形式操作集合，它就像SQL语句，我们只需告诉流需要对集合进行什么操作，它就会自动进行操作，并将执行结果交给你，无需我们自己手写代码。 因此，流的集合操作对我们来说是透明的，我们只需向流下达命令，它就会自动把我们想要的结果给我们。由于操作过程完全由Java处理，因此它可以根据当前硬件环境选择最优的方法处理，我们也无需编写复杂又容易出错的多线程代码了。

[**#**](https://pig4cloud.com/#%E6%B5%81%E7%9A%84%E7%89%B9%E7%82%B9) **流的特点**

1. 只能遍历一次 我们可以把流想象成一条流水线，流水线的源头是我们的数据源(一个集合)，数据源中的元素依次被输送到流水线上，我们可以在流水线上对元素进行各种操作。 一旦元素走到了流水线的另一头，那么这些元素就被“消费掉了”，我们无法再对这个流进行操作。当然，我们可以从数据源那里再获得一个新的流重新遍历一遍。
2. 采用内部迭代方式 若要对集合进行处理，则需我们手写处理代码，这就叫做外部迭代。 而要对流进行处理，我们只需告诉流我们需要什么结果，处理过程由流自行完成，这就称为内部迭代。

[**#**](https://pig4cloud.com/#%E6%B5%81%E7%9A%84%E6%93%8D%E4%BD%9C%E7%A7%8D%E7%B1%BB) **流的操作种类**

流的操作分为两种，分别为中间操作和终端操作。

1. 中间操作 当数据源中的数据上了流水线后，这个过程对数据进行的所有操作都称为“中间操作”。 中间操作仍然会返回一个流对象，因此多个中间操作可以串连起来形成一个流水线。
2. 终端操作 当所有的中间操作完成后，若要将数据从流水线上拿下来，则需要执行终端操作。 终端操作将返回一个执行结果，这就是你想要的数据。

[**#**](https://pig4cloud.com/#%E6%B5%81%E7%9A%84%E6%93%8D%E4%BD%9C%E8%BF%87%E7%A8%8B) **流的操作过程**

使用流一共需要三步：

1. 准备一个数据源
2. 执行中间操作 中间操作可以有多个，它们可以串连起来形成流水线。
3. 执行终端操作 执行终端操作后本次流结束，你将获得一个执行结果。

[**#**](https://pig4cloud.com/#%E4%BD%BF%E7%94%A8%E6%B5%81) **使用流**

[**#**](https://pig4cloud.com/#%E5%88%9B%E5%BB%BA%E6%B5%81) **创建流**

在使用流之前，首先需要拥有一个数据源，并通过StreamAPI提供的一些方法获取该数据源的流对象。数据源可以有多种形式：

**1. 集合**

这种数据源较为常用，通过stream()方法即可获取流对象：

List<Person> list = new ArrayList<Person>();

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

**2. 数组**

通过Arrays类提供的静态函数stream()获取数组的流对象：

String[] names = {"chaimm","peter","john"};

Stream<String> stream = Arrays.stream(names);

**3. 值**

直接将几个值变成流对象：

Stream<String> stream = Stream.of("chaimm","peter","john");

**4. 文件**

try(Stream lines = Files.lines(Paths.get(“文件路径名”),Charset.defaultCharset())){

//可对lines做一些操作

}catch(IOException e){

}

**5. iterator**

**创建无限流**

Stream.iterate(0, n -> n + 2)

.limit(10)

.forEach(System.out::println);

PS：Java7简化了IO操作，把打开IO操作放在try后的括号中即可省略关闭IO的代码。

[**#**](https://pig4cloud.com/#%E7%AD%9B%E9%80%89-filter) **筛选 filter**

filter 函数接收一个Lambda表达式作为参数，该表达式返回boolean，在执行过程中，流将元素逐一输送给filter，并筛选出执行结果为true的元素。 如，筛选出所有学生：

List<Person> result = list.stream()

.filter(Person::isStudent)

.collect(toList());

[**#**](https://pig4cloud.com/#%E5%8E%BB%E9%87%8Ddistinct) **去重distinct**

去掉重复的结果：

List<Person> result = list.stream()

.distinct()

.collect(toList());

[**#**](https://pig4cloud.com/#%E6%88%AA%E5%8F%96) **截取**

截取流的前N个元素：

List<Person> result = list.stream()

.limit(3)

.collect(toList());

[**#**](https://pig4cloud.com/#%E8%B7%B3%E8%BF%87) **跳过**

跳过流的前n个元素：

List<Person> result = list.stream()

.skip(3)

.collect(toList());

[**#**](https://pig4cloud.com/#%E6%98%A0%E5%B0%84) **映射**

对流中的每个元素执行一个函数，使得元素转换成另一种类型输出。流会将每一个元素输送给map函数，并执行map中的Lambda表达式，最后将执行结果存入一个新的流中。 如，获取每个人的姓名(实则是将Perosn类型转换成String类型)：

List<Person> result = list.stream()

.map(Person::getName)

.collect(toList());

[**#**](https://pig4cloud.com/#%E5%90%88%E5%B9%B6%E5%A4%9A%E4%B8%AA%E6%B5%81) **合并多个流**

例：列出List中各不相同的单词，List集合如下：

List<String> list = new ArrayList<String>();

list.add("I am a boy");

list.add("I love the girl");

list.add("But the girl loves another girl");

思路如下：

首先将list变成流：

list.stream();

按空格分词：

list.stream()

.map(line->line.split(" "));

分完词之后，每个元素变成了一个String[]数组。

将每个 String[] 变成流：

list.stream()

.map(line->line.split(" "))

.map(Arrays::stream)

此时一个大流里面包含了一个个小流，我们需要将这些小流合并成一个流。

将小流合并成一个大流：用 flatMap 替换刚才的 map

list.stream()

.map(line->line.split(" "))

.flatMap(Arrays::stream)

去重

list.stream()

.map(line->line.split(" "))

.flatMap(Arrays::stream)

.distinct()

.collect(toList());

[**#**](https://pig4cloud.com/#%E6%98%AF%E5%90%A6%E5%8C%B9%E9%85%8D%E4%BB%BB%E4%B8%80%E5%85%83%E7%B4%A0%EF%BC%9Aanymatch) **是否匹配任一元素：anyMatch**

anyMatch用于判断流中是否存在至少一个元素满足指定的条件，这个判断条件通过Lambda表达式传递给anyMatch，执行结果为boolean类型。 如，判断list中是否有学生：

boolean result = list.stream()

.anyMatch(Person::isStudent);

[**#**](https://pig4cloud.com/#%E6%98%AF%E5%90%A6%E5%8C%B9%E9%85%8D%E6%89%80%E6%9C%89%E5%85%83%E7%B4%A0%EF%BC%9Aallmatch) **是否匹配所有元素：allMatch**

allMatch用于判断流中的所有元素是否都满足指定条件，这个判断条件通过Lambda表达式传递给anyMatch，执行结果为boolean类型。 如，判断是否所有人都是学生：

boolean result = list.stream()

.allMatch(Person::isStudent);

[**#**](https://pig4cloud.com/#%E6%98%AF%E5%90%A6%E6%9C%AA%E5%8C%B9%E9%85%8D%E6%89%80%E6%9C%89%E5%85%83%E7%B4%A0%EF%BC%9Anonematch) **是否未匹配所有元素：noneMatch**

noneMatch与allMatch恰恰相反，它用于判断流中的所有元素是否都不满足指定条件：

boolean result = list.stream()

.noneMatch(Person::isStudent);

[**#**](https://pig4cloud.com/#%E8%8E%B7%E5%8F%96%E4%BB%BB%E4%B8%80%E5%85%83%E7%B4%A0findany) **获取任一元素findAny**

findAny能够从流中随便选一个元素出来，它返回一个Optional类型的元素。

Optional<Person> person = list.stream().findAny();

[**#**](https://pig4cloud.com/#%E8%8E%B7%E5%8F%96%E7%AC%AC%E4%B8%80%E4%B8%AA%E5%85%83%E7%B4%A0findfirst) **获取第一个元素findFirst**

Optional<Person> person = list.stream().findFirst();

[**#**](https://pig4cloud.com/#%E5%BD%92%E7%BA%A6) **归约**

归约是将集合中的所有元素经过指定运算，折叠成一个元素输出，如：求最值、平均数等，这些操作都是将一个集合的元素折叠成一个元素输出。

在流中，reduce函数能实现归约。 reduce函数接收两个参数：

1. 初始值
2. 进行归约操作的Lambda表达式

**元素求和：自定义Lambda表达式实现求和**

例：计算所有人的年龄总和

int age = list.stream().reduce(0, (person1,person2)->person1.getAge()+person2.getAge());

1. reduce的第一个参数表示初试值为0；
2. reduce的第二个参数为需要进行的归约操作，它接收一个拥有两个参数的Lambda表达式，reduce会把流中的元素两两输给Lambda表达式，最后将计算出累加之和。

**元素求和：使用Integer.sum函数求和**

上面的方法中我们自己定义了Lambda表达式实现求和运算，如果当前流的元素为数值类型，那么可以使用Integer提供了sum函数代替自定义的Lambda表达式，如：

int age = list.stream().reduce(0, Integer::sum);

Integer类还提供了 min、max 等一系列数值操作，当流中元素为数值类型时可以直接使用。

[**#**](https://pig4cloud.com/#%E6%95%B0%E5%80%BC%E6%B5%81%E7%9A%84%E4%BD%BF%E7%94%A8) **数值流的使用**

采用reduce进行数值操作会涉及到基本数值类型和引用数值类型之间的装箱、拆箱操作，因此效率较低。 当流操作为纯数值操作时，使用数值流能获得较高的效率。

**将普通流转换成数值流**

StreamAPI提供了三种数值流：IntStream、DoubleStream、LongStream，也提供了将普通流转换成数值流的三种方法：mapToInt、mapToDouble、mapToLong。 如，将Person中的age转换成数值流：

IntStream stream = list.stream().mapToInt(Person::getAge);

**数值计算**

每种数值流都提供了数值计算函数，如max、min、sum等。如，找出最大的年龄：

OptionalInt maxAge = list.stream()

.mapToInt(Person::getAge)

.max();

由于数值流可能为空，并且给空的数值流计算最大值是没有意义的，因此max函数返回OptionalInt，它是Optional的一个子类，能够判断流是否为空，并对流为空的情况作相应的处理。 此外，mapToInt、mapToDouble、mapToLong进行数值操作后的返回结果分别为：OptionalInt、OptionalDouble、OptionalLong

[**#**](https://pig4cloud.com/#%E4%B8%AD%E9%97%B4%E6%93%8D%E4%BD%9C%E5%92%8C%E6%94%B6%E9%9B%86%E6%93%8D%E4%BD%9C) **中间操作和收集操作**

| **操作** | **类型** | **返回类型** | **使用的类型/函数式接口** | **函数描述符** |
| --- | --- | --- | --- | --- |
| filter | 中间 | Stream<T> | Predicate<T> | T -> boolean |
| distinct | 中间 | Stream<T> |  |  |
| skip | 中间 | Stream<T> | long |  |
| map | 中间 | Stream<R> | Function<T, R> | T -> R |
| flatMap | 中间 | Stream<R> | Function<T, Stream<R>> | T -> Stream<R> |
| limit | 中间 | Stream<T> | long |  |
| sorted | 中间 | Stream<T> | Comparator<T> | (T, T) -> int |
| anyMatch | 终端 | boolean | Predicate<T> | T -> boolean |
| noneMatch | 终端 | boolean | Predicate<T> | T -> boolean |
| allMatch | 终端 | boolean | Predicate<T> | T -> boolean |
| findAny | 终端 | Optional<T> |  |  |
| findFirst | 终端 | Optional<T> |  |  |
| forEach | 终端 | void | Consumer<T> | T -> void |
| collect | 终端 | R | Collector<T, A, R> |  |
| reduce | 终端 | Optional<T> | BinaryOperator<T> | (T, T) -> T |
| count | 终端 | long |  |  |

[**#**](https://pig4cloud.com/#collector-%E6%94%B6%E9%9B%86) **Collector 收集**

收集器用来将经过筛选、映射的流进行最后的整理，可以使得最后的结果以不同的形式展现。 collect 方法即为收集器，它接收 Collector 接口的实现作为具体收集器的收集方法。 Collector 接口提供了很多默认实现的方法，我们可以直接使用它们格式化流的结果；也可以自定义 Collector 接口的实现，从而定制自己的收集器。

[**#**](https://pig4cloud.com/#%E5%BD%92%E7%BA%A6-2) **归约**

流由一个个元素组成，归约就是将一个个元素“折叠”成一个值，如求和、求最值、求平均值都是归约操作。

[**#**](https://pig4cloud.com/#%E4%B8%80%E8%88%AC%E6%80%A7%E5%BD%92%E7%BA%A6) **一般性归约**

若你需要自定义一个归约操作，那么需要使用 Collectors.reducing 函数，该函数接收三个参数：

* 第一个参数为归约的初始值
* 第二个参数为归约操作进行的字段
* 第三个参数为归约操作的过程

[**#**](https://pig4cloud.com/#%E6%B1%87%E6%80%BB) **汇总**

Collectors类专门为汇总提供了一个工厂方法：Collectors.summingInt。 它可接受一 个把对象映射为求和所需int的函数，并返回一个收集器；该收集器在传递给普通的 collect 方法后即执行我们需要的汇总操作。

[**#**](https://pig4cloud.com/#%E5%88%86%E7%BB%84) **分组**

数据分组是一种更自然的分割数据操作，分组就是将流中的元素按照指定类别进行划分，类似于SQL语句中的 GROUPBY。

[**#**](https://pig4cloud.com/#%E5%A4%9A%E7%BA%A7%E5%88%86%E7%BB%84) **多级分组**

多级分组可以支持在完成一次分组后，分别对每个小组再进行分组。 使用具有两个参数的 groupingBy 重载方法即可实现多级分组。

* 第一个参数：一级分组的条件
* 第二个参数：一个新的 groupingBy 函数，该函数包含二级分组的条件

**Collectors 类的静态工厂方法**

| **工厂方法** | **返回类型** | **用途** | **示例** |
| --- | --- | --- | --- |
| toList | List<T> | 把流中所有项目收集到一个 List | List<Project> projects = projectStream.collect(toList()); |
| toSet | Set<T> | 把流中所有项目收集到一个 Set，删除重复项 | Set<Project> projects = projectStream.collect(toSet()); |
| toCollection | Collection<T> | 把流中所有项目收集到给定的供应源创建的集合 | Collection<Project> projects = projectStream.collect(toCollection(), ArrayList::new); |
| counting | Long | 计算流中元素的个数 | long howManyProjects = projectStream.collect(counting()); |
| summingInt | Integer | 对流中项目的一个整数属性求和 | int totalStars = projectStream.collect(summingInt(Project::getStars)); |
| averagingInt | Double | 计算流中项目 Integer 属性的平均值 | double avgStars = projectStream.collect(averagingInt(Project::getStars)); |
| summarizingInt | IntSummaryStatistics | 收集关于流中项目 Integer 属性的统计值，例如最大、最小、 总和与平均值 | IntSummaryStatistics projectStatistics = projectStream.collect(summarizingInt(Project::getStars)); |
| joining | String | 连接对流中每个项目调用 toString 方法所生成的字符串 | String shortProject = projectStream.map(Project::getName).collect(joining(", ")); |
| maxBy | Optional<T> | 按照给定比较器选出的最大元素的 Optional， 或如果流为空则为 Optional.empty() | Optional<Project> fattest = projectStream.collect(maxBy(comparingInt(Project::getStars))); |
| minBy | Optional<T> | 按照给定比较器选出的最小元素的 Optional， 或如果流为空则为 Optional.empty() | Optional<Project> fattest = projectStream.collect(minBy(comparingInt(Project::getStars))); |
| reducing | 归约操作产生的类型 | 从一个作为累加器的初始值开始，利用 BinaryOperator 与流中的元素逐个结合，从而将流归约为单个值 | int totalStars = projectStream.collect(reducing(0, Project::getStars, Integer::sum)); |
| collectingAndThen | 转换函数返回的类型 | 包含另一个收集器，对其结果应用转换函数 | int howManyProjects = projectStream.collect(collectingAndThen(toList(), List::size)); |
| groupingBy | Map<K, List<T>> | 根据项目的一个属性的值对流中的项目作问组，并将属性值作 为结果 Map 的键 | Map<String,List<Project>> projectByLanguage = projectStream.collect(groupingBy(Project::getLanguage)); |
| partitioningBy | Map<Boolean,List<T>> | 根据对流中每个项目应用断言的结果来对项目进行分区 | Map<Boolean,List<Project>> vegetarianDishes = projectStream.collect(partitioningBy(Project::isVegetarian)); |

[**#**](https://pig4cloud.com/#%E8%BD%AC%E6%8D%A2%E7%B1%BB%E5%9E%8B) **转换类型**

有一些收集器可以生成其他集合。比如前面已经见过的 toList，生成了 java.util.List 类的实例。 还有 toSet 和 toCollection，分别生成 Set 和 Collection 类的实例。 到目前为止， 我已经讲了很多流上的链式操作，但总有一些时候，需要最终生成一个集合——比如：

* 已有代码是为集合编写的，因此需要将流转换成集合传入；
* 在集合上进行一系列链式操作后，最终希望生成一个值；
* 写单元测试时，需要对某个具体的集合做断言。

使用 toCollection，用定制的集合收集元素

stream.collect(toCollection(TreeSet::new));

还可以利用收集器让流生成一个值。 maxBy 和 minBy 允许用户按某种特定的顺序生成一个值。

[**#**](https://pig4cloud.com/#%E6%95%B0%E6%8D%AE%E5%88%86%E5%8C%BA) **数据分区**

分区是分组的特殊情况：由一个断言（返回一个布尔值的函数）作为分类函数，它称分区函数。 分区函数返回一个布尔值，这意味着得到的分组 Map 的键类型是 Boolean，于是它最多可以分为两组: true是一组，false是一组。

分区的好处在于保留了分区函数返回true或false的两套流元素列表。

[**#**](https://pig4cloud.com/#%E5%B9%B6%E8%A1%8C%E6%B5%81) **并行流**

并行流就是一个把内容分成多个数据块，并用不不同的线程分别处理每个数据块的流。最后合并每个数据块的计算结果。

将一个顺序执行的流转变成一个并发的流只要调用 parallel() 方法

public static long parallelSum(long n){

return Stream.iterate(1L, i -> i +1).limit(n).parallel().reduce(0L,Long::sum);

}

将一个并发流转成顺序的流只要调用 sequential() 方法

stream.parallel().filter(...).sequential().map(...).parallel().reduce();

这两个方法可以多次调用，只有最后一个调用决定这个流是顺序的还是并发的。

并发流使用的默认线程数等于你机器的处理器核心数。

通过这个方法可以修改这个值，这是全局属性。

System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "12");

并非使用多线程并行流处理数据的性能一定高于单线程顺序流的性能，因为性能受到多种因素的影响。 如何高效使用并发流的一些建议：

1. 如果不确定， 就自己测试。
2. 尽量使用基本类型的流 IntStream, LongStream, DoubleStream
3. 有些操作使用并发流的性能会比顺序流的性能更差，比如limit，findFirst，依赖元素顺序的操作在并发流中是极其消耗性能的。findAny的性能就会好很多，应为不依赖顺序。
4. 考虑流中计算的性能(Q)和操作的性能(N)的对比, Q表示单个处理所需的时间，N表示需要处理的数量，如果Q的值越大, 使用并发流的性能就会越高。
5. 数据量不大时使用并发流，性能得不到提升。
6. 考虑数据结构：并发流需要对数据进行分解，不同的数据结构被分解的性能时不一样的。

**流的数据源和可分解性**

| **源** | **可分解性** |
| --- | --- |
| ArrayList | 非常好 |
| LinkedList | 差 |
| IntStream.range | 非常好 |
| Stream.iterate | 差 |
| HashSet | 好 |
| TreeSet | 好 |

**流的特性以及中间操作对流的修改都会对数据对分解性能造成影响。 比如固定大小的流在任务分解的时候就可以平均分配，但是如果有filter操作，那么流就不能预先知道在这个操作后还会剩余多少元素。**

**考虑终端操作的性能：如果终端操作在合并并发流的计算结果时的性能消耗太大，那么使用并发流提升的性能就会得不偿失。**