第5章 多语言开发

Neal Ford，意见领袖[①]

用不了十年，所有程序员都将用Smalltalk编程，不论他们把它叫什么。

——Glenn Vanderburg

时间回到1995年，当时C++程序员们还在为指针、内存管理和其他怪异的技巧而身心疲惫的时候，Java出现了。它减轻了C++程序员的痛苦，从而受到热捧。程序员可以用Java更轻松地完成工作。不过为了让Java能够更成功，Java设计者们需要吸引当时主流程序员——也就是那些C++程序员——的注意。因此，Java语言的设计者有意地让Java看起来非常像C++，这在当时看来是非常合理的。如果让开发者从最基础的东西开始学习一门新的语言，他们是很难接受的。

但是走到2008年，向后兼容性的问题已经不那么重要了。Java开发新手要学很多奇怪的内容，而且这些内容大多与需要解决的问题没有关系，而仅仅是为了满足Java中的一些规矩套路。看看下面这段很多Java开发者第一次碰到的Java代码。

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

想想你要向一个开发新手解释多少事情，才能让他理解这段代码？Java自身充斥着C++的遗风（比如索引基于0的数组），以及在1995年看来合理的设计（比如区分原生数据类型和对象）——这些都成为阻碍当代程序员生产率提高的障碍了。

幸运的是，Java的设计者们在创造Java时做出了一个英明的决定：将语言和平台分离。这样，

① Meme Wrangler，ThoughtWorks公司的一种职位。——编者注

就给了开发者一个逃离Java牢笼的机会，它就是多语言开发（Polyglot Programming）。

5.1 多语言开发

单词*polyglot*是指可以讲多种语言的能力。多语言开发利用了Java中语言和平台分离的特性（C#亦如此）。开发者可以使用特定的语言解决特定的问题。现在，在Java虚拟机和.NET托管运行时上可以运行数以百计的编程语言了。然而作为开发者，我们还没有充分利用这一能力。

当然，开发者们一直都使用着多种编程语言进行开发：大多数应用程序都使用SQL访问数据库，用JavaScript为网页添加交互性，更不用说无处不在的XML配置文件。但是，多语言开发的思路与此并不相同。前面的几个例子都与JVM无关；它们运行在Java世界之外。这让人非常头疼。试想为了解决对象和基于集合论的SQL之间的阻抗不匹配，人们已经花掉了多少亿美元？这种阻抗不匹配会令开发者十分不安，因为在用到多种语言的地方，他们会感到痛苦。但是多语言开发是不同的，它利用的编程语言都会生成运行于JVM的字节码，因此不存在阻抗不匹配的问题。

多语言开发的另一个问题是你必须变换语言。在以前，变换语言通常都意味着变换平台。这对于开发者来说明显是个坏消息，因为他们不希望重写所有的库。但是对于像Java和C#这类与平台分离的语言来说，你不必再为这个问题而困惑了。多语言开发让你能够继续利用所有现有的资产，同时还能够选择更适合完成当前工作的语言。

那么，所谓“更适合完成当前工作的语言”是什么意思呢？下面几节展示了一些应用多语言开发的范例。

5.2 用 Groovy 的方式读取文件

现在有一项任务：编写程序，用来读取文本文件、打印文件的文本内容，并在每一行的前面加上行号。下面是Java代码。

下载 ./code/ford/LineNumbers.java

```
package com.nealford.polyglot.linenumbers;

import java.io.*;
import static java.lang.System.*;

public class LineNumbers {
    public LineNumbers(String path) {
        File file = new File(path);
```

```
        LineNumberReader reader = null;
        try {
            reader = new LineNumberReader(new FileReader(file));
            while (reader.ready()) {
                out.println(reader.getLineNumber() + ":"
                        + reader.readLine());
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException ignored) {
            }
        }
    }

    public static void main(String[] args) {
        new LineNumbers(args[0]);
    }
}
```

下面是能够完成相同功能的Groovy（一种在JVM上运行的脚本语言）代码。

下载 ./code/ford/LineNumbers.groovy

```
def number=0
new File (args[0]).eachLine { line ->
    number++
    println "$number: $line"
}
```

对于简单的任务，Java显得过于臃肿和复杂了。在刚才的例子中，一板一眼的异常处理代码远多于完成功能的代码。Groovy可以为你处理大部分繁琐的工作，让你更容易获得可用的代码。这段代码会编译为Java字节码，最终结果和Java代码的效果是完全相同的。不得不承认一点，Groovy版本的字节码并不是非常高效：Groovy必须对Java的字节码进行很多处理，让它变得更加动态（比如通过代理对象调用Java类）。但是在这个简单的任务中，到底是程序员的生产率更重要，还是执行的效率更重要？如果这个完成读取文件并添加行号的任务，所用的时间是500毫秒或者100毫秒，谁在乎呢？你花在编写代码上节省下来的时间是这个时间的几百万倍。挑选一个更适合工作的工具，远比优化性能重要得多。

5.3 JRuby 和 isBlank

当然，上一节的例子中，那个简单的脚本完成的是一项Java并不擅长的任务。对于在Web应用程序中验证参数不为空这种更普遍的情况呢？

下面的Java代码来自于Apache Commons项目，它满足了很多Java基础代码的常见需求。这段代码可以判断一个字符串是否为空，几乎所有的Web应用程序都要对用户输出参数进行这项判断。于是就有了StringUtils类中的isBlank()方法。

下载 ./code/ford/StringUtils.java

```
public static boolean isBlank(String str) {
    int strLen;
    if (str == null || (strLen = str.length()) == 0) {
        return true;
    }
    for (int i = 0; i < strLen; i++) {
        if ((Character.isWhitespace(str.charAt(i)) == false)) {
            return false;
        }
    }
    return true;
}
```

这段代码暴露出很多Java语言固有的缺陷。首先，由于Java不允许你改变或者扩展String类，因此这个方法是在一个名为StringUtils的类中。这是一个典型的*落单方法*（poorly hung method）——它没有被放置到自己本应属于的那个类中。另一个缺陷是，你必须不断地校验传递给你的对象是否为null。null在Java中是一个特殊类型：它既不是原生数据类型，也不是对象。很多Java代码都需要校验对象是否为null。最后，你必须要遍历字符串，确保所有的字符都是空格。当然，你无法调用每个字符上的方法做去判断（因为字符是原生类型）；而是必须使用Character封装类。

下面是完成相同功能的JRuby代码。

下载 ./code/ford/blankness.rb

```
class String
  def blank?
    empty? || strip.empty?
  end
end
```

下面是证实它能正确工作的测试。

下载 ./code/ford/test_blankness.rb

```
require "test/unit"
require "blankness"

class BlankTest < Test::Unit::TestCase
  def test_blank
    assert "".blank?
    assert " ".blank?
    assert nil.to_s.blank?
    assert ! "x".blank?
  end

end
```

注意上面代码中的几件事情。第一，Ruby允许你直接为String添加方法，从而生成带有属性的方法。第二，由于你不必区分原生类型与对象，所以代码变得非常简单。第三，在测试中，我不必担心nil的情况：在Ruby中，nil也是一个对象（像这个语言中的其他东西一样），因此如果我试图传入一个nil，它的to_s()方法（Ruby版本的toString()方法）会返回一个长度为0的字符串——也就是空字符串。

你无法将这段代码在Java中重新实现，因为在Java世界中，String类是final的。但是，如果你使用基于JRuby的Ruby on Rails，就可以这样操作Java的字符串了。

5.4 Jaskell和函数式编程

目前为止，我们看到的例子大多是关于弥补Java的语言级缺陷的。但是多语言开发还能弥补语言的基础设计决策的缺陷。比如，在Java和C#这种命令式的语言中，编写与线程相关的代码是非常困难的。你必须了解使用synchronized关键字的副作用和微妙之处，以及多线程访问共享数据时会带来哪些不同。

在多语言编程的条件下，你可以使用函数式语言，从而彻底避免这些问题。这样的语言包括Jaskell（Java版本的Haskell）以及Scala（一种为JVM编写的现代的函数式语言）。

*函数式语言*摆脱了很多命令式语言的限制。它们更加严格地遵循一些数学原理，例如函数式语言中的函数会像数学中的函数一样工作：输出完全依赖于输入。换句话说，函数在运行的过程中是不会改变其内部状态的。就好像调用数学函数，比如sin()，你不必担心某次调用会突然返回余弦值，因为sin()的任何内部状态都不会被改变。数学函数中的任何状态都不会在调用的过程中被修改。函数式语言中的函数（和方法）也以相同的方式工作。常见的函数式语言包括Haskell、O'Caml、SML、Scala、F#等。

尤其需要强调的是，函数式语言对于多线程的支持要比命令式语言好很多，因为它们拒绝使用状态。相对于命令式语言，其有利的一面是，使用函数式语言可以更容易地编写强壮的线程安全的代码。

下面进入Jaskell。Jaskell是Haskell语言的一个版本，运行在Java平台上。换句话说，它可以将Haskell代码转换为Java字节码。

下面是一个例子。假如你要使用Java实现一个类，它可以安全地访问数组中的一个元素。这个类的实现大致如下。

下载 ./code/ford/SafeArray.java

```
class SafeArray{
    private final Object[] _arr;
    private final int _begin;
    private final int _len;

    public SafeArray(Object[] arr, int len){
        _arr = arr;
        _begin = begin;
        _len = len;
    }

    public Object at(int i){
        if(i < 0 || i >= _len){
            throw new ArrayIndexOutOfBoundsException(i);
        }
        return _arr[_begin + i];
    }

    public int getLength(){
        return _len;
    }
}
```


你可以使用Jaskell中的*tuple*编写相同的功能。tuple本质上就是一个关联数组（associative array)。

下载 ./code/ford/safearray.jaskell

```
newSafeArray arr begin len = {
  length = len;
  at i = if i < begin || i >= len then
           throw $ ArrayIndexOutOfBoundsException.new[i]
         else
           arr[begin + i];
}
```

由于tuple本质上是关联数组，所以对`newSafeArray.at(3)`的调用就会转发到tuple的`at`部分，从而执行在这部分里定义的代码。尽管Jaskell不是面向对象的，但继承和多态等机制都可以用tuple来模拟，而且一些程序员们向往已久的功能——例如*mixin*——在Jaskell中也可以用tuple来实现，而在Java语言中就做不到。*mixin*允许你在不使用继承的情况下向一个类中注入代码（而不仅仅是增加方法签名），从而提供了一种取代接口与继承机制的可能性。

Haskell（以及Jaskell）的特性之一是函数的惰性求值。在Haskell中，不确定的计算在其真正需要前，是不会执行的。例如，下面的代码在Java中会导致问题，但是在Haskell中是完全合法的。

```
makeList = 1 : makeList
```

这段代码可以理解为“创建一个只包含单一元素‘1’的列表。如果还需要更多的元素，重复执行该操作。”这个函数最终会创建一个由‘1’组成的无限长度的列表。

假如你要实现一个复杂的调度算法，用Java可能需要1000行代码，但是Haskell只需要50行就够了。既然如此，为什么不充分利用Java平台支持多种语言的能力，用其他更适合这项任务的语言来编程呢？就像每个项目都有数据库管理员一样，我们以后还需要越来越多的具有特殊知识背景的人来编写代码，实现特定的功能。

不大可能只用Jaskell去实现一个完整的应用程序。但是，在那些大型的应用程序中，为什么不去好好地利用它的优势呢？比如说，你在开发一个Web应用程序，它需要一段高并发的调度代码。那么，可以只用Jaskell编写调度程序，用JRuby的Rails（或者Groovy on Grails）开发与Web相关的代码，最后利用已有的代码与老旧的大型机通信。由于有Java平台，你可以在字节码级别上把它们粘合到一起，从而提高你的生产率，因为你所用的工具更适合面前的问题。

5.5 Java测试

即使你不愿意改变现在正在做的事情，也同样可以采用多语言开发。对于现有的代码，你仍然有机会采用更适合的语言。最常见的事情之一就是测试复杂的代码。Java并不具备足够的灵活性让一个对象去模仿其他对象，所以创建符合需要的模仿（mock）对象将会非常耗费大量的时间。为什么不用更有效的语言去写测试（只是测试而已）呢？

下面的实例演示了使用JMock（一个流行的Java模仿对象库）测试Order类和Warehose类（事实上是类与接口）的交互。

下载 ./code/ford/OrderInteractionTester.java

```
package com.nealford.conf.jmock.warehouse;

import org.jmock.Mock;
import org.jmock.MockObjectTestCase;

public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new OrderImpl(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");
```

```
        //exercise
        order.fill((Warehouse) warehouseMock.proxy());

        //verify
        warehouseMock.verify();
        assertTrue(order.isFilled());
      }

    }
```

这段代码测试了Order类与Warehouse类（通过其接口）之间的交互，并且验证了相关的方法是否被调用，以及结果是否正确。

下面是利用JRuby（以及Ruby世界里强大的模仿对象库——Mocha）完成的相同的测试。

下载 ./code/ford/order_interaction_test.rb

```
require 'test/unit'
require 'rubygems'
require 'mocha'

require "java"
require "Warehouse.jar"
%w(OrderImpl Order Warehouse WarehouseImpl).each { |f|
   include_class "com.nealford.conf.jmock.warehouse.#{f}"
 }

class OrderInteractionTest < Test::Unit::TestCase
  TALISKER = "Talisker"

  def test_filling_removes_inventory_if_in_stock
    order = OrderImpl.new(TALISKER, 50)
    warehouse = Warehouse.new
    warehouse.stubs(:hasInventory).with(TALISKER, 50).returns(true)
    warehouse.stubs(:remove).with(TALISKER, 50)

    order.fill(warehouse)
    assert order.is_filled
  end

end
```

由于Ruby是动态语言，因此这段代码看上去会更加简单。JRuby会将Java对象封装在代理类中，你能够直接实例化一个接口（本例中为Warehouse）。另外，你只要在测试的开始加上require 'warehouse.jar'，就可以将所有相关的Java类导入在测试路径上了。在Java里，你不希望能这么做么？

多语言开发不一定会给你当前的工作带来天翻地覆的冲击。在大多数公司里，测试代码并不是“官方的”代码，所以甚至有可能不用获取许可就能使用JRuby开始编写测试。

5.6 多语言开发与未来之路

在2007早些时候，ThoughtWorks发布了它的商业产品处女作——Mingle，一个敏捷项目管理工具。当时，对于Mingle来说，能否尽快上市是至关重要的，所以我们决定使用Ruby on Rails进行开发。但是，我们也希望能够重用一些已经存在的库，包括Subversion支持库和一个基于Java的图表库。因此，我们最终选择用Rails on JRuby来实现：它既允许我们使用已有的Java库，又能发挥Ruby快速开发的优势。Mingle充分体现了多语言开发的精神：一方面选择使用针对当前工作的最佳工具，另一方面发挥底层平台的健壮与资源丰富的优势。

将各种解决方案强行塞入单一编程语言的日子即将一去不复返。既然有出色的托管运行时（Java和CLR），我们就应该充分利用这些平台，同时选择更好的工具。利用多语言开发，你既可以混合多种适合不同情形的语言，又不必丢弃已经存在的代码——毕竟它们还是非常重要的。在Java和.NET这两个已获得充分验证的平台上，新语言的开发正在迅猛发展。作为一名开发者，你需要学习如何利用这一发展的优势，这样才能使用更适合工作的工具编写更棒的代码。