

82 | 开源实战三（中）：剖析Google Guava中用到的几种设计模式

2020-05-11 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 11:08 大小 10.20M



上一节课，我们通过 Google Guava 这样一个优秀的开源类库，讲解了如何在业务开发中，发现跟业务无关、可以复用的通用功能模块，并将它们从业务代码中抽离出来，设计开发成独立的类库、框架或功能组件。

今天，我们再来学习一下，Google Guava 中用到的几种经典设计模式：Builder 模式、Wrapper 模式，以及之前没讲过的 Immutable 模式。



话不多说，让我们正式开始今天的学习吧！

Builder 模式在 Guava 中的应用

在项目开发中，我们经常用到缓存。它可以非常有效地提高访问速度。

常用的缓存系统有 Redis、Memcache 等。但是，如果要缓存的数据比较少，我们完全没必要在项目中独立部署一套缓存系统。毕竟系统都有一定出错的概率，项目中包含的系统越多，那组合起来，项目整体出错的概率就会升高，可用性就会降低。同时，多引入一个系统就要多维护一个系统，项目维护的成本就会变高。

取而代之，我们可以在系统内部构建一个内存缓存，跟系统集成在一起开发、部署。那如何构建内存缓存呢？我们可以基于 JDK 提供的类，比如 HashMap，从零开始开发内存缓存。不过，从零开发一个内存缓存，涉及的工作就会比较多，比如缓存淘汰策略等。为了简化开发，我们就可以使用 Google Guava 提供的现成的缓存工具类 `com.google.common.cache.*`。

使用 Google Guava 来构建内存缓存非常简单，我写了一个例子贴在了下面，你可以看下。

 复制代码

```
1 public class CacheDemo {
2     public static void main(String[] args) {
3         Cache<String, String> cache = CacheBuilder.newBuilder()
4             .initialCapacity(100)
5             .maximumSize(1000)
6             .expireAfterWrite(10, TimeUnit.MINUTES)
7             .build();
8
9         cache.put("key1", "value1");
10        String value = cache.getIfPresent("key1");
11        System.out.println(value);
12    }
13 }
```


从上面的代码中，我们可以发现，Cache 对象是通过 CacheBuilder 这样一个 Builder 类来创建的。为什么要由 Builder 类来创建 Cache 对象呢？我想这个问题应该对你来说没难度了吧。

你可以先想一想，然后再来看我的回答。构建一个缓存，需要配置 n 多参数，比如过期时间、淘汰策略、最大缓存大小等等。相应地，Cache 类就会包含 n 多成员变量。我们需要

在构造函数中，设置这些成员变量的值，但又不是所有的值都必须设置，设置哪些值由用户来决定。为了满足这个需求，我们就需要定义多个包含不同参数列表的构造函数。

为了避免构造函数的参数列表过长、不同的构造函数过多，我们一般有两种解决方案。其中，一个解决方案是使用 Builder 模式；另一个方案是先通过无参构造函数创建对象，然后再通过 setXXX() 方法来逐一设置需要的设置的成员变量。

那我再问你一个问题，为什么 Guava 选择第一种而不是第二种解决方案呢？使用第二种解决方案是否也可以呢？答案是不行的。至于为什么，我们看下源码就清楚了。我把 CacheBuilder 类中的 build() 函数摘抄到了下面，你可以先看下。

 复制代码

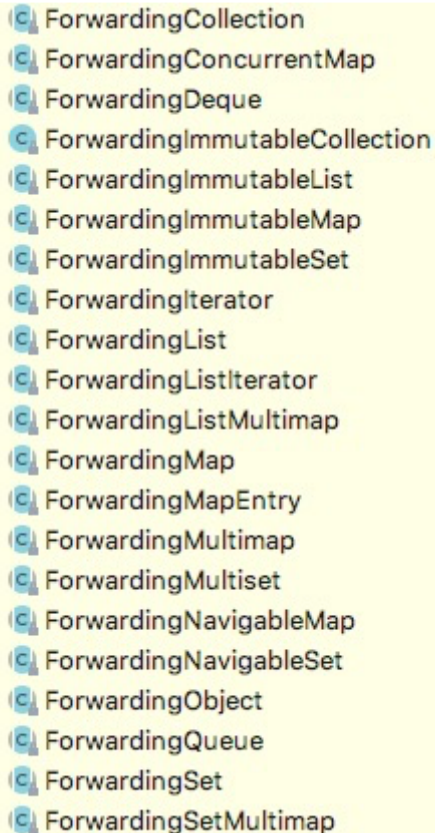
```
1 public <K1 extends K, V1 extends V> Cache<K1, V1> build() {
2     this.checkWeightWithWeigher();
3     this.checkNonLoadingCache();
4     return new LocalManualCache(this);
5 }
6
7 private void checkNonLoadingCache() {
8     Preconditions.checkState(this.refreshNanos == -1L, "refreshAfterWrite requires refreshNanos");
9 }
10
11 private void checkWeightWithWeigher() {
12     if (this.weigher == null) {
13         Preconditions.checkState(this.maximumWeight == -1L, "maximumWeight requires weigher");
14     } else if (this.strictParsing) {
15         Preconditions.checkState(this.maximumWeight != -1L, "weigher requires maximumWeight");
16     } else if (this.maximumWeight == -1L) {
17         logger.log(Level.WARNING, "ignoring weigher specified without maximumWeight");
18     }
19 }
20 }
```

看了代码，你是否有了答案呢？实际上，答案我们在讲 Builder 模式的时候已经讲过了。现在，我们再结合 CacheBuilder 的源码重新说下。

必须使用 Builder 模式的主要原因是，在真正构造 Cache 对象的时候，我们必须做一些必要的参数校验，也就是 build() 函数中前两行代码要做的工作。如果采用无参默认构造函数加 setXXX() 方法的方案，这两个校验就无处安放了。而不经校验，创建的 Cache 对象有可能是非法、不可用的。


Wrapper 模式在 Guava 中的应用

在 Google Guava 的 collection 包路径下，有一组以 Forwarding 开头命名的类。我截了这些类中的一部分贴到了下面，你可以看下。



- ForwardingCollection
- ForwardingConcurrentMap
- ForwardingDeque
- ForwardingImmutableCollection
- ForwardingImmutableList
- ForwardingImmutableMap
- ForwardingImmutableSet
- ForwardingIterator
- ForwardingList
- ForwardingListIterator
- ForwardingListMultimap
- ForwardingMap
- ForwardingMapEntry
- ForwardingMultimap
- ForwardingMultiset
- ForwardingNavigableMap
- ForwardingNavigableSet
- ForwardingObject
- ForwardingQueue
- ForwardingSet
- ForwardingSetMultimap


这组 Forwarding 类很多，但实现方式都很相似。我摘抄了其中的 ForwardingCollection 中的部分代码到这里，你可以先看下代码，然后思考下这组 Forwarding 类是干什么用的。

 复制代码

```
1 @GwtCompatible
2 public abstract class ForwardingCollection<E> extends ForwardingObject implements
3     protected ForwardingCollection() {
4     }
5
6     protected abstract Collection<E> delegate();
7
8     public Iterator<E> iterator() {
9         return this.delegate().iterator();
10    }
11
12    public int size() {
13        return this.delegate().size();
14    }
15
```

```
16  @CanIgnoreReturnValue
17  public boolean removeAll(Collection<?> collection) {
18      return this.delegate().removeAll(collection);
19  }
20
21  public boolean isEmpty() {
22      return this.delegate().isEmpty();
23  }
24
25  public boolean contains(Object object) {
26      return this.delegate().contains(object);
27  }
28
29  @CanIgnoreReturnValue
30  public boolean add(E element) {
31      return this.delegate().add(element);
32  }
33
34  @CanIgnoreReturnValue
35  public boolean remove(Object object) {
36      return this.delegate().remove(object);
37  }
38
39  public boolean containsAll(Collection<?> collection) {
40      return this.delegate().containsAll(collection);
41  }
42
43  @CanIgnoreReturnValue
44  public boolean addAll(Collection<? extends E> collection) {
45      return this.delegate().addAll(collection);
46  }
47
48  @CanIgnoreReturnValue
49  public boolean retainAll(Collection<?> collection) {
50      return this.delegate().retainAll(collection);
51  }
52
53  public void clear() {
54      this.delegate().clear();
55  }
56
57  public Object[] toArray() {
58      return this.delegate().toArray();
59  }
60
61  //...省略部分代码...
62 }
```


光看 ForwardingCollection 的代码实现，你可能想不到它的作用。我再给点提示，举一个它的用法示例，如下所示：

 复制代码

```
1 public class AddLoggingCollection<E> extends ForwardingCollection<E> {
2     private static final Logger logger = LoggerFactory.getLogger(AddLoggingColle
3     private Collection<E> originalCollection;
4
5     public AddLoggingCollection(Collection<E> originalCollection) {
6         this.originalCollection = originalCollection;
7     }
8
9     @Override
10    protected Collection delegate() {
11        return this.originalCollection;
12    }
13
14    @Override
15    public boolean add(E element) {
16        logger.info("Add element: " + element);
17        return this.delegate().add(element);
18    }
19
20    @Override
21    public boolean addAll(Collection<? extends E> collection) {
22        logger.info("Size of elements to add: " + collection.size());
23        return this.delegate().addAll(collection);
24    }
25
26 }
```

结合源码和示例，我想你应该知道这组 Forwarding 类的作用了吧？

在上面的代码中，AddLoggingCollection 是基于代理模式实现的一个代理类，它在原始 Collection 类的基础之上，针对“add”相关的操作，添加了记录日志的功能。

我们前面讲到，代理模式、装饰器、适配器模式可以统称为 Wrapper 模式，通过 Wrapper 类二次封装原始类。它们的代码实现也很相似，都可以通过组合的方式，将 Wrapper 类的函数实现委托给原始类的函数来实现。

 复制代码

```
1 public interface Interf {
2     void f1();
```

```

3   void f2();
4   }
5   public class OriginalClass implements Interf {
6       @Override
7       public void f1() { //... }
8       @Override
9       public void f2() { //... }
10  }
11
12  public class WrapperClass implements Interf {
13      private OriginalClass oc;
14      public WrapperClass(OriginalClass oc) {
15          this.oc = oc;
16      }
17      @Override
18      public void f1() {
19          //...附加功能...
20          this.oc.f1();
21          //...附加功能...
22      }
23      @Override
24      public void f2() {
25          this.oc.f2();
26      }
27  }

```

实际上，这个 ForwardingCollection 类是一个“默认 Wrapper 类”或者叫“缺省 Wrapper 类”。这类似于在装饰器模式那一节课中，讲到的 FilterInputStream 缺省装饰器类。你可以再重新看下 [第 50 讲](#) 装饰器模式的相关内容。

如果我们不使用这个 ForwardinCollection 类，而是让 AddLoggingCollection 代理类直接实现 Collection 接口，那 Collection 接口中的所有方法，都要在 AddLoggingCollection 类中实现一遍，而真正需要添加日志功能的只有 add() 和 addAll() 两个函数，其他函数的实现，都只是类似 Wrapper 类中 f2() 函数的实现那样，简单地委托给原始 collection 类对象的对应函数。


为了简化 Wrapper 模式的代码实现，Guava 提供一系列缺省的 Forwarding 类。用户在自己的 Wrapper 类的时候，基于缺省的 Forwarding 类来扩展，就可以只实现自己关心的方法，其他不关心的方法使用缺省 Forwarding 类的实现，就像 AddLoggingCollection 类的实现那样。

Immutable 模式在 Guava 中的应用

Immutable 模式，中文叫作不变模式，它并不属于经典的 23 种设计模式，但作为一种较常用的设计思路，可以总结为一种设计模式来学习。之前在理论部分，我们只稍微提到过 Immutable 模式，但没有独立的拿出来详细讲解，我们这里借 Google Guava 再补充讲解一下。

一个对象的状态在对象创建之后就不再改变，这就是所谓的不变模式。其中涉及的类就是**不变类** (Immutable Class)，对象就是**不变对象** (Immutable Object)。在 Java 中，最常用的不变类就是 String 类，String 对象一旦创建之后就无法改变。

不变模式可以分为两类，一类是普通不变模式，另一类是深度不变模式 (Deeply Immutable Pattern)。普通的不变模式指的是，对象中包含的引用对象是可以改变的。如果不特别说明，通常我们所说的不变模式，指的就是普通的不变模式。深度不变模式指的是，对象包含的引用对象也不可变。它们两个之间的关系，有点类似之前讲过的浅拷贝和深拷贝之间的关系。我举了一个例子来进一步解释一下，代码如下所示：

 复制代码

```
1 // 普通不变模式
2 public class User {
3     private String name;
4     private int age;
5     private Address addr;
6
7     public User(String name, int age, Address addr) {
8         this.name = name;
9         this.age = age;
10        this.addr = addr;
11    }
12    // 只有getter方法, 无setter方法...
13 }
14
15 public class Address {
16     private String province;
17     private String city;
18     public Address(String province, String city) {
19         this.province = province;
20         this.city= city;
21     }
22    // 有getter方法, 也有setter方法...
23 }
24
25 // 深度不变模式
26 public class User {
27     private String name;
28     private int age;
```



```

29     private Address addr;
30
31     public User(String name, int age, Address addr) {
32         this.name = name;
33         this.age = age;
34         this.addr = addr;
35     }
36     // 只有getter方法, 无setter方法...
37 }
38
39 public class Address {
40     private String province;
41     private String city;
42     public Address(String province, String city) {
43         this.province = province;
44         this.city= city;
45     }
46     // 只有getter方法, 无setter方法..
47 }

```

在某个业务场景下，如果一个对象符合创建之后就不会被修改这个特性，那我们就可以把它设计成不变类。显式地强制它不可变，这样能避免意外被修改。那如何将一个不变类呢？方法很简单，只要这个类满足：所有的成员变量都通过构造函数一次性设置好，不暴露任何 set 等修改成员变量的方法。除此之外，因为数据不变，所以不存在并发读写问题，因此不变模式常用在多线程环境下，来避免线程加锁。所以，不变模式也常被归类为多线程设计模式。


接下来，我们来看一种特殊的不变类，那就是不变集合。Google Guava 针对集合类（Collection、List、Set、Map...）提供了对应的不变集合类（ImmutableCollection、ImmutableList、ImmutableSet、ImmutableMap...）。刚刚我们讲过，不变模式分为两种，普通不变模式和深度不变模式。Google Guava 提供的不变集合类属于前者，也就是说，集合中的对象不会增删，但是对象的成员变量（或叫属性值）是可以改变的。

实际上，Java JDK 也提供了不变集合类（UnmodifiableCollection、UnmodifiableList、UnmodifiableSet、UnmodifiableMap...）。那它跟 Google Guava 提供的不变集合类的区别在哪里呢？我举个例子你就明白了，代码如下所示：

```

1 public class ImmutableDemo {
2     public static void main(String[] args) {
3         List<String> originalList = new ArrayList<>();

```

 复制代码

```
4     originalList.add("a");
5     originalList.add("b");
6     originalList.add("c");
7
8     List<String> jdkUnmodifiableList = Collections.unmodifiableList(originalLi:
9     List<String> guavaImmutableList = ImmutableList.copyOf(originalList);
10
11     //jdkUnmodifiableList.add("d"); // 抛出UnsupportedOperationException
12     // guavaImmutableList.add("d"); // 抛出UnsupportedOperationException
13     originalList.add("d");
14
15     print(originalList); // a b c d
16     print(jdkUnmodifiableList); // a b c d
17     print(guavaImmutableList); // a b c
18 }
19
20 private static void print(List<String> list) {
21     for (String s : list) {
22         System.out.print(s + " ");
23     }
24     System.out.println();
25 }
26 }
```

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天我们学习了 Google Guava 中都用到的几个设计模式：Builder 模式、Wrapper 模式、Immutable 模式。还是那句话，内容本身不重要，你也不用死记硬背 Google Guava 的某某类用到了某某设计模式。实际上，我想通过这些源码的剖析，传达给你下面这些东西。

我们在阅读源码的时候，要问问自己，为什么它要这么设计？不这么设计行吗？还有更好的设计吗？实际上，很多人缺少这种“质疑”精神，特别是面对权威（经典书籍、著名源码、权威人士）的时候。

我觉得我本人是最不缺质疑精神的一个人，我喜欢挑战权威，喜欢以理服人。就好比在今天的讲解中，我把 ForwardingCollection 等类理解为缺省 Wrapper 类，可以用在装饰器、代理、适配器三种 Wrapper 模式中，简化代码编写。如果你去看 Google Guava 在 GitHub 上的 Wiki，你会发现，它对 ForwardingCollection 类的理解跟我是不一样的。它

把 ForwardingCollection 类单纯地理解为缺省的装饰器类，只用在装饰器模式中。我个人觉得我的理解更加好些，不知道你怎么认为呢？

除此之外，在专栏的最开始，我也讲到，学习设计模式能让你更好的阅读源码、理解源码。如果我们没有之前的理论学习，那对于很多源码的阅读，可能都只停留在走马观花的层面上，根本学习不到它的精髓。这就好比今天讲到的 CacheBuilder。我想大部分人都知道它是利用了 Builder 模式，但如果对 Builder 模式没有深入的了解，很少人能讲清楚为什么要用 Builder 模式，不用构造函数加 set 方法的方式来实现。

课堂讨论

从最后一段代码中，我们可以发现，JDK 不变集合和 Google Guava 不变集合都不可增删数据。但是，当原始集合增加数据之后，JDK 不变集合的数据随之增加，而 Google Guava 的不变集合的数据并没有增加。这是两者最大的区别。那这两者底层分别是如何实现不变的呢？

欢迎留言和我分享你的想法，如果有收获，也欢迎你把这篇文章分享给你的朋友。

课程预告

5月-6月课表抢先看

充 ¥500 得 ¥580

赠 「¥99 运动水杯+ ¥129 防紫外线伞」



【点击】图片，立即查看 >>>

上一篇 81 | 开源实战三（上）：借Google Guava学习发现和开发通用功能模块

下一篇 加餐一 | 用一篇文章带你了解专栏中用到的所有Java语法

精选留言 (14)

写留言



成楠Peter

2020-05-11

JDK是浅拷贝，Guava使用的是深拷贝。一个复制引用，一个复制值。



9



hhhh

2020-05-11

猜测jdk中的不变集合保存了原始集合的引用，而guava应该是复制了原始集合的值。



2



不能忍的地精

2020-05-11

Guava里面的引用已经是一个新的集合,Jdk里面的引用还是原来的集合



Jxin

2020-05-11

- 1.两者都是生成一个新的集合对象。
- 2.前者相当于对原集合采用装饰者模式。通过复合方式限制掉原集合的写操作。实现，封装后的集合，在后续使用中不可变的特性。具有灵活性。
- 3.后者相当于新建一个不可变集合。通过原集合的元素，生成一个不可变集合。语义更加明确。...

展开



2020-05-11

课后题:

jdk的不变集合引用了原始的集合类,所以在原始集合类发生改变的时候他也会改变,他的不可变只是客户端不可变;

guava的不变集合,是在重新创建了一个原始集合对象的副本,所以改变原始类并不能改变他的数据

展开 ▾



守拙

2020-05-11

通过阅读JDK源码, 发现UnmodifiableList内部使用原始List的浅拷贝, 所以当原始list增/删时会影响UnmodifiableList. 额外说一句, UnmodifiableList实现并Override了List接口的add(), remove()等方法, 通过抛出UnsupportedOperationException来抑制add/remove等改变数据源的操作.

...

展开 ▾



小晏子

2020-05-11

JDK中的unmodifiableList的构造函数是对原始集合的浅拷贝, 而Guava.ImmutableList.copyOf是对原始集合的深拷贝。从source code可以看出来:

UnmodifiableList

```
UnmodifiableList(List<? extends E> list) {  
    super(list);...
```

展开 ▾



汝林外史

2020-05-11

我觉得 ForwardingCollection 类就应该理解为缺省的装饰器类, 前面的文章就说过代理模式、装饰器模式、适配器模式代码的写法几乎一样, 差别就是各自的使用场景, 我觉得ForwardingCollection这些类的使用场景就是作为装饰类来用的, 不会应用到代理和适配器的场景, 王老师貌似又掉入了以代码写法判断设计模式的自己说的陷阱中。

展开 ▾



Snway

2020-05-11

Jdk直接引用原来的集合, guava是拷贝了原来的集合

展开 ▾



test

2020-05-11

jdk是浅拷贝，guava是深拷贝，在修改的时候报错

展开 ▾



whistleman

2020-05-11

要多思考背后为什么要用这种设计模式，才能对使用的设计模式有更深刻的理解。打卡！



leezer

2020-05-11

我觉得我更赞同wrapper类的理解，因为装饰器的主要功能是在原始的类上做功能增强，而代理模式更多关注对非业务功能的关注。通过组合的方式我们能实现更多的Wrapper模式。这时候就不只是算装饰器的设计模式了

。

展开 ▾



Jason

2020-05-11

思考题：我猜是深拷贝和浅拷贝的区别

展开 ▾



何用

2020-05-11

我是个特别能关注到细节的人。Memcached 是个开源库，不知道为何好多人都喜欢把它叫做 Memcache，本文也不例外。

展开 ▾

