编程随想：Java新手的通病

*2009*年*1*月*29*日

概述

其实很早以前就想写这样一个文章，可惜当时我没有 Blog，所以到现在才写下来。最近几年，随着 Java 在 Web 应用和企业应用两个方面的普及，对 Java 程序员的需求量大增。因此 Java 程序员的数量也突然猛增（从 TIOBE 的排行榜可以看出来）。这虽然对 Java 社区来说是好事，但也暴露出一些问题。一方面由于大量的开发人员进入 Java 这个领域，相应的教学、培训跟不上；另一方面，很多进入 Java 领域的开发人员都比较浮躁，寄希望于"速成"，没有耐心练好基本功。

根据最近几年我面试 Java 程序员的经历以及对周围使用 Java 的同事的观察，我总结了一些共通的问题以及相应的解决方法。如果你是一个 Java 新手（刚学会 Java 不久，工作1-2年），你可以看看我说的通病是不是你也有，如果有的话，得赶紧补救一下了！

对算法和数据结构不熟悉

为什么我先拿"数据结构和算法"说事捏？这玩意是写程序最最基本的东东。不管你使用 Java 还是其它的什么语言，都离不开它。而且这玩意是跨语言的，学好之后不管在哪门语言中都能用得上。

既然"数据结构和算法"这么重要，为什么很多 Java 新手却很不熟悉捏？我琢磨了一下，估计有两种可能。有些人虽然是计算机系毕业的，但是当初压根没好好学过这门课程，到工作时早都还给老师了；还有一些人是中途转行干编程，转行后又没有好好地打基础（都指望速成）。

下面我列出几个很基本的问题，如果你每一个问题都搞得很清楚，那说明你过了这关，可以去看看下一个帖子了。否则的话，你赶紧去找本算法和数据结构的书恶补一下吧。

- 什么时候该用数组型容器、什么时候该用链表型容器？
- 什么是散列函数？HashMap 的实现原理是什么？
- 什么是递归？如果你以前从来没写过递归函数，尝试着写一个（比如用递归函数进行目录树遍历）。
- 什么是算法复杂度？
- 你是否理解空间换时间的思想？
- 写一个针对整数数组的冒泡排序函数，看看你要修改几次才能跑通。
- 写一个针对整数数组的二分查找函数，看看你要修改几次才能跑通。

后面接着说第 2 个通病：缺乏面向对象的基本功。

缺乏面向对象的基本功

按理说 Java 是一个很 OO 的语言，Java 社区也一向是充满了“对象”的氛围。但我在面试 Java 程序员时，却屡屡碰到让我大跌眼镜的事情。我碰到不止一个求职者，连什么是“多态”都讲不清楚。很多人号称用过设计模式，但一半以上都仅限于单键模式和抽象工厂模式。当我深入问他/她抽象工厂模式到底有什么好处时，很多人语焉不详。

为什么很多 Java 程序员会缺乏面向对象基本功？这得怪那些 Java 框架。现在 Java 的各种框架太发达、太傻瓜化了，导致很多程序员只需要按部就班、照着框架进行代码填空，基本已经丧失了 OOA 和 OOD 的能力。我手下有些个 Java 程序员，对 Spring、Hibernate 等框架了如指掌；但如果给他一个简单需求，让他写一个脱离 Web 框架的独立 Application，他就不知所措了。这样的开发人员，将来只能成为所谓的“软件蓝领”，岗位很难得到提升。

同上一个帖子一样，我这次也提如下几个问题：

- 基于接口的继承和基于实现的继承各有什么优点、缺点？
- 继承（包括 extend 和 implement）有什么缺点？
- 多态（polymorphism）有什么缺点？
- 为什么Java可以多继承 interface，而不可以多继承 class？
- 假如让你写一个小游戏（比如人机对战的五子棋），你会如何设计类结构？
- 类结构设计时，如何考虑可扩展性？

如果上述这些问题你都能够搞得比较清楚，说明你的 OO 基础还过得去。否则的话，我建议你一边找些 OOAD 和设计模式的书看看，同时自己动手写些简单的小程序（不依赖那些框架），把学到的模式理论结合到实践中。通过这种方式来提高自己 OOAD 的能力，效果会比较好。

后面来聊一下第 3 个通病：缺少良好的编程习惯。

缺少良好的编程习惯

上次聊了“缺乏面向对象基本功”，今天来说说编程习惯的问题。今天说的这些坏习惯大部分都是跨语言的（C++、Python 新手也有），而且大部分都需要靠平时不断地努力才能慢慢改掉。

★随意地命名

有些新手写程序，当需要定义某个变量名（也可能是函数名、类名、包名等）时，随意地一敲键盘，名字就起好了……若干星期后，碰到某 bug，再来看自己写的代码时，心中暗自嘀咕：“这代码是我写的吗？咋都看不懂捏？”

所以我常跟新来的菜鸟说，命名不规范害死人啊！鉴于该问题相当普遍，我整理了几种典型的作为反面教材，具体如下：使用单字母命名变量；使用一些没太大意义的变量名（例如 s1、s2、s3）；对同一个业务概念使用不同的术语/缩写（容易让读代码的人神经分裂）；使用拼音命名（如果你团队中有港台人士或者老外，就惨了）。

★习惯于代码的 copy & paste

这是一个很普遍的问题。很多新手写代码的时候，如果发现要写的某个函数和前几天写的某个函数差不多，就把原来的那个函数贴过来，然后稍微改几下，心中还暗喜：“又快速搞定了一个功能”.....

同学，如果你也喜欢这么干，可要注意了。这种做法是代码臭味（借用《重构 - 改善既有代码的设计》的提法）的主要来源，导致代码可维护性大大下降。当你将来需要增加功能或修改 bug 的时候，要同时改动多个地方，而那时你估计已经想不起来这砣代码有几个克隆了。

★Magic Number 满天飞

如果你没有听说过“Magic Number”，先看“这里”了解一下。

为了说明 Magic Number 的问题，咱找个例子来说事儿：假设有个业务逻辑中需要进行 10 秒的超时等待，你会怎么写这个 sleep 语句？我估计大部分人不外乎下面三种写法。

1. 直接写上 `sleep(10*1000);` 了事
2. 定义一个常量 `TIMEOUT_XXX = 10*1000;` 然后 `sleep(TIMEOUT_XXX);`
3. 在配制文件中加入一个超时项，然后程序读取配制文件获得超时值，然后调用 sleep。（此处提到的配置文件是广义的，泛指各种可用于存储配置信息的机制，如 xml 文件、数据库等）

如果你的做法类似于写法 1，你多半喜欢随手硬编码。硬编码不光缺乏可读性，而且具有和“代码拷贝粘贴”类似的代码臭味（可能会存在多个 Magic Number 克隆），不利于日后维护。

至于写法 2，比写法 1 稍好（至少可读性好了）。但是，将来一旦发生需求变更，要求在运行时调整超时间隔（甚至要求让用户来配制超时间隔），则写法 2 的缺点立马暴露无遗。

★代码耦合度太大

每当说到 MVC 或者设计模式，几乎每个 Java 开发人员都能说得头头是道？但是说归说，真正写代码的时候，鲜有人写出的代码是层次清楚的。至于说到代码耦合分别由哪些情况引起？什么是正交的设计？（关于耦合与正交设计，我后面会专门讨论一下）能完全搞明白的人就更少了。

所以很多 Java 新手的代码耦合度大也就不足为奇了。我曾经抽查过试用期员工的代码，各种业务逻辑纠缠在一起，代码臭味都要熏死人。想重构都无从下手，只好让他推倒重写。

★被 GC 宠坏

由于 Java 在语言层面提供了内存的垃圾回收机制，程序员只管申请内存，不需要再关心释放的问题。因此很多新手养成了坏习惯，对于其它资源（比如数据库连接）也只申请不释放（有些人甚至天真地以为 JVM 会帮你搞定资源回收）。

还有些人虽然知道资源需要释放，但是常常忘记（比如写了打开数据库连接和相关代码，即将写关闭数据库连接时，突然有人叫你去吃中饭，回来后就把这茬给忘了）。

这个坏习惯会导致资源的泄露，而资源泄露往往比内存泄露更要命。如果你写的程序是长时间运行的（比如运行在 WebServer 上），时间长了会由于资源耗尽而导致整个进程出问题。

下一个帖子，聊一下“异常处理使用不当”。

异常处理使用不当

上一个帖子讨论了“编程习惯的问题”，今天来聊聊关于异常处理的话题。

★空 catch 语句块

犯这种错误的人比较少，一般发生在刚学会 Java 或者刚参加工作不久的人身上。

所谓“空 catch 语句块”就是在 catch 语句块中没有对异常作任何 log 处理，导致异常信息被丢弃掉。一旦程序不能正确运行，由于查不到任何 log 信息，只好从头看代码，靠肉眼找 bug。

★没有使用 finally

很多人在 catch 语句之后不使用 finally 语句。由于在 try 语句中可能会涉及资源的申请和释放。如果在资源申请之后、资源释放之前抛出异常，就会发生资源泄露（资源泄露的严重性，上一个帖子已经聊过了）。

★笼统的catch语句块

有些人为了省事，只在自己模块的最外层代码包一个 try 语句块，然后 `catch(Exception)` 。不管捕获到什么异常，都作统一 log 了事。这种做法比“空 catch 语句块”稍好，但由于不能对具体的异常进行具体处理，对一些可恢复的异常（下面会提到），丧失了恢复的机会。而且也可能导致上述提到的资源泄露的问题。

★使用函数返回值进行错误处理

有些人放着 Java 的异常机制不用，而用函数返回值来表示成功/失败（比如返回 true 表示成功、返回 false 表示失败），简直是“捧着金碗要饭”。个人感觉，从 C 转到 Java 的人比较容易有此毛病。这种做法会导致如下几个问题：

返回值一般用整数值或布尔值表示，传递的信息过于简陋；

一旦调用者忽略了错误返回码，就会导致和“空 catch 语句块”类似的问题；

对同一个函数的多处调用，都需要对返回值进行重复判断，导致代码冗余（代码冗余的坏处，上一个帖子也已经聊过了）。

★不清楚 Checked Exception 和 Runtime Exception 的区别

这个现象比较普遍，我发现很多 2 年以上 Java 工作经验的人尚未完全搞明白两者的区别。看来这个问题得详细说一下。

当初 Java 的设计者有意区分这两种异常，是别有深意的。其中“Checked Exception”用于表示可恢复的异常（也就是你必须检查的异常）；而“Runtime Exception”表示不可恢复的异常（也就是运行时异常，主要是程序 bug 和致命错误，你不需要检查）。不过这种做法引来了很多争议（包括很多 Java 大牛），鉴于本帖子主要针对新手，以后再专门来聊这个争议的话题。

为了便于理解，下面我举一个例子来说明。假设你要写一个 Download 函数，根据传入的 URL（String 参数）返回对应网页的内容文本。这时候有两种情况你需要处理：

1、如果传入的 URL 参数是 null，这表明该函数的调用者出 bug 了，而程序本身的 bug 是很难在运行时自我恢复的。这时候 Download 函数必须抛出 Runtime Exception。并且 Download 函数的调用者不应该尝试去处理这个异常，必须让它尽早暴露出来（比如让 JVM 自己终止运行）。

2、如果传入的 URL 参数非 null，但是它包含的字符串不是一个合法的 URL 格式（可能由于用户输入错误导致）。这时候 Download 函数必须抛出 Checked Exception。并且 Download 函数的调用者必须捕获该异常并进行相应的处理（比如提示用户重新输入 URL）。

上面就是几种常见的Java异常处理的误用。下一个帖子我们来聊一下“对虚拟机（JVM）了解不足”。

不了解JVM

上次的帖子讨论了 Java 异常机制的几种误用，今天咱们来说说 JVM（以及 Java 编译器）相关的话题。为啥要聊 JVM 捏？因为有很多 Java 程序员，由于对 JVM 缺乏了解，在碰到某些技术问题时无从下手；另外，由于缺乏对 JVM 的了解，可能导致写出来的代码性能巨差或者有严重的 Bug。所以俺在之前的帖子“学习技术的三部曲：WHAT、HOW、WHY”中，强调了掌握内部机制的重要性。对于一个 Java 程序员来说，你不一定要非常清楚 JVM 的细节，但是对于一些关键的运作机制，还是要掌握大致的概念。

按照本系列的惯例，俺会问几个和 JVM 相关的问题，你如果对这些问题不是很明白，那得考虑花点时间去了解一下了。另外，鉴于有网友批评“本系列”帖子：光诊断毛病，不开出药方。（说得很形象，也很中肯）俺会针对下面提出的问题，写一些帖子来解答。

★关于基本类型和引用类型

很多新手不理解 Java 的基本类型和引用类型在本质上有什么区别。请看如下的问题：

- 这两种类型在内存存储上有什么区别？
- 这两种类型在性能上有什么区别？
- 这两种类型对于GC有什么区别？

关于前两个问题，请看之前的帖子“Java 性能优化[1]：基本类型 vs 引用类型”。

★关于垃圾回收（Garbage Collection）

很多新手不理解 GC 的实现机制。请看如下的问题：

- GC 是如何判断哪些对象已经失效？
- GC 对性能会有哪些影响？
- 如何通过 JVM 的参数调优 GC 的性能？

关于 GC 的问题，可以参见之前的帖子“Java性能优化[3]：关于垃圾回收（GC）”。

★关于字符串

对于 Java 提供的 String 和 StringBuilder，想必很多人都知道：String 用于常量字符串，StringBuilder 用于可变字符串。那 Java 当初为什么要这样设计捏？为啥不用一个类来统一搞定捏？

★关于范型（Generic Programming）

从 JDK 1.5 开始，Java 引入了一个重量级的语法：范型。不过捏，很多新手仅仅知道范型的皮毛，而对于很多本质的东东，不甚了解。

- GP 是在编译时实现的还是在运行时实现的？为什么要这么实现？
- GP 的类型擦除机制是咋回事？有啥优点/缺点？
- 使用范型容器（相对于传统容器）在性能上有啥影响？为什么？

★关于多线程

另外，多线程也是大部分Java新手的短板。所以俺最后再来提几个关于多线程的问题。

- synchronized 关键字是怎么起作用滴？
- synchronized 的颗粒度（或者说作用域）如何？是针对某个类还是针对某个类对象实例？
- synchronized 对性能有没有影响？为什么？
- volatile 关键字又是派啥用滴？啥时候需要用这个关键字捏？