

## 第3讲 | 谈谈final、finally、finalize有什么不同？

2018-05-10 杨晓峰



### 第3讲 | 谈谈final、finally、finalize有什么不同？

朗读人：黄洲君 11'03" | 5.06M

Java 语言有很多看起来很相似，但是用途却完全不同的语言要素，这些内容往往容易成为面试官考察你知识掌握程度的切入点。

今天，我要问你的是一个经典的 Java 基础题目，**谈谈 final、finally、finalize 有什么不同？**

### 典型回答

final 可以用来修饰类、方法、变量，分别有不同的意义，final 修饰的 class 代表不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。

finally 则是 Java 保证重点代码一定要被执行的一种机制。我们可以使用 try-finally 或者 try-catch-finally 来进行类似关闭 JDBC 连接、保证 unlock 锁等动作。

finalize 是基础类 java.lang.Object 的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为 deprecated。

### 考点分析

这是一个非常经典的 Java 基础问题，我上面的回答主要是从语法和使用实践角度出发的，其实还有很多方面可以深入探讨，面试官还可以考察你对性能、并发、对象生命周期或垃圾收集基本过程等方面的理解。

推荐使用 `final` 关键字来明确表示我们代码的语义、逻辑意图，这已经被证明在很多场景下是非常好的实践，比如：

- 我们可以将方法或者类声明为 `final`，这样就可以明确告知别人，这些行为是不许修改的。

如果你关注过 Java 核心类库的定义或源码，有没有发现 `java.lang` 包下面的很多类，相当一部分都被声明成为 `final class`？在第三方类库的一些基础类中同样如此，这可以有效避免 API 使用者更改基础功能，某种程度上，这是保证平台安全的必要手段。

- 使用 `final` 修饰参数或者变量，也可以清楚地避免意外赋值导致的编程错误，甚至，有人明确推荐将所有方法参数、本地变量、成员变量声明成 `final`。
- `final` 变量产生了某种程度的不可变（immutable）的效果，所以，可以用于保护只读数据，尤其是在并发编程中，因为明确地不能再赋值 `final` 变量，有利于减少额外的同步开销，也可以省去一些防御性拷贝的必要。

`final` 也许会有性能的好处，很多文章或者书籍中都介绍了可在特定场景提高性能，比如，利用 `final` 可能有助于 JVM 将方法进行内联，可以改善编译器进行条件编译的能力等等。坦白说，很多类似的结论都是基于假设得出的，比如现代高性能 JVM（如 HotSpot）判断内联未必依赖 `final` 的提示，要相信 JVM 还是非常智能的。类似的，`final` 字段对性能的影响，大部分情况下，并没有考虑的必要。

从开发实践的角度，我不想过度强调这一点，这是和 JVM 的实现很相关的，未经验证比较难以把握。我的建议是，在日常开发中，除非有特别考虑，不然最好不要指望这种小技巧带来的所谓性能好处，程序最好是体现它的语义目的。如果你确实对这方面有兴趣，可以查阅相关资料，我就不再赘述了，不过千万别忘了验证一下。

对于 `finally`，明确知道怎么使用就足够了。需要关闭的连接等资源，更推荐使用 Java 7 中添加的 `try-with-resources` 语句，因为通常 Java 平台能够更好地处理异常情况，编码量也要少很多，何乐而不为呢。

另外，我注意到有一些常被考到的 `finally` 问题（也比较偏门），至少需要了解一下。比如，下面代码会输出什么？

```
try {  
    // do something  
    System.exit(1);  
}
```

```
} finally{  
    System.out.println("Print from finally");  
}
```

上面 finally 里面的代码可不会被执行的哦，这是一个特例。

对于 finalize，我们要明确它是不推荐使用的，业界实践一再证明它不是个好的办法，在 Java 9 中，甚至明确将 `Object.finalize()` 标记为 deprecated！如果没有特别的原因，不要实现 finalize 方法，也不要指望利用它来进行资源回收。

为什么呢？简单说，你无法保证 finalize 什么时候执行，执行的是否符合预期。使用不当会影响性能，导致程序死锁、挂起等。

通常来说，利用上面的提到的 try-with-resources 或者 try-finally 机制，是非常好的回收资源的办法。如果确实需要额外处理，可以考虑 Java 提供的 Cleaner 机制或者其他替代方法。接下来，我来介绍更多设计考虑和实践细节。

## 知识扩展

### 1. 注意，final 不是 immutable！

我在前面介绍了 final 在实践中的益处，需要注意的是，final 并不等同于 immutable，比如下面这段代码：

```
final List<String> strList = new ArrayList<>();  
strList.add("Hello");  
strList.add("world");  
List<String> unmodifiableStrList = List.of("hello", "world");  
unmodifiableStrList.add("again");
```

final 只能约束 strList 这个引用不可以被赋值，但是 strList 对象行为不被 final 影响，添加元素等操作是完全正常的。如果我们真的希望对象本身是不可变的，那么需要相应的类支持不可变的行为。在上面这个例子中，[List.of 方法](#)创建的本身就是不可变 List，最后那句 add 是会在运行时抛出异常的。

Immutable 在很多场景是非常棒的选择，某种意义上说，Java 语言目前并没有原生的不可变支持，如果来实现 immutable 的类，我们需要做到：

- 将 class 自身声明为 final，这样别人就不能扩展来绕过限制了。

- 将所有成员变量定义为 `private` 和 `final`，并且不要实现 `setter` 方法。
- 通常构造对象时，成员变量使用深度拷贝来初始化，而不是直接赋值，这是一种防御措施，因为你无法确定输入对象不被其他人修改。
- 如果确实需要实现 `getter` 方法，或者其他可能会返回内部状态的方法，使用 `copy-on-write` 原则，创建私有的 `copy`。

这些原则是不是在并发编程实践中经常被提到？的确如此。

关于 `setter/getter` 方法，很多人喜欢直接用 IDE 一次全部生成，建议最好是你确定有需要时再实现。

## 2.finalize 真的那么不堪？

前面简单介绍了 `finalize` 是一种已经被业界证明了的非常不好的实践，那么为什么会导出那些问题呢？

`finalize` 的执行是和垃圾收集关联在一起的，一旦实现了非空的 `finalize` 方法，就会导致相应对象回收呈现数量级上的变慢，有人专门做过 `benchmark`，大概是 40~50 倍的下降。

因为，`finalize` 被设计成在对象被垃圾收集前调用，这就意味着实现了 `finalize` 方法的对象是个“特殊公民”，JVM 要对它进行额外处理。`finalize` 本质上成为了快速回收的阻碍者，可能导致你的对象经过多个垃圾收集周期才能被回收。

有人也许会问，我用 `System.runFinalization()` 告诉 JVM 积极一点，是不是就可以了？也许有点用，但是问题在于，这还是不可预测、不能保证的，所以本质上还是不能指望。实践中，因为 `finalize` 拖慢垃圾收集，导致大量对象堆积，也是一种典型的导致 OOM 的原因。

从另一个角度，我们要确保回收资源就是因为资源都是有限的，垃圾收集时间的不可预测，可能会极大加剧资源占用。这意味着对于消耗非常高频的资源，千万不要指望 `finalize` 去承担资源释放的主要职责，最多让 `finalize` 作为最后的“守门员”，况且它已经暴露了如此多的问题。这也是为什么我推荐，资源用完即显式释放，或者利用资源池来尽量重用。

`finalize` 还会掩盖资源回收时的出错信息，我们看下面一段 JDK 的源代码，截取自 `java.lang.ref.Finalizer`

```
private void runFinalizer(JavaLangAccess jla) {  
    // ... 省略部分代码  
    try {  
        Object finalizee = this.get();  
        if (finalizee != null && !(finalizee instanceof java.lang.Enum)) {
```

```
        jla.invokeFinalize(finalizee);

        // Clear stack slot containing this variable, to decrease
        // the chances of false retention with a conservative GC
        finalizee = null;
    }
} catch (Throwable x) { }

super.clear();
}
```

结合我上期专栏介绍的异常处理实践，你认为这段代码会导致什么问题？

是的，你没有看错，这里的`Throwable` 是被生吞了的！也就意味着一旦出现异常或者出错，你得不到任何有效信息。况且，Java 在 `finalize` 阶段也没有好的方式处理任何信息，不然更加不可预测。

### 3. 有什么机制可以替换 `finalize` 吗？

Java 平台目前在逐步使用 `java.lang.ref.Cleaner` 来替换掉原有的 `finalize` 实现。`Cleaner` 的实现利用了幻象引用（`PhantomReference`），这是一种常见的所谓 `post-mortem` 清理机制。我会在后面的专栏系统介绍 Java 的各种引用，利用幻象引用和引用队列，我们可以保证对象被彻底销毁前做一些类似资源回收的工作，比如关闭文件描述符（操作系统有限的资源），它比 `finalize` 更加轻量、更加可靠。

吸取了 `finalize` 里的教训，每个 `Cleaner` 的操作都是独立的，它有自己的运行线程，所以可以避免意外死锁等问题。

实践中，我们可以为自己的模块构建一个 `Cleaner`，然后实现相应的清理逻辑。下面是 JDK 自身提供的样例程序：

```
public class CleaningExample implements AutoCloseable {
    // A cleaner, preferably one shared within a library
    private static final Cleaner cleaner = <cleaner>;

    static class State implements Runnable {

        State(...) {
            // initialize State needed for cleaning action
        }

        public void run() {
            // cleanup action accessing State, executed at most once
        }
    }
}
```

```
    }  
  
    }  
  
    private final State;  
  
    private final Cleaner.Cleanable cleanable  
  
    public CleaningExample() {  
        this.state = new State(...);  
        this.cleanable = cleaner.register(this, state);  
    }  
  
    public void close() {  
        cleanable.clean();  
    }  
  
}
```

注意，从可预测性的角度来判断，Cleaner 或者幻象引用改善的程度仍然是有限的，如果由于种种原因导致幻象引用堆积，同样会出现问题。所以，Cleaner 适合作为一种最后的保证手段，而不是完全依赖 Cleaner 进行资源回收，不然我们就要再做一遍 finalize 的噩梦了。

我也注意到很多第三方库自己直接利用幻象引用定制资源收集，比如广泛使用的 MySQL JDBC driver 之一的 mysql-connector-j，就利用了幻象引用机制。幻象引用也可以进行类似链条式依赖关系的动作，比如，进行总量控制的场景，保证只有连接被关闭，相应资源被回收，连接池才能创建新的连接。

另外，这种代码如果稍有不慎添加了对资源的强引用关系，就会导致循环引用关系，前面提到的 MySQL JDBC 就在特定模式下有这种问题，导致内存泄漏。上面的示例代码中，将 State 定义为 static，就是为了避免普通的内部类隐含着对外部对象的强引用，因为那样会使外部对象无法进入幻象可达的状态。

今天，我从语法角度分析了 final、finally、finalize，并从安全、性能、垃圾收集等方面逐步深入，探讨了实践中的注意事项，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？也许你已经注意到了，JDK 自身使用的 Cleaner 机制仍然是有缺陷的，你有什么更好的建议吗？

请你在留言区写写你的建议，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。



# Java核心技术36讲

—— Oracle 首席工程师  
带你修炼 Java 内功 ——

杨晓峰 Oracle 首席工程师



版权归极客邦科技所有，未经许可不得转载

## 精选留言



zjh

48

一直不懂为什么这三个经常拿来一起比较，本身就一点关系都没有啊，难道仅仅是长的像。  
我觉得final倒是可以和volatile一起比较下

2018-05-11



WolvesLeader

47

能不能帮我分析一哈，匿名内部类，访问局部变量时，局部变量为啥要用final来修饰吗？

2018-05-10

### 作者回复

这个因为Java inner class实际会copy一份，不是去直接使用局部变量，final可以防止出现数据一致性问题

2018-05-10



石头狮子

44

列几个 finally 不会被执行的情况:

1. try-catch 异常退出。

```
try{
    system.exit(1)
}finally{
    print(abc)
}
```

2. 无限循环

```
try{
```

```
while(ture){  
    print(abc)  
}  
}finally{  
    print(abc)  
}
```

### 3. 线程被杀死

当执行 try, finally 的线程被杀死时。finally 也无法执行。

### 总结

- 1, 不要在 finally 中使用 return 语句。
- 2, finally 总是执行, 除非程序或者线程被中断。

2018-05-10



★神峰★

👍 22

你们都看懂了吗？我怎么什么都不知道😓

2018-05-12



有渔@蔡

👍 17

1.你说那异常被生吞，是指没写e.print...语句吧？另外我有个疑惑：super.clear()为什么写在exception里，理论上super方法写第一行，或finally里。2.在一个对象的生命周期里，其finalize方法应该只会被调用1次。3.强软弱虚引用大家都知道，这虚幻引用相比较有什么特别的吗？请再深入点。4.final是不是都在编译后确定位置？比如final List这样的，内存布局是怎样的？谢谢

2018-05-10



sharp

👍 15

这三个就是卡巴斯基和巴基斯坦的关系，有个基巴关系。。。

2018-06-12



皮卡皮卡丘

👍 15

“将 State 定义为 static，就是为了避免普通的内部类隐含着对外部对象的强引用，因为那样会使外部对象无法进入幻象可达的状态。”这个该怎么理解呢？

2018-05-10

### 作者回复

内部类如果不是static，它本身对外面那个类有引用关系，这一点其实从构造阶段就能看出来，你可以写段代码试试；有强引用就是strong reachable状态

2018-05-11



小哥。

👍 10

copy-on-write 原则，学习了



2018-05-10



Heshher

👍 9

见过一些写法是将对象手动赋值为null来让GC更快的回收，不过能起多少作用就不知道了。关于JVM中那几种引用了解不多，平时可以怎么使用呢？

2018-05-10



云学

👍 7

请问这篇文章中涉及的知识点是java中最重要的吗？我感觉有点剑走偏锋，这种知识了解就好了，应该有很多知识比这更重要的吧，虽说面试中可能会问，但不能以面试为中心，而要把实际应用中最有用的真正核心的东西分享出来，把它讲透彻，不追求面面俱到，也不想成为语言专家，我期望通过这个专栏可以获得java中最核心最实用特性的本质认识，希望有一种醍醐灌顶的感觉，在阅读java开源框架代码时不再困惑。我有多年的c++开发背景，希望通过这个专栏对java也有提纲契领的本质认识。

2018-06-12



ls

👍 6

Java中有说：finalize 有一种用途：在 Java 中调用非 Java 代码，在非 Java 代码中若调用了 C 的 malloc 来分配内存，如果不调用 C 的 free 函数，会导致内存泄露。所以需要在 finalize 中调用它。

面试中会有问：为什么 String 会设计成不可变？想听听老师的解释

2018-05-13

### 作者回复

是的，很多资源都是需要使用本地方式获取和释放

2018-05-14



refusecruder

👍 5

杨老师，关于final不能修改我想请教下，代码如下，class util {  
public final Integer info = 123;  
}

@Test

```
public void test() throws NoSuchFieldException, IllegalAccessException {  
    util util = new util();  
    Field field = util.getClass().getDeclaredField("info");  
    field.setAccessible(true);  
    field.set(util, 789);  
    System.out.println(field.get(util));  
    System.out.println(util.info);  
}
```

这里final修饰的被改了，如果不加accessible这句会报错，刚刚试了几个，似乎是基本数据类型改不了，封装类型都能改，请杨老师解答下我的疑惑，感谢。

2018-05-10

### 作者回复

setAccessible是“流氓”，不问题出在定义为基本数据类型，会被当作constant，可以反编译看看

2018-05-10



Children

4

用final修饰的class，这可以有效避免API使用者更改基础功能，某种程度上，这是保证平台安全的必要手段。这个地方真的很需要个例子去帮助理解。比如大家都知道String类是被final修饰不可被继承，但假如没有被final修饰，很好奇会出现什么样不安全的后果。

2018-05-14

### 作者回复

谢谢反馈

2018-05-14



公号-Java大后端

4

1 定义不可变对象类，当构造函数传入可变对象引用时、当getter函数返回可变对象引用时，容易掉坑。

2 在不可变对象类的构造函数中，如果传入值包括了可变对象，则clone先。

3 从不可变对象类的getter函数返回前，如果返回值为可变对象，则clone先。

4 Java默认的clone方法执行浅拷贝，对于数组、对象引用只是拷贝地址。浅拷贝在业务实现中可能是一个坑，需要多加注意。

5 如果步骤2、3中的浅拷贝无法满足不可变对象要求，请实现“深拷贝”。

2018-05-10



Do

3

final修饰变量参数的时候，其实理解为内存地址的绑定，这样理解是不是更直观，基本类型指向栈中，引用类型指向堆中。老师后期文章能不能说下java堆栈的区别，还有变量局部变量的生命周期，最好能附上图，加深理解。

2018-05-12

### 作者回复

会有

2018-05-12



echo\_陈

3

回答上面一个人的问题。

被final修饰的变量不可变。如果初始化不赋值，后续赋值，就是从null变成你的赋值，违反不可变

2018-05-10



loveluckystar

3

个人理解，finalize本身就是为了提供类似c或c++析构函数产生的，由于java中gc本身就是自动进行的，是不希望被干扰的，(就像System.gc()，并不一定起作用)所以与其费心研究如

何使用这个，不如老老实实在finally中把该做的事情做了来的实惠。

2018-05-10

### 作者回复

对，有些特殊情况需要额外处理，毕竟无法保证编程都按规范来

2018-05-10



feifei

2

JDK 自身使用的 Cleaner 机制仍然是有缺陷的，你有什么更好的建议吗？

- 1，临时对象，使用完毕后，赋值为null,可以加快对象的回收
- 2，公用资源对象，比如数据库连接，使用连接池
- 3，native调用资源的释放，比如一个进程初始化调用一次，退出调用一次，这类场景可以考虑使用cleaner
- 4，对尽量try-finally中完成资源的释放，即使用完毕就释放，最小化的使用，下次使用在申请。
- 5，可以使用钩子进行程序的正常退出清理操作。

此为我个人的一点小心得，欢迎老师指正，谢谢

2018-06-30



小绵羊拉拉

2

首先 这篇文章比上一讲明显感觉到由浅入深 很不错 有个问题请教一下finalize方法是用来回收对外内存是不是可以这么理解 类似于本地方法申请的能源 new出来的对象实现了这个方法当垃圾回收的时候会将对象放在fqueue等待被执行 不过是异步不知道啥时候被执行 可能被执行的时候对象已经置空导致不安全 可以这么理解吗 新的jdk引入的clear方法 能完全取代虚拟机中finalize方法吗

2018-05-10



说重点、

2

常被问，string类为什么用final修饰

2018-05-10