第27章

装饰器（Decorator）模式

通常，你会添加一些新类或新的方法去扩展已有的代码库。然而有时，你也想要在程序运行期间为某个对象组合新的行为。比如，借助解释器模式，你可以组合一个行为变化非常快的对象，这取决于你如何组合它。在某些情况下，你需要对象的行为发生一些细小的变化，并且这些变化可以进行组合。这时，装饰器模式就可以满足这个需求。

装饰器模式的意图是在运行时组合操作的新变化。

经典范例：流和输出器

Java 类库中输入输出流的设计思路是应用装饰器模式的一个经典范例。流是一系列字节或字符的集合，就像文档中的字符集合一样。在 Java 中，Writer 类是支持流的一个方式。某些输出器类的构造函数参数是其他的输出器，因此你可以基于一个输出器创建另一个输出器。像这种简单的组合就是装饰器模式的典型结构。Java 中的输出器就体现了装饰器模式的思路。同时，我们也应该看到，仅仅需要很少的代码量，借助于装饰器模式就可以大大扩展我们在读（输入）和写（输出）操作中融合小变化的能力。

下面是 Java 中的一个应用装饰器模式的例子，代码创建了一个小的文本文件：

```
package app.decorator;
import java.io.*;

public class ShowDecorator {
```

```
    public static void main(String[] args)
            throws IOException {
        FileWriter file = new FileWriter("sample.txt");
        BufferedWriter writer = new BufferedWriter(file);
        writer.write("a small amount of sample text");
        writer.newLine();
        writer.close();
    }
}
```

运行这段程序将会产生一个名为 sample.txt 的文本文件，其内容为很少量的简单文本。它使用 FileWriter 对象创建新文件，并把这个对象包含在 BufferedWriter 对象中。其中，最值得注意的是，我们可以从一个流组合另一个流：这段代码从 FileWriter 对象组合得到 BufferedWriter 对象。

在 Oozinoz 公司中，销售人员需要从产品数据库的文本文件中格式化出客户信息。这些信息没有使用形式多样的字体或者风格，但是现在销售人员希望信息能够进行格式化调整，使之更加整洁漂亮。为支持这个计划，我们创建了装饰器框架。这些装饰器类允许组合很多不同种类的输出过滤器。

为了开发过滤器类集合，需要首先创建一个抽象类，它将定义一些过滤器所支持的操作。通过选择已经存在于 Writer 类中的操作，可以轻松创建一个类，该类可以继承 Writer 类的所有行为。图 27.1 说明了这种思路。

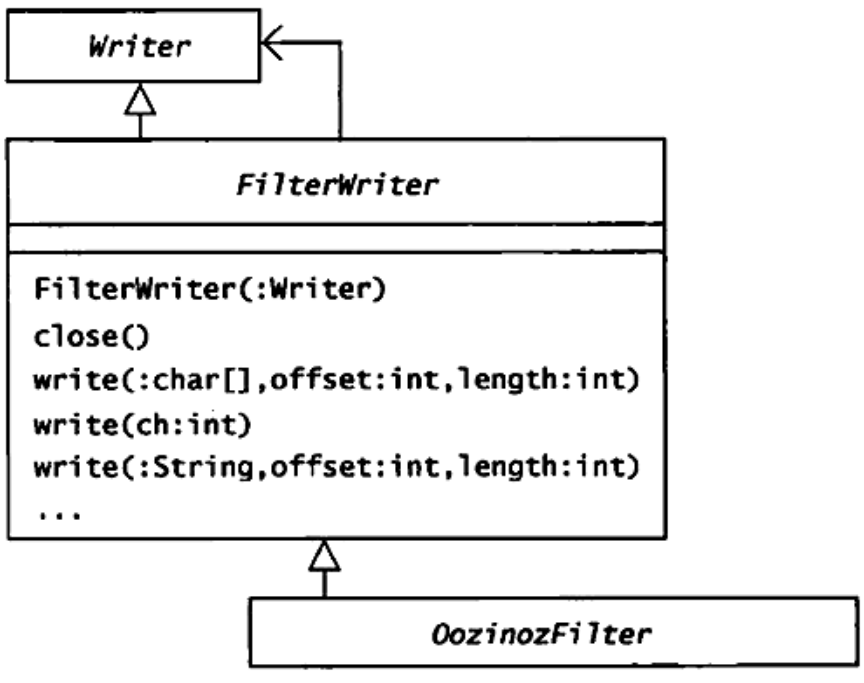

图 27.1　OozinozFilter 类是装饰输出字符流的类的超类

我们将定义一个过滤器类，将输出器作为它的构造函数参数，并且在 write()方法中混入新的行为。

为了创建一个可组合的输出流工具箱，下一步将创建一个具有多个关键属性的过滤器超类。这个过滤器类将：

- 接收一个 Writer 对象作为其构造函数参数。
- 作为一个过滤器类层次的超类。
- 提供所有 Writer 方法（write(:int)除外）的默认实现。

这种设计思路如图 27.2 所示。

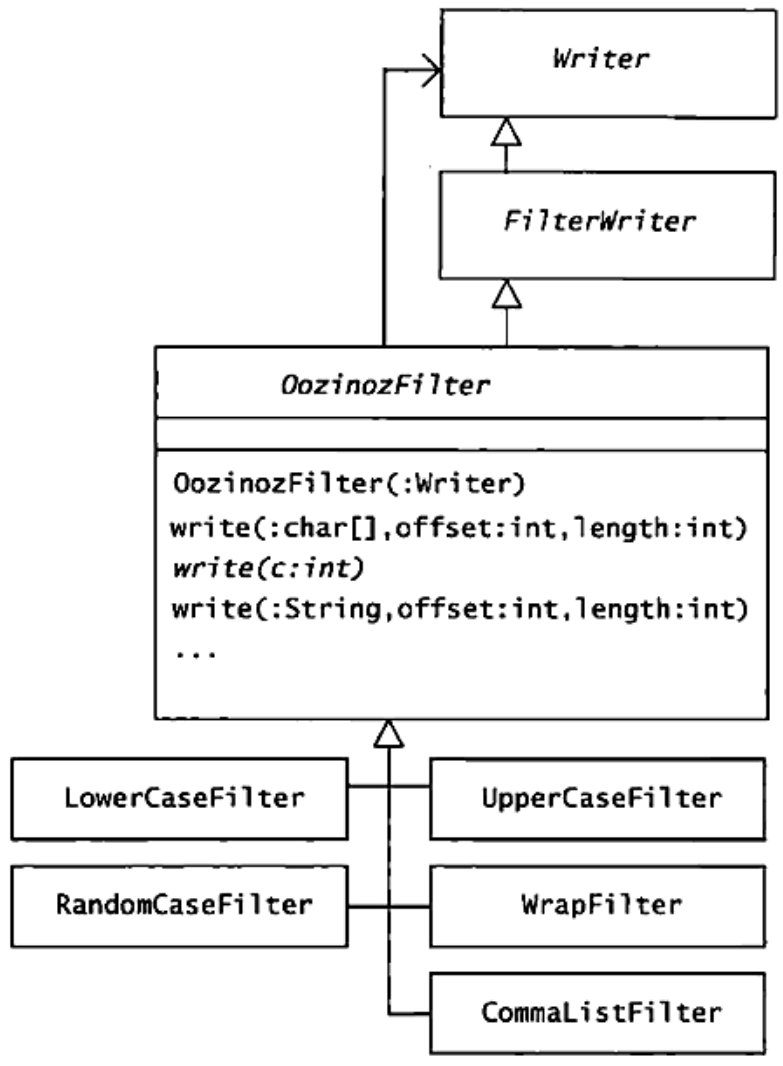

图 27.2　OozinozFilter 类的构造函数接收 Writer 类的任何子类的实例

OozinozFilter 类的设计编码实现如下所示：

```
package com.oozinoz.filter;
```

```
import java.io.*;
public abstract class OozinozFilter extends FilterWriter {
    protected OozinozFilter(Writer out) {
        super(out);
    }

    public void write(char cbuf[], int offset, int length)
                    throws IOException {
        for (int i = 0; i < length; i++)
            write(cbuf[offset + i]);
    }

    public abstract void write(int c) throws IOException;

    public void write(String s, int offset, int length)
            throws IOException {
        write(s.toCharArray(), offset, length);
    }
}
```

这段代码足以保证装饰器模式的实现。OozinozFilter 类的子类可以实现 write(:int)方法，在传递给底层流的 write(:int)方法之前对字符进行修改。OozinozFilter 类中的其他方法提供了子类需要的行为。该类把 close()方法和 flush()方法的调用丢给了它的超类 FilterWriter。OozinozFilter 类通过将 write(:int)方法定义为抽象方法，对 write(:char[])方法进行了说明。

现在很容易创建和使用新的流过滤器。比如，下面的代码把文本全部变成了小写格式：

```
package com.oozinoz.filter;
import java.io.*;

public class LowerCaseFilter extends OozinozFilter {
    public LowerCaseFilter(Writer out) {
            super(out);
    }
    public void write(int c) throws IOException {
            out.write(Character.toLowerCase((char) c));
    }
}
```

如下代码使用了这个小写过滤器：

```
package app.decorator;

import java.io.IOException;
import java.io.Writer;

import com.oozinoz.filter.ConsoleWriter;
import com.oozinoz.filter.LowerCaseFilter;

public class ShowLowerCase {
    public static void main(String[] args)
            throws IOException {
        Writer out = new ConsoleWriter();
        out = new LowerCaseFilter(out);
        out.write("This Text, notably ALL in LoWeR casE!");
        out.close();
    }
}
```

这段程序向控制台输出“this text, notably all in lower case!”。

UpperCaseFilter 类的实现代码与 LowerCaseFilter 类类似，除了 write()方法的实现略有不同，差别如下：

```
public void write(int c) throws IOException {
            out.write(Character.toUpperCase((char) c));
}
```

TitleCaseFilter 类的代码实现稍微复杂一些，因为它需要跟踪空格。

```
package com.oozinoz.filter;
import java.io.*;

public class TitleCaseFilter extends OozinozFilter {
        boolean inWhite = true;
public TitleCaseFilter(Writer out) {
            super(out);
}

public void write(int c) throws IOException {
```

```
                out.writre(
                    i nWhite
                        ? Character.toUpperCase((char) c)
                        : Character.toLowerCase((char) c));
                inWhite = Character.isWhitespace((char) c)
                || c == '"';
        }
}
```

CommaListFilter 类在元素之间添加了逗号：

```
package com.oozinoz.filter;

import java.io.IOException;
import java.io.Writer;

public class CommaListFilter extends OozinozFilter {
    protected boolean needComma = false;

    public CommaListFilter(Writer writer) {
        super(writer);
    }

    public void write(int c) throws IOException {
        if (needComma) {
            out.write(',');
            out.write(' ');
        }
        out.write(c);
        needComma = true;
    }


    public void write(String s) throws IOException {
        if (needComma)
            out.write(", ");

        out.write(s);
        needComma = true;
    }
}
```

这些过滤器的主题是一样的：开发任务主要是重写合适的 write()方法。write()方法装饰已经接收到的文本流，并把修改后的文本传递给接下来的流。

挑战 27.1

完成 RandomCaseFilter.java 的程序代码。

答案参见第 359 页

WrapFilter 类的代码比其他过滤器复杂很多。它需要将输出集中处理，因此在传递给接下来的流之前，需要缓存输出并计算字符数量。你可以从 www.oozinoz.com 网站下载查看 WrapFilter.java 代码（下载的详细信息参见附录 C）。

WrapFilter 类的构造函数接收一个 Writer 对象和一个 width 参数，其中 width 参数可以告知从何处换行。你可以将这个过滤器和其他过滤器混合使用去输出各种各样的效果。比如，如下程序会对输入文件中的文本进行换行、居中以及首字母大写等处理。

```
package app.decorator;
import java.io,*;
import com.oozinoz.filter.TitleCaseFilcer;
import com.oozinoz.filter.WrapFilter;

public class ShowFilters {
    public static void main(String args[])
            throws IOException {
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        Writer out = new FileWriter(args[1]);
        out = new WrapFilter(new Bufferedwriter(out), 0);
        out = new TitleCaseFilter(out);

        String line;
        while ((line = in.readLine()) != null)
            out.write(line + "\n");

        out.close();
        in.close();
    }
}
```

为了查看这段程序的动态运行效果，假定输入文件 adcopy.txt 包含下面的内容：

```
The "SPACESHOT" shell    hovers
          at 100 meters for 2 to 3
minutes,       erupting star bursts  every 10 seconds that
generate            ABUNDANT reading-level light for a
typical   stadium.
```

以命令行的方式执行 ShowFilters 程序：

```
>ShowFilters adcopy.txt adout.txt
```

adout.txt 文件的内容将会类似于：

```
  The "Spaceshot" Shell Hovers At 100
Meters For 2 To 3 Minutes, Erupting Star
 Bursts Every 10 Seconds That Generate
   Abundant Reading-level Light For A
            Typical Stadium.
```

如果不写入文件，也可以把输出字符直接写到控制台。图 27.3 给出了 Writer 类的子类 ConsoleWriter 的设计，它能将输出字符直接输出到控制台。

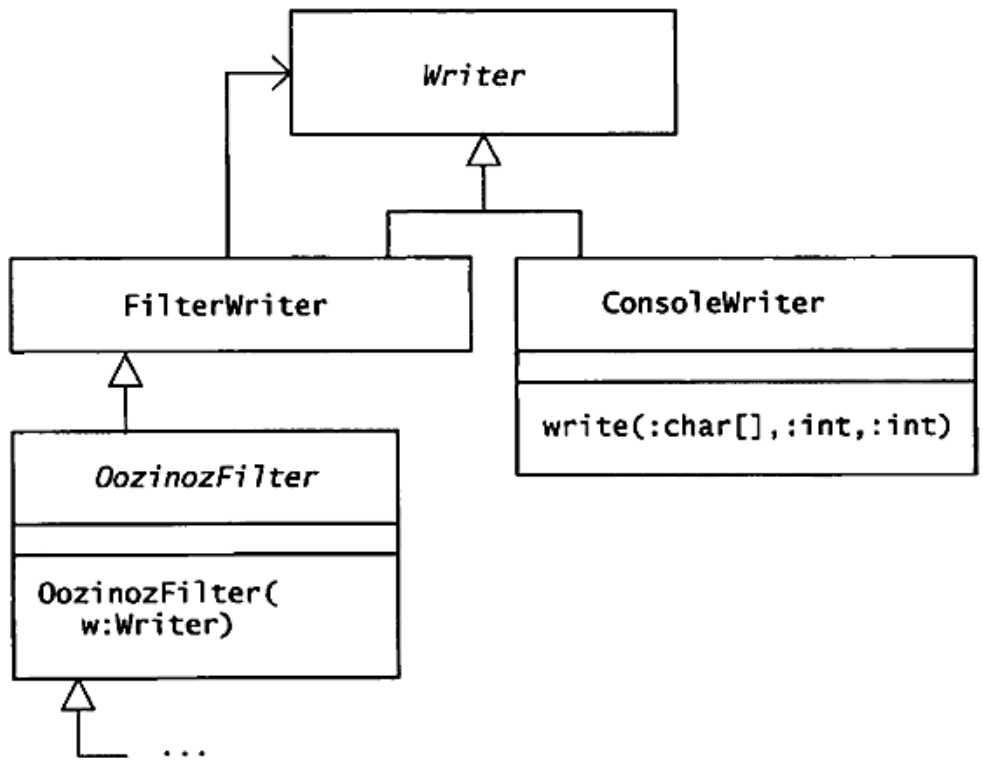

图 27.3　ConsoleWriter 对象可以作为任何 OozinozFilter 子类的构造函数参数

挑战 27.2

写出 ConsoleWriter.java 的代码实现。

答案参见第 360 页

输入输出流是关于装饰器模式如何在程序运行时组合对象行为的经典实例。装饰器模式的另一个重要应用则是在运行时创建数学函数。

函数包装器

如同装饰器模式在输入输出流应用中获得的巨大成功，通过使用装饰器模式在运行时组合新的行为并创建数学函数也取得了不错的效果。运行时创建新函数的能力可以传递给用户，使得他们通过 GUI 或者简单语言就可以实现新函数。通过把数学函数作为对象而不是新的方法来创建，可以减少代码里方法的数量，还可以获得更好的灵活性。

为了创建函数装饰器库（函数包装器），可以使用一个类似于 I/O 流的类结构。对于函数包装器超类名，我们使用 Function。对于 Function 类的初始设计，可以模仿 OozinozFilter 类的设计思路，如图 27.4 所示。

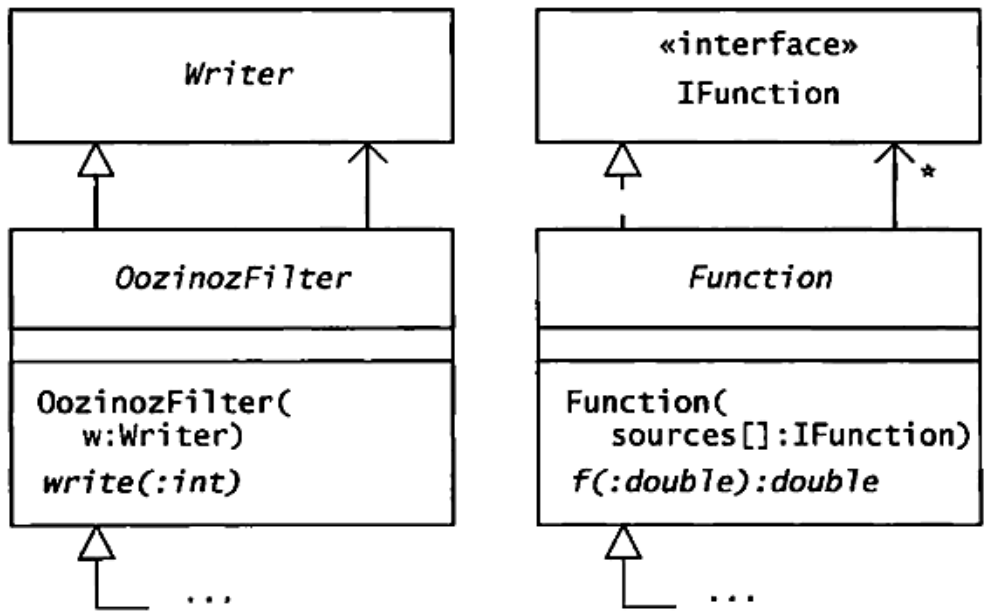

图 27.4　函数包装器（类结构）的初始设计类似于 I/O 过滤器的设计

OozinozFilter 类是 FilterWriter 类的子类，它的构造函数能够接收其他 Writer 对象。Function 类的设计思路与此类相似：不同于接收单个 IFunction 对象，Function 类接收一个数组。一些函数，比如算术函数，需要更多的下一级函数共同作用。

对于函数包装器，没有类像 Writer 类一样已经实现了我们需要的操作。因此，并不真正需要 IFunction 接口。通过删去该接口的定义，我们可以简化定义 Function 类的层次结构，如图 27.5 所示。

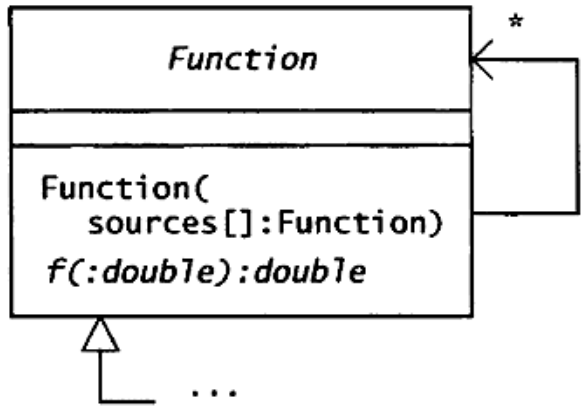

图 27.5　无须定义单独的接口，Function 类的简单设计同样可以实现需求

和 OozinozFilter 类一样，Function 类定义公共操作，其子类必须实现此公共操作。我们很自然地将这个公共操作命名为 f。同时，也可以计划实现参数函数，使所有函数基于一个标准的介于 0 到 1 之间的“time”参数。可以使用工具栏中的 Parametric Equations 菜单，在第 4 章还可以了解使用参数方程的影响力的背景。

对于希望包装的每个函数，我们都将为它们创建一个 Function 类的子类。图 27.6 是初始的 Function 类层次结构的设计思路。

Function 超类的代码主要用于声明 sources 数组：

```
package com.oozinoz.function;

public abstract class Function {
    protected Function[] sources;

    public Function(Function f) {
        this(new Function[] { f });
    }
```

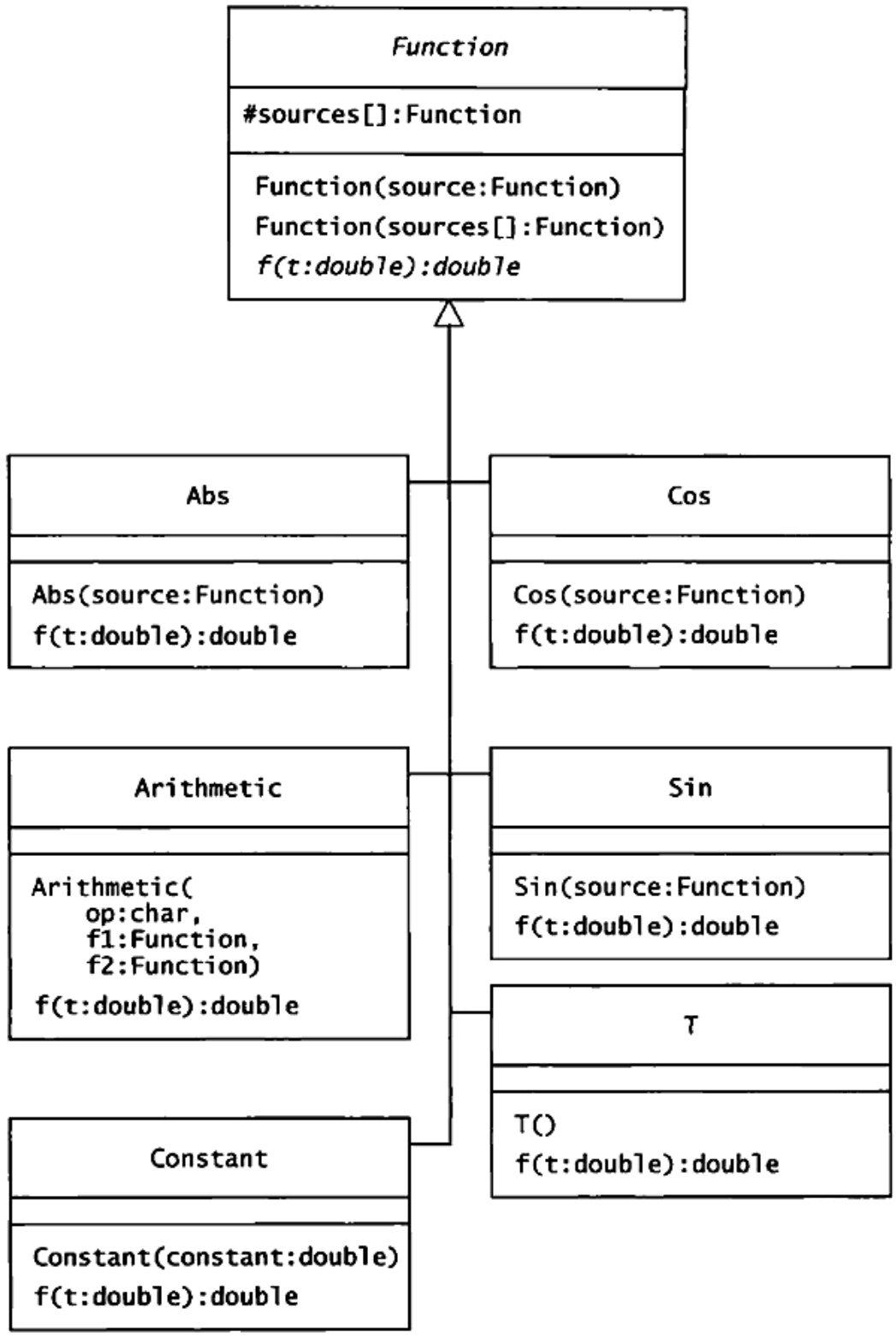

图 27.6　Function 类的每一个子类按照类名的不同含义实现 f(t)

```
public Function(Function[] sources) {
    this.sources = sources;
}
```

```
    public abstract double f(double t);

    public String toString() {
        String name = this.getClas().toString();
        StringBuffer buf = new StringBuffer(name);
        if (sources.length > 0) {
            buf.append('(');
            for (int i = 0; i < sources.length; i++) {
                if (i > 0)
                    bur.append(", ");
                buf.append(sources[i]);
            }
            buf.append(')');
        }
        return buf.toString();
    }
}
```

通常，Function 子类都比较简单。例如，下面是 Cos 类的代码：

```
package com.oozinoz.tunction;
public class Cos extends Function {
    public Cos(Function f) {
        super(f);
    }

    public double f(double t) {
        return Math.cos(sources[0].f(t));
    }
}
```

Cos 类的构造函数期待接收一个 Function 对象参数，并把这个参数传递给它的超类构造函数，参数存储在 sources 数组中。Cos.f()方法在时间 *t* 对原函数求值，将值传递给 Math.Cos()，并返回结果。

Abs 类和 Sin 类与 Cos 类几乎完全一样。借助 Constant 类，可以创建拥有一个常量值的 Function 对象，并作为 f()方法的调用返回。Arithmetic 类接收一个操作符指示器，并将其运用到 f()方法中。Arithmetic 类的代码如下所示：

```
package com.oozinoz.functlon;
```

```
public class Arithmetic extends Function {
    protected char op;

    public Arithmetic(char op, Function f1, Function f2) {
        super(new Function[] { f1, f2 });
        this.op = op;
    }

    public double f(douhle t) {
        switch (op){
        case '+':
            return sources[0].f(t) + sources[1].f(t);
        case '-':
            return sources[0].f(t) - sources[1].f(t);
        case '*':
            return sources[0].f(t) * sources[1].f(t);
        case '/':
            return sources[0].f(t) / sources[1].f(t);
        default:
            return 0 ;
        }
    }
}
```

T 类返回传入的 t 值。如果希望一个变量随着时间线性变化，那么这样做会很有用。比如，下面的代码创建了一个 Function 对象，随着时间从 0 变化到 1，该对象的 f()值从 0 变化到 2π：

```
new Arithmetic('*', new T(), new Constant(2 * Math.PI))
```

可以使用 Function 类去组合新的数学函数，而无须编写新的方法。FunPanel 类为它的 x 和 y 函数接收 Function 参数。这个类也使得这些函数可以在固定面板范围内使用。这个面板还可以和如下代码结合使用：

```
package app.decorator;

import app.decorator.brightness.FunPanel;

import com.oozinoz.function.*;
import com.oozinoz.ui.SwingFacade;
```

```
public class ShowFun {
    public Static void main(String[] args) {
        Function theta = new Arithmetic(
            '*', new T(), new Constant(2 * Math.PI));
        Function theta2 = new Arithmetic(
            '*', new T(), new Constant(2 * Math.PI * 5));
        Function x = new Arthmetic(
            '+', new Cos(theta), new Cos(theta2));
        Function y = new Arithmetic(
            '+', new Sin(theta), new Sin(theta2));

        FunPanel panel = new FunPanel(1000);
        panel,setPreferredSize(
            new java.awt.Dimension(200, 200));

        panel.setXY(x, y);
        SwingFacade.launch(panel, "Chrysanthemum");
    }
}
```

这段程序画出了一组相互嵌套的圆形，程序运行后的效果如图 27.7 所示。

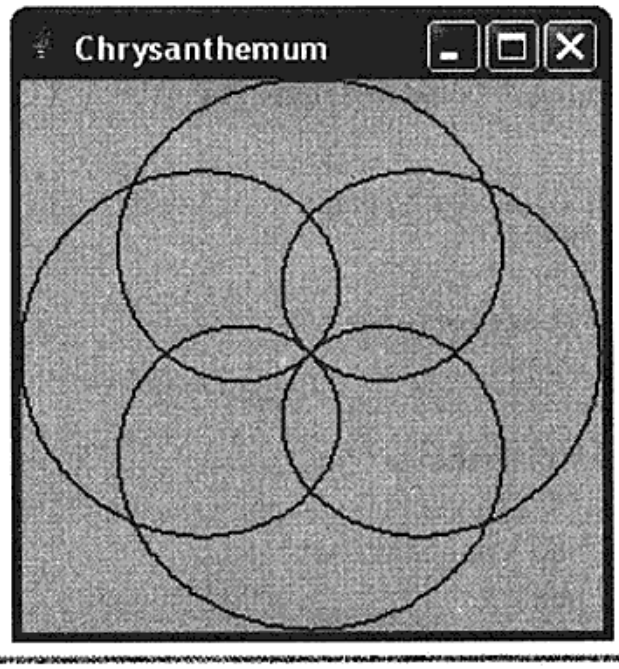

图 27.7　在没有引入任何新方法的情况下，一个复杂的数学函数被创建

通过简单地向 Function 类层次结构中加入新的数学函数，你就可以扩展你自己的函数包装器。

挑战 27.3

请写出 Exp 函数包装器类的代码实现。（试着合上书去编写代码！）

答案参见第 361 页

假定星形火药的亮度是（随时间）呈指数下降的正弦波函数：

$$brightness = e^{-4t} \cdot \sin(\pi t)$$

和前面一样，不需要编写新的类或方法就可以组合函数（绘制亮度随时间变化的曲线）：

```
package app.decorator.brightness;

import com.oozinoz.function.*;
import com.oozinoz.ui.SwingFacade;

public class ShowBrightness {
    public static void main(String args[]) {
        FunPanel panel = new FunPanel();
        panel.setPreferredSize(
            new java.awt.Dimension(200, 200));

        Function brightness = new Arithmetic(
            '*',
            new Exp(
                new Arithmetic(
                    '*',
                    new Constant(-4),
                    new T())),
                new Sin(
                    new Arithmetic(
                        '*',
                        new Constant(Math.PI),
                        new T())));

        panel.setXY(new T(), brightness);

        SwingFacade.launch(panel, "Brightness");
    }
}
```

这段代码所绘制出的曲线如图 27.8 所示。

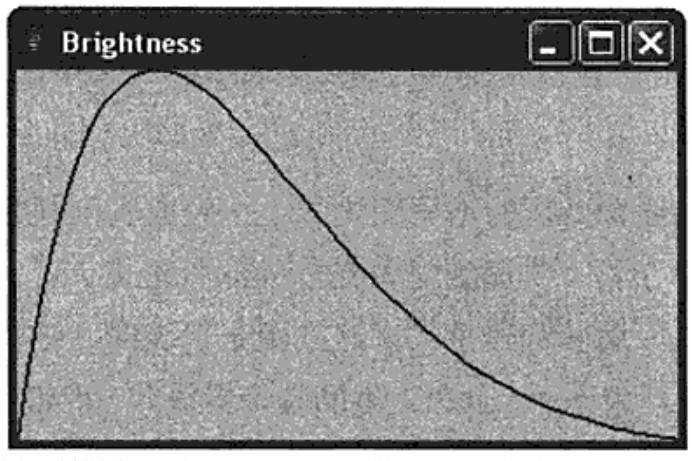

图 27.8　星形火药的亮度迅速达到顶点随后便开始衰减

挑战 27.4

请编写代码来定义表示亮度函数的 `Brightness` 对象。

答案参见第 361 页

你可以根据需要向 `Function` 类层次结构加入其他函数。例如，可能会加入 `Random` 类、`Sqrt` 类以及 `Tan` 类。也可以创建新的类层次结构来处理其他不同的类型，比如字符串类型，或者，采用不同的方法来定义 `f()`操作。例如，你可以把 `f()`定义为时间的二维或者三维函数。总而言之，不论创建了什么样的类层次结构，都可以使用装饰器模式在运行时组合出丰富的函数。

装饰器模式和其他设计模式的关系

装饰器模式的机制中包含一个跨越类层次结构实现的公共操作。从这个角度来讲，装饰器模式类似于 State（状态）模式、Strategy（策略）模式和 Interpreter（解释器）模式。在装饰器模式中，类通常拥有需要其他后继装饰器对象的构造函数。从这一点上讲，装饰器模式类似于合成模式。装饰器模式也类似于代理模式，因为装饰器类通常通过把调用转发给后继装饰器对象以实现公共操作。

小结

借助装饰器模式，可以混合操作的不同变化。一个经典实例是输入输出流，可以从其他流组合一个新的流。Java 类库在其 I/O 流实现中支持装饰器模式。可以扩展这个思路，创建自己的 I/O 过滤器集合。还可以应用装饰器模式来建立函数包装器，这样就能够根据有限的函数类集合来创建数量庞大的函数对象系列。此外，借助装饰器模式，能够灵活设计具有公共操作（这些公共操作往往具有不同的实现方式）的类，这样就可以在运行时集成新的混合的变化。