第28章

迭代器（Iterator）模式

若要通过新增集合类型扩展代码库，可以增加一个迭代器来完成扩展。本章主要讨论对集合类型进行迭代的例子。除了对新集合类型的迭代之外，在多线程环境下的迭代也会引出一系列有趣的问题。迭代看似简单，但是它并不能完全解决问题。

迭代器模式的意图是为顺序访问集合元素提供一种方式。

普通的迭代

Java 语言中提供了一系列方式来进行迭代：

- for、while 和 repeat 循环，通常使用整数索引。
- Enumeration 类（在 java.util 中）。
- Iterator 类（也在 java.util 中），在 JDK 1.2 中支持集合。
- for 循环的扩展（foreach），在 JDK 1.5 中添加。

我们将重点介绍迭代器类，而本小节将展示对循环的扩展。

一个 Iterator 类包含三个方法：hasNext()、next()和 remove()，如果迭代器不支持 remove()操作，会抛出 UnsupportedOperationException()异常。

for 循环扩展后的形式如下：

```
for (Type element : collection)
```

它将针对集合创建一个循环，每次取出集合中的一项：这里称为 element。它并不需要显式地将元素转换为特定类型，转换过程都是隐式完成的。该结构同样适用于数组。如果一个类希望支持 for 循环，必须实现 Iterable 接口，并提供一个 iterator()方法。

下面的程序演示了 Iterator 类和增强后的 for 循环：

```
package app.iterator;

import java.util.ArrayList
import java.util.Iterator;
import java.util.List;

public class ShowForeach {
    public static void main(String[] args) {
        ShowForeach example = new ShowForeach();
        example.showIterator();
        System.out.println();
        example.showForeach();
    }

    public void showIterator() {
        List names = new ArrayList();
        names.add("Fuser:1101");
        names.add("StarPress:991");
        names.add("Robot:1");

        System.out.println("JDK 1.2-style Iterator:");
        for (Iterator it = names.iterator(); it.hasNext();) {
            String name = (String) it.next();
            System.out.println(name);
        }
    }

    public void showForeach() {
        List<String> names = new ArrayList<String>();
        names.add("Fuser:1101");
```

```
        names.add("StarPress:991");
        names.add("Robot:1");

        System.out.println(
            "JDK 1.5-style Extended For Loop:");
        for (String name: names)
            System.out.println(name);
    }
}
```

当运行该程序后，将得到如下输出：

```
JDK 1.2-style Iterator:
Fuser:1101
StarPress:991
Robot:1

JDK 1.5-style Extended For Loop:
Fuser:1101
StarPress:991
Robot:1
```

到目前为止，Oozinoz 公司还在继续使用旧版本的 Iterator 类，并且暂时无法升级，除非确认客户有新版本的编译器。尽管如此，从今天开始你就该为学习扩展的 for 循环做好准备了。

线程安全的迭代

交互良好的应用程序通常使用线程来同时执行多个任务。对于那些耗费时间的任务，让它们在后台执行是极为常见的做法，因为这样就不会拖延 GUI 的响应时间。线程很有用，但也很危险。许多应用程序崩溃的原因都是由于没有协调好线程之间的交互。毫无疑问，迭代整个集合的方法常常是导致多线程程序失败的原因。

java.util.Collections 包中的集合类，通过提供一个 synchronized()方法来保证线程执行的安全。从本质上讲，该方法将确保两个线程在底层访问的是同一个集合版本，并会阻止多个线程在同一时间更改集合。

在迭代过程中，集合和它的迭代器相互协作共同检测列表是否变化，即判断列表是否被同步。为了说明这种行为的工作方式，以 Oozinoz 系统中的 Factory 单例为例，该对象能告诉我们在指定时间内启动了哪些机器，并显示这一组启动了的机器。app.iterator.concurrent 包中的示例程序在 upMachineNames()方法中硬编码了这段代码。

下面的代码显示了当前已经启动的机器列表，当新机器启动时，它还可以模拟列表的显示情况：

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentIterator implements Runnable {
    private List list;

    protected static List upMachineNames() {
        return new ArrayList(Arrays.asList(new String[] {
            "Mixer1201", "ShellAssembler1301",
            "StarPress1401", "UnloadBuffer1501" }));
    }

    public static void main(String[] args) {
        new ShowConcurrentIterator().go();
    }

    protected void go() {
        list = Collections.synchronizedList(
            upMachineNames());
        Iterator iter = list.iterator();
        int i = 0;
        while (iter.hasNext()) {
            i++;
            if (i == 2) { // 模拟唤醒线程
                new Thread(this).start();
                try { Thread.sleep(100); }
                catch (InterruptedException ignored) {}
            }
            System.out.println(iter.next());
        }
    }
```

```
    /**
     ** 在一个单独的线程中，向 list 插入一个元素
     */
    public void run() {
        list.add(0, "Fuser1101");
    }
}
```

这段代码中的 main()函数创建了该类的实例，并且调用了该实例的 go()方法。该方法负责迭代已经启动的机器列表，并构造该列表的同步版本。这段代码模拟了方法在迭代这一列表时，其他机器启动的情况。run()方法会修改列表，并运行在一个独立的线程中。

在输出一台或者两台机器后，ShowConcurrentIterator 程序就崩溃了：

```
Mixer1201
java.util.ConcurrentModificationException
atjava.util.AbstractList$Itr.checkForComodification(Unknown Source)
atjava.util.AbstractList$Itr.next(Unknown Source)
atcom.oozinoz.app.ShowConcurrent.ShowConcurrentIterator.go(Show-
ConcurrentIterator.java:49)
atcom.oozinoz.app.ShowConcurrent.ShowConcurrentIterator.main(Show-
ConcurrentIterator.java:29)
Exception in thread "main" .
```

程序崩溃的原因是因为集合和迭代对象在迭代过程中检测到了集合的变化。其实，无须使用新的线程也可以展示这种行为：创建一个程序，在迭代过程中改变其集合元素，就可以让该程序崩溃。实际上，一个多线程程序在迭代器遍历集合的过程中，更可能意外地更改该集合。

我们可以在迭代集合的过程中，设计一个线程安全的方法；然而，引起 ShowConcurrentIterator 程序崩溃的关键原因是因为它使用了迭代器。在 for 循环中使用同步列表来进行迭代，就不会引发前面程序中出现的异常，但是依然会带来一些问题。考虑下面给出的另一个版本的程序：

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentFor implements Runnable {
    private List list;
```

```
protected static List upMachineNames() {
    return new ArrayList(Arrays.asList(new String[] {
        "Mixer1201", "ShellAssembler1301",
        "StarPress1401", "UnloadBuffer1501" }));
}
public static void main(String[] args) {
    new ShowConcurrentFor().go();
}

protected void go() {
    System.out.println(
        "This version lets new things "
        + "be added in concurrently:");

    list = Collections.synchronizedList(
        upMachineNames());
    display();
}
    private void display() {
        for (inti = 0; i<list.size(); i++) {
            if (i == 1) { // 模拟唤醒线程
                new Thread(this).start();
                try { Thread.sleep(100); }
                catch (InterruptedException ignored) {}
            }
            System.out.println(list.get(i));
        }
    }

    /**
     ** 在一个单独的线程中，向 list 中插入一个元素
     */
    public void run() {
        list.add(0, "Fuser1101");
    }
}
```

运行该程序，产生的输出如下所示：

```
This version lets new things be added in concurrently:
Mixer1201
Mixer1201
ShellAssembler1301
StarPress1401
UnloadBuffer1501
```

挑战 28.1

请解释 ShowConcurrentFor 程序的输出。

答案参见第 362 页

让我们来对比一下这两个版本的程序：一个程序在运行过程中崩溃，而另一个则产生了错误的输出。这两种结果我们都不能接受，因此在迭代过程中，还需要另外的方式来保护集合列表。

在多线程程序中，有两种常见的方式为集合提供安全的迭代。这两种方式都会引入一个对象，有时称为互斥（mutex）对象。该对象被多个线程共享，用于竞争对象的控制权。在其中一种设计方式中，需要所有的线程在访问集合前都要首先获得互斥锁。下面的程序展示了这种方式：

```
package app.iterator.concurrent;
import java.util.*;

public class ShowConcurrentMutex implements Runnable {
  private List list;

  protected static List upMachineNames() {
      return new ArrayList(Arrays.asList(new String[] {
          "Mixer1201", "ShellAssembler1301",
          "StarPress1401", "UnloadBuffer1501" }));
  }

  public static void main(String[] args) {
      new ShowConcurrentMutex().go();
  }

  protected void go() {
      System.out.println(
          "This version synchronizes properly:");
```

```
        list = Collections.synchronizedList(upMachineNames());
        synchronized (list) {
            display();
        }
    }
    private void display() {
        for (inti = 0; i<list.size(); i++) {
            if (i == 1) { // 模拟唤醒线程
                new Thread(this).start();
                try { Thread.sleep(100);
                } catch (InterruptedException ignored) {}
            }
            System.out.println(list.get(i));
        }
    }

    /**
     ** 在一个单独的线程中，向 list 插入一个元素
    */
    public void run() {
        synchronized (list) {
            list.add(0, "Fuser1101");
        }
      }
    }
```

该程序将输出原始的列表：

```
This version synchronizes properly:
Mixer1201
ShellAssembler1301
StarPress1401
UnloadBuffer1501
```

这份集合的输出结果，与插入新对象之前运行 run()方法的集合结果是一致的。程序的输出是连续的，没有重复，因为该程序的逻辑要求 run()方法在等待 display()方法完成后才能执行。尽管输出正确，但这种设计却有些华而不实，毕竟当一个线程正在迭代集合时，不可能让其他线程等待。

一个替代方案是在互斥操作中克隆该集合，并对克隆后的集合进行迭代。这样做的好处是

效率高。在操作集合之前克隆集合，常常要比等待其他操作集合内容的方法执行完成要快得多 。遗憾的是，在迭代前对集合进行克隆也可能会带来一些问题。

ArrayList 的 clone()方法会产生一个浅拷贝（shallow copy）：新集合和原始集合引用相同的对象。当其他线程通过某种方式改变了引用的对象后，我们所依赖的克隆就会失效。但在某些场景下，这种风险非常小。例如，如果只是希望显示集合中的机器名，在对克隆集合进行迭代时，修改机器名就是不合理的[译注1]。

总结下来，我们已经讨论了 4 种在多线程环境下迭代集合的方式。其中，使用 synchronized()方法的两种方式都失败了，不是程序崩溃，就是产生错误的结果。后两种方式使用了锁和克隆，都能产生正确的结果，但是也会带来一些问题。

挑战 28.2

请解释反对 synchronized()方法的理由，或者基于锁的方式为何不是我们要寻求的答案？

答案参见第 362 页

Java 类库对多线程环境下的迭代提供了大量支持，但这种并发设计都未能消除复杂度。对于内置集合的迭代，Java 类库提供了很好的支持，但是倘若希望引入自己的集合类型，就可能需要引入相应的迭代器。

基于合成结构的迭代

要设计遍历合成结构的算法，访问其中的每个节点，执行一些操作，其实并不困难。可以回顾一下挑战 5.3 中多种递归访问合成结构的算法。创建一个迭代器可能比创建一个递归算法

译注1：对于并发处理过程中相关问题的解决，最佳方式就是使用不变值。这里提到的对集合进行克隆的方式，其实就是保证每个集合对象都是不变的。只是在 Java 中，由于集合对象的特点，我们很难保证集合对象的不变性。而在 JVM 中的另一门语言 Scala 中，则通过 val 关键字，很好地保证了各种类型对象的不变性。

更加复杂。复杂之处在于返回对程序中其他部分的控制，保存多种状态，并让迭代器能从上次结束的地方继续开始迭代。

合成模式为迭代器提供了一个颇具挑战性的好例子。

你或许会认为需要给领域相关的每个组件都创建一个新的迭代器。事实上，可以先设计出一个可重用的合成迭代器，然后修改各组件类，使其利用该迭代器能进行正确的迭代。

这种合成迭代器的设计从本质上来讲都是递归的。为了迭代该合成结构，我们需要迭代其子节点。乍一听，这似乎有些复杂。首先，要确定是先返回前序节点还是后续节点（分别称为前序遍历和后续遍历）。如果选择前序遍历，就必须在返回节点头后，遍历其子节点。注意，或许每个子节点本身就是一个合成结构。这种方式的关键在于需要维护两个迭代器。一个迭代器负责记录当前是哪个子节点（图 28.1 中的标记 1）。它就是一个简单的列表迭代器，负责迭代子合成结构的列表。另一个迭代器（图 28.1 中的标记 2）将当前的子节点当做合成结构进行迭代。图 28.1 给出了在合成迭代中，决定当前节点的三个方面。

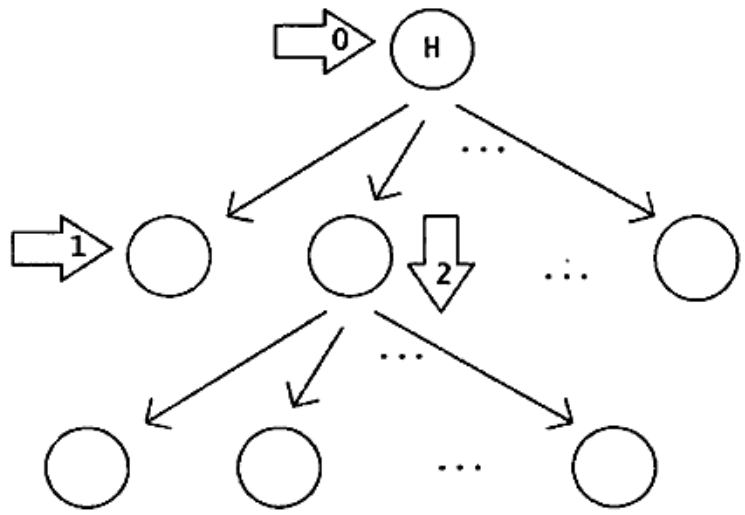

图 28.1　对合成结构进行迭代需要注意几个方面：0 代表头节点，1 代表顺序迭代其子节点，2 代表将每个子节点当做合成结构来进行迭代

要设计出合成迭代器，对叶子节点的迭代并不难，但对合成节点的迭代则可能比较困难。这些迭代器的设计如图 28.2 所示。

可以看到，类中包含了 `hasNext()`和 `next()`等方法，因此 `ComponentIterator` 类实现了 `java.util` 包中的 `Iterator` 接口。

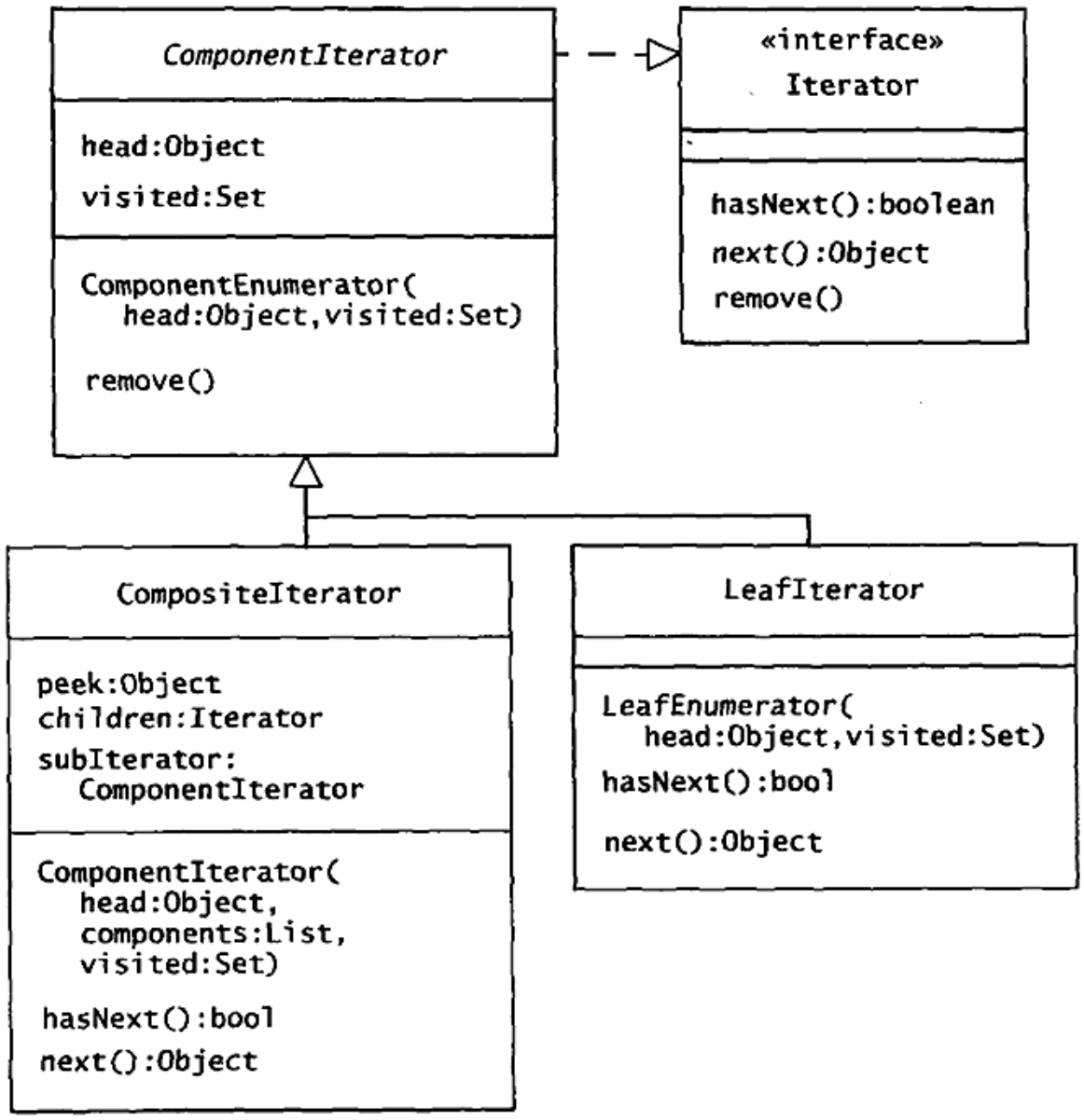

图 28.2　合成迭代器的最初设计

这一设计展现了迭代类的构造函数接收了一个对象进行迭代。在实际中，该对象将是一个合成对象，比如机器合成对象或者流程合成对象。该设计还使用了一个 visited 变量，用以记录已经迭代过的节点。这可以防止在合成结构包含了环时，不会进入无限循环。位于类层次结构顶部的 ComponentIterator 类的代码开始时如下所示：

```
package com.oozinoz.iterator;
import java.util.*;

public abstract class ComponentIterator
        implements Iterator {
    protected Object head;
```

```
    protected Set visited;
    protected boolean returnInterior = true;

    public ComponentIterator(Object head, Set visited) {
        this.head = head;
        this.visited = visited;
    }

    public void remove() {
        throw new UnsupportedOperationException(
            "ComponentIterator.Remove");
    }
}
```

该类将多数复杂的操作交给了子类去完成。

在 CompositeIterator 的子类中，我们期望使用一个列表迭代器，用来遍历合成节点的子节点。图 28.1 中标记为 1 的就是该迭代器，在图 28.2 中，使用了 children 变量来表示。合成结构也需要如图 28.1 中标记为 2 的迭代器。在图 28.2 中使用 subiterator 变量来表示。CompositeIterator 类的构造函数采用了如下形式初始化子迭代器：

```
public CompositeIterator(
        Object head, List components, Set visited) {
    super(head, visited);
    children = components.iterator();
}
```

当开始遍历一个合成结构时，返回的第一个节点是头节点（图 28.1 中标记为 H）。CompositeIterator 类中的 next()方法如下所示：

```
public Object next() {
    if (peek != null) {
        Object result = peek;
        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
```

```
            return head;
        }

        return nextDescendant();
    }
```

next()方法使用 visited 集合来记录迭代器是否已经返回了头节点。如果是，nextDescendant()方法就必须找到下一个节点。

无论何时，只要节点是合成节点，subiterator 变量就对其进行遍历。如果遍历的迭代器是有效的，CompositeIterator 类的 next()方法就可以移动这个子迭代器。倘若某个子迭代器无法移动，代码就必须移到子节点列表的下一个元素，为其分配一个新的子迭代器，并且移动该迭代器。nextDescendant()方法中的代码逻辑如下所示：

```
protected Object nextDescendant() {
    while (true) {
        if (subiterator != null) {
            if (subiterator.hasNext())
                return subiterator.next();
            }

            if (!children.hasNext()) return null;

            ProcessComponent pc =
                (ProcessComponent) children.next();
            if (!visited.contains(pc)) {
                subiterator = pc.iterator(visited);
            }
    }
```

对于支持遍历的对象类型，该方法引入了第一个约束：代码要求合成对象的子节点必须实现 iterator(:Set)方法。考虑合成结构的一个例子，比如第 5 章介绍的流程组件类层次结构。图 28.3 展示了 Oozinoz 公司对制造各种焰火弹组件的流程进行建模所形成的合成层次结构。

CompositeIterator 类的 next()方法需要去遍历合成对象的每个子节点。我们需要子类实现 iterator(:Set)方法，以便使用 next()中的代码。图 28.2 展示了这些类和接口之间的关系。

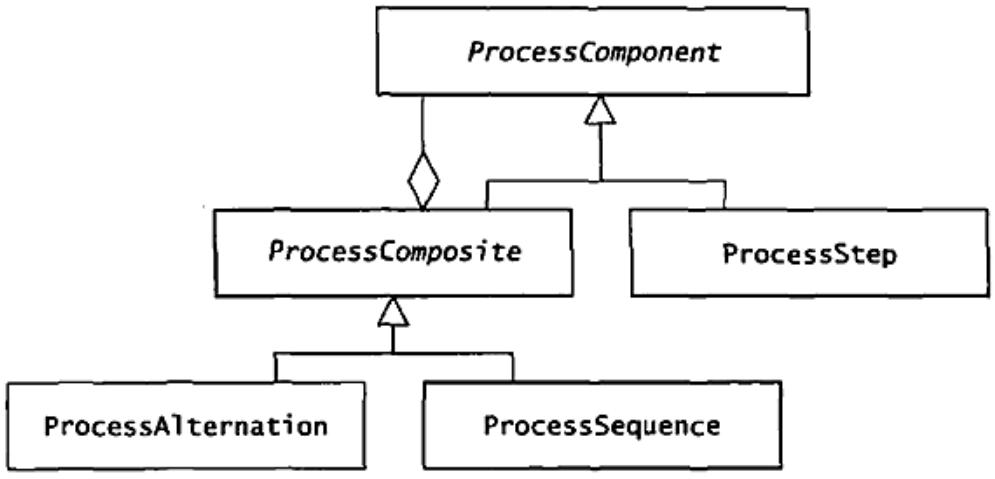

图 28.3 Oozinoz 公司的制造流程为合成结构

为了更新 ProcessComponent 类层次关系，以便能够遍历它，需要提供一个 iterator() 方法。

```
public ComponentIterator iterator() {
    return iterator(new HashSet());
}
public abstract ComponentIterator iterator(Set visited);
```

ProcessComponent 类是一个抽象类，它的抽象方法 iterator(:Set)由其子类实现。ProcessComposite 类的代码如下所示：

```
public ComponentIterator iterator(Set visited) {
    return new CompositeIterator(this, subprocesses, visited);
}
```

实现了 iterator()方法的 ProcessStep 类如下所示：

```
public ComponentIterator iterator(Set visited) {
    return new LeafIterator(this, visited);
}
```

挑战 28.3

如果让 ProcessComponent 类层次结构中的类实现 iterator()方法，从而创建合适的迭代器类实例，你会使用哪个模式？

答案参见第 362 页

在调整了 ProcessComponent 类层次结构的相应位置后，就可以为合成流程的遍历编写代码了。图 28.4 展示了 Oozinoz 公司典型的流程对象模型。

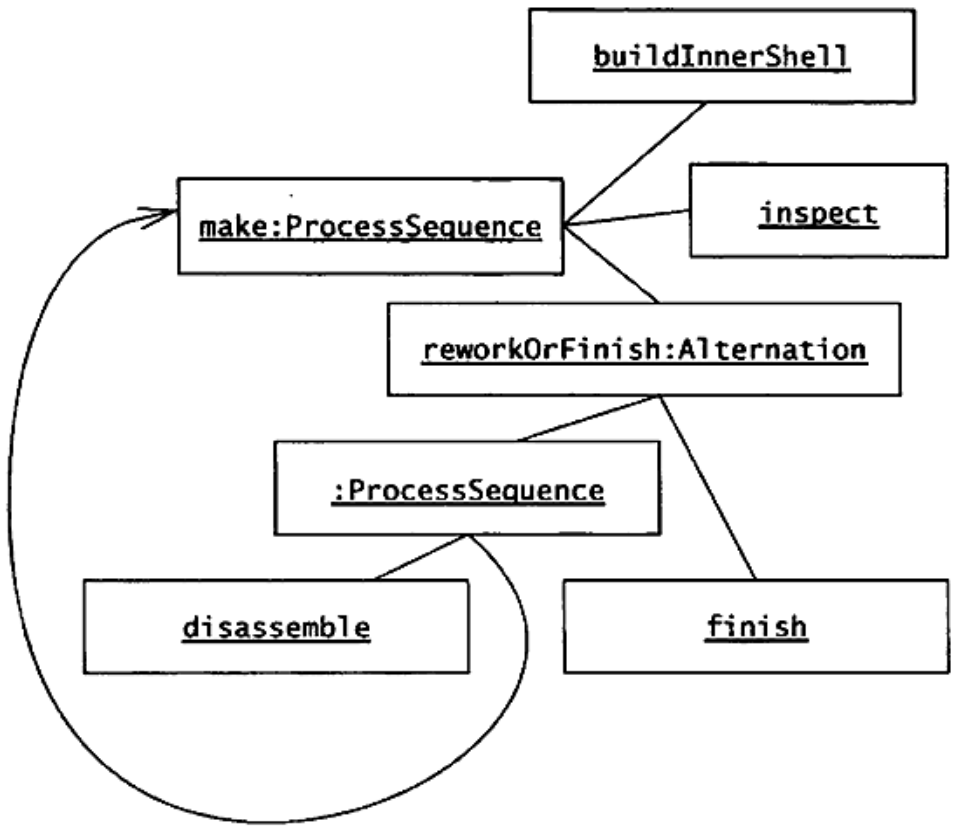

图 28.4　制作高空焰火弹的过程是一个循环的合成结构。图中的每个叶子节点都是一个 ProcessStep 类的实例。剩下的节点是 ProcessComposite 类的实例

com.oozinoz.processes 包中的 ShellProcess 类定义了一个静态的 make()方法，该方法返回图 28.4 所示的对象模型。下面的程序将会迭代模型中的所有节点。

```
package app.iterator.process;

import com.oozinoz.iterator.ComponentIterator;
import com.oozinoz.process.ProcessComponent;
import com.oozinoz.process.ShellProcess;

public class ShowProcessIteration {
    public static void main(String[] args) {
        ProcessComponent pc = ShellProcess.make();
        ComponentIteratoriter = pc.iterator();
        while (iter.hasNext())
        ·   System.out.println(iter.next());
    }
}
```

程序产生如下输出：

```
Make an aerial shell
Build inner shell
Inspect
Rework inner shell, or complete shell
Rework
Disassemble
Finish: Attach lift, insert fusing, wrap
```

输出的名字是 ShellProcess 类分配给每个步骤的名字。注意，在对象模型中，Disassemble 后的步骤是 make。输出忽略了这个步骤，因为第一行输出中已经输出过该步骤。

为合成迭代器增加深度

如果我们对每个步骤按其在模型中的深度进行缩进，则该程序的输出结果看起来将更加清晰。可以将叶迭代器的深度定义为 0，合成迭代器的当前深度是子迭代器的深度加 1。可以将超类 ComponentIterator 中的抽象方法 getDepth()定义为如下形式：

```
public abstract intgetDepth();
```

LeafIterator 类的 getDepth()方法如下所示：

```
public int getDepth() {
    return 0;
}
```

CompositeIterator.getDepth()中的代码如下所示：

```
public int getDepth() {
    if (subiterator != null)
        return subiterator.getDepth() + 1;
    return 0;
}
```

下面程序的输出具有更高的可读性：

```
package app.iterator.process;

import com.oozinoz.iterator.ComponentIterator;
import com.oozinoz.process.ProcessComponent;
```

```
import com.oozinoz.process.ShellProcess;

public class ShowProcessIteration2 {
    public static void main(String[] args) {
        ProcessComponent pc = ShellProcess.make();
        ComponentIteratoriter = pc.iterator();
        while (iter.hasNext()) {
            for (inti = 0; i< 4 * iter.getDepth(); i++)
                System.out.print(' ');
            System.out.println(iter.next());
        }
    }
}
```

程序的输出如下：

```
Make an aerial shell
    Build inner shell
    Inspect
    Rework inner shell, or complete shell
        Rework
            Disassemble
        Finish: Attach lift, insert fusing, wrap
```

另一种改进 ComponentIterator 类层次结构的方法是，只允许其遍历合成结构的叶子节点。

遍历叶子节点

假设只允许迭代返回叶子节点。如果需要了解叶子节点的属性，例如单个步骤的执行时间，这种做法是有价值的。可以为 ComponentIterator 类增加一个 returnInterior 字段，用来记录内部（非叶子）节点是否应该从遍历中返回：

```
protected boolean returnInterior = true;

public boolean shouldShowInterior() {
    return returnInterior;
}
```

```
public void setShowInterior(boolean value) {
    returnInterior = value;
}
```

在 CompositeIterator 类的 nextDescendant()方法中，当需要为合成节点的子节点创建一个新的迭代器时，需要将这一属性传递下去：

```
protected Object nextDescendant() {
    while (true) {
        if (subiterator != null) {
            if (subiterator.hasNext())
                    return subiterator.next();
        }
        if (!children.hasNext()) return null;

        ProcessComponent pc =
              (ProcessComponent) children.next();
        if (!visited.contains(pc)) {
            subiterator = pc.iterator(visited);
            subiterator.setShowInterior(
                        shouldShowInterior());
        }
    }
}
```

我们同样需要更新 CompositeEnumerator 类的 next()方法。代码如下：

```
public Object next() {
    if (peek != null) {
        Object result = peek;
        peek = null;
        return result;
    }

    if (!visited.contains(head)) {
        visited.add(head);
    }

    return nextDescendant();
}
```

挑战 28.4

更新 CompositeIterator 类中的 next()方法，使其获取 returnInterior 字段的值。

答案参见第 363 页

为某个新的集合类型创建一个迭代器（iterator），是一件非常有意义的设计任务。这样做的好处是自定义集合能够像 Java 类库中的集合那样轻松使用。

小结

迭代器模式的意图是让客户端类可以顺序地访问集合元素。Java 类库中的集合类为集合的操作提供了强大的支持，包括支持迭代或者遍历。若要创建新的集合类型，通常需要为其创建一个迭代器。与领域相关的合成结构就是一种新的集合类型。对通用集合的支持，以及对 for 循环的增强，可以使代码的可读性更好。可以设计一个通用的迭代器，并将其用到各种组件的层次结构中。

在实例化一个迭代器时，应该考虑对其集合的遍历是否更改了集合的内容。在单线程程序中，通常不会发生这种情况。但在多线程程序中，就需要确保对集合的访问是同步的。为了保证在多线程环境下迭代的安全，可以通过一个互斥对象来同步对集合的访问。当进行迭代时，可以阻塞其他对象对该集合的访问，或者通过克隆集合的方式来阻塞其他对象对该集合的访问。如果设计合理，可以为客户提供线程安全的迭代器代码。