# Thinking in Java chapter7 笔记和习题

emacsun

目录



像 Java 这样的大型程序语言，每一次升级都牵涉甚广。语言的设计者希望修改语言的某些部分使其效率更高，而语言的使用者希望接口保持不变，从而保证之前的代码能够正常运行。这就产生了矛盾：语言设计者希望修改语言，语言使用者倾向于维持接口不变。于是，语言的设计者必须保证在修改了某些部分之后保持接口不变。对语言使用者来说，语言设计者的修改是透明的。

本章介绍 Java 是如何控制访问权限，从而达到 Java 在升级换代过程中维持函数接口不变的。按照访问权限从高到低排序， Java 提供了 package , public , protected , private 这四个级别的访问控制。其中包访问规定了一组类是如何在一个库中打包的， public protected private 这三个关键词规定了类成员的访问方式。



# 1 package：库的最小单元

一个 package 包含了一组类，这些类在一个命名空间（namespace）下。Java 语言本身内置了很多可用的库，比如 utility 库，这个库在命名空间 java.util 。 java.util 的一个类叫做 ArrayList ，那么我们如何才能使用 ArrayList 呢？一种可行的方法是使用全名 java.util.ArrayList ，看代码：

```
//: access/FullQualification.java
public class FullQualification {
  public static void main(String[] args) {
    java.util.ArrayList list = new java.util.ArrayList();
  }
} ///:~
```

然而，使用全名的方法，一看就很烦，名字又臭又长。 Java 提供了一种简单的做法 import 。如果你想要 import 某个特定的类，可以使用 import 语句，看代码：

```
//: access/SingleImport.java
import java.util.ArrayList;
public class SingleImport {
  public static void main(String[] args) {
    ArrayList list = new java.util.ArrayList();
  }
} ///:~
```

现在开心了，因为使用 ArrayList 时，不用重复臭长的前缀。但是这样用一个类 import 一个类的方法也不是最有效的，尤其是当我们要用到 java.util 下的好多个类的时候。此时， * 是一个方便的符号，看：

```
import java.util.*
```

这句话把所有的 java.util 下的类都导入了。

import 的做法提供了一种命名空间的管理机制。首先自定义类成员的名字是互相隔离的，比如类 A 的方法 f() 和类 B 的方法 f() 不会造成冲突（即使他们有相同的签名）。那么是不是类成员的名字做到互相隔离就够了呢？不！假设你创建了一个 Stack 类，另外一个人也创建了一个类 Stack ，这个时候怎么办？ Java 为这种潜在的冲突提供了完整的解决方案。

截至目前我们都还没有碰到类名冲突的情况，那是因为我们的代码都很简单，当你要和别人一起合作完成较大的项目的时候，做好命名空间的保护是非常必要的。当你编写了一个 Java 文件，这个文件是一个基本的编译单元（compilation unit ，有时候也叫 translation unit ）。每一个编译单元都以



.java 结尾。在这个编译单元中一定有一个 public 类，这个类名必须和文件名字一样(包括大小写！)。 **在一个编译单元中必须只有一个 public 类，否则的话，编译器会出警告或者报错** 如果在这个编译单元中有多个类，外界是不会知道的，因为他们不是 public 的。这些类为 public 的类提供支持，是幕后的螺丝钉。

## 2 组织你的代码

当你编译 .java 文件时，编译器会为这个文件中的每一个类创建一个文件，这个文件以 .class 结尾。因此编译结束后可能会有相当多的 .class 文件出现在文件夹下面。对于 Java 来说一个可执行的程序就是一堆 .class ，这些 .class 可以通过打包程序生成 jar 文件。在执行的过程中， java 解释器搜寻，装载，解释这些文件。

一个库( library )是一组类文件，每一个源文件包含一个 public 类和任何数量的非 public 类（再次强调，一个文件只有一个 public 类）。如果想要把所有的这些编译单元（也可以说是这些文件）组织在一起，说明他们是一伙的呢？ package 关键字就是做这个活的。

在编译单元（不知道我为什么非要称一个源文件为编译单元？好吧，在以后编译单元和一个源文件是等效的。）中使用 package 时，必须把 package 放置到文件的头部，也就是说除了注释意外的第一行。当你说:

```
package access;
```

你在告诉编译器，你在告诉别的程序员：这个编译单元属于一个叫 access 的库。换句话说，这个编译单元的 public 类属于库 access 。这个类的名字被 access 罩着，任何人想要使用这个类都要使用全名或者 import 语句导入这个类，否则无法访问这个类。

举个例子，假设文件名是 MyClass.java 这意味着这个文件中一定有一个 public 类名字是 MyClass 。看代码：

```
//: access/mypackage/MyClass.java
package access.mypackage;
public class MyClass {
  // ...
} ///:~
```

假若现在有人要用 MyClass 或者 access 中的其他类，他必须用 import 导入或者全名引用，从而使得 access 可以被访问到。



全名引用的做法如下：

```
//: access/QualifiedMyClass.java
public class QualifiedMyClass {
  public static void main(String[] args) {
    access.mypackage.MyClass m =
      new access.mypackage.MyClass();
  }
} ///:~
```

使用 import 的做法如下：

```
//: access/ImportedMyClass.java
import access.mypackage.*;
public class ImportedMyClass {
  public static void main(String[] args) {
    MyClass m = new MyClass();
  }
} ///:~
```

package 和 import 可以保证不管多少人在从事同一个 Java 项目都不会造成名字冲突。

## 3 创建独一无二的 package 名字

之前我们看到 package 是如何打包库的：一个 package 不是把所有的文件合并成一个文件，相反，一个 package 可以由多个 .class 文件构成。那么这些 .class 如何组织？直观简单的做法是：把这些 .class 文件放到一个目录下，即使用操作系统提供的目录结构来组织 .class 文件。 Java 也使用这种方法。

把所有的 package 文件放到一个目录下解决了两个问题：1. 创建了唯一的 package 名字。通过这个目录的路径名字，我们可以找到该 package 的 .class 名字。通常， package 的第一部分是类创建者的域名逆序。由于域名是独一无二的，所以遵循这个规范写的类的名字也是独一无二的。2. 把所有类都保存到一个文件夹下，方便 Java 解释器寻找 .class 文件。

那么，是不是我们用了 package import 以及把类文件放到独一无二的文件路径中就不会有冲突了呢？不是的。假设两个库通过 * 导入，但是这两个库中有同名的类，这该怎么办？

```
import net.mindview.simple.*;
import java.util.*;
```



java.util* 包含了类 Vector ，如果 net.mindview.simple.* 也包含了名为 Vector 的类，怎么办？

```
Vector v = new Vector();
```

上面的一行代码到底调用的是哪里的 Vector 呢？解释器不知道，阅读代码的人也不知道。此时编译器会报错，强制写代码的人修改这个 bug 。如果你想使用 java.util.* 中的 Vector 你必须使用全名导入：

```
java.util.Vector v = new java.util.Vector();
```

了解了 package 和 import 的用法之后，基本上可以编写自己的库了。

# 4 Java 访问关键字

java 提供了 public protected 和 private 三个关键字用于控制对类的访问。这三个关键词控制了类中每一个成员的可访问程度，如果一个类的成员没有提供这三个关键字约束，那么这个成员的访问遵循 package access 也就是说，凡是能访问到这个类所属 package 的都可以访问到这个成员。不过我们这里不准备再着墨 package access ，因为此前已有不少描述。 public protected 和 private 更值得关注。

## 4.1 public

public 意味着其后面的类成员可以被任何人访问。假设你定义了一个 dessert 类在 Cookie.java 这个编译单元中。看代码：

```
//: access/dessert/Cookie.java
// Creates a library.
package access.dessert;
public class Cookie {
  public Cookie() {
    System.out.println("Cookie␣constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

谨记： Cookie.java 生成的类文件必须在 dessert 的目录下，而 dessert 又必须在 access 所在的目录下，而 access 必须在 CLASSPATH 中包含的路径下。不要错误的默认 Java 会在当前路径下寻找 .class 。如果 . 不在 CLASSPATH 中，那么 Java 不会查看当前目录。现在，我们创建一个使用 Cookie 的程序：

```
//: access/Dinner.java
// Uses the library.
import access.dessert.*;
public class Dinner {
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // ' Cant access
  }
} /* Output:
Cookie constructor
*///:~
```

在代码中创建了一个 Cookie 的对象 x，因为 Cookie 的构造函数是 public 的，这个类是 public 的。但是在 Dinner.java 中不能访问 bite() ,因为 bite() 只能有 dessert 访问（还记得 package access 么？）

## 4.2 默认的包

下面的代码能够编译通过：

```
//: access/Cake.java
// Accesses a class in a separate compilation unit.
class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} /* Output:
Pie.f()
*///:~
```

第二个文件是：

```
//: access/Pie.java
// The other class.
class Pie {
   void f() { System.out.println("Pie.f()"); }
} ///:~
```

你可能会以为这两个文件时完全独立的，但是 Cake 却能够创建 Pie 对象，并调用它的成员函数 f() 。他们之所以可以互相访问是因为他们在同一个文件夹中， Java 会把一个文件夹中的文件是做默认的包，因此在一个文件夹下的文件可以互相访问。

## 4.3 private

private 意味着除了这个类之外所有程序都不能访问该成员，包括在同一个 package 的其他类。貌似标记为 private 的类成员隔离了自己，但是细想，如果有很多人在创建同一个 package ，那么这个 private 就有意义了：只有你自己能访问，不用担心会影响到别人。

鉴于 pakcage access 已经提供了一定程度的隔离，你或许会认为 private 不那么重要。错！ private 非常重要，尤其在多线程场合（这个会在以后引入，现在只需要知道，该使用 private 的时候一定不要手软）。

```
//: access/IceCream.java
// Demonstrates "private" keyword.
class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}
public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

上面代码给出了一个使用 private 的例子：有时候你可能想控制一个类的创造过程。在上面的例子中，不能通过 Sundae() 的构造函数来创造类（因为这个构造函数是 private ），只能实用 makeASundae() 来创造这个类。

关于 private 的实用有一个准则：任何你觉得是辅助性质的方法都可以标注为 private 。对于类中的数据成员也是如此，除非你要向别人展示类的内部实现，否则类的说有数据域都应该是 private 。

## 4.4 protected

理解 protected 这个关键词可能需要对 **继承** 有一点了解。但是为了完整性，这里简单的提一下，继承是根据已有类创建新类的过程。可以再已类的基础上添加新的成员，或者改变已有成员函数的行为。为了继承已有的类，你可以说新类扩展 (extends) 了基类。

```
class Foo extends Bar{}
```

如果新类是基于已有类，那么你可以访问基类的 public 成员。（当然，如果在相同的 package 中，你还可以通过 package access 来访问一些成员）。



有时候基类创建者像创建一些只有基类能够访问的成员，而其他类不能访问。protected 就是做这个活的。 protected 同样提供 package access 也就是说 package 中的其他类也能访问该类中的 protected 成员。

# 5 总结

本文简要叙述了 java 是如何提供访问控制的。主要包括使用 package 和 import 来组织库，导入库。使用 public protected 和 private 来控制类成员的访问。