1. 构造函数

当写一个java类的时候，如果没有给出该类的任何构造函数，那么会默认有有一个无参数的构造器。在获取类的实例的时候，可以调用无参构造函数，如图1所示。

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  Test t = **new** Test();  }  } |

但是，若在声明类的时候给出了构造函数，则默认的构造函数无效。想声明该类对象，则必须调用这个类明确给出的构造函数中的某一个。

|  |
| --- |
| **public** **class** Test {  **public** Test(**int** i){  }  **public** **static** **void** main(String[] args) {  Test t = **new** Test(2);  }  } |

如果还是强行调用默认的构造函数，就相当于是调用一个没有声明的方法，就会报错。

注意：类的构造器也是静态方法，虽然构造方法前面没有加static关键字。

构造函数还有以下特点：

① 构造函数必须与类的名字相同；

② 每个类可以有多个构造器；

③ 构造函数总是伴随着new操作一起调用，且不能由程序的编写者直接调用，必须由系统调用；

④ 构造函数的主要作用是完成对象的初始化工作。

1. 继承之子类调用父类构造器

Java中声明的类只可以有一个直接父类，不像C++中单个类可以同时继承来自于两个父类。

在java中，当一个类继承自其父类之后，可以在该子类内调用父类所有没有声明为private的方法和变量。当声明一个子类对象的时候，必须在其构造方法内部显示或隐式地调用其父类中的构造器。

情况1：父类和子类都没有显示写出其构造函数，则声明子类对象的时候是隐式地调用子类和父类默认的无参数的构造方法。

|  |
| --- |
| **class** Base{  }  **public** **class** Test **extends** Base{  **public** **static** **void** main(String[] args) {  Test t = **new** Test();  }  } |

情况2：父类和子类中都是无参构造器。这与情况1一样，无需在子类的构造器中显示调用子类的构造器。

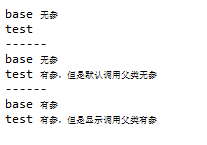
|  |
| --- |
| **class** Base{  **public** Base(){  System.*out*.println("base1");  }  }  **public** **class** Test **extends** Base{  **public** Test(){  System.*out*.println("test");  }  **public** **static** **void** main(String[] args) {  Test t = **new** Test();  }  } |

运行结果如下：

情况3：父类中有无参构造器也有有参的构造器，子类中有无参的，也有有参数的构造器。则默认调用父类的无参构造器，子类也可以显示调用父类的有参构造器。

|  |
| --- |
| **class Base{**  **public Base(){**  **System.*out*.println("base 无参");**  **}**  **public Base(int i){**  **System.*out*.println("base 有参");**  **}**  **}**  **public class Test extends Base{**  **public Test(int i){**  **System.*out*.println("test 有参，但是默认调用父类无参");**  **}**  **public Test(){**  **System.*out*.println("test");**  **}**  **public Test(int i,int j){**  **super(i);**  **System.*out*.println("test 有参，但是显示调用父类有参");**  **}**  **public static void main(String[] args) {**  **Test t = new Test();**  **System.*out*.println("------");**  **Test t1 = new Test(1);**  **System.*out*.println("------");**  **Test t2 = new Test(1,2);**  **}**  **}** |

运行结果如下：



情况4：父类中没有无参构造器，子类中的构造器则必须显示调用父类的构造器。如果子类构造器中没有显示调用父类的构造器，则会报错。

|  |
| --- |
| **class** Base{  **public** Base(**int** i){  System.*out*.println("base 有参");  }  }  **public** **class** Test **extends** Base{  /\*父类中没有无参的构造器，子类需要显示调用父类的有参构造器才可以\*/  **public** Test(**int** i){  **super**(i);//没有这一行，代码将报错  System.*out*.println("test 有参，但是默认调用父类无参");  }  **public** **static** **void** main(String[] args) {  Test t = **new** Test();  System.*out*.println("------");  Test t1 = **new** Test(1);  }  } |

总之，如果父类中有无参构造器（不管是默认还是显示），则子类构造器中可以显示调用父类中某个有参构造器或无参构造器，如果没有显示调用，则默认子类调用父类的无参构造器。如果父类中没有无参构造器，则在父类中必须显示调用父类的某个有参构造器。因为一旦父类中声明有构造器，则其默认的无参构造器则无效。

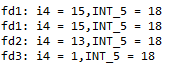
1. final 关键字
   1. final数据

在对final关键字进行定义的时候必须对其赋值初始化。对于基本类型，final是数值恒定不变；而对于对象引用，final是引用恒定不变。一旦引用被初始化为指向一个对象，就无法再把它改为指向另外一个对象，但是对象自身的值是可以修改的。（这一限制同样适用于数组，因为数组也是对象）

针对给成员变量加final关键字的示例如下，注意该示例摘自《java编程思想》

|  |
| --- |
| **class** Value{  **int** i;  **public** Value(**int** i){  **this**.i=i;  }  }  **public** **class** FinalData {  **private** **static** Random *rand* = **new** Random(47);  **private** String id;  **public** FinalData(String id){  **this**.id = id;  }  **private** **final** **int** valueOne = 9;  **private** **static** **final** **int** *VALUE\_TWO* = 99;  **public** **static** **final** **int** *VALUE\_THREE* = 39;  **private** **final** **int** i4 = *rand*.nextInt(20);//每声明一个对象不一定i4一样  **static** **final** **int** *INT\_5* = *rand*.nextInt(20);//类加载之后，这个值就固定了  **private** Value v1 = **new** Value(11);  **private** **final** Value v2 = **new** Value(22);  **private** **static** **final** Value *VAL\_3* = **new** Value(33);//静态final变量对类的所有对象都是一致的  **private** **final** **int**[] a ={1,2,3,4,5,6};  **public** String toString(){  **return** id+": "+"i4 = "+i4+",INT\_5 = "+*INT\_5*;  }  **public** **static** **void** main(String[] args){  FinalData fd1 = **new** FinalData("fd1");  //fd1.valueOne++;//valueOne是fianl成员变量，变量是基本类型，其值不可以被改变  //fd1.VALUE\_TWO++;//VALUE\_TWO是fianl成员变量，变量是基本类型，其值不可以被改变  fd1.v1 = **new** Value(9);//v2不是final对象引用，可以改变其指向的引用对象。  //fd1.v2 = new Value(10);//v2是final对象引用，不可以改变其指向的引用对象。  fd1.v2.i++;//v2是final对象引用，但是其值可以改变。  //fd1.a[i] = new int[3];//a是final对象引用，不可以改变其指向的引用对象。  **for**(**int** i=0;i<fd1.a.length;i++){//a是final对象引用，但是其值可以改变。  fd1.a[i]++;  }  System.*out*.println(fd1);  FinalData fd2 = **new** FinalData("fd2");  System.*out*.println(fd1);  System.*out*.println(fd2);  FinalData fd3 = **new** FinalData("fd3");  System.*out*.println(fd3);  }  } |

程序运行的结果为：



空白final

空白final表示被声明为final但又没有给定初值的域（但是我自己测试代码却不行？？？？？）。但是要注意的是在使用前，要确保这个变量已经被正确初始化。

final参数

Java允许在参数表中以声明的方式将参数指明为final，这就意味着在方法中更改参数引用所指向的对象。

* 1. final 方法

被声明为final的方法将不能被其子类重写，另外当调用一个声明为final的方法时，是直接讲方法主体嵌入到调用处，而不是进行方法调用。虽然从一定程度上可以提高程序的效率，但是如果final方法体过于庞大，程序代码就会膨胀，因而可能看不到内嵌带来的任何性能提高。

类中所有的private类都隐式地指定为是final的，由于无法取用private方法，因此也就无法重写它。就算子类中有某个方法与其父类中的private方法重名，但是因为这个方法在父类中是private的，那它就不是父类接口的一部分，因此子类就无法接触到这个方法。出现与父类中private方法重名的方法也就只是重名而已，并不是重写了父类中的private方法。

* 1. final类

当一个类被声明为final类的时候，这个类就不能被继承了，该类中的所有方法也不能被重写。但是不表示这个类中的所有成员不可以被改变，要想做到final类中的成员变量不可以被改变，必须给成员变量增加final修饰。需要主要的是：一个类不能同时被声明为abstract和final。

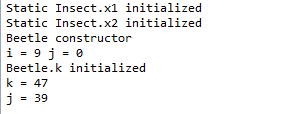
1. 初始化及类的加载

类的初始化过程与所继承的类有关系。了解包括继承在内的初始化全过程，以对所发生的一切有个全局性的把握，是很有益的。

代码样例来自于《java编程思想》。

|  |
| --- |
| public class Insect {  private int i =9;  protected int j;  Insect(){  System.*out*.println("i = "+i+" j = "+j);  j = 39;  }  private static int *x1* = *printInit*("Static Insect.x1 initialized");  static int printInit(String s){  System.*out*.println(s);  return 47;  }  }  public class Beetle extends Insect{  private int k = *printInit*("Beetle.k initialized");  public Beetle(){  System.*out*.println("k = "+k);  System.*out*.println("j = "+j);  }  private static int *x2* = *printInit*("Static Insect.x2 initialized");    public static void main(String[] args){  System.*out*.println("Beetle constructor");  Beetle b = new Beetle();  }  } |

运行结果如下所示：



分析：

在java中代码开始运行之前，首先进行加载过程，加载的时候首先加载它的基类，类加载的时候就会首先运行它包含的static部分，所以我们在代码中先看到了Static Insect.x1 initialized

Static Insect.x2 initialized

的输出，接着开始运行main函数得到了Beetle constructor的输出，顺着main函数运行到了新建得到Beetle对象b，调用这个类的构造函数，又由于在类的构造函数内部会相应的调用其父类中的构造函数，所以我们先看到了基类中i = 9 j = 0的输出。然后再得到Beetle的对象，首先初始化得到k的值，所以会看到Beetle.k initialized的输出，接着就输出了Beetle构造函数中的输出。k = 47 j = 39。

总之，java程序初始化工作可以在许多不同的代码块中来完成（例如静态代码块、构造函数等），它们执行的顺序如下：父类静态变量、父类静态代码块、子类静态变量、子类静态代码、父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码、子类构造函数。其中父类静态变量、父类静态代码块、子类静态变量、子类静态代码等四个静态部分只在类加载时执行一次。

首先如果有静态变量和静态代码则先初始化父类静态变量、父类静态代码块、子类静态变量、子类静态代码。然后如果有创建对象，则按照执行顺序:父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码、子类构造函数等依次进行初始化操作。

1. 初始化

Java尽力保证：所有变量在使用前都能得到恰当的初始化。

5.1 方法的局部变量

对于方法的局部变量，java以编译时错误的而形式来贯彻这种保证。所以，如果在某个方法内写成如下所示就会报出变量未初始化的错误。

|  |
| --- |
| **int** n;  n++;//会报错，n未初始化 |

5.2 类的成员变量

但是如果是类的成员变量是基本类型，该成员就会被自动得到一个初始化的值。如

|  |
| --- |
| **class** Book{  **int** i; //类的成员变量，没有初始化  **boolean** checkedOut = **false**;  Book(**boolean** checkOut){  i++;//使用时不出错，因为在定义的时候得到一个默认的初始化  checkedOut = checkOut;  }  **void** checkIn(){  i++;//使用时不出错，因为在定义的时候得到一个默认的初始化  checkedOut = **false**;  } |

5.3 初始化顺序

在类的内部，成员变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于方法定义之间，各个变量仍旧会在任何方法（包括构造器）被调用之前得到初始化。

初始化的顺序是先静态对象，然后再是“非静态”对象。静态初始化只在Class对象首次加载的时候进行一次。

5.4 调用构造器时的顺序

构造器的调用顺序是很重要的。

1）调用基类构造器：这个步骤会不断反复递归下去，首先是构造这种层次结构的根，然后是下一层导出类，等等，直到最底层的导出类；（即得到该基类的一个对象）。即依次得到基类的各个对象。初始化的时候分配给对象的存储空间初始化为二进制的0.

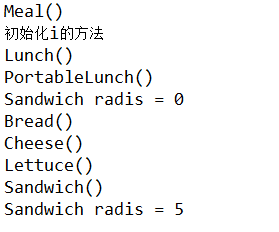
2）按声明顺序调用成员的初始化方法；

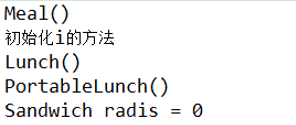
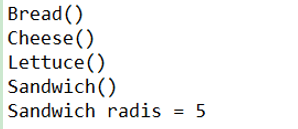
3）调用导出类构造器的主体。

具体例子如下所示：

|  |
| --- |
| **package** com.sse.zk.chapter8;  /\*\***@author** zk  \* **@date** 20160104  \* 验证构造器的调用顺序  \*/  **class** Meal{  Meal(){  System.*out*.println("Meal()");  }  }  **class** Bread{  Bread(){  System.*out*.println("Bread()");  }  }  **class** Cheese{  Cheese(){  System.*out*.println("Cheese()");  }  }  **class** Lettuce{  Lettuce(){  System.*out*.println("Lettuce()");  }  }  **class** Lunch **extends** Meal{  **private** **int** i = f();  **public** **int** f(){  System.*out*.println("初始化i的方法");  **return** 1;  }  Lunch(){  System.*out*.println("Lunch()");  }  }  **class** PortableLunch **extends** Lunch{  **public** **void** draw(){  }  PortableLunch(){  System.*out*.println("PortableLunch() ");  draw();  System.*out*.println();  }  }  **public** **class** Sandwich **extends** PortableLunch{  **private** Bread b = **new** Bread();  **private** Cheese c = **new** Cheese();  **private** Lettuce l = **new** Lettuce();  **private** **int** radis = 5;  **public** Sandwich(){  System.*out*.println("Sandwich() ");  draw();  System.*out*.println();  }  **public** **void** draw(){  System.*out*.print("Sandwich radis = "+radis);  }  **public** **static** **void** main(String[] args){  Sandwich s = **new** Sandwich();  }  } |

代码的运行结果为



由代码可知是在main方法中声明Sandwich对象s，那么在执行的时候会依次封装基类对象，封装的顺序根据继承的先后顺序为主，则依次得到每个基类对象，如，最后生成Sandwich对象，根据5.3介绍得初始化顺序依次初始化该对象内的各个变量的值。

初始化的时候分配给对象的存储空间初始化为二进制的0，所以当PortableLunch类中的draw方法被Sandwich覆盖后，在PortableLunch类构造器中调用draw方法输出radis的值为0.

总之，java程序初始化工作可以在许多不同的代码块中来完成（例如静态代码块、构造函数等），它们执行的顺序如下：父类静态变量、父类静态代码块、子类静态变量、子类静态代码、父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码、子类构造函数。其中父类静态变量、父类静态代码块、子类静态变量、子类静态代码等四个静态部分只在类加载时执行一次。

首先如果有静态变量和静态代码则先初始化父类静态变量、父类静态代码块、子类静态变量、子类静态代码。然后如果有创建对象，则按照执行顺序父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码、子类构造函数等依次进行初始化操作。

1. 多态(重载和重写)

多态见于有继承层次的类结构中。由以下代码所示，可知

1. public权限的方法有多态性，静态方法和成员变量没有多态性

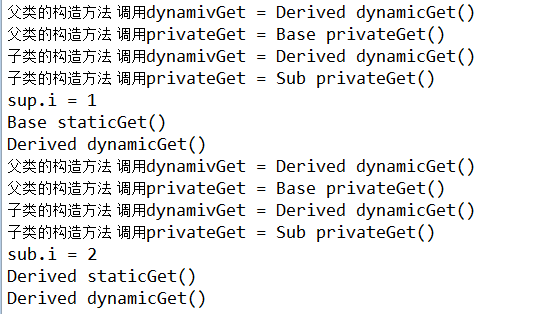
1）将子类对象可以赋给父类引用，这样调用对象的方法时会动态绑定具体的对象，父类中所有和子类中同名的public方法会被子类覆盖，当调用此类时调用的即为子类中的方法，如调用dynamivGet()方法时实际调用的是子类中的方法。但是像privateGet这样的private方法则不会被子类覆盖，在父类中的构造函数中调用privateGet方法即为父类中定义的方法。

2）静态方法不具有多态性，因为静态方法是和类绑定的，不是和对象绑定的（静态方法输出的内容取父类还是子类并不取决于创建对象的类型，而是取决于所定义变量的类型）。只有类中的方法才有多态的概念，类中成员没有多态的概念。

3）成员变量是没有办法实现多态的，成员变量的值取父类还是子类并不取决于创建对象的类型，而是取决于所定义变量的类型，这是在编译期确定的。如System.*out*.println("sup.i = "+sup.i);和System.*out*.println("sub.i = "+sub.i);两行的输出所示。

|  |
| --- |
| package java20160111;  /\*\*  \* 验证子类中对父类中同名的成员变量、静态方法和普通方法的覆盖问题。  \*/  class Super {  public int i=1;  public Super(){  System.*out*.println("父类的构造方法 调用dynamivGet = "+dynamivGet());  System.*out*.println("父类的构造方法 调用privateGet = "+privateGet());  }  public static String staticGet(){  return "Base staticGet()";  }  public String dynamivGet(){  return "Base dynamicGet()";  }  private String privateGet(){  return "Base privateGet()";  }  }  class Sub extends Super{  public int i=2;  public Sub(){  System.*out*.println("子类的构造方法 调用dynamivGet = "+dynamivGet());  System.*out*.println("子类的构造方法 调用privateGet = "+privateGet());  }    public static String staticGet(){  return "Derived staticGet()";  }  public String dynamivGet(){  return "Derived dynamicGet()";  }  private String privateGet(){  return "Sub privateGet()";  }  }  public class Polymorphism{  public static void main(String[] args){  Super sup = new Sub();//将子类对方赋给父类引用，定义为父类的类型  System.*out*.println("sup.i = "+sup.i); //变量与定义时的类型有关  System.*out*.println(sup.*staticGet*());//静态方法与定义时的类型有关  System.*out*.println(sup.dynamivGet());//普通方法与创建对象时的类型有关    Sub sub = new Sub();//定义为子类的类型  System.*out*.println("sub.i = "+sub.i);  System.*out*.println(sub.*staticGet*());  System.*out*.println(sub.dynamivGet());  }  } |

运行结果是：



2.多态的实现机制

多态的实现机制基于方法的重载和覆盖，如果父类中没有dynamivGet()方法，则sup.dynamivGet()调用这个方法将会出错。因为子类中没有对父类中的这个方法进行重载或覆盖。当子类中的方法与父类中的方法有相同的签名（相同的方法名，相同的参数个数与类型）时，子类将会覆盖父类的方法，而不会继承。

Java中提供了编译时和多态运行时多态。编译时多态是通过方法的重载实现的，运行时多态是通过方法的覆盖（子类覆盖父类的方法）实现的。

3. 重载（overload）和覆盖（override）

重载（overload）和覆盖（override），其中重载是在同一个类中多态性的表现，是指在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型（特别注意的是子类继承父类的public方法之后，该public方法作为子类中的方法之一，子类中再有其他与该public方法同名但参数不同的方法即可算作重载）。覆盖是指派生类方法覆盖基类方法，覆盖一个方法对其重写，以达到不同的作用。

3.1 使用重载时需要额外注意四点

1. 重载是通过不同的方法参数来区分的。说白了就是方法签名中方法名字相同，参数列表不同。（方法名和参数列表合称方法签名，唯一地标识出某个方法）；
2. 不能通过方法的访问权限、返回值类型和抛出的异常类型来进行重载；
3. 对于继承来说，如果基类方法的访问权限是private的，那么就不能在派生类中对其重载，如果派生类中也定义了一个同名的方法，这在子类中是一个新的方法，不会达到重载的效果；如果基类方法的访问权限是public的，在派生类中如果出现了方法名同但是参数不同的则算是重载；
4. 参数顺序的不同也可以区分两个方法，算作重载，但是一般不那么做。

3.2 在使用覆盖(重写)时需要注意三点

1. 派生类中的覆盖方法必须和基类中被覆盖的方法有相同的返回值、方法名和参数列表。注意，返回值可以是基类中返回类型的子类，这是java中SE5之后的协变返回类型。
2. 派生类中的覆盖方法必须和基类中被覆盖的方法所抛出的异常一致(但是可以抛出更少的异常或不抛出异常(编写高质量代码改善Java程序的151个建议p7，这是因为要满足里氏替换原则）))。
3. 基类中被覆盖的方法不能为private，否则其子类只是定义了一个新的方法，并没有对覆盖基类中的方法。

3.3 重载与覆盖的区别主要有以下几个方面：

1. 覆盖是子类与父类之间的关系，是垂直关系；而重载是同一个类中的多个方法之间的关系；
2. 覆盖是子类和父类中相同返回值、方法名和参数列表的方法之间的对应关系；而重载是具有相同方法名的多个方法之间的关系；
3. 覆盖要求参数列表相同；而覆盖要求参数列表不同；
4. 覆盖关系中，调用方法体是根据对象的类型（对象对应存储空间类型）来决定，而重载关系是根据调用时的实参数列表与形参列表来选择方法体的。
5. 接口与抽象类

7.1 抽象类和抽象方法

abstract关键字允许在类中创建一个或多个没有任何定义的方法即只声明了方法却没有提供方法体如abstract void f()，这些实现是由此类的继承者创建的。

包含抽象方法的类叫做抽象类，如果一个类包含一个或者多个抽象方法，该类必须被限定为抽象的。我们也可以定义一个没有任何抽象方法的抽象类。某个类称为抽象类，不需要所有的方法都是抽象的。

切记：不允许创建抽象类的对象。

7.2 接口

在接口中的每一个方法都只是一个声明，这是编译器所允许的在接口中唯一能够存在的事务，此外，在接口中没有任何方法被声明是public的，但是它们自动都是public abstract的，而且接口中的方法只能被这两个关键字修饰，在使用的时候加不加关键字都自动是public abstract的。接口也可以包含域，但是这些域隐式地是public 、static和final的。

接口中定义的域不可以是“空final”的，但是可以被非常量表达式初始化。如

|  |
| --- |
| Public interface test{  Random rand = new Random(47);  int RANDOM\_INT = rand.nextInt(10);  } |

使用接口的原因:①为了能够向上转型为多个基类型，以及由此带来的灵活性；

②防止客户端程序员创建该类的对象，并确保这仅仅是建立一个接口。

7.3 为什么java中有些接口没有任何方法

在java语言中，有些接口内部没有声明任何方法，也就是说，实现这些接口的类不需要重写任何方法，这些没有任何方法声明的接口又被叫做标识接口。标识接口对实现它的类没有任何语义上的要求，它仅仅充当一个标识的作用，用来表明实现它的类属于一个特定的类型。

7.4 抽象类与接口的相同点

1） 都不能被实例化；

2） 接口的实现类或抽象类的子类都只有实现了接口或抽象类中的方法后才可以被实例化。

7.4 抽象类与接口的不同点

1） 接口可以继承接口，抽象类既可以实现接口也可以继承具体类；

2） 接口中的成员变量是public static final，所以必须给其赋初值，所有的成员方法都是public 、abstract的，而且只能被这两个关键字修饰。而抽象类可以有自己的数据成员变量，也可以有抽象的成员方法，抽象类中的成员变量和普通类的成员变量一样的四种修饰符，抽象类的抽象方法由于要被继承的子类实现，因此不能被private、static、synchronized、native等访问修饰符修饰，同时必须以分号结尾，并且不能带有花括号。所以，当功能需要被积累时用抽象类；不需要积累时用接口。

3）接口中的方法只有定义不能实现，而抽象类中的方法可以有定义与实现；

4）接口不可以有静态的public static void main(String[] args){}方法，抽象类可以有。

5）接口是被实现，而抽象类只能被继承。一个类可以实现多个接口，但是只可以继承一个抽象类，因此使用接口可以间接地达到多重继承的目的；

6）接口强调的特定功能的实现，其设计理念是“has-a”的关系，而抽象类强调所属关系，是“is-a”的关系；

7）接口被用于实现比较常用的功能，便于日后维护或者添加删除方法；而抽象类更倾向于充当公共类的角色，不适用于日后重新对里面的代码进行修改。

总之，接口是一种特殊形式抽象类，使用接口完全有可能实现与抽象类相同的操作，但一般而言，抽象类多用于在同类事物中有无法具体描述的方法的场景，所以当子类和父类之间存在逻辑上的层次结构时，推荐使用抽象类；而接口多用于不同类之间，定义不同类之间的通信规则，所以当希望支持差别较大的两个或者更多对象之间的特定交互行为时，应该使用接口。

1. 内部类

内部类有静态内部类(嵌套内部类)、普通内部类、局部内部类、匿名内部类。

8.1嵌套内部类

内部类是static时，称作是嵌套类。则1）创建嵌套类的对象时不再需要其外围类的对象；2）不能从嵌套类的对象中访问非静态的外围类对象。

静态内部类不能访问外部类的普通成员变量，只能访问外部类中的静态成员和静态方法。

8.2普通内部类（成员内部类）

内部类与普通类的区别：1）内部类的名字是嵌套在外部类中的；2）外部类将有一个方法，该方法返回一个内部类的引用。

如果想从外部类的非静态方法之外的任意位置创建某个内部类的对象，那么必须以OuterClassName.InnerClassName的形式获取内部类对象的引用。构建内部类引用的时候需要一个指向其外围类对象的引用，如果编译器访问不到这个引用就会报错。普通内部类的字段和方法只能放在类的外部层次上，所以普通的内部类不能有static数据和static字段，也不应该包含嵌套类（静态内部类）。

如下代码所示:

|  |
| --- |
| /\*\*  \* 编写一个名为Outer的类，它包含一个名为Inner的类。在Outer类中添加一个方法，  \* 它返回一个Inner类型的对象，在main()中，创建并初始化一个指向某个Inner对象的  \* 引用。  \*/  **public** **class** Excrise1 {  /\*定义一个内部类\*/  **class** Inner{  Inner(){  System.*out*.println("得到一个内部类的引用");  }  }  /\*外部类中定义一个返回Inner类型对象的方法\*/  **public** Inner getInner(){  **return** **new** Inner();  }  **public** **static** **void** main(String[] args){  Excrise1 e = **new** Excrise1();  /\*在main()中，创建并初始化一个指向某个Inner对象的引用\*/  Excrise1.Inner i = e.getInner();//方法一：通过调用外部类返回内部类引用的方法  Excrise1.Inner i2 = e.**new** Inner();//方法二：利用外部对象引用新建一个内部类对象引用  }  } |

8.3 局部内部类

在方法或者代码的作用域内创建一个完整的类，称作局部内部类，其作用范围为所在的代码块，是内部类中最少使用的一种类型。局部内部类和局部变量一样，不能被public、protected、private和static修饰，只能访问方法中定义为final类型的局部变量。分为静态局部内部类（位于静态方法内或者静态代码块内）和非静态局部内部类。

8.4 匿名内部类

匿名内部类与正规的继承相比有些受限，因为匿名内部类既可以扩展类，也可以实现接口，但是不能两者兼备。而且如果是实现接口，也只能实现一个接口。匿名内部类没有类名的内部类，不使用关键字class、extends、implements，没有构造函数，必须继承其他类或实现其他接口。匿名内部类的好处是代码更加简洁紧凑，但坏处是易读性下降。一般应用于GUI编程中实现事件处理等。匿名内部类只能访问外部类的final类型的变量。

各个内部类可访问的变量类型总结如下表：

|  |  |
| --- | --- |
| 内部类的类型 | 可访问的外部变量 |
| 静态内部类 | Static |
| 普通内部类 | Static、普通、final |
| 局部内部类 | Final |
| 匿名内部类 | Final |

使用内部类的好处：

① 每个内部类都能独立的继承一个接口的实现，所以无论外部类是否已经继承了某个(接口的)实现，对于内部类都没有影响。内部类使得多继承的解决方案变得完整，

　　② 方便将存在一定逻辑关系的类组织在一起，又可以对外界隐藏。

　　③ 方便编写事件驱动程序

　　④ 方便编写线程代码

1. java中的clone方法

Java中的所有类默认都继承自Object类，而Object类中提供了一个clone()方法，这个方法的作用是返回一个Object对象的复制。这个复制函数返回的是一个对象而不是一个引用。以下是使用clone()方法的步骤:

1. 实现clone的类首先需要继承Cloneable接口。Cloneable接口实质是一个标识接口（没有任何接口方法）。
2. 在类中重写Object类中的clone方法。
3. 在clone方法中调用super.clone()。无论clone类的继承结构是什么，super.clone都会直接或间接地调用java.lang.Object类的clone()方法。
4. 把浅复制的引用指向原型对象新的克隆体。

如下代码所示：

|  |
| --- |
| **public** **class** Obj **implements** Cloneable{  **private** **int** i = 0;  **public** **int** getI(){  **return** i;  }  **public** **void** setI(**int** i2){  i2 = i;  }  **public** **void** changeI(){  **this**.i = 1;  }    **private** Date birth = **new** Date();  **public** Date getDate(){  **return** birth;  }  **public** **void** setDate(){  **this**.birth = birth;  }  **public** **void** changeDate(){  **this**.birth.setMonth(4);  }    **public** Object clone(){  Obj o = **null** ;  **try** {  o = (Obj)**super**.clone();  } **catch** (CloneNotSupportedException e) {  e.printStackTrace();  }  /\*实现深复制\*/  o.birth = (Date)**this**.getDate().clone();//必须保证Date类实现Cloneable接口，覆盖object的clone方法，如果把这行代码注释掉，则是浅复制，Obj对象的复制对象内Date和被复制的对象属于同一个引用  **return** o;  }    **public** **static** **void** main(String[] args){  Obj a = **new** Obj();    Obj b = (Obj) a.clone();//返回的是和a一样的对象，对b的改变不影响a中元素的值  b.changeI();  System.*out*.println("a : = "+a.getI());  System.*out*.println("b : = "+b.getI());  b.changeDate();  System.*out*.println("a : = "+a.getDate());  System.*out*.println("b : = "+b.getDate());  Obj c = a;//返回的是和a一样的引用，对c的改变会影响a中元素的值  c.changeI();  System.*out*.println("a : = "+a.getI());  System.*out*.println("c : = "+c.getI());  c.changeDate();  System.*out*.println("a : = "+a.getDate());  System.*out*.println("c : = "+c.getDate());  }  } |

引申：浅复制和深复制有什么区别？

浅复制（Shallow Clone）：被复制对象的所有变量都含有与原来对象相同的值，而所有对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不是复制它所引用的对象。

深复制(Deep Clone)：被复制对象的所有变量都含有与原来那些对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制的新对象，而不再是原有的那些被引用的对象。换言之，深复制把复制的对象所引用的对象的复制了一遍。

1. package的作用

Package主要有以下两个作用：1）提供多层命名空间，解决命名冲突，通过使用package，使得处于不同package中的类可以存在相同的名字；2）对类按功能进行分类，使项目的组织更加清晰。

1. java中的BIO、NIO、AIO

11.1 相关概念

|  |  |
| --- | --- |
| 同步IO | 用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪。比如自己亲自去银行取钱。 |
| 异步IO | 用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知（异步的特点就是通知）。比如告诉朋友银行卡密码，让他拿着银行卡去帮自己取钱。然后自己可以去干别的事。（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS） |
| 阻塞IO | 当试图对该文件描述符进行读写时, 如果当时没有东西可读,或者暂时不可写, 程序就进入等待状态, 直到有东西可读或者可写为止。比如去银行取钱，发现没有开门就会一直在那等着直到银行开门。 |
| 非阻塞IO | 非阻塞状态下, 如果没有东西可读, 或者不可写, 读写函数马上返回, 而不会等待。比如柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去。 |

1．同步和异步是针对应用程序和内核的交互而言的。

2．阻塞和非阻塞是针对于进程在访问数据的时候，根据IO操作的就绪状态来采取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

一个完整的IO读请求操作包括两个阶段：1)查看数据是否就绪；2)进行数据拷贝（内核将数据拷贝到用户线程。

那么阻塞（blocking IO）和非阻塞（non-blocking IO）的区别就在于第一个阶段，如果数据没有就绪，在查看数据是否就绪的过程中是一直等待，还是直接返回一个标志信息。阻塞过程会让出CPU，非阻塞过程不会让出CPU。

对于同步IO：当用户发出IO请求操作之后，如果数据没有就绪，需要通过用户线程或者内核不断地去轮询数据是否就绪，当数据就绪时，再将数据从内核拷贝到用户线程；

而异步IO：只有IO请求操作的发出是由用户线程来进行的，IO操作的两个阶段都是由内核自动完成，然后发送通知告知用户线程IO操作已经完成。也就是说在异步IO中，不会对用户线程产生任何阻塞。

这是同步IO和异步IO关键区别所在，同步IO和异步IO的关键区别反映在数据拷贝阶段是由用户线程完成还是内核完成。所以说异步IO必须要有操作系统的底层支持。

由上描述基本可以总结一句简短的话，同步和异步是目的，阻塞和非阻塞是实现方式。

Java BIO、NIO、AIO对比表。

|  |  |  |
| --- | --- | --- |
| 类型 | 含义 | 使用场景 |
| 同步阻塞IO（JAVA BIO） | 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。 | BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。 |
| 同步非阻塞IO(Java NIO) | 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。用户进程也需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问 | NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。 |
| 异步非阻塞IO (Java AIO(NIO.2) | 一个有效请求一个线程，客户端的I/O请求都是由操作系统先完成了再通知服务器应用去启动线程进行处理。线程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。 | AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。 |

1. 异常

12.1 概念

使用异常的一个明显好处就是，它能够降低错误处理代码的复杂度。与之前的错误处理方法相比，异常机制使得代码的阅读、编写和调试工作更加井井有条。

异常最重要的方面之一就是如果发生问题，它们将不允许程序沿着其正常的路径继续走下去。异常允许强制程序停止运行，并告诉我们出现了什么问题，或者强制程序处理问题，并返回到稳定状态。

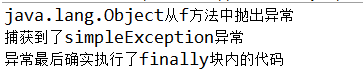
12.2 自定义异常

如果要自己自定义异常类，必须从已有的异常类继承，最好是选择意思相近的异常类继承（不过这样的异常并不容易找）。通过新建立的异常类型最简单的办法就是让编译器为产生默认的构造器。对异常来说，最重要的部分就是类名，所以下面代码中的示例在大多数情况下已经够用了。

如下代码验证了如何用try catch捕获异常，验证了如何使用自定义的异常类，验证了执行了catch语句之后会执行finally代码块中的代码。

|  |
| --- |
| /\*继承自Exception的自定义异常\*/  **class** simpleException **extends** Exception{    }  **public** **class** InherException {  **public** **void** f() **throws** simpleException{  System.*out*.println("从f方法中抛出异常");  **throw** **new** simpleException();  }  **public** **static** **void** main(String[] args){  InherException test = **new** InherException();  **try** {  test.f();  } **catch** (simpleException e) {  System.*out*.println("捕获到了simpleException异常");  } **finally**{  System.*out*.println("异常最后确实执行了finally块内的代码");  }  }  } |

运行结果如下：



12.3 异常说明

类似于public void f() throws simpleException{}这种在定义方法的时候在其名字后面加上可能抛出的异常类型throws simpleException。称作是异常说明。

12.4 永远不需要为清理异常对象而担心，因为它们都是用new在堆上创建的对象，所以垃圾回收器会自动把它们清理掉。

12.5 重新抛出异常

假如当前异常对象是Exception e，如果只是把当前异常对象e重新抛出 ，那么printStackTrace()方法显示的将是原来异常抛出的调用栈信息，而非重新抛出点的信息。想要更新这个信息，可以调用fillStackTrace()方法，抛出的是(Exception)e. fillStackTrace()这将返回一个Throwable对象，它是通过把当前调用栈信息填入原来那个异常对象而建立的。(具体内容参见P258《Thinking in java》)

12.6 异常链

在捕获一个异常后抛出另一个异常，并且希望把原始异常的信息保存下来，这被称为异常链。

在Throwable的子类中只有三种基本的异常类提供了带case参数的构造器，它们是Error，Exception和RuntimeException。如果要把其他类型的异常链接起来，应该使用initCase()方法而不是构造器。

12.7 java标准异常

Throwable是被用来表示任何可以作为异常被抛出的类。Throwable对象可分为两种类型（指从Throwable继承而得到的类型）：Error用来表示编译时和系统错误（一般不需要我们去关心这个错误，这种错误是属于JVM层次的严重错误，因此这种错误是会导致程序终止运行的）；Exception是可以被抛出的基本类型，在Java类库、用户方法以及运行时故障中都可能抛出Exception异常。所以java程序员最关心的基本类型通常是Exception。

Error为错误，是程序无法处理的，如outofmemory、ThreadDeath等，出现这种情况唯一可以做的就是听之任之，交由JVM去处理，不过JVM在大多处情况下会选择终止线程。

Exception表示可恢复的异常，是编译器可以捕获到的，它包含两种类型：检查异常(checked exception)和运行时异常(runtime exception).

1. 检查异常-------发生在编译阶段

是在程序汇总最经常碰到的异常。所有继承自Exception并且不是运行时异常的都是检查异常，比如最常见的IO异常和SQL异常。这种异常都是在编译时期，Java编译器强制程序去捕获此类的异常，即把这些可能出现异常的代码放到try块中，把对异常处理的代码放到catch块中。应用的情况有①异常的发生并不会导致程序出错，进行处理后可以继续执行后续的操作；②程序依赖于不可靠的外部条件，例如系统IO。

1. 运行时异常(非检查异常)-------发生在运行阶段

RuntimeException及其子类是一个特例，产生此类异常不需要在方法声明的时候进行异常说明也不需要try catch处理（编译器并没有强制要求这么做），它们会被Java虚拟机抛出，这种异常属于错误，将被自动捕获。不需要异常处理RuntimeException及其子类异常的原因是：1）这类错误无法预料；2）这是应该在代码中进行检查的错误。最常见的运行时异常有NullPointerException(空指针异常)、ClassCastException(类型转换异常)、ArrayIndexOutOfBoundsException(数组越界异常)、ArrayStoreException(数组存储异常)、BufferOverflowException(缓冲区溢出异常)、ArithmeticException(算数异常)。

12.8 finally

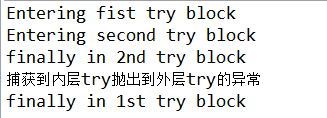
Finally子句中的代码无论try块中的异常是否抛出，都可以得到执行(除非代码没有进try或者遇到了exit)。这通常适用于内存回收之外的情况。Finally非常重要，使得程序员保证，无论try块中发生了什么，内存总能得到释放。但java有垃圾回收机制，所以内存释放不再是问题。当要把除内存之外的资源恢复到它们的初始状态时，就要用到finally子句。这种需要清理的资源包括：已经打开的文件或网络连接，在屏幕上画的图形，甚至可以是外部世界的某个开关。

总结：不管是在try执行过程中遇到return、break还是continue，代码都是会执行该try对应的finally语句。

1. 在异常没有被当前的异常处理程序捕获的情况下，异常处理机制也会在跳到更高一层的异常处理程序之前，执行finally子句。

|  |
| --- |
| **class** FourException **extends** Exception {  }  **public** **class** AlwaysFinally {  **public** **static** **void** main(String[] args) {  System.*out*.println("Entering fist try block");  **try** {  System.*out*.println("Entering second try block");  **try** {  **throw** **new** FourException();  } **finally** {/\* 内层抛出的异常在这没有被catch到 ，也要执行内层的finall块 \*/  System.*out*.println("finally in 2nd try block");  }  } **catch** (FourException e) {  System.*out*.println("捕获到内层try抛出到外层try的异常");  } **finally** {  System.*out*.println("finally in 1st try block");  }  }  } |

运行结果如下：

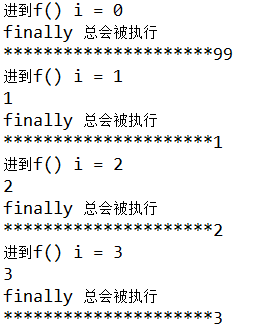


1. 在return中使用finally

Return在何处不重要，总会执行finally内的代码。

|  |
| --- |
| **public class ReturnFinally {**  **public static int f(int i){**  **System.*out*.println("进到f() i = "+i);**  **try{**  **switch(i){**  **case 4 :System.*out*.println(i);return i;//**  **case 3 :System.*out*.println(i);return i;**  **case 2 :System.*out*.println(i);return i;**  **case 1 :System.*out*.println(i);return i;**  **}**  **}finally{**  **i = 100;**  **System.*out*.println("finally 总会被执行");**  **}**  **return 99;**  **}**  **public static void main(String[] args) {**  **for(int i=0;i<4;i++){**  **int n = *f*(i);**  **System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"+n);**  **}**  **}**  **}** |

运行结果如下：



在try块中执行到return语句时会把当前的返回值保存起来，当去执行finally块中的代码就算覆盖了i的值，但是返回的结果就是在该返回的那个地方的i值。

（1）对于基本数据类型是这样，return的就是真正在try中return的值；

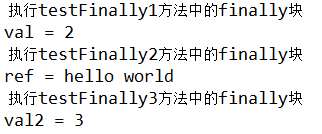
（2）如果返回的是引用类型，则其对象就会有改变（引用不变，但其内容会改变）。

（3）testFinally3（）方法验证了当在finally块中有return语句时，将会覆盖方法中其他地方的返回值。

如下代码所示。

|  |
| --- |
| **public** **class** TestReturnFinally {  **public** **static** **int** testFinally1(){  **int** result = 1;  **try**{  result = 2;  **return** result;  }**catch**(Exception e){  **return** 0;  }**finally**{  result = 3;  System.*out*.println(" 执行testFinally1方法中的finally块");  }    }    **public** **static** StringBuffer testFinally2(){  StringBuffer s = **new** StringBuffer("hello ");  **try**{  **return** s;  }**catch**(Exception e){  **return** **null**;  }**finally**{  s.append("world");  System.*out*.println(" 执行testFinally2方法中的finally块");  }  }    **public** **static** **int** testFinally3(){  **int** result = 1;  **try**{  result = 2;  **return** result;  }**catch**(Exception e){  **return** 0;  }**finally**{/\*当在finally块中有return语句时，将会覆盖方法中其他地方的返回值\*/  System.*out*.println(" 执行testFinally3方法中的finally块");  **return** 3;  }    }    **public** **static** **void** main(String[] args) {  **int** val = *testFinally1*();  System.*out*.println("val = "+val);  StringBuffer ref = *testFinally2*();  System.*out*.println("ref = "+ref);  /\*当在finally块中有return语句时，将会覆盖方法中其他地方的返回值\*/  **int** val2 = *testFinally3*();  System.*out*.println("val2 = "+val2);  }  } |

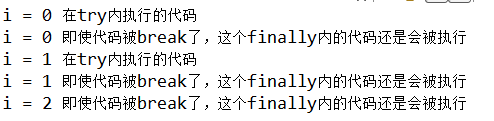
运行结果如下：



1. 即使循环中代码被break，还是会执行本次循环内后面代码中的finally部分。如下代码在i=2时跳出循环但是还是会执行本次循环中finally块中代码。

|  |
| --- |
| public class BreakFinally {  public static void main(String[] args) {  for(int i=0;i<4;i++){  try{  if(i == 2){  break;  }  System.*out*.println("i = "+i+" 在try内执行的代码");  }finally{  System.*out*.println("i = " +i+" 即使代码被break了，这个finally内的代码还是会被执行");  }  }  }  } |

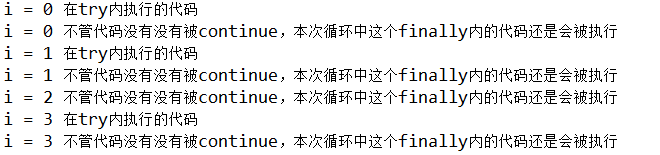
运行结果如下：



1. 即使循环中代码被continue，还是会执行本次循环内后面代码中的finally部分。如下代码在i=2时执行了continue语句，但是还是会执行本次循环中finally块中代码。

|  |
| --- |
| **public** **class** ContinueFinally {  **public** **static** **void** main(String[] args) {  **for**(**int** i=0;i<4;i++){  **try**{  **if**(i == 2){  **continue**;  }  System.*out*.println("i = "+i+" 在try内执行的代码");  }**finally**{  System.*out*.println("i = " +i+" 不管代码有没有没有被continue，本次循环中这个finally内的代码还是会被执行");  }  }  }  } |

运行结果如下：



12.9 异常匹配

抛出异常的时候，异常处理系统会按照代码的书写顺序找出“最近”的处理程序。找到匹配的处理程序之后，它就认为异常竟得到处理，然后就不再继续查找。查找的时候并不要求抛出的异常同处理程序所声明的异常完全匹配。派生类的对象也可以匹配其基类的处理程序。这一点非常有用，因为如果决定在方法里加上更多派生异常的话，只要客户程序员捕获的是基类异常，那么它们的代码就无需更改。

12.10使用异常处理的原因、原则和目标、优点

异常处理的原因：如果为每个方法所有可能发生的错误都进行处理的话，任务就会显得过于繁重，程序员也不一定会愿意这么做，这样的话就会导致错误被忽略。这就导致开发异常处理系统，其初衷是为了更好地方便程序员处理错误。

异常处理的一个重要原则是“只有在你知道如何处理的情况下才捕获异常”。

异常处理的重要目标就是把错误处理代码同错误发生的地点相分离。这样可以使得代码不会与错误处理逻辑混淆在一起，也更容易理解和维护。

异常处理的优点之一就是把错误代码和错误处理代码分开；另外一个优点就是所有的错误都以异常的形式进行报告，这也是java由于其他诸如C++这类语言的长处之一。

12.11引申：出现在java程序中的finally块是一定会被执行的吗？

答：不一定会被执行。1）当程序在进入try块之前就出现异常的时候，会直接结束，而不去会执行finally块中的代码。2）当try块中强制退出（exit（0））时，也不会去执行finally块中的代码。

1. 泛型

泛型，就是变量类型的参数化。使用泛型的典型例子，是在集合中的泛型使用，这也是设计泛型的主要目的之一。使用的时候有泛型类和泛型方法。

如泛型方法：**public** <S> **void** f(S c)

泛型类：**public** **class** Test<T> {}

泛型程序设计意味着编写的代码可以被很多不同类型的对象所重用。

泛型使用的典型例子就是集合。在使用泛型前，存入集合中的元素可以是任何类型的，当从集合中取出时，所有的元素都是Object类型，需要进行向下的强制类型转换，转换到特定的类型。泛型的思想就是**由程序员指定类型，这样集合就只能容纳该类型的元素。**

注意泛型类型参数只能被类或接口类型赋值，不能被原生数据类型赋值，原生数据类型需要使用对应的包装类。

**泛型的好处**

**1．类型安全。** 泛型的主要目标是提高 Java 程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在一个高得多的程度上验证类型假设。没有泛型，这些假设就只存在于程序员的头脑中（或者如果幸运的话，还存在于代码注释中）。

**2．消除强制类型转换。** 泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。

**3．潜在的性能收益。** 泛型为较大的优化带来可能。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，程序员会指定这些强制类型转换）插入生成的字节码中。但是更多类型信息可用于编译器这一事实，为未来版本的 JVM 的优化带来可能。由于泛型的实现方式，支持泛型（几乎）不需要 JVM 或类文件更改。所有工作都在编译器中完成，编译器生成类似于没有泛型（和强制类型转换）时所写的代码，只是更能确保类型安全而已。

Java泛型是使用擦除来实现的。所谓的擦除，仅仅是把code属性中的字节码进行擦除，实际上元数据还是保留了泛型信息，这也是能够通过反射手段取得参数化类型的根本依据。

1. 持有对象:容器

## 各种collection容器整体对比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 分类 | 描述 | 细化分类 | 描述 | 优点 | 缺点 |
| List | 以对象插入时顺序保存 | ArrayList | 基于数组实现。并未对方法做同步 | 可随机访问 | 在中间插入和删除元素较慢。 |
| Vector | 基于数组实现。有对方法做同步 | 支持多线程 |  |
| LinkedList | 基于双向循环链表实现 | 插入和删除元素方便 | 不可随机访问，只可顺序访问 |
| Set | 元素不可重复 | HashSet | 基于hash快速插入，元素间无序 | 可快速访问到元素 |  |
| TreeSet | 按照比较结果的指定顺序保存对象 | 对象有序，基于的是红黑树算法实现 |  |
| LinkedHashSet | 按照被添加的顺序保存对象 |  |  |
| Map | <key,value>形式保存键值对 | HashMap | 底层数据结构使用的是数组。将key做hash算法，然后以<key,value>存储 |  |  |
| TreeMap | 按照比较结果的指定顺序保存键 | 键值间有序 |  |
| LinkedHashMap | 按照插入顺序保存键，同时还保存HashMap的查询速度 |  |  |
| Queue | 一端插入另外一端删除 | | | | |

Java容器类库中的两种主要类型，区别在于每个“槽”保存的元素个数。Collection每个槽中只能保存一个元素，此类容器包括：List、Set、Queue；Map在每个槽内保存了两个对象，即<键,值>。

常用的List对应的实现类有ArrayList、Vector和LinkedList，其中ArrayList和Vector是基于数组实现，Vector是线程安全的，ArrayList非线程安全，LinkedList基于双向循环链表实现的。

常用的Map的实现类有HashMap、HashTable、TreeMap和LinkedHashMap。HashTable是线程安全的，HashMap基于hash表实现是非线程安全的，TreeMap是基于红黑树的数据结构来实现的，内部元素是按需排序的，LinkedHashMap是采用列表维护内部的顺序。

常用的Set对应的实现类有HashSet、TreeSet、LinkedHashSet，Set的内部实现都是基于Map的实现。

注意：List、Set、Map的使用场景是什么？

至于在什么时候使用，这个是要因地制宜的。

List：在能直接使用数组的时候，就有使用列表，如一个班的学生的成绩，成绩是可以重复的；

Set：一般用于存放无序的（指顺序不重要）不能重复的对象，如一个班的学生的学号，学号是不能重复的；

Map：用于存放具有对应关系的键值对，如一个班的学生的学号与姓名的映射，每个学号对应了一个学生的姓名，学号不能重复，但是姓名可能会重复。

## 14.1 List

最常见的List有三种：ArrayList、Vector、LinkedList。这三种List均来自Abstract的实现，Abstract直接实现了List接口，并扩展自AbstractCollection。

ArrayList和Vector几乎使用了相同的算法，唯一的区别是对多线程的支持。ArrayList没有对任何一个方法做线程同步，因此不是线程安全的。Vector对绝大部分方法都做了线程同步。

### ArrayList和LinkedList的区别

ArrayList是基于数组实现的，LinkedList是基于双向循环链表实现的。以增加和删除元素为例进行介绍它们之间的区别。

① 在列表尾端增加元素

ArrayList只需要把新加的元素追加到链表尾端即可，只有对容量的需求超过当前数组大小时才会进行扩容，但是复制的时候是调用System.arraycopy()方法，因此对ArrayList来说add()的操作效率还是蛮高的。

对LinkedList来说，是使用了链表的结构，因此不需要维护容量大小，但是每次增加元素都需要新建一个Entry对象，并进行更多的赋值操作。在繁多的系统调用中，对性能产生一定的影响。

总体来说，linkedlist在不间断地生成了新的对象，还是占用了一定的系统资源，并消耗时间。但是因为数组的连续性，所以ArrayList在尾端追加元素，只有当空间不足时才会产生数组扩容的现象。所以频繁在尾端追加元素的操作时，使用ArrayList相对来说效率会更好一点。

② 增加元素到任一位置

对ArrayList来说，在链表的任一位置插入元素会导致在插入位置之后的所有元素进行元素重组，所有的元素都需要向后移动一个；

对LinkedList来说，任意位置插入元素和在链表插入元素来说是一样的，都是新建一个entry对象然后进行赋值的操作即可。

所以当需要频繁地往链表任意位置进行插入操作时，选择LinkedList比ArrayList相对来说性能会更好一点。

③删除元素

对ArrayList来说，在链表的任一位置删除元素会导致在插入位置之后的所有元素进行元素重组，所有的元素都需要向前移动一个；

对LinkedList来说，首先通过循环链表找到要删除的元素，删除较为靠前或较为靠后的元素都是非常高效的，但是如果删除的是靠近中间的元素却几乎要遍历将近一半的链表元素 ，在列表拥有大量元素的情况下，效率较低。

1. 容量参数

是ArrayList和Vector等基于数组实现的list的特有的性能参数。它表示初始化的数组的大小。当ArrayList所存储的元素数量超过其已有大小时，便会进行扩容，扩容会导致整个数组进行一次内存复制。因此合理的数组大小有助于减少数组的扩容次数，从而提高系统性能。

## 14.2 Map

### 1.HashMap和HashTable的区别

① HashMap去掉了Hashtable中的contanins()方法，但是增加了containskey()

和containsValue()方法。

② Hashtable对大多数方法都加了同步，而HashMap没有，所以在效率上HashMap会比Hashtable好，但是HashMap不支持线程同步，但是Hashtable支持线程同步。

③HashMap中允许空的键值，但是Hashtable不允许。(奇怪的是我自己测试的时候Hashtable可以存放空的键值)

|  |
| --- |
| HashMap map = new HashMap<>();  map.put(null, "1");  map.put(1,null);  map.put(null,null);  Hashtable table = new Hashtable<>();  table.put(null, 1);  table.put(1, null);  table.put(null, null); |

④所继承的类不同

public class Hashtable extends Dictionary implements Map

public class HashMap extends AbstractMap implements Map

⑤计算key的下标索引时采用的方法不同。

Hashtable直接使用对象的hashCode，而HashMap则是重新计算hash值。

|  |
| --- |
| Hashtable的put操作中：int hash = key.hashCode();  HashMap的put操作中：**int** hash = *hash*(key.hashCode()); |

⑥默认初始化大小和扩容大小不同

HashTable中hash数组默认大小是11，扩容的方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。

### 2. HashMap的实现原理

HashMap是通过将key值做hash算法，然后把hash值映射到内存地址中，直接取得key所对应的数据。在HashMap中，底层使用的是数组 ，前面所说的内存地址也就指的是数组的下标索引。

(HashMap实际上是一个链表的数组)HashMap的内部维护这一个entry数组，每个entry项包含key、value、next、hash四项。当put操作发生冲突的时候，新的entry会被安放在对应的索引下标内，并替换原有的值，但是为了原有的旧值不丢失，会将新的entry的next指向旧值。

### 3. HashMap什么时候会进入死循环

由于HashMap不是线程安全的，所以当它在被多线程使用时，多线程同时put时，如果同时触发了rehash操作，会导致HashMap中的链表中出现死循环节点，进而使得后面get的时候会死循环。

### 4.ConcourrentHashMap

concurrenthashmap:ConcurrentHashMap是一个支持高并发的高性能的HashMap实现，它支持完全并发的读以及一定程度并发的写。采用的是锁分段技术：首先将数据分成一段一段地存储，然后给每一段数据搭配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

① 锁分离（16个hashmap）

② 使用不变性（final变量 segment所有成员都是不变的，但是数组里面的元素是可变的）和易变性（volatile变量（每段里面的count值为volatile（保证可见性））） ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。

从ConcurrentHashMap代码中可以看出，它引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的HashTable，根据key.hashCode()来决定把key放到哪个HashTable中。

在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是先根据key.hashCode()算出放到哪个Segment中。

1. **static** **final** **class** HashEntry<K,V> {
2. **final** K key;
3. **final** **int** hash;
4. **volatile** V value;
5. **final** HashEntry<K,V> next;
6. }

除了值不是final外，其它都是final，所以添加或删除都得进行逆序。但是get的时候不需要锁，同时值为volatile就能保证得到的是最新的值。put和delete需要锁，get不需要锁，containkey 也不需要锁了（因为key是final的）。

get ,最后，如果找到了所求的结点，判断它的值如果非空就直接返回，否则在有锁的状态下再读一次。（这种情况肯定很罕见了）。一旦发生这种情况，ConcurrentHashMap采取的方式是在持有锁的情况下再读一遍，这能够保证读到最新的值，并且一定不会为空值

1. V readValueUnderLock(HashEntry<K,V> e) {
2. lock();
3. **try** {
4. **return** e.value;
5. } **finally** {
6. unlock();
7. }
8. }

跨段操作的一些特性：在遍历的过程中要使用modcount监控是否发生结构变化。重试多次之后就会按顺序拿锁，这样就会出现性能消耗。

#### 删除操作

删除操作remove(key)：整个操作是先定位到段，然后委托给段的remove操作。当多个删除操作并发进行时，只要它们所在的段不相同，它们就可以同时进行。整个操作是在持有段锁的情况下执行的，空白行之前的行主要是定位到要删除的节点e。接下来，如果不存在这个节点就直接返回null，否则就要将e前面的结点复制一遍，尾结点指向e的下一个结点。e后面的结点不需要复制，它们可以重用。

整个remove实现并不复杂，但是需要注意如下几点。第一，当要删除的结点存在时，删除的最后一步操作要将count的值减一。这必须是最后一步操作，否则读取操作可能看不到之前对段所做的结构性修改。第二，remove执行的开始就将table赋给一个局部变量tab，这是因为table是volatile变量，读写volatile变量的开销很大。编译器也不能对volatile变量的读写做任何优化，直接多次访问非volatile实例变量没有多大影响，编译器会做相应优化。

#### put操作

接下来看put操作，同样地put操作也是委托给段的put方法

该方法也是在持有段锁的情况下执行的，首先判断是否需要rehash，需要就先rehash。接着是找是否存在同样一个key的结点，如果存在就直接替换这个结点的值。否则创建一个新的结点并添加到hash链的头部，这时一定要修改modCount和count的值，同样修改count的值一定要放在最后一步。put方法调用了rehash方法，reash方法实现得也很精巧，主要利用了table的大小为2^n。

#### 获取操作

同样ConcurrentHashMap的get操作是直接委托给Segment的get方法。

Get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空的才会加锁重读。Get方法立马将要使用的共享变量都定义成volatile，如用于统计当前当前segment的大小count字段和用户存储值的HashEntry的value。定义成volatile的变量，能够在多线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写。在get操作的过程中，只需要读count和value，所以可以不用加锁。

get操作不需要锁。第一步是访问count变量，这是一个volatile变量，由于所有的修改操作在进行结构修改时都会在最后一步写count变量，通过这种机制保证get操作能够得到几乎最新的结构更新。对于非结构更新，也就是结点值的改变，由于HashEntry的value变量是volatile的，也能保证读取到最新的值。接下来就是对hash链进行遍历找到要获取的结点，如果没有找到，直接访回null。对hash链进行遍历不需要加锁的原因在于链指针next是final的。但是头指针却不是final的，这是通过getFirst(hash)方法返回，也就是存在table数组中的值。这使得getFirst(hash)可能返回过时的头结点，例如，当执行get方法时，刚执行完getFirst(hash)之后，另一个线程执行了删除操作并更新头结点，这就导致get方法中返回的头结点不是最新的。这是可以允许，通过对count变量的协调机制，get能读取到几乎最新的数据，虽然可能不是最新的。要得到最新的数据，只有采用完全的同步。

最后，如果找到了所求的结点，判断它的值如果非空就直接返回，否则在有锁的状态下再读一次。这似乎有些费解，理论上结点的值不可能为空，这是因为put的时候就进行了判断，如果为空就要抛NullPointerException。空值的唯一源头就是HashEntry中的默认值，因为HashEntry中的value不是final的，非同步读取有可能读取到空值。仔细看下put操作的语句：tab[index] = new HashEntry<K,V>(key, hash, first, value)，在这条语句中，HashEntry构造函数中对value的赋值以及对tab[index]的赋值可能被重新排序，这就可能导致结点的值为空。这种情况应当很罕见，一旦发生这种情况，ConcurrentHashMap采取的方式是在持有锁的情况下再读一遍，这能够保证读到最新的值，并且一定不会为空值。

#### 跨段操作

有些操作需要涉及到多个段，比如说size(), containsValaue()。

size方法主要思路是先在没有锁的情况下对所有段大小求和，如果不能成功（这是因为遍历过程中可能有其它线程正在对已经遍历过的段进行结构性更新），最多执行RETRIES\_BEFORE\_LOCK次，如果还不成功就在持有所有段锁的情况下再对所有段大小求和。在没有锁的情况下主要是利用Segment中的modCount进行检测，在遍历过程中保存每个Segment的modCount，遍历完成之后再检测每个Segment的modCount有没有改变，如果有改变表示有其它线程正在对Segment进行结构性并发更新，需要重新计算。

14.3 Set

14.4 迭代器

迭代器是一个对象，它的工作是遍历并选择序列中的对象，而客户端程序不必知道或关心该序列底层的结构。此外，迭代器通常被称为轻量级对象：创建它的代价小。因此，经常可以看到对迭代器一些比较奇怪的限制，例如java的Iterator只能单向移动，这个Iterator只能用来：1）使用方法Iterator()要求容易返回一个Iterator。Iterator将准备好返回序列的第一个元素；2）使用next()获得序列中的下一个元素；3）使用hasNext()检查序列中是否还有元素；4)使用remove()将迭代器新近返回的元素删除，那么在调用remove()之前必须先调用next()。

|  |
| --- |
| **public** **class** TestIterator {  **public** **static** **void** main(String[] args){  ArrayList<String> list = **new** ArrayList<String>();  list.add("a");  list.add("b");  list.add("c");  /\*测试迭代器的使用\*/  Iterator it = list.iterator();  **while**(it.hasNext()){  System.*out*.println(it.next());  it.remove();  }  System.*out*.println(list.size());  }  } |

ListIterator是一个更加强大的Iterator的子类型，它只能用于各种List的访问。尽管Iterator只能向前移动，但是ListIterator可以双向移动。它还可以产生相对于迭代器在列表中指向当前位置的前一个和后一个元素的索引，并且可以使用set()方法替换它访问过的最后一个元素。

14.5 Stack

栈是指“后进先出”的容器。有时栈也被称为叠加栈，因为最后“压入”栈的元素，第一个“弹出”栈。LinkedList具有能够直接实现栈的所有功能的方法，因此可以直接将LinkedList作为栈使用，不过，有时一个真正的栈更能把事情讲清楚。

Net.mindview.util.Stack和java.util.Stack中都指Stack类，但是java.util.Stack中没有任何公共的Stack接口。尽管已经有了java.util.Stack，但是LinkedList可以产生更好的Stack，因此Net.mindview.util.Stack所采用的方式更是可取的。在使用的过程中，可以通过显式地导入来控制对“首选”Stack实现的选择：

Import Net.mindview.util.Stack

现在，任何对Stack的引用都将选择Net.mindview.util.Stack版本，而在选择java.util.Stack时，必须使用全限定名称。

14.6 Queue

队列是一个典型的“先进先出”的容器，即从容器的一端放入事务，从另一端取出，并且事务放入容器的顺序与取出的顺序是相同的。LinkedList提供了方法以支持队列的行为，并且它实现了Queue接口，因此LinkedList可以用作Queue的一种实现，比如Queue<String> queue = new LinkedList<>( String);

PriorityQueue是队列的一种，称作是优先级队列，在调用peek()、pool和remove()方法时，获取的元素将是队列中优先级最高的元素。

## 14.3. CopyOnWrite

### 概念

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后在新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器(遍历一个list的时候，还向list中添加元素)。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

### CopyOnWriteArrayList

向CopyOnWriteArrayList中add方法的实现（向CopyOnWriteArrayList里添加元素），可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。读的时候不需要加锁，如果读的时候有多个线程正在向CopyOnWriteArrayList添加数据，读还是会读到旧的数据，因为写的时候不会锁住旧的CopyOnWriteArrayList。

CopyOnWrite并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜索时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。

下面从“动态数组”和“线程安全”两个方面进一步对CopyOnWriteArrayList的原理进行说明。

1.**CopyOnWriteArrayList的“动态数组”机制** -- 它内部有个“volatile数组”(array)来保持数据。在“添加/修改/删除”数据时，都会新建一个数组，并将更新后的数据拷贝到新建的数组中，最后再将该数组赋值给“volatile数组”。这就是它叫做CopyOnWriteArrayList的原因！CopyOnWriteArrayList就是通过这种方式实现的动态数组；不过正由于它在“添加/修改/删除”数据时，都会新建数组，所以涉及到修改数据的操作，CopyOnWriteArrayList效率很低；但是单单只是进行遍历查找的话，效率比较高。

2.**CopyOnWriteArrayList的“线程安全”机制** -- 是通过volatile和互斥锁来实现的。① CopyOnWriteArrayList是通过“volatile数组”来保存数据的。一个线程读取volatile数组时，总能看到其它线程对该volatile变量最后的写入；就这样，通过volatile提供了“读取到的数据总是最新的”这个机制的保证。

② CopyOnWriteArrayList通过互斥锁来保护数据。在“添加/修改/删除”数据时，会先“获取互斥锁”，在修改完毕之后，先将数据更新到“volatile数组”中，然后再“释放互斥锁”；这样，就达到了保护数据的目的。

### CopyOnWriteMap

JDK中并没有提供CopyOnWriteMap，我们可以参考CopyOnWriteArrayList来实现一个。

代码很简单，但是使用CopyOnWriteMap需要注意两件事情：

① 减少扩容开销。根据实际需要，初始化CopyOnWriteMap的大小，避免写时CopyOnWriteMap扩容的开销。

② 使用批量添加。因为每次添加，容器每次都会进行复制，所以减少添加次数，可以减少容器的复制次数。如使用上面代码里的addBlackList方法。

### 使用时需要注意的

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所以在开发的时候需要注意一下。

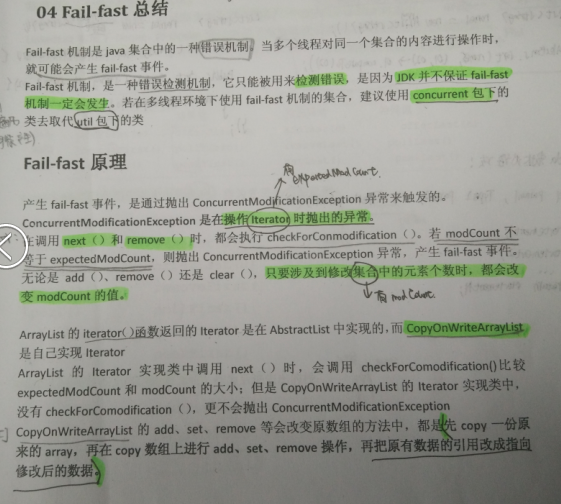
内存占用问题。因为CopyOnWrite的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时候会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占用的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候很有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite机制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全是10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他的并发容器，如ConcurrentHashMap。

数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

## 14.4. Fail-fast机制

在设计迭代器类时没有考虑到并发修改的问题，并且它们表现出的行为是“及时失败”(fail-fast)的。这就意味着容器在迭代过程中被修改时，就会被抛出ConcurrentModitificationException异常。实现的方式是：将计数器的变化与容器关联起来，如果在迭代器期间计数器被修改，那么hashNext和next将抛出。



## 14.5 迭代器

迭代器(Iterator)是一个对象，它的工作是遍历并选择序列中的对象，提供了一种访问容器对象中的各个元素，而又不暴露该对象内部细节的方法，通过迭代器，开发人员不需要了解容器底层的结构就可以实现对容器的遍历。由于创建迭代器的代价小，因此迭代器通常被称为轻量级的容器。

1. 字符串

1.不可变String

String对象是不可变的，String类中每一个看起来会修改String值的方法，实际上都是创建了一个全新的String对象，以包含修改后的字符串内容，而最初的String则丝毫未动。

|  |
| --- |
| **public** **class** TestString {  **public** **static** String g(String s){  s = s + " world";  **return** s;  }  **public** **static** **void** main(String[] args) {  String q = "hello";//q传给g方法之后再输出它的值没有发生变化  String qq = *g*(q);  System.*out*.println("q = "+q);  System.*out*.println("qq = "+qq);  }  } |

运行结果如下：



由于String的不可变性，所以任何对字符串s做的修改，得到都是一个新的String对象，除非把这个新得到的对象赋给之前的s，否则s的值是不会发生改变的。

15.2 针对常量池的优化

指当两个String对象拥有相同的值时，它们只引用常量池中的同一个拷贝。当一个相同的字符串反复出现时，这个就可以大幅度节省内存空间。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  String str1 = "abc";  String str2 = "abc";  String str3 = **new** String("abc");  System.*out*.println(str1 == str2);  System.*out*.println(str1 == str3);  System.*out*.println(str1 == str3.intern());  } |

运行结果如下：



以上代码显示str1和str2使用了相同的地址，但是str3却重新开辟了一块内存空间。但是即便如此，str3在常量池中的位置和str1是一样的，也就是说，虽然str3单独占用了堆空间，但是它所指向的实体和str1完全一样。str3.intern()方法返回了String对象在常量池中的引用。

15.3 String类的final定义

作为final类的String对象在系统中不可能有任何子类，这是对系统安全性的保护。

15.4 正则表达式

正则表达式是一种强大而灵活的文本处理工具。使用正则表达式，我们能够以编程的方式构造复杂的文本模式，并对输入的字符串进行搜索。正则表达式是一种简洁、动态的语言。正则表达式提供了一种完全通用的方式，能够解决各种字符串处理相关的问题：匹配、选择、编辑以及验证。

举例：

|  |  |
| --- | --- |
| 正则表达式 | 含义 |
| -?\\d+ | 可能有一个负号开头，后面跟着一位或多位数字 |
| \\W | 大写的W表示非单词字符 |
| \\w | 小写的w则表示一个单词字符 |
| \\w+ | 一个或多个单词字符 |

15.5 字符串分割

split()方法：将输入字符串断开成字符串数组，也可以限制将输入分割成字符串的数量。

StringTokenizer类是JDK中专门用来处理字符串分割子串的工具类使用方法如下：

|  |
| --- |
| String str = "zhang zhang zhang";  StringTokenizer st = **new** StringTokenizer(str," ");  **while**(st.hasMoreTokens()){  System.*out*.println(st.nextToken());  } |

综合比较使用StringTokenizer的效率要优于split方法。

15.6 subString()方法的内存泄露

截取字符串是字符串操作中最常用的操作之一，在java中，String类提供了两个截取字符串的方法。

public String substring(int beginIndex);

public String substring(int beginIndex, int endIndex);

通过查看其源代码可知，java是通过偏移量来决定要截取的字符串内容，String的原生内容value数组被复制到新的子字符串中，然后通过设置偏移量来确定返回的子字符串的内容。设想，如果原来的字符串很大，要截取的字符串很小，那么截取的子字符串中包含了原生字符串的所有内容，并占据了相应的内存空间，而仅仅通过偏移量和长度来决定自己的实际取值。这种算法提高了运算速度却浪费了大量的内存空间。

但是substring的执行速度非常快，只要处理好内存溢出问题就可以大胆使用。比如在返回一个字符串的子串时，使用new String（s.subString(1,2)）有优于直接返回s.subString(1,5).

15.7 StirngBuilder和StringBuffer

由于java中String对象是不可变对象，因此在需要对字符串进行修改操作时（如字符串连接、替换），String对象总是会生成一个新的Stirng对象，所以其性能相对较差。因此JDK提供了用于创建和修改字符串的工具StirngBuilder和StringBuffer。

（1）对于静态常量字符串的连接操作，Java在编译时会进行彻底的优化，将多个连接操作的字符串在编译时合成一个单独的长字符串。

（2）Java在编译时，就会对字符串进行一定的优化，因此，一些看起来会很慢的代码在执行的时候并不一定会很慢。

（3）对于String操作，类似于+和+=的运算符应该尽量少的使用。

（4）StirngBuilder和StringBuffer两个类几乎一样，区别在于StringBuffer几乎对所有的方法都做了同步，而StirngBuilder并没有做任何同步。在无需考虑线程安全的情况下，可以使用性能相对较好的StirngBuilder，但是若系统有线程安全要求，则只能用StringBuffer。

（5）容量参数

如果能够预先评估StirngBuilder的大小，将能够有效地节省在操作过程中对StirngBuilder对象进行数组扩展的操作。

15.8 String、StringBuffer、StringBuilder三者的使用场景

(1)String适用于字符串不经常变化的场景中，例如常量的声明、少量的变量声明运算等；

(2)StringBuffer适用于在频繁的进行字符串的操作(如拼接、替换、删除等)，并且是运行在多线程的环境下，例如XML解析、HTTP参数解析和封装等；

(3)StringBulider适用于频繁的对字符串进行操作但是在单线程的环境下，如SQL语句拼接、JSON封装等。

1. 数组

16.1 数组中常用知识点

数组与其他种类的容器之间的区别有三方面：效率、类型和保存基本类型的能力。在Java中，数组是一种效率最高的存储和随机访问对象引用序列的方式。

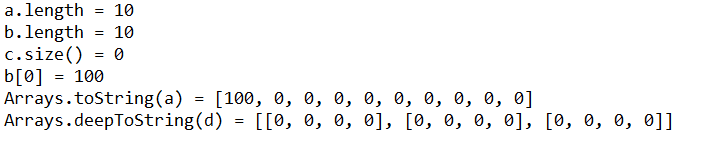
无论使用哪种类型的数组，数组标识符其实只是一个引用，指向堆中创建的一个真实对象，这个（数组）对象用以保存指向其他对象的引用。对象数组和基本类型数组在使用上几乎是相同的，唯一的区别就是对象数组保存的是引用，基本类型数组直接保存基本类型的值。

Length是表示数组的大小，而不是实际保存的元素个数。而容器中的size方法表示的是容器中当前持有的对象格式。此外 ，对于数组还可以进行数组之间对复制，这样的话两个数组引用指向堆中同一个对象引用。Arrays.toString(a)可以输出数组a的内容，如果是多维数组d则调用Arrays.deepToString(d)来打印数组d中的内容。使用Arrays.fill(a,1)可以把整个数组a都填充数值1，使用Arrays.fill(a,1,4,10)则是把数组a中下标[1，4)对应的元素填充为10.

如下代码：

|  |
| --- |
| **public** **static** **void** main(String[] args){  **int** a[] = **new** **int**[10];  **int** b[] = a;//a和b则指向堆中同一个对象。  ArrayList c = **new** ArrayList(10);  System.*out*.println("a.length = "+a.length);  System.*out*.println("b.length = "+b.length);  System.*out*.println("c.size() = "+c.size());    a[0] = 100;  System.*out*.println("b[0] = "+b[0]);  System.*out*.println("Arrays.toString(a) = "+Arrays.*toString*(a));  **int** d[][] = **new** **int**[3][4];  System.*out*.println("Arrays.deepToString(d) = "+Arrays.*deepToString*(d));  } |

输出为：



当某个方法的返回值是一个数组时，与返回任何其他对象没有区别，因为实际上返回的都是引用。

16.2 Arrays实用功能方法

① System.arraycopy()针对所有类型都做了重载，用它复制数组比用for循环复制要快得多。该方法有五个参数，分别表示：源数组、表示从源数组中的什么位置开始复制的偏移量、目标数组、表示从目标数组的什么位置开始复制的偏移量、需要复制的元素个数。在使用的过程中药注意数组不可以越界。

如果复制的是对象数组，那么只是复制了对象的引用，而不是对象本身的拷贝，这被称作是浅复制(shallow sopy)。

② 数组的比较

Arrays.*equals*(a1, a2)用于比较两个数组是否相等。两个数组相等的条件首先是元素个数必须相等，并且对应位置的元素也相等。同样，该方法也可以比较包含的元素是对象的数组。相关代码如下所示：

|  |
| --- |
| **class** Test{  }  **public** **class** MethodTest {  **public** **static** **void** main(String[] args){  **int** a1[] = **new** **int**[10];  **int** a2[] = **new** **int**[10];  Arrays.*fill*(a1, 86);  Arrays.*fill*(a2, 86);  System.*out*.println(Arrays.*equals*(a1, a2));//数组中包含的是基本类型的比较    Test t1[] = **new** Test[10];  Test t2[] = **new** Test[10];  Arrays.*fill*(t1, **new** Test());  Arrays.*fill*(t2, **new** Test());  System.*out*.println(Arrays.*equals*(t1, t2));//数组中包含的是对象    Test t = **new** Test();  Arrays.*fill*(t1, t);  Arrays.*fill*(t2, t);  System.*out*.println(Arrays.*equals*(t1, t2));//数组中包含的是对象  }  } |

运行结果：



③ 数组元素的比较

对类A，在定义的时候实现了Comparable<A>接口中的compareTo(A)方法，让当前对象与接收的参数对象进行比较（返回正值、负值和0）。如果此时有数组a包含的对象类型A，则可以调用Arrays.sort（a）方法对数组a进行排序。

④ 数组排序

可以采用内置的排序方法，可对任意的基本类型数组排序，也可以对任意的对象数组进行排序，只要该对象实现了Comparable接口或具有相关联的Comparator。

⑤ 在已排序的数组中查找

如果数组已经排好序了，就可以使用Arrays.binarySearch()执行快速查找。如果对未排好序的数组使用binarySearch()，那么将产生不可预料的结果。

1. 静态类

它们仅包含[静态成员](http://baike.baidu.com/view/534160.htm)。

它们不能被实例化。

它们是密封的。

它们不能包含实例构造函数。

因此创建静态类与创建仅包含静态成员和私有构造函数的类大致一样。私有构造函数阻止类被[实例化](http://baike.baidu.com/view/2338289.htm)。

优点：使用静态类的优点在于，[编译器](http://baike.baidu.com/view/487018.htm)能够执行检查以确保不致偶然地添加实例成员。编译器将保证不会创建此类的实例。

静态类是密封的，因此不可被继承。静态类不能包含构造函数，但仍可声明静态构造函数以分配初始值或设置某个静态状态。

1. Class对象

(1) Class简介

Class对象就是用来创建类的所有的“常规”对象的。类是程序的一部分，每个类都有一个Class对象，也就是说每当编写并且编译了一个新类，就会产生一个Class对象（更恰当地说，是被保存在了一个同名的.class文件中）。为了生成这个类的对象，运行这个程序的Java虚拟机将使用被称为“类加载器”的子系统。

所有的类都是在对其第一次使用时，动态加载到JVM中的。Class对象仅在被需要的时候才加载，static初始化是在类加载时进行的。

（2）获取class对象的三种方式

下面的代码中除了给出了三种获取class对象的方式之外，还提供了另外三个常用的接口。

知识点共有如下六个，其中1,2,3提供了三种获取class类（class类与class对象的引用说法不同，但表示的意思一样，指的都是一个java类编译之后保存的同名的.class文件的一个对象）的方式。

1. Class.forName方法返回的是给定的字符串(某个类的全限定名，包含包名如exerise20160309.FancyToy)表示的类的class对象的引用（即class类），JVM会加载这个类，同时JVM会执行该类的静态代码段；

2. obj.getClass()即实例名.getClass()返回这个实例对象的class类；

3. Toy.class即类名.class(类字面常量)返回这个类的class类;使用类字面常量返回Class类

4. c.getInterfaces()获取class类c中包含的接口；

5. c.getSuperclass()返回的是class类c的父类的class类；

6. newInstance(),获取这个class类的一个对象，并且这个class类中必须有无参的构造方法.

|  |
| --- |
| **package** exerise20160309;  **public** **class** ToyTest {    **static** **void** printInfo(Class cc){  System.*out*.println("class name :"+cc.getName() + " is interface? ["+cc.isInstance(cc)+"]");  }    **public** **static** **void** main(String[] args){  Class c = **null**;  **try** {  /\*Class.forName方法返回的是给定的字符串表示的类的class对象的引用，JVM会加载这个类，同时JVM会执行该类的静态代码段\*/  c = Class.*forName*("exerise20160309.FancyToy");  } **catch** (ClassNotFoundException e) {  System.*out*.println("没有找到FancyToy的class对象");  e.printStackTrace();  }  *printInfo*(c);  /\*c.getInterfaces()获取class对象c中包含的接口\*/  **for**(Class face : c.getInterfaces()){  *printInfo*(face);  }    /\* c.getSuperclass()返回的是其基类的class对象\*/  Class up = c.getSuperclass();  *printInfo*(up);    Toy obj = **null**;    **try** {  /\*newInstance(),获取这个class类的一个对象，并且这个class类中必须有无参的构造方法\*/  obj = (Toy) up.newInstance();  } **catch** (InstantiationException | IllegalAccessException e) {  e.printStackTrace();  }    /\*obj.getClass()返回这个实例对象的class类\*/  *printInfo*(obj.getClass());  /\*Toy.class即类名.class返回这个类的class类\*/  *printInfo*(Toy.**class**);  }  }  **interface** HasBatteries{}  **interface** Waterproof{}  **interface** Shoots{}  **class** Toy{  Toy(){}  Toy(**int** i){}  }  **class** FancyToy **extends** Toy **implements** HasBatteries,Waterproof,Shoots{  FancyToy(){  **super**(1);  }  } |

（2）如何获取父类的类名

先获取到了父类再调用的getName()方法。

|  |
| --- |
| **class** A{  }  **public** **class** Test1 **extends** A{  /\*\*  \* 这个地方输出的是Test1而不是A，原因是Java中任何类都继承自Object类，  \* getClass()方法在Object类中被定义为final与native，  \* 子类不可以覆盖这个方法。Object类中的getClass()方法的含义是:返回此Object类的运行的类。  \*  \*/  **public** **void** f(){  System.*out*.println(**super**.getClass().getName());  }  /\*\*  \* 注意这个地方可以输出父类A的名字原因是，先获取到了父类再调用的getName()方法。  \*/  **public** **void** g(){  System.*out*.println(**this**.getClass().getSuperclass().getName());  }    **public** **static** **void** main(String[] args){  **new** Test1().f();  }  } |

（3）已知的RTTI形式

RTTI：Running Time Type Identification运行时识别一个对象的类型。

使用RTTI的原因是使得代码更容易写，更容易读，且更便于维护，设计也更容易实现，理解和改变。

① 传统的类型转换，Base b = new Sub1()，Base b = new Sub2()。引用为父类型的引用只有在运行时才可以确定具体是哪个子类的类型。



② 代表对象的类型的class对象，通过查询class对象可以获取运行时所需的信息。

③ instanceof 关键字

如 b instanceof Base会返回true，如果b不是Base子类的对象的话就会返回false。这个关键字的作用是检查对象b是否从属于Base类。

使用instanceof关键字比较时需要注意的是只可将其与命名类型进行比较，而不能与Class对象进行比较。

需要注意的是instanceof关键字的左右两个操作数必须要有继承或实现关系，不能用于基本类型的判断，null instanceof 任意类型返回的都是false。

1. 反射机制

利用反射机制，它允许程序在运行时进行自我检查，同时也允许对其内部的成员进行操作。反射也可以获取和修改对象中成员变量或方法即使是非公共权限的，但是final域在遭受修改时是安全的。

(1).反射的使用

Class类与java.lang.reflect类库一起对反射的概念进行了支持，该类库包含了Field、Method以及Constructor类。这些类的对象是由JVM在运行时创建的，用以表示未知类里面的成员。

Class类的这三个常用的方法如下：

|  |  |
| --- | --- |
| [**getMethods**](mk:@MSITStore:F:\找工作\电子书\JAVA%20API官方文档中文版.CHM::/java/lang/Class.html#getMethods())() | 返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些类或接口）的公共 member 方法。 |
| [**getFields**](mk:@MSITStore:F:\找工作\电子书\JAVA%20API官方文档中文版.CHM::/java/lang/Class.html#getFields())() | 返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段。 |
| [**getConstructors**](mk:@MSITStore:F:\找工作\电子书\JAVA%20API官方文档中文版.CHM::/java/lang/Class.html#getConstructors())() | 返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法。  注意这个时候就可以利用反射机制通过非默认构造器创建对象了。 |

但是反射也是可以获取和修改对象中成员变量或方法即使是非公共权限的，但是final域在遭受修改时是安全的。上面列出的只是其中获取公共权限方法、变量和构造器的三个调用方法。

当通过反射与一个未知类型的对象打交道时，JVM只是简单地检查这个对象，看它属于哪个特定的类。在用它做某件事情之前必须加载那个类的Class对象。因此，那个类的.class文件对于JVM来说必须是可以获取到的：要么在本地上，要么可以通过网络获得。

(2)反射与RTTI的区别

所以RTTI和反射之间真正的区别在于RTTI是编译器在编译时打开和检查.class文件，而对于反射机制来说，.class文件在编译时是不可获取的，所以是在运行时打开和检查.class文件。

1. 反射的使用场景

反射的使用场景：①如果不记得一个类是否有某个方法，或者不知道一个类究竟可以做什么，而又不想通过索引或类的层次结构去查找JDK文档，这时用反射工具就能得到这个类有哪些方法。②在跨网络的远程平台上创建和运行对象。

1. 反射的优缺点

优点：增加程序的灵活性；

缺点：不恰当地使用反射机制，会严重影响系统的性能。

1. 反射提供的功能

① 得到一个对象所属的类；

② 获取一个类内所有公共权限的变量和方法；

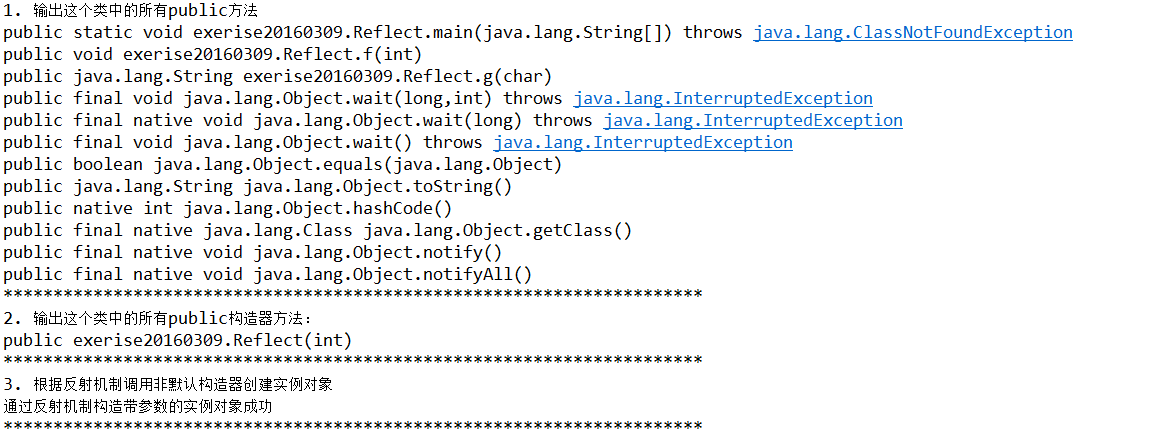
③ 获取一个类内所有public构造器方法；

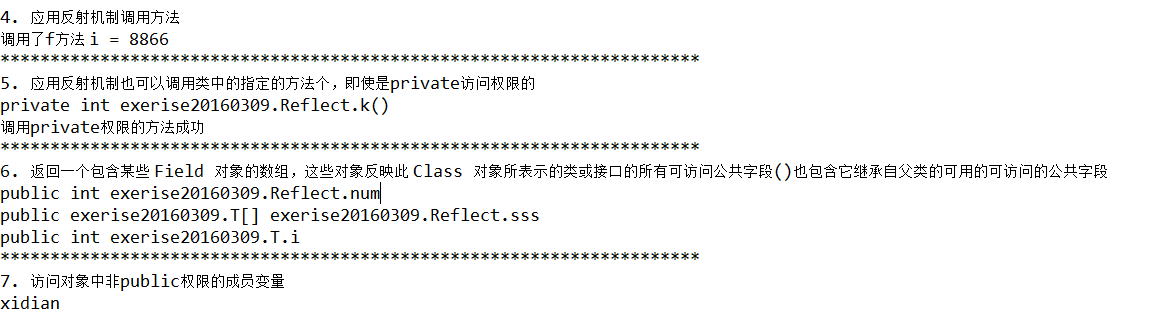
④ 运行时创建对象；

⑤ 运行时调用对象的方法。

如下代码展示了如何通过使用java.lang.reflect类库中Method和Constructor获取指定class对象中包含的方法和public访问权限的构造器方法。

|  |
| --- |
| **package** exerise20160309;  **import** java.lang.reflect.\*;  **class** T{  **public** **int** i;  }  **public** **class** Reflect **extends** T{  **public** **int** num;  **public** T[] sss;  **private** String str = "xidian";  **public** Reflect(**int** i){  System.*out*.println("通过反射机制构造带参数的实例对象成功");  }  **private** Reflect(**int** i,**int** j){  }  **public** **void** f(**int** i){  System.*out*.println("调用了f方法 i = "+i);  }  **public** String g(**char** c){  **return** "";  }  **private** **int** k(){  System.*out*.println("调用private权限的方法成功");  **return** 999;  }  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException{  Class c = Class.*forName*("exerise20160309.Reflect");  Method[] methods = c.getMethods();  Constructor[] ctors = c.getConstructors();//返回的是public权限的构造器  System.*out*.println("1. 输出这个类中的所有public方法");  **for**(Method m : methods){  System.*out*.println(m.toString());  }    System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("2. 输出这个类中的所有public构造器方法：");  **for**(Constructor con : ctors){  System.*out*.println(con.toString());  }  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("3. 根据反射机制调用非默认构造器创建实例对象");  Constructor constructor = ctors[0];  Reflect r = **null**;  **try** {  r = (Reflect) constructor.newInstance(3);  } **catch** (InstantiationException | IllegalAccessException  | IllegalArgumentException | InvocationTargetException e) {  e.printStackTrace();  }    System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("4. 应用反射机制调用方法");  **try** {  methods[1].invoke(r, 8866);  } **catch** (IllegalAccessException e1) {  e1.printStackTrace();  } **catch** (IllegalArgumentException e1) {  e1.printStackTrace();  } **catch** (InvocationTargetException e1) {  e1.printStackTrace();  }  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("5. 应用反射机制也可以调用类中的指定的方法个，即使是private访问权限的");  **try** {  Method mm = c.getDeclaredMethod("k");  System.*out*.println(mm);  mm.setAccessible(**true**);  mm.invoke(r);  } **catch** (NoSuchMethodException | SecurityException | IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");    System.*out*.println("6. 返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段()也包含它继承自父类的可用的可访问的公共字段");  Field[] fs = c.getFields();  **for**(Field f:fs){  System.*out*.println(f);  }  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.*out*.println("7. 访问对象中非public权限的成员变量");  **try** {  Field f = c.getDeclaredField("str");  f.setAccessible(**true**);  System.*out*.println(f.get(r));  } **catch** (NoSuchFieldException | SecurityException | IllegalAccessException e) {  e.printStackTrace();  }  }  } |





1. 空对象

有的时候可以使用空对象来代替null，即使空对象可以响应“实际”对象可以响应的所有消息，你仍需要某种方式去探测是否为空。要达到此目的，最简单的方式是创建一个标记接口。

1. 动态代理

(1) 设计模式中的代理

代理是基本的设计模式之一，它是为了提供额外的或不同的操作，而插入的用来代替实际对象的对象。

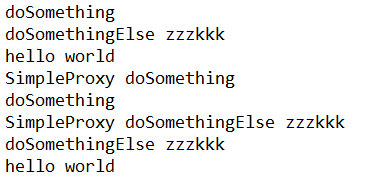
代理是保证了在任何时刻可以把额外的工作从“实际”对象中分离到不同的地方，希望可以很容易地做出修改，从没有使用这些操作到可以使用这些操作，或者是反过来，这个时候代理就显得很有用。

如下代码表示的是RealObject实现了Interface接口 ，并且完成了一定的功能，现在想在完成已有的工作的基础上，再进行一层封装，用SimpleProxy同样实现Interface接口中的方法并且代理RealObject类，在实现方法内部调用被代理实例的方法之前或之后都可以再添加其他的操作。

|  |
| --- |
| **interface** Interface{  **void** doSomething();  **void** doSomethingElse(String args);  } |

|  |
| --- |
| **class** RealObject **implements** Interface{  @Override  **public** **void** doSomething() {  System.*out*.println("doSomething");  }  @Override  **public** **void** doSomethingElse(String args) {  System.*out*.println("doSomethingElse "+args);  }    } |

|  |
| --- |
| **public** **class** SimpleProxy **implements** Interface{//代理类  **private** Interface proxied;//被代理的实例  **public** SimpleProxy(Interface proxied){  **this**.proxied = proxied;  }  @Override  **public** **void** doSomething() {  System.*out*.println("SimpleProxy doSomething");  proxied.doSomething();  }  @Override  **public** **void** doSomethingElse(String args) {  System.*out*.println("SimpleProxy doSomethingElse "+args);  proxied.doSomethingElse(args);  }    }  **public** **class** SimpleProxyDemo {  **public** **static** **void** consumer(Interface iface){  iface.doSomething();  iface.doSomethingElse("zzzkkk");  System.*out*.println("hello world");  }  **public** **static** **void** main(String[] args){  *consumer*(**new** RealObject());  *consumer*(**new** SimpleProxy(**new** RealObject()));  }  } |



(2) java中的动态代理

Java中的动态代理比设计模式中的静态代理更前进了一步，因为java中的动态代理可以动态地创建代理并动态地处理对所代理方法的调用。在动态代理上所做的所有调用都会被重定向到单一的调用处理器上，它的工作是揭示调用的类型并确定相应的对策。

动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。

java.lang.reflect 包中的InvocationHandler 接口和Proxy类提供了生成动态代理类的能力。

**InvocationHandler**

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke 方法来进行调用。

Invoke方法有三个参数：

① obj - 在其上调用方法的代理实例.

② method - (指代的是我们所要调用真实对象的某个方法的Method对象)。Method 对象的声明类将是在其中声明方法的接口，该接口可以是代理类赖以继承方法的代理接口的超接口。

③ args - 包含传入代理实例上方法调用的参数值的对象数组(指代的是调用真实对象某个方法时接受的参数)，如果接口方法不使用参数，则为 null。基本类型的参数被包装在适当基本包装器类（如 java.lang.Integer 或 java.lang.Boolean）的实例中。

**Proxy类**

Proxy类是提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。

Proxy.newProxyInstance创建动态代理，需要三个参数，Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

①是类加载器ClassLoader loader:一个ClassLoader对象，定义了由哪个ClassLoader对象来对生成的代理对象进行加载.

②是该代理实现的接口列表（不是类或抽象类）Class<?>[] interfaces:一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了.

③是InvocationHandler接口的一个实现:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上.

Proxy.newProxyInstance 创建的代理对象是在jvm运行时动态生成的一个对象，它并不是我们的InvocationHandler类型，也不是我们定义的那组接口的类型，而是在运行是动态生成的一个对象，并且命名方式都是这样的形式，以$开头，proxy为中，最后一个数字表示对象的标号。

代码示例如下：

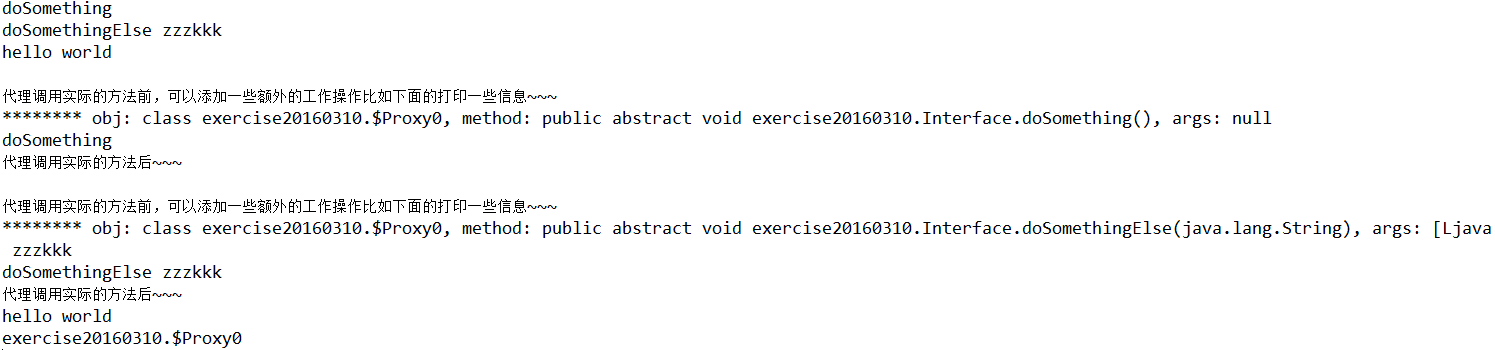
|  |
| --- |
| **package** exercise20160310;  **import** java.lang.reflect.InvocationHandler;  **import** java.lang.reflect.Method;  **class** DynamicProxyHander **implements** InvocationHandler{  **private** Object proxied;//真实的被代理的实例  **public** DynamicProxyHander(Object proxied){  **this**.proxied = proxied;  }  /\*\*  \* 在代理实例上处理方法调用并返回结果。  \* 在与方法关联的代理实例上调用方法时，将在调用处理程序上调用此方法。  \* obj - 在其上调用方法的代理实例.  \* method - 对应于在代理实例上调用的接口方法的 Method 实例(指代的是我们所要调用真实对象的某个方法的Method对象)。Method 对象的声明类将是在其中声明方法的接口，该接口可以是代理类赖以继承方法的代理接口的超接口。  \* args - 包含传入代理实例上方法调用的参数值的对象数组(指代的是调用真实对象某个方法时接受的参数)，如果接口方法不使用参数，则为 null。基本类型的参数被包装在适当基本包装器类（如 java.lang.Integer 或 java.lang.Boolean）的实例中。  \*/  @Override  **public** Object invoke(Object obj, Method method, Object[] args)  **throws** Throwable {  System.*out*.println();  System.*out*.println("代理调用实际的方法前，可以添加一些额外的工作操作比如下面的打印一些信息~~~");  System.*out*.println("\*\*\*\*\*\*\*\* obj: "+obj.getClass() +", method: "+method+", args: "+args);  **if**(args!=**null**){  **for**(Object arg : args){  System.*out*.println(" "+arg);  }  }  method.invoke(proxied, args);//把请求分发给真正代理的对象。  System.*out*.println("代理调用实际的方法后~~~");  **return** **null**;  }  } |

|  |
| --- |
| **package** exercise20160310;  **import** java.lang.reflect.\*;  **public** **class** SimpleDynamicProxy {  **public** **static** **void** consumer(Interface iface){  iface.doSomething();  iface.doSomethingElse("zzzkkk");  System.*out*.println("hello world");  }  **public** **static** **void** main(String[] args){  RealObject real = **new** RealObject();  *consumer*(real);  /\*  \* Proxy.newProxyInstance创建动态代理，需要三个参数，Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)  \* 一个是类加载器ClassLoader loader:一个ClassLoader对象，定义了由哪个ClassLoader对象来对生成的代理对象进行加载.  \* 一个是该代理实现的接口列表（不是类或抽象类）Class<?>[] interfaces:一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了.  \* 一个是InvocationHandler接口的一个实现:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上.  \*  \* 下面的这行代码表示的是动态代理RealObject的一个实例对象real，指定由Interface.class的类加载器加载这个代理的对象，该对象实现了Interface接口。  \* 注意在这里Interface只是我声明的一个接口。  \*  \* 当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke方法来进行调用.  \*/  Interface proxy = (Interface) Proxy.newProxyInstance(  Interface.class.getClassLoader(), new Class[]{Interface.class}, new DynamicProxyHander(real));  *consumer*(proxy);  System.*out*.println(proxy.getClass().getName());  }  } |

想要正确查看上面的代码运行的结果还需要下面的一个接口和一个具体的实现类。

|  |
| --- |
| **interface** Interface{  **void** doSomething();  **void** doSomethingElse(String args);  }  **class** RealObject **implements** Interface{  @Override  **public** **void** doSomething() {  System.*out*.println("doSomething");  }  @Override  **public** **void** doSomethingElse(String args) {  System.*out*.println("doSomethingElse "+args);  }  } |

输出结果如下：



上述代码的类图如下



动态代理说白了就是RealObject目前可以完成一定的工作，但是现在想要在不改变RealObject工作的基础上给他增加额外的工作，这个时候就用到了动态代理，写了一个DynamicProxyHander类实现 InvocationHandler接口，实现了接口中的invoke方法,增加完成代理类方法工作前后的一些额外工作，并且在调用的时候是通过下面这条代码实现的。

Interface proxy = (Interface) Proxy.newProxyInstance (Interface.class.getClassLoader(), new Class[]{Interface.class}, new DynamicProxyHander (real));

当我们通过代理对象调用一个方法的时候比如proxy作为参数调用consumer(proxy);时，这个方法内部有需要去调用形参proxy对象内的方法，此时这个方法的调用就会被转发为由InvocationHandler这个接口的invoke方法来进行调用。具体表示含义可见上面的讲解。这说明了动态代理会将所有调用重定向到调用处理器，因此通常调用的时候会向调用处理器的构造器传递一个实际对象的引用，从而使得调用处理器在执行其中介任务时可以将请求转发。

动态代理类的另外一个好处是，如果现在有很多种代理对象都需要被同一种代理方式进行代理，如都需要在自身方法执行前后都去做同样的事情，这个时候可以采取动态代理的方式，这样只需要写一个代理类即可，在生成代理对象的时候只需要变化这个地方的代码就可以了。

Interface proxy = (Interface) Proxy.newProxyInstance(

Interface.class.getClassLoader(), new Class[]{Interface.class}, new DynamicProxyHander(real));

比如，如果我现在加上了另外的一个接口Interface2和它的实现类如下：

|  |
| --- |
| **public** **interface** Interface2 {  **public** **void** f();  }  **public** **class** RealObject2 **implements** Interface2 {  @Override  **public** **void** f() {  System.*out*.println("0912");  }  } |

那么对ObjectReal2中的方法调用如下：

|  |
| --- |
| public static void consumer2(Interface2 iface){  iface.f();  System.out.println("hello world");  } |

在生成动态代理对象的时候

|  |
| --- |
| RealObject2 real2 = new RealObject2();  consumer2(real2);  Interface2 p2 = (Interface2) Proxy.newProxyInstance(  Interface2.class.getClassLoader(), new Class[]{Interface2.class}, new DynamicProxyHander(real2));  consumer2(p2); |

可以发现两种不一样的实际操作，但是都需要在调用其方法前后做相同的操作，所以就可以共用同一个DynamicProxyHander代理类。

(3). CGLIB代理

JDK的动态代理机制只能代理实现了接口的类，而不能实现接口的类就不能实现JDK的动态代理，cglib是针对类来实现代理的，它的原理是对指定的目标类生成一个子类，并覆盖其中方法实现增强，但因为采用的是继承，所以不能对final修饰的类进行代理。

1. **package** net.battier.dao;
3. **public** **interface** BookFacade {
4. **public** **void** addBook();
5. }
6. **package** net.battier.dao.impl;
8. /\*\*
9. \* 这个是没有实现接口的实现类
10. \*
11. \* @author student
12. \*
13. \*/
14. **public** **class** BookFacadeImpl1 {
15. **public** **void** addBook() {
16. System.out.println("增加图书的普通方法...");
17. }
18. }
19. **package** net.battier.proxy;
21. **import** java.lang.reflect.Method;
23. **import** net.sf.cglib.proxy.Enhancer;
24. **import** net.sf.cglib.proxy.MethodInterceptor;
25. **import** net.sf.cglib.proxy.MethodProxy;
27. /\*\*
28. \* 使用cglib动态代理
29. \*
30. \* @author student
31. \*
32. \*/
33. **public** **class** BookFacadeCglib **implements** MethodInterceptor {
34. **private** Object target;
36. /\*\*
37. \* 创建代理对象
38. \*
39. \* @param target
40. \* @return
41. \*/
42. **public** Object getInstance(Object target) {
43. **this**.target = target;
44. Enhancer enhancer = **new** Enhancer();
45. enhancer.setSuperclass(**this**.target.getClass());
46. // 回调方法
47. enhancer.setCallback(**this**);
48. // 创建代理对象
49. **return** enhancer.create();
50. }
52. @Override
53. // 回调方法
54. **public** Object intercept(Object obj, Method method, Object[] args,
55. MethodProxy proxy) **throws** Throwable {
56. System.out.println("事物开始");
57. proxy.invokeSuper(obj, args);
58. System.out.println("事物结束");
59. **return** **null**;

62. }
64. }
65. **package** net.battier.test;
67. **import** net.battier.dao.impl.BookFacadeImpl1;
68. **import** net.battier.proxy.BookFacadeCglib;
70. **public** **class** TestCglib {
72. **public** **static** **void** main(String[] args) {
73. BookFacadeCglib cglib=**new** BookFacadeCglib();
74. BookFacadeImpl1 bookCglib=(BookFacadeImpl1)cglib.getInstance(**new** BookFacadeImpl1());
75. bookCglib.addBook();
76. }
77. }

(4). JDK动态代理与CGLIB代理

JDK动态代理只能针对实现了接口的类生成代理。CGLIB（CODE GENERLIZE LIBRARY）代理是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的所有方法，所以该类或方法不能声明称final的。

①.JDK动态代理

此时代理对象和目标对象实现了相同的接口，目标对象作为代理对象的一个属性，具体接口实现中，可以在调用目标对象相应方法前后加上其他业务处理逻辑。

②.CGLIB代理

CGLIB（CODE GENERLIZE LIBRARY）代理是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的所有方法，所以该类或方法不能声明称final的。

如果目标对象没有实现接口，则默认会采用CGLIB代理；

如果目标对象实现了接口，可以强制使用CGLIB实现代理（添加CGLIB库，并在spring配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）。

AOP包括切面（aspect）、通知（advice）、连接点（joinpoint），实现方式就是通过对目标对象的代理在连接点前后加入通知，完成统一的切面操作。

1. Java中的数据库操作

Java数据库连接(Java DataBase Connectivity ,JDBC)用于在Java程序中实现数据库连接，它提供了执行SQL语句、访问数据库的方法，并为各种不同的数据库提供统一的操作操作接口，java.sql包中包含了JDBC操作的所有类。

JDBC驱动程序就是实现JDBC中所有接口的实现类的集合；通常会把这些实现类以JAR文件的形式推出。所以JDBC支持通过特定的程序把用于JDBC的调用映射成适当的数据库特定的调用，把这样的程序称为JDBC驱动程序。

通过JDBC访问数据库一般有如下几个步骤：

① 加载JDBC驱动程序。将数据库的JDBC驱动加载到classpath中，在基于JavaEE的web应用程序的开发过程中，通常是把目标数据库产品的JDBC驱动复制到对应的lib文件夹下；

② 加载JDBC驱动，将其注册到DriverManager中。一般使用反射Class.forName(String driverName);

③ 建立数据库连接，取得Connection对象。一般通过DriverManager.getConnection (url,username,passwd)方法实现，其中，url表示连接数据库的字符串，username表示连接数据库的用户名，passwd表示连接数据库的密码；

④ 建立Statement对象或是PreparedStatement对象；

⑤ 执行SQL语句；

⑥ 访问结果集ResultSet对象；

⑦ 依次将各个连接进行释放。

备注：Statement对象、PreparedStatement对象和CallableStatement对象有什么区别？

①Statement用于执行不带参数的sql语句，并且返回它所生成结果的对象，每次执行SQL语句时，都需要编译该SQL语句；

② PreparedStatement对象能够完成与Statement对象一样的功能，并且由于PreparedStatement对象执行SQL命令时，命令会被数据库进行编译和解析，然后放到命令缓冲区，所以后面再有相同的PreparedStatement对象时，只需要被解析但是不需要被编译；代码的可读性和可维护性好；安全性更好。

③CallableStatement对象是专门用来执行数据库函数和存储过程的Statement。是Statement的子类。

1. cookie和session的区别

Cookie是在HTTP下，服务器或脚本可以维护客户工作站上信息的一种形式，它是由web服务器保存在用户浏览器上的小文件，可以包含用户的信息如身份识别号码、密码等。Session是指用来在客户端与服务器端之间保持状态的解决方案和存储结构。

二者的区别是：

* 1. cookie保存在客户端，session保存在服务端；
  2. cookie能保存的信息只能是有限的文本信息，而会话可以保存对象，并且保存对象的数量和大小没有限制。
  3. cookie的安全性不够，由于其放在客户端上，其他人很容易获取存放在其中的东西，并进行cookie欺骗，但session的信息存档在服务器端，较为安全；
  4. cookie性能相对高一点。由于session就会一定时间内保存在服务端上，因此当访问量多时，性能就会受到影响。

1. JSON和XML

  JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。在数据传输数据过程中，传递的只是一种符合Json语法格式的字符串，归根到底，传输的还是字符串。在web应用程序开发中，可以通过json实现前台与后台之间的数据交换。

         JSON建构有两种结构：

         json简单说就是javascript中的对象和数组，所以这两种结构就是对象和数组2种结构，通过这两种结构可以表示各种复杂的结构

　　1、对象：对象在js中表示为“{}”括起来的内容，数据结构为 {key：value，key：value,...}的键值对的结构，在面向对象的语言中，key为对象的属性，value为对应的属性值，所以很容易理解，取值方法为 对象.key 获取属性值，这个属性值的类型可以是 数字、字符串、数组、对象几种。在Java中就是一种Map结构。

　　2、数组：数组在js中是中括号“[]”扩起来的内容，数据结构为 ["java","javascript","vb",...]，取值方式和所有语言中一样，使用索引获取，字段值的类型可以是 数字、字符串、数组、对象几种。在Java中是一种list结构。

XML就是扩展标记语言 (Extensible Markup Language, XML) ，用于标记电子文件使其具有结构性的标记语言，可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。 XML使用DTD(document type definition)文档类型定义来组织数据;格式统一，跨平台和语言，早已成为业界公认的标准。

XML是标准通用标记语言 (SGML) 的子集，非常适合 Web 传输。XML 提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。

(1).XML的优缺点  
<1>.XML的优点  
　　A.格式统一，符合标准；  
　　B.容易与其他系统进行远程交互，数据共享比较方便。  
<2>.XML的缺点  
　　A.XML文件庞大，文件格式复杂，传输占带宽；  
　　B.服务器端和客户端都需要花费大量代码来解析XML，导致服务器端和客户端代码变得异常复杂且不易维护；  
　　C.客户端不同浏览器之间解析XML的方式不一致，需要重复编写很多代码；  
　　D.服务器端和客户端解析XML花费较多的资源和时间。

(2).JSON的优缺点  
<1>.JSON的优点：  
　　A.数据格式比较简单，易于读写，格式都是压缩的，占用带宽小；  
　　B.易于解析，客户端JavaScript可以简单的通过eval()进行JSON数据的读取；  
　　C.支持多种语言，包括ActionScript, C, C#, ColdFusion, Java, JavaScript, Perl, PHP, Python, Ruby等服务器端语言，便于服务器端的解析；  
　　D.在PHP世界，已经有PHP-JSON和JSON-PHP出现了，偏于PHP序列化后的程序直接调用，PHP服务器端的对象、数组等能直接生成JSON格式，便于客户端的访问提取；  
　　E.因为JSON格式能直接为服务器端代码使用，大大简化了服务器端和客户端的代码开发量，且完成任务不变，并且易于维护。  
<2>.JSON的缺点  
　　A.没有XML格式这么推广的深入人心和喜用广泛，没有XML那么通用性；  
　　B.JSON格式目前在Web Service中推广还属于初级阶段。

1. java为什么要配置环境变量，配置环境变量的作用？

(1)path是配置Windows可执行文件的搜索路径，即扩展名为.exe的程序文件所在的目录，用于指定DOS窗口命令的路径。 %JAVA\_HOME%\bin; %JAVA\_HOME%\jre\bin

(2)Classpath是配置class文件所在的目录，用于指定类搜索路径，JVM就是通过它来寻找该类的class类文件的。D:\java\jdk\lib

%JAVA\_HOME% = D:\java\jdk

1. 比较一下Java 和JavaSciprt

答：JavaScript 与Java是两个公司开发的不同的两个产品。Java 是原Sun Microsystems公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而JavaScript是Netscape公司的产品，为了扩展Netscape浏览器的功能而开发的一种可以嵌入Web页面中运行的基于对象和事件驱动的解释性语言。JavaScript的前身是LiveScript；而Java的前身是Oak语言。   
下面对两种语言间的异同作如下比较：   
① 基于对象和面向对象：Java是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象；JavaScript是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象（Object-Based）和事件驱动（Event-Driven）的编程语言，因而它本身提供了非常丰富的内部对象供设计人员使用。   
② 解释和编译：Java的源代码在执行之前，必须经过编译。JavaScript是一种解释性编程语言，其源代码不需经过编译，由浏览器解释执行。（目前的浏览器几乎都使用了JIT（即时编译）技术来提升JavaScript的运行效率）   
③ 强类型变量和类型弱变量：Java采用强类型变量检查，即所有变量在编译之前必须作声明；JavaScript中变量是弱类型的，甚至在使用变量前可以不作声明，JavaScript的解释器在运行时检查推断其数据类型。   
④ 代码格式不一样。

1. J2EE、EJB、JavaBean

J2EE(Java2 Platform,Enterprise Edition)是Java平台企业版的简称，用来开发与部署企业级应用的一个架构，它提供了一个统一的、开放标准的多层平台，该平台主要由构件、服务和通信3个模块构成。构件包含客户端构件和服务端构件两种。

EJB是Enterprise JavaBean的简称，相当于分布式组件对象模型，它是一种服务器端组件体系结构，用来开发和部署多层的、分布式的以及面向对象的应用系统的跨平台体系结构。

JavaBean是可复用的组件，理论上讲，任何一个Java类都可以是一个Bean ,但是Java类要满足两个条件：首先类是公开的的并且有一个默认的无参的构造方法；其次在类中可以定义若干个私有属性，但是需要通过公共的getter/setter方法对这些属性进行赋值和取值，并且getter/setter方法不能重载。

1. New I/O

NIO是new I/O的简称，与旧式的基于流的I/O相对，表示新的一套Java I/O标准。是在Java1.4 中被纳入到JDK中，并具有以下特性：

① 为所有的原始类型提供(Buffer)缓冲支持；

② 使用java.nio.charset.Charset作为字符集编码解码解决方案；

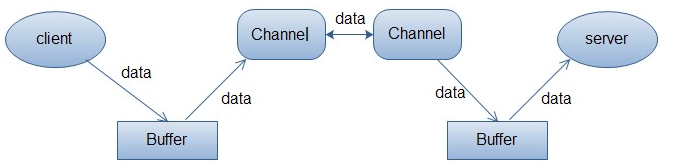
③ 增加通道(Channel)对象，作为新的原始I/O抽象；

④ 支持锁和内存映射文件的文件访问接口；

⑤ 提供了基于Selector的异步网络I/O.

与流式的I/O不同，NIO是基于块(Block)的，它以块为基本单位处理数据。在NIO中，最重要的两个组件是Buffer和Channel。缓冲是一块连续的内存块，是NIO读写数据的中转地。通道表示缓冲数据的源头或目的地，用于向缓冲区读取或者写入数据。

在NIO中和Buffer配合使用的是Channel，Channel是一个双向通道，既可读，也可写。应用程序中不能直接对Channel进行读写操作，必须把数据读进Buffer或者从Buffer中往Channel中写数据。



NIO非阻塞的实现主要是采用了Reactor(反应器)设计模式，与Observer(观察者)设计模式类似，只不过Observer设计模式只能处理一个事件源，而Reactor可以处理多个事件源。

在实现时，把需要处理的Channel的IO事件(例如connect、Read或write等)注册给Selector。Selector内部的实现原理为：对所有注册的Channel进行轮询访问，一旦轮询到一个Channel有注册的事件发生，例如有数据来了，它就通过传回Selection-Key的方式来通知开发人员对Channel进行数据的读或写操作。Key(由Selection-Key类来表示)封装一个特定的Channel和一个特定的selector之间的关系。

这种通过轮询的方式在处理多线程请求时不需要上下文的切换，而采用多线程的实现方式在线程之间切换时需要上下文的切换，同时也需要进行压栈与弹栈操作。因此，NIO有较高的执行效率。NIO在网络编程中有着非常重要的作用，与传统的Socket方式相比，由于NIO采用了非阻塞的方式，在处理大量并发请求时，使用NIO要比使用Socket效率高得多。

下面给出通过FileChannel来向文件中写入数据的一个例子：

|  |
| --- |
| public class Test {      public static void main(String[] args) throws IOException  {          File file = new File("data.txt");          FileOutputStream outputStream = new FileOutputStream(file);          FileChannel channel = outputStream.getChannel();          ByteBuffer buffer = ByteBuffer.allocate(1024);          String string = "java nio";          buffer.put(string.getBytes());          buffer.flip();     //此处必须要调用buffer的flip方法          channel.write(buffer);          channel.close();          outputStream.close();      }  } |

1. 比较一下int和Integer

## 两种类型的比较

① Ingeter是int的包装类；

② int的初值为0，Ingeter的初值为null

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int** i = 128;  Integer i2 = 128;  Integer i3 = **new** Integer(128);  //Integer会自动拆箱为int，所以为true  System.*out*.println(i == i2);//true  System.*out*.println(i == i3);//true  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  Integer i5 = 127;//java在编译的时候,被翻译成-> Integer i5 = Integer.valueOf(127);  Integer i6 = 127;  System.*out*.println(i5 == i6);//true  /\*Integer i5 = 128;  Integer i6 = 128;  System.out.println(i5 == i6);//false\*/  Integer ii5 = **new** Integer(127);  System.*out*.println(i5 == ii5); //false  Integer i7 = **new** Integer(128);  Integer i8 = **new** Integer(128);  System.*out*.println(i7 == i8); //false  } |

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Integer a = 1;  Integer b = 2;  Integer c = 3;  Integer d = 3;  Integer e = 321;  Integer f = 321;  Long g = 3L;  System.*out*.println(c == d);//true，自动拆箱  System.*out*.println(e == f);//false，两个都是非new出来的Integer，如果数在-128到127之间，则是true；不在这个范围之内都是false  System.*out*.println(c==(a+b));//true  System.*out*.println(c.equals(a+b));//a+b之后也是Integer，与c的类型相同，并且值相同，返回true  System.*out*.println(c.equals(d));//true  System.*out*.println(g == (a+b));//自动拆箱true  // System.out.println(g == c);//直接编译错误  System.*out*.println(g.equals((a+b)));//a+b之后也是Integer，equals方法计算时不进行类型转换，如果类型不同，则返回false    } |

## 比较int和Integer的值时(==)：

注意，如果是调用Integer对象的equals方法，则只要值相等，则返回true。

以下几条说都是根据==比较。

① 如果是两个int型的值进行比较时只要值相等则返回true；

② int 和Integer比较(不管是否为new出来的如Integer i=22；或Integer i = new(22))时看的是值是否相等，只要和i值相等都为true，因为会把Integer自动拆箱为int再去比；

③ 两个都是非new出来的Integer，如果数在-128到127之间，则是true；不在这个范围之内都是false；  java在编译Integer i2 = 128的时候,被翻译成-> Integer i2 = Integer.valueOf(128);而valueOf()函数会对-128到127之间的数进行缓存。在通过valueOf方法创建Integer对象的时候，如果数值在[-128,127]之间，便返回指向IntegerCache.cache中已经存在的对象的引用；否则创建一个新的Integer对象。

|  |
| --- |
| private static class IntegerCache {  static final int high;  static final Integer cache[];  static {  final int low = -128;  // high value may be configured by property  int h = 127;  if (integerCacheHighPropValue != null) {  // Use Long.decode here to avoid invoking methods that  // require Integer's autoboxing cache to be initialized  int i = Long.decode(integerCacheHighPropValue).intValue();  i = Math.max(i, 127);  // Maximum array size is Integer.MAX\_VALUE  h = Math.min(i, Integer.MAX\_VALUE - -low);  }  high = h;  cache = new Integer[(high - low) + 1];  int j = low;  for(int k = 0; k < cache.length; k++)  cache[k] = new Integer(j++);  }  private IntegerCache() {}  } |

④ 两个都是new 出来的Integer，都为false。

⑤ 无论如何，Integer与new Integer不会相等。不会经历拆箱过程，new Integer的引用指向堆，而Integer指向专门存放他的内存（常量池），他们的内存地址不一样，所以为false。

## 拆箱和装箱

注意：装箱就是自动将基本数据类型转换为包装器类型；拆箱就是自动将包装器类型转换为基本数据类型。装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。（xxx代表对应的基本数据类型）

具体内容可以参考：<http://www.cnblogs.com/dolphin0520/p/3780005.html>。

1. Java语法糖的味道

语法糖指的是在计算机语言中添加的某种语法，这种语法对于语言的功能没有多少实质性的帮助，但是更加方便供程序员使用，可以增加程序的可读性，提升语法的严谨性，减少编码出错的机会，提高代码的效率。

JVM中的语法糖：泛型、自动装箱、自动拆箱、循环遍历、变长参数、条件编译、内部类、枚举类、断言语句、对枚举的switch。

1. 注解 Annotation

注解(也被称为元数据)为我们在代码中添加信息提供了一种形式化的方法，使以后的某个时刻使用这些数据方便。使用注解能够帮助我们避免编写累赘的部署描述文件以及其他生成的文件。注解使得能够以编译器测试和验证的格式，存储有关程序的额外信息。注解提供了更加干净易读的代码以及编译期类型检查。

注解的语法比较简单，除了@符号的使用之外，基本与Java固有的语法一致。定义在java.lang中的三种标准注解：

1 @override，表示当前的方法定义将覆盖超类中的方法。如果不小心拼写错误，或者方法签名对不上被覆盖的方法，编译器就会发出错误提示。

2 @Deprecated，如果程序员使用了注解为它的元素，那么编译器就会发出警告信息。

3 @SuppressWarnings,关闭不当的编译器警告信息。

四种元注解：

1 @Target,表示注解可以用于什么地方，可能的ElementType参数包括：CONSTRUCTOR：构造器的声明；FIELD：域声明(包括enum实例)；LOCAL\_VARIABLE：局部变量声明；METHOD：方法声明；PACKAGE：包声明；PARAMETER：参数声明；TYPE：类、接口(包括注解类型)或enum声明。

2 @Rentention，表示在什么级别保存该注解信息，可选的参数有SOURCE：注解将会被编译器丢弃；CLASS：注解在class文件中可用，但会被VM丢弃；RUNTIME：VM将在运行期也保留注释，因此可以通过反射机制读取注解的信息。

3 @Documented，将此注解包含在Javadoc中。

4 @Inherited，允许子类继承父类中的注解。

1. Java Socket

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个双向链路的一端称为一个Socket。Socket也称为套接字，可以用来实现不同虚拟机或不同计算机之间的通信。在Java语言中 ，Socket可以被分为两种类型：面向连接的Socket通信协议(TCP)和面向无连接的Socket通信协议(UDP)。

任何一个socket都是由IP地址和端口号唯一确定的。

基于TCP的通信过程为三次握手。

Socket的生命周期可以分为三个阶段：打开Socket、使用Socket收发数据和关闭Socket。在Java语言中，可以使用ServerSocket来作为服务器端，Socket作为客户端来实现网络通信。

常见笔试题：

用socket实现客户端和服务端的通信，要求客户端发送数据后能够回显相同的数据。

|  |
| --- |
| **import** java.io.BufferedReader;  **import** java.io.InputStreamReader;  **import** java.io.PrintWriter;  **import** java.net.ServerSocket;  **import** java.net.Socket;  /\*\*  \* 服务端的程序  \* **@author**  \*  \*/  **public** **class** Server {  **public** **static** **void** main(String[] args){  BufferedReader br = **null**;  PrintWriter pw = **null**;  **try**{  /\*在服务器端2000端口启动监听\*/  ServerSocket server = **new** ServerSocket(2000);  /\*等待客户端的连接请求\*/  Socket socket = server.accept();  /\*获取输入流\*/  br = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));  /\*获取输出流\*/  pw = **new** PrintWriter(socket.getOutputStream(),**true**);  /\*获取接收到的内容\*/  String s = br.readLine();  /\*发送相同的数据给客户端\*/  pw.println(s);  }**catch**(Exception e){  e.printStackTrace();  }**finally**{  **try**{  br.close();  pw.close();  }**catch**(Exception e){  e.printStackTrace();  }  }  }  } |
| **import** java.io.BufferedReader;  **import** java.io.InputStreamReader;  **import** java.io.PrintWriter;  **import** java.net.Socket;  /\*\*  \* 客户端的程序  \* **@author**  \*  \*/  **public** **class** Client {  **public** **static** **void** main(String[] args){  BufferedReader br = **null**;  PrintWriter pw = **null**;    **try** {  /\*向本机的2000端口传送数据，客户端Socket会随机挑选一个本机端口发送数据\*/  Socket socket = **new** Socket("localhost",2000);  /\*获取输入流\*/  br = **new** BufferedReader(**new** InputStreamReader(socket.getInputStream()));  /\*获取输出流\*/  pw = **new** PrintWriter(socket.getOutputStream(),**true**);  /\*向服务器端发送数据\*/  pw.println("hello");  String s = **null**;  /\*等待接收服务器端返回的数据\*/  **while**(**true**){  s = br.readLine();  **if**(s!=**null**){  **break**;  }  }  System.*out*.println(s);  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  **try**{  br.close();  pw.close();  }**catch**(Exception e){  e.printStackTrace();  }  }  }  } |

1. 深入分析Java I/O的工作机制

## 基础知识

大多数web应用系统的瓶颈都是I/O瓶颈。

基于字节操作的I/O接口：InputStream和OutputStream，

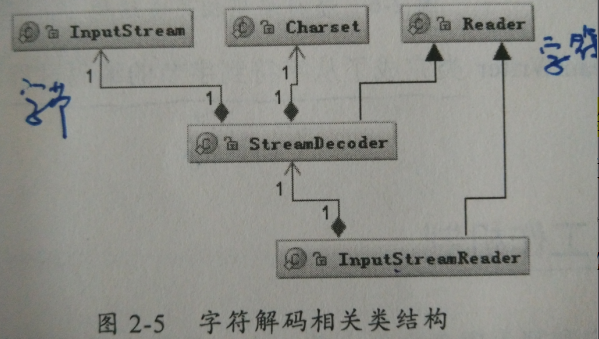
基于字符操作的I/O接口：Writer和Reader。Reader和Writer则提供兼容Unicode与面向字符的I/O功能.

基于磁盘操作的I/O接口：File

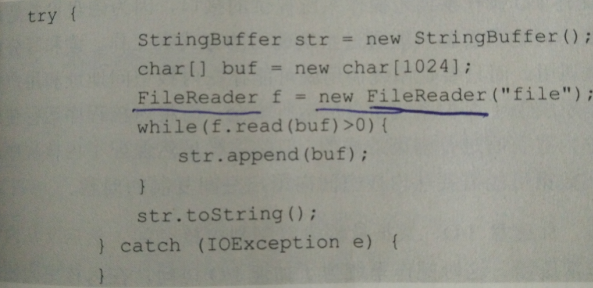
基于网络操作的I/O接口：Socket

不管是磁盘还是网络传输，最小的存储单元都是字节，而不是字符，所以I/O操作的都是字节而不是字符，但是为什么会有操作字符的I/O接口呢？这是因为在平时的程序中通常操作的数据都是字符形式的，为了操作方便所以JDK就提供了操作字符的I/O接口。

另外，数据持久化或网络传输都是以字节形式进行的，所以必须要有从字符到字节或从字节到字符的转化。



InputStreamReader类是从字节到字符的转化桥梁，从InputStream到Reader的过程要指定编码字符集，否则将采用操作系统默认的字符集，很可能会出现乱码的问题。StreamDecoder正是完成从字节到字符的解码的实现类。即当你用如下方式读取一个文件时：



FileReader类就是按照上面的工作方式读取文件的，FileReader继承了InputStreamReader类，实际上是读取文件流，然后通过StreamDecoder解码成char(字符)，只不过这里的解码字符集是默认字符集。写入的过程也是类似。

设计Reader和Writer继承层次结构主要是为了国际化。老的I/O流继承层级结构仅支持8位字节流，并且不能很好地处理16位的Unicode字符。由于Unicode用于字符国际化，所以添加Reader和Writer继承层次结构就是为了在所有的I/O操作中都支持Unicode。

最明智的做法是尽量尝试使用Reader和Writer，一旦程序代码无法成功编译，再去选择使用面向字节的InputStream和OutputStream.

## BufferedInputStream和BufferedOutputStream

BufferedInputStream：使用它可以防止每次读取时都进行实际写操作。代表“使用缓冲区”。

BufferedOutputStream：使用它可以防止每次写操作时都进行实际写操作。代表“使用缓冲区”，可以调用flush()清空缓冲区。

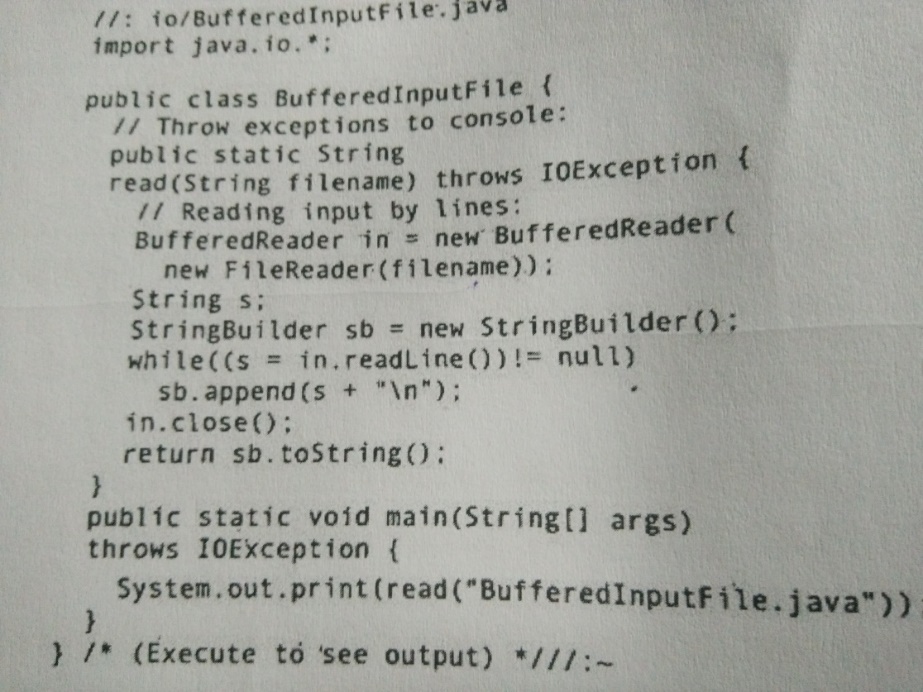
## I/O流的典型使用方式

无论何时使用readLine()，都不应该使用DataInputStream，而应该使用BufferedReader。除了这一点，DataInputStream仍是I/O类库的首选成员。

① 缓冲输入文件

如果想要打开一个文件用于字符输入，可以使用以String或File对象作为文件名的FileInputReader。为了提高速度，我们希望对那个文件进行缓冲，那么我们将所产生的引用传给一个BufferReader。由于BufferedReader也提供readLine()方法，所以这是我们的最终对象和进行读取的接口。当readLine()将返回null时，就到了文件的末尾。

字符串sb用来积累文件的全部内容(包括必须添加的换行符，因为readLine()已将它们删除掉)。最后调用close()关闭文件。



## 如何按照指定编码格式读写文件

注意：InputStreamReader类是从字节到字符的转化桥梁，OutputstreamWriter类是从字符到字节的转化桥梁。所以如果需要添加上编码时只需要在生成InputStreamReader或者OutputstreamWriter对象的时候加上编码格式即可。

对于文件，只需要从里到外包装File(可省),FileInputStream, InputStreamReader，BufferedReader即可。即先是读到字节，然后字节转化为字符，然后把字符送到缓冲区。

|  |
| --- |
| import java.io.BufferedReader;  import java.io.BufferedWriter;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.OutputStreamWriter;  public class ReadWriteFileWithEncode {  public static void write(String path, String content, String encoding)  throws IOException {  File file = new File(path);  file.delete();  file.createNewFile();  BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(  new FileOutputStream(file), encoding));  writer.write(content);  writer.close();  }  public static String read(String path, String encoding) throws IOException {  String content = "";  File file = new File(path);  BufferedReader reader = new BufferedReader(new InputStreamReader(  new FileInputStream(file), encoding));  String line = null;  while ((line = reader.readLine()) != null) {  content += line + "\n";  }  reader.close();  return content;  }  public static void main(String[] args) throws IOException {  String content = "中文内容";  String path = "c:/test.txt";  String encoding = "utf-8";  ReadWriteFileWithEncode.write(path, content, encoding);  System.out.println(ReadWriteFileWithEncode.read(path, encoding));  }  } |

## 怎样实现文件的复制

实现文件的复制主要有三步：

Step1：建立内容的输出和输入流；(因为是读文件，所以用新建一个FileInput

Stream对象即可)；

Step2：声明一个buffer数组用于读文件，初始化长度为4096，但是每次读取的时候读到=内容即返回，不一定会读满4096个字节；

Step3：从输入流中读到内容，然后写到输出流里面，最后要记得关闭输入流和输出流。

|  |
| --- |
| /\*\*  \* 实现把source路径的文件复制到target路径的文件内。  \* **@param** source  \* **@param** target  \* **@throws** IOException  \*/  **public** **void** copyFile(String source, String target) **throws** IOException{    InputStream in = **new** FileInputStream(source);  OutputStream out = **new** FileOutputStream(target);  **byte**[] bufffer = **new** **byte**[4096];  **int** len;//表示每次读到的长度  **while**((len = in.read(bufffer)) != -1){  out.write(bufffer,0,len);  }  in.close();  out.close();  } |

或者

|  |
| --- |
| public static void copyfile(String source, String target) throws IOException{  BufferedInputStream bs = new BufferedInputStream(new FileInputStream(source));  BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(target));  byte[] buffer = new byte[1024];  int len = 0;  while((len = bs.read(buffer))!=-1){  out.write(Arrays.copyOf(buffer, len));  }  bs.close();  out.close();  } |

## 怎样将GB2312编码的字符串转化为ISO-8859-1编码的字符串？

String s1 = “你好”；

String s2 = new String(s1.getBytes(“GB2312”),”ISO-8859-1”);

待验证，输出不对，需要进一步理解。

1. 两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？

答：不对，如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code）应当相同。

Java对于eqauls方法和hashCode方法是这样规定的：(1)如果两个对象相同（equals方法返回true），那么它们的hashCode值一定要相同；(2)如果两个对象的hashCode相同，它们并不一定相同；(3)如果两个对象的hashCode不同，那么他们的equals结果也肯定是不同的；(4)如果两个对象的equals方法不同，则其hashCode可以同也可以不同。

**补充：**关于equals和hashCode方法，很多Java程序都知道，但很多人也就是仅仅知道而已，在Joshua Bloch的大作《Effective Java》（很多软件公司，《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍，如果你还没看过，那就赶紧去[亚马逊](http://z.cn/)买一本吧）中是这样介绍equals方法的：首先equals方法必须满足① 自反性（x.equals(x)必须返回true）、② 对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）、③ 传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）和④ 一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。

实现高质量的equals方法的诀窍包括：①. 使用==操作符检查"参数是否为这个对象的引用"；②. 使用instanceof操作符检查"参数是否为正确的类型"；③. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；④. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；⑤. 重写equals时总是要重写hashCode；⑥. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

1. 数据类型之间的转换

## 如何将字符串转化成基本数据类型？

调用基本类型对应包装类型的parseXXX(string)或valueOf(string)即可返回相应基本类型；

## 如何将基本数据类型转化成字符串？

将基本数据类型与空字符串(“”)连接(+)即可获得其所对应的字符串；另一种方法是调用String类中的valueof()方法返回相应字符串。

1. TreeMap和TreeSet在排序时如何比较元素？Collections工具类中的sort()方法如何比较元素？

TreeSet要求存放的对象所属的类必须实现Comparable接口，该接口提供了比较元素的compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap要求存放的键值对映射的键必须实现Comparable接口从而根据键对元素进行排序。Collections工具类的sort方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象必须实现Comparable接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator接口的子类型（需要重写compare方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java中对函数式编程的支持）。

例子1：

|  |
| --- |
| public class Student implements Comparable<Student> {  private String name; // 姓名  private int age; // 年龄  public Student(String name, int age) {  this.name = name;  this.age = age;  }  @Override  public String toString() {  return "Student [name=" + name + ", age=" + age + "]";  }  @Override  public int compareTo(Student o) {  return this.age - o.age; // 比较年龄(年龄的升序)  }  }  import java.util.Set;  import java.util.TreeSet;  class Test01 {  public static void main(String[] args) {  Set<Student> set = new TreeSet<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)  set.add(new Student("Hao LUO", 33));  set.add(new Student("XJ WANG", 32));  set.add(new Student("Bruce LEE", 60));  set.add(new Student("Bob YANG", 22));  for(Student stu : set) {  System.out.println(stu);  }  // 输出结果:  // Student [name=Bob YANG, age=22]  // Student [name=XJ WANG, age=32]  // Student [name=Hao LUO, age=33]  // Student [name=Bruce LEE, age=60]  }  } |

例子2：

|  |
| --- |
| public class Student {  private String name; // 姓名  private int age; // 年龄  public Student(String name, int age) {  this.name = name;  this.age = age;  }  /\*\*  \* 获取学生姓名  \*/  public String getName() {  return name;  }  /\*\*  \* 获取学生年龄  \*/  public int getAge() {  return age;  }  @Override  public String toString() {  return "Student [name=" + name + ", age=" + age + "]";  }  }  import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;  import java.util.List;  class Test02 {  public static void main(String[] args) {  List<Student> list = new ArrayList<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)  list.add(new Student("Hao LUO", 33));  list.add(new Student("XJ WANG", 32));  list.add(new Student("Bruce LEE", 60));  list.add(new Student("Bob YANG", 22));  // 通过sort方法的第二个参数传入一个Comparator接口对象  // 相当于是传入一个比较对象大小的算法到sort方法中  // 由于Java中没有函数指针、仿函数、委托这样的概念  // 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调  Collections.sort(list, new Comparator<Student> () {  @Override  public int compare(Student o1, Student o2) {  return o1.getName().compareTo(o2.getName()); // 比较学生姓名  }  });  for(Student stu : list) {  System.out.println(stu);  }  // 输出结果:  // Student [name=Bob YANG, age=22]  // Student [name=Bruce LEE, age=60]  // Student [name=Hao LUO, age=33]  // Student [name=XJ WANG, age=32]  }  } |

1. Java中如何实现序列化，有什么意义？

答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间或者是将对象的状态持久化到数据库或文件中。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。

要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第29题）。

1. 简述正则表达式及其用途。

答：在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

**说明：**计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，正则表达式就是在进行字符串匹配和处理的时候最为强大的工具，绝大多数语言都提供了对正则表达式的支持。

1. Java中是如何支持正则表达式操作的？

答：Java中的String类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，Java中可以用Pattern类表示正则表达式对象，它提供了丰富的API进行各种正则表达式操作，请参考下面面试题的代码。

**面试题：** - 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

|  |
| --- |
| import java.util.regex.Matcher;  import java.util.regex.Pattern;  class RegExpTest {  public static void main(String[] args) {  String str = "北京市(朝阳区)(西城区)(海淀区)";  Pattern p = Pattern.compile(".\*?(?=\\()");  Matcher m = p.matcher(str);  if(m.find()) {  System.out.println(m.group());  }  }  } |

**说明：**上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的[《正则表达式30分钟入门教程》](http://www.jb51.net/tools/zhengze.html)