开篇词 | 拥抱 Java 新特性，像设计者一样工作和思考

你好，我是范学雷，欢迎加入我的课程。从今天开始，我要用 20 讲的时间，和你聊聊 JDK 8 之后 Java 最重要的一些新特性。

说到新特性啊，有些人可能会不以为然，他们会说：

- 学这些新东西有必要吗？
- 新特性好是好，还是等到用到的时候再去学习吧！
- Java 已经老了，为什么我不去学习新语言呢？
- 网上有很多关于 Java 新特性的文章，为什么还要学习你这个专栏？

我理解为什么会有这样的问题，尽管我持有不一样的观点。

作为 Oracle 的成员，Java 安全的主要推动者和贡献者之一，我从 JDK 5 开始，就一直在参与 Java 语言及其标准类库的设计和演进。我的日常工作包括关注信息安全威胁与技术进展，制定与实现 Java 安全规范，促进 Java 技术的普及与运用等等。

在每一个 JDK 的版本里，你都能看到大量我共享过和评审过的代码。在这一过程中，我也体验了很多优秀的设计和优秀的代码，见证了代码背后的各种考量和艰难取舍。

比如说，在代码安全性和性能之间，我们该如何抉择？在代码的可维护性方面我们能不能有所提高？API 的设计能不能再皮实一点？这些问题，刚开始学习编程的同学可能太不在意，但是解决好这些问题，可以使我们的工作轻松很多。

我还在极客时间上线了专栏《代码精进之路》和《实用密码学》，分享了我在 Java 和密码学领域的经验。一直以来与 Java 还有这些新特性的接触，让我对上述问题有过很认真的思考。

我认为，**学习 Java 新特性不仅很有必要，而且最好的时机就是现在。**

给你一个保守、粗暴的估计，你如果从 JDK 8 迁移到 JDK 17，并且能够恰当使用 JDK 8 以后的新特性的话，产品的代码量可以减少 20%，代码错误可以减少 20%，产品性能可以提高 20%，维护成本可以降低 20%。这些，都是实实在在的收益。

拥抱 Java 新特性，掌握主动权

为什么我会得出这样的结论呢？

从设计者的角度，我们设计一项新特性，是为了满足新的需求，为了适应更广阔的前景。而这些新特性的优越性，会随着时间的推进越来越明显。

比如说吧，JDK 1.4.2 所在的时代，用户的数量还没有这么多，服务器也不需要支持那么多的并发。所以，当时主流的客户端-服务器的设计，是使用阻塞式的套接字接口编程。现在，如果淘宝、京东还使用阻塞式的套接字接口，那是没有一点希望支持双十一的巨大流量的。

Java 的有些新技术，甚至能催生一个新行业。比如 Java 代理的技术，就至少催生了动态监控和入侵检测两大领域的颠覆性变革，并且诞生了数家明星公司和明星产品。Java 代理的技术的本意，并不是动态监控和入侵检测，但是用户创造性地使用了这项基础技术，实现了应用技术的关键突破。

现在，Java 已经迭代到了 JDK 17，不需要依靠内幕消息，我们也能知道，主流企业很快就会拥抱 JDK 11 或者 JDK 17，就像它们曾经拥抱过 JDK 7 或者 JDK 8 一样。

所以，不管你拒绝新技术、新特性的理由是什么。跟不上技术进步？认为新技术没有用？你都是时候重新审视它了。可以说，对于致力于创造新价值的我们来说，既然投身于计算机科学的领域，除了拥抱新技术，我们没有别的选择。

如果你因此退却，想要去学习一门新的语言。我也想要提醒一句，两口五尺深的井打出的水，并不一定比十尺深的井里的水甜。

那可不可以等到需要用的时候，再去学习这些新特性呢？

这么听起来好像有道理。但我想你也没法否认，很多技术，你不了解它，它是不会进入你的意识里来的，你也不会知道什么时候该使用它。因为，在你的世界里，它根本就不存在；它要解决的问题，你当然也不是很清楚。

如果你坚持不去了解不去积累，那就丧失了主动性，只能是被动的跟随者，甚至是拖后腿的反对者。**而高级工程师和初级工程师之间的差距，恰恰就是积累和见识。你打破脑袋想不出的问题，在别人那里也许看一眼就能解决。**

如果可以，为什么不现在就积累呢？尽早成为一个方案的制定者，而不是执行者，不是更好

吗？

像 Java 语言的设计者一样思考

你可能对 Java 的新特性并不了解，或者已经在网络上看到了很多讲解 Java 新特性的文章。这些资料可能会告诉你这些新特性的种类、使用方法。但在我看来，了解一个新特性背后的这些逻辑和实际运用，发掘它未来的潜力，远远比学会这个新特性更重要。

在设计新特性时，我们还要考量这项新特性是否能够持续地满足新的需求（Requirement）、增强代码安全（Security）、提高生产效率（Productivity）、提升产品性能（Performance）、降低维护成本（Maintenance）。

基于这些考量，除了对于单个的新特性的介绍，我还会和你讨论新技术组合的化学反应，比如在第 8 讲到第 9 讲，我们会讨论怎么通过封闭类、档案类以及模式匹配的叠加效果，把代码的错误处理性能提高数百倍。

无论你的基础如何，我都可以从新特性设计者的角度带你由浅及深地了解它们，而且可以让你学得更快、更精准、更深入，减少自己摸索的过程，更快地获得竞争的优势。

在使用云计算的时代，每一份性能提升，都是实实在在的成本消减；每一点工作效率的提升，都能为你争取到更多休闲时光；每一份错误的减少，都可以尽快熄灭深夜的灯光。何乐而不为呢？

我们会一起学习哪些新特性？

可是从 JDK 9 到 JDK 17，这么多的版本和新特性，到底要从哪里开始学、怎么学才能迅速、精准地抓到这些新特性的精髓，提高工作效率呢？

在这门课程里，我从 JDK 9 到 JDK 17 的新特性中筛选出了最核心、有用的 18 条特性。我会分三个模块给你讲解一般软件工程师需要经常使用的 Java 语言新技能。

在第一模块，我会给你介绍一些可以提升编码效率的特性，比如说档案类。

可以说，档案类是一个看起来不起眼的小技术。但它却具有巨大的能量。我们的代码里，存在大量的只读性质的数据。没有档案类的时候，我们要想把一个数据正确地表述好，抽象成一个类，需要上百行的代码，还要小心遵守 Java 的各种规范，比如说比较两个实例的规矩。有了档案类，完全相同的逻辑，就只需要一两行代码了。毫无疑问，档案类会把我们从千篇一律的数据表述代码里解放出来，有更多的时间专注于真正有价值的工作。

学完这一部分内容，你能够使用这些新特性，大幅度提高自己的编码效率，降低编码错误。保守估计，你的编码效率可以提高 20%。这也就意味着，如果工作量不变，每一个星期你都可以多休息一天。

在第二模块，我们会把焦点放在提升代码性能上，比如错误处理的最新成果。

使用异常来处理错误的逻辑，抛出异常和捕获异常，一直以来都是 Java 错误处理的不二选择。然而，抛出异常和捕获异常的开销是巨大的；在按计算能力付费的环境下，异常处理的意外开销是增厚账单的一个重要因素。因此，Go 语言甚至完全放弃了异常处理的方式。当 Java 语言发展到 JDK 17 的时候，我们有没有办法在 Java 语言里，使用类似于 Go 语言的错误处理方式，甚至变得更好呢？这些问题，你会在第二个模块里找到答案。

学完这一部分内容，你将能够使用这些新特性，大幅度提高软件产品的性能，帮助公司提高用户满意度，节省运营费用。保守估计，你编写代码的性能可以提高 20%，甚至更多。

在第三模块，我会跟你讲讲如何通过新特性降低维护难度，比如模块化和安全性、兼容性问题。学完这一部分内容，你将能够编写出更健壮，更容易维护的代码，并且能够知道怎么高效地把旧系统升级到 Java 的新版本。这一部分的目标，就是帮助你把代码的维护成本降低 20%或者更多。

《深入剖析 Java 新特性》课程目录

以终为始，这样学习新特性对你帮助最大

在讲解这些新特性的时候，我会以终为始，从便于你使用它们写出高质量代码的角度来展开。

首先，这门课会采用案例阅读和讨论的形式展开。

每一个新特性，我们都从阅读案例开始。这样，随着对案例的拆解和步步深入的改进，我们能够更加了解它们，理解每一个新特性诞生背后的推动力量。

这能够提高你的见识和思辨能力。让你的面试不再停留在无话可说、无题可聊的表面层次上。同时，你的代码编写能力也会有所提升。

所以，请你一定认真阅读、思考每一个案例，这是你学习这门课的基础。

然后，我还启用了多样的代码样本。

学习一门语言的新技术，最好的办法就是反复地折腾软件代码。看看什么样的写法是合法的，什么样的写法是非法的；什么样的写法更有效率，什么样的写法会是一团糟。所以，这门课的每一讲、每一个新特性，我们都会反复地讨论：以前的代码能怎么写，现在代码该怎么写，什么样的代码容易犯错，什么样的代码更养眼、更健壮，还能有什么样的改进。帮助你掌握运用新技术的方法和场景。

为了方便我们折腾，我在 GitHub 上开设了一个代码库。我希望你能认真阅读这些折腾来折腾去的样本；把这些代码下载下来，反复地修改，持续地改进。

我们都希望在最短的时间内掌握新知识，但事实是，知识与能力之间仍然有着一道巨大的鸿沟。所以，请你花更多的时间，把飘渺的知识，转化成你自己内建的能力。折腾的代码越多，你能从中学到的也就越多。这是提升你能力的关键。

最后，我还设置了代码评审这样的反馈环节。

学习最快的途径，就是请教更优秀的人；一句话的点拨，也许就胜过你几年的摸着石头过河。而加入一个社区，提交代码，接受同行和专家的评审，恐怕是目前最现实的、最有效的办法之一了。为了方便大家提交代码和评审代码，我在 GitHub 上开放了代码提交申请。

任何一行你折腾过的代码，包括每一节留下的思考题，你都可以提交一个 GitHub 的拉取请求（Pull Request），然后看看同行们的建议。当然啦，对于别人提交的代码来说，你也是他们期望寻求帮助的同行。我也会阅读一部分拉取请求，给出我的建议，我们共同进步。

所以，我建议你一定要积极地参与代码评审这样的反馈环节，给别人反馈也听取别人给你的反馈。在一定意义上，这也决定了你学习的速度和深度。

这样学习下来，不管你是刚刚开始学习 Java 语言，还是拥有扎实的 Java 语言和面向对象设计基础，都不再会被下面的问题难住：

1. 对新特性略知一二，但却不了解它为什么这么设计、可以解决什么问题、怎么用，导致在面试、在和别人交流技术时，顾左右而言他，要么和心仪的公司失之交臂，要么难以形成自己的技术影响力。
2. 了解了新技术的各种功能、各种概念，但是对它们的最佳实践、运用方法和场景知之甚少，一写代码就没了底气，写出来的代码缺东少西。
3. 看到了旧代码优化的空间，也看到了新技术的前景，但是怎么具体地改进旧代码、写好新代码，好像也想不出更好的办法来。

一句话总结，对于每一个新技术，**“面试聊得开，代码写得好，技术用得上”**，是我希望这门课能够给你带来的提升。

好了，今天的内容就先到这里。也期待你可以在留言区和我聊聊，你对这门课的期待，以及你在学习 Java 新特性时的经历。接下来，让我们正式开始学习 Java 语言，开启这段打怪升级的旅程吧！

01 | JShell：怎么快速验证简单的小问题？

你好，我是范学雷。今天，我们聊一聊 Java 的交互式编程环境，JShell。

JShell 这个特性，是在 JDK 9 正式发布的。从名字我们就能想到，JShell 是 Java 的脚本语言。一门编程语言，为什么还需要支持脚本语言呢？编程语言的脚本语言，会是什么样子的？它又能够给我们带来什么帮助呢？

让我们一起来一层一层地拆解这些问题，弄清楚 Java 语言的脚本工具是怎么帮助我们提高生产效率的。我们先从阅读案例开始。

阅读案例

学习编程语言的时候，我们可能都是从打印“Hello, world!”这个简单的例子开始的。一般来说，Java 语言的教科书也是这样的。今天，我们也从这个例子开始，温习一下 Java 语言第一课里面涉及的知识。

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

好了，有了这段可以拷贝的代码，接下来我们该怎么办呢？

首先，我们需要一个文本编辑器，比如 vi 或者类似于 IDEA 这样的集成编辑环境，把这段代码记录下来。文本编辑器，每个人都有不同的偏好，每个系统都有不同的偏好。一个软件工程师，可能需要很长时间，才能找到自己顺手的编辑器。就我自己而言，我使用了二十多年 vi 编辑器，直到这两年才发现 IDEA 的好。但是使用 IDEA 的时候，我还是会不自主地敲击 vi 的命令。不得不说，顺手，确实是一个很顽固、难改的行为习惯。

回到刚才的正题。有了文本编辑器，接下来，我们要把这段源代码编译成 Java 的字节码。编译器会帮助我们评估这段代码，看看有没有错误，有没有需要警示的地方。通常，我们使用 javac 命令行，或者通过集成编辑环境自动编译。

```
$ javac HelloWorld.java
```

编译完成之后，我们要运行编译好的字节码，把程序的结果显示出来。在这里，我们一般使用 java 命令行，或者通过集成编辑环境来运行。

```
$ java HelloWorld
```

最后一步，我们要观察运行的结果，检查一下是不是我们期望的结果。

```
Hello, world!
```

如果让我去教授 Java 语言，教到这里，我会让同学们小小地庆祝一下：我们完成了 Java 语言的第一个程序。

万事开头难，完成 Java 语言的第一个小程序，尤其难！你要学习使用编辑器、使用编译器、使用运行环境。对于一个编程语言的初学者而言，这是迈入 Java 语言世界的第一步，也是很大的一步。这当然是巨大的收获，一个小小的庆祝当然是应得的也是值得的！

当然，会有同学试着改动这段代码，享受创造的乐趣。比如说，把“Hello, world!”改成“世界你好”或者“How are you?”。 这样一来，我们就还要经历编辑、编译、运行、观察这样的过程。

```
class HowAreYou {
    public static void main(String[] args) {
        System.out.println("How are you?");
    }
}
```

毫不意外，对 Java 的了解更深之后，还会有同学继续修改代码，把 System.out 换成 System.err。然后，同样的过程还要再来一遍：编辑、编译、运行、观察。

其实，编辑、编译、运行、观察这四个步骤，就是我们学习一门新语言或者一项新特性的常规过程。如果你已经有多年的 Java 语言使用经验，想一想吧，你是怎么学习 JDK 7 的 try-with-resource 语句，又是怎么学习 JDK 8 的 Lambda 表达式的？是不是也是类似的过程？

也许，你已经习惯了这样的过程，并没有感觉得到有什么不妥当的地方。不过，如果我们看看 bash 脚本语言的处理，也许你会发现问题所在。

```
bash $ echo Hello, World!
Hello, world!
bash $
```

显然，使用 bash 编写的“Hello, world!”要简单得多。你只需要在命令行输入代码，bash 就会自动检查语法，立即打印出结果；它不需要我们调用额外的编辑器、编译器以及解释器。当然，这并不是说 bash 不需要编译和运行过程。bash 只是把这些过程处理得自动化了，不再需要我们手动处理了。

拖后腿的学习效率

没有对比，就没有伤害。一般来说，不管是初学者还是熟练的程序员，使用 bash 都可以快速编写出“Hello, world!”，不到一分钟，我们就可以观察到结果了。但是如果使用 Java，一个初学者，也许需要半个小时甚至半天才能看到输出结果；而一个熟练的程序员，也需要几分钟甚至十几分钟才能完成整个过程。

这样的学习效率差异并不是无关紧要的。有来自学校的反馈表明，老师和学生放弃 Java 的最重要的原因，就是学习 Java 的门槛太高了，尤其是入门第一课。上面的这个小小的“Hello, world!”程序，需要极大的耐心，才能看到最后的结果。这当然影响了新的小伙伴们学习 Java 的热情。而且，老朋友们学习 Java 新技术的热情，以及深入学习现有技术的热情，也会受到了极大的阻碍。

JDK 17 发布的时候，我们经常可以看到这样的评论，“然而，我还是在使用 JDK 8”。确实，没有任何人，也没有任何理由责怪这样的用户。除非有着严格的自律和强烈的好奇心，没有人喜欢学习新东西，尤其是学习门槛比较高的时候。

如果需要半个小时，我们才能看一眼一个新特性的样子，重点是，这个新特性还不一定能对我们有帮助，那很可能我们就懒得去看了。或者，我们也就是看一眼介绍新特性的文档，很难有动手试一试的冲动。最后，我们对它的了解也就仅仅停留在“听过”或者“看过”的程度上，而不是进展到“练过”或者“用过”的程度。

那你试想一下，如果仅仅需要一分钟，我们就能看到一个新特性的样子呢？我想，在稍纵即逝的好奇心消逝之前，我们很有可能会尝试着动动手，看一看探索的成果。

实际上，学习新东西，及时的反馈能够给我们极大的激励，推动着我们深入地探索下去。那 Java 有没有办法，变得像 bash 那样，一分钟内就可以展示学习、探索的成果呢？

及时反馈的 JShell

办法是有的。JShell，也就是 Java 的交互式编程环境，是 Java 语言给出的其中一个答案。JShell API 和工具提供了一种在 JShell 状态下交互式评估 Java 编程语言的声明、语句和表达式的方法。JShell 的状态包括不断发展的代码和执行状态。为了便于快速调查和编码，语句和表达式不需要出现在方法中，变量和方法也不需要出现在类中。

我们还是通过例子来理解上面的表述。

启动 JShell

JShell 的工具，是以 Java 命令的形式出现的。要想启动 JShell 的交互式编程环境，在控制台 shell 的命令行中输入 Java 的脚本语言命令“jshell”就可以了。

下面的这个例子，显示的就是启动 JShell 这个命令，以及 JShell 的反馈结果。

```
$ jshell
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
jshell>
```

我们可以看到，JShell 启动后，Java 的脚本语言就接管了原来的控制台。这时候，我们就可以使用 JShell 的各种功能了。

另外，JShell 的交互式编程环境，还有一个详细模式，能够提供更多的反馈结果。启用这个详尽模式的办法，就是使用“-v”这个命令行参数。我们使用 JShell 工具的主要目的之一，

就是观察评估我们编写的代码片段。因此，我一般倾向于启用详细模式。这样，我就能够观察到更多的细节，有助于我更深入地了解我写的代码片段。

```
$ jshell -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
```

退出 JShell

JShell 启动后，就接管了原来的控制台。要想重新返回原来的控制台，我们就要退出 JShell。退出 JShell，需要使用 JShell 的命令行。

下面的这个例子，显示的就是怎么使用 JShell 的命令行，也就是“exit”，退出 java 的交互式编程环境。需要注意的是，JShell 的命令行是以斜杠开头的。

```
jshell> /exit
|  Goodbye
```

JShell 的命令

除了退出命令，我们还可以使用帮助命令，来查看 JShell 支持的命令。比如，在 JDK 17 里，帮助命令的显示结果，其中的几行大致是下面这样：

```
jshell> /help
|  Type a Java language expression, statement, or declaration.
|  Or type one of the following commands:
|  /list [<name or id>|-all|-start]
|    list the source you have typed
... snipped ...
|  /help [<command>|<subject>]
|    get information about using the jshell tool
... snipped ...
```

熟悉 JShell 支持的命令，能给我们带来很大的便利。限于篇幅，我们这里不讨论 JShell 支持的命令。但是，我希望你可以通过帮助命令，或者其他的文档，了解这些命令。它们可以帮助你更有效率地使用这个工具。

我相信你肯定会对帮助命令显示的第一句话非常感兴趣：输入 Java 语言的表达式、语句或者声明。下面我们就来重点了解一下这一部分。

立即执行的语句

首先，我们来看一看使用 JShell 来评估 Java 语言的语句。比如，我们可以使用 JShell 来完成打印“Hello, world!”这个例子。

```
jshell> System.out.println("Hello, world!");
Hello, world!
jshell>
```

可以看到，一旦输入完成，JShell 立即就能返回执行的结果，而不再需要编辑器、编译器、解释器。

更方便的是，我们可以使用键盘的上移箭头，编辑上一次或者更前面的内容。如果我们想评估 System.out 其他的方法，比如不追加行的打印，我们编辑上一次的输入命令，把上面例子中的“println”换成“print”。就像下面这样就可以了。

```
jshell> System.out.print("Hello, world!");
Hello, world!
jshell>
```

如果我们使用了错误的方法，或者不合法的语法，JShell 也能立即给出提示。

```
jshell> System.out.println("Hello, world\!");
|  Error:
|  illegal escape character
|  System.out.println("Hello, world\!");
|                             ^
```

JShell 的这种立即执行、及时反馈的特点，毫无疑问地，加快了我们学习和评估简单 Java

代码的速度，激励着我们去学习更多的东西，更深入的技能。

可覆盖的声明

另外，JShell 还有一个特别好用的功能。那就是，它支持变量的重复声明。JShell 是一个有状态的工具，这样我们就能够很方便地处理多个有关联的语句了。比如说，我们可以先试用一个变量来指代问候语，然后再使用标准输出打印出问候语。

```
jshell> String greeting;
greeting ==> null
|  created variable greeting : String
jshell> String language = "English";
language ==> "English"
|  created variable language : String
jshell> greeting = switch (language) {
   ...>     case "English" -> "Hello";
   ...>     case "Spanish" -> "Hola";
   ...>     case "Chinese" -> "Nihao";
   ...>     default -> throw new RuntimeException("Unsupported language");
   ...> };
greeting ==> "Hello"
|  assigned to greeting : String
jshell> System.out.println(greeting);
Hello
jshell>
```

为了更方便地评估，你可以使用 JShell 运行变量的重复声明和类型变更。比如说，我们可以再次声明只带问候语的变量。

```
jshell> String greeting = "Hola";
greeting ==> "Hola"
|  modified variable greeting : String
|    update overwrote variable greeting : String
```

或者，把这个变量声明成一个其他的类型，以便后续的代码使用。

```
jshell> Integer greeting;
greeting ==> null
|  replaced variable greeting : Integer
|    update overwrote variable greeting : String
```

变量的声明可以重复，也可以转换类型，就像上一个声明并不存在一样。这样的特点和 Java 的可编译代码有所不同，在可编译的代码里，在一个变量的作用域内，这个变量的类型是不允许转变的，也不允许重复声明。

JShell 支持可覆盖的变量，主要是为了简化代码评估，解放我们的大脑。要不然，我们还得记住以前输入的、声明的变量，这可不是一个简单的任务。

也正是因为 JShell 支持可覆盖的变量，我们才能说 JShell 支持不断发展的代码，JShell 才能够更有效地处理多个关联的语句。

独白的表达式

前面我们说过，JShell 工具可以接受的输入包括 Java 语言的表达式、语句或者声明。刚才讨论了语句和声明的例子，现在我们来看看输入表达式是什么样子的。

我们知道，在 Java 程序里，语句是最小的可执行单位，表达式并不能单独存在。但是，JShell 却支持表达式的输入。比如说，输入“1+1”，JShell 会直接给出正确的结果。

```
jshell> 1 + 1
$1 ==> 2
|  created scratch variable $1 : int
```

有了独立的表达式，我们就可以直接评估表达式，而不再需要把它附着在一个语句上了。毫无疑问，这简化了表达式的评估工作，使得我们可以更快地评估表达式。下面的例子，就可以用来探索字符串常量和字符串实例的联系和区别，而不需要复杂的解释性代码。

```
jshell> "Hello, world" == "Hello, world"
$2 ==> true
|  created scratch variable $2 : boolean
jshell> "Hello, world" == new String("Hello, world")
$3 ==> false
|  created scratch variable $3 : boolean
```

总结

好，到这里，今天的课程就要结束了，我来做个小结。从前面的讨论中，我们了解了 JShell 的基本概念、它的表达形式以及编译的过程。

JShell 提供了一种在 JShell 状态下交互式评估 Java 编程语言的声明、语句和表达式的方法。JShell 的状态包括不断发展的代码和执行状态。为了便于快速调查和编码，语句和表达式不需要出现在方法中，变量和方法也不需要出现在类中。

JShell 的设计并不是为了取代 IDE。JShell 在处理简单的小逻辑，验证简单的小问题时，比 IDE 更有效率。如果我们能够在有限的几行代码中，把要验证的问题表达清楚，JShell 就能够快速地给出计算的结果。这一点，能够极大地提高我们的工作效率和学习热情。

但是，对于复杂逻辑的验证，使用 JShell 也许不是一个最优选择。这时候，也许使用 IDE 或者可编译的代码更合适。

我还拎出了几个技术要点，这些都可能在你的面试中出现。通过这一次学习，你应该能够：

- 了解 JShell 的基本概念，知道 JShell 有交互式工具，也有 API；
    - 面试问题：你使用过 JShell 吗？
- 知道 JShell 能够接收 Java 编程语言的声明、语句和表达式，以及命令行；
    - 面试问题：JShell 的代码和普通的可编译代码，有什么不一样？

这一次的讨论，主要是想让你认识到 JShell 能给我们带来的便利，知道简单的使用方法。这样，当后面我们想要讨论更多的话题时，你就可以使用 JShell 快速验证你的小问题、小想法。要想掌握 JShell 更复杂的用法，请参考相关的文档或者材料。

思考题

在前面的讨论里，我们使用了一个例子，来说明 Java 处理字符串常量的方式。

```
jshell> "Hello, world" == "Hello, world"
$2 ==> true
|  created scratch variable $2 : boolean
```

对于精通 Java 语言的同学，这个例子也许是直观的。但对部分同学来说，这个例子也许过于隐晦。过于隐晦的代码不是好的代码。同样地，过于隐晦的 JShell 片段也不是好的片段。你有没有办法，让这个例子更容易理解？使用多个 JShell 片段，是不是更好理解？这就是我们今天的思考题。

欢迎你在留言区留言、讨论，分享你的阅读体验以及你对这个思考题的处理办法。

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请放在实例匹配专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 jshell/review/xuelei 的目录下面。

02 | 文字块：怎么编写所见即所得的字符串？

你好，我是范学雷。今天，我们聊一聊 Java 的文字块（textblocks）。

文字块这个特性，首先在 JDK 13 中以预览版的形式发布。在 JDK 14 中，改进的文字块再次以预览版的形式发布。最后，文字块在 JDK 15 正式发布。

文字块的概念很简单，它是一个由多行文字构成的字符串。既然是字符串，为什么还需要文字块这个新概念呢？文字块和字符串又有什么区别呢？我们还是通过案例和代码，来弄清楚这些问题吧。

阅读案例

我们在编写代码的时候，总是或多或少地要和字符串打交道。有些字符串很简单，比如我们都知道的"Hello，World！"字符串。有些字符串很复杂，里面可能有换行、对齐、转义字符、占位符、连接符等。

比如下面的例子中，我们要构造一个简单的表示"Hello，World！"的 HTML 字符串，就需要处理好文本对齐、换行字符、连接符以及双引号的转义字符。这就使得这段代码既不美观、也不简约，一点都不自然。

```
String stringBlock =
        "<!DOCTYPE html>\n" +
        "<html>\n" +
        "    <body>\n" +
        "        <h1>\"Hello World!\"</h1>\n" +
        "    </body>\n" +
        "</html>\n";
```

这样的字符串不好写，不好看，也不好读。更糟糕的是，我们有时候需要从别的地方拷贝一段 HTML 或者 SQL 语句，然后再转换成类似于上面的字符串。是不是出力多，收效少，需要特别的耐心？遗憾的是，这样的工作还特别多，HTML,SQL,XML,JSON,HTTP,随便就可以列一大堆。

不论对于写代码的人，还是阅读代码的人来说，处理这样的字符串都不是一件赏心悦目的事情。软件的质量是一个反馈系统，糟糕的事情总是可以让事情变得更糟糕。摊开来说，这样的字符串编写起来不省心，不仅消耗了更多时间，代码质量也没有保障。与此同时，复杂的语句也容易分散评审者的精力，让疏漏和错误不易被发现。

费时费力、质量还难以控制，这让复杂字符串的处理变成了一个很没有效率的事情。没有效率，也就意味着投入产出比低，所以我们就更不愿意投入精力和时间来做好这件事情。对于用户来说，糟糕的结果也会耗费他们更多的精力和时间。用户有多少，这个糟糕的成本就放大多少倍。

如果你经常需要阅读调试日志，你可能会有更深刻的体会。难以阅读的调试日志，可能会让你产生短暂的抗拒心理，甚至暂时地放弃调试，直到你的耐心又回来了。遗憾的是，提高调试日志的可读性，似乎永远排不上开发者的日程表。

这不是一个让人愉快的事情。不过，我们似乎也不曾有过更好的办法。

所见即所得的文字块

文字块是人们在试图扭转这种糟糕局面的过程中一个最重要的尝试。文字块是一个由多行文字构成的字符串。既然是字符串，文字块能有什么影响呢？其实，文字块是使用一个新的形式，而不是传统的形式，来表达字符串的。通过这个新的形式，文字块尝试消除换行、连接符、转义字符的影响，使得文字对齐和必要的占位符更加清晰，从而简化多行文字字符串的表达。

下面的这段代码，就是我使用文字块对阅读案例所做的改进。

```
String textBlock = """
        <!DOCTYPE html>
        <html>
            <body>
```

```
                <h1>"Hello World!"</h1>
            </body>
        </html>
        """;
System.out.println(
        "Here is the text block:\n" + textBlock);
```

对比一下阅读案例里的代码，我们可以看到，下面的这些特殊的字符从这个表达式里消失了：

1. 换行字符（\n）没有出现在文字块里；
2. 连接字符（+）没有出现在文字块里；
3. 双引号没有使用转义字符（\）。

另外，出现在文字块开始和结束位置的，是三个双引号序列；而不是我们在字符串声明里看到的单个双引号。**文字块由零个或多个内容字符组成，从开始分隔符开始，到结束分隔符结束。开始分隔符是由三个双引号字符（“”“），后面跟着的零个或多个空格，以及行结束符组成的序列。结束分隔符是一个由三个双引号字符（”“”）组成的序列。**

需要注意的是，开始分隔符必须单独成行；三个双引号字符后面的空格和换行符都属于开始分隔符。所以，一个文字块至少有两行代码。即使是一个空字符，结束分隔符也不能和开始分隔符放在同一行代码里。

```
jshell> String s = """""";
|  Error:
|  illegal text block open delimiter sequence, missing line terminator
|  String s = """""";
jshell> String s = """
   ...> """;
s ==> ""
```

同样需要注意的是，**结束分隔符只有一个由三个双引号字符组成的序列。结束分隔符之前的字符，包括换行符，都属于文字块的有效内容。**

```
jshell> String s = """
   ...> OneLine""";
s ==> "OneLine"
jshell> String s = """
   ...> TwoLines
   ...> """;
s ==> "TwoLines\n"
```

由于文字块不再需要特殊字符、开始分隔符和结束分隔符这些格式安排，我们几乎就可以直接拷贝、粘贴看到的文字，而不再需要特殊的处理了。同样地，你在代码里看到的文字块是什么样子，它实际要表达的文字就是什么样子的。这也就是说，“所见即所得”。

很多系统里常见的“所见即所得”的境界，终于也能够在 Java 语言里呈现出来了。

文字块的编译过程

那么，我们用文字块改进过的阅读案例，打印结果是什么样子的呢？从下面的打印结果，我们可以看到，为了代码整洁而使用的缩进空格并没有出现在打印的结果里。

```
Here is the text block:
<!DOCTYPE html>
<html>
    <body>
        <h1>"Hello World!"</h1>
    </body>
</html>
```

也就是说，文字块的内容并没有计入缩进空格。文字块是怎么处理缩进空格的呢？这是我们学习文字块必须要了解的一个问题。

像传统的字符串一样，文字块是字符串的一种常量表达式。**不同于传统字符串的是，在编译**

期，文字块要顺序通过如下三个不同的编译步骤：

1. **为了降低不同平台间换行符的表达差异，编译器把文字内容里的换行符统一转换成 LF（\u000A）；**
2. **为了能够处理 Java 源代码里的缩进空格，要删除所有文字内容行和结束分隔符共享的前导空格，以及所有文字内容行的尾部空格；**
3. **最后处理转义字符，这样开发人员编写的转义序列就不会在第一步和第二步被修改或删除。**

首先，我们从整体上来理解一下文字块的编译期处理这种方式。阅读一下下面的代码，你能不能预测一下下面这两个问题的结果？使用传统方式声明的字符串和使用文字块声明的字符串的内容是一样的吗？这两个字符串变量指向的是同一个对象，还是不同的对象？

```
package co.ivi.jus.text.modern;
public class TextBlocks {
    public static void main(String[] args) {
        String stringBlock =
                "<!DOCTYPE html>\n" +
                "<html>\n" +
                "    <body>\n" +
                "        <h1>\"Hello World!\"</h1>\n" +
                "    </body>\n" +
                "</html>\n";
        String textBlock = """
                <!DOCTYPE html>
                <html>
                    <body>
                        <h1>"Hello World!"</h1>
                    </body>
                </html>
                """;
        System.out.println(
                "Does the text block equal to the regular string? " +
                stringBlock.equals(textBlock));
        System.out.println(
                "Does the text block refer to the regular string? " +
                (stringBlock == textBlock));
    }
}
```

第一个问题的答案应该没有意外，第二个问题的答案可能就会有意外出现了。使用传统方式声明的字符串和使用文字块声明的字符串，它们的内容是一样的，而且指向的是同一个对象。该怎么理解这样的结果呢？其实，这就说明了，文字块是在编译期处理的，并且在编译期被转换成了常量字符串，然后就被当作常规的字符串了。所以，如果文字块代表的内容，和传统字符串代表的内容一样，那么这两个常量字符串变量就指向同一内存地址，代表同一个对象。

虽然表达形式不同，但是文字块就是字符串。既然是字符串，就能够使用字符串支持的各种 API 和操作方法。比如，传统的字符串表现形式和文字块的表现形式可以混合使用：

```
System.out.println("Here is the text block:\n" +
        """
        <!DOCTYPE html>
        <html>
            <body>
                <h1>"Hello World!"</h1>
            </body>
```

```
        </html>
        """);
```

再比如，文字块可以调用字符串 String 的 API：

```
int stringSize = """
        <!DOCTYPE html>
        <html>
            <body>
                <h1>"Hello World!"</h1>
            </body>
        </html>
        """.length();
```

或者，使用嵌入式的表达式：

```
String greetingHtml = """
        <!DOCTYPE html>
        <html>
            <body>
                <h1>%s</h1>
            </body>
        </html>
        """.formatted("Hello World!");
```

巧妙的结束分隔符

好的，我们现在看看文字块编译的细分步骤。第一个和第二个步骤都很好理解。不过，第二个步骤里“删除共享的前导空格”，是一个我们可以巧妙使用的规则。通过合理地安排共享的前导空格，我们可以实现文字的编排和缩进。

为了方便理解，在下面的例子里，我们使用小数点号‘.’表示编译期要删除的前导空格，使用叹号‘!’表示编译期要删除的尾部空格。

第一个例子，我们把结束分隔符单独放在一行，和文本内容左边对齐。这时候，共享的前导空格就是文本内容本身共享的前导空格;结束分隔符仅仅是用来结束文字块的。这个例子里，我还加入了文字内容行的尾部空格，它们在编译期会被删除掉。

```
// There are 8 leading white spaces in common
String textBlock = """
........<!DOCTYPE html>
........<html>
........    <body>
........        <h1>"Hello World!"</h1>!!!!
........    </body>
........</html>
........""";
```

第二个例子，我们也把结束分隔符单独放在一行，但是放在比文本内容更靠左的位置。这时候，结束分隔符除了用来结束文字块之外，还参与界定共享的前导空格。

```
// There are 4 leading white spaces in common
String textBlock = """
....    <!DOCTYPE html>
....    <html>
....        <body>
....            <h1>"Hello World!"</h1>!!!!
....        </body>
....    </html>
....""";
```

第三个例子，我们也把结束分隔符单独放在了一行，但是放在文本内容左对齐位置的右侧。

这时候，结束分隔符的左侧，除了共享的前导空格之外，还有多余的空格。这些多余的空格，就成了文字内容行的尾部空格，它们在编译期会被删除掉。

```
// There are 8 leading white spaces in common
String textBlock = """
........<!DOCTYPE html>
........<html>
........    <body>
........        <h1>"Hello World!"</h1>!!!!
........    </body>
........</html>
........!!!!""";
```

尾部空格还能回来吗？

你可能会问，一般情况下，尾部空格确实没有什么实质性的作用。但是万一需要尾部空格，它们还能回来吗？

其实是可以的。为了能够支持尾部附带的空格，文字块还引入了另外一个新的转义字符，‘\s’，空格转义符。空格转义符表示一个空格。我们前面说过的文字块的编译器处理顺序，空格转义符不会在文字块的编译期被删除，因此空格转义符之前的空格也能被保留。所以，每一行使用一个空格转义符也就足够了。

下面的代码，就是一个重新带回尾部空格的例子，这个字符串的前两行就包含有尾部空格。

```
// There are 8 leading white spaces in common
String textBlock = """
........<!DOCTYPE html>    \s!!!!
........<html>             \s
........    <body>!!!!!!!!!!
........        <h1>"Hello World!"</h1>
........    </body>
........</html>
........""";
```

该怎么表达长段落？

但是所见即所得的文字块也有一个小烦恼。我们知道，编码规范一般都限定每一行的字节数，通常是 80 个或者 120 个字节。可是一个文本的长段落通常要超出这个限制。文字块里的换行符通常需要保留，编码规范通常要遵守，那该如何表达长段落或者长行呢？

针对这种情况，文字块引入了一个新的转义字符，‘<行终止符>’，换行转义符。换行转义符的意思是，如果转移符号出现在一个行的结束位置，这一行的换行符就会被取缔。下面的例子就使用了换行转义符，它就把分散在两行的"Hello World!"连接在一行里了。

```
String textBlock = """
        <!DOCTYPE html>
        <html>
            <body>
                <h1>"Hello \
        World!"</h1>
            </body>
        </html>
        """;
```

需要注意的是，上面的例子里，换行转义符之前，还有一个空格。这个空格会被删除吗？连接后的字符，是没有空格间隔的“HelloWorld!”，还是中间有空格的“Hello World!”？还记得我们前面说过的编译器处理顺序吗？空格处理先于转义字符处理。因此，换行转义符之前的空格不算是文字块的尾部空格，因此会得到保留。

总结

好，到这里，今天的课程就要结束了，我来做个小结。从前面的讨论中，我们了解了文字块的基本概念，它的表达形式以及编译的过程。

文字块是 Java 语言中一种新的文字。字符串能够出现的任何地方，也都可以用文字块表示。但是，文字块提供了更好的表现力和更少的复杂性。文字块“所见即所得”的表现形式，使得使用复杂字符串的代码更加清晰，便于编辑，也便于阅读。这是一个能够降低代码错误，提高生产效率的改进。

如果要丰富你的代码评审清单，学习完这一节内容后，你可以加入下面这一条：

复杂的字符串，使用文字块表述是不是更清晰？

另外，通过今天的讨论，我拎出了几个技术要点，这些都可能在你的面试中出现。通过这一次学习，你应该能够：

- 知道文字块的基本概念，以及文字块和字符串的关系；
    - 面试问题：你知道 Java 的文字块吗？它和字符串有什么区别？
- 了解文字块要解决的问题，并且能够准确使用文字块；
    - 面试问题：应当什么时候使用文字块？
- 了解文字块的表达形式，编译过程以及文字块特有的转义字符。
    - 面试问题：怎么用文字块实现文本缩进？

如果能够有意识地使用文字块，你应该能够大幅度提高复杂字符串的可读性。从而更快地编写代码，也让潜在的错误更少。毫无疑问，在面试的时候，有意识地在代码里使用文字块，除了节省时间之外，还能够让你的代码更容易阅读和接受，给面试官带来新鲜的感受。

思考题

在前面的讨论里，我们说过文字块是一个“所见即所得”的字符串表现形式。我们可以直接拷贝、粘贴文字段落到代码里，而不需要大量的调整。可是，在有些场景里，要想完全地实现“所见即所得”，仅仅使用文字块，可能还是要费一点周折的。

比如说吧，我们看到的诗，有的时候是页面居中对齐的。比如下面的这首小诗，采用的格式就是居中对齐。

```
        No man is an island,

            Entire of itself,

Every man is a piece of the continent,

            A part of the main.
```

居中对齐这种形式，在 HTML 或者文档的世界里，很容易处理，设置一下格式就可以了。如果是用 Java 语言，该怎么处理好这首小诗的居中对齐问题？这就是今天我们的思考题。

稍微提示一个，你可以使用添加缩进空格的方式对齐，也可以不局限于简单的、单纯的 Java 语言，比如添加进来 HTML 的文本。

欢迎你在留言区留言、讨论，分享你的阅读体验以及你对这个思考题的想法。

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请放在实例匹配专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 text/review/xuelei 的目录下面。

03 | 档案类：怎么精简地表达不可变数据？

你好，我是范学雷。今天，我们聊一聊 Java 的档案类。

档案类这个特性，首先在 JDK 14 中以预览版的形式发布。在 JDK 15 中，改进的档案类再次以预览版的形式发布。最后，档案类在 JDK 16 正式发布。

那么，什么是档案类呢？档案类的英文，使用的词汇是“record”。官方的说法，Java 档案类是用来表示不可变数据的透明载体。这样的表述，有两个关键词，一个是不可变的数据，另一个是透明的载体。

该怎么理解“不可变的数据”和“透明的载体”呢？我们还是通过案例和代码，一步一步地来拆解、理解这些概念。

阅读案例

在面向对象的编程语言中，研究表示形状的类是一个常用的教学案例。今天的评审案例，我们从形状的子类圆形开始，来看一看面向对象编程实践中，这个类的设计和演化。

下面的这段代码，就是一个简单的、典型的圆形类的定义。这个抽象类的名字是 Circle。它有一个私有的变量 radius，用来表示圆的半径。有一个构造方法，用来生成圆形的实例。有一个设置半径的方法 setRadius，一个读取半径的方法 getRadius。还有一个重载的方法 getArea，用来计算圆形的面积。

```
package co.ivi.jus.record.former;
public final class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }
}
```

这个圆形类之所以典型，是因为它交代了面向对象设计的关键思想，包括面向对象编程的三大支柱性原则：封装、继承和多态。

封装的原则是隐藏具体实现细节，实现的修改不会影响接口的使用。Circle 类中，表示半径的变量被定义成私有的变量。我们可以改变半径这个变量的名字，或者不使用半径而是使用直径来表示圆形。这样的实现细节的变化，并不会影响公开方法的调用。

由于需要隐藏内部实现细节，所以需要设计公开接口来访问类的相关特征，比如例子中的圆形的半径。所以上面的例子中，设置半径的方法 setRadius 和读取半径的方法 getRadius，就显得显而易见，并且顺理成章。在面向对象编程的教科书里，以及 Java 的标准类库里，我们可以看到很多类似的设计。

可是，这样的设计有哪些严重的缺陷呢？花点时间想想你能找到的问题，然后我们接下来再继续分析。

案例分析

上面这个例子，最重要的问题，就是它的接口不是多线程安全的。如果在一个多线程的环境中，有些线程调用了 setRadius 方法，有些线程调用 getRadius 方法，这些调用的最终结果是难以预料的。这也就是我们常说的多线程安全问题。

在现代计算机架构下，大多数的应用需要多线程的环境。所以，我们通常需要考虑多线程安全的问题。该怎么解决上面例子中的多线程安全问题呢？如果上述例子的实现源代码不能更改，那么就需要在调用这些接口的程序中，增加线程同步的措施。

```
synchronized (circleObject) {
    double radius = circleObject.getRadius();
    // do something with the radius.
}
```

遗憾的是，在调用层面解决线程同步问题的办法，并不总是显而易见的。不论多么资深的程序员，都有可能疏漏、忘记或者没有正确地解决好线程同步的问题。

所以，通常地，为了更皮实的接口设计，在接口规范设计的时候，就应该考虑解决掉线程同步的问题。比如说，我们可以把上面案例中的代码改成线程安全的代码。对于 Circle 类，只需要把它的公开方法都设置成同步方法，那么这个类就是多线程安全的了。具体的实现，请参考下面的代码。

```
package co.ivi.jus.record.former;
public final class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public synchronized double getArea() {
        return Math.PI * radius * radius;
    }

    public synchronized double getRadius() {
        return radius;
    }

    public synchronized void setRadius(double radius) {
        this.radius = radius;
    }
}
```

可是，线程同步并不是免费的午餐。代价有多大呢？我做了一个简单的性能基准测试，哪怕最简单的同步，比如上面代码里同步的 getRadius 方法，它的吞吐量损失也有十数倍。这相当于说，如果没有同步的应用需要一台机器支持的话，加了同步的应用就需要十多台机器来支撑相同的业务量。

这样的代价就有点大了，我们需要寻找更好的办法来解决多线程安全的问题。最有效的办法，就是在接口设计的时候，争取做到即使不使用线程同步，也能做到多线程安全。这说起来还是有点难以理解的，我们还是来看看代码吧。

下面的代码，是一个修改过的 Circle 类实现。在这个实现里，圆形的对象一旦实例化，就不能再修改它的半径了。相应地，我们删除了设置半径的方法。也就是说，这个对象是一个只读的对象，不支持修改。通常地，我们称这样的对象为不可变对象。

```
package co.ivi.jus.record.immute;
public final class Circle implements Shape {
    public final double radius;
```

```
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

对于只读的圆形类的设计，我们可以看到两个好处。

第一个好处，就是天生的多线程安全。因为这个类的对象，一旦实例化就不能再修改，所以即便在多线程环境下使用，也不需要同步。而不可变对象所承载的数据，比如上面例子中圆形的半径，就是我们前面所说的不可变的数据。这个不可变，是有一个界定范围的。这个界定范围，就是它所在对象的生命周期。如果跳出了对象的生命周期，我们可以重新生成新对象，从而实现数据的变化。

第二个好处，就是简化的代码。只读对象的设计，使得我们可以重新考虑代码的设计，这是代码简化的来源。你可能已经注意到了，在这个实现里，我们还删除了读取半径的方法。取而代之的，是公开的半径这个变量。这就是一个最直接的简化。

应用程序可以直接读取这个变量，而不是通过一个类似于 getRadius 的方法。由于半径这个变量被声明为 final 变量，所以它只可以被读取，不能被修改。这并没有破坏对象的只读性。不过，乍看之下，这样的设计似乎破坏了面向对象编程的封装原则。公开半径变量 radius，相当于公开的实现细节。如果我们改变主意，想使用直径来表示一个圆形，那么实现的修改就会显得很丑陋。

可是，如果我们认真思考一下几个简单的问题，对于封装的顾虑可能就降低很多了。比如说，使用直径来表示一个圆，这是一个真实的需求吗？这是一个必需的表达方式吗？未来的圆，会不会变得没法使用半径来表达？其实不是的，未来的圆，还是可以用半径来表达的。使用其他的办法，比如直径，来表达一个圆，其实并没有必要。

所以，公开半径这个只读变量，并没有带来违反封装原则的实质性后果。而且，从另外一个角度来看，我们可以把读取这个只读变量的操作，看成是等价的读取方法的调用。不过，虽然很多人，包括我自己，倾向于这样解读，但是这总归是一个有争议的形式。

进一步的简化

还有没有进一步简化的空间呢？我们再来看看不可变的正方形 **Square** 类的设计。具体的实现，请参考下面的代码。

```
package co.ivi.jus.record.immute;
public final class Square implements Shape {
    public final double side;
    public Square(double side) {
        this.side = side;
    }
    @Override
    public double area() {
        return side * side;
    }
}
```

如果比较一下不可变的圆形 Circle 类和正方形 Square 类的源代码，你有没有发现这两个类的代码有惊人的相似点？

第一个相似的地方，就是使用公开的只读变量（使用 final 修饰符来声明只读变量）。Circle 类的变量 **radius**，和 Square 类的变量 **side**，都是公开的只读的变量。这样的声明，是为了公开变量的只读性。

第二个相似的地方，就是公开的只读变量，需要在构造方法中赋值，而且只在构造方法中赋值，且这样的构造方法还是公开的方法。Circle 类的构造方法给 **radius** 变量赋值，Square

类的构造方法给 side 变量赋值。这样的构造方法，解决了对象的初始化问题。
第三个相似的地方，就是没有了读取的方法；公开的只读变量，替换了掉了公开的读取方法。这样的变化，使得代码量总体变少了。
这么多相似的地方，相似的代码，能不能进一步地简化呢？我知道，你可能已经开始思考这样的问题了。
对于这个问题，Java 的答案，就是使用档案类。

怎么声明档案类

我们前面说过，Java 档案类是用来表示不可变数据的透明载体。那么，怎么使用档案类来表示不可变数据呢？
我们还是一起先来看看代码吧。咱们试着把上面不可变的圆形 Circle 普通的类改成档案类，来感受下档案类到底是什么模样的。

```
package co.ivi.jus.record.modern;
public record Circle(double radius) implements Shape {
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

看到这样的代码，是不是有点出乎意料？你可以对比一下不可变的 Circle 类的代码，感受一下这两者之间的差异。
首先，最常见的 class 关键字不见了，取而代之的是 record 关键字。record 关键字是 class 关键字的一种特殊表现形式，用来标识档案类。record 关键字可以使用和 class 关键字差不多一样的类修饰符（比如 public、static 等；但是也有一些例外，我们后面再说）。
然后，类标识符 Circle 后面，有用小括号括起来的参数。类标识符和参数一起看，就像是一个构造方法。事实上，这样的表现方式，的确可以看成是构造方法。而且，这种形式，还就是当作构造方法使用的。比如下面的代码，就是使用构造方法的形式来生成 Circle 档案类实例的。

```
Circle circle = new Circle(10.0);
```

最后，在大括号里，也就是档案类的实现代码里，变量的声明没有了，构造方法也没有了。
前面我们已经知道怎么生成一个档案类实例了，但还有一个问题是，我们能读取这个圆形档案类的半径吗？
其实，类标识符声明后面的小括号里的参数，就是等价的不可变变量。在档案类里，这样的不可变变量是私有的变量，我们不可以直接使用它们。但是我们可以通过等价的方法来调用它们。变量的标识符就是等价方法的标识符。比如下面的代码，就是一个读取上面圆形档案类半径的代码。

```
double radius = circle.radius();
```

是的，在档案类里，方法调用的形式又回来了。我们前面讨论过打破封装原则的顾虑，你可能还是没有足够的信心去接受不完整的封装形式。那么现在，档案类的调用形式依然保持着良好的封装形式。打破封装原则的顾虑也就不复存在了。
需要注意的是，由于档案类表示的是不可变数据，除了构造方法之外，并没有给不可变变量赋值的方法。

意料之外的改进

上面，通过传统 Circle 类和档案 Circle 类代码的对比，我们可以感受到档案类在简化代码、提高生产力方面的努力。如果说，上面这些简化，还在我的预料之内的话；下面的简化，我刚看到的时候，是很惊喜的：“哇，这真是太奇妙了！”
我们还是通过代码来体验一下这种感受。如果我们生成两个半径为 10 厘米的圆形的实例，这两个实例是相等的吗？下面的代码，就是用来验证我们猜想的。你可以试着运行一下，看看和你猜想的结果是不是一样的。

```
package co.ivi.jus.record;
import co.ivi.jus.record.immute.Circle;
public class ImmuteUseCases {
    public static void main(String[] args) {
        Circle c1 = new Circle(10.0);
        Circle c2 = new Circle(10.0);
        System.out.println("Equals? " + c1.equals(c2));
    }
}
```

上面的代码里，使用了我们开篇案例分析中的传统 Circle 类。运行结果告诉我们，两个半径为 10 厘米的圆形的实例，并不是相等的实例。我想这应该在你的预料之内。

如果需要比较两个实例是不是相等，我们需要重载 equals 方法和 hashCode 方法。如果需要把实例转换成肉眼可以阅读的信息，我们需要重载 toString 方法。我们上面案例分析的代码中，这些方法都没有重载，因此对应的操作结果也是不可预测的。

当然，如果没有遗忘，我们可以添加这三个方法的重载实现。然而，这三个方法的重载，尤其是 equals 方法和 hashCode 方法的重载实现，一直是代码安全的重灾区。即便是经验丰富的程序员，也可能忘记重载这三个方法；就算没有遗忘，equals 方法和 hashCode 方法也可能没有正确实现，从而带来各种各样的问题。这实在难以让人满意，但是一直以来，我们也没有更好的办法。

档案类会不一样吗？

我们再来看看使用档案类的代码，结果会不会不一样呢？下面的这段代码，Circle 的实现使用的是档案类。这段代码运行的结果告诉我们，两个半径为 10 厘米的圆形的档案类实例，是相等的实例。

```
package co.ivi.jus.record;
import co.ivi.jus.record.modern.Circle;
public class ModernUseCases {
    public static void main(String[] args) {
        Circle c1 = new Circle(10.0);
        Circle c2 = new Circle(10.0);
        System.out.println("Equals? " + c1.equals(c2));
    }
}
```

看到这里，你是不是感觉到：哇！这真的是太棒了！我们并没有重载这三个方法，它们居然可以使用。

为什么会这样呢？

这是因为，档案类内置了缺省的 equals 方法、hashCode 方法以及 toString 方法的实现。一般情况下，我们就再也不用担心这三个方法的重载问题了。这不仅减少了代码数量，提高了编码的效率；还减少了编码错误，提高了产品的质量。

不可变的数据

讨论到这里，我们可以回头再看看 Java 档案类的定义了：Java 档案类是用来表示不可变数据的透明载体。“不可变的数据”和“透明的载体”是两个最重要的关键词。

我们前面讨论了不可变的数据。如果一个 Java 类一旦实例化就不能再修改，那么用它表述的数据就是不可变数据。Java 档案类就是表述不可变数据的。为了强化“不可变”这一原则，避免面向对象设计的陷阱，Java 档案类还做了以下的限制：

1. Java 档案类不支持扩展子句，用户不能定制它的父类。隐含的，它的父类是 java.lang.Record。父类不能定制，也就意味着我们不能通过修改父类来影响 Java 档案的行为。
2. Java 档案类是个终极（final）类，不支持子类，也不能是抽象类。没有子类，也就意味着我们不能通过修改子类来改变 Java 档案的行为。
3. Java 档案类声明的变量是不可变的变量。这就是我们前面反复强调的，一旦实例化

就不能再修改的关键所在。

4. Java 档案类不能声明可变的变量，也不能支持实例初始化的方法。这就保证了，我们只能使用档案类形式的构造方法，避免额外的初始化对可变性的影响。
5. Java 档案类不能声明本地（native）方法。如果允许了本地方法，也就意味着打开了修改不可变变量的后门。

通常地，我们把 Java 档案类看成是一种特殊形式的 Java 类。除了上述的限制，Java 档案类和普通类的用法是一样的。

透明的载体

好了，聊完“不可变的数据”，接下来该聊聊“透明的载体”了。

陆陆续续地，我们在前面提到过，档案类内置了下面的这些方法缺省实现：

- 构造方法
- equals 方法
- hashCode 方法
- toString 方法
- 不可变数据的读取方法

如果你注意到的话，我们使用了“缺省”这样的字眼。换一种说法，我们可以使用缺省的实现，也可以替换掉缺省的实现。下面的代码，就是我们试图替换掉缺省实现的尝试。请注意，除了构造方法，其他的替换方法都可以使用 Override 注解来标注（如果你读过《代码精进之路》，你就会倾向于总是使用 Override 注解的）。

```
package co.ivi.jus.record.explicit;
import java.util.Objects;
public record Circle(double radius) implements Shape {
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }

        if (o instanceof Circle other) {
            return other.radius == this.radius;
        }
        return false;
    }
    @Override
    public int hashCode() {
        return Objects.hash(radius);
    }
    @Override
    public String toString() {
        return String.format("Circle[radius=%f]", radius);
    }
    @Override
    public double radius() {
```

```
        return this.radius;
    }
}
```

到这里，你应该明白了“透明的载体”的意思了。透明载体的意思，通俗地说，就是档案类承载有缺省实现的方法，这些方法可以直接使用，也可以替换掉。

不过，像上面这样的替换，除了徒增烦恼，是没有实际意义的。那我们什么时候需要替换掉缺省实现呢？

重载构造方法

最常见的替换，是要在构造方法里对档案类声明的变量添加必要的检查。比如说，我们现实生活中看到的各种各样的圆形，它的半径都不会是负数。如果在这样的场景里来讨论圆形，那么表示圆形的类的半径就不应该是负数。

你应该已经意识到了，我们上面的代码，在实例化的时候，都没有检查半径的数值，包括档案类缺省的构造方法。那么这时候，我们就要替换掉缺省的构造方法。下面的代码，就是一种替换的方法。如果，构造实例的时候，半径的数值为负，构造就会抛出运行时异常 **IllegalArgumentException**。

```
package co.ivi.jus.record.improved;
public record Circle(double radius) implements Shape {
    public Circle {
        if (radius < 0) {
            throw new IllegalArgumentException(
                "The radius of a circle cannot be negative [" + radius + "]");
        }
    }
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

如果你阅读了上面的代码，应该已经注意到了一点不太常规的形式。构造方法的声明没有参数，也没有给实例变量赋值的语句。这并不是说，构造方法就没有参数，或者实例变量不需要赋值。实际上，为了简化代码，Java 编译的时候，已经替我们把这些东西加上去了。所以，不论哪一种编码形式，构造方法的调用都是没有区别的。

在上一个例子中，我们已经看到了构造方法的常规形式。在下面这张表里，我列出了两种构造方法形式上的差异，你可以看看它们的差异。

声明的形式	代码	注释

常规的形式	public Circle(double radius) {
	  if (radius < 0) {
	  // snipperd
	  }
	
	  this.radius = radius;
	}
		1、构造方法的声明包含参数。
		2、有给实例的变量赋值的语句。

简化的形式	public Circle {
	  if (radius < 0) {
	  // snipped
	  }
	}
		1、构造方法的声明不包含参数，也没有小括号。
		2、没有给实例变量赋值的语句。

重载 equals 方法

还有一类常见的替换，如果缺省的 equals 方法或者 hashCode 方法不能正常工作或者存在安全的问题，就需要替换掉缺省的方法。

如果声明的不可变变量没有重载 equals 方法和 hashCode 方法，那么这个档案类的 equals

方法和 hashCode 方法的行为就可能不是可以预测的。比如，如果不可变的变量是一个数组，通过下面的例子，我们来看看它的 equals 方法能不能正常工作。

```
jshell> record Password(byte[] password) {};
|  modified record Password
jshell> Password pA = new Password("123456".getBytes());
pA ==> Password[password=[B@2ef1e4fa]
jshell> Password pB = new Password("123456".getBytes());
pB ==> Password[password=[B@b81eda8]
jshell> pA.equals(pB);
$16 ==> false
```

这个例子里，我们设计了一个口令的档案类，其中的口令使用字节数组来存放。我们使用同样的口令，生成了两个不同的实例。然后，我们调用 equals 方法，来比较这两个实例。运算的结果显示，这两个实例并不相等。这不是我们期望的结果。其中的原因，就是因为数组这个变量的 equals 方法并不能正常工作（或者换个说法，数组变量没有重载 equals 方法）。

如果把变量的类型换成重载了 equals 方法的字符串 String，我们就能看到预期的结果了。

```
jshell> record Password(String password) {};
|  created record Password
jshell> Password pA = new Password("123456");
pA ==> Password[password=123456]
jshell> Password pB = new Password("123456");
pB ==> Password[password=123456]
jshell> pA.equals(pB);
$5 ==> true
```

一般情况下，equals 方法和 hashCode 方法是成双成对的，实现逻辑上需要匹配。所以，当我们重载 equals 方法的时候，一般也需要重载 hashCode 方法；反之亦然。

不推荐的重载

为了更个性化的显示，我们有时候也需要重载 toString 方法。但是，我们通常不建议重载不可变数据的读取方法。因为，这样的重载往往意味着需要变更缺省的不可变数值，从而打破实例的状态，进而造成许多无法预料的、让人费解的后果。

比如说，我们设想定义一个数，如果是负值的话，我们希望读取的是它的相反数。下面的例子，就是一个味道很坏的示范。

```
jshell> record Number(int x) {
   ...>     public int x() {
   ...>         return x > 0 ? x : (-1) * x;
   ...>     }
   ...> }
|  created record Number
jshell> Number n = new Number(-1);
n ==> Number[x=-1]
jshell> n.x();
$9 ==> 1
jshell> Number m = new Number(n.x());
m ==> Number[x=1]
jshell> m.equals(n);
$11 ==> false
```

在这个例子里，我们重载了读取的方法。如果一个数是负数，重载的读取就返回它的相反数。读取出来的数据，并不是实例化的时候赋于的数据。这让代码变得难以理解，很容易出错。更严重的问题是，这样的重载不再能够支持实例的拷贝。比如说，我们把实例 n 拷贝到另一个实例 m。这两个实例按照道理来说应该相等。而由于重载了读取的方法，实际的结果，这

两个实例是不相等的。这样的结果，也可能会使代码容易出错，而且难以调试。

总结

好，今天就到这里，我来做个小结。从前面的讨论中，我们了解到，Java 档案类是用来表示不可变数据的透明载体，用来简化不可变数据的表达，提高编码效率，降低编码错误。同时，我们也讨论了使用档案类的几个容易忽略的陷阱。

在我们日常的接口设计和编码实践中，为了最大化的性能，我们应该优先考虑使用不可变的对象（数据）；如果一个类是用来表述不可变的对象（数据），我们应该优先使用 Java 档案类。

如果要丰富你的代码评审清单，有了封闭类后，你可以加入下面这一条：

一个类，如果是用来表述不可变的数据，能不能使用 Java 档案类？

另外，通过今天的讨论，我拎出几个技术要点，这些都可能在你们面试中出现哦，通过学习，你应该能够：

- 知道 Java 支持档案类，并且能够有意识地使用档案类，提高编码效率，降低编码错误；
    - 面试问题：你知道档案类吗？会不会使用它？
- 了解档案类的原理和它要解决的问题，知道使用不可变的对象优势；
    - 面试问题：什么情况下可以使用档案类，什么情况下不能使用档案类？
- 了解档案类的缺省方法，掌握缺省方法的好处和不足，知道什么时候要重载这些方法。
    - 面试问题：使用档案类应该注意什么问题？

如果你能够有意识地使用不可变的对象以及档案类，并且有能力规避掉其中的陷阱，你应该能够大幅度提高编码的效率和质量。毫无疑问，在面试的时候，这也是一个能够让你脱颖而出的知识点。

思考题

在重载 equals 方法这一小节里，我们讨论了数组类型的不可变数据。我们已经知道了，这样的数据类型，需要重载 equals 方法和 hashCode 方法。其实，toString()的方法也需要重载。今天的思考题，就是请你实现这些方法的重载。

方便起见，我们假设这个数组是字节数组，用来表示社会保障号。我们都知道，社会保障号是高度敏感的信息，不能被泄漏，也不能被盗取。你来想一想，有哪些方法需要重载？为什么？代码看起来是什么样子的？有难以克服的困难吗？

我开个头，写一个空白的档案类，你来把你想添加的代码补齐。

```
record SocialSecurityNumber(byte[] ssn) {
  // Here is your code.
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及对这些问题的思考。

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 reviewtemplate 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在档案类专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 record/review/xuelei 的目录下面。

04 | 封闭类：怎么刹住失控的扩展性？

你好，我是范学雷。今天，我们聊一聊 Java 的封闭类。

封闭类这个特性，首先在 JDK 15 中以预览版的形式发布。在 JDK 16 中，改进的封闭类再次以预览版的形式发布。最后，封闭类在 JDK 17 正式发布。

那么，什么是封闭类呢？封闭类的英文，使用的词汇是"sealed classes"。从名字我们就可以感受到，封闭类首先是 Java 的类，然后它还是封闭的。

Java 的类，我们都知道什么意思。那么，“封闭”又是什么意思呢？字面的意思，就是把一些东西封存起来，里面的东西出不去，外面的东西也进不来，所以可查可数。

“封闭”、“可查可数”，这些词汇字面看起来好像很通俗，但是实际上并不容易理解。我们还是通过案例和代码，一步一步地来了解封闭类吧。

阅读案例

在面向对象的编程语言中，研究表示形状的类，是一个常用的教学案例。今天的评审案例，我们也从形状这个类开始，来研究一下怎么判断一个形状是不是正方形吧。

下面的这段代码，就是一个简单的、抽象的形状类的定义。这个抽象类的名字是 Shape。它有一个抽象方法 area()，用来计算形状的面积。它还有一个公开的属性 id，用来标识这个形状的对象。

```
package co.ivi.jus.sealed.former;
public abstract class Shape {
    public final String id;

    public Shape(String id) {
        this.id = id;
    }

    public abstract double area();
}
```

我们都知道，正方形是一个形状。正方形可以作为形状这个类的一个扩展类。它的代码可以是下面的样子。

```
package co.ivi.jus.sealed.former;
public class Square extends Shape {
    public final double side;

    public Square(String id, double side) {
        super(id);
        this.side = side;
    }

    @Override
    public double area() {
        return side * side;
    }
}
```

那么，到底怎么判断一个形状是不是正方形呢？这个问题的答案，表面上看起来很简单，只要判断这个形状的对象是不是一个正方形的实例就可以了。这个判断的例子，看起来可以是下面的样子。

```
static boolean isSquare(Shape shape) {
    return (shape instanceof Square);
}
```

你可以思考一下，这样是不是真的能判断一个形状是正方形？花几秒钟想想你的答案，我们

接下来再继续分析。

案例分析

其实，上面的这个例子，判断的只是“一个形状的对象是不是一个正方形的实例”。但实际上，一个形状的对象即使不是一个正方形的类，它也有可能是一个正方形。什么意思呢？比如说有一个对象，表示它的类是长方形或者菱形的类。如果这个对象的每一个边的长度都是一样的，其实它就是一个正方形，但是表示它的类是长方形或者菱形的类，而不是正方形类。所以，上面的这段代码还是有缺陷的，并不总是能够正确判断一个形状是不是正方形。

详细地，我们来看下一段代码，你就对这个缺陷有一个更直观的了解了。我们都知道，长方形也是一个形状，它也可以作为形状这个类的一个扩展类。下面的这段代码，定义的就是一个长方形。这个类的名字是 Rectangle，它是 Shape 的扩展类。

```
package co.ivi.jus.sealed.former;
public class Rectangle extends Shape {
    public final double length;
    public final double width;

    public Rectangle(String id, double length, double width) {
        super(id);
        this.length = length;
        this.width = width;
    }

    @Override
    public double area() {
        return length * width;
    }
}
```

代码读到这里，对于“怎么判断一个形状是不是正方形”这个问题，我觉得你可能已经有了一个更好的思路。没错，正方形是一个特殊的长方形。如果一个长方形的长和宽是相等的，那么它也是一个正方形。上面的那段“判断一个形状是不是正方形”的代码，就没有考虑到长方形的特例，所以它是有缺陷的实现。

知道了长方形这个类，我们就能改进我们的判断了。改进的代码，要把长方形考虑进去。它看起来可以是下面的样子。

```
public static boolean isSquare(Shape shape) {
    if (shape instanceof Rectangle rect) {
        return (rect.length == rect.width);
    }

    return (shape instanceof Square);
}
```

写完上面的代码，似乎就可以长舒一口气：哎，这难缠的正方形，我们终于搞定了。

但其实，这个问题我们还没有搞定。因为正方形也是一个特殊的菱形，如果一个对象是一个菱形类的实例，上面的代码就有缺陷。更令人窘迫的是，正方形还是一个特殊的梯形，还是一个特殊的多边形。随着我们学习一步一步的深入，我们知道还有很多形状的特殊形式是正方形，而且我们并不知道我们知识范围外的那些形状，当然更不能提穷举它们了。

这，实在有点让人抓狂！

问题出在哪里呢？**无限制的扩展性，是问题的根源**。正如现实世界里，我们没有办法穷举到底有多少形状的特殊形式是正方形；在计算机的世界里，我们也没有办法穷举到底有多少形状的对象可以是正方形。如果我们解决不了形状类的穷举问题，我们就不太容易使用代码来判断一个形状是不是正方形。

而解决问题的办法，就是限制可扩展类的扩展性。

怎么限制住扩展性？

你可能要问，可扩展性不是面向对象编程的一个重要指标吗？为什么要限制可扩展性呢？其实，面向对象编程的最佳实践之一，就是要把可扩展性限制在可以预测和控制的范围内，而不是无限的可扩展性。

除了上面穷举的问题之外，在极客时间专栏《代码精进之路》里，我们还讨论了继承的安全缺陷。其中，主要有两点值得我们格外小心：

一个可扩展的类，子类和父类可能会相互影响，从而导致不可预知的行为。

涉及敏感信息的类，增加可扩展性不一定是个优先选项，要尽量避免父类或者子类的影响。

虽然我们使用了 Java 语言来讨论继承的问题，但其实这些是面向对象机制的普遍问题，甚至它们也不单单是面向对象语言的问题，比如使用 C 语言的设计和实现，也存在类似的问题。

由于继承的安全问题，我们在设计 API 时，有两个要反省思考的点：

一个类，有没有真实的可扩展需求，能不能使用 final 修饰符？

一个方法，子类有没有重写的必要性，能不能使用 final 修饰符？

限制住不可预测的可扩展性，是实现安全代码、健壮代码的一个重要目标。

JDK 17 之前的 Java 语言，限制住可扩展性只有两个方法，使用私有类或者 final 修饰符。显而易见，私有类不是公开接口，只能内部使用；而 final 修饰符彻底放弃了可扩展性。要么全开放，要么全封闭，可扩展性只能在可能性的两个极端游走。全封闭彻底没有了可扩展性，全开放又面临固有的安全缺陷，这种二选一的状况有时候很让人抓狂，特别是设计公开接口的时候。

JDK 17 之后，有了第三种方法。这个办法，就是使用 Java 的 **sealed** 关键字。使用类修饰符 **sealed** 修饰的类是封闭类；使用类修饰符 **sealed** 修饰的接口是封闭接口。封闭类和封闭接口限制可以扩展或实现它们的其他类或接口。

通过把可扩展性的限制放在可以预测和控制的范围内，封闭类和封闭接口打开了全开放和全封闭两个极端之间的中间地带，为接口设计和实现提供了新的可能性。

怎么声明封闭类

那么，怎么使用封闭类呢？封闭类这个概念，涉及到两种类型的类。第一种是被扩展的父类，第二种是扩展而来的子类。通常地，我们把第一种称为封闭类，第二种称为许可类。

封闭类的声明使用 sealed 类修饰符，然后在所有的 extends 和 implements 语句之后，使用 permits 指定允许扩展该封闭类的子类。比如，使用 sealed 类修饰符，我们可以把形状这个类声明为封闭类。下面的这个例子中，Shape 是一个封闭类，可以扩展它的子类只有两个，分别为 Circle 和 Square。也就是说，这里定义的形状这个类，只允许有圆形和正方形两个子类。

```
package co.ivi.jus.sealed.modern;
public abstract sealed class Shape permits Circle, Square {
    public final String id;
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();
}
```

由 permits 关键字指定的许可子类（permitted subclasses），必须和封闭类处于同一模块（module）或者包空间（package）里。如果封闭类和许可类是在同一个模块里，那么它们可以处于不同的包空间里，就像下面的例子。

```
package co.ivi.jus.sealed.modern;
public abstract sealed class Shape
    permits co.ivi.jus.ploar.Circle,
            co.ivi.jus.quad.Square {
    public final String id;
```

```
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();
}
```

如果允许扩展的子类和封闭类在同一个源代码文件里，封闭类可以不使用 permits 语句，Java 编译器将检索源文件，在编译期为封闭类添加上许可的子类。比如下面的两种 Shape 封闭类的声明，一个封闭类使用了 permits 语句，另外一个封闭类没有使用 permits 语句。但是，这两个声明具有完全一样的运行时效果。

```
package co.ivi.jus.sealed.improved;
public abstract sealed class Shape {
    public final String id;
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();
    public static final class Circle extends Shape {
        // snipped
    }
    public static final class Square extends Shape {
        // snipped
    }
}
```

```
package co.ivi.jus.sealed.improved;
public abstract sealed class Shape
         permits Shape.Circle, Shape.Square {
    public final String id;
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();
    public static final class Circle extends Shape {
        // snipped
    }
    public static final class Square extends Shape {
        // snipped
    }
}
```

不过，如果你读过《代码精进之路》，你就会倾向于总是使用 permits 语句。因为这样的话，代码的阅读者不需要去翻找上下文，也能一目了然地知道这个封闭类支持哪些许可类。这会给代码的阅读者带来很多的便利，包括节省时间以及少犯错误。

怎么声明许可类

许可类的声明需要满足下面的三个条件：

- 许可类必须和封闭类处于同一模块（module）或者包空间（package）里，也就是说，在编译的时候，封闭类必须可以访问它的许可类；
- 许可类必须是封闭类的直接扩展类；
- 许可类必须声明是否继续保持封闭：
    - 许可类可以声明为终极类（final），从而关闭扩展性；
    - 许可类可以声明为封闭类（sealed），从而延续受限制的扩展性；
    - 许可类可以声明为解封类（non-sealed），从而支持不受限制的扩展性。

比如在下面的例子中，许可类 Circle 是一个解封类；许可类 Square 是一个封闭类；许可类 ColoredSquare 是一个终极类；而 ColoredCircle 既不是封闭类，也不是许可类。

```
package co.ivi.jus.sealed.propagate;
public abstract sealed class Shape {
    public final String id;
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();

    public static non-sealed class Circle extends Shape {
        // snipped
    }

    public static sealed class Square extends Shape {
        // snipped
    }

    public static final class ColoredSquare extends Square {
        // snipped
    }
    public static class ColoredCircle extends Circle {
        // snipped
    }
}
```

需要注意的是，**由于许可类必须是封闭类的直接扩展，因此许可类不具备传递性。**也就是说，上面的例子中，ColoredSquare 是 Square 的许可类，但不是 Shape 的许可类。

案例回顾

到这里，我们再回头看看前面的案例，怎么判断一个形状是不是正方形呢？封闭类能帮助我们解决这个问题吗？如果使用了封闭类，这个问题的答案也就呼之欲出了。

首先，我们要把形状这个类定义为封闭类。这样，所有形状的子类就可以穷举了。然后，我们寻找可以用来表示正方形的许可类。找到这些许可类后，只要我们能够判断这个形状的对象是不是一个正方形，问题就解决了。

比如下面的代码，形状被定义为封闭类 Shape。而且，Shape 这个封闭类只有两个终极的许可类。一个许可类是表示圆形的 Circle，一个许可类是表示正方形的 Square。

```
package co.ivi.jus.sealed.improved;
public abstract sealed class Shape
         permits Shape.Circle, Shape.Square {
    public final String id;
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();
    public static final class Circle extends Shape {
        // snipped
    }
    public static final class Square extends Shape {
        // snipped
    }
}
```

由于 Shape 是个封闭类，在这段代码的许可范围内，一个形状 Shape 的对象要么是一个圆形 Circle 的实例，要么是一个正方形 Square 的实例，没有其他的可能性。
这样的话，判断一个形状是不是正方形这个问题就变得比较简单了。只要能够判断出来一个形状的对象是不是一个正方形的实例，这个问题就算是解决了。

```
static boolean isSquare(Shape shape) {
    return (shape instanceof Square);
}
```

这样的逻辑在案例分析那一小节的场景中并不成立，为什么现在就成立了呢？根本的原因，在案例分析那一小节的场景中，Shape 类是一个不受限制的类，我们没有办法知道它所有的扩展类，因此我们也就没有办法穷尽正方形的所有可能性。而在使用封闭类的场景下，Shape 类的所有扩展类，我们都是已知的，所以我们就有办法检查每一个扩展类的规范，从而对这个问题做出正确的判断。

总结

好，到这里，我来做个小结。从前面的讨论中，我们了解到，可扩展性的限定方法有四个：

1. 使用私有类；
2. 使用 final 修饰符；
3. 使用 sealed 修饰符；
4. 不受限制的扩展性。

在我们日常的接口设计和编码实践中，使用这四个限定方法的优先级应该是由高到低的。最优先使用私有类，尽量不要使用不受限制的扩展性。
如果要丰富你的代码评审清单，有了封闭类后，你可以加入下面这一条：
一个类，如果有真实的可扩展需求，能不能枚举，可不可以使用 sealed 修饰符？
另外，通过今天的讨论，我拎出几个技术要点，这些都可能在你们面试中出现哦，通过学习，你应该能够：

- 知道 Java 支持封闭类，并且能够使用封闭类编写代码；
    - 面试问题：你知道封闭类吗？会不会使用它？
- 了解封闭类的原理和它要解决的问题，知道限制住扩展性的办法；
    - 面试问题：面向对象编程的可扩展性有什么问题吗？该怎么处理这些问题？
- 能够有意识地使用封闭类来限制类或者接口的扩展性。
    - 面试问题：你写的这段代码，是不是应该使用 final 修饰符或者 sealed 修饰符？

如果你的代码里使用了封闭类，无论是面试的时候还是工作的时候，一定能够给人深刻的印象。因为，这意味着你已经了解了可扩展性的危害，并且有办法降低这种危害的影响，有能力编写出更健壮的代码。

思考题

在案例回顾这一小节里，我们使用了封闭类来解决“怎么判断一个形状是不是正方形”这个问题。我们假设案例回顾这一小节的代码是版本 1.0。现在我们假设，在版本 2.0 里，需要增加另一个许可类，用来支持长方形（Rectangle）。那么：

1. 封闭类的代码该怎么改动，才能支持长方形？
2. “判断一个形状是不是正方形”的代码该怎么改动，才能适应封闭类的改变？
3. 增加一个许可类，会有兼容性的影响吗？比如说，使用版本 1.0 来判断一个形状是不是正方形的代码还能使用吗？

欢迎你在留言区留言、讨论，分享你的阅读体验以及对这些问题的思考。
注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在封闭类专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 sealed/review/xuelei 的目录下面。

05 | 类型匹配：怎么切除臃肿的强制转换？

你好，我是范学雷。今天，我们聊一聊 Java 模式匹配，主要是类型匹配。

Java 的模式匹配是一个新型的、而且还在持续快速演进的领域。类型匹配是模式匹配的一个规范。类型匹配这个特性，首先在 JDK 14 中以预览版的形式发布。在 JDK 15 中，改进的类型匹配再次以预览版的形式发布。最后，类型匹配在 JDK 16 正式发布。

那么，什么是模式匹配，什么又是类型匹配呢？这就要说到模式的组成。通常，一个模式是匹配谓词和匹配变量的组合。其中，匹配谓词用来确定模式和目标是否匹配。在模式和目标匹配的情况下，匹配变量是从匹配目标里提取出来的一个或者多个变量。

对于类型匹配来说，匹配谓词用来指定模式的数据类型，而匹配变量就是一个属于该类型的数据变量。需要注意的是，对于类型匹配来说，匹配变量只有一个。

这样的描述还是太抽象，太难理解。我们还是通过案例和代码，一点一点地来理解类型匹配吧。

阅读案例

在程序员的日常工作中，一个重要的事情，就是把相似的东西抽象出来，设计成一个通用的、可以复用的接口。

比如说，我们从正方形、长方形、圆形这些看起来差异巨大的东西出发，抽象出了形状这个接口。我们希望使用一个实例时，如果我们不能确定它是正方形还是长方形，我们至少还能确定它是一个形状。这种模模糊糊的确定性（其实也是不确定性），其实对我们编写代码有巨大的帮助，包括但是不限于简化代码逻辑，减少代码错误。

但要注意的是，每一个实例都是具体的形状。它可以是正方形的对象，可以是长方形的对象，就是不能是一个抽象的形状。也就是说，抽象的类和接口不能直接实例化。

一个方法的规范，它的输入参数可能是一个表示形状的对象，也可能是一个更一般化的对象。比如说吧，我们要设计一个方法，来判断一个形状是不是正方形。那么，就需要一个表示形状的对象，作为这个方法的输入参数。而实现这个方法的代码，仅仅知道形状这个一般化的对象是远远不够的。下面的代码，就是一个这种方法的实现代码。

```
static boolean isSquare(Shape shape) {
    if (shape instanceof Rectangle) {
        Rectangle rect = (Rectangle) shape;
        return (rect.length == rect.width);
    }
    return (shape instanceof Square);
}
```

在这个 isSquare 方法的实现代码里，我们需要使用 instanceof 运算符，来判断输入参数是不是一个长方形的实例；如果判断成立，再使用类型转换运算符，把这个实例投射成长方形的实例；最后，我们开始使用这个长方形的实例，进行更多的运算。

其实，这样的操作是一个模式化的过程。如果我们把它揉碎了来看，这个模式有三个部分。第一个部分是类型判断语句，也就是匹配谓词，使用的代码是“instanceof Rectangle”。第二个部分是类型转换语句，使用的是类型转换运算符（(Rectangle)shape）。第三个部分是声明一个新的本地变量，也就是匹配变量，来承载转换后的数据，使用的是变量声明和赋值运算符（Rectangle rect =）。第二个部分和第三个部分，只有在类型判断成立的情况下，才能够执行。

使用这样的模式化操作，是一个 Java 程序员的基本功。这个模式直观而且便于理解。可是，这个模式很乏味，也很臃肿。调用了 instanceof 之后，除了类型转换之外，我们还可以做什么呢？一般情况下，在类型判断之后，我们总是紧跟着就进行类型转换。

把类型判断和类型转换切割成两个部分，增加了错误潜入的机会，平添了许多烦恼。比如说，一个活生生的程序员或者冷冰冰的机器，有可能无意地使用了错误的类型。下面例子中的两段代码，就是两个常见的类型转换错误。第一段代码误用了变量类型，第二段代码误用了判断结果。

```
if (shape instanceof Rectangle) {
```

```
    Rectangle rect = (Rectangle) shape;
        return (rect.length == rect.width);
}
if (!(shape instanceof Rectangle) {
    Rectangle rect = (Rectangle) shape;
        return (rect.length == rect.width);
}
```

类型判断之后，我们原本就可以开始关注更重要的后续代码逻辑了，但现在不得不停下来编写类型转化代码，或者审视类型转换代码是否恰当。这当然影响力了生产效率。
我们可以用什么方法改进这个模式，提高生产效率呢？这个问题的答案就是类型匹配。

类型匹配

那么，类型匹配是怎么改进这个模式的呢？我们先来看看使用了类型匹配的代码的样子。下面的例子，就是使用类型匹配的一段代码。

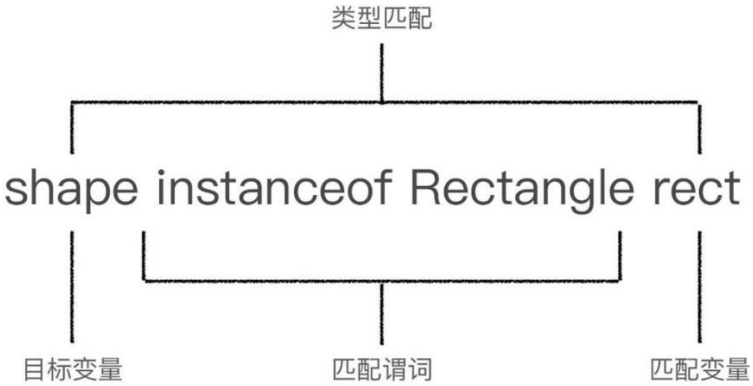
```
if (shape instanceof Rectangle rect) {
    return (rect.length == rect.width);
}
```

为了便于更直观地比较，我把传统的实现代码和使用了类型匹配的实现代码列在了下面的表格里。你可以找找其中的差异，体会下类型匹配带来的改进。

传统的实现代码	类型匹配的代码

if (shape instanceof Rectangle) {
  Rectangle rect = (Rectangle) shape;
  // snipped
}
	if (shape instanceof Rectangle rect) {
	
	  // snipped
	}

就像我们前面拆解的一样，传统的实现代码有三个部分；而使用类型匹配的代码，只有匹配谓词和本地变量两个部分，而且是在同一个语句里。为了帮助你理解这些概念，我画了下面的这张图，标记出了类型匹配的组成部分和关键概念。



你可能已经注意到了，使用类型转换运算符的语句，没有出现在使用类型匹配的代码里。但是，这并不影响类型匹配代码所要表达的基本逻辑。
这个基本逻辑就是：如果目标变量是一个长方形的实例，那么这个目标变量就会被赋值给一个本地的长方形变量，也就是我们所说的匹配变量；相反，如果目标变量不是一个长方形的实例，那么这个匹配变量就不会被赋值。
前面，我们讨论了两个常见的类型转换错误：误用变量类型和误用判断结果。在使用类型匹配的代码里，不再需要重复使用匹配类型，也不再需要使用强制类型转换符。所以，使用类型匹配的代码，不用再担心误用变量类型的错误了。
误用判断结果的错误，是不是也被解决了呢？似乎，我们还能写出下面的代码。在这样的代码里，如果目标变量不是一个长方形的实例，我们是不是也有可能使用匹配的变量呢？

```
if (!(shape instanceof Rectangle rect)) {
    return (rect.length == rect.width);
```

```
}
```
幸运的是，类型匹配已经考虑到了这个问题，Java 编译器能够检测出上面的错误，不会允许使用没有赋值的匹配变量。这样，在代码编译期间，就有机会纠正代码的错误。比如说，我们可以尝试修改成下面的逻辑：如果目标变量不是一个长方形的实例，我们就不使用匹配变量；否则，我们就使用匹配变量。把这个逻辑映射到代码，大致是下面的样子。
```
if (!(shape instanceof Rectangle rect)) {
    return false;
} else {
    return (rect.length == rect.width);
}
```
在上面的代码里，使用匹配变量的条件语句 else 分支并没有声明这个匹配变量。为什么 if 语句声明的变量，可以在 else 语句里使用呢？要弄清楚这个问题，我们还要了解匹配变量的作用域。掌握匹配变量的作用域，是学会使用类型匹配的关键。

匹配变量的作用域

匹配变量的作用域，就是目标变量可以被确认匹配的范围。如果在一个范围内，无法确认目标变量是否被匹配，或者目标变量不能被匹配，都不能使用匹配变量。如果我们从编译器的角度去理解，也就是说，在一个范围里，如果编译器能够确定匹配变量已经被赋值了，那么它就可以在这个范围内使用；如果编译器不能够确定匹配变量是否被赋值，或者确定没有被赋值，那么他就不能在这个范围内使用。
我们还是通过代码来理解这个有点抽象的概念吧。
第一段代码，我们看看最常规的使用。我们可以在确认类型匹配的条件语句之内使用匹配变量。这个条件语句之外，不是匹配变量的作用域。
```
public static boolean isSquareImplA(Shape shape) {
    if (shape instanceof Rectangle rect) {
        // rect is in scope
        return rect.length() == rect.width();
    }
    // rect is not in scope here
    return shape instanceof Square;
}
```
第二段代码，我们看看有点意外的使用。我们可以在确认类型不匹配的条件语句之后使用匹配变量。这个条件语句之内，不是匹配变量的作用域。
```
public static boolean isSquareImplB(Shape shape) {
    if (!(shape instanceof Rectangle rect)) {
        // rect is not in scope here
        return shape instanceof Square;
    }
    // rect is in scope
    return rect.length() == rect.width();
}
```
第三段代码，我们看看紧凑的方式。这一段代码的逻辑，和第一段代码一样，我们只是换成了一种更紧凑的表示方法。
在这一段代码里，我们使用逻辑与运算符表示第一段里的条件语句：类型匹配并且匹配变量满足某一个条件。这样的表示是符合匹配变量的作用域规则的。逻辑与运算符从左到右计算，只有第一个运算成立，也就是类型匹配，才能进行下一个运算。所以，我们可以在逻辑与运算的第二部分，使用匹配变量。
```
public static boolean isSquareImplC(Shape shape) {
    return shape instanceof Square ||  // rect is not in scope here
           (shape instanceof Rectangle rect &&
            rect.length() == rect.width());   // rect is in scope here
```

```
}
```

第四段代码，我们看看逻辑或运算。它类似于第三段代码，只是我们把逻辑与运算符替换成了逻辑或运算符。这时候的逻辑，就变成了“类型匹配或者匹配变量满足某一个条件”。逻辑或运算符也是从左到右计算。

不过和逻辑与运算符不同的是，一般来说，只有第一个运算不成立，也就是说类型不匹配时，才能进行下一步的运算。下一步的运算，匹配变量并没有被赋值，我们不能够在这一部分使用匹配变量。所以，这一段代码并不能通过编译器的审查。

```
public static boolean isSquareImplD(Shape shape) {
    return shape instanceof Square ||  // rect is not in scope here
           (shape instanceof Rectangle rect ||
            rect.length() == rect.width());   // rect is not in scope here
}
```

第五段代码，我们看看位与运算。

这段代码和第三段代码类似，只是我们把逻辑与运算符（&&）替换成了位与运算符（&）。和第三段代码相比，这一段代码的逻辑其实并没有变化。只不过，位与运算符两侧的表达式都要参与计算。也就是说，不管位与运算符左侧的运算是否成立，位与运算符右侧的运算都要计算出来。换句话说，无论左侧的类型匹配不匹配，右侧的匹配变量都要使用。这就违反了匹配变量的作用域原则，编译器不能够确定匹配变量是否被赋值。所以，这一段代码，也不能通过编译器的审查。

```
public static boolean isSquareImplE(Shape shape) {
    return shape instanceof Square |  // rect is not in scope here
           (shape instanceof Rectangle rect &
            rect.length() == rect.width());   // rect is in scope here
}
```

第六段代码，我们把匹配变量的作用域的影响延展一下，看看它对影子变量（Shadowed Variable）的影响。

既然我们讨论变量的作用域，我们就不能不看看影子变量。假设我们定义了一个静态变量，它和匹配变量使用相同的名字。在匹配变量的作用域内，除非特殊处理，这个静态变量就被遮掩住了。这时候，这个变量名字代表的就是匹配变量；而不是静态变量。类似地，在匹配变量的作用域之外，这个变量名字代表的就是这个静态变量。

在这段代码里，我们使用类似于第一段代码的代码组织方式，来表述类型匹配部分的逻辑。另外，我在代码里标注了变量的作用域。你可以看看，这两个变量的作用域，和你想象的作用域是不是一样的？

```
public final class Shadow {
    private static final Rectangle rect = null;
    public static boolean isSquare(Shape shape) {
        if (shape instanceof Rectangle rect) {
            // Field rect is shadowed, local rect is in scope
            System.out.println("This should be the local rect: " + rect);
            return rect.length() == rect.width();
        }
        // Field rect is in scope, local rect is not in scope here
        System.out.println("This should be the field rect: " + rect);
        return shape instanceof Shape.Square;
    }
}
```

第七段代码，我们还是来看一看影子变量。只不过，这一次，我们使用类似于第二段代码的代码组织方式，来表述类型匹配部分的逻辑。我在代码里标出的这两个变量的作用域，和你想象的作用域是一样的吗？

```
public final class Shadow {
    private static final Rectangle rect = null;
```

```
    public static boolean isSquare(Shape shape) {
        if (!(shape instanceof Rectangle rect)) {
            // Field rect is in scope, local rect is not in scope here
            System.out.println("This should be the field rect: " + rect);
            return shape instanceof Shape.Square;
        }
        // Field rect is shadowed, local rect is in scope
        System.out.println("This should be the local rect: " + rect);
        return rect.length() == rect.width();
    }
}
```

如果回头看看这七段代码，你会倾向于哪一种编码的风格？我们把这些代码放在一起，分析一下它们的特点。

第四段和第五段代码，不能通过编译器的审查，所以我们不能使用这两种编码方式。

第二段和第七段代码，匹配变量的作用域，远离了类型匹配语句。这种距离上的疏远，无论在视觉上还是心理上，都不是很舒适的选择。不舒适，就给错误留下了空间，不容易编码，也不容易排错。这种代码逻辑和语法上都没有问题，但是不太容易阅读。

第一段和第六段代码，匹配变量的作用域，紧跟着类型匹配语句。这是我们感觉舒适的代码布局，也是最安全的代码布局，不容易出错，也容易阅读。

第三段代码，它的匹配变量的作用域也是紧跟着类型匹配语句。只不过，这种代码的编排方式不太容易阅读，阅读者需要认真拆解每一个条件，才能确认逻辑是正确的。相对于第一段和第六段代码，第三段代码的组织方式，是一个次优的选择。

如果你学习过《代码精进之路》专栏，我想你会理解代码组织方式的重要性，并且能够有意识地选择简单、安全的组织方式。对于类型匹配来说，第一段和第六段代码的组织方式，是我们喜欢的方式。

实例匹配的红利

在快要结束本文写作的时候，我还是忍不住测试了一下实例匹配的性能。在我自己的笔记本电脑上，和使用类型转换运算符的代码相比，使用实例匹配代码的吞吐量提高了将近 20%。这是一个巨大的性能提升。我知道使用实例匹配会提高性能，但是没想到有这么大的提升。除了主要目标之外，这也算是使用实例匹配的一个红利吧。

```
Benchmark                  Mode  Cnt          Score          Error  Units
PatternBench.useCast      thrpt   15  263559326.599 ± 78815341.366  ops/s
PatternBench.usePattern   thrpt   15  313458467.044 ±  2666412.767  ops/s
```

总结

好，这节课的内容到这里就要结束了，我来做个小结。从前面的讨论中，我们了解了 Java 的模式匹配和 Java 的类型匹配，讨论了 Java 类型匹配要解决的问题、表现的形式，以及匹配变量的作用域。顺便，我们还讨论了我们喜欢的类型匹配代码的组织方式。

在我们日常的编码实践中，为了简化代码逻辑，减少代码错误，提高生产效率，我们应该优先考虑使用类型匹配，而不是传统的强制类型转换运算符。

如果你想要丰富你的代码评审清单，有了 Java 类型匹配后，你可以加入下面这一条：

如果需要类型转换，是不是可以使用类型匹配？

另外，我在今天的讨论中拎出了几个技术要点，这些都可能在你们面试中出现哦。通过这一次学习，你应该能够：

- 知道 Java 支持类型匹配，并且能够使用类型匹配，替换掉传统的强制类型转换运算。
    - 面试问题：你知道类型匹配吗？会不会使用它？
- 了解类型匹配的原理和它要解决的问题，知道匹配变量的作用域。
    - 面试问题：使用类型匹配有哪些好处？匹配变量什么时候可以使用？
- 了解类型匹配的代码组织方式，能够有意识地使用简单、安全的代码组织方式。

- 面试问题：你写的这段代码（如果使用了类型匹配），还有更好的表达方式吗？

如果你能够有意识地使用 Java 的类型匹配，并且有能力选择简单、安全的代码组织方式，你应该能够大幅度提高编码的效率和质量，提高代码的性能。毫无疑问，在面试的时候，这也是一个能够让你与众不同的知识点。

思考题

在“匹配变量的作用域”这一小节里，我们列举了 7 种实例匹配的代码组织方式。除了第四段代码和第五段代码，其他的五种代码都可以通过编译。为了加深你的印象，我们要动动手，验证一下每一种代码组织方式下，匹配变量的作用域。

我在下面的例子中写了一个代码小样，使用打印语句输出来验证结果。你可以试着修改成你喜欢的样子，添加更多的代码组织方式。

```
/*
 * Copyright (c) 2021, Xuelei Fan. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 */
package co.ivi.jus.instance.review.xuelei;
public sealed interface Shape
        permits Shape.Circle, Shape.Rectangle, Shape.Square {
    Shape.Rectangle rect = null;    // field variable
    record Circle(double radius) implements Shape {
        // blank
    }
    record Square(double side) implements Shape {
        // blank
    }
    record Rectangle(double length, double width) implements Shape {
        // blank
    }
    static void main(String[] args) {
        Shape shape = new Shape.Rectangle(10, 10);
        System.out.println("It should be ture that " + shape +
                " is a square: " + isSquare(shape));
        System.out.println();
        shape = new Shape.Circle(10);
        System.out.println("It cannot be ture that " + shape +
                " is a square: " + (!isSquare(shape)));
    }
    static boolean isSquare(Shape shape) {
        if (shape instanceof Rectangle rect) {
            // Field rect is shadowed, local rect is in scope
            System.out.println(
                    "This should be the local rect: " +
                     rect.equals(shape));
            return (rect.length == rect.width);
        }
        // Field rect is in scope, local rect is not in scope here
        System.out.println(
                "This should be the field rect: " + (rect == null));
        return (shape instanceof Square);
    }
```

```
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及验证的代码和结果。

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在实例匹配专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 instance/review/xuelei 的目录下面。

注：本文使用的基准性能测试代码，你也可以从 GitHub 上下载，试试你的机器是不是也有相似的性能表现。

06 | switch 表达式：怎么简化多情景操作？

你好，我是范学雷。今天，我们聊一聊 Switch 表达式。

switch 表达式这个特性，首先在 JDK 12 中以预览版的形式发布。在 JDK 13 中，改进的 switch 表达式再次以预览版的形式发布。最后，switch 表达式在 JDK 14 正式发布。

不论你学习什么样的编程语言，合理地分析、判断、处理不同的情况都是必备的基本功。比如我们使用的 if-else 语句，还有 switch 语句，都是用来处理种种不同的情况的。我们都知道 switch 语句，那么 switch 表达式又是什么呢？switch 语句和 switch 表达式又有什么不同呢？

如果你了解了 Java 的语句和表达式这两个基本概念，你的困扰也许会少一点。Java 规范里，表达式完成对数据的操作。一个表达式的结果可以是一个数值（i*4）；或者是一个变量（i=4）；或者什么都不是（void 类型）。

Java 语句是 Java 最基本的可执行单位，它本身不是一个数值，也不是一个变量。Java 语句的标志性符号是分号（代码）和双引号（代码块），比如 if-else 语句，赋值语句等。这样再来看，就很简单了：switch 表达式就是一个表达式，而 switch 语句就是一个语句。

switch 表达式是什么样子的？为什么需要 switch 表达式？我们还是通过案例和代码，一点一点地来学习 switch 表达式吧。

阅读案例

在讲解或者学习 switch 语句时，每年的十二个月或者每周的七天，是我们经常使用的演示数据。在这个案例里，我们也使用这样的数据，来看看传统的 switch 语句有哪些需要改进的地方。

下面，我们要讨论的，也是一个传统的问题：该怎么用代码计算一个月有多少天？生活中，我们熟悉这样的顺口溜，“一三五七八十腊，三十一天永不差，四六九冬三十整，平年二月二十八，闰年二月把一加”。

下面的这段代码，就是按照这个顺口溜的逻辑来计算了一下，今天所在的这个月，一共有多少天。

```
package co.ivi.jus.swexpr.former;
import java.util.Calendar;
class DaysInMonth {
    public static void main(String[] args) {
        Calendar today = Calendar.getInstance();
        int month = today.get(Calendar.MONTH);
        int year = today.get(Calendar.YEAR);
        int daysInMonth;
        switch (month) {
            case Calendar.JANUARY:
            case Calendar.MARCH:
            case Calendar.MAY:
            case Calendar.JULY:
            case Calendar.AUGUST:
            case Calendar.OCTOBER:
            case Calendar.DECEMBER:
                daysInMonth = 31;
                break;
            case Calendar.APRIL:
            case Calendar.JUNE:
            case Calendar.SEPTEMBER:
            case Calendar.NOVEMBER:
                daysInMonth = 30;
                break;
```

```
            case Calendar.FEBRUARY:
                if (((year % 4 == 0) && !(year % 100 == 0))
                        || (year % 400 == 0)) {
                    daysInMonth = 29;
                } else {
                    daysInMonth = 28;
                }
                break;
            default:
                throw new RuntimeException(
                    "Calendar in JDK does not work");
        }
        System.out.println(
            "There are " + daysInMonth + " days in this month.");
    }
}
```

这段代码里，我们使用了 switch 语句。代码本身并没有什么错误，但是，至少有两个容易犯错误的地方。

第一个容易犯错的地方，就是在 break 关键字的使用上。上面的代码里，如果多使用一个 break 关键字，代码的逻辑就会发生变化；同样的，少使用一个 break 关键字也会出现问题。

```
int daysInMonth;
switch (month) {
    case Calendar.JANUARY:
    case Calendar.MARCH:
    case Calendar.MAY:
        break;    // WRONG BREAK!!!
    case Calendar.JULY:
    case Calendar.AUGUST:
    case Calendar.OCTOBER:
    case Calendar.DECEMBER:
        daysInMonth = 31;
        break;
    // snipped
}
int daysInMonth;
switch (month) {
    // snipped
    case Calendar.APRIL:
    case Calendar.JUNE:
    case Calendar.SEPTEMBER:
    case Calendar.NOVEMBER:
        daysInMonth = 30;
                        // WRONG, NO BREAK!!!
    case Calendar.FEBRUARY:
        if (((year % 4 == 0) && !(year % 100 == 0))
                || (year % 400 == 0)) {
            daysInMonth = 29;
        } else {
            daysInMonth = 28;
        }
        break;
```

```
    // snipped
}
```

break 语句的遗漏或者冗余，这样的错误如此得常见，甚至于被单列成了一个常见软件安全漏洞。凡是使用 switch 语句的代码，都有可能成为黑客们重点关注的对象。由于逻辑的错误和黑客的特殊关照，我们在编写代码的时候，需要十二分的小心；阅读代码的时候，也需要反复地查验 break 语句的前后语境。毫无疑问，这增加了代码维护的成本，降低了生产效率。

为什么 switch 语句里需要使用 break 呢？最主要的原因，就是希望能够在不同的情况下，共享部分或者全部的代码片段。比如上面的例子中，四月、六月、九月、十一月这四种情景，可以共享每个月都是 30 天这样的代码片段。这个代码片段只需要写在十一月情景的后面，前面的四月、六月和九月这三个情景都会顺次执行下面的操作（fall-through），直到遇到下一个 break 语句或者 switch 语句终结。

现在我们都知道了，这样是一个弊大于利的设计。但很遗憾，Java 初始的设计就是采用了这样的设计思想。如果要新设计一门现代的语言，我们需要更多地使用 switch 语句，但是就不要再使用 break 语句了。不过，不同的情景共享代码片段，仍然是一个真实的需求。在废弃掉 break 语句之前，我们要找到在不同的情景间共享代码片段的新规则。

第二个容易犯错的地方，是反复出现的赋值语句。在上面的代码中，**daysInMonth** 这个本地变量的变量声明和实际赋值是分开的。赋值语句需要反复出现，以适应不同的情景。如果在 switch 语句里，**daysInMonth** 变量没有被赋值，编译器也不会报错，缺省的或者初始的变量值就会被使用。

```
int daysInMonth = 0;
switch (month) {
    // snipped
    case Calendar.APRIL:
    case Calendar.JUNE:
    case Calendar.SEPTEMBER:
    case Calendar.NOVEMBER:
        break;   // WRONG, INITIAL daysInMonth value IS USED!!!
    case Calendar.FEBRUARY:
    // snipped
}
```

在上面的例子里，初始的变量值不是一个合适的数据；当然，在另外一个例子里，缺省的或者初始的变量值也可能就是一个合适的数据了。为了判断这个本地变量有没有合适的值，我们需要通览整个 switch 语句块，确保赋值没有遗漏，也没有多余。这增加了编码出错的几率，也增加了阅读代码的成本。

那么，能不能让多情景处理的代码块拥有一个数值呢？或者换个说法，多情景处理的代码块能不能变成一个表达式？这个想法，就催生了 Java 语言的新特性：“switch 表达式”。

switch 表达式

switch 表达式是什么样子的呢？下面的这段代码，使用的就是 switch 表达式，它改进了上面阅读案例里的代码。你可以带着上面遇到的问题，来阅读这段代码。这些问题包括：

- switch 表达式是怎么表示一个数值，从而可以给变量赋值的？
- 在不同的情景间，switch 表达式是怎么共享代码片段的？
- 使用 switch 表达式的代码，有没有变得更简单、更皮实、更容易理解？

```
package co.ivi.jus.swexpr.modern;
import java.util.Calendar;
class DaysInMonth {
    public static void main(String[] args) {
        Calendar today = Calendar.getInstance();
        int month = today.get(Calendar.MONTH);
        int year = today.get(Calendar.YEAR);
```

```
        int daysInMonth = switch (month) {
            case Calendar.JANUARY,
                 Calendar.MARCH,
                 Calendar.MAY,
                 Calendar.JULY,
                 Calendar.AUGUST,
                 Calendar.OCTOBER,
                 Calendar.DECEMBER -> 31;
            case Calendar.APRIL,
                 Calendar.JUNE,
                 Calendar.SEPTEMBER,
                 Calendar.NOVEMBER -> 30;
            case Calendar.FEBRUARY -> {
                if (((year % 4 == 0) && !(year % 100 == 0))
                        || (year % 400 == 0)) {
                    yield 29;
                } else {
                    yield 28;
                }
            }
            default -> throw new RuntimeException(
                    "Calendar in JDK does not work");
        };
        System.out.println(
                "There are " + daysInMonth + " days in this month.");
    }
}
```

我们最先看到的变化，就是 switch 代码块出现在了赋值运算符的右侧。这也就意味着，这个 switch 代码块表示的是一个数值，或者是一个变量。换句话说，这个 switch 代码块是一个表达式。

```
int daysInMonth = switch (month) {
    // snipped
}
```

我们看到的第二个变化，是多情景的合并。也就是说，一个 case 语句，可以处理多个情景。这些情景，使用逗号分隔开来，共享一个代码块。而传统的 switch 代码，一个 case 语句只能处理一种情景。

```
case Calendar.JANUARY,
     Calendar.MARCH,
     // snipped
```

多情景的合并的设计，满足了共享代码片段的需求。而且，由于只使用一个 case 语句，也就不再需要使用 break 语句来满足这个需求了。所以，break 语句从 switch 表达式里消失了。

不同之处在于，传统的 switch 代码，不同的 case 语句之间可以共享部分的代码片段；而 switch 表达式里，需要共享全部的代码片段。这看似是一个损失，但其实，共享部分代码片段的能力给代码的编写者带来的困惑远远多于它带来的好处。如果需要共享部分的代码片段，我们总是可以找到替换的办法，比如把需要共享的代码封装成更小的方法。所以，我们没有必要担心 switch 表达式不支持共享部分代码片段。

下一个变化，是一个新的情景操作符，“->”，它是一个箭头标识符。这个符号使用在 case 语句里，一般化的形式是“case L ->”。这里的 L，就是要匹配的一个或者多个情景。如果目标变量和情景匹配，那么就执行操作符右边的表达式或者代码块。如果要匹配的情景有两个或者两个以上，就要使用逗号“,”用分隔符把它们分割开来。

```
case Calendar.JANUARY,
     // snipped
     Calendar.DECEMBER -> 31;
```

传统的 switch 代码，这个一般化的形式是“case L ：”，也就是使用冒号标识符。为什么不延续使用传统的情景操作符呢？这主要是出于简化代码的考虑。我们依然可以在 switch 表达式里使用冒号标识符，使用冒号标识符的一个 case 语句只能匹配一个情景，这种情况我们稍后再讨论。

下一个我们看到的变化，是箭头标识符右侧的数值。这个数值，代表的就是该匹配情景下，switch 表达式的数值。需要注意的是，**箭头标识符右侧可以是表达式、代码块或者异常抛出语句，而不能是其他的形式。**如果只需要一个语句，这个语句也要以代码块的形式呈现出来。

```
case Calendar.JANUARY,
     // snipped
     Calendar.DECEMBER -> {  // CORRECT, enclosed with braces.
    yield 31;
}
```

没有以代码块形式呈现的代码，编译的时候，就会报错。这是一个很棒的约束。代码块的形式，增强了视觉效果，减少了编码的失误。在《代码精进之路》这个专栏里，我们反复强调过这种形式的好处。

```
case Calendar.JANUARY,
     // snipped
     Calendar.DECEMBER ->   // WRONG, not a block.
    yield 31;
```

另外，箭头标识符右侧需要一个表达 switch 表达式的数值，这是一个很强的约束。如果一个语句破坏了这个需要，它就不能出现在 switch 表达式里。比如，下面的代码里的 return 语句，意图退出该方法，而没有表达这个 switch 表达式的数值。这段代码就不能通过编译器的审查。

```
int daysInMonth = switch (month) {
    // snipped
    case Calendar.APRIL,
         // snipped
         Calendar.NOVEMBER -> {
        // yield 30;
        return; // WRONG, return outside of enclosing switch expression.
    }
    // snipped
}
```

最后一个我们能够看到的变化，是出现了一个新的关键字“yield”。大多数情况下，switch 表达式箭头标识符的右侧是一个数值或者是一个表达式。如果需要一个或者多个语句，我们就要使用代码块的形式。这时候，我们就需要引入一个新的 yield 语句来产生一个值，这个值就成为这个封闭代码块代表的数值。

为了便于理解，我们可以把 yield 语句产生的值看成是 switch 表达式的返回值。所以，yield 只能用在 switch 表达式里，而不能用在 switch 语句里。

```
case Calendar.FEBRUARY -> {
    if (((year % 4 == 0) && !(year % 100 == 0))
            || (year % 400 == 0)) {
        yield 29;
    } else {
        yield 28;
    }
}
```

其实，这里还有一个我们从上述的代码里看不到的变化。在 switch 表达式里，所有的情景都要列举出来，不能多、也不能少（这也就是我们常说的穷举）。
比如说，在上面的例子里，如果没有最后的 default 情景分支，编译器就会报错。这是一个影响深远的改进，它会使得 switch 表达式的代码更加健壮，大幅度降低维护成本，如果未来需要增加一个情景分支的话，就更是如此了。

```
int daysInMonth = switch (month) {
    case Calendar.JANUARY,
         // snipped
         Calendar.DECEMBER -> 31;
    case Calendar.APRIL,
         // snipped
         Calendar.NOVEMBER -> 30;
    case Calendar.FEBRUARY -> {
             // snipped
    }
    // WRONG to comment out the default branch, 'switch' expression
    // MUST cover all possible input values.
    //
    // default -> throw new RuntimeException(
    //         "Calendar in JDK does not work");
};
```

改进的 switch 语句

通过上面的解读，我们知道了 switch 表达式里有很多积极的变化。那这些变化有没有影响 switch 语句呢？比如说，我们能够在 switch 语句里使用箭头标识符吗？我们前面说过，yield 语句是来产生一个 switch 表达式代表的数值的，因此 yield 语句只能用在 switch 表达式里，不能用在 switch 语句。
其他的变化呢？我们还是先来看下面一段代码。

```
private static int daysInMonth(int year, int month) {
    int daysInMonth = 0;
    switch (month) {
        case Calendar.JANUARY,
             Calendar.MARCH,
             Calendar.MAY,
             Calendar.JULY,
             Calendar.AUGUST,
             Calendar.OCTOBER,
             Calendar.DECEMBER ->
            daysInMonth = 31;
        case Calendar.APRIL,
             Calendar.JUNE,
             Calendar.SEPTEMBER,
             Calendar.NOVEMBER ->
            daysInMonth = 30;
        case Calendar.FEBRUARY -> {
            if (((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0)) {
                daysInMonth = 29;
                break;
            }
            daysInMonth = 28;
```

```
        }
        // default -> throw new RuntimeException(
        //         "Calendar in JDK does not work");
    }
    return daysInMonth;
}
```

在这段代码里，我们看到了箭头标识符，看到了 break 语句，看到了注释掉的 default 语句。这是一段合法的、能够工作的代码。换个说法，switch 语句可以使用箭头标识符，也可以使用 break 语句，也不需要列出所有的情景。表面上看起来，switch 语句的改进不是那么显而易见。其实，switch 语句的改进主要体现在 break 语句的使用上。

我们应该也看到了，break 语句没有出现在下一个 case 语句之前。这也就意味着，使用箭头标识符的 switch 语句不再需要 break 语句来实现情景间的代码共享了。虽然我们还可以这样使用 break 语句，但是已经不再必要了。

```
switch (month) {
    // snipped
    case Calendar.APRIL,
              // snipped
         Calendar.NOVEMBER -> {
            daysInMonth = 30;
            break;  // UNNECESSARY, could be removed safely.
        }
    // snipped
}
```

有没有 break 语句，使用箭头标识符的 switch 语句都不会顺次执行下面的操作（fall-through）。这样，我们前面谈到的 break 语句带来的烦恼也就消失不见了。

不过，使用箭头标识符的 switch 语句并没有禁止 break 语句，而是恢复了它本来的意义：从代码片段里抽身，就像它在循环语句里扮演的角色一样。

```
switch (month) {
   // snipped
   case Calendar.FEBRUARY -> {
        if (((year % 4 == 0) && !(year % 100 == 0))
                || (year % 400 == 0)) {
            daysInMonth = 29;
            break;     // BREAK the switch statement
        }

        daysInMonth = 28;
    }
   // snipped
}
```

怪味的 switch 表达式

我们前面说过，switch 表达式也可以使用冒号标识符。使用冒号标识符的一个 case 语句只能匹配一个情景，而且支持 fall-through。和箭头标识符的 switch 表达式一样，使用冒号标识符 switch 表达式也不支持 break 语句，取而代之的是 yield 语句。

这是一个充满了怪味道的编码形式，我并不推荐使用这种形式，但我可以带你略作了解。下面的这段代码，就是我们试着把箭头标识符替换成冒号标识符的一个例子。你可以比较一下使用冒号标识符和箭头标识符的两段代码，想一想两种不同形式的优劣。毫无疑问，使用箭头标识符的代码更加简洁。

```
package co.ivi.jus.swexpr.legacy;
import java.util.Calendar;
```

```
class DaysInMonth {
    public static void main(String[] args) {
        Calendar today = Calendar.getInstance();
        int month = today.get(Calendar.MONTH);
        int year = today.get(Calendar.YEAR);
    int daysInMonth = switch (month) {
        case Calendar.JANUARY:
        case Calendar.MARCH:
        case Calendar.MAY:
        case Calendar.JULY:
        case Calendar.AUGUST:
        case Calendar.OCTOBER:
        case Calendar.DECEMBER:
            yield 31;
        case Calendar.APRIL:
        case Calendar.JUNE:
        case Calendar.SEPTEMBER:
        case Calendar.NOVEMBER:
            yield 30;
        case Calendar.FEBRUARY:
            if (((year % 4 == 0) && !(year % 100 == 0))
                    || (year % 400 == 0)) {
                yield 29;
            } else {
                yield 28;
            }
        default:
            throw new RuntimeException(
                    "Calendar in JDK does not work");
        };
        System.out.println(
            "There are " + daysInMonth + " days in this month.");
    }
}
```

有了使用箭头标识符的 switch 语句和 switch 表达式之后，我们不再推荐使用冒号标识符的 switch 语句和 switch 表达式。学习并使用箭头标识符的 switch 语句和 switch 表达式，会使代码更简洁、更健壮。

总结

好，到这里，我来做个小结。从前面的讨论中，我们重点了解了 switch 表达式和改进的 switch 语句。我们还讨论了 switch 表达式带来的新概念和新的关键字，了解了这些基本概念以及它们的适用范围。

新的 switch 形式、语句和表达式，不同的使用范围，这些概念交织在一起，让 switch 的学习和使用都变成了一件有点挑战性的事情。箭头标识符的引入，简化了代码，提高了编码效率。可是，学习这么多种 switch 的表现形式，也增加了我们的学习负担。为了帮助你快速掌握这些形式，我把不同的 switch 表达形式，以及它们支持的特征，放在了下面这张表格里。

	使用箭头标识符的switch表达式	使用箭头标识符的switch语句	使用冒号标识符的switch表达式	使用冒号标识符的switch语句
break语句	✘	✔	✘	✔
yield语句	✔	✘	✔	✘
情景穷举	✔	✘	✔	✘
fall-through	✘	✘	✔	✔
多情景匹配	✔	✔	✘	✘

或者，你也可以记住下面的总结：

- break 语句只能出现在 switch 语句里，不能出现在 switch 表达式里；
- yield 语句只能出现在 switch 表达式里，不能出现在 switch 语句里；
- switch 表达式需要穷举出所有的情景，而 switch 语句不需要情景穷举；
- 使用冒号标识符的 swtich 形式，支持情景间的 fall-through；而使用箭头标识符的 swtich 形式不支持 fall-through；
- 使用箭头标识符的 swtich 形式，一个 case 语句支持多个情景；而使用冒号标识符的 swtich 形式不支持多情景的 case 语句。

使用箭头标识符的 swtich 形式，废止了容易出问题的 fall-through 这个特征。因此，我们推荐使用箭头标识符的 swtich 形式，逐步废止使用冒号标识符的 swtich 形式。在 switch 表达式和 switch 语句之间，我们应该优先使用 switch 表达式。这些选择，都可以帮助我们简化代码逻辑，减少代码错误，提高生产效率。

如果你要丰富你的代码评审清单，学习完这一节内容后，你可以加入下面这一条：

使用冒号标识符的 swtich 形式，是不是可以更改为使用箭头标识符？

使用 switch 语句赋值的操作，是不是可以更改为使用 switch 表达式？

另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你们面试中出现哦。通过这一次学习，你应该能够：

- 知道 switch 表达式，并且能够使用 switch 表达式；
    - 面试问题：你知道 switch 表达式吗？该怎么处理 switch 表达式里的语句？
- 了解 switch 表达式要解决的问题，并且知道解决掉这些问题的办法；
    - 面试问题：使用 switch 表达式有哪些好处？
- 了解不同的 switch 的表现形式，能够看得懂不同的表现形式，并且给出改进意见。
    - 面试问题：你更喜欢使用箭头标识符还是冒号标识符？

如果你能够有意识地使用箭头标识符的 switch 表达式，应该可以大幅度提高编码的效率和质量；如果你能够了解不同的 switch 表现形式，并且对每种形式都有自己的见解，你就能帮助你的同事提高编码的效率和质量。毫无疑问，在面试的时候，有意识地在代码里使用 switch 表达式，是一个能够展现你的学习能力、理解能力和对新知识的接受能力的一个好机会。

思考题

在前面的讨论里，我们说过情景穷举是一个影响深远的改进方向，它会使得 switch 表达式的代码更加健壮，大幅度降低维护成本，特别是在未来需要增加一个情景分支的情形下。但是，限于篇幅，我们并没有详细地展开讨论其中的细节。现在，我们把这个讨论当作一个稍微有点挑战的思考题。

假设有一天，地球和太阳的关系发生了变化，这种变化还没有大到毁灭人类的程度，但是也足以改变年月的关系了。于是，天文学家重新修订了日历，增加了一个新的月份，第十三个月。为了对应这种变化，JDK 的设计者们也给 Calendar 类增加了第十三个月：Calendar.AFTER DEC。那么，我们的问题就来了。

第一个问题是，我们现在的代码能够检测到这个变化吗？如果不能，是不是只有系统崩溃的时候，我们才能够意识到问题的存在？
第二个问题是，有没有更健壮的设计，能够帮助我们在系统崩溃之前就能够检测到这个意想不到的变化？从而给我们留出时间更改我们的代码和系统？
稍微提示一个，解决这个问题的其中一个思路，就是要使用有穷举能力的表达式，然后设计出可以表达穷举情景的新形式，而不是使用泛泛的整数来表达十二个月。
我在下面的例子中写了一个代码小样。这个代码小样，实现的还是一年只有十二个月的逻辑。现在我们假设，一年还是十二个月，但是我们想让这段代码健壮到能够检测到未来一年变成十一个月或者十三个月的情景。
在这个代码小样里，我也试着加入了一些提示。当然，你也可以试着找找其他的解决方案。请试着将这段代码修改成你喜欢的样子，让我们一起看看怎么解决掉这个问题。

```
package co.ivi.jus.swexpr.review.xuelei;
import java.util.Calendar;
class DaysInMonth {
    public static void main(String[] args) {
        Calendar today = Calendar.getInstance();
        int month = today.get(Calendar.MONTH);
        int year = today.get(Calendar.YEAR);
    // Hints: could we replace the integer month
    // with an exhaustive enumeration?
    int daysInMonth = switch (month) {
            case Calendar.JANUARY,
                 Calendar.MARCH,
                 Calendar.MAY,
                 Calendar.JULY,
                 Calendar.AUGUST,
                 Calendar.OCTOBER,
                 Calendar.DECEMBER -> 31;
            case Calendar.APRIL,
                 Calendar.JUNE,
                 Calendar.SEPTEMBER,
                 Calendar.NOVEMBER -> 30;
            case Calendar.FEBRUARY -> {
                if (((year % 4 == 0) && !(year % 100 == 0))
                        || (year % 400 == 0)) {
                    yield 29;
                } else {
                    yield 28;
                }
            }
            // Hints: Are we able to replace the default case by
            // enumerating all cases with case clause above?
            default -> throw new RuntimeException(
                    "Calendar in JDK does not work");
        };
        System.out.println(
            "There are " + daysInMonth + " days in this month.");
    }
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你对这个思考题的想法。
注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template

代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在实例匹配专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 swexpr/review/xuelei 的目录下面。

07 | switch 匹配：能不能适配不同的类型？

你好，我是范学雷。今天，我们聊一聊 switch 的模式匹配。

switch 的模式匹配这个特性，在 JDK 17 中以预览版的形式发布。按照通常的进度，这个特性可能还需要两到三个版本，才能最终定稿。

这个特性很简单，但是非常重要，可以帮助我们解决不少棘手而且重要的问题。我们不妨在定稿之前，就试着看看它。

前面，我们讨论了类型匹配和 switch 表达式。那 switch 的模式匹配又是什么样子的呢？为什么说 switch 的模式匹配非常重要？我们还是通过案例和代码，一步一步地了解 switch 的模式匹配吧。

阅读案例

在面向对象的编程语言中，研究表示形状的类，是一个常用的教学案例。今天的阅读案例，会涉及到表示形状的接口和类的定义，以后，我还会给出一个使用案例。通过这个案例，我们可以看到面向对象设计的一个代码在维护和发展时的难题。

假设我们定义了一个表示形状的封闭类，它的名字是 Shape；我们也定义了两个许可类：Circle 和 Square，它们分别表示圆形和正方形。下面的代码，就是一个可供你参考的实现方式。

```
public sealed interface Shape
        permits Shape.Circle, Shape.Square {
    record Circle(double radius) implements Shape {
        // blank
    }
    record Square(double side) implements Shape {
        // blank
    }
}
```

接着，我们就要使用形状这个类来处理具体的问题了。你可以先试着回答一下，给定了一个形状的对象，我们该怎么判断这个对象是不是一个正方形呢？

这是一个简单的问题。只要判断这个对象是不是一个正方形类（Square）的实例就可以了。就像下面的代码这样。

```
public static boolean isSquare(Shape shape) {
    return (shape instanceof Shape.Square);
}
```

无论是形状类的设计，还是我们处理问题的方式，看起来都没有什么问题。不过，如果我们朝前看，想一想未来的形状类的变化，问题可能就浮现出来了。

假设上面表示形状的封闭类和许可类是版本 1.0，它们被封装在一个基础 API 类库里。而判断一个表示形状的对象是不是正方形的代码，也就是 IsSquare 的实现代码，我们把它封装到另外一个 API 类库里。为了方便后面的讨论，我们把这两个类库称为基础类库和扩展类库（这两个名字并不一定契合实际）。

现在，我们升级表示形状的封闭类和许可类，新加入一个许可类，用来表示长方形。这样，我们就有了下面这样的代码。

```
public sealed interface Shape
        permits Shape.Circle, Shape.Rectangle, Shape.Square {
    /**
     * @since 1.0
     */
    record Circle(double radius) implements Shape {
        // blank
    }
    /**
     * @since 1.0
```

```
     */
    record Square(double side) implements Shape {
        // blank
    }
    /**
     * @since 2.0
     */
    record Rectangle(double length, double width) implements Shape {
        // blank
    }
}
```

在面向对象的世界里，增加一个新的子类是一种很常见的升级方法。而且，不论是出于理论还是实践，我们都没有充分的理论、也没有应有的能力杜绝掉这样的升级。所以，新加入一个表示长方形的许可类，似乎并没有什么不妥。类似这样的更改，我们也不会期待出现明显的可兼容性问题。

好了，现在我们有了 2.0 版本的基础类库。

然后，我们再来看看扩展类库。我们知道，正方形是一个特殊的长方形。如果一个长方形的长和宽是相等的，那么它也是一个正方形。所以，如果基础类库支持了长方形，我们就需要考虑正方形这个特例。不然的话，这个扩展类库的实现，就不能处理这个特例。

扩展类库的更改也很简单，只要加入处理特例的逻辑就可以了。这样，我们就有了下面这样的升级之后的代码。

```
public static boolean isSquare(Shape shape) {
    if (shape instanceof Shape.Rectangle rect) {
        return (rect.length() == rect.width());
    }
    return (shape instanceof Shape.Square);
}
```

然而，意识到扩展类库需要更改，并不是一件容易的事情。甚至，通常情况下，我们可以说它是一件非常艰苦和艰难的事情。

对于需要更改扩展类库这件事，基础类库的作者，不会通知扩展类库的作者。这绝对不是基础类库的作者的懒惰或者不负责任。一般情况下，基础类库和扩展类库是独立的产品，由不同的团队或者社区维护。所以基础类库的作者往往不太可能意识到扩展类库的存在，更不可能去研究扩展类库的实现细节。所以，修改扩展类库这件事，一般来说，是扩展类库维护者的责任。

同样地，扩展类库维护者也不会注意到基础类库的修改，更不容易想到基础类库的修改会影响到扩展类库的行为。通常地，API 的使用者依赖 API 的兼容性。也就是说，API 可以升级，但是这个升级不能影响已有代码的使用。换句话说，1.0 版本的 API 上能跑得通的代码，2.0 版本的 API 上，同样的代码也必须能跑得通。所以，扩展类库维护者，也可以把问题踢给基础类库的维护者。

那么用户呢？有时候，他们找基础类库的维护者抱怨；有时候，他们找扩展类库的维护者抱怨。谁的市场影响大，对用户更友好，谁听到的抱怨就多一点。我们也没有理由责怪用户的抱怨，毕竟是他们的业务系统，也就是现实世界的系统，遇到了真正的问题，遭受了真实的损失。

这样的问题出现的根本原因，就是我们没有在用户抱怨之前发现这样的事实：扩展类库必须做出修改，以适应升级的基础类库。

而解决这样的问题，只依靠基础类库维护者和扩展类库维护者的勤奋，是不可能实现的。

那么，我们该怎么办呢？

其中的一个思路，就是尽可能早地发现这样的兼容性问题。而我给你的其中一条解决办法，就是使用具有类型匹配能力的 switch 表达式。

模式匹配的 switch

具有模式匹配能力的 switch，说的是将模式匹配扩展到 switch 语句和 switch 表达式，允许测试多个模式，而且每一个模式都可以有特定的操作。这样，就可以简洁、安全地表达复杂的面向数据的查询了。

下面的代码，展示了如何使用具有模式匹配能力的 switch，来判断一个对象是不是正方形：

```
public static boolean isSquare(Shape shape) {
    return switch (shape) {
        case null, Shape.Circle c -> false;
        case Shape.Square s -> true;
    };
}
```

这段简短的代码里面，有几个地方是我们在 JDK 17 之前没有遇到过的。

扩充的匹配类型

第一个地方，就是 switch 要匹配的表达式，或者说数据，而不是我们熟悉的类型。我们可能都知道，JDK 17 之前的 switch 关键字可以匹配的数据类型包括数字、枚举和字符串。本质上，这三种数据类型都是整形的原始类型。而在上面的例子中，这个要匹配的目标数据类型，是一个表示形状的对象，是一个引用类型。

具有模式匹配能力的 switch，提升了 switch 的数据类型匹配能力。switch 要匹配的数据，现在可以是整形的原始类型（数字、枚举、字符串），或者引用类型。

支持 null 情景模式

第二个地方，就是空引用“null”出现在了匹配情景中。以前，switch 要匹配的数据不能是空引用。否则，就会抛出“NullPointerException”这样的运行时异常。所以，规范的、公开接口的代码，通常都要检查匹配数据是不是一个空引用，然后才能接着使用 switch 语句或者 switch 表达式。就像下面的例子这样。

```
public static boolean isSquare(Shape shape) {
    if (shape == null) {
      return false;
    }
    return switch (shape) {
        case Shape.Circle c -> false;
        case Shape.Square s -> true;
    };
}
```

然而，对于非公开接口的内部实现代码，是不是需要这样的检查，并不是显而易见的。比如说，如果所有的调用，都不会传入空的引用，当然也就不需要检查空引用。可是，这样的假设过于脆弱。而且，对于代码的阅读者来说，去检查所有可能的内部调用，真的是一件很艰难的事情。

具有模式匹配能力的 switch，支持空引用的匹配。如果我们能够有意识地使用这个特性，可以提高我们的编码效率，降低代码错误。

可类型匹配的情景

第三个地方，就是类型匹配出现在了匹配情景中。也就是说，你既可以检查类型，还可以获得匹配变量。以前，switch 要匹配的数据是一个数值，比如说星期三或者十二月。对类型匹配来说，switch 要匹配的数据是一个引用；这时候，匹配情景要做的主要判断之一，是我们希望知道的这个引用的类型。

比如说吧，如果要匹配的数据是一个表示形状的类的引用，我们希望匹配情景要能够判断出来这个引用是一个圆形类的引用，还是一个正方形类的引用。如果情景能够匹配，我们还希望能够获得匹配变量。这一点，其实就像是我们在第 5 讲说到的类型匹配。现在，类型匹配出现在了 switch 语句和 switch 表达式的使用场景里。

```
case Shape.Circle c -> false;
```

这样，我们就在 switch 语句和 switch 表达式里获得了类型匹配的好处，如果需要使用转换后的数据类型，我们就不再需要编写强制类型转换的代码了。这就简化了代码逻辑，减少了

代码错误，提高了生产效率。

穷举的匹配情景

具有模式匹配能力的 switch，是怎么解决掉阅读案例里讨论的基础类库和扩展类库协同维护问题的呢？到现在，这个问题的答案还不是很明确，虽然答案已经有了。

这就是我们要讨论的第四个地方，使用 switch 表达式，穷举出所有的情景。在 isSquare 这个方法的实现里，我们使用了 switch 表达式，并且穷举出了所有可以匹配的形状类。我们知道，switch 表达式需要穷举出所有的情景。否则，编译器就会报错。使用 switch 表达式这个特点，就是我们解决阅读案例里提到的问题的基本思路。

现在，如果我们使用 2.0 版本的基础类库，也就是新加入了表示长方形的许可类的实现，那么 isSquare 这个方法的实现就不能通过编译了。因为，这个方法的实现遗漏了长方形这个许可类，没有满足 switch 表达式需要穷举所有情景的要求。

如果代码编译期就报错，扩展类库的维护者就能够第一时间知道这个方法的缺陷。这样，他们就不用等到用户遇到真实问题的时候，才意识到要去适应升级的基础类库了。

这种提前暴露问题的方式，大大地降低了代码维护的难度，让我们有更多的精力专注在更有价值的问题上。

意识到代码需要修改，其实是最难的一步。如果已经意识到这个问题，具体的修改就很简单了。如果对实现细节感兴趣，你可以参考下面这段我修改后的代码。

```
public static boolean isSquare(Shape shape) {
    return switch (shape) {
        case null, Shape.Circle c -> false;
        case Shape.Square s -> true;
        case Shape.Rectangle r -> r.length() == r.width();
    };
}
```

改进的性能

另外，具有模式匹配能力的 switch（包括 switch 语句和 switch 表达式），还提高了多情景处理性能。

如果使用 if-else 的处理方式，每一个情景，都要至少对应一个 if-else 语句。寻找匹配情景时，需要按照 if-else 的使用顺序来执行，直到遇到条件匹配的情景为止。这样，对于 if-else 语句来说，找到匹配情景的时间复杂度是 O(N)，其中 N 指的是需要处理的情景的数量。换句话说，if-else 语句寻找匹配情景的时间复杂度和需要处理的情景数量成正比。

如果使用 switch 的处理方式，每一个情景，也要至少对应一个 case 语句。但是，寻找匹配情景时，switch 并不需要按照 case 语句的顺序执行。对于 switch 的处理方式，找到匹配的情景的时间复杂度是 O(1)。也就是说，switch 寻找匹配情景的时间复杂度和需要处理的情景数量关系不大。

情景越多，使用 switch 的处理方式获得的性能提升就越大。

什么时候使用 default？

在前面的代码里，我们并没有看到 switch 的缺省选择情景 default 关键字的使用。在 switch 的模式匹配里，我们还可以使用缺省选择情景。比如说，我们可以使用 default 来实现前面讨论的 isSquare 这个方法。

```
public static boolean isSquare(Shape shape) {
    return switch (shape) {
        case Shape.Square s -> true;
        case null, default -> false;
    };
}
```

使用了 default，也就意味着这样的 switch 表达式总是能够穷举出所有的情景。遗憾的是，这样的代码丧失了检测匹配情景有没有变更的能力；也丧失了解决阅读案例里提到的问题的能力。

所以，一般来说，**只有我们能够确信，待匹配类型的升级，不会影响 switch 表达式的逻辑的时候，我们才能考虑使用缺省选择情景**。

总结

好，到这里，我来做个小结。从前面的讨论中，我们重点了解了 switch 的模式匹配，以及如何使用 switch 表达式来检测子类扩充出现的兼容性问题。具有模式匹配能力的 switch，提升了 switch 的数据类型匹配能力。switch 要匹配的数据，现在可以是整形的原始类型（数字、枚举、字符串），或者引用类型。

在前面的讨论里，我们把重点放在了 switch 表达式上。实际上，除了情景穷举相关的内容之外，我们的讨论也适用于 switch 语句。

在我们日常的编码实践中，为了尽早暴露子类扩充出现的兼容性问题，降低代码的维护难度，提高多情景处理的性能，我们应该优先考虑使用 switch 的模式匹配，而不是传统的 if-else 语句。

如果你想要丰富你的代码评审清单，有了 switch 的模式匹配以后，你可以加入下面这几条：

处理情景选择的 if-else 语句，是不是可以使用 switch 的模式匹配？

使用了模式匹配的 switch 表达式，有没有必要使用缺省选择情景 default？

使用了模式匹配的 switch 语句和表达式，是不是可以使用 null 选择情景？

另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你们面试中出现哦。通过这一次学习，你应该能够：

- 知道 switch 能够适配不同的类型，并且能够使用 switch 的模式匹配；
    - 面试问题：你知道怎么使用 switch 匹配不同的类型吗？
- 了解 switch 的模式匹配要解决的问题，以及它的特点；
    - 面试问题：使用 switch 的模式匹配有哪些好处？
- 掌握怎么使用 switch 表达式处理子类扩充带来的兼容性问题。
    - 面试问题：子类扩充有可能遇到什么问题，该怎么解决？

子类扩充出现的兼容性问题，是面向对象编程实践中一个棘手、重要、高频的问题。如果你能够有意识地使用 switch 的模式匹配，并且编写的代码能够自动检测到子类扩充出现的变动，就可以降低代码的维护难度和维护成本，提高代码的健壮性。在面试的时候，如果你能够主动地在代码里使用 switch 的模式匹配，而不是传统的 if-else 语句，这会是一个震惊面试官的好机会。

思考题

关于 switch 的模式匹配，还有两个特点我们没有讨论。一个是匹配情景的支配地位，一个是戒备模式的匹配情景。这一次的思考题，主要是一个阅读作业，也是自学这两个特点的一个家庭作业。

希望你可以阅读 switch 的模式匹配的官方文档，然后找出并且改正下面这段代码的错误，尽可能地优化这段代码。

```
public static boolean isSquare(Shape shape) {
    if (shape == null) {
        return false;
    }

    return switch (shape) {
        case Shape.Square s -> true;
        case Shape.Rectangle r -> false;
        case Shape.Rectangle r && r.length() == r.width() -> true;
        default ->false;
    };
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及验证的代码和结果。我们下节课见！

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求

代码，请在 switch 模式匹配专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 pattern/review/xuelei 的目录下面。
注：switch 的模式匹配这个特性，在 JDK 17 还是预览版。你可以现在开始学习这个特性，但是暂时不要把它用在严肃的产品里，直到正式版发布。

08 | 抛出异常，是不是错误处理的第一选择？

你好，我是范学雷。从今天开始，我们进入这个专栏的第二个部分。在这一部分，我们重点聊一聊代码的性能。这节课呢，我想跟你讨论 Java 的错误处理。

Java 的错误处理，算不上是特性。但是 Java 错误处理的缺陷和滥用，却一直是一个很有热度的话题。其中，Java 异常的使用和处理，是滥用最严重，诟病最多，也是最难平衡的一个难题。

为了解决花样百出的 Java 错误处理问题，也有过各种各样的办法。然而，到目前为止，我们还没有看到能解决所有问题的好方法，这也是编程语言研究者们的努力方向。

不过也正是因此，我们就更需要掌握 Java 错误处理的机制，平衡使用各种解决办法，妥善处理好 Java 异常。我们还是通过案例和代码，来看看 Java 异常的滥用，以及可能的解决方案吧。

阅读案例

我们知道，Java 语言支持三种异常的状况：非正常异常（Error），运行时异常（Runtime Exception）和检查型异常（Checked Exception）。关于这三种异常状况的介绍，你可以参考《异常处理都有哪些陷阱？》这篇文章。

通常情况下，我们谈到异常的时候，除非有特别的声明，不然指的都是运行时异常或者检查型异常。

我们还知道，异常状况的处理会让代码的效率变低，所以我们**不应该使用异常机制来处理正常的状况**。一个流畅的业务，理想的情况是，在执行代码时没有任何异常发生。否则，业务执行的效率就会大打折扣。

异常处理对代码执行效率的影响有多大呢？我们先要对这个问题有一个直观的感受，然后才能体会“不应该使用异常机制来处理正常的状况”这句话的分量，认识到异常滥用的危害。

下面的这段代码，测试了两个简单用例的吞吐量。这两种状况，都试图截取一段字符串。但是其中一个基准测试没有抛出异常；另外一个基准测试，由于字符串访问越界，抛出了运行时异常。为了让两个基准测试更具有对比性，我们在两个基准测试里，使用了相同的代码结构。

```
package co.ivi.jus.agility.former;
// snipped
public class OutOfBoundsBench {
    private static String s = "Hello, world!";  // s.length() == 13.
    // snipped
    @Benchmark
    public void withException() {
        try {
            s.substring(14);
        } catch (RuntimeException re) {
            // blank line, ignore the exception.
        }
    }
    @Benchmark
    public void noException() {
        try {
            s.substring(13);
        } catch (RuntimeException re) {
            // blank line, ignore the exception.
        }
    }
}
```

基准测试的结果可能会让你大吃一惊。没有抛出异常的用例，它能够支持的吞吐量要比抛出

异常的用例大 1000 倍。

```
Benchmark                                Mode  Cnt          Score          Error  Units
OutOfBoundsBench.noException            thrpt   15  566348609.338 ± 22165278.114  ops/s
OutOfBoundsBench.withException          thrpt   15     504193.920 ±    26489.992  ops/s
```

如果用运营成本来衡量一下的话，你可以考虑按照使用的计算资源来计算费用的环境，比如云计算。如果没有抛出异常的用例要花一万块钱的话，抛出异常的用例就需要 1000 万才能支持相同数量的用户。如果一个黑客能够找到这样的运行效率问题，它足以让一个应用多掏 1000 倍的钱，或者直到应用耗尽分配的计算资源，无法继续提供服务为止。

这样的评估当然很粗陋，但是足以说明抛出异常对软件效率的影响。我们当然不希望我们编写的代码存在这么一个烧钱的问题。

这时候我们就会设想：我们的代码，能不能没有任何异常状况发生？我们前面也提到过，“一个流畅的业务，理想的情况是，在执行代码时没有任何异常状况发生”。

可惜，这几乎是无法完成的任务。随便翻一翻 Java 的代码，不管是 JDK 这样的核心类库，还是支持业务的应用软件，我们都能看到大量的异常处理代码。

比如说吧，我们要用 Java 搭建一个服务器。通常情况下，如果业务逻辑出现了问题，比如说用户输入的数据不合规范，我们都会抛出一个异常，标记出问题的数据，并且记录下来问题出现的路径。但是，无论出现什么样的业务问题，服务器崩溃都是不能接受的结果。所以，我们的服务器会捕获所有的异常，不管是运行时异常，还是检查型异常；然后从异常中恢复过来，继续提供服务。

但是场景是否异常有时候只是角度问题。比如说：输入数据不规范，从检查用户数据代码这个角度去看，这是一个不正常的情景，所以抛出异常；但是，如果从要求不间断运营的服务器的角度来看，这就只是一个需要应用程序妥善处理的正常状况，是一个正常的情景了。所以，服务器要能够从这样的异常中恢复过来，继续运行。

然而，现在稍微复杂一点的软件，都是很多类库集成的。大部分类库，都只从自己的角度考虑问题，并且使用异常来处理遇到的问题。除非是很简单的代码，不然我们很难期望一个业务执行下来没有任何异常状况发生。

毫无疑问，抛出异常影响了代码的运行效率。但是，我们又没有别的办法躲开这样的影响。所以，有些新的编程语言（比如 Go 语言）干脆就彻底抛弃了类似于 Java 这样的异常机制，重新拥抱 C 语言的错误码方式。

讨论案例

接下来的讨论，为了方便我们反复地修改代码，我会使用下面这个案例。

我们知道，在设计算法公开接口的时候，算法的敏捷性是必须要考虑的问题。因为，算法总是会演进，旧的算法会过时，新的算法会出现。一个应用程序，应该能够很方便地升级它的算法，自动地淘汰旧算法，采纳新算法，而不需要太大的改动，甚至不需要改动源代码。所以，算法的公开接口经常使用通用的参数和结构。

比如说，我们获取一个单项散列函数实例的时候，一般不会直接调用这个单项散列函数的构造函数。而是用一个类似于工厂模式的集成环境，来构造出这个单项散列函数的实例。

就像下面的这段代码里的 of 方法。这个 of 方法，使用了一个字符串作为输入参数。我们可以把它作为配置参数写在配置文件里。修改配置文件之后，不需要改动调用它的源代码就能升级算法了。

```
package co.ivi.jus.agility.former;
import java.security.NoSuchAlgorithmException;
public sealed abstract class Digest {
    private static final class SHA256 extends Digest {
        @Override
        byte[] digest(byte[] message) {
            // snipped
        }
```

```
    }

    private static final class SHA512 extends Digest {
        @Override
        byte[] digest(byte[] message) {
            // snipped
        }
    }
    public static Digest of(String algorithm) throws NoSuchAlgorithmException {
        return switch (algorithm) {
            case "SHA-256" -> new SHA256();
            case "SHA-512" -> new SHA512();
            default -> throw new NoSuchAlgorithmException();
        };
    }
    abstract byte[] digest(byte[] message);
}
```

当然，通用参数也有它自己的问题。比方说，字符串的输入参数可能有疏漏，或者不是一个可以支持的算法。这时候，站在 of 方法的角度，就需要处理这样的异常状况。反映到代码上，of 方法要声明如何处理不合法的输入参数。上面的代码，使用的办法是抛出一个检查型异常。

那么，使用这个 of 方法的代码，就需要处理这个检查型异常。下面的代码，描述的就是一个使用这个方法的典型的例子。

```
try {
    Digest md = Digest.of(digestAlgorithm);
    md.digest("Hello, world!".getBytes());
} catch (NoSuchAlgorithmException nsae) {
    // snipped
}
```

既然使用了异常处理，当然也就会有我们在阅读案例里讨论过的异常处理的性能问题。我也试着给这个方法做了异常处理方面的基准测试。测试结果显示，没有抛出异常的用例，它能够支持的吞吐量要比抛出异常的用例大了将近 2000 倍。有了前面阅读案例的知识和铺垫，你应该对这样的性能差异早已有了心理准备。

```
Benchmark                          Mode  Cnt           Score          Error  Units
ExceptionBench.noException        thrpt   15  1318854854.577 ± 14522418.634  ops/s
ExceptionBench.withException      thrpt   15      713057.511 ±    16631.048  ops/s
```

重回错误码

那么，既然异常处理的效率这么让人揪心，我们编写的 Java 代码能够像 Go 语言一样重回错误码方式吗？这是我们首先要探索的一个方向。

也就是说，如果一个方法不需要返回值，我们可以试着把它修改为返回错误码。这是一个很直观的修改方式。

```
- // no return value
- public void doSomething();
+ // return an error code if run into problems, otherwise 0.
+ public int doSomething();
```

但是，如果一个方法需要一个返回值，我们就不能使用只返回错误码这种方式了。如果有一种方法，既能返回返回值，也能返回错误码，那么代码就会得到显著的改善。因此，我们需要设计一个数据结构，来支持这样的返回方式。

下面代码里的 Coded 这个档案类，就是一个能够满足这样要求的数据结构。

```
public record Coded<T>(T returned, int errorCode) {
```

```
    // blank
};
```

如果一个方法执行成功，它的返回值应该存放在 Coded 的 returned 变量里；如果执行失败，失败的错误码应该存放在 Coded 的 errorCode 变量里。我们可以把讨论案例里的 of 方法，修改成使用错误码的形式，就像下面的这段代码这样。

```
public static Coded<Digest> of(String algorithm) {
    return switch (algorithm) {
        case "SHA-256" -> new Coded(sha256, 0);
        case "SHA-512" -> new Coded(sha512, 0);
        default -> new Coded(null, -1);
    };
}
```

对应地，这个方法的使用就需要处理错误码。下面的代码，就是一个该怎么使用错误码的例子。

```
Coded<Digest> coded = Digest.of("SHA-256");
if (coded.errorCode() != 0) {
    // snipped
} else {
    coded.returned().digest("Hello, world!".getBytes());
}
```

看了上面的代码，我想你应该已经能够判断出来它的性能状况了。我们还是用基准测试来验证一下我们猜想吧。

测试结果显示，没有返回错误码的用例，它能够支持的吞吐量和返回错误码的用例几乎没有差别。这就是我们想要的结果。

```
Benchmark                    Mode  Cnt           Score          Error  Units
CodedBench.noErrorCode      thrpt   15  1320977784.955 ± 7487395.023  ops/s
CodedBench.withErrorCode    thrpt   15  1068513642.240 ± 69527558.874  ops/s
```

重回错误码的缺陷

不过，重回错误码的选择并不是没有代价的。刚才，我们在性能优化的同时，也放弃了代码的可读性和可维护性。异常处理能够解决掉的，也就是 C 语言时代的错误处理的缺陷，又重新回来了。

需要更多的代码

使用异常处理的代码，我们可以在一个 try-catch 语句块里包含多个方法的调用；每一个方法的调用都可以抛出异常。这样，由于异常的分层设计，所有的异常都是 Exception 的子类；我们也就可以一次性地处理多个方法抛出的异常了。

```
try {
    doSomething();      // could throw Exception
    doSomethingElse();  // could throw RuntimeException
    socket.close();     // could throw IOException
} catch (Exception ex) {
    // handle the exception in one place.
}
```

如果使用了错误码的方式，每一个方法调用都要检查返回的错误码。一般情况下，同样的逻辑和接口结构，使用错误码的方式需要编写更多的代码。

对于简单的逻辑和语句，我们可以使用逻辑运算符合并多个语句。这种紧凑的方式，牺牲了代码的可读性，不是我们喜欢的编码风格。

```
if (doSomething() != 0 &&
    doSomethingElse() != 0 &&
    socket.close() != 0) {
    // handle the exception
```

```
}
```
但是，对于复杂的逻辑和语句来说，紧凑的方式就行不通了。这时候，就需要一个独立的代码块来处理错误码。这样的话，结构重复的代码就会增加，这是我们在 C 语言编写的代码里经常见到的现象。

```
if (doSomething() != 0) {
    // handle the exception
};
if (doSomethingElse() != 0) {
    // handle the exception
};
if  (socket.close() != 0) {
    // handle the exception
}
```

丢弃了调试信息

不过，重回错误码最大的代价，是可维护性大幅度降低。使用异常的代码，我们能够通过异常的调用堆栈，清楚地看到代码的执行轨迹，快速找到出问题的代码。这也是我们使用异常处理的主要动力之一。

```
Exception in thread "main" java.security.NoSuchAlgorithmException: \
        Unsupported digest algorithm SHA-128
  at co.ivi.jus.agility.former.Digest.of(Digest.java:31)
  at co.ivi.jus.agility.former.NoCatchCase.main(NoCatchCase.java:12)
```

但是，使用错误码之后，就不再生成调用堆栈了。虽然这可以让资源的消耗减少，也能够提升代码性能，但是调用堆栈能带来的好处也就没有了。

另外，能够快速地找到代码的问题，也是一个编程语言的竞争力。如果我们决定重回错误码的处理方式，千万不要忘了提供快速排查问题的替代方案。比如使用更详尽的日志，或者使用启用 JFR（Java Flight Recorder）来收集诊断和分析数据。如果没有替代方案，我相信你会非常怀念使用异常的好处。

其实呀，C 语言时代的错误码，和 Java 语言时代的异常处理机制，就像是跷跷板的两端，一端是性能，一端是可维护性。在 Java 诞生的时候，有一个假设，就是计算能力会快速演进，所以性能的分量会有所下降，而可维护性的分量会放得很重。然而，如果演进到按照计算能力计费的时代，我们可能需要重新考量这两个指标各自所占的比重了。这时候，一部分代码可能就需要把性能的分量放得更重一些了。

易碎的数据结构

如果你阅读过我的另外一个专栏《代码精进之路》，你应该能够理解，一个新机制的设计，必须要简单、皮实。所谓的皮实，就是怎么用怎么对，纪律少、要求低，不容易犯错误。我们使用这样的准则，来看看上面设计的 Coded 这个档案类，是不是足够皮实。

生成一个 Coded 的实例，需要遵守两条纪律。第一条纪律是错误码的数值必须一致，0 代表没有错误，如果是其他的值表示出现了错误；第二条纪律是不能同时设置返回值和错误码。违反了任何一条纪律，都会出现不可预测的错误。

但是，这两条纪律需要编写代码的人自觉实现，编译器不会帮助我们检查错误。

比如下面的代码，对于编译器来说就是合法的代码。但对我们来说，这样的代码很明显违反了使用错误码需要遵守的规矩。这也就意味着，生成错误码的方式，不够皮实。

```
public static Coded<Digest> of(String algorithm) {
    return switch (algorithm) {
        // INCORRECT: set both error code and value.
        case "SHA-256" -> new Coded(sha256, -1);
        case "SHA-512" -> new Coded(sha512, 0);
        default -> new Coded(sha256, -1);
    };
}
```

我们再来看看使用错误码的代码。使用错误码，也有一条铁的纪律：必须首先检查错误码，

然后才能使用返回值。同样，编译器也不会帮助我们检查违反纪律的错误。下面的代码，就没有正确使用错误码。我们需要依靠经验才能避免这样的错误。所以，使用错误码的方式，也不够皮实。

```
Coded<Digest> coded = Digest.of("SHA-256");
// INCORRECT: use returned value before checking error code.
coded.returned().digest("Hello, world!".getBytes());
```

需要的纪律越多，我们犯错的可能性就越大。那有没有改进的方案，能够减少这些额外的要求呢？

改进方案：共用错误码

我们希望，改进的方案能够同时考虑生成错误码和使用错误码两端的需求。下面这段代码就是一个改进的设计。

```
public sealed interface Returned<T> {
    record ReturnValue<T>(T returnValue) implements Returned {
    }

    record ErrorCode(Integer errorCode) implements Returned {
    }
}
```

在这个改进的设计里，我们使用了封闭类。我们知道封闭类的子类是可以穷举的，这是这项改进需要的一个重要特点。我们把 Returned 的许可类（ReturnValue 和 ErrorCode）定义成档案类，分别表示返回值和错误代码。这样，我们就有了一个精简的方案。

下面这段代码，就是用新方案生成返回值和错误码的一个例子。可以看到，相比较使用 Coded 档案类的例子，这里的返回值和错误码分离开了。一个方法，返回的要么是返回值，要么是错误码，而不是同时返回两个值。这种方式，又把我们带回到了熟悉的编码方式。

```
public static Returned<Digest> of(String algorithm) {
    return switch (algorithm) {
        case "SHA-256" -> new ReturnValue(new SHA256());
        case "SHA-512" -> new ReturnValue(new SHA512());
        case null, default -> new ErrorCode(-1);
    };
}
```

而且，生成 Coded 实例需要遵守的两条纪律，在这里也不需要了。因为，返回 ReturnValue 这个许可类，就表示没有错误；返回 ErrorCode 这个许可类，就表示出现错误。这样的设计，就变得简单、皮实多了。

接下来，我们再看看使用错误码的情况。下面的这段代码，我们使用了前面讨论过的 switch 匹配的新特性。Returned 这个封闭类被设计成了一个没有方法的接口，要想获得返回值，我们就必须要使用它的许可类 ReturnValue，或者 ErrorCode。

```
Returned<Digest> rt = Digest.of("SHA-256");
switch (rt) {
    case ReturnValue rv -> {
            Digest d = (Digest) rv.returnValue();
            d.digest("Hello, world!".getBytes());
        }
    case ErrorCode ec ->
            System.out.println("Failed to get instance of SHA-256");
}
```

如果一个方法的调用返回的是 Returned 实例，我们就知道，它要么是代表返回值的 ReturnValue 对象，要么是代表错误码的 ErrorCode 对象。而且，你要使用返回值，就必须检查它是不是一个 ReturnValue 的实例。这种情况下，使用 Coded 档案类编写代码需要遵守的纪律，也就是必须先检查错误码，在这里也不需要了。使用错误码的这一端，也变得更加

简单、皮实了。
当然，使用封闭类来分别表示返回值和错误码的方式，只是改进错误码的其中一种方式。这种方式仍然具有一些缺陷，例如它本身没有携带调试信息。在 Java 的错误处理方面，我们希望未来能够有更好的设计和更多的探索，让我们的代码更完善。

总结

好，这节课就讲到这里，我来做个小结。从前面的讨论中，我们了解了 Java 异常处理带来的性能问题，我还给你展示了使用错误码的方式进行错误处理的方案。使用错误码的方式进行错误处理，错误码不能携带调试信息，这提高了错误处理的性能，但是增加了错误排查的困难，降低了代码的可维护性。
我们在代码里，是应该使用错误码，还是应该使用异常，这是一个需要根据应用场景认真权衡的问题。Java 的新特性，尤其是封闭类和档案类，为我们在 Java 的软件里使用错误码的形式，提供了强大的支持，让我们有了新的选择。
如果你想要丰富你的代码评审清单，错误码可以作为一个可评估的选项，进入你的考察指标内：
使用异常的机制进行错误处理，是不是一个最优的选择？
另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你们面试中出现哦。通过今天的学习，你应该能够：

- 清楚 Java 异常处理所带来的性能问题，对这一问题的影响程度有一个大致的概念；
    - 面试问题：你知道 Java 异常处理会产生什么问题吗？
- 了解 Java 异常处理的替代方案，以及它的优势和劣势；
    - 面试问题：你知道怎么提高 Java 代码的性能吗？

使用封闭类和档案类这样的 Java 新技术，为 Java 的错误处理寻求一个替代方案，这是一个崭新的、尚未开发的课题。在面试的时候，我们经常会遇到对代码性能有着苛刻要求的场景，如果你能够借助新特性展示错误处理的替代方案，并且不回避这个方案存在的问题，这一定是一个彰显你创新能力的好时机。

思考题

在前面的替代方案中，我们使用封闭类来分别表示了返回值和错误码，在使用错误码的代码里，我们使用了 switch 的模式匹配。可是，直到 JDK 17，switch 的模式匹配这个特性还只是一个预览版，还没有最终定稿。一般情况下，我们可以研究探索，但是不推荐使用预览版的特性。那么，如果不使用 switch 的模式匹配，使用错误码的代码可能是什么样子的呢？这是这一次的思考题。
为了方便你阅读，我把 switch 模式匹配的代码放在了下面。你可以在这个基础上替换掉 switch 模式匹配，看看最后会是什么样子的。

```
package co.ivi.jus.error.review.xuelei;
import co.ivi.jus.error.union.Digest;
import co.ivi.jus.error.union.Returned;
public class UseCase {
    public static void main(String[] args) {
        Returned<Digest> rt = Digest.of("SHA-256");
        switch (rt) {
            case Returned.ReturnValue rv -> {
                    Digest d = (Digest) rv.returnValue();
                    d.digest("Hello, world!".getBytes());
                }
            case Returned.ErrorCode ec ->
                    System.out.println("Failed to get instance of SHA-256");
        }
    }
```

```
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及验证的代码和结果。我们下节课再见！

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在错误处理专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 error/review/xuelei 的目录下面。

09 | 异常恢复，付出的代价能不能少一点？

你好，我是范学雷。今天，我们接着讨论 Java 的错误处理。这一讲，是上一次我们讨论的关于错误处理问题的继续和升级。

就像我们上一次讨论到的，Java 的异常处理是一个对代码性能有着重要影响的因素。所以说，Java 错误处理的缺陷和滥用也成为了一个热度始终不减的老话题。但是，Java 的异常处理，有着天生的优势，特别是它在错误排查方面的作用，我们很难找到合适的替代方案。那有没有可能改进 Java 的异常处理，保持它在错误排查方面的优势的同时，提高它的性能呢？这是一个又让马儿跑，又让马儿不吃草的问题。不过，这并不妨碍我们顺着这个思路，找一找其中的可能性。

我们还是先从阅读案例开始，来试着找一找其中的蛛丝马迹吧。

阅读案例

要尝试解决一个问题，我们首先要做的，就是把问题梳理清楚，定义好。我们先来看看 Java 异常处理的三个典型使用场景。

下面的这段代码里，有三个不同的异常使用方法。在分别解析的过程中，你可能会遇到几个疑问，不过别急，带着这几个问题，我们最后来一一解读。

```
package co.ivi.jus.stack.former;
import java.security.NoSuchAlgorithmException;
public class UseCase {
    public static void main(String[] args) {
        String[] algorithms = {"SHA-128", "SHA-192"};

        String availableAlgorithm = null;
        for (String algorithm : algorithms) {
            Digest md;
            try {
                md = Digest.of(algorithm);
            } catch (NoSuchAlgorithmException ex) {
                // ignore, continue to use the next algorithm.
                continue;
            }

            try {
                md.digest("Hello, world!".getBytes());
            } catch (Exception ex) {
                System.getLogger("co.ivi.jus.stack.former")
                        .log(System.Logger.Level.WARNING,
                             algorithm + " does not work",
                             ex);
                continue;
            }

            availableAlgorithm = algorithm;
        }

        if (availableAlgorithm != null) {
            System.out.println(availableAlgorithm + " is available");
        } else {
            throw new RuntimeException("No available hash algorithm");
        }
```

```
    }
}
```

可恢复异常

第一种就是可恢复的异常处理。

这是什么意思呢？对于代码里的异常 NoSuchAlgorithmException 来说，这段代码尝试捕获、识别这个异常，然后再从异常里恢复过来，继续执行代码。我们把这种可以从异常里恢复过来，继续执行的异常处理叫做可恢复的异常处理，简称为可恢复异常。

为了深入理解可恢复异常，我们需要仔细地看看 NoSuchAlgorithmException 这个异常的处理过程。这个处理的过程，其实就只有一行有效的代码，也就是 catch 语句。

```
} catch (NoSuchAlgorithmException nsae) {
    // ignore, continue to use the next algorithm.
}
```

只要 catch 语句能够捕获、识别到这个异常，这个异常的生命周期就结束了。catch 只需要知道异常的名字，而不需要知道异常的调用堆栈。不使用异常的调用堆栈，也就意味着这样的异常处理，极大地消弱了 Java 异常在错误排查方面的作用。

既然可恢复异常不使用异常的调用堆栈，是不是可恢复异常就不需要生成调用堆栈了呢？这是我们提出的第一个问题。

从 Java 异常的性能基准测试结果看，我们知道，生成异常的调用堆栈是异常处理影响性能的最主要因素。如果不需要生成调用堆栈，那么 Java 异常的处理性能就会有成百上千倍的提升。所以，如果我们找到了第一个问题的答案，我们就解决了可恢复异常的性能瓶颈。

不可恢复异常

好了，我们再回头看看第二个使用场景。对于代码里的异常 RuntimeException 来说，上面的代码并没有尝试捕获、识别它。这个异常直接导致了程序的退出，并且把异常的信息和调用堆栈打印了出来。

```
Exception in thread "main" java.lang.RuntimeException: No available hash algorith
m
  at co.ivi.jus.stack.former.UseCase.main(UseCase.java:27)
```

这样的异常处理方式导致了程序的中断，程序不能从异常抛出的地方恢复过来。我们把这种方式，叫做不可恢复的异常处理，简称为不可恢复异常。

调用堆栈对于不可恢复异常来说至关重要，因为我们可以从异常调用堆栈的打印信息里，快速定位到出问题的代码。毫无疑问，这加快了问题排查，降低了运维的成本。

由于不可恢复异常中断了程序的运行，所以它的性能开销是一次性的。因此，不可恢复异常对于性能的影响，其实我们不用太在意。

使用了异常信息和调用堆栈，又不用担心性能的影响，不可恢复异常似乎很理想。可是，在多大的程度上，我们可以允许程序由于异常中断而退出呢？这是一个很难回答的问题。

试想一下，如果是作为服务器的程序，我们会希望它能一直运行，遇到异常能够恢复过来。所以一般情况下，服务器的场景下，不会使用不可恢复异常。

现在的客户端程序呢？比如手机里的 app，如果遇到异常就崩溃，我们就不会有耐心继续使用了。似乎，客户端的程序，也没有多少不可恢复异常的使用场景。

也许，不可恢复异常的使用场景，仅仅存在于我们的演示程序里。高质量的产品里，似乎很难允许不可恢复异常的存在。

既然我们无法忍受程序的崩溃，那么不可恢复异常还有存在的必要吗？这是我们提出的第二个问题。

记录的调试信息

最后，我们再来看看第三个使用场景。对于代码里的异常 Exception 来说，这段代码尝试捕获、识别这个异常，然后从异常里恢复过来继续执行代码。它是一个可恢复的异常。和第一个场景不同的是，这段代码还在日志里记录了下了这个异常；一般来说，这个异常的调试信息，也就是异常信息和调用堆栈，也会被详细地记载在日志里。

其实，这也是可恢复异常的一个典型的使用场景；程序可以恢复，但是异常信息可以记录待查。

我们再来仔细看看异常信息是怎么记录在案的。为了方便我们观察，我把日志记录的这几行

代码单独摘抄了出来。

```
System.getLogger("co.ivi.jus.stack.former")
        .log(System.Logger.Level.WARNING,
             algorithm + " does not work",
             ex);
```

我们可以看到，日志记录下来了如下的关键信息：

1. 在异常捕获的场景下，这个异常的记录方式，包括是否记录（“co.ivi.jus.stack.former”）；
2. 在异常捕获的场景下，这个异常的记录地点（System.getLogger()）；
3. 在异常捕获的场景下，这个异常的严重程度（Logger.Level）；
4. 在异常捕获的场景下，这个异常表示的影响（“[algorithm] does not work”）；
5. 异常生成的时候携带的信息，包括异常信息和调用堆栈（ex）。

其中，前四项信息，是在方法调用的代码里生成的；第五项，是在方法实现的代码里生成的。也就是说，记录在案的调试信息，既包括调用代码的信息，也包括实现代码的信息。

如果放弃了 Java 的异常处理机制，我们还能够获得足够的调试信息吗？换种说法，我们有没有快速定位问题的替代方案？这是我们提出的第三个问题。

改进的共用错误码

刚才，我们通过 Java 异常处理的三个典型场景，提出了三个棘手的问题：

- 既然可恢复异常不使用异常的调用堆栈，是不是可恢复异常就不需要生成调用堆栈了？
- 既然我们无法忍受程序的崩溃，那么不可恢复异常还有存在的必要吗？
- 我们有没有快速定位问题的替代方案？

带着这三个问题，我们再来看看能不能改进一下我们上一讲里讨论的共用错误码的方案。

共用错误码本身，并没有携带调试信息。为了能够快速定位出问题，我们需要为共用错误码的方案补上调试信息。

下面的两段代码，就是我们要在补充调试信息方面做的尝试。第一段代码，是我们在方法实现的代码里的尝试。在这段代码里，我们使用异常的形式补充了调试信息，包括问题描述和调用堆栈。

```
public static Returned<Digest> of(String algorithm) {
    return switch (algorithm) {
        case "SHA-256" -> new Returned.ReturnValue(new SHA256());
        case "SHA-512" -> new Returned.ReturnValue(new SHA512());
        case null -> {
            System.getLogger("co.ivi.jus.stack.union")
                    .log(System.Logger.Level.WARNING,
                         "No algorithm is specified",
                         new Throwable("the calling stack"));
            yield new Returned.ErrorCode(-1);
        }
        default -> {
            System.getLogger("co.ivi.jus.stack.union")
                    .log(System.Logger.Level.INFO,
                    "Unknown algorithm is specified " + algorithm,
                            new Throwable("the calling stack"));
            yield new Returned.ErrorCode(-1);
        }
    };
}
```

第二段代码，是我们在方法调用的代码里的尝试。在这段代码里，我们补充了调用场景的信息。

```
Returned<Digest> rt = Digest.of("SHA-128");
switch (rt) {
    case Returned.ReturnValue rv -> {
            Digest d = (Digest) rv.returnValue();
            d.digest("Hello, world!".getBytes());
        }
    case Returned.ErrorCode ec ->
        System.getLogger("co.ivi.jus.stack.union")
                .log(System.Logger.Level.INFO,
                        "Failed to get instance of SHA-128");
}
```

经过这样的调整，类似于使用异常处理的、快速定位出问题的调试信息就又回来了。

```
Nov 05, 2021 10:08:23 PM co.ivi.jus.stack.union.Digest of
INFO: Unknown algorithm is specified SHA-128
java.lang.Throwable: the calling stack
  at co.ivi.jus.stack.union.Digest.of(Digest.java:37)
  at co.ivi.jus.stack.union.UseCase.main(UseCase.java:10)
Nov 05, 2021 10:08:23 PM co.ivi.jus.stack.union.UseCase main
INFO: Failed to get instance of SHA-128
```

你一定会有这样的问题。调试信息又回来了，难道不是以性能损失为代价的吗？

是的，使用调试信息带来的性能损失，并不比使用异常性能的损失小多少。不过好在，日志记录既可以开启，又可以关闭。如果我们关闭了日志，就不用再生成调试信息了，当然它的性能影响也就消失了。当需要我们定位问题的时候，再启动日志。这时候，我们就能够把性能的影响控制到一个极小的范围内了。

那么，使用错误码的错误处理方案，是怎么处理我们在阅读案例提到的问题的呢？

其实，每一个问题的处理，都很清晰。我把问题和答案都列在了下面的表格里，你可以看一看。

问题	共用错误码的解答
可恢复异常能不能不生成调用堆栈?	可以，如果不开启日志，就不生成调用堆栈。
不可恢复异常还有存在的必要吗?	错误码方案的所有错误，都可以恢复。
有没有快速定位出问题的替代方案?	有，开启日志，就能提供调试信息了。

当然，日志并不是唯一可以记录调试信息的方式。比如说，我们还可以使用更便捷的 JFR（Java Flight Recorder）特性。

其实，错误码的调试信息使用方式，更符合调试的目的：只有需要调试的时候，才会生成调试信息。那么，**如果继续沿用 Java 的异常处理机制，调试信息能不能按需开启、关闭呢？这是我们今天的第四个问题，也是提给 Java 语言设计师的问题。**

有了今天这四个问题做铺垫，如果有一天，Java 语言的异常能够支持可以开合的异常处理机制了，想必到时候你就不会感到惊讶了。

总结

好，到这里，我来做个小结。刚才，我们了解和讨论了 Java 异常处理的两个概念：可恢复异常和不可恢复异常。我还给出了在使用错误码的场景下，快速定位问题的替代方案。

这一讲我们并没有讨论新特性，而是我们重点讨论了现在 Java 异常处理机制的几个热门话题。这节课的重点，是要开拓我们的思维。了解这些热门的话题，不仅可以增加你的谈资，还可以切实地提高你的代码性能和可维护性。

另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你的面试中出现哦。通过这一次学习，你应该能够：

- 了解可恢复异常和不可恢复异常这两个概念，以及它们的使用场景；
    - 面试问题：你的代码是怎么处理 Java 异常的？
- 了解怎么在使用错误码的方案里，添加快速定位出问题的调试信息；
    - 面试问题：你的代码，是怎么定位可能存在的问题的？

对 Java 错误处理机制的改进，这会是一个持续热门的话题。而能够了解替代方案，并且使用替代方案的软件工程师，现在还不多。如果你能够展示错误处理的替代方案，而且还不牺牲异常处理的优势，这是一个能够在面试里获得主动权，控制话语权的必杀技。

思考题

怎么通过改进 Java 的异常处理，来获取性能的提升，我们已经花了两讲的时间了。我们提出的这些改进方案，其实依然有很大的提升空间。比如说吧，我们使用了整数表示错误码，这里其实就存在很多问题。

因为有时候，我们可能需要区别不同的错误，这样我们就不能总是使用一个错误码（-1）。如果存在多个错误码，我们怎么知道方法实现的代码返回的错误码是什么呢？编译器能不能帮助我们检查错误码的使用是不是匹配？比如说错误码的检查有没有遗漏，有没有多余？如果返回的错误码从两个增加到三个，使用该方法的代码能不能自动地检测到？

解决好这些问题，能够大幅度提高代码的可维护性和健壮性。该怎么解决掉这些问题呢？这是我们今天的思考题。

为了方便你阅读，我把需要两个错误码的案例代码放在了下面。一段代码是方法实现的代码，一段代码是方法使用的代码。你可以在这两段代码的基础上改动，看看最后你是怎么处理多个错误码的。

这一段是方法实现的代码。

```
public static Returned<Digest> of(String algorithm) {
    return switch (algorithm) {
        case "SHA-256" -> new Returned.ReturnValue(new SHA256());
        case "SHA-512" -> new Returned.ReturnValue(new SHA512());
        case null -> {
            System.getLogger("co.ivi.jus.stack.union")
                    .log(System.Logger.Level.WARNING,
                        "No algorithm is specified",
                        new Throwable("the calling stack"));
            yield new Returned.ErrorCode(-1);
        }
        default -> {
            System.getLogger("co.ivi.jus.stack.union")
                    .log(System.Logger.Level.INFO,
                    "Unknown algorithm is specified " + algorithm,
                            new Throwable("the calling stack"));
            yield new Returned.ErrorCode(-2);
        }
    };
}
```

这一段是方法使用的代码。

```
Returned<Digest> rt = Digest.of("SHA-128");
switch (rt) {
    case Returned.ReturnValue rv -> {
            Digest d = (Digest) rv.returnValue();
            d.digest("Hello, world!".getBytes());
        }
```

```
    case Returned.ErrorCode ec -> {
        if (ec.errorCode() == -1) {
            System.getLogger("co.ivi.jus.stack.union")
                    .log(System.Logger.Level.INFO,
                            "Unlikedly to happen");
        } else {
            System.getLogger("co.ivi.jus.stack.union")
                    .log(System.Logger.Level.INFO,
                            "SHA-218 is not supported");
        }
    }
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及验证的代码和结果。我们下节课再见！

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在异常恢复专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 stack/review/xuelei 的目录下面。

10 | Flow，是异步编程的终极选择吗？

你好，我是范学雷。今天，我们讨论反应式编程。

反应式编程曾经是一个很热门的话题。它是代码的控制的一种模式。如果不分析其他的模式，我们很难识别反应式编程的好与坏，以及最合适它的使用场景。所以，我们今天的讨论，和以往有很大的不同。

除了反应式编程之外，我们还会花很大的篇幅讨论其他的编程模式，包括现在的和未来的。希望这样的安排，能够帮助你根据具体的场景，选择最合适的模式。

我们从阅读案例开始，先来看一看最传统的模式，然后一步一步地过渡到反应式编程，最后我们再来稍微聊几句 Java 尚未发布的协程模式。

阅读案例

我想，你和我一样，无论是学习 C 语言，还是 Java 语言，都是从打印"Hello,world!"这个简单的例子开始的。我们再来看看这个我们熟悉的代码。

```
System.out.println("Hello, World!");
```

这段代码就是使用了最常用的代码控制模式：指令式编程模型。**所谓指令式编程模型，需要我们通过代码发布指令，然后等待指令的执行以及指令执行带来的状态变化。我们还要根据目前的状态，来确定下一次要发布的指令，并且用代码把下一个指令表示出来。**

上面的代码里，我们发布的指令就是：标准输出打印“Hello, World!”这句话。然后，我们就等待指令的执行结果，验证我们编写的代码有没有按照我们的指令工作。

指令式编程模型

指令式编程模型关注的重点就在于控制状态。"Hello, world!"这个例子能看出来一点端倪，但是要了解状态变化和控制，我们需要看两行以上的代码。

```
try {
    Digest messageDigest = Digest.of("SHA-256");
    byte[] digestValue =
            messageDigest.digest("Hello, world!".getBytes());
} catch (NoSuchAlgorithmException ex) {
    System.out.println("Unsupported algorithm: SHA-256");
}
```

在上面的这段代码里，我们首先调用 Digest.of 方法，得到一个 Digest 实例；然后调用这个实例的方法 Digest.digest，获得一个返回值。第一个方法执行完成后，获得了第一个方法执行后的状态，第二个方法才能接着执行。

这种顺序执行的模式，逻辑简单直接。简单直接本身就有着巨大的能量，特别是实现精确控制方面。所以，这种模式在通用编程语言设计和一般的应用程序开发中，占据着压倒性的优势。

但是，这种模式需要维护和同步状态。如果状态数量大，我们就要把大的代码块分解成小的代码块；这样，我们编写的代码才能更容易阅读，更容易维护。而更大的问题来自于状态同步需要的顺序执行。

比如说吧，上面的例子中，Digest.of 这个方法实现，可能效率很高，执行得很快；而 Digest.digest 这个方法的实现，它的执行速度可能就是毫秒级的，甚至是秒一级别的。在要求低延迟、高并发的环境下，等待 Digest.digest 调用的返回结果，可能就不是一个好的选择。换句话说，阻塞在方法的调用上，增加了系统的延迟，降低了系统能够支持的吞吐量。

这种顺序执行的模式带来的延迟后果，在互联网时代的很多场景下是无法忍受的（比如春节的火车票预售系统，或者网上购物节的订购系统等）。存在这种问题最典型的场景之一，就是客户端-服务器这种架构下的传统的套接字编程接口。它也引发了大约 20 年前提出的 C10K 问题（支持 1 万个并发用户）。

怎样解决 C10K 问题呢？一个主要方向，就是使用非阻塞的异步编程。

声明式编程模型

非阻塞的异步编程，并不是可以通过编程语言或者标准类库就可以得到的。支持非阻塞的异步编程，需要大幅度地更改代码，转换代码编写的思维习惯。

我们可以使用打电话来做个比方。
传统的指令式编程模型，就像我们通常打电话一样。我们拨打对方的电话号码，然后等待接听，然后通话，然后挂断。当我们挂断电话的时候，打电话这一个过程也就结束了，我们也拿到了想要的结果。
而非阻塞的异步编程，更像是电话留言。我们拨打对方的电话，告诉对方方便的时候，回拨电话，然后就挂断了。当我们挂断电话的时候，打电话这一个过程当然也是结束了，但是我们没有拿到想要的结果。想要的结果，还要依靠回拨电话，才能够得到。
而类似于回拨电话的逻辑，正是非阻塞的异步编程的关键模型。映射到代码上，就是使用回调函数或者方法。
当我们试图使用回调函数时，我们编写代码的思想和模型都会产生巨大的变化。我们关注的重点，就会从指令式编程模型的“控制状态”转变到“控制目标”。这时候，我们编程模型也就转变到了**声明式的编程模型**。
如果指令式编程模型的逻辑是告诉计算机“该怎么做”，那么声明式的编程模型的逻辑就是告诉计算机“要做什么”。指令式编程模型的代码像是流水线作业的工程师，事无巨细，拧好每一个螺丝；而声明式的编程模型的代码，更像是稳坐在军帐中的军师，布置任务，运筹帷幄。
我们前面讨论的 Digest，能不能实现非阻塞的异步编程呢？答案是肯定的，不过我们需要彻底地更改代码，从 API 到实现都要转换思路。下面这段代码里声明的 API，就是我们尝试使用声明式编程的一个例子。

```
public sealed abstract class Digest {
    public static void of(String algorithm,
        Consumer<Digest> onSuccess, Consumer<Integer> onFailure) {
        // snipped
    }
    public abstract void digest(byte[] message,
        Consumer<byte[]> onSuccess, Consumer<Integer> onFailure);
}
```

转化了思路的 Digest.of 方法，就像是布置任务：如果执行成功，请继续执行 A 计划（也就是 onSuccess 这个回调函数）；否则，就继续执行 B 计划（也就是 onFailure 这个回调函数）。其实，这也就是我们前面提到的，告诉计算机“要做什么”的概念。
有了回调函数的设计，代码的实现方式就放开了管制。无论是回调函数的实现，还是回调函数的调用，都可以自由地选择是采用异步的模式，还是同步的模式。不用说，这种自由很具有吸引力。从 JDK 7 引入 NIO 新特性开始，这种模式开始进入 Java 的工业实践，并且取得了巨大的成功。出现了一大批的明星项目。
不过，回调函数的设计也有着天生的缺陷。这个缺陷，就是回调地狱（Callback Hell，常被译为回调地狱。为了更直观地表达，我更喜欢把它叫做回调堆挤）。什么意思呢？通常地，我们需要布置多个小的任务，才能完成一项大的任务。这些小任务还有可能是有因果关系的任务，这时候，就需要小任务的配合，或者按顺序执行。
比如说，上面的 Digest 设计，我们先要判断 of 方法能不能成功；如果成功的话，那么就使用这个 Digest 实例，调用它的 Digest.digest 方法。而 Digest.digest 方法的调用，也要作出 A 计划和 B 计划。这样，两个回调函数的使用，就会堆积起来。如果回调函数的嵌套增多，代码看起来就像挤在一块一样，形式上不美观，阅读起来很费解，维护起来难度很大。
下面的这段代码，就是我们使用回调函数设计的 Digest 的一个用例。这个用例里，回调函数的嵌套仅仅有两层，代码的形式已经变得很难阅读了。你可以尝试编写一个 3 层或者 5 层的回调函数的嵌套，体验一下深度嵌套的代码是什么样子的。

```
Digest.of("SHA-256",
    md -> {
        System.out.println("SHA-256 is not supported");
        md.digest("Hello, world!".getBytes(),
            values -> {
                System.out.println("SHA-256 is available");
```

```
            },
            errorCode -> {
                System.out.println("SHA-256 is not available");
            });
    },
    errorCode -> {
        System.out.println("Unsupported algorithm: SHA-256");
    });
```

如果说，回调函数带来的形式的堆积我们还可以克服的话；那这种形式上的堆积带来的逻辑堆积，我们就几乎不可承受了。**逻辑上的堆积，意味着代码的深度耦合。而深度耦合，意味着代码维护困难。深度嵌套里的一点点代码修改，都可能通过嵌套层层朝上传递，最后牵动全局。**

这就导致，使用回调函数的声明式编程模型有着严重的场景适应问题。我们通常只使用回调函数解决性能影响最大的模块，比如说网络数据的传输；而大部分的代码，依然使用传统的、顺序执行的指令式模型。

好在，业界也有很多努力，试图改善回调函数的使用困境。其中最出色也是影响最大的一个，就是反应式编程。

反应式编程

反应式编程的基本逻辑，仍然是告诉计算机“要做什么”；但是它的关注点转移到了数据的变化以及数据和变化的传递上，或者说，是转移到了对数据变化的反应上。所以，**反应式编程的核心是数据流和变化传递。**

如果我们从数据的流向角度来看的话，数据有两种基本的形式：数据的输入和数据的输出。从这两种基本的形式，能够衍生出三种过程：最初的来源，数据的传递和最终的结局。

数据的输出

在 Java 的反应式编程模型的设计里，数据的输出使用只有一个参数的 Flow.Publisher 来表示。

```
@FunctionalInterface
public static interface Publisher<T> {
    public void subscribe(Subscriber<? super T> subscriber);
}
```

在 Flow.Publisher 的接口设计里，泛型 T 表示的就是数据的类型。数据输出的对象，是使用 Flow.Subscriber 来表示的。换句话说，数据的发布者通过授权订阅者，来实现数据从发布者到订阅者的传递。一个数据的发布者，可以有多个数据的订阅者。

需要注意的是，订阅的接口，安排在了 Flow.Publisher 这个接口里。这也就意味着，订阅者的订阅行为，是由数据的发布者发起的，而不是订阅者发起的。

数据最初的来源，就是一种形式的数据输出；它只有数据输出这一个传递方向，而不能接收数据的输入。

比如下面的代码，就是一个表示数据最初来源的例子。在这段代码里，数据的类型是字节数组；而数据发布的实现，我们使用了 Java 标准类库的参考性实现 SubmissionPublisher 这个类。

```
SubmissionPublisher<byte[]> publisher = new SubmissionPublisher<>();
```

数据的输入

下面，我们再来看下数据的输入。

在 Java 的反应式编程模型的设计里，数据的输入用只有一个参数的 Flow.Subscriber 来表示。也就是我们前面提到的订阅者。

```
public static interface Subscriber<T> {
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
```

```
}
```

在 Flow.Subscriber 的接口设计里，泛型 T 表示的就是数据的类型。这个接口里一共定义了四种任务，并分别规定了下面四种情形下的反应：

1. 如果接收到订阅邀请该怎么办？这个行为由 onSubscribe 这个方法的实现确定。
2. 如果接收到数据该怎么办？这个行为由 onNext 这个方法的实现确定。
3. 如果遇到了错误该怎么办？这个行为由 onError 这个方法的实现确定。
4. 如果数据传输完毕该怎么办？这个行为由 onComplete 这个方法的实现确定。

数据最终的结局，就是一种形式的数据输入；它只有数据输入这一个传递方向，而不能产生数据的输出。

比如下面的代码，就是一个表示数据最终结果的例子。在这段代码里，我们使用一个泛型来表示数据的类型；然后，使用了一个 Consumer 函数来表示我们该怎么处理接收到的数据。这样的安排让这个例子具有了普遍的意义。只要稍作修改，就可以把它使用到实际场景中去了。

```
package co.ivi.jus.flow.reactive;
import java.util.concurrent.Flow;
import java.util.function.Consumer;
public class Destination<T> implements Flow.Subscriber<T>{
    private Flow.Subscription subscription;
    private final Consumer<T> consumer;

    public Destination(Consumer<T> consumer) {
        this.consumer = consumer;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        subscription.request(1);
        consumer.accept(item);
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onComplete() {
        System.out.println("Done");
    }
}
```

数据的控制

你可能已经注意到了，Flow.Subscriber 接口，并没有和 Flow.Publisher 直接联系。取而代之地出现了一个中间代理 Flow.Subscription。Flow.Subscription 管理、控制着 Flow.Publisher 和 Flow.Subscriber 之间的连接，以及数据的传递。

也就是说，在 Java 的反应式编程模型里，数据的传递控制从数据和数据的变化里分离了出

来。这样的分离，对于降低功能之间的耦合意义重大。

```
public static interface Subscription {
    public void request(long n);
    public void cancel();
}
```

在 Flow.Subscription 的接口设计里，我们定义了两个方法。一个方法表示订阅者希望接收的数据数量，也就是 Subscription.request 这个方法。另一个方法表示订阅者希望取消订阅，也就是 Subscription.cancel 这个方法。

数据的传递

除了最初的来源和最终的结局，数据表现还有一个过程，就是数据的传递。数据的传递这个过程，既包括接收输入数据，也包括发送输出数据。在数据传递这个环节，数据的内容可能会发生变化，数据的数量也可能会发生变化（比如，过滤掉一部分的数据，或者修改输入的数据，甚至替换掉输入的数据）。

在 Java 的反应式编程模型的设计里，这样的过程是由 Flow.Processor 表示的。Flow.Processor 是一个扩展了 Flow.Publisher 和 Flow.Subscriber 的接口。所以，Flow.Processor 有两个数据类型，泛型 T 表述输入数据的类型，泛型 R 表述输出数据的类型。

```
public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {
}
```

下面的代码，就是一个表示数据传递的例子。在这段代码里，我们使用泛型来表示输入数据和输出数据的类型；然后，我们使用了一个 Function 函数，来表示该怎么处理接收到的数据，并且输出处理的结果。这样的安排让这个例子具有了普遍的意义。稍作修改，你就可以把它用到实际场景中去了。

```
package co.ivi.jus.flow.reactive;
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;
import java.util.function.Function;
public class Transform<T, R> extends SubmissionPublisher<R>
        implements Flow.Processor<T, R> {
    private Function<T, R> transform;
    private Flow.Subscription subscription;

    public Transform(Function<T, R> transform) {
        super();
        this.transform = transform;
    }

    @Override
    public void onSubscribe(Flow.Subscription subscription) {
        this.subscription = subscription;
        subscription.request(1);
    }

    @Override
    public void onNext(T item) {
        submit(transform.apply(item));
        subscription.request(1);
    }

    @Override
    public void onError(Throwable throwable) {
        closeExceptionally(throwable);
```

```
    }

    @Override
    public void onComplete() {
        close();
    }
}
```

过程的串联

既然数据的表述方式分为输入和输出两种基本的形式，而且还提供了由此衍生出来的三种过程，我们就能够把数据的处理过程，很方便地串联起来了。

下面的代码，就是我们试图把最初的来源、数据的传递和最终的结局这三个过程，串联成一个更大的过程的例子。当然，你也可以试着串联进更多的数据处理过程。

```
private static void transform(byte[] message,
          Function<byte[], byte[]> transformFunction) {
    SubmissionPublisher<byte[]> publisher =
            new SubmissionPublisher<>();
    // Create the transform processor
    Transform<byte[], byte[]> messageDigest =
            new Transform<>(transformFunction);
    // Create subscriber for the processor
    Destination<byte[]> subscriber = new Destination<>(
            values -> System.out.println(
                    "Got it: " + Utilities.toHexString(values)));
    // Chain processor and subscriber
    publisher.subscribe(messageDigest);
    messageDigest.subscribe(subscriber);
    publisher.submit(message);
    // Close the submission publisher.
    publisher.close();
}
```

串联的形式，接藕了不同环节的关联；而且每个环节的代码也可以换个场景复用。支持过程的串联，是反应式编程模型强大的最大动力之一。像 Scala 这样的编程语言，甚至把过程串联提升到了编程语言的层面来支持。这样做，毫无疑问大幅度地提高了编码的效率和代码的美观程度。

简洁的重构

介绍完 Java 的反应式编程模型设计，我们要回头看看我们在阅读案例里提出的问题了。反应式编程，是怎么解决顺序执行的模式带来的延迟后果的呢？反应式编程，怎么解决回调函数带来的堆挤问题呢？

我们还是先看一眼使用反应式编程模型的代码，然后再来讨论这些问题吧。下面的代码，就是我们对阅读案例里 Digest 用法的改进。

```
Returned<Digest> rt = Digest.of("SHA-256");
switch (rt) {
    case Returned.ReturnValue rv -> {
        // Get the returned value
        if (rv.returnValue() instanceof Digest d) {
            // Call the transform method for the message digest.
            transform("Hello, World!".getBytes(), d::digest);
            // Wait for completion
            Thread.sleep(20000);
        } else {  // unlikely
```

```
                System.out.println("Implementation error: SHA-256");
            }
        }
        case Returned.ErrorCode ec ->
                System.out.println("Unsupported algorithm: SHA-256");
}
```

在这个例子里，我们没有发现类似于回调函数一样的堆挤现象。这里面，起重要作用的就是我们上面提到的过程的串联这种形式。Java 的反应式编程模型里的过程串联和数据控制的设计，以及数据输入和输出的分离，降低了代码的耦合，不再需要嵌套的调用了。

在这个例子里，我们还看到了 Digest.digest 方法的直接使用。为了能够使用反应式编程模型，我们没有必要去修改 Digest 代码。只要把 Digest 原来的设计和实现，恰当地放到反应式编程模型里来，就能够实现异步非阻塞的设想了。这一点，无疑具有极大的吸引力。如果不是被逼无奈，谁会去颠覆已有的代码呢？

那到底反应式编程模型是怎么支持异步非阻塞的呢？其实，和回调函数一样，反应式编程既能够支持同步阻塞的模式，也能够支持异步非阻塞的模式。如果这些接口实现是异步非阻塞模式的，这些实现的调用，也就是异步非阻塞的。当然，反应式编程模型的主要使用场景，目前还是异步非阻塞模式。

比如我们例子中的 SubmissionPublisher，就是一个异步非阻塞模式的实现。在上面的代码里，如果没有调用 Thread.sleep，我们可能还看不到 Digest 的处理结果，主线程就退出了。这就是一个非阻塞的实现表现出来的现象。

缺陷与对策

到目前为止，反应式编程模型看起来还很完美。可是，反应式编程模型的缺陷也很要命。其中最要命的缺陷，就是错误很难排查，这是异步编程的通病。而反应式编程模型的解耦设计，加剧了错误排查的难度，这会严重影响开发的效率，降低代码的可维护性。

目前来看，解决反应式编程模型的缺陷，或者说是异步编程的缺陷的方向，似乎又要回到了指令式编程模型这条老路上来了。这里最值得提及的就是协程（Fiber）这个概念（目前，Java 的协程模式还没有发布，但是我可以带你先了解一下）。

我们再来看看阅读案例里提到的这段代码。为了方便你阅读，我把它拷贝粘贴到这里来了。

```
try {
    Digest messageDigest = Digest.of("SHA-256");
    byte[] digestValue =
            messageDigest.digest("Hello, world!".getBytes());
} catch (NoSuchAlgorithmException ex) {
    System.out.println("Unsupported algorithm: SHA-256");
}
```

在 Java 的指令式编程模型里，这段代码要在一个线程里执行。我们首先调用 Digest.of 方法，得到一个 Digest 实例；然后调用这个实例的方法 Digest.digest，获得一个返回值。在每个方法返回之前，线程都会处于等待状态。而线程的等待，是造成资源浪费的最大因素。而协程的处理方式，消除了线程的等待。如果调用阻塞，就会把资源切换出去，执行其他的操作。这就节省了大量的计算资源，使得系统在阻塞的模式下，支持大规模的并发。如果指令式编程模型能够通过协程的方式支持大规模的并发，也许它是一个颠覆现有高并发架构的新技术。

目前，Java 的协程模式还没有发布。它能够给反应式编程模型带来什么样的影响，能够给我们实现大规模并发系统带来多大的便利？这些问题的答案，我们还需要等待一段时间。

总结

好，到这里，我来做个小结。前面，我们讨论了指令式编程模型和声明式编程模型，回调函数以及回调地狱，以及 Java 反应式编程模型的基本组件。

限于篇幅，我们不能展开讨论 Java 反应式编程模型的各种潜力和变化，比如“反应式宣

言”“背压”这样的热门词汇。我建议你继续深入地了解反应式编程的这些要求（比如反应式宣言和反应式系统），以及成熟的明星产品（比如 Akka 和 Spring 5+）。
由于 Java 的协程模式还没有发布，我对反应式编程的未来还没有清晰的判断。也欢迎你在留言区里留言、讨论反应式编程的现在和未来。
另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你们面试中出现哦。通过这一次学习，你应该能够：

- 了解指令式编程模型和声明式编程模型这两个术语；
    - 面试问题：你知道声明式编程模型吗，它是怎么工作的？
- 了解 Java 反应式编程模型的基本组件，以及它们的组合方式；
    - 面试问题：你知道怎么使用 Java 反应式编程模型吗？
- 知道回调函数的形式，以及回调地狱这个说法。
    - 面试问题：你知道回调函数有什么问题吗？

反应式编程是目前主流的支持高并发的技术架构思路。学会反应式编程，意味着你有能力处理高并发应用这样的需求。能够编写高并发的代码，现在很重要，以后更重要。学会使用 Java 反应式编程模型这样一个高度抽象的接口，毫无疑问能够提升你的技术深度。

思考题

今天的思考题，我们来试着使用一下 Java 反应式编程模型。在讨论反应式编程的时候，计算 a=b+c 是一个常用的范例。在这个计算里，b 和 c 随着时间的推移，会发生变化。而每一次的变化，都会影响 a 的计算结果。
现在我们假设 a 表示的数据是一件事情结束的时候是星期几，b 表示的数据是一件事情开始的时候是星期几，c 表示处理完这件事情需要多少天。你会怎么使用 Java 反应式编程模型来处理这个问题？
欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！
注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在反应式编程专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 flow/review/xuelei 的目录下面。

11 | 矢量运算：Java 的机器学习要来了吗？

你好，我是范学雷。今天，我们讨论 Java 的矢量运算。

Java 的矢量运算，我写这篇文章的时候还在孵化期，还没有发布预览版。我们之所以选取了这样一个还处于孵化期的技术，主要是因为这个技术代表了 Java 语言发展的一个重要方向，在未来一定会有着重要的影响。早一点了解这样的技术，除了扩展视野之外，还能够帮助我们制定未来几年要学习或者要使用的技术路线。

我们从阅读案例开始，看一看没有矢量运算的时候，Java 是怎么支持科学计算的；然后，我们再看看矢量运算能够带来什么样的变化。

阅读案例

我想，你对线性方程（或者说一次方程）一定不陌生。一般情况下，我们可以把线性方程表述成下面的形式。

$$y = a_0x_0 + a_1x_1 + a_2x_2 + ... + a_{n-1}x_{n-1}$$

其中 $a0$，$a1$，$an-1$ 表示的是常数，$x0$，$x1$，$xn-1$ 表示的是变量，而 y 就表示 ai 和 xi 的组合结果。n 表示未知变量的数目，通常，我们也把它称为方程的维度。

如果给定方程式右边的常数和变量，我们就能计算出方程式左边的 y 数值了。那么，该怎么用代码表示这个方程式呢？我们可以把 $a0$，$a1$，$an-1$ 表示的常数放到一个数组里，把 $x0$，$x1$，$xn-1$ 表示的变量放到另外一个数组里。下面的代码里，变量 a 和 x 就可以用来表示一个有四个维度的一次方程组。

```
static final float[] a = new float[] {0.6F, 0.7F, 0.8F, 0.9F};
static final float[] x = new float[] {1.0F, 2.0F, 3.0F, 4.0F};
```

能用 Java 的变量来表示一次方程，我们也就能够计算线性方程的结果了。下面的代码，就是一个实现的办法。

```
private static Returned<Float> sumInScalar(float[] a, float[] x) {
    if (a == null || x == null || a.length != x.length) {
        return new Returned.ErrorCode(-1);
    }
    float[] y = new float[a.length];
    for (int i = 0; i < a.length; i++) {
        y[i] = a[i] * x[i];
    }
    float r = 0F;
    for (int i = 0; i < y.length; i++) {
        r += y[i];
    }
    return new Returned.ReturnValue<>(r);
}
```

在上面的代码里，我们先计算 ai 和 xi 的乘积，然后再计算乘积结果的总和。其中的乘法运算，就是我们常说的标量运算。为了方便讨论，我把乘法运算的代码单独拿出来，粘贴在下面。

```
float[] y = new float[a.length];
for (int i = 0; i < a.length; i++) {
    y[i] = a[i] * x[i];
}
```

如果我们仔细观察线性方程就会发现，对于每一个纬度，ai 和 xi 是互不影响的，当然它们的乘积也是互不影响的。既然每个维度的计算都互不影响，那么我们能不能并行计算呢？

矢量运算

Java 的矢量运算就是使用单个指令并行处理多个数据的一个尝试（单指令多数据，Single

Instruction Multiple Data)。
在现代的微处理器（CPU）中，一个控制器可以控制多个平行的处理单元；在现代的图形处理器（GPU）中呢，更是拥有强大的并发处理能力和可编程流水线。这些处理器层面的技术，为软件层面的单指令多数据处理提供了物理支持。Java 矢量运算的设计和实现，也是希望能够借助现代处理器的这种能力，提高运算的性能。
为了使用单指令多数据的指令，我们需要把不同数据的运算独立出来，让并行运算成为可能。而数学里的矢量运算，恰好就能满足这样的要求。
如果使用矢量，我们可以把线性方程表述成下面的形式（使用向量的数量积形式）：

$$y' = ax$$

$$y = \sum_{0}^{n-1} y'_i$$

其中，a，x 和 y' 是三个 n 维的矢量。

$$a = [a_0, a_1, a_2, \ldots a_{n-1}]$$

$$y' = [y'_0, y'_1, y'_2, \ldots, y'_{n-1}]$$

好了，现在我们可以看看 Java 是怎么表达矢量的了。下面代码里的变量 a，和前面阅读案例里 a 是一样的，它以数组的形式表示；变量 va，就是变量 a 的矢量表达形式。fromArray 这个方法，可以把一个数组变量，转换成一个矢量的变量。

```
static final float[] a = new float[] {0.6F, 0.7F, 0.8F, 0.9F};
static final FloatVector va =
        FloatVector.fromArray(FloatVector.SPECIES_128, a, 0);

static final float[] x = new float[] {1.0F, 2.0F, 3.0F, 4.0F};
static final FloatVector vx =
        FloatVector.fromArray(FloatVector.SPECIES_128, x, 0);
```

有了表示矢量的办法，我们就可以试着使用矢量运算的办法，来计算线性方程的结果了。下面的代码，就是一个简化了的实现。

```
private static Returned<Float> sumInVector(FloatVector va, FloatVector vx) {
    if (va == null || vx == null || va.length() != vx.length()) {
        return new Returned.ErrorCode(-1);
    }

    // FloatVector vy = va.mul(vx);
    float[] y = va.mul(vx).toArray();

    float r = 0F;
    for (int i = 0; i < y.length; i++) {
        r += y[i];
    }
    return new Returned.ReturnValue<>(r);
}
```

这个运算的关键部分是其中的矢量运算，也就是下面这行代码。

```
FloatVector vy = va.mul(vx);
```

和上面的标量运算的办法相比，矢量运算的代码精简了很多。这是矢量运算的第一个优点。但它的优点还不止于此。

float[] y = new float[a.length];
for (int i = 0; i < a.length; i++) {
  y[i] = a[i] * x[i];
}
	float[] y = va.mul(vx).toArray();

飙升的性能

我们前面提到，Java 矢量运算的设计，主要是为了性能。那么，性能的提升能有多大呢？我自己做了一个性能测试。虽然这个特性还处于孵化期，但是它的性能测试结果还是很令人振奋的。就上面这个简单的、四维的矢量来说，和我们在阅读案例里使用的标量运算相比，矢量运算的性能提高了足足有 10 倍。

```
Benchmark                                Mode  Cnt            Score            Error  Units
VectorBench.scalarComputation           thrpt   15    180635563.597 ±   30893274.582  ops/s
VectorBench.vectorComputation           thrpt   15   1839556188.443 ±  153876900.442  ops/s
```

对于一个还处于孵化阶段的实现来说，这么大的性能提升是有点超出预料的。

在密码学和机器学习领域，通常需要处理几百甚至几千维的数据。一般情况下，为了能够使用处理器的计算优势，我们经常需要特殊的设计以及内嵌于 JVM 的本地代码来获得硬件加速。这样的限制，让普通代码的计算很难获得硬件加速的好处。

希望成熟后的 Java 矢量运算，能在这些领域有出色的表现，让普通的代码获得处理器的单指令多数据的强大运算能力。毕竟，只有单指令多数据的优势能够被普通的 Java 应用程序广泛使用，Java 才能在机器学习、科学计算这些领域获得计算优势。

如果从机器学习在未来的重要性来说，Java 在科学计算领域的拓展来得也许正是时候。

总结

好，到这里，我来做个小结。前面，我们讨论了 Java 的矢量运算这个尚处于孵化阶段的新特性，对 Java 的矢量运算这个新特性有了一个初始的印象。

如果 Java 矢量运算成熟起来，许多领域都可以从这个新特性中受益，包括但是不限于机器学习、线性代数、密码学、金融和 JDK 本身的代码。

这一次学习的主要目的，就是让你对矢量运算有一个基本的印象。这样的话，如果你的代码里有大量的数值计算，也许可以考虑在将来使用矢量运算获得硬件的并行计算能力，大幅度提高代码的性能。

由于矢量运算尚处于孵化阶段，目前我们还不需要学习它的 API，知道 Java 有这个发展方向，并且能够思考你的代码潜在的改进空间就足够了。知道了这个方向，等 Java 矢量运算正式发布的时候，你就可以尽早地改进你的代码，从而获得领先的优势了。

如果面试中聊到了数值计算的性能，你应该知道有矢量运算这么一个潜在的方向，以及“单指令多数据”这么一个术语。

思考题

其实，今天的这个新特性，是练习使用 JShell 快速学习新技术的一个好机会。使用阅读案例里提供的数据，你能够使用 JShell，快速地表示出下面的这个矢量吗？

$y' = ax$

需要注意的是，要想使用孵化期的 JDK 技术，需要在 JShell 里导入孵化期的 JDK 模块，就像下面的例子这样。

```
$ jshell --add-modules jdk.incubator.vector -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
jshell> import jdk.incubator.vector.*;
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！
注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在矢量运算专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 vector/review/xuelei 的目录下面。

12 | 外部内存接口：零拷贝的障碍还有多少？

你好，我是范学雷。今天，我们来讨论 Java 的外部内存接口。

Java 的外部内存接口这个新特性，现在还在孵化期，还没有发布预览版。我之所以选取了这样一个还处于孵化期的技术，主要是因为这个技术太重要了。我们需要提前认识它；然后在这项技术出来的时候，尽早地使用它。

我们从阅读案例开始，看一看 Java 在没有外部内存接口的时候，是怎么支持本地内存的；然后，我们再看看外部内存接口能够给我们的代码带来什么样的变化。

阅读案例

在我们讨论代码性能的时候，内存的使用效率是一个绕不开的话题。像 TensorFlow、Ignite、Flink 以及 Netty 这样的类库，往往对性能有着偏执的追求。为了避免 Java 垃圾收集器不可预测的行为以及额外的性能开销，这些产品一般倾向于使用 JVM 之外的内存来存储和管理数据。这样的数据，就是我们常说的堆外数据（off-heap data）。

使用堆外存储最常用的办法，就是使用 ByteBuffer 这个类来分配直接存储空间（direct buffer）。JVM 虚拟机会尽最大努力直接在直接存储空间上执行 IO 操作，避免数据在本地和 JVM 之间的拷贝。

由于频繁的内存拷贝是性能的主要障碍之一。所以为了极致的性能，应用程序通常也会尽量避免内存的拷贝。理想的状况下，一份数据只需要一份内存空间，这就是我们常说的零拷贝。

下面的这段代码，就是用 ByteBuffer 这个类来分配直接存储空间的方法。

```
public static ByteBuffer allocateDirect(int capacity);
```

ByteBuffer 所在的 Java 包是 java.nio。从这个 Java 包的命名我们就能感受到，ByteBuffer 设计的初衷是用于非阻塞编程的。的确，ByteBuffer 是异步编程和非阻塞编程的核心类，几乎所有的 Java 异步模式或者非阻塞模式的代码，都要直接或者间接地使用 ByteBuffer 来管理数据。

非阻塞和异步编程模式的出现，起始于对于阻塞式文件描述符（File descriptor）（包括网络套接字）读取性能的不满。而诞生于 2002 年的 ByteBuffer，最初的设想也主要是用来解决当时文件描述符的读写性能的。所以，它的设计也不能跳脱出当时的客观需求。

如果站在现在的角度重新审视这个类的设计，我们会发现它主要有两个缺陷。

第一个缺陷是没有资源释放的接口。一旦一个 ByteBuffer 实例化，它占用了内存的释放，就会完全依赖 JVM 的垃圾回收机制。使用直接存储空间的应用，往往需要把所有潜在的性能都挤压出来。依赖于垃圾回收机制的资源回收方式，并不能满足像 Netty 这样的类库的理想需求。

第二个缺陷是存储空间尺寸的限制。ByteBuffer 的存储空间的大小，是使用 Java 的整数来表示的。所以，它的存储空间，最多只有 2G。这是一个无意带来的缺陷。在网络编程的环境下，这并不是一个问题。可是，超过 2G 的文件，一定会越来越多；2G 以上的文件，映射到 ByteBuffer 上的时候，就会出现文件过大的问题。而像 Memcahed 这样的分布式内存，也会让应用程序需要控制的内存超越 2G 的界限。

这两个缺陷，也是横隔在“零拷贝”这个理想路上的两个主要设计障碍。

对于第一个缺陷，我们还可以在 ByteBuffer 的基础上修改，并且保持这个类的优雅。但是第二个缺陷，由于 ByteBuffer 类里到处都在使用的整数类型，我们就很难找到办法既保持这个类的优雅，又能够突破存储空间的尺寸限制了。

一个合理的改进，就是重新建造一个轮子。这个新的轮子，就是外部内存接口。

外部内存接口

外部内存接口沿袭了 ByteBuffer 的设计思路，但是使用了全新的接口布局。我们先来看看使用外部内存接口的代码看起来是什么样子的。下面的这段代码，要分配一段外部内存，并且存放 4 个字母 A。

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment segment = MemorySegment.allocateNative(4, scope);
    for (int i = 0; i < 4; i++) {
```

```
        MemoryAccess.setByteAtOffset(segment, i, (byte)'A');
    }
}
```

现在，我们通过这个小例子，来看看外部内存接口的布局。

第一行的 ResourceScope 这个类，定义了内存资源的生命周期管理机制。这是一个实现了 AutoCloseable 的接口。我们就可以使用 try-with-resource 这样的语句，及时地释放掉它管理的内存了。这样的设计，就解决了 ByteBuffer 的第一个缺陷。

第二行的 MemorySegment 这个类，定义和模拟了一段连续的内存区域。第三行的 MemoryAccess 这个类，定义了可以对 MemorySegment 执行读写操作。在 ByteBuffer 的设计里，内存的表达和操作，是在 ByteBuffer 这一个类里完成的。在外部内存接口的设计里，把对象表达和对象的操作，拆分成了两个类。这两类的寻址数据类型，使用的是长整形（long）。这样，长整形的寻址类型，就解决了 ByteBuffer 的第二个缺陷。

超预期的演进

无论是在我们生活的现实世界里，还是在软件的虚拟世界里，只要我们超前迈出了第一步，后续的发展往往会超出我们的预料。外部内存接口的出现，虽然还处在孵化期，也带来了远远超出预期的精彩局面。

在计算机的世界里，代码主要和两类计算资源打交道。一类是负责控制和运算的处理器；一类是临时存放运算数据的存储器。表现到编程语言的层面，就是函数和内存。函数之间的数据传递，也是用过内存的形式进行的。

现在，外部内存接口为我们提供了一个统一的内存操作接口。对应地，外部函数之间的数据传递问题也就有了思路。既然能够解决函数之间的数据传递问题，那么，不同语言间的函数调用能不能变得更简单、更有效率呢？

这个问题，就是我们下一次要讨论的内容。如果说，设计外部内存接口的最初动力是为了解决 ByteBuffer 的两个缺陷。那研发的持续推进，则给外部内存接口赋予了更大的责任和能量。

总结

好，到这里，我来做个小结。前面，我们讨论了 Java 的外部内存接口这个尚处于孵化阶段的新特性，对外部内存接口这个新特性有了一个初始的印象。

设计外部内存接口的最初动力，是为了解决 ByteBuffer 的两个缺陷。也就是 ByteBuffer 占用的资源不能及时释放，以及它的寻址空间太小这两个问题。但是外部内存接口的更大使命，是和外部函数接口联系在一起的。我们下一次再讨论这个更大的使命。

如果外部内存接口正式发布出来，现在使用 ByteBuffer 的类库（比如 Flink 和 Netty，甚至 JDK 本身），应该可以考虑切换到外部内存接口来获取性能的提升。

这一次学习的主要目的，就是让你对外部内存接口有一个基本的印象。由于外部内存接口尚处于孵化阶段，现在我们还不需要学习它的 API。只要知道 Java 有这个发展方向，能够了解 ByteBuffer 的这两个缺陷能够给你的程序带来的影响就足够了。

如果面试中聊到了 ByteBuffer，你应该可以聊一聊零拷贝，以及 ByteBuffer 的这两个缺陷，还有未来的 Java 要做的改进。

思考题

其实，今天的这个新特性，也是练习使用 JShell 快速学习新技术的一个好机会。我们在前面的讨论里，分析了下面的这段代码。为了方便你阅读，我把这段代码重新拷贝到下面了。

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment segment = MemorySegment.allocateNative(4, scope);
    for (int i = 0; i < 4; i++) {
        MemoryAccess.setByteAtOffset(segment, i, (byte)'A');
    }
}
```

虽然我们提到了使用 try-with-resource 这样的语句，可以及时地释放掉它管理的内存。但是，我们并没有验证这一说法。你能不能使用 JShell，快速地验证它的资源释放效果呢？需要注意的是，要想使用孵化期的 JDK 技术，需要在 JShell 里导入孵化期的 JDK 模块。就像下面的例子这样。

```
$ jshell --add-modules jdk.incubator.foreign -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
jshell> import jdk.incubator.foreign.*;
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在外部内存接口专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 memory/review/xuelei 的目录下面。

13 | 外部函数接口，能不能取代 Java 本地接口？

你好，我是范学雷。今天，我们一起来讨论 Java 的外部函数接口。

Java 的外部函数接口这个新特性，我写这篇文章的时候，还在孵化期，还没有发布预览版。由于孵化期的特性还不成熟，不同的版本之间的差异可能会很大。我建议你使用最新版本，现在来说就是 JDK 17 来体验孵化期的特性。

Java 的外部函数接口这个特性，有可能会是 Java 自诞生以来最重要的两个特性之一，它和外部内存接口一起，会极大地丰富 Java 语言的生态环境。提前了解一下这样的新特性，有助于我们思考现在的技术手段和未来的技术规划。

我们从阅读案例开始，来看一看 Java 的外部函数接口为什么可能会带来这么大的影响，以及它能够给我们的代码带来什么样的变化吧。

阅读案例

我们知道，像 Java 或者 Go 这样的通用编程语言，都需要和其他的编程语言或者环境打交道，比如操作系统或者 C 语言。Java 是通过 Java 本地接口（Java Native Interface，JNI）来支持这样的做法的。本地接口，拓展了一门编程语言的生存空间和适用范围。有了本地接口，就不用所有的事情都在这门编程语言内部实现了。

比如下面的代码，就是一个使用 Java 本地接口实现的“Hello, world!”的小例子。其中的 sayHello 这个方法，使用了修饰符 native，这表明它是一个本地的方法。

```
public class HelloWorld {
    static {
        System.loadLibrary("helloWorld");
    }
    public static void main(String[] args) {
        new HelloWorld().sayHello();
    }
    private native void sayHello();
}
```

这个本地方法，可以使用 C 语言来实现。然后呢，我们需要生成这个本地方法对应的 C 语言的头文件。

```
$ javac -h . HelloWorld.java
```

有了这个自动生成的头文件，我们就知道了 C 语言里这个方法的定义。然后，我们就能够使用 C 语言来实现这个方法了。

```
#include "jni.h"
#include "HelloWorld.h"
#include <stdio.h>
JNIEXPORT void JNICALL Java_HelloWorld_sayHello(JNIEnv *env, jobject jObj) {
    printf("Hello World!\n");
}
```

下一步，我们要把 C 语言的实现编译、链接放到它的动态库里。这时候，就要使用 C 语言的编译器了。

```
$ gcc -I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/darwin \
      -dynamiclib HelloWorld.c -o libhelloWorld.dylib
```

完成了这一步，我们就可以运行这个 Hello World 的本地实现了。

```
java -cp . -Djava.library.path=. HelloWorld
```

你看，一个简单的“Hello, world!”的本地接口实现，需要经历下面这些步骤：

1. 编写 Java 语言的代码（HelloWorld.java）；
2. 编译 Java 语言的代码（HelloWorld.class）；
3. 生成 C 语言的头文件（HelloWorld.h）；
4. 编写 C 语言的代码（HelloWorld.c）；
5. 编译、链接 C 语言的实现（libhelloWorld.dylib）；

6. 运行 Java 命令，获得结果。

其实，在 Java 本地接口的诸多问题中，像代码实现的过程不简洁这样的问题，还属于可以克服的小问题。

Java 本地接口面临的比较大的问题有两个。

一个是 C 语言编译、链接带来的问题，因为 Java 本地接口实现的动态库是平台相关的，所以就没有了 Java 语言“一次编译，到处运行”的跨平台优势；另一个问题是，因为逃脱了 JVM 的语言安全机制，JNI 本质上是不安全的。

Java 的外部函数接口，是 Java 语言的设计者试图解决这些问题的一个探索。

外部函数接口

Java 的外部函数接口是什么样子的呢？下面的代码，就是一个使用 Java 的外部函数接口实现的“Hello, world!”的小例子。我们来一起看看，Java 的外部函数接口是怎么工作的。

```
import java.lang.invoke.MethodType;
import jdk.incubator.foreign.*;
public class HelloWorld {
    public static void main(String[] args) throws Throwable {
        try (ResourceScope scope = ResourceScope.newConfinedScope()) {
            CLinker cLinker = CLinker.getInstance();
            MemorySegment helloWorld =
                    CLinker.toCString("Hello, world!\n", scope);
            MethodHandle cPrintf = cLinker.downcallHandle(
                    CLinker.systemLookup().lookup("printf").get(),
                    MethodType.methodType(int.class, MemoryAddress.class),
                    FunctionDescriptor.of(CLinker.C_INT, CLinker.C_POINTER));
            cPrintf.invoke(helloWorld.address());
        }
    }
}
```

在这段代码里，try-with-resource 语句里使用的 ResourceScope 这个类，定义了内存资源的生命周期管理机制。

第 8 行代码里的 CLinker，实现了 C 语言的应用程序二进制接口（Application Binary Interface，ABI）的调用规则。这个接口的对象，可以用来链接 C 语言实现的外部函数。

接下来，也就是第 12 行代码，我们使用 CLinker 的函数标志符（Symbol）查询功能，查找 C 语言定义的函数 printf。在 C 语言里，printf 这个函数的定义就像下面的代码描述的样子。

```
int printf(const char *restrict format, ...);
```

C 语言里，printf 函数的返回值是整型数据，接收的输入参数是一个可变长参数。如果我们要使用 C 语言打印“Hello, world!”，这个函数调用的形式就像下面的代码。

```
printf("Hello World!\n");
```

接下来的两行代码（第 13 行和第 14 行代码），就是要把这个调用形式，表达成 Java 语言外部函数接口的形式。这里使用了 JDK 7 引入的 MethodType，以及尚处于孵化期的 FunctionDescriptor。MethodType 定义了后面的 Java 代码必须遵守的调用规则。而 FunctionDescriptor 则描述了外部函数必须符合的规范。

好了，到这里，我们找到了 C 语言定义的函数 printf，规定了 Java 调用代码要遵守的规则，也有了外部函数的规范。调用一个外部函数需要的信息就都齐全了。接下来，我们生成一个 Java 语言的方法句柄（MethodHandle）（第 11 行），并且按照前面定义的 Java 调用规则，使用这个方法句柄（第 15 行），这样我们就能够访问 C 语言的 printf 函数了。

对比阅读案例里使用 JNI 实现的代码，使用外部函数接口的代码，不再需要编写 C 代码。当然，也不再需要编译、链接生成 C 的动态库了。所以，由动态库带来的平台相关的问题，也就不存在了。

提升的安全性

更大的惊喜，来自于外部函数接口在安全性方面的提升。

从根本上说，任何 Java 代码和本地代码之间的交互，都会损害 Java 平台的完整性。链接到预编译的 C 函数，本质上是不可靠的。Java 运行时，无法保证 C 函数的签名和 Java 代码的期望是匹配的。其中一些可能会导致 JVM 崩溃的错误，这在 Java 运行时无法阻止，Java 代码也没有办法捕获。

而使用 JNI 代码的本地代码则尤其危险。这样的代码，甚至可以访问 JDK 的内部，更改不可变数据的数值。允许本地代码绕过 Java 代码的安全机制，破坏了 Java 的安全性赖以存在的边界和假设。所以说，JNI 本质上是不安全的。

遗憾的是，这种破坏 Java 为台完整系的风险，对于应用程序开发人员和最终用户来说，几乎是无法察觉的。因为，随着系统的不断丰富，99%的代码来自于夹在 JDK 和应用程序之间的第三方、第四方、甚至第五方的类库里。

相比之下，大部分外部函数接口的设计则是安全的。一般来说，使用外部函数接口的代码，不会导致 JVM 的崩溃。也有一部分外部函数接口是不安全的，但是这种不安全性并没有到达 JNI 那样的严重性。可以说，使用外部函数接口的代码，是 Java 代码，因此也受到 Java 安全机制的约束。

JNI 退出的信号

当出现了一个更简单、更安全的方案后，原有的方案很难再有竞争力。外部函数接口正式发布后，JNI 的退出可能也就要提上议程了。

在外部函数接口的提案里，我们可以看到这样的描述：

JNI 机制是如此危险，以至于我们希望库在安全和不安全操作中都更喜欢纯 Java 的外部函数接口，以便我们可以在默认情况下及时全面禁用 JNI。这与使 Java 平台开箱即用、缺省安全的更广泛的 Java 路线图是一致的。

安全问题往往具有一票否决权，所以，JNI 的退出很可能比我们预期的还要快！

总结

好，到这里，我来做个小结。前面，我们讨论了 Java 的外部函数接口这个尚处于孵化阶段的新特性，对外部函数接口这个新特性有了一个初始的印象。外部内存接口和外部函数接口联系在一起，为我们提供了一个崭新的不同语言之间的协作方案。

如果外部函数接口正式发布出来，我们可能需要考虑切换到外部函数接口，逐步退出传统的、基于 JNI 的解决方案。

这一次学习的主要目的，就是让你对外部函数接口有一个基本的印象。由于外部函数接口尚处于孵化阶段，所以我们不需要学习它的 API。只要知道 Java 有这个发展方向，目前来说就足够了。

如果面试中聊到了 Java 的未来，你不妨聊一聊外部内存接口和外部函数接口，它们要解决的问题，以及能带来的变化。

思考题

其实，今天的这个新特性，也是练习使用 JShell 快速学习新技术的一个好机会。我们在前面的讨论里，分析了下面这段代码。为了方便你阅读，我把这段代码重新拷贝到下面了。

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    CLinker cLinker = CLinker.getInstance();
    MemorySegment helloWorld =
            CLinker.toCString("Hello, world!\n", scope);
    MethodHandle cPrintf = cLinker.downcallHandle(
            CLinker.systemLookup().lookup("printf").get(),
            MethodType.methodType(int.class, MemoryAddress.class),
            FunctionDescriptor.of(CLinker.C_INT, CLinker.C_POINTER));
    cPrintf.invoke(helloWorld.address());
```

```
}
```

你能不能找一个你熟悉的 C 语言标准函数，试着修改上面的代码，快速地验证一下外部函数接口能不能按照你的预期工作？

需要注意的是，要想使用孵化期的 JDK 技术，需要在 JShell 里导入孵化期的 JDK 模块。就像下面的例子这样。

```
$ jshell --add-modules jdk.incubator.foreign -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
jshell> import jdk.incubator.foreign.*;
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在外部函数接口专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 memory/review/xuelei 的目录下面。

14 | 禁止空指针，该怎么避免崩溃的空指针？

你好，我是范学雷。今天，我们讨论 Java 的空指针。

我们都知道空指针，它的发明者开玩笑似的，称它是一个价值 10 亿美元的错误；同时呢，他还称 C 语言的 get 方法是一个价值 100 亿美元的错误。空指针真的错得这么厉害吗？get 方法又有什么问题？我们能够在 Java 语言里改进或者消除空指针吗？

我们从阅读案例开始，来看一看该怎么理解这些问题，以及怎么降低这些问题的影响。

阅读案例

通常地，一个人的姓名包括两个部分，姓（Last Name）和名（First Name）。在有些文化里，也会使用中间名（Middle Name）。所以，我们通常可以使用姓、名、中间名这三个要素来标识一个人的姓名。用代码的形式表示出来，就是下面的代码这样。

```
public record FullName(String firstName,
        String middleName, String lastName) {
    // blank
}
```

中间名并不是必需的，因为有的人使用中间名，有的人不使用。现在我们假设，需要判断一个人的中间名是不是黛安（Diane）。这个判断的逻辑，可能就像下面的代码这样。

```
private static boolean hasMiddleName(
        FullName fullName, String middleName) {
    return fullName.middleName().equals(middleName);
}
```

这个判断的逻辑是没有问题的。但是它的代码实现，就存在没有校验空指针的错误。如果一个人不使用中间名，那么 FullName.middleName 这个方法的返回值就是一个空指针。如果一个对象是空指针，那么调用它的任何方法，都会抛出空指针异常（NullPointerException）。

我们可以试着使用 JDK 11 的 JShell，看一看空指针异常的异常信息是什么样子的。

```
$ jshell -v
|  Welcome to JShell -- Version 11.0.13
|  For an introduction type: /help intro
jshell> String a = null;
a ==> null
|  created variable a : String
jshell> a.equals("b");
|  Exception java.lang.NullPointerException
|        at (#2:1)
```

然后，我们再试试看 JDK 17 里，空指针异常信息是什么样的。

```
$ jshell -v
|  Welcome to JShell -- Version 17
|  For an introduction type: /help intro
jshell> String a = null;
a ==> null
|  created variable a : String
jshell> a.equals("b");
|          Exception      java.lang.NullPointerException:      Cannot      invoke
"String.equals(Object)" because "REPL.$JShell$11.a" is null
|        at (#2:1)
```

对比一下，我们可以看到，JDK 17 的异常信息里，包含了调用者（REPL.*JShell*11.a）和被调用者（String.equals(Object)）的信息；而 JDK 11 里，调用者的信息需要从调用堆栈里寻找，而且没有被调用者的信息。

这是空指针异常的一个小的改进。它简化了问题排查的流程，提高了问题排查的效率。

好的，我们再回到主题，看一看空指针异常到底有什么危害。按照我们前面讨论过的中间名

的逻辑，有的人不使用中间名。那么，如果一个对象的中间名是空值，也就意味着他没有中间名。可是，在上面的实现代码里，如果中间名是空值，hasMiddleName 抛出了空指针异常，而不是通过返回值来表示这个对象没有中间名。
这当然是一个错误。我们**需要检查返回值有没有可能是空指针，然后才能继续使用返回值。**这是一个 C 语言或者 Java 语言软件工程师需要掌握的基本常识。当然，这也是一个我们编码的时候，需要遵守的纪律。
检查返回值有没有可能是空指针需要额外的代码，而且不符合我们的思维习惯。下面的代码，我添加了空指针的检查，这就让它看起来就有点臃肿。这就是精准控制的代价。

```
private static boolean hasMiddleNameImplA(
        FullName fullName, String middleName) {
    if (fullName.middleName() != null) {
        return fullName.middleName().equals(middleName);
    }
    return middleName == null;
}
```

空指针的问题，其实是我们人类行为方式的一个反映。无论是纪律还是常识，如果没有配以强制性的手段，都没有办法获得 100%的执行。如果不能 100%地执行，一个危害就会从一个小小的局部，蔓延到一个庞大的系统。
今天的应用程序，我们几乎可以肯定地说，都是由很多小的部件组合起来的。其中，99%以上的部件，我们都不了解，甚至都不知道它们的存在。任何一个小的部件出了问题，都会蔓延开来，酝酿出一个更大的问题。
在 C 语言和 Java 语言里，存在着大量的空指针。不管我们怎么努力，也不管我们经验多么丰富，总是会时不时地就忘了检查空指针。而忘了检查这样的小错误，很可能就蔓延成严重的事故。所以，空指针发明者称它是一个价值 10 亿美元的错误。
那有什么办法能够降低空指针的负面影响呢？

避免空指针

降低空指针的负面影响的最重要的办法，就是不要产生空指针。没有空指针的代码，代码更简洁，风险也更小。
比如说，我们可以使用空字符串来替代字符串的空指针。如果用这种思路，我们就可以把阅读案例里 FullName 档案类，修改成不使用空指针的版本了。

```
public record FullName(String firstName,
        String middleName, String lastName) {
    public FullName(String firstName,
            String middleName, String lastName) {
        this.firstName = firstName == null ? "" : firstName;
        this.middleName = middleName == null ? "" : middleName;
        this.lastName = lastName == null ? "" : lastName;
    }
}
```

这样，我们就不用检查空指针了；因此，也就不用担心空指针带来的问题了。所以，代码的使用也就变得简洁了起来。

```
private static boolean hasMiddleName(
        FullName fullName, String middleName) {
    return fullName.middleName().equals(middleName);
}
```

在很多场景下，我们都可以使用空值来替代空指针，比如，空的字符串、空的集合。在 API 设计的时候，如果碰到了使用空指针的规范或者代码，我们要停下来想一想，有没有替代空指针的办法？如果能够避免空指针，我们的代码会更健壮，更容易维护。

强制性检查

不过，不是在所有的情况下我们都能够避免空指针的。如果空指针不能避免，降低空指针的负面影响的另外一个办法，就是在使用空指针的时候，执行强制性的检查。所谓强制性的检查，对于编程语言来说，指的是我们通常能够依赖的是编译器的能力，以及新的接口设计思路。

不尽人意的 Optional

在 JDK8 正式发布，而后在 JDK9 和 11 持续改进的 Optional 工具类是 JDK 试图降低空指针风险的一个尝试。

设计 Optional 的目的，是希望开发者能够先调用它的 Optional.isPresent 方法，然后再调用 Optional.get 方法获得目标对象。按照设计者的预期，这个 Optional 类的使用应该像下面的代码这样。

```
private static boolean hasMiddleName(
        FullName fullName, String middleName) {
    if (fullName.middleName().isPresent()) {
        return fullName.middleName().get().equals(middleName);
    }
    return middleName == null;
}
```

当然，我们还需要修改 FullName 的 API，就像下面的代码这样。

```
public final class FullName {
    // snipped
    public Optional<String> middleName() {
        return Optional.ofNullable(middleName);
    }
    // snipped
}
```

遗憾的是，我们也可以不按照预期的方式使用它，比如下面的代码，我们就没有调用 Optional.isPresent 方法，而是直接使用了 Optional.get 方法。这不在设计者的预期之内，但是这是合法的代码。

```
private static boolean hasMiddleName(FullName fullName, String middleName) {
    return fullName.middleName().get().equals(middleName);
}
```

如果Optional指代的对象不存在,或者是个空指针,Optional.get方法就会抛出NoSuchElementException 异常。和空指针异常一样，这个异常也是运行时异常。虽然这个异常的名字不再叫做空指针异常，但它实质上依然是空指针异常。当然，这个异常也具有和空指针异常相同的问题。

如果你对比一下使用空指针的代码和使用 Optional 类的代码,就会发现这两个类型的代码，不论是正确的使用方法还是错误的使用方法，它们在形式上是相似的。Optional 带来了不必要的复杂性，然而它并没有简化开发者的工作，也没有解决掉空指针的问题。

被寄予厚望的 Optional 的设计，不能尽如人意。

新特性带来的新希望

那么，对于空指针的检查，我们能不能借助编译器，让它变得更强硬一点呢？下面的例子，就是我们使用新特性来解决空指针问题的一个新的探索。

我们希望返回值的检查是强制性的。如果不检查，就没有办法得到返回值指代的真实对象。

实现的思路，就是使用封闭类和模式匹配。

首先呢，我们定义一个指代返回值的封闭类 Returned。为什么使用封闭类呢，因为封闭类的子类可查可数。可查可数，也就意味着我们可以有简单的模式匹配。

```
public sealed interface Returned<T> {
```

```
    Returned.Undefined UNDEFINED = new Undefined();
    record ReturnValue<T>(T returnValue) implements Returned {
    }
    record Undefined() implements Returned {
    }
}
```

然后呢，我们就可以使用 Returned 来表示返回值了。

```
public final class FullName {
    // snipped
    public Returned<String> middleName() {
        if (middleName == null) {
            return Returned.UNDEFINED;
        }
        return new Returned.ReturnValue<>(middleName);
    }
    // snipped
}
```

最后，我们来看看 Returned 是怎么使用的。

```
private static boolean hasMiddleName(FullName fullName, String middleName) {
    return switch (fullName.middleName()) {
        case Returned.Undefined undefined -> false;
        case Returned.ReturnValue rv -> {
            String returnedMiddleName = (String)rv.returnValue();
            yield returnedMiddleName.equals(middleName);
        }
    };
}
```

这种使用了封闭类和模式匹配的设计，极大地压缩了开发者的自由度，强制要求开发者的代码必须执行空指针的检查，只有这样才能编写下一步的代码。这种看似放弃了灵活性的设计，恰恰把开发者从低级易犯的错误中解救了出来。不论是对写代码的开发者，还是对读代码的开发者来说，这都是一件好事。

好事情的背后，往往都意味着一些妥协。比如说吧，使用空指针的代码，我们可以轻松地使用档案类；使用 Optional 和 Returned 的代码，我们就要重新回到传统的类上面来了。

无论档案类、封闭类还是模式匹配，对于 Java 来说，都还是新鲜的技术。要想让这些技术之间熟练配合，还需要一些这样或者那样的磨练，包括不停地改进，组合效应的新研究等。

总结

好，到这里，我来做个小结。前面，我们讨论了空指针带来的问题，以及降低空指针负面影响的一些办法。

总体来说，在我们的代码里，尽量不要产生空指针。没有空指针，也就没有了空指针的烦恼。

如果避免不了空指针，我们就要看看能不能执行强制性的检查。比如使用封闭类和模式匹配的组合形式，让编译器和接口设计帮助我们实施这种强制性。

如果不能实施强制性的检查，我们就要遵守空指针的编码纪律。也就是说，对于可能是空指针的变量，先检查后使用。

如果面试中聊到了空指针的问题，你可以聊一聊空指针的危害，以及我们这一次学习到的解决办法。

思考题

今天，我们使用封闭类和模式匹配来降低空指针危害的例子，有点像我们前面提到过的替代异常处理的错误码方案。其实，一个带有返回值的方法，通常要考虑三种情况：正常情况、

异常情况以及空指针。我们可以把空指针解读为正常情况，也可以解读为异常情况。
如果要在返回值这个封闭类里考虑进这三种情况，我们该怎么设计这个封闭类以及它的许可类呢？这是我们这一次的思考题。
为了方便你阅读，我把我们这次讨论用到的 Returned 的实现代码拷贝到了下面。你可以在这个基础上修改。

```
public sealed interface Returned<T> {
    Returned.Undefined UNDEFINED = new Undefined();
    record ReturnValue<T>(T returnValue) implements Returned {
    }
    record Undefined() implements Returned {
    }
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！
注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在外部函数接口专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 nullp/review/xuelei 的目录下面。

15 | 现代密码：你用的加密算法过时了吗？

你好，我是范学雷。今天，我想和你聊聊 JDK 里的密码学算法相关的问题。

Java 语言安全的基础，主要有两块内容。一块是 Java 语言的安全设计，比如字节码的校验，内存保护机制等等；另外一块是 Java 平台的保护机制，比如签名的类库，资源的认证授权等等。而 Java 平台的保护机制，是建立在密码学的基础之上的。

这一次的讨论，我们从故事开始，来看看现在我们应该采用的密码学的技术，以及应该抛弃的密码学技术。

阅读案例

1976 年，是现代密码学的奠基之年。这一年，Diffie－Hellman 密钥交换协议公开发表。这是由 Ralph Merkle 构思并以 Whitfield Diffie 和 Martin Hellman 命名的第一个公钥协议。这是最早为公众所知的，提出公钥和私钥思想的著作。从这一年开始，在非安全通道上建立安全通信的想法，有了理论上的依据；现代互联网的安全，也终于有了稳固的基石。

Diffie－Hellman 密钥交换协议的论文，为密码学家展示了一个全新的大陆。有了这个方向的指引，接下来很快就有了更多的脚步踏出了新的道路。1977 年，受 Diffie－Hellman 密钥交换协议的启发，Ron Rivest、Adi Shamir 和 Leonard Adleman 公开发表了基于公开密钥的电子签名算法，也就是 RSA 算法。从此以后，要在非安全通道上识别身份、建立信任的想法，也有了理论上的依据。

至此，加上传统的加密技术，解决信息安全基本问题的三大技术就已经集结完成了。随着互联网的发展，这些技术大放异彩，成为了互联网基础设施最终的环节之一。

后面的事情也就顺理成章了。1982 年，Ron Rivest、Adi Shamir 和 Leonard Adleman 成立了 RSA 公司，公司主要提供基于 RSA 算法的产品和服务。1991 年，RSA 公司推出了 RSA 大会以及 RSA 算法的分解挑战。2006 年 RSA 公司被 EMC 收购，收购价达到 21 亿美元。2007 年，RSA 算法分解挑战终止。而 RSA 大会，则发展成了信息安全领域最富盛名的的大会。

为什么 RSA 分解挑战终止了呢？按照官方的声明，因为：“现在业界对常见对称密钥和公钥算法的密码分析强度有了更深入的了解，这些挑战不再活跃。”

那分解挑战的成果是什么样子的呢？我想你也一定感兴趣。

1991 年 3 月 18 日，RSA 公司推出 RSA 算法的分解挑战。不到两个星期，也就是 1991 年的 4 月 1 日，330 位的 RSA 密钥被破解。随后，更高强度的 RSA 算法被破解。其中，768 位的 RSA 密钥在 2009 年被破解，这是一个 RSA 命运的分水岭。从此以后，小于或者等于 1024 位的 RSA 密钥，都被认为是不安全的密钥。现在的 RSA 算法，应该是用至少 2048 位的密钥。

对 RSA 算法的破解研究，并不仅仅局限于因式分解这样的纯计算游戏。比如说，早在 1998 年，就有密码学家发现了对 RSA 算法进行旁路攻击的办法。现在，如果是用测时攻击（timing attacks），对于 1024 位的密钥，破解传统的 RSA 实现也就是分分钟钟的事情。

虽然，我们可以通过复杂的 RSA 实现来化解这样的攻击。但是，复杂的实现，意味着性能的损失以及维护的困难。到这里，我们已经可以依稀地听到 RSA 算法要告别历史舞台的声音了。

其实，任何一个密码学的算法，都有它的生命周期。从看似完美的问世，到实际破落的境地，也就是数十年的时间。

看向未来

但是，我们的隐私数据却需要上百年，甚至是永远的保护。有生命周期的算法，似乎满足不了这样的要求。密码学要始终看向未来。如果站在十年后看现在，我们怎么能保证万无一失呢？

十多年后，量子计算机大概率就能够问世了。而量子计算机的计算能力是非常恐怖的。现在我们常见的非对称密码算法所能提供的计算强度，在量子计算时代，也许就像是小孩子的玩具一样脆弱。所以，密码学家和各种组织都在紧锣密鼓地遴选“后量子时代”的非对称密码算法。

显然，我们不能等到“后量子时代”的非对称密码算法问世以后，再来保护我们的隐私数据。现在我们就需要这样的保护。而这其中最重要的方案，就是使用前向保密（Forward Secrecy）

的安全协议。前向保密也就意味着，即使未来我们反复使用的密钥被破解，我们的数据依然能够得到保护。如果你想了解更多的关于前向保密的细节，请参考我在另外一个专栏里的讨论《量子时代，你准备好了吗？》。
在 Java 的设计和实现里，前向保密是 JDK 缺省的选择。这是 JDK 8 之后，JDK 做的一个重要的安全策略调整。这个调整，涉及到的大都是 JDK 实现的小细节，比如缺省 JDK 升级到 TLS 1.3 这样的变动。
JDK 的安全是 Java 语言的头等大事。所以，JDK 的安全改进一般情况下，都会向后移植，直到我们没有能力移植为止。前向保密的策略，也已经向后移植，进入到 JDK 8 了。

关注变化

既然密码学的算法有生命周期，我们就需要了解这个生命周期，及时地停止使用危险的、过期的算法。那么，哪些密码算法如今已经过期或者存在安全隐患？我们又能从哪里找到这方面最新的信息呢？
JDK 8 之后，Java 安全策略的另外一个重要的调整，就是公开发布 JDK 的密码路线图。在这个路线图里，JDK 会声明哪些密钥算法是危险的，哪些是过期的，以及 JDK 根据密码学的进展作出的变动。
如果你的产品或者代码涉及到了密码相关的内容，你就要密切关注这个路线图的更新，及时地调整产品里涉及到密码算法了。
另外，密钥算法的废弃，总是会带来这样或者那样的兼容性问题。当安全性和兼容性相遇的时候，我们应该毫不犹豫地选择选择安全性，及时解决掉兼容性问题。安全性问题，时间上千万不要拖。软件系统的漏洞，一般情况下，攻击者知道的比你还要早。我们拖拉的每一秒钟，都是留给攻击者的时间窗口。

应该抛弃的算法

下面我罗列了一些曾经流行的，JDK 支持的，但是我们不应该使用的密码学算法或者协议。继续使用这些算法，会给你的系统带来难以预料的灾难。而且，使用的系统也很容易成为黑客攻击的目标。

- MD2
- MD5
- SHA-1
- DES
- 3DES
- RC4
- SSL 3.0
- TLS 1.0
- TLS 1.1
- 密钥小于 1024 位的 RSA 算法
- 密钥小于 1024 位的 DSA 算法
- 密钥小于 1024 位的 Diffie-Hellman 算法
- 密钥小于 256 位的 EC 算法

应该退役的算法

下面我罗列了一些曾经流行的，JDK 支持的，我们可以使用，但是应该尽快替换掉的算法。这些算法，目前来看还是安全的，但是已经处于危险的边缘了。如果你的系统计划运行五年以上，这些算法的安全性值得担忧。

- 密钥大于 1024 位小于 2048 位的 RSA 算法。
- 密钥大于 1024 位小于 2048 位的 DSA 算法。
- 密钥大于 1024 位小于 2048 位的 Diffie-Hellman 算法。
- RSA 签名算法
- 基于 RSA 的密钥交换算法

- 128 位的 AES 算法

推荐使用的算法

下面我罗列了一些现在流行的，JDK 支持的，我们推荐使用的密码学算法。这些算法，目前看还没有发现值得重视的安全问题，是可以信任的算法。如果一个系统计划运行五年以上，你应该使用这些算法。

- 256 位的 AES 算法
- SHA-256、SHA-512 单向散列函数
- RSASSA-PSS 签名算法
- X25519/X448 密钥交换算法
- EdDSA 签名算法

我们前面提到过，安全改进一般都会向后移植，但是也有我们没有能力移植的例子。上面提到的推荐使用的算法中，JDK 8 不支持 X25519/X488 密钥交换算法，也不支持 EdDSA 签名算法。一个最重要的原因，就是这些算法需要使用新的公开接口。

一般情况下，小版本的 JDK 升级，不能变更公开接口。这就让 JDK 8 有了安全上的短板。目前看，这个短板还不足以构成安全威胁。但是停留在 JDK 8 意味着我们放弃了更好的密码算法，包括安全性的提高和性能的提升。

我上面列举的算法，大部分开发者应该接触不到。因为，它们是 Java 语言和 Java 平台的一部分，是计算机基础设施的一部分。我们天天使用它们，但是没有多少人意识到它们的存在。如果你需要使用密码，比如签名 Java 包，或者使用数字证书，请留意这些数字内容使用的密码算法，尽量使用推荐的算法，千万不要使用已经抛弃的算法。

总结

好，到这里，我来做个小结。通过今天的讨论，我们知道，任何一个密码学的算法，都有它的生命周期。所以，我们要能够管理它们的生命周期。反映到代码里，就是要使用前向保密的安全协议以及当前推荐的算法；及时替换掉过期的算法。

对于 JDK 的开发者来说，我们要关注 JDK 的密码路线图，了解 JDK 根据密码学的进展作出的变动，及时解决自己代码里的兼容性问题。

如果面试中聊到了密码学算法的问题，你可以聊聊前向保密，以及我们推荐的密码学算法。

思考题

今天的思考题，是一个拓展阅读。在上面推荐的算法里，除了 AES 算法之外，其他的三个算法，如果不是关注密码学进展的话，你可能都没有听说过。密码学进展很快，十多年前的主流算法，在今天几乎都要进入退休的年龄了。我们也要随时更新对密码学基本现状的认识。

如果有时间，你可以去搜索一下 RSASSA-PSS 签名算法，X25519 密钥交换算法以及 EdDSA 签名算法的相关介绍。不需要了解技术细节，知道大致是怎么回事就行。

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的拓展阅读内容。我们下节课见！

16 | 改进的废弃，怎么避免使用废弃的特性？

你好，我是范学雷。今天，我们讨论 Java 公开接口的废弃。

像所有的事物一样，公开接口也有生命周期。要废弃那些被广泛使用的、或者还有人使用的公开接口，是一个非常痛苦的过程。该怎么废弃一个公开接口，该怎么减少废弃接口对我们的影响呢？这是这一次我们要讨论的话题。

我们先来看看阅读案例。

阅读案例

在 JDK 中，一个公开的接口，可能会因为多种多样的原因被废弃。比如说，这个接口的设计是危险的，或者有了更新的、更好的替代接口。不管是什么原因，废弃接口的使用者们都需要尽快迁移代码，转换到替代方案上来。

在 JDK 中，公开接口的废弃需要使用两种不同的机制，也就是“Deprecated”注解（annotation）和“Deprecated”文档标记（JavaDoc tag）。

Deprecated 的注解会编译到类文件里，并且可以在运行时查验。这就允许像 javac 这样的工具检测和标记已废弃接口的使用情况了。

Deprecated 文档标记用于描述废弃接口的文档中。除了标记接口的废弃状态之外，一般情况下，我们还要描述废弃的原因和替代的方案。

下面的这段代码，就是使用 Java 注解和文档标记来废弃一个公开接口的例子。

```
public sealed abstract class Digest {
    /**
     * -- snipped
     *
     * @deprecated This method is not performance friendly. Use
     *             {@link #digest(byte[], byte[]) instead.
     */
    @Deprecated
    public abstract byte[] digest(byte[] message);
    // snipped
    public void digest(byte[] message, byte[] digestValue) {
        // snipped
    }
}
```

如果一段程序使用了废弃接口，编译的时候，就会提出警告。但是，有很多编译环境的配置，把编译警告看作是编译错误。为了解决这样的问题，JDK 还提供了“消除使用废弃接口的编译警告”的选项。也就是 SuppressWarnings 注解。

```
@SuppressWarnings("deprecation")
public static void main(String[] args) {
    try {
        Digest.of("SHA-256")
              .digest("Hello, world!".getBytes());
    } catch (NoSuchAlgorithmException ex) {
        // ignore
    }
}
```

公开接口的废弃机制，是在 JDK 1.5 的时候发布的。这种机制像一座设计者和使用者之间的沟通桥梁，减轻了双方定义或者使用废弃接口的痛苦。

遗憾的是，直到现在，公开接口的废弃，依然是一个复杂、痛苦的过程。一个公开的接口，从声明废弃，到彻底删除是一个漫长的过程。在 JDK 中，还存在着大量废弃了 20 多年都无法删除的公开接口。

为什么删除废弃的公开接口这么困难呢？如果从废弃机制本身的角度来思考，下面几个问题

延迟了废弃接口使用者的迁移意愿和努力。
第一个问题，也是最重要的问题，就是 SuppressWarnings 注解的使用。SuppressWarnings 注解的本意是消除编译警告，保持向后的编译兼容性。可是一旦编译警告消除，SuppressWarnings 注解也就抵消了 Deprecated 注解的功效。代码的维护者一旦使用了 SuppressWarnings 注解，就很难再有更合适的工具，让自己知道还在使用的废弃接口有哪些了。不知道，当然就不会有行动。
第二个问题，就是废弃接口的使用者并不担心使用废弃接口。虽然我们都知道不应该使用废弃的接口，但是因为一些人认为没有紧急迁移的必要性，也不急着制定代码迁移的时间表，所以倾向于先使用 SuppressWarnings 注解把编译警告消除了，以后再说迁移的事情。然后，就掉入了第一个问题的陷阱。
第三个问题，就是废弃接口的使用者并不知道接口废弃了多久。在接口使用者的眼里，废弃了十年，和废弃了一年的接口，没有什么区别。可是，在接口维护者的眼里，废弃了十年的接口，应该可以放心地删除了。然而，使用者并没有感知到这样的区别。没有感知，当然也就没有急迫感了。
一旦一个接口被声明为废弃，它的问题也就再难进入接口维护者的任务列表里了。所以，这个接口的实现可能充满了风险和错误。于是局面就变成了，接口维护者难以删除废弃的接口，接口的使用者又不能获得必要的提示，这种情况实在有点尴尬。

改进的废弃

上面这些问题，在 JDK 9 的接口废弃机制里有了重大的改进。
第一个改进是添加了一个新的工具，jdeprscan。有了这个工具，就可以扫描编译好的 Java 类或者包，看看有没有使用废弃的接口了。即使代码使用了 SuppressWarnings 注解，jdeprscan 的结果也不受影响。这个工具解决了我们在阅读案例里提到的第一个问题。
另外，如果我们使用第三方的类库，或者已经编译好的类库，发现对废弃接口的依赖关系很重要。如果将来废弃接口被删除，使用废弃接口的类库将不能正常运行。而 jdeprscan 允许我们在使用一个类库之前进行废弃依赖关系检查，提前做好风险的评估。
第二个改进是给 Deprecated 注解增加了一个“forRemoval”的属性。如果这个属性设置为“true”，那就表示这个废弃接口的删除已经提上日程了。两到三个版本之后，这个废弃的接口就会被删除。这样的改进，强调了代码迁移的紧急性，它给了使用者一个明确的提示。这个改进，解决了我们在阅读案例里提到的第二个问题。
第三个改进是给 Deprecated 注解增加了一个“since”的属性。这个属性会说明这个接口是在哪一个版本废弃的。如果我们发现一个接口已经废弃了三年以上，就要考虑尽最大努力进行代码迁移了。这样的改进，给了废弃接口的使用者一个时间上的概念，也方便开发者安排代码迁移的时间表。这个改进，解决了我们在阅读案例里提到的第三个问题。
下面的这段代码，就是一个使用了这两种属性的例子。

```
public sealed abstract class Digest {
    /**
     * -- snipped
     *
     * @deprecated This method is not performance friendly. Use
     *             {@link #digest(byte[], byte[]) instead.
     */
    @Deprecated(since = "1.4", forRemoval = true)
    public abstract byte[] digest(byte[] message);
    // snipped
    public void digest(byte[] message, byte[] digestValue) {
        // snipped
    }
}
```

如果在 Deprecated 注解里新加入“forRemoval”属性，并且设置为“true”，那么以前的 SuppressWarnings 就会失去效果。要想消除掉编译警告，我们需要使用新的选项。就像下面

的例子这样。

```
@SuppressWarnings("removal")
public static void main(String[] args) {
    try {
        Digest.of("SHA-256")
              .digest("Hello, world!".getBytes());
    } catch (NoSuchAlgorithmException ex) {
        // ignore
    }
}
```

当一个废弃接口的删除提上日程的时候，添加“forRemoval”属性让我们又有一次机会在代码编译的时候，重新审视还在使用的废弃接口了。

废弃三部曲

有了 JDK 9 的废弃改进，我们就能够看到接口废弃的一般过程了。
第一步，废弃一个接口，标明废弃的版本号，并且描述替代方案；
第二步，添加“forRemoval”属性，把删除的计划提上日程；
第三步，删除废弃的接口。
对于接口的使用者，我们应该尽量在第一步就做好代码的迁移；如果我们不能在第一步完成迁移，当看到第二步的信号时，我们也要把代码迁移的工作提高优先级，以免影响后续的版本升级。
对于接口的维护者，我们需要尽量按照这个过程退役一个接口，给接口的使用者充分的时间和信息，让他们能够完成代码的迁移。

总结

好，到这里，我来做个小结。刚才，我们讲了接口废弃的现实问题，以及接口废弃的三部曲。
总体来说，我们要管理好废弃的接口。接口的废弃要遵守程序，有序推进；代码的迁移要做好计划，尽快完成。
另外，我们要使用好 jdeprscan 这个新的工具。在使用一个类库之前，要有意识地进行废弃依赖关系检查，提前做好代码风险的评估。
如果面试中聊到了接口废弃的问题，你可以聊一聊接口废弃的三部曲，以及每一步应该使用的 Java 注解形式。

思考题

今天的思考题，我们来练习一下接口废弃的过程。前面，我们练习过表示形状的封闭类。假设要废弃表示正方形的许可类，我们该怎么做呢？代码该怎么改动呢？
为了方便你阅读，我把表示形状的封闭类的代码拷贝到了下面。请再一次阅读“废弃三部曲”这一小节，然后试着修改下面的代码。

```
package co.ivi.jus.retire.review.xuelei;
public abstract sealed class Shape {
    public final String id;
    public Shape(String id) {
        this.id = id;
    }
    public abstract double area();
    public static final class Circle extends Shape {
        public final double radius;
        public Circle(String id, double radius) {
            super(id);
            this.radius = radius;
```

```
        }
        @Override
        public double area() {
            return Math.PI * radius * radius;
        }
    }
    public static final class Square extends Shape {
        public final double side;
        public Square(String id, double side) {
            super(id);
            this.side = side;
        }
        @Override
        public double area() {
            return side * side;
        }
    }
    // Here is your code for Rectangle.
    // Here is the test for circle.
    public static boolean isCircle(Shape shape) {
        // Here goes your update.
        return (shape instanceof Circle);
    }
    // Here is the code to run your test.
    public static void main(String[] args) {
        // Here is your code.
    }
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从 GitHub 下载，你可以通过修改 GitHub 上 review template 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在外部函数接口专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 retire/review/xuelei 的目录下面。

17 | 模块系统：为什么 Java 需要模块化？

你好，我是范学雷。今天，我们一起来讨论 Java 平台模块系统（Java Platform Module System，JPMS）。

Java 平台模块系统是在 JDK 9 正式发布的。为了沟通起来方便，我们有时候就直接简称为 Java 模块。Java 平台模块系统，可以说是自 Java 诞生以来最重要的新软件工程技术了。模块化可以帮助各级开发人员在构建、维护和演进软件系统时提高工作效率。软件系统规模越大，我们越需要这样的工程技术。

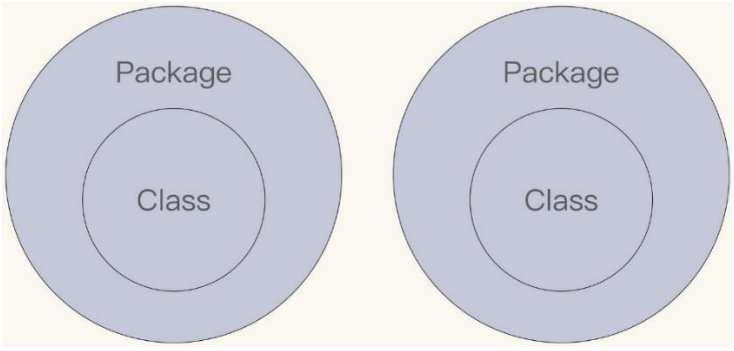
实现 Java 平台的模块化是具有挑战性的，Java 模块系统（Module System）的最初设想可以追溯到 2005 年的 Java 7，但是最后的发布，是在 2017 年的 JDK 9。它的设计和实现，花了十多年时间，我们可以想象它的复杂性。

令人满意的是，Java 平台模块系统最终呈现的结果是简单、直观的。我们并不需要太长的时间，就能快速掌握这一技术。

我们先来了解 Java 模块化背后的动力，和它能够带来的工程效率提升。除非特别说明，这一次的讨论，说的都是 JDK 8 及以前的版本的事情。下一次，我们再来讨论 JDK 9 之后，我们应该怎么使用 Java 平台模块系统。

缺失的访问控制

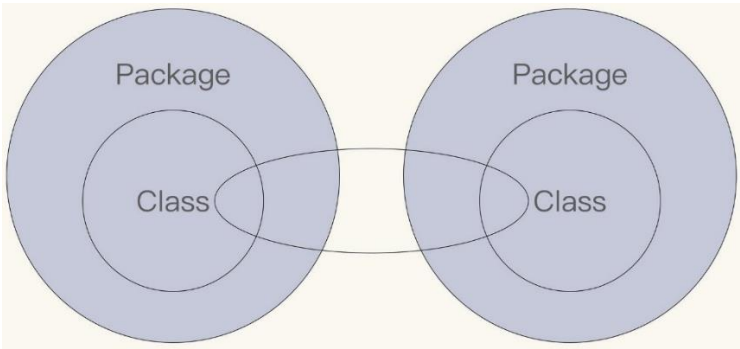
我们都清楚并且能够熟练地使用 Java 的访问修饰符。这样的访问修饰符一共有三个：public、protected，以及 private。如果什么修饰符都不使用，那就是缺省的访问修饰符，这也算是一种访问控制。所以，Java 语言一共定义了四种类型的访问控制。



private 访问修饰符修饰的对象，在同一个类里是可见的；缺省访问修饰符修饰的对象，在同一个 Java 包里是可见的；pubic 访问修饰符修饰的对象，在不同的 Java 包里也是可见的。

有了 private、public 和缺省的访问修饰符，看起来我们已经能解决大部分的问题了。不过这里还欠缺了重要的一环。

当我们设计对象的扩展能力的时候，我们可能期待扩展的子类处于不同的 Java 包里。但是，其中的一些数据信息，子类需要访问，但又因为它们是接口实现的细节，不应该对外公开。所以这时候，就需要一个能够穿越 Java 包，传递到子类的访问修饰符。这个访问修饰符就是 protected。protected 访问修饰符，在 Java 包之间打通了一条继承类之间的私密通道。



我们可以用下面这张表来总结 Java 语言访问修饰符的控制区域。

访问修饰符	当前类	当前包	其他包的子类	其他包
private	可见			
default	可见	可见		
protected	可见	可见	可见	
public	可见	可见	可见	可见

从这个列表看，Java 语言访问修饰符似乎覆盖了所有的可能性，这好像是一个完备的定义。遗憾的是，Java 语言访问修饰符遗漏了很重要的一种情况，那就是 Java 包之间的关系。Java 包之间的关系，并不是要么全开放，要么全封闭这么简单。

类似于继承类之间的私密通道，Java 包之间也有这种类似私密通道的需求。比如说，我们在 JDK 的标准类库里，可以看到像 java.net 这样的放置公开接口的包，也可以看到像 sun.net 这样的放置实现代码的包。

公开接口，当然需要定义能够广泛使用的类，比如 public 修饰的 Socket 类。

```
package java.net;
public class Socket implements java.io.Closeable {
    // snipped
}
```

让人遗憾的是，放置公开接口实现代码的包里，也需要定义 public 的类。这就让本来只应该由某个公开接口独立使用的代码变得所有人都可以使用了。

比如说，用来实现公开接口 Socket 类的 PlatformSocketImpl 类，就是一个使用 public 修饰的类。

```
package sun.net;
public interface PlatformSocketImpl {
    // snipped
}
```

虽然 PlatformSocketImpl 是一个 public 修饰的类，但是我们并不期望所有的开发者都能够使用它。这是一个用来支持公开接口 Socket 实现的类。除了实现公开接口 Socket 的代码之外，它不应该被任何其他的代码和开发者调用。

然而，PlatformSocketImpl 是一个 public 修饰的类。这也就意味着任何代码和开发者都可以使用它。这显然是不符合设计者的预期的。

在 JDK 8 及以前的版本里，一个对象在两个包之间的访问控制，要么是全封闭的，要么是全开放的。所以，JDK 9 之前的 Java 世界里，它的设计者没有办法强制性地设定 PlatformSocketImpl，给出一个恰当的访问控制范围。

两个包之间，没有一个定向的私密通道。换句话说，JDK 9 之前的 Java 语言没有描述和定义包之间的依赖关系，也没有描述和定义基于**包的依赖关系**的访问控制规则。这是一个缺失的访问控制。

这种缺失的关系，带来了严重的后果。

松散的使用合约

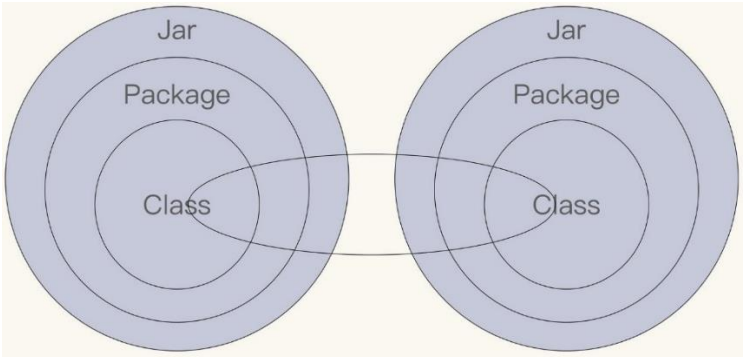
按照 JDK 的期望，一个开发者应该只使用公开接口（比如上面提到的 Socket 类），而不能使用实现细节的内部接口（比如上面提到的 PlatformSocketImpl 接口）。无论是公开接口，还是内部接口，都可以使用 public 修饰符。那么，该怎么判断一个接口是公开接口，还是内部接口呢？

解决的办法，是依靠 Java 接口的使用规范这样的纪律性合约，而不是依靠编译器强制性的检查。**在 JDK 里，以 java 或者 javax 命名开头的 Java 包，是公开的接口；其他的包是内**

部的接口。按照 Java 接口的使用规范，一个开发者应该只使用公开的接口，而不能使用内部的接口。不过，这是一个完全依靠自觉的纪律性约束；Java 的编译器和解释器，并不会禁止开发者使用内部接口。
内部接口的维护者可能会随时修改甚至删除内部的接口。使用内部接口的代码，它的兼容性是不受保护的。这是一个危险的依赖，应该被禁止。
遗憾的是，这种纪律性合约是松散的，它很难禁止开发者使用内部接口。我们能够看到大量的、没有遵守内部接口使用合约的应用程序。内部接口的不合规使用，也成了 Java 版本升级的重要障碍之一。松散的纪律性合约既伤害了内部接口的设计者，也伤害了它的使用者和最终用户。
我们前面提到过，Java 平台模块化的设计和实现，花了十多年时间。而内部接口的不合规使用，就是这项工作复杂性的主要来源。
我们认为，**如果一件事情应该禁止，那么最好的办法就是让这件事情没有办法发生；而不是警告发生以后的的后果，或者依靠事后的惩罚。**
那怎么能够更有效的限制内部接口的使用，提高 Java 语言的可维护能力呢？这是 Java 语言要解决的一个重要问题。

手工的依赖管理

Java 语言没有描述和定义包之间的依赖关系，这就直接增加了应用程序部署的复杂性。
公开接口的定义和实现，并不一定是放置在同一个 Java 包。比如上面我们提到的 Socket 类和 PlatformSocketImpl 类就位于不同的 Java 包。
因为通常情况下，我们使用 Jar 文件来分发和部署 Java 应用，所以，公开接口的定义和实现，也不一定是放置在同一个 Jar 文件里。比如一个加密算法的实现，它的公开接口一般是由 JDK 定义的；但是它的实现，可能是由一个第三方的类库完成的。



Java 的编译器只需要知道公开接口的规范，并不会去检查实现的代码，也不会去链接实现的代码。可是，Java 在运行时，不仅需要知道公开接口的字节码，还需要知道实现的字节码。这就导致了编译和运行的脱节。一个能通过编译的代码，运行时可能也会遇到找不到实现代码的错误。
而且，Java 的编译器不会在字节码里添加类和包的依赖关系。我们在编译期设置的依赖类库，在运行期还需要重新设置。编译器环节和运行环节是由两个独立的 Java 命令执行的，所以这种依赖关系也不会从编译期传导到运行期。
由于依赖关系的缺失，Java 运行的时候，可能不会完全按照它的设计者的意图工作。这就给 Java 应用的部署带来很多问题。这一类的问题如此让人讨厌，以至于它还有一个让人亲切不起来的外号，Jar 地狱。
为了解决依赖关系的缺失带来的种种问题，业界现在也有了一些解决方案，比如使用 Maven 和 Gradle 来管理项目。然而，由于 Java 没有内在的依赖关系规范，现有的解决方案也就只能依赖人工。依赖人工的手段，也就意味着效率和质量上的潜在风险。

缓慢的实现加载

Java 语言没有描述和定义包之间的依赖关系，还直接影响了 Java 应用程序的启动效率。
我们都知道像 Spring 这样的框架，它缓慢的启动一直都是一个大问题。影响 Java 应用启动速度的最主要原因，就是类的加载。导致类加载缓慢的一个重要原因，就是很难查找到要加

载的类的实现代码。
假设我们设置的 class path 里有很多 Jar 文件，对于一个给定名称的 class，Java 怎么才能找到实现这个类的字节码呢？由于 Jar 文件里没有描述类的依赖关系的内容，Java 的类加载器只能线性地搜索 class path 下的 Jar 文件，直到发现了给定的类和方法。这种线性搜索方法当然不是高效的。class path 下的 Jar 文件越多，类加载得就越慢。
更糟糕的是，这种线性搜索的方式，还带来了不可预测的副作用。其中，影子类（Shadowing classes）和版本冲突是最常见的两个副作用。
因为在不同的 Jar 文件里，可能会存在两个有着相同命名，但是行为不同的类。给定了类的名称，哪一个 Jar 文件里的类会被首先加载呢？这依赖于 Jar 文件被检索的顺序。在不同的运行空间，class path 的设置可能是不同的，Jar 文件被检索的顺序可能也是不同的；所以，实际加载的类就有可能是不同的，最终的运行结果当然也是不同的。这样的问题，可能会导致难以预料的结果，而且非常难以排查。
如果一个类的不同版本的实现都出现在了 class path 里，也会出现类似的问题。

新的思路

我们可以看到，这些问题的根源，都来自于 Java 语言没有描述和定义包之间的依赖关系。
那么，我们能不能通过扩展访问修饰符来解决这些问题呢？
答案可能没有这么简单。多个节点之间的依赖关系描述，需要使用的是数学逻辑图。而单个的修饰符，不足以表达复杂的图的逻辑。
另外，Jar 文件虽然是 Java 语言的一种必不可少的代码组织方式，但是它却不是由我们编写的代码直接控制的。我们编写的代码，可以控制 Java 包，可以控制 Java 类，但是管不了 Jar 文件的内容和形式。
所以，要解决这些问题，需要新的思路。而 JDK 9 发布的 Java 平台模块系统，就是解决这些问题的一个尝试。

总结

好，到这里，我来做个小结。前面，我们讨论了 JDK 8 及其以前版本的访问控制缺陷，以及由此带来的种种问题。
总体来说，Java 语言没有描述和定义包之间的依赖关系。这个缺失，导致了无法有效地封闭实现的细节；无法有效地管理应用的部署；无法精准地控制类的检索和加载，也影响了应用启动的效率。
那能不能在 Java 语言里添加进来这个缺失的关系呢？该怎么做？这是我们下一次要讨论的话题。
如果面试的时候，讨论到了 Java 的访问修饰符，你不妨聊一聊这个缺失的环节，以及 Jar 地狱这样的问题。我相信，这是一个有意思、有深度的话题。

思考题

在前面的讨论中，我们提到了使用 Maven 或者 Gradle 来管理项目，以此解决依赖关系的缺失。但是，我们并没有展开讨论这些问题是怎么解决的。
如果熟悉 Maven、Gradle 或者类似的工具的话，你能不能聊一聊，这样的工具，是怎么解决依赖关系缺失这样的问题的？它们哪些地方做得比较好，哪些地方还有待改进？这样的讨论，也许有助于我们深入了解我们这一次讨论到的问题。
欢迎你在留言区留言、讨论，分享你的阅读体验以及你对 Maven 或者 Gradle 的了解。我们下节课见！

18 | 模块系统：怎么模块化你的应用程序？

你好，我是范学雷。今天，我们继续讨论 Java 平台模块系统（Java Platform Module System，JPMS）。

Java 平台模块系统是在 JDK 9 正式发布的。在上一讲我们也说过，这项重要的技术从萌芽到诞生，花费了十多年的时间，堪称 Java 出现以来最重要的新软件工程技术。

模块化可以帮助各级开发人员在构建、维护和演进软件系统时提高工作效率。更让人满意的是，它还非常简单、直观。我们不需要太长的学习时间就能快速掌握它。

这一节课，我们就一起来看看应该怎么使用 Java 平台模块系统。

阅读案例

在前面的课程里，我们多次使用了 Digest 这个案例来讨论问题。在这些案例里，我们把实现的代码和接口定义的代码放在了同一个文件里。对于一次 Java 新特性的讨论来说，这样做也许是合适的。我们可以使用简短的代码，快速、直观地展示新特性。

```
public sealed abstract class Digest {
    private static final class SHA256 extends Digest {
        // snipped, implementation code.
    }
    private static final class SHA512 extends Digest {
        // snipped, implementation code.
    }
    public static Returned<Digest> of(String algorithm) {
        // snipped, implementation code.
    }
    public abstract byte[] digest(byte[] message);
}
```

但是，如果放到生产环境，这样的示例就不一定是一个好的导向了。因为，Digest 的算法可能有数十种。其中有老旧废弃的算法，有即将退役的算法，还有当前推荐的算法。把这些算法的实现都装到一个瓶子里，似乎有点拥挤。

而且，不同的算法，可能有不同的许可证和专利限制；实现的代码也可能是由不同的个人或者公司提供的。同一个算法，可能还会有不同的实现：有的实现需要硬件加速，有的实现需要使用纯 Java 代码。这些情况下，这些实现代码其实都是没有办法装到同一个瓶子里的。

所以，典型的做法就是分离接口和实现。

首先，我们来看一看接口的设计。下面的代码就是一个接口定义的例子。

```
package co.ivi.jus.crypto;
import java.util.ServiceLoader;
public interface Digest {
    byte[] digest(byte[] message);
    static Returned<Digest> of(String algorithm) {
        ServiceLoader<DigestManager> serviceLoader =
                ServiceLoader.load(DigestManager.class);
        for (DigestManager cryptoManager : serviceLoader) {
            Returned<Digest> rt = cryptoManager.create(algorithm);
            switch (rt) {
                case Returned.ReturnValue rv -> {
                    return rv;
                }
                case Returned.ErrorCode ec -> {
                    continue;
                }
            }
```

```
        }
        return Returned.UNDEFINED;
    }
}
```

在这个例子里，我们只定义了 Digest 的公开接口，以及实现获取的方法（使用 ServiceLoader），而没有实现具体算法的代码。同时呢，我们希望 Digest 接口所在的包也是公开的，这样应用程序可以方便地访问这个接口。

有了 Digest 的公开接口，我们还需要定义连接公开接口和私有实现的桥梁，也就是实现的获取和供给办法。下面这段代码，定义的就是这个公开接口和私有实现之间的桥梁。Digest 公开接口的实现代码需要访问这个桥梁接口，所以它也是公开的接口。

```
package co.ivi.jus.crypto;
public interface DigestManager {
    Returned<Digest> create(String algorithm);
}
```

然后，我们来看看 Digest 接口实现的部分。有了 Digest 的公开接口和实现的桥梁接口，Digest 的实现代码就可以放置在另外一个 Java 包里了。比如，下面的例子里，我们把 Sha256 的实现，放在了 co.ivi.jus.impl 这个包里。

```
package co.ivi.jus.impl;
import co.ivi.jus.crypto.Digest;
import co.ivi.jus.crypto.Returned;
final class Sha256 implements Digest {
    static final Returned.ReturnValue<Digest> returnedSha256;
    // snipped
    private Sha256() {
        // snipped
    }
    @Override
    public byte[] digest(byte[] message) {
        // snipped
    }
}
```

因为这只是一个算法的实现代码，我们不希望应用程序直接调用实现的子类，也不希望应用程序直接访问这个 Java 包。所以，Sha256 这个子类，使用了缺省的访问修饰符。

同时，在这个 Java 包里，我们也实现了 Sha256 的间接获取方式，也就是实现了桥梁接口。

```
package co.ivi.jus.impl;
// snipped
public final class DigestManagerImpl implements DigestManager {
    @Override
    public Returned<Digest> create(String algorithm) {
        return switch (algorithm) {
            case "SHA-256" -> Sha256.returnedSha256;
            case "SHA-512" -> Sha512.returnedSha512;
            default -> Returned.UNDEFINED;
        };
    }
}
```

稍微有点遗憾的是，由于 ServiceLoader 需要使用 public 修饰的桥梁接口，所以我们不能使用除了 public 以外的访问修饰符。也就是说，如果应用程序加载了这个 Java 包，它就可以直接使用 DigestManagerImpl 类。这当然不是我们期望的使用办法。

我们并不希望应用程序直接使用 DigestManagerImpl 类，然而 JDK 8 之前的 Java 世界里，我们并没有简单有效的、强制性的封装办法。所以，我们的解决办法通常是对外宣称：“co.iv

i.jus.impl”这个包是一个内部 Java 包，请不要直接使用。这需要应用程序的开发者仔细地阅读文档，分辨内部包和公开包。
但在 Java 9 之后的 Java 世界里，我们就可以使用 Java 模块来限制应用程序使用 DigestManagerImpl 类了。

使用 Java 模块

下面我们来一起看看，Java 模块是怎么实现这样的限制的。

模块化公开接口

首先呢，我们把公开接口的部分，也就是 co.ivi.jus.crypto 这个 Java 包封装到一个 Java 模块里。我们给这个模块命名为 jus.crypto。Java 模块的定义，使用的是 module-info.java 这个文件。这个文件要放在源代码的根目录下。下面的代码，就是我们封装公开接口的部分的 module-info.java 文件。

```
module jus.crypto {
    exports co.ivi.jus.crypto;
    uses co.ivi.jus.crypto.DigestManager;
}
```

第一行代码里的“module”，就是模块化定义的关键字。紧接着 module 的就是要定义的模块的名字。在这个例子里，我们定义的是 jus.crypto 这个 Java 模块。
第二行代码里的“exports”，说明了这个模块允许外部访问的 API，也就是这个模块的公开接口。“模块的公开接口”，是一个 Java 模块带来的新概念。
没有 Java 模块的时候，除了内部接口，我们可以把 public 访问修饰符修饰的外部接口看作是公开的接口。这样的规则，需要我们去人工分辨内部接口和外部接口。
但有了 Java 模块之后我们就知道，使用了“exports”模块定义、并且使用了 public 访问修饰符修饰的接口，就是公开接口。这样，公开接口就有了清晰的定义，我们就不用再去人工分辨内部接口和外部接口了。
而第四行代码里的“uses”呢，则说明这个模块直接使用了 DigestManager 定义的服务接口。
你看，这么简短的五行代码，就把 co.ivi.jus.crypto 这个 Java 模块化了。它定义了公开接口以及要使用的服务接口。

模块化内部接口

然后呢，我们要把内部接口的部分，也就是 co.ivi.jus.impl 这个 Java 包也封装到一个 Java 模块里。下面的代码，就是我们封装内部接口部分的 module-info.java 文件。

```
module jus.crypto.impl {
    requires jus.crypto;
    provides co.ivi.jus.crypto.DigestManager with
co.ivi.jus.impl.DigestManagerImpl;
}
```

在这里，第一行代码定义了 jus.crypto.impl 这个 Java 模块。
第二行代码里的“requires”说明，这个模块需要使用 jus.crypto 这个模块。也就是说，定义了这个模块的依赖关系。有了这个明确定义的依赖关系，加载这个模块的时候，Java 运行时就不再需要地毯式地搜索依赖关系了。
第四行代码里的“provides”说明，这个模块实现了 DigestManager 定义的服务接口。同样的，有了这个明确的定义，服务接口实现的搜索，也不需要地毯式地排查了。
需要注意的是，这个模块并没有使用“exports”定义模块的公开接口。这也就意味着，虽然在 co.ivi.jus.impl 这个 Java 包里，有使用 public 访问修饰符修饰的接口，它们也不能被模块外部的应用程序访问。这样，我们就不用担心应用程序直接访问 DigestManagerImpl 类了。取而代之的，应用程序只能通过 DigestManager 这个公开的接口，间接地访问这个实现类。这是我们想要的封装效果。

模块化应用程序

有了公开接口和实现，我们再来看看该怎么模块化应用程序。下面的代码，是我们使用了 Digest 公开接口的一个小应用程序。

```
package co.ivi.jus.use;
import co.ivi.jus.crypto.Digest;
import co.ivi.jus.crypto.Returned;
public class UseCase {
    public static void main(String[] args) {
        Returned<Digest> rt = Digest.of("SHA-256");
        switch (rt) {
            case Returned.ReturnValue rv -> {
                Digest d = (Digest) rv.returnValue();
                d.digest("Hello, world!".getBytes());
            }
            case Returned.ErrorCode ec ->
                    System.getLogger("co.ivi.jus.stack.union")
                        .log(System.Logger.Level.INFO,
                              "Failed to get instance of SHA-256");
        }
    }
}
```

下面的代码，就是我们封装这个应用程序的 module-info.java 文件。

```
module jus.crypto.use {
    requires jus.crypto;
}
```

在这里，第一行代码定义了 jus.crypto.use 这个 Java 模块。
第二行代码里的“requires”，说明这个模块需要使用 jus.crypto 这个模块。
需要注意的是，这个模块并没有使用“exports”定义模块的公开接口。那么，我们该怎么运行 UseCase 这个类的 main 方法呢？其实，和传统的 Java 代码相比，模块的编译和运行有着自己的特色。

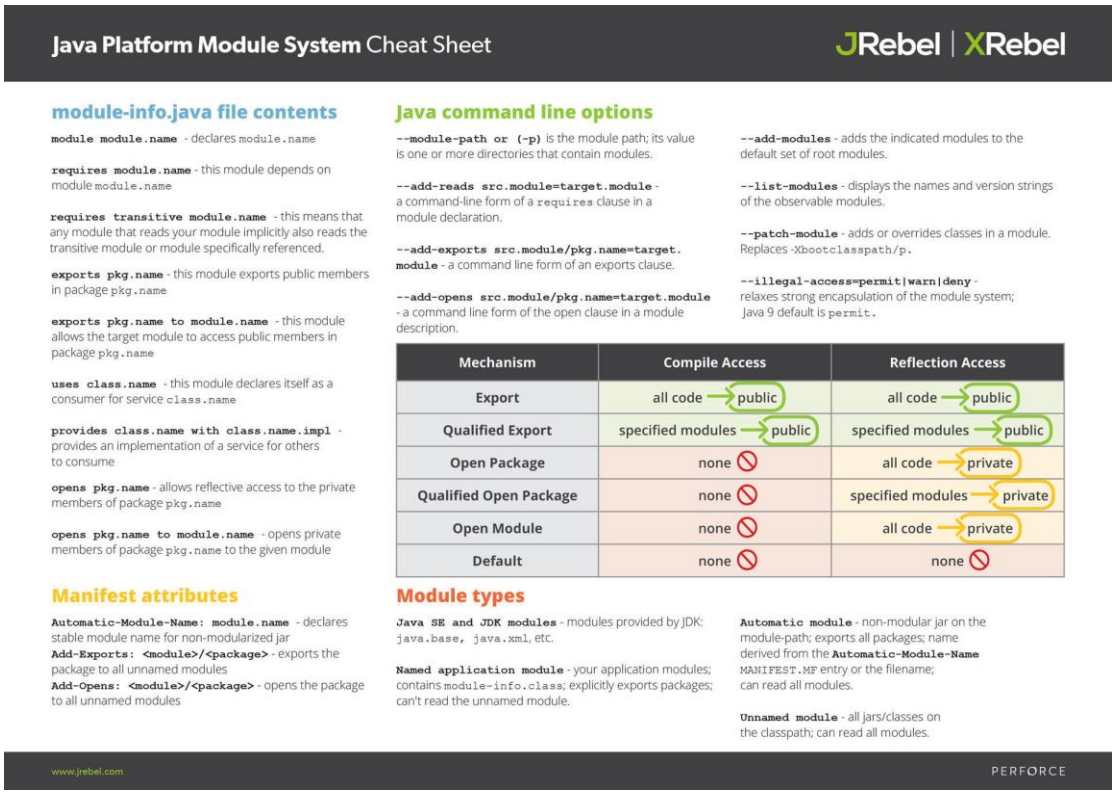
模块的编译和运行

在 javac 和 java 命令行里，我们可以使用“-module-path”指定 java 模块的搜索路径。
在 Jar 命令行里，我们可以使用“-main-class”指定这个 Jar 文件的 main 函数所在的类。
在 Java 命令里，我们可以使用“-module”指定 main 函数所在的模块。
有了这些选项的配合，在上面的例子里，我们就不需要把 UseCase 在模块里定义成公开类了。我们来看看这些选项是怎么使用的。

```
$ cd jus.crypto
$ javac --enable-preview --release 17 \
      -d classes src/main/java/co/ivi/jus/crypto/* \
      src/main/java/module-info.java
$ jar --create --file ../jars/jus.crypto.jar -C classes .
$ cd ../jus.crypto.impl
$ javac --enable-preview --release 17 \
      --module-path ../jars -d classes \
      src/main/java/co/ivi/jus/impl/* \
      src/main/java/module-info.java
$ jar --create --file ../jars/jus.crypto.impl.jar -C classes .
$ cd ../jus.crypto.use
$ javac --enable-preview --release 17 \
      --module-path ../jars -d classes \
      src/main/java/co/ivi/jus/use/* \
      src/main/java/module-info.java
$ jar --create --file ../jars/jus.crypto.use.jar \
      --main-class co.ivi.jus.use.UseCase \
```

```
      -C classes .
$ java --enable-preview --module-path ../jars --module jus.crypto.use
```

我在专栏里不会讲解这些选项的细节。具体的用法，我更希望你去找第一手的资料。下面的这个备忘单是我看到的一个比较好的总结。你可以打印下来备用，用熟了之后再丢掉。



Java Platform Module System Cheat Sheet

JRebel | XRebel

module-info.java file contents

module module.name - declares module.name

requires module.name - this module depends on module module.name

requires transitive module.name - this means that any module that reads your module implicitly also reads the transitive module or module specifically referenced.

exports pkg.name - this module exports public members in package pkg.name

exports pkg.name to module.name - this module allows the target module to access public members in package pkg.name

uses class.name - this module declares itself as a consumer for service class.name

provides class.name with class.name.impl - provides an implementation of a service for others to consume

opens pkg.name - allows reflective access to the private members of package pkg.name

opens pkg.name to module.name - opens private members of package pkg.name to the given module

Java command line options

--module-path or (-p) is the module path; its value is one or more directories that contain modules.

--add-reads src.module=target.module - a command-line form of a requires clause in a module declaration.

--add-exports src.module/pkg.name=target.module - a command line form of an exports clause.

--add-opens src.module/pkg.name=target.module - a command line form of the open clause in a module description.

--add-modules - adds the indicated modules to the default set of root modules.

--list-modules - displays the names and version strings of the observable modules.

--patch-module - adds or overrides classes in a module. Replaces -Xbootclasspath/p.

--illegal-access=permit|warn|deny - relaxes strong encapsulation of the module system; Java 9 default is permit.

Mechanism	Compile Access	Reflection Access
Export	all code → public	all code → public
Qualified Export	specified modules → public	specified modules → public
Open Package	none	all code → private
Qualified Open Package	none	specified modules → private
Open Module	none	all code → private
Default	none	none

Manifest attributes

Automatic-Module-Name: module.name - declares stable module name for non-modularized jar
Add-Exports: <module>/<package> - exports the package to all unnamed modules
Add-Opens: <module>/<package> - opens the package to all unnamed modules

Module types

Java SE and JDK modules - modules provided by JDK: java.base, java.xml, etc.

Named application module - your application modules; contains module-info.class; explicitly exports packages; can't read the unnamed module.

Automatic module - non-modular jar on the module-path; exports all packages; name derived from the **Automatic-Module-Name** MANIFEST.MF entry or the filename; can read all modules.

Unnamed module - all jars/classes on the classpath; can read all modules.

www.jrebel.com

PERFORCE

总结

好，到这里，我来做个小结。前面，我们讨论了怎么使用 Java 模块封装我们的代码，了解了 module-info.java 文件以及它的结构和关键字。

总体来看，Java 模块的使用是简单、直观的。Java 模块的使用，实现了更好的封装，也定义了模块和 Java 包之间的依赖关系。有了依赖关系，Java 语言就能够实现更快的类检索和类加载了。这样的性能提升，通过模块化就能实现，还不需要更改代码。

如果面试的时候，讨论到了 Java 平台模块系统，你可以聊一聊 Java 模块封装的关键字，以及这些关键字能够起到的作用。我相信，这是一个有意思、有深度的话题。

思考题

在前面的讨论中，我们把 DigestManager 定义成了公开接口。我们希望 Digest 的实现可以使用这个桥梁接口，但是我们又不希望应用程序直接使用它。取而代之的，应用程序应该使用 Digest.of 方法获得算法的实现。从这个意义上说，我们前面的案例，并没有做好封装。那么，有没有更好的办法，把 DigestManager 也封装起来，让应用程序无法调用呢？这是我们这一次、也是最后一次的思考题。

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的改进。

注：本文使用的完整代码可以从 GitHub 下载。

结束语 | Java 的未来，依然是星辰大海

你好，我是范学雷。

最后要和你说再见了，这是我们这个专栏的最后一课，感谢你能坚持和我一起学完。一路上，我们聊了很多的技术和技术背后的故事。在专栏结束的时候，我想和大家聊一聊 Java 的未来。

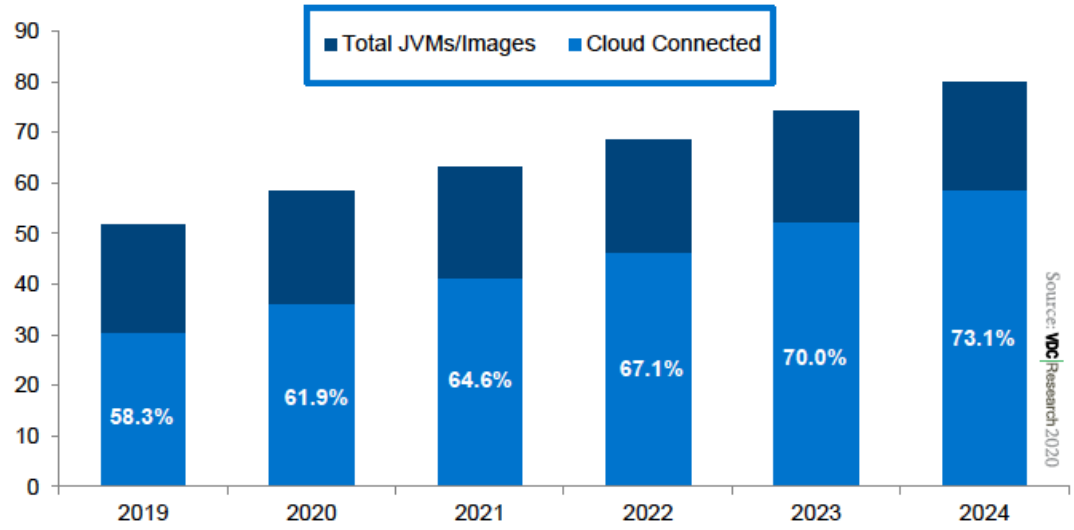
未来这个东西，看不见摸不着，能够影响它的因素实在太多了。能对未来有实质性影响的因素，也许是我们现在看不起、看不见的东西。所以，当我们说一个事情的未来的时候，总是有点算命先生的味道。

不过，看清未来对我们来说至关重要。一个人或者一个公司的命运和归宿，在很大程度上取决于我们的选择。而我们的选择，在很大程度上取决于我们对未来的看法。所以，即便对未来没有充分的把握，我们总是想探讨一下未来、预测一下未来。

那么，Java 还有没有未来？Java 的未来在哪里呢？我想，Java 的未来，依然是星辰大海。我们常说，“后视镜里看不到未来”。不过，我们也说，“以史为鉴可以知兴替”。历史的数据和故事，是我们试图描绘未来的依据。

2020 年，VDC 发布了一份关于 Java 的研究报告。下面的柱状图，就是来源于这份研究报告。2020 年，大约有 600 亿份运行的 Java 虚拟机。其中大约 61.9%，也就是大约 372 亿份 Java 虚拟机是运行在云计算的环境下的。到 2023 年，预计将会有 700 亿份运行的 Java 虚拟机；而 70%以上的 Java 虚拟机将在云上运行。

Exhibit 6: Total Cloud-Connected Active JVMs/Images Worldwide (Billions)



数字也许很冰冷，数字背后的逻辑可能更有意思。

600 亿份运行的 Java 虚拟机，这是一个庞大的数字。这个数字表示的是 Java 用户的庞大基数，以及 Java 应用的广泛部署。600 亿份运行的 Java 虚拟机，是 Java 生态系统的庞大资产，也是 Java 能够持续繁荣的背后力量。

这个庞大的数字暗示了两个逻辑。

第一个逻辑是 Java 平台的兼容性至关重要，这是对 Java 开发者的要求。任何看似微不足道的兼容性问题，都可能导致数亿台设备的故障。兼容性会导致软件的复杂性，而软件的复杂性会影响软件的持续进化。Java 的设计者可能会畏手畏脚，前怕狼后怕虎；而事实情况也是前有狼后有虎。所以，600 亿份运行的 Java 虚拟机，既是庞大的资产，也是庞大的责任。怎么在不影响兼容性的情况下，保持持续的进化，这是 Java 语言面临的最现实的挑战。

如果 Java 社区能够解决好兼容和进化的矛盾，Java 的地位是无法撼动的。

第二个逻辑是强悍的杠杆效应，小技术也能带来大影响。即便是一个微小的改进，加上 600 亿倍的杠杆，也能有巨大的效应。对于平常的代码开发，我们可能不太满意百分之一或者百分之二的性能提升；我们可能也不太愿意花时间让接口更简单、更皮实。但是，在 600 亿倍的杠杆的支撑下，任何微小的改进和努力，都会会有巨大的生态效应。也正是因为这样强悍

的杠杆效应，更多的、更广泛的开发者，愿意投入到微小的改进中。这样，Java 社区就能吸引更多的公司、更多的开发者参与其中。而那些起初不起眼的微小改进，后来都有可能产生重量级的影响。

如果 Java 社区能够持续地吸引开发者，Java 的繁荣就能够得以持续和发展。

如果进化和社区的问题能够处理好，我相信 Java 的未来依然是美好的。如果你回头看看这个专栏，重新从进化和社区的角度仔细品味每一个新特性，我想你能够看到 Java 在这方面做出的持续努力。

你和我，都是推动 Java 向前走的力量。祝愿 Java 有一个美好的未来，也祝愿你有一个更美好的未来！