学生应知信息知识

Java知识手册（十一）

狄登峰 主编

目　　录

Java 中的抽象数据类型探讨

在本文中，我们将考察 Java 中的数据类型，同时将介绍抽象数据类型（ADT）的概念。我们还将通过介绍 Java Collections Framework（Java 集合架构）来学习 Java 定义的一些 ADT。

ADT

一个 ADT 是一个仅由保存的数据类型和可能在这个数据类型上进行的操作定义的。开发者们只能通过 ADT 的操作方法来访问 ADT 的属性，而且他们不会知道这个数据类型内部各种操作是如何实现的。

在 Java 中，我们常常使用一个接口来给出一个操作集合而不需要透露这些操作实现的细节。记住一个接口定义了一个方法集而 Java 类必须实现这个集合以便满足它的强制性条件或者实现这个接口的一个实例。

线性表，堆栈和队列

当我们谈论 ADT 的时候，经常会说到线性表，堆栈和队列。我们不会讨论这些数据结构的细节，但我们会讨论为什么它们被称为 ADT。

一个线性表是有限个元素的集合，其元素以线性的方式进行排列并提供对它的元素的直接访问。一个堆栈是一个后进先出（LIFO）的有序线性表，元素从堆栈头

加入，并从堆栈头取出。一个队列是一个先进先出的有序线性表，元素从队列尾加入，并从队列头取出。

线性表，堆栈和队列的内部结构可以用许多方式实现。例如，我们可以使用一个有序数组或者一个链表来实现每个结构。关键的一点是不论你如何实现其内部结构，它对外的接口总是不变的。这使得你能够修改或者升级底层的实现过程而不需要改变公共接口部分。

Java 集合架构

Java 2 软件开发包（SDK）提供了一些新类来支持大多数常用的 ADT。这些类被称为 Java 集合类（类似于 MFC 中的集合类），它们协同工作从而形成 Java 集合架构。这个集合架构提供了一套将数据表示成所谓的集合抽象数据的接口和类。

java.util.Collection 接口被用来表示任意的成组的对象，也就是元素。这个接口提供基本的诸如添加，删除，和查询这样的操作。Collection 接口还提供了一个 iterator 方法。iterator 方法返回 java.util.Iterator 接口的一个实例。而 Iterator 接口又提供了 hasNext, next, 和 remove 方法。使用 Iterator 接口提供的方法，你可以从头到尾循环遍历一个 Collection 对象中的实例并能够安全的删除 iterator(游标)所表示的元素。

java.util.AbstractCollection 是所有集合架构类的基础。AbstractCollection 类提供了对 java.util.Collection 接口中除 iterator 和 size 方法以外的所有方法的实现。

这两个例外的方法由所有继承 java.util.AbstractCollection 的子类实现。

实现一个接口的类必须提供对所有接口方法的实现。因为集合架构中的一些接口方法是可选的，所以必须有一种方法来通知调用者某种方法没有实现。当一个可选的方法被实现而这个方法又并没有被实现的时候，就会抛出一个 UnsupportedOperationException 异常。UnsupportedOperationException 类继承了 RuntimeException 类。这使得调用者能够调用所有的集合操作而不需要把每次调用都放在一个 try-catch 对里。

List 线性表

List 接口继承了 Collection 接口并定义了一个允许相同元素存在的有序集合。List 接口还附加了一些使用一个数值型索引值并基于元素在线性表中的位置来操作 Collection 中元素的方法。这些操作包括 add,get,set 和 remove。

List 接口还提供了 listIterator 方法。这个方法返回 java.util.ListIterator 接口的一个实例，这个实例能够让你从头至尾或者从尾至头的遍历一个线性表。java.util.ListIterator 继承了 java.util.Iterator 接口。因此，它支持对它代表的 Collection 中的元素的添加和修改。

下面的例子演示了如何从后向前遍历一个列表的元素。要完成这个工作，必须在遍历开始之前把 ListIterator 定位于列表最后一个元素之后。

```
ListIterator iter = aList.listIterator(aList.size());
while (iter.hasPrevious())
System.out.println(iter.previous().toString());
}
```

集合架构提供了对 List **接口的两个实现：**LinkedList **（链表）和** ArrayList**（数组列表，即静态列表）。这两个实现都支持对其元素的随机访问。一个** ArrayList **实例支持数组风格的操作并支持数组大小的改变操作。一个** LinkedList **的实例则提供了在列表开始和结尾添加，删除和提供元素的显式的支持。使用这些新方法，一个程序员可以简单的把一个** LinedList **当做堆栈或者队列使用，如下：**

```
LinkedList aQueue = new LinkedList(aCollection);
aQueue.addFirst(newElement);
Object anElement = aQueue.removeLast();
LinkedList aStack = new LinkedList(aCollection);
aStack.addFirst(newElement);
Object anElement= aStack.removeFirst();
```

表 A 中的代码片段使用 java.util.ArrayList 和 java.util.LinkedList 演示了对 java.util.List 接口的实现实例的一些常用的操作。这些操作包括添加元素，随机访问元素和显式的在列表尾删除元素。

知其然不知其所以然是大有好处的

ADT 提供了一个将对象公共接口中的操作和其具体的实现分开的强有力的工具。这使得一个 ADT 的实现可以不断变化和演化同时保持其公共接口不变。Java 集合架构提供了大量的接口和其实现用来代表基本元素的集合并可以用来创建有用的 ADT。

用 Java 实现 HTTP 断点续传

(一)断点续传的原理

其实断点续传的原理很简单，就是在 Http 的请求上和一般的下载有所不同而已。打个比方，浏览器请求服务器上的一个文时，所发出的请求如下：

假设服务器域名为 wwww.sjtu.edu.cn，文件名为 down.zip。

GET /down.zip HTTP/1.1

Accept: image/gif, image/x-xbitmap, image/jpeg,

image/pjpeg, application/vnd.ms-

excel, application/msword, application/vnd.ms-powerpoint, */*

Accept-Language: zh-cn

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT 5.0)

Connection: Keep-Alive

服务器收到请求后，按要求寻找请求的文件，提取文件的信息，然后返回给浏览器，返回信息如下：

200

Content-Length=106786028

Accept-Ranges=bytes

Date=Mon, 30 Apr 2001 12:56:11 GMT

ETag=W/"02ca57e173c11:95b"

Content-Type=application/octet-stream

Server=Microsoft-IIS/5.0

Last-Modified=Mon, 30 Apr 2001 12:56:11 GMT

所谓断点续传，也就是要从文件已经下载的地方开始继续下载。所以在客户端浏览器传给

Web **服务器的时候要多加一条信息--从哪里开始。**

下面是用自己编的一个"浏览器"来传递请求信息给 Web **服务器，要求从** 2000070 **字节开始。**

GET /down.zip HTTP/1.0

User-Agent: NetFox

RANGE: bytes=2000070-

Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2

仔细看一下就会发现多了一行 RANGE: bytes=2000070- **这一行的意思就是告诉服务器** down.zip **这个文件从** 2000070 **字节开始传,前面的字节不用传了。**

服务器收到这个请求以后，返回的信息如下：

206

Content-Length=106786028

Content-Range=bytes 2000070-106786027/106786028

Date=Mon, 30 Apr 2001 12:55:20 GMT

ETag=W/"02ca57e173c11:95b"

Content-Type=application/octet-stream

Server=Microsoft-IIS/5.0

Last-Modified=Mon, 30 Apr 2001 12:55:20 GMT

和前面服务器返回的信息比较一下，就会发现增加了一行：

Content-Range=bytes 2000070-106786027/106786028

返回的代码也改为 206 **了，而不再是** 200 **了。**

知道了以上原理，就可以进行断点续传的编程了。

(**二**)Java **实现断点续传的关键几点**

(1)**用什么方法实现提交** RANGE: bytes=2000070-**。**

当然用最原始的 Socket 是肯定能完成的，不过那样太费事了，其实 Java 的 net 包中提供了这种功能。代码如下：

```
URL url = new URL(" http://www.sjtu.edu.cn/down.zip";;);

HttpURLConnection httpConnection = (HttpURLConnection)url.openConnection

();

//设置 User-Agent

httpConnection.setRequestProperty("User-Agent","NetFox");

//设置断点续传的开始位置

httpConnection.setRequestProperty("RANGE","bytes=2000070");

//获得输入流

InputStream input = httpConnection.getInputStream();
```

从输入流中取出的字节流就是 down.zip 文件从 2000070 开始的字节流。大家看，其实断点续传用 Java 实现起来还是很简单的吧。接下来要做的事就是怎么保存获得的流到文件中去了。

保存文件采用的方法

我采用的是 IO **包中的** RandAccessFile **类。**

操作相当简单，假设从 2000070 **处开始保存文件，代码如下：**

```
RandomAccess oSavedFile = new RandomAccessFile("down.zip","rw");

long nPos = 2000070;

//定位文件指针到 nPos 位置

oSavedFile.seek(nPos);

byte[] b = new byte[1024];

int nRead;

//从输入流中读入字节流，然后写到文件中

while((nRead=input.read(b,0,1024)) > 0)

{

   oSavedFile.write(b,0,nRead);

}
```

接下来要做的就是整合成一个完整的程序了。包括一系列的线程控制等等。

(三)断点续传内核的实现

主要用了 6 个类，包括一个测试类：

SiteFileFetch.java 负责整个文件的抓取，控制内部线程(FileSplitterFetch 类)。

FileSplitterFetch.java 负责部分文件的抓取。

FileAccess.java 负责文件的存储。

SiteInfoBean.java 要抓取的文件的信息，如文件保存的目录，名字，抓取文件的 URL 等。

Utility.java 工具类，放一些简单的方法。

TestMethod.java 测试类。

下面是源程序：

```
/*
**SiteFileFetch.java
*/
package NetFox;
import java.io.*;
import java.net.*;
```

```
public class SiteFileFetch extends Thread {
SiteInfoBean siteInfoBean = null; //文件信息 Bean
long[] nStartPos; //开始位置
long[] nEndPos; //结束位置
FileSplitterFetch[] fileSplitterFetch; //子线程对象
long nFileLength; //文件长度
boolean bFirst = true; //是否第一次取文件
boolean bStop = false; //停止标志
File tmpFile; //文件下载的临时信息
DataOutputStream output; //输出到文件的输出流
public SiteFileFetch(SiteInfoBean bean) throws IOException
{
   siteInfoBean = bean;
        //tmpFile = File.createTempFile ("zhong","11
File(bean.getSFilePath()));
        tmpFile = new File(bean.getSFilePath()+File.separa
bean.getSFileName()+".info");
```

```
    if(tmpFile.exists ())
    {
bFirst = false;
read_nPos();
    }
    else
    {
nStartPos = new long[bean.getNSplitter()];
nEndPos = new long[bean.getNSplitter()];
    }
}
public void run()
{
    //获得文件长度
    //分割文件
    //实例 FileSplitterFetch
```

```
    //启动 FileSplitterFetch 线程
    //等待子线程返回
    try{
if(bFirst)
{
    nFileLength = getFileSize();
    if(nFileLength == -1)
    {
System.err.println("File Length is not known!");
    }
    else if(nFileLength == -2)
    {
System.err.println("File is not access!");
    }
else
{
```

```
    for(int i=0;i<nStartPos.length;i++)
    {
nStartPos[i] = (long)(i*(nFileLength/nStartPos.length));
    }
    for(int i=0;i<nEndPos.length-1;i++)
    {
nEndPos[i] = nStartPos[i+1];
    }
    nEndPos[nEndPos.length-1] = nFileLength;
}
    }
    //启动子线程
    fileSplitterFetch = new FileSplitterFetch[nStartPos.length];
    for(int i=0;i<nStartPos.length;i++)
    {
fileSplitterFetch[i] = new FileSplitterFetch(siteInfoBean.getSSiteURL(
siteInfoBean.getSFilePath() + File.separator + siteInfoBean.getSFile
```