> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

《手册》的异常处理规约对异常的处理方式提出了一些指导规范，如：

> 【强制】异常不要用来做流程控制，条件控制。
>
> 【强制】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回滚事务。

常规 try - catch 捕捉异常非常简单，相信大家都很熟悉，这里就不作展开。

本节重点讲述一些异常处理姿势不正确导致的各种 BUG， 并给出一些异常处理的建议。

在实际开发中关于异常的一个重要问题是：要不要 “吞掉” 异常。

所谓 “吞掉” 异常是指：处理后不再将异常传给上层。其中包括 catch 到异常并处理（打印日志、发通知等）后不再扔给上层；捕捉到异常后给上层返回 null 值等行为。

其中 “**有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回 滚事务。**” 就属于其中一例。

那么为什么要手动回滚呢?

我们看下事务的执行入口：

`TransactionInterceptor#invoke`

调用到了这里 `TransactionAspectSupport#invokeWithinTransaction` ：

```
    @Nullable
    protected Object invokeWithinTransaction(Method method, @Nullable Class<?>
targetClass,
            final InvocationCallback invocation) throws Throwable {


        TransactionAttributeSource tas = getTransactionAttributeSource();
        final TransactionAttribute txAttr = (tas != null ?
tas.getTransactionAttribute(method, targetClass) : null);
        final PlatformTransactionManager tm =
determineTransactionManager(txAttr);
        final String joinpointIdentification = methodIdentification(method,
targetClass, txAttr);

        if (txAttr == null || !(tm instanceof
CallbackPreferringPlatformTransactionManager)) {

            TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr,
joinpointIdentification);
            Object retVal = null;
            try {


                retVal = invocation.proceedWithInvocation();
            }
            catch (Throwable ex) {

                completeTransactionAfterThrowing(txInfo, ex);
                throw ex;
```

```
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
            commitTransactionAfterReturning(txInfo);
            return retVal;
        }



    }
```

带 @Transaction 注解的事务函数中捕获到异常后，执行如下代码：

`TransactionAspectSupport#completeTransactionAfterThrowing`

```
protected void completeTransactionAfterThrowing(@Nullable TransactionInfo txInfo,
Throwable ex) {
   if (txInfo != null && txInfo.getTransactionStatus() != null) {
      if (logger.isTraceEnabled()) {
         logger.trace("Completing transaction for [" +
txInfo.getJoinpointIdentification() +
               "] after exception: " + ex);
      }
      if (txInfo.transactionAttribute != null &&
txInfo.transactionAttribute.rollbackOn(ex)) {
         try {

txInfo.getTransactionManager().rollback(txInfo.getTransactionStatus());
         }
         catch (TransactionSystemException ex2) {
            logger.error("Application exception overridden by rollback
exception", ex);
            ex2.initApplicationException(ex);
            throw ex2;
         }
         catch (RuntimeException | Error ex2) {
            logger.error("Application exception overridden by rollback
exception", ex);
            throw ex2;
         }
      }
      else {


         try {

txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
         }
         catch (TransactionSystemException ex2) {
            logger.error("Application exception overridden by commit exception",
ex);
            ex2.initApplicationException(ex);
            throw ex2;
         }
```

```
          catch (RuntimeException | Error ex2) {
             logger.error("Application exception overridden by commit exception",
ex);
             throw ex2;
          }
       }
    }
}
```

可以看到，如果设置了事务属性且当前异常满足 rollbackOn 指定的异常（默认为 RuntimeException 类型及其的子类以及 Error 及其子类），则会将当前事务回滚，否则提交。

因此如果 catch 异常后没有再次将异常抛出或者不手动回滚，将会导致事务提交。

在封装二方接口很多人也会选择 “吞掉” 异常，示意代码如下：

```
@Component
public class DemoClient {

    @Resource
    private XXServcie xxServcie;

    public XXInfo someMethod(Long id) {
        try {
            return xxServcie.getXXInfo(id);
        } catch (Exception e) {
            log.warn("调用xx服务异常，参数：{}", id, e);
            return null;
        }
    }
}
```


当调用发生异常时打印异常信息后直接返回 null。

此时如果上层调用方直接拿到返回值对象未做判空处理直接使用其属性，很容易报 NPE。

另外由于此处 “吞掉” 了二方接口的异常，有些业务异常中包含的错误原因（如包含 xxx 敏感词汇、标题不能超过 20 个字等）无法传给上层再封装给前端，可能会造成出错后用户懵逼，增加很多用户咨询。

比如用户输入了某个敏感词汇，调用二方接口时 “吞掉” 了敏感词汇的业务异常提示（输入中包含 xx 敏感词），用户通过技术支持咨询，开发人员要查日志才能知道具体的错误原因（如果此处没打印日志，可能连日志都没得查），非常低效。

开发中要根据具体业务场景慎重确定是否要 “吞掉” 异常，一个 “不经意” 的写法可能会造成很多线上咨询甚至线上 BUG。

【参考】特别注意循环的代码异常处理的对程序的影响。

我们先看下面的例子：

```
    public static void main(String[] args) throws InterruptedException {

        List<String> data = new ArrayList<>();
```

```
        data.add("a");
        data.add("ab");
        data.add("abc");
        data.add("abcd");

        for (String str : data) {

            String result = doSomeRemoteInvoke(str);
            System.out.println(result);
        }


    }
```

在写代码时这种场景非常常见，需要注意的是如果不对循环代码进行捕捉，如果循环中出现异常，后续代码则无法执行。

但是如果在 for 循环外部捕捉异常，虽然 for 循环后如果有代码依然可以执行，但是列表中的非最后一个元素作为参数调用 `doSomeRemoteInvoke` 出现异常，后续数据无法继续执行。

```
try {
    for (String str : data) {

        String result = doSomeRemoteInvoke(str);
        System.out.println(result);
    }
} catch (Exception e) {
    log.error("程序出错，参数data:{},错误详情", JSON.toJSONString(data), e);
}
```


因此需要对 for 循环代码内对可能出现的异常进行捕捉：

```
for (String str : data) {
    try {

        String result = doSomeRemoteInvoke(str);
        System.out.println(result);
    } catch (Exception e) {
        log.error("程序出错，参数data:{},错误详情", JSON.toJSONString(data), e);
    }
}
```

我们再看下面一个例子，思考两个问题：

分别调用两个函数 `pirntList1` 和 `printList2` 输出的结果有何不同？

哪个不需要捕捉异常也不会造成中间有一个出错后续处理中断？

代码如下：

```
public class ExceptionDemo {
```

```
    public static void main(String[] args) throws InterruptedException {

        ExecutorService executorService = Executors.newSingleThreadExecutor();
        List<String> data = new ArrayList<>();
        data.add("a");
        data.add("ab");
        data.add("abc");
        data.add("abcd");

        printList1(data, executorService);

    }

    private static void printList1(List<String> data, ExecutorService
executorService) {
        for (String str : data) {
            executorService.execute(() -> {

                if (str.length() == 2) {
                    throw new IllegalArgumentException();
                }
                System.out.println(str);
            });
        }
    }

    private static void printList2(List<String> data, ExecutorService
executorService) {
        executorService.execute(() -> {
            for (String str : data) {

                if (str.length() == 2) {
                    throw new IllegalArgumentException();
                }
                System.out.println(str);
            }
        });
    }
}
```


让我们来分析这两个函数的区别：

在函数 `pirntList1` 中和上面的代码非常相似，for 循环在线程池执行代码外部，每次循环调用线程池去执行判断和打印语句。

此时依次传入 a、ab、abc、abcd 四个字符串；当执行到 ab 时会抛出 `IllegalArgumentException`，此时线程池中的唯一的线程销毁；当执行到 abc 字符串时，再次在线程池中执行，线程池创建新的线程来执行，依然可以正常执行。

而在函数 `pirntList2` 中 for 循环在 线程池 `execute` 参数的 lambda 表达式内，所有的循环执行都在同一个线程内。当执行到 ab 字符串时，抛出了异常，导致整个线程销毁，无法继续执行。

因此为了不让一个数据出错导致后续的代码都无法执行，如果采用第二种方式来执行可以对代码做出如下修改：

```
private static void printList2(List<String> data, ExecutorService
executorService) {
    executorService.execute(() -> {
        for (String str : data) {
            try {

                if (str.length() == 2) {
                    throw new IllegalArgumentException();
                }

                System.out.println(Thread.currentThread().getName() + "-->" +
str);
            } catch (Exception e) {
                log.error("程序出错，参数data:{},错误详情", JSON.toJSONString(data),
e);
            }
        }
    });
}
```


在实际业务开发过程中，这种问题比较隐蔽，尤其是在异步线程中执行时，如果不加留意，很容易出现上面所描述的问题。

【建议】慎重思考是否 “吞掉” 异常，在二方服务封装时，如捕捉异常，应打印出查询参数和异常详情。

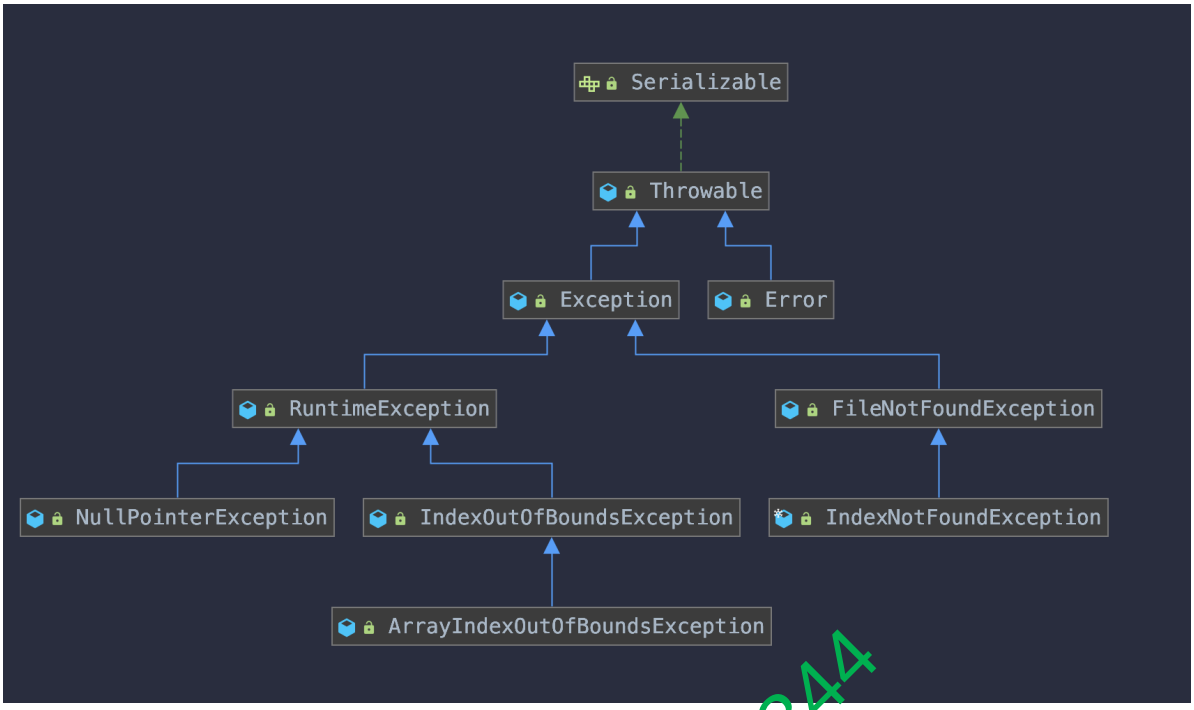
实际开发中，一般都不会吞掉异常，遇到 “吞掉” 异常的场景要慎重思考是否合理。

另外，正如第二部分给出的范例所示，如果调用二方接口出现异常没有打印日志，将对排查问题造成很大的困难。

【建议】要理解好受检异常和非受检异常的区别，避免误用。

Java 中的异常主要分为两类：受检异常和非受检异常。

根据 JLS 异常部分的相关描述，我们可知受检异常主要指编译时强制检查的异常，包括非受检异常之外的其他 `Throwable` 的子类；非受检异常主要指编译器免检异常，通常包括运行时异常类和 Error 相关类。



Error 和 Exception 都是 Throwable 的子类。 RuntimeException 和其子类都属于运行时异常。Error 类和其子类都属于错误类。RuntimeException 及其子类 和 Error 类及其子类 属于非受检异常，除此之外的 其他 Throwable 子类属于受检异常。

大家可以使用 IDEA 自带的类图功能，绘制出自己感兴趣的异常类型，通过上述原则分析其是否属于受检异常。

通常开发中自定义的业务异常（BusinessException）属于非受检异常，会定义为 RuntimeException 的子类。

有些朋友可能会将业务异常定义为受检异常，导致底层抛出后上层调用每层都要被迫处理它。

【建议】努力使失败保持原子性。

正如《Effective Java》第 3 版 第 76 条 努力使失败保持原子性 [^2] 所提到的那样。

我们可以在函数核心代码执行前对参数进行检查，对不满足的条件抛出适当的异常。

实际开发中通常可以使用 `com.google.common.base.Preconditions` 或者 `org.apache.commons.lang3.Validate` 第三方库提供的参数检查工具类来实现。

【建议】如果忽略异常，请给出理由

如果 catch 住异常却没有进行编写任何处理代码，请在注释中给出充分的理由，避免其他人产生困惑，避免留坑。

大家可以参考 `org.springframework.context.support.AbstractApplicationContext#close` 源码：

```
@Override
public void close() {
   synchronized (this.startupShutdownMonitor) {
      doClose();


      if (this.shutdownHook != null) {
```

```
            try {
                Runtime.getRuntime().removeShutdownHook(this.shutdownHook);
            }
            catch (IllegalStateException ex) {

            }
        }
    }
}
```

上面的源码捕捉到 `IllegalStateException` 异常以后没有处理，给出了处理方式和原因: 忽略此异常，因为虚拟机已经正在关闭。

本节主要讲异常的一些处理建议，包括是否要 “吞掉” 异常，循环中的异常处理，以及一些补充建议。希望大家可以重视异常，少趟坑。

下一节我们将讲解打印日志的目的，该打印哪些日志，不该打印哪些日志以及忘记打印日志该怎么办等问题。

}