76-开源实战一（上）：通过剖析JavaJDK源码学习灵活应用设计模式

从今天开始，我们就正式地进入到实战环节。实战环节包括两部分，一部分是开源项目实战，另一部分是项目实战。

在开源项目实战部分，我会带你剖析几个经典的开源项目中用到的设计原则、思想和模式，这其中就包括对Java JDK、Unix、Google Guava、Spring、MyBatis这样五个开源项目的分析。在项目实战部分，我们精心挑选了几个实战项目，手把手地带你利用之前学过的设计原则、思想、模式，来对它们进行分析、设计和代码实现，这其中就包括鉴权限流、幂等重试、灰度发布这样三个项目。

接下来的两节课，我们重点剖析Java JDK中用到的几种常见的设计模式。学习的目的是让你体会，在真实的项目开发中，要学会活学活用，切不可过于死板，生搬硬套设计模式的设计与实现。除此之外，针对每个模式，我们不可能像前面学习理论知识那样，分析得细致入微，很多都是点到为止。在已经具备之前理论知识的前提下，我想你可以跟着我的指引自己去研究，有哪里不懂的话，也可以再回过头去看下之前的理论讲解。

话不多说，让我们正式开始今天的学习吧！

工厂模式在Calendar类中的应用

在前面讲到工厂模式的时候，大部分工厂类都是以Factory作为后缀来命名，并且工厂类主要负责创建对象这样一件事情。但在实际的项目开发中，工厂类的设计更加灵活。那我们就来看下，工厂模式在Java JDK中的一个应用：java.util.Calendar。从命名上，我们无法看出它是一个工厂类。

Calendar类提供了大量跟日期相关的功能代码，同时，又提供了一个getInstance()工厂方法，用来根据不同的TimeZone和Locale创建不同的Calendar子类对象。也就是说，功能代码和工厂方法代码耦合在了一个类中。所以，即便我们去查看它的源码，如果不细心的话，也很难发现它用到了工厂模式。同时，因为它不单单是一个工厂类，所以，它并没有以Factory作为后缀来命名。

Calendar类的相关代码如下所示，大部分代码都已经省略，我只给出了getInstance()工厂方法的代码实现。从代码中，我们可以看出，getInstance()方法可以根据不同TimeZone和Locale，创建不同的Calendar子类对象，比如BuddhistCalendar、JapaneseImperialCalendar、GregorianCalendar，这些细节完全封装在工厂方法中，使用者只需要传递当前的时区和地址，就能够获得一个Calendar类对象来使用，而获得的对象具体是哪个Calendar子类的对象，使用者在使用的时候并不关心。

```
public abstract class Calendar implements Serializable, Cloneable, Comparable<Calendar> {
  //...
  public static Calendar getInstance(TimeZone zone, Locale aLocale){
    return createCalendar(zone, aLocale);
  }

  private static Calendar createCalendar(TimeZone zone,Locale aLocale) {
    CalendarProvider provider = LocaleProviderAdapter.getAdapter(
        CalendarProvider.class, aLocale).getCalendarProvider();
    if (provider != null) {
      try {
        return provider.getInstance(zone, aLocale);
      } catch (IllegalArgumentException iae) {
        // fall back to the default instantiation
      }
```

```
    }

    Calendar cal = null;
    if (aLocale.hasExtensions()) {
      String caltype = aLocale.getUnicodeLocaleType("ca");
      if (caltype != null) {
        switch (caltype) {
          case "buddhist":
            cal = new BuddhistCalendar(zone, aLocale);
            break;
          case "japanese":
            cal = new JapaneseImperialCalendar(zone, aLocale);
            break;
          case "gregory":
            cal = new GregorianCalendar(zone, aLocale);
            break;
        }
      }
    }
    if (cal == null) {
      if (aLocale.getLanguage() == "th" && aLocale.getCountry() == "TH") {
        cal = new BuddhistCalendar(zone, aLocale);
      } else if (aLocale.getVariant() == "JP" && aLocale.getLanguage() == "ja" && aLocale.getCountry() == "
        cal = new JapaneseImperialCalendar(zone, aLocale);
      } else {
        cal = new GregorianCalendar(zone, aLocale);
      }
    }
    return cal;
  }
  //...
}
```

建造者模式在Calendar类中的应用

还是刚刚的Calendar类，它不仅仅用到了工厂模式，还用到了建造者模式。我们知道，建造者模式有两种实现方法，一种是单独定义一个Builder类，另一种是将Builder实现为原始类的内部类。Calendar就采用了第二种实现思路。我们先来看代码再讲解，相关代码我贴在了下面。

```
public abstract class Calendar implements Serializable, Cloneable, Comparable<Calendar> {
  //...
  public static class Builder {
    private static final int NFIELDS = FIELD_COUNT + 1;
    private static final int WEEK_YEAR = FIELD_COUNT;
    private long instant;
    private int[] fields;
    private int nextStamp;
    private int maxFieldIndex;
    private String type;
    private TimeZone zone;
    private boolean lenient = true;
    private Locale locale;
    private int firstDayOfWeek, minimalDaysInFirstWeek;

    public Builder() {}

    public Builder setInstant(long instant) {
        if (fields != null) {
            throw new IllegalStateException();
```

```
        }
        this.instant = instant;
        nextStamp = COMPUTED;
        return this;
    }
    //...省略n多set()方法

    public Calendar build() {
      if (locale == null) {
        locale = Locale.getDefault();
      }
      if (zone == null) {
        zone = TimeZone.getDefault();
      }
      Calendar cal;
      if (type == null) {
        type = locale.getUnicodeLocaleType("ca");
      }
      if (type == null) {
        if (locale.getCountry() == "TH" && locale.getLanguage() == "th") {
          type = "buddhist";
        } else {
          type = "gregory";
        }
      }
      switch (type) {
        case "gregory":
          cal = new GregorianCalendar(zone, locale, true);
          break;
        case "iso8601":
          GregorianCalendar gcal = new GregorianCalendar(zone, locale, true);
          // make gcal a proleptic Gregorian
          gcal.setGregorianChange(new Date(Long.MIN_VALUE));
          // and week definition to be compatible with ISO 8601
          setWeekDefinition(MONDAY, 4);
          cal = gcal;
          break;
        case "buddhist":
          cal = new BuddhistCalendar(zone, locale);
          cal.clear();
          break;
        case "japanese":
          cal = new JapaneseImperialCalendar(zone, locale, true);
          break;
        default:
          throw new IllegalArgumentException("unknown calendar type: " + type);
      }
      cal.setLenient(lenient);
      if (firstDayOfWeek != 0) {
        cal.setFirstDayOfWeek(firstDayOfWeek);
        cal.setMinimalDaysInFirstWeek(minimalDaysInFirstWeek);
      }
      if (isInstantSet()) {
        cal.setTimeInMillis(instant);
        cal.complete();
        return cal;
      }

      if (fields != null) {
        boolean weekDate = isSet(WEEK_YEAR) && fields[WEEK_YEAR] > fields[YEAR];
        if (weekDate && !cal.isWeekDateSupported()) {
          throw new IllegalArgumentException("week date is unsupported by " + type);
        }
        for (int stamp = MINIMUM_USER_STAMP; stamp < nextStamp; stamp++) {
          for (int index = 0; index <= maxFieldIndex; index++) {
```

```
            if (fields[index] == stamp) {
              cal.set(index, fields[NFIELDS + index]);
              break;
             }
           }
         }

         if (weekDate) {
           int weekOfYear = isSet(WEEK_OF_YEAR) ? fields[NFIELDS + WEEK_OF_YEAR] : 1;
           int dayOfWeek = isSet(DAY_OF_WEEK) ? fields[NFIELDS + DAY_OF_WEEK] : cal.getFirstDayOfWeek();
           cal.setWeekDate(fields[NFIELDS + WEEK_YEAR], weekOfYear, dayOfWeek);
         }
         cal.complete();
       }
       return cal;
     }
   }
}
```

看了上面的代码，我有一个问题请你思考一下：既然已经有了getInstance()工厂方法来创建Calendar类对象，为什么还要用Builder来创建Calendar类对象呢？这两者之间的区别在哪里呢？

实际上，在前面讲到这两种模式的时候，我们对它们之间的区别做了详细的对比，现在，我们再来一块回顾一下。工厂模式是用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。建造者模式用来创建一种类型的复杂对象，通过设置不同的可选参数，“定制化”地创建不同的对象。

网上有一个经典的例子很好地解释了两者的区别。

> 顾客走进一家餐馆点餐，我们利用工厂模式，根据用户不同的选择，来制作不同的食物，比如披萨、汉堡、沙拉。对于披萨来说，用户又有各种配料可以定制，比如奶酪、西红柿、起司，我们通过建造者模式根据用户选择的不同配料来制作不同的披萨。

粗看Calendar的Builder类的build()方法，你可能会觉得它有点像工厂模式。你的感觉没错，前面一半代码确实跟getInstance()工厂方法类似，根据不同的type创建了不同的Calendar子类。实际上，后面一半代码才属于标准的建造者模式，根据setXXX()方法设置的参数，来定制化刚刚创建的Calendar子类对象。

你可能会说，这还能算是建造者模式吗？我用第46讲的一段话来回答你：

> 我们也不要太学院派，非得把工厂模式、建造者模式分得那么清楚，我们需要知道的是，每个模式为什么这么设计，能解决什么问题。只有了解了这些最本质的东西，我们才能不生搬硬套，才能灵活应用，甚至可以混用各种模式，创造出新的模式来解决特定场景的问题。

实际上，从Calendar这个例子，我们也能学到，不要过于死板地套用各种模式的原理和实现，不要不敢做丝毫的改动。模式是死的，用的人是活的。在实际上的项目开发中，不仅各种模式可以混合在一起使用，而且具体的代码实现，也可以根据具体的功能需求做灵活的调整。

装饰器模式在Collections类中的应用

我们前面讲到，Java IO类库是装饰器模式的非常经典的应用。实际上，Java的Collections类也用到了装饰

器模式。

Collections类是一个集合容器的工具类，提供了很多静态方法，用来创建各种集合容器，比如通过unmodifiableColletion()静态方法，来创建UnmodifiableCollection类对象。而这些容器类中的UnmodifiableCollection类、CheckedCollection和SynchronizedCollection类，就是针对Collection类的装饰器类。

因为刚刚提到的这三个装饰器类，在代码结构上几乎一样，所以，我们这里只拿其中的UnmodifiableCollection类来举例讲解一下。UnmodifiableCollection类是Collections类的一个内部类，相关代码我摘抄到了下面，你可以先看下。

```
public class Collections {
  private Collections() {}

  public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c) {
    return new UnmodifiableCollection<>(c);
  }

  static class UnmodifiableCollection<E> implements Collection<E>,   Serializable {
    private static final long serialVersionUID = 1820017752578914078L;
    final Collection<? extends E> c;

    UnmodifiableCollection(Collection<? extends E> c) {
      if (c==null)
        throw new NullPointerException();
      this.c = c;
    }

    public int size()                   {return c.size();}
    public boolean isEmpty()            {return c.isEmpty();}
    public boolean contains(Object o)   {return c.contains(o);}
    public Object[] toArray()           {return c.toArray();}
    public <T> T[] toArray(T[] a)       {return c.toArray(a);}
    public String toString()            {return c.toString();}

    public Iterator<E> iterator() {
      return new Iterator<E>() {
        private final Iterator<? extends E> i = c.iterator();

        public boolean hasNext() {return i.hasNext();}
        public E next()          {return i.next();}
        public void remove() {
          throw new UnsupportedOperationException();
        }
        @Override
        public void forEachRemaining(Consumer<? super E> action) {
          // Use backing collection version
          i.forEachRemaining(action);
        }
      };
    }

    public boolean add(E e) {
      throw new UnsupportedOperationException();
    }
    public boolean remove(Object o) {
       hrow new UnsupportedOperationException();
    }
    public boolean containsAll(Collection<?> coll) {
```

```
      return c.containsAll(coll);
    }
    public boolean addAll(Collection<? extends E> coll) {
      throw new UnsupportedOperationException();
    }
    public boolean removeAll(Collection<?> coll) {
      throw new UnsupportedOperationException();
    }
    public boolean retainAll(Collection<?> coll) {
      throw new UnsupportedOperationException();
    }
    public void clear() {
      throw new UnsupportedOperationException();
    }

    // Override default methods in Collection
    @Override
    public void forEach(Consumer<? super E> action) {
      c.forEach(action);
    }
    @Override
    public boolean removeIf(Predicate<? super E> filter) {
      throw new UnsupportedOperationException();
    }
    @SuppressWarnings("unchecked")
    @Override
    public Spliterator<E> spliterator() {
      return (Spliterator<E>)c.spliterator();
    }
    @SuppressWarnings("unchecked")
    @Override
    public Stream<E> stream() {
      return (Stream<E>)c.stream();
    }
    @SuppressWarnings("unchecked")
    @Override
    public Stream<E> parallelStream() {
      return (Stream<E>)c.parallelStream();
    }
  }
}
```

看了上面的代码，请你思考一下，为什么说UnmodifiableCollection类是Collection类的装饰器类呢？这两者之间可以看作简单的接口实现关系或者类继承关系吗？

我们前面讲过，装饰器模式中的装饰器类是对原始类功能的增强。尽管UnmodifiableCollection类可以算是对Collection类的一种功能增强，但这点还不具备足够的说服力来断定UnmodifiableCollection就是Collection类的装饰器类。

实际上，最关键的一点是，UnmodifiableCollection的构造函数接收一个Collection类对象，然后对其所有的函数进行了包裹（Wrap）：重新实现（比如add()函数）或者简单封装（比如stream()函数）。而简单的接口实现或者继承，并不会如此来实现UnmodifiableCollection类。所以，从代码实现的角度来说，UnmodifiableCollection类是典型的装饰器类。

适配器模式在Collections类中的应用

在第51讲中我们讲到，适配器模式可以用来兼容老的版本接口。当时我们举了一个JDK的例子，这里我们再

重新仔细看一下。

老版本的JDK提供了Enumeration类来遍历容器。新版本的JDK用Iterator类替代Enumeration类来遍历容器。为了兼容老的客户端代码（使用老版本JDK的代码），我们保留了Enumeration类，并且在Collections类中，仍然保留了enumaration()静态方法（因为我们一般都是通过这个静态函数来创建一个容器的Enumeration类对象）。

不过，保留Enumeration类和enumeration()函数，都只是为了兼容，实际上，跟适配器没有一点关系。那到底哪一部分才是适配器呢？

在新版本的JDK中，Enumeration类是适配器类。它适配的是客户端代码（使用Enumeration类）和新版本JDK中新的迭代器Iterator类。不过，从代码实现的角度来说，这个适配器模式的代码实现，跟经典的适配器模式的代码实现，差别稍微有点大。enumeration()静态函数的逻辑和Enumeration适配器类的代码耦合在一起，enumeration()静态函数直接通过new的方式创建了匿名类对象。具体的代码如下所示：

```
/**
 * Returns an enumeration over the specified collection.  This provides
 * interoperability with legacy APIs that require an enumeration
 * as input.
 *
 * @param  <T> the class of the objects in the collection
 * @param c the collection for which an enumeration is to be returned.
 * @return an enumeration over the specified collection.
 * @see Enumeration
 */
public static <T> Enumeration<T> enumeration(final Collection<T> c) {
  return new Enumeration<T>() {
    private final Iterator<T> i = c.iterator();

    public boolean hasMoreElements() {
      return i.hasNext();
    }

    public T nextElement() {
      return i.next();
    }
  };
}
```

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，我重点讲了工厂模式、建造者模式、装饰器模式、适配器模式，这四种模式在Java JDK中的应用，主要目的是给你展示真实项目中是如何灵活应用设计模式的。

从今天的讲解中，我们可以学习到，尽管在之前的理论讲解中，我们都有讲到每个模式的经典代码实现，但是，在真实的项目开发中，这些模式的应用更加灵活，代码实现更加自由，可以根据具体的业务场景、功能需求，对代码实现做很大的调整，甚至还可能会对模式本身的设计思路做调整。

比如，Java JDK中的Calendar类，就耦合了业务功能代码、工厂方法、建造者类三种类型的代码，而且，在建造者类的build方法中，前半部分是工厂方法的代码实现，后半部分才是真正的建造者模式的代码实现。这也告诉我们，在项目中应用设计模式，切不可生搬硬套，过于学院派，要学会结合实际情况做灵活调整，做到心中无剑胜有剑。

课堂讨论

在Java中，经常用到的StringBuilder类是否是建造者模式的应用呢？你可以试着像我一样从源码的角度去剖析一下。

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：

- Darren 2020-04-27 23:53:00
  我觉得是，因为StringBuilder的主要方法append，其实就是类似于建造者模式中的set方法，只不过构建者模式的set方法可能是对象的不同属性，但append其实是在一直修改一个属性，且最后没有build(),但StringBuilder出现的目的其实是为了解决String不可变的问题，最终输出其实是String，所以可以类比toString()就是build()，所以认为算是建造者模式。 [2赞]

- QQ怪 2020-04-27 07:50:33
  StringBuilder的append()方法使用了建造者模式，StringBuilder把构建者的角色交给了其的父类AbstractStringBuilder，最终调用的是父类的append（）方法 [2赞]

- 小晏子 2020-04-27 14:28:17
  课后思考：
  我的答案是算也不算…，如果按照学院派的思想，stringbuilder和GOF中的对于builder模式的定义完全不同，stringbuilder并不会创建新的string对象，只是将多个字符连接在一起，而builder模式的基本功能是生成新对象，两个本质就不一样了，从这个角度来讲，stringbuilder不能算是builder模式。
  那为什么又说算呢？这样从另外一个角度想，stringbuilder得到的字符串是一步一步append出来的，这个builder模式的一步一步set在build的行为很像，因为文中也说了不要太学院派，要灵活使用，那么从这个角度来说可以认为stringbuilder也是属于builder的一种实现，只是它不是典型实现。 [1赞]

- Demon.Lee 2020-04-27 23:48:12
  如果说要创建一个复杂的String对象，那么通过StringBuilder的append()方法会非常方便，最后通过toString()方法返回，从这个角度看算建造者模式。
  @Override
  public String toString() {
  // Create a copy, don't share the array
  return new String(value, 0, count);
  }

- Jxin 2020-04-27 13:05:28
  回答课后题：

  我认为应该算是建造者模式。

  拿餐厅类比，对于披萨加番茄，起司等不同配料来定制披萨，这属于建造者模式要解决的问题（实例对象的定制）。而stringbuild的应用场景，更像是对披萨加一个番茄还是两个番茄更或者三个番茄的定制方式（某个字段的定制）。

所以strbuild的应用场景比传统的建造者模式更细更具体（前者实现字段的定制，后者实现对象的定制）。但字段定制依旧属于对象定制的范涛，所以我认为其依旧算是建造者模式。

- Jeff.Smile 2020-04-27 12:15:56
  总结：
  1 工厂模式：简单工厂模式+工厂方法模式
  ● 简单工厂模式直接在条件判断中根据不同参数将目标对象new了出来。
  ● 工厂方法模式是将目标对象的创建过程根据参数分类抽取到各自独立的工厂类中，以应对目标对象创建过程的复杂度。条件分支可以使用map来缓存起来！
  案例：java.util.Calendar的getInstance() 方法可以根据不同 TimeZone 和 Locale，创建不同的 Calendar 子类对象。

  2 建造者模式：
  通过内部的Builder类主导目标对象的创建同时校验必选属性之间的依赖关系。

  3 装饰器模式：
  装饰器模式把每个要装饰的功能放到单独的类中，并让这个类包装它所需要装饰的对象，从而实现对原始类的增强。
  案例：java的IO类库比如InputStream和BufferInputStream。还有Collections类。

- Heaven 2020-04-27 11:12:05
  个人认为,是属于建造者模式的,在其中,最主要的append方法,是将其抛给了父类AbstractStringBuilder,然后返回自己,其父类AbstractStringBuilder中维护了一个数组,并且可以动然扩容,在我们最后获取结果的toString()方法中,就是直接new String对象,这种模式其实更像是装饰器模式的实现

- 守拙 2020-04-27 10:21:44
  课堂讨论:
  StringBuilder应用了Builder模式. 其主要方式是append(), 即通过不断append创建复杂对象.
  不同于传统Builder模式的是:
  1. StringBuilder的目的是创建String, 但StringBuilder并不是String的内部类.
  2. StringBuilder的创建过程可以断续, 传统的Builder模式一次性填入参数后调用build()方法创建对象.
  3. StringBuilder通过内部维护字符数组(char[])的方式实现拼接.

- 全炸攻城狮 2020-04-27 09:57:28
  感觉不像建造者。append在StringBuilder的创建过程中不起任何作用，append真正用到的地方是StringBuilder创建好以后，对字符串的拼接。StringBuilder的创建是调用的构造方法

- Keep-Moving 2020-04-27 09:47:35
  感觉UnmodifiableCollection 更像是 Collection 类的代理类