章节51-章节60

章节51 课时 198 04054_打印流（问题引出）

问题引出
现在已经清楚了InputStream和OutputStream两个类的基本作用，但是此时有一个小小的问题，如果要进行输出只能使用OutputStream类完成，但是OutputStream类在输出上是否真的方便？
如果要使用OutputStream输出数据，假设要输出的是String，需要将String变为字节数组后再输出，那么如果是int，那么需要将int变为字符串而后再变成字节数组输出，同样，如果是boolean呢？或者说是double呢？

```
package cn.mldn.demo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
class PrintUtil{
    private OutputStream output;
    public PrintUtil(OutputStream output){
        this.output=output;
    }
    public void print(int x){
        this.print(String.valueOf(x));
    }
    public void print(String x){
        try {
            this.output.write(x.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void print(double x){
        this.print(String.valueOf(x));
    }
    public void println(int x){
        this.println(String.valueOf(x));
    }
    public void println(String x){
        this.print(x.concat("\n"));
    }
    public void println(double x){
        this.println(String.valueOf(x));
    }
    public void close(){
        try {
            this.output.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
public class TestDemo{
    public static void main(String args[])throws Exception{
        PrintUtil pu=new PrintUtil(new FileOutputStream(new File("d:"+File.separator+"my.txt")));
        pu.print("Hello");
        pu.println("World");
        pu.println(1+1);
        pu.println(1.1+1.1);

    }
}
```

章节51 课时 199 04055_打印流（打印流）

为了解决输出数据时的功能不足，所以在java.io包里面又提供了一套专门的用于输出数据的类；PrintStream(打印字节流)、PrintWriter(打印字符流)。
以PrintStream类为例，观察一下这个类的继承结构和构造方法。
java.lang.Object
java.io.OutputStream
java.io.FilterOutputStream
java.io.PrintStream

PrintStream是一个装饰类，为了完善OutputStream类的功能。

```
package cn.mldn.demo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
public class TestDemo{
    public static void main(String args[])throws Exception{
        PrintStream pu=new PrintStream(new FileOutputStream(new File("d:"+File.separator+"my.txt")));
        pu.print("Hello");
        pu.println("World");
        pu.println(1+1);
        pu.println(1.1+1.1);

    }
}
```

章节51 课时 200 04056_打印流（JDK 1.5支持）

JDK1.5的改进
所有的输出数据都要求使用打印流完成，但是在JDK1.5之后考虑到市场因素，所以增加了一种新的输出，称为格式化输出。

```
public PrintStream printf(String format,
                          Object... args)
```

如果要格式化输出就需要一些标记：整数（%d）、字符串（%s）、小数（%m.nf）、字符（%c）描述。
范例格式化输出：

```
package cn.mldn.demo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;
public class TestDemo{
    public static void main(String args[])throws Exception{
        String name="陈冠";
        int age=18;
        double score=59.43234123412341;
        PrintStream pu=new PrintStream(new FileOutputStream(new File("d:"+File.separator+"my.txt")));
        pu.printf("姓名:%s,年龄:%d,成绩:%5.2f", name,age,score);
        pu.close();
    }
}
```

在开发之中，几乎不会使用到此类输出，此类的输出只是作为使用特色而已。

在JDK1.5之后，String类也发生了一些变化，定义了一个新的功能：格式化字符串；
格式化字符串的方法：

```
public static String format(String format,
                            Object... args)
```

范例：

```
package cn.mldn.demo;
public class TestDemo{
    public static void main(String args[])throws Exception{
        String name="陈冠";
        int age=18;
        double score=59.43234123412341;
        String str=String.format("姓名:%s,年龄:%d,成绩:%5.2f", name,age,score);
        System.out.println(str);
    }
}
```

String类的所有核心功能就算讲解完成了。

总结：
如果在日后进行程序输出数据操作的话，绝对要使用打印流完成功能。

章节52 课时 201 04057_System类对IO的支持（输出）

学习完PrintStream（PrintWriter）类之后会发现里面的方法很熟悉
了解一下System类之中对于IO操作的支持。
具体内容（了解）
在System类里面为了支持IO操作专门提供有三个常量：
错误输出：public static final PrintStream err
输出到标准输出设备（显示器）public static final PrintStream out
从标准输入设备输入（键盘）：public static final InputStream in

错误输出：
System.err是PrintStream类对象，此对象专门负责进行错误信息的输出操作。
严格来讲System.err和System.out的功能是完全一样的，之所以这样设计，主要的目的是希望err输出不让用户看见的错误，而out输出的是可以让用户看见的信息，但是现在基本上没人再去分了。

信息输出：System.out
System.out是在Java之中专门准备的支持屏幕输出信息的操作对象（此对象由系统负责实例化），下面就可以利用System.out来实现一个简单的输出操作（有些啰嗦）。
范例：利用OutputStream实现屏幕输出。
```
package cn.mldn.demo;
import java.io.OutputStream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  OutputStream out=System.out;//现在OutputStream就变成了屏幕输出
  out.write("helloworld".getBytes());//屏幕输出
 }
}
```
另外，要结合JDK1.8的函数式的功能接口，它可以为消费型函数式接口做方法引用。
```
package cn.mldn.demo;
import java.io.OutputStream;
import java.util.function.Consumer;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Consumer<String> con=System.out::println;
  con.accept("Hello World!");
 }
}
```
在java系统里面，System.out是默认提供好的实例化对象，不再需要用户进行明确的实例化操作。

章节52 课时 202 04058_System类对IO的支持（输入）

系统输入：System.in
在任何语言里面都有一种功能：键盘输入的操作。但是Java本身并没有提供，但是在System里面有一个in的对象。此对象的类型是InputStream。
范例：实现键盘的数据输入
```
package cn.mldn.demo;
import java.io.InputStream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  InputStream input=System.in;
  byte data[]=new byte[1024];
  System.out.println("请输入数据：");
  int len=input.read(data);
  System.out.println("输入数据为："+new String(data,0,len));

 }
}
```
与之前最大的不同只是更换了一个实例化对象。现在已经实现了键盘数据的输入操作，但是有一个问题出现了，此时输入的时候开辟了一个数组，那么如果说数组的容量小于输入的长度呢？

那么现在会发现一个非常严重的问题，超过数组长度的数据将不会被保存。实际上，如果在开发之中，你永远也无法指定用户可能输入的数据是多少，所以这种设置输入长度的操作不可能使用。
那么干脆就别设置长度了
范例：
```
package cn.mldn.demo;
import java.io.InputStream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  InputStream input=System.in;
  StringBuffer buf=new StringBuffer();
  System.out.println("请输入数据：");
  int temp=0;
  while((temp=input.read())!=-1){
   if(temp=='\n'){
    break;
   }
   buf.append((char)temp);
  }
  System.out.println("输入数据为："+buf);

 }
}
```
但是中文输出会出错，因为一个汉字2个字节
System类中对于IO的各种操作实际上并不会过多的使用到，因为唯一可以使用到的只是System.out.println();的方法，但是对于此部分你至少应该清楚一点：System.out使用的是PrintStream类对象进行的标准输出设备显示。

章节53 课时 203 04059_缓冲输入流

缓冲输入流是在开发之中，也会被经常大量使用到的工具类，其目的是解决数据的乱码问题。
现在最直观的解决方式就是System.in所带来的问题。
如果要进行中文数据的处理首先想到的一定是字符流，并且要想完整的处理数据，那么一定需要缓冲区，可以对于缓冲区的操作有两种流：
 字符缓冲区流：BufferedReader、BufferedWriter
 字节缓冲区流：BufferedInputStream、BufferedOutputStream
在给出的缓冲区数据输入流上有两个，其中最为重要的就是BufferedReader，因为在BufferedReader类里面提供有一个重要的读取方法：
public String readLine()throws IOException，读取一行数据，以分隔符（换行）为界。最重要的是返回的是String类型。

下面继续来认真的观察一下BufferedReader类的继承结构以及构造方法。
```
java.lang.Object
 java.io.Reader
  java.io.BufferedReader
```
public BufferedReader(Reader in)
但是如果要想使用BufferedReader类来处理System.in的操作就比较麻烦了，因为System.in是InputStream类型。之前学习过一个类InputStreamReader
范例：键盘输入数据的标准格式
```
package cn.mldn.demo;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class TestDemo{
 public static void main(String args[])throws Exception{
  //System.in是InputStream类的对象
  //BufferedReader的构造方法里面需要接收Reader类对象
  //利用InputStreamReader将字节流变为字符流
  BufferedReader buf=new BufferedReader(new InputStreamReader(System.in));
  System.out.print("请输入数据：");
  String str=buf.readLine();
  System.out.println("请输出数据："+str);
 }
}
```

此时输入数据没有长度限制，并且得到的还是String型数据，那么这样就可以实现键盘输入数据的操作了，只不过这种操作一般意义不大。
一直强调使用BufferedReader是因为她可以实现字符串数据的接收，所以现在可以基于正则进行判断。
范例：判断输入内容

```
package cn.mldn.demo;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class TestDemo{
 public static void main(String args[])throws Exception{
  //System.in是InputStream类的对象
  //BufferedReader的构造方法里面需要接收Reader类对象
  //利用InputStreamReader将字节流变为字符流
  BufferedReader buf=new BufferedReader(new InputStreamReader(System.in));
  boolean flag=true;
  while(flag){
   System.out.print("请输入年龄：");
   String str=buf.readLine();//以回车作为换行
   if(str.matches("\\d{1,3}")){//输入数据由数字组成
    System.out.println("年龄是："+Integer.parseInt(str));
    flag=false;
   }else{
    System.out.println("年龄输入错误，应该由数字所组成");
   }
  }
 }
}
```

正是因为此处可以利用正则进行操作验证，所以在开发之中，只要是能够接收的类型是String，那么是最方便的。
除了可以接收输入信息之外，也可以利用缓冲区读取进行文件的读取。

```
package cn.mldn.demo;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
public class TestDemo{
 public static void main(String args[])throws Exception{
  File f=new File("d:"+File.separator+"my.txt");
  BufferedReader buf=new BufferedReader(new FileReader(f));
  String str=null;
  while((str=buf.readLine())!=null){
   System.out.println(str);
  }
  buf.close();
 }
}
```

与InputStream（Reader）类相比，使用BufferedReader在进行文件信息读取的时候会更加的方便，但是以上的代码只适合读取文字。二进制的不适合。
总结：
读取数据不再直接使用InputStream，就好比输出不再直接使用OutputStream一样。

章节53 课时 204 04060_扫描流：Scanner

Scanner类的主要特点以及操作形式；
利用Scanner解决数据输入流的操作。
具体内容
如果要改进输出功能不足提供有打印流，随后又利用了BufferedReader解决了大本文数据的读取操作，但是BufferedReader类有两个问题：
它读取数据的时候只能够按照字符串返回：public String readLine()throws IOException
所有的分隔符都是固定的。
在JDK1.5后提供有一个java.util.Scanner的类，这个类专门负责解决所有输入流的操作问题。
构造方法：public Scanner(InputStream source)，接收有一个InputStream类对象，表示的是由外部设置输入的位置。
在Scanner类里面定义了两大组方法：
判断是否有指定数据：public boolean hasNextXxx()
取出数据：public 数据类型 nextXxx()
范例：以键盘输入数据为例

```
package cn.mldn.demo;
import java.util.Scanner;
public class TestDemo{
 public static void main(String args[])throws Exception{
```

```
  Scanner scan=new Scanner(System.in);
  System.out.println("请输入内容：");
  if(scan.hasNext()){
   System.out.println("输入的内容："+"\n"+scan.next());
  }
  scan.close();
 }
}
```

Scanner与BufferedReader类的操作相比，Scanner更加的容易，并且操作更为直观。

但是需要提醒的是，如果现在输入的是字符串，是否存在有hasNext()方法意义不大。但是如果是其他类型，这个hasNextXxx()就有意义了，为了保持操作的统一性，建议不管是否有用都先加上hasNext()判断。

范例：输入一个数字double

```
package cn.mldn.demo;
import java.util.Scanner;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Scanner scan=new Scanner(System.in);
  System.out.println("请输入成绩：");
  if(scan.hasNextDouble()){//表示输入的是一个小数
   double score=scan.nextDouble();//省略了转型
   System.out.println("输入的内容："+"\n"+score);
  }else{//表示输入的不是一个小数
   System.out.println("输入的不是数字，错误");
  }
  scan.close();
 }
}
```

除了以上支持的各种类型外，也可以在Scanner输入数据的时候设置正则验证。

范例：正则验证

```
package cn.mldn.demo;
import java.util.Scanner;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Scanner scan=new Scanner(System.in);
  System.out.println("请输入生日：");
  if(scan.hasNext("\\d{4}-\\d{2}-\\d{2}")){//表示输入的是一个小数
   String str=scan.next("\\d{4}-\\d{2}-\\d{2}");
   System.out.println("输入的内容："+"\n"+str);
  }else{//表示输入的不是一个小数
   System.out.println("输入的不是生日，错误");
  }
  scan.close();
 }
}
```

在Scanner类的构造里面由于接收的类型是InputStream，所以此时依然可以设置一个文件的数据流，但是进行文件读取的时候需要考虑到分隔符的问题。

设置分隔符：public Scanner useDelimiter(String pattern)

```
package cn.mldn.demo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Scanner;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Scanner scan=new Scanner(new FileInputStream(new File("d:"+File.separator+"my.txt")));
  scan.useDelimiter("\n");
  while(scan.hasNext()){
   System.out.println(scan.next());
  }
  scan.close();
 }
}
```

现在使用Scanner读取数据的时候综合来讲的确要比BufferedReader简单一些，所以在以后的开发之中，程序输入数据使用打印流，输入数据使用Scanner（如果发现Scanner不好用了，使用BufferedReader）。

总结：

InputStream类的功能不足已经被Scanner解决了。
Reader类的功能不足被BufferedReader解决了。
OutputStream类的功能不足被PrintStream解决了。
Writer类的功能不足被PrintWriter解决了。

章节53 课时 205 04061_对象序列化

本次预计讲解的知识点：
对象序列化的意义以及实现；
了解对象输入、输出流的使用；
理解transient关键字。

对象序列化（重点）
所谓的对象序列化指的就是将保存在内存中的对象数据转换为二进制数据流进行传输的操作。但是并不是所有类的对象都可以进行序列化，如果要被序列化的对象，那么其所在的类一定要实现java.io.Serializable接口。但是这个接口里面并没有任何的操作方法存在，因为它是一个标识接口，表示一种能力。
范例：定义一个可以被序列化对象的类

```
package cn.mldn.demo;
import java.io.Serializable;
@SuppressWarnings("serial")
class Book implements Serializable{//此类对象可以被序列化
 private String title;
 private double price;
 public Book(String title,double price){
  this.title=title;
  this.price=price;
 }
 public String toString() {
  return "书名：" +this.title+"价格：" +this.price;
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
 }
}
```

这个类的对象就可以实现二进制传输了。

实现序列化与反序列化
由于现在只有单机程序，所以下面将对象序列化到文件里面，随后再通过文件反序列化到程序之中。如果要想实现这样的操作需要两个类的支持：
序列化类：ObjectOutputStream，将对象变味了指定格式的二进制数据；
反序列化类：ObjectInputStream，可以将序列化的对象再转换回对象内容。
范例：实现序列化对象操作——ObjectOutputStream
构造方法：public ObjectOutputStream(OutputStream out)throws IOException
输出对象：public final void writeObject(Object obj)throws IOException

```
package cn.mldn.demo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
@SuppressWarnings("serial")
class Book implements Serializable{//此类对象可以被序列化
 private String title;
 private double price;
 public Book(String title,double price){
  this.title=title;
  this.price=price;
 }
 public String toString() {
  return "书名：" +this.title+"价格：" +this.price;
 }
}
public class TestDemo{
```

```
 public static void main(String args[])throws Exception{
  ser();
 }
 public static void ser()throws Exception{
   ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream(new File("d:"+File.separator+"book.ser") ));
   oos.writeObject(new Book("java开发",79.8));
   oos.close();
 }
}
```

范例：实现反序列化操作——ObjectInputStream
构造方法：public ObjectInputStream(InputStream in)throws IOException
读取方法：public final Object readObject()
　　throws IOException,
　　ClassNotFoundException

```
package cn.mldn.demo;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
@SuppressWarnings("serial")
class Book implements Serializable{//此类对象可以被序列化
 private String title;
 private double price;
 public Book(String title,double price){
  this.title=title;
  this.price=price;
 }
 public String toString() {
  return "书名："+this.title+"价格："+this.price;
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  dser();
 }
 public static void ser()throws Exception{
   ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream(new File("d:"+File.separator+"book.ser") ));
   oos.writeObject(new Book("java开发",79.8));
   oos.close();
 }
 public static void dser()throws Exception{
   ObjectInputStream ois=new ObjectInputStream(new FileInputStream(new File("d:"+File.separator+"book.ser") ));
   Object obj=ois.readObject();
   Book book=(Book) obj;
   System.out.println(book);
   ois.close();
 }
}
```

在以后的实际开发之中，会由容器帮助用户自动完成以上的操作。

transient关键字（理解）

以上虽然实现了序列化，但是会发现序列化操作时是将整个对象的所有属性内容进行了保存，那么如果说现在某些属性的内容不需要被保存，就可以通过transient关键字来定义。

```
private transient String title;
```

此时title属性将无法被序列化，但是大部分情况下不需要使用此关键字。

对象序列化本身就是一个非常简单的概念，但是由于其在开发之中使用很官方，所以现在对于编写代码中必须要清楚Serializable接口的作用，需要注意的是：不是所有的类都需要被序列化，只有需要传输的对象所在的类才需要进行序列化操作。

章节54 课时 206 04062_网络编程（简介）

章节54 课时 207 04063_网络编程（基本实现）

开发第一个网络程序
如果要想进行网络程序的开发，那么最为核心的两个类：
服务器类：ServerSocket，主要工作在服务器端，用于接收用户的请求；
Socket，每一个连接到服务器上的用户都通过Socket表示；
范例：定义服务器端——主要使用ServerSocket
构造方法：（设置监听接口）public ServerSocket(int port)throws IOException
接收客户端连接：public Scoket accept()throws IOException

开发第一个网络程序
如果要想进行网络程序的开发，那么最为核心的两个类：
服务器类：ServerSocket，主要工作在服务器端，用于接收用户的请求；
Socket，每一个连接到服务器上的用户都通过Socket表示；
范例：定义服务器端——主要使用ServerSocket
构造方法：（设置监听接口）public ServerSocket(int port)throws IOException
接收客户端连接：public Scoket accept()throws IOException
取得客户端的额输出功能，Socket类定义方法：
public InputStream getInputStream()throws IOException

```
package cn.mldn.demo;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
public class TestDemo{
 public static void main(String args[])throws Exception{
  ServerSocket server=new ServerSocket(9999);//所有的服务器必须有端口
  System.out.println("等待客户端连接...");
  Socket client=server.accept();//等待客户端连接
  //因为OutputStream不方便进行内容的输出，所以使用PrintStream打印流输出。
  PrintStream out=new PrintStream(client.getOutputStream());
  out.println("Hello World!");
  out.close();
  client.close();
  server.close();
 }
}
```

此时的服务器端只是输出一个Hello World字符串而后就关闭服务器操作了。只能够处理一次客户端的请求。
范例：编写客户端——Socket
构造方法：public Socket(String host,int port)throws UnknownHostException,IOException
host表示主机的IP地址，如果是本机直接方法那么使用localhost（127.0.0.1）代替IP；
得到输入数据：public InputStream getInputStream()throws IOException

```
package cn.mldn.demo;
import java.net.Socket;
import java.util.Scanner;
public class HelloClient {
 public static void main(String[] args)throws Exception {
  Socket client=new Socket("localhost",9999);//连接服务器端
  //取得客户端的输入数据流对象，表示接收服务器端的输出信息。
  Scanner scan=new Scanner(client.getInputStream());
  scan.useDelimiter("\n");
  if(scan.hasNext()){
   System.out.println("回应数据"+scan.next());
  }
  scan.close();
  client.close();
 }
}
```

客户端现在也只是连接一次服务器，并且接收输入数据输出后结束操作。
Socket定义的就是客户端

章节54 课时 208 04064_网络编程（Echo模型）

Echo程序
在网络编程之中Echo是一个经典的程序开发模型，本程序的意义在于：客户端随意输入信息并且将信息发送到服务器端，服务器端接收后前面加上一个“ECHO:”的标记返回。

本程序设计如下：
由于需要采用多次输入的形式，所以不能够每次连接后立刻关闭服务器端；
可以设置一个字符串，如果输入了“byebye”，那么才表示结束本次的Echo操作。
范例：实现服务器端

```
package cn.mldn.demo;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Scanner;
public class EchoServer {
 public static void main(String[] args) throws Exception{
  ServerSocket server=new ServerSocket(9999);//建立服务器端口
  Socket client=server.accept();//连接客户端
  //得到客户端输入数据以及向客户端输出数据的对象
  Scanner scan=new Scanner(client.getInputStream());//输入流
  PrintStream out=new PrintStream(client.getOutputStream());//输出流
  boolean flag=true;//控制多次接收操作
  while(flag){
   if(scan.hasNext()){
    String str=scan.next().trim();//得到客户端发送的内容,加trim去掉左右空格
    if(str.equalsIgnoreCase("byebye")){//程序要结束
     out.println("拜拜，下次再会！");
     flag=false;
    }else{//回应输入信息
     out.println("ECHO"+str);
    }
   }
  }
  scan.close();
  out.close();
  server.close();
  client.close();
 }
}
```

章节55 课时 209 04065_类集简介

类集就是一组Java实现的数据结构。或者再简单一点，所谓的累就指的就是对象数组的应用。
在之前讲解的时候就强调过，如果要想保存多个对象，应该使用对象数组，但是传统的对象数组有一个问题，长度是固定的（数组一般不会使用）。后来使用了链表来实现了一个动态的对象数组，但是对于链表的开发的最大的感受
链表的开发实在是太麻烦了；
如果要考虑到链表的操作性能太麻烦了；
链表使用了Object类行进保存，所有的对象必须发生向上以及强制向下转型操作；
综合以上的问题，得出的结论：如果在开发项目里面由用户自己去实现一个链表，那么这种项目的开发难度实在是太高了。并且在所有的项目里面都会存在有数据结构的应用，那么在Java设计之初就考虑到此类问题，所以提供了一个与链表类似的工具类——Vector（向量），但是后来随着时间的推移，发现这个类并不能很好的描述出数据结构这个概念，所以从java2（JDK1.2之后）提供了一个专门实现数据结构的开发框架——类集框架，在JDK1.5之后，由于泛型技术的引入，又解决了类集框架之中，所有的操作类型都使用Object所带来的安全隐患。
随后在JDK1.8里面，又针对于类集的大数据的操作环境下推出了数据流的分析操作功能；
在整个类集里面一共有一下的几个核心接口：
Collection、list、Set；
Map；
Iterator、Enumeration;
类集就是java数据结构的实现。类集就是动态对象数组。

章节55 课时 210 04066_Collection接口

连接Collection接口，以及它的相关常用操作方法定义。
具体内容（重点）
Collection是整个类集之中单值保存的最大父接口。即：每一次可以向集合里面保存一个对象。
所有的类集都在java.util包中定义。
Collection接口的定义：public interface Collection<E> extends Iterable<E>

在Collection接口里面定义有如下的几个常用操作方法。

NO	方法名称	类型	描述
1	public boolean add(E e)	普通	向集合里面保存数据
2	public boolean addAll (Collection<? extends E> c)	普通	追加一个集合
3	public void clear()	普通	清空集合，根元素为null
4	public boolean contains(Object o)	普通	判断是否包含有指定的内容，需要equals()支持
5	public boolean isEmpty()	普通	判断是否是空集合
6	public boolean remove(Object o)	普通	删除对象，需要equals()支持
7	public int size()	普通	取得集合中的元素个数
8	public Object[] toArray()	普通	将集合变为对象数组保存
9	public Iterator<E> iterator()	普通	为Iterator接口实例化

在所有的开发之中add()与iterator()两个方法的使用几率是最高的，其他的方法几乎可以忽略，但是你必须知道。但是千万要记住，continue()与remove()两个方法一定要依靠equals()方法执行。

从一般的道理来讲，现在已经知道了Collection接口的方法了，就应该使用子类为这个接口实例化，但是现在的开发由于要求的严格性，所以不会再直接使用Collection接口。而都会去使用它的两个子接口：List(允许重复)、Set(不允许重复)。

历史回顾：

最终Java刚刚推出类集框架的时候，使用最多的就是Collection接口，最大的使用环境就是EJB上。在Java的一个开源项目上——PetShop，就出现了一个问题。由于此项目是属于java业余爱好者共同开发的，没有考虑过多的性能问题以及代码或数据库的设计，就导致了整个程序的技术都是很牛的，但是性能都是很差的，于是此时正赶上微软准备退出.net平台。微软使用.net重新设计并且开发了PetShop，对外宣布，性能比Java好。于是乎这个事情就发生了本质的改变，人们开始认为.net平台性能很高（实际上和Java没什么区别），后来SUN官方重新编写了PetShop并且发布了测试报告，但是由于此时微软的宣传已经进行了，所以基本上就已经变成了性能上的差距事实了。实际不存在。

因为代码的规范化的产生，所以从PetShop开始就不再使用模糊不清的Collection接口了，而都使用List或Set子接口进行开发。

总结

1、Collection接口几乎不会再直接使用了；

2、一定要将Collection接口的方法全部记住。

章节55 课时 211 04067_List子接口

1、使用List子接口验证Collection接口中所提供的操作方法；

2、掌握List子接口的操作特点以及常用子类（Arraylist、Vector）

List子接口是Collection中最为常用的一个子接口，但是这个接口对Collection接口进行了一些功能的扩充。

NO	方法名称	类型	描述
1	public E get(int index)	普通	取得索引编号的内容
2	public E set(int index,E element)	普通	修改指定索引编号的内容
3	public ListIterator<E> listIterator()	普通	为ListIterator接口实例化

而List本身是属于接口，所以如果要想使用此接口进行操作，那么就必须存在有子类，而ArrayList是List的子类（还有另外一个Vector子类，90%的情况下选择ArrayList）

新的子类：ArrayList

ArrayList类是List接口最为常用的一个子类。下面将利用此类来验证所学习到的操作方法

范例：List的基本操作

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
public class TestDemo{
 public static void main(String args[])throws Exception{
  //设置了泛型，保证集合中所有数据的类型都是一致的
  List<String> all=new ArrayList<String>();
  System.out.println("长度："+all.size()+"是否为空："+all.isEmpty());
```

```
 all.add("Hello");
 all.add("Hello");//重复元素
 all.add("World!");
 System.out.println("长度："+all.size()+"是否为空："+all.isEmpty());
 //Collection接口定义了size()方法可以取得集合长度
 //List接口扩充了一个get()方法，可以根据索引取得数据
 for(int x=0;x<all.size();x++){
  String str=all.get(x);//取得索引数据
  System.out.println(str);
 }
}
}
```

通过演示可以发现，List集合之中所保持的数据是按照保存的顺序存放，而且允许存在有重复数据，但是一定要记住的是，List子接口扩充有get()方法。Collection没有。

范例：为Collection接口实例化（只举例帮助理解，代码无意义）

ArrayList是List接口的子类，而List又是Collection子接口，自然可以通过ArrayList为Collection接口实例化。

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.Collection;
public class TestDemo{
 public static void main(String args[])throws Exception{
  //设置了泛型，保证集合中所有数据的类型都是一致的
  Collection<String> all=new ArrayList<String>();
  all.add("Hello");
  all.add("Hello");//重复元素
  all.add("World!");
  Object obj[]=all.toArray();
  for(int x=0;x<obj.length;x++){
   System.out.println(obj[x]);
  }
 }
}
```

Collection接口与List接口相比，功能会显得有所不足，而且以上所讲解的输出方式并不是集合所会使用到的标准输出结构，只是做一个基础的展示。

范例：在集合里面保存对象

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
class Book{
 private String title;
 private double price;
 public Book(String title,double price){
  this.title=title;
  this.price=price;
 }
 @Override
 public boolean equals(Object obj) {
  if(this==obj){
   return true;
  }
  if(obj==null){
   return false;
  }
  if(!(obj instanceof Book)){
   return false;
  }
  Book book=(Book) obj;
  if(this.title.equals(book.title)&&this.price==book.price){
   return true;
  }
  return false;
 }
 @Override
 public String toString() {
```

```
  return "书名："+this.title+"价格："+this.price;
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<Book> all=new ArrayList<Book>();
  all.add(new Book("Java开发",79.8));
  all.add(new Book("JSP开发",69.8));
  all.add(new Book("Oracle开发",89.8));
  //任何情况下集合数据的删除与查询都必须提供有equals()方法，所以在Book里面要覆写equals()方法。
  all.remove(new Book("Oracle开发",89.8));
  System.out.println(all);
 }
}
```

与之前的链表相比，几乎是横向替代就是替换了一个类名称而已，因为给出的链表就是按照Collection与List接口的方法标准定义的。

旧的子类：Vector

在最早JDK1.0的时候就已经提供有Vector类，并且这个类被大量的使用。但是到了JDK1.2的时候由于类集框架的引入，所以对于整个集合的操作就有了新的标准，那么为了可以继续保留下Vector类，所以让这个类多实现了一个List接口。

范例：使用Vector

```
package cn.mldn.demo;
import java.util.List;
import java.util.Vector;
public class TestDemo{
 public static void main(String args[])throws Exception{
  //设置了泛型，保证集合中所有数据的类型都是一致的
  List<String> all=new Vector<String>();
  System.out.println("长度："+all.size()+"是否为空："+all.isEmpty());
  all.add("Hello");
  all.add("Hello");//重复元素
  all.add("World!");
  System.out.println("长度："+all.size()+"是否为空："+all.isEmpty());
  //Collection接口定义了size()方法可以取得集合长度
  //List接口扩充了一个get()方法，可以根据索引取得数据
  for(int x=0;x<all.size();x++){
   String str=all.get(x);//取得索引数据
   System.out.println(str);
  }
 }
}
```

面试题：请解释ArrayList与Vector的区别？

NO	区别点	ArrayList(90%)	Vector(10%)
1	推出时间	JDK1.2推出，属于新的类	JDK1.0推出，属于旧的类
2	性能	采用异步处理	采用同步处理
3	数据安全性	非线程安全	线程安全
4	输出	Iterator、ListIterator、foreach	Iterator、ListIterator、foreach、Enumeration

在以后的开发之中，如果使用了List子接口就用ArrayList子类。

总结：

1、List中的数据保存顺序就是数据的添加顺序；

2、List集合中可以保存有重复的元素；

3、List子接口比Collection接口扩充了一个get()方法；

4、List选择子类就是用ArrayList。

章节55 课时 212 04068_Set子接口

Set子接口的操作特点以及常用子类

深入分析两个常用子类的操作特征。

具体内容

在Collection接口下又有另外一个比较常用的子接口为Set接口（20%），Set接口并不像List接口那样对于Collection接口进行了大量的扩充，而是简单的继承了Collection接口。也就没有了之前List集合所提供的get()方法。

Set接口下有两个常用的子类：HashSet、TreeSet

范例：观察HashSet子类的特点

```
package cn.mldn.demo;
import java.util.HashSet;
import java.util.Set;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Set<String> all=new HashSet<String>();
  all.add("NIHAO");
  all.add("Hello");
  all.add("Hello");
  all.add("World");
  System.out.println(all);
 }
}
```

通过演示发现，Set集合下没有重复元素（这一点是Set接口的特征），同时发现在里面保存的数据是没有任何顺序的（我怎么感觉像是按首字母大小排序呢），即：HashSet子类的特征属于无序排列。

范例：使用TreeSet子类

```
package cn.mldn.demo;
import java.util.Set;
import java.util.TreeSet;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Set<String> all=new TreeSet<String>();
  all.add("X");
  all.add("B");
  all.add("B");
  all.add("A");
  System.out.println(all);
 }
}
```

此时的程序使用了TreeSet子类，发现没有重复数据，以及所保存的内容自动排序。

关于数据排序的说明

既然TreeSet子类保存的内容可以进行排序，那么下面不如就编写一个自定义的类来完成保存。

集合就是一个动态的对象数组，那么如果要想为一组对象进行排序，在Java里面必须要使用比较器，应该使用Comparable完成比较。在比较方法里面需要将这个类的所有属性都一起参与到比较之中。

范例：TreeSet排序

```
package cn.mldn.demo;
import java.util.Set;
import java.util.TreeSet;
class Book implements Comparable<Book>{
 private String title;
 private double price;
 public Book(String title,double price){
  this.title=title;
  this.price=price;
 }
 public String toString() {
  return "书名："+this.title+",价格："+this.price+"。";
 }

 public int compareTo(Book o){
  if(this.price>o.price){
   return 1;
  }else if(this.price<o.price){
   return -1;
  }else{
   return this.title.compareTo(o.title);//调用了String类的比较大小
  }
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  Set<Book> all=new TreeSet<Book>();
```

```
  all.add(new Book("Java开发",79.8));
  all.add(new Book("Java开发",79.8));//完全重复
  all.add(new Book("JSP开发",79.8));//价格重复
  all.add(new Book("Android开发",89.8));//都不重复
  System.out.println(all);
 }
}
```

通过检测可以发现TreeSet类主要是依靠Comparable接口中的compare To()方法判断是否是重复数据，如果返回的是0，那么它就认为是重复数据，不会被保存。

关于重复元素的说明

很明显，Comparable接口只能够负责TreeSet子类进行重复元素的判断，它并不是真正的用于能够进行重复元素验证的操作。如果要想判断重复元素那么稚嫩巩固依靠Object类中所提供的方法：

取得哈希码：public int hashCode()

先判断对象的哈希码是否相同，依靠哈希码取得一个对象的内容；

对象比较：public boolean equals(Object obj)

再将对象的属性进行一次的比较

范例：利用eclipse自带的工具生成hashCode()和equals()

写完Book类后点击Source选择 Generate hashCode() and equals()

```
package cn.mldn.demo;
import java.util.HashSet;
import java.util.Set;
class Book  {
 @Override
 public int hashCode() {
  final int prime = 31;
  int result = 1;
  long temp;
  temp = Double.doubleToLongBits(price);
  result = prime * result + (int) (temp ^ (temp >>> 32));
  result = prime * result + ((title == null) ? 0 : title.hashCode());
  return result;
 }
 @Override
 public boolean equals(Object obj) {
  if (this == obj)
   return true;
  if (obj == null)
   return false;
  if (getClass() != obj.getClass())
   return false;
  Book other = (Book) obj;
  if (Double.doubleToLongBits(price) != Double.doubleToLongBits(other.price))
   return false;
  if (title == null) {
   if (other.title != null)
    return false;
  } else if (!title.equals(other.title))
   return false;
  return true;
 }
 private String title;
 private double price;
 public Book(String title,double price){
  this.title=title;
  this.price=price;
 }
 public String toString() {
  return "书名：" +this.title+",价格：" +this.price+"。";
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  Set<Book> all=new HashSet<Book>();
  all.add(new Book("Java开发",79.8));
```

```
  all.add(new Book("Java开发",79.8));//完全重复
  all.add(new Book("JSP开发",79.8));//价格重复
  all.add(new Book("Android开发",89.8));//都不重复
  System.out.println(all);
 }
}
```

以后再非排序的情况下，只要是判断重复元素依靠的永远都是hashCode()与equals()。

总结

1、在开发之中，Set接口绝对不是首选，如果真要使用也建议使用HashSet子类；

2、Comparable这种比较器大部分情况下只会存在于Java理论范畴内，例如：要进行TreeSet；

3、Set不管如何操作，必须始终保持一个前提：数据不能够重复。

章节55 课时 213 04069_集合输出

Collection、List、Set三个接口里面只有List接口是最方便进行输出操作的，所以本次要讲解集合的四种输出操作形式。

集合在JDK1.8之前支持四种输出：Iterator(95%)、ListIterator(0.05%)、Enumeration(4.9%)、foreach(0.05%)。

迭代输出：Iterator（核心）

如果遇见了集合操作，那么一般而言都会使用Iterator接口进行集合的输出操作，首先来观察一下Iterator接口的定义

```
public interface Iterator<E>{
 public boolean?hasNext();
 public E Next();
}
```

Iterator接口如果要想取得本节口实例化只能够依靠Collection接口，在Collection接口里面定义有如下的一个操作方法：

```
public Iterator<E> iterator();
```

范例：使用Iterator输出集合，既然Collection有这个方法List和Set全有了。

```
package cn.mldn.demo;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Set <String> all=new HashSet<String> ();
  all.add("姜笨");
  all.add("与蠢");
  all.add("与蠢");
  Iterator <String> iter=all.iterator();
  while(iter.hasNext()){
   String str=iter.next();
   System.out.println(str);
  }
 }
}
```

从今以后，只要是遇见集合的输出不需要做任何复杂的大脑思考，直接使用Iterator接口输出。

双向迭代：ListIterator（了解）

Iterator本身只具备有钱向后的输出（99%的情况下都是这么做的），但是有些人认为应该让其支持双向的输出，即可以实现由前向后，也可以实现由后向前。那么就可以使用Iterator的子接口——ListIterator接口。在这个接口里面主要是两个方法：

判断是否有前一个元素：public boolean hasPrevious()

取得前一个元素：public E previous();

ListIterator是专门为List子接口定义的输出接口，方法：

```
public ListIterator<E> listIterator()
```

范例：完成双向迭代

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List <String> all=new ArrayList<String> ();
  all.add("A");
  all.add("B");
  all.add("C");
  System.out.println("由前向后输出：");
  ListIterator<String> iter=all.listIterator();
```

```
  while(iter.hasNext()){
   String str=iter.next();
   System.out.print(str+"、");
  }
  System.out.println("\n由后向前输出：");
  while(iter.hasPrevious()){
   System.out.print(iter.previous()+"、");
  }
 }
}
```

如果要想实现由后向前的输出操作之前，一定要首先发生由前向后的输出操作。

foreach输出

理论上，foreach输出还是挺方便的使用，但是如果你们现在过多的使用foreach不利于理解程序。foreach本身可以方便的输出数组，但是集合也可以。

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List <String> all=new ArrayList<String> ();
  all.add("A");
  all.add("B");
  all.add("C");
  for(String str:all){
   System.out.println(str);
  }
 }
}
```

在初学阶段还是强烈建议使用Iterator操作。

Enumeration输出

Enumeration是与Vector类一起在JDK1.0的时候退出的输出接口，即：最早的Vector如果要想输出就需要使用Enumeration接口完成，那么此接口定义如下：

```
public interface Enumeration<E>{
 public boolean hasMoreElements();//判断是否有下一个元素，等同于hasNext()
 public E nextElement();//取出当前元素，等同于Next()
}
```

但是如果要想取得Enumeration接口的实例化对象只能够依靠Vector子类。在Vector子类里面定义有如下的操作方法：

取得Enumeration操作对象：

```
package cn.mldn.demo;
import java.util.Enumeration;
import java.util.Vector;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Vector <String> all=new Vector<String> ();
  all.add("A");
  all.add("B");
  all.add("C");
  Enumeration<String> enu=all.elements();
  while(enu.hasMoreElements()){
   String str= enu.nextElement();
   System.out.println(str);
  }
 }
}
```

在一些古老的操作方法上，此接口依然会使用到，所以必须掌握。

总结：

虽然集合的输出提供有四种形式，但是一定要以Iterator和Enumeration为主，并且一定要记清楚这两个接口的核心操作方法。

章节55 课时 214 04070_Map接口

Collection每一次都只会保存一个对象，而Map主要是负责保存一对对象的信息。

1、Map接口的主要操作方法；

2、Map接口的常用子类。

如果说现在要保存一对关联数据（key=value）的时候，那么如果直接使用Collection就不能直接满足于要求，可以使用Map接口实现此

类数据的保存，并且Map接口还提供有根据key查找value的功能。
在Map接口里面定义有如下的常用方法：

NO	方法名称	类型	描述
1	public V put(K key,V value)	普通	向集合中保存数据
2	public V get(Object key)	普通	根据key查找对应的value数据
3	public Set<Map.Entry<K,V>> entrySet()	普通	将Map集合转化为Set集合
4	public Set<K> keySet()	普通	取出全部的key

在Map的接口下有两个常用子类：HashMap、Hashtable
范例：观察HashMap的使用

```
package cn.mldn.demo;
import java.util.HashMap;
import java.util.Map;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Map <String,Integer> map=new HashMap<String,Integer>();
  map.put("壹", 1);
  map.put("贰", 2);
  map.put("叁", 3);
  map.put("叁", 33);
  System.out.println(map);//key数据重复
 }
}
```

通过以上操作可以发现如下特点：
使用HashMap定义的Map集合是无序存放的（顺序无用）；
如果发现出现了重复的key会进行覆盖，使用新的内容替换掉旧的内容。
在Map接口里面提供有get()方法，这个方法的主要功能是根据key查找所需要的value
范例：查询操作

```
package cn.mldn.demo;
import java.util.HashMap;
import java.util.Map;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Map <String,Integer> map=new HashMap<String,Integer>();
  map.put("壹", 1);
  map.put("贰", 2);
  map.put("叁", 3);
  map.put(null, 0);
  System.out.println(map.get("壹"));
  System.out.println(map.get("陆"));
  System.out.println(map.get(null));
 }
}
```

通过以上的代码可以发现，Map存放数据的最终目的实际上就是为了信息的查找，但是Collection存放数据的目的是为了输出。

范例：取得全部的key

```
package cn.mldn.demo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Map <String,Integer> map=new HashMap<String,Integer>();
  map.put("壹", 1);
  map.put("贰", 2);
  map.put("叁", 3);
  map.put(null, 0);
  Set<String> set=map.keySet();//取得全部的key
  Iterator <String> iter=set.iterator();
```

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

在Map接口下还有一个Hashtable的子类，此类在JDK1.0的时候提供的，属于最早的Map集合的实现操作，在JDK1.2的时候让其多实现了一个Map接口，从而保存下来继续使用。

范例：使用Hashtable

```
package cn.mldn.demo;
import java.util.Hashtable;
import java.util.Map;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Map <String,Integer> map=new Hashtable<String,Integer>();
  map.put("壹", 1);
  map.put("贰", 2);
  map.put("叁", 3);
  System.out.println(map);
 }
}
```

现在发现Hashtable里面对于key和value的数据都不允许设置为null

面试题：请解释HashMap与Hashtable的区别

NO	区别点	HashMap(90%)	Hashtable(10%)
1	推出时间	JDK1.2推出，属于新的类	JDK1.0推出，属于旧的类
2	性能	采用异步处理	采用同步处理
3	数据安全性	非线程安全	线程安全
4	设置null	允许key或value为null	不允许设置为null

而在实际的使用中，遇见了Map接口基本上就是用HashMap类。

关于Iterator输出的问题（核心）

在之前强调过，只要是集合的输出那么一定要使用Iterator完成，但是在整个Map接口里面并没有定义任何的可以返回Iterator接口对象的方法，所以下面如果要想使用Iterator输出Map集合，首先必须要针对于Map集合和Collection集合保存数据的特点进行分析后才能够实现。

每当御用使用put()方法向Map集合里面保存一堆数据的时候，实际上所有的数据都会被自动的封装为Map.Entry接口对象。那么来观察Map.Entry接口的定义。

```
//使用static定义的内部接口就是外部接口
public static interface Map.Entry<K,V>
```

在这个接口里面定义了两个操作：

取得key: public K getKey();

取得vlaue:public V getValue();

在Map接口里面定义有一个将Map集合转化为Set集合的方法：

```
public Set<Map.Entry<K,V>> entrySet()
```

Map集合利用Iterator接口输出的步骤：

1、利用Map接口的entrySet()方法将Map集合变为Set集合，里面的泛型是Map.Entry
2、利用Set集合中的iterator()方法将Set集合进行Iterator输出；
3、每一次Iterator循环取出的都是Map.Entry接口对象，利用此对象进行key与value的取出。

范例：利用Iterator实现Map接口的输出

```
package cn.mldn.demo;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Map <String,Integer> map=new Hashtable<String,Integer>();
  map.put("壹", 1);
  map.put("贰", 2);
  map.put("叁", 3);
  //将Map集合变为Set集合，目的是为了使用iterator()方法
  Set<Map.Entry<String,Integer>> set=map.entrySet();
  Iterator<Map.Entry<String,Integer>> iter=set.iterator();
  while(iter.hasNext()){
   Map.Entry<String, Integer> me=iter.next();
   System.out.println(me.getKey()+"="+me.getValue());
  }
 }
}
```

以上的代码会不定期的出现，一定要掌握步骤，并且一定要熟练掌握。

关于Map集合中key的说明

在使用Map接口的时候可以发现，几乎可以使用任意的类型来作为key或value的存在，那么也就表示也可以使用自定义的类型作为key.那么这个作为key的自定义类必须要覆写Object类之中的hashCode()与equals()，只要靠这两个方法才能够确定元素是否重复，而在Map中指的是是否能够找到。

```
package cn.mldn.demo;
import java.util.HashMap;
import java.util.Map;
class Book{
 private String title;
 public Book(String title){
  this.title=title;
 }
 @Override
 public String toString() {
  return "书名：" +this.title;
 }
 @Override
 public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((title == null) ? 0 : title.hashCode());
  return result;
 }
 @Override
 public boolean equals(Object obj) {
  if (this == obj)
   return true;
  if (obj == null)
   return false;
  if (getClass() != obj.getClass())
   return false;
  Book other = (Book) obj;
  if (title == null) {
   if (other.title != null)
    return false;
  } else if (!title.equals(other.title))
   return false;
  return true;
 }
```

```
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  Map <Book,String> map=new HashMap<Book,String>();
  map.put(new Book("Java开发"), new String("Java"));
  System.out.println(map.get(new Book("Java开发")));
 }
}
```

在以后使用Map集合的时候，首选的key的类型是String，尽量不要去使用自定义的类型作为key.

总结：

1、Map集合保存数据的目的是为了查询使用，而Collection集合保存数据的目的是为了输出使用。

2、Map使用Iterator接口输出的步骤以及具体实现代码要背诵；

3、HashMap可以保存null（Hashtable不能），key如果出现重复会覆盖。

章节55 课时 215 04071_Stack子类

Stack表示的是栈操作，栈是一种先进后出的数据结构。而Stack是Vector的子类。

public class Stack<E> extends Vector<E>

但是需要注意的是，虽然Stack是Vector子类，可是它不会使用Vector类的方法，它使用自己的方法。

入栈：public E push(E item)

出栈：public E pop()

范例：观察栈的操作

```
package cn.mldn.demo;
import java.util.Stack;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Stack<String> all=new Stack<String>();
  all.push("A");
  all.push("B");
  all.push("C");
  System.out.println(all.pop());
  System.out.println(all.pop());
  System.out.println(all.pop());
  System.out.println(all.pop());//EmptyStackException
 }
}
```

在进行栈操作的过程之中，如果栈已经没有数据了，那么无法继续出栈。

总结

栈的这种操作现在唯一还算是有点能够编程的应用，就在Android上。

章节55 课时 216 04072_Properties子类

本次预计讲解的知识点：

国际化程序特点：同一个程序，根据不同的语言环境选择资源文件，所有的资源文件后缀必须是“*.properties”

Properties类的操作特点

Properties是Hashtable的子类，主要是进行属性的操作（属性的最大特点是利用字符串设置key和value）。首先来观察Properties类的定义结构：

public class Properties extends Hashtable<Object,Object>

在使用Properties类的时候不需要设置泛型类型，因为从它一开始出现就只能够保存String。在Properties类中主要使用一下方法：

设置属性：public Object setProperty(String key,String value)

取得属性：public String getProperty(String key)，如果key不存在返回null

取得属性：public String getProperty(String key,String defaultValue)，如果key不存在返回默认值。

范例：属性的基本操作

```
package cn.mldn.demo;
import java.util.Properties;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Properties pro=new Properties();
  pro.setProperty("BJ", "北京");
  pro.setProperty("TJ", "天津");
  System.out.println(pro.getProperty("BJ"));
  System.out.println(pro.getProperty("GZ"));
  System.out.println(pro.getProperty("GZ","没有此记录"));
```

```
 }
}
```
在Properties类里面提供有数据的输出操作
public void store(OutputStream out,String comments)throws IOException
范例：将属性信息保存在文件里
```
package cn.mldn.demo;
import java.io.File;
import java.io.FileOutputStream;
import java.util.Properties;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Properties pro=new Properties();
  pro.setProperty("BJ", "北京");
  pro.setProperty("TJ", "天津");
  //一般而言后缀可以随意设置，但是标准来讲，既然是属性文件，后缀必须是*.properties,这样也是为了与国际化对应
  pro.store(new FileOutputStream(new File("d:"+File.separator+"area.properties")), "Area Info");
 }
}
```
也可以从指定的输入流中读取属性信息：public void load(InputStream inStream)
        throws IOException
范例：通过文件流读取属性内容
```
package cn.mldn.demo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
public class TestDemo{
 public static void main(String args[])throws Exception{
  Properties pro=new Properties();
  pro.load(new FileInputStream(new File("d:"+File.separator+"area.properties")));
  System.out.println(pro.getProperty("BJ"));
 }
}
```
对于属性文件（资源文件）除了可以使用Properties类读取之外，也可以使用ResourceBundle类读取，这也就是将输出的属性文件统一设置的后缀为*.properties的原因所在。
总结
1、资源文件的特点：
 key=value
 key=value
2、资源文件的数据一定都是字符串。

章节55 课时 217 04073_Collections工具类

在Java提供类库的时候考虑到用户的使用方便性，所以专门提供了一个集合的工具类——Collections，这个工具类可以实现List、Map、Set集合的操作。

 为集合追加数据：public static <T> boolean addAll(Collection<? super T> c,T... elements)
```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  Collections.addAll(all,"A","B","C","D","E");
  System.out.println(all);
  Collections.reverse(all);
  System.out.println(all);
 }
}
```
面试题：请解释Collection与Collections的区别？
Collection是集合操作的接口；
Collections是集合操作的工具类，可以进行List、Map、Set集合的操作。
总结：
这个类我们不会使用到，但是作为知识点清楚有这么一个类的存在就够了。

章节55 课时 218 04074_数据流：Stream

本次预计讲解知识点：

1、离不开Lamda表达式；

2、方法引用、四个函数式接口；

3、如何使用Stream数据流进行集合的辅助操作；MapReduce的使用过程。

具体内容

在JDK1.8开始发现整个类集里面所提供的接口都出现了大量的default或者static方法，以Collection的父接口Iterable接口里面定义的一个方法来观察：

default void forEach(Consumer<? super T> action)。

范例：利用forEach()方法输出

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Hello");
  all.add("Hello");
  all.add("World");
  all.forEach(System.out::println);
 }
}
```

不会采用以上的方式完成，因为forEach()只能够实现输出，但是很多时候我们在进行集合数据输出的同事还需要针对于数据进行处理，也就是说Iterator输出时主要使用的方式。

除了使用Iterator迭代取出数据并且处理之外，在JDK1.8里面又提供了一个专门可以进行数据处理的类：Stream，这个类的对象可以利用Collection接口提供的方法进行操作：default Stream<E>stream()。

范例：取得Steam对象

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Hello");
  all.add("Hello");
  all.add("World");
  Stream<String> stream=all.stream();//取得Stream类的对象
  System.out.println(stream.count());//取得数据个数
 }
}
```

既然取得了Stream类的对象那么下面进行数据的加工处理操作。

范例：取消掉重复数据

Stream类里面提供了一个消除重复的方法，public Stream<T>distinct();

收集器（最后使用收集）：public <R,A> R collect(Collector<? super T,A,R> collector)

|-Collectors类：public static <T> Collector<T,?,List<T>> toList()

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Hello");
  all.add("Hello");
  all.add("World");
  all.add("World");
  all.add("NIHAO");
  all.add("NIHAO");
  Stream<String> stream=all.stream();//取得Stream类的对象
  //System.out.println(stream.distinct().count());//取得数据个数
  List<String> newAll=stream.distinct().collect(Collectors.toList());
```

```
  newAll.forEach(System.out::println);
 }
}
```
既然Stream类是进行数据处理的，那么在数据处理过程之中就不可能不去思考数据的筛选问题（过滤），Stream类里面支持有数据的过滤操作：public Stream<T> filter(Predicate<? super T> predicate)
//Predicate断言型的函数式接口
范例：数据过滤

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Android");
  all.add("Java");
  all.add("Ios");
  all.add("jsp");
  all.add("ORACLE");
  Stream<String> stream=all.stream();//取得Stream类的对象
  //增加了数据的过滤操作，使用了断言型的函数式接口，使用了String类中的contains()方法。
  List<String> newAll=stream.distinct().filter((x)->x.contains("a")).collect(Collectors.toList());
  newAll.forEach(System.out::println);
 }
}
```
整个现在的数据操作已经成功的实现了过滤应用，但是有一个缺点，发现这个时候的过滤是区分大小写的。那么证明在对数据过滤之前需要对数据进行一些额外的处理，例如：转大写或统一转小写。在Stream接口里面提供有专门的处理方法public <R> Stream<R> map(Function<? super T,? extends R> mapper)
范例：数据处理后过滤
```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Android");
  all.add("Java");
  all.add("Ios");
  all.add("jsp");
  all.add("ORACLE");
  Stream<String> stream=all.stream();//取得Stream类的对象
  //增加了数据的过滤操作，使用了断言型的函数式接口，使用了String类中的contains()方法。
  List<String> newAll=stream.distinct().map((x)->x.toLowerCase()).filter((x)->x.contains("a")).collect(Collectors.toList());
  newAll.forEach(System.out::println);
 }
}
```
在Stream接口里面提供有进行集合数据分页的操作：
  设置跳过的数据行数：public Stream<T>skip(long n);
  设置取出的数据个数：public Stream<T>limit(long maxSize)
```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Android");
  all.add("Java");
  all.add("Ios");
```

```
  all.add("jsp");
  all.add("ORACLE");
  Stream<String> stream=all.stream();//取得Stream类的对象
  //增加了数据的过滤操作，使用了断言型的函数式接口，使用了String类中的contains()方法。
  List<String> newAll=stream.distinct().map((x)->x.toLowerCase()).skip(2).limit(2).collect(Collectors.toList());
  newAll.forEach(System.out::println);
 }
}
```

在Stream接口里面还可以进行数据的全匹配或部分匹配

全匹配：public boolean allMatch(Predicate<? super T> predicate)

匹配任意一个：public boolean anyMatch(Predicate<? super T> predicate)

范例：实现数据的匹配查询

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Android");
  all.add("Java");
  all.add("Ios");
  all.add("jsp");
  all.add("ORACLE");
  Stream<String> stream=all.stream();//取得Stream类的对象
  if(stream.anyMatch((x)->x.contains("jsp"))){
   System.out.println("数据存在");
  }
 }
}
```

在实际之中有可能会出现多个匹配的条件，在断言型的函数式接口里面提供有如下的方法：

或操作：default Predicate<T> or(Predicate<? super T> other)

与操作：default Predicate<T> and(Predicate<? super T> other)

范例：设置多个条件

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Stream;
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<String> all=new ArrayList<String>();
  all.add("Android");
  all.add("Java");
  all.add("Ios");
  all.add("jsp");
  all.add("ORACLE");
  Predicate<String> p1=(x)->x.contains("jsp");
  Predicate<String> p2=(x)->x.contains("Ios");
  Stream<String> stream=all.stream();//取得Stream类的对象
  if(stream.anyMatch(p1.or(p2))){//同时使用2个条件
   System.out.println("数据存在");
  }
 }
}
```

利用这样的匹配条件，可以针对于数据进行方便的查询操作。

如果要想更好的发挥出Stream的操作优势，必须结合MapReduce一起观察。

数据分析方法：public Optional<T> reduce(BinaryOperator<T> accumulator)

|-就是做数据统计使用的。

范例：实现一个MapReduce

```
class ShopCar{
 private String pname;
 private double price;
 private int amount;
```

```
 public ShopCar(String pname,double price,int amount) {
  this.pname=pname;
  this.price=price;
  this.amount=amount;
 }
 public String getPname() {
  return pname;
 }
 public double getPrice() {
  return price;
 }
 public int getAmount() {
  return amount;
 }
}
```

本类设计的时候专门设计出了商品的单价和数量，这样如果要想取得某一个商品的花费就必须使用数量*单价

范例：进行数据的保存于初步的处理

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
class ShopCar{
 private String pname;
 private double price;
 private int amount;
 public ShopCar(String pname,double price,int amount) {
  this.pname=pname;
  this.price=price;
  this.amount=amount;
 }
 public String getPname() {
  return pname;
 }
 public double getPrice() {
  return price;
 }
 public int getAmount() {
  return amount;
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<ShopCar> all=new ArrayList<ShopCar>();
  all.add(new ShopCar("王惊雷娃娃",800.0,20));
  all.add(new ShopCar("生姜",1.0,2000));
  all.add(new ShopCar("鸭子",19.0,20));
  all.add(new ShopCar("盆",20.0,10));
  all.stream().map((x)->x.getPrice()*x.getAmount()).forEach(System.out::println);;

 }
}
```

此时已经针对于每一个数据进行了处理。但是你这个时候的处理没有总价。

于是要对处理后的数据进行统计操作，要用reduce()完成。

范例：统计处理数据

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.List;
class ShopCar{
 private String pname;
 private double price;
 private int amount;
 public ShopCar(String pname,double price,int amount) {
  this.pname=pname;
  this.price=price;
  this.amount=amount;
```

```
 }
 public String getPname() {
  return pname;
 }
 public double getPrice() {
  return price;
 }
 public int getAmount() {
  return amount;
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<ShopCar> all=new ArrayList<ShopCar>();
  all.add(new ShopCar("王惊雷娃娃",800.0,20));
  all.add(new ShopCar("生姜",1.0,2000));
  all.add(new ShopCar("鸭子",19.0,20));
  all.add(new ShopCar("盆",20.0,10));
  double s=all.stream().map((x)->x.getPrice()*x.getAmount()).reduce((sum,m)->sum+m).get();
  System.out.println("花费总金额:"+s);
 }
}
```

以上只是实现了一个最简单的MapReduce，但是所完成的统计功能实在是过于有限，如果要完成更完善的统计操作，需要使用Stream接口里面定义的一下方法：

按照Double处理：public DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)

按照Int处理：public IntStream mapToInt(ToIntFunction<? super T> mapper)

按照long处理：public LongStream mapToLong(ToLongFunction<? super T> mapper)

（Stream是BaseStream的子接口，而DoubleStream也是BaseStream的子接口）

BaseStream：（DoubleStream, IntStream, LongStream, Stream<T>）

范例：实现数据的统计操作

```
package cn.mldn.demo;
import java.util.ArrayList;
import java.util.DoubleSummaryStatistics;
import java.util.List;
class ShopCar{
 private String pname;
 private double price;
 private int amount;
 public ShopCar(String pname,double price,int amount) {
  this.pname=pname;
  this.price=price;
  this.amount=amount;
 }
 public String getPname() {
  return pname;
 }
 public double getPrice() {
  return price;
 }
 public int getAmount() {
  return amount;
 }
}
public class TestDemo{
 public static void main(String args[])throws Exception{
  List<ShopCar> all=new ArrayList<ShopCar>();
  all.add(new ShopCar("王惊雷娃娃",800.0,20));
  all.add(new ShopCar("生姜",1.0,2000));
  all.add(new ShopCar("鸭子",19.0,20));
  all.add(new ShopCar("盆",20.0,10));
  DoubleSummaryStatistics dss=all.stream().mapToDouble((sc)->sc.getAmount()*sc.getPrice()).summaryStatistics();
  System.out.println("商品数量"+dss.getCount());
  System.out.println("总共花费"+dss.getSum());
  System.out.println("平均花费"+dss.getAverage());
```

```
  System.out.println("最大花费"+dss.getMax());
  System.out.println("最小花费"+dss.getMin());
 }
}
```

这种操作实在是太麻烦了，但是它的确是简化了代码的编写。

总结：

以上的操作忘了吧，就像它从未出现过一样。。。

MapReduce：Map处理数据、Reduce分析数据。

章节56 课时 219 04075_JDBC简介

在Java语言设计的识货除了考虑到平台的编程技术之外，为了方便用户进行各种情况下的开发还提供有一系列的服务，而数据库的操作就属于Java的服务范畴。服务的最大特点：所有的操作部分几乎都是固定的流程，也就是说服务几乎没有技术含量，属于应用，而对于所有的应用，代码的流程是固定的，只有多写才能记住。

JDBC（Java Database Connective），Java数据库连接技术，即：是有Java提供的一组与平台无关的数据库的操作标准。（是一组接口的组成），由于数据库属于资源操作，所以所有的数据库操作的最后必须要关闭数据库连接。

在JDBC技术范畴里面实际上规定了四种Java数据库操作的形式。

形式一：JDBC-ODBC桥接技术（100%不用）

|-在Windows中有ODBC技术，ODBC指的是开发数据库连接，是由微软提供的数据库连接应用，而Java可以利用JDBC间接操作ODBC技术，从而实现数据库的连接；

|-流程：程序→JDBC→ODBC→数据库，性能室最差的，支持的版本是最新的；

形式二：JDBC直接连接：

|-直接由不同的数据库生产商提供指定的数据库连接驱动程序（实现了Java的数据操作标准的一群类），此类方式由于是JDBC直接操作数据库，所以性能是最好的，但是支持的JDBC版本不是最新的。

形式三：JDBC的网络连接：

|-使用专门的数据库的网络连接指令进行指定主机的数据库操作，此种方式使用的最多。

形式四：模拟指定数据库的通讯协议自己编写数据库操作。

Java几乎连接任何数据库性能都是很高的，但是只有一个数据库性能室最初的：SQL Server

总结：

在国内使用最多的集中数据库：Oracle、MySQL（MariaDB）、MongoDB。

章节56 课时 220 04076_连接Oracle数据库

本次预计讲解的知识点：

1、使用JDBC技术连接Oracle数据库；

2、观察JDBC中常用类与接口的使用结构

具体内容

在Java之中，所有数据库操作的类和接口都保存在了java.sql包里面，在这个包里面核心的组成如下：

一个类：DriverManager类；

四个接口：Connection、Statement、ResultSet、PreparedStatement。

所有的JDBC连接数据库的操作流程都是固定的，按照如下的几步完成：

1、加载数据库的驱动程序（向容器加载）

2、进行数据库连接（通过DriverManager类完成，Connection表示连接）；

3、进行数据的CRUD（Statement、PreparedStatement、ResultSet）

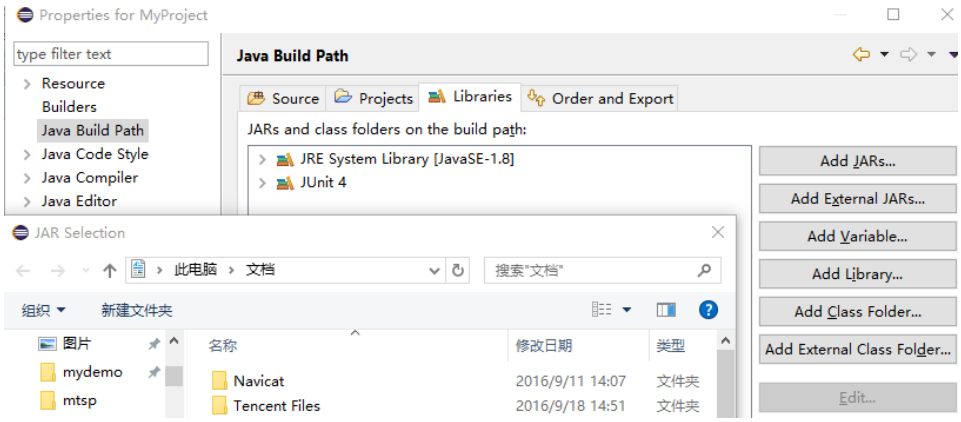
4、关闭数据库操作以及连接（直接关闭连接就够了）；

一、加载驱动程序

所有的JDBC实际上都是由各个不同的数据库生产商提供的数据库驱动程序，这些驱动程序都是以*.jar文件的方式给出来的，所以如果要使用就需要为其配置CLASSPATH,而后要设置驱动程序的类名称（包.类）；

驱动程序：E:\app\mldn\product\11.2.0\dbhome_1\jdbc\lib\bjdbc6.jar

我自己的目录：D:\Oracle\jdbc\lib\bjdbc6.jar



Oracle驱动程序类：oracle.jdbc.driver.OracleDriver

加载类使用：Class.forName("oracle.jdbc.driver.OracleDriver");

二、连接数据库

如果要想连接数据库需要提供有如下的几个信息：

（前提：数据库服务要打开，OracleOraDb11g_home1TNSListener，OracleServiceORCL）

数据库的连接地址：jdbc:oracle:连接方式@ip地址:端口号:服务器名称（即数据库SID）

|-要连接本机的ORCL数据库：jdbc:oracle:thin:@localhost:1521:ORCL

http://zhidao.baidu.com/question/400501853.html?qbl=relate_question_1&word=jdbc:oracle:thin:

数据库的用户名：scott

数据库的密码：tiger

要连接数据库必须依靠DriverManager类完成，在此类定义有如下的方法：

连接数据库：public static Connection getConnection(String url,String user,String password)throws SQLException

在JDBC里面，每一个数据库连接都要求使用一个Connection对象进行封装，所以只要有一个新的Connection对象就表示要连接一次数据库。

四、关闭数据库

Connection接口提供有close()方法：public void close()throws SQLException

```
package cn.mldn.demo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class TestDemo{
 private static final String DBDRIVER="oracle.jdbc.dirver.OracleDriver";
 private static final String USER="scott";
 private static final String PASSWORD="tiger";
 private static final String DBURL="jdbc:oracle:thin:@localhost:1521:ORCL";
 public static void main(String[] args)throws Exception {
  //第一步，加载数据库驱动程序
  Class.forName(DBDRIVER);
  //第二步：连接数据库
  Connection conn=DriverManager.getConnection(DBURL,USER,PASSWORD);
  System.out.println(conn);
  //第三步：增加数据
  Statement stmt=conn.createStatement();
            //如果SQL语句太长需要换行，一定要在各行前后加上空格。
  String sql=" INSERT INTO member(mid,name,birthday,age,note) VALUES "
    +" (myseq.nextval,'张三',TO_DATE('1998-10-2','yyyy-mm-dd'),17,'是个人')";
  int len=stmt.executeUpdate(sql);
  System.out.println("更新的行数是："+len);
  //第四步：关闭数据库
  conn.close();
 }
}
```

查询数据：

```
package cn.mldn.demo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
public class TestDemo{
```

```
 private static final String DBDRIVER="oracle.jdbc.dirver.OracleDriver";
 private static final String USER="scott";
 private static final String PASSWORD="tiger";
 private static final String DBURL="jdbc:oracle:thin:@localhost:1521:ORCL";
 public static void main(String[] args)throws Exception {
  //第一步，加载数据库驱动程序
  Class.forName(DBDRIVER);
  //第二步：连接数据库
  Connection conn=DriverManager.getConnection(DBURL,USER,PASSWORD);
  System.out.println(conn);
  //第三步：查询数据
  Statement stmt=conn.createStatement();
  String sql="SELECT mid,name,birthday,age,note FROM member";
  ResultSet rs=stmt.executeQuery(sql);
  while(rs.next()){//循环取出返回的每一行数据
   int mid=rs.getInt(1);
   String name=rs.getString(2);
   Date birthday=rs.getDate(3);
   int age=rs.getInt(4);
   String note=rs.getString(5);
   System.out.println(mid+","+name+","+birthday+","+age+","+note);
  }
  //第四步：关闭数据库
  rs.close();//此步可以省略，因为Connection接口的conn.close();关闭了也会把Result Set关闭
  stmt.close();//此步可以省略，理由同上。
  conn.close();
 }
}
```

章节56 课时 222 04078_PreparedStatement接口

```
package cn.mldn.demo;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.util.Date;
public class TestDemo{
 private static final String DBDRIVER="oracle.jdbc.dirver.OracleDriver";
 private static final String USER="scott";
 private static final String PASSWORD="tiger";
 private static final String DBURL="jdbc:oracle:thin:@localhost:1521:ORCL";
 public static void main(String[] args)throws Exception {
  String name="Mr'SMITH";
  Date birthday=new Date();
  int age=18;
  String note="是个歪果仁";
  //第一步，加载数据库驱动程序
  Class.forName(DBDRIVER);
  //第二步：连接数据库
  Connection conn=DriverManager.getConnection(DBURL,USER,PASSWORD);
  System.out.println(conn);
  //第三步：进行数据库的数据操作，执行了完整的SQL
  String sql="INSERT INTO member(mid,name,birthday,age,note) VALUES(myseq.nextval,?,?,?,?)";
  PreparedStatement ps=conn.prepareStatement(sql);
  ps.setString(1, name);
  ps.setDate(2, new java.sql.Date(birthday.getTime()));
  ps.setInt(3, age);
  ps.setString(4, note);
  int len=ps.executeUpdate();
  System.out.println("影响的行数："+len);
  //第四步：关闭数据库
  ps.close();//此步可以省略。
  conn.close();
```

}
}

获取结果集元数据！

- 得到元数据：rs.getMetaData()，返回值为 ResultSetMetaData；
- 获取结果集列数：int getColumnCount()
- 获取指定列的列名：String getColumnName(int colIndex)

```
int count = rs.getMetaData().getColumnCount();
while(rs.next()) {//遍历行
    for(int i = 1; i <= count; i++) {//遍历列
        System.out.print(rs.getString(i));
        if(i < count) {
            System.out.print(", ");
        }
    }
    System.out.println();
}
```

```
int count=rs.getMetaData().getColumnCount();
  while(rs.next()){
   for (int i = 0; i < count; i++) {
    Object obj=rs.getObject(i);
    System.out.print(obj);
    if(i<count){
     System.out.print(", ");
    }
   }
   System.out.println();
  }
```