
Table of Contents

Introduction	1.1
第一章 从数据类型开始	1.2
第二章 值类型	1.3
第三章 引用类型	1.4
第四章 从编程范式开始	1.5
第五章 面向对象语言	1.6
第六章 函数式语言	1.7

写在前面

写这本书的初衷是因为曾经跟一个朋友调侃说：“我写一本书送你好。”

那之后我就开始构思一本既适合初入编程的新手，又能为有一定编程经验的朋友提供帮助的书。于是我根据读书工作过程中陆陆续续接触过的编程语言，汇总整理出它们的相似与区别，以数据类型和编程范式这两方面作为学习编程语言的切入点，分章节进行说明介绍。相信读者对这两方面有了一定的了解后，再学习编程，或者接触新的编程语言也就有了更加明确的学习方向。

从数据类型聊编程语言的划分

通常说的程序由数据和算法两部分组成，因此从数据类型开始说明编程语言是再好不过的切入点，这一章我讲主要对数据类型几个相对的特性进行说明，这几个相对的特性也直接的影响着编程语言本身的特性。所以第一章也可以看做是以数据类型作为切入点说明编程语言间的对立与区分。

1. 第一节将引入数据类型的概念。
2. 第二节在第一节的基础上，针对数据类型转换机制不同引入数据类型强弱的概念。
3. 第三节在第一节的基础上，针对类型检查机制的不同引入数据类型动静的概念。
4. 第四节和数据类型关系不大，这里说明了编译类型语言和解释类型语言的区别，主要为了对比区分第二节、第三节中涉及到的这三组概念。
5. 第五节回到数据类型相关概念，对常见的两大类数据类型：值类型和引用类型进行概述说明。

1.1 类型

数据类型

在谈语言设计上针对数据类型采取的不同策略之前，我们首先简单说明下什么是数据类型。

数据类型可以看作是内存对采用统一存储方式的数据而定义的一个统一称谓，简单的说就是数据是如何在内存中存储的。

比如在Java中用4个字节来存储的数字就是整型的，如果同一个数字用8个字节来存储，那么它就是长整型的。而在JavaScript中存储这个数字必然要花费掉8个字节，因为JavaScript中数字类型就是8字节的。

更简单的说，数据类型是用来区分不同种类的数据而定义的统一称谓。

比如Java中整型（int）数据的数值范围要比长整型小；整型存储整数，而浮点型存储小数。它们都代表不同的数据。在JavaScript中数字类型就是Number，没有对小数和整数进行区分，所以这个时候所有的数字被统一看作为一种数据。

另外一种区分数据类型的的说法是，数据类型是用来划分数据间是否可以直接进行运算的。

比如数字和数字是可以直接求和运算的，因为数字1和数字2都被划分为数字类型，而数字和字符则不能直接相加。

常见数据类型

常用的数据类型有数字类型、字符类型、布尔类型、集合类型、散列类型、函数类型、自定义类型等。其中不同的语言又有所不同。比如Java把数字类型又划分为Long Number、Short Number、 Decimal Number；C用Char来表示字符类型，后续的大多数语言则提供了String。

有些语言也会定义一些不常见的类型，比如集合类型下的Range，用来表示一个范围的数值，Range通常作为循环迭代子来使用。

总之不同的语言会对数据类型进行不同精度的划分，不同的语言也可能使用不同的别名，但是数据的用途都是相同的。

好了，有了数据类型点概念就可以说明强弱类型和动静类型了。

1.2 强类型和弱类型

这里的强与弱是对类型检查严格程度的描述，强类型语言中对类型转换的要求相对与弱类型语言来说要更加严格。

我想通过对比几种语言的类型转换来说明强弱类型的区别，而不同数据类型之间的运算最能体现不同语言对类型转换的不同处理方式。

下面的例子中会用到Java、JavaScript、C++和Ruby，它们分别是强类型、弱类型、弱类型、强类型。

类型转换

在强类型语言中，不同类型的变量是无法直接运算的，而弱类型语言在处理这种运算时不会抛出异常，来看下面几个例子。

在Java中int类型不能和String类型直接运算，在加法运算中int类型会被隐式转换为String类型，然后执行String + String的字符串合并运算。

而在减法运算中，因为字符串不存在减法运算，所以int类型不会隐式转换为String类型进行String - String运算。而Java中String类型无法隐式转换为int类型，所以int与String的减法运算在编译时会报错。

```
//Java 隐式转换下整型与字符型加减运算

int number = 1;
String string = "this is a string";
String stringNumber = "1";

System.out.println(number + string);           //1this is a string
System.out.println(number + stringNumber); //11

System.out.println(number - string);
//error: bad operand types for binary operator '-'
```

接下来我们看下Java显示转换下会如何处理上述运算。

```
//Java 显示转换下整型与字符型加减运算

int number = 1;
String string = "this is a string";
String stringNumber = "1";

System.out.println(number + Integer.parseInt(stringNumber)); //2
System.out.println(number - Integer.parseInt(stringNumber)); //0

System.out.println(number - Integer.parseInt(string));
//Exception in thread "main" Java.lang.NumberFormatException: For input string: "t
his is a string"
```

可以看出当我们显示的把一个内容为数字的String转换为int时，会按照两个int类型的运算规则进行加减运算。

而当我们显示的把一个内容为非数字的String转换为int时会抛出一个运行时异常：字符串无法格式化为数字。

同样在Ruby中对String和Number进行加法运算也会报运行时错误。

```
#Ruby 整型与字符型加减运算

1 + "1"                      #String can't be coerced into Fixnum
1 + "this is a string" #String can't be coerced into Fixnum
```

接下来我们看下弱类型的C++和JavaScript是如何处理两种不同数据类型的运算的。

```
//C++ 整型与字符型加减运算

#include <iostream>
using namespace std;

int main()
{
    int number = 1;
    char character1 = '1';
    char characterA = 'A';

    cout << number + character1 << endl;      //50
    cout << number + characterA << endl;      //66
    cout << number - character1 << endl;      //-48

    return 0;
}
```

可以看到无论char是字母还是数字在C++中都会转换为表示这个字符的ASCII码（数字表示），然后与int进行运算。

接下来看下JavaScript中是如何处理的。

```
//JavaScript 整型与字符型加减运算

var number = 1
,   string = "this is a string"
,   stringNumber = "1";

console.log(number + string);          //1this is a string
console.log(number + stringNumber);    //11
console.log(number - string);         //NaN
console.log(number - stringNumber);    //0
```

可以看出JavaScript的运算结果是在Java的基础上进行了变化，number + string 依旧是做字符串合并运算，而 number - string 则变成了数字减法运算，而对无法转换为数字的字符串进行数学运算也不会返回错误，取而代之的是返回NaN（Not a Number，是JavaScript内置的一个特殊值）。

值得一提的是，在大多数语言中除零运算都会报错，比如Java和Ruby中会报运行错误，而C++中会报编译错误，但是在JavaScript中除零会返回另外一个特殊值 Infinity（表示无限大的值）。

也许通过上面的几个例子，你会总结出：可以用 int - string 来判断一个语言是强类型还是弱类型，确实这个看上去很取巧，但是事实总在意料之外，来看下面这段Clojure代码。

```
;Clojure 整型与字符串加减运算

;Clojure是前置运算符, (+ 1 "1")即1 + "1"

(+ 1 "1") ;"11"
(- 1 "1") ;0
```

Clojure是强类型语言，但是可以看出其 number + string 返回的是string，而 number - string 返回的是int，所以判断一门语言是强类型还是弱类型，最好还是看看官方定位。

究竟一个语言是强类型还是弱类型还是取决于语言设计者所做出的偏好，弱类型语言可以让你的代码更简单明了，逻辑更加清晰易懂。但为此付出的代价就是你可能会因为一时的疏忽而导致一些难以发现的问题（trap）。

通常我在用JavaScript处理字符串与数字转换时我完全可以避免语言繁琐的API，而只用 + 或 - 操作符。

```
//JavaScript 弱类型语言代码更加简洁

"1" - 0; //将字符串转换为数字
1 + "0"; //将数字转换为字符串
```

如我之前所说的，这样的隐式转换会隐藏着一些小的陷阱。

```
"1" == 1; //true
```

所以在一些if判断时你不得不用这样的操作符。

```
"1" === 1; //false
```

总之强弱类型决定着你是用简短的方式编码，还是扎实的处理变量类型避免一些不必要的问题。不过，既然你选择了要使用的语言，或者你被迫要使用这样的语言，那么你只能遵从语言设计时所采取的折中，你在只能知道这样做有什么好处又什么坏处，而不是能决定不这么做。

好，强/弱类型就到此为止吧，接下来说一说动/静类型，猜一猜动/静类型将会在哪一方面做出折中，而这种折中又会影响到语言在哪方面的表现呢。

1.3 静态类型和动态类型

这里的动与静是针对变量在声明（declare）时是否绑定了指定的数据类型而说的，通常静态语言在声明时要指定一个具体的数据类型，而动态语言则不需要。

静态类型在编译时确定具体数据类型，而动态类型是在运行时确定具体数据类型。

经常有人把动态类型和弱类型混在一起，他们觉得用 var 去声明一个变量不去指定具体的类型，那么这样的语言就是动态的，当然不能说这样的解释是完全错误的，但是确实是不准确的。

和对比强弱类型类似，我想通过变量声明和参数传递来说明动静类型的区别。

变量声明

静态类型语言在变量声明时是要指定数据类型的，而动态类型语言中变量在声明时是无法确定具体类型，只有在使用时才能明确。

```
int intVariable; //这里很明显这个变量是个数字。
```

```
var unknowVariable; //这里通过变量声明无法判断其具体类型。
```

在动态类型语言中你甚至可以不声明而直接使用一个变量。

```
//JavaScript 动态语言声明变量无须显式指定类型  
  
undeclaredVariable = "this variable is not declared";  
//undeclaredVariable是一个未声明的变量，在JavaScript中这个变量会变为Global对象的一个属性
```

也许你想说我在用Java范型时也无法得知变量的具体类型，但不得不承认你所用的范型是你自己定义的一个类，实质上类本身就是数据类型。而在JavaScript中我们无法判断unknowVariable是一个值还是一个对象。

除此之外你可能还会拿出Scala说它的val和var也是无法区分类型的，但在Scala的代码中你随处都可以看见 a: Int、b: Double 这样的变量声明，因为Scala确实是一个静态类型语言。而 val 和 var 只是Scala引入的类型推断机制，这一点可以在Scala用val或者var声明变量必须初始化看出来。

```
//Scala 在使用val声明变量时必须初始化，否则无法推断变量类型  
  
val uninitializedVariable;  
//error: only classes can have declared but undefined members
```

参数传递

因为动态类型语言对变量与数据类型绑定要求宽松，导致在函数传参时相对与静态类型语言也要灵活的多。

```
//Java 函数参数类型检查更加严格

public class StaticTyped{
    public void mustBeInt(int params){
        System.out.println("params is permitted");
    }
    public void mustBeString(String params){
        System.out.println("params is permitted");
    }
    public static void main(String []args){
        StaticTyped StaticTyped = new StaticTyped();
        String stringParams = "this params is a string";

        StaticTyped.mustBeInt(stringParams);
        //error: method mustBeInt in class StaticTyped cannot be applied to given
        types:
        StaticTyped.mustBeString(stringParams);
        //params is permitted
    }
}
```

```
//JavaScript 函数参数类型检查宽松

function paramsIsDynamicTyped(params){
    if(typeof params === "number") console.log("params is number");
    if(typeof params === "string") console.log("params is string");
}

var number = 1
,   string = "this is a string";

paramsIsDynamicTyped(number);      //params is number
paramsIsDynamicTyped(string);     //params is string
```

动态类型提供了更加灵活的编程方式，你不必因为函数参数类型而困扰，你也可以在需要使用一个变量时省去变量声明。不过同介绍强弱类型时所描述的一样，你在获得更高的灵活性的同时，也要付出更多的代价保证代码的准确，因为一个很常见的拼写错误在动态类型下是无法察觉的，而静态类型语言却会给出一个明确的编译错误。正因为这样的特性，静态类型语言往往更容易提供一个友好的IDE。

在JavaScript一个拼写错误的变量初始化会变为全局对象的一个属性。

```
//JavaScript 弱类型语言难以对错误拼写给出警告

var zero;
zaro = 0;
console.log(window.zaro, zero);      //0 undefined
```

而在Java中因为变量必须声明才能使用，所以会抛出一个编译错误。

```
//JavaScript 强类型语言变量必须声明并绑定具体类型，所以可以提供拼写错误警告

int zero;
zero = 0;      //error: cannot find symbol
```

其实强弱类型和动静类型一样，都是在灵活性和安全性这个数轴上寻找一个折中点。动态类型使语言更加灵活，但更容易出错。与此相对的，强类型使语言更加安全，但代码累赘。虽然两者都是在灵活性与安全性上做出折中，但两者是不同的概念，也就是说静态类型语言可能是强类型的，也可能是弱类型的。

下表就是一个好的例子：

	静态类型	动态类型
强类型	Java	Ruby
弱类型	C++	JavaScript

最后补充一点，所谓的强弱并不是绝对的概念，比如C++可能是有一点弱类型的，所谓的“有一点”只是在强弱类型的数轴上更加偏向弱类型而已。

1.4 编译型与解释型

聊过了强弱类型和动静类型后，剩下一个容易混进来的概念就是编译型和解释型了。有些人会把编译类型和静态类型混淆，就像把弱类型和动态类型混淆一样，好了，在开始编译型与解释型的介绍之前你最好先问问自己弱类型和动态类型分别是怎么确定的？

编译型和解释型并不是语言设计上的概念，而是语言实现上的特性。所以我们说某个语言只能是编译型或者解释型是不准确的。因为任何语言都是可以是编译型的或者解释型的，这取决于你是如何实现这个语言的，而不是语言就是如此设计的。

编译型

那么什么算作是编译型呢，将程序转换为特定的机器能运行的机器码，特定的机器可以直接运行编译后的机器码，这样运行的语言就是编译型的。

编译型的好处是源码是私密的，相对要安全些（因为你只需要提供机器码就可以运行程序了）。而且相比解释型语言其速度要更快。

但是弊端是无法跨平台，也就是机器码会受到操作系统的限制，并不是任何系统都可以正确运行的。另外它难以调试的（指针对于用于运行的机器码），因为你不知道它的源码是如何实现的。

C、C++、Object-C都是编译型实现的语言。

解释型

解释型语言不是由机器直接运行的，解释型语言会通过一个中间程序：解释器，来执行代码。解释器会将源代码一行一行的转换为当前机器能够执行的代码然后运行。

因此解释型语言的源码是公开的，而且因为逐行转换的缘故其速度相比编译型要慢。

但是因为解释器缘故它可以是跨平台的（只要解释型可以在任何操作系统运行），同时公开的源码和逐行解释也让其容易测试和调试。

PHP、JavaScript是解释型语言。

混合型

像在动静类型和强弱类型里所描述的一样，语言并非是完完全全按照这样或者那样的规格来的。也就是说并非所有语言都必须是完完全全的编译型或者解释型。

有一些语言会将源码编译为中间码（字节码），然后目标机器再通过解释器解释中间码来运行。这样做就中和了两种实现类型的优缺点。

这也是为什么有人说Java是编译类型，有人却说它是解释型的。其实这样的语言只是折中了这两种方案罢了。所以一定要给它定义个实现类型，那就叫做混合型好了。

Java、Python、Ruby都是混合型的。

基础类型和引用类型

抛开在1.1中列举的数据类型分类方式，这里我想从另外一种分类方式来说明：基础类型（有些语言喜欢称作值类型，比如C#）和引用类型。

基础类型与引用类型对比

基础类型

1. 基础类型存储的是直接值
2. 基础类型的值存储在栈中
3. 基础类型是不可修改的
4. 基础类型进行比较时，直接值相等就相等
5. 基础类型在作为参数传递时，传递的是值
6. 一般数字类型、字符串类型、布尔类型都是基础类型

引用类型

1. 引用类型存储的是地址
2. 引用类型的值存储在堆中
3. 引用类型是可修改的（可变的）
4. 引用类型进行比较时，值相等，且地址相等才相等
5. 引用类型作为参数传递时，传递的是引用
6. 一般数组类型、对象类型、函数类型都是引用类型

接下来对上面的五点分别举例说明。

值的存储

基础类型在栈中存储直接值，而引用类型在栈中存储引用的地址，在堆中存储实际值。

对于编译型语言

而言栈是在编译

时处理内存的分

配的，在运行时

处理堆的内存分

配。所以栈的内

存分配是静态

的，堆的内存分

配是动态的；栈

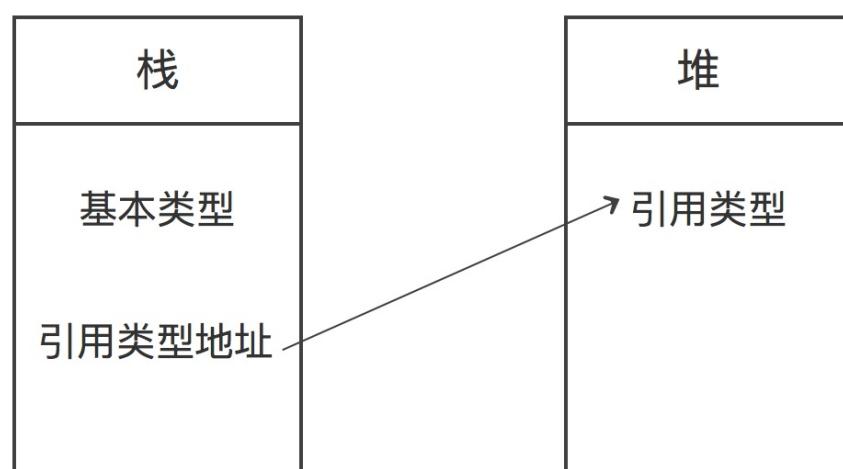
中存储的数据是

固定大小的，而

堆中的数据是可

变大小的。因为

编译型语言在程序编译时处理了基础类型的内存分配，所以相比解释型语言在运行时的处理速度效率自然要高。



对于基础类型和引用类型的存储情况可以参考右图。

是否可变

基础类型是不可变的，这里的不可变不是指变量的值不可变，而是存储在栈中指定地址的值是不允许改变的。我们通过Ruby代码来说明值不可变的意思（因为获取一个值的栈地址对于现在的编程语言来说并没有实际作用，所以Java一类的语言本身并没有提供类似的功能，多亏Ruby是支持的）。

```
#Ruby 获取基础类型栈地址

a = 5
a.object_id      #11

a = 3
a.object_id      #7

b = 5
b.object_id      #11
```

这里我们先对变量a赋值为5，然后输出5对应在栈中的地址11（转换为二进制是0b1011），然后我们修改a的值，再次输出a的地址，会发现地址改变了，这说明a的值改变了，是因为a的地址变了。然后我们对b赋值为5并输出地址，会发现其地址也是11，这说明地址为11的内存中存储的值就是5，是不会改变的，变量值的改变是因为指向的地址发生了改变。

引用类型是可变的，根据上述描述，可以猜测到所谓可变是指存储引用类型的地址所存储的值是可以改变的。

数组是引用类型的，我们修改数组中元素的值，然后输出数组的地址信息进行对比。

```
Ruby 获取引用类型栈地址

a = [1, 2, 3]
a.object_id      #70324083600480
a[0].object_id   #3

a[0] = 0
a.object_id      #70324083600480
a[0].object_id   #1
```

可以发现数组的值改变了，但是其地址信息没有改变，说明引用类型的值是可变的。当输出`a[0].object_id`发现这个地址改变了，因为`a[0]`是个数字类型，数字类型是基本类型，他的值是不可变的，所以`a[0]`指向了改变后的值的地址。但是数组`a`的地址不会因为元素值的修改而修改，除非你对`a`重新赋值（Java是不允许这样做的，Java只允许数组初始化时整体赋值或者循环每一个元素分别赋值）。

```
#Ruby 重新赋值引用类型，引用地址会改变

a = [1, 2, 3]
a.object_id      #70324083600480

a = [0, 1, 2]
a.object_id      #70324083654580
```

相等比较

判断基础类型是否相等，只要值相等就是相等的。而引用类型相等的条件是保证引用的对象是同一个，也就是说栈中存储的地址是指向堆中同一个位置的。

```
//JavaScript 基础类型、引用类型比较

a = 1;
b = 1;
a === b;      //true

a = [1, 2, 3];
b = [1, 2, 3];
a === b;      //false
```

```
//Java 基础类型、引用类型比较

int a = 1;
int b = 1;
System.out.print(a == b);      //true

int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };
System.out.print(a == b);      //false
```

```
#Ruby 基础类型、引用类型比较
```

```
a = 1
b = 1
a.equal? b      #true

a = [1, 2, 3]
b = [1, 2, 3]
a.equal? b      #false
```

这里单独说明下ruby用 `==` 判断两个对象的值是否相等，用 `equal?` 判断是否为同一引用。

```
#Ruby ==和equal?的区别
```

```
a = {:type=>"Object"}
b = {:type=>"Object"}
c = a

a == b      #true
a == c      #true
a.equal? b  #false
a.equal? c  #true
```

参数传递

基础类型传递的是值，而引用类型传递的是引用。基础类型发生修改是无副作用的（不会修改原变量），引用类型修改是副作用的。

```
//JavaScript 基础类型、引用类型参数传递
```

```
var a = 1;
var o = { b: 1 };
var fa = function(a){ a = 2; }
var fo = function(o){ o.b = 2; }

f(a);
f(o);

console.log(a);      //1
console.log(o.b);    //2
```

```
//Java 基础类型、引用类型参数传递

class Demo{
    public static void mi(int a){ a = 2; }
    public static void mr(int[] a){ a[0] = 2; }
    public static void main(String args[]){
        int a = 1;
        int[] aa = { 1, 2, 3 };
        Demo.mi(a);
        Demo.mr(aa);

        System.out.print(a);          //1
        System.out.print(aa[0]);      //2
    }
}
```

对于变量拷贝也是如此，基础类型拷贝的是值，修改是无副作用的；引用类型拷贝的是引用，修改是有副作用的。

```
//JavaScript 基础类型、引用类型拷贝

var a = 1
,   b = a
,   oa = { a: 1 }
,   ob = oa;
b = 2;
ob.a = 2;

console.log(a);          //1
console.log(oa.a);      //2
```

```
//Java 基础类型、引用类型参数传递

int a = 1, b = a;
int aa[] = { 1, 2, 3 }, ab[] = aa;
b = 2;
ab[0] = 2;

System.out.print(a);      //1
System.out.print(aa[0]);  //2
```

基础类型和栈内存分配

字符串类型是基础类型，所以在做字符串修改、合并、截取等操作时，字符串的每次改变都会在栈中分配一块新的内存记录新的值。

```
#Ruby 字符串修改会从栈分配新内存

a = [ "A", "B", "C"]
s = "Array a contains: "
a.each do |i|
  s += i
  puts "address of s is #{s.object_id}"
end
puts s
# output
# address of s is 70123772400880
# address of s is 70123772400780
# address of s is 70123772400700
# Array a contains: ABC
```

可以看出，如果循环的集合对象中存在大量的元素，这时对栈内存而言是一种极大的开销，对于这种需要多次修改的字符串，Java提供了StringBuffer类，Ruby提供了StringIO类，我们仍旧以Ruby为例来演示StringIO的效果。

```
#Ruby StringIO类降低字符串修改的内存开销
require 'stringio'
a = ["a", "b", "c"]
s = StringIO.new
s << "Array a contains: "
a.each do |i|
  s << i
  puts "address of s is #{s.object_id}"
end
puts s.string
# output
# address of s is 70280643727180
# address of s is 70280643727180
# address of s is 70280643727180
# Array a contains: abc
```

可以看出这时s指向的是同一块地址，这样就可以避免频繁修改同一字符串造成栈内存大量开销的问题了。

引用类型深度拷贝

有时候我们在使用一个引用类型时希望操作后能保证原引用是不可破坏的（值不会被修改），这个时候我们就要用到拷贝，拷贝分为浅拷贝和深度拷贝，简单的说浅拷贝是只拷贝引用类型的第一层（假设一个对象的一个属性b也是对象，那么浅拷贝的对象产生修改时是无法保证b对象的属性不被破坏的），而深度拷贝会逐层拷贝这个引用类型的所有引用类型的属性。

可惜的是无论是Java还是JavaScript都没有提供深度拷贝的API，所有只能自己编写或者使用第三方库来实现深度拷贝。

当然最常用的实现思路是遍历这个对象的每一个属性，如果是引用类型则继续遍历这个引用，如果是基础类型就拷贝。另外一种相对简单但不稳定的思路是将对象序列化为JSON串，然后拷贝这个字符串，再反序列化为对象。我们采用第二个思路来看如下例子。

```
//JavaScript 序列化方式实现深度拷贝

function deepClone(obj) {
    return JSON.parse(JSON.stringify(obj));
}

var o = {
    a: { b: 1, c: 2 },
    d: 3
}
var b = deepClone(o);

console.log(b);
//{ a: { b: 1, c: 2 }, d: 3 }
```

第二章 数据类型中的值类型

继第一章第五节提到的值类型概念，这里将详细说明主流编程语言中常见的值类型，以及与这些数据类型相关的概念。

1. 介绍布尔类型、以布尔类型为主的类型转换问题。
2. 介绍数字类型、数字在内存中如何存储。
3. 介绍字符串类型、常用的字符编码。
4. 最后介绍不常见的原子类型。

布尔类型

布尔类型用于逻辑真假判断，表示真时值为true，表示假时值为false。

```
//Java 布尔类型
if(true)      System.out.println("true");    //true
```

布尔类型最基础的用法是逻辑判断，但不同语言类型转换处理方式的不同，使布尔类型在使用上有些区别，我想通过类型转换来进一步说明在不同语言下布尔类型使用上的一些差异。

类型转换

隐式转换

弱类型语言中可以用数字类型代替逻辑判断。

```
//Javascript 布尔类型使用
if(1)      console.log("true");    //true
```

其原理是弱类型语言会将数字类型隐式转换为布尔值。

我准备用JavaScript为例来说明boolean类型和其他类型的隐式转换。

```
//JavaScript 布尔类型隐式转换
```

```
1 == true;           //true, [1]
0 == false;          //true, [1]
1 === true;          //false, [2]
'' == true;          //false, [3]
'' == false;          //true, [3]
's' == true;          //fasle, [4]
's' == false;          //false, [4]
!'' == true;          //true, [5]
!'s' == true;          //false, [5]
!!'s' == true;          //true, [5]
Boolean('s') == true; //true, [6]
Boolean('') == false; //true, [6]
undefined == false; //false, [7]
!undefined == true; //true, [7]
null == false; //false, [7]
!null == true; //true, [7]
```

1. 在JavaScript中boolean会转化为数字，其中true会转换为1，false会转换为0，所以相等。
2. 而强制等于 === 会禁止类型转换，所以这里1不等于true
3. 空字符会转换为true。
4. 非空字符和布尔类型始终不等。
5. 通过4可知是非空字符本身即不是true也不是false，但是取反后是false，两次取反自然就是true。
6. 通过Boolean强制转换非空字符就是true,空字符是false。
7. 这里undefined和null类型与布尔类型比较都返回false，但是取反后表示真，所以有些教程会说undefined、nill和NaN都表示false，这个说法不是很准确，但是在if表达式中确实又是如此应用的。

```
//JavaScript 布尔类型隐式转换
```

```
if(undefined)  console.log("will not execute"); //undefined返回为false, 不执行后续操作
if(!null)      console.log("will execute"); //will execute
```

和JavaScript中0会转换为false不同，并非所有弱类型语言都会把0当做false看待。

```
#Ruby 数字0转换为布尔类型true
if(0) puts "zero equals true" end #zero equals true
```

可以看出在ruby中0等价于true，所以在使用0, 1做条件判断时最稳妥办法还是用==运算返回一个布尔值。

```
//Java 数字类型与布尔类型转化

int a = 1;
if(a == 1) System.out.println("equals true"); //equals true
```

```
#Ruby 数字类型与布尔类型转化

a = 1
puts "equals true" if a != 0 #equals true
```

显式转换

一般强类型语言在处理布尔类型与其他类型间转换时，不像弱类型语言那么随意，通常需要显式转换。

```
//Java

boolean a = true == 1;
//incomparable types: boolean and int [1]

String a = "True";
String b = "another";
boolean c = true == Boolean.parseBoolean(a);
//true [2]

boolean d = true == Boolean.parseBoolean(b);
//false [2]
```

1. Java中不允许数字类型与布尔类型间转换。
2. Boolean.parseBoolean会将字符串true（不区分大小写）转换为布尔值true，而其他字符串都会转换为false。

另外Java可以对一个布尔类型实例化为null，但是null本身是不能和boolean值进行比较的。

```
Boolean a = true == null; //incomparable types: boolean and <null>

Boolean a = null
if(a == null) System.out.println("true"); //true
```

除了布尔类型本身条件判断外，布尔类型间的与或运算也有着特殊的用途。

与或运算

与运算 (&&)： [表达式A]&&[表达式B]，其中表达式A和表达式B必须同时为真才返回true，否则都返回false。

或运算 (||)： [表达式A]||[表达式B]，其中表达式A和表达式B必须同时为假才返回false，否则都返回true。

基于如上运算规则，只要与运算中表达式A为假则返回值就必然为false，所以表达式B本身就不需要计算了。同理在或运算中只要表达式A为真则表达式B也不需要计算了。

```
//Java 与运算中如果表达式A为假，则表达式B不执行

boolean a = false;
boolean b = true;
boolean c = a && (b = !b);

System.out.println(b);      //true
```

```
//Java 或运算中如果表达式A为真，则表达式B不执行

boolean a = true;
boolean b = false;
boolean c = a || (b = !b);

System.out.println(b);      //false
```

在动态类型语言中配合与或运算的特性，可以简化一些赋值语句。

```
//JavaScript 通过逻辑判断进行赋值

var a, b = "";
if(b == "") a = "b should not empty";
else        a = b;

console.log(a);      //b should not empty
```

上述代码中如果b不为空则把a赋值为b，如果b为空则a赋值为"b should not empty"。使用或运算简化if语句为：

```
//JavaScript 动态类型语言通过与或运算简化赋值操作

var b = ""
, a = b || "b should not empty";

console.log(a); //b should not empty
```

与JavaScript类似，Ruby更是提供了`||=`运算符。

```
#Ruby ||=运算符使用
a ||= "default value"
```

`||=`和`+=`类似，即：`a = a || "default value"`，因为`a`本身为空，所以表达式A返回`false`，继续执行表达式B对`a`赋值：`default value`。于是在Ruby中的`||=`就变成了：“如果`a`没有被赋值过，则赋值为...”这样的命令，其实`||=`本质还是或运算特性的使用，只是Ruby中固定了一个运算符来表述。

数字类型

关于数字类型我想先简单讲下其在内存中的存储，然后通过Java和JavaScript中数字类型的划分来进一步说明。

正整数

首先我们用二进制表示一个数字，比如123。

$$123 = 1 + 2 + 8 + 16 + 32 + 64$$

$$123 = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6$$

换作二进制表示为：1111011，低位从右向左排序，即右边第一个数字表示 2^0 。

我们以Java的short类型为例，short类型采用16位存储一个数字，目前存储数字123需要使用7位空间，还剩下9位，剩余的高位(左侧)用0来填充，因此最终表示为 00000000 01111011。同理如果你用int类型来存储的话就是一长串的0： 00000000 00000000 00000000 01111011，因为int类型用32位来存储一个数字。

负整数与符号位

如果你还记得初中数学课本里讲的数轴就能猜到如何表示负数，没错，因为正数和负数是对称的，所以我们只需要在正数的基础上加上一个符号就可以了，这个符号就是符号位，符号位在最高位，这样一个16位的short类型能表示的正整数就只能折半了，即 2^{15} 个，所以short类型正整数能表示的最大值就是 $2^{15}-1 = 32767$ 。

方便理解我们用3位来说明：

正整数	二进制	负整数	二进制
1	001	-1	101
2	010	-2	110
3	011	-3	111
0	000	-0	100

可以看出3位能表达的正整数为 $2^{3-1}-1=3$ 个数。

负整数与补码

只说数据类型不说类型间的运算是无意义的。接下来我们看下正整数的加减运算如何处理。

这里以4+7为例来说明加法运算：

$$\begin{array}{r}
 +4: 0|0100 \\
 (+) \\
 +7: 0|0111 \\
 \hline
 = 0|1011
 \end{array}$$

+11: +| 8+0+2+1

以12-9为例来说明减法运算：

$$\begin{array}{r}
 +12: 0|1100 \\
 (-) \\
 +9: 0|1001 \\
 \hline
 = 0|0011
 \end{array}$$

+3: 0| 0+0+2+1

可以看出正整数加减法和我们想要的结果是一致的。

接下来看下负整数加减法，这里说明下负整数与负整数加减运算不运算符号位。

以(-8)+(-6)为例说明加法

$$\begin{array}{r} -8: \quad 1| \quad 1 \ 0 \ 0 \ 0 \\ (+) \\ -6: \quad 1| \quad 0 \ 1 \ 1 \ 0 \\ \hline = \quad \quad 1| \quad 1 \ 1 \ 1 \ 0 \\ -14: \quad -| \quad 8+4+2+0 \end{array}$$

以(-6)-(-5)为例

$$\begin{array}{r} -6: \quad 1| \quad 0 \ 1 \ 1 \ 0 \\ (-) \\ -5: \quad 1| \quad 0 \ 1 \ 0 \ 1 \\ \hline = \quad \quad 1| \quad 0 \ 0 \ 0 \ 1 \\ -1: \quad -| \quad 0+0+0+1 \end{array}$$

好极了，到目前为止都是理想中的结果，那么接下来看个不如人意的例子。

$$\begin{array}{r} -3: \quad 1| \quad 0 \ 0 \ 1 \ 1 \\ (+) \\ +3: \quad 0| \quad 0 \ 0 \ 1 \ 1 \\ \hline = \quad \quad 1| \quad 0 \ 1 \ 1 \ 0 \\ -6: \quad -| \quad 0+4+2+0 \end{array}$$

可以看出 $-3+3=-6$ 这是不正确的运算结果，我们期待的结果应该是 $-3+3=0$ 。这里为了得到正确的结果，我们先反向构造出一个和为0的情况：

$$\begin{array}{r} -3: \quad 1| \quad 0 \ 0 \ 1 \ 1 \\ (+) \\ +3: \quad 0| \quad 1 \ 1 \ 0 \ 1 \\ \hline = \quad \quad 1| \quad 0 \ 0 \ 0 \ 0 \\ 0: \quad -| \quad 0+0+0+0 \end{array}$$

可以看出 0011 按位取反在加 $1 * 2^0 = 1$ 即为 1101 ，这里按位取反再加一后得到的 1101 就是 0011 的补码。我们可以通过补码的定义修补数学意义上相反数和为零的漏洞。

接下来我们用补码来表示负数

正整数	二进制	负整数	二进制
1	001	-1	111
2	010	-2	110
3	011	-3	101
0	000	-0	100

因为000已经可以表示0了所以为了避免资源的浪费我们可以用100表示-4，这样一来一个有符号的、16位的、补码表示的类型就可以表示 2^{15} - 1个正整数和 2^{15} 个负整数了，范围就是 $[-32768 - 32767]$ 。

小数和精度

二进制与十进制的小数转换同整数一样。我们用110.11来说明二进制小数到十进制的转换。

$$110.11 = 1 * 2^2 + 1 * 2^1 + 0 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2} = 6.75$$

十进制小数转换为二进制，规则为

1. 将小数位乘以二
2. 取结果的整数位，作为高位
3. 取结果的小数位重复上述操作，如果结果小数位为0则结束运算

我们将0.725转换为二进制。

1. $0.125 * 2 = 0.25$, 取整为0, 取0.25循环
2. $0.25 * 2 = 0.5$, 取整为0, 取0.5循环
3. $0.5 * 2 = 1.0$, 取整为1, 取0结束
4. 按顺序拼接取整后数字到高位: 001

这样0.625表示为二进制就是001 对于小数的存储，常见的方式有32位和64位两种 在32位中把第32位作为符号位，第31位到23位这8位作为小数位，剩下的23位用来表示整数 在64位中把第64位作为符号位，第63到52位这11位作为小数位，剩下的52位用来表示整数 以上小数存储规则可以参考IEEE754规范

小数运算误差

如果你尝试过用二进制表示0.3或者0.2就会发现，即使用64位（双精度）来表示，其实也只能是一个近似值，因为0.3这种小数是无法通过有限位的二进制表示的。

这样就引出了一个有趣的问题，我们以JavaScript来说明，在JavaScript中所有数字都是按照64位（11位小数位，52位整数位）方式存储的，这样我们得到0.3,0.2,0.1的二进制表示如下：

```
0.3: 0.0100 1100 110  
0.2: 0.0011 0011 001  
0.1: 0.0001 1001 100
```

我们取0.3与0.2的后三位做减法与0.1的后三位比较

```
0.3-0.2: 101  
0.1:      100
```

很明显两者是不相等的，所以如下代码是合乎情理的。

```
//JavaScript 0.2-0.3 != 0.1  
  
0.3 - 0.2 === 0.2 - 0.1;  
//false
```

这不是任何一门编程语言的问题，这是采用了IEEE754规范的计算机语言都兼备的问题。所以在计算机程序运算中尽可能采用小单位运算，比如 $0.3T = 300KG$ ，因为你知道的计算机存储300要比存储0.3准确的多。

基本的数字存储说完后，我们看看不同语言对数字类型的划分是什么样的：

Java的数字类型

名称	类型	位数	符号位	范围
字节型	byte	8	有	[-128, 127]
短整型	short	16	有	[-32768, -32767]
整型	int	32	有	$[-2^{31}, 2^{31}-1]$
长整型	long	64	有	$[-2^{63}, 2^{63}-1]$
浮点型	float	32	有	IEEE 754 binary floating point
双精度浮点型	double	64	有	IEEE 754 binary floating point

- 一般小计量的数字存储，可以用byte和short来节省内存开支。
- 对于货币这种需要精确数值的，不能用float和double，可以用Java提供的BigDecimal类。
- 在Java8以后int和long可以用来表示无符号位的整数，取值范围分别为： $[0, 2^{32}-1]$, $[0, 2^{64}-1]$

其他语言数字类型

我们以Java的数字类型为标准，来对比下其他语言的数字类型在定义上有什么区别。

JavaScript和Ruby

同Java不同，JavaScript和Ruby这样的弱类型语言，本身数字类型没有做特别细致的划分，它们用Number类型表示数字。但细节处理上JavaScript和Ruby又是不同的。

JavaScript并没有区分整数和小数，所有的数字都是遵循IEEE 754 标准采用64位float进行表示，而Ruby本身则是有区分Integer（Integer又分为Fixnum和Bignum）、Float和Rational（实数，类似于Java的BigDecimal类）的。

Python

对比Ruby提供的实数类型，Python和R则提供了一个更广泛的数据类型：complex number（复数类型），它表示为 $a+bi$ 的形式，其中a和b都是浮点类型，a表示实部，b表示虚部。

另外Python的数字类型还包括int、long和float，这一点和Ruby其实是相似的，不同的是Python的long类型是没有长度限制的，而int类型超过长度限制则会转换为long类型（这就是为什么有人会说Python只有long和float两种数字类型）。

基于JVM的二代语言

另外像Scala、Groovy这种基于JVM的二代语言，其数字类型是沿用自Java的，无非是Scala作为弱类型语言提供了类型推断机制，在使用数字类型时无需特别声明。

另外一门基于JVM的Lisp语言方言Clojure，它没有效仿Java的数字类型，而是采用了Integers和Float两种类型，其中Integers又区分为Decimal Integers（整数类型）、Octal Numbers（八进制数）、Hexadecimal Numbers（十六进制数）、Radix Numbers（进制类型）。类似的Perl语言也提供了Hexadecimal（十六进制）和Octal（八进制）两种类型。

Clojure的进制类型可以把任意进制数转换为十进制数，因此如果你用的语言提供了这样的类型，在处理进制转换时相当方便。

这里单独说明下Clojure的进制类型，上面我们讲过了二进制的算法，同八进制和十六进制的计算方法是类似的。只是为了和十进制区分，八进制以0开头，而十六进制以0x开头。

```
0175
```

```
0x1A
```

我们将这两个数字表示为10进制：

$$\begin{aligned}0175 &= 1 * 8^2 + 7 * 8^1 + 5 * 8^0 = 125 \\0x2E &= 2 * 16^2 + 14 * 16^1 + 10 * 16^0 = 746\end{aligned}$$

注意16进制中用A表示10，B表示11，C表示12，依此类推，所以E表示14。而Clojure的Radix Numbers可以定义进制的底数，比如刚刚的0175就可以表示为8r175，0x2EA就可以表示为16r2ea，r前表示底数，r后表示数字。因为数字从0-9共10个，字母从a-z共26个，所以底数的范围是1-36，即Clojure的Radix Numbers最高可以计算36进制数转换。

```
(println (format "%s" 11r12))  
;;13
```

微软系语言F#、C#、VB.net

我们以C#为例说明，微软除了提供了像Java数字类型中byte、short、int、long、double、float这些类型外，还特别针对有符号的short、int、long分别提供了无符号的ushort、uint、ulong类型，这些类型都无法表示负整数，但相反的却可以表示两倍于对应符号类型的正整数。

字符类型

在说明字符类型之前，我想先说明下字符编码。

字符编码

计算机是只识别二进制的，所以和数字一样我们要想办法把字符也映射为一串01的记号，数字类型采用十进制与二进制的转换解决了这个问题。对于字符我们只需要规范一个二进制数和字符之间对应关系的表就可以了。

ASCII

我们用两个字节，即8位（相应可以表示256个符号）来表示一些我们常用的字符，这一套字符对照表就是ASCII码，但是ASCII码只使用了后7位（第八位置为0）规定了与常用的128个字符的对应关系，这128个字符是a-z、A-Z、0-9以及一些标点符号和控制符号。

ASCII码是针对英文字符进行的规范，但不同的国家使用的语言不同（很明显我们就不用英文字符），不同国家规定的对照表也就不同，用多种不同的对照表去解析一个字符就会出现不同的结果。正因为不同编码规定的同一个二进制数表示的字符不同，所以如果编码和解码的规则不一致，就会出现乱码问题。因此我们需要一个通用的编码、解码规范（即是一个能将世界上常用的字符和二进制数进行对应的对照表）。

Unicode

为了解决上述问题，提出了Unicode字符集，它规定了字符和二进制数的对应关系，但没有给出具体的存储方式。因为英文字符用一个字节就足以表示，但是汉语字符可能要需要2个字节才能表示。如果以两个字节为单位进行存储，英文字符又会浪费掉一部分存储空间。

基于上述问题，就出现了不同的存储方案。

UTF-8

UTF-8采用了一种变长的存储方式解决了上述问题。为了让计算机能区分变长字符存储所需要的字节数，UTF-8做出如下规定：

1. 对于单字节字符，字节第一位是0，后面七位为这个符号的Unicode码，因此英文字母的Unicode码实际上和ASCII码是一致的。
2. 对于需要N个字节来存储的符号，第一个字节的前N位置为1，第N+1位置为0；后面N-1个字节的前两位都是10。比如三个字节存储的字符对应为：1110xxxx 10xxxxxx
10xxxxxx

这里UTF-8只是Unicode编码的一种存储实现，相应的还有UTF-16和UTF-32。

UCS-2

早期在做字符集统一的时候存在两个团队，一个是Unicode团队，另一个就是UCS团队，UCS团队提出的UCS-2采用两个字节16位的方式来存储一个字符，UCS-2是向下兼容ASCII的（也就是说UCS-2对ASCII中规定的字符与二进制数的对应关系予以保留，并追加更多字符）。但随后Unicode推出了UCS-2的超集Unicode-16，因为不需要两套规范，所以UCS-2就被并入Unicode-16。

UTF-16

UTF-16规定用十六进制、两个字节来存储字符，其范围是`0x0000 - 0xFFFF` 但两字节只能存储 2^{16} 即65535个字符，这是不够的，因此要存储更多字符就需要两个存储单元。但是这就又出现一个问题，怎么判断两个存储单元是表示两个字符还是表示一个字符。为了解决这个问题，UTF-16编码规范将`0xD800 - 0xDFFF` 这段设置为空，不对应任何字符。然后又把这一段空出来的空间分成两段，`0xD800-0xDBFF` 和 `0xDC00-0xDFFF`，对于超出`0xFFFF` 的字符追加的那个存储单元映射为这两段的数值，并放在前面。这样计算机检测到第一个存储单元的两个字节在`0xD800-0xDBFF` 和 `0xDC00-0xDFFF` 范围时就知道需要和后一个存储单元配合来定位为一个字符了。正是因为这个策略导致单个存储单元的UTF-16编码所能表示的字符数量要少。

目前Java、C#、Object-C、JavaScript这些语言内部编码都采用的UTF-16（严格的说JavaScript采用的是UCS-2编码）。

字符长度

刚刚也说过有些特殊字符需要两个存储单元来保存，所以采用UTF-16编码的语言通过length函数来取这个字符的长度时，就会返回2。

```
//Java 字符 长度  
System.out.println("".length()); //2
```

```
//JavaScript 字符 长度  
console.log("".length); //2
```

```
#Ruby 字符 长度  
puts "".length #1
```

注意Ruby、Go、Rust这些语言已经开始使用UTF-8作为内部编码了，所以Ruby对特殊字符取长度返回1。

字符和字符串

在Java和C++中字符串并不是一个原始的数据类型，而是一个字符序列，是一个类，字符才是基本数据类型。但是JavaScript和Ruby中字符串是作为基本数据类型提供的，相应的并没有细化字符的概念，字符只是长度为1的字符串。

```
//Java 字符和字符串

public static void main(String args[]){
    char normal = 'a';
    char unicode = '\u039A';
    String str = "string";
    char[] chars = {'s', 't', 'r', 'i', 'n', 'g'};
    String charstr = new String(chars);

    System.out.println(normal);      //a
    System.out.println(unicode);     //K
    System.out.println(str);        //string
    System.out.println(charstr);    //string
}
```

```
char = 'a'
string = 'string'

puts char      #a
puts string    #string
```

注意虽然Java中String是一个类，但字符串拷贝的时候是值类型，而不是引用类型。

```
//Java 字符串是值类型，拷贝修改不会影响源

public static void main(String args[]){
    char[] chars = {'s', 't', 'r', 'i', 'n', 'g'};
    String charStr = new String(chars);
    String equalCharStr = charStr;

    charStr = "string charStr";
    equalCharStr = "string equalCharStr";

    System.out.println(charStr);           //string charStr
    System.out.println(equalCharStr);      //string equalCharStr

    String str = "string";
    String equalStr = str;

    str = "string str";
    equalStr = "string equalStr";

    System.out.println(str);              //string str
    System.out.println(equalStr);         //string equalStr
}
```

Ruby和JavaScript中字符串本身就作为原始类型定义，所以字符串拷贝自然也是值类型拷贝。

```
#Ruby 字符串拷贝，也是值类型拷贝

str = "string"
equalStr = str

str = "string str"
equalStr = "string equalStr"

puts str      #string str
puts equalStr #string equalStr
```

字符串修改及内存优化

这部分内容已经在2.1 基础类型与栈内存分配中有介绍，因为字符串是值类型，所以字符串频繁修改时会消耗大量内存，因此部分语言本身提供了类来优化内存的使用。

```
//Java StringBuffer优化字符串内存占用

public static void main(String args[]){
    char[] chars = {'a', 'b', 'c'};
    String s = new String();
    StringBuffer sb = new StringBuffer();

    for(char c : chars){
        s+= c;
        sb.append(c);
    }
    System.out.println(s);      //abc
    System.out.println(sb);     //abc
}
```

原子类型

原子类型是函数式语言中常用的数据类型，它就像是一个常用的、不变的字符，因为有些字符会被多次使用，对于这样的字符多次赋值给不同的字符串有些没有必要，同时也存在内存的浪费。所以一些语言中加入了原子类型专门存储这种常用的字符串文本。

Ruby中用`:`修饰原子类型，比如`:red`就是文本值为`red`的原子类型（注意Ruby中原子类型其实叫做符号类型symbol）。Erlang中用`' '`单引号包裹表示原子类型，比如`'red'`就是文本值为`red`的原子类型（Erlang中原子类型用单引号包裹，字符串类型用双引号包裹）。

```
#Ruby 字符串类型与原子类型比较

string_a = "string"
string_b = "string"
symbol_a = :string
symbol_b = :string

puts string_a.object_id      #70329489970500
puts string_b.object_id      #70329489970480
puts symbol_a.object_id       #158248
puts symbol_b.object_id       #158248
```

可以看出虽然都是相同的文本字符，但是用字符串类型会分别存储，而符号类型则会指向同一个存储地址，对于一个不变的文本信息用符号类型可以降低内存的开销，并提高程序性能。

另外symbol也常作为对象的key来使用，因为key是不允许改变的，而且同一模板下的对象key值也是相同的。

```
#Ruby 用符号表示散列的键
```

```
prices = {  
  :banana => 2,  
  :apple   => 3,  
  :pear    => 2.5  
 #ruby中用=>绑定Hash键值对。  
}
```

第三章 与值类型相对的引用类型

继第一章第五节和第二章，这一章主要说明数据类型中的引用类型，这里汇总大多数语言中常用的引用类型，进行了归类划分，主要区分为集合类型，其中集合类型分为按顺序索引的数组及按键索引的散列，函数类型和自定义类型，这些介绍中混合了命令式语言和函数式语言两种范式下相关的引用类型。

1. 介绍命令式下的数组、链表及声明式下的列表。
2. 介绍命令式下的散列、枚举及声明式下的元组。
3. 介绍命令式和声明式两个范式下和函数相关的概念。
4. 介绍了命令式下的结构体、类和声明式下的类型。

集合类型

这里我准备介绍的集合类型主要指命令式语言下常用的数组类型和声明式语言下常用的列表类型。因为语言本身设计上的缘故，通常静态类型语言所使用的集合多是定长的，也就是不能动态拓展，而动态类型语言相对是随时可以拓展的。同样弱类型语言的集合类型是异质的，也就是数组元素可以不同数据类型。这里因为Java语言设计上的缘故，提出了数组、链表、集的概念，这些概念通常为集合类型，所以我也用集合类型来统一称谓了。

接下来我们先看下最基本的数组类型，然后以Java语言为主说明如何通过集合类型完善Java数组对比其他语言所不具备的功能。

数组

在Java中数组类型是同质定长的，也就是说Java的数组是不支持不同类型数据混合的，且其长度是在实例化时指定的。

```
// Java 数组类型

int[] a = { 1, 2, "three" };
//error incompatible types: String connot be converted to int

int[] b = new int[10];
b[10] = 11;
//exception in thread main java.lang.ArrayIndexOutOfBoundsException
```

可以看出声明一个异质的数组会报编译时错误，而对超出数组索引的元素赋值会报运行时异常（数组越界异常）。

于此不同JavaScript和Ruby的数组就是异质变长的，不过JavaScript似乎并不提供无序数组的概念，也就是说数组是有且只有顺序的。

```
//JavaScript 数组类型

var a = [1, 2, 3];
console.log(a.length);      //3

a[3] = 4;
console.log(a.length);      //4

a[6] = 7;
console.log(a);            // [1, 2, 3, 4, undefined, undefined, 7]
console.log(a.length);      //7
```

在数组方面Ruby和JavaScript表现是一致的。

```
#Ruby 数组类型

a = [1, 2, "three"]
puts a.length          #3

a[3] = 4
puts a.length          #4

a[6] = 7
puts a                #[1, 2, 3, 4, nil, nil, 7]
puts a.length          #7
```

其实Ruby和JavaScript支持异质数组是因为Ruby中一切都是对象，而JavaScript中一切都是原型，采用相同的逻辑，如果Java中定义类型为基类（Object）的数组时，同样也是支持异质结构的。

```
//Java 异质数组

Object[] o = { "string", 1, false };
System.out.println(o[0]);        //string
System.out.println(o[1]);        //1
System.out.println(o[2]);        //false
System.out.println(o.length);    //3
```

而关于Java如何实现变长数组会在介绍了数组长度相关的几个概念后再详细说明。

数组长度

上面已经说过Java的数组是定长的，所以对长度为3的数组进行`a[4]=5`这赋值操作会报数组越界的异常。

而JavaScript和Ruby中数组是变长的，也就是你可以对长度为3的数组执行类似`a[4]=5`这样的赋值操作，但是不同的是Ruby会把中间遗漏掉的元素用`nil`补齐，而JavaScript则直接跳过，因为JavaScript本身没有数组类型，数组实际上是Key值为数字的Object，你可以给一个Object定义任何键值，所以跳过数组索引顺序赋值本身也是合法的（这样的数组叫做稀疏数组）。

```
#Ruby 变长数组

a = [1, 2, 3]
a[5] = 6

puts a          #[1, 2, 3, nil, nil, 6]
puts a.length   #6
```

```
//JavaScript 变长数组

var a = [1, 2, 3];
a[5] = 6;

console.log(a);           // [1, 2, 3, , , 6]
console.log(a.length);    // 6
```

与Ruby不同，JavaScript甚至支持对数组长度进行赋值，当设置的`length`值大于数组长度时，JavaScript用`undefined`进行填充，形成一个相同长度的数组。假如设定的`length`值小于数组实际长度，数组就会进行截断处理。

```
//JavaScript 对数组长度赋值

var a = [1, 2, 3];
console.log(a.length);    // 3

a.length = 5;
console.log(a[4]);        // undefined
console.log(4 in a);      // false
console.log(a);           // [1, 2, 3, , , ], 这时a是一个稀疏数组

a.length = 2;
console.log(a[3]);        // undefined
console.log(3 in a);      // false
console.log(a);           // [1, 2]
```

稀疏数组

在JavaScript初始化一个数组时，跳过某些元素不指定具体值，这样的数组就是稀疏数组。当遍历这个数组时会跳过这些没有值的元素（这些元素缺省值为undefined，但是不存在对应的key）。

```
//Javascript 稀疏数组

var a = [ , , 3];
a.forEach(function(v){
    console.log(v)
});
//3
```

如果我们初始化的时候指定了缺省值，这时数组就不是稀疏数组了。

```
//Javascript 指定数组元素为undefined

var a = [undefined, 1]
,   b = [ , 1];

console.log(0 in a);      //true
console.log(0 in b);      //false
Object.keys(a);           //["0", "1"]
Object.keys(b);           //["1"]
```

因为JavaScript中数组本身是key值为数字的Object，所以可以用in来判断对象是否存在键。上面的例子可以发现a[0]是存在的，而b[0]是不存在的，所以a是稠密数组，而b是稀疏数组。

因为Java数组默认都是有缺省值的，所以Java语言不存在稀疏数组的概念。

```
//Java 数组元素缺省值
int a[] = new int[3];
a[2] = 3;

for(int i=0; i<a.length; i++){
    System.out.println(a[i]);
}
//003
```

Java中整型数组的缺省值为0，字符型数组缺省值为null。

不过在Android SDK中提供了稀疏数组类SparseArray，它是用来处理稀疏矩阵的，和JavaScript中定义的稀疏数组是两种概念，简单说明下稀疏矩阵就能清楚的分辨出两者的区别了。

假设我们用一个二维数组来表示一个5*5矩阵

```
1 0 0 0 0  
0 2 0 0 0  
0 7 0 0 0  
0 0 0 0 9  
0 0 0 0 0
```

上面这个矩阵中有效的数值其实只有4个，剩下的21个都是空，这时用一个二维数组来存储这个矩阵会浪费大量的存储空间，我们考虑通过记录有效数据来压缩这个矩阵，如下：

```
5 5 4 //原矩阵的 行 列 有效数字  
1 1 1 //第一个元素所在 行 列 及数值  
2 2 2 //第二个元素所在 行 列 及数值  
3 2 7 //第三个元素所在 行 列 及数值  
4 5 9 //第四个元素所在 行 列 及数值
```

这样就把一个5*5的矩阵压缩为3*4的矩阵了，这个压缩后的矩阵就是稀疏矩阵，因为采用了数组类型所以笼统的就叫做稀疏数组了（注意Java SE本身没有特别定义稀疏数组的概念）。

链表

为了方便理解你可以把链表看做是一个变长的数组，JavaScript和Ruby所提供的数组本身就是变长的，那么抛开这两门语言，单独说下Java的链表。

链表

List是Java的一个接口，它有三个实现类ArrayList、LinkedList和Vector，List允许你根据需求动态的增加数组元素。

最常用的List实现类是ArrayList，只有在频繁的向数组头部插入元素，或者通过遍历数组删除元素时才会用LinkedList。而Vector和ArrayList的不同在于Vector是同步的，在多线程编程时使用Vector更加稳妥。我们来看下通过ArrayList追加元素的例子。

```
//Java 链表

/* 这里需要引入 List 接口 和 ArrayList 实现类 */
import java.util.List;
import java.util.ArrayList;
class Demo {
    public static void main(String args[]){
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.set(0, 3);
        System.out.println(list.get(0));      //3
        System.out.println(list.get(1));      //2
        System.out.println(list.size());      //2
    }
}
```

`ArrayList.set(index, value)` 可以将索引为`index`的元素修改为`value`，需要注意你不能用这个方法对长度为3的链表取第四个元素赋值，这样做会和数组一样抛出数组越界异常。

```
//Java 通过ArrayList的add方法拓展数组元素

List<Integer> list = new ArrayList<Integer>();
list.add(1);
list.set(3, 3);
//Exception in thread "main" java.lang.IndexOutOfBoundsException
```

另外和数组不同`List`的不是存储在一段连续的空间的（注意数组是存储在一段连续空间的），同时数组必须初始化大小，而`List`可以不初始化大小，`List`可以删除和插入元素，但是数组不可以。不过相比之下数组效率比`List`高一些。

集

除了`List`外Java还提供了`Set`，和`List`不同的是`Set`不允许重复元素。`Set`也是一个接口，实现类有`HashSet`、`TreeSet`、`LinkedHashSet`。`HashSet`是最常用的，它通过散列（hash）来存储节点，因此是无序的。`TreeSet`通过红黑树存储节点，所以元素的顺序按照红黑树的遍历顺序计算，效率不如`HashSet`。`LinkedHashSet`从名字可以看出，它的节点是由一个链表串联起来的，所以元素的顺序跟插入顺序有关。

同样以最常用的`HashSet`为例对比链表来说明集更强调元素的唯一。

```
//Java 集

import java.util.*;
class Demo {
    public static void main(String args[]){
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(1);
        list.add(2);
        System.out.println(list);           // [1, 2, 2]
        System.out.println(list.size());    // 3

        Set<Integer> set = new HashSet<Integer>();
        set.add(1);
        set.add(1);
        set.add(2);
        System.out.println(list);           // [1, 2]
        System.out.println(set.size());    // 2
    }
}
```

注意集本身是用来存储不重复数据集的，通常的操作是判断元素是否存在于集中，很少直接获取指定元素，所以集本身不提供get方法。关于集不提供get方法你也可以理解为因为集不像数组或链表一样按索引排序，同时也不像散列一样可以通过键来定位元素，所以集无法定位到具体某一个元素，也就不提供get方法了。

如果说链表是一个变长的数组，那么集就可以看做是一个无序的链表。为了方便理解，我将List和Set划分到数组类型下。至于究竟是用Array还是List，是HashSet还是LinkedHashSet就取决于你具体面对的数据格式和业务场景了，如果你不够熟悉这些类，那么ArrayList是最通用的了，就像Ruby和JavaScript并没有对数组进行变长定长、有序无序这样的划分，但是不得不说它们的数组类型本身就足够灵活了，Java的ArrayList也是这样的工具类，所以不妨试试看吧。

列表

介绍完命令式语言中常用的数组类型后，接下来说下通数组类型类似，在声明式语言中起着重要作用的列表类型。

因为大多数函数式语言都是静态类型的，所以列表基本上也都是非异质的。不同于之前的语言，函数式语言针对列表提供了更丰富的操作，比如你可以通过一些高阶函数直接过滤列表元素并进行运算、操作，也可以使用更加丰富的API对列表进行合并、截断，正因为需要支持灵活的操作，列表本身也是变成的。

我们来看下Haskell针对列表提供的一些操作。

合并列表

```
-- Haskell 合并列表
```

```
[1, 2, 3] ++ [4, 5]  
-- [1, 2, 3, 4, 5]
```

对列表所有元素取绝对值

```
-- Haskell 操作列表
```

```
map abs [1, -2, 3]  
-- [1, 2, 3]
```

过滤出大于三的元素

```
-- Haskell 过滤列表
```

```
filter (>3) [1, 4, 7, 2]  
-- [4, 7]
```

取出列表中元素值小于5，且能被三整除的元素并取反

```
-- Haskell 列表推导
```

```
[ x | -x <- [1, 3, 5, 6], x `mod` 3 == 0, x < 5]  
--- 3
```

取得数组第三个元素

```
-- Haskell 列表取值
```

```
head . tail . tail $ [1, 2, 3, 4, 5]  
-- 3
```

在函数式语言中高阶函数配合列表操作可以以简短的代码完成复杂的任务，具体示例我会在函数式编程中详细说明。

散列类型

散列是一种由键值对构成的数据类型，拿他与数组比较的话，可以理解为数组是以升序数字为键的散列，散列是索引为字符串的数组。

通数组类似，我会先说明命令式语言下的散列类型，然后再介绍声明式语言下同等功能的元组。

散列

在Ruby中散列是作为基本数据类型定义的，在JavaScript中对象类型就是散列结构，因此也可以认为散列是JavaScript的基本数据类型，最后Java中散列同List、Set类似通过Map接口实现。

```
#Ruby 散列

language = { "name"=> "ruby", "type"=> "dynamic" }
puts language["name"]      #ruby
```

```
//JavaScript 对象

var language = { name: "javascript", type: "dynamic" }
console.log(language.name);    //javascript
```

```
//Java 散列

Map<String, String> language = new HashMap<String, String>(){
{
    put("name", "java");
    put("type", "static");
}
};

System.out.println(language.get("name"));    //java
```

注意散列和数组一样，是引用类型，在使用时要特别小心，如果不想破坏原引用最好进行深度拷贝。

```
#Ruby 散列属于引用类型

a = language
a["name"] = "Ruby"

puts language["name"]      #Ruby
```

```
//JavaScript 散列属于引用类型

var a = language;
a.name = "JavaScript";

console.log(language.name); //JavaScript
```

```
//Java 散列属于引用类型

Map<String, String> a = new HashMap<String, String>();
a = language;
a.put("name", "Java");

System.out.println(language.get("name")); //Java
```

键值改变

在使用散列时保证键值不可修改是很重要的，如果我们通过变量定义散列的键，就会存在一种额外的风险。

我们以JavaScript为例说明。

```
//Javascript 变量作为属性名

var name = "name"
, language = {};

language[name] = "javascript";
console.log(language, language[name]);
//{ name: 'javascript' } 'javascript'

name = "rename"
console.log(language, language[name]);
//{ name: 'javascript' } undefined
/* 这里因为不存在键rename, 所以language[name]返回undefined */
```

从上面的例子可以看出如果用变量作散列的键，当变量发生改变时，再用这个变量获取散列的值时会出现问题。

在Ruby中可以使用符号来作为散列的键，符号是不可修改的，这样就避免了如上问题。

```
#Ruby 符号作为散列的键
language = { :name=> "ruby", :type=> "dynamic" }
```

Ruby中用符号作为键还有另外一个优势，因为符号是同一个引用，所以这样的散列相比字符串索引运算性能要高。

```
#Ruby 符号做键与字符串做键查询性能对比

require 'benchmark/ips'
STRING_HASH = { "foo" => "bar" }
SYMBOL_HASH = { :foo => "bar" }
Benchmark.ips do |x|
  x.report("string") { STRING_HASH["foo"] }
  x.report("symbol") { SYMBOL_HASH[:foo] }
end

# Warming up -----
# string    102.788k i/100ms
# symbol    116.332k i/100ms
# Calculating -----
# string      3.799M (± 6.5%) i/s -      18.913M
# symbol      7.810M (± 6.8%) i/s -      38.855M
# 可以看出符号的运算速度是字符串的两倍多。
```

同理Java下键值通过变量来存储，发生改变后也会返回null值。

```
//Java 变量作为散列的键

public static void main(String args[]){
  Map<String, String> language = new HashMap<String, String>();
  String name = "name";
  language.put(name, "Java");
  System.out.println(language +" "+ language.get(name));
  //{name=Java} Java

  name = "rename";
  System.out.println(language +" "+ language.get(name));
  //{name=Java} null
}
```

可以将变量修饰为final，或者用匿名内部类传入形参不可变的特性来保证键值不可修改。

```
//Java 匿名内部类保证键值不可修改

public static void main(String args[]){
    String name = "name";
    Map<String, String> language = new HashMap<String, String>(){
        {
            put(name, "java");
            put("type", "static");
        }
    };

    name = "NAME";
    //error: local variables referenced from a inner class must be final or effectively final
}
```

上面例子可以看出如果你试图修改匿名内部类中引用的变量name，程序在编译时就会抛出一个编译错误。

枚举

枚举类型可以看做是一个键为常量，值为数字类型（默认值从0开始自增长）的散列，其主要用来存储一些列性质相同的常量元素。

目前主流语言都加入了枚举类型：

```
//C 枚举

enum season {
    Spring = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 3;
}
```

```
//Java v1.5后引入枚举

public enum Season {
    SPRING, SUMMER, AUTUMN, WINTER;
}
```

在分支语句中使用枚举能提高代码的可读性：

```
//Java 分支语句中使用枚举

public static void main(String args[]){
    Color color = Color.RED;

    switch(color){
        case RED:
            System.out.println("red");
            break;
        case GREEN:
            System.out.println("green");
            break;
        case YELLOW:
            System.out.println("yellow");
            break;
    }
}
```

枚举的另外一个主要用途是充当静态常量，对比静态常量枚举类型更加安全。

```
//Java 常量在使用时需要合法性验证

interface Color{
    static final int RED = 1;
    static final int GREEN = 2;
    static final int YELLOW = 3;
}

public class Main{
    public Boolean isRed(int i){
        //为保证参数有效，需要合法性验证
        if(i < 1 || i > 3) System.out.println("error code");

        if(i == 1) return true;
        else return false;
    }
    public static void main(String args[]){
        Main main = new Main();
        if(main.isRed(Color.RED)) System.out.println("red");      //red
    }
}
```

```
//Java 枚举类型比常量更加安全

enum Color{
    RED, GREEN, YELLOW;
}
public class Main{
    public Boolean isRed(Color color){
        //枚举类型限定了输入参数，保证了合法性输入
        if(color.ordinal() == 0)  return true;
        else                      return false;
    }
    public static void main(String args[]){
        Main main = new Main();
        Color red = Color.RED;
        if(main.isRed(red))  System.out.println("red");      //red
    }
}
```

合理的通过枚举替换常量的使用是一种良好的编程习惯，当然抛去枚举的概念不用也不会产生什么严重的问题，枚举的使用与否也是个人习惯的一种选择，这里只是讲到散列类型联系到枚举，简单进行说明。

元组

除了列表外函数式语言中另外一种常用的数据结构就是元组了，元组和列表的区别在于元组允许异质类型，不过元组多数都是定长的。

鉴于元组如上的性质，一般元组用来存储数据，而不处理数据，当你在一个函数中需要返回多种类型的数值时，元组就是一个很好的包装。

我们来看下如何定义元组，并从中取值。

定义一个元组

```
--Haskell 元组
(1, "two", "3", "three")
```

元组是定长的，不支持列表一样的合并运算

```
--Haskell 元组无法合并
(1, "two", 3) ++ ("three", 4)
--Couldn't match expected type
```

元组不支持列表一样的函数应用，因此无法处理数据

```
--Haskell 元组无法处理数据

map (+3) (1, 2, 3)
--Couldn't match expected type
```

定义一个函数third，取得元组的第三个值

```
--Haskell 从元组中获取元素

third:: (a, b, c, d) -> c
third (_ , _ , c , _ ) = c

third (1, "two", "3", "three")
--3
```

大多数函数式语言都针对二元组（有两个元素的元组）提供了常用的函数来简化了元组的访问。

```
--Haskell 二元组

fst ("name", "haskell")      --name
snd ("name", "haskell")      --haskell
```

这里二元组就相当于命令式下的散列类型，第一个元素作用等同于散列的键，第二个元素相当于散列的值。当然也可以采用多元组（奇数位存键，偶数位存值）或者元组列表来模拟多键值对的存储。

最后我们看下如何配合列表来处理稍微复杂一点的数据类型：

```
--Haskell 取列表中第三个元组第二个元素的值

snd . last . take 2 $ [("language", "ruby"), ("language", "java"), ("language", "haskell"),
("language", "javascript")]
--haskell
```

函数类型

函数

函数是程序中的一段子程序，它用来执行一定的操作或运算，通常函数包括一个入口和一个出口。入口即参数输入（输入可以为空，即不输入参数），出口即函数返回值（输出可以为空，即不返还值），简单的说在你使用一个函数时，需要输入一些参数，函数会根据输入参数进行运算，然后会返回给你一个运算结果。

比如如下一个除运算函数，需要输入除数b与被除数a，函数进行除法运算后会返回一个商 a/b 。

```
//Java 函数

public static double devide(double a, double b){
    return a/b;
}

public static void main(String args[]){
    double r = devide(3, 2);
    System.out.println(r);      //1.5
}
```

分类

函数分为标准函数和自定义函数，所谓标准函数就是语言本身提供的函数，通常这是一些通用的函数，视具体语言不同标准函数库也稍有差别。

比如针对数组排序的函数sort，在java中该函数会破坏原数组：

```
//Java sort方法会破坏原数组

public static void main(String args[]){
    int[] a = {4, 7, 5, 1, 2};
    int[] b = Arrays.sort(a);
    for(int i in a) {
        System.out.print(a[i] + " ");
        //1 2 4 5 7
    }
}
```

可以看出数组a的排序也发生了改变。对比Java，Ruby提供了两种排序方法sort和sort!，其中sort方法排序数组会保留原数组，返回一个新的排序后的数组，而sort!方法和上述中Java的Arrays.sort()方法类似，会破坏原数组。值得一提的是Ruby对大多数会产生破坏性的方法都提供了一组这样的函数，其中！结尾的方法表示会破坏原结构，之所以用！大概是想提示你这个函数要慎重使用。

```
#Ruby sort和sort!方法对比
```

```
a = [4, 7, 5, 1, 2]
a.sort
puts a    #[4, 7, 5, 1, 2]

a = [4, 7, 5, 1, 2]
a.sort!
puts a    #[1, 2, 5, 5, 7]
```

对于大多数语言而言最基本的函数都是有提供的，比如字符串处理函数：substring：字符串截取、concat：字符串合并、replace：字符串替换、toUpperCase/toLowerCase：大/小写转换、charAt：截取字符、startsWith/endsWith：开始/结束字符等；以及数组处理函数：join：数组转字符串、reverse：数组翻转、sort：数组排序、concat：数组组合并、slice：数组截取、splice：插入删除等。

而自定义函数就是用户自己定义的函数。

抛开这种分类方式，我准备按照两种通用的编程范式来区分介绍函数，即面向对象下的函数和函数式下的函数。面向对象的核心就是以对象的方式处理问题，所以这部分函数通常都和类/对象相关，而函数式则以函数的方式处理问题，所以这部分的函数通常是一些函数的组合。

面向对象下函数

函数重载

函数重载是针对于多个函数而言的，对于像C++、Java、C#这些静态类型语言而言，函数参数是需要声明具体类型的，相同名称不同参数类型的函数可以看做是不同的函数。重载就是针对这一特点而言的，函数重载允许创建输入参数或输出参数不同，但函数名相同的函数。

来看下Java的函数重载：

```
//Java 函数重载

public static double add(double a, double b){
    return a+b;
}

public static int add(int a, int b){
    return a+b;
}
```

上面两个函数都叫做add，但是输入参数类型和返回结果类型不同，这就是函数重载。因为语言特性不同，对于动态语言而言，函数重载是没有意义的，因为动态语言本身是支持类型推导，所以没必要重复声明同一个函数应付不同的参数类型。

```
#Ruby 动态类型语言不必重复接受不同参数的同一函数
```

```
def add(a, b)
  a+b
end

puts add(1, 2)      #3
puts add(1.3, 1.2) #2.5
```

Ruby会根据输入参数进行类型推导，并根据实际传入类型进行运算，输出对应类型的返回值。

运算符重载

如果函数重载是对一个函数重新定义，那么运算符重载就是对运算符对应的运算规则重新定义。出于语言设计者的考虑，运算符重载也不是所有语言都支持的。比如Java语言就不提供运算符重载，而C++和Ruby则提供了运算符重载。

因为Ruby的运算符本身就是一个函数，只是函数名称是运算符号而已，所以重新定义对应符号名的函数就相当于运算符重载了。

比如这个例子：假设我们有一个班级类Class，Class下有两个属性：boy_numbers和girl_numbers，分别记录了班级里男生人数和女生人数，现在我们想不区分性别，通过一个简单的加法运算来求出两个班级的总人数。这里我们用Ruby来重写 + 运算。

```
#Ruby (+)运算符重载

class Class
  attr_accessor :boy_numbers, :girl_numbers
  def initialize(boys, girls)
    @boy_numbers = boys
    @girl_numbers = girls
  end

  def +(c)
    @boy_numbers + c.boy_numbers + @girl_numbers + c.girl_numbers
  end
end

puts Class.new(27, 24) + Class.new(26, 23)    #100
```

可以看出我们重写了+运算符，使他对传入的对象c取得boy_numbers和girl_numbers的值并与自身的boy_numbers和girl_numbers值求和，最后返回总人数。

构造函数

构造函数是类在实例化一个对象时执行的函数，一般用于初始化对象的属性。

Java构造函数

在Java中构造函数名与类名一致，你可以显示的定义构造函数，并执行一些赋值操作。

```
//Java 显式定义构造函数并执行赋值操作

class Demo{
    public int a, b;
    public Demo(int a, int b){
        //构造函数
        this.a = a;
        this.b = b;
    }
    public static void main(String args[]){
        Demo d = new Demo(1, 2);

        System.out.println(d.a +" , "+ d.b);      //1, 2
    }
}
```

Ruby构造函数

在Ruby中类的构造函数都用initialize命名，同样你也可以显示的定义，并进行赋值操作。

```
#Ruby 构造函数

class Demo
    attr_accessor :a, :b      #配置属性可读可写
    def initialize(a, b)
        #构造函数
        @a = a
        @b = b
    end
end
d = Demo.new(1, 2)

puts d.a, d.b      #1 2
```

析构函数(destructor)

与构造函数相反，析构函数在对象结束调用后执行，用来释放对象所占用的内存空间，回收资源。现在的高级语言都加入了垃圾回收机制帮我们处理废弃的对象并回收内存空间。所以通常你应该也不会去定义析构函数了。

在C++中析构函数采用在构造函数名前加~的方式来命名。

```
//C++ 构造函数和析构函数

class Demo{
public:
    Demo(){};      //构造函数
    ~Demo();       //析构函数
}
```

实例方法和类方法

对于面向对象语言而言，实例方法（也可以叫作成员方法）和类方法是不得不说的一个概念。

简单的说实例方法是属于对象的函数，而类方法是属于类的函数，也就是说前者通过对象调用，而后者则通过类调用。

Java中的实例方法和类方法

在Java中类级别的方法或者属性通过static关键字修饰。

```
//Java 类方法和实例方法

class Animal{
    public static int numbers;
    public int index;

    public static int getNumbers(){
        return numbers;
    }
    public int getIndex(){
        return index;
    }
    public Animal(){
        numbers++;
        index++;
    }
    public static void main(String args[]){
        Animal cat = new Animal();
        Animal dog = new Animal();

        System.out.println(Animal.getNumbers());      //2
        System.out.println(dog.getIndex());           //1
    }
}
```

上述例子中声明了类属性number以及类方法getNumber， 实例属性（也有叫成员变量的）index和实例方法getIndex。输出结果中可以看出，每次实例对象都会有一个新的index，但是number却是共享的，因为它是类级别的。

Ruby中的实例方法和类方法

Ruby则通过 @@ 来定义类属性，通过self关键字来定义类方法， self代表当前运行时的上下文，这里所指上下文就是类本身。

```
#Ruby 类方法和实例方法

class Animal
  @@numbers = 0
  def initialize()
    @@numbers += 1
    @index = 1
  end
  def self.get_numbers
    @@numbers
  end
  def get_index
    @index
  end
end
cat = Animal.new
dog = Animal.new

puts Animal.get_numbers      #2
puts dog.get_index          #1
```

上面的例子和Java一样，只是语法上稍有不同，Ruby通过`@`定义成员属性，通过`@@`定义类属性。

私有方法

私有方法可以看做是一种特殊的实例方法和类方法，私有方法是只有在当前类（上下文）下才能调用的方法，通常用`private`关键字修饰，如果这个类存在子类，那么子类是无法访问该方法的（注意子类会继承该方法，只是无法访问）。

```
//Java 类的私有方法

class A{
    private void privateMethod(){
        System.out.println("This is a private method of class A");
    }
    public void callPrivateMethod(){
        this.privateMethod();
    }
}

class B extends A{



public class Demo{

    public static void main(String args[]){
        A a = new A();
        B b = new B();

        a.privateMethod();
        //error: privateMethod() has private access in A

        a.callPrivateMethod();
        //This is a private method of class A

        b.privateMethod();
        //error: cannot find symbol b.privateMethod();

        b.callPrivateMethod();
        //This is a private method of class A
    }
}
}
```

可以看出privateMethod只有在类A的上下文下才可以访问，在外部通过a对象调用会报权限错误，而子类调用则找不到该方法。

```
#Ruby 类的私有方法

class A
    private
        def privateMethod
            puts "this is private method of class A"
        end
    public
        def callPrivateMethod
            privateMethod()
        end
end
class B < A
end

a = A.new
b = B.new

a.privateMethod
#private method `privateMethod' called for #<A:0x007fde9309e038> (NoMethodError)

a.callPrivateMethod
#this is private method of class A

b.privateMethod
#private method `privateMethod' called for #<A:0x007fde9309e038> (NoMethodError)

b.callPrivateMethod
#this is private method of class A
```

可以看出Ruby中的私有方法调用结果和Java一样，对于类A所在的上下文外是无法访问的。

友元函数

友元函数是针对一个类而言的，当一个函数定义为一个类的友元函数，那么这个函数就可以无视访问权限关键字（private、protected）直接访问这个类的属性。因为友元函数的设计降低了封装性，和面向对象设计的根本原则有所违背，所以在后续的语言中不再提供这个概念，这里也不过多赘述了。

属性访问器

同私有方法类似，通过private修饰的属性来是无法在外部直接访问的。因此需要提供一个属性读与写的入口，这就是属性访问器。通常属性访问器用set/get表示，set表示写，get表示读。

```
//Java 属性构造器

class A{
    private String privateVariable;

    public void setVariable(String a){
        this.privateVariable = a;
    }
    public String getVariable(){
        return this.privateVariable;
    }
}

public class Demo{
    public static void main(String args[]){
        A a = new A();

        a.privateVariable = "private variable";
        //error: privateVariable has private access in A

        System.out.println(a.privateVariable);
        //error: privateVariable has private access in A

        a.setVariable("private variable");
        System.out.println(a.getVariable());
        //private variable

    }
}
```

可以看出无法在类外直接访问类的私有属性，需要通过set和get方法进行读写操作，这样做可以保证数据读写的安全。

接下来我们看看Ruby中是如何实现属性构造器的。

```
#Ruby 属性构造器

class A
    attr_accessor :privateVariable
end

a = A.new
a.privateVariable = "private variable"

puts a.privateVariable    #private variable
```

在Ruby中，类的所有属性都是private的。Ruby提供了对属性读写的配置：attr_reader、attr_writer和attr_accessor，其中attr_reader表示属性可读，相当于Java的get方法，attr_writer表示属性可写（配置）相当于set方法，最后这里用到的attr_accessor表示属性既可读又可配置，相当于同时定义了set方法和get方法。属性具备了读写特性后就可以通过`.`运算进行读写操作了。

虚函数（virtual function）和函数重写（override）

虚函数是C++中提出的一个概念，当子类继承父类时，如果父类中某个函数定义为虚函数，那么子类可以重写这个函数的实现。虚函数需要用virtual关键字修饰。

```
//C++ 虚函数

class A{
public:
    virtual void VirtualFunction(){
        cout << "This is virtual function of class A";
    }
};

class B : public A{
public:
    void VirtualFunction(){
        cout << "Rewrite virtual function in class B";
    }
};
```

虚函数是面向对象语言中实现多态的一种重要手段。在Java语言中所有函数默认都是虚函数，也就是子类可以重写所有父类的函数，因此Java中称作函数重写。

```
//Java 函数重写

class A{
    public void overrideFunction(){
        System.out.println("function of class A");
    }
}

class B extends A{
    public void overrideFunction(){
        System.out.println("function override by class B");
    }
}
class C extends A{
    public void overrideFunction(){
        System.out.println("function override by class C");
    }
}

public class Main{
    public static void main(String args[]){
        A a = new A();
        B b = new B();
        C c = new C();

        a.overrideFunction();      //function of class A
        b.overrideFunction();      //function override by class B
        c.overrideFunction();      //function override by class C
    }
}
```

函数式语言中的函数

匿名函数

匿名函数是无需显示声明的函数，一般用于基本不会复用或使用次数有限的函数。匿名函数可以使代码简化，方便调用。

除此之外匿名函数也常用作高阶函数的参数或返回值，这一点会在介绍函数式的时候详细说明。

JavaScript中匿名函数

```
//JavaScript 匿名自执行函数
console.log(function(){ return 2 }());    //1
```

我们在log函数里传入了一个匿名函数 `function(){ return 2 }`，紧接着通过`()`运算符执行了这个匿名函数。

接下来再看下匿名函数作为返回值的例子：

```
//JavaScript 匿名函数作为返回值

function add (a){
    return function(b, c){ a+b+c };
}

add(1)(2, 3);      //6
```

上面的例子中add函数返回了一个匿名函数 `function(b, c){ a+b+c }`，当调用add(1)时会返回一个函数 `function(b, c){ 1+b+c }`，所以我们可以继续用`()`运算符传入参数b与c执行返回的匿名函数。

Java匿名内部类和lambda表达式

Java本身是不支持匿名函数的，但是可以通过匿名内部类实现匿名函数的功能。

```
//Java 匿名内部类模拟匿名函数

class Demo{
    public static void main(String args[]){
        new Object(){
            void add(int a, int b){ System.out.println(a + b); }
        }.add(1, 2);
    }
}
```

上面我们在输出方法中实例化了一个匿名内部类，并在类中定义了add方法，紧接着用这个实例化的对象调用了add方法，早期版本的Java不支持匿名函数与其初衷的设计也是有关的。在后续版本Java8中引入了lambda，通过lambda表达式也可以避开丑陋的匿名内部类直接实现匿名函数。

```
//Java 对比匿名内部类与lambda表达式

/* 匿名内部类实现 */
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("anonymous inner class");
    }
}).start();

/* lambda表达式实现 */
new Thread(() -> System.out.println("lambda express")).start();
```

在这里还要强调一点，虽然Java8引入了lambda的特性，但其实并非是单纯的匿名函数，和内部类一样只是一种匿名函数的实现手段，lambda表达式和匿名函数最大的区别就在于lambda并不是一个可以通过`()`运算符直接调用的函数，它只是一段带有作用域的代码，其主要目的是数据及代码的传递，如果从这方面看匿名函数的话，也可以理解为匿名函数是一种以函数的形式来传递数据和代码的手段。

Ruby中的proc和lambda

在Ruby中匿名函数也是通过lambda实现的，不同的是Ruby中的lambda是一个proc对象，Ruby用proc和block两个概念来包装代码段，区别在于proc是Proc类的对象，而block只是单纯的代码段，所以block只能作为代码段使用，而proc可以像变量一样传递。

虽然lambda是一个proc对象，但两者也存在区别：lambda会检查参数的合法性（比如实参个数与形参是否匹配），而proc则不会如此。

```
#Ruby 匿名自执行函数

lambda{ |x| puts x + 1 }.call(2)      #3
Proc.new { |x| puts x + 1 }.call(2)      #3
```

上面代码定义了一个匿名的lambda，并且通过call（等同于上面JavaScript代码中的`()`）执行了这个lambda。第二个例子则创建了一个匿名的Proc对象并执行。

因为proc和lambda都是对象，所以是可以赋值给其他变量的。

```
#Ruby lambda作为参数传递

a = lambda { |x| puts x*2 }
[1, 2, 3].each(&a)      #2 4 6
```

上面代码中 `&` 运算符是将`proc`对象转为`block`，相当于把代码段 `puts x*2` 带入到`each`中，当然这样的写法就不能算作纯粹的匿名函数了，因为我们给`lambda`指定了变量名`a`。

```
#Ruby 匿名函数作为参数传递

[1, 2, 3].each(&lambda { |x| puts x*2 })      #2 4 6
```

纯函数

纯函数是指没有副作用的函数，所谓副作用是指会对外界变量、状态产生影响的语句，也就是说纯函数不会改变函数外变量的值，因此纯函数对相同的参数输入会返回相同的计算结果。

这种禁止修改外部状态的策略，也是函数式语言的一种策略，对于纯函数式语言，所有的函数必须都是纯函数（比如Haskell），正因为纯函数语言不存在可变状态，使其在并发编程下具备绝对的优势。

我们来通过Ruby的`reverse`和`reverse!`对比下纯函数和非纯函数的区别。

```
#Ruby reverse函数

a = [1, 2, 3]

puts a.reverse      #[3,2,1]
puts a            #[1,2,3]
```

```
#Ruby reverse!函数
a = [1, 2, 3]

puts a.reverse!    #[3,2,1]
puts a            #[3,2,1]
```

上面的例子中`reverse`就是一个纯函数，`reverse`翻转一个数组后不会改变原数组，而`reverse!`则相反，它会修改原数组，所以`reverse!`不是一个纯函数，包括上面函数分类用的`sort`和`sort!`也是如此的，Ruby本身提供的函数是纯函数，根据需求你也可以用相应的`!`函数改变源数据。对比Java（Java的`reverse`是破坏性的）和Haskell，Ruby提供了更多的可选条件。

谓词函数

谓词函数是值返回值为布尔类型的函数，一般常在函数式语言中使用来配合构成高阶函数。

```
//JavaScript 谓词函数

var isString = function(s){ return typeof s === "string" };

isString("it's a string");      //true
isString(1);                  //false
```

上述代码定义一个字符串判断函数，如果输入参数为字符串则返回true，否则返回false。

同样还是特别介绍下Ruby的谓词函数

```
#Ruby 谓词函数

"it's a string".is_a? String      #true
1.is_a? String                      #false
```

函数`is_a?`用来判断参数类型是否匹配，匹配则返回true，否则返回false，注意观察会发现这个谓词函数后面跟着一个`?`，和破坏性函数类型，Ruby用`?`来特别标识一个函数为谓词函数。

因此如果我们用Ruby来定义一个谓词函数时按照规范应该在函数名后标识一个`?`。

```
#Ruby 定义谓词函数

def is_string? s
  s.is_a? String
end

is_string? "it's a string"      #true
is_string? 1                   #false
```

高阶函数

高阶函数是一种参数、返回值为函数的函数。高级函数作为函数式编程中最为基础和根本的内容，我会在后续章节中用一节的内容详细介绍，现在你只需要记得高阶函数可以把其他函数作为参数传入，也可以指定一个函数作为计算结果返回。

类、类型类

结构体

在说明类之前我想先聊一聊结构体。

结构体在C语言中是用来打包存储多种数据类型的数据类型，它可以在一块内存区存储多种变量，并通过一个名字来访问这些变量。

```
//C 定义一个结构体

struct account {
    int account_number;
    char *first_name;
    char *last_name;
    float balance;
};

struct account s;
```

简单的说，结构体就是一种可以包含多种数据类型的数据类型。

类和结构体类似，也是打包不同数据类型的数据类型，类通常由属性和方法构成，其中属性用来存储变量，方法则是函数操作。

对象及其实例化

实例和对象

如上所述类其实就是一系列数据类型的封装，其本身也是一种数据类型。通过类声明的一个变量就叫做对象或者实例。

```
//Java

int a;          //声明了一个Int类型的变量a, a是一个int类型的变量
Pet dog;       //声明了一个Pet类的对象dog, dog是Pet类的一个实例
```

new运算符

`new`操作符会为对象分配一块内存并返回这个内存的引用，同时它也会调用构造函数，我们把这个过程叫做实例化。接下来构造函数会对对象的属性进行初始化赋值，我们把这个过程叫做初始化。

```
//Java 实例化类并初始化对象

Pet cat = new Pet(); //实例化cat对象
//注意Java中构造函数与类名相同，因此前一个Pet是对象的类型，后一个Pet是用来初始化对象的构造函数的
名称。
```

创建一个对象

综上所述创建一个对象包括三个步骤

1. 声明一个类的对象
2. 通过new操作符实例化对象
3. 通过这个类的构造函数初始化对象

类与对象的关系

其实对象就是一系列状态和行为的打包，比如说铅笔有笔芯和笔杆（对象的状态），铅笔可以写字、画画（对象的行为）。而类是对象的模板，比如说铅笔有笔芯笔杆，可以写作绘画，油性笔也有笔芯笔杆，也可以写作绘画，像这样的有笔芯有笔杆可以写作绘画的东西就叫做笔。那么笔就是模板，铅笔、油性笔就是对象，而这个模板所打包的状态就是笔芯、笔杆，打包的动作就是写作、绘画。

```
//Java 作为笔的类

class Pen {
    public Refill r;
    public Shaft s;
    public void write(){}
    public void draw(){}
}
```

有了类与变量之间关系的认识，就可以更书面的说明类的定义了：

类是一个模板，它描述一类对象的行为和状态。

类和变量

通常在类中有三种变量：局部变量、实例变量和类变量。

实例变量和类变量在实例函数和类函数中已经有过说明，可以通过类直接访问的变量就是类变量，一般用static关键词修饰。通过对对象访问的变量是实例变量。

局部变量是指那些在函数中定义的变量，是无法通过类、对象直接访问的。

```
//Java 类变量、实例变量和局部变量

class Demo{
    public static String classVariable = "类变量";
    public String instanceVariable = "实例变量";

    public static void(String args[]){
        String localVariable = "局部变量";
    }
}
```

更多关于类的概念我会在后续章节面向对象特点下详细说明。

鸭子类型

在说明类型类之前，我想先介绍下Ruby语言中提出的鸭子类型概念，这样方便我们区分联系类及类型类之间的关系。

当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。

鸭子类型中关注的不是对象本身是什么，而是其表现出的行为，也就是它的使用场景。还是以之前的笔为例子说明，我们用木枝在沙滩上写字绘画，很明显木枝并不是笔的一种，但是它能作为笔来使用，实现写的功能，那么这是我们就可以把木枝称作为“笔”。这就是鸭子类型的特点，不注重对象本身，而是对象其行为。

有了鸭子类型的概念我们继续下面的类型类。

类型类

如果说类是Java中实现复用的重要手段，那么类型类就是Haskell中实现复用的重要手段。

类型类和鸭子类型的思维方式类似，它也强调行为功能。通常类型类描述一些列的行为，属于这个类型类的类型就实现了其定义的行为（你也可以把类型类理解为Java的接口，它是描述行为的一种抽象）。

我们来看下Haskell中函数(>)的定义：

```
-- Haskell (>) 定义

(>):: (Ord a)=> a -> a -> Bool
```

这里Ord就是一个类型类，属于Ord的类型a就具有排序的行为，这个定义说明传入两个a类型，a类型具有比较行为，进行比较运算后返回一个布尔类型。

在Haskell中除了Ord还有诸如Enum、Num、Bounded等类型类，它们分别具备成员是否连续、是否为数字、成员的上下限等行为。类型具备了类型类的行为才能实现出相等、比较、字符串转换这些运算。

类、类型类、类型及类类型

这些概念放在一起确实有些容易混淆，所以我想在最后简单区分下。

类和类型类已经说明过了，类是面向对象程序中非常重要的一个概念，而类型类是Haskell语言中抽象行为的一种方式。类型就是我们这两章中在说明的概念，那么类类型是什么，很简单，就是“类 类型”。如果把类看做是声明的变量，那么类类型就是对应的类型，就像 `int a`，如果a是一个类，那么int就像类类型。

第四章 从算法聊编程语言的划分

和第一章相对，程序由数据和算法组成，第一章从数据类型方面对编程语言进行了对比，这一章将从算法方面对编程语言进行对比。当然所说的算法并非计算机算法课程里所说的数学相关的逻辑知识，就像程序通过计算机的数据类型来表达数据的概念，也通过范式表达算法的概念，这里所说的算法就是编程范式。

1. 命令式编程范式相关概念及子范式。
2. 声明式编程范式相关概念及子范式。

命令式编程

命令式范式侧重于描述程序的执行过程，它由一系列的命令组成，通过语句 (statements) 改变程序中的状态 (status) 。

```
if(x > 0)  y = x;  
else        y = -x;
```

语句 (statements)

语句分为基本语句和复合语句两种，其中基本语句包括：断言 (assertion) 、赋值 (assignment) 、跳转 (goto) 、返回 (return) 、调用 (call) 。复合语句又包括区块 (block) 、循环 (do-loop, for-loop, while-loop) 条件 (if, if-else) 、分支 (switch) 、伴随 (with) 、异常 (try-catch) 。

基本语句

基本语句通常用来声明、赋值、返回变量，执行函数。

赋值

赋值语句用于对变量进行赋值操作。

```
//Java 变量赋值  
int a = 2;
```

返回

返回语句用于返回计算结果。

```
//Java 函数返回值  
public static int add(int a, int b){ return a + b;}
```

```
//JavaScript 函数返回值  
function abs(a){ return a > 0 ? a : -a;}
```

断言

断言一般用来判断表达式结果的真假。

```
//Java 断言  
  
assert(1 > 0);System.out.println("1 large than 0");  
//1 large than 0  
  
assert(1 < 0);System.out.println("1 small than 0");  
//Exception in thread "main" java.lang.AssertionError
```

也有些测试库用断言来比较期望值与实际值是否相等。

```
//Java junit断言  
assertEquals(2, 1+1); //true
```

```
//JavaScript jasmine断言  
expect(1+1).toBe(2); //true
```

调用

调用一般用于执行定义的函数。

```
add(1, 3);  
abs(-4)
```

跳转

跳转可以使程序直接到指定标记处继续执行，正因为如此goto语句使程序难以维护和理解，所以大多数语言在设计层面已经舍弃了goto，即使在保留了goto语句的高级语言（C、C++）中也不建议使用。

```
//C goto语句  
  
int n = 0;  
loop: if(n < 3){ n++; goto loop;}
```

复合语句

命令式编程中最具代表性的语句就是条件、循环和分支。这些语句负责程序的逻辑控制。

条件

if: 如果表达式成立，则执行语句。

```
if([表达式]) [语句]
```

```
//Java 条件判断  
if(x == 1) System.out.println("one"); //one
```

if-else if: 依次判断表达式，如果表达式n成立则执行语句n，结束判断。

```
if([表达式1]) [语句1]  
else if([表达式2]) [语句2]  
else if([表达式3]) [语句3]
```

```
int x = 2;  
if(x == 1) System.out.println("one");  
else if(x == 2) System.out.println("x is two");  
else if(x == 2) System.out.println("x is 2");  
//x is two
```

if-else: 如果表达式成立，则执行语句1，否则执行语句2。

```
if([表达式]) [语句1]  
else [语句2]
```

```
int x = 1;  
if(x == 1) System.out.println("x is 1");  
else System.out.println("x is not 1");  
//x is 1
```

循环

for: 执行初始化表达式，然后执行条件判断表达式，如果成立则执行语句和增量表达式并重复之前的执行过程，如果不成立则结束循环。

```
for([初始化表达式]; [条件判断表达式]; [增量表达式]) [语句]
```

```
//Java for循环
```

```
for(int i=0; i<=2; i++)  
    System.out.print(i+1);      //123
```

while: 执行条件判断表达式，如果满足则执行语句。

```
while([条件判断表达式]) [语句]
```

```
int i = 1;  
while(i<=3){  
    System.out.print(i);  
    i+=1;  
}  
//123
```

do-while: 首先执行语句，然后执行条件判断表达式，如果成立则重复上述操作，否则结束循环。

```
int i = 1;  
do{  
    System.out.println(i);  
    i++;  
}while(i<=3);  
//123
```

分支

分支语句和条件分支功能类似，部分语言对switch语句有优化，所以多路分支时switch效率相对要高，不过switch语句不易维护，所以出于这方面考虑建议用if-else if语句替换，大多数情况下switch和if-else if可以相互替换，但是对于不可列举的情况下只能用if-else if。

switch: 依次判断变量表达式和常量判断表达式n是否相等，如果相等则执行语句n，否则执行默认分支语句。

```
switch([变量表达式]){
    case [常量判断表达式1]
        [语句1]
        break;
    case [常量判断表达式2]
        [语句2]
        break;
    case [常量判断表达式3]
        [语句3]
        break;
    default
        [默认分支语句]
}
```

```
int a = 1;
switch(a){
    case 1:
        System.out.println(1);
        break;
    case 2:
        System.out.println(2);
        break;
    case 3:
        System.out.println(3);
        break;
    default:
        System.out.println("other number");
        break;
}
//1
```

异常

异常处理的作用是在程序正常运行中对不合法的输入进行控制和提示，防止程序崩溃。

try-catch-finally: 执行异常语句如果发现异常则执行异常处理语句，最后无论是否存在异常都会执行终止语句。

```
try { [异常语句]; }
catch{ [异常处理语句]; }
finally{ [终止语句]; }
```

```
try{
    int a = 1/0;
}catch(Exception e){
    System.out.println(e.toString());
    //java.lang.ArithmetricException: / by zero
}finally{
    System.out.println("finally");
    //finally
}
```

命令式语言发展历程

机器语言

机器语言是由计算机直接执行的语言，它们一般由八进制或者十六进制的字节码组成，这种代码只能在指定的机器上运行，因此很难移植。

```
4944 3303 0000 001f 1c09 5453 5345 0000
000d 0000 004c 6176 6635 362e 342e 3130
```

1960年汇编语言被开发出来，汇编语言提供了一些标签和符号，通过汇编语言可以编写出可读性更高的源代码，然后将源代码编译为机器可执行的字节码。

汇编语言的出现，使程序的编写得到了极大的便利。

```
;汇编语言
INC COUNT          ;计数器COUNT加1
MOV TOTAL, 48      ;变量TOTAL赋值为48
ADD AH, BH         ;将寄存器AH和BH相加
```

可以看出除了复杂的语言特性外，从汇编语言开始就有了类似变量赋值、运算的概念。

过程式语言

过程式语言按照一定的流程顺序执行，在执行过程中可以调用其他程序或函数。其主要用到的语句有流程控制语句(条件、分支、循环)和函数调用。

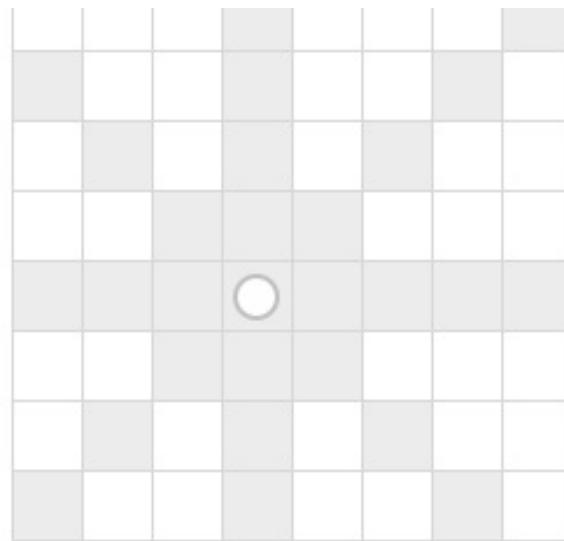
这里我们用C语言来看下过程式语言是如何解决八皇后问题的。

八皇后问题最早是由国际象棋棋手马克斯·贝瑟尔（Max Bezzel）于1848年提出，问题在于如何在一个8×8的国际象棋棋盘上放置八个皇后，使得任何一个皇后都无法直接吃掉其他的皇后（任两个皇后都不能处于同一条横行、纵行或斜线）。



如上图我们在棋盘第四列第五行位置放置了一个皇后，按照八皇后的游戏规则，图中所有灰色部分都是不允许放置皇后的，为了方便说明我们把这些区域称作不可置位置；相反所有白色区域是允许放置皇后的，我们称作可置位置。然后我们用坐标

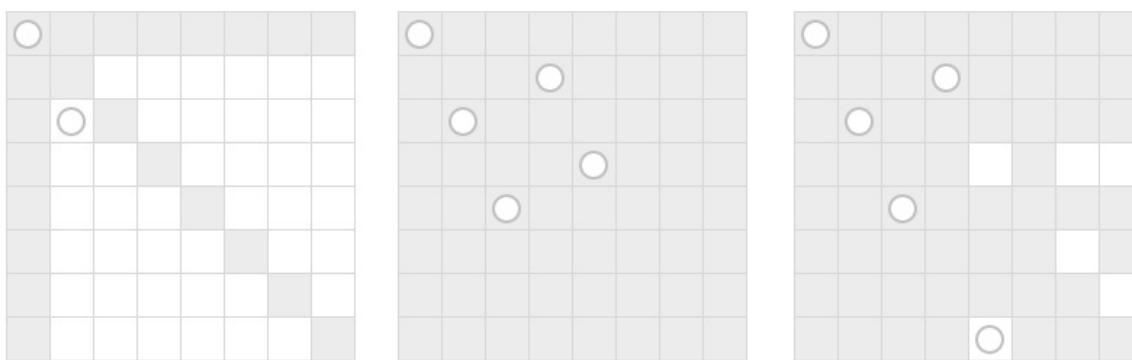
(4, 3) 来表示皇后的位置，并规定从左到右依次为首列、第一列、第二列、第三列...从上到下依次为首行、第一行、第二行...



有了如上约定后，我们就可以介绍如何通过回溯法获得八皇后的结果集了。

1. 从首列按首行到第七行的顺序依次放置一个皇后，每当放置一个皇后执行下一步。
2. 在第一列中选择第一个可置区域放置一个皇后（行坐标最小的那个），执行下一步。
3. 跳转到下一列，查找最小行标的可置区域放置皇后，如果不存在可置区域，则回溯到上一行，重置上一行的皇后到下一个最小行标的可置区域。
4. 重复第三步，直到到达第七列在可置区域放置皇后，此时为一个有效解。
5. 将有效的皇后位置记录，输出。

我们配合下面这张图来说明下回溯的过程：



1. 当我们在(0, 0)位置放置一个皇后后，第一列最小行标的可置区域为(1, 2)，在此放置皇后。
2. 重复上述流程中的第四步操作，当第五列时发现无可置区域，回溯到第四列。
3. 回溯到第四列选择下一个最小行标的可置区域(4, 7)后，放置皇后，继续下一列。

接下来我们看看过程式的C代码是如何描述上述执行过程的：

```
//C 八皇后问题

#include <stdio.h>
int is_safe(int rows[8], int x, int y)
{
    int i;
    if (y == 0)    return 1;
    for (i=0; i < y; ++i) {
        if (rows[i] == x || rows[i] == x + y - i || rows[i] == x - y +i)
            return 0;
    }
    return 1;
}

void putboard(int rows[8])
{
    static int s = 0;
    int x, y;
    printf("\nresult %d:\n", ++s);
    for (y=0; y < 8; ++y) {
        for (x=0; x < 8; ++x)
            printf(x == rows[y] ? "| o " : "|   ");
        printf("|\n");
    }
}

void eight_queens(int rows[8], int y)
{
    int x;
    for (x=0; x < 8; ++x) {
        if(is_safe(rows, x, y)) {
            rows[y] = x;
            if(y == 7)    putboard(rows);
            else         eight_queens(rows, y+1);
        }
    }
}

int main()
{
    int rows[8];
    eight_queens(rows, 0);
    return 0;
}
```

这里简单说明下这段代码。

1. 先看主函数main，这里声明了一个数组rows，用来描述从首行到第七行。

2. main函数中调用了eight_queens函数，传入了rows和0，rows上面已经说过了，用来描述行。0表示从第一列开始插入皇后到可置区域。
3. 在eight_queens函数中x表示当前列下行坐标，is_safe函数返回当前坐标(x, y)是否为可置区域。如果是可置区域则放置皇后并记录皇后坐标 `rows[y] = x`，这里为了节约内存，用了一维数组来记录，数组的索引用来记录y坐标，值用来记录横坐标，则皇后坐标实际为 `(rows[y], y)`。
4. 然后我们来看看is_safe函数，这个函数会检查当前这个位置（坐标）是否会与之前列下的皇后同行、同对角、反对角线（即判断这个位置的横坐标rows[i]是否会与之前任何一个皇后的横坐标x，对角线横坐标 $x + y - i$ ，斜对角线横坐标 $x - y + i$ 相等）。如果有相等则说明该坐标为不可置区域，返回0，否则说明是可置区域返回1。
5. 最后回到eight_queens，当列坐标y=7且is_safe返回1时，说明回溯算法走到最后一列并且找到了一个可置区域放置了皇后，那么这个路径就是个有效解，执行putboard输出所有皇后的坐标。我们通过rows[y] = x来判断的这个位置是不是皇后，是的话输出o来模拟棋子。

来看一看输出的结果：

```
result 1:  
| o | | | | | | | | |
| | | | | o | | | |  
| | | | | | | | o | |  
| | | | | | o | | | |  
| | | | o | | | | | |  
| | | | | | | | o | |  
| | | | | o | | | | |  
  
result 2:  
| o | | | | | | | | |
| | | | | | o | | | |  
| | | | | | | | o | |  
| | | | | | o | | | |  
| | | | | o | | | | |  
| | | | | | o | | | |  
| | | | | | | o | | |  
| | | | | | | | o | |  
  
...  
result 92:  
| | | | | | | | o |  
| | | | | o | | | |  
| o | | | | | | | |  
| | | | o | | | | |  
| | | | | o | | | |  
| | | | | | o | | |  
| | | | | | | o | |  
| | | | | | | | o |
```

八皇后问题有92个解，这里就不一一列举了。

面向对象语言

在1980年左右，在命令式范式的基础上追加了对象特性的语言开始快速发展起来，这就是面向对象语言。

1. 1980年Smalltalk
2. 1985年C++
3. 1987年Perl
4. 1990年Python
5. 1991年Visual C++
6. 1994年PHP和Java
7. 1995年Ruby
8. 2002年.NET Framework(C#、VB.NET)

面向对象语言将数据和方法打包到一个结构中，通常称作为对象。对象内的数据和方法只有对象本身可以访问。程序不再由流程式的函数调用组成，而是通过对象及对象间的关系构成。

这里用C++来重复上面的八皇后问题，对比下面向对象和过程式两种范式的区别。

```
//C++ 八皇后问题

#include <iostream>
using namespace std;

class EightQueens{
    int rows[8];
public:
    bool IsSafe(int x, int y){
        int i;
        if (y == 0)    return 1;
        for (i=0; i < y; ++i) {
            if (rows[i] == x || rows[i] == x + y - i || rows[i] == x - y +i)
                return false;
        }
        return true;
    }

    void PutBoard(){
        static int s = 0;
        int x, y;
        cout << "\nresult" << ++s << ":\n";
        for (y=0; y < 8; ++y) {
            for (x=0; x < 8; ++x)
                x == rows[y] ? cout << "| o " : cout << "|   ";
            cout << "|\\n";
        }
    }

    void Solution(int y){
        int x;
        for (x=0; x < 8; ++x) {
            if (IsSafe(x, y)) {
                rows[y] = x;
                if(y == 7)    PutBoard();
                else          Solution(y+1);
            }
        }
    }
};

int main()
{
    EightQueens q;
    q.Solution(0);
}
```

这里整体逻辑和C语言实现是相似的，唯一不同的在于代码的组织方式，像C++这种面向对象语言，我们可以把数据与函数打包到对象中管理，在这里我们将八皇后问题所需要用到的方法和数据都打包到EightQueens对象中，对比C语言你会发现所有函数都不需要显示传递参数rows了。rows作为EightQueens对象的属性允许对象内的任何函数直接访问。最后声明了EightQueens的一个实例q，然后通过对象q调用Solution函数，传入起始行参数0来查询八皇后问题的有效解。

对比过程式语言，面向对象语言更侧重于对数据与函数的封装，以及封装后对象与对象之间关系的处理。后面我会用一个章节来详细说明面向对象语言的特点，这里就不过多解释了。

声明式编程

声明式范式的编程风格是只描述问题的逻辑关系，而不关系具体解决过程。通常声明式程序由一系列的逻辑条件组成，程序执行时会找出满足这些逻辑的情况作为结果返回。简单的说声明式编程解决问题的思路是通过陈述（声明）语句描述问题是什么而不是通过流程控制语句描述问题该如何解决。这样做的好处能尽可能的减少副作用，甚至无副作用，因此声明式编程在处理并行问题时更有优势。

满足声明式编程风格的范式或者说子范式（我个人觉得范式之间没有明确的继承关系，而是互相穿插的）包括如下：

1. 约束编程
2. 领域专属语言
3. 逻辑编程
4. 函数式编程

在介绍声明式编程的子范式之前，我想先介绍一些声明式及其子范式下的编程技巧（或者说是编程风格）。

声明式编程技巧

列表推导式

声明式编程中常用的数据结构为列表和元组，通常一个列表作为输入，输出一个变化后的新列表。列表推导式也是这样的技巧，将一个规则约束于原列表，推导出一个满足这个规则的新列表。

```
-- Haskell 列表推导  
[ x * 2 | x <- [1,2,3,4], x > 2]      -- [6,8]
```

列表推导式语法和数学中集合推导式是一致的，`{x * 2 | x ∈ N, 2 < x < 5}` 表示取2到5之间的正整数乘2。同理，上述代码表示取集合中满足 $x > 2$ 的元素做 $x * 2$ 运算，然后将结果集打包成一个新列表 `([6, 8])` 返回。

之前强调过声明式语言的编程特点是不描述具体执行过程，只描述具体规则，由机器推导运算结果。那么应用到列表推导式是如何表现的呢？

我们来看这个例子，求解三边长度都在20以内的直角三角形各边长分别为多少。

```
-- Haskell 通过列表推导式过滤出符合条件的三角形

[(a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b], c^2 == b^2 + a^2]
-- [(3,4,5), (6,8,10), (5,12,13), (9,12,15), (8,15,17), (12,16,20)]
```

上面代码为了排除重复的三角形，限制了三边长度关系 `a < b < c`。同样如果我们想在上述条件不变的情况下，取出三边长度总和在10到25之间的三角形，只需要追加约束条件 `a+b+c > 10, a+b+c < 25`。

```
-- Haskell 通过列表推导式过滤出符合条件的三角形

[(a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b], c^2 == b^2 + a^2, a+b+c > 10, a
+b+c < 25]
-- [(3,4,5), (6,8,10)]
```

模式匹配

命令式编程中用来控制分支逻辑的语法有`case`和`if`两种，因为`case`难以阅读，所以多数情况下使用`if-else`分支结构，对于多种情况就会出现大量的`if-else`分支，形成一个树一样的结构。

声明式编程中通过模式匹配解决了分支难以阅读的问题，模式匹配通过检查数据结构与条件匹配来决定是否执行相应操作。下面通过一个简单的例子来看看模式匹配是如何工作的。

```
--Haskell 对输入的阿拉伯数字进行匹配返回对应的英文

numberToEnglish:: Int->String
numberToEnglish 1 = "one"
numberToEnglish 2 = "two"
numberToEnglish 3 = "three"
numberToEnglish 4 = "four"
numberToEnglish 5 = "five"
numberToEnglish 6 = "six"
numberToEnglish 7 = "seven"
numberToEnglish 8 = "eight"
numberToEnglish 9 = "nine"
numberToEnglish 0 = "zero"
numberToEnglish _ = "not between 0 to 9"

numberToEnglish 4      --four

numberToEnglish 12     --not between 0 to 9
```

这里我们定义了函数numberToEnglish并规定了输入类型Int及返回类型String，然后按顺序定义了十个模式，分别对参数为0-10的情况定义了处理逻辑，最后用`_`表示任意参数，相当于default或else。

我们以参数param = 4为例来说明执行过程：

1. 首先会匹配第一个模式param = 1，不符合条件，则匹配下一个模式param = 2，
2. 重复1，当匹配到param = 4时，匹配成功，则返回"four"。

同理param = 12对前面10个模式都不匹配，所以执行param = `_` 返回"not between 0 to 9"。

哨兵 (Guard)

和模式匹配一样，哨兵也是声明式语言中的分支逻辑，如果一定要对比的话，模式匹配有点像case而哨兵更像if-else，每一个哨兵都是一个布尔表达式，如果表达式值为true，就会执行对应的操作。值为false则对下一个哨兵求值。

我们通过BMI（体重指数）计算来看下哨兵是如何工作的。

BMI指数（Body Mass Index），即身体质量指数，是用体重公斤数除以身高米数平方得出的数字，是目前国际上常用的衡量人体胖瘦程度以及是否健康的一个标准。

```
--Haskell 计算体质数

getBMI :: Double->String
getBMI bmi
| bmi <= 18.5 = "underweight"
| bmi <= 25.0 = "normal"
| bmi <= 30.0 = "overweight"
| otherwise = "obese"

getBMI 27      --overweight
```

我们以**bmi = 27**为例来说明：

1. 首先计算第一个哨兵 $27 \leq 18.5$ 计算结果为false，则计算下一个哨兵。
2. 当计算第三个哨兵 $27 \leq 30.0$ 时，计算结果为true，则执行对应操作返回字符串"overweight"。
3. 最后的otherwise和模式匹配里的_作用类似相当于命令式语言中的else。

合一、绑定

合一和绑定都是赋值操作，我们先看看 prolog 的合一。

```
%Prolog 合一

combine(X, Y, Z) :- X = 1, Y = 2, Z = 3.

combine(1, 2, 3).    %yes
```

`combine`对元组(X, Y, Z)执行了合一操作，这时元组中满足 $X = 1, Y = 2, Z = 3$ ，因此执行 `combine(1, 2, 3).` 返回yes。

好，我们了解了合一的语法后，接下来看看如何用haskell的合一和绑定来重写上面的 `getBMI`函数。

```
--Haskell 通过绑定重构getBMI

getBMI :: Double->Double->String
getBMI weight height
| bmi <= normal      = "underweight"
| bmi <= overweight  = "normal"
| bmi <= obese        = "overweight"
| otherwise            = "boese"
where bmi = weight/height^2
      (normal, overweight, obese) = (18.5, 25.0, 30.0)

getBMI 70 1.72    --normal
```

在讲解这段代码之前我们先说下**where**的作用：**where**相当于指令式语言里的赋值操作，它可以记录一个操作结果，方便多次引用。其作用域限定在当前函数定义内。

接下来我们来看看**getBMI**函数，与之前不同，它传入了两个参数：**weight**和**height**，用来计算BMI。然后用**where**将**weight/height^2**的计算结果绑定到**bmi**，以方便多次使用。同时用合一对元组(normal, overweight, obese)进行赋值，用来作为BMI参数的临界值。

满足声明式风格的编程范式

接下来我会简单介绍下约束编程和领域专属语言，然后相对详细的说明下逻辑编程，至于函数式编程，我会拿出一章来详细说明，和面向对象一样也不详细解释了。

约束编程

约束编程规定了变量之间的一种约束关系，它不强调具体要执行哪一步计算，只是规定了变量的一些属性。简单的说，就是数学中方程式的概念，约束只定义了方程式的定义域，并没有指明如何求解具体解，但解必须是满足这个域的。

比如以 $y = x + 1, x \in \{1, 2\}$ 为例，依照这个方程式的约束我们可以得知y的值域是 {2,3}。

约束式编程一般作为其他范式的一种补充，我们来看看在基于逻辑范式的prolog和基于函数范式的haskell下是如何表现的：

```
%Prolog 方程式求解

equation(X, Y) :- X == 1, Y is X + 1.
equation(X, Y) :- X == 2, Y is X + 1.
```

执行：

```
equation(1, Y).      %Y = 2
equation(3, Y).      %no
```

这里定义了一个推断equation，当断言 $x == 1$ 成立时，则执行is运算将Y绑定为 $X+1$ ，Prolog通过断言和推断返回查询结果 $Y=2$ 。如果断言不成立，程序无法查询到Y值，则返回no。

```
-- Haskell 方程式求解
```

```
equation x
| x == 1 = x + 1
| x == 2 = x + 1
| otherwise = error "no"
```

执行：

```
equation 1      --2
equation 3      --Exception: no
```

这里定义了一个函数equation，对传入参数x依次匹配各个模式，如果满足当前模式则返回操作结果，否则执行下一个模式。当传入参数为1时，满足第一个模式返回 $x+1=2$ ，当传入参数为3时，前两个模式都不匹配，因此执行otherwise抛出一个异常。

领域专属语言

领域专属语言（DSLs：Domain-specific languages）是针对某一特定问题而设计的语言，常见的领域专属语言如正则表达式、结构化查询语言（linq）、标记语言（html）。

应用场景：

1. 作为命令行工具或者编译程序的标准用户输入接口，比如grep的正则表达式匹配。
2. 通过编程语言的宏机制实现内部DSL拓展语言的表述能力，比如Lisp的macro，Ruby的DSL。
3. 作为某语言的内置库，增强其表达能力，比如微软的Linq。
4. 用通用编程语言解决一个特定的问题，嵌入到宿主程序中，比如用perl实现一个正则引擎。

逻辑编程

逻辑编程由三个重要部分组成，分别是：断言、推断和查询，断言用来描述客观事实；推断是针对断言作出的推测；查询就是要解决的一个问题。一般逻辑式语言通过断言和推断描述出问题是什么，然后通过查询语句将问题交给程序运算，程序会根据断言和推断计算

出符合查询的结果。

谈到逻辑式编程，最具代表性的语言就是prolog，因此接下来的例子都用prolog来说明。

我们用断言和推断来实现一个等量代换的例子，在数学中等量代换是指如果 $a = b$, $b = c$, 那么 $a = c$ 。这里断言是 $a = b$ 和 $b = c$, 推断是 $a = c$ 。那么用prolog来实现断言和推断则如下：

```
equal(a, b).  
equal(c, b).  
  
equalToo(X, Y) :- equal(X, Z), equal(Y, Z).
```

其中前两行是断言描述了 $a=b$, $c=b$, 最后一行推断表示： $a = c$ 的前提条件是： $a = b$ 且 $c = b$ 。

我们运行这段程序，然后执行如下两个查询：

```
equalToo(a, c). %yes
```

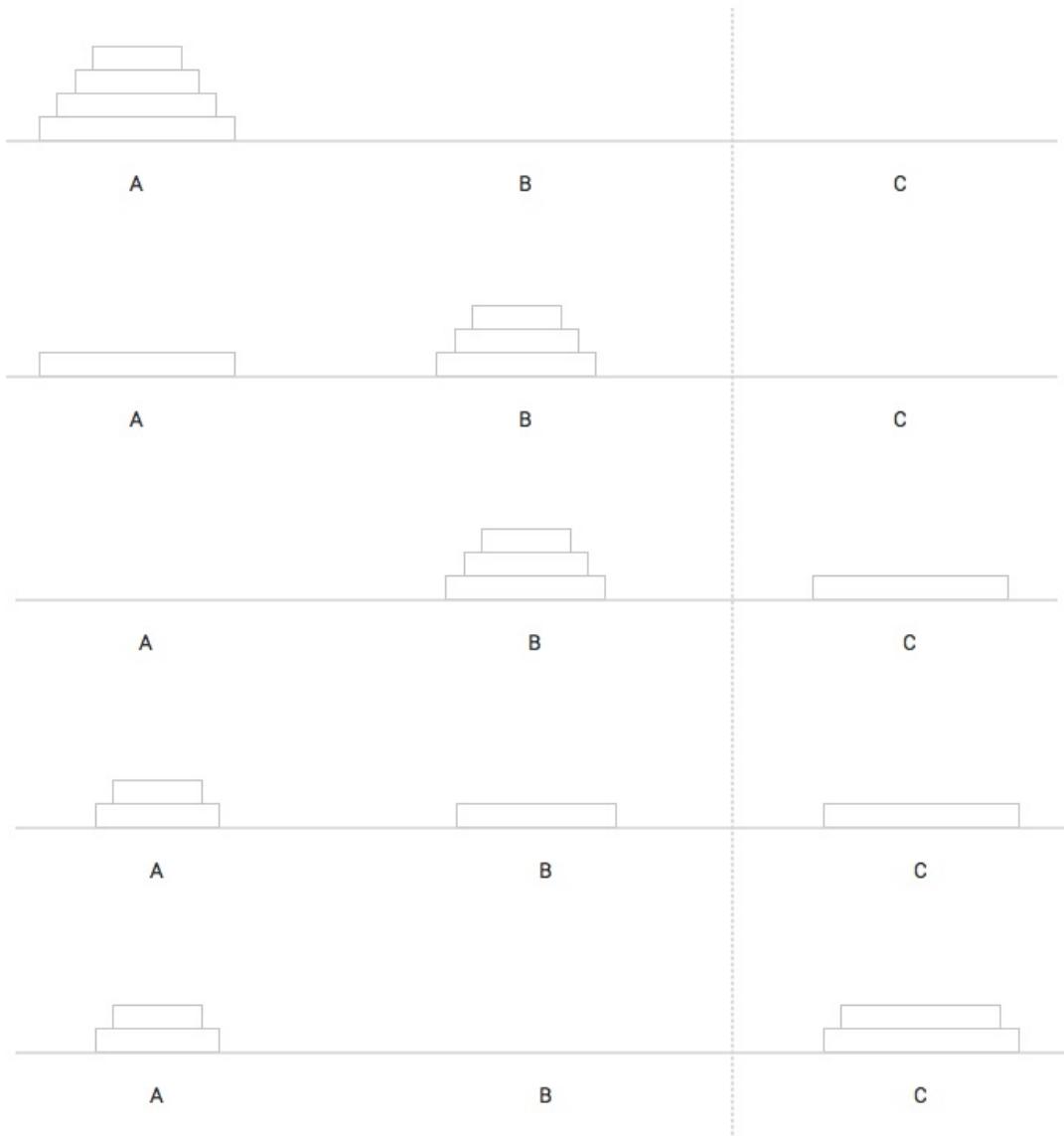
接下来看看prolog是如何解决汉诺塔问题的

汉诺塔是法国数学家爱德华·卢卡斯提出的一个数学问题，传说印度某间寺院有三根柱子，柱子上串有64个金盘。寺院里的僧侣依照一个古老的预言，以一定的规则来移动这些盘子；预言说当这些盘子移动完毕，世界就会灭亡。

汉诺塔的游戏规则如下：

1. 有三根杆子A, B, C。A杆上有N个 ($N > 1$) 穿孔圆盘，盘的尺寸由下到上依次变小。
2. 要求按3、4的规则将所有圆盘移至到C杆。
3. 每次只能移动一个圆盘。
4. 大盘不能叠在小盘上面。

在说明具体实现代码之前，我们先理清汉诺塔的解决思路。



上图可以看出我们将A中圆盘移动到C的过程中有一个关键的状态点：一个柱子上按顺序叠放着 $n-1$ 个圆盘，而另一个柱子上放着当前最大的圆盘（抛开C不管，只看柱子A、B）。

在这个状态点我们把最大的圆盘移至到C，就恢复到了最初的状态：一个柱子上按顺序叠放着 n 个圆盘，而另一个柱子上没有圆盘。实际上要把圆盘按顺序叠放到C柱子，我们只需要不断的重复操作达到这两个状态点就可以了。

因为声明式语言不关心具体的操作步骤，所以我们根本不需要考虑通过怎样的移动可以达到这两个状态，我们只需要强调这两种状态的产生时机就可以了。

```
%Prolog 汉诺塔问题
```

```
move(1, A, B, C) :-  
    write('move top disk from ' ),  
    write(A),  
    write(' to ' ),  
    write(C),  
    nl.  
move(N, A, B, C) :-  
    N > 1,  
    M is N-1,  
    move(M, A, C, B),  
    move(1, A, B, C),  
    move(M, B, A, C).
```

我们定义了一个移动函数，第一个参数表示当前存在的圆盘数，其余A、B、C参数分别表示图示中的三根圆柱，A，作为起始圆柱，B作为辅助圆柱，C作为终止圆柱。

1. 当只存在一个圆盘时，我们可以直接把圆盘从起始圆柱A移动到终止圆柱C。
2. 当存在一个以上（N个）圆盘时，我们先把上面N-1个圆盘从起始圆柱A移动到辅助圆柱B，然后再将起始柱A最下面的圆盘移动到终止圆柱C。这时B就变成了之前的起始圆柱，而A则变成了辅助圆柱，继续重复之前的逻辑将剩余的N-1个圆盘从起始圆柱B移动到终止圆柱C，而A则作为辅助圆柱。

最后我们定义起始圆柱A、辅助圆柱B、终止圆柱C、圆盘数4，然后输出。

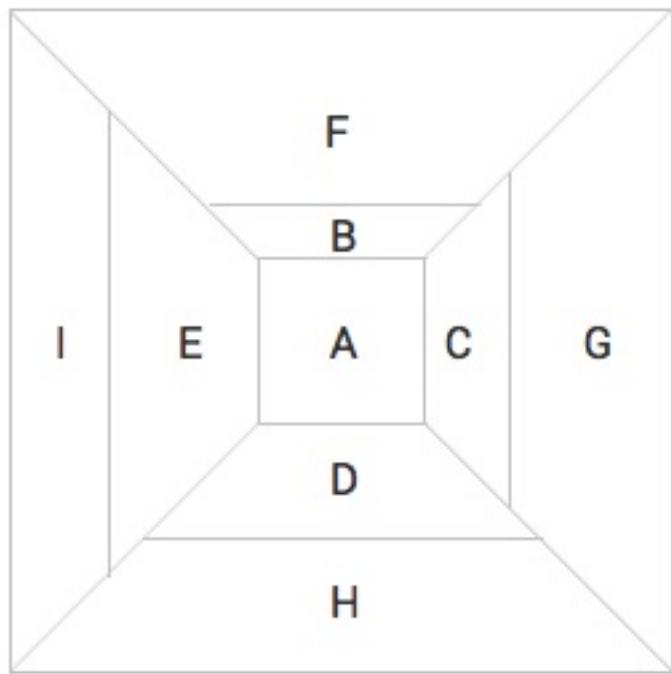
```
move(3, 'A', 'B', 'C').  
  
/* 输出  
move top disk from A to C  
move top disk from A to B  
move top disk from C to B  
move top disk from A to C  
move top disk from B to A  
move top disk from B to C  
move top disk from A to C  
*/
```

可以看到在上述代码中我们并没有关心具体细节，只需要串联关键状态，计算机会自动帮我们计算具体步骤，这就是逻辑式语言的优势。

最后我们再来看一个更加纯粹的只关心规则声明的例子：四色定理。

如果在平面上划出一些邻接的有限区域，那么可以用四种颜色来给这些区域染色，使得每两个邻接区域染的颜色都不一样。

html 图片



我们用四种颜色（红、绿、蓝、黄）对左侧图中9个区域进行染色，根据四色定理我们可以使任意相邻的两个区域都不同色。

首先列出相邻两个区域不同颜色着色的所有方案：

```
%Prolog 断言相邻区域不同色
```

```
different(red, green). different(red, blue). different(red, yellow).
different(green, red). different(green, blue). different(green, yellow).
different(blue, red). different(blue, green). different(blue, yellow).
different(yellow, red). different(yellow, green). different(yellow, blue).
```

接下来我们以从A到I的顺序确保其相邻区域都满足于着色方案：

```
%Prolog 图示相邻区域颜色不同
```

```
coloring(A, B, C, D, E, F, G, H, I) :-  
    different(A, B), different(A, C), different(A, D), different(A, E),  
    different(B, C), different(B, E), different(B, F),  
    different(C, D), different(C, F), different(C, G),  
    different(D, E), different(D, G), different(D, H),  
    different(E, F), different(E, H), different(E, I),  
    different(F, G), different(F, I),  
    different(G, H), different(H, I).
```

最后对所选11个区域着色，执行查询：

```
coloring(A, B, C, D, E, F, G, H, I).
```

得到结果：

```
A = red  
B = green  
C = blue  
D = green  
E = blue  
F = red  
G = yellow  
H = red  
I = green?  
Action (; for next solution, a for all solutions, RET to stop) ?
```

如果输入a我们还能得到其他九中涂色方案，就不一一列举了。

函数式编程

函数式语言主要特点在于其减少甚至避免了程序中的副作用，保证了函数的输入相同则输出相同。因此函数式编程适用于并行开发以及一些严格需求无副作用的业务。

副作用：在函数或者表达式中如果其计算对外部产生影响，比如全局变量的修改、参数值的修改、抛出异常或者调用有副作用的函数，那么这个函数或者表达式就是有副作用的。

函数式语言具有函数作为一等公民、引用透明、值不变等特性，它通过函数组织代码逻辑，将元组和列表作为基本数据结构，通过高阶函数强化元组和列表的功能；通过柯里化单一化函数参数；通过惰性求值优化执行性能；

这些特性和技巧我将会用专门的一章来详细说明。

为了展现函数式语言的特点，我们用haskell来实现汉诺塔

```
--Haskell 汉诺塔问题
```

```
hanoi :: Integer -> a -> a -> a -> [(a,a)]
hanoi 0 _ _ _ = []
hanoi n a b c = hanoi (n - 1) a c b ++ [(a,c)] ++ hanoi (n - 1) b a c
hanoiIO :: Integer -> IO ()
hanoiIO n = mapM_ f $ hanoi n "A" "B" "C" where
  f (x,y) = putStrLn $ "move top disk from " ++ show x ++ " to " ++ show y
```

```
hanoiIO 2
```

```
{- 输出
move top disk from "A" to "B"
move top disk from "A" to "C"
move top disk from "B" to "C"
-}
```

主要来看第三行，和之前的逻辑一样：

- 首先将前n-1个圆盘，从起始圆柱a移动到辅助圆柱b。
- 接下来将最后一个圆盘从起始圆柱a移动到终止圆柱c，然后递归这个函数，将剩下的n-1个圆盘从新的起始圆柱b借助新的起始圆柱a移动到终止圆柱。

这里因为每次移动返回的是一个元组类型的列表，代表从哪个圆柱移动到哪个圆柱，所以我们可以直接对最后一个圆盘的移动绑定为[(a,c)],也就是起始圆柱到终止圆柱。而我们在第二行中规定了当n=0时返回空，所以当n=1时函数实际执行了[]++[(a,c)]++[],其中++是haskell中列表合并运算，所以n=1本身就返回了[(a,c)]，类似的我们上面执行的n=2则返回了[(a,b), (a,c), (b,c)]。

其递归过程如下：

```
hanoi 2 a b c => hanoi 1 a c b + [(a,c)] + hanoi 1 b a c
hanoi 1 a c b => hanoi 0 a b c + [(a,b)] + hanoi 0 c a b
hanoi 1 b a c => hanoi 0 b c a + [(b,c)] + hanoi 0 a b c
hanoi 0 _ _ _ => []
-----
hanoi 2 a b c => []+[(a,b)]+[]+[(a,c)]+[]+[(b,c)]+[]
=> [(a,b), (a,c), (b,c)]
```

下面的hanoiIO只是将列表中的移动信息文本化输出，这里可以不必关心。

第五章 命令式下的面向对象概念

这里我同第一章到第二章处理方式相同，在第四章引入命令式和声明式概念后，这里将主要说明命令式范式下的面向对象语言。

1. 介绍面向对象语言中对象相关概念及对象实现的三种常见模式。
2. 介绍面向对象语言的特点，及实现相应特点引入的概念。
3. 介绍面向对象语言为处理常见问题而引入的设计模式概念。

面向对象及模式

在说明面向对象之前先简单说明下什么是对象。

对象

对象本身也是一种数据类型，所以它本质上是一段存储着值的内存地址。它与其他数据类型的区别在于它将数据和方法打包成一个整体，即对象内部包含一定的数据（值类型或引用类型）和对数据操作的方法（函数）。所以对象是涵盖多个数据类型的数据类型。

面向对象

面向对象就是通过对象将程序组织到一起，对象之间存在着相互的关系，可以访问修改关联对象的数据，这样做的好处是提高了代码的复用性、灵活性和拓展性，使程序更易于维护和拓展。

接下来我们看看对象在实现上常用的几种模式。

模式

静态类模式

实例和类：类是一系列实例的抽象，实例则是类的一个具现。

实例中通常包括属性（对象的描述）和方法（对象的行为），在静态类模式下实例所拥有的属性和方法是由类的模板定义的，而且模板一旦定义后是不允许再次修改的。

```
//伪代码 类与实例

C = class{ int:a, int:b, void:method }
c1 = { a: 10, b: 20 }
c2 = { a: 20, b: 30 }
```

类模式通过类的继承来实现代码复用，A继承自B，那么A就拥有B的属性和方法，B继承自C，那么A、B都有C的属性和方法。如果尝试取得A的一个属性/方法：a，那么首先会在A中寻找a，如果不存在就去B中寻找，如果B中不存在则去C中寻找，如果C中也不存在则A不存在这个属性/方法。

另外要说明的是如果A继承自B，那么A会把B的所有属性都拷贝一遍（即使A类可能根本不需要这个属性，但是方法不会拷贝），所以如果A中某个属性与B的同名，则会以A的这个属性为准，只有A中没有的属性才会到B中寻找。

总结

1. 需要定义一个对象的模板：类。
2. 根据模板来创建对象，对象中包含属性和方法。
3. 类定义后无法修改，对象的属性和方法依赖于类。
4. 类的继承使子类包含父类的所有属性和方法（有些语言提供了访问权限解决了这个问题）。
5. 实例无法拥有自己特有的属性和方法。

原型模式

动态和可拓展对象：没有类模板概念，通过对对象本身定义属性和方法。

```
//伪代码 对象拓展

object1 = { a: 10, b: 20 }
object2 = { a: 20, b: 30 }
```

对象可以动态的添加、修改属性和方法。如果我们尝试取得a的一个属性/方法b，a会检查是否存在b，如果存在则更新b，如果不存在则创建b。

对象模式通过原型来实现代码的复用，原型也是一个对象（原型是对象的一个属性）。任何一个对象都可以作为其他对象的原型来使用。

```
//伪代码 一个对象作为另一个对象的原型实现继承

object1 = { a: 10, b: 20, method1 }
object2 = { a: 20, c: 30, method2 }
object2.[[Prototype]] = object1

show object2
//{ a: 20, b: 20, c: 30, method1, method2 }

delete object2.a
object2.d = 40

show object2
//{ a: 10, b: 20, c: 30, d: 40, method1, method2 }
```

一个作为原型的对象本身也可以指定原型，一个对象a以b为原型，b以c为原型，则abc就构成了一个原型链，原型链的概念就和类模式的继承一样了。不同点在于，因为对象是可拓展的，所以原型链的继承关系和结构是有可能改变的。

另外除了通过指定原型属性外还有另外一种代码复用的实现，就是完全拷贝，这样做的好处是修改不会对原型对象产生影响，但相应的需要更多的内存开销。

在静态类模式下我们是通过类模板来描述对象的，比如说类模板定义了class Car，那么Car的实例很明显都是汽车，而原型模式下则是通过鸭子类型（如果一个对象表现的行为是car，那么它就是Car）来表述这一概念的，简单的说当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。在鸭子类型中关注的不是对象本身是否是鸭子，而是它在使用上是否像鸭子。

```
//伪代码 鸭子类型：如果一个对象表现出鸭子的特性，那么它就是个鸭子

object1.active == 'duck' ? 'object1 is duck' : 'object1 is not duck'
object2.active == 'duck' ? 'object2 is duck' : 'object2 is not duck'
```

总结

1. 对象本身是动态的、可拓展的。
2. 对象本身不需要模板来定义属性和方法。
3. 对象可以通过原型实现代码的复用，同样原型也是随时可以改变的。
4. 当原型发生改变时，会影响到原型链下层的对象。
5. 对象的不是通过特定的类型或继承关系定义，而是通过当前所拥有的特性（鸭子类型）定义。

动态类模型

之前的两种模式可以对比为类和模型，通常静态语言使用类模式，而动态语言常使用原型模式。当然也存在像Python或Ruby这样的语言，他们属于动态类型语言，却使用类模式。

他们通过定义类模板来规范对象的属性和方法，但又可以在运行中改变类的定义和对象的原型。我们来看下面的例子：

```
#Ruby 对类拓展属性

class Demo
  attr_accessor :a
end

d = Demo.new
d.a = 1
d.b = 2
#undefined method `b=' for #<Demo:0x007fe81a9caa68 @a=1> (NoMethodError)

# 给Demo类拓展属性b
Demo.class_eval do
  attr_accessor :b
end

d.b = 2
puts d.a, d.b
#1
#2
```

可以看出Ruby中可以通过class_eval来拓展类的定义，这种采用对象和类的方式组织数据，但却允许在程序运作过程中动态改变类的定义的模式就是动态类模式。

面向对象特点

多态

多态是指同一个消息传递给不同的对象时会产生不同的动作，比如Animal类下有duck和dog两个对象，当他们收到相同的run消息后表现出跑的动作就是不同的。

对多态有了基本的印象后，我们从如下几个方面展开说明：函数多态、变量多态、子类型多态。

函数多态 (Ad hoc polymorphism)

函数多态接受不同类型参数，并根据不同的参数类型返回不同的值。函数重载和操作重载是最常见的函数多态。

```
//Java 函数重载

class Demo{

    public static int add(int a, int b){
        return a + b;
    }

    public static String add(String a, String b){
        return a + b;
    }

    public static void main(String args[]){
        int result1 = Demo.add(1, 2);
        String result2 = Demo.add("String ", "combined!");

        System.out.println(result1);      //3
        System.out.println(result2);      //String combined!
    }
}
```

Java本身不提供运算符重载，而Ruby的操作符运算本身就是一个函数，所以Ruby中的运算符重载只需要重载函数即可。

```
#Ruby 运算符重载

class Demo
  def initialize a
    @a = a
  end
  def +(b)
    "plus b equal to" + (@a+b).to_s
  end
end

a = Demo.new(2)
puts a + 2      #4
```

变量多态（parametric polymorphism）

参数多态允许函数或数据类型采用同一定义，但是可以处理不同类型的变量。变量多态在保证静态类型安全的情况下使语言的表述更加清晰简明。

在Java语言中通过泛型来实现变量多态，当然动态类型语言在运行时会自动适配参数的数据类型，因此实现上加简洁。

```
//Java 泛型

public class Tree<T>{
    private T value;
    private Tree<T> left;
    private Tree<T> right;

    public void replaceAll(T value){
        this.value = value;
        if(left != null)  left.replaceAll(value);
        if(right != null)  right.replaceAll(value);
    }
}
```

子类型多态 (Subtyping)

一些语言提供了子类型多态的概念，子类型多态允许你在定义函数的时候使用父类型做形参，但实际使用的实参可以是其子类型。

```
//Java 子类型多态

abstract class Pet {
    abstract String run();
}

class Dog extends Pet {
    String run() {
        return "run with four legs";
    }
}

class Duck extends Pet {
    String run() {
        return "run with two legs";
    }
}

public class Main{
    public static void petRun(Pet p){
        System.out.println(p.run());
    }
    public static void main(String args[]){
        Main.petRun(new Dog());      //run with four legs
        Main.petRun(new Duck());    //run with two legs
    }
}
```

上述代码中定义了petRun方法，形参为Pet类，实参为Pet类的子类Dog或Duck，因此实际上调用的run方法是子类的一个实现。

这里特别说明一点，子类型多态和继承是两个不同的概念，子类型多态强调的是类型之间的关系，即子类型可以替代父类型；而继承强调的是对象间的关系，即一类对象可以从另一类对象衍生出来。

最后我们通过变量多态和子类型多态组合使用的例子来加深对多态的认识。

```
//Java 多态

interface Runner{ String run(); }

class Dog implements Runner{
    public String run(){
        return "run with four legs";
    }
}

class Duck implements Runner{
    public String run(){
        return "run with two legs";
    }
}

class Pet<T extends Runner>{
    private T pet;
    public Pet(T pet){
        this.pet = pet;
    }
    public String run(){
        return pet.run();
    }
}

public class Main{
    public static void main(String args[]){
        Pet<Dog> dog = new Pet<Dog>(new Dog());
        Pet<Duck> duck = new Pet<Duck>(new Duck());

        System.out.println(dog.run());      //run with four legs
        System.out.println(duck.run());     //run with two legs
    }
}
```

我们整理下上述代码的逻辑：

1. 首先声明了一个Runner的接口，接口中定义了一个名为run的行为。
2. 然后创建了Dog类和Duck类，他们分别实现了Runner接口的run行为。
3. 接着定义了一个Pet类，这个类需要传递一个实现了Runner接口的泛型（类）。
4. 在Pet类定义了一个run方法，这个方法返回的是实现了Runner接口的类型T的run方法。

5. 执行主函数，声明了两个Pet类，一个传入的类型T是Dog，另一个是Duck，最后分别调用这两个Pet的run方法。

封装

封装是通过类将数据和函数包装起来，封装具有如下作用：

1. 将数据、函数的实现细节隐藏，简化使用
2. 防止外界直接访问包装内的数据、函数

与封装密切相关的一个概念是访问级别，它定义了外部访问包装内数据和函数的规则。

访问级别 (Access Level)

访问级别用来控制一个类的属性或方法能否被其他类访问，以Java的访问级别为例来说明包括：private、protected、public和no modifier（不定义）这四种，其对应的访问权限如下表(Y表示可访问，N表示不可)

修饰符/场景	当前类	当前包	子类	全局
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

假设A是B的父类，如果A中的一个属性p是用private修饰的话，那么B是无法访问这个属性的。相对的如果属性p是用public修饰的，那么B就可以访问这个属性。

```
//Java 访问权限

class A {
    protected String a = "protected prop can access subclass";
    private int b = "private prop can not access in subclass";
}

class B extends A {}

public class Main{
    public static void main(String args[]){
        B b = new B();
        System.out.println(b.b);
        //error: b has private access in A
    }
}
```

JavaScript中没有提供私有属性的概念，通常的人为的用`_`为开头命名一个私有变量，使用时避免在外部访问某对象的以`_`为开头命名的变量就可以了。

继续刚才的例子，如果你就想用B来访问A的私有属性`p`，那么你可以在A中定义一个`public/protected`修饰的方法，然后通过这个方法返回`p`。

```
//Java 通过公开的方法访问私有属性

class A {
    private int p = 1;
    protected int getP(){
        return this.p;
    }
}

class B extends A {}

public class Main{
    public static void main(String args[]){
        B b = new B();
        System.out.println(b.getP());      //1
    }
}
```

或者通过Java的反射机制获取。

```
//Java 反射

import java.lang.reflect.*;
class A {
    private int p = 1;
}

public class Main{
    public static void main(String args[])throws Throwable {
        A a = new A();
        Field field = a.getClass().getDeclaredField("p");
        field.setAccessible(true);
        Object b = (int)field.get(a);
        System.out.println(b);      //1
    }
}
```

Python和Ruby则提供了直接访问私有属性的语法糖。

```
#Ruby 访问私有成员

class A
  def initialize
    @a = 10
  end

  private
  def private_method(b)
    return @a + b
  end
end

a = A.new
puts a.send(:private_method, 20)      #30
puts a.instance_variable_get(:@a)      #10
```

继承

类或对象继承自另一个类或对象就能具备其属性和方法，通常继承分为三种：单继承、多重继承和多级继承。

B继承自A这种叫做单继承；C继承自A和B叫做多重继承；C继承自B，B继承自A叫做多级继承。

单继承和多级继承本身不存在什么问题，但多重继承却存在很多的问题，最典型的问题就是菱形继承问题（diamond problem）。

如图B、C都继承自A，并且重写了A的metho的方法，而D继承自B和C，同时D没有重写method方法，那么这时D的method方法究竟该继承自B，还是继承自C，这就是菱形继承问题。

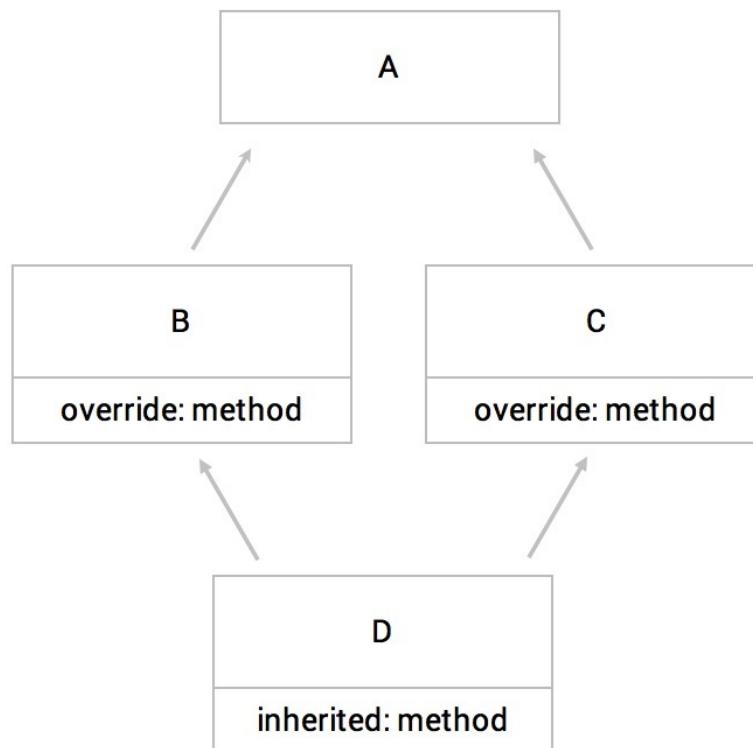
因为多重继承暴露出太多的问题，所以多数语言规定不允许多重继承，但多重继承又是不可或缺的功能，比如铅笔即属于文具，又是木制品，那么它要满足这两点，就需要继承文具类和木制品类，那么该如何处理这种需求呢？

不同的语言提出了不同的方案：

混合（Mixin）

Mixin也是一种代码复用的策略，任何对象都可以通过Mixin拓展方法，与类的多重继承一样，一个对象可以引用多个Mixin。

Ruby就是采用Mixin的方式实现多重继承的代表。



#Ruby 混合

```
module Action
  def jump
    "jump with legs"
  end
end

module Sound
  def say
    "say #{@sound}"
  end
end

class Cat
  def initialize
    @sound = "mew"
  end
  include Action
  include Sound
end

cat = Cat.new
puts cat.jump      #jump with legs
puts cat.say       #say mew
```

上面我们定义了两个Mixin，一个用于描述动作，一个用于描述叫声。我们只需要在类的定义时通过include关键字引入Action和Sound就可以调用相关的方法了。

注意Ruby的类中如果存在和module（Mixin）里定义的同名的方法，则以类中的方法体为准，如果多个module定义了同一方法，就以最后拓展（include）的那个为准。Ruby正是通过这种设计避免了多重继承中的菱形继承的问题。

Trait

Trait和Mixin类似也是用来拓展类的方法的，只是Trait中不允许实例化对象，同时如果引用的两个Trait存在同名函数，那么将不会采取任何策略解决这个问题，而会直接抛出异常。

我们来看下Scala中Trait的使用。

```
//Scala Trait

class Person
trait Nice{
    def greet() = println("Hello")
}

class Character extends Person with Nice

val american = new Character()
american.greet //Hello
```

上面我们定义了一个父类Person以及一个Trait，然后定义了一个Character类，让他继承自Person，同时使用了Trait拓展。这样Character在没有定义任何方法时也是可以访问Nice的greet方法的。

接口（Interface）

接口和Mixin、Trait类似，也是用来给类拓展方法的，与Mixin、Trait不同的是，接口只定义方法，但不实现。接口的实现由其实现类实现。

```
//Java 接口

interface Move{
    String jump();
}

interface Sound{
    String say();
}

class Cat implements Move, Sound{
    public String jump(){
        return "jump with legs";
    }

    public String say(){
        return "mew";
    }
}

public class Main{
    public static void main(String args[]){
        Cat cat = new Cat();
        System.out.println(cat.jump());      //jump with legs
        System.out.println(cat.say());       //mew
    }
}
```

因为接口规定了实现由类完成，所以多个接口中存在同名函数时也是不会有任何问题，因为类的函数实现肯定是唯一的。

注意Java中的接口和抽象类不同，抽象类的抽象方法和接口类似。在只需要拓展一个接口的情况下可以用抽象方法取代接口，但需要拓展多个接口的时候抽象类是无法取代的，所以接口主要处理的问题还是多重继承。

组合（composition）

组合也是一种实现多重继承的方式，简单的说就是一个类的属性是另外一个类，此时这个类中就可以通过这个属性来调用另一个类的方法了，出于理解方便你可以把这个类看做是一个Mixins。一般我们称继承为is a关系，称组合为has a关系，即：

1. A是B的一个子类，那么A自然是B，所以A is a B
2. A通过组合引用了B，那么A中有B，所以A has a B

```
//Java 组合

class Move{
    public String jump(){
        return "jump";
    }
}

class Sound{
    public String say(String word){
        return word;
    }
}

class Cat{
    String word;
    Move move;
    Sound sound;
    Cat(){
        this.word = "mew";
        this.move = new Move();
        this.sound = new Sound();
    }

    public String jump(){
        return this.move.jump();
    }

    public String say(){
        return this.sound.say(this.word);
    }
}

public class Main{
    public static void main(String args[])
    {
        Cat cat = new Cat();
        System.out.println(cat.jump());      //jump
        System.out.println(cat.say());       //mew
    }
}
```

我们来看下上面这段代码：

1. 首先定义了一个Move类，实现了jump方法
2. 然后定义了一个Sound类，实现了say方法
3. 接着定义Cat类，组合Move类和Sound类作为属性
4. 重写Cat类的构造函数，实例化move对象和sound对象。
5. 在Cat类中定义jump方法和say方法，方法体内分别调用move的jump方法及sound的say方法

抽象

抽象用来隐藏复杂的底层细节，将关注点放在呈现的结果，一般而言抽象分两种。

- 操作抽象，比如编程语言中的`+`运算符，你只需要知道它能够处理加法运算就行，不需要知道底层的实现细节
- 数据抽象，比如编程语言中的`Number`类型，你只需知道它用来存储数字类型就行，不需要知道底层的存储细节

面向对象中的抽象也是用于隐藏实现细节的。在Java语言下抽象通过抽象（`abstract`）和接口（`interface`）两种方式实现。

这里就不再重复说明接口了，抽象和接口功能类似，它可以用来修饰类也可以修饰方法。通过`abstract`修饰的类叫做抽象类，修饰的方法叫做抽象方法。抽象类不能实例化，一般作为父类使用，抽象方法不需要定义方法体，方法体由子类实现。另外因为抽象方法需要子类来实现，所以`abstract`不能和`private`（子类是没有访问父类的私有属性的权限的）一起使用。

```
//Java 抽象

abstract class Animal{
    public abstract String say();
}

class Cat extends Animal{
    public String say(){
        return "mew";
    }
}

public class Main{
    public static void main(String args[]){
        Animal animal = new Animal();
        //error: Animal is abstract; cannot be instantiated

        Cat cat = new Cat();
        System.out.println(cat.say());      //mew
    }
}
```

抽象作为一种多态的实现机制对于Ruby这种动态类型的语言而言是不需要的，因为在动态类型语言中函数参数和返回值无需强制声明，解释器会根据类型推导机制自动判断。

```
//Java 函数重载
public String getA(String a){
    return a;
}

public int getA(int a){
    return a;
}
```

```
#Ruby 函数定义
def getA(a)
    return a
end
```

从上面例子可以看出Java这种静态类型语言需要显式的声明函数的参数类型和返回值类型。因为参数类型或返回值类型不同的同名函数是两个不同的函数，所以Java中抽象的概念是必要的。而在Ruby这种动态类型语言中，参数类型和返回值类型是程序在运行时动态决定的，函数本身就能够接收不同类型的输入参数，所以抽象类和抽象方法是没有必要的，这也是Ruby不支持抽象类和抽象方法的原因。

虽然Ruby本身没有提供对抽象的支持，但是还是可以通过抛出异常的方式来模拟抽象方法或者抽象类的。

```
#Ruby 模拟抽象方法

class Animal
    def say
        raise "abstract method"
    end
end

class Cat < Animal
    def say
        return "mew"
    end
end

c = Cat.new
puts c.say      #mew
```

我们在Animal里定义了一个say方法，该方法没有执行任何操作，只是抛出一个异常（防止直接调用这个方法），然后我们声明了个Cat类继承了Animal，并重写了say方法。这样通过抛出异常的方式限制父类的say方法直接调用，来模拟Java的抽象方法。

设计模式

设计模式 (design pattern) 是指对软件设计中常见的各种问题，提出一套通用可行的解决方案，从而避免重复设计方案来处理同一类问题。

设计模式的提出可以提高代码的复用性、易读性、可靠性，使程序开发更加工程化。

背景

1. 1987年Kent Beck和Ward Cunningham参照建筑设计领域的思想提出了设计模式并应用在Smalltalk的图形接口上。
2. 1989年Erich Gamma在他的博士论文里开始尝试把这种思想用于软件开发。
3. 1989年Erich Gamma拿到博士学位后在美国与Richard Helm, Ralph Johnson, John Vlissides 合作出版了《Design Patterns》一书，书中提到了23种设计模式。
4. 经过20多年的发展，软件开发中套用模式已经成为一种流行的设计方法。

由于《设计模式》这本书的作者有四人，所以在软件开发领域称他们为四人帮（Gang of Four），简写作GoF，因此GoF也就成了软件设计模式的代称。

分类

设计模式总体来说分为三大类：

1. 创建型，共五种：工厂模式、抽象工厂模式、单例模式、建造者模式、原型模式。
2. 结构型，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
3. 行为型，共十一种：策略模式、模板模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

原则

1. 开闭原则 (Open Close Principle)

开闭原则是指对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，从而使程序易于维护和升级。一般通过使用接口和抽象类来达到这个目的。

2. 里氏代换原则 (Liskov Substitution Principle)

里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键就是抽象化。而父类与子类的继承关系就是抽象化的一种具体实现。里氏代换原则下，任何父类都可以被它的子类替换，只有这样才能保证父类是可复用的。

3. 依赖倒置原则 (Dependence Inversion Principle)

依赖倒置强调针对接口编程，不是依赖与具体的实现类，而是依赖于这个实现类的抽象（接口或抽象父类）。

4. 接口隔离原则 (Interface Segregation Principle)

将可以拆分的接口拆分为多个，目的在与降低接口之间的耦合性，降低类与类，类与接口间的耦合性能使程序更易于维护。

5. 迪米特法则 (Demeter Principle)

最少知道原则同接口隔离原则类似，最少知道原则是针对实体而言，强调实体间应该尽量少的相互作用，从而保证功能模块的独立，本质上还是降低耦合性使程序易于维护。

6. 合成复用原则 (Composite Reuse Principle)

尽量使用组合的方式，而不是继承。

几种设计模式

我不准备把23种设计模式一一列举，毕竟这不是一本讲设计模式的书，我只是想说明设计模式是面向对象编程下一个必不可少的要素。所以我将从中挑选出几个比较经典的设计模式进行介绍，当然介绍设计模式没有比Java语言更合适的了。

工厂模式

在开发过程中经常需要根据不同的条件实例化不同的对象，每次都重复实现这个逻辑显然是不理想的，所以我们可以把这样的逻辑封装到一个类中，这个类就是工厂类，通过类和条件参数返回需要实例化的对象，这个模式就是工厂模式。

```
//Java 工厂模式

interface Language{
    public void sayHello();
}

class Java implements Language{
    public void sayHello(){
        System.out.println("say hello by Java");
    }
}

class Ruby implements Language{
    public void sayHello(){
        System.out.println("say hi by Ruby");
    }
}

class LanguageFactory{
    public Language setLanguage(String type){
        if("Java".equals(type))      return new Java();
        else if("Ruby".equals(type)) return new Ruby();
        else{
            System.out.println("type error");
            return null;
        }
    }
}

public class Main{
    public static void main(String args[]){
        LanguageFactory languageFactory = new LanguageFactory();
        Language java = languageFactory.setLanguage("Java");

        java.sayHello();    //say hello by Java
    }
}
```

来看下上面的代码：

1. 创建一个Language接口，并定义了一个sayHello方法。
2. 分别创建Java类和Ruby类，实现Language接口的sayHello方法。
3. 创建一个LanguageFactory类并实现setLanguage方法，这个方法的返回值是一个Language接口。我们可以根据type参数进行逻辑判断，返回一个实现了Language接口的类。
4. 调用LanguageFactory的setLanguage得到需要的对象。

单例模式

对于需要频繁实例化的类，每次调用都实例化一个对象会对内存造成很大的开销。考虑到内存的开销，我们是否可以在实例化这个类的时候判断下是否存在该类的对象，如果存在则直接返回这个对象而不是实例化一个新的对象。

单例模式就是这样的作用，它可以避免多次实例化一个类的对象从而节省内存开销。另外在类似交易系统中，如果多次创建控制类对象会导致系统流程控制混乱，这样的场景下也是离不开单例模式的。

因为单例模式经常与工厂模式配合（通常工厂类只需要实例化一次就可以），所以我们上面的工厂类改为单例类。

```
//Java 单例模式

class LanguageFactory{

    private static LanguageFactory instance = null;
    private LanguageFactory(){}

    public static LanguageFactory getInstance(){
        if(instance == null) instance = new LanguageFactory();
        return instance;
    }

    public void demoMethod(){
        System.out.println("Singleton pattern");
    }
}

public class Main{
    public static void main(String args[]){
        LanguageFactory.getInstance().demoMethod();
        //Singleton pattern
    }
}
```

单例模式用到了访问权限的特性，来看下上面的代码：

1. 保证单例类不能被实例化的方式就是让其构造函数变为私有属性，这样就不能通过构造函数构造当前类的对象了。
2. 因为不能在外面实例化对象，所以我们需要单例类本身提供一个自己的实例（对象）作为属性，并且这个属性是类级别的（否则会变成一个死循环，如果是成员属性就必须有对象，而对象本身是通过这个属性暴漏出来的）。
3. 最后需要暴漏一个类方法来返回这个属性（当前类的一个实例）。
4. 这样就可以直接通过LanguageFactory的getInstance拿到这个类的唯一实例了。

适配器模式

适配器的作用主要用于将一个接口转换为一个期望的接口，从而消除两个接口间不兼容的问题。其本质上也是一种封装、多态的设计。

适配器可以将代码细节更好的隐藏，实现上和工厂模式类似，适配器要根据输入参数选择合适的处理类及方法（不同的类实现了不同的接口方法）。

来看下面这个播放器的例子：

```
//Java 适配器模式

interface MediaPlayer{
    public void play(String audioType);
}

interface AdvancedMediaPlayer{
    public void playFlv();
    public void playMp4();
}

class FlvPlayer implements AdvancedMediaPlayer{
    public void playFlv(){
        System.out.println("play flv");
    }
    public void playMp4(){
        //do nothing
    }
}
class Mp4Player implements AdvancedMediaPlayer{
    public void playFlv(){
        //do nothing
    }
    public void playMp4(){
        System.out.println("play mp4");
    }
}

class AdapterPlayer implements MediaPlayer{
    AdvancedMediaPlayer advancedMediaPlayer;

    public AdapterPlayer(String type){
        if("flv".equals(type)) advancedMediaPlayer = new FlvPlayer();
        else if("mp4".equals(type)) advancedMediaPlayer = new Mp4Player();
        else System.out.println("type error");
    }
    public void play(String audioType){
        if("flv".equals(audioType)) advancedMediaPlayer.playFlv();
        else if("mp4".equals(audioType)) advancedMediaPlayer.playMp4();
        else System.out.println("audio type error");
    }
}
```

```
class AudioPlayer implements MediaPlayer{
    AdapterPlayer adapterPlayer;
    public void play(String audioType){
        adapterPlayer = new AdapterPlayer(audioType);
        adapterPlayer.play(audioType);
    }
}
public class Main{
    public static void main(String args[]){
        AudioPlayer audioPlayer = new AudioPlayer();
        audioPlayer.play("mp4");      //play mp4
        audioPlayer.play("flv");      //play flv
    }
}
```

简单分析下上述代码：

1. 我们有一个要适配的接口AdvancedMediaPlayer和用来包装的接口MediaPlayer，MediaPlayer定义了一个play方法，而AdvancedMediaPlayer对play细化定义了playFlv和playMp4。
2. 创建FlvPlayer和Mp4Player类，一个实现了AdvancedMediaPlayer接口的playFlv方法，另一个实现了AdvancedMediaPlayer接口的playMp4方法。
3. 定义了适配器AdapterPlayer，适配器实现了包装接口MediaPlayer的play方法，同时适配器同工厂类似根据输入参数返回具体的advancedMediaPlayer对象。
4. 创建AudioPlayer类，该类通过适配器AdapterPlayer实现了包装接口MediaPlayer的play方法，并根据play方法输入的参数实例化AdapterPlayer对象，然后调用adapterPlayer的play方法。这样AudioPlayer就通过适配器就隐藏了play的细节。

装饰模式

装饰模式可以在不修改现有结构的情况下，对当前对象追加新特性，装饰模式会创建一个装饰类来包装这个对象，然后通过装饰类来拓展新特性。

相当于我们要对一个类进行拓展，可以通过组合的方式把它包装到装饰类里，通过对装饰类定义属性和方法来拓展新特性。

来看下面这段代码：

```
//Java 装饰模式

interface Animal{
    public void say();
}

class Dog implements Animal{
    public void say(){
        System.out.println("bark");
    }
}

class Cat implements Animal{
    public void say(){
        System.out.println("mew");
    }
}

abstract class AnimalDecorator implements Animal{
    protected Animal animal;
    public AnimalDecorator(Animal animal){
        this.animal = animal;
    }
    public void say(){
        animal.say();
    }
}

class LovelyAnimalDecorator extends AnimalDecorator{
    public LovelyAnimalDecorator(Animal animal){
        super(animal);
    }
    public void say(){
        setCharacter();
        animal.say();
    }
    public void setCharacter(){
        System.out.print("lovely ");
    }
}

public class Main{
    public static void main(String args[]){
        Animal cat = new Cat();
        Animal kitten = new LovelyAnimalDecorator(cat);
        kitten.say();      //lovely mew
    }
}
```

上面代码如下：

1. 首先创建了一个Animal接口并定义了say方法，然后定义了实现这个接口的两个类Cat和Dog。
2. 定义一个装饰类的抽象类，抽象类中定义了一个Animal类型的属性，方便子类包装使

用。

3. 定义了一个装饰类LovelyAnimalDecorator，继承了抽象类AnimalDecorator，并添加了新特性，重写了say方法。
4. 实例化一个原始类对象cat，把cat作为参数传递给包装类LovelyAnimalDecorator，调用包装类的say方法。
5. 包装类的say方法执行包装类定义的新特性，并执行原始类的say方法。

代理模式

代理模式是指当需要执行一个类的某个操作时，不是直接调用这个类的方法，而是通过一个代理类进行包装，调用这个代理类的方法，由代理类来执行真正的需要调用的操作。

这个模式有点像中介服务，比如你可以直接从卖家手里购买商品，也可以选择通过电商平台来与卖家交易。在卖家看来与他交易的是电商平台，而实质上购物的真实操作是你完成的，电商平台只是把你的钱给了卖家，那么电商平台就是你的代理类，替你来买执行购物的操作。

来看下面这段代码：

```
//Java 代理模式

interface Trade{
    public void buy(String good);
}

class Buyer implements Trade{
    public void buy(String good){
        System.out.print("buy " + good + " ");
    }
}

class Taobao implements Trade{
    private Buyer buyer;
    public void buy(String good){
        if(buyer == null)    buyer = new Buyer();
        buyer.buy(good);
        System.out.println("by taobao");
    }
}

public class Main{
    public static void main(String args[]){
        Trade taobao = new Taobao();
        taobao.buy("cd game");    //buy cd game by taobao
    }
}
```

分析以上代码

1. 创建了Trade接口并定义了购买方法。 2. 创建Buyer类并实现Trade接口。 3. 定义了代理类Taobao，拥有一个Buyer属性，实现Trade接口，并在buy方法中调用buyer对象的buy方法。

观察者模式

观察者模式通常在如下场景使用：当一个对象的属性发生改变时，所有依赖它的对象都要收到相应的通知。

类似的使用场景比如消息订阅，当消息发生改变时，所有订阅这个消息的用户都应该收到通知。

我们以天气预报为例来说明：

```
//Java 观察者模式

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

class WeatherForecast{
    private List<Observer> observers = new ArrayList<Observer>();
    private String weather;

    public String getWeather(){
        return this.weather;
    }
    public void setWeather(String weather){
        this.weather = weather;
        this.notifyObservers();
    }
    public void subscribe(Observer observer){
        observers.add(observer);
    }
    public void notifyObservers(){
        for (Observer observer : observers) {
            observer.notified();
        }
    }
}
abstract class Observer{
    protected WeatherForecast weatherForecast;
    public abstract void notified();
}
class You extends Observer{
    public You(WeatherForecast weatherForecast){
        this.weatherForecast = weatherForecast;
        this.weatherForecast.subscribe(this);
    }
    public void notified(){

    }
}
```

```
System.out.println("notified you, "+ weatherForecast.getWeather());
}
}

class YourFriend extends Observer{
    public YourFriend(WeatherForecast weatherForecast){
        this.weatherForecast = weatherForecast;
        this.weatherForecast.subscribe(this);
    }
    public void notified(){
        System.out.println("notified your friend, "+ weatherForecast.getWeather());
    }
}

public class Main{
    public static void main(String args[]){
        WeatherForecast weatherForecast = new WeatherForecast();
        new You(weatherForecast);
        new YourFriend(weatherForecast);
        weatherForecast.setWeather("rain");
        //notified you, rain
        //notified your friend, rain
    }
}
```

简单的说观察者模式就是两个类之间消息的传递，消息类状态发生改变时会通知观察者执行相应操作，而观察者在初始化时会通知消息类把自己加入到观察者注册列表中。具体如下：

1. 首先创建一个消息类，消息类中有一个消息状态及状态的访问器，这里定义了用来描述天气的属性weather及weather的属性访问器setWeather和getWeather。
2. 消息类还定义了一个观察者列表及相关的方法。观察者列表observers用来存储所有订阅该消息的观察者，subscribe方法用来添加新的观察者对象到观察者列表，notifyObservers方法用来通知观察者消息改变了。
3. 其中notifyObservers方法遍历观察者列表中每一个观察者，并调用他们的notified方法。
4. 接着定义了观察者抽象类Observer，这个抽象类拥有消息类WeatherForecast及notified方法。
5. 然后定义了两个观察者You和YourFriend，分别在构造函数中调用了消息类weatherForecast的观察者注册方法subscribe，这样在消息发生改变时所有在观察者列表中的观察者都会收到相应的通知并调用notified方法。
6. 最后让每个观察者类都实现notified方法，当收到消息类发送的通知时分别执行各自的处理方法。

责任链模式

责任链是将一系列对象像链表一样链接起来，其中每个对象完成一个指定的任务，然后将任务传递给下一个对象。

比如生产线上每一个车间完成一个加工任务，然后流水线到下一个车间。

我们来看如下代码：

```
//Java 责任链模式

abstract class Task{
    protected Task next;

    public void nextTask(Task task){
        this.next = task;
    }
    protected abstract void work();
    public void start(){
        work();
        if(this.next != null)  this.next.start();
    }
}

class Write extends Task{
    public void work(){
        System.out.print("write ");
    }
}

class Compile extends Task{
    public void work(){
        System.out.print("compile ");
    }
}

class Link extends Task{
    public void work(){
        System.out.print("link ");
    }
}

class Run extends Task{
    public void work(){
        System.out.print("run");
    }
}

class TaskChain{
    public static Task getTaskChain(){
        Task write = new Write();
        Task compile = new Compile();
        write.next = compile;
        compile.next = new Link();
        link.next = new Run();
        return write;
    }
}
```

```

Task link = new Link();
Task run = new Run();
write.nextTask(compile);
compile.nextTask(link);
link.nextTask(run);
return write;
}
}

public class Main{
    public static void main(String args[]){
        Task taskChain = TaskChain.getTaskChain();
        taskChain.start();    //write compile link run
    }
}

```

这里我们以Java语言从代码到运行这个过程为例进行说明（Java从代码到执行要经过编写、编译、链接、运行四个步骤）。

1. 首先我们创建一个Task抽象类，Task类定义了一个Task类型的属性next来指明下一个任务节点。
2. Task类中包含一个抽象方法work，交由子类实现。同时还定义了一个start方法，用来执行当前Task的work方法并判断是否存在下一个任务节点，如果存在，则执行这个任务节点的start方法。这样就可以构成一个任务链了。
3. 然后定义Write、Compile、Link、Run四个任务，都继承自Task类，并实现了work方法。
4. 定义一个责任链类TaskChain，类中定义了一个静态方法getTaskChain用来获取任务链，方法中创建了任务类对象，并通过nextTask方法组成一个任务类链表，然后将任务链的第一个任务返回。
5. 最后调用taskChain.start方法触发任务链的执行，start方法会执行当前任务的work方法并判断是否存在下一个任务，如果存在，则执行下一个任务的start方法，否则结束任务，以此类推。

设计模式和语言特性

设计模式本身是一种规范与技巧，其存在的目的是完善或者弥补面向对象语言在编程开发上的不足。一些语言无法更好解决的问题，开发中可以通过适当的使用设计模式来妥善处理，但是如果语言本身具备了所需要的特性，那么设计模式就可以抛到一边了。就像Java提供了C++中没有的垃圾回收机制，你就无需在Java中手动释放内存了，Scala提供了Java中没有的线程管理机制，你就无需在Scala中加锁一样。如果语言本身提供了更好的特性，你也就不必为了提高代码的复用性和维护性采用各种复杂设计模式了。

我们可以通过以下两个例子来对比说明设计模式是如何弥补语言设计上的缺陷的，以及某些语言是如何隐藏掉设计模式的。

Java8接口方法替代工厂模式

在讲匿名函数时提到过Java8新增了lambda特性，基于lambda的特性对应的也提出了接口方法，简单的说接口的函数体（方法实现）可以像变量被赋值，而旧版本的Java必须通过实现类来实现接口。因为这个特性的增加，我们可以跳出必须通过实现类实现接口方法的约束，直接在接口声明时指定接口方法的实现。

```
//Java8 接口方法替代工厂模式

interface Language{
    public void sayHello();
}

class NewFeatureReplaceFactory{
    private Language language;
    private String type;

    public NewFeatureReplaceFactory(String type){
        this.type = type;
    }
    public void sayHello(){
        if("Java".equals(this.type))
            language = ()-> System.out.println("hello Java8");
        else if("Ruby".equals(this.type))
            language = ()-> System.out.println("hello Ruby");
        else
            language = ()-> System.out.println("type error");
        language.sayHello();
    }
}

public class Main{
    public static void main(String args[]){
        NewFeatureReplaceFactory java = new NewFeatureReplaceFactory("Java");
        java.sayHello();    //hello java8
    }
}
```

从上面代码可以看出，我们没有定义Language接口的实现类，而是在工厂类NewFeatureReplaceFactory的sayHello方法中根据type类型不同，对Language接口指定了不同的函数体实现（注意使用接口函数时，接口只能定义一个函数，解释器会自动把定义的函数体和函数定义绑定），这段逻辑其实和定义不同的类来实现Language的sayHello方法是等效的。

Io语言和单例模式

我们再来一个更加明显的例子，这里将引入一门基于原型的嵌入式语言Io，Io是基于原型的语言，支持面向对象编程范式，但是没有模板的概念，而是通过原型链来处理继承层次，原型的方法查找也像其他基于原型的语言一样在原型链中一层一层的向上查找，有就返回，没有就创建。

Io中通过clone方法来链接原型之间的层级关系，`B := A clone` 表示原型链中A在B的上级，同时clone方法默认也会返回一个新的Object原型。假设一个原型重写了clone方法，并指定返回值就是当前原型本身，那会如何？没错，你没办法通过clone来创建一个新的Object原型，取代的是你始终会得到这个原型本身，这不正是单例模式要实现的功能么。

Io 通过将原型链指向自身实现单例模式

```
Singleton clone := Singleton
isSingleton := Singleton clone
isSingleton = Singleton
//true
```

关于设计模式的争论

设计模式是面向对象发展过程中演化出来的编程技巧，适当的使用可以使代码更易于维护和拓展。关于设计模式有两种极端的人持有各自的想法。

1. 通常根深蒂固于面向对象思维方式下的人会认为设计模式是一种信仰，设计模式是评判一个人技术水平的基本准则，不使用设计模式的程序一文不值。
2. 一些函数式拥护者认为，语言本身存在重大的缺陷才需要设计模式来弥补，足够优秀的语言是不需要设计模式这种繁琐冗杂的技巧来打补丁的。

个人认为这两种想法都是不合理的，目前主流语言都纷纷加入函数式特性，部分模式是可以通过新特性化简的，设计模式确实可以体现出一个编程人员的水平，但没必要过分依赖和看重。另外函数式向工程化发展的过程中也渐渐意识到设计模式的重要性，渐渐的也提出了函数式下常用的设计模式，所以并非完备的语言是不需要模式的。这是语言本身特性和通过模式弥补之间的一个折中。

设计模式是一个复杂而高级的编程技巧，适当的使用可以使代码更健壮、更易维护，但是不要当做是一种信仰过分的依赖，比较所有技术手段和技巧其核心目的无非是提供更健壮、更易拓展的代码。如果你使用的语言本身有更好的特性，那么用语言的特性也是能够达到同样的效果的；同理再优秀的语言，也存在一定的弊端，设计模式是在经验积累下发展出来，弥补语言缺陷的重要手段之一。

和我之前在强弱类型，动静类型中所表述的一样，没必要在一个数轴上过分偏向哪一端，遵循语言的设计初衷一样，设计模式的使用也是如此，如果你对新特性够熟悉当然可以选择跳过设计模式的短板，同理如果语言需要设计模式的补充，也没必要否认。

第六章 声明式下的函数式概念

继第四章和第五章，这里针对第四章提到的声明式，取目前比较流行的函数式语言，说明函数式语言相关概念。

1. 介绍函数式语言的特点。
2. 介绍高阶函数及高阶函数引入的概念。
3. 介绍函数式语言其他概念。

函数式编程

函数式编程通过数学函数求值来处理程序计算，它具有代码简洁易读，更接近于自然语言，开发效率高等优点。

函数式特点

他有如下特点：

1. 函数为一等公民
2. 声明式
3. 值不可变
4. 引用透明

函数为一等公民

一等公民（first class）简单的说就是可以当做值来使用的，比如作为变量的值，作为函数的返回值。而函数作为一等公民就是说函数也可以作为参数进行传递，或者作为另一个函数的返回值。因为函数是一段代码，所以函数作为一等公民就意味着代码段及代码段内包含的变量也是可以传递的。

```
-- Haskell 函数作为参数传递给另一个函数
map (+3) [1,2,3]
```

如上 `map` 是一个函数，`(+3)` 也是一个函数（这个函数传递了变量3作为加法的右操作数），`[1,2,3]` 是另一个参数，这里把函数 `(+3)` 当做参数传递给了另一个函数 `map`。

声明式

函数式编程是一种声明式编程范式，声明式通过表达式计算、函数声明来代替命令式编程中的逻辑控制。

```
--Haskell 函数定义

isZero:: Int->String
isZero 0 = "yes"
isZero _ = "no"

isZero 3    --no
```

如上声明了一个函数isZero，其中Int->String表示接受一个整数类型，返回一个字符串类型。然后声明了 `isZero 0 = "yes"` 表示当参数为0时返回yes。最后 `isZero _ = "no"` 表示除此之外任意参数都返回no。

isZero会按照声明的顺序对输入参数进行判断，先判断输入参数是否为0，如果为0则返回yes，否则执行下一个模式，一般最后一个模式都用 `_` 匹配任意值，相当于if else中的else。

对比声明式编程看下命令式如何实现isZero。

```
//Java 命令式编程
public static String isZero(int num){
    if(num == 0)    return "yes";
    else           return "no";
}

System.out.print(isZero(0));    //yes
```

值不可变

1. 函数式编程是没有副作用的；
2. 函数式要求值是不可修改的，状态也是不可改变的；
3. 对输入参数进行函数运算不允许修改源数据，相应的会生成一个新的拷贝作为运算结果。
4. 函数运算只依赖于输入参数，不会改变函数外部数据的值或状态。
5. 面向对象语言通过封装解决状态改变的问题，而函数式语言通过函数变换避免状态改变。

引用透明

引用透明是指函数的运算不会受到外部变量的影响，其计算结果只依赖于输入参数，输入参数相同则返回结果相同。

引用透明的特性使函数式语言使其具有如下优势：

1. 如果当前函数表达式返回值不再需要，可以直接删除，不会对函数表达式外数据产生影响。
2. 多次调用一个纯函数输入相同参数，返回结果相同，且不存在副作用。
3. 替换两个纯函数表达式的执行顺序不会影响执行结果，因此适合做并行开发。
4. 无副作用的表达式运算允许编译器自由结合表达式的运算结果，可以将多个函数组合使用。

对比命令式编程、面向对象编程

1. 命令式编程是按照程序是一系列改变状态的命令来建模的一种编程风格，函数式编程则将程序描述为表达式和变换，以数学方程的形式建立模型。
2. 命令式编程一次循环完成多个任务，更注重性能，而函数式一个任务可能需要多次循环，更注重语义。
3. 面向对象是对数据进行封装，而函数式则对行为进行封装。
4. 面向对象编程中通过封装不确定因素使代码更容易被人理解，而函数式编程通过尽量减少不确定因素来使代码容易被人理解。
5. 面向对象提倡对具体的问题建立专门的数据结构及操作，而函数式则提倡使用几种基本的数据结构（数组、列表）及针对这几种结构高度优化的操作来完成程序任务。
6. 函数式编程将程序描述为表达式和变换，像数学方程一样建立模型。更注重高层的抽象，而非底层的细节，开发者需要用高阶函数调整底层运转。
7. 函数式编程能在更细小的层面上重用代码。

完美数的例子：

上面像说明书一样列举了函数式的特点，并对比了与命令式的区别。但这些终究是教科书一样的文案，没有实际的例子是难以认清函数式编程的真面目的，下面我们来通过完美数例子来对比命令式编程与函数式编程。

完美数是指除了自身以外的所有他的正约数之和等于其本身的数字，比如
6($1+2+3=6$,其中6的非自身正约数有1, 2, 3)。

在说明编码之前，先说明下如何判断一个数是否为完美数。

1. 取1到这个数之间的所有数。
2. 把这些数中能整除它的数筛选出来。
3. 对筛选出来的数字求和。
4. 判断是否与这个数自身相等。

这里我们用JavaScript实现完美数例子：

```
//JavaScript 命令式实现完美数

(function(){
    var factors = []
    ,   number
    ,   sum = 0
    ,   result;

    number = 496;

    getFactors();
    aliquoSum();
    result = isPerfect();

    if(result)  console.log(number + " is perfect");
    else        console.log(number + " is not perfect");

    //取得所有正公约数
    function getFactors(){
        for(i = 1; i < number; i = i+1){
            if(isFactor(i)) factors.push(i);
        }
    }

    //判断是否为公约数
    function isFactor(pontential){
        return number % pontential === 0;
    }

    //对所有正公约数求和
    function aliquoSum(){
        for(i = 0; i < factors.length; i = i+1){
            sum = sum + factors[i];
        }
    }

    //判断是否为完美数
    function isPerfect(){
        return sum === number;
    }

})();
//496 is perfect
```

可以看到我们先用getFactors方法取得1到当前数之间所有数字，然后通过isFactor方法找出可以整除的数记录到factor数组。最后把数组中所有元素相加，判断是否与这个数字相等，如果相等则说明这个数字是完美数。

好，接下来我们看看函数式是如何实现的（这里引用了JavaScript的函数式库 `underscore.js`）。

```
//JavaScript 函数式编程实现完美数

var _ = require("underscore")
, number = 496;

if(
  _.isEqual(number,
    _.reduce(
      _.filter(
        _.range(1, number)
        , function(n){ return number%n == 0 })
      , function(p,n){ return p+n }
      , 0)
    )
  ){
  console.log(number + " is perfect");
  //496 is perfect
}
```

我们从最里层的调用开始看`_.range(1, number)`会帮我们生成一个从1到number的数组，然后我们把这个数组作为参数应用到`filter`函数，这个函数会把表达式 `number%n == 0` 为真的数字返回，也就是能整除的数字。然后我们把这些过滤后得到的数字传递给`reduce`函数，并通过匿名函数 `function(p,n){ return p+n }` 求和，最后将求和得到的结果传入`isEqual`函数，判断是否与`number`相等。

这里可以看出函数式编程通常将数据作为输入，通过一个接一个的函数进行映射、过滤、折叠处理，最终返回出一个计算结果。

对比命令式编程，函数式代码更加简洁清晰，容易阅读。

针对上述代码，我们还可以通过链式调用进行优化，使代码更加简洁清晰。

```
//JavaScript 通过链式调用使函数式代码更加语义化

var number = 496
, isPerfect = _.chain(_.range(1, 300))
  .filter(function(n){ return 300%n == 0 })
  .reduce(function(p,n){ return p+n }, 0)
  .value();
if(isPerfect) console.log(number, "is perfect");
//496 is perfect
```

我们通过`_chain`函数可以将数组传入管道，然后就可以通过链式调用对数据进行处理，最后调用`value`函数计算最终结果。

高阶函数

在函数类型介绍时已经提到过高阶函数的概念，高阶函数必须至少满足以下两个条件之一：

1. 传入参数为函数类型
2. 返回值为函数类型

简单的说将函数类型当做值一样使用，在函数中传递或返回，这样的函数就是高阶函数。

工具类高阶函数

在介绍其他和高阶函数相关的概念之前，我们先看下针对列表操作而设计的三个最基础的高阶函数。

映射（map）

`map`函数会接受一个集合类型（比如列表、数组、链表）参数和一个函数类型参数，然后将集合中的每一个元素应用到函数，并把运算结果组装成一个新的集合返回。

我们来看下下面这个例子：

```
// Java 把集合中每个元素值加三

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3);
        List<Integer> mappedNumbers = numbers
            .stream()
            .map(i -> i + 3)
            .collect(Collectors.toList());
        System.out.println(mappedNumbers); // [4, 5, 6]
    }
}
```

Java8中加入了stream库来提供对函数式编程的支持，通过stream库可以使用流畅的API来操作数据（类似于之前用underscore.js介绍链式调用使用的chain）。

简单说明下上面的代码：

1. 首先声明一个数组numbers，并通过stream方法打包成提供集合操作的stream对象。
2. 执行map操作，取出数组中每个元素做+3运算，这里用到了Java8的lambda表达式，i就相当于传入的元素。
3. 通过collect方法求值并返回新的数组赋值给mappedNumbers。

接下来对比下纯函数式语言Haskell的map函数

```
-- Haskell map函数  
map (+3) [1, 2, 3]      --[4, 5, 6]
```

同样这里的(+3)是一个函数类型参数，[1, 2, 3]是一个列表（你可以理解为数组，因为Haskell中列表是定长非异质的，这跟Java的数组是一样的）。map的作用就是将列表中的每一个元素都应用到(+3)函数上。

关于上面的(+3)函数，这里想多说明一些细节，其实你也可以像Java8中使用lambda一样，通过匿名函数将(+3)函数细节补全。

```
-- Haskell 匿名函数补全(+3)函数实现  
map (x\ -> x+3) [1, 2, 3]      --[4, 5, 6]
```

而这里直接隐藏掉参数x传入的细节，使代码更加简洁。

过滤 (filter)

filter函数也是接受一个集合类型参数和一个函数类型参数，同map一样将集合中每个元素应用到函数，并将计算结果为真的元素组装成一个新的集合返回。

```
//Java 通过filter返回数组中值小于3的元素

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
        List<Integer> filteredNumbers = numbers
            .stream()
            .filter(i -> i < 3)
            .collect(Collectors.toList());
        System.out.println(filteredNumbers);      // [1, 2]
    }
}
```

和map函数执行类似，唯一不同的是map会返回所有计算后的元素，而filter只返满足计算结果的元素。

同样haskell中也是接受一个列表[1, 2, 3, 4]并将其中每个元素应用到函数(<3)，最后把计算结果为真的元素组装成新的列表返回。

```
--Haskell filter函数
filter (<3) [1, 2, 3, 4]      -- [1, 2]
```

折叠 (fold/reduce)

折叠的调用相对复杂一些，折叠函数将一个初始值和集合中的第一个元素应用到函数执行第一次累计运算，然后将返回结果与集合中第二个元素应用到函数进行下一次累计运算，如此反复，直到最后一个元素结束累计运算，最终返回一个累计的结果。当没有初始值作为参数传入时，第一次累计运算就会将集合中第一个和第二个元素作为参数，后续运算同上不变。

```
//Java 利用折叠对集合中所有元素求和

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
        Integer total = numbers
            .stream()
            .reduce(0, (i,j) -> i+j);
        System.out.println(total);      //10
    }
}
```

如上reduce方法接受两个参数，第一个是累计初始值，第二个是一个累计函数，这个累计函数又需要两个参数，一个是累计结果，一个是集合类型中的下一个元素。整个方法可以理解为把集合中所有元素依次执行累计运算（这里是加法运算），第一次执行时因为没有累计结果，所以传入了一个缺省值0。

```
--Haskell foldl函数
foldl (+) 0 [1, 2, 3, 4]      --10
```

同理Haskell中0为累计操作的缺省值。

需要额外说明的是，折叠方法一般分为两种，左折叠和右折叠。两者的区别是

1. 左折叠从第一个元素开始累计，右折叠从最后一个开始累计。
2. 左折叠中列表元素作为右操作数参与运算，而右折叠中列表元素作为左操作数参与运算。

关于第二条来看如下例子：

```
foldl (-) 0 [1, 2]      -- -3
foldr (-) 0 [1, 2]      -- -1
```

foldl中拿列表中最左边元素1和缺省值0进行减法运算，1作为右操作数，所以是 $0-1=-1$ ，接着-1作为累计结果和列表下一个元素继续运算，2作为右操作数，即 $-1-2=-3$ ，所以左折叠结果为-3。

foldr中拿列表中最右边元素2和缺省值0进行减法运算，2作为左操作数，所以是 $2 - 0 = 2$ ，接着2作为累计结果和列表上一个元素继续运算，1作为左操作数，即 $1 - 2 = -1$ ，所以右折叠结果为-1。

通常依据将列表中元素作为左操作数还是右操作数来决定是用左折叠还是右折叠，但对于满足交换律的运算，比如加法、乘法，左折叠和右折叠的运算结果是相同的。

另外右折叠因为是从最后一个元素开始累计，所以右折叠可以计算无限列表，但是左折叠却不行。

其他高阶函数

一般提供函数式编程的语言都会提供map、filter、reduce三个函数，对于列表操作，这三个函数就像命令式语言里的for、if语句一样不可或缺。

除此之外还有另外一些高级函数也是很重要的，但是并非所有支持函数式的语言都提供这些函数。

1. flip翻转函数的两个参数
2. max返回两个数字中大的一个
3. repeat无限循环一个数据
4. take取列表前面指定个数元素等

这里单独说明下zipWith函数。

zipWith会将两个列表作为参数按顺序依应用到函数，然后将运算结果组装成一个新的列表返回。

```
-- Haskell zipWith函数
zipWith (*) [1,2,3] [2,3,4]      -- [2, 6, 12]
```

zipWith会拿第一个列表的第一个元素1和第二个列表的第一个元素2应用到函数(*)，并把运算结果作为第一个元素填充到新的列表，重复上述逻辑对后续元素依次执行运算，最终得到列表[2, 6, 12]($1 * 2 = 2$, $2 * 3 = 6$, $3 * 4 = 12$)。

柯里化

柯里化

柯里化的主要目的就是单一化函数参数，也就是通过函数可以作为值传递这种特性，将一个参数与要应用的函数打包成新的函数。通过这种方式将所有函数都封装为单一参数的形式。

比如说上面Haskell中常见的加法运算正常情况下 `(+) 1 2` 应该这样执行，`(+)`是函数名（haskell中加法运算符是个函数）而1和2是两个参数。实际上haskell所有函数都是单参数的，其执行过程是函数`(+)`和参数1组合成一个新的函数`(+1)`，然后参数2再应用到函数`(+1)`。这就是将多个参数通过柯里化分割为单个参数的过程。

我们用Java重写上面的例子看看：

```
//Java 柯里化单一化函数参数

import java.util.function.IntFunction;
import java.util.function.IntUnaryOperator;

public class Main {
    public static void main(String[] args) {
        IntFunction<IntUnaryOperator> curriedAdd = a -> b -> a + b;
        IntUnaryOperator adder5 = curriedAdd.apply(5);

        System.out.println(adder5.applyAsInt(4)); //9
        System.out.println(curriedAdd.apply(5).applyAsInt(6)); //11
    }
}
```

简单说明下上述代码：

- 首先声明了一个参数和返回类型都为Int的函数接口curriedAdd，在lambda表达式 `a -> b -> a + b` 中a和b表示参数，`a+b`表示返回值，每一个参数或返回值之间都用符号`->`链接。
- 在curriedAdd的基础上应用了一个参数5声明了一个参数和返回值都为Int的函数接口adder5。
- adder5就是柯里化后的函数，当一个参数作用于它时，他会把5作为默认参数和接收到的参数应用到curriedAdd然后执行加法操作并返回运算结果。

部分施用

和柯里化类似，部分施用也是用来减少参数个数的，不同的是部分施用是通过提取代入一部分参数值，使一个多参数函数变为较少数目参数的函数，而不是必须转换为单参数的函数。

我们来看下下面这个`a+b+c`的函数，是如何通过部分施用将函数和一个参数打包成一个需要传入两个参数的新函数的。

```
//Java 部分施用封装部分参数

@FunctionalInterface
interface TriFunction<T, U, V, R> {
    R apply(T a, U b, V c);
}

public class Main {
    public static int add(int x, int y, int z) {
        return x + y + z;
    }

    public static <T, U, V, R> BiFunction<U, V, R> partial(TriFunction<T, U, V, R> f,
    T x) {
        return (y, z) -> f.apply(x, y, z);
    }

    public static void main(String[] args) {
        BiFunction<Integer, Integer, Integer> partialFunction = partial(Main::add, 1);

        System.out.print(partialFunction.apply(2,3));           //6
        System.out.print(partial(Main::add, 2).apply(3,4));   //9
    }
}
```

简单说明下上述代码：

1. 首先定义一个接口TriFunction，这个接口接受三个参数T、U、V并返回R，这里R是一个函数。
2. 声明一个原始的a+b+c方法add。
3. 声明了一个返回了BiFunction接口（它和自定义的TriFunction接口类似，接受两个参数U和V并返回R）的方法partial，它接受两个参数TriFunction和T，其中TriFunction是一个具有默认函数的接口，T是TriFunction的第一个参数。同样partial方法也返回一个函数（接口），这个函数需要两个参数y和z。
4. 通过partial构造一个新的方法partialFunction，在构造这个方法时需传入要打包的方法和它的一个参数。之后我们就可以通过打包后的函数，传入其他两个参数来等价执行add方法了。
5. 我们也可以像 `partial(Main::add, 2).apply(3,4)` 这样不显示的指定打包函数，就像之前的stream操作一样支持连续传递参数。

细心观察的话会发现我在上述说明中用到 `函数（接口）` 这样的说明，这是因为在Java8中函数仍旧不能作为值传递和返回，诸如TriFunction、BiFunction这些函数接口，本质上是一种接口而不是函数，实际上返回的仍旧是接口类型。只不过Java8中新增了接口的默认函数，这些函数接口都包含一个默认函数，而接口类型的apply方法正是执行了默认函数。因此看起来这个函数接口就像一个函数一样在运作。

总之我们还是可以通过打包函数的方式将一个需要传递三个参数的函数调用拆分为两次，第一次传递一个参数，第二次传递两个参数。这样通过函数打包的方式来减少参数个数的方式就是部分施用。

总结

最后我们还是以Haskell为例来对比下普通的函数调用、部分施用和柯里化间的区别。

这里我们定义一个函数，接受三个参数，第一个参数表示一个区间的下限，第二个参数表示这个区间的上限，第三个参数用来判断是否在这个区间，并把比较结果以字符串的方式返回。

```
-- Haskell inRange函数判断数字是否在区间内

inRange :: Int -> Int -> Int -> String
inRange a b c
| a > c = "small than minimum"
| b < c = "large than maximum"
| otherwise = "in range"
```

不用担心上面的语法，你只要记得这个inRange函数的作用就可以，我们主要来看下面几种调用：

```
(inRange 1 10 5)          --in range
((inRange 1 10) -2)        --small than minimum
(((inRange 1) 10) 12)      --large than maximum
```

第一个调用我们打包了一个函数，这个函数传入了三个参数，这有点像我们在命令式编程里书写的代码，拿到三个参数后比较，然后分情况返回结果。

第二个调用我们打包了两个函数，第一个函数接受两个参数用来生成一个区间，第二个函数接受一个参数作为匹配值，这就像部分施用，我们通过打包函数的方式分组或者简化了参数个数。

第三个调用，每次传入一个参数，我们把每次传入的参数记录起来，并在最后一个函数中执行所有的判断，这就是柯里化。

当然Haskell是自动柯里化的，也就是说无论你函数的参数列表定义了多少参数，实际执行的时候都会被抽离成单参数的形式，也就是说第一个调用及第二个调用实际上和第三个调用的执行过程是一样的。我这里这么说明只是帮你区分部分施用和柯里化的区别。

函数组合

如果说柯里化是将一整个函数拆分成一段一段的单个函数，然后依次传参执行。那么函数组合更像是反过来的过程，是将几个函数组合成一个新的函数然后传参执行。不过不管是柯里化还是函数组合最终都是打包成一个只接收一个参数的函数。

柯里化是把参数传入第一个函数，执行后返回一个新的函数，然后把第二个参数应用到这个新函数，依次；而函数组合是把参数传递给第一个函数，执行后返回一个值，然后把值传入第二个函数，依次。所以柯里化看起来是一个函数连续调用了几个参数，而函数组合是多个函数嵌套调用了一个参数。

在数学中 $f(g(x))$ 可以写作 $f \cdot g(x)$ ，同样为了便于书写和阅读，Haskell也提供了 $f.g(x)$ 这样的表达方式。

我们通过加法函数的嵌套来说明下函数组合。

```
-- Haskell 函数嵌套  
(+3)((+5) 1)      -- 9
```

上面是正常的函数嵌套，首先1应用到 $(+5)$ 函数，返回结果为6，然后6应用到 $(+3)$ 函数，返回结果9。

接下来看下函数组合如何表达

```
-- Haskell 函数组合  
(+3).(+5) $ 1      -- 9
```

上面我们用`.`将 $(+3)$ 和 $(+5)$ 打包成了一个新函数，方便理解我们把这个新函数叫做 $(+3).(+5)$ ，然后我们通过`$`运算符指明将1应用到新函数 $(+3).(+5)$ 。

函数组合本身还是按照正常的函数调用顺序执行，也就是说上面的例子其实还是1先应用到 $(+5)$ 返回6， $(+3).(+5)$ 这种写法只是为了方便表述和阅读。

lambda表达式

我们在讲匿名函数的时候已经提到过lambda，其实你也可以把lambda看做是一次性的匿名函数（至少Haskell是如此的）。

我们来看一个例子，对比一下lambda的作用：

假如我们想要计算一个数组中所有数字的平方和是多少，在不使用lambda的情况下我们必须先要给出一个求平方和的函数作为foldl的一个参数。

```
-- Haskell 求平方和函数

squareSum:: Num a => a->a->a
squareSum a b = a + b ^2

foldl squareSum 0 [1,2,3,4]      --30
```

但是这个平方和函数不容易维护，如果我现在想要的操作是立方和，那么我出于语义的方便还需要修改squareSum的名字，而且这个看起来也并不是很简洁。

替换为lambda表达式：

```
-- Haskell 匿名函数替换平方和函数
foldl (\a,b -> a+b^2) 0 [1,2,3,4]      --30
```

在haskell中\用来声明一个lambda表达式，因为它看起来像是 λ 所以叫做lambda表达式。

同样的例子我们来看下Java版的实现。

```
//Java lambda表达式

public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
    Integer total = numbers
        .stream()
        .reduce(0, (i,j) -> i+j*j);
    System.out.println(total);      //30
}
```

可以看到和Haskell类似 $(i,j) \rightarrow i+j*j$ 就是一个lambda表达式，我们会传入i和j两个参数，并返回 $i+j*j$ 。

其他概念

闭包

闭包 (closure) 是函数及其相关引用环境组合而成的一个整体，为了方便解释，你可以理解为闭包就是一个容器，在这个容器中包含一个函数，和几个参数，函数可以像访问自己内部定义的变量一样访问容器中的参数。

先看这段代码

```
-- Haskell map' 函数

map' list = map (+3) list
map' [1,2,3]      -- [4,5,6]
```

这是一个高阶函数，我们把列表中每一项都应用了(+3)函数，我们对它进行抽象，让他变成 `map (+x) list`。

```
-- Haskell map' 函数闭包应用

map' x list = map (+x) list
map' 3 [1,2,3]      -- [4,5,6]
```

做了如上改进，我们让`map'`传入两个参数，一个是加法运算的操作数另一个是被操作的列表，其中`map`就是一个容器，`(+x)`就是容器中的函数。接下来在`map'`得到参数`x`为3时，就变成了 `map (+3) list`，我们再把`(+3)`用匿名函数的方式展开就变成了：`map (\x->x+3) list`，这时我们看到3变成了函数 `(\x->x+3)` 中的一部分，而3本身不是函数中定义的，而是通过容器`map'`传入的。

这种函数自由访问容器中外部变量的特性就是闭包，整个容器就是这个函数和它外部变量的 闭包。

递归

在命令式编程中我们通常是通过索引来获取数组中的元素，但在函数式中我们通常会将列表分成头和尾两部分，然后通过递归尾部（将尾部再次分为头和尾，尾部越来越短直至为空）来访问列表中的每一个元素。

```
-- Haskell 通过递归将列表中元素取反

negatedEvery :: Num a=>[a]->[a]
negatedEvery [] = []
negatedEvery (x:xs) = -x : negatedEvery xs

negatedEvery [1,2,3,4]      -- [-1,-2,-3,-4]
```

上面代码中函数negatedEvery作用是将列表元素取反。

首先我们看第三行，`x:xs`是将列表拆分为头部`x`和尾部`xs`，然后分别对`x`和`xs`赋值。此时头部元素是一个数值，所以可以直接对其取反，而尾部元素仍旧是一个列表，所以尾部将作为一个新的列表重新应用到`negatedEvery`函数。

然后我们看第二行，这里表示如果元组为空则返回空。也就是说当我们不断递归`xs`到`negatedEvery`函数时，只要列表不是无限的终究会执行到`negatedEvery [] = []`，此时列表将执行`[] = []`退出递归，结束函数执行。最终返回一个所有元素都执行了取反运算的列表。

接下来我们用递归来解决斐波那契数列问题，斐波那契数列满足如下计算规则：

1. $F(0) = 1$
2. $F(1) = 1$
3. $F(n) = F(n-2) + F(n-1)$

斐波那契数列本身就是典型的递归实现，我们根据计算规则可以明确三种情况：

1. $n=0$ 时值为0
2. $n=1$ 时值为1
3. n 为其他值时，递归 $F(n-1)+F(n-2)$

先用Java实现如上逻辑：

```
//Java 通过递归实现斐波那契数列

class Recursive {
    static LongUnaryOperator func =
        x -> (x == 1 || x == 0) ? x :
        Recursive.func.applyAsLong(x - 1) + Recursive.func.applyAsLong(x - 2);

    public static long fibonacci(int n){
        return func.applyAsLong(n);
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.print(Recursive.fibonacci(30));
    }
}
```

简单说明上述代码：

- 首先声明一个LongUnaryOperator函数接口，这个函数接口约定传入参数和返回类型都是long类型。
- 接着用lambda表达式对输入参数x应用三元运算符，如果x值为0或者1则返回1，否则返回 $F(n-1) + F(n-2)$ 。
- 最后声明一个取斐波那契数的方法，将传入参数转换为long类型应用到之前声明的 LongUnaryOperator函数接口。

同理Haskell的实现方式类似：

```
--Haskell 实现斐波那契数列

fibonacci::Integer->Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

尾调用

尾调用是指一个函数里的最后一个动作是一个函数调用，这个调用的返回将作为函数的返回值返回。如果函数在最后调用了自身，则称作尾递归。

通常情况下函数在调用时会记录当前调用位置以及内部变量等信息，但是尾调用函数本身就是函数的返回值，所以记录尾调用函数的调用位置和内部变量等信息是没有必要的。针对尾调用，不记录调用位置及内部变量等信息就可以减少栈空间的使用。

单纯的尾调用即使存储了不必要的调用信息也不会浪费太多栈内存，但是如果是尾递归调用的话就有区别了，我们来对比看下记录调用信息和不记录调用信息两种情况下栈的变化。

先看下这段没采用尾调用代码。

```
--Haskell 阶乘函数

factorial 0 = 1
factorial n = n * factorial(n-1)

factorial 3
```

```
--Haskell 阶乘函数运算过程
```

```
factorial 3
3 * (factorial 2)
3 * (2 * (factorial 1))
3 * (2 * (1 * (factorial 0)))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
```

每次调用需要记录函数调用位置，以及内部变量，然后再依次求值，释放对应空间。

用尾递归重写上面的代码：

```
--Haskell 尾递归重写阶乘函数

tail_factorial x = factorial' x 1 where
  factorial' 1 y = y
  factorial' x y = factorial' (x-1) $! (x*y)

tail_factorial 3
```

尾递归优化后，只需要记录当前函数下的内部变量，不需要花费更多额外的空间记录调用函数的位置及调用函数的内部变量。

```
-- Haskell 阶乘函数尾递归优化后运算过程

factorial' 3 1
factorial' 2 3
factorial' 1 6
6
```

再Haskell中尾递归优化是默认支持的，而像Java、Ruby、Python等这些面向对象语言，解释器并没有针对尾递归调用进行优化处理，不过Java8提供函数式支持后有针对尾递归进行优化，ES6标准也规定了JavaScript语言必须实施尾递归优化。也就是说在这些后续支持了尾递归优化的语言，在使用尾递归处理问题时不会像以前一样再出现栈溢出的问题了。

记忆

记忆的作用是用来存储一个函数的返回值，因为函数式的不变性，输入相同的参数必然返回相同的计算结果，正是为了避免重复计算同一个函数调用而引入了记忆机制。

通常我们会把输入参数作为key，把输出结果作为value存在一个缓存结构中（哈希或者元组，函数式用元组来存储key-value）中，在函数调用时先遍历这个缓存结构中是否存在当前调用的key，如果存在则返回缓存中的value，不存在则计算，并插入这对键值到缓存结构。

我们还是以斐波那契数列为例说明，在不使用缓存机制时斐波那契数列的计算是非常慢的。

```
-- Haskell 不使用记忆，执行斐波那契函数

fibonacci 30
{-
  832040
  (2.09 secs, 554,439,712 bytes)
-}
```

添加记忆机制后

```
--Haskell 添加记忆后执行斐波那契函数

memoized_fib :: Int -> Integer
memoized_fib = (map fib [0 ..] !!)
  where fib 0 = 0
        fib 1 = 1
        fib n = memoized_fib (n-2) + memoized_fib (n-1)

memoized_fib 30
{-
 832040
(0.01 secs, 100,520 bytes)
-}
```

这样每当执行memoized_fib函数时都会把计算结果缓存起来，下次执行的时候就可以省去计算直接使用了。因为斐波那契数列存在大量的重复运算，所以缓存的添加让执行效率有了质的改变。

来看下Java8版本的实现：

```
//Java 添加记忆重写斐波那契数列

public class Main {
    private static Map<Integer, Long> memo = new HashMap<>();
    static {
        memo.put(0, 0L);
        memo.put(1, 1L);
    }

    public static long fibonacci(int x) {
        return memo.computeIfAbsent(x, n -> (fibonacci(n-1) + fibonacci(n-2)));
    }

    public static void main(String[] args) {
        System.out.print(fibonacci(30));
    }
}
```

我们重点来看fibonacci这个函数，它调用了memo的computeIfAbsent方法，这里computeIfAbsent是Java8对Map接口新增的方法，它传入两个参数，一个是键值，另一个是函数接口。当key（这里key就是x）值不在map中时会将key作为键，以key为参数应用到函数运算得到的返回值作为value追加到map中。

最后说明下因为lambda表达式中没有区分当x=0和x=1这两种特殊情况，所以我们干脆直接在map中添加key为0和1的值，这样当x=0或者x=1时就避开了错误的函数调用。

惰性求值

惰性求值是指表达式不会在它被绑定到变量之后就立刻求值，而是在该值被真正使用的时候才求值，惰性求值有两个优点：

1. 可以提高计算性能
2. 能构造无限列表（因为惰性求值只关心列表中要求算的那个数，并不关心后面是否还有其他数据）。

某些编程语言默认就是惰性求值的，比如Haskell，另外也有些语言提供了惰性求值的函数或语法，比如Java8的stream操作就是惰性求值的。

我们先通过haskell看一个无限列表的例子。

假设我们有这样一个需求：求所有三的倍数的数字之和

```
--Haskell 惰性求值实现无限列表

let mod3 = filter (\x -> mod x 3 == 0) [1..]
take 5 mod3    --[3, 6, 9, 12, 15]
```

我们创建了一个mod3函数，这个函数过滤出所有三的倍数（但不会马上计算）。之后用take先后取了前五个三的倍数，因为惰性求值的缘故，mod3此时才会计算，并且只会计算到第五个元素。这就是一个惰性求值实现无限列表求值的例子。

接下来我们看看Java8中stream是如何表现出惰性求值的。

```
//Java stream惰性求值实例

public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    numbers.stream().filter(n-> {
        System.out.print(n);
        return n > 2;
    });
}
```

执行上面这段代码不会输出任何结果，说明filter内的函数并没有执行。这是因为stream是惰性求值的，如果想得到链式操作的返回结果需要在最后执行一个及时求值的方法。

```
//Java 通过及时求值函数计算stream返回值

public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
    List<Integer> results = numbers.stream().filter(n-> {
        System.out.print(n); //=>12345
        return n > 2;
    }).collect(Collectors.toList());
}
```

和之前不同我们在Stream操作最后调用了collect(Collectors.toList())方法，这是一个及时求值的方法，它会把之前的表达式执行并返回结果，因此这时 `System.out.print(n);` 就会把numbers的所有元素都打印出来了。

不相交联合体（disjoint union）

Either

在函数式编程中经常会遇到需要返回两种不同类型的情况，为了满足这种需求设计了不相交联合体。不相交联合体可以存储两种不同类型中的某一种类型。一般会用Either类来表述这种结构，Either有一个左值和一个右值，但是左值和右值只能选择一个。

```
--Haskell 左值和右值

Left "This is left value"
Right 2
```

Either类型另外一个重要的作用就是处理异常，很多函数式语言是没有异常处理机制的，因为异常的设计和函数式语言无副作用的思想是冲突的，异常是有副作用的，同时异常会引导程序进入异常处理流程而不是期望的返回值。而函数式编程是以值为根本的，所以在函数式语言中通过把异常信息记录在Either类型的左值中，正确值记录在Either类型的右值中来避免异常产生的副作用。

```
--Haskell 通过Either包装异常信息

div3:: Float->Float->Either String Float
div3 x 0 = Left "Divison by zero"
div3 x y = Right (x / y)

div3 3 0    --Exception: divide by zero
div3 3 3    --1
```

上面定义了一个除3的方法，当除数为0时，返回左值，给出错误信息 除数为0，其他情况下执行除运算并返回右值。

Java本身没有提供Either类，开发人员可以通过这种思想构造Either类，其他基于函数式语言诸如Scala本身是内置Either类的。

Maybe/Optional

Maybe(Java/Scala中称作Optional)类型和Either类型类似，你可以把它看做是一种简单的异常场景，它也存储两个值，一个为空，另一个为有效值。你可以认为Maybe类型只对返回结果做了失败还是成功的判断，如果失败则返回空，成功则返回计算结果的有效值。

```
--Haskell Maybe类型

returnEven:: Int -> Maybe Int
returnEven a
| a `mod` 2 == 0 = Just a
| otherwise = Nothing

returnEven 2      --Just 2
returnEven 1      --Nothing
```

如上定义了一个返回偶数的函数returnEven，函数返回了一个Maybe类型。如果计算结果为偶数则返回值（Just是Haskell对Maybe类型中有效值的包装，这里不用在意），否则返回Nothing。

Java8也提供了Optional类型用来代替null值。通过Optional代替null表示值不存在，可以避免一些不必要的麻烦。

Optional也可以用来判断一个变量是否存在值。

```
//Java Optional使用

Optional emptyOptional = Optional.empty();
Optional<String> a = Optional.of("a");

System.out.print(emptyOptional.get());
//java.util.NoSuchElementException: No value present

System.out.print(a.get());
//a
```

纯度

我之前有在介绍函数式特点的时候有提及过纯度（purely）这个概念，纯度的特性有一定的优点但也存在很多问题，比如很多操作（混合运算）并非是顺序无关的，这些操作是有副作用的（输入结果相同，函数执行顺序不同，返回结果不同），因此对于纯函数式的语言而言就要面对一些棘手的问题（但并非所有支持函数式编程的语言都是纯函数式的）。

我们来通过一个过平衡木的例子来说明Haskell是如何处理因函数执行顺序而产生的副作用的。

平衡木游戏的规则如下：

1. 用向左倾斜和向右倾斜的程度表示一个人的平衡程度，向左倾斜用负数表示，向右倾斜用正数表示
2. 向左平衡程度和向右平衡程度的和的绝对值表示平衡值，如果平衡程度值小于3则人是平衡的
3. 人每向前移动一步，则随机追加左右倾斜值，计算平衡值。将平衡值传给下一步

```
-- Haskell 平衡木游戏

type Rate = Int
type Balance = (Rate, Rate)

toLeft :: Rate -> Balance -> Balance
toLeft n (left, right) = (left - n, right)

toRight :: Rate -> Balance -> Balance
toRight n (left, right) = (left, right + n)
```

我们用Rate表示平衡倾斜程度，用Balance表示平衡程度，其中left指向左倾斜程度，right指向右倾斜程度。这样我们定义了向左倾斜函数toLeft和向右倾斜函数toRight，这两个函数都要求输入当前平衡度及平衡倾斜数，然后返回新的平衡度。

我们做如下调用，向左倾斜1 -> 向右倾斜1 -> 向左倾斜2

```
-- Haskell 平衡木游戏
toLeft 2 (toRight 1 (toLeft 1 (0, 0)))    -- (-3, 1)
```

到目前为止是没有问题的，我们在这个基础上继续，向右倾斜1 -> 向左倾斜3 -> 向右倾斜2

```
toRight 1 (toLeft 3 (toRight 2 (-3, 1)))    -- (-6, 4)
```

这时返回的结果仍旧是保持平衡的： $|-6+4|<3$ 。但实际上在向左倾斜3的时候就已经失去平衡而摔倒了，所以是不可能再向右倾斜2恢复到平衡状态的（此时人已经掉下平衡木了）。

那么我们需要在失去平衡的时候返回一个失败信息，并且这个信息会一致传递下去。

我们做如下修改：

```
-- Haskell 平衡木游戏

type Rate = Int
type Balance = (Rate, Rate)

toLeft :: Rate -> Balance -> Maybe Balance
toLeft n (left, right)
| abs (left -n + right) < 3 = Just (left-n, right)
| otherwise = Nothing

toRight :: Rate -> Balance -> Maybe Balance
toRight n (left, right)
| abs (left + right + n) < 3 = Just (left, right+n)
| otherwise = Nothing
```

我们对要传递给下一步的平衡状态包装为Maybe类型，如果失去平衡则返回Nothing，否则返回Just。这样当我们接收到Nothing的时候就只会传递Nothing到下一步了。

为了让一个值应用到函数并返回一个包装类型我们要用到monad（就是下面的 `>>=` 函数）。

```
return (0,0) >>= toLeft 2 >>= toRight 1 >>= toLeft 1      --Just(-3,1)
return (-3,1) >>= toRight 1 >>= toLeft 3 >>= toRight 2      --Nothing
```

可以看到这回返回了“摔倒”，同时monad不仅将值应用到了函数包装类型，同时还提供了函数的链式调用，这种写法更加干净易懂。

另外，针对 `toRight 1 (toLeft 3 (toRight 2 (-3,1)))` 这种嵌套调用，我们也可以定义一个 `-:` 方法实现函数的链式调用。

```
-- Haskell 实现链式调用

x :- f = f x
(0,0) -: toRight 1 -: toLeft 3 -: toRight 2      --(-3,3)
```

通过上面的例子可以看到为了处理产生副作用的调用，Haskell提供了诸如Maybe、Monad这样的概念。而对于允许副作用的语言处理其相同的问题就简单容易的多，这也是纯函数式对比命令式的一个缺点。