# 为什么需要引入流

集合是Java中使用最多的API。但是集合中只提供了简单的使用，可以存储，遍历。一些复杂的操作类似数据库的方式筛选是不支持的。例如：
菜按照类别进行分组 ，或查找出最贵的菜。数据库里面可以写sql语句来实现，而集合里面只能写笨拙的代码遍历，判断。

流是Java API的新成员，它允许你以声明性方式处理数据集合。可以把流看成遍历数据集的高级迭代器，简短的定义流：从支持数据处理操作的源生成的元素序列

通过集合来学习流是最简短的方式，还有很多其他地方也可以获取到流，比如文件IO

使用流处理集合有三个优点：

声明性——更简洁，更易读
可复合——更灵活
可并行——性能更好

# 流与集合的区别

比如DVD里面的电影，这就是一个集合。这是完整的
而从互联网播放的一个视频文件，这里播放的就是流媒体，流是一个完整的数据中的一部分。

只能遍历一次
　　流只能遍历一次。遍历完之后，我们就说这个流已经被消费掉了。你可以从原始数据源那里再获得一个新的流

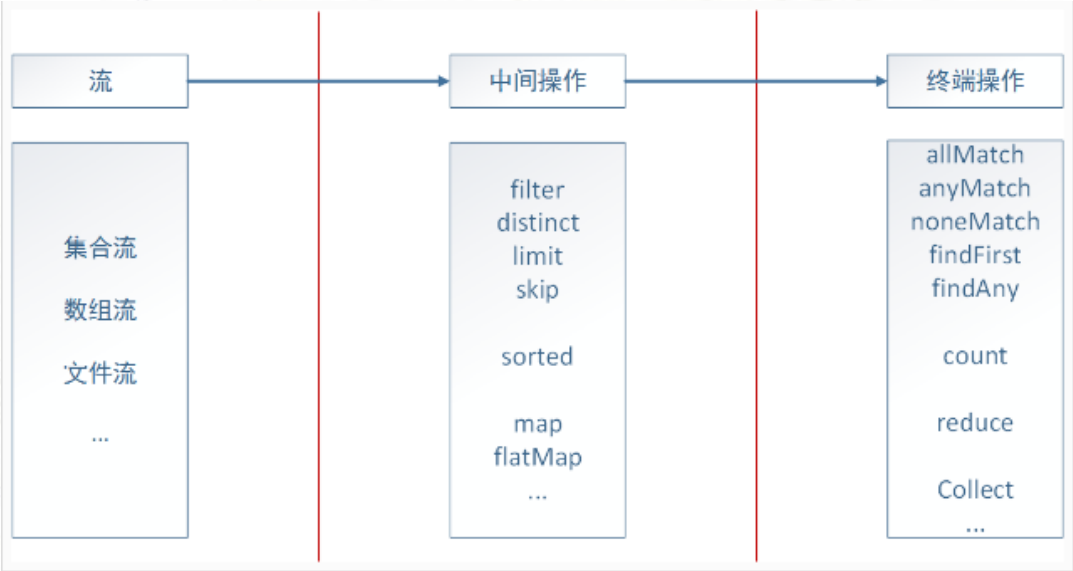
# 流的操作

```
1. List<String> names = menu.stream()
2.                          .filter(d -> d.getCalories() >
300)
3.                          .map(Dish::getName)
4.                          .limit(3)
5.                          .collect(toList());
```

你可以看到两类操作：

filter、 map和limit可以连成一条流水线；
collect触发流水线执行并关闭它

**中间操作**：filter或map或limit等中间操作会返回另一个流。这让多个操作可以连接起来形成一个查询
**终端操作**：终端操作会关闭流，并且从流的流水线生成结果




总而言之，流的使用一般包括三件事：

一个数据源（如集合，文件IO）来执行一个查询；
一个中间操作链，形成一条流的流水线；
一个终端操作，执行流水线，并能生成结果

# 流的具体使用

## 数据源

```
1. public class Student {
2.
3.     /** 学号 */
4.     private long id;
5.
6.     private String name;
7.
8.     private int age;
9.
10.     /** 年级 */
11.     private int grade;
12.
13.     /** 专业 */
14.     private String major;
15.
16.     /** 学校 */
17.     private String school;
18.
19.     // 省略getter和setter
20. }
```

```
1. List<Student> students = new ArrayList<Student>() {
2.     {
3.         add(new Student(20160001, "孔明", 20, 1, "土木工程", "武汉大学"));
4.         add(new Student(20160002, "伯约", 21, 2, "信息安全", "武汉大学"));
5.         add(new Student(20160003, "玄德", 22, 3, "经济管理", "武汉大学"));
6.         add(new Student(20160004, "云长", 21, 2, "信息安全", "武汉大学"));
7.         add(new Student(20161001, "翼德", 21, 2, "机械与自动化", "华中科技大学"));
8.         add(new Student(20161002, "元直", 23, 4, "土木工程", "华中科技大学"));
9.         add(new Student(20161003, "奉孝", 23, 4, "计算机科学", "华中科技大学"));
10.         add(new Student(20162001, "仲谋", 22, 3, "土木工程", "浙江大学"));
11.         add(new Student(20162002, "鲁肃", 23, 4, "计算机科学", "浙江大学"));
12.         add(new Student(20163001, "丁奉", 24, 5, "土木工程", "南京大学"));
13.     }
14. };
```

## 中间操作

### 过滤

#### filter： 获取武汉大学的学生

```
List<Student> whuStudents = students.stream()
                                    .filter(student ->
"武汉大学".equals(student.getSchool()))
.collect(Collectors.toList());
```

#### distinct： 去重

```
List<Integer> num = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
List<Integer> disList = num.stream()
                           .distinct()
.collect(Collectors.toList());
```

#### limit： 截断流，只获取2个土木工程的学生

```
List<Student> civilStudents = students.stream()
                                      .filter(student ->
"土木工程".equals(student.getMajor())).limit(2)
.collect(Collectors.toList());
```

#### sorted： 排序，只获取2个土木工程的学生，根据年龄排序

```
List<Student> sortedCivilStudents = students.stream()
.filter(student -> "土木工
程".equals(student.getMajor())).sorted((s1, s2) ->
s1.getAge() - s2.getAge())
                                                .limit(2)
.collect(Collectors.toList());
```

#### skip： 跳过前面的几个，只获取后面的

```
List<Student> civilStudents = students.stream()
                                      .filter(student ->
"土木工程".equals(student.getMajor()))
                                      .skip(2)
.collect(Collectors.toList());
```

### 映射

#### map：

筛选出所有专业为计算机科学的学生姓名，那么我们可以在filter筛选的基础之上，通过map将学生实体映射成为学生姓名字符串，具体实现如下：

```
List<String> names = students.stream()
                                .filter(student -> "计算机科学".equals(student.getMajor()))
.map(Student::getName).collect(Collectors.toList());  //将Student List 转换成 Sutdent的名称List
```

计算所有专业为计算机科学学生的年龄之和

```
int totalAge = students.stream()
                    .filter(student -> "计算机科学".equals(student.getMajor()))
                    .mapToInt(Student::getAge).sum();
```

flatmap：

flatMap与map的区别在于 flatMap是将一个流中的每个值都转成一个个流，然后再将这些流扁平化成为一个流

我们有一个字符串数组String[] strs = {"java8", "is", "easy", "to", "use"};，我们希望输出构成这一数组的所有非重复字符，那么我们可能首先会想到如下实现：

```
List<String[]> distinctStrs = Arrays.stream(strs)
                                .map(str -> str.split(""))  // 映射成为Stream<String[]>
                                .distinct()
.collect(Collectors.toList());
```

而用flatmap直接可以实现

```
List<String> distinctStrs = Arrays.stream(strs)
                                .map(str -> str.split(""))  // 映射成为Stream<String[]>
.flatMap(Arrays::stream)  // 扁平化为Stream<String>
                                .distinct()
.collect(Collectors.toList());
```

终端操作

## 查找

### allMatch

allMatch用于检测是否全部都满足指定的参数行为，如果全部满足则返回true，检测是否所有的学生都已满18周岁：

```
1. boolean isAdult = students.stream().allMatch(student ->
student.getAge() >= 18);
```

### anyMatch

anyMatch则是检测是否存在一个或多个满足指定的参数行为，如果满足则返回true，检测是否有来自武汉大学的学生：

```
1. boolean hasWhu = students.stream().anyMatch(student ->
"武汉大学".equals(student.getSchool()));
```

### noneMatch

noneMatch用于检测是否不存在满足指定行为的元素，如果不存在则返回true，检测是否不存在专业为计算机科学的学生：

```
1. boolean noneCs = students.stream().noneMatch(student ->
"计算机科学".equals(student.getMajor()));
```

### findFirst

findFirst用于返回满足条件的第一个元素，比如我们希望选出专业为土木工程的排在第一个学生，那么可以实现如下：

```
1. students.stream().filter(student -> "土木工
程".equals(student.getMajor())).findFirst();
```

### findAny

findAny相对于findFirst的区别在于，findAny不一定返回第一个，而是返回任意一个

实际上对于顺序流式处理而言，findFirst和findAny返回的结果是一样的，至于为什么会这样设计，是因为在下一篇我们介绍的并行流式处理，当我们启用并行流式处理的时候，查找第一个元素往往会有很多限制，如果不是特别需求，在并行流式处理中使用findAny的性能要比findFirst好

## 规约

```
// 前面例子中的方法
int totalAge = students.stream()
                    .filter(student -> "计算机科学".equals(student.getMajor()))
                    .mapToInt(Student::getAge).sum();
// 归约操作
int totalAge = students.stream()
                    .filter(student -> "计算机科学".equals(student.getMajor()))
                    .map(Student::getAge)
                    .reduce(0, (a, b) -> a + b);

// 进一步简化
int totalAge2 = students.stream()
                     .filter(student -> "计算机科学".equals(student.getMajor()))
                     .map(Student::getAge)
                     .reduce(0, Integer::sum);

// 采用无初始值的重载版本，需要注意返回Optional
Optional<Integer> totalAge = students.stream()
                    .filter(student -> "计算机科学".equals(student.getMajor()))
                    .map(Student::getAge)
                    .reduce(Integer::sum);  // 去掉初始值
```

map和reduce的连接通常称为map-reduce模式

## 收集器

*高级规约*： 收集器也提供了规约服务

求学生的总人数求学生的总人数

```
long count = students.stream().collect(Collectors.counting());

// 进一步简化
long count = students.stream().count();
```

求年龄的最大值和最小值

```
// 求最大年龄
Optional<Student> olderStudent = students.stream().collect(Collectors.maxBy((s1, s2) -> s1.getAge() - s2.getAge()));

// 进一步简化
Optional<Student> olderStudent2 =
```

```
students.stream().collect(Collectors.maxBy(Comparator.
comparing(Student::getAge)));
6.
7. // 求最小年龄
8. Optional<Student> olderStudent3 =
students.stream().collect(Collectors.minBy(Comparator.
comparing(Student::getAge)));
```

求年龄总和

```
1. int totalAge4 =
students.stream().collect(Collectors.summingInt(Studen
t::getAge));
```

求年龄的平均值

```
1. double avgAge =
students.stream().collect(Collectors.averagingInt(Stud
ent::getAge));
```

字符串拼接

```
1. String names =
students.stream().map(Student::getName).collect(Collec
tors.joining());
2. // 输出：孔明伯约玄德云长翼德元直奉孝仲谋鲁肃丁奉
3. String names =
students.stream().map(Student::getName).collect(Collec
tors.joining(", "));
4. // 输出：孔明, 伯约, 玄德, 云长, 翼德, 元直, 奉孝, 仲谋,
鲁肃, 丁奉
```

*分组*：在数据库操作中，我们可以通过GROUP BY关键字对查询到的数据进行分组，java8的流式处理也为我们提供了这样的功能Collectors.groupingBy来操作集合

按学校对上面的学生进行分组

```
1. Map<String, List<Student>> groups =
students.stream().collect(Collectors.groupingBy(Studen
t::getSchool));
```

多级分组，在按学校分组的基础之上再按照专业进行分组

```
1. Map<String, Map<String, List<Student>>> groups2 =
students.stream().collect(
2.
Collectors.groupingBy(Student::getSchool,  // 一级分
组，按学校
3.
Collectors.groupingBy(Student::getMajor)));  // 二级分
组，按专业
```

groupingBy的第二个参数不是只能传递groupingBy，还可以传递任意Collector类型，比如我们可以传递一个Collector.counting，用以统计每个组的个数

```
1. Map<String, Long> groups = 
students.stream().collect(Collectors.groupingBy(Studen
t::getSchool, Collectors.counting()));
```

如果不添加第二个参数，则编译器会默认帮我们添加一个Collectors.toList()。

*分区：*分区可以看做是分组的一种特殊情况，在分区中key只有两种情况：true或false，目的是将待分区集合按照条件一分为二

将学生分为武大学生和非武大学生，那么可以实现如下

```
1. Map<Boolean, List<Student>> partition = 
students.stream().collect(Collectors.partitioningBy(studen
t -> "武汉大学".equals(student.getSchool())));
```