

12 | 异常处理：别让自己在出问题的时候变为瞎子

2020-04-04 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 18:25 大小 12.66M



你好，我是朱晔。今天，我来和你聊聊异常处理容易踩的坑。

应用程序避免不了出异常，捕获和处理异常是考验编程功力的一个精细活。一些业务项目中，我曾看到开发同学在开发业务逻辑时不考虑任何异常处理，项目接近完成时再采用“流水线”的方式进行异常处理，也就是统一为所有方法打上 `try...catch...` 捕获所有异常记录日志，有些技巧的同学可能会使用 AOP 来进行类似的“统一异常处理”。

其实，这种处理异常的方式非常不可取。那么今天，我就和你分享下不可取的原因、 处理相关的坑和最佳实践。

捕获和处理异常容易犯的错

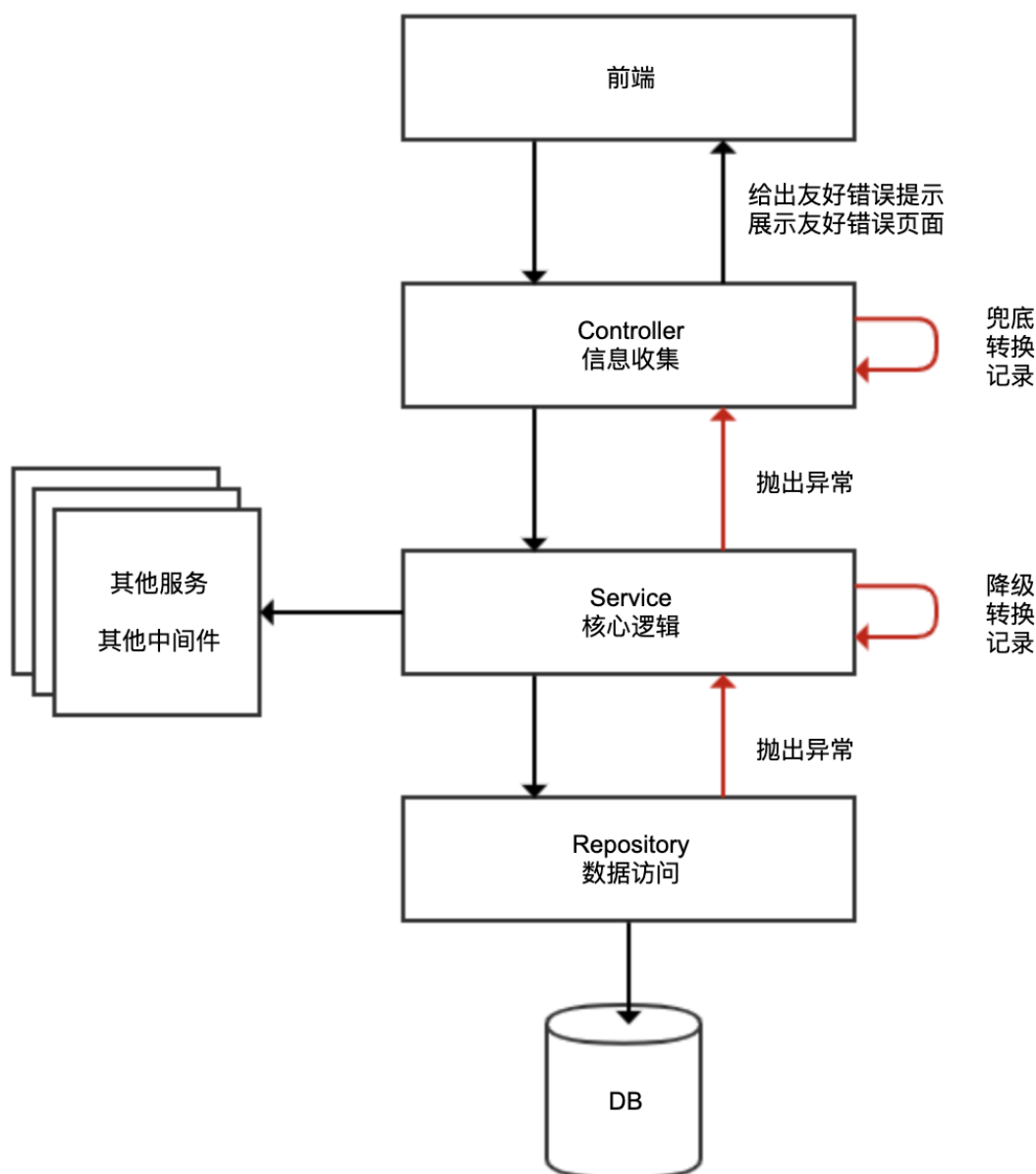
“统一异常处理”方式正是我要说的第一个错：**不在业务代码层面考虑异常处理，仅在框架层面粗犷捕获和处理异常。**

为了理解错在何处，我们先来看看大多数业务应用都采用的三层架构：

Controller 层负责信息收集、参数校验、转换服务层处理的数据适配前端，轻业务逻辑；

Service 层负责核心业务逻辑，包括各种外部服务调用、访问数据库、缓存处理、消息处理等；

Repository 层负责数据访问实现，一般没有业务逻辑。



每层架构的工作性质不同，且从业务性质上异常可能分为业务异常和系统异常两大类，这就决定了很难进行统一的异常处理。我们从底向上看一下三层架构：

Repository 层出现异常或许可以忽略，或许可以降级，或许需要转化为一个友好的异常。如果一律捕获异常仅记录日志，很可能业务逻辑已经出错，而用户和程序本身完全感知不到。

Service 层往往涉及数据库事务，出现异常同样不适合捕获，否则事务无法自动回滚。此外 Service 层涉及业务逻辑，有些业务逻辑执行中遇到业务异常，可能需要在异常后转入分支业务流程。如果业务异常都被框架捕获了，业务功能就会不正常。


如果下层异常上升到 Controller 层还是无法处理的话，Controller 层往往会给予用户友好提示，或是根据每一个 API 的异常表返回指定的异常类型，同样无法对所有异常一视同仁。

因此，我不建议在框架层面进行异常的自动、统一处理，尤其不要随意捕获异常。但，框架可以做兜底工作。如果异常上升到最上层逻辑还是无法处理的话，可以以统一的方式进行异常转换，比如通过 `@RestControllerAdvice + @ExceptionHandler`，来捕获这些“未处理”异常：

对于自定义的业务异常，以 Warn 级别的日志记录异常以及当前 URL、执行方法等信息后，提取异常中的错误码和消息等信息，转换为合适的 API 包装体返回给 API 调用方；

对于无法处理的系统异常，以 Error 级别的日志记录异常和上下文信息（比如 URL、参数、用户 ID）后，转换为普适的“服务器忙，请稍后再试”异常信息，同样以 API 包装体返回给调用方。

比如，下面这段代码的做法：

 复制代码

```
1 @RestControllerAdvice
2 @Slf4j
3 public class RestControllerExceptionHandler {
4     private static int GENERIC_SERVER_ERROR_CODE = 2000;
5     private static String GENERIC_SERVER_ERROR_MESSAGE = "服务器忙，请稍后再试";
6
7     @ExceptionHandler
8     public APIResponse handle(HttpServletRequest req, HandlerMethod method, Exception ex) {
9         if (ex instanceof BusinessException) {
10             BusinessException exception = (BusinessException) ex;
```

```

11         log.warn(String.format("访问 %s -> %s 出现业务异常!", req.getRequestI
12         return new APIResponse(false, null, exception.getCode(), exception
13     } else {
14         log.error(String.format("访问 %s -> %s 出现系统异常!", req.getReques
15         return new APIResponse(false, null, GENERIC_SERVER_ERROR_CODE, GENI
16     }
17 }
18 }

```

出现运行时系统异常后，异常处理程序会直接把异常转换为 JSON 返回给调用方：

```

▼ {
    "success": false,
    "data": null,
    "code": 2000,
    "message": "服务器忙，请稍后再试"
}

```

要做得更好，你可以把相关出入参、用户信息在脱敏后记录到日志中，方便出现问题时根据上下文进一步排查。

第二个错，**捕获了异常后直接生吞**。在任何时候，我们捕获了异常都不应该生吞，也就是直接丢弃异常不记录、不抛出。这样的处理方式还不如不捕获异常，因为被生吞掉的异常一旦导致 Bug，就很难在程序中找到蛛丝马迹，使得 Bug 排查工作难上加难。

通常情况下，生吞异常的原因，可能是不希望自己的方法抛出受检异常，只是为了把异常“处理掉”而捕获并生吞异常，也可能是想当然地认为异常并不重要或不可能产生。但不管是什么原因，不管是你认为多么不重要的异常，都不应该生吞，哪怕是一个日志也好。

第三个错，**丢弃异常的原始信息**。我们来看两个不太合适的异常处理方式，虽然没有完全生吞异常，但也丢失了宝贵的异常信息。

比如有这么一个会抛出受检异常的方法 `readFile`：

[复制代码](#)

```
1 private void readFile() throws IOException {
2     Files.readAllLines(Paths.get("a_file"));
3 }
```

像这样调用 `readFile` 方法，捕获异常后，完全不记录原始异常，直接抛出一个转换后异常，导致出了问题不知道 `IOException` 具体是哪里引起的：

[复制代码](#)

```
1 @GetMapping("wrong1")
2 public void wrong1(){
3     try {
4         readFile();
5     } catch (IOException e) {
6         //原始异常信息丢失
7         throw new RuntimeException("系统忙请稍后再试");
8     }
9 }
```

或者是这样，只记录了异常消息，却丢失了异常的类型、栈等重要信息：

[复制代码](#)

```
1 catch (IOException e) {
2     //只保留了异常消息，栈没有记录
3     log.error("文件读取错误，{}", e.getMessage());
4     throw new RuntimeException("系统忙请稍后再试");
5 }
```

留下的日志是这样的，看完一脸茫然，只知道文件读取错误的文件名，至于为什么读取错误、是不存在还是没权限，完全不知道。

[复制代码](#)

```
1 [12:57:19.746] [http-nio-45678-exec-1] [ERROR] [.g.t.c.e.d.HandleExceptionCont
```

这两种处理方式都不太合理，可以改为如下方式：

[复制代码](#)

```
1 catch (IOException e) {  
2     log.error("文件读取错误", e);  
3     throw new RuntimeException("系统忙请稍后再试");  
4 }
```

或者，把原始异常作为转换后新异常的 cause，原始异常信息同样不会丢：

[复制代码](#)

```
1 catch (IOException e) {  
2     throw new RuntimeException("系统忙请稍后再试", e);  
3 }
```

其实，JDK 内部也会犯类似的错。之前我遇到一个使用 JDK10 的应用偶发启动失败的案例，日志中可以看到出现类似的错误信息：

[复制代码](#)

```
1 Caused by: java.lang.SecurityException: Couldn't parse jurisdiction policy file  
2   at java.base/javax.crypto.JceSecurity.setupJurisdictionPolicies(JceSecurity.  
3   at java.base/javax.crypto.JceSecurity.access$000(JceSecurity.java:73)  
4   at java.base/javax.crypto.JceSecurity$1.run(JceSecurity.java:109)  
5   at java.base/javax.crypto.JceSecurity$1.run(JceSecurity.java:106)  
6   at java.base/java.security.AccessController.doPrivileged(Native Method)  
7   at java.base/javax.crypto.JceSecurity.<clinit>(JceSecurity.java:105)  
8   ... 20 more
```

查看 JDK JceSecurity 类 setupJurisdictionPolicies 方法源码，发现异常 e 没有记录，也没有作为新抛出异常的 cause，当时读取文件具体出现什么异常（权限问题又或是 IO 问题）可能永远都无法知道了，对问题定位造成了很大困扰：


```

329     try (DirectoryStream<Path> stream = Files.newDirectoryStream(
330         cryptoPolicyPath, "{default,exempt}*.policy")) {
331         for (Path entry : stream) {
332             try (InputStream is = new BufferedInputStream(
333                 Files.newInputStream(entry))) {
334                 String filename = entry.getFileName().toString();
335
336                 CryptoPermissions tmpPerms = new CryptoPermissions();
337                 tmpPerms.load(is);
338
339                 if (filename.startsWith("default_")) {
340                     // Did we find a default perms?
341                     defaultPolicy = ((defaultPolicy == null) ? tmpPerms :
342                         defaultPolicy.getMinimum(tmpPerms));
343                 } else if (filename.startsWith("exempt_")) {
344                     // Did we find a exempt perms?
345                     exemptPolicy = ((exemptPolicy == null) ? tmpPerms :
346                         exemptPolicy.getMinimum(tmpPerms));
347                 } else {
348                     // This should never happen. newDirectoryStream
349                     // should only throw return "{default,exempt}*.policy"
350                     throw new SecurityException(
351                         "Unexpected jurisdiction policy files in : " +
352                         cryptoPolicyProperty);
353                 }
354             } catch (Exception e) {
355                 throw new SecurityException(
356                     "Couldn't parse jurisdiction policy files in: " +
357                     cryptoPolicyProperty);
358             }
359         }

```

第四个错，**抛出异常时不指定任何消息**。我见过一些代码中的偷懒做法，直接抛出没有 message 的异常：

```
1 throw new RuntimeException();
```

复制代码

这么写的同学可能觉得永远不会走到这个逻辑，永远不会出现这样的异常。但，这样的异常却出现了，被 ExceptionHandler 拦截到后输出了下面的日志信息：

```

1 [13:25:18.031] [http-nio-45678-exec-3] [ERROR] [c.e.d.RestControllerExceptionHandler:
2 java.lang.RuntimeException: null
3 ...

```

复制代码

这里的 null 非常容易引起误解。按照空指针问题排查半天才发现，其实是异常的 message 为空。

总之，如果你捕获了异常打算处理的话，**除了通过日志正确记录异常原始信息外，通常还有三种处理模式：**

转换，即转换新的异常抛出。对于新抛出的异常，最好具有特定的分类和明确的异常消息，而不是随便抛一个无关或没有任何信息的异常，并最好通过 `cause` 关联老异常。

重试，即重试之前的操作。比如远程调用服务端过载超时的情况，盲目重试会让问题更严重，需要考虑当前情况是否适合重试。

恢复，即尝试进行降级处理，或使用默认值来替代原始数据。

以上，就是通过 `catch` 捕获处理异常的一些最佳实践。

小心 `finally` 中的异常


有些时候，我们希望不管是否遇到异常，逻辑完成后都要释放资源，这时可以使用 `finally` 代码块而跳过使用 `catch` 代码块。

但要千万小心 `finally` 代码块中的异常，因为资源释放处理等收尾操作同样也可能出现异常。比如下面这段代码，我们在 `finally` 中抛出一个异常：

 复制代码


```
1 @GetMapping("wrong")
2 public void wrong() {
3     try {
4         log.info("try");
5         //异常丢失
6         throw new RuntimeException("try");
7     } finally {
8         log.info("finally");
9         throw new RuntimeException("finally");
10    }
11 }
```

最后在日志中只能看到 `finally` 中的异常，**虽然 `try` 中的逻辑出现了异常，但却被 `finally` 中的异常覆盖了**。这是非常危险的，特别是 `finally` 中出现的异常是偶发的，就会在部分时候覆盖 `try` 中的异常，让问题更不明显：

 复制代码

```
1 [13:34:42.247] [http-nio-45678-exec-1] [ERROR] [.a.c.c.C.[.[./].[dispatcherSe
2 java.lang.RuntimeException: finally
```


至于异常为什么被覆盖，原因也很简单，因为一个方法无法出现两个异常。修复方式是，`finally` 代码块自己负责异常捕获和处理：

 复制代码


```
1 @GetMapping("right")
2 public void right() {
3     try {
4         log.info("try");
5         throw new RuntimeException("try");
6     } finally {
7         log.info("finally");
8         try {
9             throw new RuntimeException("finally");
10        } catch (Exception ex) {
11            log.error("finally", ex);
12        }
13    }
14 }
```

或者可以把 `try` 中的异常作为主异常抛出，使用 `addSuppressed` 方法把 `finally` 中的异常附加到主异常上：

 复制代码


```
1 @GetMapping("right2")
2 public void right2() throws Exception {
3     Exception e = null;
4     try {
5         log.info("try");
6         throw new RuntimeException("try");
7     } catch (Exception ex) {
8         e = ex;
9     } finally {
10        log.info("finally");
11        try {
12            throw new RuntimeException("finally");
13        } catch (Exception ex) {
14            if (e != null) {
15                e.addSuppressed(ex);
16            } else {
17                e = ex;
18            }
19        }
20    }
21    throw e;
22 }
```

运行方法可以得到如下异常信息，其中同时包含了主异常和被屏蔽的异常：

 复制代码


```
1 java.lang.RuntimeException: try
2   at org.geekbang.time.commonmistakes.exception.finallyissue.FinallyIssueContr
3   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
4   ...
5   Suppressed: java.lang.RuntimeException: finally
6     at org.geekbang.time.commonmistakes.exception.finallyissue.FinallyIssueCon
7     ... 54 common frames omitted
```

其实这正是 try-with-resources 语句的做法，对于实现了 AutoCloseable 接口的资源，建议使用 try-with-resources 来释放资源，否则也可能会产生刚才提到的，释放资源时出现的异常覆盖主异常的问题。比如如下我们定义一个测试资源，其 read 和 close 方法都会抛出异常：

 复制代码


```
1 public class TestResource implements AutoCloseable {
2     public void read() throws Exception{
3         throw new Exception("read error");
4     }
5     @Override
6     public void close() throws Exception {
7         throw new Exception("close error");
8     }
9 }
```

使用传统的 try-finally 语句，在 try 中调用 read 方法，在 finally 中调用 close 方法：

 复制代码


```
1 @GetMapping("userresourcewrong")
2 public void userresourcewrong() throws Exception {
3     TestResource testResource = new TestResource();
4     try {
5         testResource.read();
6     } finally {
7         testResource.close();
8     }
9 }
```

可以看到，同样出现了 finally 中的异常覆盖了 try 中异常的问题：

 复制代码


```
1 java.lang.Exception: close error
2   at org.geekbang.time.commonmistakes.exception.finallyissue.TestResource.close
3   at org.geekbang.time.commonmistakes.exception.finallyissue.FinallyIssueContr
```

而改为 try-with-resources 模式之后：

 复制代码

```
1 @GetMapping("userresourceright")
2 public void userresourceright() throws Exception {
3     try (TestResource testResource = new TestResource()){
4         testResource.read();
5     }
6 }
```

try 和 finally 中的异常信息都可以得到保留：

 复制代码

```
1 java.lang.Exception: read error
2   at org.geekbang.time.commonmistakes.exception.finallyissue.TestResource.read
3   ...
4   Suppressed: java.lang.Exception: close error
5       at org.geekbang.time.commonmistakes.exception.finallyissue.TestResource.cl
6       at org.geekbang.time.commonmistakes.exception.finallyissue.FinallyIssueCon
7       ... 54 common frames omitted
```


千万别把异常定义为静态变量

既然我们通常会自定义一个业务异常类型，来包含更多的异常信息，比如异常错误码、友好的错误提示等，那就需要在业务逻辑各处，手动抛出各种业务异常来返回指定的错误码描述（比如对于下单操作，用户不存在返回 2001，商品缺货返回 2002 等）。

对于这些异常的错误代码和消息，我们期望能够统一管理，而不是散落在程序各处定义。这个想法很好，但稍有不慎就可能会出现把异常定义为静态变量的坑。

我在救火排查某项目生产问题时，遇到了一件非常诡异的事情：我发现异常堆信息显示的方法调用路径，在当前入参的情况下根本不可能产生，项目的业务逻辑又很复杂，就始终没往异常信息是错的这方面想，总觉得是因为某个分支流程导致业务没有按照期望的流程进行。

经过艰难的排查，最终定位到原因是把异常定义为了静态变量，导致异常栈信息错乱，类似于定义一个 Exceptions 类来汇总所有的异常，把异常存放在静态字段中：

 复制代码

```
1 public class Exceptions {
2     public static BusinessException ORDEREXISTS = new BusinessException("订单已!
3     ...
4 }
```

把异常定义为静态变量会导致异常信息固化，这就和异常的栈一定是需要根据当前调用来动态获取相矛盾。


我们写段代码来模拟下这个问题：定义两个方法 createOrderWrong 和 cancelOrderWrong 方法，它们内部都会通过 Exceptions 类来获得一个订单不存在的异常；先后调用两个方法，然后抛出。

 复制代码

```
1 @GetMapping("wrong")
2 public void wrong() {
3     try {
4         createOrderWrong();
5     } catch (Exception ex) {
6         log.error("createOrder got error", ex);
7     }
8     try {
9         cancelOrderWrong();
10    } catch (Exception ex) {
11        log.error("cancelOrder got error", ex);
12    }
13 }
14
15 private void createOrderWrong() {
16     //这里有问题
17     throw Exceptions.ORDEREXISTS;
18 }
19
20 private void cancelOrderWrong() {
21     //这里有问题
```


```
22     throw Exceptions.ORDEREXISTS;
23 }
```

运行程序后看到如下日志，cancelOrder got error 的提示对应了 createOrderWrong 方法。显然，cancelOrderWrong 方法在出错后抛出的异常，其实是 createOrderWrong 方法出错的异常：

 复制代码

```
1 [14:05:25.782] [http-nio-45678-exec-1] [ERROR] [.c.e.d.PredefinedExceptionCont
2 org.geekbang.time.commonmistakes.exception.demo2.BusinessException: 订单已经存在
3   at org.geekbang.time.commonmistakes.exception.demo2.Exceptions.<clinit>(Exce
4   at org.geekbang.time.commonmistakes.exception.demo2.PredefinedExceptionContr
5   at org.geekbang.time.commonmistakes.exception.demo2.PredefinedExceptionContr
```

修复方式很简单，改一下 Exceptions 类的实现，通过不同的方法把每一种异常都 new 出来抛出即可：


 复制代码

```
1 public class Exceptions {
2     public static BusinessException orderExists(){
3         return new BusinessException("订单已经存在", 3001);
4     }
5 }
```

提交线程池的任务出了异常会怎么样？

在 [第 3 讲](#) 介绍线程池时我提到，线程池常用作异步处理或并行处理。那么，把任务提交到线程池处理，任务本身出现异常时会怎样呢？

我们来看一个例子：提交 10 个任务到线程池异步处理，第 5 个任务抛出一个 RuntimeException，每个任务完成后都会输出一行日志：

 复制代码


```
1 @GetMapping("execute")
2 public void execute() throws InterruptedException {
3
4     String prefix = "test";
5     ExecutorService threadPool = Executors.newFixedThreadPool(1, new ThreadFac
```

```

6      //提交10个任务到线程池处理，第5个任务会抛出运行时异常
7      IntStream.rangeClosed(1, 10).forEach(i -> threadPool.execute(() -> {
8          if (i == 5) throw new RuntimeException("error");
9          log.info("I'm done : {}", i);
10     }));
11
12     threadPool.shutdown();
13     threadPool.awaitTermination(1, TimeUnit.HOURS);
14 }

```

观察日志可以发现两点：

 复制代码


```

1  ...
2  [14:33:55.990] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:26 ] - I
3  Exception in thread "test0" java.lang.RuntimeException: error
4      at org.geekbang.time.commonmistakes.exception.demo3.ThreadPoolAndExceptionHandlerCo
5      at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:
6      at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:
7      at java.lang.Thread.run(Thread.java:748)
8  [14:33:55.990] [test1] [INFO ] [e.d.ThreadPoolAndExceptionHandler:26 ] - I
9  ...

```

任务 1 到 4 所在的线程是 test0，任务 6 开始运行在线程 test1。由于我的线程池通过线程工厂为线程使用统一的前缀 test 加上计数器进行命名，因此**从线程名的改变可以知道因为异常的抛出老线程退出了，线程池只能重新创建一个线程**。如果每个异步任务都以异常结束，那么线程池可能完全起不到线程重用的作用。

因为没有手动捕获异常进行处理，ThreadGroup 帮我们进行了未捕获异常的默认处理，向标准错误输出打印了出现异常的线程名称和异常信息。**显然，这种没有以统一的错误日志格式记录错误信息打印出来的形式，对生产级代码是不合适的**，ThreadGroup 的相关源码如下所示：

 复制代码

```

1  public void uncaughtException(Thread t, Throwable e) {
2      if (parent != null) {
3          parent.uncaughtException(t, e);
4      } else {
5          Thread.UncaughtExceptionHandler ueh =
6              Thread.getDefaultUncaughtExceptionHandler();
7          if (ueh != null) {
8              ueh.uncaughtException(t, e);

```


```

9         } else if (!(e instanceof ThreadDeath)) {
10             System.err.print("Exception in thread \""
11                               + t.getName() + "\" ");
12             e.printStackTrace(System.err);
13         }
14     }
15 }

```

修复方式有 2 步：

1. 以 execute 方法提交到线程池的异步任务，最好在任务内部做好异常处理；
2. 设置自定义的异常处理程序作为保底，比如在声明线程池时自定义线程池的未捕获异常处理程序：


 复制代码

```

1 new ThreadFactoryBuilder()
2     .setNameFormat(prefix+"%d")
3     .setUncaughtExceptionHandler((thread, throwable)-> log.error("ThreadPool {} ;
4     .get()

```

或者设置全局的默认未捕获异常处理程序：

 复制代码

```

1 static {
2     Thread.setDefaultUncaughtExceptionHandler((thread, throwable)-> log.error(
3 }

```

通过线程池 `ExecutorService` 的 `execute` 方法提交任务到线程池处理，如果出现异常会导致线程退出，控制台输出中可以看到异常信息。那么，把 `execute` 方法改为 `submit`，线程还会退出吗，异常还能被处理程序捕获到吗？

修改代码后重新执行程序可以看到如下日志，说明线程没退出，异常也没记录被生吞了：

 复制代码

```

1 [15:44:33.769] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
2 [15:44:33.770] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
3 [15:44:33.770] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I


```



```
4 [15:44:33.770] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
5 [15:44:33.770] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
6 [15:44:33.770] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
7 [15:44:33.770] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
8 [15:44:33.771] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
9 [15:44:33.771] [test0] [INFO ] [e.d.ThreadPoolAndExceptionHandler:47 ] - I
```

为什么会这样呢？

查看 `FutureTask` 源码可以发现，在执行任务出现异常之后，异常存到了一个 `outcome` 字段中，只有在调用 `get` 方法获取 `FutureTask` 结果的时候，才会以 `ExecutionException` 的形式重新抛出异常：

 复制代码

```
1 public void run() {
2     ...
3     try {
4         Callable<V> c = callable;
5         if (c != null && state == NEW) {
6             V result;
7             boolean ran;
8             try {
9                 result = c.call();
10                ran = true;
11            } catch (Throwable ex) {
12                result = null;
13                ran = false;
14                setException(ex);
15            }
16        }
17    }
18
19    protected void setException(Throwable t) {
20        if (UNSAFE.compareAndSwapInt(this, stateOffset, NEW, COMPLETING)) {
21            outcome = t;
22            UNSAFE.putOrderedInt(this, stateOffset, EXCEPTIONAL); // final state
23            finishCompletion();
24        }
25    }
26
27    public V get() throws InterruptedException, ExecutionException {
28        int s = state;
29        if (s <= COMPLETING)
30            s = awaitDone(false, 0L);
31        return report(s);
32    }
```

```
33 private V report(int s) throws ExecutionException {
34     Object x = outcome;
35     if (s == NORMAL)
36         return (V)x;
37     if (s >= CANCELLED)
38         throw new CancellationException();
39     throw new ExecutionException((Throwable)x);
40 }
41
```

修改后的代码如下所示，我们把 submit 返回的 Future 放到了 List 中，随后遍历 List 来捕获所有任务的异常。这么做确实合乎情理。既然是以 submit 方式来提交任务，那么我们应该关心任务的执行结果，否则应该以 execute 来提交任务：

 复制代码

```
1 List<Future> tasks = IntStream.rangeClosed(1, 10).mapToObj(i -> threadPool.subm
2     if (i == 5) throw new RuntimeException("error");
3     log.info("I'm done : {}", i);
4 })).collect(Collectors.toList());
5
6 tasks.forEach(task-> {
7     try {
8         task.get();
9     } catch (Exception e) {
10         log.error("Got exception", e);
11     }
12 });
```

执行这段程序可以看到如下的日志输出：

 复制代码

```
1 [15:44:13.543] [http-nio-45678-exec-1] [ERROR] [e.d.ThreadPoolAndExceptionCont
2 java.util.concurrent.ExecutionException: java.lang.RuntimeException: error
```

重点回顾

在今天的文章中，我介绍了处理异常容易犯的几个错和最佳实践。

第一，注意捕获和处理异常的最佳实践。首先，不应该用 AOP 对所有方法进行统一异常处理，异常要么不捕获不处理，要么根据不同的业务逻辑、不同的异常类型进行精细化、针对

性处理；其次，处理异常应该杜绝生吞，并确保异常栈信息得到保留；最后，如果需要重新抛出异常的话，请使用具有意义的异常类型和异常消息。

第二，务必小心 finally 代码块中资源回收逻辑，确保 finally 代码块不出现异常，内部把异常处理完毕，避免 finally 中的异常覆盖 try 中的异常；或者考虑使用 addSuppressed 方法把 finally 中的异常附加到 try 中的异常上，确保主异常信息不丢失。此外，使用实现了 AutoCloseable 接口的资源，务必使用 try-with-resources 模式来使用资源，确保资源可以正确释放，也同时确保异常可以正确处理。

第三，虽然在统一的地方定义收口所有的业务异常是一个不错的实践，但务必确保异常是每次 new 出来的，而不能使用一个预先定义的 static 字段存放异常，否则可能会引起栈信息的错乱。

第四，确保正确处理了线程池中任务的异常，如果任务通过 execute 提交，那么出现异常会导致线程退出，大量的异常会导致线程重复创建引起性能问题，我们应该尽可能确保任务不出异常，同时设置默认的未捕获异常处理程序来兜底；如果任务通过 submit 提交意味着我们关心任务的执行结果，应该通过拿到的 Future 调用其 get 方法来获得任务运行结果和可能出现的异常，否则异常可能就被生吞了。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

思考与讨论

1. 关于在 finally 代码块中抛出异常的坑，如果在 finally 代码块中返回值，你觉得程序会以 try 或 catch 中返回值为准，还是以 finally 中的返回值为准呢？
2. 对于手动抛出的异常，不建议直接使用 Exception 或 RuntimeException，通常建议复用 JDK 中的一些标准异常，比如 [IllegalArgumentExcpetion](#)、[IllegalStateException](#)、[UnsupportedOperationException](#)，你能说说它们的适用场景，并列更多常用异常吗？

不知道针对异常处理，你还遇到过什么坑，还有什么最佳实践的心得吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 空值处理：分不清楚的null和恼人的空指针

下一篇 13 | 日志：日志记录真没你想象的那么简单

精选留言 (9)

 写留言



Darren

2020-04-05


这篇文章收获很大，因为我们现在的系统就是用的统一异常处理，使用的就是老师提到的兜底异常，就是简单的分为业务异常和非业务异常，提示语不同而已。

试着回答下问题：

第一个问题：

肯定是以finally语句块为准。...

展开 

作者回复: ，你总结了比较重要的两点，JVM采用异常表控制try-catch的跳转逻辑；对于finally中的代码块其实是复制到try和catch中的return和throw之前的方式来处理的。

 4

 10



行者

2020-04-04

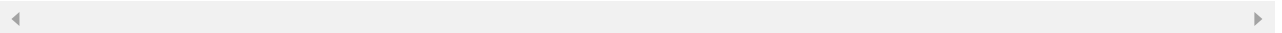
老师，关于 千万别把异常定义为静态变量，麻烦分析下为什么cancelOrderRight抛出的异常信息指向createOrderWrong所在的行~

展开 ▾

作者回复: 创建异常的时候一次性fillInStackTrace了，除非这样：

```
BusinessException ex = Exceptions.ORDEREXISTS;  
ex.fillInStackTrace();  
throw ex;
```

(这样同样不是线程安全的)



💬 1

👍 4



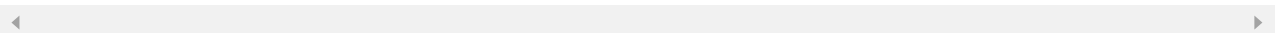
行者

2020-04-04

IllegalArgumentException: 入参错误，比如参数类型int输入string。
IllegalStateException: 状态错误，比如订单已经支付完成，二次请求支付接口。
UnsupportedOperationException: 不支持操作错误，比如对一笔不能退款的订单退款。
其他异常
SecurityException: 权限错误，比如未登陆用户调用修改用户信息接口。

展开 ▾

作者回复: 不错



💬

👍 3



梦倚栏杆

2020-04-04

现在出问题就是瞎子，一点一点的日志的打，上线，哎，这哪是个合格的RD呀

💬

👍 3



努力奋斗的Pisces

2020-04-04

1.得看finally里面是怎么处理的了，除非finally没有执行到，或者是finally里面报错了，不然都是按照finally里面的返回值做最终的返回吧

展开 ▾



3



梦倚栏杆

2020-04-06

遇到一个坑(也可以说不理解), 和该篇文章没关系, 反馈一下

mysql 占位符问题

```
prepare sqltpl from 'select id,name from table1 where id in (?);  
set @a='1,2,3,4,5,6,7,8,9';...
```

展开 ▾

作者回复: 嗯, 原因是:

- 1、id是数字型, 传入的1,2,3,4,5,6,7,8,9的并不能转换为数字, 所以截断为1 (第一个逗号之前的数字)
- 2、zhangsan,lisi,wangwu整个当做一个字符串来查询了, 所以数据库中根本查不到name='zhangsan,lisi,wangwu'这样的记录



1

2



小杰

2020-04-05

老师, 看不懂那个异常定义为静态变量的例子, 异常打印的是堆栈信息, 堆栈信息必须是对象是这个意思吗?

作者回复: 仔细观察一下栈, 数据串了, 取消订单的异常栈显示了下单的方法



2



终结者999号

2020-04-05

关于行者的问题, 请老师再解释一下可以吗? 貌似没有看懂那个回复, 什么叫一次性fillstacktrace?

作者回复: 就是Throwable的stacktrace只是在其new出来的时候才初始化 (调用fillInStackTrace方法) 是一次性的 (除非你手动调用那个方法), 而非getStackTrace的时候去获得stacktrace, 毕竟我们关心的是异常抛出时的栈。





2



旭东

2020-04-04

订单存在异常，这个例子实际开发中真需要区分吗？个人觉得无效参数的例子更好些



2