J2EE项目异常处理

为什么要在J2EE项目中谈异常处理呢？可能许多java初学者都想说：“异常处理不就是try….catch…finally吗？这谁都会啊！”。笔者在初学java时也是这样认为的。如何在一个多层的j2ee项目中定义相应的异常类？在项目中的每一层如何进行异常处理？异常何时被抛出？异常何时被记录？异常该怎么记录？何时需要把checked Exception转化成unchecked Exception ，何时需要把unChecked Exception转化成checked Exception?异常是否应该呈现到前端页面？如何设计一个异常框架？本文将就这些问题进行探讨。

1. JAVA异常处理

在面向过程式的编程语言中，我们可以通过返回值来确定方法是否正常执行。比如在一个c语言编写的程序中，如果方法正确的执行则返回1.错误则返回0。在vb或delphi开发的应用程序中，出现错误时，我们就弹出一个消息框给用户。

通过方法的返回值我们并不能获得错误的详细信息。可能因为方法由不同的程序员编写，当同一类错误在不同的方法出现时，返回的结果和错误信息并不一致。所以java语言采取了一个统一的异常处理机制。

什么是异常？运行时发生的可被捕获和处理的错误。

在java语言中，Exception是所有异常的父类。任何异常都扩展于Exception类。Exception就相当于一个错误类型。如果要定义一个新的错误类型就扩展一个新的Exception子类。采用异常的好处还在于可以精确的定位到导致程序出错的源代码位置,并获得详细的错误信息。

Java异常处理通过五个关键字来实现，try,catch,throw ,throws, finally。具体的异常处理结构由try….catch….finally块来实现。try块存放可能出现异常的java语句，catch用来捕获发生的异常，并对异常进行处理。Finally块用来清除程序中未释放的资源。不管理try块的代码如何返回，finally块都总是被执行。

一个典型的异常处理代码

java 代码

```
1. public String getPassword(String userId)throws DataAccessException{
2. String sql = “select password from userinfo where userid=’”+userId
+”’”;
3. String password = null;
4. Connection con = null;
5. Statement s = null;
6. ResultSet rs = null;
```

```
7. try{
8. con = getConnection();//获得数据连接
9. s = con.createStatement();
10. rs = s.executeQuery(sql);
11. while(rs.next()){
12. password = rs.getString(1);
13. }
14. rs.close();
15. s.close();
16. }
17. Catch(SqlException ex){
18. throw new DataAccessException(ex);
19. }
20. finally{
21. try{
22. if(con != null){
23. con.close();
24. }
25. }
26. Catch(SQLException sqlEx){
27. throw new DataAccessException("关闭连接失败!",sqlEx);
28. }
29. }
30. return password;
31. }
```

可以看出Java的异常处理机制具有的优势：

给错误进行了统一的分类，通过扩展Exception类或其子类来实现。从而避免了相同的错误可能在不同的方法中具有不同的错误信息。在不同的方法中出现相同的错误时，只需要throw 相同的异常对象即可。

获得更为详细的错误信息。通过异常类，可以给异常更为详细，对用户更为有用的错误信息。以便于用户进行跟踪和调试程序。

把正确的返回结果与错误信息分离。降低了程序的复杂度。调用者无需要对返回结果进行更多的了解。

强制调用者进行异常处理，提高程序的质量。当一个方法声明需要抛出一个异常

时，那么调用者必须使用try....catch块对异常进行处理。当然调用者也可以让异常继续往上一层抛出。

2． Checked 异常 还是 unChecked 异常?

Java异常分为两大类:checked 异常和unChecked 异常。所有继承java.lang.Exception 的异常都属于checked异常。所有继承java.lang.RuntimeException的异常都属于unChecked异常。

当一个方法去调用一个可能抛出checked异常的方法，必须通过try...catch块对异常进行捕获进行处理或者重新抛出。

我们看看Connection接口的createStatement()方法的声明。

public Statement createStatement() throws SQLException;

SQLException是checked异常。当调用createStatement方法时，java强制调用者必须对SQLException进行捕获处理。

java 代码

```
1. public String getPassword(String userId){
2. try{
3. ……
4. Statement s = con.createStatement();
5. ……
6. Catch(SQLException sqlEx){
7. ……
8. }
9. ……
10. }
```

或者

java 代码

```
1. public String getPassword(String userId)throws SQLException{
2. Statement s = con.createStatement();
3. }
```

(当然，像Connection,Satement这些资源是需要及时关闭的，这里仅是为了说明checked 异常必须强制调用者进行捕获或继续抛出)

unChecked异常也称为运行时异常，通常RuntimeException都表示用户无法恢复的异常，如无法获得数据库连接，不能打开文件等。虽然用户也可以像处理checked异常一样捕获unChecked异常。但是如果调用者并没有去捕获unChecked异常时，编译器并不会强制你那么做。

比如一个把字符转换为整型数值的代码如下：

java 代码

```
1. String str = “123”;
2. int value = Integer.parseInt(str);
```

parseInt的方法签名为：

java 代码

```
1. public static int parseInt(String s) throws NumberFormatException
```

当传入的参数不能转换成相应的整数时，将会抛出NumberFormatException。

因为NumberFormatException扩展于RuntimeException，是unChecked异常。

所以调用parseInt方法时无需要try...catch

因为java不强制调用者对unChecked异常进行捕获或往上抛出。所以程序员总是喜欢抛出unChecked异常。或者当需要一个新的异常类时，总是习惯的从RuntimeException扩展。当你去调用它些方法时，如果没有相应的catch块，编译器也总是让你通过,同时你也根本无需要去了解这个方法倒底会抛出什么异常。看起来这似乎倒是一个很好的办法，但是这样做却是远离了java异常处理的真实意图。并且对调用你这个类的程序员带来误导，因为调用者根本不知道需要在什么情况下处理异常。而checked异常可以明确的告诉调用者，调用这个类需要处理什么异常。如果调用者不去处理，编译器都会提示并且是无法编译通过的。当然怎么处理是由调用者自己去决定的。

所以Java推荐人们在应用代码中应该使用checked异常。就像我们在上节提到运用异常的好外在于可以强制调用者必须对将会产生的异常进行处理。包括在《java Tutorial》等java官方文档中都把checked异常作为标准用法。

使用checked异常，应意味着有许多的try...catch在你的代码中。当在编写和处理越来越多的try...catch块之后，许多人终于开始怀疑checked异常倒底是否应该作为标准用法了。

甚至连大名鼎鼎的《thinking in java》的作者Bruce Eckel也改变了他曾经的想法。Bruce Eckel甚至主张把unChecked异常作为标准用法。并发表文章，以试验checked异常是否应该从java中去掉。Bruce Eckel语：“当少量代码时，checked异常无疑是十分优雅的构思，并有助于避免了许多潜在的错误。但是经验表明，对大量代码来说结果正好相反”

关于checked异常和unChecked异常的详细讨论可以参考

Alan Griffiths http://www.octopull.demon.co.uk/java/ExceptionalJava.html

Bruce Eckel http://www.mindView.net/Etc/Disscussions/CheckedExceptions

《java Tutorial》

http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html
使用checked异常会带来许多的问题。
checked异常导致了太多的try...catch 代码
可能有很多checked异常对开发人员来说是无法合理地进行处理的，比如SQLException。而开发人员却不得不去进行try...catch。当开发人员对一个checked异常无法正确的处理时，通常是简单的把异常打印出来或者是干脆什么也不干。特别是对于新手来说，过多的checked异常让他感到无所适从。
java 代码

```
1. try{
2. ……
3. Statement s = con.createStatement();
4. ……
5. Catch(SQLException sqlEx){
6. sqlEx.PrintStackTrace();
7. }
8. 或者
9. try{
10. ……
11. Statement s = con.createStatement();
12. ……
13. Catch(SQLException sqlEx){
14. //什么也不干
15. }
```

checked异常导致了许多难以理解的代码产生
当开发人员必须去捕获一个自己无法正确处理的checked异常，通常的是重新封装成一个新的异常后再抛出。这样做并没有为程序带来任何好处。反而使代码晚难以理解。
就像我们使用JDBC代码那样，需要处理非常多的try...catch.，真正有用的代码被包含在try...catch之内。使得理解这个方法变理困难起来
checked异常导致异常被不断的封装成另一个类异常后再抛出
java 代码

```
1. public void methodA()throws ExceptionA{
2. .....
3. throw new ExceptionA();
```

```
4. }
5. public void methodB()throws ExceptionB{
6. try{
7. methodA();
8. ……
9. }catch(ExceptionA ex){
10. throw new ExceptionB(ex);
11. }
12. }
13. Public void methodC()throws ExceptinC{
14. try{
15. methodB();
16. …
17. }
18. catch(ExceptionB ex){
19. throw new ExceptionC(ex);
20. }
21. }
```

我们看到异常就这样一层层无休止的被封装和重新抛出。

checked异常导致破坏接口方法

一个接口上的一个方法已被多个类使用，当为这个方法额外添加一个checked异常时，那么所有调用此方法的代码都需要修改。

可见上面这些问题都是因为调用者无法正确的处理checked异常时而被迫去捕获和处理，被迫封装后再重新抛出。这样十分不方便，并不能带来任何好处。在这种情况下通常使用unChecked异常。

chekced异常并不是无一是处，checked异常比传统编程的错误返回值要好用得多。通过编译器来确保正确的处理异常比通过返回值判断要好得多。

如果一个异常是致命的，不可恢复的。或者调用者去捕获它没有任何益处，使用unChecked异常。

如果一个异常是可以恢复的，可以被调用者正确处理的，使用checked异常。

在使用unChecked异常时，必须在在方法声明中详细的说明该方法可能会抛出的unChekced异常。由调用者自己去决定是否捕获unChecked异常

倒底什么时候使用checked异常，什么时候使用unChecked异常？并没有一个绝对的标准。但是笔者可以给出一些建议

当所有调用者必须处理这个异常，可以让调用者进行重试操作；或者该异常相当于该方法的第二个返回值。使用checked异常。
这个异常仅是少数比较高级的调用者才能处理，一般的调用者不能正确的处理。使用unchecked异常。有能力处理的调用者可以进行高级处理，一般调用者干脆就不处理。
这个异常是一个非常严重的错误，如数据库连接错误，文件无法打开等。或者这些异常是与外部环境相关的。不是重试可以解决的。使用unchecked异常。因为这种异常一旦出现，调用者根本无法处理。
如果不能确定时，使用unchecked异常。并详细描述可能会抛出的异常,以让调用者决定是否进行处理。
3. 设计一个新的异常类
在设计一个新的异常类时，首先看看是否真正的需要这个异常类。一般情况下尽量不要去设计新的异常类，而是尽量使用java中已经存在的异常类。
如
java 代码

1. IllegalArgumentException, UnsupportedOperationException

不管是新的异常是chekced异常还是unChecked异常。我们都必须考虑异常的嵌套问题。
java 代码

1. public void methodA()throws ExceptionA{
2. .....
3. throw new ExceptionA();
4. }

方法methodA声明会抛出ExceptionA.
public void methodB()throws ExceptionB
methodB声明会抛出ExceptionB,当在methodB方法中调用methodA时，ExceptionA是无法处理的,所以ExceptionA应该继续往上抛出。一个办法是把methodB声明会抛出ExceptionA.但这样已经改变了MethodB的方法签名。一旦改变，则所有调用methodB的方法都要进行改变。
另一个办法是把ExceptionA封装成ExceptionB，然后再抛出。如果我们不把ExceptionA封装在ExceptionB中，就丢失了根异常信息，使得无法跟踪异常的原始出处。
java 代码

1. public void methodB()throws ExceptionB{

```
2. try{
3. methodA();
4. ……
5. }catch(ExceptionA ex){
6. throw new ExceptionB(ex);
7. }
8. }
```

如上面的代码中，ExceptionB嵌套一个ExceptionA.我们暂且把ExceptionA称为“起因异常”,因为ExceptionA导致了ExceptionB的产生。这样才不使异常信息丢失。

所以我们在定义一个新的异常类时，必须提供这样一个可以包含嵌套异常的构造函数。并有一个私有成员来保存这个“起因异常”。

java 代码

```
1. public Class ExceptionB extends Exception{
2. private Throwable cause;
3. public ExceptionB(String msg, Throwable ex){
4. super(msg);
5. this.cause = ex;
6. }
7. public ExceptionB(String msg){
8. super(msg);
9. }
10. public ExceptionB(Throwable ex){
11. this.cause = ex;
12. }
13. }
```

当然，我们在调用printStackTrace方法时，需要把所有的“起因异常”的信息也同时打印出来。所以我们需要覆写printStackTrace方法来显示全部的异常栈跟踪。包括嵌套异常的栈跟踪。

java 代码

```
1. public void printStackTrace(PrintStrean ps){
2. if(cause == null){
3. super.printStackTrace(ps);
4. }else{
```

```
5. ps.println(this);
6. cause.printStackTrace(ps);
7. }
8. }
```

一个完整的支持嵌套的checked异常类源码如下。我们在这里暂且把它叫做NestedException

java 代码

```
1. public NestedException extends Exception{
2. private Throwable cause;
3. public NestedException (String msg){
4. super(msg);
5. }
6. public NestedException(String msg, Throwable ex){
7. super(msg);
8. This.cause = ex;
9. }
10. public Throwable getCause(){
11. return (this.cause == null ? this :this.cause);
12. }
13. public getMessage(){
14. String message = super.getMessage();
15. Throwable cause = getCause();
16. if(cause != null){
17. message = message + “;nested Exception is ” + cause;
18. }
19. return message;
20. }
21. public void printStackTrace(PrintStream ps){
22. if(getCause == null){
23. super.printStackTrace(ps);
24. }else{
25. ps.println(this);
26. getCause().printStackTrace(ps);
27. }
```

```
28. }
29. public void printStackTrace(PrintWrite pw){
30. if(getCause() == null){
31. super.printStackTrace(pw);
32. }
33. else{
34. pw.println(this);
35. getCause().printStackTrace(pw);
36. }
37. }
38. public void printStackTrace(){
39. printStackTrace(System.error);
40. }
41. }
```

同样要设计一个unChecked异常类也与上面一样。只是需要继承RuntimeException。

4． 如何记录异常

作为一个大型的应用系统都需要用日志文件来记录系统的运行，以便于跟踪和记录系统的运行情况。系统发生的异常理所当然的需要记录在日志系统中。

java 代码

```
1. public String getPassword(String userId)throws NoSuchUserException{
2. UserInfo user = userDao.queryUserById(userId);
3. If(user == null){
4. Logger.info(“找不到该用户信息,userId=”+userId);
5. throw new NoSuchUserException(“找不到该用户信
息,userId=”+userId);
6. }
7. else{
8. return user.getPassword();
9. }
10. }
11. public void sendUserPassword(String userId)throws Exception {
12. UserInfo user = null;
13. try{
```

```
14. user = getPassword(userId);
15. //.....
16. sendMail();
17. //
18. }catch(NoSuchUserException ex)(
19. logger.error(“找不到该用户信息:”+userId+ex);
20. throw new Exception(ex);
21. }
```

我们注意到，一个错误被记录了两次.在错误的起源位置我们仅是以info级别进行记录。而在sendUserPassword方法中，我们还把整个异常信息都记录了。

笔者曾看到很多项目是这样记录异常的，不管三七二一，只有遇到异常就把整个异常全部记录下。如果一个异常被不断的封装抛出多次，那么就被记录了多次。

那么异常倒底该在什么地方被记录?

异常应该在最初产生的位置记录!

如果必须捕获一个无法正确处理的异常，仅仅是把它封装成另外一种异常往上抛出。不必再次把已经被记录过的异常再次记录。

如果捕获到一个异常，但是这个异常是可以处理的。则无需要记录异常

java 代码

```
1. public Date getDate(String str){
2. Date applyDate = null;
3. SimpleDateFormat format = new SimpleDateFormat(“MM/dd/yyyy”);
4. try{
5. applyDate = format.parse(applyDateStr);
6. }
7. catch(ParseException ex){
8. //乎略，当格式错误时，返回null
9. }
10. return applyDate;
11. }
```

捕获到一个未记录过的异常或外部系统异常时，应该记录异常的详细信息

java 代码

```
1. try{
2. ……
3. String sql=”select * from userinfo”;
```

```
4. Statement s = con.createStatement();
5. ……
6. Catch(SQLException sqlEx){
7. Logger.error(“sql执行错误”+sql+sqlEx);
8. }
```

究竟在哪里记录异常信息，及怎么记录异常信息，可能是见仁见智的问题了。甚至有些系统让异常类一记录异常。当产生一个新异常对象时，异常信息就被自动记录。

java 代码

```
1. public class BusinessException extends Exception {
2. private void logTrace() {
3. StringBuffer buffer=new StringBuffer();
4. buffer.append("Business Error in Class: ");
5. buffer.append(getClassName());
6. buffer.append(",method: ");
7. buffer.append(getMethodName());
8. buffer.append(",messsage: ");
9. buffer.append(this.getMessage());
10. logger.error(buffer.toString());
11. }
12. public BusinessException(String s) {
13. super(s);
14. race();
15. }
```

这似乎看起来是十分美妙的，其实必然导致了异常被重复记录。同时违反了“类的职责分配原则”，是一种不好的设计。记录异常不属于异常类的行为，记录异常应该由专门的日志系统去做。并且异常的记录信息是不断变化的。我们在记录异常同应该给更丰富些的信息。以利于我们能够根据异常信息找到问题的根源，以解决问题。

虽然我们对记录异常讨论了很多，过多的强调这些反而使开发人员更为疑惑，一种好的方式是为系统提供一个异常处理框架。由框架来决定是否记录异常和怎么记录异常。而不是由普通程序员去决定。但是了解些还是有益的。

5. J2EE项目中的异常处理

目前,J2ee项目一般都会从逻辑上分为多层。比较经典的分为三层：表示层，业

务层，集成层(包括数据库访问和外部系统的访问)。
J2ee项目有着其复杂性，J2ee项目的异常处理需要特别注意几个问题。
在分布式应用时，我们会遇到许多checked异常。所有RMI调用（包括EJB远程接口调用）都会抛出java.rmi.RemoteException;同时RemoteException是checked异常，当我们在业务系统中进行远程调用时，我们都需要编写大量的代码来处理这些checked异常。而一旦发生RemoteException这些checked异常对系统是非常严重的，几乎没有任何进行重试的可能。也就是说，当出现RemoteException这些可怕的checked异常，我们没有任何重试的必要性，却必须要编写大量的try...catch代码去处理它。一般我们都是在最底层进行RMI调用，只要有一个RMI调用，所有上层的接口都会要求抛出RemoteException异常。因为我们处理RemoteException的方式就是把它继续往上抛。这样一来就破坏了我们业务接口。RemoteException这些J2EE系统级的异常严重的影响了我们的业务接口。我们对系统进行分层的目的就是减少系统之间的依赖，每一层的技术改变不至于影响到其它层。
java 代码

```
1. //
2. public class UserSoaImpl implements UserSoa{
3. public UserInfo getUserInfo(String userId)throws RemoteException{
4. //.....
5. 远程方法调用.
6. //.....
7. }
8. }
9. public interface UserManager{
10. public UserInfo getUserInfo(Stirng userId)throws RemoteException;
11. }
```

同样JDBC访问都会抛出SQLException的checked异常。
为了避免系统级的checked异常对业务系统的深度侵入，我们可以为业务方法定义一个业务系统自己的异常。针对像SQLException,RemoteException这些非常严重的异常，我们可以新定义一个unChecked的异常，然后把SQLException，RemoteException封装成unChecked异常后抛出。
如果这个系统级的异常是要交由上一级调用者处理的，可以新定义一个checked的业务异常，然后把系统级的异常封存装成业务级的异常后再抛出。
一般地，我们需要定义一个unChecked异常，让集成层接口的所有方法都声明

抛出这unChecked异常。

java 代码

```
1. public DataAccessException extends RuntimeException{
2. ……
3. }
4. public interface UserDao{
5. public String getPassword(String userId)throws DataAccessException;
6. }
7. public class UserDaoImpl implements UserDAO{
8. public String getPassword(String userId)throws DataAccessException{
9. String sql = “select password from userInfo where userId=
‘”+userId+”’”;
10. try{
11. …
12. //JDBC调用
13. s.executeQuery(sql);
14. …
15. }catch(SQLException ex){
16. throw new DataAccessException(“数据库查询失败”+sql,ex);
17. }
18. }
19. }
```

定义一个checked的业务异常，让业务层的接口的所有方法都声明抛出Checked异常.

java 代码

```
1. public class BusinessException extends Exception{
2. …..
3. }
4. public interface UserManager{
5. public Userinfo copyUserInfo(Userinfo user)throws BusinessException{
6. Userinfo newUser = null;
7. try{
8. newUser = (Userinfo)user.clone();
9. }catch(CloneNotSupportedException ex){
```

```
10. throw new BusinessException(“不支持clone方
法:”+Userinfo.class.getName(),ex);
11. }
12. }
13. }
```

J2ee表示层应该是一个很薄的层，主要的功能为：获得页面请求，把页面的参数组装成POJO对象，调用相应的业务方法，然后进行页面转发，把相应的业务数据呈现给页面。表示层需要注意一个问题，表示层需要对数据的合法性进行校验，比如某些录入域不能为空，字符长度校验等。

J2ee从页面所有传给后台的参数都是字符型的，如果要求输入数值或日期类型的参数时，必须把字符值转换为相应的数值或日期值。

如果表示层代码校验参数不合法时，应该返回到原始页面，让用户重新录入数据，并提示相关的错误信息。

通常把一个从页面传来的参数转换为数值，我们可以看到这样的代码

java 代码

```
1. ModeAndView handleRequest(HttpServletRequest
request,HttpServletResponse response)throws Exception{
2. String ageStr = request.getParameter(“age”);
3. int age = Integer.parse(ageStr);
4. ……
5. String birthDayStr = request.getParameter(“birthDay”);
6. SimpleDateFormat format = new SimpleDateFormat(“MM/dd/yyyy”);
7. Date birthDay = format.parse(birthDayStr);
8. }
```

上面的代码应该经常见到，但是当用户从页面录入一个不能转换为整型的字符或一个错误的日期值。

Integer.parse()方法被抛出一个NumberFormatException的unChecked异常。但是这个异常绝对不是一个致命的异常,一般当用户在页面的录入域录入的值不合法时，我们应该提示用户进行重新录入。但是一旦抛出unchecked异常，就没有重试的机会了。像这样的代码造成大量的异常信息显示到页面。使我们的系统看起来非常的脆弱。

同样，SimpleDateFormat.parse()方法也会抛出ParseException的unChecked异常。

这种情况我们都应该捕获这些unChecked异常，并给提示用户重新录入。

java 代码

```
1. ModeAndView handleRequest(HttpServletRequest
request,HttpServletResponse response)throws Exception{
2. String ageStr = request.getParameter(“age”);
3. String birthDayStr = request.getParameter(“birthDay”);
4. int age = 0;
5. Date birthDay = null;
6. try{
7. age=Integer.parse(ageStr);
8. }catch(NumberFormatException ex){
9. error.reject(“age”,”不是合法的整数值”);
10. }
11. ……
12. try{
13. SimpleDateFormat format = new SimpleDateFormat(“MM/dd/yyyy”);
14. birthDay = format.parse(birthDayStr);
15. }catch(ParseException ex){
16. error.reject(“birthDay”,”不是合法的日期，请录入’MM/dd/yyy’格式的
日期”);
17. }
18. }
```

在表示层一定要弄清楚调用方法的是否会抛出unChecked异常，什么情况下会抛出这些异常，并作出正确的处理。

在表示层调用系统的业务方法，一般情况下是无需要捕获异常的。如果调用的业务方法抛出的异常相当于第二个返回值时，在这种情况下是需要捕获