50 | 装饰器模式：通过剖析Java IO类库源码学习装饰器模式

王争 2020-02-26

00:00 08:18
讲述：冯永吉 大小：6.66M

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。

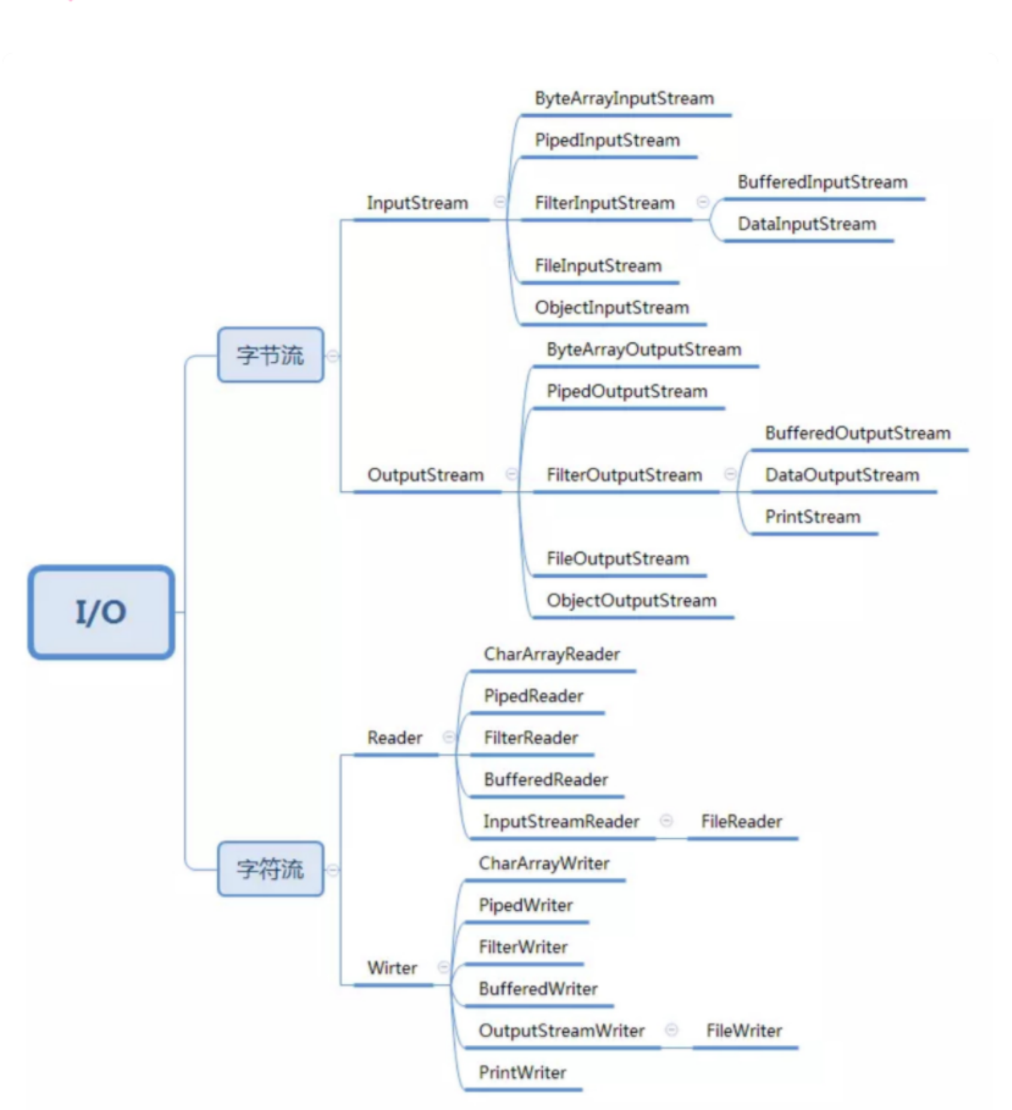
话不多说，让我们正式开始今天的学习吧！

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：



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。

复制代码

```
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？

复制代码

```
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。

复制代码

```
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。

复制代码

```
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。

复制代码

```
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。

复制代码

```
// 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。

复制代码

```
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。

复制代码

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

  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 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。