Java、C++和 Python 的 OO 特性调研报告

（中国工程物理研究院　赖辉源）

面向对象编程（Object Oriented Programming），简称 OOP，是一种程序设计思想。OOP 把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计是把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度，而面向对象的程序设计是把计算机程序视为一组对象的集合，每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

本次调研主要针对于 Java、C++和 Python 这三门面向对象语言，针对于它们的设计思想和面向对象特征进行简单的阐述和比较。

一、Python 简介

Python 是一种面向对象的解释型计算机程序设计语言，它具有丰富和强大的库。它常被昵称为胶水语言，因为它能够把用其他语言制作的各种模块（尤其是 C/C++）很轻松地联结在一起。在 Python 中，一切数据类型都可以视为对象，当然使用者也可以自定义对象，自定义的对象数据类型就是面向对象中的类（Class）的概念。

1.1　Python 的对象

Python 语言中，一切皆对象,这句话概括地说就是在 Python 语言中涉及到的所有东西都是“对象”。比如，函数、各种数值（比如整数值、浮点数值、布尔值）、模块（类似于 Java 的 package）、None（类似于 Java 的空引用 null、C++的空指针 NULL）等都是对象。

但是与之相对比下的 C++和 Java 的语法：只有类的实例才能算得上是对象。连基本类型（比如 int、char、float、等）都算不上对象，至于函数，也就更不是对象了。

1.1.1 对象的属性

Python 的对象都具有若干个属性。我们可以通过内置的 dir()函数进行反射，从而了解到某个对象分别都包含哪些属性，而且 Python 还提供了若干内置的函数，这用于在运行时操作指定对象的属性。具体如下：

```
hasattr(obj, name)          #判断 obj 对象是否具有名为 name 的属性
setattr(obj, name, value)   #设置 obj 对象的名为 name 的属性值为 value
getattr(obj, name)          #获取 obj 对象的名为 name 的属性值
delattr(obj, name)          #删除 obj 对象的名为 name 的属性
```

1.1.2 对象的类型

Python 的所有对象都可以通过内置的 type()函数以获取该对象的类型。这实际上就是 Python 的 RTTI 机制的体现，该函数与 C++的 typeid 关键字和 Java 的的 instanceof 关键字是类似的。

1.1.3 一切皆对象的优势

Java 由于基本类型继承自 Object 类引出不少麻烦，当初 Java 刚开始设计容器类（比如 Vector、 ArrayList 等）的时候，颇费了一番功夫。因为容器里面放置的必须是 Object，为了让容器能适应基本类型，只好给每一种基本类型分别 对应一个派生自 Object 的包装类(Integer 类对应 int,Float 类对应 float 等)；后来又平添了自动装箱/拆箱的概念，而 Python 就没有这方面的困扰。

然后再“反射”的问题上，虽然 Java 语言支持对象的反射，但是 Java 的 package 不是 Object，所以也就无法对 package 进行反射。而 Python，任何一个 module（相当于 Java 的 package）import 之后，都可以直接通过前面提到的 dir()函数进行反射，得知该 module 包含了哪些东西，我们只需要 2 行代码，如：

```
import xxx
dir(xxx)
```

1.2 封装（Encapsulation）

为了避免歧义，以及叙述方便，我们把 OOP 的封装，分为狭义和广义两种进行讨论。

1.2.1 广义的封装

OOP 强调以数据为中心，所以 OOP 的广义封装，就是把数据和操作数据的行

为，打包到一起。比如 C++/Java 里的 class，可以同时包含数据成员和函数成员，就算是满足广义的封装了。对于 Python 而言，其 class 关键字类似于 C++和 Java，也已经具有广义的封装性了。

1.2.2 狭义封装

OOP 的狭义封装，则更进一步，增加了信息隐藏（Information Hiding)。比如 C++和 Java 的 public、protected、private 关键字，就是通过访问控制来达到信息隐藏的效果。Python 虽然没有针对访问控制的关键字来修饰类成员，但是 Python 采用了另外一套机制，即根据命名来约定。在 Python 的对象中，如果某个属性以双下划线开头来命名（比如 __name)，就可以起到类似于 private 的效果。

1.3 继承（Inheritance）

1.3.1 Python 的继承

Python 没有像 Java 那样，区分出类继承（OO 的术语中也叫“实现继承”)、接口继承；也没有像 C++那样，区分出公有继承、私有继承、保护继承等。Python 就只有一种继承方式。

1.3.2 继承的语法

Python 的继承语法，相比 C++/Java 而言，也更加简洁。比如子类 Child 需要继承父类 Parent，代码只需如下：

```
class Child(Parent):
```

如果是多继承，代码大同小异：

```
class Child(Parent1, Parent2, Parent3):
```

如果想知道某个类有哪些父类（基类)，只需要通过 Child.__bases__便可知晓。

1.3.3 继承的动态性

作为一种动态语言，Python 可以在运行时修改类的继承关系。这个特性可以说是 C++/Java 所望尘莫及的。比如下面的例子：

```
class Parent1 :
    def dump(self) :
        print("parent1")
```

```
class Parent2 :
    def dump(self) :
        print("parent2")

class Child :
    def dump(self) :
        print("child")

print(Child.__bases__)
Child.__bases__ += (Parent1, Parent2) # 动态追加了 2 个父类
print(Child.__bases__)                # 打印出的父类信息中，已经包含 Parent1、Parent2
```

1.3 多态（Polymorphism）

Python 的多态，和传统的 OO 语言类似，没有太多的不同之处，用上述的 3 个类作为例子，然后再另外增加一个 test()函数如下：

```
def test(obj) :
    obj.dump()
```

然后对 test()函数分别传入不同的类型的对象。

```
c = Child()
test(c)                               # 打印出 child
p1 = Parent1()
test(p1)                              # 打印出 parent1
```

由于对 Python 所知甚少，以上内容主要是从一些资料里收集整理而得，仅仅对 Python 在面向对象方面的特性有一个粗浅的认识和描述。

二、C++简介

C++是 C 语言的继承，它既可以进行 C 语言的过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。C++擅长面向对象程序设计的同时，还可以进行基于过程的程序设计，因而 C++就适应的问题规模而论，大小由之。 C++不仅拥有计算机高效运行的实用性特征，同时还致力于提高大规模程序的编程质量与程序设计

语言的问题描述能力。

2.1 类的基本概念

2.1.1 类的定义、构造函数

下面是一个 C++类的例子：

```
class User {
  string name;
  int age;
public:
  User(string str, int i) {name = str; age = i;}
  void print() {cout << "name: " << name << " age: " << age;}
};
```

这个类包含两个数据成员，前者的类型是 string，后者是 int，此外还包括一个成员函数（方法）print，用来将该类对象的信息打印出来。我们可以看到在 C++中需要用一个分号来表示类定义的结束，然后写了 User 类的一个构造函数，构造函数是与类同名的特殊成员函数，没有返回值，用来对该类的对象进行初始化。有了这一构造函数，我们就可以用下列两种方法之一来创建这个类的一个对象实例：

```
User u("Bob", 20);
User *p = new User("Bob", 20);
```

第一种方法在栈上为对象 u 分配内存，当 u 离开它的作用域之后，其内存会自动释放。第二种方法在堆上为对象分配内存，并将这块内存的地址赋给指针 p。以后这块地址需要调用 delete 操作符显式地进行释放。

C++允许类的一个成员函数的实现代码可以位于类定义的外部。在 C++中，如果一个成员函数的实现代码是在类定义本身提供的，这个函数便被认为是内联的（inline）。函数是否为内联的会影响编译器执行某些优化措施的能力。通过显式地声明，一个在类定义外部定义的成员函数也可以是内联的。

2.1.2 访问控制

类的每个成员都有一个相关的访问控制属性。在 C++中，成员的访问控制属性是 private（私有）、protected（保护）、public（公共）三者之一。如果没

有显式地声明一个成员的访问控制属性，那么它就按照缺省情况作为这个类的私有成员。因此，在上面User 类的例子中，数据成员 name 和 age 的访问控制属性是 private，而构造函数和方法 print 的访问控制属性是 public。对于某个类的 private 成员，它的名字将只能由该类的成员函数和友元使用。对于某个类的 protected 成员，它的名字只能由该类的成员函数和友元，以及由该类的子类的成员函数和友元使用。而对于某个类的public 成员，它的名字可以由任何函数使用。

2.1.3 对象的复制与赋值

在C++中，按照默认规定，类对象可以进行复制。特别是可以用同一个类的对象的复制对该类的其他对象进行初始化。例如：

```
X  a=b;                                    //通过复制初始化
```

按照默认方式，类对象的复制就是其中各个成员的复制。如果某个类X所需要的不是这种默认方式，那么就可以定义一个复制构造函数 X::X(const X&)，由它提供所需要的行为。类似地，类对象也可以通过赋值进行按默认方式的复制。例如：

```
a=b;
```

其中 a 和 b 都是类 X 的对象。在这里也一样，默认的语义就是按成员复制。如果对于某个类而言这种方式不是正确选择，那么用户可以重载赋值运算符，由它提供所需要的行为。

2.1.4 对象的终结处理

在对象销毁之前，可能需要做一些最后动作，例如释放对象所占用的各种资源，维护有关状态等等。在C++中，终结动作以类的析构函数的形式定义。析构函数的名称就是类名前面加一个波浪号。它不应接受任何参数，也不应返回任何值。对于堆栈上的类的对象，在其作用域退出时，它的析构函数会被自动调用。对于堆上的类的对象，我们需要显式地释放它们所占的存储。在释放之前，析构函数会被自动调用。如果一个类没有显式地提供一个析构函数，那么系统在销毁这个类的对象之前会执行析构函数的缺省行为。这个缺省行为就是依次调用类的每个数据成员的析构函数。

2.2 类继承

2.2.1 子类的定义及相关概念

继承是面向对象程序设计的基本概念之一。通过继承一个已有的类可以对类进行扩展，针对于上述 User 类的例子，接着定义该类的一个子类 StudentUser，即：

```
class StudentUser : public User {
    string school;
public:
    StudentUser(string str, int i, string s) : User(str, i) {
        school = s;}
    void print() {
        User::print();
        cout << " School: " << school; }
};
```

在上述定义的基础上，一般称 User 是一个父类、基类或超类，StudentUser 是一个子类、派生类或扩展类。

2.2.2 C++的继承方式

C++提供了三种继承方式，以控制对基类成员的访问。第一种是 public 继承，也就是通常情况下的继承方式。记子类为 D，基类为 B，那么子类头部的写法是“class D : public B”。在 public 继承中，基类的 public 成员变成子类的 public 成员，protected 成员变成子类的 protected 成员。需要注意的是基类的 private 成员在子类中不可访问。

第二种是 protected 继承，子类头部的写法是“class D : protected B”。在 protected 继承中，基类的 public 成员和 protected 成员都变成派生类的 protected 成员。

第三种是 private 继承，子类头部的写法是“class D : private B”。在 private 继承中，基类 public 成员和 protected 成员都变成派生类的 private 成员。在实际应用中可以根据需要来选择适当的继承方式，根据经验一般 public 继承是最为常用的。

2.3 方法的动态约束

2.3.1 方法覆盖的限制

在 C++中，当一个方法在基类中被声明为虚拟方法时，我们可以在派生类中用一个同名的方法对它进行覆盖。然而，覆盖不是任意的，要受到一些条件的限制。

第一，当基类方法的返回值是基本类型时，派生类的覆盖方法的返回类型必须和基类方法相同；当方法的返回类型是一个指向类 A 类型的指针或引用时，派生类的覆盖方法的返回类型可以是指向类 A 的子类型的指针或引用类型。

第二，基类虚拟方法的访问控制属性对于派生类的覆盖定义是否合法没有影响。即使基类的虚拟方法位于保护或私有部分，派生类的覆盖方法也可以出现在派生类的任何一个部分。

第三，如果基类的虚拟方法带有异常说明，那么派生类的覆盖方法就不允许抛出这个异常说明之外的异常。

2.4 抽象类、接口

一般而言我们是无法从一个抽象类创建对象的。在 C++中，如果一个类的一个或多个成员函数被声明为纯虚函数，那么我们就无法从这个类创建对象。在 Java 中，如果我们对一个类显式地使用关键字 abstract 进行声明，那么我们也不能从这个类创建任何对象。如果我们无法由一个类创建对象，那么这个类有什么用呢？首先，抽象类可以在一个类层次体系中起到组织其它类的作用。其次，抽象类可以表示一种特殊的行为，当它与其它类混合使用时，可以允许我们创建具有这种行为的类。此外，抽象类可以帮助我们以增量的方式创建类的实现代码。

在面向对象程序设计中，抽象类可以作为一个包含了一些具体类的类层次体系的根系，把那些分散封装在各个子类中的共同概念归纳在一起。例如，圆形和矩形等都是形状，我们可以写一个具体类 Circle 来描述圆形，写一个具体类 Rectangle 来描述矩形，我们也可以写一个抽象类 Shape 来描述形状。显然，为 Circle 和 Rectangle 类创建对象是合理的，而为 Shape 类来创建对象就显得没有意义。Shape 类在这里扮演了一个关键的角色，它把其它的类组织到一个单一的类层次体系中。需要说明的是，“组织在一起”这个概念实际上比看上去要复杂的多，它涉及到了继承和动态约束。利用继承，我们可以把各个子类中的

公共代码放在 Shape 类中，使程序的效率更高。假如我们有一个 Shape 变量（引用或指针）的列表，其中有些变量所引用的实际是 Circle 类型的对象，还有一些引用的则是 Rectangle 等类型的对象。我们可以在整个列表中调用一个诸如 area 的方法，动态约束会使列表中每个变量都调用到适当的 area 方法。

下面是是 C++中定义抽象类的一个简单例子。

```
class Shape {
    public:
        virtual double area() = 0;};
```

其中，最后一行的符号“= 0”告诉编译器这个函数是纯虚函数。编译器要求纯虚函数不包含任何实现代码，而如果一个 C++类具有至少一个纯虚函数，那么这个类就是一个抽象类。

3.5 C++的多重继承

C++允许一个类从多个基类继承实现代码，这称为多重继承。在面向对象程序设计中，一个实体可能同时从多个来源继承了属性。在这种情况下，采用多重继承就显得是一件自然的事情。一方面，多重继承是一种功能强大的编程工具；然而另一方面，多重继承会带来一些缺点。在一个类层次结构中，当一个派生类从两条不同的路径继承了某个基类的相同成员时，会导致许多复杂的问题。此外，多重继承也会带来实现方面的困难。

另外 Python 也是支持多重继承的，而需要注意的是在 Java 中不允许一个类从多个基类继承实现代码， Java 允许一个类从多个接口继承各种行为。我们称之为混入式继承。

三、Java 简介

3.1 类的基本概念

3.1.1 类的定义、构造函数

在 Java 中，一个类定义的例子如下:

```
class User {
    private String name;
    private int age;
```

```
    public User(String str, int i) {
        name = str;
        age = i;}
    public void print() {
        System.out.println("name: " + name + " age: " + age);}
}
```

在这个类中，我们定义了两个数据成员。其中，name 成员的类型是 String，这是 Java 的字符串类型。与 C++不同的是，Java 的类定义不需要以一个分号结束。

在上述的类中提供了一个构造函数。有了这一构造函数，我们就可以象下面这样而且只能这样创建这种类型的对象。

```
User  u = new  User("Bob", 20);
```

该表达式右边部分的调用创建了一个 User 类型的新对象并返回对该对象的引用。

3.1.2 访问控制

在 Java 中，限定符 public 和 private 与它们在 C++中的含义相同。至于 protected，含义与 C++中的类似，只不过在同一个程序包中这类成员就像是公共成员一样。也就是说，我们可以在同一个程序包的所有类中访问一个类的保护成员，但是在这个包之外，只有这个类的子类可以访问这些成员。除了 private、protected 和 public 之外，Java 中还有一个访问控制限定符 package。对于 package 成员，在同一个程序包内它们就像是公共成员，而在包的外部它们就像是私有成员。此外，如果我们没有为一个成员指定访问限定符，那么这个成员的访问控制属性就是 package。

3.1.3 对象的复制与赋值

在 Java 中，情况要简单一些。对于语句：X a = b;

其语义并不是对象的复制，而是变量 a 得到了变量 b 所保存的那个对象的引用的一份拷贝。也就是说，在初始化之后，a 和 b 都是对同一个对象的引用。如果我们希望 a 所引用的是 b 所引用的那个对象的一份拷贝，那么我们可以通过调用 clone 方法来实现。该方法是从根类 Object 继承而来的。

3.1.4 对象的终结处理

Java 的对象销毁方式与C++有着很大的不同。因为Java 提供了自动废料收集的机制。在一个 Java 程序中，如果没有任何变量保存了对一个对象的引用，那么这个对象就成为了废料收集的候选目标。在实际销毁一个对象之前，如果该对象存在 finalize 方法，那么废料收集器就会执行其中的代码。Java 的finalize方法所执行的任务类似于C++中程序员所提供的析构函数。然而，执行终结动作的时间是不可预计的，存在时间上和顺序上的不确定性。

3.2 类继承

3.2.1 子类的定义、相关概念

对于上面给出的 User 类的例子，我们可以定义该类的一个子类StudentUser。

```
class StudentUser extends User {
    private String school;
    public StudentUser(String str, int i, String s) {
        super(str, i);
        school = s;
    }
    public void print() {
        super.print();
        System.out.print(" School: " + school); }
}
```

在定义子类时， Java 和 C++中的书写格式有所不同。在 Java 中的写法则是“class StudentUser extends User”。子类能够继承基类的所有成员，但它不能访问基类的私有成员。在上面的例子中，由于name 和age 是基类User 的私有成员，因此子类StudentUser 中的任何成员函数（方法）都无法访问它们。

我们分别通过“User(str, i)”和“super(str, i)”调用了父类的构造函数，对那些只能由基类的成员函数进行访问的数据成员进行赋值。实际上，一个派生类的构造函数在执行任何任务之前必须调用它的基类的构造函数。如果在派生类的构造函数中并没有显式地调用基类的构造函数，系统将会试图调用基类的

无参构造函数（缺省构造函数）。

在上面的例子中，子类 StudentUser 的 print 函数的定义覆盖了它从基类 User 继承而来的同名函数的定义。然而，我们仍然可以访问被隐藏的基类函数，例如 C++中的“User::print();”以及 Java 中的“super.print();”。最后需要指出的是，基类的指针或引用可以安全引用子类的对象，但子类的指针或引用不能引用基类的对象。这一点在 C++和 Java 中是一致的。

3.2.2 Java 中的关键字 final

有的时候，我们会有这样的一种需要，就是希望某个类不能被继承。比如说，编译程序如果预先能够知道一个类不能被扩展，那么它可能会在函数调用的数据访问方式的优化上做得更好。另外，出于安全方面的一些考虑，我们也有可能决定不让用户对销售商所提供的类进行扩展。Java 提供了一种方便的方法，如果我们在一个类的头部以关键字 final 作为前缀，那么这个类就不能被扩展。

在 Java 中，我们也可以只把一个类的其中一些方法声明为 final，从而实现有选择地控制继承的目的。当超类中的一个方法被声明为 final 时，我们就无法在子类中对它进行覆盖。相比之下，C++并没有提供一个类似 final 的关键字来防止一个类被扩展或者有选择地对覆盖机制进行阻断。然而我们知道，一个派生类的构造函数在执行任何任务之前必须调用它的基类的构造函数。因此，如果我们把基类的构造函数放在类的私有部分，那么这个类就不能被扩展

3.3 方法的动态约束

面向对象语言里的方法调用通常采用“x.m(...)”的形式。其中，x 是一个指向或者引用对象的变量，m 是 x 的定义类型的一个方法。如果上述形式调用的方法是根据变量 x 的类型静态确定，我们就称之为静态约束；如果是根据方法调用时被指向或引用的对象的类型确定，我们就称之为动态约束。动态约束是面向对象程序设计的一个非常重要的概念。动态约束带来的优点是明显的，它使得我们能够编写程序来通用化地处理一个层次中的所有类的对象。然而，采用动态约束调用时会带来一些额外开销，如果需要调用的方法能够静态确定的话，采用静态约束有速度优势。动态约束方法的另一个缺点是不能做在线展开（inline）处理。

在 Java 语言里，所有方法都采用动态约束。与此不同的是，C++提供了静

态约束和动态约束两种方式，以静态约束作为默认方式，而把动态约束的方法称为虚方法（virtual 方法）。

C++之所以会这么做，是与它的 C 基础有关。下面是一个简单的 Java 例子来说明方法的动态约束。

```
class User {
    private String name;
    private int age;
    public User(String str, int i) {
        name = str;
        age = i;}
        public void print() {
        System.out.print ("name: " + name + " age: " + age);}
}
class StudentUser extends User {
    private String school;
    public StudentUser(String str, int i, String s) {
        super(str, i);
        school = s;
    }
    public void print() {
        super.print();
        System.out.print(" School: " + school); }
}
class Test {
    public static void main(String[] args) {
        User u = new User("Bob", 20);
        User su = new StudentUser("Chris", 25, "Math");
        u.print();
        System.out.println();
```

```
        su.print();
        System.out.println(); }
}
```

这是一个完整的 Java 程序。在 main 方法中，我们定义了两个 User 类型的变量。变量 u 引用了一个 User 类型的对象，而变量 su 则引用了一个 StudentUser 类型（User 的子类）的对象。我们对变量 u 调用方法 print，与对变量 su 调用方法 print，调用的是不同的方法。这是由于 Java 采用的是方法的动态约束（假如采用的是静态约束的话，调用的就是同一个方法）。该程序输出的结果是：

```
name: Bob age: 20
name: Chris age: 25 School: Math
```

假如采用的是静态约束的话，输出结果中就不会有“School: Math”。

3.3.1 方法覆盖的限制

在 Java 中，关于方法的覆盖也有一些限制。第一，派生类覆盖方法的返回类型必须和基类中被覆盖方法的返回类型相同。与 C++的规定不同，Java 对覆盖方法的返回类型的限制与返回类型是基本类型还是类类型无关。第二，覆盖方法的访问限制不能比基类的被覆盖方法的访问限制更严格。因此，如果基类方法的访问限制属性是保护，那么派生类的覆盖方法的访问限制可以是保护或公共，但不可以是私有。与 C++不同的是，Java 中在类的私有部分声明的方法是不能被覆盖的。第三，派生类中一个覆盖方法的异常说明必须是基类中被覆盖方法的异常说明的一个子集。这一点与 C++中的相应限制是类似的。

3.4 抽象类、接口

Java 要求抽象类的头部以关键字 abstract 开头。同时，如果我们的类包含了任何我们不想提供实现代码的方法，我们在这个方法的声明中也包含这个关键字。

除了抽象类之外，Java 还支持接口的概念。接口是一种只由抽象方法和常量组成的类。Java 中声明接口的语法如下。

```
interface Collection {
    public boolean add(Object o);
```

```
    public boolean remove(Object o); }
```

在接口中声明的方法访问控制类型始终是公共的，即使它们并没有显式地用 public 关键字进行声明也是如此。而且，接口中的方法也不允许声明为静态方法。此外，Java 的接口还可以包含常量。在接口中定义的常量被隐式地作为了 final 和静态处理。当一个类继承了该接口时，这些常量就像是在这个类中局部定义的一样。需要说明一下该例中的类 Object。Java 采用的是单根的类层次结构。也就是说，Java 有一个唯一的根超类，这个类就是 Object，其它所有的类都是由它直接或间接地派生出来的。与此不同的是，C++采用的是任意的类层次结构。

另外由于 C++允许多重继承而 Java 不允许，Java 只允许从一个基类继承实现代码，虽然这个限制消除了多重继承所带来的许多编程难题，但它同时也导致了一些自身的设计限制，不得不使用接口的概念予以解决。一个 Java 类只可以是一个基类的子类，却可以是任意多个接口的子类。

四、结语

本文旨在讨论 Python、C++与 Java 这三种语言的基本语言特征和面向对象机制方面的异同。由于个人水平所限与知之甚少，文中只是讨论了这些语言中非常浅层的一部分，并且还有许多方面未涉及到的，例如异常处理等等方面。

Python、C++和 Java 都是当前非常流行的面向对象程序设计语言，它们在许多方面都有着值得比较和称赞的地方。Python 由于开源以及语法简单等优势，在广大的工作者中广受欢迎，使用者越来越多，C++的许多规定都比较灵活，甚至把一些细节留给了实现，而 Java 的设计追求安全性和平台无关性，因而 Java 几乎把所有的问题都规定得很严格，没有什么更改的余地，这三门编程语言都有自身的独特之处，并且在自己擅长的领域里起着重要的作用和地位，它们都是非常优秀的程序设计语言。