Thinking in java读书笔记

# 对象导论

# 一切都是对象

# 操作符

# 控制流程

# 初始化与清理

初始化和清理正是涉及安全的两个问题。C++中采用构造器和析构函数来实现。Java通过构造器和“垃圾回收”来实现。

## 5.1 构造器的初始化顺序

## 5.2 垃圾回收机制

## 5.3 枚举类的初始化

## 5.4 本章相关概念及问题

# 访问权限控制

# 复用类

# 多态

# 接口

# 内部类

# 持有对象

# 通过异常处理错误

## 12.1 本章相关概念及问题

1. 异常的终极目标：你的应用程序中没有未处理的错误。

2. 异常创建的地点：异常使得我们每件事都当作一个事务来处理。同java创建对象一样，将使用new关键字在堆上创建异常对象。若当前执行路径被终止，从当前环境中弹出对异常对象的引用。

3.异常处理的两种模式：终止和恢复。在终止模式，将假设错误非常关键，以至于程序无法返回到异常发生的地方继续执行。一旦异常被抛出，就表明错误已经无法返回，也不能回来继续执行。在恢复模式，异常处理程序的工作是修正错误，然后重新尝试调用出问题的方法，并认为第二次能成功。

4.创建自定义异常：必须从已有的异常类继承，最好是选择意思相近的异常类继承。建立新的异常类型最简单的方法就是让编译器为你产生默认构造器。

5.通过java.util.logging工具将输出记录到日志中或者通过写入System.err而将错误发送给标准错误流。

6.捕获所有异常：通过捕获异常类的基类Exception可以做到这点。

7.重新抛出异常：通过调用fillInStackTrace()方法，将返回一个Throwable对象，它是通过把当前调用栈信息填入到原来那个异常对象而建立的。

8.异常对象是通过new在堆上创建的，所以，垃圾回收器会自动把它们清理掉。

9.Java标准异常：Throwable对象可以分为两种类型：Error用来表示编译时和系统错误；Exception是可以被抛出的基本类型。

10.异常丢失：异常作为程序出错的标志，决不应该被忽略，但如果return在finally中会抛出异常，但看不到任何输出。

11.构造器中的异常处理：如果在构造器中抛出了异常，这些清理行为可能就不能正常工作了。最安全的使用方法是使用嵌套的try-catch语句。

12.异常匹配：如果把捕获基类的catch子句放在最前面，则派生类的异常将会全部“屏蔽”掉。

13.异常处理的目的：把错误处理的代码和错误发生的地点相分离。异常处理的重要原则是“只有你知道如何处理异常时才捕获异常”。

14.反射和泛型就是用来补偿静态类型检查所带来的过多限制。

15.异常使用指南：

1）在恰当的级别处理问题。（在知道该如何处理的情况下才捕获异常。）

2）解决问题并且重新调用产生异常的方法。

3）进行少许修补，然后绕过异常发生的地方继续执行。

4）用别的数据进行计算，以代替方法预计会返回的值。

5）把当前运行环境下能坐的事情尽量做完，然后把异常抛到更高层。

6）把“被检查异常”转换为“不检查异常”。

# 字符串

## 13.1 String，StringBuffer，StringBuilder

String对象是不可变的。String类中的每一个看起来会修改String值的方法，实际上是创建一个全新的String对象。当需要改变字符串的内容时，String类的方法可以返回一个新的String对象。同时，如果内容没有发生改变。String的方法只是返回指向原对象的引用而已。

String s=“ABC”+”mango”+”def”+47。编译器会自动引入java.lang.StringBuilder类。虽然我们在源码中没有使用StringBuilder类，编译器会自作主张使用了它。StringBuilder是Java SE5引入的，在这之前Java用的是StringBuffer。StringBuffer是线程安全的，因此开销也会更大些。如果你已经知道最终的字符串大概有多长，那预先指定StringBuilder的大小可以避免多次重新分配缓冲。

注意：如果你希望toString()方法打印出对象的内存地址，应该调用Object.toString()方法。使用this.toString()或者super.toString()。

## 13.2 String相关的类

1.格式化输出

1)System.out.printf()

2)System.out.format()：java5中引入的format方法可用于PrintStream或者PrintWriter对象。

3)Formatter类：在java中，所有新的格式化功能都由java.util.Formatter类处理。可以将Formatter看作一个翻译器，它将你的格式化字符串与数据翻译成需要的结果。对其它基本类型的参数，制药该参数不为null，那转换的结果永远都是true。即使是数字0，转换结果依然是true。而这在其他语言中（包括C），往往转换为false。

2.扫描输入

Scanner类：Scanner的构造器可以接受任何类型的输入对象。包括File对象、InputStream对象、String或者Readable 对象。

Scanner有一个假设，在输入结束时会抛出IOException，所以，Scanner会把IOException吞掉。

Scanner的定界符：在默认情况下，Scanner根据空白字符对输入进行分词，但是你可以用正则表达式指定自己所需的定界符。使用useDelimiter()来设定界符。

3.分词-StringTokenizer：在1.4中引入，因为在5中有Scanner和正则表达式，StringTokenizer基本作废。

## 13.3 正则表达式及相关类

# 类型信息

## 14.1 为什么需要类型信息

RTTI（run-time type information）运行时类型信息

## 14.2 Class对象

## 14.3 类型转换前先做检查

## 14.4 注册工厂

## 14.5 instanceof与Class的等价性

## 14.6 反射

1.获取某个类型的确切信息的方法。1）在编译时已知。2）通过反射获取。

2.Class类和java.lang.reflect类库一起对反射的概念进行了支持，该类库包含了Field、Method已经Constructor（构造器）类（每个类都实现了Member接口）。这些类型的对象是由JVM在运行时创建的，用来表示未知类里对应的成员。你可以通过Constructor创建新的对象。用get()和set()方法读取和修改与Field对象相关的字段，用invoke()方法调用与Method对象关联的方法。还可以调用getField(),getMethods(),getConstructors()返回表示字段、方法以及构造器的对象的数组。这样，匿名对象的类信息就能在运行时被完全确定下来，而在编译时不需要知道任何事情。

3.RTTI和反射之间的真正区别：通过反射与一个未知类型的对象打交道时，JVM只是简单地检查这个对象，看它属于哪个特定的类（就像RTTI那样）。在用它做其他事情之前必须先加载那个类的class对象。因此，那个类的.class文件对于JVM来说必须是可获取的：要么在本地机器上，要么可以通过网络取得。真正区别在于：对RTTI来说，编译器在编译时打开和检查.class文件。（换句话说，我们可以通过“普通”方式调用对象的所有方法。）而对于反射机制来说，.class文件在编译时是不可获取的，所以是在运行时打开和检查.class文件。

## 14.7 动态代理

动态代理采用了代理这种设计模式。但java动态代理思想比代理的思想更向前迈进了一步。因为它可以动态地创建代理并动态地处理对所有代理方法的调用。通过调用静态方法Proxy.newProxyInstance()可以创建动态代理，这个方法需要得到一个类加载器（通常从已经被加载的对象中获取其类加载器，如：Interface.class.getClassLoader()），一个你希望该代理实现的接口列表（不是类和抽象类）,以及InvocationHandler接口的一个实现。invoke()方法中传递进来了代理对象，然而，在invoke()内部，在代理上调用方法时需要格外小心，因为对接口的调用将被重定向为对代理的调用。通常，你会执行被代理的操作，然后使用Method.invoke()将请求转发给被代理对象，并传入必须的参数。

## 14.8 空对象

## 14.9 接口与类型信息

通过反射，仍旧可以到达并调用所有方法，甚至是private方法（通过在Method对象上调用setAccessible(true)）。但是final域实际上在遭遇修改时时安全的。运行时系统会在不抛出异常的情况下接受任何修改尝试，但是实际上不会发生任何修改。

# 泛型

## 15.1 泛型的概念

泛型实现了参数化类型的概念，在你创建参数化类型的一个实例时，编译器会为你负责转型操作并保证类型的正确性。我们只需要指定它们的名称以及类型参数列表即可。

## 15.2 泛型类

元组：它是将一组对象直接打包存储于其中的一个单一对象。这个容器对象允许读取其中元素，但是不允许向其中存放新的对象。通常，元组可以具有任意长度，同时，元组中的对象可以是任意不同类型。

## 15.3 泛型接口

一般而言，一个生成器只定义一个方法，该方法用以产生新的对象。Java泛型的一个局限性：基本类型无法作为类型参数。

## 15.4 泛型方法

类中包含参数化方法，而这个方法所在的类可以是泛型类，也可以不是泛型类。无论何时，只要你能做到，你就应该尽量使用泛型方法。对于一个static的方法而言，无法访问泛型类的类型参数，所以，如果static方法需要使用泛型能力，就必须成为泛型方法。要定义泛型方法，只需要将泛型参数列表置于返回值之前。如果调用f()时传入基本类型，自动打包机制就会介入其中，将基本类型的值包装为对应的对象。在泛型方法中，可以显式地指明类型，不过这种语法很少使用。要显式地指明类型，必须在点操作符与方法名之间插入尖括号，然后把类型置于尖括号内。

## 15.5 擦除

在Java1.0中就已经使用了泛型，但是它不是使用擦除来实现，而是使用具体化。为了兼容之前的代码，引入了擦除的概念。在泛型代码内部，无法获取任何有关泛型参数的类型信息。通过Class.getTypeParameters()将“返回一个TypeVariable对象数组，表示有泛型声明所声明的类型参数”。但你能发现的只有参数占位符的标识符。Java泛型是使用擦除来实现的，这意味着你在使用泛型时，任何具体的类型信息都被擦除了，你唯一知道的就是你在使用一个对象。因此List<String>和List<Integer>在运行时事实上是相同的类型。这两种形式都被擦除为“原生”类型，即List。泛型类型只有在静态类型检查期间才出现，在此之后，程序中的所有泛型类型都将被擦除，替换为它们的非泛型上界。如List<T>这样的类型注解将被擦除为List，而普通的类型变量在未指定边界的情况下将被擦除为Object。当你在编写泛型代码时，必须时刻提醒自己，你只是看起来好像拥有有关参数的类型信息而已。因为擦除在方法体中移除了类型信息，所以在运动时的问题就是边界：即对象进入和离开方法的地点。

## 15.6 擦除的补偿

因为泛型擦除了丢失在泛型代码中执行某些操作的能力，但是编译器还是能识别这个类，只是某些操作需要知道泛型所真正代表的类。必须通过引入类型标签来对擦除进行补偿。这意味着你需要显式地传递你的类型的Class对象，以便你可以在类型表达式中使用它。Java中有两种解决办法：（1）传递一个工厂对象，并使用它来创建新的实例。最便利的工厂对象就是Class对象。（2）使用模版方法设计模式。

# 数组

## 16.1 Arrays实用功能

它有一套用于数组的static实用方法：equals()用于比较两个数组内容是否相等（deepEquals()用于多维数组）;fill();sort();binarySearch();toString();hashCode();asList()。

1. System.arraycopy()用它复制数组比用for循环要快很多。并针对很多类型做了重载。如果复制对象数组，那么只是复制了对象的引用-而不是对象本身的拷贝（浅复制）。
2. 数组元素的比较：1）实现java.lang.Comparable接口。重写compareTo()方法。此方法接收另一个Object为参数，如果当前对象小于参数则返回负值，如果相等则返回零，如果当前对象大于参数则返回正值。2）Collections.reverseOrder()。

## 16.2 本章相关概念及问题

1. 应该优先使用容器而不是数组。

2.数组和其他种类的容器之间的区别：效率、类型和保存基本类型的能力。在java中，数组是一种效率最高的存储和随机访问对象引用序列的方式。数组是一个简单的线性序列，这使得元素访问非常快速。但是为这种速度所付出的代价是数组对象的大小被固定，并且在其生命周期内不可改变。数组之所以优于泛型之前的容器，就是因为你可以创建一个数组去持有某种具体类型。这意味着你可以通过编译期检查，来防止插入错误类型和抽取不当类型。

3.对象数组和基本类型数组在使用上几乎是相同的；唯一的区别是对象数组保存的是引用，基本类型数组保存基本类型的值。

4.length是数组的大小，而不是实际保存的元素个数。

5.如果需要返回一组值，可以使用数组。

6.数组和泛型：class classParameter<T>{}是参数化类；class MethodParameter{public static <T> T[] f(T[] arg){}}是参数化方法。参数化方法常常是首选。

# 容器深入研究

# Java I/O系统

存在各种I/O源端和想要与之通信的接收端（文件、控制台、网络链接），而且有不同的方式进行通信（顺序、随机存储、缓存、二进制、按字符、按行、按字）。

## 18.1 File 类

File既能代表一个特定文件的名称，又能代表一个目录下的一组文件的名称。包括目录的检查和创建，删除等操作。

## 18.2 输入和输出类

1.我们通常通过叠合多个对象来提供所期望的功能（这是装饰器设计模式）。

2.InputStream类型：InputStream是用来表示那些从不同数据源产生输入的类。这些数据源包括：1）字节数组（ByteArrayInputStream）2）String对象（StringBufferInputStream）3）文件（FileInputStream）4）“管道”（PipedInputStream）5）一个由其他种类的流组成的序列（SequenceInputStream）6）其他数据源（如：网络）。

3.OutputStream类型：该类别的类决定了输出所要去往的目标。字节数组，文件或者管道。1）ByteArrayOutputSteam。2）FileOutputStream。3）PipedOutputStream。4）FilterOutputStream。

4.FilterInputStream和FilterOutputStream是用来提供装饰器类接口以控制特定输入流（InputStream）和输出流（OutputStream）的两个类。属于FilterInputStream类的有：DataInputStream，BufferedInputStream，LineNumberInputStream，PushbackInputStream。属于FilterOutPutStream类的有：DataOutputStream，PrintStream，BufferedOutputStream（常用）。

## 18.3 Reader和Writer

1.Reader和Writer则提供兼容Unicode与面向字符的功能。

2.有时我们必须把来自于“字节”层次结构的类和“字符”层次结构的类结合起来使用。为了这个目的，需要使用适配器类：InputStreamReader可以把InputStream转换为Reader，而OutputStream可以把OutputStream转换为Writer。

3.几乎所有原始的Java I/O流类都有相应的Reader和Writer类来提供天然的Unicode操作。常用的做法是：尽量尝试使用Reader和Writer，一旦程序代码无法成功编译，我们就会发现自己不得不使用面向字节的类库。

4.注：java.util.zip类库就是面向字节的而不是面向字符的。

5.自我独立的类：RandomAccessFile。RandomAccessFile的工作方式类似于把DataInputStream和DataOutputStream组合起来使用。只有RandomAccessFile支持搜索方法，并且只适用于文件。在JDK1.4中，RandomAccessFile的大多数功能（但不是全部）由nio存储映射文件所取代。

## 18.4 I/O流的典型使用方式

1.输入方式

1）缓存输入文件。如果想要打开一个文件用于字符输入，可以使用以String或File对象作为文件名的FileInputReader。为了提高速度，采用缓存，将所产生的引用传给一个BufferedReader构造器。

2）从内存输入。StringReader in = new StringReader(BufferedInputFile.read(“aa.java”));

3）格式化的内存输入。要格式化数据，可以使用DataInputStream，它是一个面向字节的I/O类，因此，我们必须使用InputStream类而不是Reader类。

2.输出方式

1）基本的文件输出：FileWriter对象可以向文件写入数据。通过创建一个与指定文件连接的FileWriter，实际上，我们通常会用BufferedWriter将其包装起来用以缓存输出。

## 18.5 标准I/O

标准I/O:程序的所有输入都可以来自于标准输入，它的所有输出也都可以发送到标准输出，以及所有的错误信息都可以发送到标准错误。

Java提供了System.in，System.out，System.err。其中System.out，System.err事先已经包装成了printStream对象。而System.in却是一个没有被包装过的未加工的InputStream。

Java提供了简单的静态方法，允许我们对标准输入、输出和错误I/O流进行重定向。如果显示器上创建大量的输出而无法阅读时，重定向写入文件是极为有用的。I/O重定向操作的是字节流，而不是字符流。因此，我们使用的是InputStream和OutputStream，而不是Reader和Writer。

## 18.6 进程控制

进程控制可以将标准输入附接到文件上，并将标准输出和标准错误重定向到另外一个文件。注意，它在程序开头处存储了对最初的System.out对象的引用，并且在结尾处将系统输出恢复到了该对象上。

Process process = new ProcessBuilder(command.split(“”)).start()。

process.getInputStream().

## 18.7 NIO

1.NIO在速度提高上有很大的优势，速度的提高在文件I/O和网络I/O中都有可能发生。速度的提高来自于所使用的结构更接近于操作系统执行I/O的方式：通道和缓冲器。然而，我们并没有直接和通道交互，我们只是和缓冲器交互，并把缓冲器派送到通道。唯一直接和通道交互的缓冲器是ByteBuffer，它可以存储未加工字节的缓冲器。

2.旧I/O类中修改了FileInputStream，FileOutputStream以及用于既读又写的RandomAccessFile三个类，用以产生FileChannel。Reader和Writer这种字符模式类不能用于产生通道；但是java.nio.channels.Channels类提供了实用方法，用以在通道中产生Reader和Writer。

3.对于某些流类，getChannel()将会产生一个FileChannel()。可以向通道传送用于读写的ByteBuffer，并且可以锁定文件的某些区域用于独占式访问。

4.下面是通道的相关方法。特殊方法有：transferTo()和transferFrom()则允许我们将一个通道和另外一个通道直接相连。

package com.swad.util;  
  
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.RandomAccessFile;  
import java.nio.ByteBuffer;  
import java.nio.channels.FileChannel;  
  
*/\*\*  
 \* Created by gonghaiyu on 07/05/2017.  
 \*/*public class GetChannel {  
 private static final int *BSIZE* = 1024;  
 public static void main(String [] args) throws Exception{  
 //write a file  
 FileChannel fc = new FileOutputStream("data.txt").getChannel();  
 fc.write(ByteBuffer.*wrap*("Some text ".getBytes()));  
 fc.close();  
 //在文本的结尾处加入内容  
 fc = new RandomAccessFile("data.txt","rw").getChannel();  
 fc.position(fc.size());//移动到文档结尾处  
 fc.write(ByteBuffer.*wrap*("Some more".getBytes()));  
 fc.close();  
 //read the file  
 fc = new FileInputStream("data.txt").getChannel();  
 //分配ByteBuffer的大小  
 ByteBuffer buffer = ByteBuffer.*allocate*(*BSIZE*);  
  
 //从通道中读取，往缓存中存储字节。  
 fc.read(buffer);  
 //让缓冲器做好读的准备  
 buffer.flip();  
 while (buffer.hasRemaining())  
 System.*out*.print((char)buffer.get());  
 }  
}

5.缓冲器容纳的是普通的字节，为了把它们转化成字符，我们要么在输入它们的时候对其进行编码，要么在将其从缓冲器输出时对它们解码。可以使用java.nio.charset.Charset类实现这些功能。尽管ByteBuffer只能保存字节类型的数据，但是它具有可以从其所容纳的字节中产生出各种不同基本类型值的方法。

6.字节存放次序：不同的机器可能会使用不同的字节排序方法来存储数据。“big endian”（高位优先）将最重要的字节存放在地址最低的存储器单元。而“little endian”（低位优先）则是将最重要的字节放在地址最高的存储器单元。ByteBuffer是以高位优先的形式存储数据的，并且数据在网上传送时也常常使用高位优先的形式。

7.内存映射文件：内存映射文件允许我们创建和修改那些因为太大而不能放入内存的文件。为了既能写又能读，我们先由RandomAccessFile开始，获取该文件上的通道，然后调用map()产生MappedByteBuffer，这是一种特殊类型的直接缓冲器。注意：我们必须指定映射文件的初始位置和映射区域的长度，这意味着我们可以映射某个大文件的较小部分。映射文件中的所有输出必须使用RandomAccessFile。

package com.swad.util;  
  
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.RandomAccessFile;  
import java.nio.ByteBuffer;  
import java.nio.channels.FileChannel;  
  
*/\*\*  
 \* Created by gonghaiyu on 07/05/2017.  
 \*/*public class GetChannel {  
 private static final int *BSIZE* = 1024;  
 public static void main(String [] args) throws Exception{  
 //write a file  
 FileChannel fc = new FileOutputStream("data.txt").getChannel();  
 fc.write(ByteBuffer.*wrap*("Some text ".getBytes()));  
 fc.close();  
 //在文本的结尾处加入内容  
 fc = new RandomAccessFile("data.txt","rw").getChannel();  
 fc.position(fc.size());//移动到文档结尾处  
 fc.write(ByteBuffer.*wrap*("Some more".getBytes()));  
 fc.close();  
 //read the file  
 fc = new FileInputStream("data.txt").getChannel();  
 //分配ByteBuffer的大小  
 ByteBuffer buffer = ByteBuffer.*allocate*(*BSIZE*);  
  
 //从通道中读取，往缓存中存储字节。  
 fc.read(buffer);  
 //让缓冲器做好读的准备  
 buffer.flip();  
 while (buffer.hasRemaining())  
 System.*out*.print((char)buffer.get());  
 }  
}

8.文件加锁

JDK1.4中引入了文件加锁机制，允许我们同步访问某个作为共享资源的文件。不过，竞争同一文件的两个线程可能在不同的Java虚拟机上，或者一个是Java线程，另一个是操作系统中其他的某个本地线程。文件锁对其他的操作系统进行是可见的，因为java文件加锁机制映射到了本地操作系统的加锁工具。

通过对FileChannel调用tryLock()或lock()。就可以获取整个文件的FileLock。tryLock()是非阻塞式的，它设法获取锁，但是如果不能获取（当其他一些进程已经持有相同的锁，并且不共享时），它将直接从方法调用返回。而lock()则是阻塞式的，它要阻塞进行直到锁可以获得，或调用lock()的线程中断，或调用lock()的通道关闭。使用FileLock.release()可以释放锁。

也可以使用如下方法对文件的一部分上锁：

tryLock(long position, long size,Boolean shared)

或者

lock(long position,long size,Boolean shared)

其中，加锁的区域由size-position决定。第三个参数指定是否是共享锁。数据库就是这样，因此多个用户可以同时访问到它。

对独占锁或者共享锁的支持必须由底层的操作系统提供。如果操作系统不支持共享锁并为每个请求都创建一个锁，那么它就会使用独占锁。锁的类型（共享或独占）可以通过FileLock.isShared()进行查询。

## 18.8 压缩

1.压缩类库是按字节方式而不是字符方式处理的。尽管存在有许多种压缩算法，但是Zip和GZIP可能是最常用的。压缩的基类：DeflaterOutputStream。解压缩的基类：InflaterInputStream。Zip压缩显示了用Checksum类来计算和校验文件的校验和的方法。有两种Checksum类型：Adler32（它快一些）和CRC32（慢一些，但更准确）。对于多个文件压缩，都必须调用putNextEntry()，并将其传递给一个ZipEntry对象。尽管Zip格式提供了设置密码的方法，但Java的Zip类库并不提供这方面的支持。虽然CheckedInputStream和CheckedOutputStream都支持Adler32和CRC32两种类型的校验和，但是ZipEntry类只有一个支持CRC的接口。

## 18.9 对象序列化

1.对象序列化：Java对象序列化将那些实现了Serializable接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。因这过程可以通过网络，序列化机制能自动弥补不同操作系统之间的差异。

2.transient（瞬时关键字）：当我们对序列化进行控制时，可能某个特定子对象不想让Java的序列化机制自动保存与恢复。如果子对象是我们不希望将其序列化的敏感信息（如密码），可以用transient（瞬时）关键字逐个字段地关闭序列化。由于Externalizable对象在默认情况下不保存它们的任何字段，所以transient关键字只能和Serializable对象一起使用。对象序列化只是java的解决方案，一种更具互操作性的解决方案是将数据转化为XML格式。

# 枚举类型

# 注解

# 并发

# 图形化用户界面