50-装饰器模式：通过剖析JavaIO类库源码学习装饰器模式

上一节课我们学习了桥接模式，桥接模式有两种理解方式。第一种理解方式是“将抽象和实现解耦，让它们能独立开发”。这种理解方式比较特别，应用场景也不多。另一种理解方式更加简单，类似“组合优于继承”设计原则，这种理解方式更加通用，应用场景比较多。不管是哪种理解方式，它们的代码结构都是相同的，都是一种类之间的组合关系。

今天，我们通过剖析Java IO类的设计思想，再学习一种新的结构型模式，装饰器模式。它的代码结构跟桥接模式非常相似，不过，要解决的问题却大不相同。

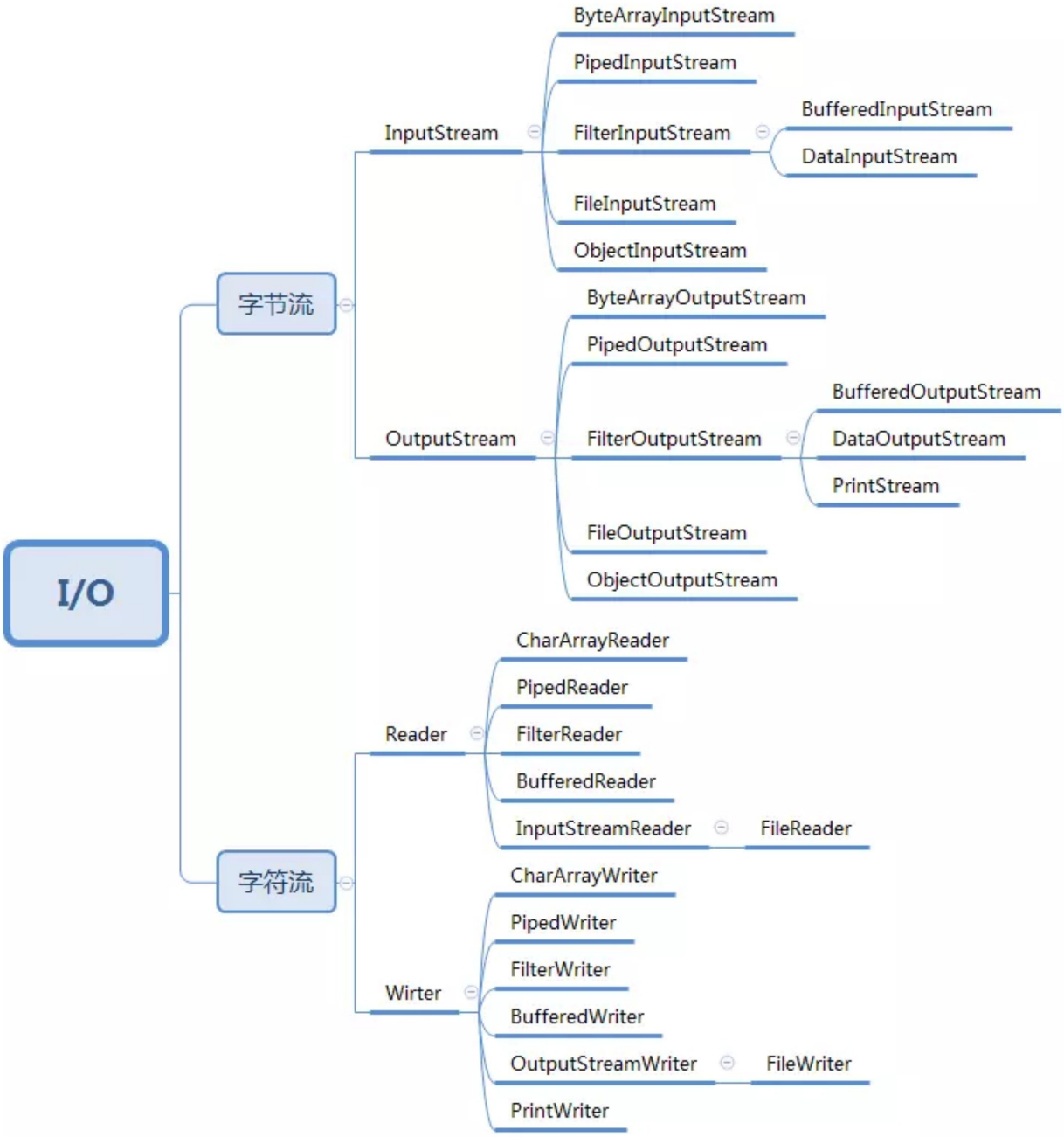
话不多说，让我们正式开始今天的学习吧！

Java IO类的“奇怪”用法

Java IO类库非常庞大和复杂，有几十个类，负责IO数据的读取和写入。如果对Java IO类做一下分类，我们可以从下面两个维度将它划分为四类。具体如下所示：

	字节流	字符流
输入流	InputStream	Reader
输出流	OutputStream	Writer

针对不同的读取和写入场景，Java IO又在这四个父类基础之上，扩展出了很多子类。具体如下所示：



极客时间

在我初学Java的时候，曾经对Java IO的一些用法产生过很大疑惑，比如下面这样一段代码。我们打开文件test.txt，从中读取数据。其中，InputStream是一个抽象类，FileInputStream是专门用来读取文件流的子类。BufferedInputStream是一个支持带缓存功能的数据读取类，可以提高数据读取的效率。

```
InputStream in = new FileInputStream("/user/wangzheng/test.txt");
InputStream bin = new BufferedInputStream(in);
byte[] data = new byte[128];
while (bin.read(data) != -1) {
  //...
}
```

初看上面的代码，我们会觉得Java IO的用法比较麻烦，需要先创建一个FileInputStream对象，然后再传递给BufferedInputStream对象来使用。我在想，Java IO为什么不设计一个继承FileInputStream并且支持缓存的BufferedFileInputStream类呢？这样我们就可以像下面的代码中这样，直接创建一个

BufferedFileInputStream类对象，打开文件读取数据，用起来岂不是更加简单？

```
InputStream bin = new BufferedFileInputStream("/user/wangzheng/test.txt");
byte[] data = new byte[128];
while (bin.read(data) != -1) {
  //...
}
```

基于继承的设计方案

如果InputStream只有一个子类FileInputStream的话，那我们在FileInputStream基础之上，再设计一个孙子类BufferedFileInputStream，也算是可以接受的，毕竟继承结构还算简单。但实际上，继承InputStream的子类有很多。我们需要给每一个InputStream的子类，再继续派生支持缓存读取的子类。

除了支持缓存读取之外，如果我们还需要对功能进行其他方面的增强，比如下面的DataInputStream类，支持按照基本数据类型（int、boolean、long等）来读取数据。

```
FileInputStream in = new FileInputStream("/user/wangzheng/test.txt");
DataInputStream din = new DataInputStream(in);
int data = din.readInt();
```

在这种情况下，如果我们继续按照继承的方式来实现的话，就需要再继续派生出DataFileInputStream、DataPipedInputStream等类。如果我们还需要既支持缓存、又支持按照基本类型读取数据的类，那就要再继续派生出BufferedDataFileInputStream、BufferedDataPipedInputStream等n多类。这还只是附加了两个增强功能，如果我们需要附加更多的增强功能，那就会导致组合爆炸，类继承结构变得无比复杂，代码既不好扩展，也不好维护。这也是我们在第10节中讲的不推荐使用继承的原因。

基于装饰器模式的设计方案

在第10节中，我们还讲到“组合优于继承”，可以“使用组合来替代继承”。针对刚刚的继承结构过于复杂的问题，我们可以通过将继承关系改为组合关系来解决。下面的代码展示了Java IO的这种设计思路。不过，我对代码做了简化，只抽象出了必要的代码结构，如果你感兴趣的话，可以直接去查看JDK源码。

```
public abstract class InputStream {
  //...
  public int read(byte b[]) throws IOException {
    return read(b, 0, b.length);
  }

  public int read(byte b[], int off, int len) throws IOException {
    //...
  }

  public long skip(long n) throws IOException {
    //...
  }
```

```
  public int available() throws IOException {
    return 0;
  }

  public void close() throws IOException {}

  public synchronized void mark(int readlimit) {}

  public synchronized void reset() throws IOException {
    throw new IOException("mark/reset not supported");
  }

  public boolean markSupported() {
    return false;
  }
}

public class BufferedInputStream extends InputStream {
  protected volatile InputStream in;

  protected BufferedInputStream(InputStream in) {
    this.in = in;
  }

  //...实现基于缓存的读数据接口...
}

public class DataInputStream extends InputStream {
  protected volatile InputStream in;

  protected DataInputStream(InputStream in) {
    this.in = in;
  }

  //...实现读取基本类型数据的接口
}
```

看了上面的代码，你可能会问，那装饰器模式就是简单的“用组合替代继承”吗？当然不是。从Java IO的设计来看，装饰器模式相对于简单的组合关系，还有两个比较特殊的地方。

第一个比较特殊的地方是：装饰器类和原始类继承同样的父类，这样我们可以对原始类“嵌套”多个装饰器类。比如，下面这样一段代码，我们对FileInputStream嵌套了两个装饰器类：BufferedInputStream和DataInputStream，让它既支持缓存读取，又支持按照基本数据类型来读取数据。

```
InputStream in = new FileInputStream("/user/wangzheng/test.txt");
InputStream bin = new BufferedInputStream(in);
DataInputStream din = new DataInputStream(bin);
int data = din.readInt();
```

第二个比较特殊的地方是：装饰器类是对功能的增强，这也是装饰器模式应用场景的一个重要特点。实际上，符合“组合关系”这种代码结构的设计模式有很多，比如之前讲过的代理模式、桥接模式，还有现在的装饰器模式。尽管它们的代码结构很相似，但是每种设计模式的意图是不同的。就拿比较相似的代理模式和装饰器模式来说吧，代理模式中，代理类附加的是跟原始类无关的功能，而在装饰器模式中，装饰器类附加

的是跟原始类相关的增强功能。

```
// 代理模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
  void f();
}
public class A impelements IA {
  public void f() { //... }
}
public class AProxy impements IA {
  private IA a;
  public AProxy(IA a) {
    this.a = a;
  }

  public void f() {
    // 新添加的代理逻辑
    a.f();
    // 新添加的代理逻辑
  }
}

// 装饰器模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
  void f();
}
public class A impelements IA {
  public void f() { //... }
}
public class ADecorator impements IA {
  private IA a;
  public ADecorator(IA a) {
    this.a = a;
  }

  public void f() {
    // 功能增强代码
    a.f();
    // 功能增强代码
  }
}
```

实际上，如果去查看JDK的源码，你会发现，BufferedInputStream、DataInputStream并非继承自InputStream，而是另外一个叫FilterInputStream的类。那这又是出于什么样的设计意图，才引入这样一个类呢？

我们再重新来看一下BufferedInputStream类的代码。InputStream是一个抽象类而非接口，而且它的大部分函数（比如read()、available()）都有默认实现，按理来说，我们只需要在BufferedInputStream类中重新实现那些需要增加缓存功能的函数就可以了，其他函数继承InputStream的默认实现。但实际上，这样做是行不通的。

对于即便是不需要增加缓存功能的函数来说，BufferedInputStream还是必须把它重新实现一遍，简单包裹对InputStream对象的函数调用。具体的代码示例如下所示。如果不重新实现，那BufferedInputStream类就无法将最终读取数据的任务，委托给传递进来的InputStream对象来完成。这一部分稍微有点不好理解，你自己多思考一下。

```
public class BufferedInputStream extends InputStream {
  protected volatile InputStream in;

  protected BufferedInputStream(InputStream in) {
    this.in = in;
  }

  // f()函数不需要增强，只是重新调用一下InputStream in对象的f()
  public void f() {
    in.f();
  }
}
```

实际上，DataInputStream也存在跟BufferedInputStream同样的问题。为了避免代码重复，Java IO抽象出了一个装饰器父类FilterInputStream，代码实现如下所示。InputStream的所有的装饰器类（BufferedInputStream、DataInputStream）都继承自这个装饰器父类。这样，装饰器类只需要实现它需要增强的方法就可以了，其他方法继承装饰器父类的默认实现。

```
public class FilterInputStream extends InputStream {
  protected volatile InputStream in;

  protected FilterInputStream(InputStream in) {
    this.in = in;
  }

  public int read() throws IOException {
    return in.read();
  }

  public int read(byte b[]) throws IOException {
    return read(b, 0, b.length);
  }

  public int read(byte b[], int off, int len) throws IOException {
    return in.read(b, off, len);
  }

  public long skip(long n) throws IOException {
    return in.skip(n);
  }

  public int available() throws IOException {
    return in.available();
  }

  public void close() throws IOException {
    in.close();
  }

  public synchronized void mark(int readlimit) {
    in.mark(readlimit);
  }

  public synchronized void reset() throws IOException {
    in.reset();
  }
```

```
  public boolean markSupported() {
    return in.markSupported();
  }
}
```

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。

装饰器模式主要解决继承关系过于复杂的问题，通过组合来替代继承。它主要的作用是给原始类添加增强功能。这也是判断是否该用装饰器模式的一个重要的依据。除此之外，装饰器模式还有一个特点，那就是可以对原始类嵌套使用多个装饰器。为了满足这个应用场景，在设计的时候，装饰器类需要跟原始类继承相同的抽象类或者接口。

课堂讨论

在上节课中，我们讲到，可以通过代理模式给接口添加缓存功能。在这节课中，我们又通过装饰者模式给InputStream添加缓存读取数据功能。那对于“添加缓存”这个应用场景来说，我们到底是该用代理模式还是装饰器模式呢？你怎么看待这个问题？

欢迎留言和我分享你的思考，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：

- 下雨天 2020-02-26 10:50:18
  你是一个优秀的歌手，只会唱歌这一件事，不擅长找演唱机会，谈价钱，搭台，这些事情你可以找一个经纪人帮你搞定，经纪人帮你做好这些事情你就可以安稳的唱歌了，让经纪人做你不关心的事情这叫代理模式。
  你老爱记错歌词，歌迷和媒体经常吐槽你没有认真对待演唱会，于是你想了一个办法，买个高端耳机，边唱边提醒你歌词，让你摆脱了忘歌词的诟病，高端耳机让你唱歌能力增强，提高了基础能力这叫装饰者模式。 [19赞]

- 小晏子 2020-02-26 10:03:00
  对于添加缓存这个应用场景使用哪种模式，要看设计者的意图，如果设计者不需要用户关注是否使用缓存功能，要隐藏实现细节，也就是说用户只能看到和使用代理类，那么就使用proxy模式；反之，如果设计者需要用户自己决定是否使用缓存的功能，需要用户自己新建原始对象并动态添加缓存功能，那么就使用decorator模式。 [15赞]

- rammelzzz 2020-02-26 01:40:16
  对于无需Override的方法也要重写的理解：
  虽然本身BufferedInputStream也是一个InputStream，但是实际上它本身不作为任何io通道的输入流，而传递进来的委托对象InputStream才能真正从某个“文件”（广义的文件，磁盘、网络等）读取数据的输入流。因此必须默认进行委托。 [7赞]

- 守拙 2020-02-26 10:40:26
  补充关于Proxy Pattern 和Decorator Pattern的一点区别:

  Decorator关注为对象动态的添加功能, Proxy关注对象的信息隐藏及访问控制.
  Decorator体现多态性, Proxy体现封装性.

reference:
https://stackoverflow.com/questions/18618779/differences-between-proxy-and-decorator-pattern [5赞]

- Jxin 2020-02-26 13:01:57
今天的课后题：
1.有意思，关于代理模式和装饰者模式，各自应用场景和区别刚好也想过。

1.代理模式和装饰者模式都是 代码增强这一件事的落地方案。前者个人认为偏重业务无关，高度抽象，和稳定性较高的场景（性能其实可以抛开不谈）。后者偏重业务相关，定制化诉求高，改动较频繁的场景。

2.缓存这件事一般都是高度抽象，全业务通用，基本不会改动的东西，所以一般也是采用代理模式，让业务开发从缓存代码的重复劳动中解放出来。但如果当前业务的缓存实现需要特殊化定制，需要揉入业务属性，那么就该采用装饰者模式。因为其定制性强，其他业务也用不着，而且业务是频繁变动的，所以改动的可能也大，相对于动代，装饰者在调整（修改和重组）代码这件事上显得更灵活。 [3赞]

- iLeGeND 2020-02-26 16:38:43

```
// 代理模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
void f();
}
public class A impelements IA {
public void f() { //... }
}
public class AProxy impements IA {
private IA a;
public AProxy(IA a) {
this.a = a;
}

public void f() {
// 新添加的代理逻辑
a.f();
// 新添加的代理逻辑
}
}

// 装饰器模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
void f();
}
public class A impelements IA {
public void f() { //... }
}
public class ADecorator impements IA {
private IA a;
public ADecorator(IA a) {
this.a = a;
```

```
}

public void f() {
// 功能增强代码
a.f();
// 功能增强代码
}
}
```

老师 上面代码结构完全一样啊 不能因为 f() 中写的 逻辑不同 就说是两种模式吧 [2赞]

- 唐朝农民 2020-02-26 12:13:08
  订单的优惠有很多种，比如满减，领券这样的是不是可以使用decorator 模式来实现 [2赞]

- 岁月神偷 2020-02-27 09:00:34
  我觉得应该用代理模式，当然这个是要看场景的。代理模式是在原有功能之外增加了其他的能力，而装饰器模式则在原功能的基础上增加额外的能力。一个是增加，一个是增强，就好比一个是在手机上增加了一个摄像头用于拍照，而另一个则是在拍照这个功能的基础上把像素从800W提升到1600W。我觉得通过这样的方式区分的话，大家互相沟通起来理解会统一一些。 [1赞]

- 守拙 2020-02-26 10:33:26
  如果仅添加缓存一个功能, 使用Proxy好一些, 如果还有其他需求, 使用Decorator好一些.
  如果让我选择的话, 我宁愿选择Decorator, 方便扩展, 符合开闭原则. [1赞]

- 忆水寒 2020-02-26 09:36:24
  如果只是需要对所有 对象的缓存功能进行增强（相当于缓存是新的功能了），则可以使用代理模式。
  如果只是对某一类对象进行增强，而这类对象有共同的接口或父类，则可以使用装饰模式。 [1赞]

- 木头 2020-02-26 07:33:39
  看业务场景，如果只是针对某个类的某一个对象，添加缓存，那么就使用装饰模式。如果是针对这个类的所有对象，添加缓存。那么就使用代理模式 [1赞]

- 松花皮蛋me 2020-02-26 00:23:12
  通过将原始类以组合的方式注入到装饰器类中，以增强原始类的功能，而不是使用继承，避免维护复杂的继承关系。另外，装饰器类通常和原始类实现相同的接口，如果方法不需要增强，重新调用原始类的方法即可。
  [1赞]

- Frank 2020-02-27 21:13:01
  打卡 设计模式-装饰器模式
  装饰器模式是一种类似于代理模式的结构型模式。主要意图是增强原始类的功能，可以实现多个功能的增强（即不同的功能单独一个类维护，使用该模式将其功能组合起来）。该模式主要是为了解决为了实现某些功能导致子类膨胀的问题。个人觉得主要体现了单一职责、组合优先于继承原则。主要应用场景有Java IO 流设计。但是有个疑惑，在Reader和Writer体系结构的设计中，并没有像InputStream和OutputStream那样设计一个过滤流类，而BufferedReader等直接继承了Reader。按照作者本文的分析，字符输入流直接跳过了使用中间类来继承的步骤，这样的设计又该如何理解？
  对于课堂讨论，我觉得应该使用装饰器模式，因为“添加缓存”这个功能跟原始功能是由直接关系的。而代理模式所面向主要是将框架代码与业务代码解耦合。

- Yayu 2020-02-27 19:00:55
  请问老师适配器模式和装饰器模式一样吗？有何本质不同？

- 往事随风，顺其自然 2020-02-27 07:01:50
  装饰器模式和代理模式结构完全一样，这个对？如果真的一样，无论选择哪个都可以，只是说附加功能区分类别不一样，与原始类无关或者有关，人为区分不一样，其实本质是一样。可以这样理解？勇接口实现是一样的，结构代码

- Boogie 捷 2020-02-26 23:14:50
  介于JAVA8可以支持default method 了，设计decorate pattern 的时候可以把 最底层的abstract 类（比如文中的InputStream）换成 interface 么？

- 黄林晴 2020-02-26 22:59:32
  打卡 白天在家办公
  晚上来着精华

- 高源 2020-02-26 22:31:59
  老师讲的很好啊😁这回知道应用场景了

- hanazawakana 2020-02-26 21:30:08
  因为inputstream里的方法有些是抽象的，还没实现。所以需要一个fileinputstream来实现抽象方法。这样bufferedinputstream只要继承需要装饰的方法

- 辣么大 2020-02-26 20:44:38
  装饰者模式：不创建对象，给已有对象新增功能。
  代理模式：控制对象访问，创建了新的对象。通常实现框架级别的缓存。

  代理模式现实生活中，让我想起了申请专利的代理。你写好技术交底书，专利代理人员帮你写申请材料，帮你交专利申请。
  装饰者模式相当于产品定制服务。例如我买了一台自行车（有基本功能），要给自行车配自己喜欢的配件（装饰）。