第2讲 | Exception和Error有什么区别？

2018-05-08 杨晓峰

第2讲 | Exception和Error有什么区别？
朗读人：黄洲君　11′14″ | 5.15M

世界上存在永远不会出错的程序吗？也许这只会出现在程序员的梦中。随着编程语言和软件的诞生，异常情况就如影随形地纠缠着我们，只有正确处理好意外情况，才能保证程序的可靠性。

Java 语言在设计之初就提供了相对完善的异常处理机制，这也是 Java 得以大行其道的原因之一，因为这种机制大大降低了编写和维护可靠程序的门槛。如今，异常处理机制已经成为现代编程语言的标配。

今天我要问你的问题是，请对比 Exception 和 Error，另外，运行时异常与一般异常有什么区别？

典型回答

Exception 和 Error 都是继承了 Throwable 类，在 Java 中只有 Throwable 类型的实例才可以被抛出（throw）或者捕获（catch），它是异常处理机制的基本组成类型。

Exception 和 Error 体现了 Java 平台设计者对不同异常情况的分类。Exception 是程序正常运行中，可以预料的意外情况，可能并且应该被捕获，进行相应处理。

Error 是指在正常情况下，不大可能出现的情况，绝大部分的 Error 都会导致程序（比如 JVM 自身）处于非正常的、不可恢复状态。既然是非正常情况，所以不便于也不需要捕获，常见的比如 OutOfMemoryError 之类，都是 Error 的子类。

Exception 又分为可检查（checked）异常和不检查（unchecked）异常，可检查异常在源代码里必须显式地进行捕获处理，这是编译期检查的一部分。前面我介绍的不可查的 Error，是 Throwable 不是 Exception。

不检查异常就是所谓的运行时异常，类似 NullPointerException、ArrayIndexOutOfBoundsException 之类，通常是可以编码避免的逻辑错误，具体根据需要来判断是否需要捕获，并不会在编译期强制要求。

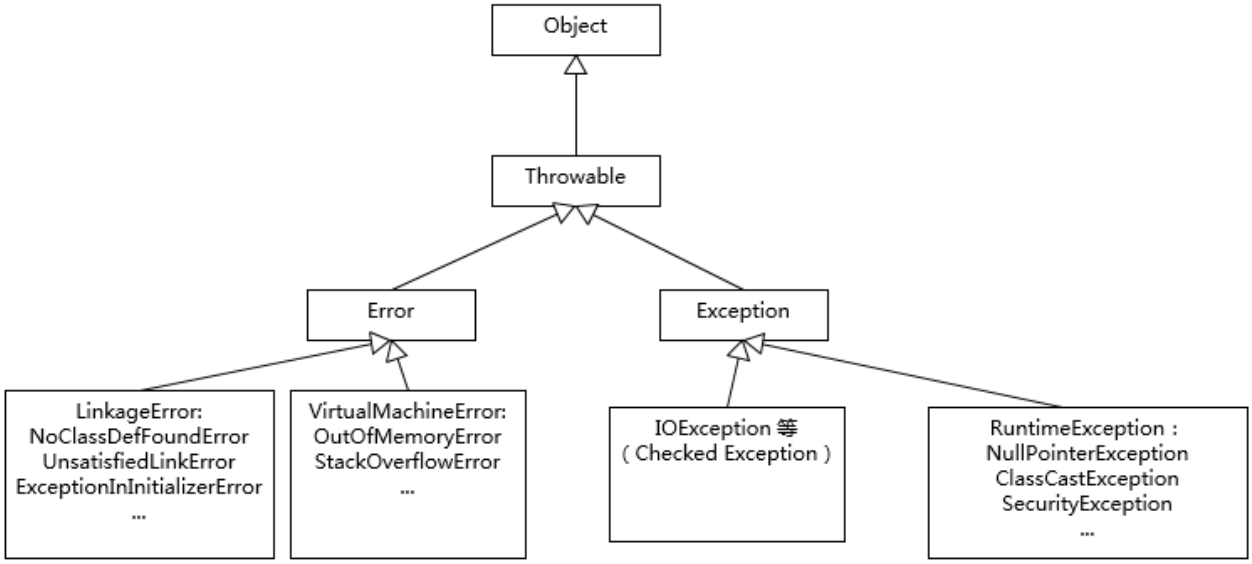
考点分析

分析 Exception 和 Error 的区别，是从概念角度考察了 Java 处理机制。总的来说，还处于理解的层面，面试者只要阐述清楚就好了。

我们在日常编程中，如何处理好异常是比较考验功底的，我觉得需要掌握两个方面。

第一，理解 Throwable、Exception、Error 的设计和分类。比如，掌握那些应用最为广泛的子类，以及如何自定义异常等。

很多面试官会进一步追问一些细节，比如，你了解哪些 Error、Exception 或者 RuntimeException？我画了一个简单的类图，并列出来典型例子，可以给你作为参考，至少做到基本心里有数。



其中有些子类型，最好重点理解一下，比如 NoClassDefFoundError 和 ClassNotFoundException 有什么区别，这也是个经典的入门题目。

第二，理解 Java 语言中操作 Throwable 的元素和实践。掌握最基本的语法是必须的，如 try-catch-finally 块，throw、throws 关键字等。与此同时，也要懂得如何处理典型场景。

异常处理代码比较繁琐，比如我们需要写很多千篇一律的捕获代码，或者在 finally 里面做一些资源回收工作。随着 Java 语言的发展，引入了一些更加便利的特性，比如 try-with-resources 和 multiple catch，具体可以参考下面的代码段。在编译时期，会自动生成相应的处理逻辑，比如，自动按照约定俗成 close 那些扩展了 AutoCloseable 或者 Closeable 的对象。

```
try (BufferedReader br = new BufferedReader(…);
     BufferedWriter writer = new BufferedWriter(…)) {// Try-with-resources
// do something
catch ( IOException | XEception e) {// Multiple catch
   // Handle it
}
```

知识扩展

前面谈的大多是概念性的东西，下面我来谈些实践中的选择，我会结合一些代码用例进行分析。

先开看第一个吧，下面的代码反映了异常处理中哪些不当之处？

```
try {
  // 业务代码
  // …
  Thread.sleep(1000L);
} catch (Exception e) {
  // Ignore it
}
```

这段代码虽然很短，但是已经违反了异常处理的两个基本原则。

第一，尽量不要捕获类似 Exception 这样的通用异常，而是应该捕获特定异常，在这里是 Thread.sleep() 抛出的 InterruptedException。

这是因为在日常的开发和合作中，我们读代码的机会往往超过写代码，软件工程是门协作的艺术，所以我们有义务让自己的代码能够直观地体现出尽量多的信息，而泛泛的 Exception 之类，恰恰隐藏了我们的目的。另外，我们也要保证程序不会捕获到我们不希望捕获的异常。比如，你可能更希望 RuntimeException 被扩散出来，而不是被捕获。

进一步讲，除非深思熟虑了，否则不要捕获 Throwable 或者 Error，这样很难保证我们能够正确程序处理 OutOfMemoryError。

第二，不要生吞（swallow）异常。这是异常处理中要特别注意的事情，因为很可能会导致非常难以诊断的诡异情况。

生吞异常，往往是基于假设这段代码可能不会发生，或者感觉忽略异常是无所谓的，但是千万不要在产品代码做这种假设！

如果我们不把异常抛出来，或者也没有输出到日志（Logger）之类，程序可能在后续代码以不可控的方式结束。没人能够轻易判断究竟是哪里抛出了异常，以及是什么原因产生了异常。

再来看看第二段代码

```
try {
   // 业务代码
   // …
} catch (IOException e) {
    e.printStackTrace();
}
```

这段代码作为一段实验代码，它是没有任何问题的，但是在产品代码中，通常都不允许这样处理。你先思考一下这是为什么呢？

我们先来看看printStackTrace()的文档，开头就是“Prints this throwable and its backtrace to the standard error stream”。问题就在这里，在稍微复杂一点的生产系统中，标准出错（STERR）不是个合适的输出选项，因为你很难判断出到底输出到哪里去了。

尤其是对于分布式系统，如果发生异常，但是无法找到堆栈轨迹（stacktrace），这纯属是为诊断设置障碍。所以，最好使用产品日志，详细地输出到日志系统里。

我们接下来看下面的代码段，体会一下Throw early, catch late 原则。

```
public void readPreferences(String fileName){
     //...perform operations...
    InputStream in = new FileInputStream(fileName);
     //...read the preferences file...
}
```

如果 fileName 是 null，那么程序就会抛出 NullPointerException，但是由于没有第一时间暴露出问题，堆栈信息可能非常令人费解，往往需要相对复杂的定位。这个 NPE 只是作为例子，实际产品代码中，可能是各种情况，比如获取配置失败之类的。在发现问题的时候，第一时间抛出，能够更加清晰地反映问题。

我们可以修改一下，让问题“throw early”，对应的异常信息就非常直观了。

```
public void readPreferences(String filename) {
    Objects. requireNonNull(filename);
    //...perform other operations...
    InputStream in = new FileInputStream(filename);
     //...read the preferences file...
}
```

至于“catch late”，其实是我们经常苦恼的问题，捕获异常后，需要怎么处理呢？最差的处理方式，就是我前面提到的“生吞异常”，本质上其实是掩盖问题。如果实在不知道如何处理，可以选择保留原有异常的 cause 信息，直接再抛出或者构建新的异常抛出去。在更高层面，因为有了清晰的（业务）逻辑，往往会更清楚合适的处理方式是什么。

有的时候，我们会根据需要自定义异常，这个时候除了保证提供足够的信息，还有两点需要考虑：

- 是否需要定义成 Checked Exception，因为这种类型设计的初衷更是为了从异常情况恢复，作为异常设计者，我们往往有充足信息进行分类。
- 在保证诊断信息足够的同时，也要考虑避免包含敏感信息，因为那样可能导致潜在的安全问题。如果我们看 Java 的标准类库，你可能注意到类似 java.net.ConnectException，出错信息是类似“ Connection refused (Connection refused)”，而不包含具体的机器名、IP、端口等，一个重要考量就是信息安全。类似的情况在日志中也有，比如，用户数据一般是不可以输出到日志里面的。

业界有一种争论（甚至可以算是某种程度的共识），Java 语言的 Checked Exception 也许是个设计错误，反对者列举了几点：

- Checked Exception 的假设是我们捕获了异常，然后恢复程序。但是，其实我们大多数情况下，根本就不可能恢复。Checked Exception 的使用，已经大大偏离了最初的设计目的。
- Checked Exception 不兼容 functional 编程，如果你写过 Lambda/Stream 代码，相信深有体会。

很多开源项目，已经采纳了这种实践，比如 Spring、Hibernate 等，甚至反映在新的编程语言设计中，比如 Scala 等。 如果有兴趣，你可以参考：

http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/。

当然，很多人也觉得没有必要矫枉过正，因为确实有一些异常，比如和环境相关的 IO、网络等，其实是存在可恢复性的，而且 Java 已经通过业界的海量实践，证明了其构建高质量软件的能力。我就不再进一步解读了，感兴趣的同学可以点击链接，观看 Bruce Eckel 在 2018 年全球软件开发大会 QCon 的分享 Failing at Failing: How and Why We've Been Nonchalantly Moving Away From Exception Handling。

我们从性能角度来审视一下 Java 的异常处理机制，这里有两个可能会相对昂贵的地方：

- try-catch 代码段会产生额外的性能开销，或者换个角度说，它往往会影响 JVM 对代码进行优化，所以建议仅捕获有必要的代码段，尽量不要一个大的 try 包住整段的代码；与此同时，利用异常控制代码流程，也不是一个好主意，远比我们通常意义上的条件语句（if/else、switch）要低效。
- Java 每实例化一个 Exception，都会对当时的栈进行快照，这是一个相对比较重的操作。如果发生的非常频繁，这个开销可就不能被忽略了。

所以，对于部分追求极致性能的底层类库，有种方式是尝试创建不进行栈快照的 Exception。这本身也存在争议，因为这样做的假设在于，我创建异常时知道未来是否需要堆栈。问题是，实际上可能吗？小范围或许可能，但是在大规模项目中，这么做可能不是个理智的选择。如果需要堆栈，但又没有收集这些信息，在复杂情况下，尤其是类似微服务这种分布式系统，这会大大增加诊断的难度。

当我们的服务出现反应变慢、吞吐量下降的时候，检查发生最频繁的 Exception 也是一种思路。关于诊断后台变慢的问题，我会在后面的 Java 性能基础模块中系统探讨。

今天，我从一个常见的异常处理概念问题，简单总结了 Java 异常处理的机制。并结合代码，分析了一些普遍认可的最佳实践，以及业界最新的一些异常使用共识。最后，我分析了异常性能开销，希望对你有所帮助。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？可以思考一个问题，对于异常处理编程，不同的编程范式也会影响到异常处理策略，比如，现在非常火热的反应式编程（Reactive Stream），因为其本身是异步、基于事件机制的，所以出现异常情况，决不能简单抛出去；另外，由于代码堆栈不再是同步调用那种垂直的结构，这里的异常处理和日志需要更加小心，我们看到的往往是特定 executor 的堆栈，而不是业务方法调用关系。对于这种情况，你有什么好的办法吗？

请你在留言区分享一下你的解决方案，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

精选留言

公众号:代码荣耀 80

在Java世界里，异常的出现让我们编写的程序运行起来更加的健壮，同时为程序在调试、运行期间发生的一些意外情况，提供了补救机会；即使遇到一些严重错误而无法弥补，异常也会非常忠实的记录所发生的这一切。以下是文章心得感悟:

1 不要推诿或延迟处理异常，就地解决最好，并且需要实实在在的进行处理，而不是只捕捉，不动作。

2 一个函数尽管抛出了多个异常，但是只有一个异常可被传播到调用端。最后被抛出的异常时唯一被调用端接收的异常，其他异常都会被吞没掩盖。如果调用端要知道造成失败的最初原因，程序之中就绝不能掩盖任何异常。

3 不要在finally代码块中处理返回值。

4 按照我们程序员的惯性认知：当遇到return语句的时候，执行函数会立刻返回。但是，在Java语言中，如果存在finally就会有例外。除了return语句，try代码块中的break或continue语句也可能使控制权进入finally代码块。

5 请勿在try代码块中调用return、break或continue语句。万一无法避免，一定要确保finally的存在不会改变函数的返回值。

6 函数返回值有两种类型：值类型与对象引用。对于对象引用，要特别小心，如果在finally代码块中对函数返回的对象成员属性进行了修改，即使不在finally块中显式调用return语句，这个修改也会作用于返回值上。

7 勿将异常用于控制流。

8 如无必要，勿用异常。

2018-05-08

coder王 14

留言中凸显高手。

2018-05-08

钱宇祥 11

1.异常：这种情况下的异常，可以通过完善任务重试机制，当执行异常时，保存当前任务信息加入重试队列。重试的策略根据业务需要决定，当达到重试上限依然无法成功，记录任务执行失败，同时发出告警。
2.日志：类比消息中间件，处在不同线程之间的同一任务，简单高效一点的做法可能是用traceId/requestId串联。有些日志系统本身支持MDC/NDC功能，可以串联相关联的日志。

2018-05-08

作者回复

很棒的总结

2018-05-08

欧阳田 10

1.Error:系统错误，虚拟机出错，我们处理不了，也不需要我们来处理。
2.Exception，可以捕获的异常，且作出处理。也就是要么捕获异常并作出处理，要么继续抛出异常。
3.RuntimeException，经常性出现的错误，可以
捕获，并作出处理，可以不捕获，也可以不用抛出。ArrayIndexOutOfBoundsException像这种异常可以不捕获，为什么呢？在一个程序里，使用很多数组，如果使用一次捕获一次，则很累。
4.继承某个异常时，重写方法时，要么不抛出异常，要么抛出一模一样的异常。
5.当一个try后跟了很多个catch时，必须先捕获小的异常再捕获大的异常。
6.假如一个异常发生了，控制台打印了许多行信息，是因为程序中进行多层方法调用造成的。关键是看类型和行号。
7.上传下载不能抛异常。上传下载一定要关流。
8.异常不是错误。异常控制代码流程不利于代码简单易读。
9.try catch finally执行流程，与 return，break，continue等混合使用注意代码执行顺序。不是不可以，而是越是厉害的人，代码越容易理解。

2018-05-08

涟漪 8

非常感谢作者以及评论中的高手们！我很喜欢作者能够精选评论。

2018-05-09

猿工匠 8

每天早上学习与复习一下😁😁

2018-05-08

迷途知返 7

我比较菜 在听到“NoClassDefFoundError 和 ClassNotFoundException 有什么区别，这也是个经典的入门题目。 ”这一段的时候 我以为会讲这两个的区别呢 我觉得这个区别详细讲讲 就是干货！文章总结性的语言比较多 并不具体

2018-05-17

Alphabet 7

老师可以在文章末尾推荐一些基础和进阶的Java学习书籍或是资料吗？最好是使用较新版本jdk的

2018-05-10

飞云 7

能不能讲下怎么捕捉整个项目的全局异常，说实话前两篇额干货都不多，希望点更实在的干货

2018-05-08

作者回复

谢谢建议，极客课程设计是尽量偏向通用场景，我们掉坑里，往往都不是在高大上的地方；全局异常Spring MVC的方式就很实用；对与干货，你是希望特定场景，特定问题吗？说说你的想法

2018-05-08

小绵羊拉拉 6

看完文章简单认识一些浅层的意思 但是我关注的 比如try catch源码实现 涉及 以及 文章中提到 try catch 产生 堆栈快照 影响jvm性能等 一笔带过 觉得不太过瘾。只是对于阿里的面试 读懂这篇文章还是不够。还希望作者从面试官的角度由浅入深的剖析异常处理 最后还是 谢谢分享

2018-05-08

作者回复

谢谢反馈，如果不做jvm或非常底层开发，个人没有看到这些细节的实际意义，如果非要问可以鄙视他：-）

创建Throwable因为要调用native方法fillInStacktrace；至于try catch finally，jvms第三章有细节，也可以自己写一段程序，用javap反编译看看 goto、异常表等等

2018-05-09

Jerry银银 5

由于反应式编程是异步的，基于事件的，所以异常肯定不能直接抛出，如果直接抛出，随便一个异常都会引起程序崩溃，直接影响到对后续事件处理。个人觉得一种处理方式是：当某个事件发生异常时，为了不影响对后续事件的处理，可以对当前发生异常的事件进行拦截处理，然后将异常信息发送出去。

至于发生异常时，堆栈信息只是关于特定executor框架中的，不知道是否可以将之前事件的“上下文”带到executor，再传递给观察者？
（对反应式编程不太了解，尝试作答^_^）

2018-05-08

曹铮 4

先说问题外的话，Java的checked exception总是被诟病，可我是从C#转到Java开发上来的，中间经历了go，体验过scala。我觉得Java这种机制并没有什么不好，不同的语言体验下来，错误与异常机制真是各有各的好处和槽点，而Java我觉得处在中间，不极端。当然老师提到lambda这确实是个问题...
至于响应式编程，我可以泛化为异步编程的概念嘛？一般各种异步编程框架都会对异常的传递和堆栈信息做处理吧？比如promise/future风格的。本质上大致就是把lambda中的异常捕获并封装，再进一步延续异步上下文，或者转同步处理时拿到原始的错误和堆栈信息

2018-05-08

作者回复

是的，非常棒的总结，归根结底我们需要一堆人合作构建各种规模的程序，Java异常处理有槽点，但实践证明了其能力；

类似第二点，我个人也觉得可以泛化为异步编程的概念，比如Future Stage之类使用ExecutionException的思路

2018-05-08

三军 3

Java语言规范将派生于Error类或RuntimeException类的所有异常称为未检查（unchecked）异常，所有其他的异常成为已检查（checked）异常。

编译器将检查是否为所有的已检查异常提供了异常处理器。

这是经典，要好好理解。

平时我们使用throws往外抛错，或者try-catch这类异常处理器不就是处理已检查异常吗☺

未检查异常就是潜在的，编译器无需提供异常处理器进行处理。

2018-05-10

张世杰 3

老师总结的类图，对理解Throwable,Exception,Error非常的直观！但再说掌握的两个方面，第二方面时候，仅仅提到懂得如何处理典型场景！如果能详细描述一下什么样的典型场景，会对深入理解，使用Exception,Error非常有帮助！

2018-05-08

James 3

个人觉得checked exception / unchecked exception 分别翻译为 检查型异常/非检查型异常 更加好理解。

可检查异常容易被理解为可以不检查。

2018-05-08

作者回复

有道理，谢谢指出

2018-05-08

Ccook 2

大佬能介绍下，线程间调用导致异常信息丢失的问题吗

2018-05-10

作者回复

Thtead 一个UnCaughtExceptionHandler

2018-05-10

whhbbq 2

关于检查型异常，因为要强制捕获或者在函数签名声明，导致要写好多的代码。现在都改为用运行时异常，根据业务定义好编码和消息，然后通过全局的异常处理器处理。一些特殊的需要携带更多信息的异常，会自定义异常类，当然它也是运行时异常。

2018-05-09

fangxuan 2

有种浅尝辄止的感觉，希望老师能再深入一点。另外对于程序中到底是该抛出异常还是默默的处理掉，把我不好，希望老师能给点这方面的最佳实践。函数式编程中遇到checkedexception怎么处理能详细指点一下吗？

2018-05-08

13683815260 2

个人想的有三步:
1，完善的异常记录。包括调用的上下文信息，如果在同一个进程内考虑ThreadLocal传递参数。如果分布式，把核心的参数封装传递。
2，在基础1之上构建traceid之类的调用链跟踪。
3，基于回调机制，发生异常时以事件的方式通知调用方。

另 对学习的的结果做个小总结。首先二者继承体系的异同。设计里面也不同，error一般表示不可自主从异常中恢复，Exception意味着可能可以恢复。其中Exception分为两类检查异常，非检查异常。
最佳实践，try中的代码块不宜过长，捕获时不宜大而全，finally里只释放资源不要有业务逻辑，尤其是修改返回值。用新语法可增强代码的可读性和简洁性。
业务异常可继承runntimeexception，封装applicationexception.
finally中的代码始终是执行的，用途为清理资源。
对于线程池注意runntimeexception导致的线程逃逸现象。

2018-05-08

风动静泉 2

"Exception 又分为可检查（checked）异常和不检查（unchecked）异常"这句话本身没问题，但是不够全面吧。查了下<<JAVA核心技术 卷Ⅰ>> 第9版，pp.474 "JAVA语言规范将派生于Error类或RuntimeException类的所有异常称为未检查(unchecked)异常，所有其他的异常称为已检查(checked)异常。"

2018-05-08

作者回复

没错，看描述的角度和范围，不然让人晕了

2018-05-08