Java SE 7을 통한 향상된 리소스 관리 : 구문 설탕을 넘어서

Julien Ponge 작성
2011 년 5 월

이 기사에서는 *try-with-resources* 문 이라고하는 Project Coin의 일부로 제안 된 새로운 언어 구조의 형태로 자동 리소스 관리 문제에 대한 Java 7 답변을 제공합니다 .

다운로드 :

Java SE 7 미리보기
예제 소스 파일 (zip)

소개

일반적인 Java 애플리케이션은 파일, 스트림, 소켓 및 데이터베이스 연결과 같은 여러 유형의 리소스를 조작합니다. 이러한 리소스는 운영을위한 시스템 리소스를 확보하기 때문에 매우주의해서 처리해야합니다. 따라서 오류가 발생하더라도 해제되도록해야합니다. 실제로 잘못된 리소스 관리는 프로덕션 응용 프로그램에서 오류의 일반적인 원인이며, 일반적인 함정은 코드의 다른 곳에서 예외가 발생한 후에도 열린 상태로 남아있는 데이터베이스 연결 및 파일 설명자입니다. 이로 인해 운영 체제 및 서버 응용 프로그램은 일반적으로 자원에 대한 상한 제한이 있기 때문에 자원이 고갈 될 때 응용 프로그램 서버가 자주 다시 시작됩니다.

Java의 자원 및 예외 관리에 대한 올바른 관행은 잘 문서화되어 있습니다. 성공적으로 초기화 된 리소스의 경우 해당 `close()` 메서드에 대한 해당 호출 이 필요합니다. 이를 위해서는 `try/catch/finally`리소스 열기의 실행 경로가 결국 이를 닫는 메서드에 대한 호출에 도달하도록 블록을 규칙적으로 사용해야합니다. FindBugs와 같은 정적 분석 도구는 이러한 유형의 오류를 식별하는 데 큰 도움이됩니다. 그러나 경험이없는 개발자와 경험이있는 개발자 모두 리소스 관리 코드가 잘못되어 기껏해야 리소스 누수가 발생하는 경우가 많습니다.

그러나 리소스에 대한 올바른 코드를 작성하려면 중첩 된 `try/catch/finally`블록 형태의 상용구 코드가 많이 필요하다는 점을 알아야합니다 . 이러한 코드를 올바르게 작성하는 것은 그 자체로 문제가됩니다. 한편, Python 및 Ruby와 같은 다른 프로그래밍 언어는 이 문제를 해결하기 위해 자동 리소스 관리 라는 언어 수준의 기능을 제공하고 있습니다 .

이 기사는 프로젝트 코인의 일부로 제안되고 *try-with-resources* 문 이라고하는 새로운 언어 구조의 형태로 자동 리소스 관리 문제에 대한 Java SE (Java Platform, Standard Edition) 7 답변을 제공합니다 . 우리가 보게 될 것처럼, Java SE 5의 향상된 for 루프처럼 단순한 구문 적 설탕 이상의 의미를 지닙니다. 실제로 예외는 서로를 가릴 수있어 근본 문제 원인을 식별하는 것이 때때로 디버그하기 어렵게 만듭니다.

이 기사 `try-with-resources`는 Java 개발자 관점에서 명령문 의 필수 사항을 소개하기 전에 리소스 및 예외 관리에 대한 개요로 시작 합니다. 그런 다음 이러한 명령문을 지원하기 위해 클래스를 준비하는 방법을 보여줍니다. 다음으로 예외 마스킹의 문제와이를 해결하기 위해 Java SE 7이 어떻게 진화했는지에 대해 설명합니다. 마지막으로, 언어 확장 뒤에 있는 통사론 적 설탕을 이해하고 토론과 결론을 제공합니다.

참고 :이 기사에서 설명하는 예제의 소스 코드는 sources.zip에서 다운로드 할 수 있습니다.

리소스 및 예외 관리

다음 코드 발췌부터 시작하겠습니다.

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

부

첫눈에이 방법은별로 해를 끼치 지 않습니다.라는 파일을 열고 `data`정수와 문자열을 씁니다. `java.io`패키지 의 스트림 클래스 디자인은 데코레이터 디자인 패턴을 사용하여 결합 될 수 있도록합니다.

예를 들어, a `DataOutputStream`와 `FileOutputStream`. 스트림이 닫히면 장식중인 스트림도 닫힙니다. 예제로 다시 돌아가서, 때 `close()`의 인스턴스라고 `DataOutputStream`그래서입니다, `close()`지분법 `FileOutputStream`.

그러나이 메서드에는 `close()`메서드 호출과 관련하여 심각한 문제가 있습니다. 기본 파일 시스템이 가득 차서 정수 또는 문자열을 쓰는 동안 예외가 발생했다고 가정합니다. 그러면 `close()`메서드가 호출 될 가능성이 없습니다.

기본 데이터 유형을 인코딩하고 바이트 배열로 쓰기 위해의 `DataOutputStream`인스턴스에서만 작동하기 때문에,는 별로 문제가되지 않습니다 `OutputStream`. 실제 문제는 on입니다. `FileOutputStream`, `close()`호출 될 때만 해제되는 파일 설명자에 운영 체제 리소스를 내부적으로 보유하고 있기 때문 입니다. 따라서이 방법은 리소스를 누출합니다.

이 문제는 수명이 짧은 프로그램의 경우 대부분 무해하지만 Java EE (Java Platform, Enterprise Edition) 애플리케이션 서버에서 볼 수 있듯이 장기 실행 애플리케이션의 경우 전체 서버를 다시 시작해야 할 수 있습니다. 기본 운영 체제에서 허용하는 최대 열린 파일 설명자 수에 도달하기 때문입니다.

이전 방법을 다시 작성하는 올바른 방법은 다음과 같습니다.

```
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();
           }
       }
   }
```

부

모든 경우에 throw 된 예외가 메서드 호출자에게 전파되지만 `finally`블록 뒤의 `try`블록은 `close()`데이터 출력 스트림 의 메서드가 호출되도록합니다. 그러면 기본 파일 출력 스트림의 `close()`메서드도 호출되어 파일과 관련된 운영 체제 리소스를 적절하게 해제 할 수 있습니다.

참을성이없는 사람들을위한 *try-with-resources* 진술

이전 예제에는 리소스가 제대로 닫혔는지 확인하기 위해 많은 상용구 코드가 있습니다. 더 많은 스트림, 네트워크 소켓 또는 JDBC (Java Database Connectivity) 연결을 사용하면 이러한 상용구 코드는 메서드의 비즈니스 논리를 읽기 어렵게 만듭니다. 더 나쁜 것은 오류 처리 및 리소스 닫기 논리를 잘못 작성하기 쉽기 때문에 개발자의 규율이 필요하다는 것입니다.

그 동안 다른 프로그래밍 언어에서는 이러한 경우 처리를 단순화하기위한 구성을 도입했습니다. 예를 들어, 이전 메소드는 Ruby에서 다음과 같이 작성됩니다.

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

부

그리고 파이썬에서는 다음과 같이 작성됩니다.

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

부

Ruby에서 `File.open`메서드는 실행할 코드 블록을 가져 와서 블록 실행에서 예외가 발생하더라도 열린 파일이 닫히도록합니다.

Python 예제는 특수 `with`문이 `close`메서드와 코드 블록 이있는 객체를 취 한다는 점에서 유사 합니다. 다시 말하지만 예외가 발생했는지 여부에 관계없이 적절한 리소스 닫기를 보장합니다.

Java SE 7은 Project Coin의 일부로 유사한 언어 구조를 도입했습니다. 예제는 다음과 같이 다시 작성할 수 있습니다.

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

부

새로운 구조 `try`는 `for`루프를 사용 하는 경우처럼 리소스를 선언하기 위해 블록을 확장 합니다 . `try`블록 열기 내에서 선언 된 모든 리소스 가 닫힙니다. 따라서 새로운 구성 은 적절한 리소스 관리에 전념 `try`하는 해당 `finally`블록 과 블록 을 쌍으로 연결하지 않아도됩니다 . 세미콜론은 각 리소스를 구분합니다. 예를 들면 다음과 같습니다.

```
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
```

부

마지막으로 Java SE 7 이전의 일반 try 문과 마찬가지로 이러한 `try-with-resources`문 뒤에 `catch`및 `finally`블록이 올 수 있습니다 .

자동 종료 가능한 클래스 만들기

이미 짐작했듯이 `try-with-resources`명령문은 모든 클래스를 관리 할 수 없습니다. 호출 된 새 인터페이스 `java.lang.AutoCloseable`가 Java SE 7에 도입되었습니다 . `close()`확인 된 예외 (`java.lang.Exception`)를 throw 할 수 있는 void 메서드를 제공하는 것뿐입니다 . `try-with-resources`명령문 에 참여하려는 모든 클래스 는이 인터페이스를 구현해야합니다. 클래스 및 하위 인터페이스를 구현하는 것이보다 정확한 예외 유형을 `java.lang.Exception`선언하거나 호출 `close()`이 실패하지 않아야 하는 경우 예외 유형을 전혀 선언하지 않는 것이 좋습니다 .

이러한 `close()`메서드는 `java.io, java.nio, javax.crypto, java.security, java.util.zip, java.util.jar, javax.net,` 및을 포함하여 표준 Java SE 런타임 환경의 여러 클래스에 개조되었습니다 `java.sql packages`. 이 접근 방식의 가장 큰 장점은 기존 코드가 이전과 동일하게 계속 작동하는 반면 새 코드는 `try-with-resources`명령문을 쉽게 활용할 수 있다는 것입니다 .

다음 예를 살펴 보겠습니다.

```
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);
        }
    }
```

부

이 `AutoClose`클래스 `AutoCloseable`는 `main()`메소드에 설명 된대로 try-with-resources 문의 일부로 구현 되므로 사용할 수 있습니다 . 의도적으로 일부 콘솔 출력을 추가 했으며 클래스 의 `work()`및 `close()`메서드 모두에서 예외를 throw 합니다. 프로그램을 실행하면 다음과 같은 출력이 생성됩니다.

```
>>> 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)
```

출력 은 예외를 처리해야하는 블록에 `close()` 들어가기 전에 실제로 호출 되었음을 분명히 증명합니다 `catch`. 그러나 Java SE 7을 발견 한 Java 개발자는 " `Suppressed: (…)` " 접두사가 붙은 예외 스택 추적 줄을보고 놀랄 수 있습니다 . `close()` 메소드에서 발생한 예외와 일치 하지만 Java SE 7 이전에는 이러한 형태의 스택 추적을 만날 수 없었습니다. 여기서 무슨 일이 벌어지고 있습니까?

예외 마스킹

이전 예에서 무슨 일이 일어 났는지 이해 `try-with-resources`하기 위해 잠시 문을 제거 하고 올바른 리소스 관리 코드를 수동으로 다시 작성 하겠습니다 . 먼저 메서드에서 호출 할 다음 정적 메서드를 추출해 보겠습니다 `main`.

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

부

그런 다음 `main` 그에 따라 메서드를 .

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

부

이제 프로그램을 실행하면 다음과 같은 출력이 제공됩니다.

```
>>> 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)
```

Java SE 7 이전의 적절한 리소스 관리를위한 관용어 인이 코드는 한 예외가 다른 예외에 의해 마스킹되는 문제를 보여줍니다. 실제로 `runWithMasking()` 메서드에 대한 클라이언트 코드는 메서드에서 throw되는 예외에 대해 알림을 받지만 `close()` 실제로는 첫 번째 예외가 메서드에서 throw되었습니다.`work()` .

그러나 한 번에 하나의 예외 만 발생할 수 있습니다. 즉, 올바른 코드라도 예외를 처리하는 동안 정보가 누락됩니다. 리소스를 닫는 동안 추가 예외가 발생하여 주 예외가 마스크되면 개발자는 디버깅 시간을 상당히 잃게됩니다. 기민한 독자는 결국 예외가 중첩 될 수 있기 때문에 그러한 주장에 의문을 제기 할 수 있습니다. 그러나 중첩 된 예외는 일반적으로 응용 프로그램 아키텍처의 상위 계층을 대상으로하는 하위 수준 예외를 하나의 예외와 다른 예외 사이의 인과 관계에 사용해야합니다. 좋은 예는 소켓 예외를 JDBC 연결로 래핑하는 JDBC 드라이버입니다. 여기에는 실제로 두 가지 예외가 있습니다. 하나는에 `work()`, 하나는 `close()`에 있으며 둘 사이에는 인과 관계가 전혀 없습니다.

"억제 된"예외 지원

예외 마스킹은 실제로 매우 중요한 문제이기 때문에 Java SE 7은 "억제 된"예외가 기본 예외에 첨부 될 수 있도록 예외를 확장합니다. 이전에 "마스킹 된"예외라고 불렀던 것은 실제로는 억제되고 기본 예외에 첨부되는 예외입니다.

의 확장자 `java.lang.Throwable`는 다음과 같습니다.

- `public final void addSuppressed(Throwable exception)` 예외 마스킹을 피하기 위해 억제 된 예외를 다른 예외에 추가합니다.
- `public final Throwable[] getSuppressed()` 예외에 추가 된 억제 된 예외를 가져옵니다.

이러한 확장은 특히 `try-with-resources` 문 예외 마스킹 문제를 수정 .

이전 `runWithMasking()`방법으로 돌아가서 억제 된 예외에 대한 지원을 염두에두고 다시 작성하겠습니다.

```
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();
            }
        }
    }
```

부

분명히 이것은 단일 자동 종료 가능 클래스의 두 가지 예외 발생 메서드를 적절하게 처리하기위한 상당한 양의 코드를 나타냅니다! 지역 변수는 기본 예외, 즉 `work()` 메서드가 throw 할 수 있는 예외를 캡처하는 데 사용됩니다 . 이러한 예외가 발생하면 캡처 된 다음 즉시 다시 발생하여 나머지 작업을`finally` 블록 .

`finally`블록을 입력하면 1 차 예외에 대한 참조가 확인됩니다. 예외가 throw되면 `close()` 메서드가 throw 할 수있는 예외가 억제 된 예외로 첨부됩니다. 그렇지 않으면`close()` 메서드가 호출되고 예외가 발생하면 실제로 기본 예외이므로 다른 예외를 마스킹하지 않습니다.

이 새로운 방법으로 수정 된 프로그램을 실행 해 보겠습니다.

```
>>> 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
```

보시다시피 `try-with-resources`이전 에 성명서 의 동작을 수동으로 재현했습니다 .

Syntantic Sugar Demystified

`runWithoutMasking()` 우리가 구현 한 방법은 `try-with-resources`리소스를 적절하게 닫고 예외 마스킹을 방지하여 명령문 의 동작을 재현합니다 . 실제로 Java 컴파일러 `runWithoutMasking()` 는 `try-with-resources`명령문을 사용하는 다음 메소드 의 코드와 일치하는 코드로 확장됩니다 .

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

부

이것은 디 컴파일을 통해 확인할 수 있습니다. `javap`JDK (Java Development Kit) 바이너리 도구의 일부인를 사용하여 바이트 코드를 비교할 수 있지만 대신 바이트 코드-자바 소스 코드 디 컴파일러를 사용하겠습니다. 의 코드는 도구에 `runInARM()` 의해 다음과 같이 추출됩니다 `JD-GUI`(형식 변경 후).

```
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();
               }
           }
```

```
        }
    }
```

부

As we can see, the code that we manually wrote shares the same resource management canvas as the one inferred by the compiler on a `try-with-resources` statement. It should also be noted that the compiler handles the case of possibly `null` resource references to avoid null pointer exceptions when invoking `close()` on a `null` reference by adding extra `if` statements in `finally` blocks to check whether a given resource is `null` or not. We did not do that in our manual implementation, because there is no chance that the resource is `null`. The compiler systematically generates such code, however.

Let us now consider another example, this time involving three resources:

```
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);
           }
       }
   }
```

Copy

이 메서드는 파일을 압축하기 위해 세 가지 리소스를 조작합니다. 하나는 읽기 용 스트림, 하나는 압축 용 스트림, 다른 하나는 출력 파일에 대한 스트림입니다. 이 코드는 리소스 관리 관점에서 정확합니다. Java SE 7 이전에는이 메서드를 포함하는 클래스를 디 컴파일하여 얻은 코드와 유사한 코드를 작성해야했습니다 `JD-GUI`.

```
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(localFile
            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();
            }
        }
    }
}
```

부

`try-with-resources`Java SE 7 에있는 명령문 의 이점은 이러한 예에 대해 설명이 필요하지 않습니다. 작성할 코드가 적고 코드를 읽기가 더 쉬우 며 마지막으로 코드가 리소스를 유출하지 않습니다!

토론

인터페이스 의 `close()` 메서드 정의에는 `java.lang.AutoCloseable`a `java.lang.Exception`가 발생할 수 있음이 언급되어 있습니다. 그러나 이전 `AutoClose`예제에서는 확인 된 예외를 언급하지 않고이 메서드를 선언했습니다. 일부는 예외 마스킹을 설명하기 위해 의도적으로 수행했습니다.

자동 종료 가능한 클래스 `java.lang.Exception`에 대한 사양은 특정 확인 된 예외를 위해 throw 를 피하고 `close()` 메서드가 실패 할 것으로 예상되지 않는 경우 확인 된 예외를 언급하지 않도록 제안합니다 . 또한 억제해서는 안되는 예외를 선언하지 않는 것이 `java.lang.InterruptedException`가장 좋은 예입니다. 실제로이를 억제하고 다른 예외에 연결하면 스레드 인터럽트 이벤트가 무시되고 응용 프로그램이 일관성없는 상태가 될 수 있습니다.

try-with-resources성명 사용에 관한 합법적 인 질문은 적절한 리소스 관리 코드를 수동으로 작성하는 것과 비교하여 성능에 미치는 영향입니다. 이전 예제에서 디 컴파일을 통해 설명한 것처럼 컴파일러는 모든 예외를 적절하게 처리하기 위해 최소한의 올바른 코드를 유추하기 때문에 실제로 성능에 영향을 미치지 않습니다.

하루가 끝날 때 try-with-resources문은 for반복자를 통해 루프를 확장하기 위해 Java SE 5에 도입 된 향상된 루프와 같은 구문 설탕 입니다.

즉, try-with-resources문장 확장 의 복잡성을 제한 할 수 있습니다 . 일반적으로 try블록이 리소스를 선언할수록 생성되는 코드는 더 복잡해집니다. 이전 compress() 방법은 세 개가 아닌 두 개의 리소스로 다시 작성하여 예외 처리 블록을 덜 생성 할 수 있습니다.

```
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);
            }
        }
    }
```

부

try-with-resourcesJava 에서 성명 이 등장하기 전의 경우와 마찬가지로 일반적으로 개발자는 리소스 인스턴스화를 연결할 때 항상 장단점을 이해해야합니다. 이를 위해 가장 좋은 방법은 각 리소스의 close() 방법 사양을 읽고 의미와 의미를 이해하는 것입니다.

기사의 시작 부분에있는 writingWithARM () 예제로 돌아 가면에서 DataOutputStream예외를 throw 할 가능성이 없기 때문에 연결이 안전 합니다 close(). 그러나 마지막 예제에서는 GZIPOutputStream나머지 압축 데이터를 close() 메서드의 일부로 쓰려고 하기 때문에 그렇지 않습니다 . 압축 된 파일을 작성하는 동안 예외가 더 일찍 발생하면 close() in 메서드 GZIPOutputStream가 추가 예외를 throw 할 가능성이 높으므로 close() 메서드 FileOutputStream가 호출되지 않고 파일 설명자 리소스가 누출됩니다.

좋은 방법은 try-with-resources파일 디스크립터, 소켓 또는 JDBC 연결과 같은 중요한 시스템 자원을 보유하는 각 자원에 대해 명령문에 별도의 선언을 갖는 것입니다 . 여기서 close() 메소드가 최종적으로 호출 되는지 확인해야합니다 . 그렇지 않으면 관련 리소스 API가이를 허용하는 경우 할당을 연결하는 것은 단순한 편의가 아닙니다. 또한 리소스 누출을 방지하면서 더 간결한 코드를 생성합니다.

결론

이 기사에서는 안전한 자원 관리를 위해 Java SE 7의 새로운 언어 구조를 소개했습니다. 이 확장은 문법적인 설탕보다 더 많은 영향을 미칩니다. 실제로 개발자를 대신하여 올바른 코드를 생성하므로 틀리기 쉬운 상용구 코드를 작성할 필요가 없습니다. 더 중요한 것은 이러한 변화가 하나의 예외를 다른 예외에 연결하는 진화와 함께 진행되어 서로를 은폐하는 잘 알려진 예외 문제에 대한 우아한 솔루션을 제공한다는 것입니다.

또한보십시오

다음은 몇 가지 추가 리소스입니다.

- 자바 SE 7 미리보기 : http://jdk7.java.net/preview/
- Joshua Bloch의 자동 리소스 관리에 대한 원본 제안, 2009 년 2 월 27 일 : http://mail.openjdk.java.net/pipermail/coin-dev/2009-February/000011.html 및 //docs.google.com/View ? id =

ddv8ts74_3fs7483dp

- 프로젝트 코인 : http://openjdk.java.net/projects/coin/
- JSR334 초기 초안 미리보기 : http://jcp.org/aboutJava/communityprocess/edr/jsr334/index.html
- JSR334 공개 검토 : http://jcp.org/aboutJava/communityprocess/pr/jsr334/index.html
- *Java Puzzlers :* Joshua Bloch 및 Neal Gafter의 함정, 함정 및 코너 케이스 (Addison-Wesley Professional, 2005)
- Joshua Bloch의 효과적인 *Java* 프로그래밍 언어 가이드 (Addison-Wesley Professional, 2001)
- 데코레이터 디자인 패턴 : http://en.wikipedia.org/wiki/Decorator_pattern
- 정적 코드 분석 도구 인 FindBugs : http://findbugs.sourceforge.net/
- JD-GUI, 자바 바이트 코드 디 컴파일러 : http://java.decompiler.free.fr/?q=jdgui
- Python : http://www.python.org/
- 루비 : http://www.ruby-lang.org/

저자 정보

Julien Ponge 는 오랫동안 오픈 소스 장인입니다. 그는 IzPack 설치 프로그램 프레임 워크를 만들었 으며 Sun Microsystems와 협력하여 GlassFish 애플리케이션 서버를 포함한 여러 다른 프로젝트에 참여했습니다. 박사 학위 보유 UNSW Sydney와 UBP Clermont-Ferrand에서 컴퓨터 과학을 전공 한 그는 현재 INSA de Lyon의 컴퓨터 과학 및 공학 부교수이자 INRIA Amazones 팀의 연구원입니다. 산업 언어와 학술 언어를 모두 구사하는 그는 이러한 세계 간의 시너지 효과를 더욱 발전시키는 데 큰 동기를 부여했습니다.