Guava中这些Map的骚操作，让我的代码量减少了50%！

macrozheng 2022-03-18 09:02

以下文章来源于码农参上 ，作者Dr Hydra

码农参上

专注后端技术分享，有趣、深入、直接，与你聊聊技术。

Guava是google公司开发的一款Java类库扩展工具包，内含了丰富的API，涵盖了集合、缓存、并发、I/O等多个方面。使用这些API一方面可以简化我们代码，使代码更为优雅，另一方面它补充了很多jdk中没有的功能，能让我们开发中更为高效。

今天Hydra要给大家分享的就是Guava中封装的一些关于 `Map` 的骚操作，在使用了这些功能后，不得不说一句真香。先引入依赖坐标，然后开始我们的正式体验吧~

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>30.1.1-jre</version>
</dependency>
```

Table - 双键Map

java中的 `Map` 只允许有一个 `key` 和一个 `value` 存在，但是guava中的 `Table` 允许一个 `value` 存在两个 `key` 。 `Table` 中的两个 `key` 分别被称为 `rowKey` 和 `columnKey` ，也就是行和列。（但是个人感觉将它们理解为行和列并不是很准确，看作两列的话可能会更加合适一些）

举一个简单的例子，假如要记录员工每个月工作的天数。用java中普通的 `Map` 实现的话就需要两层嵌套：

```
Map<String,Map<String,Integer>> map=new HashMap<>();
//存放元素
Map<String,Integer> workMap=new HashMap<>();
workMap.put("Jan",20);
workMap.put("Feb",28);
```

```
map.put("Hydra",workMap);

//取出元素
Integer dayCount = map.get("Hydra").get("Jan");
```

如果使用 `Table` 的话就很简单了，看一看简化后的代码：

```
Table<String,String,Integer> table= HashBasedTable.create();
//存放元素
table.put("Hydra", "Jan", 20);
table.put("Hydra", "Feb", 28);

table.put("Trunks", "Jan", 28);
table.put("Trunks", "Feb", 16);

//取出元素
Integer dayCount = table.get("Hydra", "Feb");
```

我们不需要再构建复杂的双层 `Map` ，直接一层搞定。除了元素的存取外，下面再看看其他的实用操作。

1、获得key或value的集合

```
//rowKey或columnKey的集合
Set<String> rowKeys = table.rowKeySet();
Set<String> columnKeys = table.columnKeySet();

//value集合
Collection<Integer> values = table.values();
```

分别打印它们的结果， `key` 的集合是不包含重复元素的， `value` 集合则包含了所有元素并没有去重：

```
[Hydra, Trunks]
[Jan, Feb]
[20, 28, 28, 16]
```

2、计算key对应的所有value的和

以统计所有 `rowKey` 对应的 `value` 之和为例：

```
for (String key : table.rowKeySet()) {
    Set<Map.Entry<String, Integer>> rows = table.row(key).entrySet();
    int total = 0;
    for (Map.Entry<String, Integer> row : rows) {
        total += row.getValue();
    }
    System.out.println(key + ": " + total);
}
```

打印结果：

```
Hydra: 48
Trunks: 44
```

3、转换rowKey和columnKey

这一操作也可以理解为行和列的转置，直接调用 `Tables` 的静态方法 `transpose` ：

```
Table<String, String, Integer> table2 = Tables.transpose(table);
Set<Table.Cell<String, String, Integer>> cells = table2.cellSet();
cells.forEach(cell->
    System.out.println(cell.getRowKey()+","+cell.getColumnKey()+":"+cell.getValue())
);
```

利用 `cellSet` 方法可以得到所有的数据行，打印结果，可以看到 `row` 和 `column` 发生了互换：

```
Jan,Hydra:20
Feb,Hydra:28
Jan,Trunks:28
Feb,Trunks:16
```

4、转为嵌套的Map

还记得我们在没有使用 `Table` 前存储数据的格式吗，如果想要将数据还原成嵌套 `Map` 的那种形式，使用 `Table` 的 `rowMap` 或 `columnMap` 方法就可以实现了：

```
Map<String, Map<String, Integer>> rowMap = table.rowMap();
Map<String, Map<String, Integer>> columnMap = table.columnMap();
```

查看转换格式后的 `Map` 中的内容，分别按照行和列进行了汇总：

```
{Hydra={Jan=20, Feb=28}, Trunks={Jan=28, Feb=16}}
{Jan={Hydra=20, Trunks=28}, Feb={Hydra=28, Trunks=16}}
```

BiMap - 双向Map

在普通 `Map` 中，如果要想根据 `value` 查找对应的 `key` ，没什么简便的办法，无论是使用 `for` 循环还是迭代器，都需要遍历整个 `Map` 。以循环 `keySet` 的方式为例：

```
public List<String> findKey(Map<String, String> map, String val){
    List<String> keys=new ArrayList<>();
    for (String key : map.keySet()) {
        if (map.get(key).equals(val))
            keys.add(key);
    }
    return keys;
}
```

而guava中的 `BiMap` 提供了一种 `key` 和 `value` 双向关联的数据结构，先看一个简单的例子：

```
HashBiMap<String, String> biMap = HashBiMap.create();
biMap.put("Hydra","Programmer");
biMap.put("Tony","IronMan");
biMap.put("Thanos","Titan");
//使用key获取value
System.out.println(biMap.get("Tony"));
```

```
BiMap<String, String> inverse = biMap.inverse();
//使用value获取key
System.out.println(inverse.get("Titan"));
```

执行结果，：

```
IronMan
Thanos
```

看上去很实用是不是？但是使用中还有几个坑得避一下，下面一个个梳理。

1、反转后操作的影响

上面我们用 `inverse` 方法反转了原来 `BiMap` 的键值映射，但是这个反转后的 `BiMap` 并不是一个新的对象，它实现了一种视图的关联，所以对反转后的 `BiMap` 执行的所有操作会作用于原先的 `BiMap` 上。

```
HashBiMap<String, String> biMap = HashBiMap.create();
biMap.put("Hydra","Programmer");
biMap.put("Tony","IronMan");
biMap.put("Thanos","Titan");
BiMap<String, String> inverse = biMap.inverse();

inverse.put("IronMan","Stark");
System.out.println(biMap);
```

对反转后的 `BiMap` 中的内容进行了修改后，再看一下原先 `BiMap` 中的内容：

```
{Hydra=Programmer, Thanos=Titan, Stark=IronMan}
```

可以看到，原先值为 `IronMan` 时对应的键是 `Tony` ，虽然没有直接修改，但是现在键变成了 `Stark` 。

2、value不可重复

`BiMap` 的底层继承了 `Map` ，我们知道在 `Map` 中 `key` 是不允许重复的，而双向的 `BiMap` 中 `key` 和 `value` 可以认为处于等价地位，因此在这个基础上加了限制， `value` 也是不允许重复的。看一下下面的代码：

```
HashBiMap<String, String> biMap = HashBiMap.create();
biMap.put("Tony","IronMan");
biMap.put("Stark","IronMan");
```

这样代码无法正常结束，会抛出一个 `IllegalArgumentException` 异常：

如果你非想把新的 `key` 映射到已有的 `value` 上，那么也可以使用 `forcePut` 方法强制替换掉原有的 `key` ：

```
HashBiMap<String, String> biMap = HashBiMap.create();
biMap.put("Tony","IronMan");
biMap.forcePut("Stark","IronMan");
```

打印一下替换后的 `BiMap` ：

```
{Stark=IronMan}
```

顺带多说一句，由于 `BiMap` 的 `value` 是不允许重复的，因此它的 `values` 方法返回的是没有重复的 `Set` ，而不是普通 `Collection` ：

```
Set<String> values = biMap.values();
```

Multimap - 多值Map

java中的 `Map` 维护的是键值一对一的关系，如果要将一个键映射到多个值上，那么就只能把值的内容设为集合形式，简单实现如下：

```
Map<String, List<Integer>> map=new HashMap<>();
List<Integer> list=new ArrayList<>();
list.add(1);
list.add(2);
map.put("day",list);
```

guava中的 `Multimap` 提供了将一个键映射到多个值的形式，使用起来无需定义复杂的内层集合，可以像使用普通的 `Map` 一样使用它，定义及放入数据如下：

```
Multimap<String, Integer> multimap = ArrayListMultimap.create();
multimap.put("day",1);
multimap.put("day",2);
multimap.put("day",8);
multimap.put("month",3);
```

打印这个 `Multimap` 的内容，可以直观的看到每个 `key` 对应的都是一个集合：

```
{month=[3], day=[1, 2, 8]}
```

1、获取值的集合

在上面的操作中，创建的普通 `Multimap` 的 `get(key)` 方法将返回一个 `Collection` 类型的集合：

```
Collection<Integer> day = multimap.get("day");
```

如果在创建时指定为 `ArrayListMultimap` 类型，那么 `get` 方法将返回一个 `List` ：

```
ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.create();
List<Integer> day = multimap.get("day");
```

同理，你还可以创建 `HashMultimap` 、 `TreeMultimap` 等类型的 `Multimap` 。

`Multimap` 的 `get` 方法会返回一个非 `null` 的集合，但是这个集合的内容可能是空，看一下下面的例子：

```
List<Integer> day = multimap.get("day");
List<Integer> year = multimap.get("year");
System.out.println(day);
System.out.println(year);
```

打印结果：

```
[1, 2, 8]
[]
```

2、操作get后的集合

和 `BiMap` 的使用类似，使用 `get` 方法返回的集合也不是一个独立的对象，可以理解为集合视图的关联，对这个新集合的操作仍然会作用于原始的 `Multimap` 上，看一下下面的例子：

```
ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.create();
multimap.put("day",1);
multimap.put("day",2);
multimap.put("day",8);
multimap.put("month",3);

List<Integer> day = multimap.get("day");
List<Integer> month = multimap.get("month");

day.remove(0);//这个0是下标
month.add(12);
System.out.println(multimap);
```

查看修改后的结果：

```
{month=[3, 12], day=[2, 8]}
```

3、转换为Map

使用 `asMap` 方法，可以将 `Multimap` 转换为 `Map<K,Collection>` 的形式，同样这个 `Map` 也可以看做一个关联的视图，在这个 `Map` 上的操作会作用于原始的 `Multimap` 。

```
Map<String, Collection<Integer>> map = multimap.asMap();
for (String key : map.keySet()) {
    System.out.println(key+" : "+map.get(key));
}
map.get("day").add(20);
System.out.println(multimap);
```

执行结果：

```
month : [3]
day : [1, 2, 8]
{month=[3], day=[1, 2, 8, 20]}
```

4、数量问题

`Multimap` 中的数量在使用中也有些容易混淆的地方，先看下面的例子：

```
System.out.println(multimap.size());
System.out.println(multimap.entries().size());
for (Map.Entry<String, Integer> entry : multimap.entries()) {
    System.out.println(entry.getKey()+","+entry.getValue());
}
```

打印结果：

```
4
4
month,3
day,1
```

```
day,2
day,8
```

这是因为 `size()` 方法返回的是所有 `key` 到单个 `value` 的映射，因此结果为4， `entries()` 方法同理，返回的是 `key` 和单个 `value` 的键值对集合。但是它的 `keySet` 中保存的是不同的 `key` 的个数，例如下面这行代码打印的结果就会是2。

```
System.out.println(multimap.keySet().size());
```

再看看将它转换为 `Map` 后，数量则会发生变化：

```
Set<Map.Entry<String, Collection<Integer>>> entries = multimap.asMap().entrySet();
System.out.println(entries.size());
```

代码运行结果是2，因为它得到的是 `key` 到 `Collection` 的映射关系。

RangeMap - 范围Map

先看一个例子，假设我们要根据分数对考试成绩进行分类，那么代码中就会出现这样丑陋的 `if-else` ：

```
public static String getRank(int score){
    if (0<=score && score<60)
        return "fail";
    else if (60<=score && score<=90)
        return "satisfactory";
    else if (90<score && score<=100)
        return "excellent";
    return null;
}
```

而guava中的 `RangeMap` 描述了一种从区间到特定值的映射关系，让我们能够以更为优雅的方法来书写代码。下面用 `RangeMap` 改造上面的代码并进行测试：

```
RangeMap<Integer, String> rangeMap = TreeRangeMap.create();
rangeMap.put(Range.closedOpen(0,60),"fail");
rangeMap.put(Range.closed(60,90),"satisfactory");
rangeMap.put(Range.openClosed(90,100),"excellent");

System.out.println(rangeMap.get(59));
System.out.println(rangeMap.get(60));
System.out.println(rangeMap.get(90));
System.out.println(rangeMap.get(91));
```

在上面的代码中，先后创建了 `[0,60)` 的左闭右开区间、 `[60,90]` 的闭区间、 `(90,100]` 的左开右闭区间，并分别映射到某个值上。运行结果打印：

```
fail
satisfactory
satisfactory
excellent
```

当然我们也可以移除一段空间，下面的代码移除了 `[70,80]` 这一闭区间后，再次执行 `get` 时返回结果为 `null` ：

```
rangeMap.remove(Range.closed(70,80));
System.out.println(rangeMap.get(75));
```

ClassToInstanceMap - 实例Map

`ClassToInstanceMap` 是一个比较特殊的 `Map` ，它的键是 `Class` ，而值是这个 `Class` 对应的实例对象。先看一个简单使用的例子，使用 `putInstance` 方法存入对象：

```
ClassToInstanceMap<Object> instanceMap = MutableClassToInstanceMap.create();
User user=new User("Hydra",18);
Dept dept=new Dept("develop",200);

instanceMap.putInstance(User.class,user);
instanceMap.putInstance(Dept.class,dept);
```

使用 getInstance 方法取出对象：

```
User user1 = instanceMap.getInstance(User.class);
System.out.println(user==user1);
```

运行结果打印了 true ，说明了取出的确实是我们之前创建并放入的那个对象。

大家可能会疑问，如果只是存对象的话，像下面这样用普通的 Map<Class,Object> 也可以实现：

```
Map<Class,Object> map=new HashMap<>();
User user=new User("Hydra",18);
Dept dept=new Dept("develop",200);
map.put(User.class,user);
map.put(Dept.class,dept);
```

那么，使用 ClassToInstanceMap 这种方式有什么好处呢？

首先，这里最明显的就是在取出对象时省去了复杂的强制类型转换，避免了手动进行类型转换的错误。其次，我们可以看一下 ClassToInstanceMap 接口的定义，它是带有泛型的：

```
public interface ClassToInstanceMap<B> extends Map<Class<? extends B>, B>{...}
```

这个泛型同样可以起到对类型进行约束的作用， value 要符合 key 所对应的类型，再看看下面的例子：

```
ClassToInstanceMap<Map> instanceMap = MutableClassToInstanceMap.create();
HashMap<String, Object> hashMap = new HashMap<>();
TreeMap<String, Object> treeMap = new TreeMap<>();
ArrayList<Object> list = new ArrayList<>();

instanceMap.putInstance(HashMap.class,hashMap);
instanceMap.putInstance(TreeMap.class,treeMap);
```

这样是可以正常执行的，因为 `HashMap` 和 `TreeMap` 都集成了 `Map` 父类，但是如果想放入其他类型，就会编译报错：

所以，如果你想缓存对象，又不想做复杂的类型校验，那么使用方便的 `ClassToInstanceMap` 就可以了。

总结

本文介绍了guava中5种对 `Map` 的扩展数据结构，它们提供了非常实用的功能，能很大程度的简化我们的代码。但是同时使用中也有不少需要避开的坑，例如修改关联的视图会对原始数据造成影响等等，具体的使用中大家还需要谨慎一些。

推荐阅读

- 升级 **SpringBoot 2.6.x** 版本后，**Swagger** 没法用了！
- 想要逃离北上广了！
- **2022**数据库排行榜新鲜出炉！**MySQL**大势已去，**PostgreSQL**即将崛起！
- 还在用 **RedisTemplate**？试试 **Redis** 官方 **ORM** 框架吧，用起来够优雅！
- 比 **Elasticsearch** 更快！**RediSearch + RedisJSON =** 王炸！
- 颜值爆表！**Redis** 官方可视化工具来啦，功能真心强大！

- **40K+Star！Mall电商实战项目开源回忆录！**
- **mall-swarm 微服务电商项目发布重大更新，打造Spring Cloud最佳实践！**

macrozheng

专注Java技术分享，涵盖SpringBoot、SpringCloud、Docker、中间件等实用技术，...

211篇原创内容

公众号

喜欢此内容的人还喜欢

低代码平台边界探索：多技术栈支持及高低代码混合开发

InfoQ

老板：你来弄一个团队代码规范！？

程序员黑叔