利用 Java SE 7 更好地管理资源：不仅仅是语法糖

2011 年 5 月发布

作者： Julien Ponge

本文介绍 Java 7 针对自动资源管理问题给出的解决办法，即 Coin 项目中提出的新语言结构 try-with-resources **语句。**

下载：

: Java SE 7 预览版

: 示例源文件 (zip)

简介

典型的 Java 应用程序可以处理多种类型的资源，如文件、流、套接字和数据库连接。必须谨慎处理这些资源，因为对它们的操作会占用系统资源。因此，需要确保即便在出错的情况下也能释放这些资源。实际上，不正确的资源管理是生产应用中常见的故障根源，常见的错误是，代码中其他位置出现异常后，数据库连接和文件描述符依然处于打开状态。由于操作系统和服务器应用程序通常有一个资源上限，因此在资源耗竭时这会导致应用服务器频繁重启。

针对 Java 中资源管理和异常管理的正确做法已经有了很好的文档说明。对于任何已成功初始化的资源，都需要相应地调用它的 `close()` 方法。这就要求严格遵守 `try/catch/finally` 块的用法，以确保任何从资源打开时起的执行路径最终都能调用一个方法来关闭资源。静态分析工具（如 FindBugs）在识别此类错误时很有帮助。然而通常的情况是，经验不足的开发人员和经验丰富的开发人员都会编写错误的资源管理代码，从而导致资源泄漏甚至更严重的后果。

然而，应该承认，编写正确的资源代码需要大量采用嵌套了 `try/catch/finally` 块的样板代码，您在后文中将看到这一点。正确编写这种代码本身很快就会成为难题。与此同时，Python 和 Ruby 等其他编程语言已经提供了语言级工具（即*自动资源管理*）来解决这一问题。

本文介绍 Java Platform, Standard Edition (Java SE) 7 针对自动资源管理问题给出的解决办法，即 Coin 项目中提出的新语言结构 try-with-resources 语句。我们将看到，该语句的好处远不止像 Java SE 5 的循环语句增强一样地加入更多语法糖。实际上，异常会彼此屏蔽，从而导致有时难以找到问题的根源。

本文首先将概述资源和异常管理，然后将从 Java 开发人员的视角介绍 `try-with-resources` 语句的要点。随后将展示如何准备一个类，使之支持此类语句。接下来，将讨论异常屏蔽的问题，以及 Java SE 7 做了哪些改变来解决此类问题。最后，本文将揭开语言扩展背后语法糖的神秘面纱，进行讨论并给出结论。

注意：本文所述示例的源代码可从这里下载：sources.zip

管理资源和异常

我们先从下面节选的一段代码开始：

```
private void incorrectWriting() throws IOException {
       DataOutputStream out = new DataOutputStream(new FileOutputStream("data"));
       out.writeInt(666);
       out.writeUTF("Hello");
       out.close();
   }
```

乍一看，此方法似乎不会造成什么损害：它打开一个名为 `data` 的文件，随后写入一个整数和一个字符串。`java.io` 程序包中对流类的设计使之能够通过修饰设计模式进行组合。

例如，我们可以在 `DataOutputStream` 与 `FileOutputStream` 之间添加一个用于压缩数据的输出流。关闭一个流时，也会关闭它所修饰的流。重新回到这个示例，在对 `DataOutputStream` 的实例调用 `close()` 时，同样也会调用 `FileOutputStream` 的 `close()` 方法。

然而，关于在这种方法中对 `close()` 方法的调用存在一个严重的问题。假设在写入整数或字符串时因底层文件系统已满而抛出一个异常。那么，将不再有机会调用 `close()` 方法。

这对于 `DataOutputStream` 不是什么严重的问题，因为它仅对 `OutputStream` 实例进行操作，用于将基本数据类型解码并把它们写入字节数组中。真正的问题在于 `FileOutputStream`，因为它在一个文件描述符内部保留了一个操作系统资源，仅在调用 `close()` 时才能释放该资源。因此，这种方法会泄漏资源。

这个问题对短时运行的程序基本无碍，但对于建立在 Java Platform, Enterprise Edition (Java EE) 应用服务器上的、长期运行的应用程序来说，由于达到了底层操作系统所允许打开的文件描述符的最大数量，可能会导致整个服务器重启。

一种正确地重写前述方法的方式如下：

```
private void correctWriting() throws IOException {
       DataOutputStream out = null;
       try {
           out = new DataOutputStream(new FileOutputStream("data"));
           out.writeInt(666);
           out.writeUTF("Hello");
       } finally {
           if (out != null) {
               out.close();
           }
       }
   }
```

在任何情况下，抛出的异常都会传播给方法的调用者，但 `try` 块后的 `finally` 块能确保调用数据输出流的 `close()` 方法。这相应地确保了底层文件输出流的 `close()` 方法同样获得调用，从而正确释放与文件关联的操作系统资源。

适合缺乏耐心者的 try-with-resources **语句**

不可否认，前例中存在大量确保正确关闭资源的样板代码。如果存在更多的流、网络套接字或 Java 数据库连接 (JDBC) 连接，此类样板代码会使您更难以阅读一个方法的业务逻辑。更糟糕的是，它需要开发人员的自律，因为在编写错误处理和资源关闭逻辑时非常容易出错。

与此同时，其他编程语言已经引入了简化此类情况处理的结构。例如，上一个方法可以使用 Ruby 写成如下所示的样子：

```
def writing_in_ruby
       File.open('rdata', 'w') do |f|
           f.write(666)
           f.write("Hello")
       end
   end
```

用 Python 可写成这个样子：

```
def writing_in_python():
       with open("pdata", "w") as f:
           f.write(str(666))
           f.write("Hello")
```

在 Ruby 中，`File.open` 执行了一个代码块，即便在该块的执行出现异常时也能确保关闭所打开的文件。

Python 的示例与之相似，其特殊的 `with` 语句采用一个带有 `close` 方法和一个代码块的对象。同样，无论是否抛出异常，都能确保正确关闭资源。

Java SE 7 在 Coin 项目中引入了类似的语言结构。之前的示例可重写为如下所示：

```
private void writingWithARM() throws IOException {
       try (DataOutputStream out
               = new DataOutputStream(new FileOutputStream("data"))) {
           out.writeInt(666);
           out.writeUTF("Hello");
       }
   }
```

新结构扩展了 `try` 块，按照与 `for` 循环相似的方式声明了资源。在 `try` 块中声明打开的任何资源都会关闭。因此，这个新结构使您不必配对使用 `try` 块与对应的 `finally` 块，后者专用于正确的资源管理。使用分号分隔每个资源，例如：

```
try (
       FileOutputStream out = new FileOutputStream("output");
       FileInputStream  in1 = new FileInputStream("input1");
       FileInputStream  in2 = new FileInputStream("input2")
   ) {
       // Do something useful with those 3 streams!
   }   // out, in1 and in2 will be closed in any case
```

最后需要提到的是，这样一条 `try-with-resources` 语句后面可能跟 `catch` 和 `finally` 块，就像 Java SE 7 之前的常规 try 语句一样。

构造可自动关闭的类

您可能已经猜到了，`try-with-resources` 语句无法管理所有类。Java SE 7 引入了一个新接口 `java.lang.AutoCloseable`。它的作用就是提供一个名为 `close()` 的 void 方法，该方法可能抛出一个检查到的异常 (`java.lang.Exception`)。任何希望在 `try-with-resources` 语句中使用的类都应实现该接口。强烈建议，实现的类和子接口应声明一种比 `java.lang.Exception` 更精确的异常类型，当然，更好的情况是，如果调用 `close()` 方法不会导致失败，就根本不用声明异常类型。

此类 `close()` 方法已经进行了改进，包含在标准 Java SE 运行时环境的许多类中，这些类包括 `java.io`、`java.nio`、`javax.crypto`、`java.security`、`java.util.zip`、`java.util.jar`、`javax.net` 和 `java.sql packages`。这种方法的主要优点在于，现有代码可继续像以前那样工作，而新代码可以轻松利用 `try-with-resources` 语句。

我们来看看以下示例：

```
public class AutoClose implements AutoCloseable {

       @Override
       public void close() {
           System.out.println(">>> close()");
           throw new RuntimeException("Exception in close()");
       }

       public void work() throws MyException {
           System.out.println(">>> work()");
           throw new MyException("Exception in work()");
       }

       public static void main(String[] args) {
           try (AutoClose autoClose = new AutoClose()) {
               autoClose.work();
           } catch (MyException e) {
               e.printStackTrace();
           }
       }
   }
class MyException extends Exception {

       public MyException() {
           super();
       }

       public MyException(String message) {
           super(message);
       }
   }
```

`AutoClose` 类实现了 `AutoCloseable`，因此可用作 try-with-resources 语句的一部分，如 `main()` 方法中所示。我们特意添加了一些控制台输出，并在该类的 `work()` 和 `close()` 方法中抛出异常。运行该程序将产生以下输出：

```
>>> work()
   >>> close()
   MyException: Exception in work()
          at AutoClose.work(AutoClose.java:11)
          at AutoClose.main(AutoClose.java:16)
          Suppressed: java.lang.RuntimeException: Exception in close()
                 at AutoClose.close(AutoClose.java:6)
                 at AutoClose.main(AutoClose.java:17)
```

输出显然证实了在进入应处理异常的 `catch` 块之前，确实调用了 `close()`。然而，Java 开发人员意外地发现，在 Java SE 7 中出现了以“`Suppressed:(…)`”为前缀的异常堆栈跟踪行。它相当于 `close()` 方法抛出的异常，但在 Java SE 7 之前，您可能从未遇到过这种形式的堆栈跟踪。这是怎么回事？

异常屏蔽

为了理解前面示例中所发生的情况，让我们暂时抛开 `try-with-resources` 语句，手动重新编写正确的资源管理代码。首先，我们提取将由 `main` 方法调用的以下静态方法：

```
public static void runWithMasking() throws MyException {
       AutoClose autoClose = new AutoClose();
       try {
           autoClose.work();
       } finally {
           autoClose.close();
       }
   }
```

随后，相应地改造 `main` 方法：

```
public static void main(String[] args) {
       try {
           runWithMasking();
       } catch (Throwable t) {
           t.printStackTrace();
       }
   }
```

现在，运行程序后会给出以下输出：

```
>>> work()
   >>> close()
   java.lang.RuntimeException: Exception in close()
          at AutoClose.close(AutoClose.java:6)
          at AutoClose.runWithMasking(AutoClose.java:19)
          at AutoClose.main(AutoClose.java:52)
```

这段代码是在 Java SE 7 之前惯用的正确资源管理方法，它显示了一个异常被另一个异常屏蔽的问题。实际上，调用 `runWithMasking()` 方法的客户端代码将获知 `close()` 方法抛出一个异常，尽管实际上是 `work()` 方法先抛出了异常。

然而，一次只能抛出一个异常，这就意味着在处理异常时即使正确的代码也会遗漏一些信息。如果一个重要异常被关闭资源时进而抛出的另一个异常所屏蔽，开发人员就要浪费大量时间进行调试。敏锐的读者可能会对此提出异议，毕竟异常是可以嵌套的。然而，仅应对彼此之间存在因果关系的异常使用嵌套，通常将一个低级异常包装在位于应用程序架构较高层的异常中。一个很好的例子是 JDBC 驱动程序将套接字异常包装在一个 JDBC 连接中。我们的示例中实际上有两个异常：一个在 `work()` 中，一个在 `close()` 中，两者之间绝对不存在因果关系。

支持“被抑制的”异常

由于异常屏蔽在实际中是如此重要的一个问题，因此 Java SE 7 扩展了异常，这样就可以将“被抑制的”异常附加到主异常上。我们之前所说的“屏蔽的”异常实际上就是一个被抑制并附加到主异常的异常。

`java.lang.Throwable` 的扩展如下：

`public final void addSuppressed(Throwable exception)` 将一个被抑制的异常附加到另一个异常上，从而避免异常屏蔽。
`public final Throwable[] getSuppressed()` 获取添加到一个异常中的被抑制的异常。
这些扩展是专门为支持 `try-with-resources` 语句和修复异常屏蔽问题而引入的。

回到之前的 `runWithMasking()` 方法，我们在考虑支持被抑制异常的前提下重新编写此方法：

```
public static void runWithoutMasking() throws MyException {
       AutoClose autoClose = new AutoClose();
       MyException myException = null;
       try {
           autoClose.work();
       } catch (MyException e) {
           myException = e;
           throw e;
       } finally {
           if (myException != null) {
               try {
                   autoClose.close();
               } catch (Throwable t) {
                   myException.addSuppressed(t);
               }
           } else {
               autoClose.close();
           }
       }
   }
```

很明显，这里使用了大量代码，其目的仅仅是正确处理一个可自动关闭类的两个异常抛出方法！一个局部变量用于捕获主异常，也就是 `work()` 方法可能抛出的异常。如果抛出这样一个异常，则捕获该异常，随后立即再次抛出，以便将其余工作委托给 `finally` 块。

进入 `finally` 块，检查对主异常的引用。如果抛出了一个异常，则 `close()` 方法可能抛出的异常将作为被抑制异常附加到此异常。否则将调用 `close()` 方法，如果该方法抛出一个异常，那么该异常实际上就是主异常，因此不会屏蔽其他异常。

我们来运行使用这个新方法修改后的程序：

```
>>> work()
   >>> close()
   MyException: Exception in work()
          at AutoClose.work(AutoClose.java:11)
          at AutoClose.runWithoutMasking(AutoClose.java:27)
          at AutoClose.main(AutoClose.java:58)
          Suppressed: java.lang.RuntimeException: Exception in close()
                 at AutoClose.close(AutoClose.java:6)
                 at AutoClose.runWithoutMasking(AutoClose.java:34)
                 ... 1 more
```

正如您所见到的那样，我们手动重现了前文所述的 `try-with-resources` 语句的行为。

语法糖揭秘

我们实现的 `runWithoutMasking()` 方法通过正确关闭资源以及防止异常屏蔽来重现了 `try-with-resources` 语句的行为。实际上，Java 编译器将以下方法的代码扩展为与 `runWithoutMasking()` 代码一致的情形，使用了 `try-with-resources` 语句：

```
public static void runInARM() throws MyException {
       try (AutoClose autoClose = new AutoClose()) {
           autoClose.work();
       }
   }
```

可以通过反编译来进行检查。虽然我们可以使用 Java Development Kit (JDK) 二进制工具中包含的 `javap` 来比较字节码，但我们把它当作一个字节码到 Java 源代码的反编译器来使用。`JD-GUI` 工具提取出的 `runInARM()` 代码如下（经过重新排版）：

```
public static void runInARM() throws MyException {
       AutoClose localAutoClose = new AutoClose();
       Object localObject1 = null;
       try {
           localAutoClose.work();
       } catch (Throwable localThrowable2) {
           localObject1 = localThrowable2;
           throw localThrowable2;
       } finally {
           if (localAutoClose != null) {
               if (localObject1 != null) {
                   try {
```

```
                        localAutoClose.close();
                    } catch (Throwable localThrowable3) {
                        localObject1.addSuppressed(localThrowable3);
                    }
                } else {
                    localAutoClose.close();
                }
            }
        }
    }
```

可以看到，我们手动编写的代码使用的资源管理画布与编译器根据 `try-with-resources` 语句推断出的画布相同。还应注意到，编译器处理了可能为 `null` 的资源引用，在 `finally` 块中添加了额外的 `if` 语句来检查给定资源是否为 `null`，从而避免对 `null` 引用调用 `close()` 时的空指针异常。我们的手动实现中并未这样做，因为资源不可能为 `null`。但编译器会系统性地生成此类代码。

现在，我们来考虑另外一个示例，这次涉及三个资源：

```
private static void compress(String input, String output) throws IOException {
       try(
           FileInputStream fin = new FileInputStream(input);
           FileOutputStream fout = new FileOutputStream(output);
           GZIPOutputStream out = new GZIPOutputStream(fout)
       ) {
           byte[] buffer = new byte[4096];
           int nread = 0;
           while ((nread = fin.read(buffer)) != -1) {
               out.write(buffer, 0, nread);
           }
       }
   }
```

这个方法操纵三个资源来压缩一个文件：一个流用于读取、一个流用于压缩、一个流指向输出文件。从资源管理的视角来看，这样的代码是正确的。在 Java SE 7 之前，您不得不类似于下面的代码，这段代码是再次使用 `JD-GUI` 来反编译包含此方法的类获得的：

```
private static void compress(String paramString1, String paramString2)
          throws IOException {
       FileInputStream localFileInputStream = new FileInputStream(paramString1);
    Object localObject1 = null;
       try {
           FileOutputStream localFileOutputStream = new FileOutputStream(paramString2);
        Object localObject2 = null;
           try {
               GZIPOutputStream localGZIPOutputStream = new GZIPOutputStream(localFileOutputStream);
            Object localObject3 = null;
               try {
                   byte[] arrayOfByte = new byte[4096];
                   int i = 0;
                   while ((i = localFileInputStream.read(arrayOfByte)) != -1) {
                       localGZIPOutputStream.write(arrayOfByte, 0, i);
                   }
               } catch (Throwable localThrowable6) {
                   localObject3 = localThrowable6;
                   throw localThrowable6;
               } finally {
                   if (localGZIPOutputStream != null) {
                       if (localObject3 != null) {
                           try {
                               localGZIPOutputStream.close();
                           } catch (Throwable localThrowable7) {
                               localObject3.addSuppressed(localThrowable7);
                           }
                       } else {
                           localGZIPOutputStream.close();
                       }
                   }
               }
           } catch (Throwable localThrowable4) {
               localObject2 = localThrowable4;
               throw localThrowable4;
           } finally {
               if (localFileOutputStream != null) {
                   if (localObject2 != null) {
                       try {
                           localFileOutputStream.close();
                       } catch (Throwable localThrowable8) {
                           localObject2.addSuppressed(localThrowable8);
                       }
                   } else {
                       localFileOutputStream.close();
                   }
               }
           }
       } catch (Throwable localThrowable2) {
           localObject1 = localThrowable2;
           throw localThrowable2;
       } finally {
           if (localFileInputStream != null) {
               if (localObject1 != null) {
                   try {
                       localFileInputStream.close();
                   } catch (Throwable localThrowable9) {
                       localObject1.addSuppressed(localThrowable9);
                   }
               } else {
                   localFileInputStream.close();
               }
           }
       }
   }
```

对于这样的示例来说，Java SE 7 中的 `try-with-resources` 语句的好处是不言而喻的：要编写的代码很少、代码的可读性更高，最后但并非最不重要的是，代码不会泄漏资源！

讨论

`java.lang.AutoCloseable` 接口中 `close()` 方法的定义意味着可能抛出 `java.lang.Exception`。然而，前面的 `AutoClose` 示例对该方法进行声明，但并未提及任何检查到的异常，这是我们有意为之，部分是为了说明异常屏蔽。

可自动关闭类的规范建议避免抛出 `java.lang.Exception`，优先使用具体的受检异常，如果预计 `close()` 方法不会失败，就不必提及任何受检异常。此外还建议，不要声明任何不应被抑制的异常，`java.lang.InterruptedException` 就是最好的例子。实际上，抑制该异常并将其附加到另一个异常可能会导致忽略线程中断事件，使应用程序处于不一致的状态。

一个关于 `try-with-resources` 语句使用的合理问题是，与手动编写的正确资源管理代码相比，其对性能的影响如何。实际上并不存在性能方面的影响，因为编译器为所有异常的正确处理推断出尽可能少的正确代码，正如我们在之前示例中通过反编译所演示的那样。

最后要说的是，`try-with-resources` 语句是语法糖，就像 Java SE 5 为扩展迭代器循环而引入的增强 `for` 循环一样。

话虽如此，我们仍然可以限制 `try-with-resources` 语句扩展的复杂程度。一般来说，一个 `try` 块声明的资源越多，所生成的代码也就越复杂。之前的 `compress()` 方法可重写成仅使用两个资源而不是三个，从而生成更精简的异常处理块：

```
private static void compress(String input, String output) throws IOException {
       try(
           FileInputStream fin = new FileInputStream(input);
           GZIPOutputStream out = new GZIPOutputStream(new FileOutputStream(output))
       ) {
           byte[] buffer = new byte[4096];
           int nread = 0;
           while ((nread = fin.read(buffer)) != -1) {
               out.write(buffer, 0, nread);
           }
       }
   }
```

就像 Java 中出现 `try-with-resources` 语句之前的情况一样，一般经验是，开发人员在链接资源实例化时应始终明白需要取舍的东西。为此，最好的方法就是阅读每个资源的 `close()` 方法的规范，理解其语义和影响。

回到本文最初的 writingWithARM() 示例，链接是安全的，因为 `DataOutputStream` 不可能在 `close()` 上抛出异常。但是，这不适用于最后一个示例，因为 `GZIPOutputStream` 会尝试写入其余压缩数据作为 `close()` 方法的一部分。如果在写入压缩文件时，抛出异常的时间较早，`GZIPOutputStream` 中的 `close()` 方法更有可能进而抛出另一个异常，导致不会调用 `FileOutputStream` 中的 `close()` 方法，从而泄漏一个文件描述符资源。

好的做法是在 `try-with-resources` 语句中为每一个持有关键系统资源（如文件描述符、套接字或者 JDBC 连接）的每个资源进行单独声明，必须确保 `close()` 方法最终得到调用。否则，如果相关资源 API 允许，选择链接分配就不仅是一种惯例：在防止资源泄漏的同时还能得到更为紧凑的代码。

结论

本文介绍了 Java SE 7 中一种新的用于安全管理资源的语言结构。这种扩展带来的影响不仅仅是更多的语法糖。事实上，它能位开发人员生成了正确的代码，消除了编写容易出错的样板代码的需要。更重要的是，这种变化还伴随着将一个异常附加到另一个异常的改进，从而为众所周知的异常彼此屏蔽问题提供了完善的解决方案。

另请参见
下面是其他一些资源：

Java SE 7 预览版： http://jdk7.java.net/preview/
Java SE 7 API： http://download.java.net/jdk7/docs/api/
Joshua Bloch 针对自动资源管理给出的最初建议，2009 年 2 月 27 日：http://mail.openjdk.java.net/pipermail/coin-dev/2009-February/000011.html 和 https://docs.google.com/View?id=ddv8ts74_3fs7483dp
Coin 项目：更新的 ARM 规范，2010 年 7 月 15 日： http://blogs.sun.com/darcy/entry/project_coin_updated_arm_spec
Coin 项目：JSR 334 EDR 现已提供，2010 年 1 月 11 日： http://blogs.sun.com/darcy/entry/project_coin_edr
Coin 项目：如何终止 try-with-resources，2011 年 1 月 31 日： http://blogs.sun.com/darcy/entry/project_coin_how_to_terminate
Coin 项目：空资源上的 try-with-resources，2011 年 2 月 16 日： http://blogs.sun.com/darcy/entry/project_coin_null_try_with
Coin 项目：JSR 334 进入公示期，2011 年 3 月 24 日： http://blogs.sun.com/darcy/entry/project_coin_jsr_334_pr
Coin 项目： http://openjdk.java.net/projects/coin/
JSR334 早期草案审议： http://jcp.org/aboutJava/communityprocess/edr/jsr334/index.html
JSR334 进入公示期： http://jcp.org/aboutJava/communityprocess/pr/jsr334/index.html
Java *难题：误区、陷阱和极端情况*，作者：Joshua Bloch 和 Neal Gafter（Addison-Wesley Professional，2005）
高效 Java *编程语言指南*，作者：Joshua Bloch（Addison-Wesley Professional，2001）
修饰设计模式： http://en.wikipedia.org/wiki/Decorator_pattern
静态代码分析工具 FindBugs： http://findbugs.sourceforge.net/
Java 字节码反编译器 JD-GUI： http://java.decompiler.free.fr/?q=jdgui
Python： http://www.python.org/
Ruby：http://www.ruby-lang.org/

关于作者

Julien Ponge 是一位长期从事开源工作的技术高人。他创建了 IzPack 安装程序框架，还参与了其他几个项目，包括与 Sun Microsystems 合作的 GlassFish 应用服务器。他拥有 UNSW Sydney 和 UBP Clermont-Ferrand 的计算机科学博士学位，目前是 INSA de Lyon 计算机科学与工系程的副教授，并且是 INRIA Amazones 团队的一名研究人员。由于熟练掌握行业和学术两个领域中的语言，因此他正在积极推进这两个领域之间更进一步的协作。