什么是javabean

作者：[GoOnDrift](http://my.csdn.net/GoOnDrift)

javabean其实就是一个java类，不过他是有某些规定的java类，例如它有属性，且属性名为全小写，对每个属性有get和set方法，叫做设置器与获取器，这个类还必须有一个无参数的构造函数。

javabean的提出是为了实现可重用组件的需求提出的。其实jdk里面的大部分类都是javabean。在通常情况下javabean被用作处理业务逻辑来使用。

<http://www.zhihu.com/question/19773379>

Java语言欠缺属性、事件、多重继承功能。所以，如果要在Java程序中实现一些面向对象编程的常见需求，只能手写大量胶水代码。Java Bean正是编写这套胶水代码的惯用模式或约定。这些约定包括getXxx、setXxx、isXxx、addXxxListener、XxxEvent等。遵守上述约定的类可以用于若干工具或库。  
  
举个例子，假如有人要用Java实现一个单向链表类，可能会这样写：

// 编译成 java-int-list\_1.0.jar

**public** **final** **class** **JavaIntList** **{**

**static** **class** **Node** **{**

**public** Node next**;**

**public** **int** value**;**

**}**

**public** Node head**;**

**public** **int** size**;**

**}**

上述实现为了能够快速获取链表的大小，把链表大小缓存在size变量中。用法如下：

JavaIntList myList **=** **new** JavaIntList**();**

System**.**out**.**println**(**myList**.**size**);**

JavaIntList的作者很满意，于是开源了java-int-list库的1.0版。文件名是java-int-list\_1.0.jar。发布后，吸引了许多用户来使用java-int-list\_1.0.jar。  
有一天，作者决定要节省内存，不要缓存size变量了，把代码改成这样：

// 编译成 java-int-list\_2.0.jar

**public** **final** **class** **JavaIntList** **{**

**static** **final** **class** **Node** **{**

**public** Node next**;**

**public** **int** value**;**

**}**

**public** Node head**;**

**public** **int** **getSize()** **{**

Node n **=** head**;**

**int** i **=** 0**;**

**while** **(**n **!=** **null)** **{**

n **=** n**.**next**;**

i**++;**

**}**

**return** i**;**

**}**

**}**

然后发布了2.0版：java-int-list\_2.0.jar。发布后，原有java-int-list\_1.0.jar的用户纷纷升级版本到2.0。这些用户一升级，就发现自己的程序全部坏掉了，说是找不到什么size变量。于是这些用户就把作者暴打一顿，再也不敢用java-int-list库了。  
  
这个故事告诉我们，如果不想被暴打致死，你就必须保持向后兼容性。太阳公司在设计Java语言时，也懂得这个道理。所以Java标准库中，绝对不会出现public int size这样的代码，而一定会一开始就写成：

**private** **int** size**;**

**public** **int** **getSize()** **{** **return** size**;** **}**

让用户一开始就使用getSize，以便有朝一日修改getSize实现时，不破坏向后兼容性。这种public int getSize() { return size; }的惯用手法，就是Java Bean。  
  
现在是2014年，C#、Scala等比Java新的面向对象语言自身就提供了语言特性来实现这些常用需求，所以根本不需要Java Bean这样繁琐的约定。  
  
比如，假如有个Scala版的ScalaIntList：

// 编译成 scala-int-list\_1.0.jar

**object** **ScalaIntList** **{**

**final** **case** **class** **Node(**next**:** **Node,** value**:** **Int)**

**}**

**final** **class** **ScalaIntList** **{**

**var** head**:** **ScalaIntList.Node** **=** **null**

**var** size**:** **Int** **=** 0

**}**

用户这样用：

**val** myList **=** **new** **ScalaIntList**

println**(**myList**.**size**)**

有一天你心血来潮改成这样：

// 编译成 scala-int-list\_2.0.jar

**object** **ScalaIntList** **{**

**final** **case** **class** **Node(**next**:** **Node,** value**:** **Int)**

**}**

**final** **class** **ScalaIntList** **{**

**var** head**:** **ScalaIntList.Node** **=** **null**

**final** **def** size**:** **Int** **=** **{**

**var** n **=** head

**var** i **=** 0

**while** **(**n **!=** **null)** **{**

n **=** n**.**next

i**++**

**}**

i

**}**

**}**

用户还是照样能用，根本不破坏向后兼容性。所以Scala程序只要不考虑和Java交互，一般就不需要类似Java Bean这样的约定。  
  
顺便说一句，向后兼容性分为源代码级和二进制级，Scala的var或val改为final def的话，无论源代码级的向后兼容性，还是二进制级的向后兼容性，都不遭受破坏。但C#的字段改为属性的话，虽然不破坏源代码级的向后兼容性，但是会破坏二进制级的向后兼容性。这是C#的设计缺陷，导致微软的编码规范不得不禁止使用公有字段。