目录

14 简化工作：Guava Lists Maps 实际工作运用和源码

更新时间：2019-09-24 10:32:00

> 如果不想在世界上虚度一生，那就要学习一辈子。
>
> ——高尔基

引导语

在日常工作中，我们经常会使用一些三方的 API 来简化我们的工作，Guava 就是其中一种，Guava 是 Google 开源的技术框架，使用率高，社区活跃度也很高。

本小节我们从工作中对 Guava 集合的使用入手，然后深入的看下其底层的实现，最后总结其设计思想，感兴趣的同学也可以下载源码学习，GitHub 地址：https://github.com/google/guava，源码中 guava 的文件夹为其源码。

1 运用工厂模式进行初始化

在集合类初始化方面，Guava 比 Java 原生的 API 更加好用，还发明了很多新的功能，比如说在 JDK 7 之前，我们新建集合类时，声明和初始化都必须写上泛型说明，像这样：`List<泛型> list = new ArrayList<泛型>();`，JDK 7 之后有所改变，我们只需要在声明处写上泛型说明，像这样：`List<泛型> list = new ArrayList<>();`。

Guava 提供了更加方便的使用姿势，采用了工厂模式，把集合创建的逻辑交给了工厂，开发者无需关注工厂底层是如何创建的，只需要关心，工厂能产生什么，代码于是变成了这样：`List<泛型> list = Lists.newArrayList();`，Lists 就是 Guava 提供出来的，方便操作 List 的工具类。

这种写法其实就是一种简单的工厂模式，只需要定义好工厂的入参和出参，就能对外隐藏其内部的创建逻辑，提供更加方便的使用体验。

2 Lists

2.1 初始化

Lists 最大的功能是能帮助我们进行 List 的初始化，比如我们刚说的 newArrayList 这种：

```
List<String> list = Lists.newArrayList();
public static <E> ArrayList<E> newArrayList() {
  return new ArrayList<>();
}
// 这种底层是帮助我们写好了泛型，E 代表泛型，表示当前返回的泛型类型和声明的一致即可，在编
```

如果你清楚 List 的大小，我们也可以这样做：

```
// 可以预估 list 的大小为 20
List<String> list = Lists.newArrayListWithCapacity(20);
// 不太肯定 list 大小是多少，但期望是大小是 20 上下。
List<String> list = Lists.newArrayListWithExpectedSize(20);
```

newArrayListWithCapacity(20) 方法内部实现是：`new ArrayList<>(20);`，而 newArrayListWithExpectedSize 方法内部实现是对 List 大小有一个计算公式的，计算公式为：`5L + arraySize + (arraySize / 10)`，arraySize 表示传进来的值，公式简化下就是 5 + 11/10 * arraySize，因为这个方法表示期望的大小，所以这里取的约是期望值的十分之十一，比传进来的值约大十分之一，所以根据 20 最终计算出来的值是 27。

Lists 在初始化的时候，还支持传迭代器的入参（只适合小数据量的迭代器的入参），源码如下：

```
public static <E> ArrayList<E> newArrayList(Iterator<? extends E> elements) {
  ArrayList<E> list = newArrayList();
  // addAll 方法底层其实通过迭代器进行 for 循环添加
  Iterators.addAll(list, elements);
  return list;
}
```

从 Lists 对 List 初始化进行包装的底层源码来看，底层源码非常简单的，但我们还是愿意使用这种方式的包装，主要是因为这种工厂模式的包装，使我们的使用姿势更加优雅，使用起来更加方便。

2.2 分组和反转排序

除了初始化之外，Lists 还提供了两个比较实用的功能，分组和反转排序功能，我们分别来演示一下：

```
// 演示反转排序
public void testReverse(){
  List<String> list = new ArrayList<String>(){{
    add("10");
    add("20");
    add("30");
    add("40");
```

```
  log.info("反转之后： "+JSON.toJSONString(list));
}
// 打印出来的结果为：
反转之前：["10","20","30","40"]
反转之后：["40","30","20","10"]
```

reverse 方法底层实现非常巧妙，底层覆写了 List 原生的 get(index) 方法，会把传进来的 index 进行 (size - 1) - index 的计算，使计算得到的索引位置和 index 位置正好相反，这样当我们 get 时，数组索引位置的 index 已经是相反的位置了，达到了反转排序的效果，其实底层并没有进行反转排序，只是在计算相反的索引位置，通过计算相反的索引位置这样简单的设计，得到了反转排序的效果，很精妙。

在工作中，有时候我们需要把一个大的 list 进行切分，然后再把每份丢给线程池去运行，最后将每份运行的结果汇总，Lists 工具类就提供了一个对 list 进行切分分组的方法，演示 demo 如下：

```
// 分组
public void testPartition(){
  List<String> list = new ArrayList<String>(){{
    add("10");
    add("20");
    add("30");
    add("40");
  }};
  log.info("分组之前： "+JSON.toJSONString(list));
   List<List<String>> list2 = Lists.partition(list,3);
  log.info("分组之后： "+JSON.toJSONString(list2));
}
输出结果为：
分组之前：["10","20","30","40"]
分组之后：[["10","20","30"],["40"]]
```

partition 方法的第二个参数的意思，你想让分组后的 List 包含几个元素，这个方法的底层实现其实就是 subList 方法。

有一点需要我们注意的是这两个方法返回的 List 并不是 ArrayList，是自定义的 List，所以对于 ArrayList 的有些功能可能并不支持，使用的时候最好能看下源码，看看底层有无支持。

2.3 小结

Lists 上述的方法大大的方便了我们进行开发，简化了使用姿势，但其内部实现却非常简单巧妙，比如说 reverse 方法可以输出相反排序的 List，但底层并没有实现排序，只是计算了索引位置的相反值而已，这点值得我们学习。

3 Maps

3.1 初始化

Maps 也是有着各种初始化 Map 的各种方法，原理不说了，和 Lists 类似，我们演示下如何使用：

```
Map<String,String> hashMap = Maps.newHashMap();
Map<String,String> linkedHashMap = Maps.newLinkedHashMap();
```

3.2 difference

Maps 提供了一个特别有趣也很实用的方法：difference，此方法的目的是比较两个 Map 的差异，入参就是两个 Map，比较之后能够返回四种差异：

1. 左边 Map 独有 key。
2. 右边 Map 独有 key。
3. 左右边 Map 都有 key，并且 value 相等。
4. 左右边 Map 都有 key，但是 value 不等。

我们用代码来演示一下：

```
// ImmutableMap.of 也是 Guava 提供初始化 Map 的方法，入参格式为 k1,v1,k2,v2,k3,v3……
Map<String,String> leftMap = ImmutableMap.of("1","1","2","2","3","3");
Map<String,String> rightMap = ImmutableMap.of("2","2","3","30","4","4");
MapDifference difference = Maps.difference(leftMap, rightMap);
log.info("左边 map 独有 key：{}",difference.entriesOnlyOnLeft());
log.info("右边 map 独有 key：{}",difference.entriesOnlyOnRight());
log.info("左右边 map 都有 key，并且 value 相等：{}",difference.entriesInCommon());
log.info("左右边 map 都有 key，但 value 不等：{}",difference.entriesDiffering());
最后打印结果为：
左边 map 独有 key：{1=1}
右边 map 独有 key：{4=4}
左右边 map 都有 key，并且 value 相等：{2=2}
左右边 map 都有 key，但 value 不等：{3=(3, 30)}
```

从这个 demo 我们可以看到此方法的强大威力，我们在工作中经常遇到 Map 或者 List 间比较差异的任务，我们就可以直接使用该方法进行对比，List 可以先转化成 Map。

而且 difference 底层的实现也算是最优的实现了，只需要循环一遍，就可得到上述四种差异结果，源码解析如下：

```
// 对比两个 map 的差异
private static <K, V> void doDifference(
    Map<? extends K, ? extends V> left,
    Map<? extends K, ? extends V> right,
    Equivalence<? super V> valueEquivalence,
    // key 只在左边 map 出现
    Map<K, V> onlyOnLeft,
    // key 只在右边 map 出现，调用 doDifference 方法前已经包含了全部右边的值
    Map<K, V> onlyOnRight,
    // key 在左右 map 中都出现过，并且 value 都相等
    Map<K, V> onBoth,
    // key 在左右 map 中都出现过，但 value 不等
    Map<K, MapDifference.ValueDifference<V>> differences) {
  // 以左边 map 为基准进行循环
  for (Entry<? extends K, ? extends V> entry : left.entrySet()) {
    K leftKey = entry.getKey();
    V leftValue = entry.getValue();
    // 右边 map 包含左边的 key
    if (right.containsKey(leftKey)) {
      // onlyOnRight 已经包含全部右边的值 所以需要删除当前 key
      V rightValue = onlyOnRight.remove(leftKey);
      // key 相等，并且 value 值也相等
      if (valueEquivalence.equivalent(leftValue, rightValue)) {
```

```
      differences.put(leftKey, ValueDifferenceImpl.create(leftValue, rightValue));
    }
   // 右边 map 不包含左边的 key，就是左边 map 独有的 key
   } else {
    onlyOnLeft.put(leftKey, leftValue);
   }
  }
}
```

这是一种比较优秀的，快速比对的算法，可以好好看下上面的源码，然后把这种算法背下来，或者自己再次实现一次。

Sets 的使用方式和 Lists 和 Maps 很类似，没有太大的亮点，我们就不说了。

4 总结

这一小节主要都是实战内容，在实际工作中可以用起来。

在 Guava 对集合的设计中，有两个大点是非常值得我们学习的：

1. Lists、Maps 的出现给我们提供了更方便的使用姿势和方法，我们在实际工作中，如果碰到特别繁琐，或者特别难用的 API，我们也可以进行一些包装，使更好用，这个是属于在解决目前的痛点的问题上进行创新，是非常值得提倡的一件事情，往往可以帮助你拿到更好的绩效。
2. 如果有人问你，List 或者 Map 高效的差异排序算法，完全可以参考 Maps.difference 的内部实现，该方法只使用了一次循环，就可得到所有的相同或不同结果，这种算法在我们工作中也经常被使用。

了解更多，可以直接前往 Guava 的代码库查看：https://github.com/google/guava

← 13 差异对比：集合在 Java 7 和 8 有何不同和改进

15 CopyOnWriteArrayList 源码解析和设计思路 →

精选留言 2

欢迎在这里发表留言，作者筛选后可公开显示

南门祥老大

不错不错

0 回复 2019-12-09

937587592

讲解的很好，简明扼要！！

0 回复 2019-12-03

回复 2019-12-08 13:53:26

目录

千学不如一看，千看不如一练