Java程序设计

(下)

2016.10.24

isszym　sysu.edu.cn

抽象类和接口

- 当不能确定所有子类的行为（方法实现）时就可以使用抽象类。以后增加新的子类也不会影响根据该抽象类所编程序。
- 只要一个类有一个抽象方法(可以是继承来的)，则该类要定义为抽象类。抽象方法只有声明没有方法体。不能采用抽象类定义对象。

```
public abstract class ShapeAbs {          // 抽象类
    String color;                         // 变量或属性(field)
    public ShapeAbs() {                   // 构造函数(constructor)
        System.out.println("Shape Initialized!");
        color = "black";
    }
    public abstract void draw();          // 抽象函数或方法(method)

    public void setColor(String color) {  //方法: 设置颜色
        this.color = color;               //this.color表示本类的属性
    }

    public String getColor() {            //方法: 取出颜色
        return this.color;
    }
}
```

```
public class CircleA extends ShapeAbs {
   public CircleA() {        // 构造函数(constructor)
      System.out.println("CircleA Initialized!");
   }
   public void draw() {          // 定义方法draw()
      System.out.println("CircleA draw() is called!");
   }
}
public class RectangleA extends ShapeAbs {
   public RectangleA() {
      System.out.println("RectangleA Initialized!");
   }
   public void draw() {
      System.out.println("RectangleA draw() is called!");
   }
}
public class ShapeInherit {
   public static void main(String args[]){
      Circle circle = new Circle();
      circle.draw();

      Rectangle rectangle = new Rectangle();
      rectangle.draw();
   }
}
```

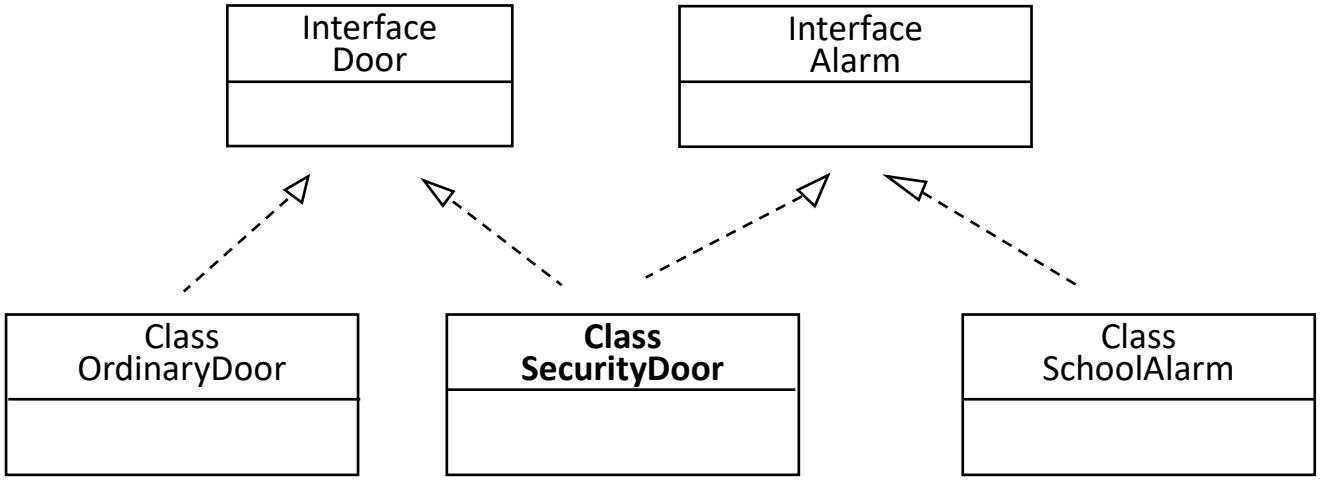
执行结果:

```
Shape Initialized!
CircleA Initialized!
CircleA draw() is called!
Shape Initialized!
RectangleA Initialized!
RectangleA draw() is called!
```

接口是更加抽象的一种类，接口的所有方法都只有声明而没有方法体。接口不能定义成员变量，但是可以定义常量。

```
public interface Door {
        void open();   // 开门
        void close();  // 关门
}
public class OrdinaryDoor implements Door {
        public void open(){
                System.out.println("open door!");
        }

        public void close(){
                System.out.println("close door!");
        }
}
public interface Alarm {
        void alarm();   // 拉响警报
}
public class SchoolAlarm implements Alarm {
        public void alarm(){
                System.out.println("alarm!");
        }
}
```

如果要实现一个安全门，就要用到多重继承。和C++不同，Java没有多重继承，不过多重继承通过实现多个接口来实现。




```
public class SecurityDoor implements Door,Alarm {
    public void open(){
        System.out.println("open door!");
    }

    public void close(){
        System.out.println("close door!");
    }

    public void alarm(){
        System.out.println("alarm!");
    }
}
```

通过接口提供的方法可以使用接口子类提供的功能。即使子类发生变化或增加了新的子类，只要接口不变，使用接口的模块也不用改变。

```
public class InterfaceTest {
   public static void main(String args[]){
      OrdinaryDoor door1 = new OrdinaryDoor();
      SchoolAlarm alarm1 = new SchoolAlarm();
      SecurityDoor door2 = new SecurityDoor();
      enter(door1);
      enter(door2);
      alarm(alarm1);
      alarm(door2);
    }
  public static void enter(Door door){
     door.open();
     door.close();
        ...
  }
  public static void alarm(Alarm alarm){
      ...
     alarm.alarm();
  }
}
```

一个类除了可以实现多个接口，还可以同时继承一个基类。例如：下面ClassA继承ClassB，实现了接口InterfaceA和InterfaceB。

```
class ClassA extends ClassB implements InterfaceA, InterfaceB  {

}
```

可以通过多个接口定义更大的接口:

```
public interface SecurityDoorInterface implements Door,Alarm {
    public void radio(); //除了继承Door和Alarm的方法，还增加了新方法radio()
}
```

可以通过Class定义接口:

```
public interface IntA extends ClassC  {

}
```

内部类

在类的内部定义的类为内部类，可以定义在类或类的方法内。内部类可以被内部方法所使用，并对外部实现了隐藏。在内部类中可以直接使用其它外部类。

```
class InClass {
     int x=0;
     class B {
        void f1(){
          x = 5;
        }
     }
     B getB(){
        return new B();
     }
     public static void main(String[] args){
         InClass a1= new InClass();
         System.out.println(""+a1.x); //输出0
         B b1 = a1.getB();
         b1.f1();
         System.out.println(""+a1.x);  //输出5
     }
}
```

try和throws

```
public class CatchError {
   public static void main(String args[]){
      int x;
      try {                                    // 打开例外处理语句
         for (int i = 5; i >= -2; i--) {
            x = 12 / i;                        // 出现例外(x==0)后将不执行后面的语句
            System.out.println("x=" + x);// 直接跳到catch内执行
         }
      }
      catch (Exception e) {                  // 捕捉例外信息。可以并列用多个catch
         System.out.println("Error:"+e.getMessage()); // 显示当前错误信息
         // e.printStackTrace();             // 显示系统错误信息
      }
      finally{                               // 出现例外必须执行这里的语句
          x=0;
      }
      System.out.println(x);

   }
}
```

```
执行结果:
x=2
x=3
x=4
x=6
x=12
Error:/ by zero
0
```

如果一个方法不愿意处理一些例外，可以采用throws定义方法的例外，把这些例外交给调用者去处理。

```
public class DivideClass {
    void divide() throws Exception {       // 出错后交给调用程序处理
      for (int i = 5; i >= -2; i--) {
         int x = 12 / i;                   // 出现例外(x==0)后将不执行后面的语句
         System.out.println("x=" + x);
      }
    }
}
public class ThrowError  {
   public static void main(String args[]){     // main为主程序入口
      try {                                     // 打开例外处理语句
           DivideClass div = new DivideClass();
           div.divide();
      }
      catch (Exception e) {                     // 捕捉例外信息。可以并列用多个catch
         System.out.println("Error:"+e.getMessage()); // 显示当前错误信息
      }
   }
}
```

执行结果：

```
x=2
x=3
x=4
x=6
x=12
Error:/ by zero
```

构造器

- 构造器(constructor)主要用于初始化对象中的成员变量。每个类可以定义一个或多个构造器。构造器可以带或不带参数。如果一个类没有定义构造器，Java系统会为它生成一个默认的构造器。
- 如果定义了构造器，则只能调用这些构造器，而不能再调用默认构造器。
- 一个类的构造器的名字要与其类名完全相同。 构造器默认的访问类型为public，如果定义为private，则不能在外部直接建立对象，而要使用静态方法建立对象。
- 构造器也可以重载，此时要根据新建对象带入的参数选择构造器。
- 当建立导出类对象时，要先调用基类构造器。如果基类没有构造器，则会自动调用默认构造器。如果基类有构造器但是没有参数，则它会被自动调用。如果基类构造器有参数，则不会被自动调用。此时需要在导出类的构造器中用super(参数)调用基类的构造器。

```
class BaseA {
   int x1;
   BaseA(){
       x1=5;
   }
   BaseA(int y){
      x1=y;
   }
}
class DerivedA extends BaseA{
   String s1;
   DerivedA(String s2,int y){
      super(y);
      this.s1="'"+s2+"'";
   }
}
class TestStatic {
     public static void main(String[] args){
        DerivedA o1= new DerivedA("",12);
        System.out.println(o1.s1+o1.x1);

     }
}
```

方法参数与重载

- Java的方法可以采用数组和对象作为参数，也用可变参数类型。数组和对象也可以作为返回类型。
- 因为Java取消了指针类型，所以Java只有传值参数没有传址参数。采用基本数据类型和字符串的参数不能作为传址参数（也称为引用参数），即不能带回返回值。如果需要带回参数值，只能用自定义类作为参数类型，因为这种参数传入的是对象的引用。
- 一个类中定义多个同名方法的做法叫重载(overload)。这些方法主要通过参数进行区分，包括参数个数、类型和顺序。不能使用返回值的类型进行重载。
- 子类也可以重载父类的方法。如果参数完全相同，则会覆盖(override)基类的方法，使父类的同名方法不能使用(可以通过super.method()调用)。
- 在覆盖方法前加上@Override，编译器会根据这个指示进行检查是否父类有这个相同的方法。

可变参数

如果希望方法带入不同个数的参数，就可以使用可变参数。

```
class Example{
  static void print(String...  args){
    System.out.println("***"+args.length);//参数个数
    for(String temp:args)
       System.out.println(temp);
   }
   public static void main(String[] args){
       print();
       print("1","2");
       print("1","2","3");
       String a[]={"a","b","c","d"};
       print(a);
   }
}
```

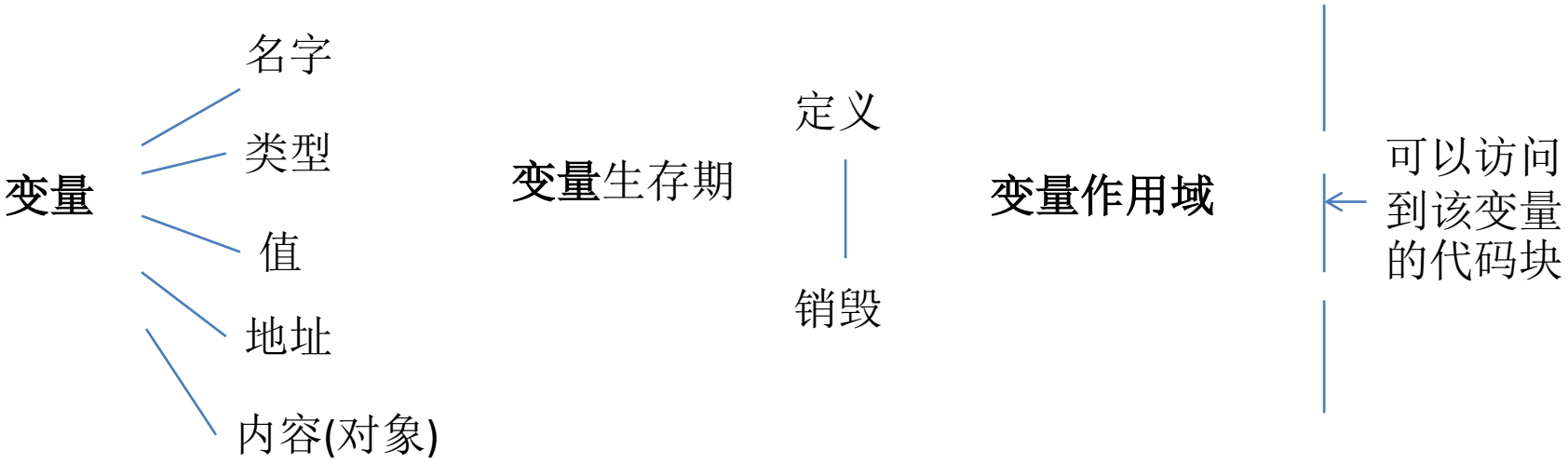
结果：

```
***0
***2
1
2
***3
1
2
3
***4
a
b
c
d
```

可变参数只能作为最后一个参数使用。args[i]可以取出第i个可变参数，用args.length可以得到可变参数的个数。

http://blog.csdn.net/testcs_dn/article/details/38920323

Java变量的作用域和生存期



- **变量**具有名字、类型、地址和值四个部分。变量的类型指出值的类型，变量的地址指出值存放的地址。**对象变量的值**为一个指针，指向对象的存放位置。
- 何时可以对变量赋值和引用它取决于变量的生存期和作用域。**变量的作用域**是指可以访问到该变量的代码范围。**变量的生存期**是指变量占用内存的时间，如果超出变量的生存期，即使在作用域内也不能访问该变量。

```
class  MyMath {
  String msg;
  int sum(int n){
    int res=0;
    for(int i=1;i<n;i++){
       n+=i;
    }
    return res;
  }
}
```

局部变量i的作用域：定义它的块

局部变量i的生存期：只在方法被调用的时候存在

- 一个方法的**非静态局部变量**只有在该方法被调用时存在，一旦退出方法，非静态局部变量所占空间将被释放。**静态局部变量的生存期**为从第一次调用其所在方法直到程序结束。
- **所有局部变量的作用域**都是定义它的块(由{}括住)。同名的局部变量不能在某个块及其子块同时定义（C语言可以，只使用最近定义的）。
- **非静态成员变量(数据域)的生存期**与对象相同。**静态成员变量的生存期**从第一次使用该变量直到程序结束。
- **非静态和静态成员变量的作用域**由其访问权限确定。静态成员变量和方法一种使用方法是作为类似C++的全局变量和函数进行使用。

静态设置

- 类的方法和成员变量可以定义为静态的，方法内部的局部变量也可以定义为静态的。它们在第一次被访问时分配内存空间，然后一直保留到程序结束才释放。
- 静态变量在堆中分配空间。静态方法和静态成员变量可以直接通过类名直接访问而不需要建立对象。
- 静态成员变量和静态局部变量在堆中所分配的空间，可以被所有处于其作用域的方法所使用。因此，静态方法和静态成员变量类似C++语言的全局函数和全局变量。
- main()就是静态方法，它也不用预先建立对象就可以让系统直接调用。
- 同一个类的非静态方法不能被直接使用。在不建立对象时，静态方法只能访问静态方法和静态数据域。只有建立对象才能使用非静态数据域和非静态方法。
- import语句后加入static可以令该包的类的所有方法变为static。

```
import static  com.group.show.*;
```

ClassA.java

```
class ClassA {
    static  int  x1 =0;
    static String s1;
    {
        ClassA.s1 = “abcd”;  //错误的!
    }
    static void f1(){
        System.out.println("Hello!");
    }
}
class TestStatic {
     public static void main(String[] args){
          ClassA.f1();
          System.out.println(ClassA.x1);
          System.out.println(ClassA.s1);
     }
}
```

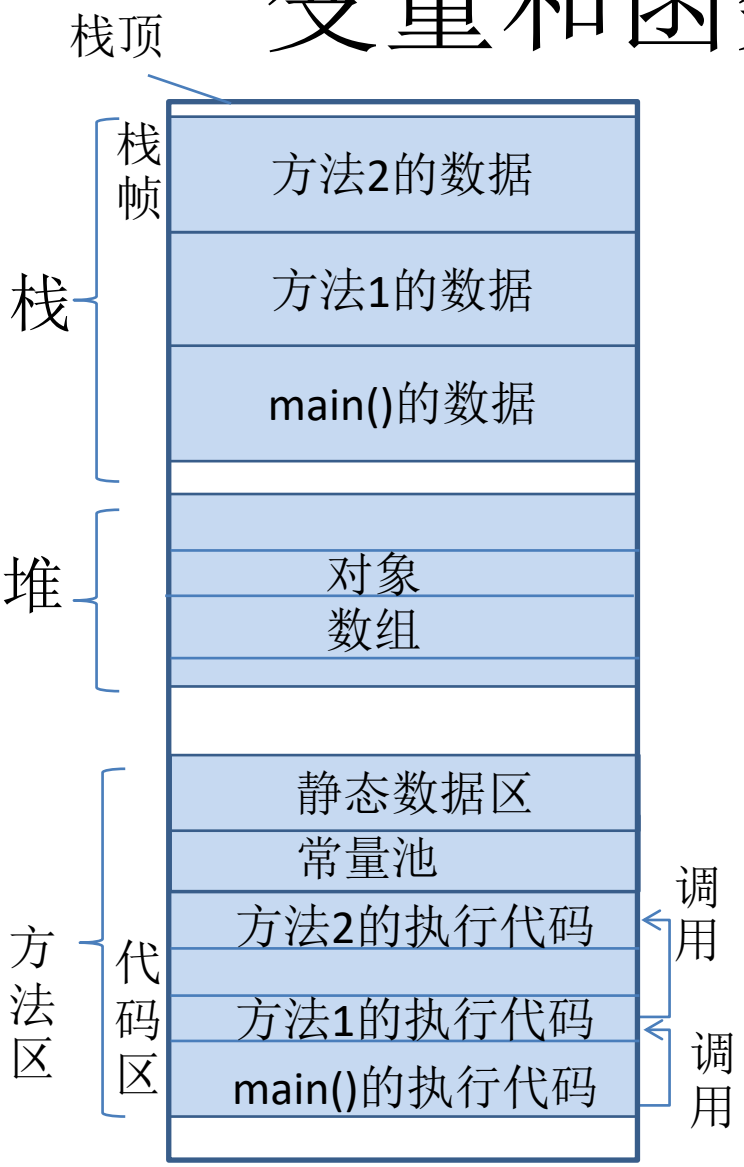
显示： Hello
0
null

final设置

- 在Java中，如果成员变量（数据域）、局部变量或者参数加上final的修饰词，表示初始化后不能被修改。
- 定义为final的成员变量的初始化可以在定义之后进行，但是只能初始化一次。
- 定义为final的方法不能被导出类所覆盖(override)。
- 定义为final的类不能被继承。

```
private static Random rand = new Random(47);
static final int INT_5 = rand.nextInt(20);     //0-19
```

变量和函数的内存分配

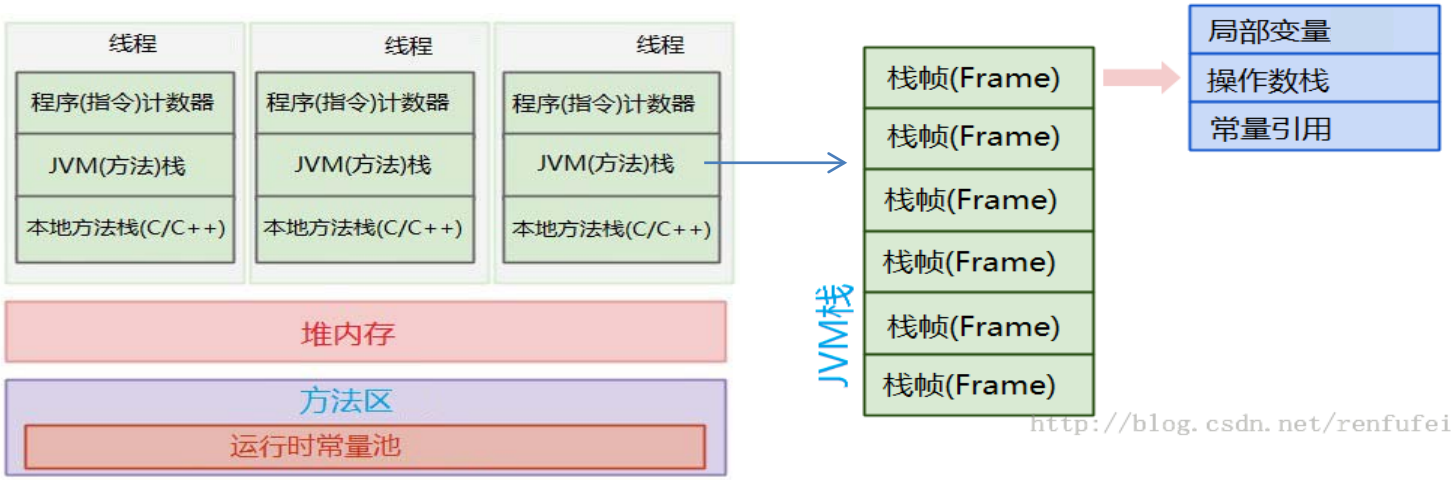


栈(Stack)区：每个线程都PC(程序计数器),JVM(方法)栈,以及本地方法栈(调用C语言等的动态链接库用)。每个栈帧存放用于一个方法的非静态局部变量、方法参数、方法返回值、操作数栈（计算表达式）、常量引用。对于基本数据类型变量，直接存放值；对于对象变量，只存放对象引用 。由于编译器可以直接确定局部变量等的相对地址(相对栈帧基址)，所以访问速度很快。

堆(heap)区用于存放(new)数组和对象。堆中对象当没有任何引用时，其空间通过垃圾回收进行释放。每一个Java应用都唯一对应一个JVM实例，每一个实例唯一对应一个堆。

方法区包含静态数据区、常量池和代码区。常量池存储了常量、类有关的信息。代码区保存方法和构造器的代码。常量以及静态变量的引用直接放在程序的指令或栈帧中。

* 定义对象变量只是定义了对象的引用（在栈中）只有new之后才在堆中为对象分配内存。



常量池的常量分为字面量和引用量。文本字符串、final变量等都是字面量，类信息、接口信息、数据段信息、方法信息都属于引用量。引用量最常见的一种用法是在调用方法的时候，根据方法名找到方法的引用，并以此定位到函数体进进行函数代码的执行。

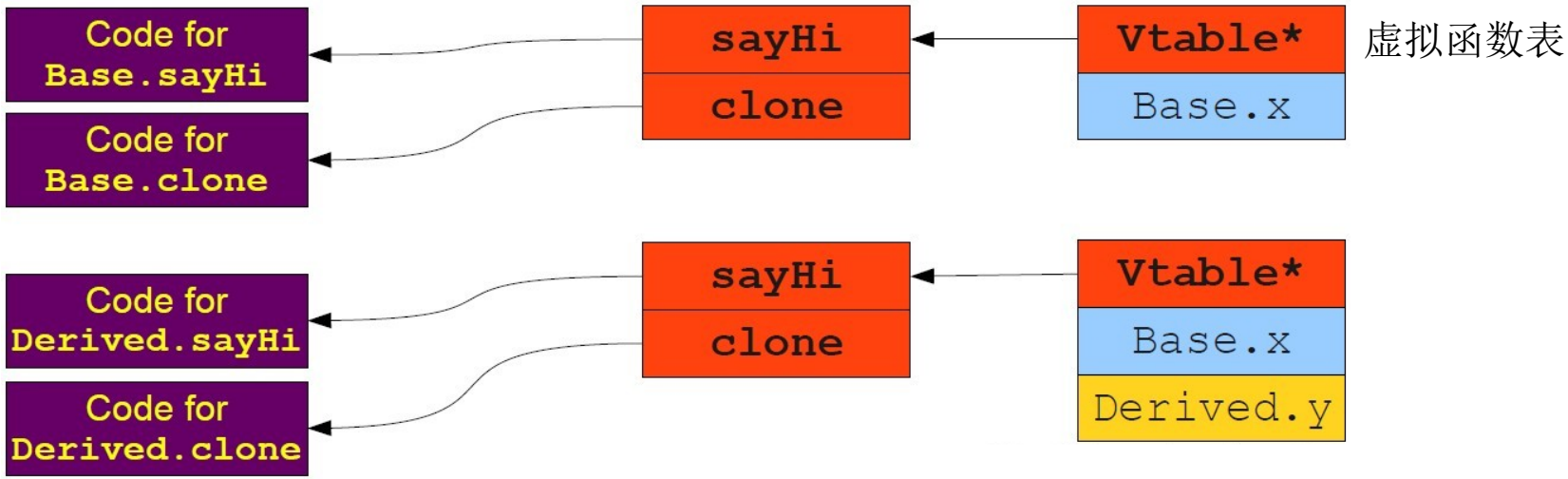
JVM把常量池按数据类型存放常量，并通过索引访问它们的入口。常量池在编译期间就被确定，并被保存在已编译的.class文件中。Java编译器会自动对常量进行优化，会查找并使用常量池已有的常量。见下例：

```
String str1 = new String("abc");
String str2 = new String("abc");
String str3 = "abc";
String str4 = "abc";
String str5 = "ab"+"c";
System.out.println(str1 == str2);      //false
System.out.println(str1 == str3);      //false
System.out.println(str3 == str4);      //true
System.out.println(str4 == str5);      //true
```

新建对象时JVM要在常量池中找到对象信息，然后在堆中建立对象，再把地址填入对象变量中。**Java对象在内存中是怎样分配的呢?** 一旦对象在堆中分配了空间,那本质上就是一系列的字节. 那么如何找到对象中某个特定的属性域呢? 编译器通过一个内部表来保存每个域的偏移量。

```
class Base {
    int x;
    void sayHi() {
        Print("Base");
    }
    Base clone() {
        return new Base;
    }
}
```

```
class Derived extends Base {
    int y;
    void sayHi() {
        Print("Derived");
    }
    Derived clone() {
        return new Derived;
    }
}
```



垃圾回收

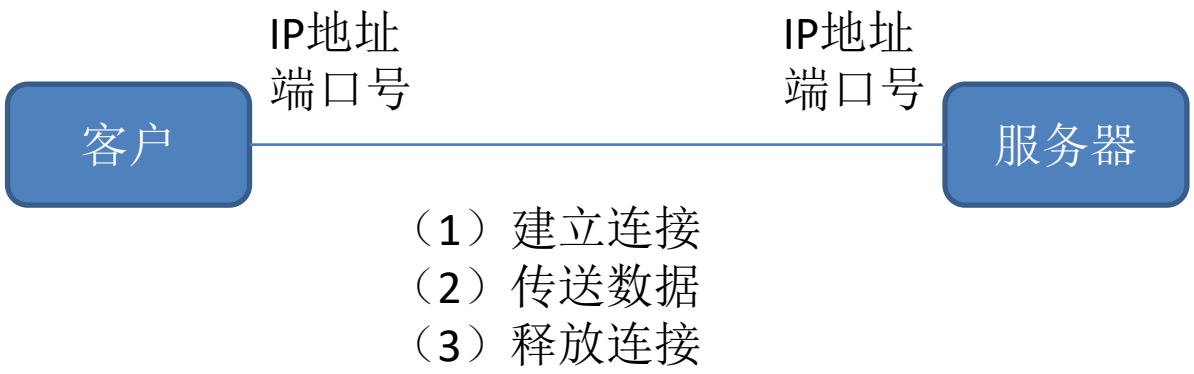
- Java语言没有析构器(destructor)，不能主动销毁对象，所有对象都由垃圾回收器自动进行回收。
- 如果一个对象如果没有任何引用，则可以被回收。Java的垃圾回收器只有在内存缺乏时才会去销毁这些没有任何引用的对象，并触发对象的finalize事件。因此，回收工作也许在程序结束都不会发生。
- 可以把对象的变量赋值为null来删除变量对该对象的引用。局部变量在退出所在方法后引用会被系统自动清除。
- 如果需要在对象使用结束前做一下清理工作，例如，关闭文件，把一些引用类变量设置为null，可以在对象中专门定义一个dispose方法来处理。

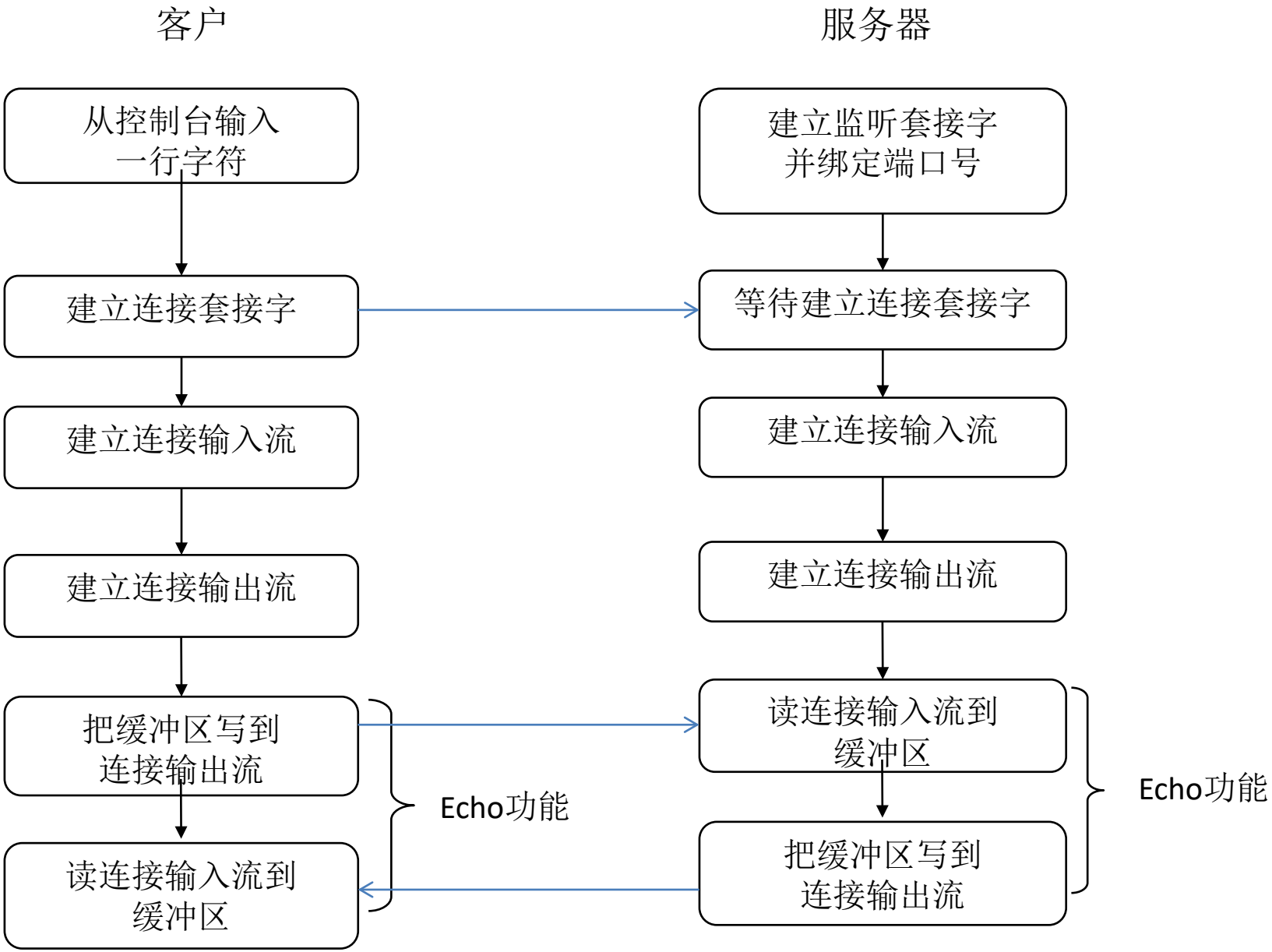
TCP编程

- 概述

（1）套接字编程包括TCP编程和UDP编程。它们都需要编写服务器程序和客户端程序。用TCP编程传送数据是可靠的，用UDP编程传送数据是不可靠的。

（2）TCP编程需要建立连接。为了建立连接，服务器必须公开其IP地址(或域名)和端口号，客户端是利用该IP地址找到服务器，并用端口号与服务器程序建立连接，然后传送数据，最后断开连接。




客户
服务器
从控制台输入
一行字符
建立连接套接字
建立连接输入流
建立连接输出流
把缓冲区写到
连接输出流
读连接输入流到
缓冲区
建立监听套接字
并绑定端口号
等待建立连接套接字
建立连接输入流
建立连接输出流
读连接输入流到
缓冲区
把缓冲区写到
连接输出流
Echo功能
Echo功能

- TCP服务器程序

```
//Echo程序：接收客户端的字符串，然后加上字符串"Echo:",并送回客户端。
import java.net.*;
import java.io.*;
public class Server {
 public static void main(String[] args) throws Exception{
   ServerSocket serverSocket = new ServerSocket(8000); //建立监听套接字
   Socket socket = serverSocket.accept();               //建立连接套接字
   DataInputStream inputFromClient =
          new DataInputStream(socket.getInputStream());//建立连接输入字节流
   DataOutputStream outputToClient =
          new DataOutputStream(socket.getOutputStream());//建立连接输出字节流
   String s1 = inputFromClient.readUTF();                //读入UTF-8编码的字符
   outputToClient.writeUTF("Echo:" +s1);                 //输出UTF-8编码的字符
   socket.close();                                       //关闭连接
 }
}
//Java内部采用Unicode编码，当读写UTF-8编码的文件时需要进行转换。
```

- TCP客户端程序

```
//服务器位于本机(localhost或127.0.0.1),端口号：8000。
//    与服务器建立连接后，输入一行字符然后发送给服务器，收到服务器响应后关闭连接
import java.util.Scanner;
import java.net.*;
import java.io.*;
public class Client {
  public static void main(String[] args) throws Exception {
   Scanner scanner = new Scanner(System.in);
   System.out.print("Enter a string: ");
   String s1 = scanner.nextLine();
   Socket socket = new Socket("localhost",8000);  //与服务器 建立连接。
   DataInputStream fromServer =
                    new DataInputStream(socket.getInputStream());
   DataOutputStream toServer =
                    new DataOutputStream(socket.getOutputStream());
   toServer.writeUTF(s1);
   String s2 = fromServer.readUTF();
   System.out.println(s2);
   socket.close();  //关闭连接
 }
}
```

UDP编程

- UDP服务器端

```
import java.net.*;
public class EchoUdpServer {
  public static void main(String[] args) throws Exception {
     InetSocketAddress socketAddress = new InetSocketAddress("localhost", 5050);
     DatagramSocket ds = new DatagramSocket(socketAddress);

     final byte[] buffer = new byte[1024];
     DatagramPacket packetA = new DatagramPacket(buffer, buffer.length);
     ds.receive(packetA);
     String infoRecvd = new String(packetA.getData(), 0, packetA.getLength());
     System.out.println("接收信息: " + infoRecvd);
     String clientAddress = packetA.getAddress().getHostAddress();
     int clientPort = packetA.getPort();
     String echoInfo =  clientAddress+":"+clientPort+ "    " + infoRecvd;
     DatagramPacket packetB = new DatagramPacket(buffer, buffer.length,
                                        packetA.getAddress(), packetA.getPort());
     packetB.setData(echoInfo.getBytes());
     ds.send(packetB);
  }
}
```

- UDP客户端程序

```
import java.net.*;
import java.util.*;
public class EchoUdpClient {
  public static void main(String[] args) throws Exception {
    System.out.print("Enter a string: ");
    Scanner scanner = new Scanner(System.in);
    String s1 = scanner.nextLine();
    final byte[] bufferA = s1.getBytes();
    DatagramSocket ds = new DatagramSocket();
    ds.setSoTimeout(2000);
    DatagramPacket packetA = new DatagramPacket(bufferA, bufferA.length,
                                    InetAddress.getByName("localhost"), 5050);
    ds.send(packetA);

    final byte[] bufferB = new byte[1024];
    DatagramPacket packetB = new DatagramPacket(bufferB, bufferB.length,
                                    InetAddress.getByName("localhost"), 5050);
    ds.receive(packetB);
    String info = new String(packetB.getData(), 0, packetB.getLength());
    System.out.println("服务端回应数据: " + info);
 }
}
```

- InetAddress对象

功能： 用于描述和包装一个Internet IP地址。
方法：
getLocalhost()：返回封装本地地址的实例。
getAllByName(String host)：返回封装Host地址的InetAddress实例数组。
getByName(String host)：返回一个封装Host地址的实例。其中，Host可以是域名或者是一个合法的IP地址。
getByAddress(addr)：根据地址串返回InetAddress实例。
getByAddress(host, addr)：根据主机字符串和地址串返回InetAddress实例。

- DatagramSocket对象

功能：用于接收和发送UDP的Socket实例。该类有3个构造函数：
方法：
DatagramSocket()：通常用于客户端编程，它并没有特定监听的端口，仅仅使用一个临时的。程序会让操作系统分配一个可用的端口。
DatagramSocket(int port)：创建实例，并固定监听Port端口的报文。通常用于服务端
DatagramSocket(int port, InetAddress localAddr)：这是个非常有用的构建器，当一台机器拥有多于一个IP地址的时候，由它创建的实例仅仅接收来自LocalAddr的报文。
receive(DatagramPacket d)：接收数据报文到d中。receive方法产生一个“阻塞”。“阻塞”是一个专业名词，它会产生一个内部循环，使程序暂停在这个地方，直到一个条件触发。

send(DatagramPacket dp)：发送报文dp到目的地。
setSoTimeout(int timeout)：设置超时时间，单位为毫秒。
close()：关闭DatagramSocket。在应用程序退出的时候，通常会主动释放资源，关闭Socket，但是由于异常地退出可能造成资源无法回收。所以，应该在程序完成时，主动使用此方法关闭Socket，或在捕获到异常抛出后关闭Socket。

- DatagramPacket

功能：
用于处理报文，它将Byte数组、目标地址、目标端口等数据包装成报文或者将报文拆卸成Byte数组。应用程序在产生数据包是应该注意，TCP/IP规定数据报文大小最多包含65507个，通常主机接收548个字节，但大多数平台能够支持8192字节大小的报文。

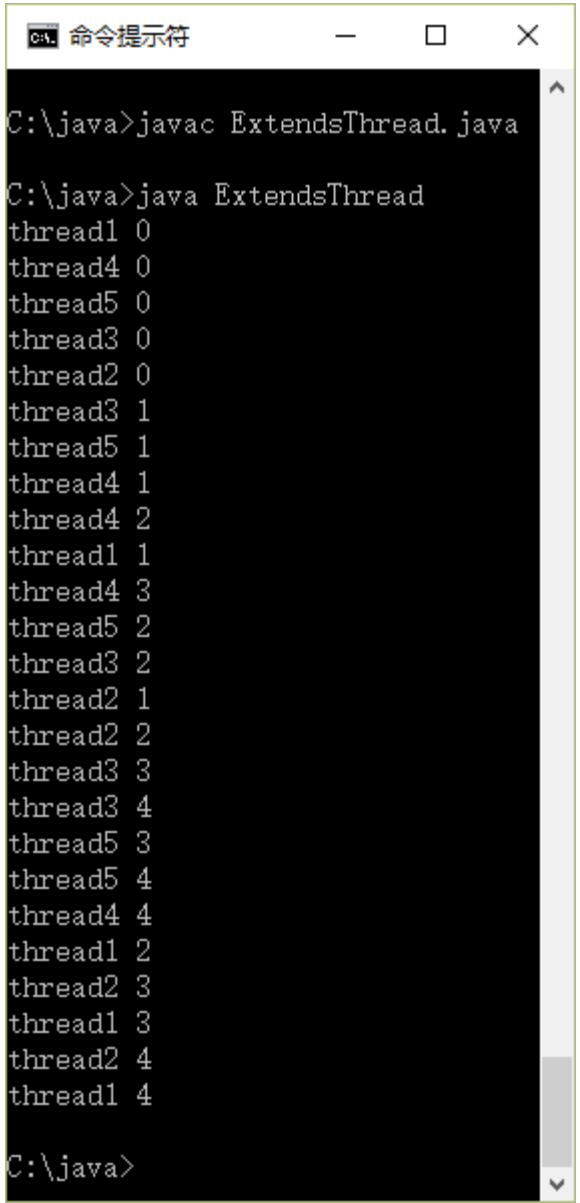
方法：
DatagramPacket(byte[] buf, int length)：将数据包中Length长的数据装进 Buf数组，一般用来接收客户端发送的数据。
DatagramPacket(byte[] buf, int offset, int length)：将数据包中从Offset开始、Length长的数据装进Buf数组。
DatagramPacket(byte[] buf, int length, InetAddress clientAddress,int clientPort)：从Buf数组中，取出Length长的数据创建数据包对象，目标是clientAddress地址，clientPort端口,通常用来发送数据给客户端。
DatagramPacket(byte[] buf, int offset, int length, InetAddress clientAddress, int clientPort)：从Buf数组中，取出Offset开始的、Length长的数据创建数据包对象，目标是clientAddress地址，clientPort端口，通常用来发送数据给客户端。
getData()：从实例中取得报文的Byte数组编码。
setData(byte[] buf)：将byte数组放入要发送的报文中。

进程与线程编程

- 概述
  - ✓ 现代操作系统可以同时运行多个程序。每个java程序的运行都会形成一个进程（process）。因为进程都在自己的地址空间中执行，所以，进程之间相互访问是比较复杂而低效的。
  - ✓ 如果一个进程只能顺序执行，就不能同时完成多个子任务，例如：等待输入时就不能进行计算。此时就需要引入线程(thread)。在一个进程中可以建立多个线程来同时完成多个子任务 。
  - ✓ 由于一个进程的所有线程都使用该进程的地址空间，所以它们之间通信十分容易。线程之间一般采用共享变量的方法进行通信。
  - ✓ 因为多个线程同时去操作一个共享变量会引起重写问题，所以，必须引入同步机制。
  - ✓ Java中每个线程都是独立的，因此主线程执行完毕并不会导致其它线程退出。
  - ✓ 下面我们先讲如何建立线程，然后讲如何通过同步方法访问共享变量。

- 继承Thread类创建线程

```
public class ExtendsThread extends Thread{
     private final static int THREAD_NUM = 5;
     private final static int COUNT = 5;

     public static void main(String[] args){
         for (int i = 1; i <=THREAD_NUM; i++) {
             new ExtendsThread("thread"+i).start();
         }
     }
     public  ExtendsThread(String name){
         super(name);
     }
     @Override
     public void run() {
         // TODO Auto-generated method stub
         for (int i = 0; i < this.COUNT; i++) {
             System.out.println(this.getName()+" "+i);
         }
     }
 }
```

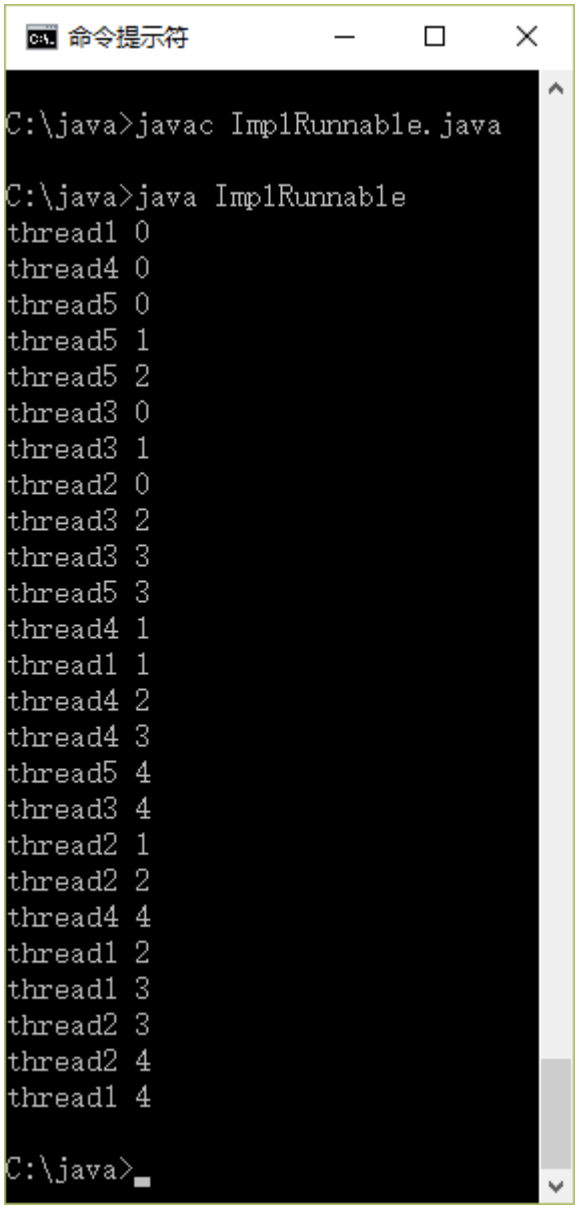


http://blog.csdn.net/luyee2010/article/details/8539815

- 实现runnable接口创建线程

```
public class ImplRunnable implements Runnable {
     private static final int THREAD_NUM = 5;
     private static final int COUNT = 5;
     @Override
     public void run() {
         for (int i = 0; i < COUNT; i++) {
            System.out.println(
               Thread.currentThread().getName()+" "+i);
         }
     }

     public static void main(String[] args) {
         for (int j = 0; j <= THREAD_NUM; j++) {
             ImplRunnable implRunnable= new ImplRunnable();
             new Thread(implRunnable,"thread"+j).start();
         }
     }
 }
```

ThreadName



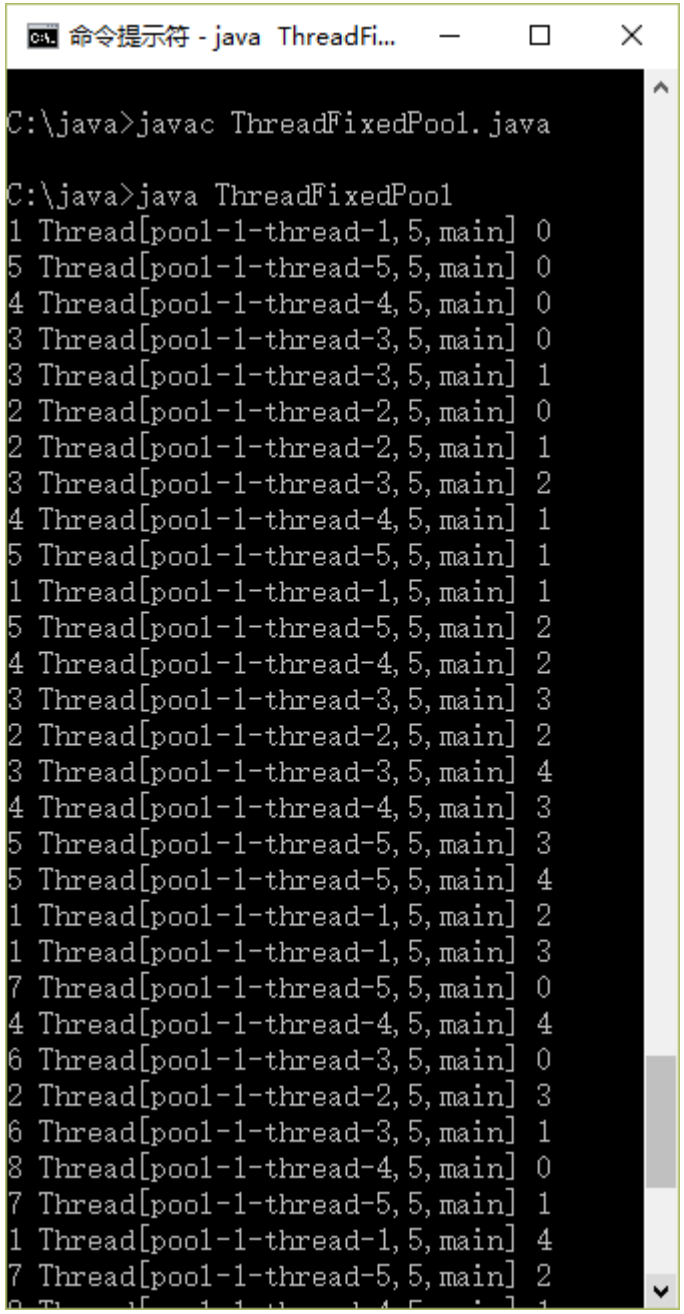
- 线程池

每次建立新线程都会有开销，线程太多会导致系统开销大增。Java提供了线程池来解决这个问题。线程池就是可以缓存线程的地方。采用线程池可以重复使用以前建立的线程，可以控制最大并发线程数和更好地管理线程。

Java通过Executors提供四种线程池，分别为：

- **newCachedThreadPool** 新线程会尽量重用线程池中的线程，如果没有可用的，则创建一个新线程并添加到池中。 60 秒钟未被使用的线程会被终止并从缓存中移除。对于执行大量短期异步任务的程序而言，这种线程池通常可提高程序性能。
- **newFixedThreadPool** 创建一个定长线程池，用来控制线程最大并发数，线程池没用线程可用时新线程会在队列中等待。
- **newScheduledThreadPool** 创建一个定长线程池，支持定时及周期性任务执行。
- **newSingleThreadExecutor** 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadFixedPool {
     private  static int POOL_NUM = 10;
     public static void main(String[] agrs){
         ExecutorService executorService
                = Executors.newFixedThreadPool(5);
         for (int i = 1; i < POOL_NUM+1; i++) {
             RunnableThread thread
                = new RunnableThread(i);
             executorService.execute(thread);
         }
     }
}
class RunnableThread implements  Runnable{
     private int COUNT = 5;
     private int index;
     RunnableThread(int index){
       this.index = index;
     }
     public void run() {
         for (int i = 0; i < COUNT; i++) {
             System.out.println(""+index+" "
                +Thread.currentThread()+" "+i);
         }
     }
}
```



```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadCachedPool {
     private  static int POOL_NUM = 10;
     public static void main(String[] agrs){
         ExecutorService executorService
               = Executors.newCachedThreadPool();
         for (int i = 1; i < POOL_NUM+1; i++) {
             RunnableThread thread
                 = new RunnableThread(i);
             executorService.execute(thread);
         }
     }
}
class RunnableThread implements  Runnable{
     private int COUNT = 5;
     private int index;
     RunnableThread(int index){
       this.index = index;
     }
     public void run() {
         for (int i = 0; i < COUNT; i++) {
             System.out.println(""+index+" "
                 +Thread.currentThread()+" "+i);
         }
     }
}
```

- 线程同步

设置同步(synchronized)方法可以对其所在对象加锁，使得只有一个线程可以执行该对象方法(但是其它线程可以在其中等待执行)。对于静态方法，同步方法直接对其所在的类加锁。

```
import java.util.concurrent.*;
public class AccountWithSync {
  private static Account account = new Account();
  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }
    executor.shutdown(); // 不再接收新任务
    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }
    System.out.println("What is balance? " + account.getBalance());
  }
  // A thread for adding a penny to the account
  private static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }
```

```
// An inner class for account
 private static class Account {
   private int balance = 0;
   public int getBalance() {
     return balance;
   }
   public synchronized void deposit(int amount) {
     int newBalance = balance + amount;
     // This delay is deliberately added to magnify the
     // data-corruption problem and make it easy to see.
     try {
       Thread.sleep(5);
     }
     catch (InterruptedException ex) {
     }
     balance = newBalance;
   }
 }
}
```

可替换为

同步可以直接对对象加锁

```
public void deposit(int amount) {
   synchronized(this) {
      int newBalance = balance + amount;
      try {
        Thread.sleep(5);
      }
      catch (InterruptedException ex) {
      }
      balance = newBalance;
    }
}
```

- 线程加锁

通过可重入锁（ReentrantLock）可以显示地创建排斥锁。所有进入加锁区域的线程，只能有一个线程处于执行状态。

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class AccountWithSyncUsingLock {
  private static Account account = new Account();
  public static void main(String[] args) {
    ExecutorService executor = Executors.newCachedThreadPool();
    // Create and launch 100 threads
    for (int i = 0; i < 100; i++) {
      executor.execute(new AddAPennyTask());
    }
    executor.shutdown(); // 不再接收新任务
    // Wait until all tasks are finished
    while (!executor.isTerminated()) {
    }
    System.out.println("What is balance ? " + account.getBalance());
  }
  // A thread for adding a penny to the account
  public static class AddAPennyTask implements Runnable {
    public void run() {
      account.deposit(1);
    }
  }
```

```
// An inner class for account
  public static class Account {
    private static Lock lock = new ReentrantLock(); // 创建(公平)锁  --
    private int balance = 0;                        //  等待最久的线程首先获得锁
    public int getBalance() {
      return balance;
    }
    public void deposit(int amount) {
      lock.lock(); // 获得锁
      try {
        int newBalance = balance + amount;
        // This delay is deliberately added to magnify the
        // data-corruption problem and make it easy to see.
        Thread.sleep(5); //睡眠并让出cpu给其他线程，但是不释放锁，执行yield()也会让出CPU。
        balance = newBalance;
      }
      catch (InterruptedException ex) {
      }
      finally {
        lock.unlock(); // 释放锁
      }
    }
  }
}
```

在调用sleep()方法的过程中，线程不会释放锁，其它持有该锁的线程不能执行。

- 线程间协作

在拥有可重入锁之后，执行await的线程将会进入等待状态，等待该条件对象执行signal()或signalAll()时才可以恢复为就绪态。 signal()只激活一个等待线程， signalAll()激活全部线程。对下面程序通过为存款和取款分别建立线程，并通过线程间协作保证当存款足够时客户可以及时取到款。

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class Hello {  //ThreadCooperation
  private static Account account = new Account();
  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new DepositTask());
    executor.execute(new WithdrawTask());
    executor.shutdown();
    System.out.println("Thread 1\t\tThread 2\t\tBalance");
  }
```

```
  // A task for adding an amount to the account
  public static class DepositTask implements Runnable {
    public void run() {
      try { // Purposely delay it to let the withdraw method proceed
        while (true) {
          account.deposit((int)(Math.random() * 10) + 1);
          Thread.sleep(1000);
        }
      }
      catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
// A task for subtracting an amount from the account
  public static class WithdrawTask implements Runnable {
    public void run() {
      while (true) {
        account.withdraw((int)(Math.random() * 10) + 1);
      }
    }
  }
  // An inner class for account
  private static class Account {
    // Create a new lock
    private static Lock lock = new ReentrantLock(); // 重入锁
    // Create a condition
    private static Condition newDeposit = lock.newCondition();
    private int balance = 0;
    public int getBalance() { return balance; }
```

```
  public void withdraw(int amount) {
    lock.lock(); // 获得锁（每个时刻只能有一个线程保持锁）
    try {
      while (balance < amount) {
        System.out.println("\t\t\tWait for a deposit");
        newDeposit.await(); // wait时将线程将进入休眠状态，此时其它线程可以使用本锁。
      }
      balance -= amount;
      System.out.println("\t\t\tWithdraw " + amount +
        "\t\t" + getBalance());
    }
    catch (InterruptedException ex) {
      ex.printStackTrace();
    }
    finally {
      lock.unlock(); // 释放锁
    }
  }
  public void deposit(int amount) {
    lock.lock(); // Acquire the lock
    try {
      balance += amount;
      System.out.println("Deposit " + amount +
        "\t\t\t\t\t" + getBalance());
      newDeposit.signalAll(); // 激活所有在此条件等待的线程
    }
    finally {
      lock.unlock(); // Release the lock
    }
  }
}
```

结果

```
Deposit 44
Thread 1Thread 2Balance
Withdraw 31
Wait for a deposit
Deposit 23
Wait for a deposit
Deposit 47
Withdraw 43
Wait for a deposit
Deposit 14
Wait for a deposit
Deposit 711
Withdraw 56
Withdraw 60
...
```

在调用sleep()方法的过程中，线程不会释放对象锁。而当调用await()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有被signal之后本线程才进入对象锁定池准备。

- 用lock实现生产消费者

下面程序通过进队列和出队列说明使用重入锁如何解决生产者和消费者同步问题。

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
public class ConsumerProducerLock {  //ConsumerProducer
  private static Buffer buffer = new Buffer();
  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }
  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes " + i);
          buffer.write(i++); // Add a value to the buffer
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
  }
```

```
// A task for reading and deleting an int from the buffer
 private static class ConsumerTask implements Runnable {
   public void run() {
     try {
       while (true) {
         System.out.println("\t\t\tConsumer reads " + buffer.read());
         // Put the thread into sleep
         Thread.sleep((int)(Math.random() * 10000));
       }
     } catch (InterruptedException ex) {
       ex.printStackTrace();
     }
   }
 }
 // An inner class for buffer
 private static class Buffer {
   private static final int CAPACITY = 1; // buffer size
   private java.util.LinkedList<Integer> queue =
     new java.util.LinkedList<Integer>();
   // Create a new lock
   private static Lock lock = new ReentrantLock();
   // Create two conditions
   private static Condition notEmpty = lock.newCondition();
   private static Condition notFull = lock.newCondition();
```

```
  public void write(int value) {
      lock.lock(); // Acquire the lock
      try {
        while (queue.size() == CAPACITY) {
          System.out.println("Wait for notFull condition");
          notFull.await();
        }
        queue.offer(value);
        notEmpty.signal(); // Signal notEmpty condition
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      } finally {
        lock.unlock(); // Release the lock
      }
    }
    public int read() {
      int value = 0;
      lock.lock(); // Acquire the lock
      try {
        while (queue.isEmpty()) {
          System.out.println("\t\t\tWait for notEmpty condition");
          notEmpty.await();
        }
        value = queue.remove();
        notFull.signal(); // Signal notFull condition
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      } finally {
        lock.unlock(); // Release the lock
        return value;
      }
    }
  }
}
```

对象还可以采用wait()、notify()和notifyAll()进行同步。当前对象采用this. wait()、 this. notify()、 this. notifyAll()。

- Lock用来保护代码片段，允许多个线程进入这段代码，但是任何时刻只有一个线程执行这段代码。
- 一个Lock对象可以拥有多个Condition对象，它们可以被用来管理进入代码段的线程。

• 用信号量实现生产者消费者

```
import java.util.concurrent.*;
public class ProducerConsumerSemp {
   public static void main(String[] args) {
     ExecutorService exec = Executors.newCachedThreadPool();
     final Semaphore semp = new Semaphore(5);     // 最大许可数：5
     for (int index = 0; index < 20; index++) {  // 建立20个线程，模拟20个客户端访问
         final int NO = index;
         Runnable run = new Runnable() {
            public void run() {
               try {
                     semp.acquire();    //从信号量获取一个许可，
                                        //  如果没有许可可用，则会阻塞线程直到有许可可用
                     System.out.println("Accessing: " + NO);
                     Thread.sleep((long) (Math.random() * 10000));
                     semp.release();    // 释放一个许可给信号量
                     System.out.println("-----------------"+semp.availablePermits());
               } catch (InterruptedException e) {
                     e.printStackTrace();
               }
             }
         };
         exec.execute(run);
   }
   exec.shutdown();
  }
}
```

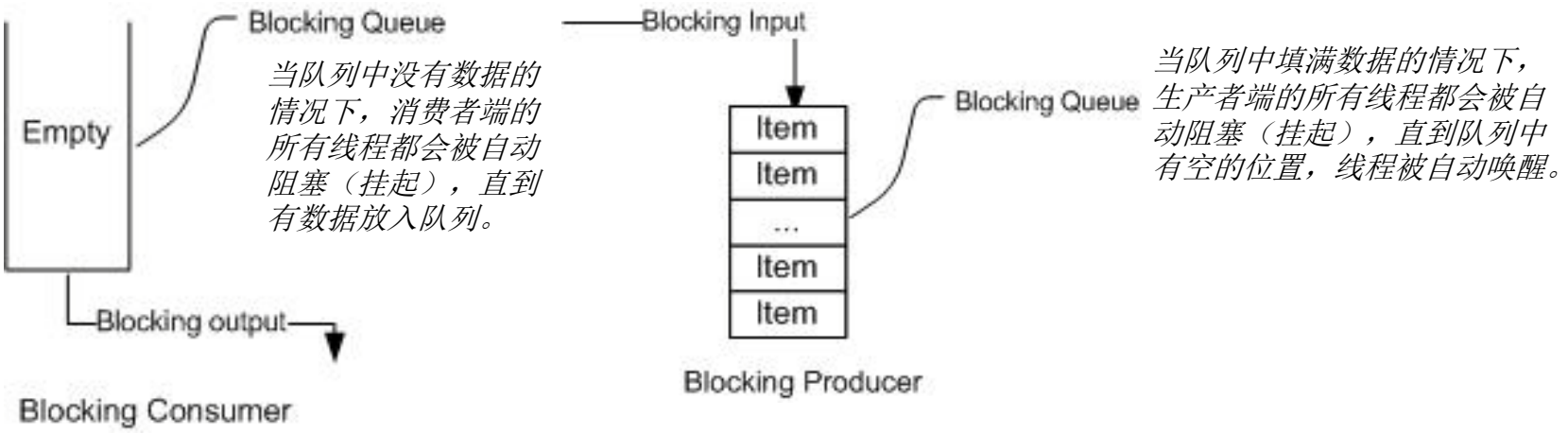
结果：

```
Accessing: 3
Accessing: 2
Accessing: 0
Accessing: 1
Accessing: 5
----------0
Accessing: 4
----------1
Accessing: 6
----------1
Accessing: 10
Accessing: 8
----------0
----------1
Accessing: 9
Accessing: 17
----------0
Accessing: 19
----------0
----------1
Accessing: 18
----------1
Accessing: 12
----------1
Accessing: 13
Accessing: 16
----------0
----------1
Accessing: 7
----------0
Accessing: 14
----------1
Accessing: 15
----------1
Accessing: 11
----------1
----------2
----------3
----------4
----------5
```

http://www.cnblogs.com/whgw/archive/2011/09/29/2195555.html

- 阻塞队列（BlockingQueue）

概述



http://blog.csdn.net/rwecho/article/details/38313685
http://blog.csdn.net/defonds/article/details/44021605#t7
http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/BlockingQueue.html
http://blog.csdn.net/defonds/article/details/44021605#t8　所有并发数据结构

BlockingQueue的主要方法：

放入数据：

add(E e): 如果将对象e成功加入BlockingQueue，返回true，否则出错IllegalStateException 。

offer(E e): 如果将对象e成功加入BlockingQueue，返回true，否则返回false。(不阻塞)

offer(E e, long timeout, TimeUnit unit)：如果在设定时间内可以将e加到BlockingQueue里，返回true，否则返回false。（不阻塞）

put(E e): 将对象e加入BlockingQueue，如果队列满，则等待直到成功加入 。(void)

获取数据：

poll(long timeout, TimeUnit unit)：在指定时间内取走并返回队头或者超时未取到返回null。

take()：等待直到可以取走并返回队头为止 。(E)

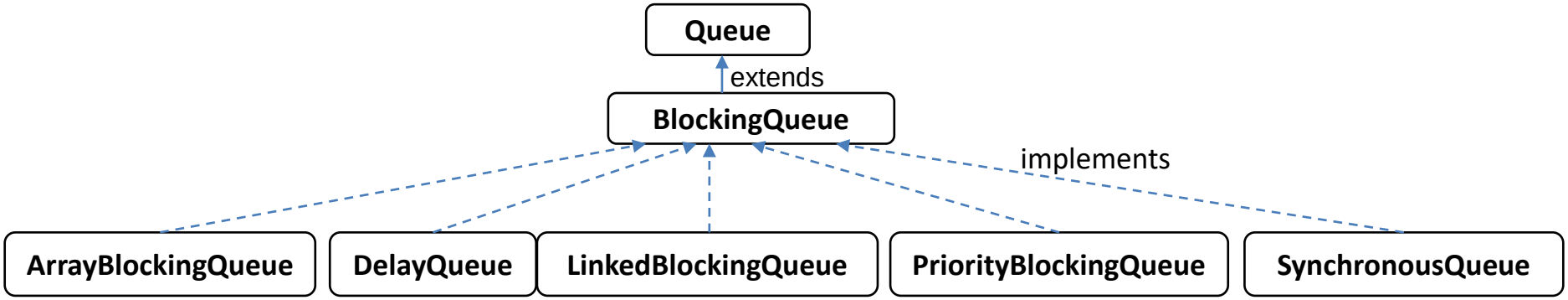
drainTo(Collection<? super E> c): 取走所有元素(可限制个数)并放入集合c， 返回取到的个数。

其它方法：

boolean contains(Object o) int remainingCapacity() boolean remove(Object o) E peek()

* 枚举类型：TimeUnit.NANOSECONDS TimeUnit.MILLISECONDS TimeUnit.MICROSECONDS TimeUnit.SECONDS...
* 这里的“取走”都有取出并移除(remove)的含义。
* <? super E>表示泛型采用E的父类， <? extends E>表示泛型采用E的子类
* peek()只取出不移除。
* 构造函数：DelayQueue( ) DelayQueue(Collection<? extends E> c)

BlockingQueue的类型：



* `public interface BlockingQueue<E> extends Queue<E>`

* **`PriorityBlockingQueue`** 是一个无界的并发队列。它使用了和类 `java.util.PriorityQueue` 一样的排序规则。你无法向这个队列中插入 `null` 值。所有插入到 `PriorityBlockingQueue` 的元素必须实现 `java.lang.Comparable` 接口。因此该队列中元素的排序就取决于你自己的 `Comparable` 实现。注意 `PriorityBlockingQueue` 对于具有相等优先级(`compare() == 0`)的元素并不强制任何特定行为。同时注意，如果你从一个 `PriorityBlockingQueue` 获得一个 `Iterator` 的话，该 `Iterator` 并不能保证它对元素的遍历是以优先级为序的。

* **`SynchronousQueue`** 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。把这个类称作一个队列显然是夸大其词了，它更多像是一个汇合点。

* 其它并发结构：**阻塞双端队列 BlockingDeque**和LinkedBlockingDeque，**并发 Map(映射) ConcurrentMap** 和 **ConcurrentHashMap**，**ConcurrentSkipListSet**，**ConcurrentLinkedQueue**等。

http://janeky.iteye.com/blog/770671

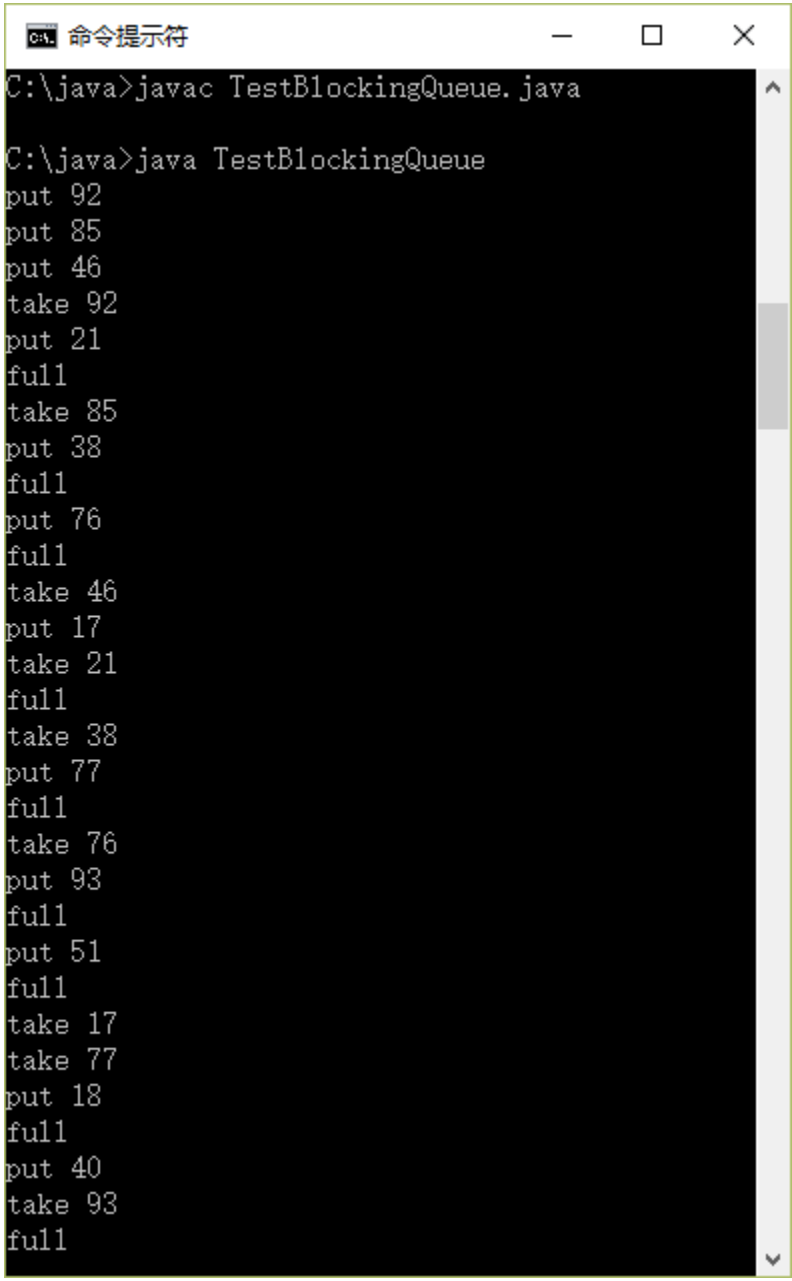
LinkedBlockingQueue示例

```
import java.util.Random;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
public class TestBlockingQueue {
   public static void main(String[] args) {
      final BlockingQueue<Integer> queue = new LinkedBlockingQueue<Integer>(3);
      final Random random=new Random();    /*省略上限(3)，则以Integer.MAX_VALUE 作为上限*/

      class Producer implements Runnable{
         @Override
         public void run() {
            while(true){
               try {
                  int j=random.nextInt(100);
                  queue.put(j);//满则阻塞
                  System.out.println("put "+j);
                  if(queue.size()==3){
                     System.out.println("full");
                  }
               } catch (InterruptedException e) {
                  e.printStackTrace();
               }
            }
         }
      }
```

双端队列LinkedBlockingDeque的其他操作：

```
queue.addFirst("1");
queue.addLast("2");

String two = queue.takeLast();
String one = queue.takeFirst();
```

```
    class Consumer implements Runnable{
        @Override
        public void run() {
            while(true){
                try {
                    int i = queue.take();//空则阻塞
                    System.out.println("take "+i);
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    new Thread(new Producer()).start();
    new Thread(new Consumer()).start();
  }
}
```



DelayedQueue示例

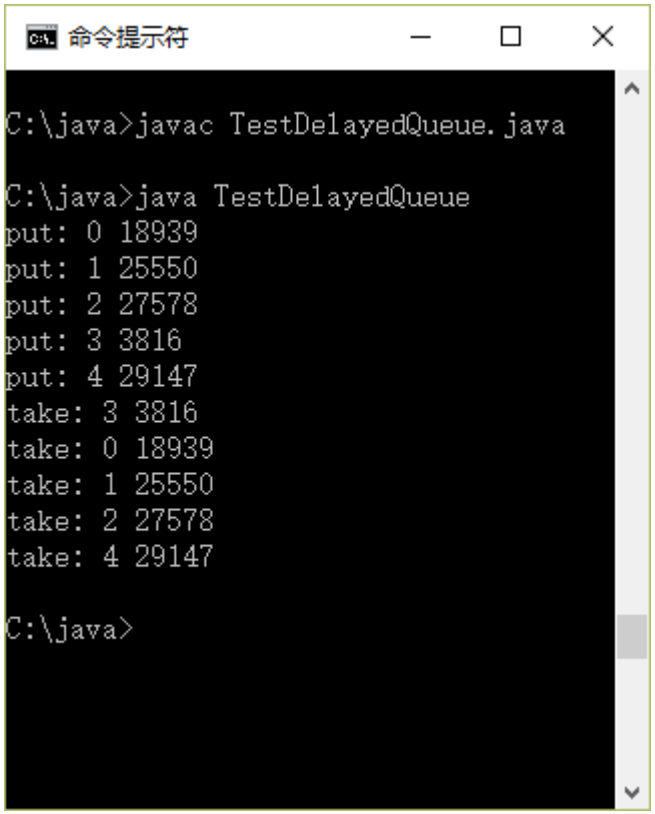
*DelayedQueue 注入的元素必须实现 Delayed 接口。

```
public interface Delayed extends Comparable<Delayed> {
        public long getDelay(TimeUnit timeUnit);
}
```

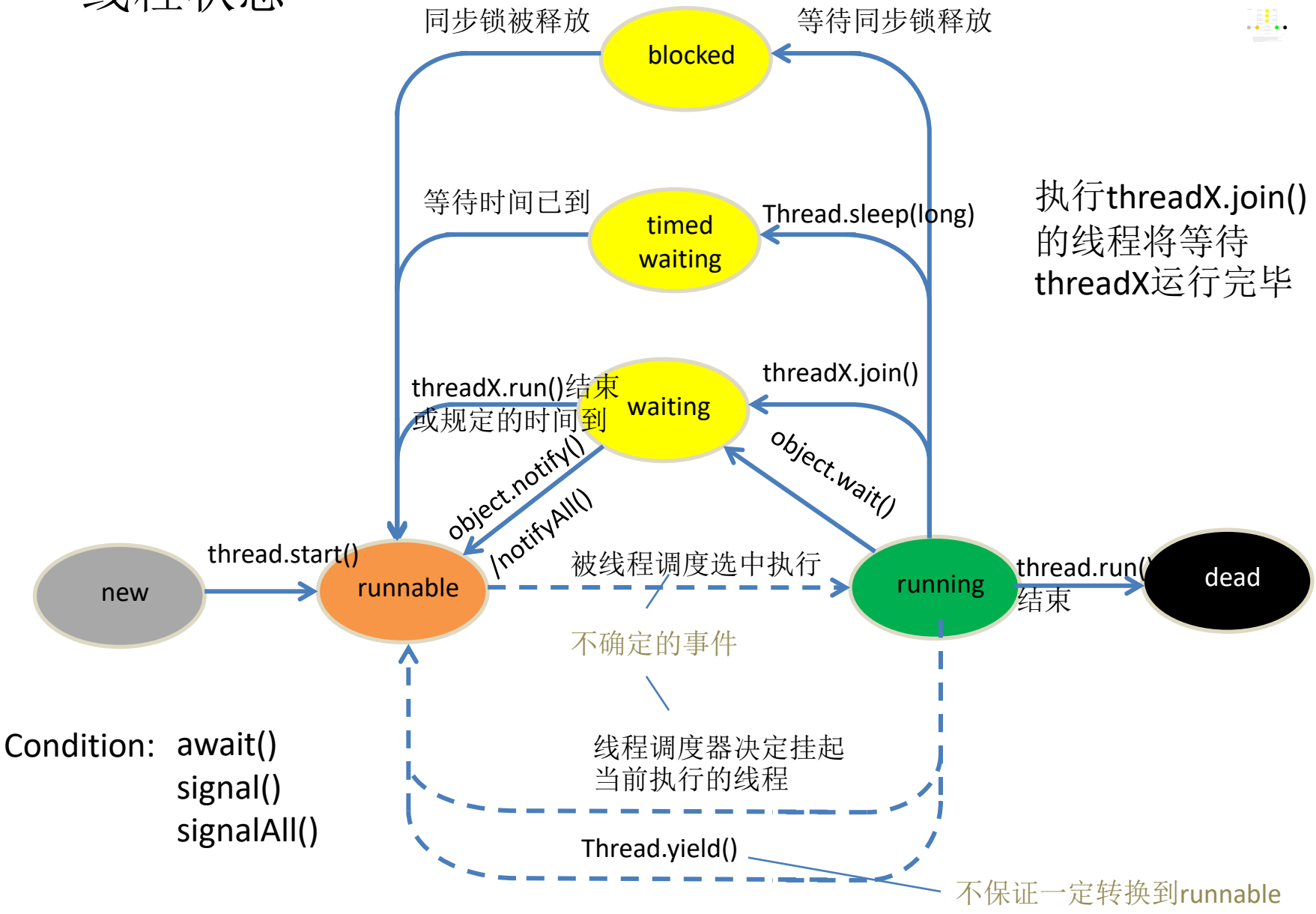
* DelayQueue的元素必须过期才可以被取走。
getDelay()返回的值为还需要延迟的时间，返回的是0 或者负值表示已过期。

```
import java.util.Random;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;
public class TestDelayedQueue {
   private class Stadium implements Delayed{
      long delay, trigger;
      int index;
      public Stadium(int index, long delay){
         this.index = index;
         this.delay = delay;
         trigger=System.currentTimeMillis()+delay;
         print("put");
      }
      void print(String st){System.out.println(st+": "+index+" " + delay);}
      @Override
      public long getDelay(TimeUnit arg0) {
         long n=trigger-System.currentTimeMillis();
         return n;
      }
      @Override
      public int compareTo(Delayed arg0) {
         return (int)(this.getDelay(TimeUnit.MILLISECONDS)
                  -arg0.getDelay(TimeUnit.MILLISECONDS));
      }
      public long getTriggerTime(){
         return trigger;
      }
   }
```

```
public static void main(String[] args)throws Exception {
      Random random=new Random();
      DelayQueue<Stadium> queue=new DelayQueue<Stadium>();
      TestDelayedQueue t=new TestDelayedQueue();

      for(int i=0;i<5;i++){
         queue.add(t.new Stadium(i,random.nextInt(30000)));
      }
      Thread.sleep(2000);

      while(true){
         Stadium s=queue.take();//延时时间未到就一直等待
         if(s!=null){
            s.print("take");
         }
         if(queue.size()==0)
            break;
      }
   }
}
```



- 线程状态


同步锁被释放
blocked
等待同步锁释放
等待时间已到
timed waiting
Thread.sleep(long)
执行threadX.join()的线程将等待threadX运行完毕
threadX.run()结束或规定的时间到
waiting
threadX.join()
object.notify()
/notifyAll()
object.wait()
new
thread.start()
runnable
被线程调度选中执行
running
thread.run()结束
dead
不确定的事件
线程调度器决定挂起当前执行的线程
Thread.yield()
不保证一定转换到runnable
Condition: await()
signal()
signalAll()

定时器

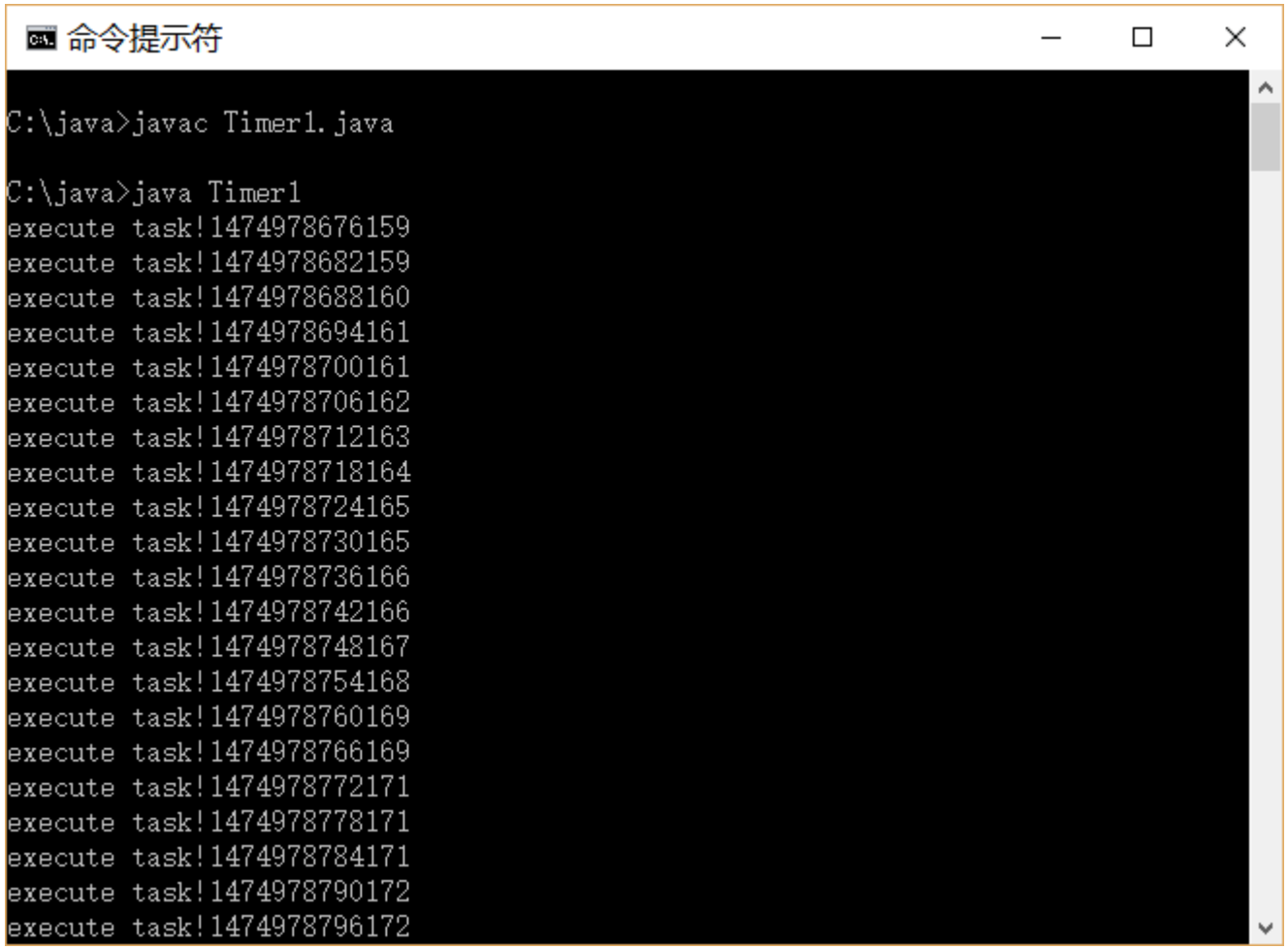
参考

- Java采用Timer类来完成定时或周期性任务。调用其方法schedule和scheduleAtFixedRate可以从某个时候或者从多少毫秒开始周期性地执行任务。
- 使用schedule方法在每次计时时间到时执行一个新线程，在完成该线程时启动下一次计时，而使用scheduleAtFixedRate方法在每次时间到后都会立即启动下一次计时并执行一个新线程。
- Timer类的cancel方法可以取消定时器。
- 下页例子中的 scheduledExecutionTime() 方法用于返回最近执行这项任务被调度执行时的时间。
- 格式：

```
Timer.schedule(TimerTask task, long delay, long period)
Timer.schedule(TimerTask task, Date  startTime, long period)
Timer.scheduleAtFixedRate(TimerTask task, long delay, long period)
Timer.scheduleAtFixedRate(TimerTask task, Date startTime, long period)
```

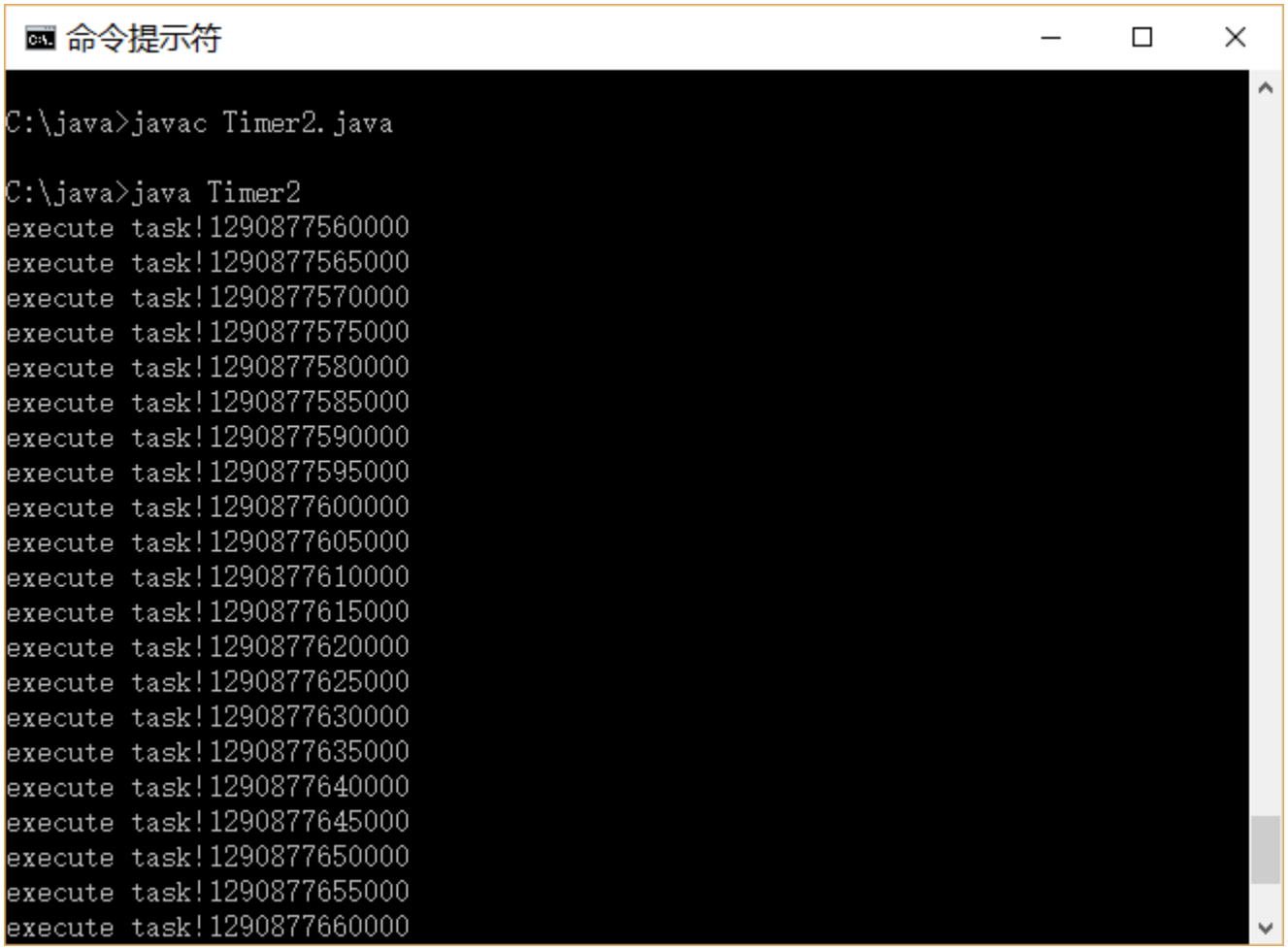
使用匿名类的例子：

```
import java.text.ParseException;  import java.text.SimpleDateFormat;
import java.util.Date;import java.util.Timer;  import java.util.TimerTask;

public class Timer1 {
    public static void main(String[] args) throws ParseException {
       SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");
       Date startDate = dateFormatter.parse("2016/9/27 17:40:00");
       Timer timer = new Timer();
       timer.schedule(new TimerTask()  {    // schedule 在某个时间之后开始计时，到时
        public void run() {                 //    则执行线程，完成后再开始下一次的计时
            try {
                Thread.sleep(6000);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            System.out.println("execute task!"+ this.scheduledExecutionTime());
         }
       }, startDate, 5 * 1000); // 第二个参数还可以采用毫秒，例如: 2000(2秒后)
   }                            // 第三个参数为周期，单位为毫秒，这里是5秒一次
}
```



这里例子把前面的日期改成了3000。显示的结果为毫秒，第一个结果是过了9000毫秒显示的，然后基本上是每个隔6秒多。

如果把上面程序中的timer.schedule改为timer. scheduleAtFixedRate (Timer2.java)，执行的结果如下：

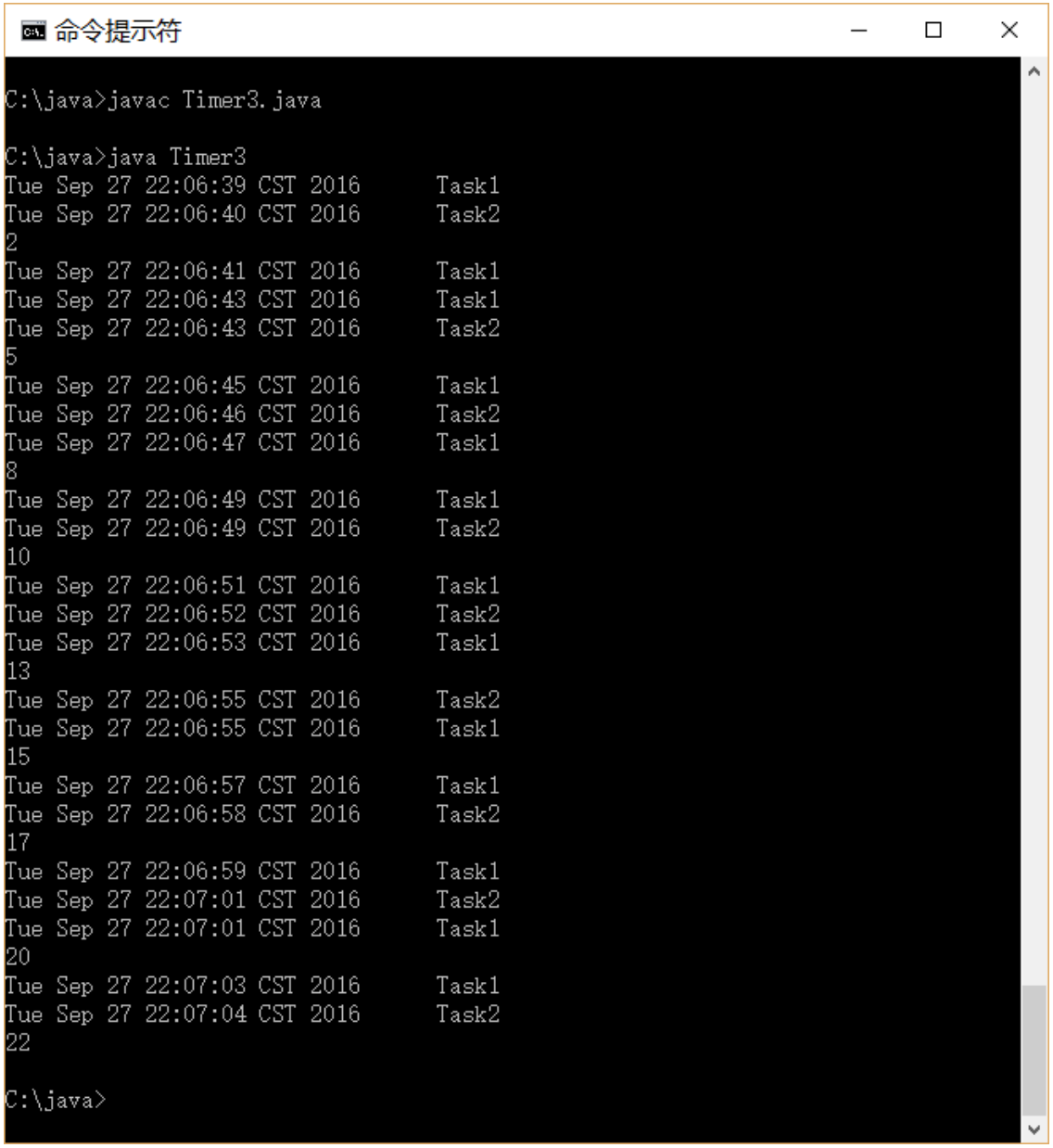


第一个结果是过了9000毫秒显示的，然后每个任务隔5秒，非常准时。

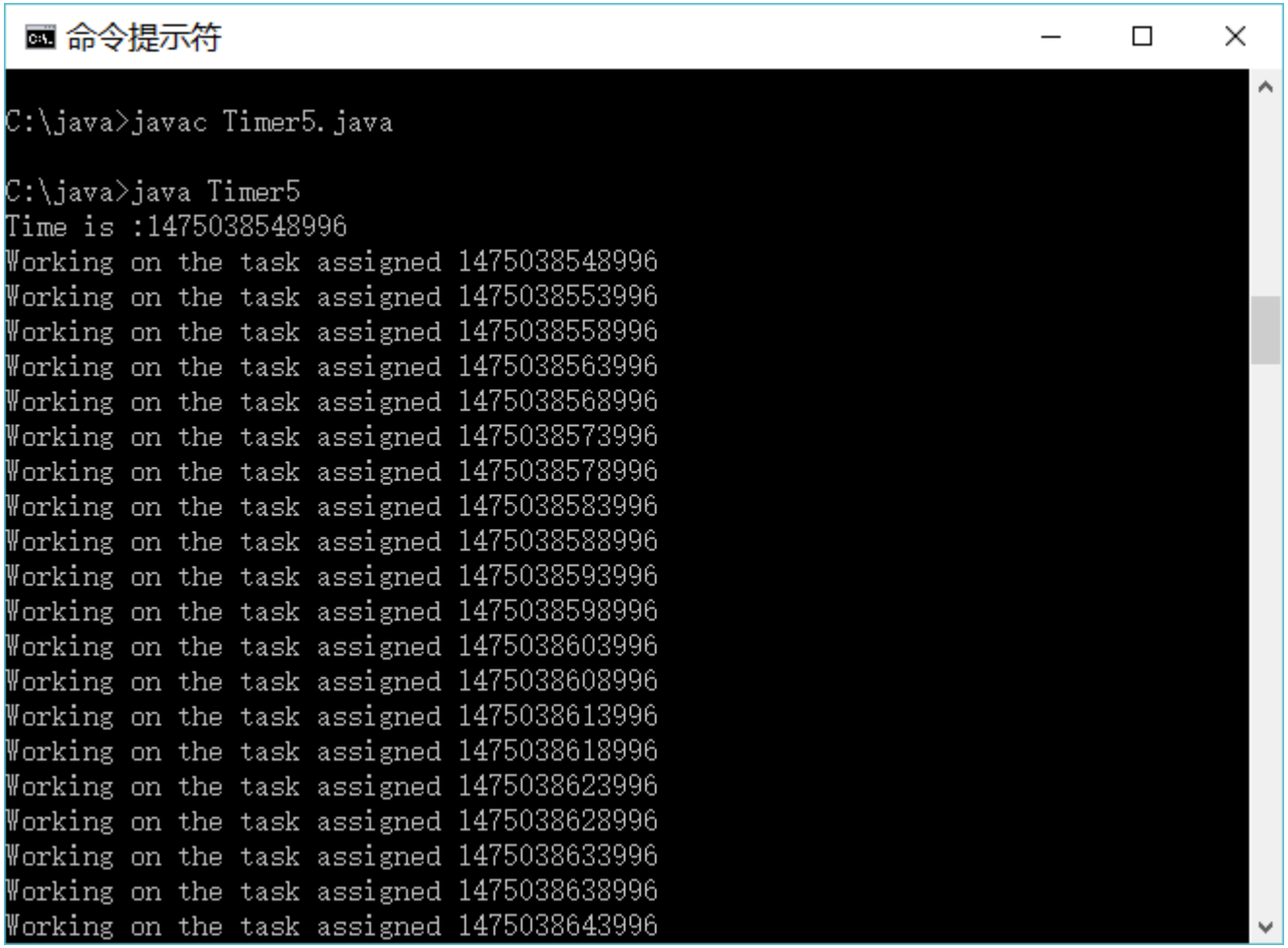
使用命名类的例子：

```
import java.io.IOException;  import java.util.Date;  import java.util.Timer;

public class Timer3{
  static int count = 0;
  public static void main(String[] args){
     Timer timer = new Timer();
     MyTask myTask1 = new MyTask("Task1");
     MyTask myTask2 = new MyTask("Task2");
     timer.schedule(myTask1, 1000, 2000);
     timer.scheduleAtFixedRate(myTask2, 2000, 3000);
     while (count<20){
        try {
            Thread.sleep(3000);
            System.out.println(count);
         } catch (InterruptedException e) {
             e.printStackTrace();
         }
      }
      timer.cancel();
 }
```

```
static class MyTask extends java.util.TimerTask{
        String info = "INFO";
        MyTask(String info){
                this.info = info;
        }
        @Override
        public void run(){
           System.out.println(new Date() + "      " + info);
           count++;
        }
    }
}
```


命令提示符
C:\java>javac Timer3.java
C:\java>java Timer3
Tue Sep 27 22:06:39 CST 2016 Task1
Tue Sep 27 22:06:40 CST 2016 Task2
2
Tue Sep 27 22:06:41 CST 2016 Task1
Tue Sep 27 22:06:43 CST 2016 Task1
Tue Sep 27 22:06:43 CST 2016 Task2
5
Tue Sep 27 22:06:45 CST 2016 Task1
Tue Sep 27 22:06:46 CST 2016 Task2
Tue Sep 27 22:06:47 CST 2016 Task1
8
Tue Sep 27 22:06:49 CST 2016 Task1
Tue Sep 27 22:06:49 CST 2016 Task2
10
Tue Sep 27 22:06:51 CST 2016 Task1
Tue Sep 27 22:06:52 CST 2016 Task2
Tue Sep 27 22:06:53 CST 2016 Task1
13
Tue Sep 27 22:06:55 CST 2016 Task2
Tue Sep 27 22:06:55 CST 2016 Task1
15
Tue Sep 27 22:06:57 CST 2016 Task1
Tue Sep 27 22:06:58 CST 2016 Task2
17
Tue Sep 27 22:06:59 CST 2016 Task1
Tue Sep 27 22:07:01 CST 2016 Task2
Tue Sep 27 22:07:01 CST 2016 Task1
20
Tue Sep 27 22:07:03 CST 2016 Task1
Tue Sep 27 22:07:04 CST 2016 Task2
22
C:\java>

```
import java.util.*;
import java.io.IOException;
public class Timer5 {
  public static void main(String[] args) {
    TimerTask task = new TimerTaskCancel();
    Timer timer = new Timer();
    timer.scheduleAtFixedRate(task, new Date(), 5000);
    System.out.println("Time is :"+task.scheduledExecutionTime());
  }
  static class TimerTaskCancel extends java.util.TimerTask{
    public void run() {
      try {
          Thread.sleep(3000);
         System.out.println("Working on the task assigned "+this.scheduledExecutionTime());
       }
       catch (Exception e) {
             System.out.println("Error:"+e.getMessage());
       }
    }
  }
}
```


命令提示符
C:\java>javac Timer5.java
C:\java>java Timer5
Time is :1475038548996
Working on the task assigned 1475038548996
Working on the task assigned 1475038553996
Working on the task assigned 1475038558996
Working on the task assigned 1475038563996
Working on the task assigned 1475038568996
Working on the task assigned 1475038573996
Working on the task assigned 1475038578996
Working on the task assigned 1475038583996
Working on the task assigned 1475038588996
Working on the task assigned 1475038593996
Working on the task assigned 1475038598996
Working on the task assigned 1475038603996
Working on the task assigned 1475038608996
Working on the task assigned 1475038613996
Working on the task assigned 1475038618996
Working on the task assigned 1475038623996
Working on the task assigned 1475038628996
Working on the task assigned 1475038633996
Working on the task assigned 1475038638996
Working on the task assigned 1475038643996

Java的正则表达式

- 如果你需要在字符串中查找或替换具有某种模式的子串，你就可以正则表达式(Regular Expression)。

  例如：模式[Jj]ava.+可以用来查找以Java或java开头的字符串。其中，“[Jj]”表示第一个字符必须是J或j，然后必须出现三个字符“ava”，最后“.+”表示需要以一个或若干个任意字符结尾。字符串“Javanese”可以匹配该模式，而“core java”和“Java”则不能匹配该模式。

- 单字符选择匹配：[Jj]表示匹配一个字符J或j，其它例子：

[...]	位于括号之内的任意字符
[^...]	不在括号之中的任意字符
[abc]	查找方括号之间的任何字符，即a或b或c。
[^abc]	查找任何不在方括号之间的字符，即不包含字符a b c
[0-9]	查找任何从 0 至 9 的数字
[a-z]	查找任何从小写 a 到小写 z 的字符
[A-z]	查找任何从大写 A 到小写 z 的字符。
[A-Z0-9]	查找任何大写字母和数字

- 如果要表示所有字母数字，可以采用[a-zA-Z0-9]，有简化的写法吗？如果是任意字符怎么表示？可以用“\w” 表示[a-zA-Z0-9]，可以用“.”表示任意字符。这些都是转义字符的表示方法。转义字符还可以用于表示一些不可显示的字符。

\f	换页符
\n	换行符
\r	回车
\t	制表符
\v	垂直制表符
\/	字符/
\\	字符 \
\.	字符.
*	字符 *
\+	字符+
\?	字符?

|	字符|
\(	字符(
\)	字符)
\[	字符 [
\]	字符]
\{	字符 {
\}	字符 }
\0	字符nul
\nnn	十进制数 nnn指定的ASCII码字符
\xnn	十六进制数 nn 指定的ASCII码字符
\unn	十六进制数 nn 指定UNICODE码字符
\cX	控制字符^X。例如，\cI等价于\t，\cJ等价于 \n

.	除了换行符之外的任意字符，等价于 [^\n]
\w	任何单个字母数字，等价于[a-zA-Z0-9]
\W	任何非单个字母数字，等价于[^a-zA-Z0-9]
\s	任何空白符,等价于[\t\n\r\f\v]
\S	任何非空白符,等价于[^ \t\n\r\f\v]
\d	任何数字,等价于[0-9]
\D	除了数字之外的任何字符,等价于[^0-9]
\b	单词边界
\B	非单词边界
[\b]	一个退格直接量(特例)

- 量词模式“.+”可以用来匹配一个或若干字符，其它的量词匹配为:

x{n}	x出现n次
x{n, m}	x出现n~m次
x{n, }	x出现至少n次
x?	x出现0次或1次，等价于 {0, 1}
x+	x出现至少1次，等价于{1,}
x*	x出现至少0次，等价于{0,}

例子：

```
\d{2, 4}          //匹配2到4个数字。
\w{3} \d?         //匹配三个单字字符和一个数字。
\s+java\s+        //匹配字符串“java“ ,前后至少有一个空白符。
[^”] *            //匹配零个或多个非引号字符。
```

- 如果要表示以java结尾，可以使用位置匹配模式“java$”，所有的位置匹配模式如下：

x$	匹配任何结尾为 x 的字符串。
^x	匹配任何开头为 x 的字符串。
?=x	匹配任何其后紧接指定字符串 x 的字符串。
?!x	匹配任何其后没有紧接指定字符串 x 的字符串。

- 怎么表示分别以3个数字开头，然后大写字母\数字、大写字母、数字、…?

\d\d\d[A-Z]\d+ 可以吗？

\d\d\d([A-Z]\d)+ //( )表示一个整体，用于隔开前面的内容。

- 怎么表示分别3个小写字母，加3个数字或者6个数字，再加4个小写字母的模式呢?

  [a-z]{3}\d{3}|\d{6}[a-z]{4}

怎么表示3个小写字母接3个数字或者6个数字接4个小写字母的模式?

([a-z]{3}\d{3})|(\d{6}[a-z]{4}) // 这是一种多模式选择方式

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegEx {
   public static void main(String[] args) {
        Pattern pattern = Pattern.compile("b*g");
        Matcher matcher = pattern.matcher("bbg");
        System.out.println(matcher.matches());  //true
        System.out.println(pattern.matches("b*g","bbg")); //true

        getDate("Nov 10,2009");  //nov
        charReplace();           //okA，seeA A youA
        getChinese("Java,welcome,java!"); //Javajava
   }
   public static void getDate(String str){
        String regEx="([a-zA-Z]+)|//s+[0-9]{1,2},//s*[0-9]{4}";
        Pattern pattern = Pattern.compile(regEx);
        Matcher matcher = pattern.matcher(str);
        if(!matcher.find()){
            System.out.println("日期格式错误!");
            return;
        }
        //分组的索引值是从1开始的而不是0。
        System.out.println(matcher.group(1)); //nov
   }
```

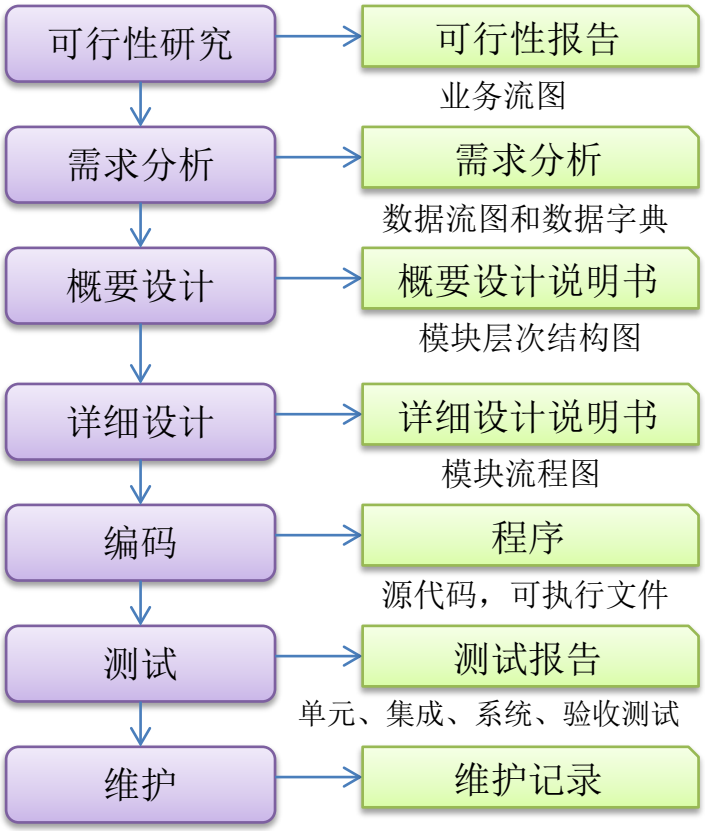
```
    public static void charReplace(){
        String regex = "a+";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher("okaa，seeaaa aa youa");
        String s = matcher.replaceAll("A");
        System.out.println(s); //okA，seeA A youA
    }
    public static void getChinese(String str){
        String regex = "[Jj]ava";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(str);
        StringBuffer sb = new StringBuffer();
        while(matcher.find()){
                sb.append(matcher.group());
        }
        System.out.println(sb); //Javajava
    }
}
```

软件设计

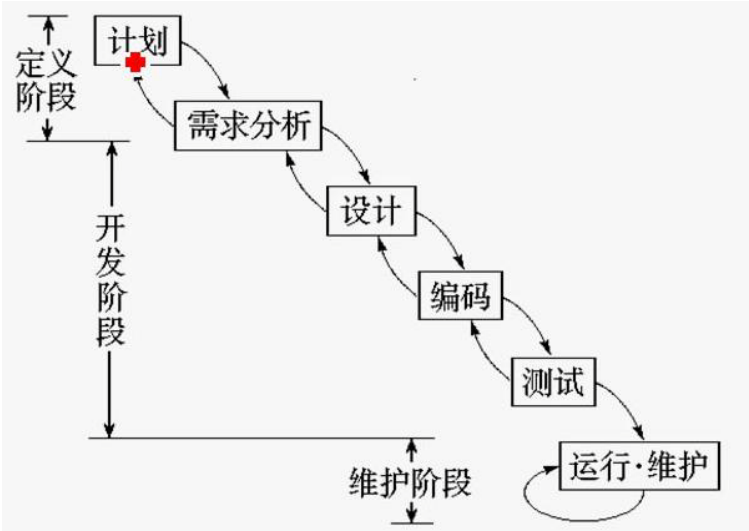
- 概述
  - 设计软件要采用工程化的方法进行，也就是采用**软件工程**的方法。软件工程需要对整个软件开发的整个过程进行计划和管理，并有一套控制软件产品质量的方法。
  - 软件开发过程的**管理方法**有瀑布模式、快速原型法、螺旋模型、敏捷开发四种模式。
  - 软件开发过程的**质量管理体系**主要有ISO9001 质量管理体系和软件能力成熟度模型。
  - 软件开发的**程序设计方法**主要有面向过程的程序设计方法和面向对象的程序设计方法。
  - 数据流图分析方法和UML建模方法分别为面向过程和面向对象目前使用的主要**分析方法**。
  - 就像一座高楼需要有好的结构，一个好的软件系统还需要有好的**软件系统构架**（框架），例如：三层结构（中间件）和MVC构架。

• 软件开发过程的管理方法

瀑布模型完成一个阶段后需要验收，然后只关注下一个阶段。

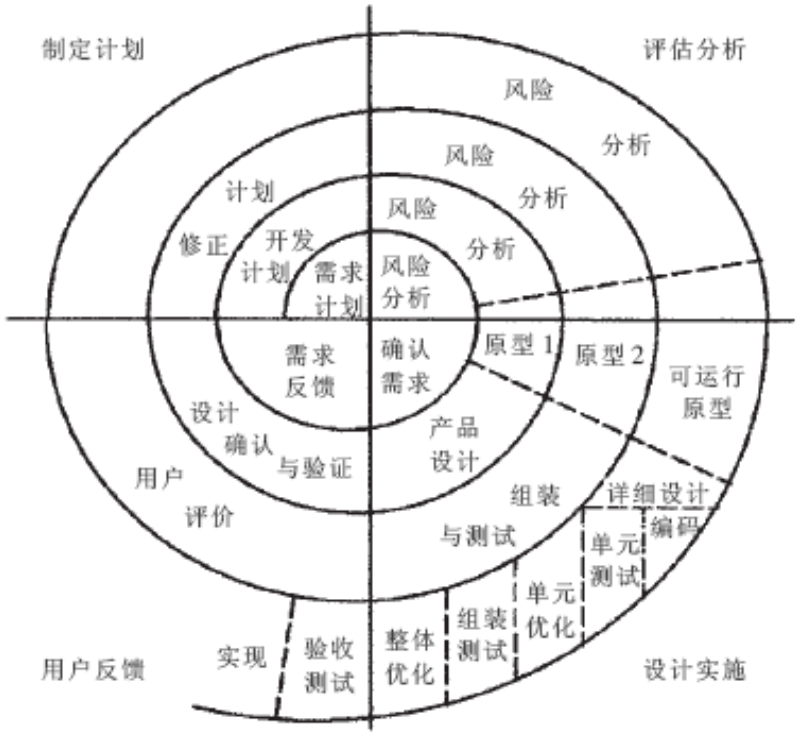


快速原型法(rapid prototyping)是一种迭代方法。它首先构造一个功能简单的原型系统，然后通过对原型系统逐步求精，不断扩充完善得到最终的软件系统。



快速迭代

螺旋模型将瀑布模型和快速原型模型结合起来，采用风险驱动的方法，在进入每个阶段及经常发生的循环之前都必须首先进行风险评估。螺旋模型的整个过程沿着螺线进行若干次迭代。



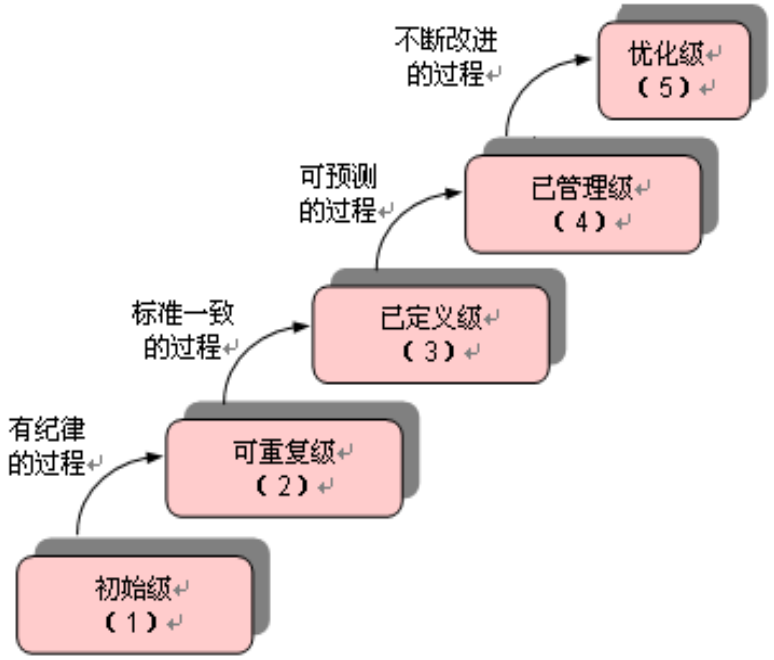
敏捷开发是1990年代出现的一种以人为核心、迭代、循序渐进的开发方法。在敏捷开发中，软件项目的构建被切分成多个子项目，各个子项目的成果都经过测试，具备集成和可运行的特征。换言之，就是把一个大项目分为多个相互联系，但也可独立运行的小项目，并分别完成，在此过程中软件一直处于可使用状态。

• 软件质量管理体系

ISO 9001质量认证体系用于证实组织具有提供满足顾客要求和适用法规要求的产品的能力，目的在于增进顾客满意。

1、强化品质管理，提高企业效益。
2、增强客户信心，扩大市场份额，在产品品质竞争中永远立于不败之地。
3、提高全员质量意识，改善企业文化。
4、第三方认证，提供最广泛的认可，节省了第二方审核的精力和费用。
5、有效地避免产品责任。
6、获得了国际贸易"通行证"，消除了国际贸易壁垒。
7、法律责任减免：如更容易的许可，更少的检查以及简化的报告要求等。
8、公众形象及社会关系，为消费者选择提供信心。

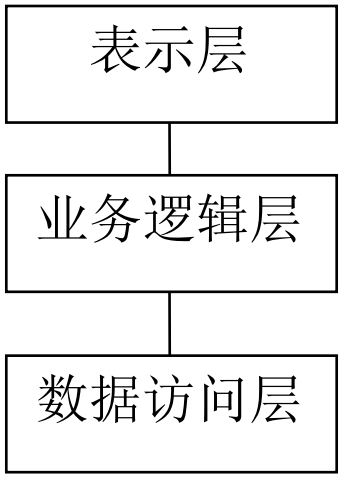
软件能力成熟度模型（Capability Maturity Model for Software，SW-CMM或CMM）是对于软件组织在定义、实施、度量、控制和改善其软件过程的实践中各个发展阶段的描述。



- 软件设计构架

➢三层体系结构

所谓**三层体系结构**，是在客户端与数据库之间加入了一个中间层，用来处理业务规则、数据访问、合法性校验。客户端不直接与数据库进行交互，而是通过与中间层访问数据库。C/S和B/S开发模式都可以采用三层架构。



接收用户的输入并把业务逻辑层处理的结果提交给用户，表现为人机接口或客户端界面。

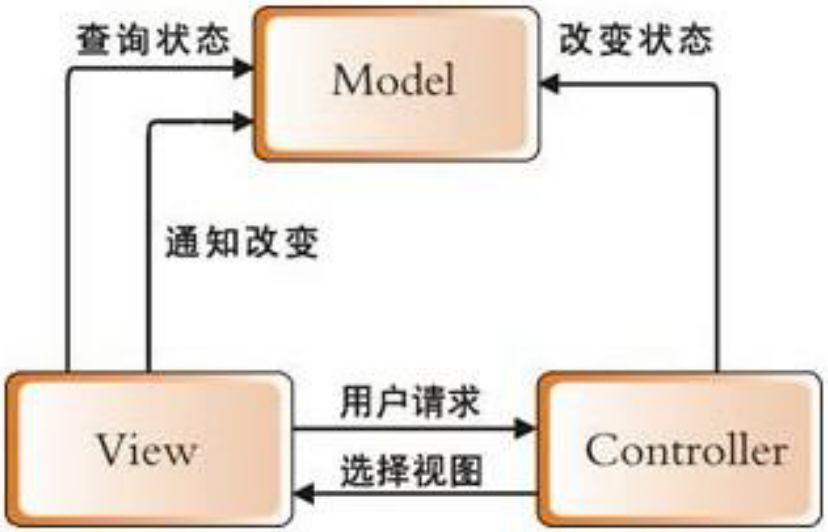
根据业务规则处理表示层和数据访问层提供的数据，并把结果交给数据访问层保存或在表示层显示出来。

为业务逻辑层或表示层提供数据服务。

➢MVC构架

MVC(Model-View-Controller)构架 由用户界面(View)、数据模型(Model)和控制器(Controller)三部分组成。

模型负责对数据访问；**视图**负责将数据显示和输入；**控制器**负责处理系统中的业务逻辑，并在需要时更新模型和视图。控制器解耦了模型和视图。



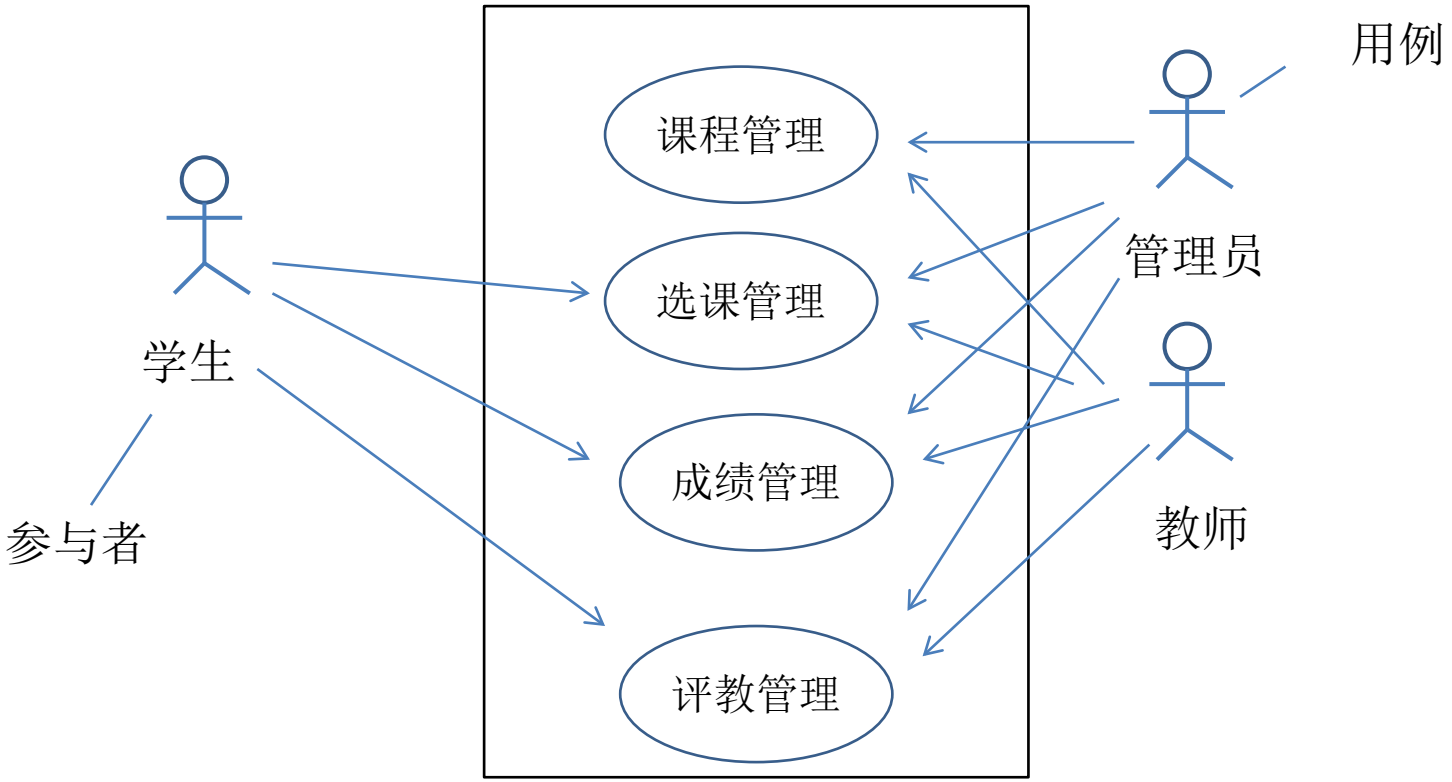
- UML

统一建模语言(Unified Modeling Language，UML) 是一个支持整个软件系统开发过程的建模语言。其基本模型包括：

用例图：表达系统外部的执行者与系统用例之间的关系。
类图：展示系统中类的静态结构
对象图：一种实例化类图
包图： 表示包与包之间的关系
状态图：描述一类对象具有的所有可能的状态及其转移关系
时序图/顺序图：展示对象之间随时间推移交换消息的过程
合作图：展示对象间动态协作关系，突出消息收发关系
活动图：展示系统中各种活动的执行流程和执行顺序
构件图：展示程序代码的组织结构和各种构件之间的依赖关系
配置图：展示软件在硬件环境中(特别是在分布式及网络环境中)的配置关系

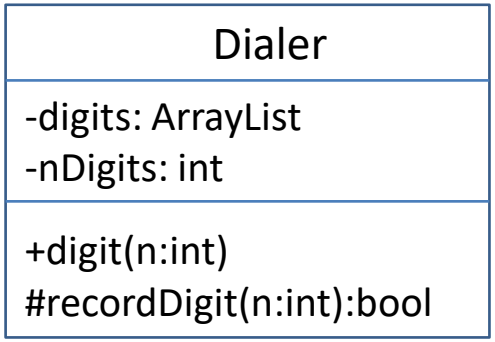
用例图

由参与者（Actor）、用例（Use Case）以及它们之间的关系构成的用于描述系统功能的静态视图称为**用例图**。



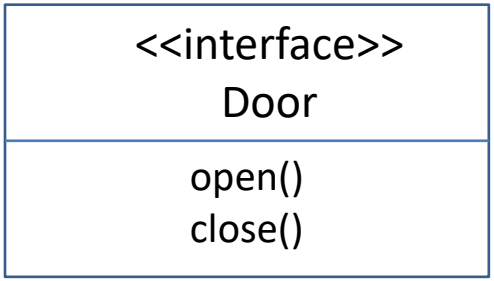
类的表示


```
public class Dialer{

}
```

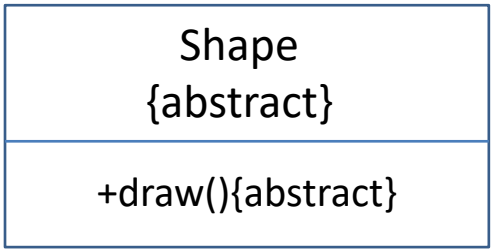



```
public class Dialer{
    private ArrayList digits;
    private int nDigits;
    public void digit(int n){...};
    protected bool recordDigit(int n){}...;
}
```

- private
+public
#protected

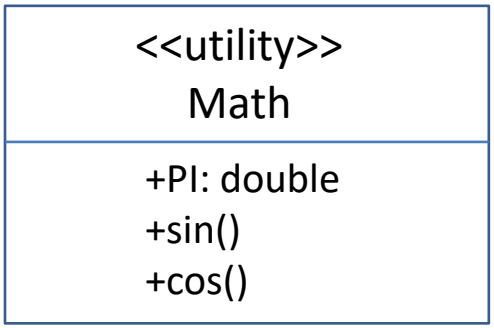



```
public interface Door {
   void open();
   void close();
}
```




```
public abstract class Shape {
   public void draw();
}
```

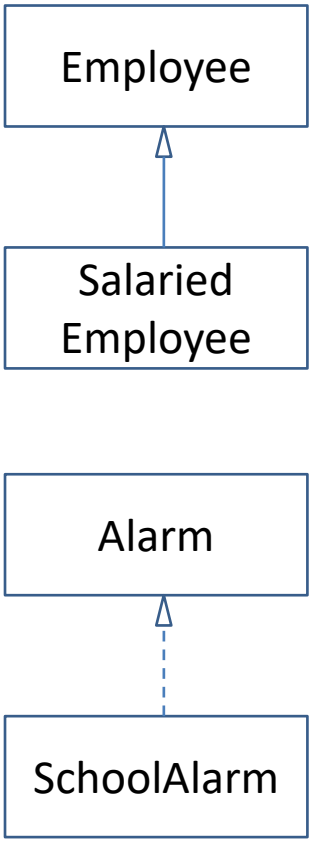
其它表示法：把名字写成斜体或在框上加(A)




```
public class Math{
   public static double PI=3.1415926
   public static double sin(double theta){...};
   public static double cos(double theta){...};
}
```

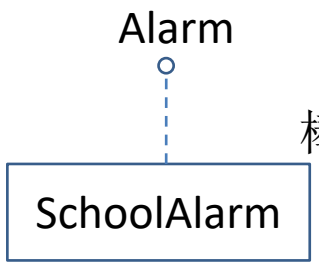
工具类：全部采用静态变量和静态方法

继承




```
public class Employee{
   ...
}

public class SalariedEmployee  extends Employee {
   ...
}
```

```
public interface Alarm{
   ...
}
public class SchoolAlarm  implements  Alarm {
   ...
}
```

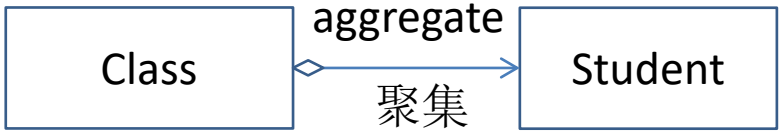


棒棒糖形状表示接口

关联

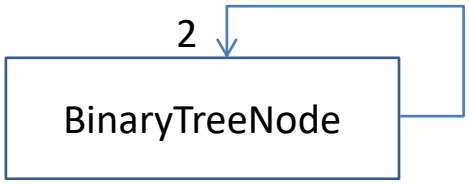

```
public class Car{
   private Wheel[4] itsWheels;
}
```


```
public class Phonebook{
   private ArrayList itsPnos;
}
```

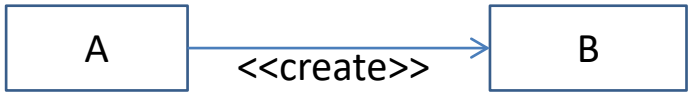



```
public class Class{
   private Student[4] students;
}
```

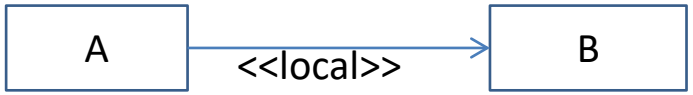

```
public class House{
   private Door itsDoor;
}
```



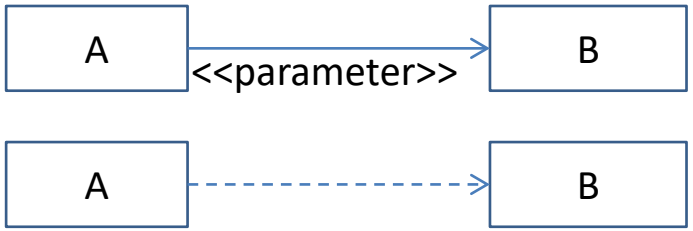

```
public class BinaryTreeNode{
   private BinaryTreeNode leftNode;
   private BinaryTreeNode rightNode;
}
```




```
public class A{
   private B makeB(){
     return new B();
   }
}
```

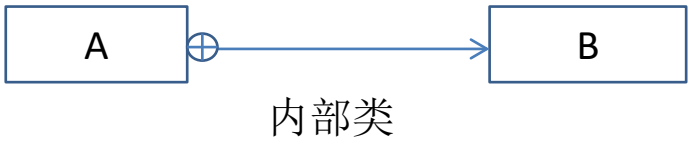



```
public class A{
   private void F(){
     B b=new B();
   }
}
```

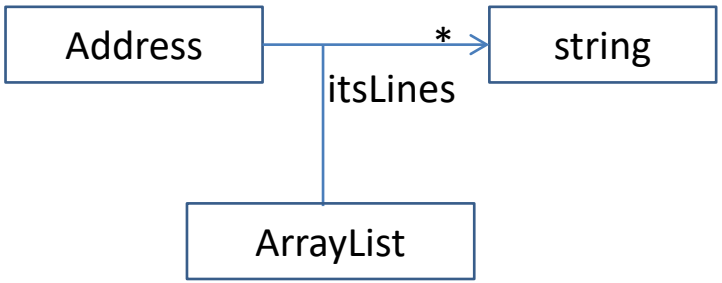



```
public class A{
   public void F(B b){
   }
}
```


```
public class A{
   private B itsB;
   public void F(){
      itsB.F();
   }
}
```



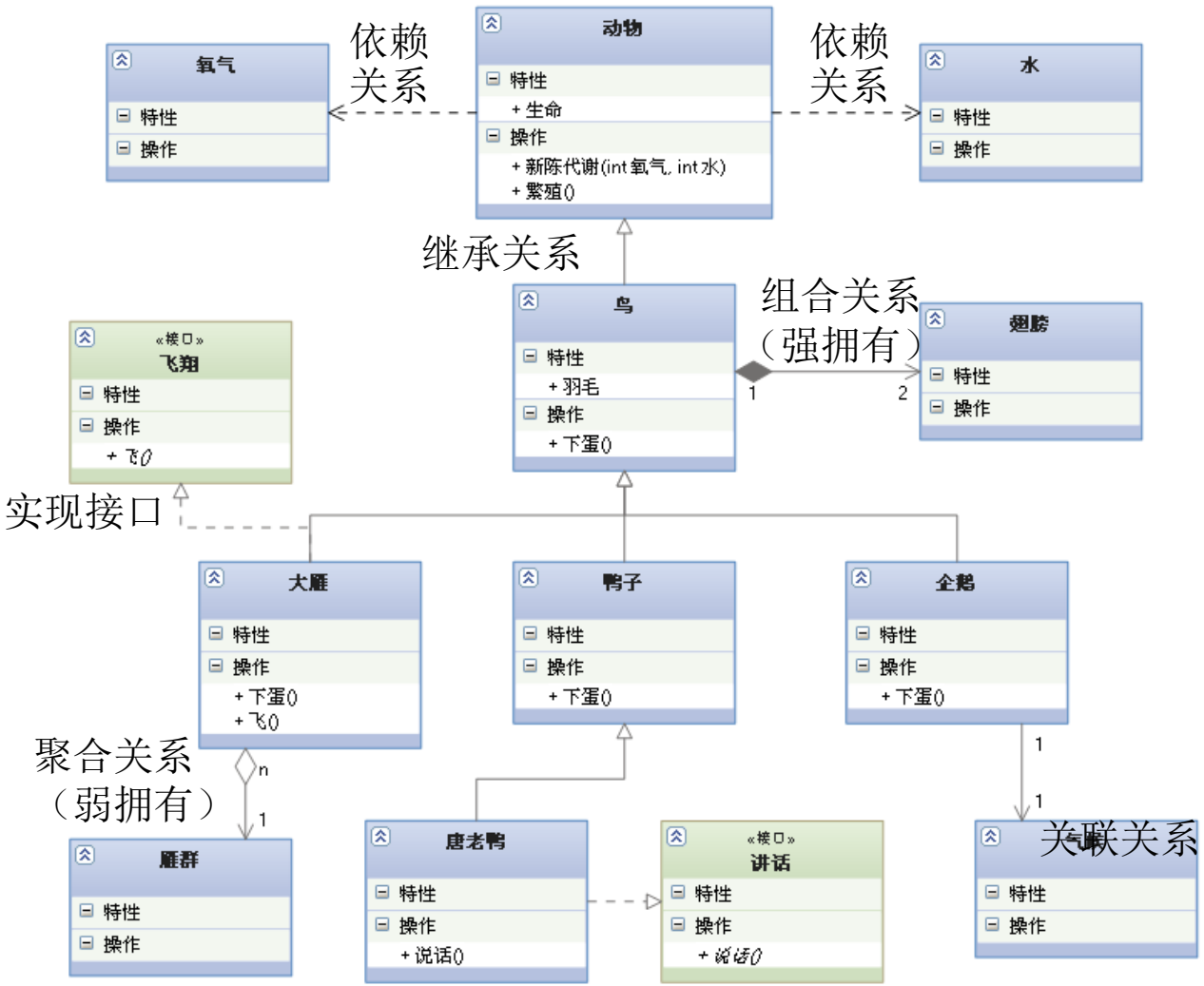

```
 public class A{
    private class B {
       ...
    }
 }
```




```
  public class Adress{
     private ArrayList itsLines;
  }
```

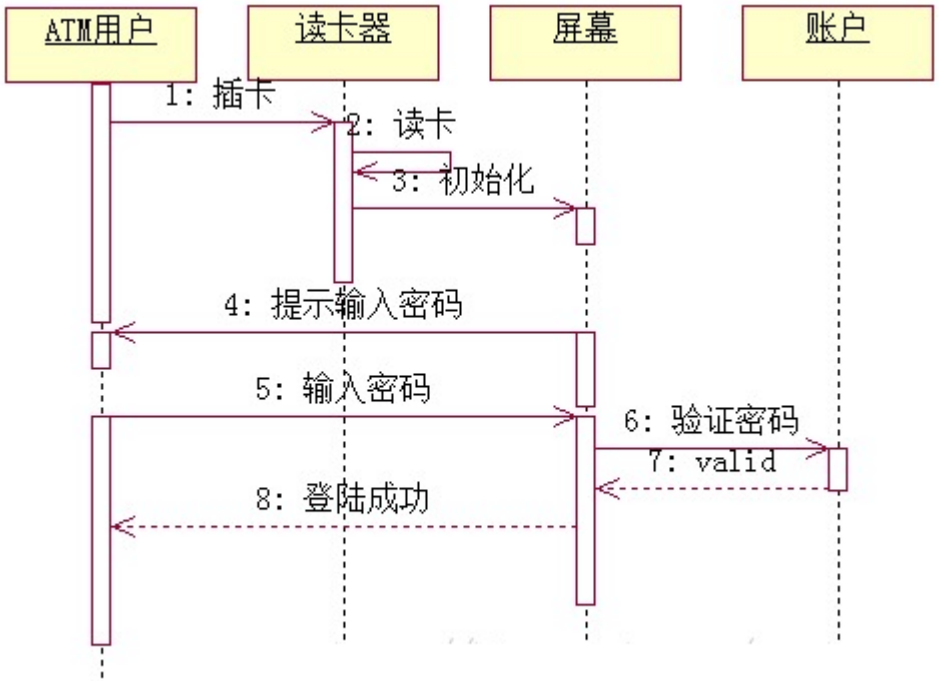

```
    public class LoginTransaction {
       private String empid
       private ArrayList itsLines;
    }
```

类图


氧气
特性
操作
依赖关系
动物
特性
+ 生命
操作
+ 新陈代谢(int 氧气, int 水)
+ 繁殖()
依赖关系
水
特性
操作
继承关系
«接口»
飞翔
特性
操作
+ 飞()
鸟
特性
+ 羽毛
操作
+ 下蛋()
组合关系
（强拥有）
1
2
翅膀
特性
操作
实现接口
大雁
特性
操作
+ 下蛋()
+ 飞()
鸭子
特性
操作
+ 下蛋()
企鹅
特性
操作
+ 下蛋()
聚合关系
（弱拥有）
n
1
雁群
特性
操作
唐老鸭
特性
操作
+ 说话()
«接口»
讲话
特性
操作
+ 说话()
1
1
关联关系
特性
操作

顺序图

顺序图描述了对象之间传送消息的时间顺序，用来表示用例中的行为顺序。当执行用例时，顺序图中的每条消息对应了一个类操作或者引起转换的触发事件。



敏捷软件开发的原则

- 敏捷软件开发遵循的原则有开闭原则(Open Closed Principle，OCP)、里氏代换原则(Liskov Substitution Principle，LSP)、依赖倒转原则(Dependency Inversion Principle，DIP)、接口隔离原则(Interface Segregation Principle，ISP)、合成复用原则(Composite/Aggregate Reuse Principle，CARP)、最小知识原则(Principle of Least Knowledge，PLK，也叫迪米特法则)。具体说明如下：

开闭原则： 模块应尽量在不修改原代码(闭)的情况下进行扩展(开)。

里氏代换原则： 如果调用的是父类的话，那么换成子类也完全可以运行。

依赖倒转原则： 抽象不应该依赖于细节，细节应当依赖于抽象。

接口隔离原则： 每一个接口应该是一种角色，不干不该干的事，该干的事都要干。

合成复用原则： 要尽量使用组合/聚合，尽量不要使用继承。只有"Is-A"或"Is-Like-A"关系才符合继承关系，“Has-A”关系应当使用聚合来描述。

最少知识原则： 不要和陌生人说话，即一个对象应对其它对象有尽可能少的了解。

* 组合：一辆车包含车轮、底盘、发动机
* 聚合：一个班级包含很多学生

设计模式

- 为了代码重用和增加软件可维护性，1994年，E. Gamma等人出版了一本关于设计模式（Design pattern）的书，提供了一套解耦复杂模块的模板或方法 。
- 设计模式举例：
  - ✓ **工厂（Factory）模式**用于不指定特定的"类"而生成某类的对象。
  - ✓ **单一状态（Monostate）模式**用同一个类的对象共享状态。
  - ✓ **单件（SingleTon）模式**只产生一个对象，要利用private构造函数和静态方法实现。
  - ✓ **适配器（Adapter）模式**将一个类的接口转换成客户希望的另外一个接口。
  - ✓ **观察者（observer）模式**让多个观察者对象同时监听某一个主题对象。这个主题对象在其状态发生变化时会通知所有观察者对象，使它们能够做出响应。
  - ✓ **外观（facade）模式**为子系统中的各类（或结构与方法）提供一个简明一致的界面，隐藏子系统的复杂性，使子系统更加容易使用。
  - ✓ **职责链（chain of responsibility）模式**采用令多个对象都有可能处理请求的方式解耦请求和接收者对象。将这些接收者对象形成一条链，让请求沿着这条链传递，直到有一个对象处理它为止。
  - ✓ **委托（Delegate）模式**是通过一个对象调用另一个对象的方法。
  - ✓ **代理（proxy）模式**用于间接访问一个对象。
  - ✓ **表数据通路（Table Data Gateway）模式**采用gateway隔离业务和数据操作，类似于三层设计。每个表对应一个类，对该表(table)的操作要通过Gateway接口。 Gateway接口的一个实现（类）专门处理该表的插入、删除、查询和更新操作。

- # 工厂模式

工厂（Factory）模式用于不指定特定的"类"而生成某类的对象。例如，只需指定父类，就可以创建子类的对象。

```
public class TestCreateCircle{
  public static void main(String[] args){
    IShapeFactory factory = new ShapeFactory();
    Shape s = factory.Make("Circle");
    s.draw();
  }
}
abstract class Shape{
    String type;
    String color;
    abstract void draw();
}
class Circle extends Shape {  //还可以定义其它形状
    void draw(){
         System.out.print("Circle is drawn!");
    }
}
interface IShapeFactory {
    Shape Make(String name);
}
class ShapeFactory implements IShapeFactory{
    public Shape Make(String shape){    //用Make（）产生新形状
         if(shape.equals("Circle"))return new Circle();
    }
}
```

• 单一状态模式

单一状态（Monostate）模式用同一个类的对象共享状态，采用静态成员变量实现。

```
public class TestMonostate {
  public static void main(String[] args){
      Monostate s1 = new Monostate();
      s1.setState(10);
      Monostate s2 = new Monostate();
      s2.setState(12);
      System.out.println(s1.getState()); //显示12
      System.out.println(s2.getState()); //显示12
  }
}
class Monostate{
    private static int state=0;  //可以定义多个状态
    MonoState(int state){
         Monostate.state = state;
    }
    void setState(int state){
        Monostate.state = state;
    }
    int getState(){
        return Monostate.state;
    }
}
```

- 单件模式

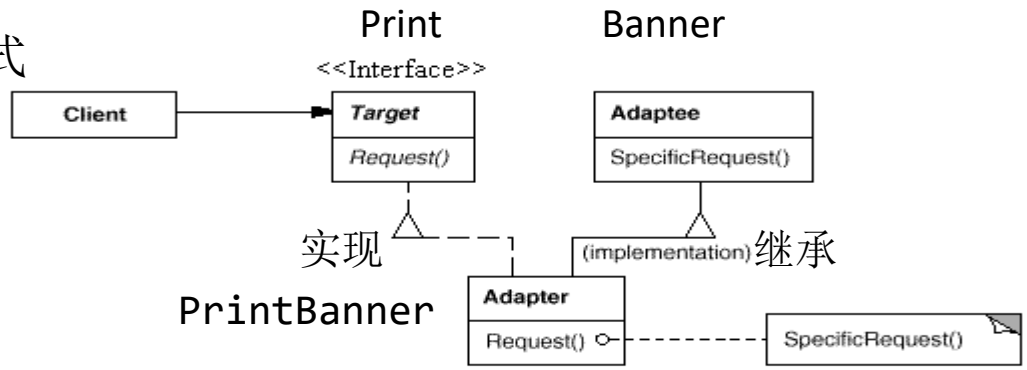
单件（SingleTon）模式只产生一个对象，要利用private构造函数和静态方法实现。

```
public class TestSingleton {
    public static void main(String[] args){
          Singleton s1 = Singleton.makeSingleton();
          s1.setName("Hello, Li!");
          Singleton s2 = Singleton.makeSingleton();
          s2.setName("Hello, Wang!");
          System.out.println(s1.getName());  //显示Hello, Wang!
          System.out.println(s2.getName());  //显示Hello, Wang!
    }
}
class Singleton {
    private String name;
    private static Singleton single = new Singleton();
    private Singleton(){   //不允许直接创建对象
    }
    void setName(String name){
        this.name=name;
    }
    String getName(){
        return this.name;
    }
    static Singleton makeSingleton(){  //这里并不建造新对象
        return single;
    }
}
```

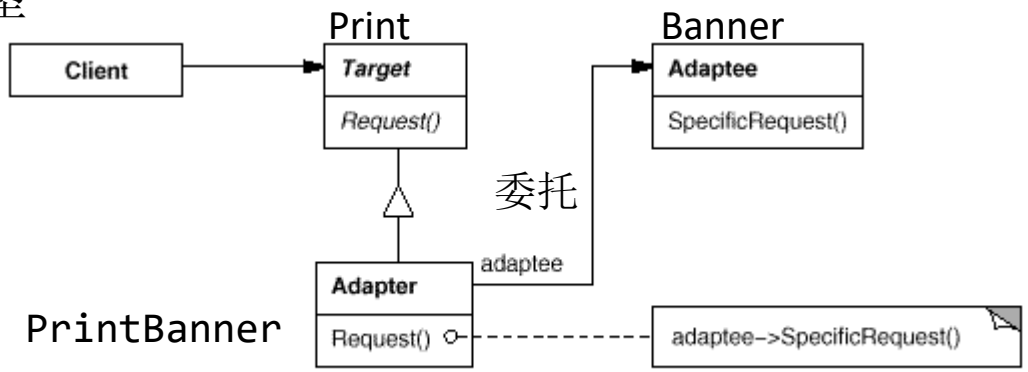
- Adapter模式

适配器（Adapter），也叫包装器（Wrapper），将一个类的接口转换成客户希望的另外一个接口。常用于将“既有内容”转换成“需要结果”，是代码重用的利器之一。

类适配器模式



对象适配器模型



示例-类适配器（继承）：

	电源	程序示例
既有内容	交流电220V	Banner类（showWithParen,showWithAster）
转换装置	适配器	PrintBanner类
需要的结果	直流电12V	Print接口 （PrintWeak， PrintStrong

```
public class TestAdapter {
  public static void main(String[] args){
     Print p = new PrintBanner("Hello");
     p.printWeak();
     p.printStrong();
  }
}
class Banner{
  private String strings;
  public Banner(String strings){
    this.strings = strings;
  }
  public String getBannerWithParen(){
     return "("+strings+")";
  }
  public String getBannerWithAster(){
     return "*"+strings+"*";
  }
}
```

```
class PrintBanner extends Banner implements Print{
  public PrintBanner(String strings){
    super(strings);
  }
  public void printWeak(){
    System.out.println(getBannerWithParen());
  }
  public void printStrong(){
    System.out.println(getBannerWithAster());
  }
}
interface Print{
  public abstract void printWeak();
  public abstract void printStrong();
}
```

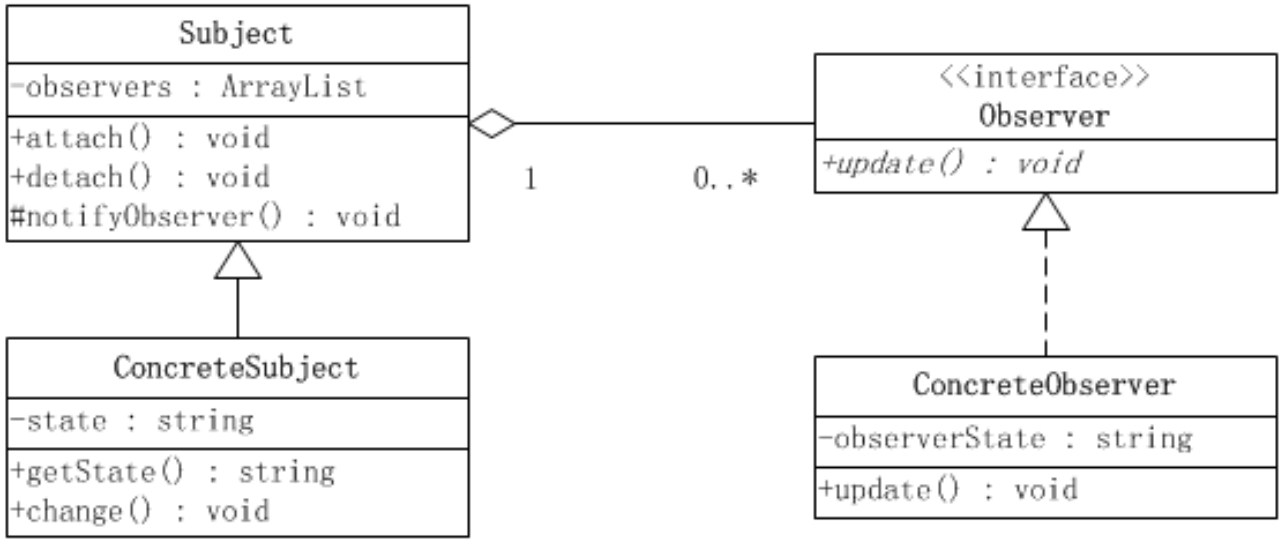
Java语言不允许多重继承！

示例-对象适配器（委托）：

```
// ... 这部分与上面的继承模式相同

class PrintBanner extends Print{
    private Banner banner;
    public PrintBanner(String strings){
       this.banner = new Banner(strings);
    }
    public void printWeak(){
        System.out.println(banner.getBannerWithParen());
    }
    public void printStrong(){
        System.out.println(banner.getBannerWithAster());
    }
}
abstract class Print{
  public abstract void printWeak();
  public abstract void printStrong();
}
```

- 观察者模式

观察者模式让多个观察者对象同时监听某一个主题对象。这个主题对象在其状态发生变化时会通知所有观察者对象，使它们能够做出响应。观察者模式又叫发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。

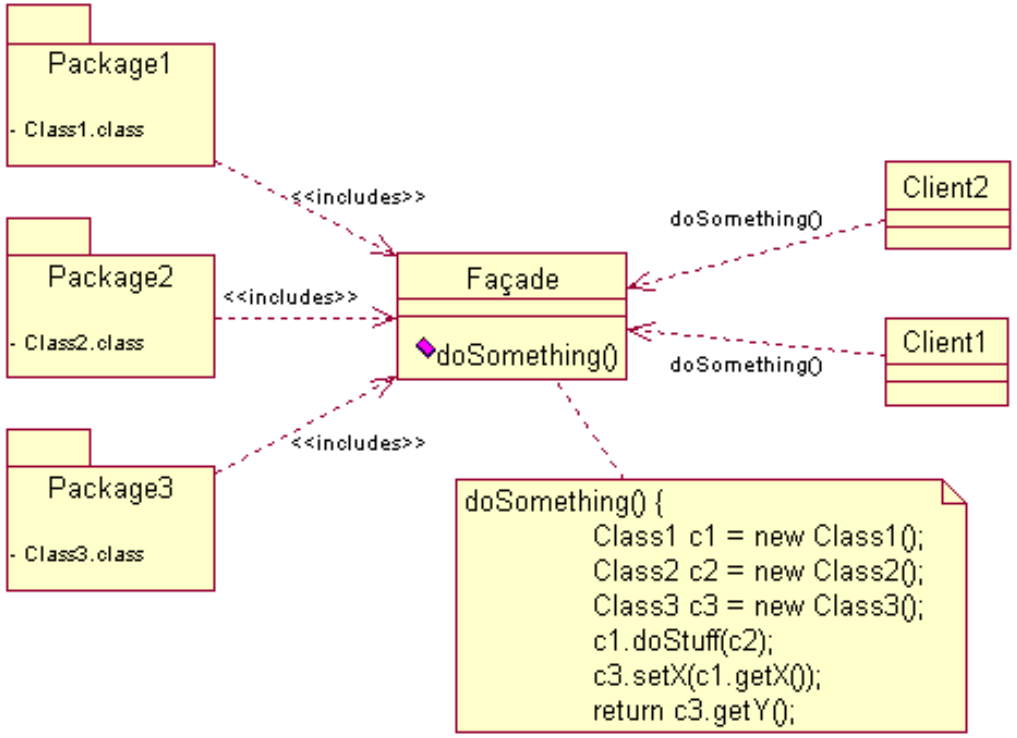


```
public abstract class Subject {
   private List<Observer> list = new ArrayList<Observer>(); //注册的观察者对象
   public void attach(Observer observer){              // 注册观察者对象
        list.add(observer);
        System.out.println("Attached an observer");
  }
  public void detach(Observer observer){               // 删除观察者对象
         list.remove(observer);
    }
    public void nodifyObservers(String newState){     //通知所有注册的观察者对象
        for(Observer observer : list){
            observer.update(newState);
        }
    }
}
public class ConcreteSubject extends Subject{
    private String state;
    public String getState() {
        return state;
    }
    public void change(String newState){
        state = newState;
        System.out.println("主题状态为: " + state);
        this.nodifyObservers(state); //状态发生改变，通知各个观察者
    }
}
```

```java
public interface Observer {
     public void update(String state);        //更新的状态
}
public class ConcreteObserver implements Observer {
    private String observerState;             //观察者的状态
    @Override
    public void update(String state) {
        observerState = state;    //更新观察者的状态，使其与目标的状态保持一致
        System.out.println("状态为："+observerState);
    }
}

public class Client {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject(); //创建主题对象

        Observer observer = new ConcreteObserver();     //创建观察者对象
        subject.attach(observer);              //将观察者对象登记到主题对象上

        subject.change("new state");           //改变主题对象的状态
    }
}
```

- 外观模式(facade)

外观模式为子系统中的各类（或结构与方法）提供一个简明一致的界面，隐藏子系统的复杂性，使子系统更加容易使用。也就是，只需要通过外观类中的方法使用，而不需要了解复杂的子系统。



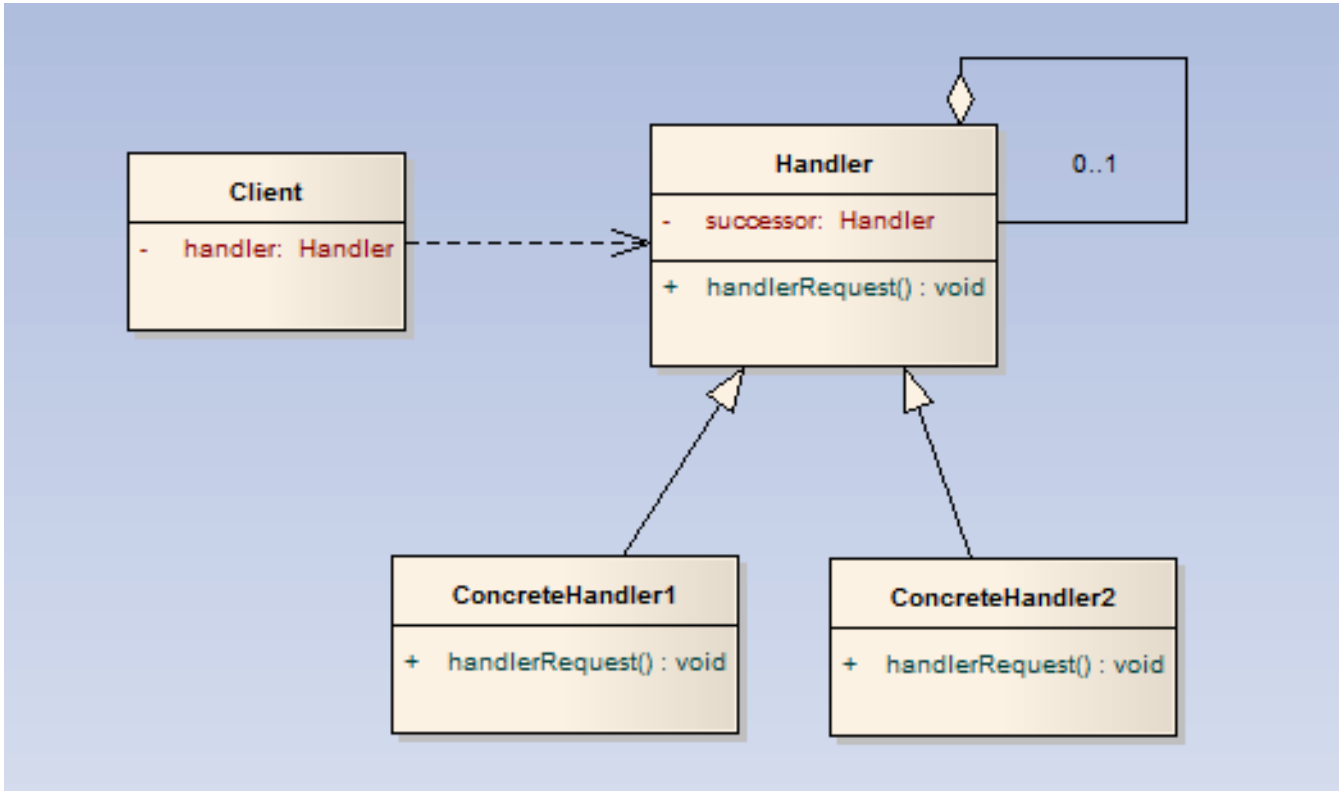
```
public class ServiceA {
        public void methodA() {
                System.out.println( "methodA--> is runing" );
        }
}

public class ServiceB {
        public void methodB() {
                System.out.println( "methodB--> is runing" );
        }
}

public class ServiceC {
        public void methodC() {
                System.out.println( "methodC--> is runing" );
        }
}
```

```java
/* 外观模式类  */
public class Facade {
    ServiceA sa;
    ServiceB sb;
    ServiceC sc;
    public Facade() {
        sa = new ServiceA();
        sb = new ServiceB();
        sc = new ServiceC();
    }
    public void method1() {
        sa.methodA();
        sb.methodB();
    }
    public void method2() {
        sb.methodB();
        sc.methodC();
    }
    public void method3() {
        sc.methodC();
        sa.methodA();
    }
}
```

```java
/* 测试 */
public class Client {
  public static void main(String[] args) {
    ServiceA sa = new ServiceA();
    ServiceB sb = new ServiceB();
    sa.methodA();
    sb.methodB();
    System.out.println("============");
    Facade f = new Facade();
    f.method1();
    f.method2();
    f.method3() ;
  }
}
```

- 职责链模式

职责链模式（chain of responsibility）采用令多个对象都有可能处理请求的方式解耦请求和接收者对象。将这些接收者对象形成一条链，让请求沿着这条链传递，直到有一个对象处理它为止。



```
public abstract class Handler {
  protected Handler successor;
  public abstract void handlerRequest(String condition);
  public Handler getSuccessor() {
    return successor;
  }
  public void setSuccessor(Handler successor) {
    this.successor = successor;
  }
}
public class Handler1 extends Handler {
  @Override
  public void handlerRequest(String condition) {
    if(condition.equals("Handler1")){       // 自己处理或传给下家
      System.out.println( "Handler1 handled ");
      return;
    }else{
      System.out.println( "Handler1 passed ");
      getSuccessor().handlerRequest(condition);
    }
  }
}
```

```
public class Handler2 extends Handler {
 @Override
 public void handlerRequest(String condition) {
   if(condition.equals("Handler2")){
     System.out.println( "Handler2 handled ");
     return ;
   }else{
     System.out.println( "Handler2 passed ");
     getSuccessor().handlerRequest(condition);
   }
 }
}
/* ..... */
public class HandlerN extends Handler {
  /** 如果不是最后一个节点必须处理掉，也可以改变某些条件后，交给前面的节点处理，
      这会出现环形或树形传递。 */
  @Override
  public void handlerRequest(String condition) {
    System.out.println( "HandlerN handled");

  }
}
```

```
public class Client {
  public static void main(String[] args) {

    Handler handler1 = new Handler1();
    Handler handler2 = new Handler2();
    Handler handlern = new HandlerN();

    //把所有接收对象链起来
    handler1.setSuccessor(handler2);
    handler2.setSuccessor(handlern);

    //假设这个请求是Handler2的责任
    handler1.handlerRequest("Handler2");
  }
}
```

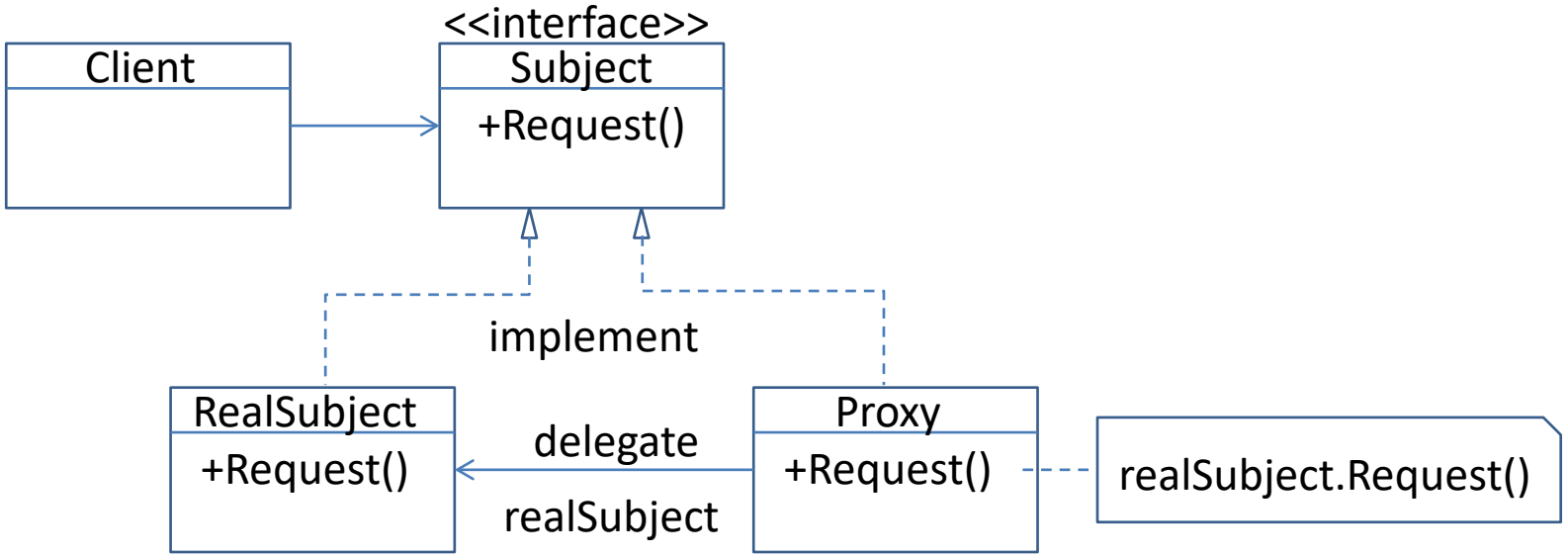
- 委托模式

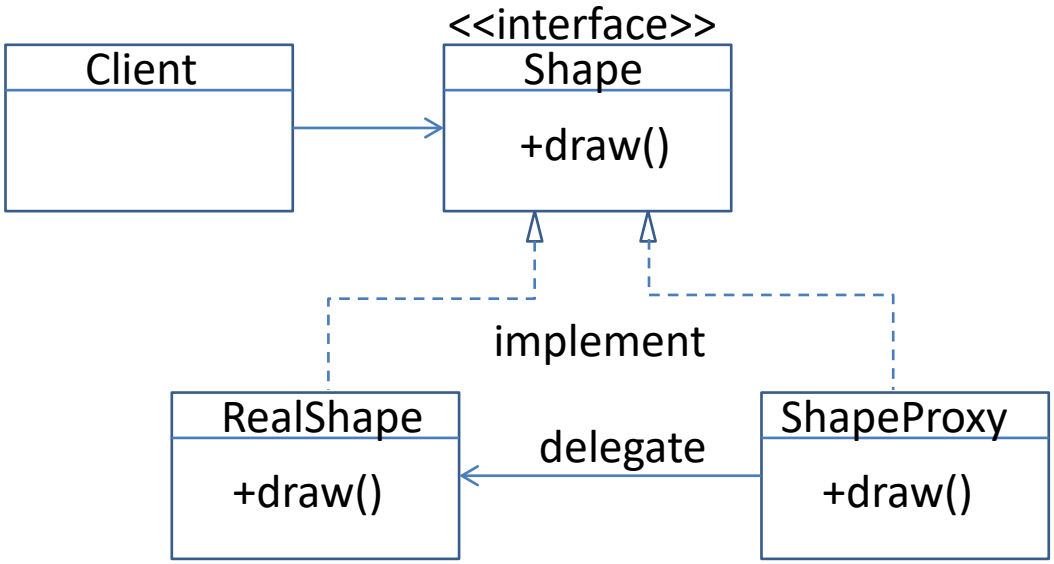
委托（Delegate）模式是通过一个对象调用另一个对象的方法。

```
public class TestDelegate {
      public static void main(String[] args){
          Car car = new Car();
          car.run();
          car.stop();
      }
}
class Engine{
    void run(){System.out.println("run!");}
    void stop(){System.out.println("stop!");}
}
class Car{
    Engine engine;
    Car(){ engine = new Engine(); }
    void run(){
        engine.run();
    }
    void stop(){
        engine.stop();
    }
}
```

- 代理模式

如果不希望直接访问一个对象，就可以使用代理（proxy）模式。代理模式能够隔离调用者和被调用者，降低系统的耦合度，但是使用代理模式可能会让系统变得非常复杂并且令系统处理速度变慢。





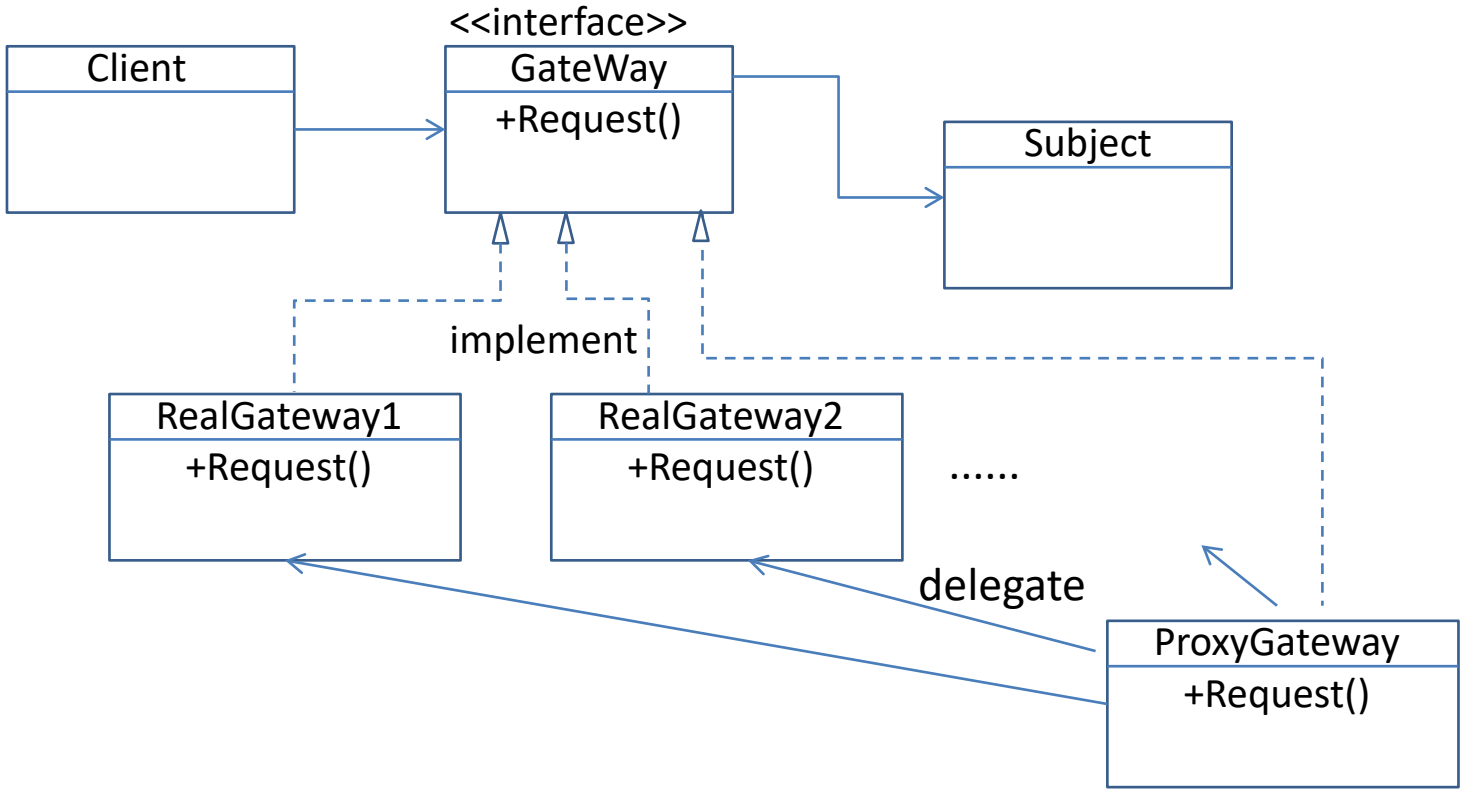
RealShape（Circle，Square）实现接口Shape，ShapeProxy通过委托的方法也实现了接口Shape。Client通过ShapeProxy使用RealShape。

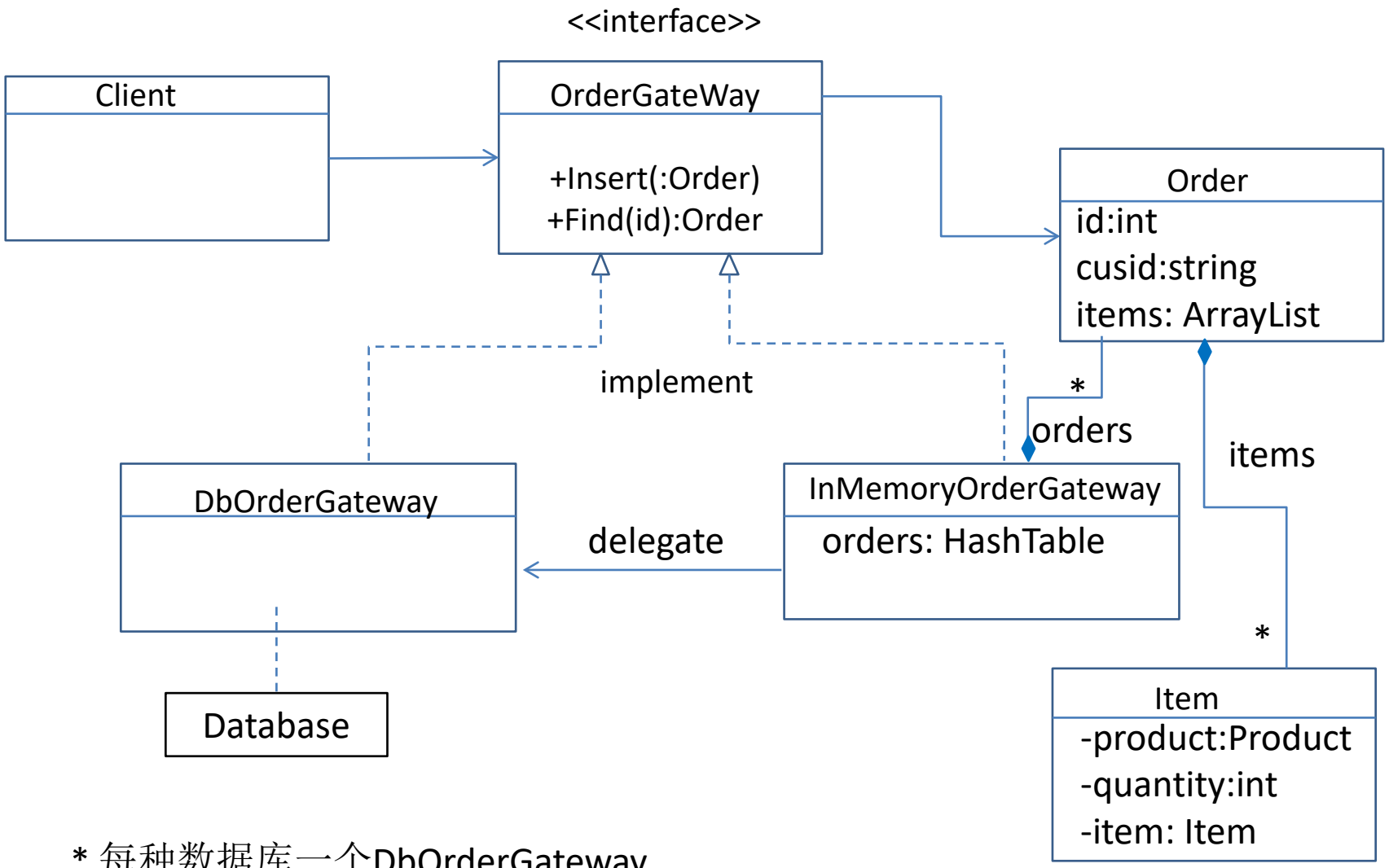
```
public interface Shape {
  public void draw();
}
public class Circle implements Shape {
  public void draw() { System.out.println( "Draw a circle!"); }
}
public class Square implements Shape {
  public void draw() { System.out.println( "Draw a square!"); }
}
public class ShapeProxy implements Shape {
  Shape shape;
  ShapeProxy(String shapeType) {
     if(shapeType.equals("circle")){
       shape = new Circle();
     }
     else if(shapeType.equals("square")){
       shape = new Square();
     }
     else {  System.out.println( "Shape Type not exist!");
     }
  }
  public void draw() {shape.draw();}
}
```

```
public class Client {
  public static void main(String[] args) {
        ShapeProxy shape1 = new ShapeProxy("circle");
        ShapeProxy shape2 = new ShapeProxy("square");
        draw(shape1);
        draw(shape2);
  }
  public static void draw(ShapeProxy shape) {
          shape.draw();
  }
}
```

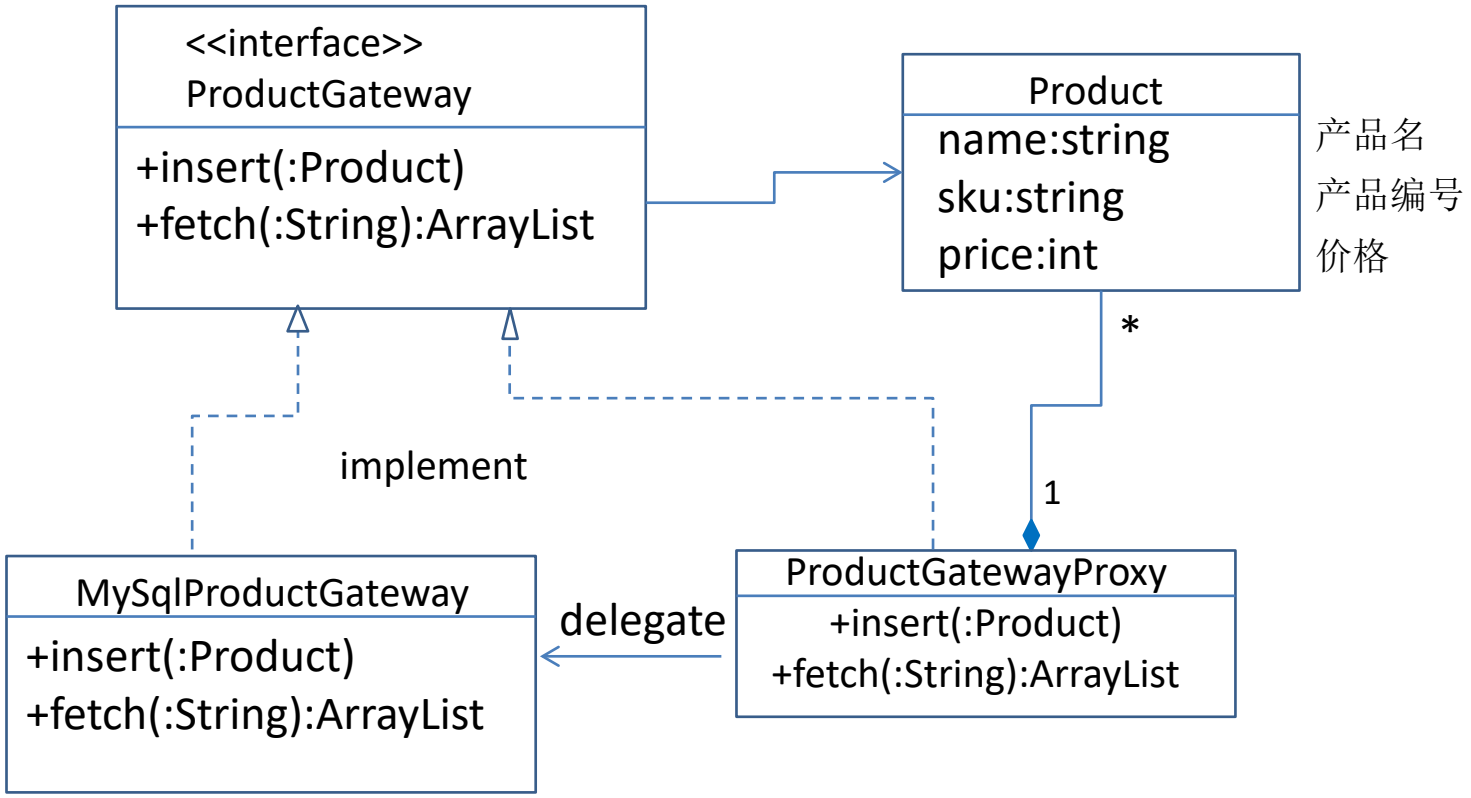
- Table Data Gateway模式

表数据通路（Table Data Gateway）模式采用gateway隔离业务和数据操作，类似于三层设计。每个表对应一个类，对该表(table)的操作要通过Gateway接口。 Gateway接口的一个实现（类）专门处理该表的插入、删除、查询和更新操作。





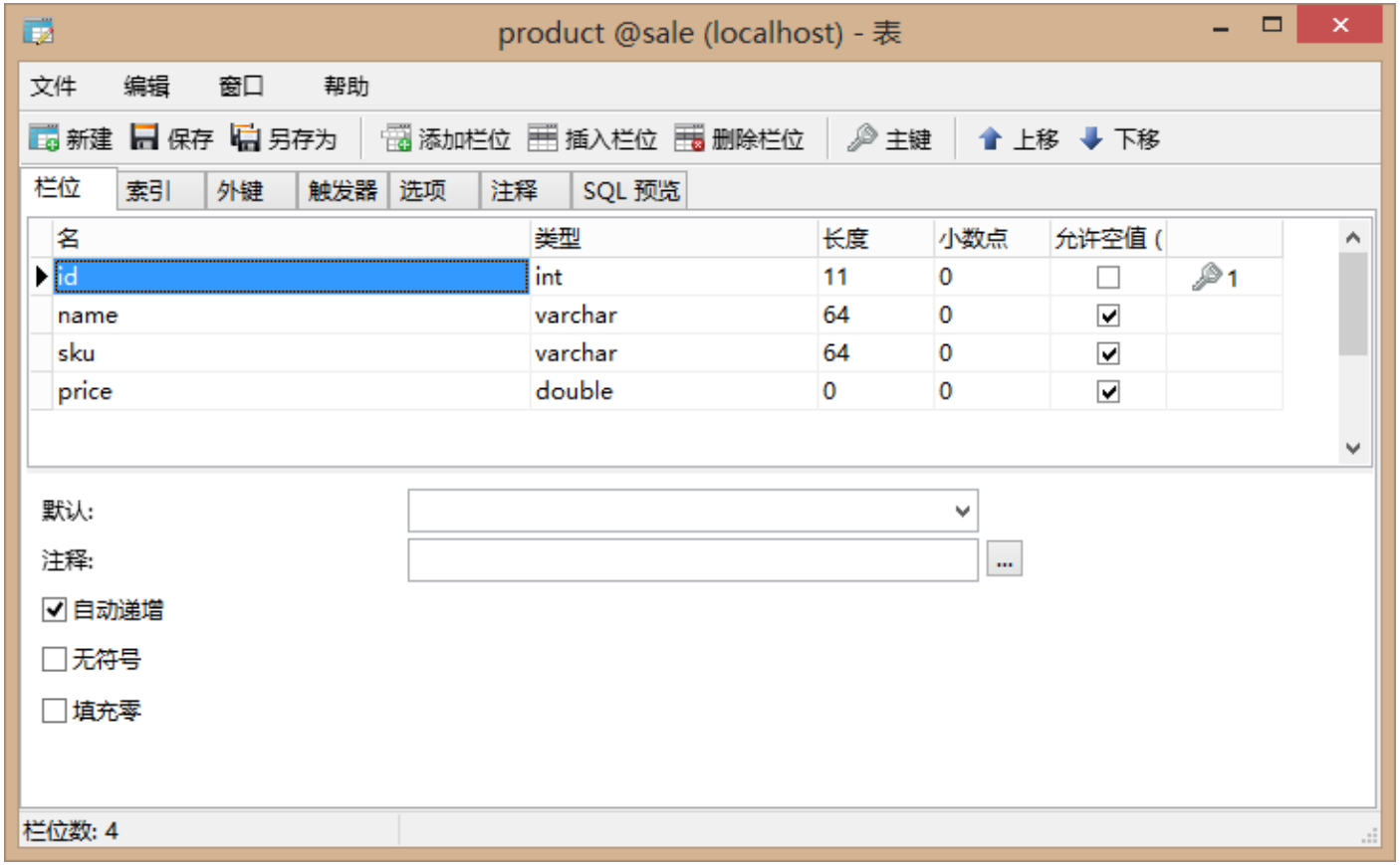
* 每种数据库一个DbOrderGateway



这里采用了MySql数据库，改为其它数据库或其他数据存储方式时业务和显示部分只需要做很少的改动。

SKU（Stock Keeping Unit）原来表示库存量单位，现在表示产品统一编号，每个产品具有唯一的SKU。

数据库：sale
表：product



```java
//Product.java
package bean;
public class Product {
        private String name;    //产品名
        private String sku;     //库存单位
        private double price;       //单价
        public String getName(){
                return name;
        }
        public String getSku(){
                return sku;
        }
        public double getPrice(){
                return price;
        }
        public Product(String name,String sku,double price){
                this.name=name;
                this.sku=sku;
                this.price=price;
        }
}
```

```java
//ProductGateway.java
package bean;
import java.util.*;
public interface ProductGateway {
        public boolean insert(Product product);
        public ArrayList<Product> fetch(String condition) throws Exception;
        public String getMessage();
}

//MySqlProductGateway
package bean;
import java.sql.*;
import java.util.ArrayList;
public class MySqlProductGateway implements ProductGateway{
        String msg="";
        DataConnection conn;

        public MySqlProductGateway(DataConnection conn){
                this.conn=conn;
        }

        public String getMessage(){
                return msg;
        }
```

```
public boolean insert(Product product){
        if(!conn.isUnique("product", "sku", product.getSku())){
                msg = "sku已存在！";
                return false;
        }
        String sql=String.format("insert into product(sku,name,price)"
                                        + "values('%s','%s',%f)",
                        product.getSku(), product.getName(),
                        product.getPrice());
        if(!conn.executeUpdate(sql)){
                msg = conn.getMessage();
                return false;
        }
        msg = "操作成功！";
        return true;
}
```

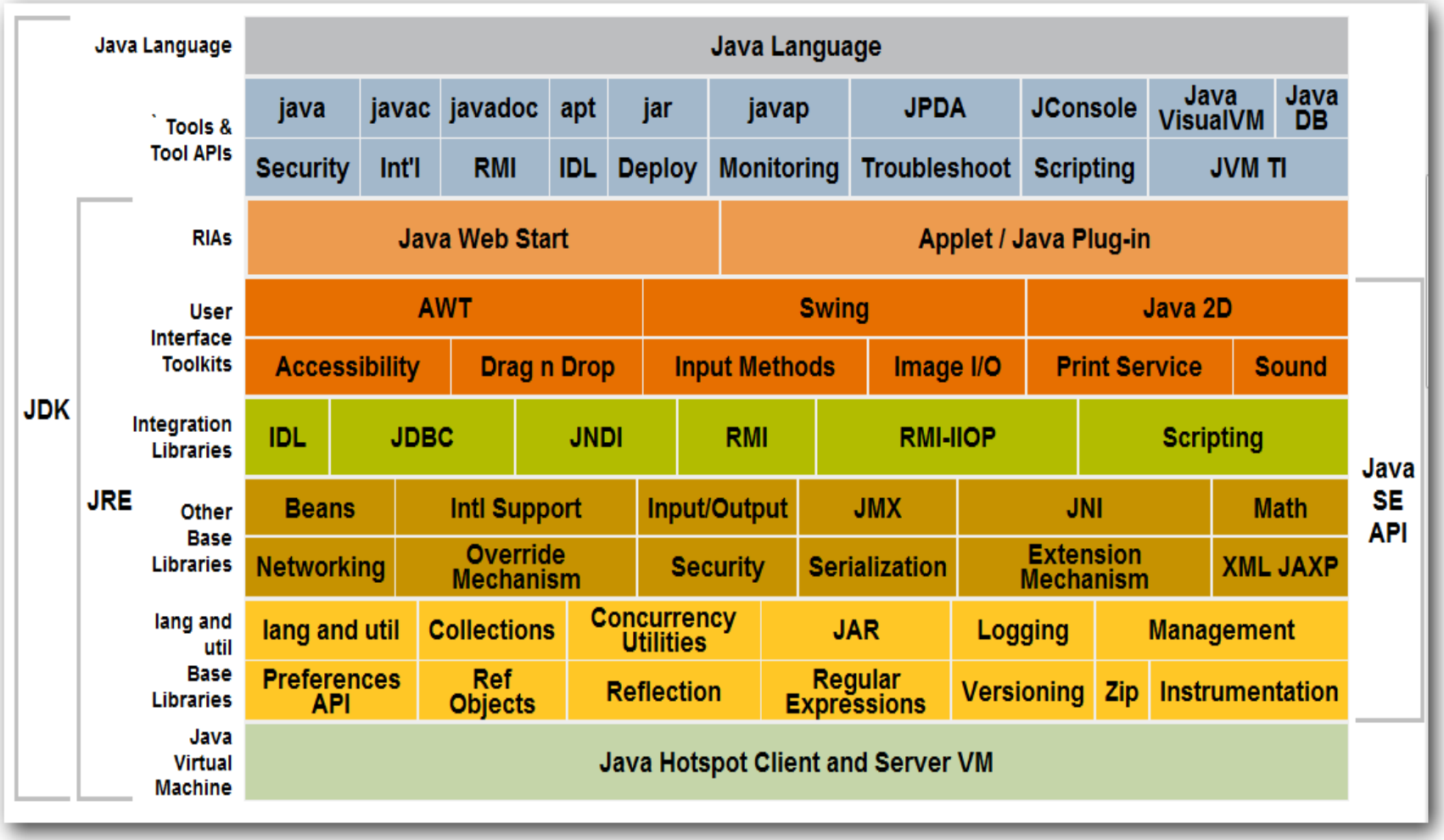
```
        public ArrayList<Product> fetch(String condition)
                        throws Exception{
                ArrayList<Product> products= new ArrayList<Product>();
                String sql=String.format("select * from product %s",
                                (condition.isEmpty())?"":" where "+condition);
                ResultSet rs = conn.executeQuery(sql);
                msg = conn.getMessage();
                while(rs!=null && rs.next()){
                     products.add(new Product(rs.getString("name"),
                                rs.getString("sku"),rs.getDouble("price")));
                }
                return products;
        }
}
```

如果是订单类还要用聚合的方法(ArrayList)包含订单明细（Item类）。 Item类包含了产品编号和购买单价。订单类和Item类也要定义Gateway访问数据库。

```
//ProductGatewayProxy
package bean;
import java.sql.*;
import java.util.ArrayList;
public class ProductGatewayProxy implements ProductGateway{
        String msg="";
        DataConnection conn;
        ProductGateway prodGateway;

        public ProductGatewayProxy(DataConnection conn,String dbType){
            this.conn=conn;
            if(dbType.toLowerCase().equals("mysql")){
              prodGateway = new MySqlProductGateway(conn);
            }
        }

        public String getMessage(){
                return prodGateway.getMessage();
        }
        public boolean insert(Product product){
        }
        public ArrayList<Product> fetch(String condition)
                        throws Exception{
           return prodGateway.fetch(condition) ;
        }
}
```

附录、Java平台的组件

- 下面的概念图展示了Java平台的所有组件以及其它们如何组合起来的。



- JDK的主要程序

javac	把Java源代码文件(.java)编译为Java字节码文件(.class) 。 $javac ShowPhoto.java
java	用于在Java虚拟机上执行Java字节码文件。$java ShowPhoto
appletviewer	用来执行HTML文件中的Java小程序代码。
javadoc	根据Java源程序中的说明语句生成HTML文档。
javap	用来显示字节码文件中字节码含义以及可访问方法和数据。
jar	Java用于发布的打包和解压软件(ZIP格式)。

直接执行命令java和javac可以看出是否了Java虚拟机和JDK。

```
$jar  cvf  ShowPhoto  ShowPhoto.class  pictureDir  videoDir  //压缩
$jar  tvf  ShowPhoto.jar                                    //显示
$jar  xf  ShowPhoto.jar                                     //解压
```

参考资料

- Bruce Eckel，Java编程思想（第4版），机械工业出版社，2012
- Cay S.Horstmann, G.Cornell, Java核心技术，机械工业出版社，2012
- Y. Daniel Liang，Java语言程序设计（第8版），机械工业出版社，2011
- http://docs.oracle.com/javase/8/docs/
- http://api.apkbus.com/reference/java/io/package-summary.html
- http://docs.oracle.com/javase/tutorial/
- Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides，Design Patterns：Elements of Reusable Object-Oriented Software，1994