# 目录

# Java 多线程实现

## 线程与进程

Java 是一门为数不多的支持多线程编程的语言。

进程？在操作系统的定义中，进程指的是一次程序的完整运行，这个运行的过程之中内存、处理器、IO 等资源操作都要为这个进程服务。

在最早的 dos 系统的时代，有一个特点：如果你电脑病毒发作了，那么你电脑几乎就不能动了。因为所有的资源都被病毒软件所占用，其他的程序无法抢占资源。但是到了后来 Windows 时代，这一情况发生了改变，电脑即使有病毒，电脑也可以运行（就是慢点）。

Windows 属于多进程的操作系统。但是有一个问题出现了：每一个进程都需要有资源的支持，那么多个进程怎么去分配资源呢？

在同一个时间段上，会有多个进程轮流去抢占资源。但是在某一个时间点上只能有一个进程。

线程？线程是在进程的基础上进一步划分的结果，即：一个进程上可以同时创建多个线程。

线程是比进程更快的处理单元，而且所占的资源也小。那么多线程的应用也是性能最高的应用。

总结：线程的存在离不开进程。进程如果消失后，线程一定会消失。反之线程消失了进程未必会消失。

## Thread 类实现

掌握 java 中 3 种多线程的实现方式（JDK1.5 之后增加了第 3 种）。

如果想在 java 中实现多线程，有两种途径：

1. 继承 Thread 类；
2. 实现 Runable 接口（Callable 接口）。

Thread 类是一个支持多线程的功能类，只要有一个类继承了 Thread 那么它就是一个多线程的类。

```
Class MyThread extends Thread{}
```

所有程序的起点是 main（）方法。**但是所有线程也有自己的起点，即：run（）方法。在多线程的每个主体类之中都必须覆写 Thread 类中所提供的 run（）方法。**

```
public void run(){}
```

这个方法没有返回值，那么也就表示线程一旦开始那么就要一直执行，不能够返回内容。

```
class MyThread extends Thread{
    private String name;//定义属性
    public MyThread(String name){//定义构造方法
        this.name=name;
    }
    @Override
    public void run() {//覆写run方法，作为线程的主体方法
        for(int x=0;x<200;x++){
            System.out.println(this.name+"-------"+x);
        }
    }

}
```

本线程类的功能是进行循环的输出操作。所有的线程跟进程一样的，都必须轮流抢占资源。**所以多线程的执行应该是多个线程彼此交替执行，也就是说如果直接调用了 run（）方法，那么**

**就不能够启动多线程，多线程启动的唯一方法就是 Thread 类中的 start()方法：public void start()，调用此方法执行的方法体是 run（）方法定义的。**

```
package cn.mldn.util;
class MyThread extends Thread{
    private String name;//定义属性
    public MyThread(String name){//定义构造方法
        this.name=name;
    }
    @Override
    public void run() {//覆写run方法，作为线程的主体方法
        for(int x=0;x<100;x++){
            System.out.println(this.name+"-------"+x);
        }
    }

}
public class TestDemo{
    public static void main(String args[]){
        MyThread mt1=new MyThread("线程A");
        MyThread mt2=new MyThread("线程B");
        MyThread mt3=new MyThread("线程C");
        mt1.start();
        mt2.start();
        mt3.start();
    }
}
```

此时每一个线程对象交替执行。

疑问？为什么多线程启动不是调用 run（），而必须调用 start（）？

Thread 的 start（）方法不仅仅要启动多线程的执行代码，还要去根据不同的操作系统进行资源的分配。

## Runnable 接口实现

虽然 Thread 类可以实现多线程的主体类定义，但是它有一个问题，java 具有单继承局限，正因为如此，在任何情况下针对于类的继承都应该是回避的问题，那么多线程也一样，为了解决单继承的限制，在 java 里面专门提供了 Runable 接口：

```
@FunctionalInterface
public interface Runnable{//函数式接口，特征是一个接口只能有一个方法
public void run();
}
```

**在接口里面任何的方法都是 public 定义的权限，不存在默认的权限。**

那么只需要让一个类实现 Runnable 接口，并且也需要覆写 run（）方法。

与继承 Thread 类相比，此时的 MyThread 类在结构上是没有区别的，但是有一点是有严重区别的：如果此时继承了 Thread 类，那么可以直接继承 start（）方法；但如果实现的是 Runnable 接口，并没有 start（）方法可以被继承。

```
class MyThread implements Runnable{
    private String name;//定义属性
    public MyThread(String name){//定义构造方法
```

```
        this.name=name;
    }
    @Override
    public void run() {//覆写run方法，作为线程的主体方法
        for(int x=0;x<100;x++){
            System.out.println(this.name+"-------"+x);
        }
    }

}
```

不管何种情况下，如果要想启动多线程，一定依靠 Thread 类完成，在 Thread 类里定义有如下的构造方法：

public Thread(Runnable target)，接收的是 Runnable 接口对象；

范例：

```
package cn.mldn.util;
class MyThread implements Runnable{
    private String name;//定义属性
    public MyThread(String name){//定义构造方法
        this.name=name;
    }
    @Override
    public void run() {//覆写run方法，作为线程的主体方法
        for(int x=0;x<200;x++){
            System.out.println(this.name+"-------"+x);
        }
    }

}
public class TestDemo{
    public static void main(String args[]){
        MyThread mt1=new MyThread("线程A");
        MyThread mt2=new MyThread("线程B");
        MyThread mt3=new MyThread("线程C");
        Thread m1=new Thread(mt1);
        Thread m2=new Thread(mt2);
        Thread m3=new Thread(mt3);
        m1.start();
        m2.start();
        m3.start();
    }
}
```

此时就避免了单继承局限，那么也就是说在实际工作中使用接口是最合适的。

## 两种实现方式的区别（面试题）

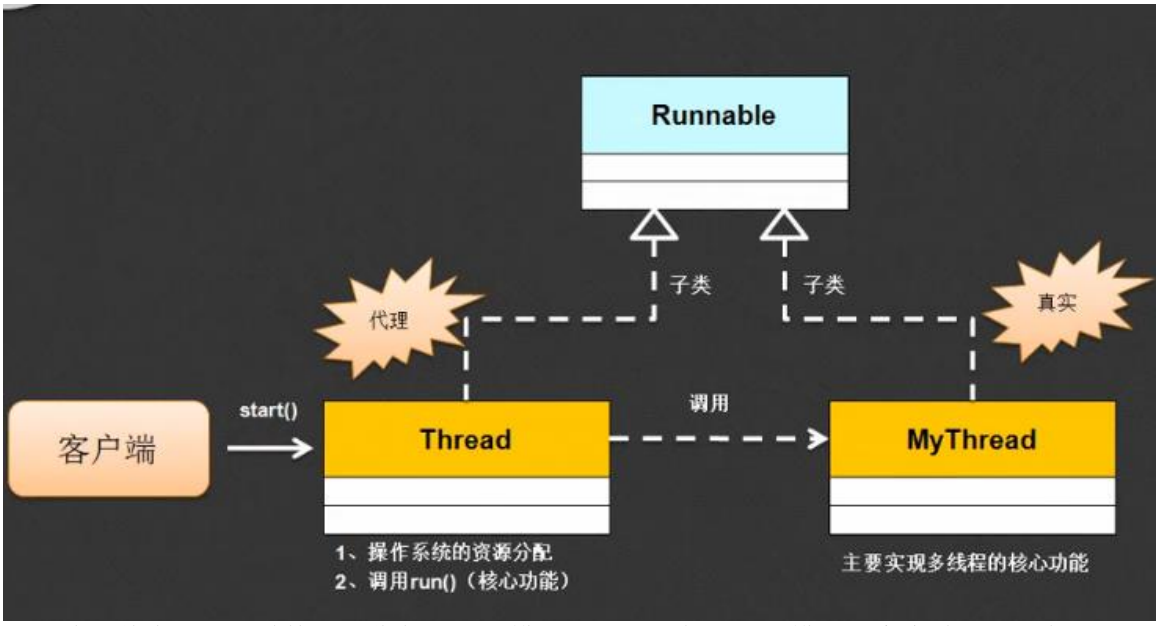
通过讲解已经清楚多线程两种实现方式，这两种方式有哪些区别？

首先我们要明确的是使用 Runnable 接口与 Thread 相比，解决了单继承局限，所以不管后面的区别与联系，至少这一点就已经下了死定义。如果要使用，一定使用 Runnable 接口。

观察 Thread 类定义：

```
public class Thread
extends Object
implements Runnable
```

发现 Thread 类实现了 Runnable 接口。




此时整个的定义结构非常像代理设计模式，如果是代理设计模式，客户端调用的代理类的方法应该是接口里提供的方法，那么也应该是 run（）才对。

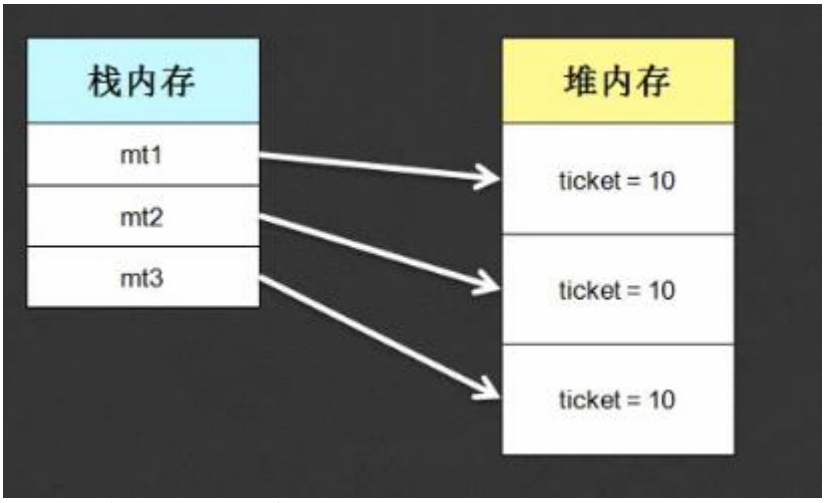
除了以上的联系之外，还有一点：使用 Runnable 接口可以比 Thread 类更好的描述出数据共享这一概念。此时的数据共享指的是多个线程访问同一资源的操作。

范例:观察代码 Thread 实现（每一个线程对象都必须通过 start（）启动）

```
package cn.mldn.util;
class MyThread extends Thread{
    private int ticket=10;//定义属性
    @Override
    public void run() {//覆写run方法，作为线程的主体方法
        for(int x=0;x<100;x++){
            if(this.ticket>0){
            System.out.println("卖票，ticket="+this.ticket--);
            }
        }
    }

}
public class TestDemo{
    public static void main(String args[]){
        //由于Mythread类有start()方法，所以每一个MyThread类就是一个线程对象，可以直接启动
        MyThread mt1=new MyThread();
        MyThread mt2=new MyThread();
        MyThread mt3=new MyThread();
        mt1.start();
        mt2.start();
```

```
        mt3.start();
    }
}
```

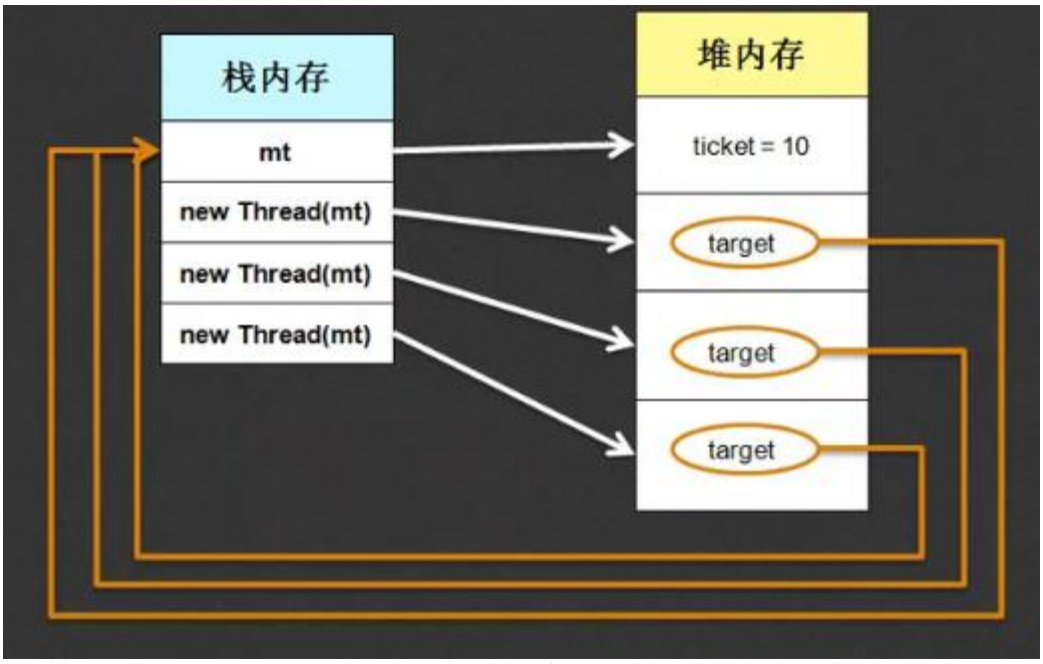
本程序声明了 3 个 MyThread 类对象，并且分别调用了 3 次 start（）方法启动线程对象。结果是每一个线程对象都在卖各自的票。此时并不存在数据共享。




范例：利用 Runnable 实现

```
package cn.mldn.util;
class MyThread implements Runnable{
    private int ticket=10;//定义属性
    @Override
    public void run() {//覆写run方法，作为线程的主体方法
        for(int x=0;x<100;x++){
            if(this.ticket>0){
            System.out.println("卖票，ticket="+this.ticket--);
            }
        }
    }

}
public class TestDemo{
    public static void main(String args[]){
        MyThread mt1=new MyThread();
        new Thread(mt1).start();
        new Thread(mt1).start();
        new Thread(mt1).start();

    }
}
```

此时也属于 3 个线程对象，唯一的区别是这 3 个线程对象都占用了同一个 MyThread 类的对象引用，也就是这 3 个线程对象都直接访问同一个数据资源。




面试题：请解释 Thread 类与 Runnable 接口实现多线程的区别？（请解释多线程两种实现方式的区别？）

Thread 类是 Runnable 接口的子类，使用 Runnable 接口实现可以避免单继承局限；

Runnable 接口实现的多线程可以比 Thread 类实现的多线程更加清楚的描述数据共享的概念。

请写出多线程两种实现操作？

把 Thread 类继承的方式和 Runnable 接口实现的方式都写出来。

## Callable 接口（理解）

使用 Runnable 接口实现的多线程可以避免单继承局限，但是有一个问题，Runnable 接口里面的 run() 方法不能返回操作结果。为了解决这个矛盾，提供了一个新的接口：java.util.concurrent.Callable 接口。

```
public Interface Callable<V>{
    public V call()throws Exception;
}
```

Call（）方法完成线程的主体功能后可以返回一个结果，而这个结果的类型由 Callable 接口的泛型决定。

范例：定义一个线程主体类

```
import java.util.concurrent.Callable;

class MyThread implements Callable<String>{
   private int ticket=10;//定义属性
   @Override
   public String call() throws Exception{
      for(int x=0;x<100;x++){
         if(this.ticket>0){
         System.out.println("卖票，ticket="+this.ticket--);
         }
      }
      return "票已卖光！！！";
```

```
    }

}
```

此时观察 Thread 类里面并没有直接支持 Callable 接口的多线程应用。

从 JDK1.5 开始提供有 java.util.concurrent.FutureTask<V>类。这个类主要是负责 Callable 接口对象的操作的，这个接口的定义结构：

```
public class FutureTask<V> extends Object implements RunnableFuture<V>
```

```
public interface RunnableFuture<V> extends Runnable, Future<V>
```

FutureTask 有如下的构造方法：FutureTask(Callable<V> callable)。

接收的目的只有一个，那么就是去的 call（）方法的返回结果。

```
package cn.mldn.util;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
class MyThread implements Callable<String>{
    private int ticket=10;//定义属性
    @Override
    public String call() throws Exception{
        for(int x=0;x<100;x++){
            if(this.ticket>0){
            System.out.println("卖票，ticket="+this.ticket--);
            }
        }
        return "票已卖光！！！";
    }

}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt1=new MyThread();
        MyThread mt2=new MyThread();
        FutureTask<String> fu1=new FutureTask<String>(mt1);//目的是取得call（）的返回结果
        FutureTask<String> fu2=new FutureTask<String>(mt2);//目的是取得call（）的返回结果
        new Thread(fu1).start();//启动多线程
        new Thread(fu2).start();//启动多线程
        //多线程执行完毕后可以取得内容，依靠FutureTask的父接口Future中的get()方法完成
        System.out.println("A线程的返回结果"+fu1.get());
        System.out.println("B线程的返回结果"+fu2.get());
    }
}
```

第三种方法最麻烦的在于需要接收返回值信息，并且又要与原始的多线程的实现靠拢（向 Thread 类靠拢）。

**总结：**

1. **对于多线程的实现重点在于 Runnable 接口与 Thread 类启动的配合上。**
2. **对于 JDK1.5 的新特性了解就行，知道区别在于返回结果上。**

# 线程常用操作方法

多线程里有很多方法定义，但是大部分方法都在 Thread 里定义。强调几个与开发有关的方法。

## 命名和取得

所有的线程程序的执行，每一次都是不同的结果，因为它会根据自己的情况进行资源的抢占，所以如果要想区分每一个线程，那么就必须依靠线程的名字。对于线程名字一般而言会在启动之前定义，不建议对已经启动的线程进行更改名称、或者为不同的线程设置重名。

如果想进行线程名称的操作，可以使用 Thread 类如下的方法：

**构造方法：public Thread(Runnable target,String name);**

**设置名字：public final void setName(String name);（final 表示不能覆写）**

**取得名字：public final String getName();**

对于线程名字的操作会出现一个问题，这些方法是属于 Thread 类的，可是如果回到线程类（Runnable 子类），这个类并没有继承 Thread 类，如果要想取得线程名字，那么能够取得的就是执行本方法的线程名字。所以在 Thread 类里面提供有一个方法：

**取得当前线程对象：public static Thread currentThread()。**

范例：观察线程的命名

```
package cn.mldn.util;
class MyThread implements Runnable{
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        new Thread(mt).start();
        new Thread(mt).start();
        new Thread(mt).start();
    }
}
```

如果再实例化 Thread 类对象的时候没有为其设置名字，那么会自动的进行编号命名，保证线程对象的名字不重复。

范例：观察设置名字

```
package cn.mldn.util;
class MyThread implements Runnable{
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        new Thread(mt,"自己的线程A").start();
        new Thread(mt).start();
        new Thread(mt,"自己的线程B").start();
```

```
    }
}
```

观察：

```
package cn.mldn.util;
class MyThread implements Runnable{
    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        new Thread(mt,"自己的线程").start();
        mt.run();
    }
}
```

结果：

```
main
自己的线程
```

原来主方法就是一个线程：main 线程。

通过以上的代码发现：线程其实一直都存在，可是进程去哪里了？

每当使用 java 命令去解释一个程序类的时候，对于操作系统而言，都相当于启动了一个新的进程，而 main 只是新进程上的一个子线程而已。

**提问：每个 JVM 进程启动的时候至少启动几个线程呢？**

1. main 线程:程序的主要执行，以及启动子线程；
2. gc 线程：负责垃圾收集。

## 线程的休眠

所谓线程休眠，指的是让线程执行的速度变慢一点，休眠方法：

public static void sleep(long millis) throws InterruptedException

long：用来表示日期、时间与文件大小。

范例：观察休眠的特点

```
package cn.mldn.util;
class MyThread implements Runnable{
    @Override
    public void run(){
        for(int x=0;x<10000;x++){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

    System.out.println(Thread.currentThread().getName()+",X="+x);
        }
    }
}
```

```
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        new Thread(mt,"自己的线程").start();
    }
}
```

因为每一次执行到 run（）方法的线程对象都会休眠，所以执行的速度就会变慢。
默认情况下在休眠的时候如果设置了多个线程对象，那么所有的线程对象将会一起进入到 run（）方法（所谓一起进入因为先后顺序实在太短，肉眼忽略）。

## 线程优先级

所谓优先级指的是越高的优先级越有可能先执行。在 Thread 类里提供有以下两个方法进行优先级的操作：

设置优先级：public final void setPriority(int newPriority)；
取得优先级：public final int getPriority()。

发现设置和取得优先级都是使用 int 型数据，对于此内容有 3 种取值：

最高优先级：public static final int MAX_PRIORITY，10；
中等优先级：public static final int NORM_PRIORITY，5；
最低优先级：public static final int MIN_PRIORITY，1。

范例：

```
package cn.mldn.util;
class MyThread implements Runnable{
    @Override
    public void run(){
        for(int x=0;x<20;x++){
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

    System.out.println(Thread.currentThread().getName()+",X="+x);
        }
    }
}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        Thread mt1=new Thread(mt,"自己的线程A");
        Thread mt2=new Thread(mt,"自己的线程B");
        Thread mt3=new Thread(mt,"自己的线程C");
        mt3.setPriority(Thread.MAX_PRIORITY);
        mt1.start();
        mt2.start();
        mt3.start();
    }
}
```

范例：主线程优先级是多少？

```
public class TestDemo{
    public static void main(String args[]) throws Exception{
        System.out.println(Thread.currentThread().getPriority());
    }
}
```

结果：5

所以主线程是中等优先级。

总结：

1. **Thread.currentThread()可以取得当前线程类对象；**
2. Thread.sleep()主要是休眠，如果设置多个线程对象的话，感觉是一起休眠，但实际是由先后顺序的。
3. 清楚优先级越高的线程对象越有可能先执行。

# 线程的同步与死锁（了解）

线程的同步产生原因
线程的同步处理操作
线程的死锁

## 同步问题引出

所谓同步指的是多个线程访问同一资源时所需要考虑到的问题。

范例：观察非同步情况下的操作

```
package cn.mldn.util;
class MyThread implements Runnable{
    public int ticket=5;
    @Override
    public void run(){
        for(int x=0;x<20;x++){
            if(this.ticket>0){
                System.out.println(Thread.currentThread().getName()+",
卖票，ticket="+this.ticket--);

            }
        }
    }
}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        new Thread(mt,"票贩子A").start();
        new Thread(mt,"票贩子B").start();
        new Thread(mt,"票贩子C").start();
        new Thread(mt,"票贩子D").start();
    }
}
```

此时 4 个线程对象一起出售 5 张票。此时没有出现问题是因为在一个 JVM 下运行，并且没有

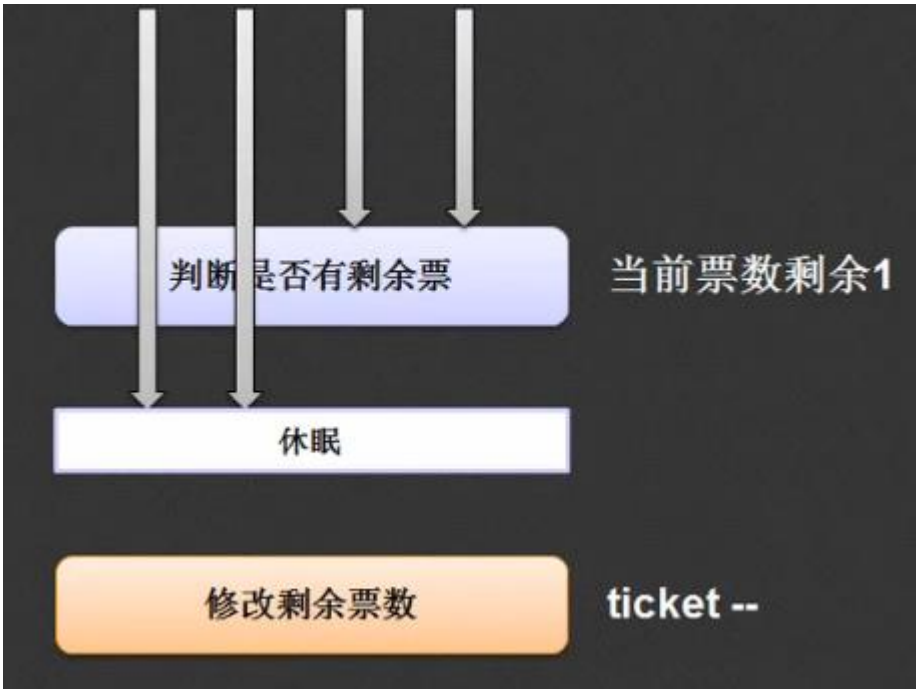
受到任何影响。

```
class MyThread implements Runnable{
    public int ticket=5;
    @Override
    public void run(){
        for(int x=0;x<20;x++){
            if(this.ticket>0){
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()+",
卖票，ticket="+this.ticket--);

            }
        }
    }
}
```

增加延迟之后发现执行的结果出现了负数，那么着就叫不同步的情况，到底如何造成不同步呢？整个卖票的步骤分为 2 步：

第一步：判断是否还有剩余票数；

第二步：减少剩余票数。

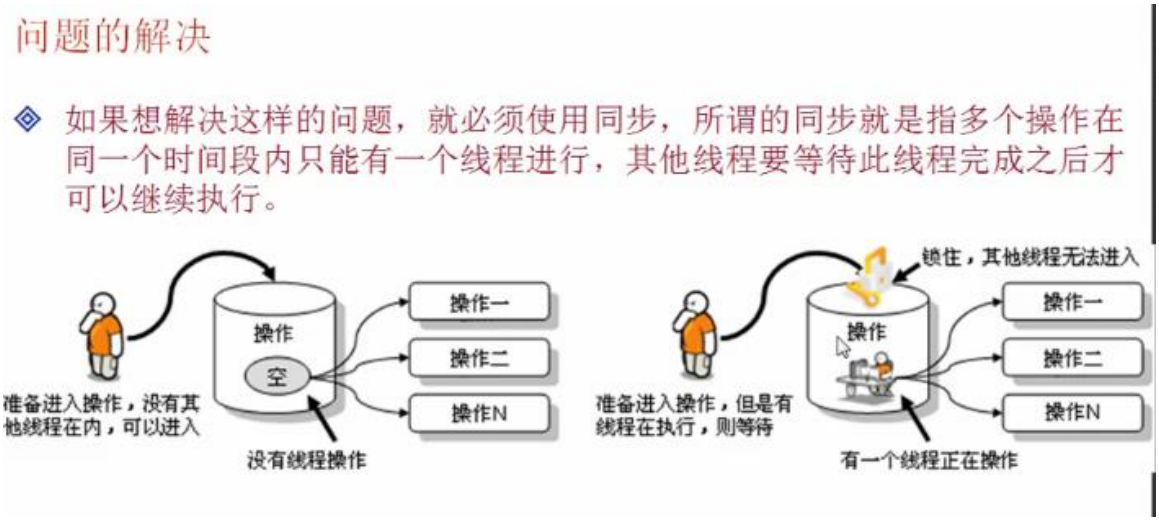



目前互联网上的应用都属于异步。异步操作就会存在数据的安全隐患。

## 同步操作

通过观察可以发现以上程序发生的最大问题是：判断和修改数据是分开完成的，即：某几个线程可以同时执行这些功能。


问题的解决：




在 java 里面如果想要实现线程的同步可以使用 **synchronized** 关键字。而这个关键字可以通过两种方式使用：

第一种：同步代码块；

第二种：同步方法。

在 java 里面有四种代码块：普通代码块、构造块、静态块和同步块。

范例：观察同步块。

```
package cn.mldn.util;
class MyThread implements Runnable{
    public int ticket=54;
    @Override
    public void run(){
        for(int x=0;x<200;x++){
```

```
            synchronized(this){//当前操作每次只允许一个对象进入
                if(this.ticket>0){
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

    System.out.println(Thread.currentThread().getName()+",卖票,
ticket="+this.ticket--);
                }
            }
        }
    }
}
public class TestDemo{
    public static void main(String args[]) throws Exception{
        MyThread mt=new MyThread();
        new Thread(mt,"票贩子A").start();
        new Thread(mt,"票贩子B").start();
        new Thread(mt,"票贩子C").start();
        new Thread(mt,"票贩子D").start();
    }
}
```

范例：观察同步方法解决

```
class MyThread implements Runnable{
    public int ticket=54;
    @Override
    public void run(){
        for(int x=0;x<200;x++){
                this.sale();//调用同步方法
        }
    }
    public synchronized void sale() {
        if(this.ticket>0){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName()+",卖
票，ticket="+this.ticket--);
        }
    }

}
```

同步操作与同步操作相比，异步操作的执行速度要高于同步操作，但是同步操作的数据安全性更高，属于安全的线程操作。

## 死锁

通过分析可以发现，所谓同步指的是一个线程对象等待另一个线程对象执行完毕后的操作形式。线程同步过多就有可能造成死锁。

范例：没有任何意义，仅用观察。

以上的代码只是为了说明死锁而举的例子，死锁是程序开发之中由于某种逻辑上的错误所造成的问题，并且不会简单就出现的。

**面试题：请解释多个线程访问同一资源时需要考虑哪些情况？有可能带来哪些问题？**

**多个线程访问同一资源时一定处理好同步，可以使用同步代码块或同步方法解决；**

**同步代码块:synchronized(同步对象){代码};**

**同步方法：public synchronized 返回值 方法名称(){代码};**

**但是过多的使用同步有可能造成死锁。**

总结：

1. 最简单的理解同步和异步操作那么就可以通过：synchronized 来实现；
2. 死锁是一种不定的状态。

## “生产者-消费者”模型-多线程操作经典案例

### 问题引出

生产者和消费者指的是两种不同的线程类对象，操作同一资源的情况，具体的操作流程如下：

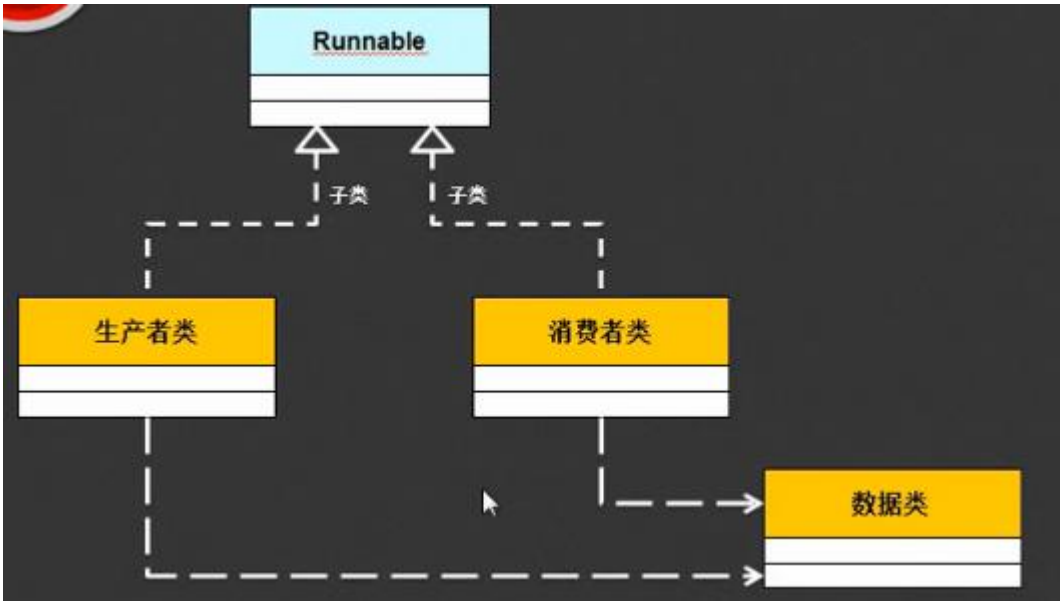
生产者负责生产数据，消费者负责取走数据；

生产者每生产完一组数据后，消费者就要取走一组数据。

那么假设要生产的数据如下：

第一组：title=王惊雷，content=好学生一枚；

第二组：title=可爱的萌动物，content=草泥马。




范例：程序基本模型。

```
package cn.mldn.util;
class Info{
   private String title;
```

```
    private String content;
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public void setContent(String content) {
        this.content = content;
    }
    public String getContent() {
        return content;
    }
}
class Productor implements Runnable{
    private Info info;
    public Productor(Info info){
        this.info=info;
    }
    @Override
    public void run(){
    for(int x=0;x<100;x++){
    if(x%2==0){
        this.info.setTitle("王惊雷");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        this.info.setContent("好学生一枚！");
    }
    else{
        this.info.setTitle("可爱的萌动物");
        try {
            //Thread.sleep(100);
            Thread.sleep(200);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        this.info.setContent("草泥马！");
    }
    }
    }
}
class Customer implements Runnable{
    private Info info;
    public Customer(Info info){
        this.info=info;
    }
```

```
        @Override
        public void run() {
            for(int x=0;x<100;x++){
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

    System.out.println(this.info.getTitle()+"-"+this.info.getConten
t());
            }
        }

    }
    public class TestDemo{
        public static void main(String args[]){
            Info info=new Info();
            new Thread(new Productor(info)).start();
            new Thread(new Customer(info)).start();

        }
    }
```

现在实际上通过以上的代码可以发现两个严重的问题：

- 数据错位，发现不再是一个所需要的完整数据；
- 数据重复取出，数据重复设置。

## 解决数据错乱问题-同步处理

数据的错位完全是因为非同步的操作所造成的，所以应该使用同步处理。因为取出和设置是两个不同的操作。要想进行同步控制，那么就要将其定义在一个类里面。

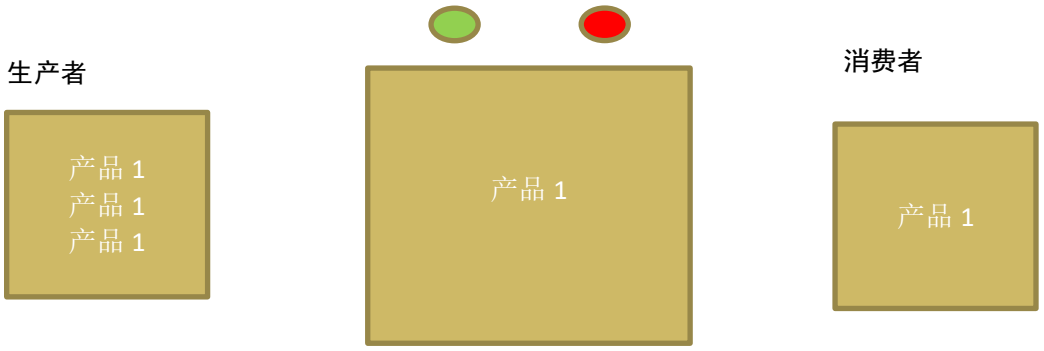
代码如下：

```
    package cn.mldn.util;
    class Info{
        private String title;
        private String content;
        public synchronized void set(String title,String content){
            this.title=title;
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            this.content=content;
        }
        public synchronized void get(){
            try {
                Thread.sleep(200);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(this.title+"-"+this.content);
        }
    }
    class Productor implements Runnable{
        private Info info;
        public Productor(Info info){
            this.info=info;
        }
        @Override
        public void run(){
        for(int x=0;x<100;x++){
        if(x%2==0){
            this.info.set("王惊雷","好学生一枚！");
        }
        else{
            this.info.set("可爱的萌动物","草泥马！");
        }
        }
        }
    }
    class Customer implements Runnable{
        private Info info;
        public Customer(Info info){
            this.info=info;
        }
        @Override
        public void run() {
            for(int x=0;x<100;x++){
                this.info.get();
            }
        }

    }
    public class TestDemo{
        public static void main(String args[]){
            Info info=new Info();
            new Thread(new Productor(info)).start();
            new Thread(new Customer(info)).start();

        }
    }
```

此时数据的错位问题很好的得到了解决，但是重复操作问题更加严重。

## 解决重复问题-Object 类支持




中间的公共区域的指示灯：

绿灯，可以生产。生产后变为红灯，表示已经生产过了但是还没取出，消费者可以取出，红灯不能生产。

消费者取出后再次变为绿灯，可以生产。

如果要想实现整个代码的操作，必须加入等待与唤醒机制。在 Object 类里面存在有专门的处理方法。

- 等待：public final void wait() throws InterruptedException
- 唤醒第一个等待线程：public final void notify()
- 唤醒第全部等待线程，哪个优先级高就先执行：public final void notifyAll()

范例：只是在 Info 类里加入了等待机制：

```
package cn.mldn.util;
class Info{
    private String title;
    private String content;
    private boolean flag=true;
    //flag=true 表示可以生产但是不可以取出
    //flag=false 表示可以取出但是不可以生产
    public synchronized void set(String title,String content){
        if(this.flag==false){
            try {
                wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        this.title=title;
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.content=content;
        this.flag=false;
        super.notify();
    }
    public synchronized void get(){
        if(this.flag==true){
```

```
            try {
                wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(this.title+"-"+this.content);
        this.flag=true;
        super.notify();
    }
}
class Productor implements Runnable{
    private Info info;
    public Productor(Info info){
        this.info=info;
    }
    @Override
    public void run(){
    for(int x=0;x<100;x++){
    if(x%2==0){
        this.info.set("王惊雷","好学生一枚！");
    }
    else{
        this.info.set("可爱的萌动物","草泥马！");
    }
    }
    }
}
class Customer implements Runnable{
    private Info info;
    public Customer(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            this.info.get();
        }
    }

}
public class TestDemo{
    public static void main(String args[]){
        Info info=new Info();
        new Thread(new Productor(info)).start();
```

```
            new Thread(new Customer(info)).start();

        }
    }
```

现在可以实现生产一个取走一个，但是速度有点慢。

**面试题：请解释 sleep（）与 wait（）的区别？**

**Sleep（）是 Thread 类定义的方法，而 wait（）是 Object 类定义的方法。**

**Sleep（）可以设置休眠时间，时间一到自动唤醒，而 wait（）需要 notify（）唤醒。**

**总结：**

**这是一个非常经典的多线程的处理模型，掌握它是个人能力的提升，同时可以更加理解 Object 类的应用。**

# Java 基础类库

# StringBuffer 类

预计知识点：

1. StringBuffer 类的主要特点。
2. Stringbuffer,StringBuilder,String 的区别。

【String 类的特点】：

- String 类对象有两种实例化方法：

  | -直接赋值：只开辟一块堆内存空间，可以直接入池。

  | -构造方法：开辟两块堆内存空间，不会直接入池,使用 intern（）手工入池。
- 任何一个字符串都是 String 类的匿名对象。
- 字符串一旦声明则不可改变，可以改变的只是 String 类对象的引用。

虽然在所有的项目里面，String 类都要使用，可是 String 类有一个问题不能不重视，String 类不能被修改。为此在 java 里存在有另一个类—StringBuffer 类（里面的内容可以改变）。

String 类的对象可以使用的“+”进行字符串的操作，但是 StringBuffer 类里面必须使用 append（）方法进行追加：public StringBuffer append(数据类型 变量)

范例：观察 StringBuffer 类的基本使用：

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
      //String类可以直接赋值实例化，但是StringBuffer类不行。
      StringBuffer buf=new StringBuffer();
      buf.append("Hello").append(" World").append(" !!!");
      fun(buf);
      System.out.println(buf);
  }
  public static StringBuffer fun(StringBuffer temp){
      temp.append("\n").append("Hello mldn!!!");
      return temp;
  }
}
```

发现 StringBuffer 类的内容是可以修改的，而 String 类的内容是不能修改的。

清楚了 StringBuffer 类的基本使用与基本操作之后，现在来看一下这两个类的定义情况。

| String 类 | StringBuffer 类 |
|---|---|

| | |
|---|---|
| public final class String<br>extends Object<br>implements Serializable,<br>Comparable<String>, CharSequence | public final class StringBuffer<br>extends Object<br>implements Serializable, CharSequence |

发现 String 类与 StringBuffer 类都是 CharSequence 接口的子类。而在以后的开发中，如果你看到某些方法的操作上出现的是 CharSequence 接口，那么应该立刻清楚只需要传递字符串即可。

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     CharSequence seq="Hello";//向上转型
     System.out.println(seq);//String类覆写的toString（）
  }
}
```

虽然 String 和 StringBuffer 有着共同的接口，但是这两个类的对象之间如果要转换不能直接转换：

1. 将 String 转换为 Stringbuffer：
   a) 利用 StringBuffer 类 的构造方法完成：public StringBuffer(String str)

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
        StringBuffer buf=new StringBuffer("Hello");
           System.out.println(buf);
        }
     }
```

   b) 利用 append()方法：public StringBuffer append(String str)

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     StringBuffer buf=new StringBuffer();
     buf.append("Hello");
        System.out.println(buf);
  }
   }
```

2. 将 StringBuffer 类变为 String 类：
   a) 利用 toString（ ）可以讲 StringBuffer 转换为 String：

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     StringBuffer buf=new StringBuffer("Hello");
     String str=buf.toString();
     System.out.println(str);
  }
}
```

b) 也可以利用 String 类的构造方法：public String(StringBuffer buffer)

c) 在 String 类里面也存在一个和 StringBuffer 类比较的方法：

public boolean contentEquals(StringBuffer sb)

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     StringBuffer buf=new StringBuffer("Hello");
     System.out.println("Hello".contentEquals(buf));
  }
}
```

StringBuffer 类里也定义了许多方法，有些正好和 String 类互补：

1.字符串反转：public StringBuffer reverse()

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     StringBuffer buf=new StringBuffer("Hello");
     System.out.println(buf.reverse());
  }
}
```

2. 在指定的索引位置增加数据：public StringBuffer insert(int offset,数据类型 变量)

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     StringBuffer buf=new StringBuffer("Hello");
     buf.insert(0,"MLDN ").insert(0,"你好 ");
     System.out.println(buf);
  }
}
```

（这个操作 String 类没有）

3.删除部分数据：public StringBuffer delete(int start,int end)

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     StringBuffer buf=new StringBuffer("Hello World MLDN");
     buf.delete(5,11);
     System.out.println(buf);
  }
}
```

从 JDK1.5 之后增加了一个新的 String 操作类，StringBuilder

定义结构：

```
public final class StringBuilder
extends Object
```

```
implements Serializable, CharSequence
```

发现 StringBuffer 与 StringBuilder 在定义上非常相似，几乎连方法也一样。

面试题：请解释 String、StringBuffer、StringBuilder 的区别？

- String 类的内容一旦声明则不可改变，StringBuffer 和 StringBuilder 的内容可以改变。
- StringBuffer 类中提供的方法都是同步方法，属于安全的线程操作；而 Stringbuilder 类中的方法都是异步方法，属于非线程安全的操作。

日后在开发中见到字符串的应用，不需要思考 95%使用 String 类，只有在需要频繁修改的时候才需要使用 StringBuffer 或者 StringBuilder。

# Runtime 类（了解）

在每一个 JVM 进程里面都会有一个 Runtime 类的对象，这个类的主要功能是取得一些与运行时有关的环境的属性或者创建新的进程等操作。

在 Runtime 类定义的时候他的构造方法就已经被私有化了，这就属于单例设计模式的应用，因为要保证在整个进程里只有一个 Runtime 类的对象。所以在 Runtime 类里面提供了一个 static 型的方法，用于取得 Runtime 类的实例化对象：public static Runtime getRuntime()

Runtime 类是直接与本地运行有关的所有属性的集合，所以在 Runtime 类里面定义有如下的方法：

- 返回所有可用内存空间：public long totalMemory()
- 返回最大可用内存：public long maxMemory()
- 返回空余内存空间：public long freeMemory() （注意 long 返回的是字节）

**范例**：观察内存大小

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     Runtime run=Runtime.getRuntime();
     System.out.println("max="+run.maxMemory());
     System.out.println("total="+run.totalMemory());
     System.out.println("free="+run.freeMemory());
     String str="";
     for(int x=0;x<9000;x++){
        str+=x;
     }
     System.out.println("max="+run.maxMemory());
     System.out.println("total="+run.totalMemory());
     System.out.println("free="+run.freeMemory());
  }
}
```

一旦产生过多垃圾后，就会改变可用的内存空间大小。

可是在 Runtime 类里面有一个方法：可以释放掉垃圾空间：public void gc()

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
     Runtime run=Runtime.getRuntime();
```

```
        System.out.println("max="+run.maxMemory());
        System.out.println("total="+run.totalMemory());
        System.out.println("free="+run.freeMemory());
        String str="";
        for(int x=0;x<9000;x++){
            str+=x;
        }
        System.out.println("max="+run.maxMemory());
        System.out.println("total="+run.totalMemory());
        System.out.println("free="+run.freeMemory());
        run.gc();//释放垃圾空间
        System.out.println("max="+run.maxMemory());
        System.out.println("total="+run.totalMemory());
        System.out.println("free="+run.freeMemory());
    }
}
```

**面试题：**请解释什么叫 GC，如何处理？

- GC（Garbage Collector）垃圾收集器，指的是释放无用的内存空间；
- GC 会由系统不定期的进行自动的回收，或者调用 Runtime 类中的 gc（）方法手工回收。

实际上 Runtime 类还有一个更加有意思的功能，就是他可以调用本机的可执行程序，并且创建进程。

执行程序：public Process exec(String command) throws IOException

范例：执行进程（看一下）

```
package cn.mldn.util;
import java.io.IOException;
public class TestDemo{
  public static void main(String args[]) throws Exception{
      Runtime run=Runtime.getRuntime();
      Process  pro = run.exec("mspaint.exe");
      Thread.sleep(2000);
      pro.destroy();
  }
}
```

这样的操作一般意义不大。

**总结：**

1. **Runtime 类使用了单例设计模式，所以必须通过内部的 getRuntime（）方法才可以取得 Runtime 类对象。**
2. **Runtime 类提供了 gc（）方法，可以用于手工释放内存。**

# System 类（了解）

**本次讲解预计知识点：**

1. 如何计算某个代码的执行时间；
2. 进行垃圾收集操作。

**具体内容（了解）**

之前一直使用的“System.out.println()”就属于 System 类的操作，只不过这个功能由于牵扯到 IO 部分，所以以后再讲。

在 Sysytem 类里面之前也使用过 System.Arraycopy()方法实现数组拷贝，而这个方法的真实定义如下：

方法拷贝：public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

在 System 类里面定义有一个重要的方法：

取得当前的系统时间：public static long currentTimeMillis()

**范例：**请统计出某项操作的执行时间。

```
package cn.mldn.util;
import java.io.IOException;
public class TestDemo{
  public static void main(String args[]) throws Exception{
     Long start=System.currentTimeMillis();
     String str="";
     for(int x=0;x<30000;x++){
        str+=x;
     }
     Long end=System.currentTimeMillis();
     System.out.println("本次操作执行时间："+(end-start));
  }
}
```

要想统计出所花费的毫秒时间，就使用 Long 型数据直接进行数学计算得来。

在 System 里定义了一个操作方法：public static void gc()，这个 gc（）方法并不是一个新定义的方法，而是间接调用了 Runtime 类中的 gc（）方法。

对象产生一定会调用构造方法，可以进行一些处理的操作，但是某一个对象如果要被回收了，那么连一个收尾的机会都没有。如果需要给对象一个收尾的机会，那么就需要覆写 finalize（）方法：protected void finalize() throws Throwable，在对象回收时就算抛出任何异常也不会影响整个程序的正常执行。

```
package cn.mldn.util;
class Member{
  public Member(){
     System.out.println("噼里啪啦，祸害诞生了！");
  }
  @Override
  protected void finalize() throws Throwable {
     System.out.println("祸害死了，全世界欢庆！");
     throw new Exception("老子十八年后继续祸害全世界，然后自己死了！");
  }
}
public class TestDemo{
  public static void main(String args[]){
     Member me=new Member();
```

```
        me=null;
        System.gc();
        }
}
```

构造方法是留给对象初始化使用的，而 finalize（）方法时留给对象回收使用的。

**面试题：**请解释 final、finally 与 finalize 的区别?

Final 定义不能被继承的类、不能被覆写的方法和常量。

Finally：异常的统一出口。

Finalize：这是一个 Object 类提供的方法：protected void finalize() throws Throwable，指的是对象回收前的收尾方法，即使出现了异常也不会使整个程序中断执行。

**总结：**

1. System 类可以使用 currentTimeMillies（）取得当前的系统时间。
2. System 类的 gc（）方法，相当于调用了“Runtime.getruntime（）.gc()”

# 对象克隆（了解）

对象克隆本身不重要，主要是清楚对象克隆的操作结构，巩固接口的作用。

对象克隆指的是对象的复制操作。

对象克隆：protected Object clone() throws CloneNotSupportedException

此方法上抛出一个“cloneNotSupportedException”异常，如果要使用对象克隆的类没有实现叫 Cloneable 接口，那么就会抛出此异常。但是 Cloneable 接口没有方法，此为标识接口，表示一种操作能力。

**范例：对象克隆**

```
package cn.mldn.util;
class Book implements Cloneable{
  private String title;
  private double price;
  public Book(String title,double price) {
      this.title=title;
      this.price=price;
  }
  @Override
  public String toString() {
      return "书名: "+this.title+", 价格: "+this.price;
  }
  //由于此处需要对象克隆操作，所以才需要进行方法的覆写
  @Override
  public Object clone() throws CloneNotSupportedException {
      return super.clone();//调用父类的克隆方法
  }
  public double getPrice() {
      return price;
  }public String getTitle() {
      return title;
  }public void setPrice(double price) {
      this.price = price;
  }public void setTitle(String title) {
      this.title = title;
```

```
    }
}
public class TestDemo{
  public static void main(String args[]) throws Exception{
      Book ba=new Book("java开发",79.8);
      Book bb=(Book)ba.clone();
      bb.setTitle("Jsp开发");
      System.out.println(ba);
      System.out.println(bb);
      }
}
```

对象克隆的理论价值大于实际价值，因为在实际的工作里面很少会用到克隆的操作。重点在于标识接口上，以后依然会见到没有方法的接口，这样的接口就好比通行证，表示的是能力。

**总结：**标识接口，没有任何方法定义，只是一个接口的声明。

## 数字操作类

本次主要讲解数学操作类的使用。

1. Math 类
2. Random 类
3. 大数字操作类

## Math 类

Math 就是一个专门进行数学计算的类，里面提供了一系列的数学计算方法。

在 Math 类里提供的方法都是 static 方法，因为 Math 类里面没有普通属性。

在整个 Math 类里面，实际上只有一个方法能够引起我们的注意：

- 四舍五入：public static long round（double a）

**范例：**观察四舍五入：（有可能会出现面试题）

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

      System.out.println(Math.round(15.5));
      System.out.println(Math.round(-15.5));
      System.out.println(Math.round(-15.51));
      }
}
```

如果进行负数四舍五入的时候，操作的数据小数大于 0.5 才进位小于等于 0.5 不进位。

## Random 类

这个类的主要功能是取得随机数的操作类。

**范例：**产生 10 个不大于 100 的正整数（0~99）。

```
package cn.mldn.util;
import java.util.Random;
public class TestDemo{
```

```
    public static void main(String args[]) throws Exception{
        Random an=new Random();
        for(int x=0;x<10;x++){
            System.out.println(an.nextInt(100)+"、");
        }
    }
}
```

既然 Random 可以产生随机数，下面就实现 36 选 7 的功能。

最大值是 36，所以设置的边界值是 37，并且里面不能有 0 或者重复的数据。

**范例：**36 选 7 的程序

```
package cn.mldn.util;
import java.util.Random;
public class TestDemo{
    public static void main(String args[]){
        Random rand=new Random();
        int[] data=new int[7];
        int foot=0;//脚标
        while(foot<7){//不知道多少次循环可以保存完整数据，所以使用while循环
            int t=rand.nextInt(37);//产生一个不大于36的随机数
            if(!isReapt(data,t)){
                data[foot]=t;
                foot++;
            }
        }
        java.util.Arrays.sort(data);//排序
        for(int x=0;x<data.length;x++){
            System.out.print(data[x]+"、");
        }
    }
    /*
     * 此方法主要判断是否有重复的内容
     * @temp 指的是已经保存的数据
     * @num 新生成的数据
     * @return 如果存在返回true，否则返回false
     */
    public static boolean isReapt (int temp[],int num){
        if(num==0){
            return true;
        }
        for(int x=0;x<temp.length;x++){
            if(num==temp[x]){
                return true;//表示后面的数据不再进行判断了
            }
        }
        return false;

    }
}
```

在很多的开发之中随机数都一定会有。

## BigInteger 大整数操作类

如果说现在要操作的数据值很大，首先想到的是 double，如果计算的结果超过了 double 呢？

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[]){
  System.out.println(Double.MAX_VALUE*Double.MAX_VALUE);//Infinit
y
  }
}
```

现在发现此时的计算结果并不存在，因为已经超过了 double 的范围。

**面试题：**假设现在又两个很大的数字进行数据计算（超过了 double 的范围），你该怎么做？

如果真的超过了 double 的范围，肯定不能使用 double 进行保存数据，只能使用 String 来保存数据，如果很大的数据要进行数学计算，只能将其变为 String 型，而后按位取出每一个字符保存的数据，进行手工的计算。

所以在 java 里面考虑到了此类情况，所以专门进行了大数字的操作类，其中就有 BigInteger 和 BigDecimal。

BigInteger 构造方法：public BigInteger(String val)，它接收的是 String 型。

```
package cn.mldn.util;
import java.math.BigInteger;
public class TestDemo{
  public static void main(String args[]){
      BigInteger biga=new BigInteger("7878798979");
      BigInteger bigb=new BigInteger("2132454578");
      System.out.println("加法操作："+(biga.add(bigb)));
      System.out.println("减法操作："+(biga.subtract(bigb)));
      System.out.println("乘法操作："+(biga.multiply(bigb)));
      System.out.println("除法操作："+(biga.divide(bigb)));
      BigInteger resault[]=biga.divideAndRemainder(bigb);
      System.out.println("商："+resault[0]+",余数："+resault[1]);
  }
}
```

在 java 里面虽然提供了大数据操作类，很多时候我们的项目开发可能对数字要求更加敏感，而这个时候 java 本身所提供的数字类帮助不大。

## BigDecimal 大浮点数操作类（重要）

BigInteger 不能保存小数，而 BigDecimal 可以保存小数。在 BigDecimal 里提供有如下的构造：

构造一: public BigDecimal(String val)

构造二：public BigDecimal(double val)

与 BigInteger 一样，BigDecimal 也支持基本的数学计算，可是我们使用 BigDecimal 还有一个非常重要的目的，即利用它可以准确的实现四舍五入操作。

我们之前使用过 Math.round（）实现过四舍五入的操作，但是这样的操作有一个问题，所有的小数位都四舍五入了。那么假设有一家公司的年收入按亿进行计算，今年收入：3.45678 亿，按照 Math.round()的做法，相当于只有 3 亿。

遗憾的是 BigDecimal 并没有直接提供四舍五入的支持，可是我们可以利用除法实现：

除法操作：public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)

|-BigDecimal divisor：被除数

|- int scale：保留的小数位

|- int roundingMode：进位模式（public static final int ROUND_HALF_UP）

**范例：**实现准确的四舍五入

```
package cn.mldn.util;
import java.math.BigDecimal;
class MyMath{
  /*实现准确位数的四舍五入操作
   * @num 要进行四舍五入的操作数
   * @scale 要保留的小位数
   * return 处理后的四舍五入数
   */
  public static double round(double num,int scale){
      BigDecimal biga=new BigDecimal(num);
      BigDecimal bigb=new BigDecimal(1);
      return
biga.divide(bigb,scale,BigDecimal.ROUND_HALF_UP).doubleValue();

  }
}
public class TestDemo{
  public static void main(String args[]){
      System.out.println(MyMath.round(-15.5, 0));
      System.out.println(MyMath.round(15.5, 0));
  }
}
```

此类的操作功能在日后的开发之中一定要会使用，属于工具类的范围。

**总结：**

1.Math 类重点要清楚 round（）方法的坑。

2.Random 类生成随机数。

3.如果数据量大就是要 BigInteger 或 BigDecimal，这两个类是 Number 的子类。

# 日期操作类

预计知识点：

1. Date 类的使用；
2. Calendar 类的使用；
3. SimpleDateFormat 类的使用。

## Date 类

在之前一直在编写简单 java 类，但是所编写的数据表与简单 java 类的转换里面缺少了 Date 数据类型。本部分就属于简单 java 类的最后一块重要的拼板。

在 java 里面提供有一个 java.util.Date 的类

**范例**：获取当前日期时间：

```
package cn.mldn.util;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]){
      Date da=new Date();
      System.out.println(da);
  }
}
```

这个时候是获取了当前日期时间，但是格式实在是难看（Sat Dec 24 15:03:55 CST 2016）。

一直以来强调过一个概念，long 可以描述日期时间数据，那么这一点在 Date 类的方法上也是可以看到的。在 Date 类里面定义了如下几个重要方法：

- 无参构造：public Date()
- 有参构造：public Date(long date)，接受 long 型数据。
- 转换为 long 型: public long getTime()

**范例**：Date 与 long 间的转换

```
package cn.mldn.util;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]){
      long cur=System.currentTimeMillis();//取得当前系统时间，以long返回
      Date da=new Date(cur);
      System.out.println(da);
      System.out.println(da.getTime());

  }
}
```

以后的代码编写之中，依然需要以上的转换操作，尤其是 getTime（）方法很重要。

## SimpleDateFormat（核心）

Java.text 是一个专门实现国际化程序的开发包，而 SimpleDateFormat 是专门处理格式化日期的工具类，即将 Date 型对象转化为 String 的形式出现。而主要使用的是已下方法：

- 构造方法：public SimpleDateFormat(String pattern)，需要传递转换格式
- 将 Date 转化为 String：public final String format(Date date)
- 将 String 转化为 Date：public Date parse(String source)throws ParseException

现在的关键就在于转换格式上，常用的有年（yyyy）、月（MM）、日（dd）、时（HH）、分（mm）、秒（ss）、毫秒（SSS）。

**范例**：将日期格式化显示（Date 型数据转换为 String 型）

```
package cn.mldn.util;
import java.text.SimpleDateFormat;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]){
      Date da=new Date();
      SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd
```

```
HH:mm:ss.SSS");
        String str=sdf.format(da);//将Date型变为String型
        System.out.println(str);
    }
```

除了可以讲日期转换为字符串，还可以将字符串转换为日期。

**范例**：将字符串转换为日期：

```
package cn.mldn.util;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]) throws ParseException{
        String str="2011-11-11 11:11:11.111";
        SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS");
        Date date=sdf.parse(str);//将字符串转换为日期
        System.out.println(date);
    }
}
```

在将字符串变为日期型数据的时候，如果日期型数据给的月不对，那么会自动进行进位。如果给的字符串与要转换的格式不符合，同样会出现异常。

**总结：关于数据类型的转换**

在数据表的操作里面重点说过了几个常用类型，特别是 VARCHAR2（String），CLOB（String），Number（Double、int），Date（java.util.Date）。

Date 与 String 类之间的转换依靠的是 StringDateFormat；

String 与基本数据类型之间的转换依靠的是包装类与 String.valueOf()方法；

Long 与 Date 之间转换依靠的是 Date 类提供的构造以及 getTime（）方法。

# Calendar 类

Date 类与 SimpleDate 类两个都是一起使用的，但是 Calendar 主要是进行一些简单的日期计算的。

Calendar，（Java.util.Calendar）定义如下：

```
public abstract class Calendar
extends Object
implements Serializable, Cloneable, Comparable<Calendar>
```

它是一个抽象类，应该依靠子类进行对象的实例化操作。但是在这个类里面它提供一个 static 方法，此方法返回的正式本类对象：public static Calendar getInstance()

**范例**：取得当前的日期时间

```
package cn.mldn.util;
import java.util.Calendar;
public class TestDemo{
  public static void main(String args[]){
        Calendar cal=Calendar.getInstance();
        StringBuffer buf=new StringBuffer();
        buf.append(cal.get(Calendar.YEAR)).append("-");
```

```
        buf.append(cal.get(Calendar.MONTH)+1).append("-");
        buf.append(cal.get(Calendar.DAY_OF_MONTH)).append(" ");
        buf.append(cal.get(Calendar.HOUR_OF_DAY)).append(":");
        buf.append(cal.get(Calendar.MINUTE)).append(":");
        buf.append(cal.get(Calendar.SECOND)).append("");
        System.out.println(buf);
    }
}
```

但是这个类在取得的时候可以进行一些简单的计算，例如若干天后的日期。如果是日期计算要比 Date 省事。如果是 Date 进行日期的计算，就需要使用 long 进行。一般而言不会轻易使用这个类。

**总结：**

1. 以后数据库中的日期型就使用 java.util.date 表示；
2. 重点强调了 SimpleDateFormat 实现 String 与 Date 间的互换。

# 比较器

**预计知识点：**

1. 重新认识 Arrays 类；
2. 两种比较器的使用；
3. 数据结构之二叉树（Binary Tree）

## Arrays 类（了解）

在之前一直使用的“java.util.Arrays.sort()”可以实现数组的排序，而 Arrays 类实际上就是 java.util 包中提供的一个工具类，这个工具类主要是完成所有与数组有关的操作。

在这个类里面存在**二分查找法：public static int binarySearch(数据类型 a,数据类型 key)**。如果大于 0 表示查找到了。但是使用二分查找有一个前提，即数组必须是排序后的。

```
package cn.mldn.util;
import java.util.Arrays;
public class TestDemo{
  public static void main(String args[])throws Exception{
      int data[]=new int[]{1,5,6,2,3,4,9,8,7,10};
      java.util.Arrays.sort(data);
      System.out.println(Arrays.binarySearch(data, 9));
  }
}
```

**Arrays 类提供了数组比较：public static boolean equals(short[] a,short[] a2)**。只不过借用了 Object 类的 equals 方法名称，与 Object 类的 equals()屁关系都没有。

```
package cn.mldn.util;
public class TestDemo{
  public static void main(String args[])throws Exception{
      int dataA[]=new int[]{1,2,3};
      int dataB[]=new int[]{2,1,3};
      System.out.println(dataA.equals(dataB));
  }
}
```

要想判断数组相同，需要顺序也一致。

**其他操作：**

填充数组：public static void fill(数据类型 a,数据类型 val)。以一个指定的内容将数组填满。
将数组变为字符串输出：public static String toString(数据类型 a)。

```
package cn.mldn.util;
import java.util.Arrays;
public class TestDemo{
  public static void main(String args[])throws Exception{
     int data[]=new int[10];
     Arrays.fill(data,3);
     System.out.println(Arrays.toString(data));
  }
}
```

以上属于数组的基本操作，只不过这样的操作在实际开发里很少出现。

## Comparable 接口（核心）

下面观察 Arrays 类中提供的数组排序方法：
对象数组排序：public static void sort(Object[] a)
发现 Arrays 类里面可以直接利用 sort（）方法实现对象数组排序。
范例：编写测试代码

```
package cn.mldn.tpf;
import java.util.Arrays;
class Book{
  private String title;
  private double price;
  public Book(String title,double price){
     this.title=title;
     this.price=price;
  }
  @Override
  public String toString() {
     return "书名: "+this.title+"价格: "+this.price;
  }
}
public class TestDemo{
  public static void main(String args[])throws Exception{
     Book books[]=new Book[]{
           new Book("java开发",79.8),
           new Book("jsp开发",69.8),
           new Book("Oracle开发",99.8),
           new Book("Android开发",89.8)
     };
     Arrays.sort(books);//对象数组排序
     System.out.println(Arrays.toString(books));
  }
}
```

结果：

```
Exception in thread "main" java.lang.ClassCastException:
cn.mldn.util.Book cannot be cast to java.lang.Comparable
   at java.util.Arrays.mergeSort(Arrays.java:1144)
   at java.util.Arrays.sort(Arrays.java:1079)
            at cn.mldn.util.TestDemo.main(TestDemo.java:23)
```

造成此异常的原因只有一个：两个没有关系的类对象发生了强制的转换。

每一个对象实际上只保留有地址信息，地址里面是有内容的，所以如果是普通的 int 型数组要进行比较，只要判断大小就可以。但是如果是对象数组，里面包含的如果只是编码（地址），比较是没有意义的。就按上面来讲，应该按照价格排序才是有意义的。所以此处必须明确的设置出比较的规则：

比较的规则就是 Comparable 接口定义的。此接口定义：public interface Comparable<T>

```
public interface Comparable<T>{
    public int compareTo(T o);
}
```

实际上 String 类就是 Comparable 接口的子类，之前使用的 compareTo（）就是比较的操作功能。如果用户现在要针对于对象进行比较，建议 compareTo()返回三类数据：1（大于），0（等于），-1（小于）。

**范例：**使用比较器

```
package cn.mldn.tpf;
import java.util.Arrays;
class Book implements Comparable<Book>{//实现比较
  private String title;
  private double price;
  public Book(String title,double price){
     this.title=title;
     this.price=price;
  }

  @Override
  public String toString() {
     return "书名: "+this.title+"价格: "+this.price+"\n";
  }
  @Override
     public int compareTo(Book o) {//Arrays.sort()会自动调用此方法比较
     if(this.price>o.price){
        return 1;
     }else if(this.price<o.price){
        return -1;
     }else{
        return 0;
     }
     }
}
public class TestDemo{
  public static void main(String args[])throws Exception{
     Book books[]=new Book[]{
           new Book("java开发",79.8),
           new Book("jsp开发",69.8),
           new Book("Oracle开发",99.8),
           new Book("Android开发",89.8)
     };
     Arrays.sort(books);
```

```
        System.out.println(Arrays.toString(books));
    }
}
```

compareTo（）由 Arrays.sort()自动调用。

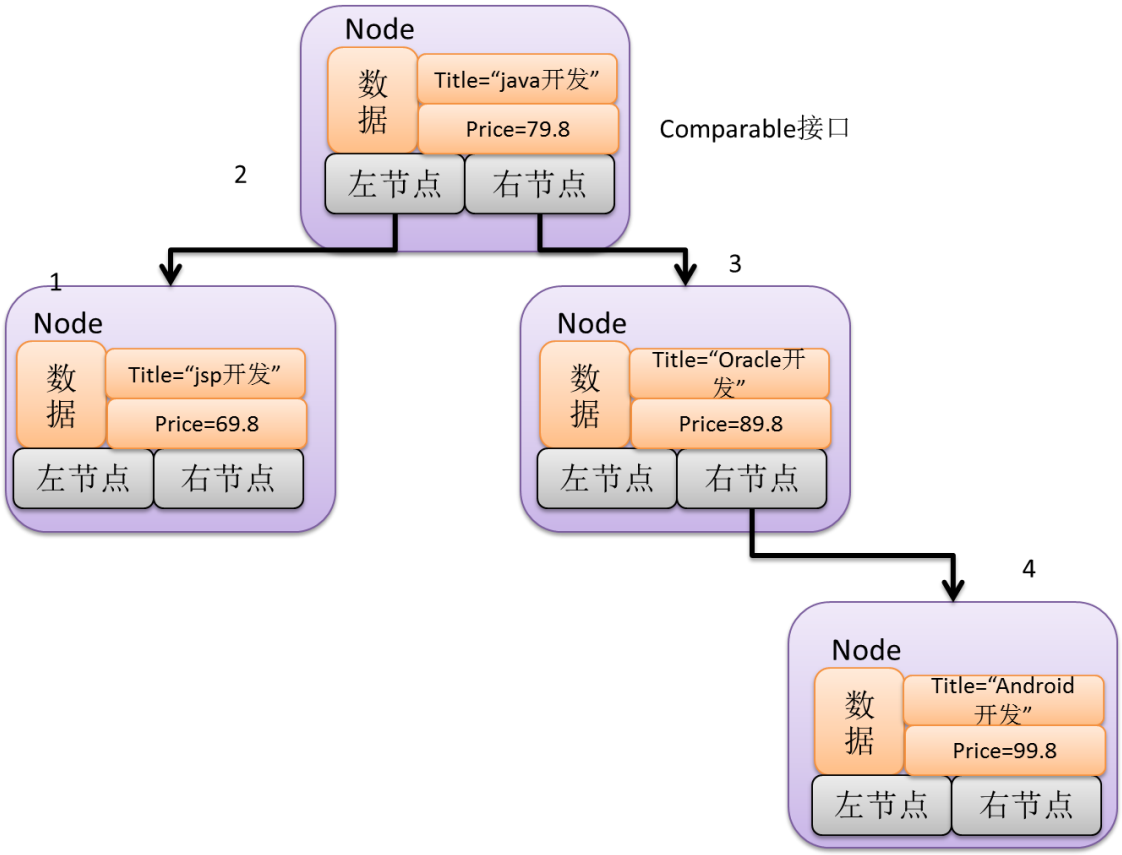
**总结：以后不管何种情况下，只要一种对象要排序，对象所在的类一定要实现 Comparable 接口。**

## 二叉树实现（了解）

树是一种比链表更为复杂概念应用，本质也属于动态对象数组，但是与链表不同在于，树的最大特征是可以根据数据排序。

树的操作原理：选择第一个数据作为根节点，而后比根节点小的放在根节点的左子树（左节点），比根节点大的放在右子树（右节点），取得的时候按照中序遍历的方式取出（左-中-右）。

在任何数据结构里面，Node 类的核心功能是保存真实数以及配置节点关系。




**范例：**实现二叉树

- 定义出要使用的数据，数据所在的类实现 Comparable 接口

```
class Book implements Comparable<Book>{//实现比较
  private String title;
  private double price;
  public Book(String title,double price){
     this.title=title;
```

```
        this.price=price;
    }
    @Override
    public String toString() {
        return "书名: "+this.title+"价格: "+this.price+"\n";
    }
    @Override
        public int compareTo(Book o) {//Arrays.sort()会自动调用此方
法比较
        if(this.price>o.price){
            return 1;
        }else if(this.price<o.price){
            return -1;
        }else{
            return 0;
        }
        }
  }
```

- 定义二叉树，所有的数据结构都需要有 Node 类的支持。

完整代码：

```
package cn.mldn.util;
import java.util.Arrays;
class Book implements Comparable<Book>{//实现比较
  private String title;
  private double price;
  public Book(String title,double price){
      this.title=title;
      this.price=price;
  }
  @Override
  public String toString() {
      return "书名: "+this.title+"价格: "+this.price+"\n";
  }
  @Override
      public int compareTo(Book o) {//Arrays.sort()会自动调用此方法比较
      if(this.price>o.price){
          return 1;
      }else if(this.price<o.price){
          return -1;
      }else{
          return 0;
      }
      }
}
class BinaryTree{
  private class Node{
      private Comparable data;//排序的依据就是Comparable
      private Node left;
      private Node right;
```

```
        @SuppressWarnings("unchecked")
        public Node(Comparable data) {
            this.data=data;
        }
        public void addNode(Node newNode){
            if(this.data.compareTo(newNode.data)>0){
                if(this.left==null){
                    this.left=newNode;
                }else{
                    this.left.addNode(newNode);
                }
            }else{
                if(this.right==null){
                    this.right=newNode;
                }else{
                    this.right.addNode(newNode);
                }
            }
        }
        public void toArrayNode(){
            if(this.left!=null){
                this.left.toArrayNode();//表示有左节点
            }
            BinaryTree.this.retData[BinaryTree.this.foot++]=this.data;
            if(this.right!=null){
                this.right.toArrayNode();//表示有右节点
            }
        }
    }
    private int count;
    private Node root;
    private Object [] retData;
    private int foot;
    public void add(Object obj){//进行数据的追加
        Comparable com=(Comparable)obj;//必须变为Comparable才可以实现数据保存
        Node newNode=new Node(com);//创建新的节点
        if(this.root==null){//现在不存在根节点
            this.root=newNode;//保存根节点
        }else{
            this.root.addNode(newNode);//交给Node类处理
        }
        this.count++;

    }
    public Object [] toArray(){
        if(this.root==null){
            return null;
        }
        this.foot=0;
        this.retData=new Object[this.count];
```

```
        this.root.toArrayNode();
        return this.retData;
    }
}
public class TestDemo{
    public static void main(String args[])throws Exception{
        BinaryTree bt=new BinaryTree();
        bt.add(new Book("java开发",79.8));
        bt.add(new Book("jsp开发",69.8));
        bt.add(new Book("Oracle开发",99.8));
        bt.add(new Book("Android开发",89.8));
        Object obj[]=bt.toArray();
        System.out.println(Arrays.toString(obj));
    }
}
```

可是这些内容 java 的类库都有了自己的实现。

## Comparator 接口

Comparable 接口的主要特征是在类定义的时候默认实现好的接口。现在如有有一个类已经是开发的完善了，但是由于初期的设置并没有安排对象数组的排序，那么后来又需要对象数组的排序，那么在不能修改 Book 类的情况下是不可能使用 Comparable 接口的，为此在 java 里又提供了一个新的比较器：java.util.Comparator，原本在 Comparator 接口里有两个方法。

```
@FunctionalInterface
public interface Comparator<T>{
    public int compare(T o1,T o2);
    public boolean equals(Object obj);
}
```

而真正要使用的只有 compare（）方法，需要单独准备出一个类来实现 Comparator 接口，这个将作为指定类的排序类。

**范例：**定义排序的工具类

```
class BookComparator implements Comparator<Book>{
    @Override
    public int compare(Book o1, Book o2) {
        if(o1.getPrice()>o2.getPrice()){
            return 1;
        }else if(o1.getPrice()<o2.getPrice()){
            return -1;
        }else{
        return 0;
        }
    }
}
```

之前使用 Comparable 接口的时候用的是 Arrays 类中的 sort（）方法，可是现在更换了接口之后也可以使用另外一个被重载的 sort（）方法：public static <T> void sort(T[] a, Comparator<? super T> c)

范例：实现排序

```
package cn.mldn.util;
```

```
import java.util.Arrays;
import java.util.Comparator;
class Book{
  private String title;
  public String getTitle() {
     return title;
  }
  public void setTitle(String title) {
     this.title = title;
  }
  public double getPrice() {
     return price;
  }
  public void setPrice(double price) {
     this.price = price;
  }
  private double price;
  public Book() {
  }
  public Book(String title,double price){
     this.title=title;
     this.price=price;
  }
  @Override
  public String toString() {
     return "书名: "+this.title+"价格: "+this.price+"\n";
  }
}

class BookComparator implements Comparator<Book>{
  @Override
  public int compare(Book o1, Book o2) {
     if(o1.getPrice()>o2.getPrice()){
        return 1;
     }else if(o1.getPrice()<o2.getPrice()){
        return -1;
     }else{
     return 0;
     }
  }
}
public class TestDemo{
  public static void main(String args[])throws Exception{
     Book books[]=new Book[]{
           new Book("java开发",79.8),
           new Book("jsp开发",69.8),
           new Book("Oracle开发",99.8),
           new Book("Android开发",89.8)
     };
     Arrays.sort(books,new BookComparator());
     System.out.println(Arrays.toString(books));
```

```
    }
}
```

很明显，使用 Comparator 比较麻烦，还要定义一个专门的排序类，而且调用排序的时候也要明确的指明一个排序规则类。

**面试题：**请解释 Comparable 和 Comparator 的区别？（请解释两种比较器的区别？）（掌握）

1. 如果对象数组要进行排序，那么必须设置排序规则，可以使用 Comparable 或者 Comparator 接口实现。
2. Java.lang.comparable 是在一个类定义的时候实现好的接口，这样本类的对象数组就可以进行排序，在 Comparable 接口里定义有一个 public int compareTo（）方法；
3. Java.util.Comparator 是专门定义一个指定类的比较规则，属于挽救的比较操作，里面有两个方法：public int compare（），public Boolean equals（）。

**总结：**

1. **以后不管在何种情况下只要牵扯到对象数组的排序一定使用 Comparable 接口；**
2. **根据自己的情况决定是否要熟练编写链表。**

# 正则表达式

所有的开发一定有正则的支持：

1. 记下常用的正则标记；
2. 掌握 String 类对正则的支持。

# 正则引出

为了更好的说明正则的应用，现在来编写一个程序：判断一个字符串是否由数字所组成。

实现原理：将字符串变为字符数组，判断每一个字符是否在“‘0’ ~ ‘9’”范围之间。

**范例：**实现字符串的判断

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[])throws Exception{
     String str="123a";
     System.out.println(isNumber(str));
  }
  public static boolean isNumber(String temp){
     char data[]=temp.toCharArray();
     for(int x=0;x<data.length;x++){
        if(data[x]>'9'||data[x]<'0'){
           return false;
        }
     }
     return true;
  }
}
```

此时判断字符串是否由字符串组成，是一个很容易实现的功能，但是就这样一个简短的操作竟然用了 8 行代码，那么如果是更加复杂的操作呢？

**范例：**更简单的做法

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[])throws Exception{
```

```
        String str="123";
        System.out.println(str.matches("\\d+"));
    }
}
```

一个写了很多行的代码，最后只是写了一行简单的操作就实现了，而其中出现的\\d+就是正则表达式。

正则是从 JDK1.4 的时候正式引入 java 的工具类，所有正则支持的类都定义在 java.util.regex 包里面。在 JDK1.4 之前要想使用正则，则需要单独下载一个正则表达式的开发包后才可以使用。

在 java.util.regex 包里定义有两个主要的类：

Matcher ：通过 Pattern 类取得。

Pattern ：此类对象如果想要取得必须使用 compile（）方法，方法的功能是编译正则；

## 正则标记（背）

所有的正则可以使用的标记都在 java.util.regex.Pattern 类里定义。

**1.单个字符（数量：1）**

**字符：表示由一位字符组成；**

**\\:表示转义字符“\”；**

**\t:表示一个“\t”符号；**

**\n:匹配换行（\n）符号；**

**2.字符集（数量：1）**

**[abc]:表示可能是字符 a 或者是字符 b 或者是字符 c 中的任意一位。**

**[^abc]:表示不是 a、b、c 中的任意一位；**

**[a-zA-Z]:表示任意一位字母，不区分大小写；**

**[a-z]:表示任意一位小写字母；**

**[0-9]:表示任意的一位数字；**

**3.简化的字符集表达式（数量：1）**

**.（点）:表示任意的一位字符；**

**\d:等价于“[0-9]”，属于简化写法（注意应该写\\d）；**

**\D:等价于“[^0-9]”,也属于简化写法；**

**\s:表示任意的空白字符，例如：“\t”、”\n”;**

**\S:表示任意的非空白字符；**

**\w:等价于”[a-zA-Z_0-9]”表示由任意的字母、数字、_组成；**

**\W: 等价于”[^a-zA-Z_0-9]”表示不是由任意的字母、数字、_组成；**

**4.边界匹配（不要在 java 中使用，在 javascript 里使用）**

**^:正则的开始；**

**$:正则的结束；**

**5.数量表达式**

**正则？：表示此正则可以出现 0 次或 1 次；**

**正则+：表示此正则可以出现 1 次或 1 次以上；**

**正则*：表示此正则可以出现 0 次、1 次或多次；**

**正则{n}：表示此正则正好出现 n 次；**

**正则{n，}：表示此正则出现 n 次及以上；**

**正则{n，m}：表示此正则出现 n 到 m 次；**

**6.逻辑运算**

**正则 1 正则 2：正则 1 判断完成后继续判断正则 2；**

**正则 1|正则 2：正则 1 或者正则 2 有一组满足即可；**

（正则）：将多个正则作为一组，可以为这一组单独设置出现的次数；

## String 类对正则的支持（重点）

在 JDK1.4 之后，由于正则的引入，所以 String 类里也相应的增加了新的方法支持。

| No | 方法名称 | 类型 | 描述 |
|---|---|---|---|
| 1 | public boolean matches(String regex) | 普通 | 正则验证，使用指定的字符串判断其是否符合给出的正则表达式结构 |
| 2 | public String replaceAll(String regex, String replacement) | 普通 | 全部替换 |
| 3 | public String replaceFirst(String regex, String replacement) | 普通 | 替换首个 |
| 4 | public String[] split(String regex) | 普通 | 全部拆分 |
| 5 | public String[] split(String regex,int limit) | 普通 | 部分拆分 |

给出的几个方法里面，对于替换和拆分难度不高，最关键是正则匹配，在验证上使用的特别多。

**范例**：实现字符串替换,只保留小写字母

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]){
      String str="hakaHDAH*(dfgjskgj*&KLM##()";
      String regex="[^a-z]";
      System.out.println(str.replaceAll(regex, ""));
  }
}
```

**范例**：字符串拆分，按数字拆分

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]){
      String str="hak56aHDA565H*(d4564fgjskgj*&K46456LM##()";
      String regex="[0-9]+";
      String result[]=str.split(regex);
      for(int x=0;x<result.length;x++){
          System.out.println(result[x]);
      }
  }
}
```

所有正则之中最应该引起我们兴奋的事情是因为可以使用它进行验证。

**范例**：验证一个字符串是否是数字，如果是直接变为 double 型

数字可能是整数（10）也可能是小数（10.2）；

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]){
      String str="10";
      String regex="\\d+(\\.\\d+)?";
      System.out.println(str.matches(regex));
      if(str.matches(regex)){
          System.out.println(Double.parseDouble(str));
```

```
        }
    }
}
```

**范例：**判断给定的字符串是不是一个 IP 地址（IPV4）？

IP 地址：172.16.1.31；每一个字段最多保存 3 个长度。

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]){
      String str="172.16.1.31";
      String regex="(\\d{1,3}\\.){3}\\d{1,3}";
      System.out.println(str.matches(regex));
  }
}
```

范例：给定一个字符串，判断是否是日期格式，如果是则转为 Date 型数据。

```
package cn.mldn.tpf;
import java.text.ParseException;
import java.text.SimpleDateFormat;
public class TestDemo{
  public static void main(String args[]) throws ParseException{
      String str="2016-12-08";
      String regex="\\d{4}-\\d{1,2}-\\d{1,2}";
      //System.out.println(str.matches(regex));
      if(str.matches(regex)){
          SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd");
          System.out.println(sdf.parse(str));
      }
  }
}
```

**范例：**判断电话号码，一般电话号码以下几种格式都是满足的：

格式 1：022-65378132

格式 2：65378132

格式 3：（022）-65378132

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]){
      String str="(022)-65378132";
      String regex="((\\d{3,4}-)?|(\\(\\d{3,4}\\)-)?)\\d{7,8}";
      System.out.println(str.matches(regex));
  }
}
```

**范例：**验证 Email 地址

要求格式 1：email 由字母、数字、下划线组成，\w 正好能描述；

要求格式 2：用户名要求由字母、数字、_、.组成，其中必须以字母开头和结尾，且长度不超过 30；根域名只能是.com、.cn、.net、.com.cn、.net.cn、.edu、.gov、.org

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]){
      String str="ma_dan.12dgdhdhdhdfgd3ling@clinbox.com.cn";
      String
regex="[a-zA-Z][a-zA-Z0-9_\\.]{0,28}[a-zA-Z]@\\w+\\.(net|cn|com|co
```

```
m\\.cn|net\\.cn|edu|gov|org)";
        System.out.println(str.matches(regex));
    }
}
```

以上是几个典型的应用，而以后接触到类似验证的时候，一定要求自己可以直接写出。

## java.util.regex 包支持（理解）

在大多数情况下使用正则的时候都会采用 String 类完成，正则最原始的开发包是 java.util.regex，这个开发包里提供有两个类。

**范例**:Pattern 类

```
package cn.mldn.tpf;
import java.util.Arrays;
import java.util.regex.Pattern;
public class TestDemo{
  public static void main(String args[]){
      String str="a1b22c333d4444e55555f666666";
      String regex="\\d+";
      Pattern pt=Pattern.compile(regex);//编译正则
      String result[]=pt.split(str);
      System.out.println(Arrays.toString(result));
  }
}
```

**范例**：字符串验证

```
package cn.mldn.tpf;
import java.util.Arrays;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class TestDemo{
  public static void main(String args[]){
      String str="7568658";
      String regex="\\d+";
      Pattern pt=Pattern.compile(regex);//编译正则
      Matcher mat=pt.matcher(str);//进行正则匹配
      System.out.println(mat.matches());
  }
}
```

正是因为 String 类本身就已经支持这两种操作了，所以对于 String 类而言，由于所有接收的数据也都是字符串，所以就很少利用 Pattern 和 Matcher 类进行操作了。

**总结：1.利用正则实现验证代码可以最少化；**

**2．一定要清楚 String 类对正则支持的几个方法，以及所有讲解过的相关程序；**

## 反射机制

认识反射

理解反射的作用

利用反射来调用类的结构

# 认识反射

反射先通过“反”来理解，有反就一定有正。在正常情况下一定是先有类而后再产生对象。所谓的反，就是可以利用对象找到对象的出处，在 Object 类里面提供有一个方法：

取得 class 对象：public final Class<?> getClass()

**范例**：观察反

```
package cn.mldn.tpf;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]){
      Date date=new Date();
      System.out.println(date.getClass());
  }
}
```

结果：class java.util.Date

## 实例化 Class 类对象

java.lang.Class 类是反射操作的源头，即：所有的反射都要从此类开始，这个类有三种实例化方式：

第一种：调用 Object 类的 getClass 方法（用的几率很小）：

```
package cn.mldn.tpf;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]){
      Date date=new Date();
      Class<?> cls=date.getClass();
      System.out.println(cls);
  }
}
```

第二种：使用“类.class”取得，以后讲解 Hibernate、MyBatis、Spring；

```
package cn.mldn.tpf;
import java.util.Date;
public class TestDemo{
  public static void main(String args[]){
      Class<?> cls=Date.class;
      System.out.println(cls);
  }
}
```

之前是在产生了类的实例化对象之后取得的 Class 类对象，但是此时并没有实例化对象的产生。

第三种：调用 Class 类提供的一个方法：

实例化 Class 类对象：

public static Class<?> forName(String className)  throws ClassNotFoundException

```
package cn.mldn.tpf;
public class TestDemo{
  public static void main(String args[]) throws
ClassNotFoundException{
      Class<?> cls=Class.forName("java.util.Date");
```

```
        System.out.println(cls);
    }
}
```

此时可以不是要 import 语句导入一个明确的类，而类名称是采用字符串的形式进行描述的。

## 反射实例化对象

当拿到一个类的时候，肯定要使用关键字 new 进行对象的实例化操作，这属于习惯性做法。但是如果有了 Class 类对象那么就可以做到利用反射来实现对象实例化操作：

实例化对象方法：public T newInstance()throws InstantiationException,IllegalAccessException

**范例：**利用反射实例化对象

```
package cn.mldn.tpf;
class Book{
  public Book() {
      System.out.println("*****无参构造***********");
  }
  @Override
  public String toString() {
      return "这是一本书！";
  }
}
public class TestDemo{
  public static void main(String args[]) throws Exception{
      //Book b=new Book();
      //System.out.println(b);
      Class<?> cls=Class.forName("cn.mldn.tpf.Book");
      Object obj=cls.newInstance();//相当于使用new调用无参构造
      Book b=(Book)obj;
      System.out.println(b);
  }
}
```

有了反射之后，以后进行对象实例化的操作不再只是单独的依靠关键字 new 完成了，反射也可以完成，但是这并不表示关键字 new 就被完全取代了。

在任何的开发之中，new 是造成耦合的最大元凶，一起的耦合都起源于 new。

**范例：**观察工厂设计模式：

```
package cn.mldn.test;
interface Fruit{
  public void eat();
}
class Apple implements Fruit{
  @Override
  public void eat() {
      System.out.println("***吃苹果！");
  }
}
class Factory{
  public static Fruit getInstance (String className){
      if("apple".equals(className)){
          return new Apple();
```

```
        }
        return null;
    }
}
public class testFactory {
    public static void main(String [] args){
        Fruit f=Factory.getInstance("apple");
        f.eat();
    }
}
```

如果此时增加了 Fruit 接口子类，那么就表示程序要修改工厂类。如果随时都可能增加子类呢？工厂类要被一直进行修改。因为现在工厂类中的对象都是通过关键字 new 直接实例化的，而 new 就成了所有问题的关键点。要想解决这一问题就要依靠反射完成。

```
package cn.mldn.test;
interface Fruit{
    public void eat();
}
class Apple implements Fruit{
    @Override
    public void eat() {
        System.out.println("***吃苹果！");
    }
}
class Orange implements Fruit{
    @Override
    public void eat() {
        System.out.println("***吃橘子！");
    }
}
class Factory{
    public static Fruit getInstance (String className){
        Fruit f=null;
        try {
            f=(Fruit)Class.forName(className).newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return f;
        }
}
public class testFactory {
    public static void main(String [] args){
        Fruit f=Factory.getInstance("cn.mldn.test.Orange");
        f.eat();
    }
}
```

此时的程序就真正完成了解耦合的目的，而且可扩展性非常强。

## 调用构造方法

在之前所编写的代码实际上发现都默认调用了类之中的无参构造方法，可是类中还有可能不提供无参构造呢？

**范例：**观察当前程序的问题

```
package cn.mldn.th;
public class Book {
  private String title;
  public String getTitle() {
     return title;
  }
  public void setTitle(String title) {
     this.title = title;
  }
  public double getPrice() {
     return price;
  }
  public void setPrice(double price) {
     this.price = price;
  }
  private double price;
  public Book(String title,double price){
     this.title=title;
     this.price=price;
  }
  @Override
  public String toString() {
     return "书名: "+this.title+",价格: "+this.price+"\n";
  }
}
```

另一个包中：

```
package cn.mldn.tpf;
import cn.mldn.th.Book;
public class TestDemo{
  public static void main(String args[]) throws Exception{
     //Book b=new Book();
     //System.out.println(b);
     Class<?> cls=Class.forName("cn.mldn.th.Book");
     Object obj=cls.newInstance();
     Book b=(Book)obj;
     System.out.println(b);
  }
}
```

由于此时 Book 类没有提供无参构造方法，所以出错了。

```
Exception in thread "main" java.lang.InstantiationException:
cn.mldn.th.Book
   at java.lang.Class.newInstance0(Class.java:340)
   at java.lang.Class.newInstance(Class.java:308)
   at cn.mldn.tpf.TestDemo.main(TestDemo.java:8)
```

以上的错误指的是当前 Book 类里面没有无参构造方法，所以程序无法进行对象的实例化。在这种情况下，只能明确的调用有参构造方法。

在 Class 类里面有一个方法可以取得构造：

取得全部构造：public Constructor<?>[] getConstructors()throws SecurityException

取得一个指定参数顺序的构造：public Constructor<T> getConstructor(Class<?>... parameterTypes)throws NoSuchMethodException,SecurityException

以上两个方法返回的都是“java.lang.reflect.Constructor”类的对象，在这个类中提供有一个明确传递有参构造内容的实例化对象方法：public T newInstance(Object... initargs)throws InstantiationException,IllegalAccessException,IllegalArgumentException,InvocationTargetException

**范例：**明确调用类的有参构造

```
package cn.mldn.tpf;
import java.lang.reflect.Constructor;
import cn.mldn.th.Book;
public class TestDemo{
  public static void main(String args[]) throws Exception{
      //Book b=new Book();
      //System.out.println(b);
      Class<?> cls=Class.forName("cn.mldn.th.Book");
      Constructor<?>
con=cls.getConstructor(String.class,double.class);
      Object obj=con.newInstance("java开发",78.8);
      Book b=(Book)obj;
      System.out.println(b);
  }
}
```

简单 java 类的开发之中不管提供有多少构造方法，请至少保留有无参构造。

## 调用普通方法

类中的普通方法只有在一个类产生实例化对象之后才可以调用。并且实例化对象的访视有三种（new、clone、反射）。

**范例：**定义一个类

```
package cn.mldn.th;
public class Book {
   private String title;
   private double price;
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public double getPrice() {
      return price;
   }
   public void setPrice(double price) {
      this.price = price;
   }
   @Override
   public String toString() {
      return "书名: "+this.title+",价格: "+this.price+"\n";
   }
}
```

这个类有无参构造方法，所以实例化对象的时候可以直接 Class 类中的 newInstance（）方法。
在 Class 类中提供有一下取得 Method 的操作：

- 取得类中的全部方法：public Method[] getMethods()throws SecurityException
- 取得指定方法：public Method getMethod(String name,Class<?>... parameterTypes) throws NoSuchMethodException,SecurityException

以上的两个操作返回的是 java.lang.reflect.Method 类的对象，在这个类里面重点关注一个方法：调用方法：public Object invoke(Object obj,Object... args)throws IllegalAccessException, IllegalArgumentException,InvocationTargetException

**范例：**反射调用方法

```
package cn.mldn.tpf;
import java.lang.reflect.Method;
public class TestDemo{
    public static void main(String args[]) throws Exception{
        String fieldName="title";
        Class<?> cls=Class.forName("cn.mldn.th.Book");
        Object obj=cls.newInstance();
        Method setMet=cls.getMethod("set"+initcap(fieldName),
String.class);
        Method getMet=cls.getMethod("get"+initcap(fieldName));
        setMet.invoke(obj,"java开发");
        System.out.println(getMet.invoke(obj));
    }
    public static String initcap(String str){
        return str.substring(0,1).toUpperCase()+str.substring(1);
    }
}
```

此时完全看不到具体的操作类型，也就是说利用反射可以实现任意类的指定方法的调用。

## 调用成员

类中的属性一定要在本类实例化对象产生之后才可以分配内存空间。在 Class 类里面提供有取得成员的方法：

- 取得全部成员：public Field[] getDeclaredFields()throws SecurityException
- 取得指定成员：public Field getDeclaredField(String name) throws NoSuchFieldException,SecurityException

返回的类型是 java.lang.reflect.Field，在这个类里面有两个重要的方法：

- 取得属性内容：public Object get(Object obj) throws IllegalArgumentException,IllegalAccessException
- 设置属性内容：public void set(Object obj,Object value) throws IllegalArgumentException,IllegalAccessException

在 java.lang.reflect.AccessibleObject 类下（JDK1.8 之后修改）：

Executable：下面继续继承了 Constructor, Method；

Field：

在这个类下面提供有一个方法：public void setAccessible(boolean flag)throws SecurityException，设置是否取消封装。

**范例：**现在提供有如下的类

```
package cn.mldn.th;
public class Book {
    private String title;
```

```
}
```

这个类里只定义了私有属性，按照原始的方法，此时它一定无法被外部使用。

**范例：**反射调用

```
package cn.mldn.tpf;
import java.lang.reflect.Field;
public class TestDemo{
    public static void main(String args[]) throws Exception{
        Class<?> cls=Class.forName("cn.mldn.th.Book");
        Object obj=cls.newInstance();
        Field titleField=cls.getDeclaredField("title");
        titleField.setAccessible(true);
        titleField.set(obj,"java开发");
        System.out.println(titleField.get(obj));
    }
}
```

构造方法与普通方法也可以取消封装，只不过很少这样去做，对于属性还是建议使用 setter、getter 方法完成。

**总结：**

1. **实例化对象的方式又增加了一种反射；**
2. **对于简单 java 类的定义应该更加清晰了；**
3. **反射调用类结构只是一个开始。**

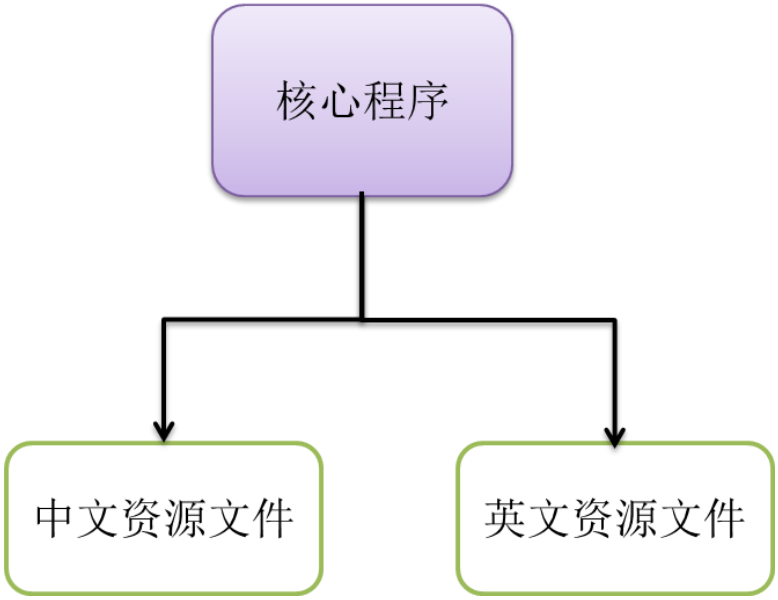
# 国际化

# 国际化程序实现

在工作开发里面，国际化应用一定会存在，只不过，如果你只是针对于使用情况下开发，国际化基本上就是应用。

如果说现在又一套程序，中国、美国、俄罗斯都要使用，很明显，不管哪块使用，程序的核心功能不会被改变，唯一可能被改变是语言。

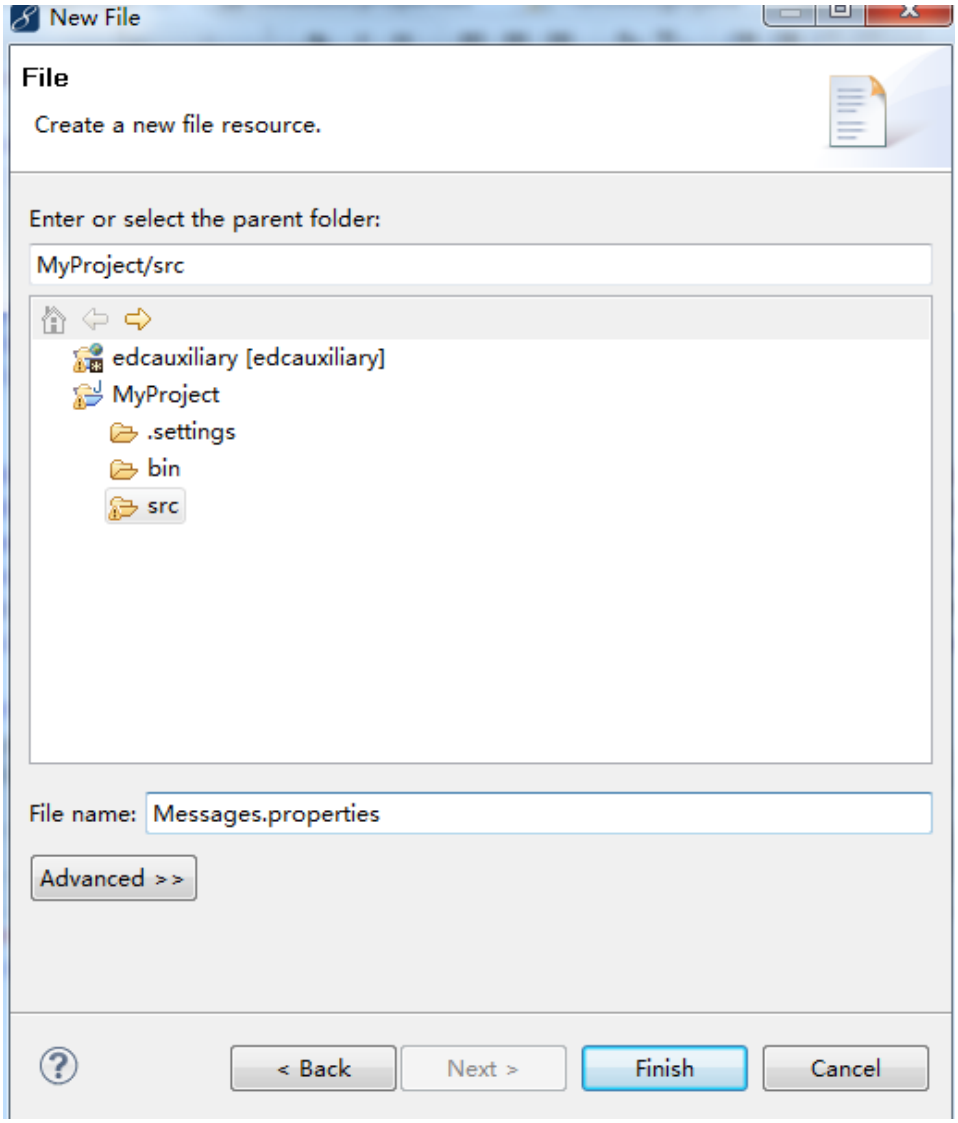
如果想实现语言的统一，那么唯一能够做的方式就是将所有需要显示的语言定义在各自的资源文件里面。




所谓的国际化应用指的就是根据当前的语言环境读取指定的语言资源文件。

如果想要实现国际化的操作，那么首先要解决的问题就是如何读取资源文件。

所谓的资源文件指的就是后缀名称为“*.properties”里面保存的内容按照“key=value”的形式保存，而且资源文件的命名标准与 java 类完全一样。




**范例：**定义一个 *Messages*.properties

如果保存的是中文信息，必须将其变为 Unicode 编码。

```
Info=中华人民共和国
```

这里面保存的 info 是这个信息的 key，以后要根据 key 取得对应的 Value。

如果要读取资源文件的信息读取“java.util.ResourceBundle”类。这是一个抽象类，但是这个类的内容也提供有一个 static 方法，用于取得本类对象：

根据当前语言环境取出：public static final ResourceBundle getBundle(String baseName)

设置指定语言环境：public static final ResourceBundle getBundle(String baseName,Locale locale)

当取得了 ResourceBundle 类对象之后可以通过以下的方法读取数据：

简单读取：public final String getString(String key)

Java.text 是专门负责国际化处理的程序包，在这个包里面还有一个专门处理占位数据的操作类：MessageFormat 类，格式化文本：public static String format(String pattern,Object... arguments)

**范例：**读取普通文本

```
package cn.mldn.de1;
import java.util.ResourceBundle;
public class TestDemo {
    public static void main(String []args){
```

```
        //访问的时候不要加上后缀，因为默认后缀就是*.properties
        //此时的Messages.properties一定要放在CLASSPATH路径下
        ResourceBundle rb=ResourceBundle.getBundle("Messages");
        System.out.println(rb.getString("info"));
    }
}
```

很多时候数据是会被改变的。

**范例：**修改 Messages.properties 文件

```
Wel.msg=欢迎{0}光临，现在时间是：{1}!
```

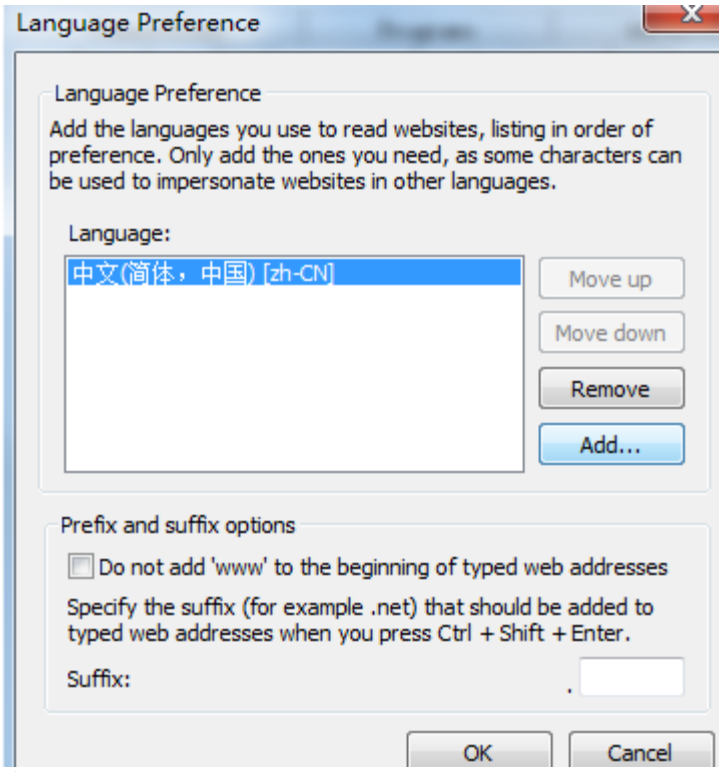
**范例：**设置读取的可变内容

```
package cn.mldn.de1;
import java.text.MessageFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.ResourceBundle;
public class TestDemo {
    public static void main(String []args){
        //访问的时候不要加上后缀，因为默认后缀就是*.properties
        //此时的Messages.properties一定要放在CLASSPATH路径下
        ResourceBundle rb=ResourceBundle.getBundle("Messages");
        String str=rb.getString("wel.msg");
        System.out.println(MessageFormat.format(str, "阿佑",new
SimpleDateFormat("yyyy-MM-dd").format(new Date())));
    }
}
```

如果只是从事应用开发，那么不需要编写以上的代码，你所要编写的只是一个资源文件而已。

国际化程序应该要根据所在国家的不同显示不同的内容，可是只是提供了一个资源文件，那么怎么进行不同语言的显示呢？那么就需要 Locale 类的帮忙了。

Locale 类保存的是一个国家的区域和语言编码(可以通过 ie 浏览器查 alt，工具，Internet 选项，语言，add 可以看其他的)：




中国：zh_CN

美国：en_US

可以在定义资源文件的时候加上指定的语言编码

**范例**：定义中文的资源文件—Messages_zh_CN.properties

```
Wel.msg=欢迎{0}光临！
```

**范例**：定义英文的资源文件—Messages_en_US.properties

```
Wel.msg=Welcome {0}!
```

设置的 baseName 设置的一定是 Messages，所有的语言代码有 Locale 类设置，在 Locale 类里提供有以下方法：

构造方法：public Locale(String language,String country)；

取得当前语言环境：public static Locale getDefault()；

**范例**：读取中文文件

```
package cn.mldn.de1;
import java.text.MessageFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class TestDemo {
    public static void main(String []args){
        Locale loc=new Locale("zh","CN");
        ResourceBundle rb=ResourceBundle.getBundle("Messages",loc);
        String str=rb.getString("wel.msg");
        System.out.println(MessageFormat.format(str, "阿佑"));
    }
}
```

**范例**：读取英文

```
package cn.mldn.de1;
import java.text.MessageFormat;
```

```
import java.util.Locale;
import java.util.ResourceBundle;
public class TestDemo {
    public static void main(String []args){
        Locale loc=new Locale("en","US");
        ResourceBundle rb=ResourceBundle.getBundle("Messages",loc);
        String str=rb.getString("wel.msg");
        System.out.println(MessageFormat.format(str, "阿佑"));
    }
}
```

特定语言的资源文件读取的优先级会高于公共语言资源文件读取的优先级。

总结：

1. 资源文件的要求是文件名称每个单词首字母大写，而后缀必须是*.properties
2. 通过 ResourceBundle 可以读取指定的 CLASSPATH 下的资源文件，读取时不需要输入文件后缀。，注意动态的占位文本格式化：MessageFormat。
3. Locale 类用于指定读取的资源文件的语言环境。

# 文件操作

## 基本操作

在 java 里面，对于所有的初学者而言，可能最麻烦的就是 javaIO，因为这里面所牵扯到的父类与子类实在太多了。但是只要本着一个原则：抽象类中定义的抽象方法会根据实例化其子类的不同，也会完成不同的功能。

使用 File 类进行文件的操作。

如果要进行文件及文件内容的开发操作，应该使用的是 java.io 包完成，而在 java.io 包里面一共有五个核心类和一个核心接口：

五个核心类：File、InputStream、OutputStream 、Reader、Writer；

一个核心接口：Serializable。

在整个 java.io 包里面，File 类是唯一一个与文件本身操作有关的类，但不涉及文件的具体内容。所谓的文件本身指的是文件的创建、删除等操作。

如果要想使用 File 类，那么首先就需要通过它提供的构造方法定义一个要操作文件的路径。

设置完整路径：public File(String pathname)，大部分情况下都使用此类操作；

设置父路径与子文件路径：public File(String parent,String child)，在 Android 上用的比较多。

**范例：**操作文件

创建文件：public boolean createNewFile()throws IOException；

抛出异常？如果目录不能访问；如果文件重名；文件名称错误；

删除文件：public boolean delete()

判断文件是否存在：public boolean exists()

```
package cn.mldn.th;
import java.io.File;
public class TestDmo{
    public static void main(String args[])throws Exception{
        File file=new File("e:\\demo.txt");//设置文件的路径
        if(file.exists()){
            file.delete();
        }else{
```

```
        System.out.println(file.createNewFile());
        }
    }

}
```

以上的程序已经完成了具体的文件创建于删除，但是此时的程序会存在有两个问题：

1.在 Windows 系统里面支持的是“\”路径分隔符；Linux 下使用的是“/”路径分隔符；

解决方法：在 File 类里面提供有一个常量：public static final String separator；

```
File file=new File("e:"+File.separator+"demo.txt");//设置文件的路径
```

2.在进行 java.IO 操作的过程之中会出现延迟情况，因为现在的问题是 java 的程序是通过 JVM 间接的调用操作系统的文件处理函数进行的文件处理操作，所以中间会出现延迟。

## File 类操作方法

以上已经实现了文件的创建操作，但是这个时候是直接创建在了根路径下，下面来创建包含有子目录的文件：

```
File file=new File("e:"+File.separator+"test"+File.separator+"demo.txt");//设置文件的路径
```

由于“test”目录不存在，所以系统会认为此时的路径是不能够使用的，所以就会出现错误。所以必须要想想办法判断父路径是否存在？

找到父路径：public File getParentFile()；

创建目录：public boolean mkdirs()；

```
package cn.mldn.th;
import java.io.File;
public class TestDmo{
    public static void main(String args[])throws Exception{
        File file=new
File("e:"+File.separator+"test"+File.separator+"hello"+File.separa
tor+"mldn"+File.separator+"demo.txt");//设置文件的路径
        if(!file.getParentFile().exists()){
            file.getParentFile().mkdirs();
        }
        if(file.exists()){
            file.delete();
        }else{
        System.out.println(file.createNewFile());
        }
    }

}
```

在 File 类里面还提供有一系列取得文件信息内容的操作功能：

取得文件大小，按照字节返回：public long length()；

判断是否是文件：public boolean isFile()；

判断是否是目录：public boolean isDirectory()；

最近一次修改日期：public long lastModified()；

```
package cn.mldn.th;
import java.io.File;
import java.math.BigDecimal;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
public class TestDmo{
    public static void main(String args[])throws Exception{
        File file=new File("e:"+File.separator+"my.jpg");//设置文件的路
径
        if(file.exists()){
            System.out.println("是否是文件？"+file.isFile());
            System.out.println("是否是目录？"+file.isDirectory());
            System.out.println("文件大小："+(new
BigDecimal((double)file.length()/1024/1024)).divide(new
BigDecimal(1), 2, BigDecimal .ROUND_HALF_UP)+"M");
            System.out.println("上次修改时间："+new
SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
Date(file.lastModified())));
        }
    }

}
```

整个取得过程都是取得文件相关信息，但是并不取得文件内容。

## 操作目录

以上的所有操作都是围绕着文件进行的，但是在整个磁盘上除了文件之外，还会有目录，对于目录而言最为常用的功能就是列出目录组成，在 File 类里面定义有如下两个列出目录的方法：

列出目录信息：public String[] list()；

列出所有的信息以 File 类包装：public File[] listFiles()；

**范例：**列出信息

```
package cn.mldn.th;
import java.io.File;
public class TestDmo{
    public static void main(String args[])throws Exception{
        File file=new File("e:"+File.separator);//设置文件的路径
        if(file.isDirectory()){
            String result[]=file.list();
            for(int x=0;x<result.length;x++){
                System.out.println(x+":"+result[x]);
            }
        }
    }

}
```

此时的确可以列出目录中的内容了，但是所列出来的是目录下的子目录或文件的名字。

**范例：**列出全部的 File 类对象

```
package cn.mldn.th;
import java.io.File;
public class TestDmo{
    public static void main(String args[])throws Exception{
        File file=new File("e:"+File.separator);//设置文件的路径
        if(file.isDirectory()){
            File result[]=file.listFiles();
```

```
            for(int x=0;x<result.length;x++){
                System.out.println(x+":"+result[x]);
            }
        }
    }

}
```

为了更好的体验出以上操作的好处，下面输出一个类似于资源管理器的界面。

```
package cn.mldn.th;
import java.io.File;
import java.math.BigDecimal;
import java.text.SimpleDateFormat;
import java.util.Date;
public class TestDmo{
    public static void main(String args[])throws Exception{
        File file=new File("e:"+File.separator);//设置文件的路径
        if(file.isDirectory()){
            File result[]=file.listFiles();
            for(int x=0;x<result.length;x++){
                System.out.println(result[x].getName()+"\t\t\t"+
                        new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date(result[x].lastModified()))+"\t\t"+
                        (result[x].isDirectory()?"文件夹":"文件")+"\t\t"+
                        (result[x].isFile()?new
BigDecimal(((double)result[x].length()/1024)).divide(new
BigDecimal(1), 2, BigDecimal.ROUND_HALF_UP)+"K":""));
            }
        }
    }

}
```

通过一系列的验证可以发现取得文件的对象列表会更加方便，因为可以继续取出更多内容。

**思考题：**列出指定目录下的所有子路径？

原则：如果现在给定的路径依然是一个目录，则应该向里面继续列出所有的组成，应该使用递归的方式完成。

```
package cn.mldn.th;
import java.io.File;
import java.math.BigDecimal;
import java.text.SimpleDateFormat;
import java.util.Date;
public class TestDmo{
                        public static void main(String args[])throws Exception{
                          File file=new File("d:"+File.separator);//设置文件的路径
                          print(file);

                        }
                        public static void print(File file){
                          if(file.isDirectory()){
                                File result[]=file.listFiles();
                                if(result!=null){
```

```
                    for(int x=0;x<result.length;x++){
                        print(result[x]);
                    }
                    }
                }
                    System.out.println(file);
            }

}
```

如果将以上的输出操作更换为了删除呢？这就成为了一个恶性程序了。

总结：

1. File 类本身只是操作文件的，不涉及内容。
2. File 类中的重要方法：

   设置完整路径：public File(String pathname)，大部分情况下都使用此类操作；

   删除文件：public boolean delete();

   判断文件是否存在：public boolean exist();

   找到父路径：public File getParentFile();

   创建目录：public boolean mkdirs();

3.路径分隔符 File. separator

# 方法的定义与使用

## 基本概念

方法：指的是定义在主类之中，并且由主方法直接调用的方法。

在方法返回值类型为 void 的时候，可以使用 return 结束方法，后面的代码不执行了。（很少用）

## 方法重载（重要）

如果说现在又一个方法名称，有可能要执行多项操作。例如 add()方法可能要执行两个整数的相加，也可能要执行三个整数的相加，或者可能执行两个小数的相加。所以要为 add()方法实现多个功能的实现，此类的功能就成为重载。要求方法的名称相同、参数的类型及个数不同。

方法重载的概念本身很容易理解，但是对于方法重载有两点说明：

1. 在进行方法重载的时候一定要考虑到参数类型的统一，虽然可以实现重载方法返回类型不同，但是从标准的开发来讲，建议所有重载后的方法是用同一种返回类型
2. 方法重载的时候重点根据参数类型及个数来区分不同的方法，而不是依靠返回值的不同来确定的。

## 递归调用（了解）

递归调用是迈向数据结构开发的第一步，如果真想把递归操作掌握熟练，那么需要大量的代码积累。在标准的项目开发里面，很难去应用递归操作的。

所谓的递归调用指的是一个方法自己调用自己的情况，但是如果要想实现自己调用自己，一定需要一个结束的条件。并且每次调用的时候都需要去修改结束条件。

之所以递归操作在开发之中尽量减少使用，是因为如果处理不当，可能会出现内存的溢出问题。

范例：计算从 1 加到 100 的值。

```
package cn.mldn.tpf;
public class TestDemo{
    public static void main(String args[]){
        System.out.println(sum(100));
        System.out.println(sum(200));
    }
    public static int sum(int t){
        if(t==1){
            return 1;
        }
        return t+sum(t-1);
    }
}
```

总结：

1. 将一些可以重复执行的代码定义在方法里面，方法（Method）在有些书上也被成为函数（Function）。
2. 本次所讲解的方法是有它自己的局限性，定义主类，并且由主方法直接调用。
3. 方法返回值一旦定义了就要使用 return 返回相应数据。
4. 方法重载（Overloading）指的是方法名称相同，参数的类型和个数不同，同时尽量保持返回值类型相同。
5. 递归调用需要明确的设置一个结束的调用，如果处理数据量太多，就有可能出现溢出。

# 字节流与字符流

## 简介

File 类虽然可以操作文件，但并不是操作文件的内容，如果要进行内容的操作只能够通过两种途径完成：字节流和字符流。

如果要进行输入、输出操作一般按照如下的步骤进行（以文件操作为例）：

- 通过 File 类定义一个要操作文件的路径；
- 通过字节流或字符流的子类对象为父类对象实例化；
- 进行数据的读（输入）、写（输出）操作；
- 数据流属于资源操作资源操作必须关闭。

对于 java.io 包定义了两类流：

- 字节流（JDK1.0）：InputStream、OutputStream；
- 字符流（JDK1.1）：Reader、Writer；

## 输出流：OutputStream

OutputStream 类是一个专门进行字节数据输出的一个类，这个类定义如下：

```
public abstract class OutputStream
extends Object
implements Closeable, Flushable
```

发现 OutputStream 类实现了两个接口：Closeable、Flushable，这两个接口定义如下：

| Closeable（JDK1.5） | Flushable（JDK1.5） |
| --- | --- |
| public interface Closeable<br>extends AutoCloseable{<br>public void close() throws IOException<br>} | public interface Flushable{<br>public void flush()throws IOException<br>} |

但是 OutputStream 类是在哪 JDK1.0 的时候就提供的，这个类原本就定义了 close（）与 flush（）两个操作方法，所以以上的两个接口就几乎可以忽略了。

在 OutputStream 类里面一共提供有三个输出的方法：

输出单个字节：public void write(byte[] b)throws IOException；

输出全部字节数组：public void write(byte[] b)throws IOException；

输出部分字节数组：public void write(byte[] b,int off,int len)throws IOException。

这三个方法，最重要的是输出部分字节数组。

OutputStream 类属于抽象类，如果要想为抽象类进行对象的实例化操作，那么一定要使用抽象类的子类，由于要进行的文件操作，所以可以使用 FileOutputStream 子类。在这个子类里定义有如下构造方法：

```

```

```

```

```

```

# 打印流

## 问题引出

1. 打印流的实现原理；
2. 打印流操作类的使用。

打印流属于整个 java 开发过程之中，非常重要的一个组成概念。

现在我们已经清楚了 InputStream 和 OutputStream 两个类的

# 类集框架

## 类集简介（了解）

类集就是一组 java 实现的数据结构，或者，所谓的类集指的就是对象数组的应用。

在之前讲解的时候，如果要想保存多个数据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）。后来使用了链表实现了动态的对象数组，但是对于链表的开发最大感受：

1. 链表开发实在太麻烦了；
2. 如果要考虑到链表的操作性能太麻烦了；
3. 链表使用了 Object 类进行保存，所有的对象必须发生向上以及强制的向下转型操作。

综合以上的问题得出了结论：在开发项目里面由用户自己去实现一个链表，那么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 设计之初就考虑到了此类问题，所以提供了一个跟链表类似的工具类—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的概念，所以从 java2（JDK1.2）之后提供了一个专门实现数据结构的框架—类集框架，并且在 JDK1.5 之后，由于泛型技术的引用，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以下几个核心接口：

1. Collection、List、Set;
2. Map;
3. Iterator、Enumeration。

总结：类集就是 java 数据结构实现、类集就是动态对象数组。

## Collection 接口（重点）

了解 Collection 接口，以及它的相关常用操作方法定义。

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保存一个对象。观察 Collection 接口的定义：

```
public interface Collection<E>
extends Iterable<E>
```

在 Collection 里定义有如下几个常用的操作方法：

| 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 |
| 4 | public boolean contains(Object o) | 普通 | 判断是否包含有指定的内容，需要 equals |
| 5 | public boolean isEmpty() | 普通 | 判断是否是空集合（不是 null） |
| 6 | public boolean remove(Object o) | 普通 | 删除对象，需要 equals 支持 |
| 7 | public int size() | 普通 | 取得原集合中保存的元素个数 |
| 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（99%），其它的方法几乎可以忽略。但是千万要记住，contains（）与 remove（）两个方法一定要依靠 equals（）支持。

从一般的道理讲，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并且使用。但是现在的开发由于要求的严格性，所以不会在直接使用 Collection 接口，而都会使用它的子接口：List、Set。

历史回顾：最早 java 刚刚推出类集框架的时候，使用最多的是 Collection 接口，最大的使用环境是在 EJB 上。于是在 java 的一个开源项目上—PetShop，就出现了一个问题，由于此项目属于 java

的一个业余爱好者共同开发的，所以没有考虑过多的性能问题以及代码或数据库设计，那么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推出.net 平台。所以微软使用.net 重新设计并开发了 PetShop，对外宣布，性能比 java 好，于是事情就发生了本质改变，人们就认为.net 平台性能很高（实际上和 java 没什么区别），后来 SUN 的官方重新编写了 PetShop，并且发布了测试报告，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。

因为代码规范化的产生，所以从 PetShop 开始就不再使用迷糊不清的 Collection 接口了，而是使用 List（允许重复）或 Set（不允许重复）子接口进行开发。

总结：

1. Collection 接口几乎不会直接使用了；
2. 一定要将 Collection 接口的方法全部记住。

# 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 子接口的操作特点及常用子类（ArrayList、Vector）。

List 子接口（80%）是 Collection 接口中最常用的子接口，但是这个接口对于 Collection 接口进行了一些功能的扩充。在 List 子接口里面，重点要掌握以下方法的使用：

| 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，可以使用 ArrayList 子类实现操作（还有 Vector 子类，90%使用 ArrayList）。

ArrayList 类是 List 接口最为常用的子类。下面将利用此类来验证所学到的操作方法。

范例：List 基本操作。

```
package cn.mldn.de1;
import java.util.ArrayList;
import java.util.List;
public class TestDemo {
    public static void main(String []args){
        //设置泛型可以保证集合里的元素都一致
        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子接口扩充了get方法，可以根据索引取得数据
        for(int x=0;x<all.size();x++){
            System.out.println(all.get(x));
        }
    }
}
```

结果：

```
长度：0，是否为空：true
长度：3，是否为空：false
Hello
Hello
World!
```

通过演示可以发现，List 集合之中所保存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，List 子接口扩充有 get（）方法。

范例：为 Collection 接口实例化。

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

```
package cn.mldn.de1;
import java.util.ArrayList;
import java.util.Collection;
public class TestDemo {
    public static void main(String []args){
        //设置泛型可以保证集合里的元素都一致
        Collection<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子接口扩充了get方法，可以根据索引取得数据
        Object obj[]=all.toArray();//变为对象数组取得
        for(int x=0;x<obj.length;x++){
            System.out.println(obj[x]);
        }
    }
}
```

结果：

```
长度：0，是否为空：true
长度：3，是否为空：false
Hello
Hello
World!
```

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的标准输出结构钢，只是做基础的展示。

范例：在集合里保存对象

```
package cn.mldn.de1;

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 String toString() {
        return "书名: "+this.title+",价格: "+this.price+"\n";
    }
    @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;
    }
}
public class TestDemo {
    public static void main(String []args){
        List<Book> all=new ArrayList<Book>();
        all.add(new Book("java开发",88.9));
        all.add(new Book("jsp开发",78.9));
        all.add(new Book("oracle开发",68.9));
        //任何情况下集合数据的删除与内容的查询都必须提供有equals（）方法
        all.remove(new Book("oracle开发",68.9));
        System.out.println(all);
    }
}
```

结果：

```
[书名：java开发,价格：88.9
, 书名：jsp开发,价格：78.9
]
```

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

Vector 子类：

在最早 JDK1.0 的时候就已经提供了 Vector 类，并且这个类被大量使用。但是到了 JDK1.2 的时候，由于类集框架的引用，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 Vector，所以让这个类多实现了一个 List 接口。

```
package cn.mldn.de1;
import java.util.List;
```

```
import java.util.Vector;
public class TestDemo {
    public static void main(String []args){
        //设置泛型可以保证集合里的元素都一致
        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子接口扩充了get方法，可以根据索引取得数据
        for(int x=0;x<all.size();x++){
            System.out.println(all.get(x));
        }
    }
}
```

面试题：请解释 ArrayList 与 Vector 的区别？

| No | 区别点 | **ArrayList（90%）** | Vector |
|---|---|---|---|
| 1 | 推出时间 | JDK1.2 | JDK1.0 |
| 2 | 性能 | 异步处理 | 同步处理 |
| 3 | 数据安全 | 非线程安全 | 线程安全 |
| 4 | 输出 | 支持 Iterator、ListIterator、foreach | 支持 Iterator、ListIterator、foreach、Enumeration |

在以后的开发之中，如果使用了 List 子接口，就使用 ArrayList 子类。

总结：

1. List 中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 选择子类就使用 ArrayList。

## Set 子接口

1. Set 子接口的操作特点及常用子类；
2. 深入分析两个常用子类的操作特征。

在 Collection 接口下，又有一个常用子接口 Set（20%）。Set 接口并不像 List 接口那样对 Collection 接口进行了方法扩充，而是简单了继承了 Collection 接口。也就没有了之前 List 集合所提供了 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.de1;
import java.util.HashSet;
import java.util.Set;
public class TestDemo {
    public static void main(String []args){
        Set<String> all=new HashSet<String>();
```

```
        all.add("NIHAO");
        all.add("hello");
        all.add("hello");//重复数据
        all.add("world");
        System.out.println(all);
    }
}
```

通过演示可以发现，Set 集合下没有重复元素，同时在里面所保存的数据是无序的。即 HashSet 的子类特征是无序排列。

范例：使用 TreeSet

```
package cn.mldn.de1;
import java.util.Set;
import java.util.TreeSet;
public class TestDemo {
    public static void main(String []args){
        Set<String> all=new TreeSet<String>();
        all.add("X");
        all.add("B");
        all.add("B");//重复数据
        all.add("A");
        all.add("C");
        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 子类的保存的内容可以自动排序，那么下面就编写一个自定义的类来完成数据的保存。

```
package cn.mldn.de1;
import java.util.Set;
import java.util.TreeSet;
class Book{
    private String title;
    private double price;
    public Book(String title,double price){
        this.title=title;
        this.price=price;
    }
    @Override
    public String toString() {
        return "书名: "+this.title+",价格: "+this.price+"\n";
    }
}
public class TestDemo {
    public static void main(String []args){
        Set<Book> all=new TreeSet<Book>();
        all.add(new Book("java开发",88.9));
        all.add(new Book("java开发",88.9));//全部信息重复
```

```
        all.add(new Book("jsp开发",88.9));//价格信息重复
        all.add(new Book("oracle开发",68.9));//都不重复
        System.out.println(all);
    }
}
```

结果：

```
Exception in thread "main" java.lang.ClassCastException: cn.mldn.de1.Book cannot be cast to java.lang.Comparable
        at java.util.TreeMap.put(TreeMap.java:542)
        at java.util.TreeSet.add(TreeSet.java:238)
        at cn.mldn.de1.TestDemo.main(TestDemo.java:20)
```

集合就是一个动态的对象数组，如果要想为一组数据进行排序，在 java 里必须要使用比较器，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。

```
package cn.mldn.de1;
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;
    }
    @Override
    public String toString() {
        return "书名: "+this.title+",价格: "+this.price+"\n";
    }
    @Override
    public int compareTo(Book o) {
        if(this.price>o.price){
            return 1;
        }else if(this.price<o.price){
            return -1;
        }else{
        return 0;//认为两个对象是一个对象
        }
    }
}
public class TestDemo {
    public static void main(String []args){
        Set<Book> all=new TreeSet<Book>();
        all.add(new Book("java开发",88.9));
        all.add(new Book("java开发",88.9));//全部信息重复
        all.add(new Book("jsp开发",88.9));//价格信息重复
        all.add(new Book("oracle开发",68.9));//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 compareTo（）方法判断是否是重复数据，如果返回是 0，那么它就认为是重复数据，不会保存。

**正确程序代码：**

```
package cn.mldn.de1;
```

```
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;
    }
    @Override
    public String toString() {
        return "书名: "+this.title+",价格: "+this.price+"\n";
    }
    @Override
    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);//认为两个对象是一个对象
        }
    }
}
public class TestDemo {
    public static void main(String []args){
        Set<Book> all=new TreeSet<Book>();
        all.add(new Book("java开发",88.9));
        all.add(new Book("java开发",88.9));//全部信息重复
        all.add(new Book("jsp开发",88.9));//价格信息重复
        all.add(new Book("oracle开发",68.9));//都不重复
        System.out.println(all);
    }
}
```

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，它并不是真正的进行重复元素验证的操作。如果要想判断重复元素，只能用 Object 类中所提供的方法：

1. 取得哈希码：public int hashCode()；
   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；
2. 对象比较：public boolean equals(Object obj)。
   再将对象的属性进行依次的比较。

范例：代码

```
package cn.mldn.de1;
import java.util.HashSet;
import java.util.Set;
class Book {
    private String title;
    private double price;
    public Book(String title,double price){
        this.title=title;
```

```
        this.price=price;
    }
    @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;
    }
    @Override
    public String toString() {
        return "书名: "+this.title+",价格: "+this.price+"\n";
    }
}
public class TestDemo {
    public static void main(String []args){
        Set<Book> all=new HashSet<Book>();
        all.add(new Book("java开发",88.9));
        all.add(new Book("java开发",88.9));//全部信息重复
        all.add(new Book("jsp开发",88.9));//价格信息重复
        all.add(new Book("oracle开发",68.9));//都不重复
        System.out.println(all);
    }
}
```

以后在非排序的情况下，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 的理论范围内，例如要进行 TreeSet；
3. Set 不管如何操作，必须始终保持一个前提：数据不能重复。

## 集合输出

Collection、List 和 Set 接口里只有 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的形式。

集合在 JDK1.8 之前支持 4 中输出：**Iterator（95%）**、ListIterator、**Enumeration（4.9%）**、foreach。

**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，首先来观察一下 Iterator 接口的定义。

```
public interface Iterator<E>{
    public boolean hasNext()
    public E next()
}
```

Iterator 本身是一个接口，如果要想取得本接口实例化只能依靠 Collection 接口，在 Collection 接口里定义有如下的操作方法：public Iterator<E> iterator();

范例：使用 Iterator 输出集合

```
package cn.mldn.de1;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class TestDemo {
    public static void main(String []args){
        Set<String> all=new HashSet<String>();
        all.add("小马");
        all.add("美好");
        all.add("美好");
        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的大脑思考，直接使用 Iterator 就行啦。

**双向迭代：ListIterator（了解）**

Iterator 本身只具备由前向后输出（99%），但是有些人认为应该让其支持双向输出，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。

public interface ListIterator<E>；

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.de1;
import java.util.ArrayList;
import java.util.List;
```

```
import java.util.ListIterator;
public class TestDemo {
    public static void main(String []args){
        List<String> all=new ArrayList<String>();
        all.add("小");
        all.add("马");
        all.add("美");
        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，一定要先发生由前向后的输出。

**Foreach 输出**

理论上说 foreach 的输出还是挺方便的，但是如果你们过多使用 foreach 不利于理解程序。本身可以方便的输出数组，集合也可以。

```
package cn.mldn.de1;
import java.util.ArrayList;
import java.util.List;
public class TestDemo {
    public static void main(String []args){
        List<String> all=new ArrayList<String>();
        all.add("小");
        all.add("马");
        all.add("美");
        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 的时候推出的输出接口，即：最早的 Vector 如果要想输出，就使用 Enumeration 接口。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 子类里定义有如下的操作方法：

取得 Enumeration 接口对象：public Enumeration<E> elements();

```
package cn.mldn.de1;
import java.util.Enumeration;
import java.util.Vector;
public class TestDemo {
    public static void main(String []args){
        Vector<String> all=new Vector<String>();
        all.add("小");
        all.add("马");
        all.add("美");
        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，此接口仍然会使用到，所以必须掌握。

**总结：**

虽然集合的操作提供有四种形式，但是请一定要以 Iterator 和 Enumeration 为主，并且一定要注意到这两个接口的核心操作方法。

## Map 接口

Collection 每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（public interface Map<K,V>）里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 找到对应的 value** |
| **3** | **public Set<Map.Entry<K,V>> entrySet()** | **普通** | **将 Map 集合转化为 Set** |
| 4 | public Set<K> keySet() | 普通 | 取出全部的 key |

在 Map 接口下有两个常用子类：HashMap、HashTable。

范例：观察 HashMap 的使用。

```
package cn.mldn.de1;
import java.util.HashMap;
import java.util.Map;
public class TestDemo {
    public static void main(String []args){
        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);
```

```
    }
}
```

通过以上可以发现：

1. 使用 HasshMap 定义的 Map 集合是无序存放的（顺序无用）；
2. 如果发现了重复的 key 会进行覆盖，使用新的内容替换掉旧的内容。

在 Map 接口里提供有 get（）方法，这个方法的主要功能是根据 key 查找需要的 value。

范例：查询操作。

```
package cn.mldn.de1;
import java.util.HashMap;
import java.util.Map;
public class TestDemo {
    public static void main(String []args){
        Map<String,Integer> map=new HashMap<String,Integer>();
        map.put("壹", 1);
        map.put("贰", 2);
        map.put("叁", 3);
        map.put(null, 33);
        System.out.println(map);
        System.out.println(map.get("壹"));
        System.out.println(map.get("陆"));//如果key不存在，返回null
        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.de1;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
public class TestDemo {
    public static void main(String []args){
        Map<String,Integer> map=new HashMap<String,Integer>();
        map.put("壹", 1);
        map.put("贰", 2);
        map.put("叁", 3);
        map.put(null, 33);
        Iterator<String> iter=map.keySet().iterator();
        while(iter.hasNext()){
            System.out.print(iter.next()+"、");
        }
    }
}
```

在 Map 接口下还有 HashTable 子类，此类实在 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.de1;
import java.util.Hashtable;
import java.util.Map;
```

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

发现 HashTable 里面对于 key 和 value 的数据都不许设为 null。

面试题：请解释 HashMap 与 HashTable 的区别？

| No | 区别点 | HashMap（**90%**） | HashTable |
|---|---|---|---|
| 1 | 推出时间 | JDK1.2 | JDK1.0 |
| 2 | 性能 | 异步处理 | 同步处理 |
| 3 | 数据安全 | 非线程安全 | 线程安全 |
| 4 | 设置 null | 允许 key 或 value 为 null | Key 或 value 都不允许为空 |

而在实际的应用之中，基本上就使用 HashMap。

**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 接口里并没有定义任何的可以返回 Iterator 接口对象的方法。如果要想使用 Iterator 输出 Map 集合，首先要针对于 Map 集合与 Collection 集合保存数据的特点进行分析后才能实现。

每当用户使用 put（）方法向 Map 集合里面保存一对数据的时候，实际上所有的数据都会被自动的封装为 Map.Entry（内部接口）接口对象。那么来观察 Map.Entry 接口定义：

```
public static interface Map.Entry<K,V>
```

Stati 定义的内部接口就是外部接口。

在这个接口里定义了两个操作：

取得 key：public K getKey()；

取得 value：public V getValue()。

在 Map 接口里面定义有一个将 Map 接口转化为 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.de1;
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){
        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，那么也就表示可以使用自定义的类型作为 key。那么这个作为 key 的自定义的类必须要覆写 Object 类中的 hashCode（）与 equals（）两个方法，因为只有靠这两个方法才能够确定元素是否重复，在 Map 中指的是是否能找到。

在以后使用 Map 集合的时候，首选的 key 类型是 String，尽量不要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，而 Collection 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；
3. HashMap 可以保存 null，key 重复会出现覆盖。

## Stack 子类

了解 Stack 的使用。

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 的子类，可是它不会使用 Vector 的方法，操作方法只有两个：

入栈：public E push(E item)；

出栈：public E pop()。

范例：观察栈的操作（了解）

```
package cn.mldn.de1;
import java.util.Stack;
public class TestDemo {
    public static void main(String []args){
        Stack<String> st=new Stack<String>();
        st.push("A");
        st.push("B");
        st.push("C");
        System.out.println(st.pop());
        System.out.println(st.pop());
        System.out.println(st.pop());
        System.out.println(st.pop());//EmptyStackException
    }
}
```

在栈操作的过程之中，如果栈已经没有数据了，就无法继续出栈。

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。

## Properties 子类

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“*.Proterties”。

Properties 类的操作子类。

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 类里面主要使用如下的操作方法：

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。

```
package cn.mldn.de1;
import java.util.Properties;
public class TestDemo {
    public static void main(String []args){
        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("SH"));
        System.out.println(pro.getProperty("SH", "不存在该城市！"));
    }
}
```

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.de1;
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，后缀就必须是
*.properties,这样做的目的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.de1;
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（资源）文件，除了可以使用 Properties 类读取外，还可以使用 ResourceBundle 类读取，这也就是将输出的属性文件后缀统一设置为”*.properties”的原因。

总结：

1.资源文件的特点：

Key=value

Key=value

2.资源文件中的数据一定都是字符串，其他的意义不大。

# Collections 工具类

了解 Collections 类的功能。

在 java 提供类库的时候考虑到用户使用的方便性，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、Set 和 Map 的操作。

为集合追加数据：public static <T> boolean addAll(Collection<? super T> c,T... elements)；

反转：public static void reverse(List<?> list);

```
package cn.mldn.de1;
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");
        Collections.reverse(all);
        System.out.println(all);
    }
}
```

面试题：请解释 Collection 与 Collections 的区别？

1. Collection 是集合操作的接口；
2. Collections 是集合操作的工具类，可以进行 Set、List、Map 集合的操作。

总结：这个工具类不会用到，知道就行了。

## Stream(JDK1.8)

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。

   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);

# Java 数据库编程（JDBC）

## JDBC 简介

在 java 语言设计的时候除了考虑到了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，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，只有多写才能记下来。

JDBC（java database connective），java 数据库的连接技术，即：由 java 提供的一组与平台无关的数据库操作标准。（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的连接。

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，指的是开放数据库连接，是有微软提供的数据库的连接应用，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，性能是最差的，支持的版本是最新的；

形式二：JDBC 技术直接连接；

|-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 的数据库操作标准的一群类），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 版本不见得是最新的。

**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定能都是很高的，但是只有一个数据库性能是最差的：SQL server。

在国内使用使用最多的数据库：oracle、MySQL、MongoDB。

# 连接 Oracle 数据库

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。

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、ResuleSet、PreparedStatement。

**所有 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、ResultSet、PreparedStatement）；**

**4.关闭数据库操作以及连接（直接关闭连接就行）。**

## 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：D:\app\Administrator\product\11.2.0\client_1\jdbc\lib\ojdbc6.jar；

步骤：右键点 Properties → Java Build Path → Librarie → Add External JARS。

Oracle 驱动程序类：oracle.jabc.driver.OracleDviver。

加载类使用：Class.forName(“oracle.jabc.driver.OracleDviver”);

## 二、连接数据库

如果要想连接数据库，需要提供几个有如下的几个信息（前提，数据库服务要打开）：

|-数据库的连接地址：jdbc:oracle:连接方式:主机名称:端口名称:数据库的 SID

|-**url=jdbc\:oracle\:thin**:@60.175.166.136\:1521\:cboxcz

|-数据库的用户名；

|-数据库的密码。

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

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。

## 三、

## 四、 关闭数据库

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

```
package clinbox.cn.net;
import java.sql.Connection;
import java.sql.DriverManager;
public class TestDemo {
   private static final String
DBDRIVER="oracle.jdbc.driver.OracleDriver";
   private static final String
DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
   private static final String USER="scott";
```

```
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        Connection con=DriverManager.getConnection(DUBRL, USER,
PASSWORD);
        System.out.println(con);
        con.close();
    }
}
```

可是很多时候是连接不上 Oracle 数据库的。

可能的原因：

1. 修改计算机名：

2.监听服务出现错误（可能性很高）；

|-监听的主机名称不是本机的计算机名称，也不要使用 IP 地址（有可能是动态的）；

|-监听配置文件路径：D:\app\Administrator\product\11.2.0\client_1\network\admin，这里面有两个文件监听文件：listener.ora;和监听的名称文件：tnsnames；

如果监听有问题，那么将出现如下的错误提示：

解决方法时修改这两个文件的主机名称跟当前的主机保持一致，然后重启监听。

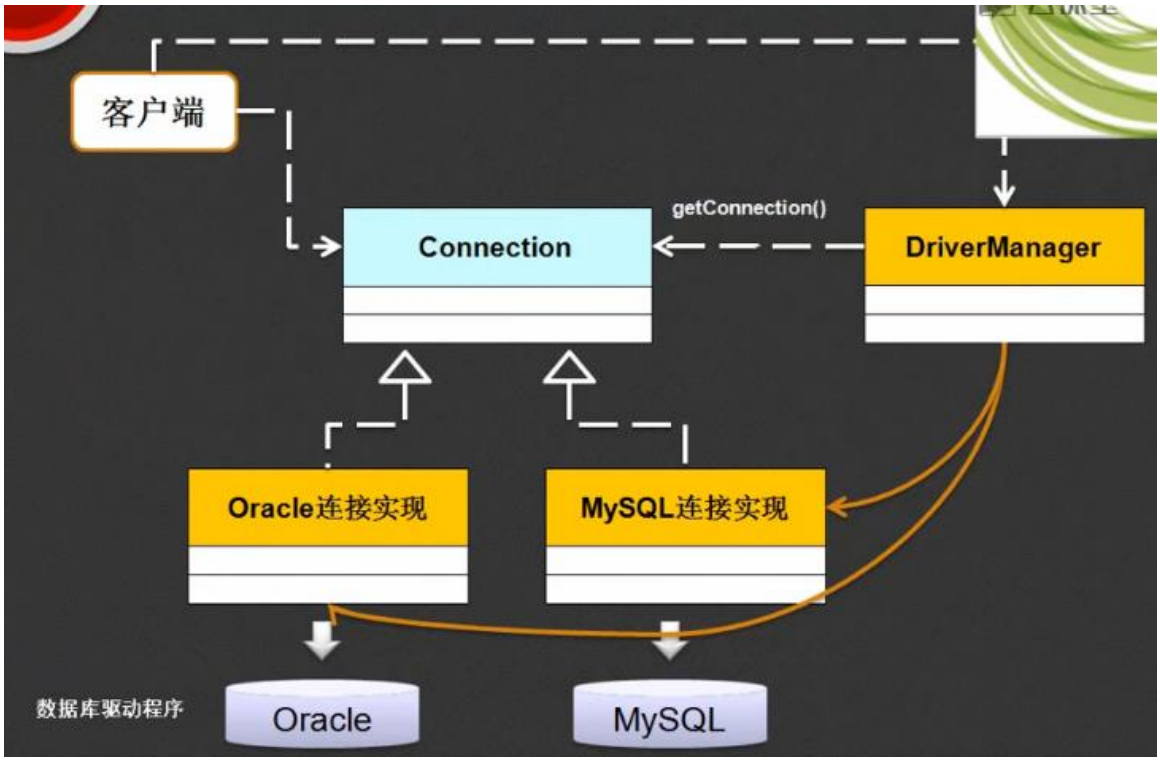
3.不能够找到指定的 SID：

数据库的名字就是 SID 的名字，但是很多时候你会发现该名称不会自动注册，也就是说只有数据库名称没有对应的 SID 名称。于是可以打开数据库的网络管理工具：开始菜单→ oracle → 配置和移植工具 → Net Manager；然后选择：本地 → 监听程序 → LISTENER → 数据库服务，然后如果没有服务的话，选择添加，数据库名称与 SID 名称一致，然后文件，保存配置。

结构总结：

通过以上的操作可以发现，整个数据库进行连接操作的时候，都是按照同样的步骤进行:

|-DriverManager 取得 Connection 接口对象；




原来 JDBC 在操作之中，在驱动数据库连接对象时，采用的工厂设计模式，而 DriverManager 就是一个工厂类。客户端在调用的时候会完全隐藏具体的实现子类。

总结：

1. 以后不管连接何种关系型数据库，都一定通过 DriverManager 进行数据库连接；
2. 每一个 Connection 接口对象就表示一个数据库连接，程序最后必须关闭数据库连接。

## Statement 接口

利用 Statement 接口实现数据表的更新与查询操作。

当取得了数据库连接对象之后，那么实际上就意味着可以进行数据库操作了，可以使用最简单的 Statement 接口完成。

那么如果想要取得 Statement 接口的实例化对象则需要依靠 Connection 接口提供的方法完成。

取得 Statement 接口对象：public Statement createStatement() throws SQLException；

当取得 Statement 接口对象之后可以使用以下两个方法实现数据库操作：

|-数据更新：public int executeUpdate(String sql) throws SQLException；返回更新行数；

|-数据查询：public ResultSet executeQuery(String sql) throws SQLException。

范例：编写数据库创建脚本。

```
DROP TABLE member PURGE;
DROP SEQUENCE myseq;
CREATE SEQUENCE myseq;
CREATE TABLE member(
                        mid             NUMBER,
                        name            VARCHAR2(20),
                        birthday        DATE    DEFAULT SYSDATE,
                        age             NUMBER(3),
                        note            CLOB,
                        constraint    pk_mid    PRIMARY KEY(mid)
```

```
);
```

以上有序列，mid 通过序列生成，序列操作“nextval”伪列取得下一个增长的数值。

**1. 数据更新操作。**

数据更新操作最关键的问题是每次更新完成之后都一定会返回影响的数据行数。

INSERT INTO 表名称 （列，列，……） VALUES (值，值，……)；

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class TaxRate {
   private static final String
DBDRIVER="oracle.jdbc.driver.OracleDriver";
   private static final String
DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
   private static final String USER="scott";
   private static final String PASSWORD="tiger";
   public static void main(String[] args) throws Exception {
       //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
       Class.forName(DBDRIVER);
       Connection con=DriverManager.getConnection(DUBRL, USER,
PASSWORD);
       Statement stmt=con.createStatement();
       String sql="INSERT INTO member (mid,name,birthday,age,note)
VALUES (myseq.nextval,'张三',To_Date('1998-10-10',yyyy-mm-dd),17,'是
个人')";
       int len=stmt.executeUpdate(sql);
       System.out.println("影响的数据行"+len);
       con.close();
   }
}
```

范例：数据的修改

范例：删除操作。

2、数据查询

每当使用 SELECT 查询，实际上会将所有的查询结果返回给用户显示，而显示的基本结构就是表的形式。可是如果要进行查询，这些查询的结果应该返回给程序，由用户来进行处理。那就必须有一种类型能够接收所有的返回结果。在数据库里，虽然可能有几百张数据表，但是整个数据表的组成数据类型都是固定的，所以在 ResultSet 在设计的过程之中它是按照数据类型的方式来保存返回数据的。

在 ResultSet 接口里面定义了如下的方法：

- 向下移动指针并判断是否有数据行：boolean next() throws SQLException：
  |-移动之后就可以直接取得当前数据行中所有数据列的内容了。
- 取出数据列的内容：getInt()、getDouble()、getString()、getDate()。

范例：实现数据的查询。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Date;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
```

```
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        Statement stmt=con.createStatement();
        String sql="SELECT mid,name,birthday,age,note FROM member"; //不允许写*，需要哪个查询哪
个。
        ResultSet rs=stmt.executeQuery(sql);
        while(rs.next()){
            int mid=rs.getInt("mid");
            String name=rs.getString("name");
            Date birthday=rs.getDate("birthday");
            int age=rs.getInt("age");
            String note=rs.getString("note");
            System.out.println(mid+","+name+", "+birthday+","+age+","+note);
        }
        con.close();
    }
}
```

关于 ResultSet 的使用，有如下几点忠告：

1. 在使用 getXxx（）取出数据的时候，强烈建议按照给定的顺序取。
2. 每一列的数据都按照顺序只能取一次。

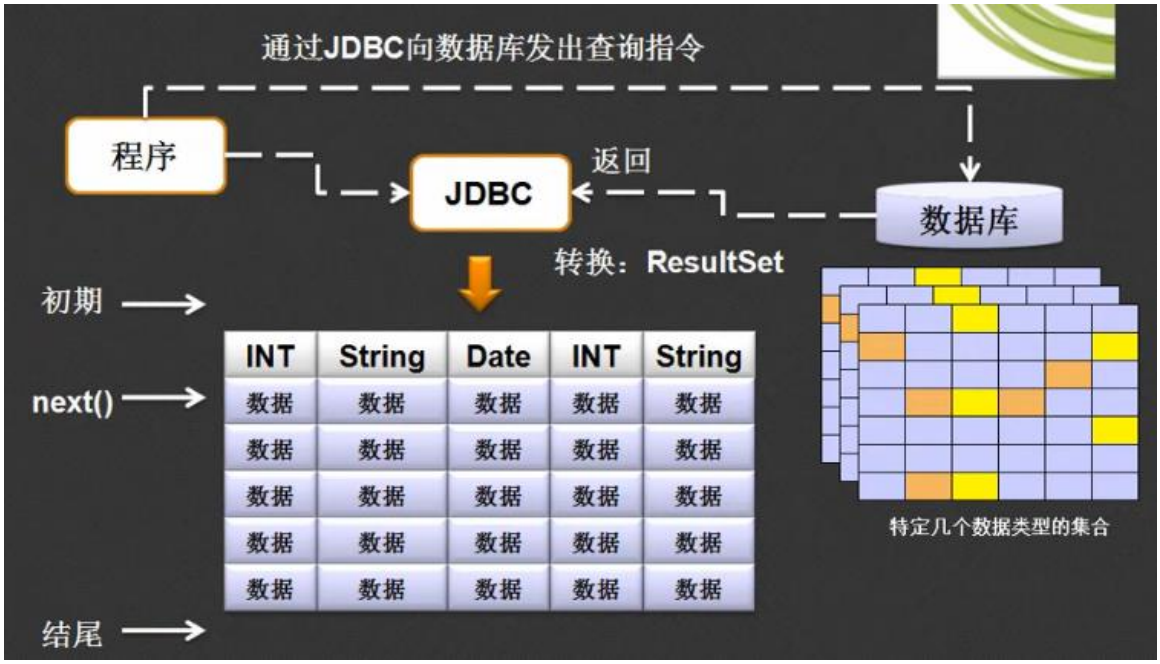
以上的代码在取出内容的时候重复编写了列名称，实际上这一点可以忽略，因为在写 SQL 语句的时候就给出了列名称，可以按照序号取出。

```
while(rs.next()){
            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);
        }
```

既然已经可以执行查询了，那么就可以继续添加各种复杂查询。

总结：

通过演示可以发现，Statement 接口的执行步骤几乎都是固定的，并且最方便的是可以直接执行 SQL 的字符串操作。但是任何的开发都不可能使用 Statement，以后都要去使用 PreparedStatement 接口。




# PreparedStatement 接口

**预计讲解的知识点**

将分析 Statement 接口操作的不足，以及通过各种实例讲解 PreparedStatement 接口的使用。本次讲解是 JDBC 操作中最重要的内容。

**具体内容**

虽然 JDBC 里提供有 Statement 接口，但是从实际来讲，Statement 接口存在有严重的操作缺陷，是不可能在实际的工作中使用的。

范例：以数据增加为例，观察 Statement 的问题。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        String name="Mr'SMITH";
        String birthday="1998-10-10";
        int age=18;
        String note="是个外国人";
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        Statement stmt=con.createStatement();
        String sql="INSERT INTO member (mid,name,birthday,age,note) VALUES
(myseq.nextval,'"+name+"',To_Date('"+birthday+"',yyyy-mm-dd),"+age+",'"+note+"')";
        System.out.println(sql);
        int len=stmt.executeUpdate(sql);
```

```
        System.out.println("影响的数据行"+len);
        con.close();
    }
}
```

发现报错。

Statement 如果想要变为灵活的应用，那么就必须采用拼凑字符串的方式完成，可是如果输入的内容存在有单引号',那么整个 sql 就会出错，也就是说 Statement 的执行模式不适合处理一些敏感字符。

PreparedStatement 操作

Statement 执行关键性的问题是在于它需要一个完整的字符串来定义要使用的 sql 语句，所以这就导致在使用中药进行大量的 SQL 的拼凑。而 PreparedStatement 与 Statement 不同的地方在于，它执行的是一个完整的具备特殊占位标记的 SQL 语句，并且可以动态的设置所需要的数据。

PreparedStatement 属于 Statement 的一个子接口，但是要想取得这个子接口的实例化对象，依然需要使用 Connection 接口提供的方法:Public PreparedStatement prepareStatement(String sql) throws SQLException，里面在执行的时候需要传入一个 SQL 语句，这个 SQL 是一个具备特殊标记的完整的 SQL，但是此时没有内容，当取得了 PreparedStatement 接口对象后，需要使用一系列的 setXxx（）方法，用于为所使用的标记设置具体内容。而后对于执行操作，有两个方法支持。

更新操作：public int executeUpdate()  throws SQLException；

查询操作：public ResultSet executeQuery() throws SQLException。

当使用 PreparedStatement 接口操作时，最需要注意的是里面的 setDate（）方法，因为此方法使用的是 java.sql.Date，而不再是 java.util.Date。

在 java.util.Date 类下有三个子类都是在 java.sql 包中的。

|-Java.sql.Date:描述的是日期。

|-Java.sql.Time:描述的是时间。

|-Java.sql.Timestamp:描述的是时间戳（包含日期时间）。

如果要将 java.util.Date 变为 java.sql.Date(Time,Timestamp)，只能够依靠 long 完成。

|-java.util.Date:public long getTime()，可以将 Date 变为 long。

|-java.sql.Date:public Date(long Date),将 long 变为 java.sql.Date。

范例：改进数据增加

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.util.Date;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        String name="Mr'SMITH";
        Date birthday=new Date();
        int age=18;
        String note="是个外国人";
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        //第二步：连接数据库
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        //在编写sql的过程里面，如果太长的时候需要加换行，那么请一定记住前后加上空格
        String sql=" INSERT INTO member (mid,name,birthday,age,note) " +
                " VALUES (myseq.nextval,?,?,?,?) ";
        //第三步：进行数据库的数据操作，执行了完整的SQL
```

```
            PreparedStatement stmt=con.prepareStatement(sql);
            stmt.setString(1,name);
            stmt.setDate(2,new java.sql.Date(birthday.getTime()));
            stmt.setInt(3,age);
            stmt.setString(4,note);
            System.out.println(sql);
            int len=stmt.executeUpdate();
            System.out.println("影响的数据行"+len);
            //第四步：关闭数据连接
            con.close();
        }
}
```

更新与删除操作和增加操作的区别不大。

整个数据库操作里面，更新操作几乎都是固定的模式，但是查询操作是最为复杂的。

范例：查询全部数据

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Date;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        //第二步：连接数据库
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        //在编写sql的过程里面，如果太长的时候需要加换行，那么请一定记住前后加上空格
        String sql="SELECT mid,name,birthday,age,note FROM member ORDER  BY mid";
        //第三步：进行数据库的数据操作，执行了完整的SQL
        PreparedStatement stmt=con.prepareStatement(sql);
        ResultSet rs=stmt.executeQuery();
        while(rs.next()){
            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);
        }
        int len=stmt.executeUpdate();
        System.out.println("影响的数据行"+len);
        //第四步：关闭数据连接
        con.close();
    }
}
```

范例：模糊查询：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
import java.util.Date;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        String keyword="李";
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        //第二步：连接数据库
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        //在编写sql的过程里面，如果太长的时候需要加换行，那么请一定记住前后加上空格
        String sql="SELECT mid,name,birthday,age,note FROM member where name=? ORDER  BY mid";
        //第三步：进行数据库的数据操作，执行了完整的SQL
        PreparedStatement stmt=con.prepareStatement(sql);
        stmt.setString(1,"%"+keyword+"%");
        ResultSet rs=stmt.executeQuery();
        while(rs.next()){
            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);
        }
        int len=stmt.executeUpdate();
        System.out.println("影响的数据行"+len);
        //第四步：关闭数据连接
        con.close();
    }
}
```

范例：分页显示

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Date;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        int currentpage=1;
        int linesize=5;
        String keyword="李";
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        //第二步：连接数据库
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        //在编写sql的过程里面，如果太长的时候需要加换行，那么请一定记住前后加上空格
        String sql=" SELECT * FROM " +
                " (SELECT mid,name,birthday,age,note,ROWNUM rn FROM member where name LIKE ? AND
```

```
ROWNUM <=? ) temp " +
                    " WHERE temp.rn>? ";
          //第三步：进行数据库的数据操作，执行了完整的SQL
          PreparedStatement stmt=con.prepareStatement(sql);
          stmt.setString(1,"%"+keyword+"%");
          stmt.setInt(2, currentpage*linesize);
          stmt.setInt(3,(currentpage-1)*linesize);
          ResultSet rs=stmt.executeQuery();
          while(rs.next()){
               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);
          }
          int len=stmt.executeUpdate();
          System.out.println("影响的数据行"+len);
          //第四步：关闭数据连接
          con.close();
     }
}
```

范例：统计数据量

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.Date;
public class TaxRate {
     private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
     private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
     private static final String USER="scott";
     private static final String PASSWORD="tiger";
     public static void main(String[] args) throws Exception {
          String keyword="李";
          //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
          Class.forName(DBDRIVER);
          //第二步：连接数据库
          Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
          //在编写sql的过程里面，如果太长的时候需要加换行，那么请一定记住前后加上空格
          String sql=" SELECT COUNT(mid) FROM member where name LIKE ? ";
          //第三步：进行数据库的数据操作，执行了完整的SQL
          PreparedStatement stmt=con.prepareStatement(sql);
          stmt.setString(1,"%"+keyword+"%");
          ResultSet rs=stmt.executeQuery();
          if(rs.next()){
               int count=rs.getInt(1);
               System.out.println(count);
          }
          int len=stmt.executeUpdate();
          System.out.println("影响的数据行"+len);
          //第四步：关闭数据连接
          con.close();
     }
}
```

以上所讲解的为 PreparedStatement 接口操作的核心功能，必须掌握。

总结：

在以后的任何开发中永远不会使用 Statement 接口，只会是 PreparedStatement 接口。

# 批处理与事物处理

**预计知识点**

1、 掌握批处理的操作流程；

2、 掌握 JDBC 提供的事物处理操作。

**具体内容**

在之前所使用的数据库操作严格来讲是属于 JDBC1.0 中就规定的操作模式，而最新的 JDBC 是 4.0，但是没人去用它。从 JDBC2.0 开始，增加了一些神奇的功能：可滚动的结果集、可以利用结果集执行增加、更新、删除操作、批处理操作。

所谓批处理操作是一次性向数据库之中发出多条操作命令，一起执行。如果要想操作批处理，主要还是在 Statement 与 PreparedStatement 接口上定义的。

- Statement 接口定义的方法：
  |-增加批处理语句：public void addBatch(String sql) throws SQLException；
  |-执行批处理：public int[] executeBatch() throws SQLException；
  |-返回的数组是包含了所有批处理语句的执行结果；
- PreparedStatement 接口定义的方法：
  |-增加批处理语句：public void addBatch() throws SQLException；
  |-

范例：执行批处理

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Arrays;
import java.util.Date;
public class TaxRate {
                        private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
                        private               static               final               String
DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
                        private static final String USER="scott";
                        private static final String PASSWORD="tiger";
                        public static void main(String[] args) throws Exception {
                          String keyword="李";
                          //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管
理
                          Class.forName(DBDRIVER);
                          //第二步：连接数据库
                          Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
                          //第三步：进行数据库的数据操作，执行了完整的 SQL
                          Statement stmt=con.createStatement();
                          stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试
A')");
                          stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试
B')");
                          stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试
C')");
```

```
                stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试
D')");
                stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试
E')");
                stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试
F')");
                int rs[]=stmt.executeBatch();//执行批处理
                System.out.println(Arrays.toString(rs));
                //第四步：关闭数据连接
                con.close();
            }
}
```

如果假设以上的批处理属于一组关联的操作，其中一条语句执行失败，其他的不应该成功。

修改以上代码，使其中一条出现错误（出现单引号即报错），发现，在批处理操作的过程中，由于 JDBC 具有自动的事物提交，所以一旦中间的语句出现了错误，那么结果就是错误前的语句正常执行，错误后的语句执行不了，很明显这不应该。

可以使用 JDBC 提供的事物处理操作来进行手工的事物控制，所有的操作方法都在 Connection 接口里定义：

|-事物提交：public void commit() throws SQLException；

|-事物回滚：public void rollback() throws SQLException；

|-设置是否为自动提交：public void setAutoCommit(boolean autoCommit) throws SQLException。

范例：利用事物处理

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Arrays;
import java.util.Date;
public class TaxRate {
    private static final String DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String DUBRL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String USER="scott";
    private static final String PASSWORD="tiger";
    public static void main(String[] args) throws Exception {
        String keyword="李";
        //第一步：加载数据库驱动程序，此时不需要实例化，因为还有容器自己负责管理
        Class.forName(DBDRIVER);
        //第二步：连接数据库
        Connection con=DriverManager.getConnection(DUBRL, USER, PASSWORD);
        //第三步：进行数据库的数据操作，执行了完整的SQL
        Statement stmt=con.createStatement();
        con.setAutoCommit(false);//取消自动提交
        try{
        stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试A')");
        stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试B')");
        stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试C')");
        stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试D')");
        stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试E')");
        stmt.addBatch("INSERT INTO member(mid,name) VALUES (myseq.nextval,'测试F')");
        int rs[]=stmt.executeBatch();//执行批处理
        System.out.println(Arrays.toString(rs));
        con.commit();//如果没有错误，进行提交
```

```
            }catch(Exception e){
                e.printStackTrace();
                con.rollback();//如果出现异常，则进行回滚
            }
            //第四步：关闭数据连接
            con.close();
        }
}
```

以上的只是演示一下如何手工处理事物，而在实际的工作上，这些操作意义不大，会由日后的容器帮我们自动处理。

**总结：**

掌握批处理的执行操作。

Connection 接口定义了事物的处理方法：commit（），rollback（），setAutoCommit(boolean autoCommit)。

# DAO 设计模式—设计分层初步

## 分层设计思想

**预计知识点**

1、 理解程序设计分层的思想

2、 DAO 设计模式的组成以及各部分的开发。

**具体内容（核心）**

本次讲解之中，除了 IO 部分暂时不会用到之外所有 java 的重点的核心部分都会采用到。

1、 程序的分层

实际上在任何的环境下分层的概念都会存在，例如：在公司里面可以按照职位分层。每一层的开发都是完全独立的，并且可以与其他层进行完整的交互。

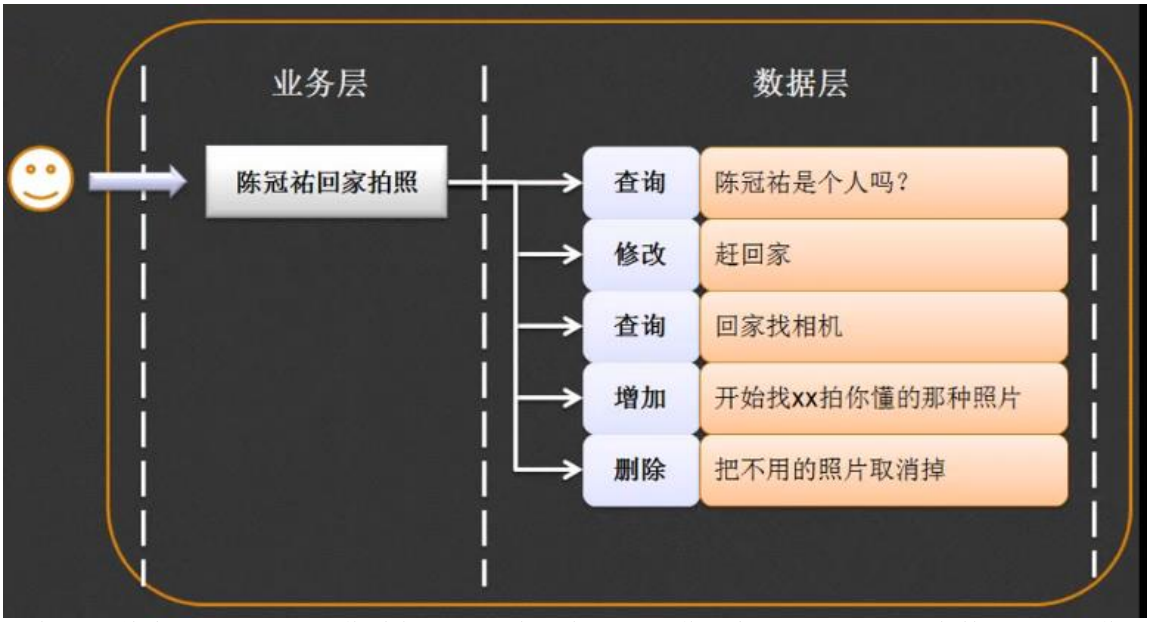
现在以人类交谈为例做一个简单的分层：

- 大脑作为所有信息存储的单位存在，可以理解为数据层；
- 依靠语言或各个行为模式将分散的数据组合在一起；
- 还是需要有一些辅助性的外表的支持。

如果要程序进行简单的划分那么最常见的划分方式：显示层+控制层+业务层+数据层+数据库。


在整个项目中，后台业务是最为核心的部分。因为现在移动应用的火爆问题，所以对于前台层已经不再单独局限于一个简单的 Web 层了，而是可能用 Android、IOS，而且随着技术的发展，对于前台的开发，还可能不使用 java 了，使用 Python 或者是 Node.JS 进行包装。

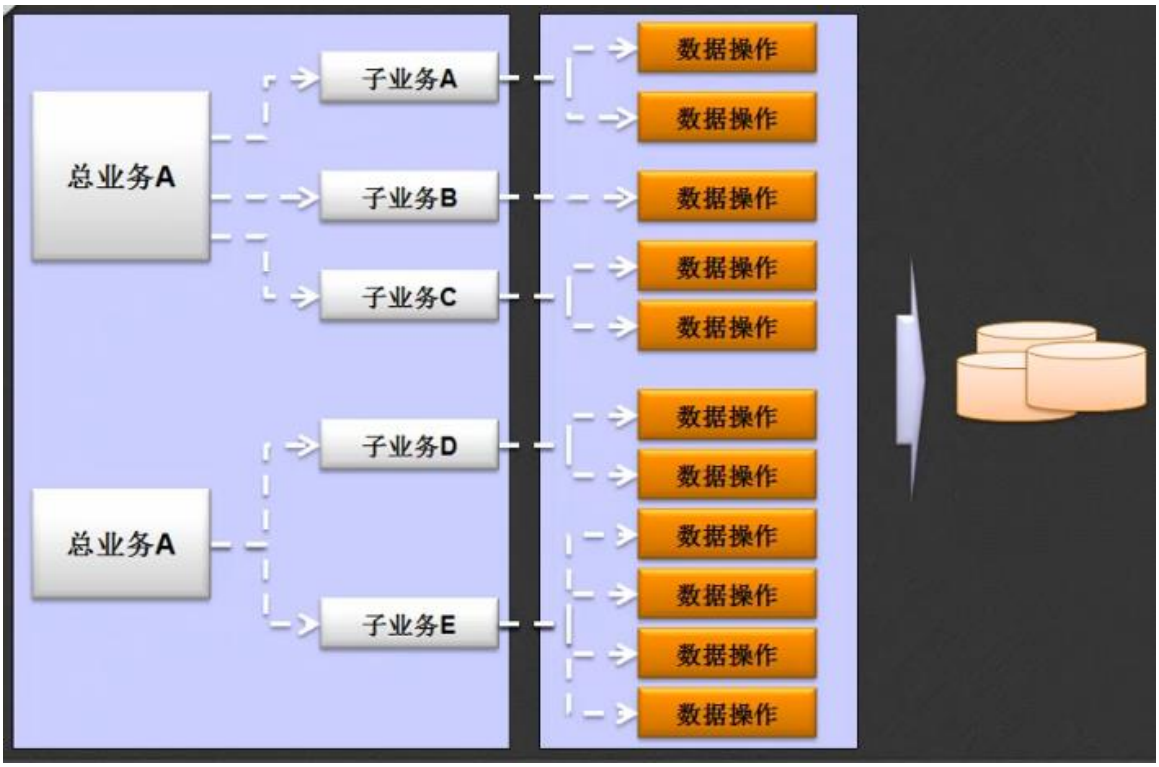
那么既然整个项目的核心是后台业务层，那么什么叫业务？什么叫数据呢？以“陈冠佑回家拍照”这个操作为例，




业务层是整个程序提供的操作功能，而一个业务层的操作要想完成，需要多个数据层的操作一起完成。

整个过程之中，数据层完成的只是一个个原子性的数据库开发。而在实际的开发之中，每个业务往往需要牵扯到多个原子性的操作，也就是说所有原子性的操作业务最终在业务层中完成。

在实际的开发之中，业务的设计是非常复杂的，本次的操作只是简单的区分了业务层与数据层基础关系。而如果业务非常复杂，那么往往需要一个总业务层，而后会牵扯到若干个子业务层，每一个子业务层又去执行多个数据层。




数据层：又被称为数据访问层（Data Access Object，DAO），是专门进行数据库的原子性操作，也就是说在数据层之中最需要控制的就是 JDBC 中的 PreparedStatement 接口的使用。

业务层：又被称为业务对象（Business Object，BO），但是现在又有一部分人认为应该称为服务层（Serveice），业务层核心的目的是调用多个数据层的操作，以完成整体的项目的业务设计，这是整个项目的核心所在。

# 业务设计实例

现在要求使用 emp 表（empno，ename，job，hiredate，sal，comm）实现如下的操作功能：

- 【业务层】实现雇员数据的添加，但是保证添加雇员编号不会重复；
  |-〖数据层〗判断要增加的雇员编号是否存在；
  |-〖数据层〗如果雇员编号不存在，则进行数据的保存操作；
- 【业务层】实现雇员数据的修改操作；
  |-〖数据层〗执行数据的修改操作；
- 【业务层】实现雇员数据的删除操作；
  |-〖数据层〗执行雇员的限定删除操作；
- 【业务层】可以根据雇员编号查找到雇员的信息；
  |-〖数据层〗根据雇员编号查询指定的雇员数据；
- 【业务层】可以查询所有雇员的信息；
  |-〖数据层〗查询全部雇员数据；
- 【业务层】可以实现数据的分页显示，同时又可以返回所有的雇员数量。
  |-〖数据层〗雇员数据的分页查询；
  |-〖数据层〗使用 COUNT（）函数统计出所有的雇员数量。

结论：用户所提出的所有的需求都应该划分为业务层，因为它指的是功能，而开发人员必须要根据业务层去进行数据层的设计。

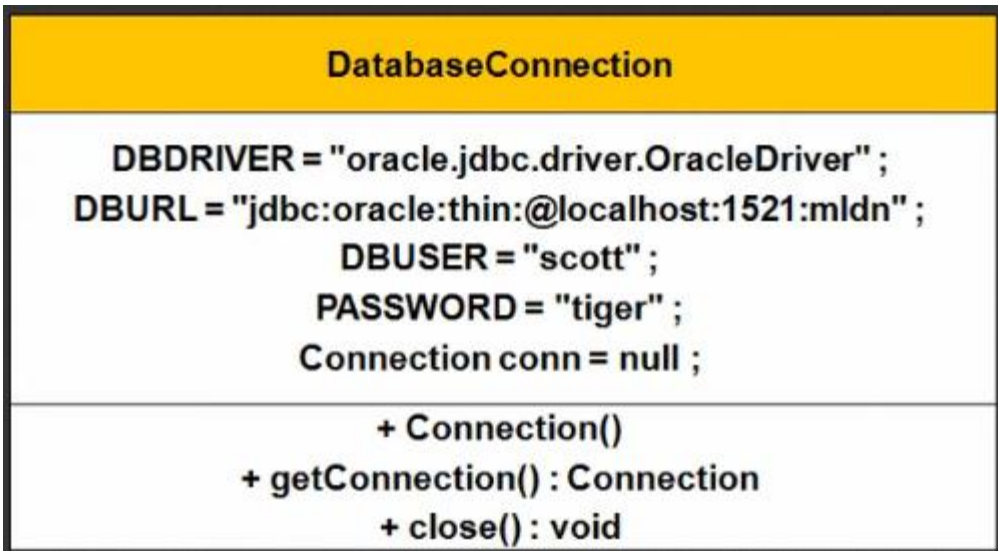
# DAO 设计模式—开发准备

首页可以设置一个项目名称：DAOProject，并且由于此项目需要使用 Oracle 数据库，需要为其配置好数据库的驱动程序。请保证数据库已打开监听与实例服务。

为了方便的进行程序的统一管理，所有的项目的父包名称统一设定为：cn.mldn。而子包要根据不同的功能模块进行划分。

# 数据库连接类

本次的操作既然要进行数据库的开发，那么就必须进行数据库的连接取得与关闭才可以正常操作，那么几乎所有的数据的连接操作都是固定的步骤，那么就可以单独定义一个 DatabaseConnection 类，这个类主要负责数据库连接对象的取得以及数据库的关闭工作。既然是一个专门用于数据库的连接操作，那么可以将其保存在 dbc 子包中。




范例：定义数据库的连接类

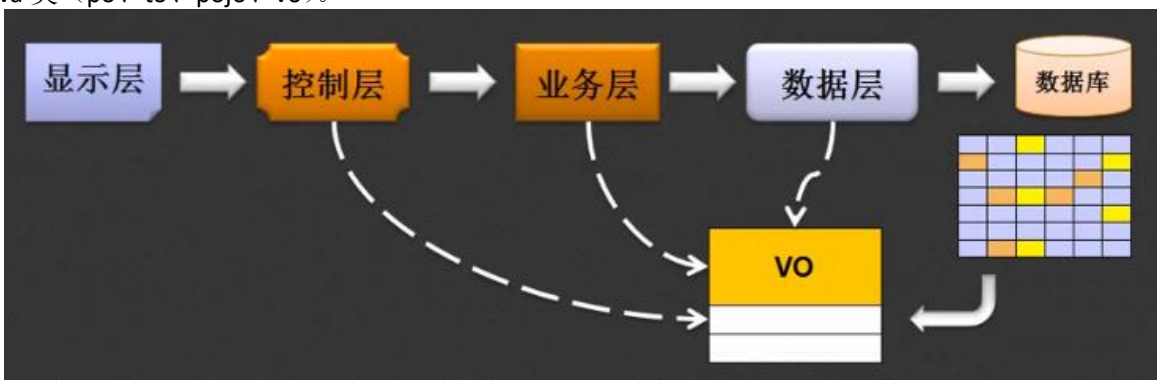
```
package cn.mldn.dbc;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
/**
 * 本类专门负责数据库的连接与关闭操作，在实例化本类对象时就意味着要进行数据库的开发
 * 所以在本类的构造方法里要进行数据库驱动加载与数据库连接取得
 * @author Administrator
 *
 */
public class DatabaseConnection {
    private static final String  DBDRIVER="oracle.jdbc.driver.OracleDriver";
    private static final String  DBURL="jdbc:oracle:thin:@localhost:1521:mldn";
    private static final String  DBUSER="scott";
    private static final String  DBPASSWORD="tiger";
    private Connection conn=null;
    /**
     * 在构造方法里面为conn对象进行实例化，可以直接取得数据库的连接对象<br>
     * 由于所有的操作都是基于数据库完成的，如果数据库取得不到连接，那么也就意味着所有的操作都可以停止了
     * @throws ClassNotFoundException
     */
    public DatabaseConnection(){
        try {
            Class.forName(DBDRIVER);
            this.conn=DriverManager.getConnection(DBURL, DBUSER,DBPASSWORD);
        } catch (Exception e) {//虽然此处有异常，但是抛出的意义不大
            e.printStackTrace();
        }
    }
    /**
     * 取得数据库的连接对象
     * @return Connection的实例化对象
     */
    public Connection getConnection(){
        return this.conn;
```

```
    }
    /**
     * 负责数据库的关闭
     */
    public void close(){
        if(this.conn!=null){
            try {
                this.conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

整个操作过程之中，DatabaseConnection 只是无条件的提供有数据库连接，而至于说有多少个线程需要找到此类要连接对象，它都不关心。

# 开发 ValueObject 类

现在的程序严格来讲已经给出了四个层次。不同层次之间一定要进行数据的传递，但是既然要操作的是指定的数据表，所以数据的结构必须要与表的结构一一对应，那么自然就可以想到简单 java 类（po、to、pojo、vo）。




（利用简单 java 类将需要的数据层连接起来，可以将数据传递到需要的数据层上。）

**在实际的工作中，针对简单 java 类的开发给出如下要求：**

- **考虑到日后程序有可能出现的分布式应用问题，所有的简单 java 类必须要实现 java.io.Serialziable 接口；**
- **简单 java 类的名称必须与表名称保持一致；**
  **|-实际上表名有可能采用这样的名字：student_info，那么类名称为 StudentInfo；**
- **类的属性不允许使用基本数据类型，都必须使用基本数据类型的包装类；**
  **|-基本数据类型的默认数值是 0，而包装类默认值是 null；**
- **类中的属性必须使用 private 封装，封装后的属性必须有 setter 和 getter 方法。**
- **类中可以定义有多个构造方法，但是必须要保留一个无参构造方法；**
- 【可选要求，基本不用】覆写 equals（）、toString（）、hashCode（）。

将所有的简单 java 类保存在 vo 包中。

范例:定义 Emp.java 类

新建 Emp.java 类，属于 cn.mldm.vo 包，实现 java.io.Serialziable 接口。

注意类中的属性使用包装类定义。

```
package cn.mldn.vo;
import java.io.Serializable;
```

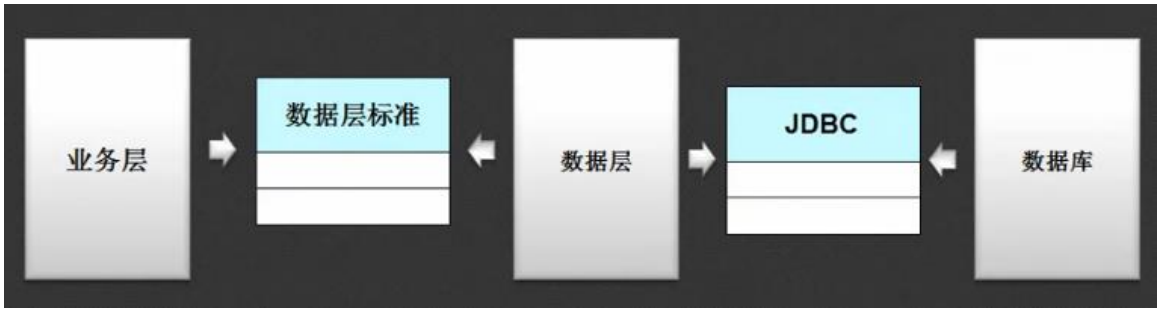
```
import java.util.Date;
public class Emp implements Serializable {
    private Integer empno;
    private String ename;
    private String job;
    private Date hiredate;
    private Double sal;
   private Double comm;
   //setter 和 getter 略
}
```

不管有多少张表，只要是实体表，一定要写简单 java 类，而且不要试图想着一次性将所有的表都转换到位。

# DAO 设计模式—数据层开发

## 数据层接口

数据层最中是交给业务层调用的，所以业务层必须知道数据的执行标准。即：业务层需要明确知道数据层的操作方法，但是不需要知道它的具体实现。




（不同层之间如果要进行访问，需要一个标准。）

不同层之间如果要进行访问，那么必须要提供接口，以定义操作标准，那么对于数据层也是一样的，数据层最终交给业务层，所以需要定义数据层接口。

**对于数据层接口，给出如下开发要求：**

- **数据层既然是进行数据操作的，那么就将其保存在 dao 包下；**
- **既然不同的数据表的操作有可能使用不同的数据层开发，那么就根据数据表进行命名。**
  **|-例如 emp 表，那么数据层接口就应该定义为：IEmpDAO。**
- **对于整个数据层的开发，严格来讲就只有两类功能：**
  **|-数据的更新：建议它的操作方法以 doXxx（）的形式命名，例如：doCreate（）、doUpdate（）、doRemove（）。**
  **|-数据的查询：对于查询分为两种形式：**
  **|-查询表中数据：以 findXxxx（）形式命名，例如：findByid()、findByName（）、findAll（）；**
  **|-统计表中数据：以 getXxx（）形式命名，例如：getAllCount（）。**

范例：定义 IEmpDAO 接口。

```
package cn.mldn.dao;

import java.util.List;
import java.util.Set;
```

```
import cn.mldn.vo.Emp;
/**
 * 定义emp表的数据层的操作标准
 * @author Administrator
 *
 */
public interface IEmpDAO {
    /**
     * 实现数据的增加操作
     * @param vo 包含了要增加数据的VO对象
     * @return 数据保存成功返回true，否则返回false
     * @throws Exception SQL执行异常
     */
    public boolean doCreate(Emp vo) throws Exception;
    /**
     * 实现数据的修改操作，本次修改是根据id进行全部字段数据的修改
     * @param vo 包含了要修改数据的信息，一定要提供有id内容
     * @return 数据保存成功返回true，否则返回false
     * @throws Exception SQL执行异常
     */
    public boolean doUpdate(Emp vo) throws Exception;
    /**
     * 执行数据的批量删除操作，所有要删除的数据以Set集合的形式保存
     * @param ids 包含了所有要删除的id，不包含有重复内容
     * @return 数据保存成功返回true（删除的数据个数与要删除的数据个数相同），否则返回false
     * @throws Exception SQL执行异常
     */
    public boolean doRemoveBatch(Set<Integer> ids) throws Exception;
    /**
     * 根据雇员编号查询指定的雇员信息
     * @param id 要查询的雇员编号
     * @return 如果雇员信息存在，则将数据以VO类对象的形式返回，如果不存在则返回null
     * @throws Exception SQL执行异常
     */
    public Emp finById(Integer id) throws Exception;
    /**
     * 查询指定数据表的全部记录，并且以集合的形式返回
     * @return 如果表中有数据，则所有数据会封装为VO对象而后利用List集合返回，<br>
     * 如果没有数据，那么集合的长度为0，size()==0，不是null
     * @throws Exception SQL执行异常
     */
    public List<Emp> finAll()throws Exception;
    /**
     * 分页进行数据的模糊查询，查询结果以集合的形式返回
     * @param currentPage 表示当前所在的页
     * @param lineSize 每页所显示的数据行数
     * @param column 要进行模糊查询的数据列
     * @param keyWord 模糊查询的关键字
     * @return 如果表中有数据，则所有数据会封装为VO对象而后利用List集合返回，<br>
     * 如果没有数据，那么集合的长度为0，size()==0，不是null
     * @throws Exception SQL执行异常
     */
    public List<Emp> findAllSplit(Integer currentPage,Integer lineSize,String column,String
keyWord)throws Exception;
    /**
```

```
     * 进行模糊查询数据量的统计
     * @param column 要进行模糊查询的数据列
     * @param keyWord 模糊查询的关键字
     * @return 返回表中的数据量，如果没有返回0
     * @throws Exception SQL执行异常
     */
    public Integer getAllCount(String column,String keyWord) throws Exception;
}
```


# 数据层类实现

数据层要被业务层调用，数据层需要进行数据库的执行（PreparedStatement），由于在开发之中，一个业务层操作，需要执行多个数据层的调用，所以数据库的打开和关闭操作应该由业务层控制会比较合理。

所有的数据层实现类要求保存在 dao.impl 子包下。

范例：EmpDAOImpl 子类。

在 cn.mldn.dao.impl 子包下；

实现 IEmpDAO 接口；

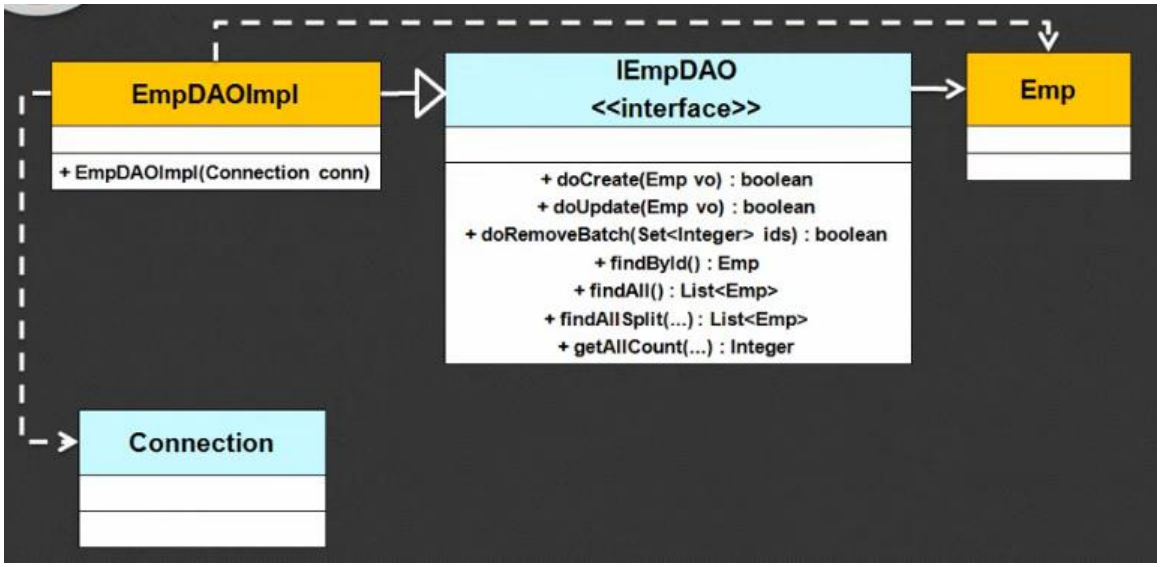
```
package cn.mldn.dao.impl;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

import cn.mldn.dao.IEmpDAO;
import cn.mldn.vo.Emp;

public class EmpDAOImpl implements IEmpDAO {
                        private Connection conn;//需要利用 Connection 对象操作
                        private PreparedStatement pstmt;
                        /**
                         *  如果想要使用数据层进行原子性的功能操作实现，必须要提供有 Connection
```

```
接口对象<br>
                         * 另外由于开发之中业务层要调用数据层，所以数据库的打开与关闭交由业务层
处理
                         * @param conn 表示数据库连接对象
                         */
                        public EmpDAOImpl(Connection conn){
                          this.conn=null;
                        }
                        public boolean doCreate(Emp vo) throws Exception {
                          String sql="INSERT INTO emp(empno,ename,job,hiredate,sal,comm) VALUES
(?,?,?,?,?,?)";
                          this.pstmt=this.conn.prepareStatement(sql);
                          this.pstmt.setInt(1,vo.getEmpno());
                          this.pstmt.setString(2,vo.getEname());
                          this.pstmt.setString(3, vo.getJob());
                          this.pstmt.setDate(4, new java.sql.Date(vo.getHiredate().getTime()));
                          this.pstmt.setDouble(5, vo.getSal());
                          this.pstmt.setDouble(6, vo.getComm());
                          return this.pstmt.executeUpdate()>0;
                        }

                        public boolean doRemoveBatch(Set<Integer> ids) throws Exception {
                          if(ids==null||ids.size()==0){//没有要删除的数据
                               return false;
                          };
                          StringBuffer sql=new StringBuffer();
                          sql.append("DELETE FROM emp WHERE empno IN (" );
                          Iterator<Integer> iter=ids.iterator();
                          while(iter.hasNext()){
                               sql.append(iter.next()).append(",");
                          }
                          sql.delete(sql.length()-1, sql.length());
                          sql.append(")");
                          this.pstmt=conn.prepareStatement(sql.toString());
                          return this.pstmt.executeUpdate()==ids.size();
                        }

                        public boolean doUpdate(Emp vo) throws Exception {
                          String sql="UPDATE  emp SET ename=?,job=?,hiredate?,sal?,comm=? WHERE
empno=?";
                          this.pstmt=this.conn.prepareStatement(sql);
                          this.pstmt.setString(1,vo.getEname());
                          this.pstmt.setString(2, vo.getJob());
                          this.pstmt.setDate(3, new java.sql.Date(vo.getHiredate().getTime()));
                          this.pstmt.setDouble(4, vo.getSal());
                          this.pstmt.setDouble(5, vo.getComm());
                          this.pstmt.setInt(6,vo.getEmpno());
                          return this.pstmt.executeUpdate()>0;
                        }

                        public List<Emp> finAll() throws Exception {
                          String sql="SELECT empno,ename,job,hiredate,sal,comm FROM emp";
                          this.pstmt=conn.prepareStatement(sql);
                          ResultSet rs=this.pstmt.executeQuery();
                          List<Emp> ls=new ArrayList<Emp>();
```

```
                while(rs.next()){
                    Emp vo=new Emp();
                    vo.setEmpno(1);
                    vo.setEname(rs.getString(2));
                    vo.setJob(rs.getString(3));
                    vo.setHiredate(rs.getDate(4));
                    vo.setSal(rs.getDouble(5));
                    vo.setComm(rs.getDouble(6));
                    ls.add(vo);
                }
                return ls;
            }


            public Emp finById(Integer id) throws Exception {
                Emp vo=null;
                String sql="SELECT empno,ename,job,hiredate,sal,comm FROM emp WHERE
empno=?";
                this.pstmt=conn.prepareStatement(sql);
                this.pstmt.setInt(1, id);
                ResultSet rs=this.pstmt.executeQuery();
                if(rs.next()){
                    vo=new Emp();
                    vo.setEmpno(1);
                    vo.setEname(rs.getString(2));
                    vo.setJob(rs.getString(3));
                    vo.setHiredate(rs.getDate(4));
                    vo.setSal(rs.getDouble(5));
                    vo.setComm(rs.getDouble(6));
                }
                return vo;
            }

            public List<Emp> findAllSplit(Integer currentPage, Integer lineSize,
                    String column, String keyWord) throws Exception {
                String sql=" SELECT * FROM " +
                        " (SELECT empno,ename,job,hiredate,sal,comm,ROWNUM rn FROM emp
WHERE "+column+" LIKE ? AND ROWNUM <=?) temp " +
                        " WHERE temp.rn>?";
                this.pstmt=conn.prepareStatement(sql);
                this.pstmt.setString(1,"%"+keyWord+"%");
                this.pstmt.setInt(2, currentPage*lineSize);
                this.pstmt.setInt(3, (currentPage-1)*lineSize);
                ResultSet rs=this.pstmt.executeQuery();
                List<Emp> ls=new ArrayList<Emp>();
                while(rs.next()){
                    Emp vo=new Emp();
                    vo.setEmpno(1);
                    vo.setEname(rs.getString(2));
                    vo.setJob(rs.getString(3));
                    vo.setHiredate(rs.getDate(4));
                    vo.setSal(rs.getDouble(5));
                    vo.setComm(rs.getDouble(6));
                    ls.add(vo);
                }
```

```
            return ls;
        }

        public Integer getAllCount(String column, String keyWord) throws Exception
{
            String sql="SELECT COUNT(*) FROM emp WHERE ? LIKE ?";
            this.pstmt=conn.prepareStatement(sql);
            this.pstmt.setString(1, column);
            this.pstmt.setString(2, "%"+keyWord+"%");
            ResultSet rs=this.pstmt.executeQuery();
            if(rs.next()){
                return rs.getInt(1);
            }
            return 0;
        }
}
```

子类对象唯一要注意的一点就是构造方法一定要接收一个 Connection 对象。



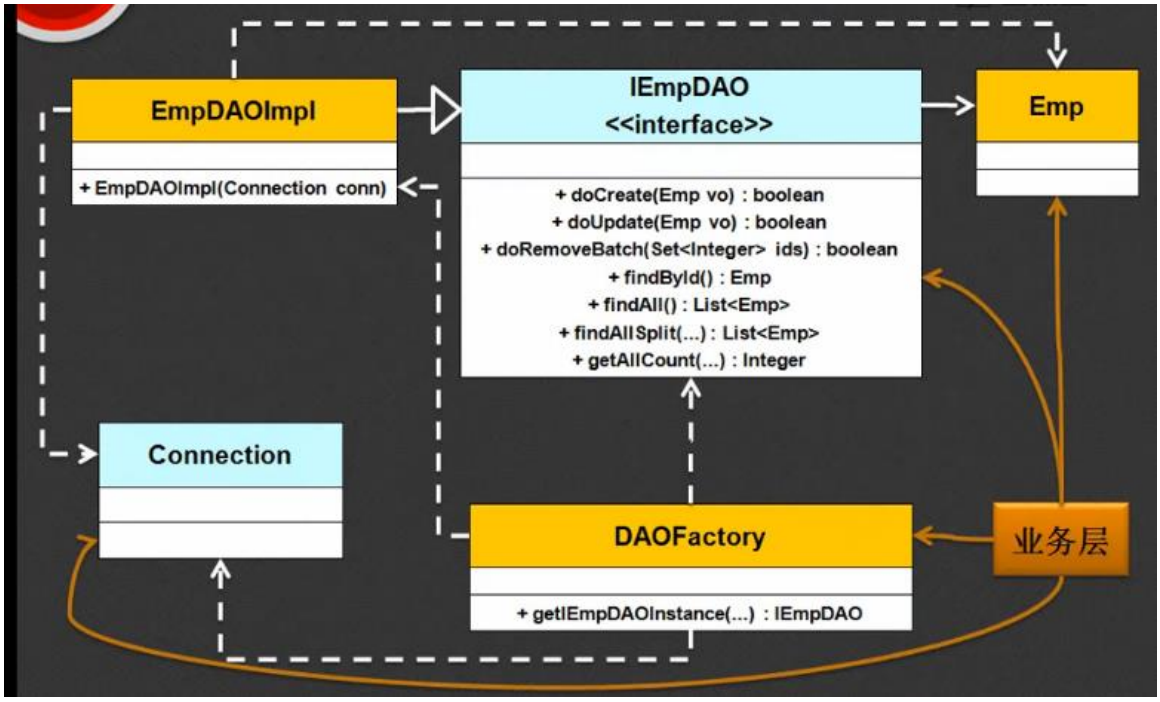

# 数据层工厂类

业务层要想进行数据层的调用，那么必须要取得 IEmpDAO 接口对象，但是不同层之间要想取得接口对象实例需要使用工厂设计模式。这个工厂类，将保存在 factory 子包下。

范例：定义工厂类。

```
package cn.mldn.factory;

import java.sql.Connection;

import cn.mldn.dao.IEmpDAO;
import cn.mldn.dao.impl.EmpDAOImpl;

public class DAOFactory {
    public static IEmpDAO getIEmpDAOInstance(Connection conn){
        return new EmpDAOImpl(conn);
    }
}
```

使用工厂的特征是外层不需要知道具体的子类。




# DAO 设计模式--业务层开发

## 业务层接口

业务层是真正留给外部调用的，可能是控制层，或者是直接调用，既然业务层也是由不同的层进行调用的，所以业务层开发的首要任务就是定义业务层的操作标准—IempService。

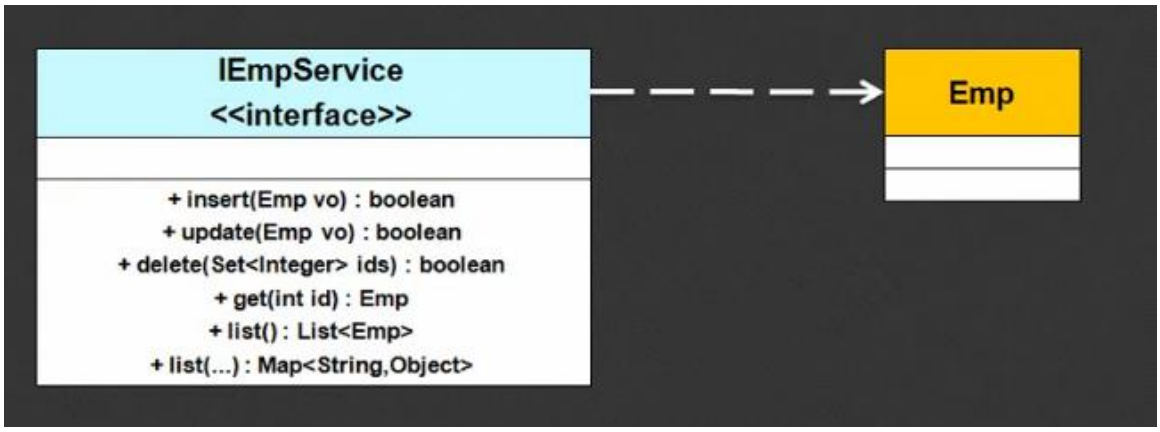
业务层也可以说是 Service 层，既然是描述的 emp 表的操作，所以名称就定义为 IEmpService，并且保存在 Service 子包下。但是对于业务层的方法并没有明确的要求。强烈建议还是写上有意义的统一名称。

范例：定义 IEmpService 操作标准。

```
package cn.mldn.service;

import java.util.List;
import java.util.Map;
import java.util.Set;

import cn.mldn.vo.Emp;
//业务员层用基本类型，数据层用包装类完成
/**
 * 定义emp表的执行业务标准,此类一定要负责数据库的打开与关闭操作
 * 此类可以通过DAOFactory类取得IEmpDAO接口对象
 * @author Administrator
 *
 */
public interface IEmpService {
    /**
     * 实现雇员数据的增加操作，本次操作要调用IEmpDAO接口的如下方法：<br>
```

```
     * <li> 需要调用IEmpDAO.findByID()f方法，判断要增加的id是否已经存在；<br>
     * <li> 如果要增加的数据编号不存在，则调用IEmpDAO.create()方法，返回操作的结果；
     * @param vo 包含了要增加数据的vo对象
     * @return 如果增加ID重复或保存失败返回false，否则返回true
     * @throws Exception SQL执行异常
     */
    public boolean insert(Emp vo)throws Exception;
    /**
     * 实现雇员数据的修改操作，本次操作调用IEmpDAO.doUpdate()方法，本次修改属于全部内容的修改；
     * @param vo 包含了本次修改数据的vo对象
     * @return 修改成功返回true，否则返回false
     * @throws Exception SQL执行异常
     */
    public boolean update(Emp vo)throws Exception;
    /**
     * 执行雇员数据的删除操作，可以删除多个雇员的信息，调用IEmpDAO.doRemoveBatch()方法
     * @param ids 包含了全部要删除的数据的集合，没有重复的数据
     * @return 删除成功返回true，否则返回false
     * @throws Exception SQL执行异常
     */
    public boolean delete(Set<Integer> ids)throws Exception;
    /**
     * 根据雇员编号查找雇员的完整信息，调用IEmpDAO.findById()方法
     * @param id 要查找的雇员编号
     * @return 如果找到了以vo对象返回，否则返回null
     * @throws Exception SQL执行异常
     */
    public Emp get(Integer id) throws Exception;
    /**
     * 查询全部雇员信息，调用IEmpDAO.findAll()方法
     * @return 查询结果以List集合的形式返回，如果没有数据则List集合的长度为0
     * @throws Exception SQL执行异常
     */
    public List<Emp> list()throws Exception;
    /**
     * 实现数据的模糊查询与统计，要调用IEmpDAO接口的两个方法：<br>
     * <li> 调用IEmpDAO.findAllSplit(),查询出所有的数据，返回List<Emp>;
     * <li> 调用IEmpDAO.getAllCount()，查询所有的数据量，返回Integer类型。
     * @param currentPage 当前所在页
     * @param lineSize 每页显示的记录数
     * @param column 模糊查询的数据列
     * @param keyWord 模糊查询的关键字
     * @return 本方法由于返回多种数据类型，    所以使用Map集合返回，由于类型不统一，所以value类型设
置为Object。返回内容如下：
     * <li> key=allEmps,value=IEmpDAO.findAllSplit()返回结果，List<Emp>;
     * <li> key=empCount,value=IEmpDAO.getAllCountf返回结果，Integer
     * @throws Exception SQL执行异常
     */
    public Map<String,Object> list(int currentPage,int lineSize,String column,String keyWord)throws
Exception;
}
```

本接口中方法的设计完全符合之前的分析过程。




# 业务层实现类

业务层实现类的核心功能：

- 负责数据库的打开和关闭，当存在了业务层对象后，其目的就是为了操作数据库，即业务层对象实例化之后就必须准备好数据库连接；
- 负责根据 DAOFactory 调用 getIEmpDAOInstance（）方法，而后取得 IEmpDAO 接口对象。

业务层的实现类保存在 dao.impl 子包中。

范例：定义 EmpServiceImpl 子类。

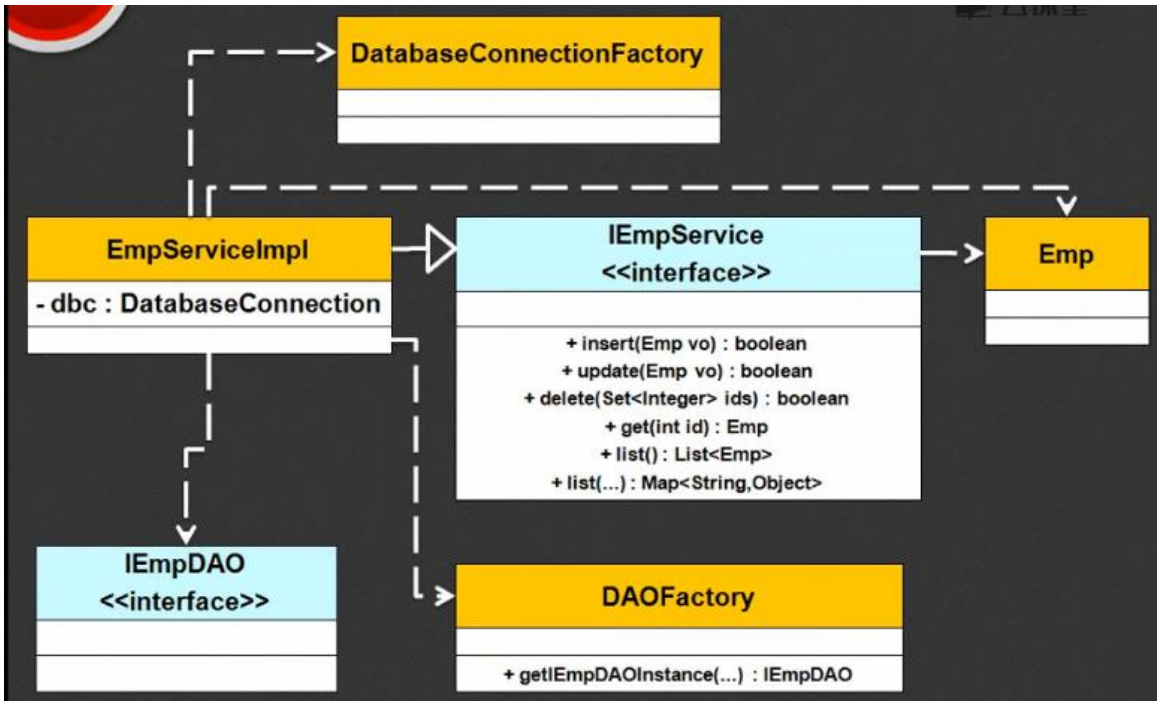
```
package cn.mldn.service.impl;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;

import cn.mldn.dbc.DatabaseConnection;
import cn.mldn.factory.DAOFactory;
import cn.mldn.service.IEmpService;
import cn.mldn.vo.Emp;

public class EmpServiceImpl implements IEmpService {
                        //在这个类的对象内部就提供有一个数据库连接类的实例化对象
                        private DatabaseConnection dbc=new DatabaseConnection();

                        public boolean delete(Set<Integer> ids) throws Exception {
                          try{
                               return
DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).doRemoveBatch(ids);
                          }catch(Exception e){
                               throw e;
                          }finally{
                               this.dbc.close();
                          }
                        }

                        public Emp get(Integer id) throws Exception {
                          try{
                               return
DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).findById(id);
```

```
                }catch(Exception e){
                    throw e;
                }finally{
                    this.dbc.close();
                }
            }

            public boolean insert(Emp vo) throws Exception {
                try{//要增加的雇员编号不存在，则 findById（）返回的结果就是 null，表示可以信息新雇员的保存

            if(DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).findById(vo.getEmpno())==null){
                        return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).doCreate(vo);
                    }
                    return false;
                }catch(Exception e){
                    throw e;
                }finally{
                    this.dbc.close();
                }
            }

            public List<Emp> list() throws Exception {
                try{
                    return DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).finAll();
                }catch(Exception e){
                    throw e;
                }finally{
                    this.dbc.close();
                }
            }

            public Map<String, Object> list(int currentPage, int lineSize,
                    String column, String keyWord) throws Exception {
                try{
                    Map<String,Object> map=new HashMap<String,Object>();
                    map.put("allEmps", DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).findAllSplit(currentPage, lineSize, column, keyWord));
                    map.put("empCount", DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).getAllCount(column, keyWord));
                    return map;
                }catch(Exception e){
                    throw e;
                }finally{
                    this.dbc.close();
                }
            }

            public boolean update(Emp vo) throws Exception {
                try{
                    return
```

```
DAOFactory.getIEmpDAOInstance(this.dbc.getConnection()).doUpdate(vo);
                            }catch(Exception e){
                                throw e;
                            }finally{
                                this.dbc.close();
                            }
                        }
}
```

不同层之间的访问，依靠的就是工厂类和接口进行操作。

（一般写了功能类和接口，一般使用工厂类）。



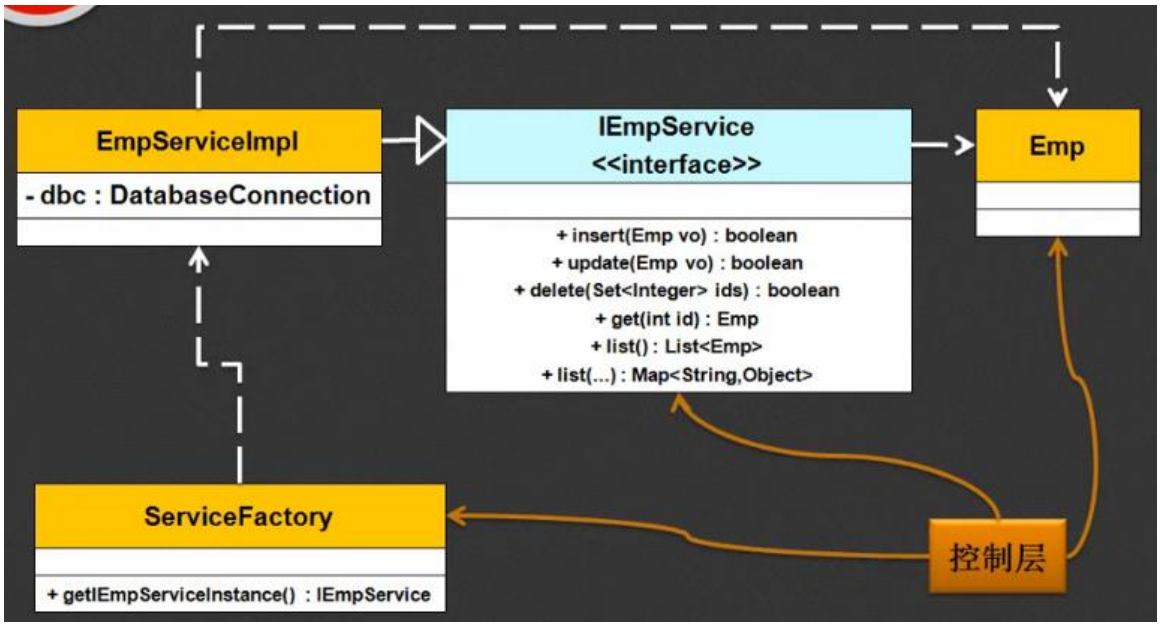

# 业务层工厂类

业务层依然需要背其他的层所使用，所以依然需要为其定义工厂类，该类同样保存在 factory 子包下。如果从实际的开发来讲，业务层应该分为两种：

- 前台业务逻辑：可以保存在 service.front 包中，工厂类：ServiceFrontFactory；
- 后台业务逻辑：可以将其保存在 service.back 包中，工厂类：ServiceBackFactory。

范例：定义 ServiceFactory。

```
package cn.mldn.service;

import cn.mldn.service.impl.EmpServiceImpl;

public class ServiceFactory {
    public static IEmpService getIEmpServiceInstance(){
        return new EmpServiceImpl();
    }
}
```




在实际的编写之中，子类永远都是不可见的，同时在整个操作里面，控制层完全看不到任何的数据库操作（没有任何的 JDBC 代码）。