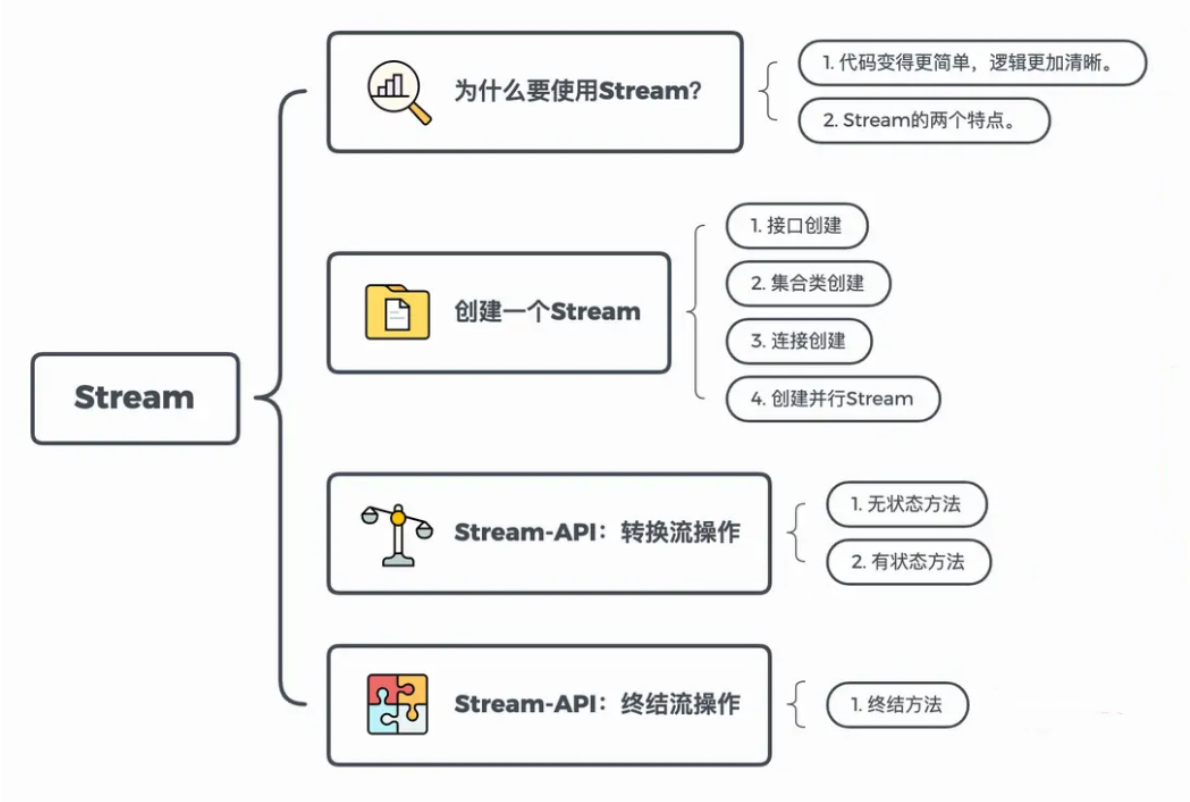
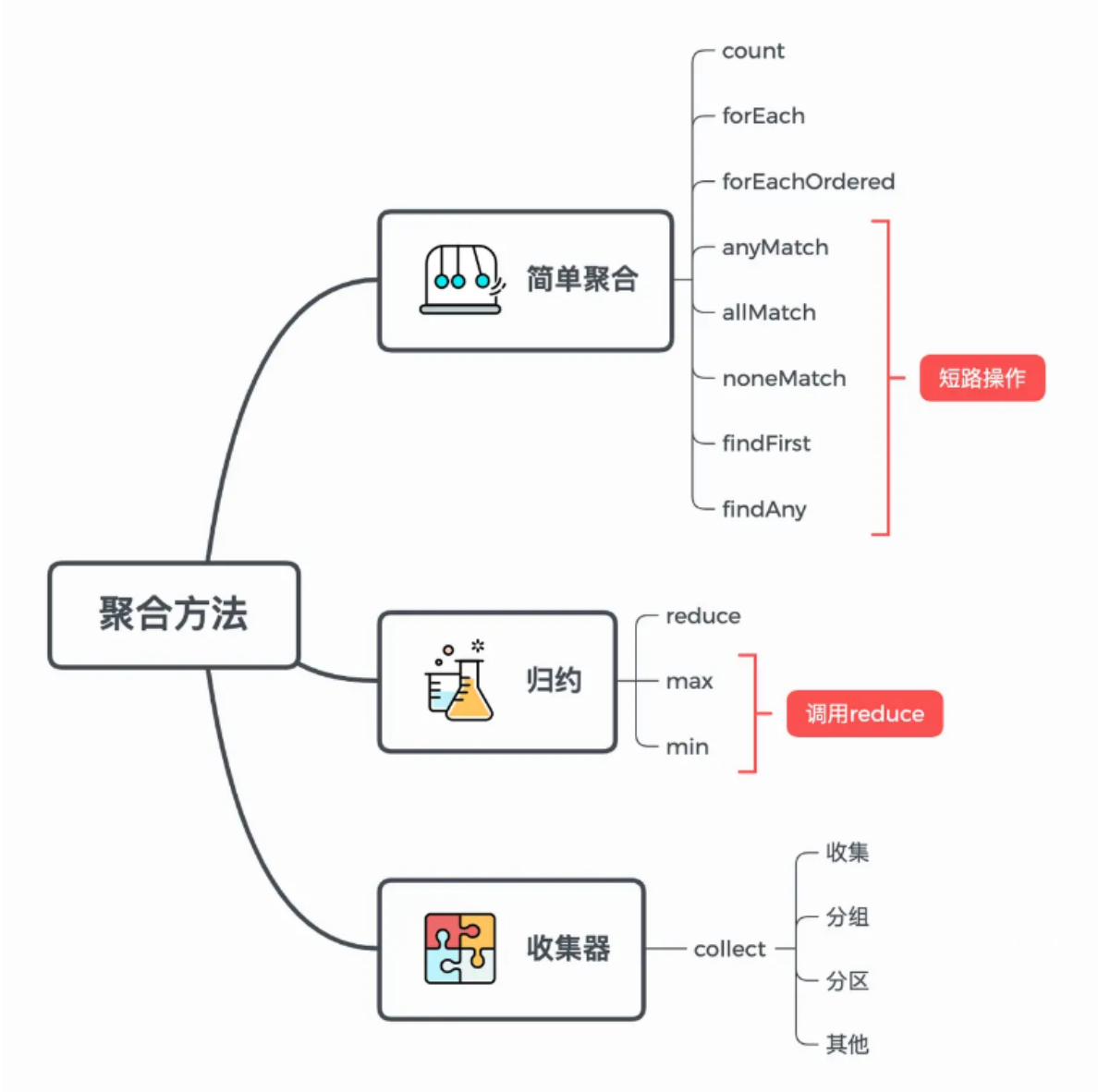
JavaStream数据处理:



聚合方法(终结方法)：

1. 聚合方法代表着整个流计算的最终结果，所以它的返回值都不是Stream。
2. 聚合方法返回值可能为空，比如filter没有匹配到的情况，JDK8中用Optional来规避NPE(空指针异常)。
3. 聚合方法都会调用evaluate方法，这是一个内部方法，看源码的过程中可以用它来判定一个方法是不是聚合方法。

聚合方法分类：



Stream的聚合方法是我们在使用Stream中的必用操作:

1. 简单聚合方法

Stream的聚合方法,先来说说这部分方法:

- `count()`：返回Stream中元素的size大小。
- `forEach()`：通过内部循环Stream中的所有元素，对每一个元素进行消费，此方法没有返回值。
- `forEachOrder()`：和上面方法的效果一样，但是这个可以保持消费顺序，哪怕是在多线程环境下。
- `anyMatch(Predicate predicate)`：这是一个短路操作，通过传入断言参数判断是否有元素能够匹配上断言。
- `allMatch(Predicate predicate)`：这是一个短路操作，通过传入断言参数返回是否所有元素都能匹配上断言。
- `noneMatch(Predicate predicate)`：这是一个短路操作，通过传入断言参数判断是否所有元素都无法匹配上断言，如果是则返回true，反之则false。
- `findFirst()`：这是一个短路操作，返回Stream中的第一个元素，Stream可能为空所以返回值用Optional处理。
- `findAny()`：这是一个短路操作，返回Stream中的任意一个元素，串型流中一般是第一个元素，Stream可能为空所以返回值用Optional处理。

虽然以上都比较简单，但是这里面有五个涉及到短路操作的方法：

首先是`findFirst()`和`findAny()`这两个方法， 由于它们只需要拿到一个元素就能方法就能结束，所以短路效果很好理解。

接着是`anyMatch`方法，它只需要匹配到一个元素方法也能结束，所以它的短路效果也很好理解。

最后是`allMatch`方法和`noneMatch`，乍一看这两个方法都是需要遍历整个流中的所有元素的，其实不然，比如allMatch只要有一个元素不匹配断言它就可以返回false了，noneMatch只要有一个元素匹配上断言它也可以返回false了，所以它们都是具有短路效果的方法。

2. 归约

2.1 reduce：反复求值

下面是归约的定义：

> 将一个Stream中的所有元素反复结合起来，得到一个结果，这样的操作被称为归约。

注：在函数式编程中，这叫做折叠(fold)。

举个很简单的例子，我有1、2、3三个元素，我把它们俩俩相加，最后得出6这个数字，这个过程就是归约。

再比如，我有1、2、3三个元素，我把它们俩俩比较，最后挑出最大的数字3或者挑出最小的数字1，这个过程也是归约。

下面我举一个求和的例子来演示归约，归约使用reduce方法：

```
        Optional<Integer> reduce = List.of(1, 2, 3).stream()
                .reduce((i1, i2) -> i1 + i2);
```

首先你可能注意到了，我在上文的小例子中一直在用俩俩这个词，这代表归约是俩俩的元素进行处理然后得到一个最终值，所以reduce的方法的参数是一个二元表达式，它将两个参数进行任意处理，最后得到一个结果，其中它的参数和结果必须是同一类型。

比如代码中的，i1和i2就是二元表达式的两个参数，它们分别代表元素中的第一个元素和第二个元素，当第一次相加完成后，所得的结果会赋值到i1身上，i2则会继续代表下一个元素，直至元素耗尽，得到最终结果。

如果你觉得这么写不够优雅，也可以使用Integer中的默认方法：

```
        Optional<Integer> reduce = List.of(1, 2, 3).stream()
                .reduce(Integer::sum);
```

这也是一个以`方法引用`代表lambda表达式的例子。

你可能还注意到了，它们的返回值是Optional的，这是预防Stream没有元素的情况。

你也可以想办法去掉这种情况，那就是让元素中至少要有一个值，这里reduce提供一个重载方法给我们：

```
        Integer reduce = List.of(1, 2, 3).stream()
                .reduce(0, (i1, i2) -> i1 + i2);
```

如上例，在二元表达式前面多加了一个参数，这个参数被称为初始值，这样哪怕你的Stream没有元素它最终也会返回一个0，这样就不需要Optional了。

在实际方法运行中，初始值会在第一次执行中占据i1的位置，i2则代表Stream中的第一个元素，然后所得的和再次占据i1的位置，i2代表下一个元素。

不过使用初始值不是没有成本的，它应该符合一个原则：`accumulator.apply(identity, i1) == i1`，也就是说在第一次执行的时候，它的返回结果都应该是你Stream中的第一个元素。

比如我上面的例子是一个相加操作，则第一次相加时就是`0 + 1 = 1`，符合上面的原则，作此原则是为了保证并行流情况下能够得到正确的结果。

如果你的初始值是1，则在并发情况下每个线程的初始化都是1，那么你的最终和就会比你预想的结果要大。

2.2 max：利用归约求最大

max方法也是一个归约方法，它是直接调用了reduce方法。

先来看一个示例：

```
        Optional<Integer> max = List.of(1, 2, 3).stream()
                .max((a, b) -> {
                    if (a > b) {
                        return 1;
                    } else {
                        return -1;
                    }
                });
```

没错，这就是max方法用法，这让我觉得我不是在使用函数式接口，当然你也可以使用Integer的方法进行简化：

```
        Optional<Integer> max = List.of(1, 2, 3).stream()
                .max(Integer::compare);
```

在max方法里面传参数是为了让我们自己自定义排序规则，也有一个默认按照自然排序进行排序的方法。

基础类型Stream：

```
        OptionalLong max = LongStream.of(1, 2, 3).max();
```

果然，我能想到的，类库设计者都想到了~

注：OptionalLong是Optional对基础类型long的封装。

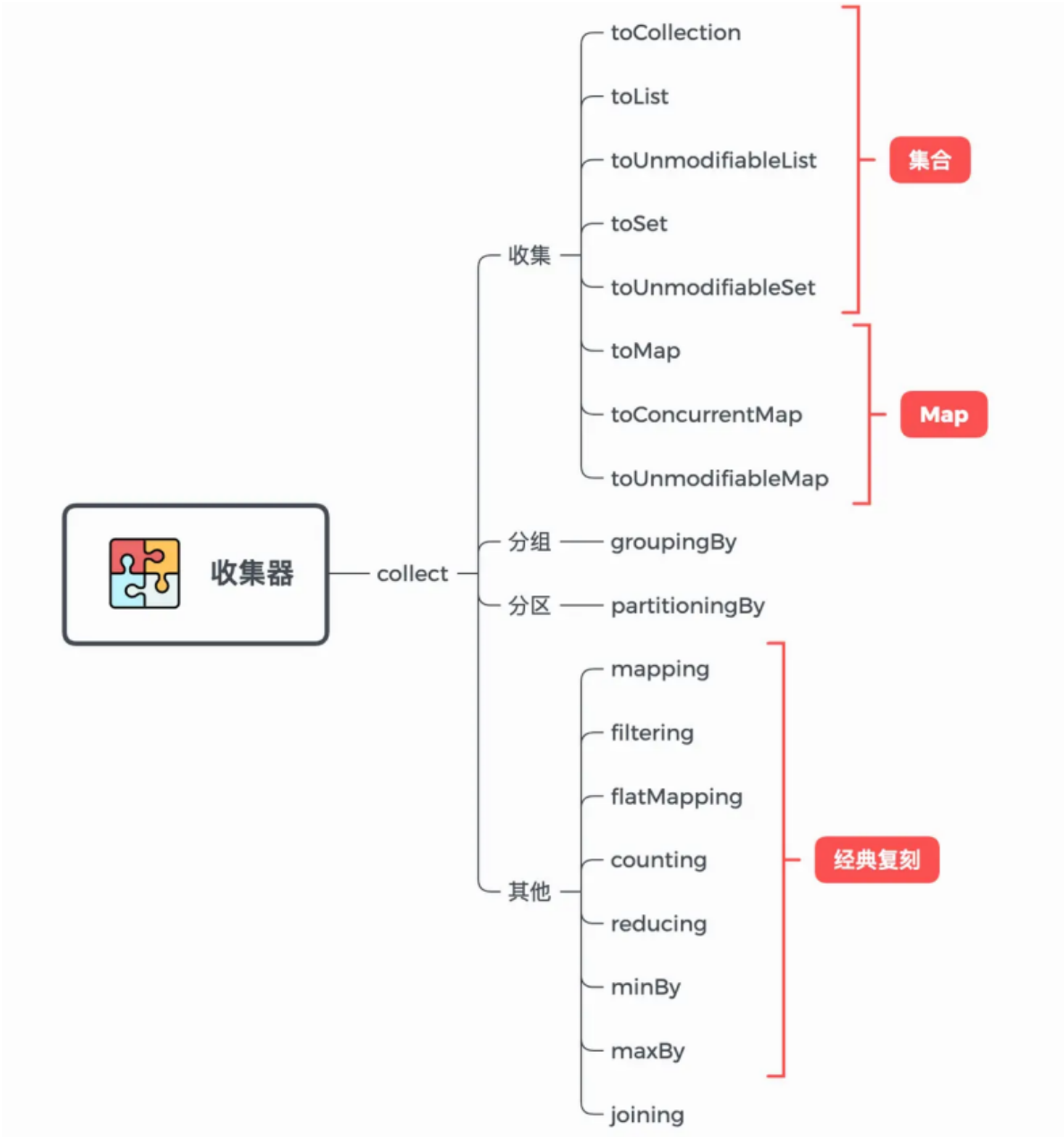
2.3 min：利用归约求最小

min还是直接看例子吧：

```
Optional<Integer> max = List.of(1, 2, 3).stream()
        .min(Integer::compare);
```

它和max区别就是底层把 `>` 换成了 `<`。

3. 收集器



收集器的方法名是collect，它的方法定义如下：

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

顾名思义，收集器是用来收集Stream的元素的，最后收集成什么我们可以自定义，但是我们一般不需要自己写，因为JDK内置了一个Collector的实现类——Collectors。

3.1 收集方法

通过Collectors我们可以利用它的内置方法很方便的进行数据收集：

比如你想把元素收集成集合，那么你可以使用toCollection或者toList方法，不过我们一般不使用toCollection，因为它需要传参数，没人喜欢传参数。

你也可以使用toUnmodifiableList，它和toList区别就是它返回的集合不可以改变元素，比如删除或者新增。

再比如你要把元素去重之后收集起来，那么你可以使用toSet或者toUnmodifiableSet。

接下来放一个比较简单的例子：

```
        // toList
        List.of(1, 2, 3).stream().collect(Collectors.toList());

        // toUnmodifiableList
        List.of(1, 2, 3).stream().collect(Collectors.toUnmodifiableList());

        // toSet
        List.of(1, 2, 3).stream().collect(Collectors.toSet());

        // toUnmodifiableSet
        List.of(1, 2, 3).stream().collect(Collectors.toUnmodifiableSet());
```

以上这些方法都没有参数，拿来即用，toList底层也是经典的ArrayList，toSet 底层则是经典的HashSet。

也许有时候你也许想要一个收集成一个Map，比如通过将订单数据转成一个订单号对应一个订单，那么你可以使用toMap()：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<String, Order> map = orders.stream()
                .collect(Collectors.toMap(Order::getOrderNo, order -> order));
```

toMap() 具有两个参数：

1. 第一个参数代表key，它表示你要设置一个Map的key，我这里指定的是元素中的orderNo。
2. 第二个参数代表value，它表示你要设置一个Map的value，我这里直接把元素本身当作值，所以结果是一个Map<String, Order>。

你也可以将元素的属性当作值：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<String, List<Item>> map = orders.stream()
                .collect(Collectors.toMap(Order::getOrderNo,
Order::getItemList));
```

这样返回的就是一个订单号+商品列表的Map了。

toMap() 还有两个伴生方法：

- toUnmodifiableMap()：返回一个不可修改的Map。
- toConcurrentMap()：返回一个线程安全的Map。

这两个方法和toMap() 的参数一模一样，唯一不同的就是底层生成的Map特性不太一样，我们一般使用简简单单的toMap() 就够了，它的底层是我们最常用的HashMap() 实现。

toMap() 功能虽然强大也很常用，但是它却有一个致命缺点。

我们知道HahsMap遇到相同的key会进行覆盖操作，但是toMap() 方法生成Map时如果你指定的key出现了重复，那么它会直接抛出异常。

比如上面的订单例子中，我们假设两个订单的订单号一样，但是你又将订单号指定了为key，那么该方法会直接抛出一个IllegalStateException，因为它不允许元素中的key是相同的。

所以,toMap又有一个重载方法，第三个参数当key冲突时进行处理,并且官方推荐使用带第三个参数的重载方法

```
toMap(Order::getOrderNo, e -> e , (k1,k2) -> k1)
```

当然toMap还有容易踩到的坑，在toMap的时候key为null会报错，那如果value为null会报错吗？答案是——会NPE！！！那有人就说了，就要这个value为null，还就要用stream().collect()这种高逼格写法，那怎么做呢，可以用如下写法：
HashMap hashMap = personList.stream().collect(HashMap::new, (newMap, entity) -> newMap.put(entity.getName(), entity.getPhone()), HashMap::putAll);

3.2 分组方法

如果你想对数据进行分类，但是你指定的key是可以重复的，那么你应该使用groupingBy 而不是toMap。

举个简单的例子，我想对一个订单集合以订单类型进行分组，那么可以这样：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<Integer, List<Order>> collect = orders.stream()
                .collect(Collectors.groupingBy(Order::getOrderType));
```

直接指定用于分组的元素属性，它就会自动按照此属性进行分组，并将分组的结果收集为一个List。

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<Integer, Set<Order>> collect = orders.stream()
                .collect(Collectors.groupingBy(Order::getOrderType, toSet()));
```

groupingBy还提供了一个重载，让你可以自定义收集器类型，所以它的第二个参数是一个Collector收集器对象。

对于Collector类型，我们一般还是使用Collectors类，这里由于我们前面已经使用了Collectors，所以这里不必声明直接传入一个toSet()方法，代表我们将分组后的元素收集为Set。

groupingBy还有一个相似的方法叫做groupingByConcurrent()，这个方法可以在并行时提高分组效率，但是它是不保证顺序的，这里就不展开讲了。

3.3 分区方法

接下来我将介绍分组的另一种情况——分区，名字有点绕，但意思很简单：

> 将数据按照TRUE或者FALSE进行分组就叫做分区。

举个例子，我们将一个订单集合按照是否支付进行分组，这就是分区：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<Boolean, List<Order>> collect = orders.stream()
                .collect(Collectors.partitioningBy(Order::getIsPaid));
```

因为订单是否支付只具有两种状态：已支付和未支付，这种分组方式我们就叫做分区。

和groupingBy一样，它还具有一个重载方法，用来自定义收集器类型：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<Boolean, Set<Order>> collect = orders.stream()
                .collect(Collectors.partitioningBy(Order::getIsPaid, toSet()));
```

3.4 经典复刻方法

经典复刻,换言之，就是Collectors把Stream原先的方法又实现了一遍，包括：

1. **map** → `mapping`
2. **filter** → `filtering`
3. **flatMap** → `flatMapping`
4. **count** → `counting`
5. **reduce** → `reducing`
6. **max** → `maxBy`
7. **min** → `minBy`

这些方法的功能我就不一一列举了，之前的文章已经讲的很详尽了，唯一的不同是某些方法多了一个参数，这个参数就是我们在分组和分区里面讲过的收集参数，你可以指定收集到什么容器内。

我把它们抽出来主要想说的为什么要复刻这么多方法处理，个人理解的话。

我觉得主要是为了功能的组合。

什么意思呢？比方说我又有一个需求：使用订单类型对订单进行分组，并找出每组有多少个订单。

订单分组我们已经讲过了，找到其每组有多少订单只要拿到对应list的size就行了，但是我们可以不这么麻烦，而是一步到位，在输出结果的时候键值对就是订单类型和订单数量：

```
        Map<Integer, Long> collect = orders.stream()
                .collect(Collectors.groupingBy(Order::getOrderType,
counting()));
```

就这样，就这么简单，就好了，这里等于说我们对分组后的数据又进行了一次计数操作。

上面的这个例子可能不太明显，当我们需要对最后收集之后的数据在进行操作时，一般我们需要重新将其转换成Stream然后操作，但是使用Collectors的这些方法就可以让你很方便的在Collectors中进行数据的处理。

再举个例子，还是通过订单类型对订单进行分组，但是呢，我们想要拿到每种类型订单金额最大的那个，那么我们就可以这样：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<Integer, Optional<Order>> collect2 = orders.stream()
                .collect(groupingBy(Order::getOrderType,
                        maxBy(Comparator.comparing(Order::getMoney))));
```

更简洁，也更方便，不需要我们分组完之后再去一一寻找最大值了，可以一步到位。

再来一个分组之后，求各组订单金额之后的：

```
        List<Order> orders = List.of(new Order(), new Order());

        Map<Integer, Long> collect = orders.stream()
                .collect(groupingBy(Order::getOrderType,
summingLong(Order::getMoney)));
```

不过summingLong这里我们没有讲，它就是一个内置的请和操作，支持Integer、Long和Double。

还有一个类似的方法叫做averagingLong看名字就知道，求平均的，都比较简单。

最后一个方法joining()，用来拼接字符串很实用：

```
        List<Order> orders = List.of(new Order(), new Order());

        String collect = orders.stream()
                .map(Order::getOrderNo).collect(Collectors.joining(","));
```

这个方法的方法名看着有点眼熟，没错，String类在JDK8之后新加了一个`join()`方法，也是用来拼接字符串的，Collectors的joining不过和它功能一样，底层实现也一样，都用了StringJoiner类。