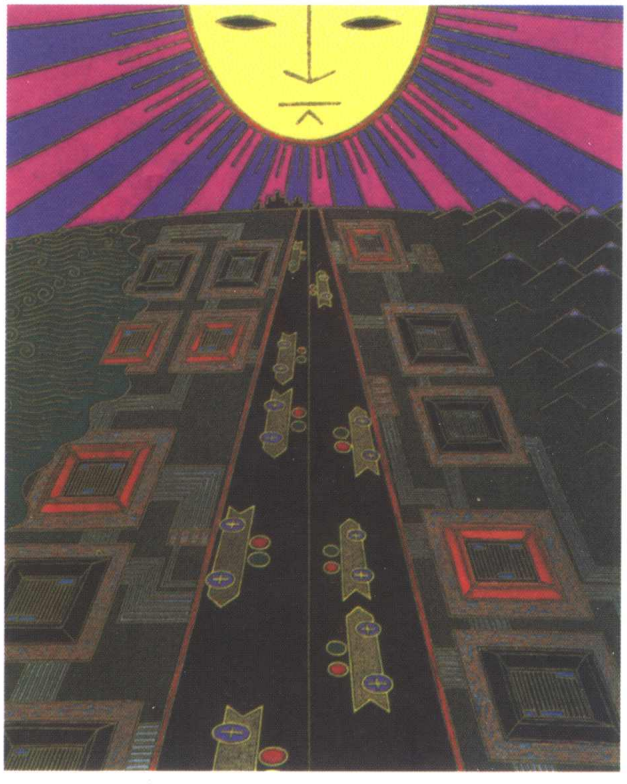
科海电脑技术丛书

Program Generators with XML and Java

用XML与JAVA

创建程序生成器



- 创建程序生成器的突破性技术，定制应用程序开发的自动化向导。
- 包括大量示例与实用指南。
- 提高软件开发效率的域工程技术。
- 光盘中包括示例程序，可执行代码和相关 XML 规范。

美] J.Craig Cleaveland 著　胡俊 刘吉强 译　黄厚宽 审校

Pearson Education

科学出版社

PH PTR Pearson Education 培生教育出版集团

Prentice-Hall 科海电脑技术丛书

用 XML 与 JAVA 创建程序生成器

Program Generators with XML and JAVA

[美] J.Craig Cleaveland 著

胡 俊 刘吉强 译

黄厚宽 审校

科 学 出 版 社
Pearson Education培生教育出版集团
2002.6

著作权合同登记号：01-2002-1511

内 容 提 要

在程序设计和开发过程中，使用程序生成器可以减少代码的编制工作。

本书将 XML，JAVA，JSP 及程序生成器等概念与方法结合起来，通过域分析的思路、方法、过程以及一些易于理解的示例，来介绍使用 XML 与 JAVA 创建程序生成器的方法与过程。书中并比较了不同方法的优劣，还介绍了许多创建程序生成器的捷径。

本书系统地说明了如何规划、设计及建立程序生成器。可作为程序开发人员及对程序生成器感兴趣的人员的参考书。

Program Generators with XML and Java

图书在版编目（CIP）数据

用 XML 与 JAVA 创建程序生成器/（美）克利夫兰（Cleaveland,J.C.）著；胡俊等译.
—北京：科学出版社，2002.5

ISBN 7-03-010434-X

Ⅰ. 用... Ⅱ. ①克...②胡... Ⅲ. ①可扩充语言，XML—程序设计
②JAVA 语言—程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2002）第 030824 号

科学出版社 出版

北京东黄城根北街 16 号

邮政编码：100717

北京市耀华印刷有限公司印刷

科学出版社发行 各地新华书店经销

*

2002 年 6 月第一版 开本：787×1092 1/16

2002 年 6 月第一次印刷 印张：18

印数：1-4000 字数：437 760

定价：36.00 元（含光盘）

前　言

很多年来我都在梦想写一本这样的书，但一直没能如愿。XML 和 JAVA 是不曾在我梦想中出现过的新技术。我已经从事程序生成器和专业工程方面的工作近 20 年，主要是用 C 语言和纯文本语言。XML 和 JAVA 给这些已有的技术提供了新的应用背景，并且两者相互协调补充。回顾多年来的经历，看到现在运用 XML 和 JAVA 进行那些艰苦的工作是如此简便而从容，真是一件令人高兴的事。

我体味着这本书的标题，感到编写一个程序生成器有许多的乐趣，因为它意味着不仅仅是写一个程序，而是要写一个可以写出许多程序的程序。想想做一个属于自己的程序生成器是多么聪明的一件事。有的程序生成器称为“向导”。程序生成器的效益非常高，并且也是许多软件工程开发环境中的重要部分，特别是在用户界面、数据库、中间件、语法分析和词法分析等方面。

当写过两三个程序生成器后，就会想，是否有一个更好的方法可以判断程序生成器应该完成什么？回答是肯定的，但答案仍是一个秘密。这个更好的方法称为**域工程**（domain engineering），它是一个确定某个专业的重要组成与需求的系统方法。这些信息对如何高效地建立一个满足客户需要的程序生成器是必要的。

许多计算机科学家，包括我自己，都着迷于自指示结构[1]。这样的结构是**程序生成器的生成器**（program generator generator）。程序生成器的生成器就是生成程序生成器的程序生成器，即一个能够编写可自动编程的程序的程序。第 12 章概述了我在贝尔实验室时在这方面所做的一些工作，但这里也是运用 XML 和 JAVA，以更简单而且优雅的风格体现的。任何人要想在几天之内创建自己的程序生成器的生成器都将是一件简单而容易的事。

---

(1) 指示自身的结构（例如这个句子）。

目　录

第 1 章　引言：字典问题

- 什么是程序生成器
- 程序生成器的结构
- 程序生成器的示例
- 为什么用程序生成器

Jack 和 Jill 有一个问题。他们正在写一个拼字（Scrabble）游戏的软件程序，并且需要一个用于查找单词的字典。做这件事最自然的方式就是在程序开始启动时读入一个单词文件。这个文件可能非常长，大约有 50 000 个单词，这取决于字典的容量，Jack 的工作是给一个名为 word.txt 的文件收集单词，该文件中每行写一个单词。文件开始时只有 7 个单词，计划明天或后天增加另外 49 993 个单词，如示例 1-1 所示。

示例 1-1　words.txt 文件

```
hill
fetch
pail
water
up
down
crown
```

其间，Jill 开始编写 Java 字典。她用了一种非常简单的方法。当建立 Dictionary0 对象时，将从 words.txt 文件中读入单词，并将它们存入 words 向量中。向量是一个变长的项目列表，如示例 1-2 所示。

示例 1-2　Dictionary0.Java

```
import java.io.*;
import java.util.*;

class Dictionary0 {
   Vector words = new Vector();

   Dictionary0() {
      loadWords();
```

```
    }

    void loadWords() {
        try {
            BufferedReader f = new BufferedReader(
                new FileReader("words.txt"));
            String word = null;
            while ((word = f.readLine())!=null) {
                words.addElement(word);
            }
        } catch (Exception e) {
            System.err.println("Unable to read from words.txt");
            // continue with empty dictionary
        }
    }

    public boolean isWord(String w) {
        for (int j=0; j<words.size(); ++j) {
            String w2 = (String) words.elementAt(j);
            if (w.equals(w2)) {
                return true;
            }
        }
        return false;
    }
}
```

Jill 仅需用 isword 方法就可以确定一个单词是否在该字典中，该方法对拼字游戏非常有用。但她却开始考虑，如果哪天自己想写其他的游戏程序，这个字典对象也许还能有用。她想，向量可能不是存储单词的最好的方式，线性搜索可能也不是最好的选择。但是作为一个好的工程师，她知道自己首先应保证这种方法正确，然后才可以使其做得更快。她真正要做的是测试程序并且保证每件事都能流畅而快速地执行。

第二天，Jack 完成了给字典增加另外 49 993 个单词的任务，而 Jill 在运行程序时发现了一个较大的问题。正如前面所提到的那样，线性搜索的速度真是太慢了。但由于在拼字游戏中并不经常查单词，所以这件事还不是太糟。真正的问题是需要花太长的时间去启动程序，读入一个包含 50 000 个单词的文件所付出的代价实在太高。

Jill 发出感叹，但随后想起，拼字游戏将以一个小应用程序的形式公布在 Web 上，而后它将被安装在一个 Java 手持游戏机上。但在这两种情况下，无论如何她都不能很容易地从一个文件中读入单词，因此，Jill 编写了一个更简单并且速度更快的程序。在这个程序中，单词被

存储在一个初始数 4 组中[1]，而不是从一个文件中读取，如示例 1-3 所示。

示例 1-3　Dictionary1.java

```
class Dictionary1 {

    String[] words = {
        "hill",
        "fetch",
        "pail",
        "water",
        "up",
        "down",
        "crown"
    };

    boolean isWord(String w) {
        for (int j=0; j<words.length; ++j) {
            if (w.equals(words[j])) {
                return true;
            }
        }
        return false;
    }
}
```

字典是程序的一部分，当程序启动时，单词不需要读入。接着，她运行新的程序并对它进行测试，结果令人满意。

同其他任何一个优秀的软件工程师一样，Jill 对两个程序间的不同之处作出了如下评估。如图 1-1 所示。

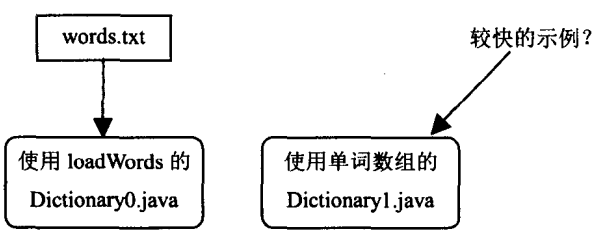


图 1-1　Dictionary0 与 Dictionary1

1. **性能**：两个程序在速度上不同。在 `Dictionary0` 中，必须从一个文件中读入单词并将

(1) 有些 Java 程序的执行在有大量的元素需要进行初始化时可能有困难。

其存储在一个数据结构中，在每次程序执行时都需要做这种运算。在 Dictionary1 中，该项工作在编译时完成，可执行的结果代码更长，因此进行加载需要更多的时间。但总体上看，这种方法比第一种方法快（对于 Java 手持游戏机如同闪电般地快），而且编译优化也可以促进性能的提高。words 数组可以声明为 final，这样，编译器可以据此创建更快更短的代码[2]。

2. **简洁**：简洁是对观看者的视觉而言。第一眼看上去，Dictionary1 显得比较简单，因为在此不需要初始化代码或 loadWords 方法。Jack 将会怎样想呢？

当 Jack 知道 Jill 的改变后有些心烦，因为他花费了整整一天的时间将 49 993 个单词键入到 word.txt 文件中去。Jack 将会怎么做呢？他确实不愿意带着引号和逗号将这些单词再输入一次。另外，与那些包含引号和逗号的杂乱的文件相比，他更喜欢自己的简单的文件。这时，他想出一个主意：写一个程序去读入文件 word.txt 中的单词，加入引号和逗号，并将数组数据输出到 Dictionary1 程序中，如图 1-2 和图 1-3 所示。他可以重用 Dictionary0 中的 loadWords 方法，且仅需要执行下列几行语句即可：

```
for  (int j=0; j<words.size(); ++j)  {
     System.out.println("   \""+words.elementAt(j)+"\", ");
}
```

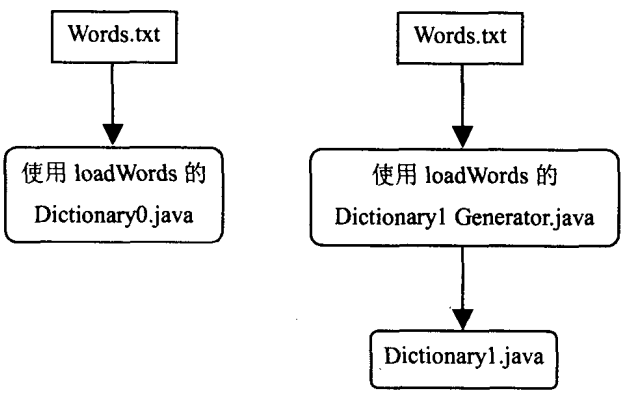


图 1-2 Dictionary1Generator 创建 Dictionary1.Java

Jack 喜欢这样做，因为这样他能够继续使用 word.txt，而转换程序将创建 Jill 在自己的程序中可用的文件。Jill 也认为这是一个很棒的想法，但她不希望将单词分割并传送到自己的程序中去。所以，她提议，既然生成一个有 50 000 行的文件，那为什么不直接生成整个 50 014 行程序呢？也就仅仅长了 14 行而已。这样就有了程序生成器 ——program generator 的前身，如示例 1-4 所示。

(2) 例如，通过移去某些数组索引上不需要的边界检测，如常量，可以获得更快更短的代码。

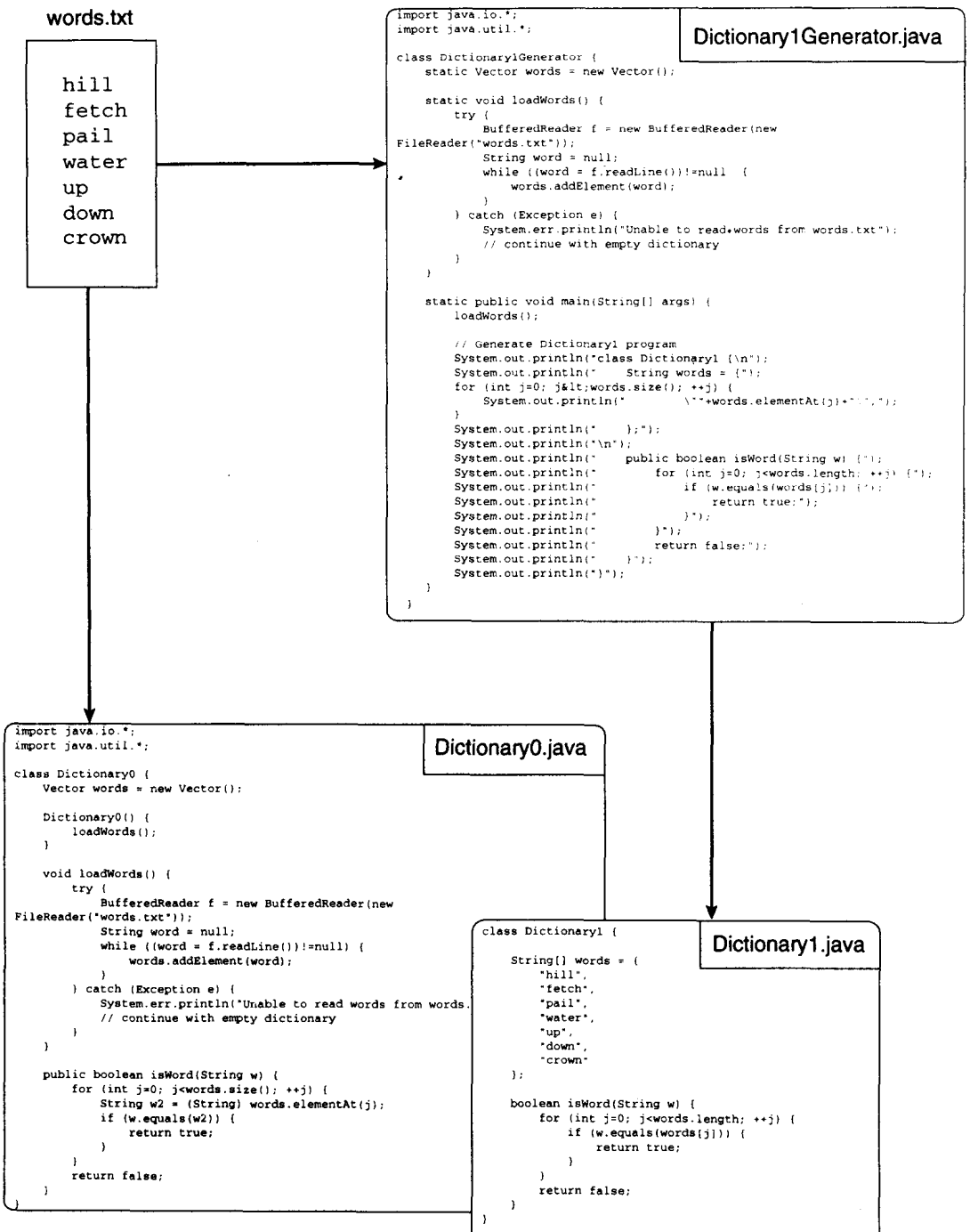

words.txt
hill
fetch
pail
water
up
down
crown
Dictionary1Generator.java
import java.io.*;
import java.util.*;
class Dictionary1Generator {
static Vector words = new Vector();
static void loadWords() {
try {
BufferedReader f = new BufferedReader(new
FileReader("words.txt"));
String word = null;
while ((word = f.readLine())!=null) {
words.addElement(word);
}
} catch (Exception e) {
System.err.println("Unable to read words from words.txt");
// continue with empty dictionary
}
}
static public void main(String[] args) {
loadWords();
// Generate Dictionary1 program
System.out.println("class Dictionary1 {\n");
System.out.println(" String words = {");
for (int j=0; j<words.size(); ++j) {
System.out.println(" \""+words.elementAt(j)+"\",");
}
System.out.println(" };");
System.out.println("\n");
System.out.println(" public boolean isWord(String w) {");
System.out.println(" for (int j=0; j<words.length; ++j) {");
System.out.println(" if (w.equals(words[j])) {");
System.out.println(" return true;");
System.out.println(" }");
System.out.println(" }");
System.out.println(" return false;");
System.out.println(" }");
System.out.println("}");
}
}
Dictionary0.java
import java.io.*;
import java.util.*;
class Dictionary0 {
Vector words = new Vector();
Dictionary0() {
loadWords();
}
void loadWords() {
try {
BufferedReader f = new BufferedReader(new
FileReader("words.txt"));
String word = null;
while ((word = f.readLine())!=null) {
words.addElement(word);
}
} catch (Exception e) {
System.err.println("Unable to read words from words.
// continue with empty dictionary
}
}
public boolean isWord(String w) {
for (int j=0; j<words.size(); ++j) {
String w2 = (String) words.elementAt(j);
if (w.equals(w2)) {
return true;
}
}
return false;
}
}
Dictionary1.java
class Dictionary1 {
String[] words = {
"hill",
"fetch",
"pail",
"water",
"up",
"down",
"crown"
};
boolean isWord(String w) {
for (int j=0; j<words.length; ++j) {
if (w.equals(words[j])) {
return true;
}
}
return false;
}
}

图 1-3　文件内容流程图

示例 1-4 Dictionary1Generator.java

```
import java.io.*;
import java.util.*;

class Dictionary1Generator {
   static Vector words = new Vector();

   static void loadWords() {
      try {
         BufferedReader f =
            new BufferedReader(new FileReader("words.txt"));
         String word = null;
         while ((word = f.readLine())!=null) {
            words.addElement(word);
         }
      } catch (Exception e) {
          System.err.println("Unable to read words from words.txt");
          // continue with empty dictionary
      }
   }

   static public void main(String[] args) {
      loadWords();

      // Generate Dictionary1 program
      System.out.println("class Dictionary1 {\n");
      System.out.println("   String words = {");
      for (int j=0; j<words.size(); ++j) {
          System.out.println("       \""+words.elementAt(j)+"\",");
      }
      System.out.println("   };");
      System.out.println("\n");
      System.out.println("   public boolean isWord(String w) {");
      System.out.println("    for (int j=0;j<words.length;++j){");
      System.out.println("             if (w.equals(words[j])) {");
      System.out.println("                 return true;");
      System.out.println("             )");
      System.out.println("         )");
      System.out.println("         return false;");
```

```
            System.out.println("    }");
            System.out.println("}");
        }
    }
```

为了完全分辨这两种方法的区别，就需要运用约束时间（binding time）这个概念，在此时间，某些信息被程序确定或者绑定在一个程序上。考虑到字典中单词的约束时间，在第一种情况下，单词在运行时读入，这种约束在单词被读入并存储于向量时产生。在第二种情况下，单词在编译之前读入。看起来这也许不是一个大问题，但这意味着，如果要在单词中作改变，则必须重新编译程序。除了传统的两种方法外，程序生成器引入第三种约束时间，如图 1-4 所示。

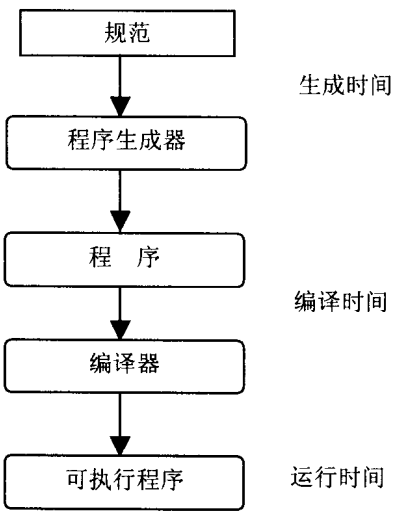


图 1-4　约束时间

- **生成时间**（Generation time）　是指在读入一个程序的规范后生成程序的时间（在此，规范很简单，只是一个单词列表）。
- **编译时间**（Compile time）　是指编译生成的程序的时间。
- **运行时间**（Run time）　是指执行生成的程序的时间。

Jill 再次估量自己的这两个不同程序间的区别。像以前一样，她观察到一个有趣的情形，一些运行时间的计算转移到了其他的时间段。特别是，`loadWords` 方法已经被逐字地迁出运行时间（在 `Dictionary0` 中）而移入到生成时间（在 `Dictionary1Generator` 中）。这样很好，因为这就意味着任务运行速度更快。

Jack 与 Jill 两人都对这个结果感到高兴。Jack 可以继续给 words.txt 文件增加单词。无论什么时候需要，新的程序 Dictionary1.Java 就可以再生成出来。Jill 同样感到满意，因为她确实不想让 Jack 动她的程序。另外，两个人同时修改同一个文本文件实在是自找麻烦。