深入理解 Java try-with-resource 语法糖

阅读 2252 收藏 76 2016-10-07
原文链接：www.kissyu.org

> 子曾经曰过：所有的炒冷饭都是温故而知新。

背景

众所周知，所有被打开的系统资源，比如流、文件或者Socket连接等，都需要被开发者手动关闭，否则随着程序的不断运行，资源泄露将会累积成重大的生产事故。

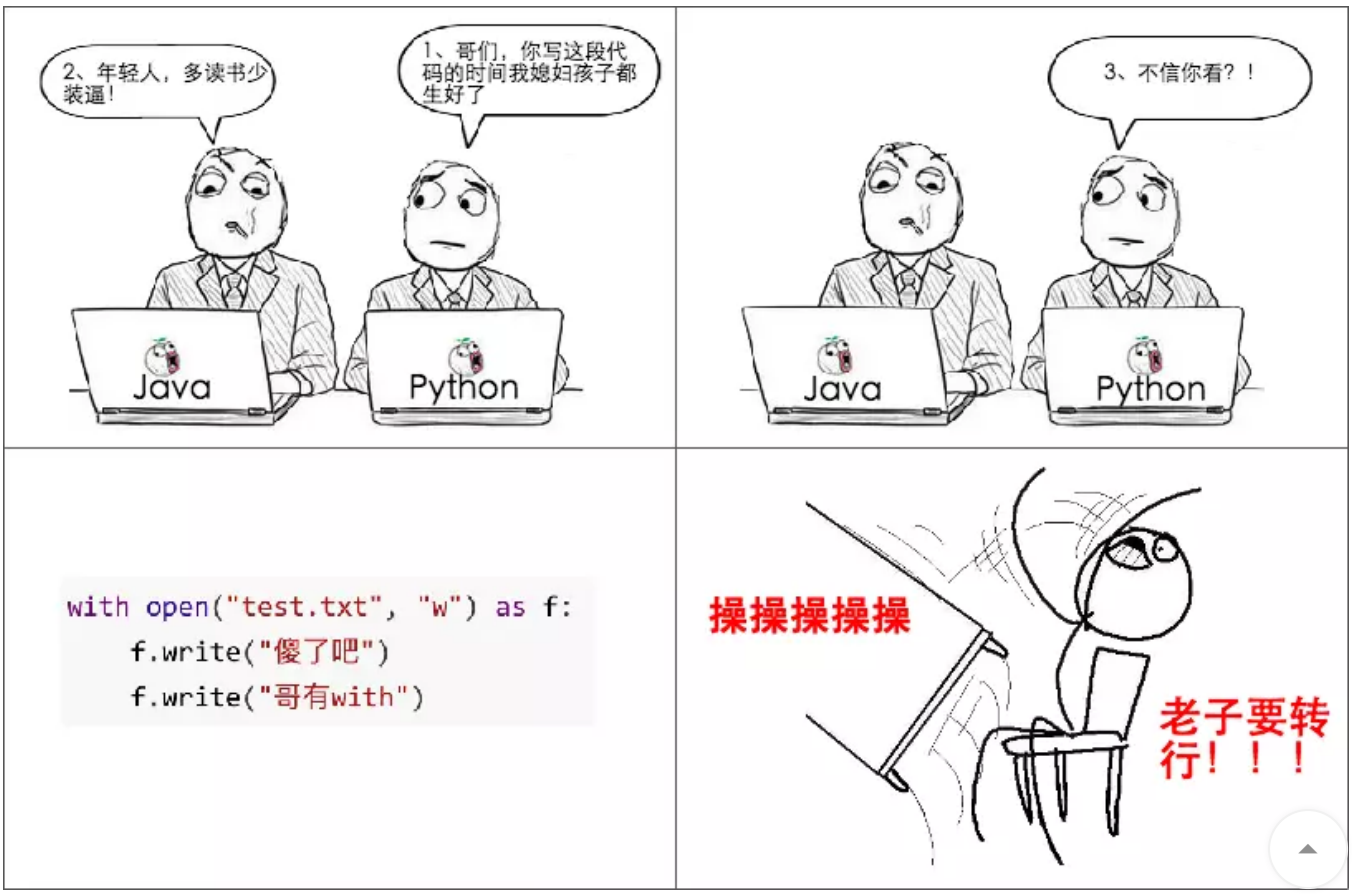
在Java的江湖中，存在着一种名为finally的功夫，它可以保证当你习武走火入魔之时，还可以做一些自救的操作。在远古时代，处理资源关闭的代码通常写在finally块中。然而，如果你同时打开了多个资源，那么将会出现噩梦般的场景：

```
public class Demo {
    public static void main(String[] args) {
        BufferedInputStream bin = null;
        BufferedOutputStream bout = null;
        try {
            bin = new BufferedInputStream(new FileInputStream(new File("test.txt")));
            bout = new BufferedOutputStream(new FileOutputStream(new File("out.txt")));
            int b;
            while ((b = bin.read()) != -1) {
                bout.write(b);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (bin != null) {
                try {
                    bin.close();
                }
                catch (IOException e) {
```

```
                finally {
                    if (bout != null) {
                        try {
                            bout.close();
                        }
                        catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }
}
```

Oh My God！！！**关闭资源的代码竟然比业务代码还要多**！！！这是因为，我们不仅需要关闭`BufferedInputStream`，还需要保证如果关闭`BufferedInputStream`时出现了异常，`BufferedOutputStream`也要能被正确地关闭。所以我们不得不借助finally中嵌套finally大法。可以想到，打开的资源越多，finally中嵌套的将会越深！！！

更为可恶的是，Python程序员面对这个问题，竟然微微一笑很倾城地说：“这个我们一点都不用考虑的嘞~”：



但是兄弟莫慌！我们可以利用Java 1.7中新增的try-with-resource语法糖来打开资源，而无需码农们自己书写资源来关闭代码。妈妈再也不用担心我把手写断掉了！我们用try-with-resource来改写刚才的例子：

```
public class TryWithResource {
    public static void main(String[] args) {
        try (BufferedInputStream bin = new BufferedInputStream(new FileInputStream(new File
             BufferedOutputStream bout = new BufferedOutputStream(new FileOutputStream(new F
            int b;
            while ((b = bin.read()) != -1) {
                bout.write(b);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

是不是很简单？是不是很刺激？再也不用被Python程序员鄙视了！好了，下面将会详细讲解其实现原理以及内部机制。

动手实践

为了能够配合try-with-resource，资源必须实现 `AutoClosable` 接口。该接口的实现类需要重写 `close` 方法：

```
public class Connection implements AutoCloseable {
    public void sendData() {
        System.out.println("正在发送数据");
    }
    @Override
    public void close() throws Exception {
        System.out.println("正在关闭连接");
    }
}
```

调用类：

```
public class TryWithResource {
```

```
            conn.sendData();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行后输出结果：

```
正在发送数据
正在关闭连接
```

通过结果我们可以看到，close方法被自动调用了。

原理

那么这个是怎么做到的呢？我相信聪明的你们一定已经猜到了，其实，这一切都是编译器大神搞的鬼。我们反编译刚才例子的class文件：

```
public class TryWithResource {
    public TryWithResource() {
    }
    public static void main(String[] args) {
        try {
            Connection e = new Connection();
            Throwable var2 = null;
            try {
                e.sendData();
            } catch (Throwable var12) {
                var2 = var12;
                throw var12;
            } finally {
                if(e != null) {
                    if(var2 != null) {
                        try {
                            e.close();
                        } catch (Throwable var11) {
                            var2.addSuppressed(var11);
                        }
                    } else {
```

```
            }
        } catch (Exception var14) {
            var14.printStackTrace();
        }
    }
}
```

看到没，在第15~27行，编译器自动帮我们生成了finally块，并且在里面调用了资源的close方法，所以例子中的close方法会在运行的时候被执行。

异常屏蔽

我相信，细心的你们肯定又发现了，刚才反编译的代码（第21行）比远古时代写的代码多了一个 `addSuppressed` 。为了了解这段代码的用意，我们稍微修改一下刚才的例子：我们将刚才的代码改回远古时代手动关闭异常的方式，并且在 `sendData` 和 `close` 方法中抛出异常：

```
public class Connection implements AutoCloseable {
    public void sendData() throws Exception {
        throw new Exception("send data");
    }
    @Override
    public void close() throws Exception {
        throw new MyException("close");
    }
}
```

修改main方法：

```
public class TryWithResource {
    public static void main(String[] args) {
        try {
            test();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    private static void test() throws Exception {
        Connection conn = null;
        try {
            conn = new Connection();
            conn.sendData();
```

```
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

运行之后我们发现：

```
basic.exception.MyException: close
        at basic.exception.Connection.close(Connection.java:10)
        at basic.exception.TryWithResource.test(TryWithResource.java:82)
        at basic.exception.TryWithResource.main(TryWithResource.java:7)
        ......
```

好的，问题来了，由于我们一次只能抛出一个异常，所以在最上层看到的是最后一个抛出的异常——也就是 `close` 方法抛出的 `MyException`，而 `sendData` 抛出的 `Exception` 被忽略了。这就是所谓的异常屏蔽。由于异常信息的丢失，异常屏蔽可能会导致某些bug变得极其难以发现，程序员们不得不加班加点地找bug，如此毒瘤，怎能不除！幸好，为了解决这个问题，从Java 1.7开始，大佬们为 `Throwable` 类新增了 `addSuppressed` 方法，支持将一个异常附加到另一个异常身上，从而避免异常屏蔽。那么被屏蔽的异常信息会通过怎样的格式输出呢？我们再运行一遍刚才用try-with-resource包裹的main方法：

```
java.lang.Exception: send data

        at basic.exception.Connection.sendData(Connection.java:5)
        at basic.exception.TryWithResource.main(TryWithResource.java:14)
        ......
        Suppressed: basic.exception.MyException: close
                at basic.exception.Connection.close(Connection.java:10)
                at basic.exception.TryWithResource.main(TryWithResource.java:15)
                ... 5 more
```

可以看到，异常信息中多了一个 `Suppressed` 的提示，告诉我们这个异常其实由两个异常组成，`MyException` 是被Suppressed的异常。可喜可贺！

一个小问题

在使用try-with-resource的过程中，一定需要了解资源的 `close` 方法内部的实现逻辑。否则还是

举个例子，在Java BIO中采用了大量的装饰器模式。当调用装饰器的 `close` 方法时，本质上是调用了装饰器内部包裹的流的 `close` 方法。比如：

```
public class TryWithResource {
    public static void main(String[] args) {
        try (FileInputStream fin = new FileInputStream(new File("input.txt"));
                GZIPOutputStream out = new GZIPOutputStream(new FileOutputStream(new File("
            byte[] buffer = new byte[4096];
            int read;
            while ((read = fin.read(buffer)) != -1) {
                out.write(buffer, 0, read);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

在上述代码中，我们从 `FileInputStream` 中读取字节，并且写入到 `GZIPOutputStream` 中。`GZIPOutputStream` 实际上是 `FileOutputStream` 的装饰器。由于try-with-resource的特性，实际编译之后的代码会在后面带上finally代码块，并且在里面调用fin.close()方法和out.close()方法。我们再来看 `GZIPOutputStream` 类的close方法：

```
public void close() throws IOException {
    if (!closed) {
        finish();
        if (usesDefaultDeflater)
            def.end();
        out.close();
        closed = true;
    }
}
```

我们可以看到，out变量实际上代表的是被装饰的 `FileOutputStream` 类。在调用out变量的 `close` 方法之前， `GZIPOutputStream` 还做了 `finish` 操作，该操作还会继续往 `FileOutputStream` 中写压缩信息，此时如果出现异常，则会 `out.close()` 方法被略过，然而这个才是最底层的资源关闭方法。正确的做法是应该在try-with-resource中单独声明最底层的资源，保证对应的 `close` 方法一定能够被调用。在刚才的例子中，我们需要单独声明每个 `FileInputStream` 以及 `FileOutputStream`：

```
public class TryWithResource {
```

```
                FileOutputStream fout = new FileOutputStream(new File("out.txt"));
                GZIPOutputStream out = new GZIPOutputStream(fout)) {
            byte[] buffer = new byte[4096];
            int read;
            while ((read = fin.read(buffer)) != -1) {
                out.write(buffer, 0, read);
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

由于编译器会自动生成 `fout.close()` 的代码，这样肯定能够保证真正的流被关闭。

总结

怎么样，是不是很简单呢，如果学会了话

参考资料

1. 官方文档

Java

相关热门文章

数据库中间件 MyCAT 源码分析 —— 调试环境搭建

Java公众号_芋道源码_每日更新 21

从团队自研的百万并发中间件系统的内核设计看Java并发性能优化【石杉的架构笔记】

石杉的架构笔记 13 3

BAT 经典算法笔试题 —— 磁盘多路归并排序

老錢 7

教你用认知和人性来做最棒的程序员

刘轶 59 29

兄弟，用大白话给你讲小白都能看懂的分布式系统容错架构【石杉的架构笔记】

石杉的架构笔记 61 9

评论

输入评论...

chenxuxu android

我想请教一下，为什么要 finally 嵌套？上面的资源关闭发生异常有catch，不会影响下面的资源关闭。

7月前 回复

xesam 野生程序员 @ 车来了

这个语法自动关闭的坑，如果出错，挺难查找的

2年前

掘金 首页

2年前

掘金 一个帮助开发者成长的社区 打开应用