提升Java编程质量

（如何进行异常处理）

系统架构部

2016 / 12　　李俊超

JD.京东.COM

关于异常几个问题

- 1. 如何界定异常与主流程？
- 2. 截获异常之后：继续抛出、就地处理及异常转化之间的如何权衡？
- 3. 是否所有可能出现的异常都需要截获处理？
- 4. 异常和错误码之间如何选择？
- 5. Checked Exception与Unchecked Exception之间如何选择？
- 6. 使用自定义异常要注意哪些问题？
- 7. Finally 与 return 的执行关系是什么，能否在finally里面使用return，能否从Finally中再次抛出异常？
- 8. 能否通过减少异常使用的方式来优化性能？
- 9. 既不能通过throws子句抛出异常，也不合适在当前环境处理异常时，如何处理？

大纲

JD.COM 京东

1. 正确的认识异常
2. 异常处理的不良习惯
3. 异常处理最佳实践
4. Java经典及JDK官方对异常处理的阐述
5. Java Exception 经验总结

引入异常的深层次原因

JD.COM 京东

传统的错误（异常）处理方法：

- 返回一个错误码

  错误类型和它们的列举值必须标准化

  对于每一个返回码都必须查阅和解释

- 求助于全局标记

  方法标准化

  这多线程环境下，多个线程操作同一个错误码

- 终止程序

  在标准C的函数库中有两个函数用来终止一个程序：exit()和abort()

  容错水平低下：关键性的程序不应该在任何运行期错误存在的情况下突然终止

  abort() 和exit()不调用局部对象的析构函数

引入异常的深层次原因

JD.COM 京东

异常机制的引入

- 把错误处理和真正的工作分开来

- 系统更安全，不会因为小的疏忽使程序意外崩溃

  提供更加可靠的异常处理模型

- 支持分层嵌套

  程序的控制流可以安全的跳转到上层（或上上层）的错误处理模块中去

  不同于return语句，异常处理的控制流是可以安全地跨越一个或多个函数

异常的本质

JD.COM 京东

异常表示没有遵守契约

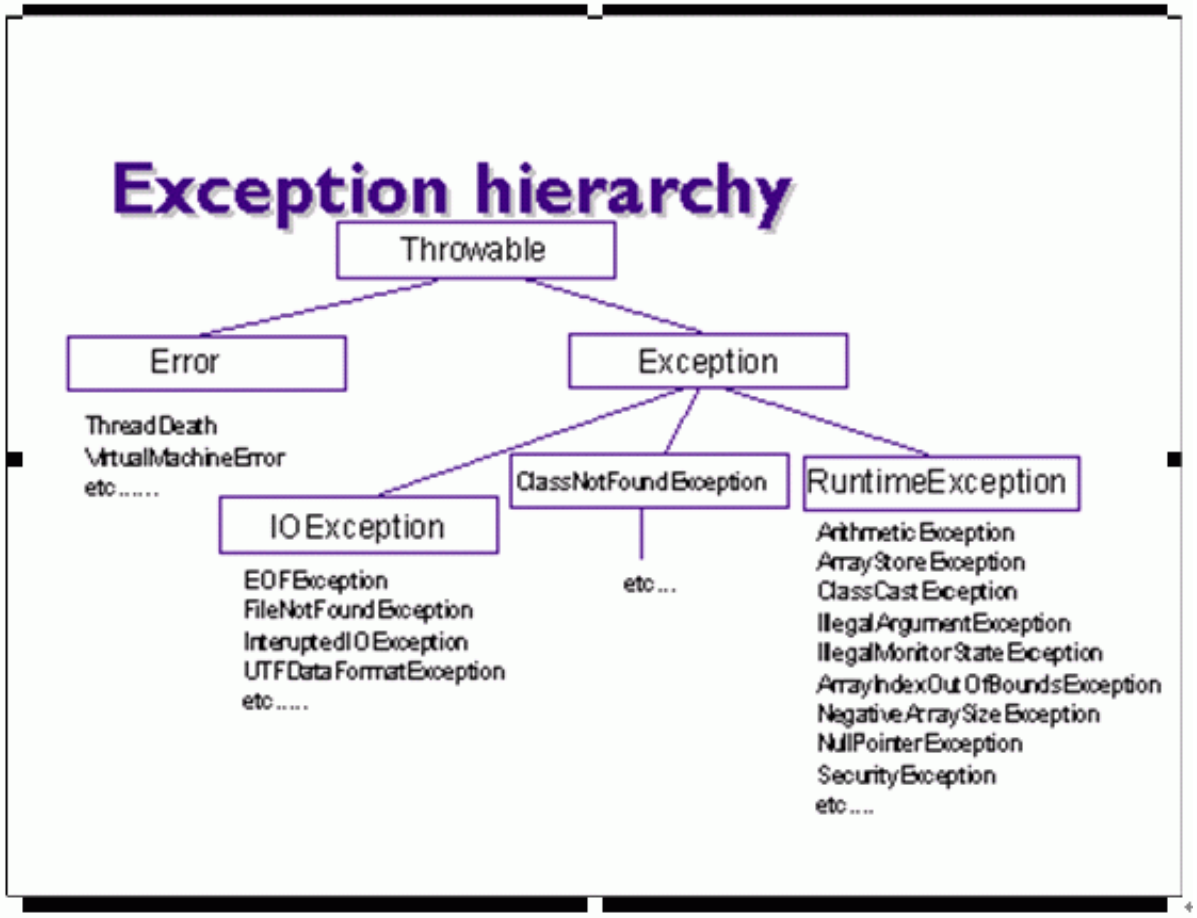
一个情况经常发生，往往意味着它是程序流程的组成部分

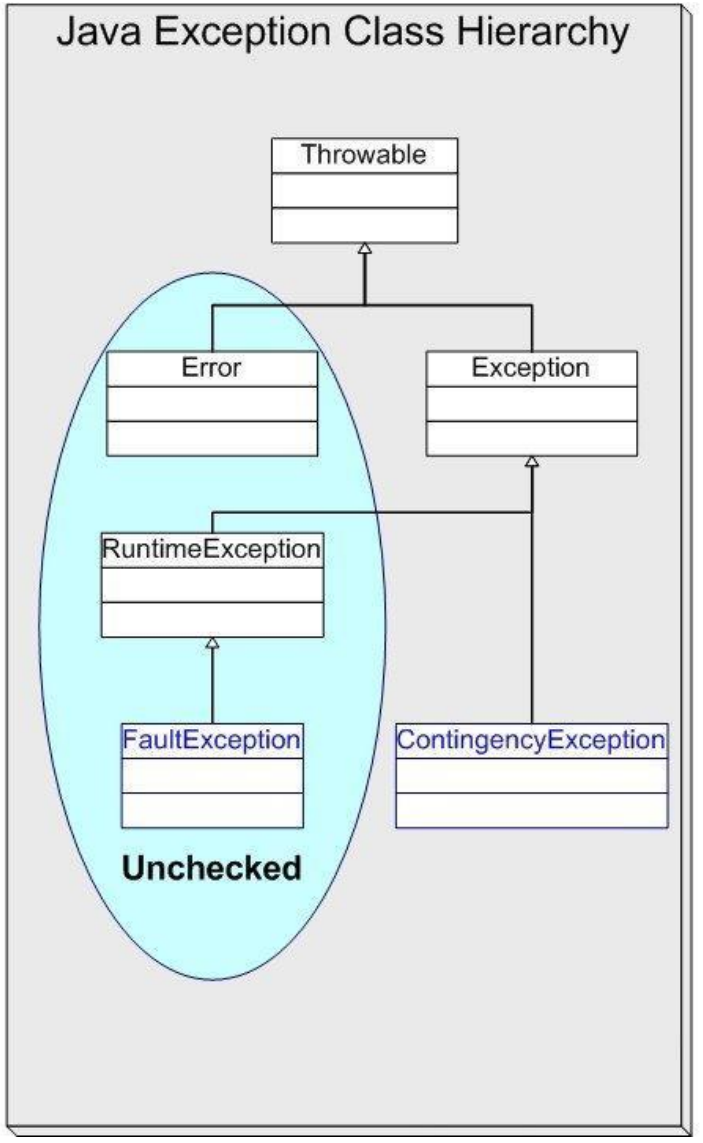
Effective java：只针对非正常的条件才使用异常。

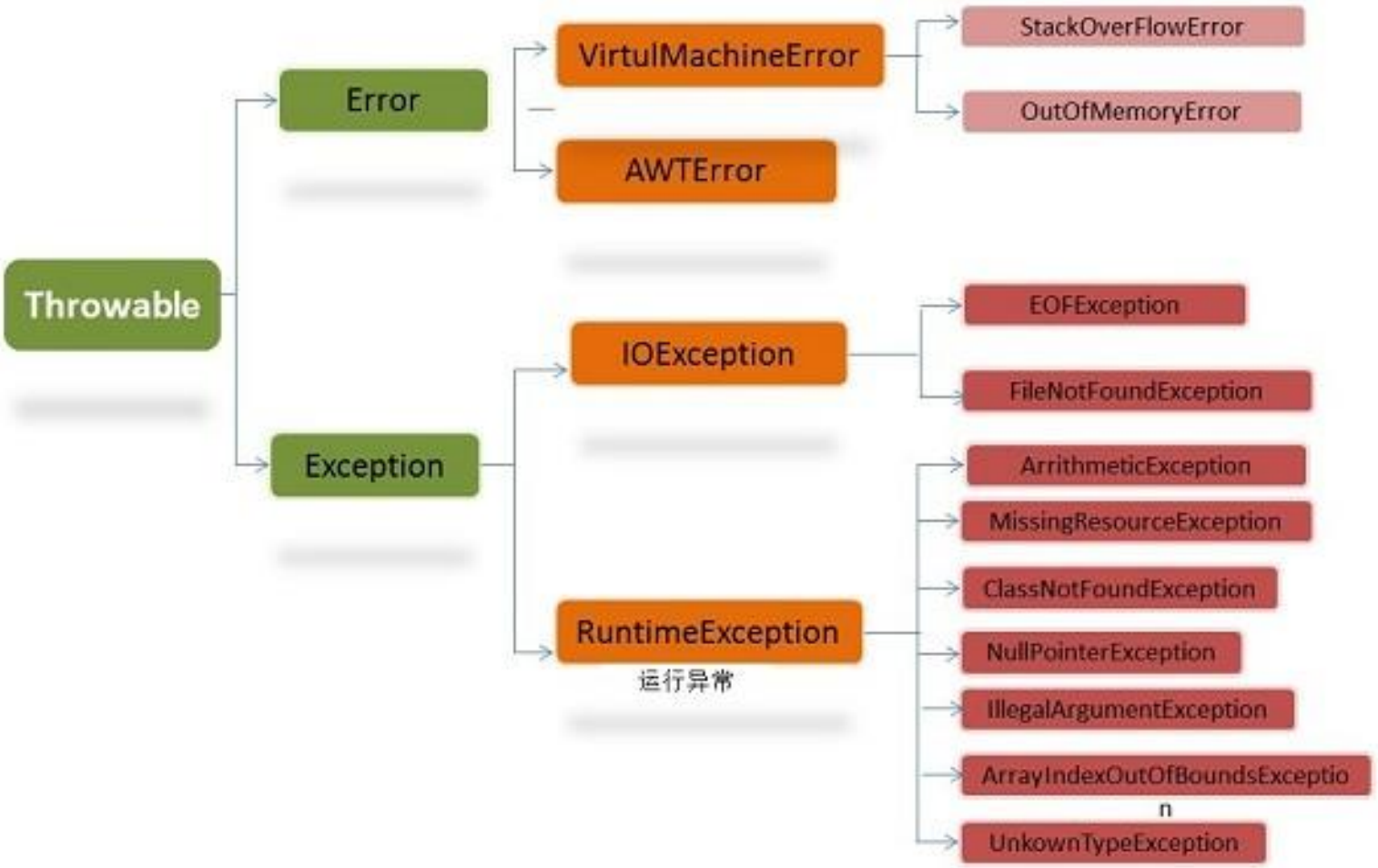
非正常：不知道如何处理的意外情况

反过来：在有充足理由将某情况视为该方法的典型功能的部分时，避免使用异常。

Java异常结构体系


Exception hierarchy
Throwable
Error
Exception
Thread Death
VirtualMachineError
etc.....
IOException
ClassNotFoundException
RuntimeException
EOFException
FileNotFoundException
InterruptedIOException
UTFDataFormatException
etc.....
etc...
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
ArrayIndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
etc....


Java Exception Class Hierarchy
Throwable
Error
Exception
RuntimeException
FaultException
ContingencyException
Unchecked


Throwable
Error
VirtulMachineError
StackOverFlowError
OutOfMemoryError
AWTError
Exception
IOException
EOFException
FileNotFoundException
RuntimeException
运行异常
ArrithmeticException
MissingResourceException
ClassNotFoundException
NullPointerException
IllegalArgumentException
ArrayIndexOutOfBoundsExceptio
n
UnkownTypeException

异常使用的常见不良习惯

```
OutputStreamWriter out = ...
java.sql.Connection conn = ...
try {
   Statement stat = conn.createStatement();
   ResultSet rs = stat.executeQuery("select uid, name from user");
   while (rs.next())
   {
      out.println("ID：" + rs.getString("uid") +    "，姓名：" +
                        rs.getString("name"));
   }
   conn.close();
   out.close();
  }
  catch(Exception ex)
  {
    ex.printStackTrace();
  }
}
```

异常使用的常见不良习惯

- 一、丢弃（忽略）异常：捕获了异常，不作任何**有意义**的处理

```
catch(……)
{
        ex.printStackTrace();
}
```

Java编程中最常见的陋习

异常使用的常见不良习惯

- 二、不指定具体的异常

  catch(Exception ex)

- 用一个catch语句捕获所有的异常！
  - 绝大多数异常都直接或间接从 Exception 派生，catch (Exception ex)就相当于说我们想要处理几乎所有的异常。
- Catch异常时思考一下：真正想要捕获的异常是什么？
  - 不能也不应该去捕获可能出现的所有异常，程序的其他地方还有捕获异常的机会，直至最后由JVM处理；
- 在catch语句中尽可能指定具体的异常类型，必要时使用多个catch。不要试图处理所有可能出现的异常。

异常使用的常见不良习惯

- 三、占用资源不释放
  - 如果程序用到了文件、Socket、JDBC连接之类的资源，即使遇到了异常，也要保证正确释放占用的资源
  - finally保证在try/catch/finally块结束之前，执行清理任务的代码总是有机会执行

异常使用的常见不良习惯

- 四、不记录或传递异常的详细信息
  - 建议：在出现异常时，提供当前正在执行的类、方法和当前正在执行的操作等状态信息，以更适合阅读的方式整理和组织printStackTrace提供的信息。

异常使用的常见不良习惯

- 五、过于庞大的try块
  - 图省事，把大量的语句装入单个巨大的try块
  - 建议：分离各个可能出现异常的段落并分别捕获其异常

异常使用的常见不良习惯

- 六、导致数据不完整或不一致
  - 例子：在循环中发生异常，造成数据不完整的情况。
  - 参考Effective Java中的描述： 失败的方法调用，应该使对象保持在被调用之前的状态，即让失败保持原子性。

如何让失败保持原子性？

- 在所有输入检查完成之后再进行具体的操作；
- 编写一段恢复代码，使对象回滚到操作开始之前的状态；

异常使用的常见不良习惯

重新编写一个异常使用的例子：

```
try {
    resource1 = allocateResource();
    resource2 = allocateAnotherResource();
    resource1.workWith(resource2);
} catch (MyApplicationException mae) {
    handleException(mae);
} finally {
    try {
        if (resource1 != null)
        resource1.close();
    } catch (CloseException ce) {
        handleException(ce);
    } finally {
        try {
            if (resource2 != null)
                resource2.close();
            } catch (CloseException ce) {
                handleException(ce);
            }
        }
    }
```

【注意】：多个资源需要释放的写法

Java异常设计规范

- 一、不要简单地捕获顶层的异常
  - 通常情况下，去捕获当前代码可能抛出的特定的异常，而不是简单的去捕获Exception，Throwable，RuntimeException
- 二、不要在正常的控制流中使用异常
  - 能通过正常的程序逻辑处理的，不要通过异常来处理
- 三、在抛出异常时为开发人员提供非富而有意义的错误消息
  - 使用异常构造器去设置本地化消息
- 四、不要把异常作为公有成员的返回值或输出参数
- 五、不要捕获不应该捕获的异常，不知道在本地如何处理的异常，应当让异常沿着调用栈向上传递。

Java异常设计规范

- 六、避免异常黑洞，忽略异常不做任何处理
  - 切记不要用空的catch块，或者么也不做，或仅仅打印异常信息来忽略异常，以至于好像什么也没有发生一样
  - 正确的处理是对异常进行恢复，对于有些异常在产生时无能为力，可以终止，如果产生时不知道如何处理的，应当抛出有上层进行处理

- 七、代码遇到了严重问题且无法继续安全的执行时，要采取终止模式，不要再抛出异常

- 八、不创建没有意义的异常
  - 不要仅仅为了拥有自己的异常而创建并使用新的异常
  - 创建新的异常类型来传达独一无二的程序错误

- 九、不要对用户手动输入校验抛出异常，使用异常用于参数校验

- 十、使用try-finally来做必要的清理工作

错误码与异常之间的选择

- 所有的系统内部的错误处理一律用异常方式

- 系统之间的调用
  - 引用Jar的方式： 异常处理
  - http/webservice调用：将异常映射为错误码

Checked exception VS unChecked exception

- 权威观点（JDK文档采纳）：

  http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

  Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

  总的原则：

  如果客户端期望并能够从异常中恢复，使用检查型异常；

  如果客户端对于从异常中恢复无能为力，使用非检查型异常；

- 经验：
- 1. 通常将来自外部，不可控因素导致的异常定义为可检查异常

  FileNotFoundException，SocketException，BindException，SAXParseException，SQLException，RMIException，JMSException

- 2. 来自内部，“失误”导致的异常为运行时异常（不可检查异常）

  IllegalArgumentException，NullPointerException，ClassCastException，IndexOutOfBoundsException，UnsupportedOperationException

checked exception VS unchecked exception

- 不要在自己定义的公共API里显式或隐式的抛出 ArithmeticExecption，NullPointerException，ClassCastException，NegativeArrayException，ArrayIndexOutOfBoundsException，NoSuchMethodException……

  出现以上异常，大多数情况表示代码存在缺陷，这些留给JVM去处理

  参考：
  http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

  Generally speaking, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Checked Exception	unChecked Exception
对应Exception类	对应RuntimeException类
必须处理	可不处理
用户可以纠正，然后程序继续正常运行	表明程序有问题，只有开发人员才能处理

Java经典关于异常使用的阐述

《Think in Java》中关于异常的总结：

- 1. Handle problems at the appropriate level. (Avoid catching exceptions unless you know what to do with them).

  在合适的地方处理异常（避免在不知道该如何处理的情景下去捕捉异常）。
- 2. Fix the problem and call the method that caused the exception again.

  解决问题，再调用引发异常的方法。
- 3. Patch things up and continue without retrying the method.

  修复问题，在不重新尝试（造成异常的）方法的前提下继续。
- 4. Calculate some alternative result instead of what the method was supposed to produce.

  用一些别的结果，取代那个（产生异常的） 方法返回的结果。
- 5. Do whatever you can in the current context and rethrow the same（or different） exception to a higher context.

  在当前环境下尽可能解决问题，（在当前环境下不能解决所有问题的情况下）把相同（或者不同的<这里指转化过的>）的异常抛到更高层继续处理。

Java经典关于异常使用的阐述

《Think in Java》中关于异常的总结：

- 7. Terminate the program.

  中止程序。（注：当当前异常导致应用无法继续执行时，选择终止程序）

- 8. Simplify. (If your exception scheme makes things more complicated, then it is painful and annoying to use.)

  简化编码（如果异常把事情搞得更复杂，用起来将非常痛苦）（注：这里指不要去创造太多的异常，使用太多的继承层次去设计异常，导致异常体系过于复杂）

- 9. Make your library and program safer. (This is a short-term investment for debugging, and a long-term investment for application robustness.)

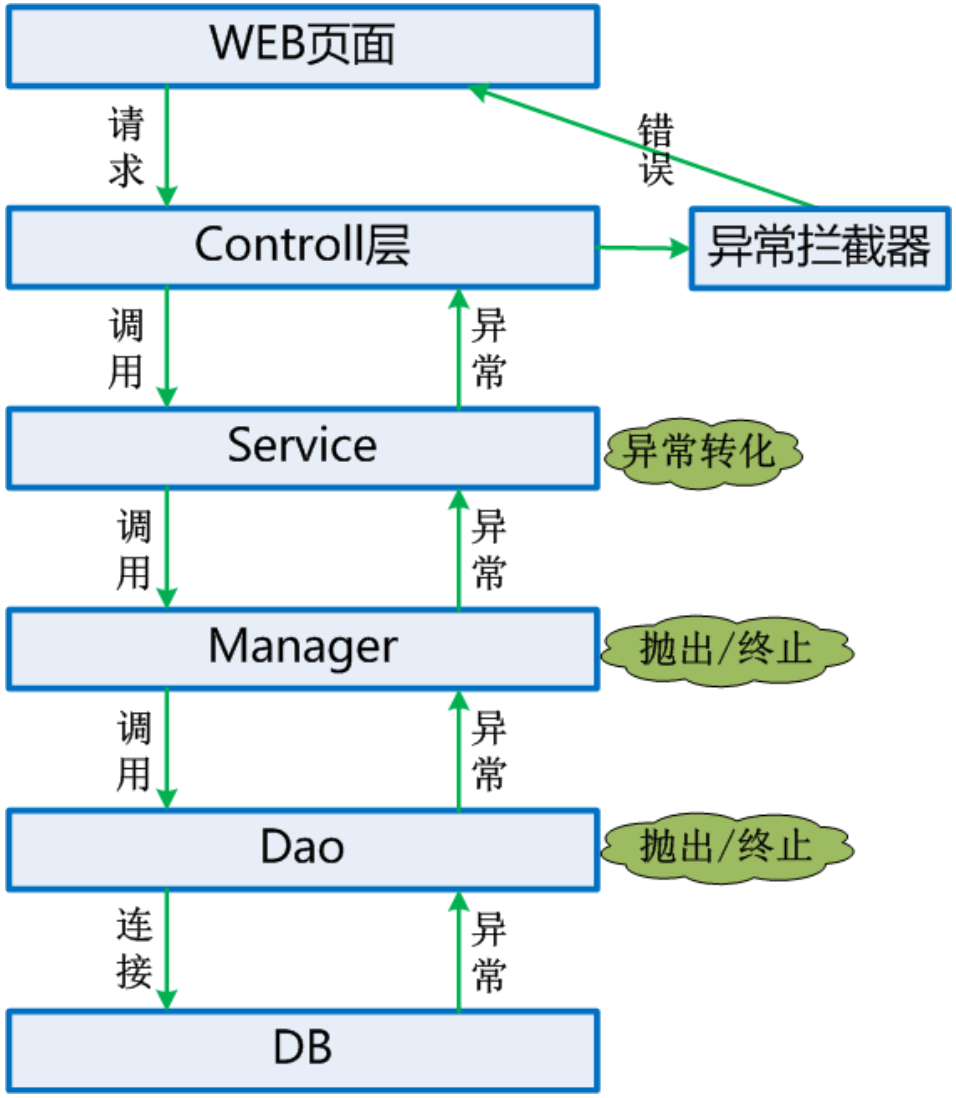
  使你的库和程序更安全（这既是为程序便于调试做短期投资，也是为程序的健壮性做长期投资）。

Java经典关于异常使用的阐述

《Effective Java》中关于异常处理的条款：

- 1. 只针对不正常的条件才使用异常（第39条 57）
- 2. 对于可恢复的条件使用被检查的异常，对于程序错误使用运行时异常（第40条 58）
- 3. 避免不必要地使用被检查的异常（第41条 59）
- 4. 尽量使用标准的异常（第42条 60）
- 5. 抛出的异常要适合于相应的抽象（第43条 61）
- 6. 每个方法抛出的异常都要有文档（第44条 62）
- 7. 在细节消息中包含失败 - 捕获信息（第45条 63）
- 8. 努力使失败保持原子性（第46条 64）
- 9. 不要忽略异常（第47条 65）

Java web项目中的异常



Java web项目中的异常

- 为业务层定义自己的异常
  - ✓ 避免系统级的checked异常对业务系统的深度侵入；
  - ✓ 避免过多的自定义异常；
- 业务层自定义异常
  - ✓ 按子系统来划分；
  - ✓ 从逻辑上划分异常类型，比如：权限相关的，安全相关的；
- 异常的记录
  - ✓ 捕捉到异常就记录是一个不好的习惯；
  - ✓ 异常应该在最初产生的位置记录；
  - ✓ 如果异常是可以处理的，则无需要记录异常；
- 在表示层调用业务方法，一般情况下无需要捕获异常

自定义异常

可以参照Java IOException 的例子：

```
class IOException extends Exception {
    static final long serialVersionUID = 7818375828146090155L;

    public IOException() {
        super();
    }

    public IOException(String message) {
        super(message);
    }

    public IOException(String message, Throwable cause) {
        super(message, cause);
    }

    public IOException(Throwable cause) {
        super(cause);
    }
}
```

异常使用的经验

- 1. 不要因异常可能对性能造成负面影响而使用错误码；
- 2. 避免太深的异常继承层次；
- 3. 要使用合理的，最具针对性的（最底层派生类）异常；
- 4. 不要在在异常消息中泄漏安全信息；
  - 把与安全性有关的信息保存在私有的异常状态中，并确保只有可信赖的代码才能访问
- 5. 不要让调用方根据某个选项来决定是否抛出异常；
- 6. 避免显示的从finally块中抛出异常；

异常使用的经验

- 7. 不要为了仅仅为了通报错误而创建新的异常类型；
  - 不要创建新的异常类型——如果对该错误的处理和对框架中已有异常的并没有什么不同；
- 8. 不要把特定的异常转化为更通用的异常；
- 9. 不要在自己定义的公共API里显式或隐式的抛出ArithmeticExecption，NullPointerException等运行时异常；
- 10. 禁止在finally内部使用return语句，也不建议在catch块中用return的方式处理；

谢谢！
Thank you!

地址：北京市海淀区杏石口路65号益园文创基地C区11号楼
邮编：100195

JD.京东.COM