```
mTestObjects = Arrays.asList(
                new StreamTestObject(1, 2),
                new StreamTestObject(3, 4),
                new StreamTestObject(5, 6));
        // 便于观察变化
        Stream<StreamTestObject> stream = streamTestObjects.stream();
        Stream<Integer> id1Stream =
streamTestObjects.stream().map(StreamTestObject::getId1);
```

```
```


看代码我们可以看到，map方法将对象的stream流映射为了其中 id1 这个字段的stream流

拿到这个字段的流后，可以做些什么呢?

最常用的方法之一就是与 集合 **collect()** 搭配起来使用。

那么 **collect()** 方法能做写什么呢?

* **用途**：将流中的元素累积成一个汇总结果，我们可以按照自己的需求将结果汇总为一个 List、Set、Map 等

如下代码所示

```
```

```
        List<StreamTestObject> streamTestObjects = Arrays.asList(
                new StreamTestObject(1, 2),
                new StreamTestObject(3, 4),
                new StreamTestObject(5, 6));
```

```
        Stream<StreamTestObject> stream = streamTestObjects.stream();
        Stream<Integer> id1Stream =
streamTestObjects.stream().map(StreamTestObject::getId1);

        List<Integer> collectList = id1Stream.collect(Collectors.toList());
//        Set<Integer> collectSet = id1Stream.collect(Collectors.toSet());

        // 连起来使用一行代码可以写成这样
        collectList =
streamTestObjects.stream().map(StreamTestObject::getId1).collect(Colle
ctors.toList());
//        collectSet =
streamTestObjects.stream().map(StreamTestObject::getId1).collect(Colle
ctors.toSet());
        System.out.println("collectList:" + collectList);
// 输出结果 collectList:[1, 3, 5]
```


结果能够把 id1 成功收集起来，代码的易读性也体现在其中。我们一眼就能看出这行代码映射了id1 这个字段为一个 **List** 或 **Set**。

> ###### 多字段映射

那如果我们想要对象集合中的 id1 和 id2 都汇总到一个 List 集合里，应该如何操作呢。

这里我们可以使用一个 **flatMap()** 方法

* **用途**：将流中的每个元素都转换成另一个流，然后将所有流连接成一个流。

直接上代码

```
        List<StreamTestObject> streamTestObjects = Arrays.asList(
                new StreamTestObject(1, 2),
```

```
                new StreamTestObject(3, 4),
                new StreamTestObject(5, 6));
        List<Integer> collectList = streamTestObjects.stream()
                .flatMap(object ->Stream.of(object.getId1(),
object.getId2())).collect(Collectors.toList());
        System.out.println("collec            .filter(object -> {
                    int id1 = 1;
                    int id2 = 4;
                    return object.getId1().equals(id1) ||
object.getId2().equals(id2);
                }).collect(Collectors.toList());
        System.out.println("collectList:" + collectList);
        //输出结果 collectList:[StreamTestObject(id1=1, id2=2),
StreamTestObject(id1=3, id2=4)]
```

```




在代码块里可以编辑自己**自定义的过滤逻辑**

这里要注意返回值是一个布尔值，如果为 true，则保留这项数据，不满足，则进行一项数据处理。

> #### Stream流的其他方法

前文是 Stream 流比较常见的方法案例，它还提供了很多其他的接口来实现对应的场景，如：

1. **sorted()**

* **用途**：对流中的元素进行自然排序（需实现 Comparable 接口），返回排序后的 Stream。
* **示例**：对用户列表按年龄进行排序。
2. **limit(long maxSize)**

* **用途**：截断流，使其包含不超过给定数量的元素，返回截断后的 Stream。
* **示例**：只取用户列表中的前三个用户。
3. **skip(long n)**
```

* **用途**：跳过流中的前 n 个元素，返回剩下的元素的 Stream。
* **示例**：跳过用户列表中的前两个用户，取后面的用户。
4. **forEach(Consumer<? super T> action)**

* **用途**：这是大家比较熟悉的操作，在代码编写中可以省去 .Stream() 的写法。意为对流中的每个元素执行提供的操作，这是一个终结操作。
* **示例**：遍历用户列表并打印每个用户的名字。

> ### 使用Stream流的弊端

学习了Stream流 的优点之后，也需要知道随之产生的弊端有短些，这里我列举几个主要的内容

* **性能问题**：

1. **多次遍历**：有时为了完成一个操作，可能需要多次遍历数据源。例如，先过滤（**filter**）再映射（**map**）最后收集（**collect**），这会导致数据被多次遍历。
2. **并行流开销**：虽然并行流可以加速处理过程，但它们引入了额外的线程管理开销，并且不总是能带来性能提升，尤其是在数据源较小或操作相对简单时。
3. **懒加载导致的意外行为**：Stream操作是懒加载的，这意味着它们直到需要结果时才会执行。这可能导致在调试时难以追踪问题的源头，或者在某些情况下，当流操作依赖于外部状态时，可能导致不可预测的行为。

---

* **可读性和维护性**：

？？：前面不是可读性强吗，怎么有问题了？如果嵌套太多层的操作方法，也会使得表达式的可读性降低

1. **复杂逻辑难以追踪**：对于包含多个复杂操作（如多重过滤、映射、归约等）的Stream链，其逻辑可能变得难以理解和追踪。

2. **调试困难**：由于Stream操作的延迟执行和中间操作的无状态性，调试Stream代码可能会比传统循环更加困难。

---

* **错误处理**：

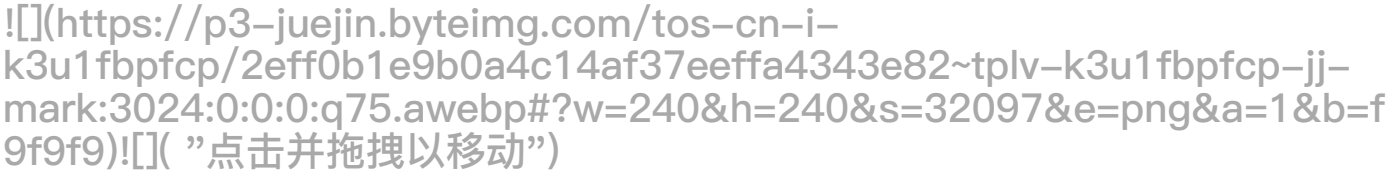
1. **异常处理复杂**：在Stream操作中处理异常（如尝试映射一个可能抛出异常的函数）比在传统循环中更复杂。Stream API没有直接支持异常处理机制，通常需要通过try-catch块或自定义函数来处理。

---

* **内存消耗**：

1. **中间结果存储**：Stream API在执行过程中可能会创建中间结果的临时集合，尤其是在进行复杂操作时，这可能会增加内存消耗。

到这里，同学们可以多实操一下这些方法来巩固知识。文章如有遗漏或建议更改的部分欢迎佬们指出。



原文链接: https://juejin.cn/post/7388064351503843343