82-开源实战三（中）：剖析GoogleGuava中用到的几种设计模式

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。

```
public class CacheDemo {
  public static void main(String[] args) {
    Cache<String, String> cache = CacheBuilder.newBuilder()
            .initialCapacity(100)
            .maximumSize(1000)
            .expireAfterWrite(10, TimeUnit.MINUTES)
            .build();

    cache.put("key1", "value1");
    String value = cache.getIfPresent("key1");
    System.out.println(value);
  }
}
```

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。

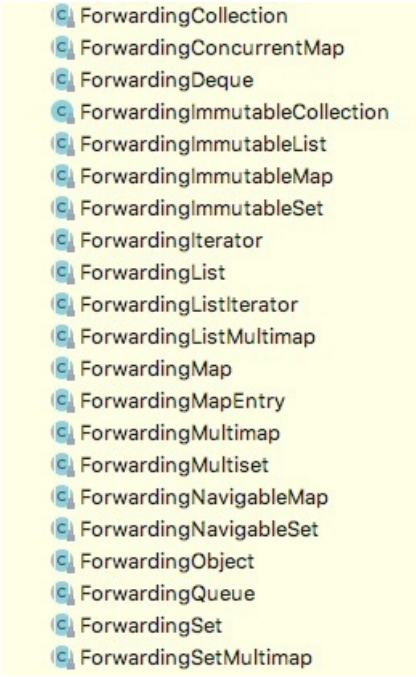
```
public <K1 extends K, V1 extends V> Cache<K1, V1> build() {
  this.checkWeightWithWeigher();
  this.checkNonLoadingCache();
  return new LocalManualCache(this);
}

private void checkNonLoadingCache() {
  Preconditions.checkState(this.refreshNanos == -1L, "refreshAfterWrite requires a LoadingCache");
}

private void checkWeightWithWeigher() {
  if (this.weigher == null) {
    Preconditions.checkState(this.maximumWeight == -1L, "maximumWeight requires weigher");
  } else if (this.strictParsing) {
    Preconditions.checkState(this.maximumWeight != -1L, "weigher requires maximumWeight");
  } else if (this.maximumWeight == -1L) {
    logger.log(Level.WARNING, "ignoring weigher specified without maximumWeight");
  }

}
```

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。而不经过校验，创建的Cache对象有可能是不合法、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。



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，你可以下先看下代码，然后思考下这组Forwarding类是干什么用的。

```
@GwtCompatible
public abstract class ForwardingCollection<E> extends ForwardingObject implements Collection<E> {
  protected ForwardingCollection() {
  }

  protected abstract Collection<E> delegate();

  public Iterator<E> iterator() {
    return this.delegate().iterator();
  }

  public int size() {
    return this.delegate().size();
  }

  @CanIgnoreReturnValue
  public boolean removeAll(Collection<?> collection) {
    return this.delegate().removeAll(collection);
  }

  public boolean isEmpty() {
    return this.delegate().isEmpty();
  }

  public boolean contains(Object object) {
    return this.delegate().contains(object);
  }

  @CanIgnoreReturnValue
  public boolean add(E element) {
    return this.delegate().add(element);
  }

  @CanIgnoreReturnValue
  public boolean remove(Object object) {
    return this.delegate().remove(object);
  }
```

```
  public boolean containsAll(Collection<?> collection) {
    return this.delegate().containsAll(collection);
  }

  @CanIgnoreReturnValue
  public boolean addAll(Collection<? extends E> collection) {
    return this.delegate().addAll(collection);
  }

  @CanIgnoreReturnValue
  public boolean retainAll(Collection<?> collection) {
    return this.delegate().retainAll(collection);
  }

  public void clear() {
    this.delegate().clear();
  }

  public Object[] toArray() {
    return this.delegate().toArray();
  }

  //...省略部分代码...
}
```

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：

```
public class AddLoggingCollection<E> extends ForwardingCollection<E> {
  private static final Logger logger = LoggerFactory.getLogger(AddLoggingCollection.class);
  private Collection<E> originalCollection;

  public AddLoggingCollection(Collection<E> originalCollection) {
    this.originalCollection = originalCollection;
  }

  @Override
  protected Collection delegate() {
    return this.originalCollection;
  }

  @Override
  public boolean add(E element) {
    logger.info("Add element: " + element);
    return this.delegate().add(element);
  }

  @Override
  public boolean addAll(Collection<? extends E> collection) {
    logger.info("Size of elements to add: " + collection.size());
    return this.delegate().addAll(collection);
  }

}
```

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。

```
public interface Interf {
  void f1();
  void f2();
}
public class OriginalClass implements Interf {
  @Override
  public void f1() { //... }
  @Override
  public void f2() { //... }
}

public class WrapperClass implements Interf {
  private OriginalClass oc;
  public WrapperClass(OriginalClass oc) {
    this.oc = oc;
  }
  @Override
  public void f1() {
    //...附加功能...
    this.oc.f1();
    //...附加功能...
  }
  @Override
  public void f2() {
    this.oc.f2();
  }
}
```

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。用户在实现自己的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：

```
// 普通不变模式
public class User {
  private String name;
  private int age;
  private Address addr;

  public User(String name, int age, Address addr) {
    this.name = name;
    this.age = age;
    this.addr = addr;
  }
  // 只有getter方法，无setter方法...
}

public class Address {
  private String province;
  private String city;
  public Address(String province, String city) {
    this.province = province;
    this.city= city;
  }
  // 有getter方法，也有setter方法...
}

// 深度不变模式
public class User {
  private String name;
  private int age;
  private Address addr;

  public User(String name, int age, Address addr) {
    this.name = name;
    this.age = age;
    this.addr = addr;
  }
  // 只有getter方法，无setter方法...
}

public class Address {
  private String province;
  private String city;
  public Address(String province, String city) {
    this.province = province;
    this.city= city;
  }
```

```
  // 只有getter方法，无setter方法..
}
```

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：

```
public class ImmutableDemo {
  public static void main(String[] args) {
    List<String> originalList = new ArrayList<>();
    originalList.add("a");
    originalList.add("b");
    originalList.add("c");

    List<String> jdkUnmodifiableList = Collections.unmodifiableList(originalList);
    List<String> guavaImmutableList = ImmutableList.copyOf(originalList);

    //jdkUnmodifiableList.add("d"); // 抛出UnsupportedOperationException
    // guavaImmutableList.add("d"); // 抛出UnsupportedOperationException
    originalList.add("d");

    print(originalList); // a b c d
    print(jdkUnmodifiableList); // a b c d
    print(guavaImmutableList); // a b c
  }

  private static void print(List<String> list) {
    for (String s : list) {
      System.out.print(s + " ");
    }
    System.out.println();
  }
}
```

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。

精选留言：

- 成楠Peter 2020-05-11 08:27:49
  JDK是浅拷贝，Guava使用的是深拷贝。一个复制引用，一个复制值。 [11赞]

- hhhh 2020-05-11 06:13:22
  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赞]

- 辣么大 2020-05-11 20:39:07
  在JDK中只是将list的地址赋给了UnmodifiableList
  final List<? extends E> list;
  UnmodifiableList(List<? extends E> list) {
  super(list);
  this.list = list;
  }
  在Guava中不可变集合是“保护性”拷贝，创建的不可变集合可以理解为常量。
  要创建真正的不可变集合，集合中的对象还要是真正的不可变。
  下面我举个反例，各位看看：
  public static void main(String[] args) {
  List<Student> ori = new ArrayList<>();
  ori.add(new Student("xiaoqiang", 10));

  Student mutable = new Student("wangz", 8);

```
ori.add(mutable);

ori.add(new Student("lameda", 12));
List<Student> jdkCopy = Collections.unmodifiableList(ori);

List<Student> guavaCopy = ImmutableList.copyOf(ori);

ori.add(new Student("wawa", 20));

System.out.println(jdkCopy);
System.out.println(guavaCopy);

mutable.name = "mutable";
System.out.println(guavaCopy);
// [Student{age=10, name='xiaoqiang'}, Student{age=8, name='mutable'}, Student{age=12, name='la
meda'}]

} [1赞]
```

- 小晏子 2020-05-11 10:18:02

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);
this.list = list;
}
Guava.ImmutableList.copyOf
public static <E> ImmutableList<E> copyOf(Collection<? extends E> elements) {
if (elements instanceof ImmutableCollection) {
@SuppressWarnings("unchecked") // all supported methods are covariant
ImmutableList<E> list = ((ImmutableCollection<E>) elements).asList();
return list.isPartialView() ? ImmutableList.<E>asImmutableList(list.toArray()) : list;
}
return construct(elements.toArray());
}
/** Views the array as an immutable list. Checks for nulls; does not copy. */
private static <E> ImmutableList<E> construct(Object... elements) {
return asImmutableList(checkElementsNotNull(elements));
}

/**
* Views the array as an immutable list. Does not check for nulls; does not copy.
*
* <p>The array must be internally created.
*/
static <E> ImmutableList<E> asImmutableList(Object[] elements) {
return asImmutableList(elements, elements.length);
}
```

```
/**
* Views the array as an immutable list. Copies if the specified range does not cover the complete
* array. Does not check for nulls.
*/
static <E> ImmutableList<E> asImmutableList(Object[] elements, int length) {
switch (length) {
case 0:
return of();
case 1:
return of((E) elements[0]);
default:
if (length < elements.length) {
elements = Arrays.copyOf(elements, length);
}
return new RegularImmutableList<E>(elements);
}
} [1赞]
```

- jaryoung 2020-05-11 22:44:47
  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。为啥要设置成跟jdk不一样？换句话说，我觉得应该是，如果jdk和guava功能都一摸一样，就没有存在的必要了。底层的实现，jdk如下：
  final List<? extends E> list，guava是对集合内容的对象进行逐一拷贝。
  本来不想查源码，但是不想误导别人，还是把源码看了一下。

- Frank 2020-05-11 22:20:01
  unmodifiableList 内部还是使用了Warpper模式，重新实现了某些方法，比如add,remove等，当调用这些方法时，抛出异常，而有些方法还是委托给原始list进行操作，比如get操作。所以这里在原始类添加元素后，使用不jdk的变类可以打印出新添加的元素。而Guava 中的ImmutableList 时采用拷贝的方式将原始集合中的数据拷贝到一个对象数组中，后续原始集合添加，删除元素，其结果都不会影响该Immutable List。

- do it 2020-05-11 20:32:27
  没看过源码，猜测是浅拷贝与深拷贝的区别

- 不能忍的地精 2020-05-11 17:16:21
  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 13:12:29
  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。

  4.前者通过按需操作，具备灵活性。但在集合接口加缺省方法时，可能会有bug。毕竟它是以复合实现功能的。后者语义更明确，不具备前者的灵活性。但在集合接口加缺省方法时，一般不会有bug。因为它是操作自身数据结构实现的功能，与原集合无关联。

- ， 2020-05-11 11:03:31
  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

- 守拙 2020-05-11 10:28:14
  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.

  Guava包下的ImmutableList.copyOf(Collection<? extends E> elements)内部调用了construct(elements.toArray())方法, 内部维护了源List的数组copy, 属于深拷贝范畴. 执行construct(elements.toArray())后, ImmutableList内部维护数组作为数据源, 与源List完全隔离, 所以源List的add/remove等操作不会影响到ImmutableList.

  源码参考:
  java.util.collections 1337行开始;(内部类UnmodifiableList)
  com.google.common.collect.ImmutableList 238行开始.(copyOf方法)

- 汝林外史 2020-05-11 09:56:30
  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。

- Snway 2020-05-11 09:51:11
  Jdk直接引用原来的集合，guava是拷贝了原来的集合

- test 2020-05-11 09:18:14
  jdk是浅拷贝，guava是深拷贝，在修改的时候报错

- whistleman 2020-05-11 09:00:21
  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 08:53:06
  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
  。

- Jason 2020-05-11 07:37:45
  思考题：我猜是深拷贝和浅拷贝的区别

- 何用 2020-05-11 07:20:50
  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。