笔记1

第11章
异常 断言 日志与调试

在哪些时候会出现异常呢

1 用户输入错误
2 设备错误
3 物理限制
4 代码错误

关于异常的分类
所有的异常都是继承与Thorwable 然后下一层立即分解为俩个分支 一个是error 一个是exception
error是java内部的运行时的系统错误和资源耗尽错误 这种情况程序没有办法解决
主要是exception 这个又分为俩个分支 一个runtimeexception
另一个是包含其他异常 由程序错误导致的异常属于RuntimeException 而程序本身没有问题 但由于像I/O错误这类问题导致异常属于其他异常
派生于RuntimeException的异常包含下面几种情况
1 错误的类型转换
2 数组访问越界
3 访问空指针
不是派生于RuntimeException的异常包括
1 试图在文件尾部后面读取数据
2 试图打开一个不存在的文件
3 试图根据给定的字符串查找Class对象 而这个字符串表示的类并不存在

java语言规范将派生于Error类或RuntimeException类的所有异常称为未检查异常 所有其他的异常被称为已检查异常

声明已检测异常 就是在方法上使用throws子句
在遇到以下情况下应该抛出异常
1 调用一个已抛出检测异常的方法
2 程序运行时发现错误 并且利用throw子句抛出一个已检测异常
3 程序出现错误
4 java虚拟机和运行时出现的内部错误

在前面看到 对于一个已经存在的异常类 将其抛出非常容易 在这种情况下 :
1 找到一个合适的异常类
2 创建这个类的一个对象
3 将对象抛出
一旦方法抛出了异常 这个方法就不可能返回到调用者 也就是说 不必为返回的默认值或错误代码担忧
抛出异常在代码中经常使用 用来明确错误以及程序在出现错误的情况下安全退出

```
/**
 * 自定义异常
* Created by han on 2016/7/13.
 */
public class ZIdingyiException extends RuntimeException {
    ZIdingyiException(){

    }
    ZIdingyiException(String username){
      super(username);
       }
}
```

这里的自定义异常的具体通知消息是可以通过Throwable来通过toString方法来打印出来

注意 如果在编写一个覆盖超类的方法 而这个方法又没有抛出异常 那么这个方法就必须捕获方法代码中出现的每一个已检测异常 不允许在子类的throws说明符中出现超过超类方法所列出的异常类范围

try{

access the databases

}catch(Exception e){
 logger.log(level,message,e);
throw e;
}

现在编译器已经相对以前改进了不少 以前java编译器会查看catch块中的throw子句 然后查看e的类型 会指出这个方法可以抛出任何Exception而不是SqlException 现在改进了 编译器会跟踪到e来着try块 假设try块中仅有的已检测异常是SQLException实例，另外 假设e在catch块中未改变 将外围方法声明为throws SQLException就是合法的

建议独立使用try/catch 与try/finally 语句块 可以变得清晰 如下所示

```
InputStream in;
try{
try {
//代码
}finally {
        in.close();
}

}catch (IOException e){
    e.printStackTrace();
}
```

当finally包含return子句时会出现想不到的后果 会进行覆盖原有的return子句

带资源的try语句

```
try(Scanner in=new Scanner(FileInputStream("D://目录.txt"))){

while (in.hasNext())
        System.out.println(in.next());
}
```

这有就可以自己自动的进行关闭close的操作 这样就相当于常规的try/finally 关闭方式

这样可以获得在这个方法内部的步骤出错导致发生的异常 可以通过se.getCause（）得到原始异常

```
try {
servletTest.ceshi();
} catch (FileNotFoundException se) {
se.printStackTrace();
Throwable e = se.getCause();
System.out.println(e);
```

这样能让用户抛出子系统中的高级异常 而不会丢失原始异常的细节

分析堆栈跟踪元素

一种方法是Throwable类的printStackTrace方法访问堆栈跟踪的文本描述信息
另一种更灵活的方法是使用getStackTrace方法 它会到得到StackTraceElement对象的一个数组 可以在你的程序中分析这个对象数组

```
public  static int factorial(int n){
System.out.println("factorial("+n+");");
Throwable t=new Throwable();
StackTraceElement[] frames=t.getStackTrace();//当前的线程的文件名和当前执行的代码行号的方法
for (StackTraceElement f:frames){
System.out.println(f);
int r;
if(n <=1)r=1;
else r=n*factorial(n-1);
System.out.println("return"+r);
return r;
    }
return Integer.parseInt(null);
}
```

stackTraceElement还有能够获得文件名和当前执行的代码行号的方法 同时 还有能够获得类名和方法名的方法 toString方法将产生一个格式化的字符串 其中包含所获得的信息

还有静态的**Thread.getAllStackTraces();**方法能够获得所有线程的堆栈跟踪

```
Map<Thread,StackTraceElement[]> map=Thread.getAllStackTraces();
for (Thread t:map.keySet()){
    StackTraceElement[] frames=map.get(t);
//。。。代码。。
}
```

使用异常的技巧

1异常处理不能代替简单的测试

2 不要过分的细化异常

3 利用异常层次结构

4不要压制异常

5当检测错误时 苛刻比放任更好

6不要羞于传递异常

断言

http://blog.csdn.net/z_dendy/article/details/12008629这里是关于断言讲解的博客
1 assert <boolean表达式>
如果<boolean表达式>为true，则程序继续执行。
如果为false，则程序抛出AssertionError，并终止执行。

2、assert <boolean表达式>：<错误信息表达式>
如果<boolean表达式>为true，则程序继续执行。
如果为false，则程序抛出java.lang.AssertionError，并输入<错误信息表达式>。

关于断言的陷阱说明
assert关键字用法简单，但是使用assert往往会让你陷入越来越深的陷阱中。应避免使用。笔者经过研究，总结了以下原因：

1、assert关键字需要在运行时候显式开启才能生效，否则你的断言就没有任何意义。而现在主流的Java IDE工具默认都没有开启-ea断言检查功能。这就意味着你如果使用IDE工具编码，调试运行时候会有一定的麻烦。并且，对于Java Web应用，程序代码都是部署在容器里面，你没法直接去控制程序的运行，如果一定要开启-ea的开关，则需要更改Web容器的运行配置参数。这对程序的移植和部署都带来很大的不便。

2、用assert代替if是陷阱之二。assert的判断和if语句差不多，但两者的作用有着本质的区别：**assert关键字本意上是为测试调试程序时使用的**，但如果不小心用assert来控制了程序的业务流程，那在测试调试结束后去掉assert关键字就意味着修改了程序的正常的逻辑。

3、assert断言失败将面临程序的退出。这在一个生产环境下的应用是绝不能容忍的。一般都是通过异常处理来解决程序中潜在的错误。但是使用断言就很危险，一旦失败系统就挂了。

但是注意 在maven中的test模块中 就是开启断言机制的。所以可以使用 还是值得使用的

在java中有三种处理系统错误的机制
1 异常
2 断言
3 日志

关于日志

在日志中有几个部分
1 基本日志

2 高级日志
3 处理器
4 过滤器
5 格式化器

这是一条基本日志的语句

```
Logger.getGlobal().info("File->Open menu item selected");
```

高级日志

在日志记录器中也是有层次结构的
有7个级别
1 SERVERE
2 WARNING
3 INFO
4 CONFIG
5 FINE
6 FINEST
默认情况下 只记录前三个级别 也可以设置为其他的级别

关于本地化

将日志本地化的话 本地化的应用程序包含资源包中的本地特定信息 资源包由各个地区的映射集合组成 在请求日志记录器的适合可以指定一个资源包 就是下面的getLogger方法

```
Logger logger = Logger.getLogger(logname."com.company.ThreadStackTrace");
```

然后 为日志信息指定资源包的关键字 而不是实际的日志消息字符串

```
logger.info("readingFile")
```

通常需要在本地化的消息中添加一些参数 比如占位符 {0}，{1}等

比如 Reading FILE{0}

然后 通过调用下面的一个方法向占位符传递具体的值

```
logger.info(Level.ALL,"readfile",filename);
```

```
private  static  final  Logger
myLogger=Logger.getLogger("com.company.ThreadStackTrace");
```

设置为其他级别

```
Logger logger=new Logger("ceshi");
logger.setLevel(Level.FINE);
```

```
logger.warning(AssertTest);
logger.fine(AssertTest);
```

关于日志的讲解http://blog.csdn.net/PacosonSWJTU/article/details/50272839

这是记录fine类型的日志 修改默认的日志记录级别和处理器级别
我们还可以绕过配置文件 安装自己的处理器

```
Logger logger1=Logger.getLogger("com.company");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler=new ConsoleHandler();
handler.setLevel(Level.FINE);
```

```
logger.addHandler(handler);
```

http://blog.csdn.net/u011794238/article/details/50719775关于日志的记录器的说明

关于处理器

日志记录器将记录发送到ConsoleHandler 并由它输出到System.err流中 特别是 日志处理器还会将记录发送到父处理器中
而最终的处理器(命名为"")有一个ConsoleHandler
处理器也有日志处理级别 对于一个要被记录的日志记录 它的日志记录级别必须高于日志记录器和处理器的阈值 默认日志管理器配置文件设置的默认控制台处理器的日志记录为
java.util.loggging.ConsoleHandler.level =INFO;
如果想要将日志记录发送到其他地方 就需要添加其他的处理器 下面的FileHandler就是很有用的一个
FileHandler可以直接搜集文件中的记录 可以像下面这样直接将记录发送到默认文件的处理器
另外还提供了一个处理器 这就是SocketHandler 这个处理器将记录发送到特定的主机和端口

```
FileHandler fileHandler=new FileHandler();
logger.addHandler(fileHandler);
```

这些记录被发送到用户主目录的javan.log中 n是文件名的唯一编号
如果想不用默认的日志记录文件名的话 是可以修改的
如果有多个应用程序（或者同一个应用程序的多个副本）使用同一个日志文件 就应该开启append标识 另外 应该在文件名模式中使用%u以便每个应用程序创建日志的唯一副本
可以通过设置日志管理器配置文件的不同参数 或者利用其他的构造器来修改文件处理器的默认行为
另外 还可以扩展Handler类或者StramHandler类自定义处理器

http://www.shinater.com/docs/csharp/Logging/FileHandler.html这是关于FileHandler类的具体说明

过滤器是根据日志记录的级别进行过滤 每个日志过滤器和处理器都可以有一个可选的过滤器来完成附加的过滤 另外 还可以通过实现Filter接口并定义下列方法来自定义过滤器
boolean isLoggable(LogRecord record)在这个方法中 可以利用自己喜好的标准 对日志记录进行分析 返回true表示这些记录应该包含在日志中
注意 要想将一个过滤器安装到一个日志记录器或处理器中 只需要调用getFilter方法就可以了 注意 同一时刻 最多只能有一个过滤器

格式化器

ConsoleHandler类和FileHandler类可以生成文本和xml格式的日志记录 格式化器就是进行相关的日志信息的格式化的 也可以自定义格式
需要扩展Formatter类并覆盖下面这个方法
String format(LogRecord record)
可以更具自己想法对记录中的信息进行格式化 并返回字符串 在format方法中 有可能调用下面这个方法
String formatMessage(LogRecord record)
这个方法对记录中的部分信息进行格式化 参数替换和本地化应用操作
很多文件格式需要在已格式化的记录前后加上一个头部和尾部
需要覆盖下面俩个方法
String getHead(Handler h)
String getTail（Handler h）
最后 调用setFormatter方法将格式化器安装到处理器中

关于日志的记录说明

1 为一个简单的应用程序 选择一个日志记录器 并把日志记录命名为与主应用程序包一样的名字

2 默认的日志配置将基本等于或高于INFO级别的所有信息记录到控制台

3 所有级别为INFO,WARNUNG和SERVER的消息都将显示到控制台上 因此最好只将对程序用户有意义的消息设置为这儿的级别 将程序员想要的日志记录 设定为FINE是一个很好的选择

关于调试技巧 有以下几个方面

1
可以使用打印或者记录以下的值
System.out.println("x="+x);
或

Logger.getGlobal().info("x="+x);

2 在要测试的类中放一个main方法 来进行每一个类的单元测试

3junit这个框架

4 日志代理
在调用方法的同时进行输出这个方法的具体日志
5 利用printStackTrace方法 可以从任何一个异常对象中获取堆栈情况

6 可以利用printStackTrace方法将它发送到一个文件中

7 将一个程序的错误信息保存在一个文件中是非常有用的

8 可以调用静态的Thread.setDeaultUncaughtExceptionHandler方法改变非捕获异常的处理器

9 使用-verbose标识启动java虚拟机

10 使用xlint选项告诉编译器对一些普遍容易出错的代码问题进行检查

11 java虚拟机增加了对java应用程序进行监控和管理是支持
Jconsole processID

12 可以使用jmap工具获得一个堆的转储 其中显示了堆了每个对象

13 如果使用-Xprof标识运行java虚拟机 就会运行一个基本的剖析器来跟踪那写代码中经常被调用的方法

调试工具 eclipse中的debug
IDEA中也有这么一个工具