[键入公司名称]

2014

Java文档

java

Me

[键入公司地址]

[一、 java编译与查看运行时内部结构 3](#_Toc485149078)

[二、 Java基本知识 4](#_Toc485149079)

[(一) java数据类型 4](#_Toc485149080)

[(二) 八种基本类型+void 4](#_Toc485149081)

[(三) Java引用类型 6](#_Toc485149082)

[(四) JVM的内存空间： 7](#_Toc485149083)

[(五) 类型转换 7](#_Toc485149084)

[三、 java初始化顺序 10](#_Toc485149085)

[四、 java序列化与反序列化 11](#_Toc485149086)

[五、 java遍历 13](#_Toc485149087)

[(六) java遍历：for、do while、while、foreach、iterator的用法和区别 13](#_Toc485149088)

[(七) java遍历集合 14](#_Toc485149089)

[六、 java泛型 16](#_Toc485149090)

[(一) 泛型类： 16](#_Toc485149091)

[(二) 泛型方法： 17](#_Toc485149092)

[七、 java类型之间的关系判断（instanceof、isAssignableFrom） 19](#_Toc485149093)

[1) instanceof 19](#_Toc485149094)

[2) isAssignableFrom 19](#_Toc485149095)

[八、 Java知识外延 20](#_Toc485149096)

[(一) java.lang.Class类 20](#_Toc485149097)

[1) getName(),getSimpleName(),getCanonicalName ()三者之间的区别 20](#_Toc485149098)

[(二) 数据库访问部分 20](#_Toc485149099)

[1) javax.persistence.EntityManager; 20](#_Toc485149100)

[2) javax.persistence.Query 21](#_Toc485149101)

[3) 关于sql语句的说明 21](#_Toc485149102)

[4) 注意事项 21](#_Toc485149103)

# java编译与查看运行时内部结构

注：TestLambda.java 文件编写TestLambda类，并且其包名为package com.test;

编译：

1. 直接编译

javac com/test/TestLambda.java

1. 指定编码

javac -encoding UTF-8 com/test/TestLambda.java

查看运行时内部结构：

1. 直接反编译查看结构

javap -p com/test/TestLambda.class

1. 输出运行时生成的内部类和方法

java -Djdk.internal.lambda.dumpProxyClasses com/test/TestLambda

# Java基本知识

## java数据类型

数据类型

基本数据类型

引用数据类型

类(class)

接口(interface)

数组

注解

布尔型(boolean)

字符型(char)

数值型

整数类型（byte, short, int, long）

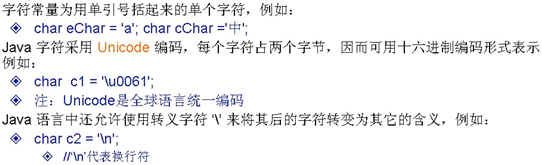
浮点类型（float, double）

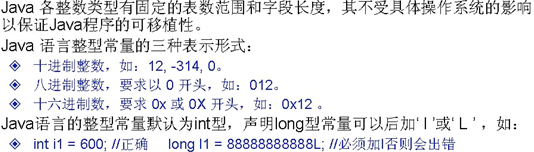
数值类型不存在无符号的，它们的取值范围是固定的，不会随着机器硬件环境或者操作系统的改变而改变。实际上，JAVA中还存在另外一种基本类型void，它也有对应的包装类 java.lang.Void，不过我们无法直接对它们进行操作。

## 八种基本类型+void

* 八种类型表示范围如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 封装类 | 存储空间 | 范围 |
| boolean | Boolean | 1位 | true或false；Boolean. TRUE; Boolean.FALSE; |
| char | Character | 2字节,16位  Character.SIZE | '\u0000'~'\uFFFF'；  Character. MIN\_VALUE; Character. MAX\_VALUE; |
| byte | Byte | 1字节,8位  Byte.SIZE | [-2^7,2^7-1]；  即：-128 ~ 127  Byte. MIN\_VALUE; Byte. MAX\_VALUE; |
| short | Short | 2字节,16位  Short.SIZE | [-2^15,2^15-1]；  即：-32768 ~ 32767  Short. MIN\_VALUE; Short. MAX\_VALUE; |
| int | Integer | 4字节,32位  Integer.SIZE | [-2^31,2^31-1]；  即：-2147483648 ~ 2147483647 共10位  Integer. MIN\_VALUE; Integer. MAX\_VALUE;  十六进制整型常量：以十六进制表示时，需以0x或0X开头，如0xff,0X9A。  八进制整型常量：八进制必须以0开头，如0123，034 |
| long | Long | 8字节,64位  Long.SIZE | [-2^63,2^63-1]；  即：-9223372036854775808到9223372036854775807共19位  Long. MIN\_VALUE; Long. MAX\_VALUE;  长整型必须以l/L作结尾，如9L,342L。 |
| float | Float | 4字节,32位  Float.SIZE | Float. MIN\_VALUE; Float. MAX\_VALUE;  数值必须以f/F做结尾 |
| double | Double | 8字节,64位  Double.SIZE | Double. MIN\_VALUE; Double. MAX\_VALUE;  小数可以以d/D结尾，默认类型，可以忽略 |
| void | Void |  |  |





注意：float、double两种类型的最小值与Float.MIN\_VALUE、 Double.MIN\_VALUE的值并不相同，实际上Float.MIN\_VALUE和Double.MIN\_VALUE分别指的是 float和double类型所能表示的最小正数。也就是说存在这样一种情况，0到±Float.MIN\_VALUE之间的值float类型无法表示，0 到±Double.MIN\_VALUE之间的值double类型无法表示。这并没有什么好奇怪的，因为这些范围内的数值超出了它们的精度范围。

在数学中0到1有无数个浮点数；而计算机是离散的，所以表示的时候有误差，计算机用精度（小数点后几位来表示正确），比较浮点数时a==0.1是不合适的，应该a-0.1==0;如果a是0.1,则即使有误差 a-0.1==0因为a和0.1都被表示为一个有误差的计算机二进制;

float与double表面相同的值，未必相等;

比如

0.1f == 0.1d ，返回false

1.1f == 1.1d ，返回false

1f == 1d ，返回true

Float和Double的最小值和最大值都是以科学记数法的形式输出的，结尾的"E+数字"表示E之前的数字要乘以10的多少倍。比如3.14E3就是3.14×1000=3140，3.14E-3就是3.14/1000=0.00314。

Java基本类型存储在栈中，因此它们的存取速度要快于存储在堆中的对应包装类的实例对象。从Java5.0（1.5）开始，JAVA虚拟机（Java Virtual Machine）可以完成基本类型和它们对应包装类之间的自动转换。因此我们在赋值、参数传递以及数学运算的时候像使用基本类型一样使用它们的包装类，但这并不意味着你可以通过基本类型调用它们的包装类才具有的方法。另外，所有基本类型（包括void）的包装类都使用了final修饰，因此我们无法继承它们扩展新的类，也无法重写它们的任何方法。

* 基本类型与封装类型优缺点

基本类型的优势：数据存储相对简单，运算效率比较高

封装类的优势：有的容易，比如集合的元素必须是对象类型，满足了java一切皆是对象的思想

* 一些常用的转义字符：

①\r表示接受键盘输入，相当于按下了回车键；

②\n表示换行；

③\t表示制表符，相当于Table键；

④\b表示退格键，相当于Back Space键；

⑤\'表示单引号；

⑥\''表示双引号；

⑦\\表示一个斜杠\。

数据类型之间的转换

1).简单类型数据间的转换,有两种方式:自动转换和强制转换,通常发生在表达式中或方法的参数传递时。

## Java引用类型

Java有 5种引用类型（对象类型）：**类、接口、数组、枚举、注解**

引用类型：底层结构和基本类型差别较大

## JVM的内存空间：

（1）. Heap 堆空间：分配对象 new Student（）

（2）. Stack 栈空间：临时变量 Student stu

（3）.Code 代码区 ：类的定义，静态资源 Student.class

eg：Student stu = new Student（）； //new 在内存的堆空间创建对象

stu.study(); //把对象的地址赋给stu引用变量

上例实现步骤：

a.JVM加载Student.class 到Code区

b.new Student()在堆空间分配空间并创建一个Student实例

c.将此实例的地址赋值给引用stu， 栈空间

## 类型转换

具体地讲,当一个较"小"数据与一个较"大"的数据一起运算时,系统将自动将"小"数据转换成"大"数据,再进行运算。而在方法调用时,实际参数较"小",而被调用的方法的形式参数数据又较"大"时(若有匹配的,当然会直接调用匹配的方法),系统也将自动将"小"数据转换成"大"数据,再进行方法的调用,自然,对于多个同名的重载方法,会转换成最"接近"的"大"数据并进行调用。这些类型由"小"到"大"分别为 (byte，short，char)--int--long--float—double。这里我们所说的"大"与"小",并不是指占用字节的多少,而是指表示值的范围的大小。

①下面的语句可以在Java中直接通过：

byte b;int i=b; long l=b; float f=b; double d=b;

②如果低级类型为char型，向高级类型（整型）转换时，会转换为对应ASCII码值，例如

char c='c'; int i=c;

System.out.println("output:"+i);输出：output:99;

③对于byte,short,char三种类型而言，他们是平级的，因此不能相互自动转换，可以使用下述的强制类型转换。

short i=99 ; char c=(char)i; System.out.println("output:"+c);输出：output:c;

强制转换

将"大"数据转换为"小"数据时，你可以使用强制类型转换。即你必须采用下面这种语句格式： int n=(int)3.14159/2;可以想象，这种转换肯定可能会导致溢出或精度的下降。

2)表达式的数据类型自动提升, 关于类型的自动提升，注意下面的规则。

①所有的byte,short,char型的值将被提升为int型；

②如果有一个操作数是long型，计算结果是long型；

③如果有一个操作数是float型，计算结果是float型；

④如果有一个操作数是double型，计算结果是double型；

例， byte b; b=3; b=(byte)(b\*3);//必须声明byte。

3)包装类过渡类型转换

一般情况下，我们首先声明一个变量，然后生成一个对应的包装类，就可以利用包装类的各种方法进行类型转换了。例如：

①当希望把float型转换为double型时：

float f1=100.00f;

Float F1=new Float(f1);

double d1=F1.doubleValue();//F1.doubleValue()为Float类的返回double值型的方法

②当希望把double型转换为int型时：

double d1=100.00;

Double D1=new Double(d1);

int i1=D1.intValue();

简单类型的变量转换为相应的包装类，可以利用包装类的构造函数。即：Boolean(boolean value)、Character(char value)、Integer(int value)、Long(long value)、Float(float value)、Double(double value)

而在各个包装类中，总有形为××Value()的方法，来得到其对应的简单类型数据。利用这种方法，也可以实现不同数值型变量间的转换，例如，对于一个双精度实型类，intValue()可以得到其对应的整型变量，而doubleValue()可以得到其对应的双精度实型变量。

4)字符串与其它类型间的转换

其它类型向字符串的转换

①调用类的串转换方法:X.toString();

②自动转换:X+"";

③使用String的方法:String.volueOf(X);

字符串作为值,向其它类型的转换

①先转换成相应的封装器实例,再调用对应的方法转换成其它类型

例如，字符中"32.1"转换double型的值的格式为:new Float("32.1").doubleValue()。也可以用:Double.valueOf("32.1").doubleValue()

②静态parseXXX方法

String s = "1";

byte b = Byte.parseByte( s );

short t = Short.parseShort( s );

int i = Integer.parseInt( s );

long l = Long.parseLong( s );

Float f = Float.parseFloat( s );

Double d = Double.parseDouble( s );

③Character的getNumericValue(char ch)方法

5）Date类与其它数据类型的相互转换

整型和Date类之间并不存在直接的对应关系，只是你可以使用int型为分别表示年、月、日、时、分、秒，这样就在两者之间建立了一个对应关系，在作这种转换时，你可以使用Date类构造函数的三种形式：

①Date(int year, int month, int date)：以int型表示年、月、日

②Date(int year, int month, int date, int hrs, int min)：以int型表示年、月、日、时、分

③Date(int year, int month, int date, int hrs, int min, int sec)：以int型表示年、月、日、时、分、秒

在长整型和Date类之间有一个很有趣的对应关系，就是将一个时间表示为距离格林尼治标准时间1970年1月1日0时0分0秒的毫秒数。对于这种对应关系，Date类也有其相应的构造函数：Date(long date)。

获取Date类中的年、月、日、时、分、秒以及星期你可以使用Date类的getYear()、getMonth()、getDate()、getHours()、getMinutes()、getSeconds()、getDay()方法，你也可以将其理解为将Date类转换成int。

而Date类的getTime()方法可以得到我们前面所说的一个时间对应的长整型数，与包装类一样，Date类也有一个toString()方法可以将其转换为String类。

有时我们希望得到Date的特定格式，例如20020324，我们可以使用以下方法，首先在文件开始引入，

import java.text.SimpleDateFormat;

import java.util.\*;

java.util.Date date = new java.util.Date();

//如果希望得到YYYYMMDD的格式

SimpleDateFormat sy1=new SimpleDateFormat("yyyyMMDD");

String dateFormat=sy1.format(date);

//如果希望分开得到年，月，日

SimpleDateFormat sy=new SimpleDateFormat("yyyy");

SimpleDateFormat sm=new SimpleDateFormat("MM");

SimpleDateFormat sd=new SimpleDateFormat("dd");

String syear=sy.format(date);

String smon=sm.format(date);

String sday=sd.format(date);

总结：只有boolean不参与数据类型的转换

（1）.自动类型的转换：

a.常数在表数范围内是能够自动类型转换的

b.数据范围小的能够自动数据类型大的转换（注意特例）

int到float，long到float，long到double 是不会自动转换的，不然将会丢失精度

c.引用类型能够自动转换为父类的

d.基本类型和它们包装类型是能够互相转换的

（2）.强制类型转换：用圆括号括起来目标类型，置于变量前

# java初始化顺序

java类初始化顺序：

父类--静态变量、静态初始化块：按照定义的顺序初始化

子类--静态变量、静态初始化块：按照定义的顺序初始化

子类main方法

父类--变量、初始化块：按照定义的顺序初始化

父类--构造器

子类--变量、初始化块：按照定义的顺序初始化

子类--构造器

# java序列化与反序列化

1. serialVersionUID 生成规则：只有静态属性以及构造方法变化时不改变其值，其余皆改变
2. 当使用writeObject 与 readObject时：请注意写入属性的顺序和读取属性的顺序应该一致
3. 反序列化时：通过二进制流直接生成对象，并不调用构造方法；

生成类的方式：new、Class.forName("xxx")、反射、反序列化；

其中new、Class.forName("xxx")、反射都是调用构造器，而反序列化不是；

反序列化：不调用构造器，自身相当于构造器的另一种写法；

1. 如果继承自Serialiable则：

序列化函数：

private Object writeReplace()throws ObjectStreamException

private void writeObject(ObjectOutputStream out)throws IOException

注：系统总是先调用writeReplace()方法，然后调用writeObject方法；

反序列化函数：

private void readObject(ObjectInputStream in)throws IOException,ClassNotFoundException

private Object readResolve() throws ObjectStreamException

注：系统总是先调用readObject ()方法，然后调用readResolve方法；

注：如果writeReplace()方法改变了对象类型，那么将会调用真实类型的writeObject ，反序列化时调用真实类型的readObject 以及真实类型的readResolve ;

1. 如果继承自Externalizable则：

不像Serializable接口只是一个标记接口，里面的接口方法都是可选的（可实现可不实现，如果不实现则启用其自动序列化功能），而Externalizable接口不是一个标记接口，它强制你自己动手实现串行化和反串行化算法：

public interface Externalizable extends java.io.Serializable {

// 序列化

void writeExternal(ObjectOutput out) throws IOException;

/ / 反序列化

void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;

}

该接口直接继承了Serializable接口，其两个方法其实就对应了Serializable的writeObject和readObject方法，实现方式也是一模一样；

ObjectOutput和ObjectInput的使用方法和ObjectOutputStream和ObjectInputStream一模一样（readObject、readInt之类的，完全一模一样）

因此Externalizable就是强制实现版的Serializable罢了；

# java遍历

## java遍历：for、do while、while、foreach、iterator的用法和区别

相同点： 五个都可以用来遍历数组和集合

不同点：

1. 形式差别

do while的形式是

do{ xxxx }while(yyyy);

while的形式是

while(yyyy){ xxxxxx }

for的形式是

for（int i=0;i<arr.size();i++）{...}

foreach的形式是

for（int　i：arr）{...}

iterator的形式是

Iterator it = arr.iterator();

while(it.hasNext()){ object o =it.next(); ...}

1. 条件差别

for需要知道集合或数组的大小，而且需要是有序的，不然无法遍历；

foreach和iterator都不需要知道集合或数组的大小，他们都是得到集合内的每个元素然后进行处理；

1. 多态差别

for和foreach都需要先知道集合的类型，甚至是集合内元素的类型，即需要访问内部的成员，不能实现态；

iterator是一个接口类型，他不关心集合或者数组的类型，而且他还能随时修改和删除集合的元素，举个例子：

public void display（Iterator<object> it）{

while(it.hasNext()){

system.out.print(it.next()+"");

}

}

当我们需要遍历不同的集合时，我们只需要传递集合的iterator（如arr.iterator()）看懂了吧，这就 是iterator的好处，他不包含任何有关他所遍历的序列的类型信息，能够将遍历序列的操作与序列底层的结构分离。迭代器统一了对容器的访问方式。这也是接口的解耦的最好体现。

1. 用法差别

for循环一般用来处理比较简单的有序的，可预知大小的集合或数组

foreach可用于遍历任何集合或数组，而且操作简单易懂，他唯一的不好就是需要了解集合内部类型

iterator是最强大的，他可以随时修改或者删除集合内部的元素，并且是在不需要知道元素和集合的类型的情况下进行的（原因可参考第三点：多态差别），当你需要对不同的容器实现同样的遍历方式时，迭代器是最好的选择！

注：

for，while是一样的，形式上的不同

foreach，Iterator也是一样的，形式上的不同

性能肯定是用迭代器的好一些。迭代器能直接访问集合内部元素。而且有的集合没有索引，没法用for来遍历，只有用迭代器。

## java遍历集合

1. 对set的遍历

迭代遍历

Set<String> set = new HashSet<String>();

Iterator<String> it = set.iterator();

while (it.hasNext()) {

String str = it.next();

System.out.println(str);

}

for循环遍历：

for (String str : set) {

System.out.println(str);

}

优点还体现在泛型 假如 set中存放的是Object

for循环遍历：

Set<Object> set = new HashSet<Object>();

for (Object obj: set) {

if(obj instanceof Integer){

int aa= (Integer)obj;

}else if(obj instanceof String){

String aa = (String)obj

}

........

}

1. 对map的遍历

迭代遍历

Set set = map.keySet();

Iterator it = set.iterator();

while(it.hasNext()){

String str = it.next().toString();

}

Iterator it = paraMap.entrySet().iterator();

while (it.hasNext())

{

Map.Entry pairs = (Map.Entry)it.next();

System.out.println(pairs.getKey() + " = " + pairs.getValue());

}

for循环遍历：

Map<String,String> paraMap=new HashMap<>();

for(String dataKey : paraMap.keySet())

{

System.out.println(dataKey );

}

这里要注意的是,paraMap是怎么样定义的,如果是简单的Map paraMap = new HashMap();那前面的String就只能换成Object了.

对整Map的key和value都进行循环,如下:

Java代码

复制代码 代码如下:

for(Map.Entry<String, Object> entry : paraMap.entrySet())

{

System.out.println(entry.getKey()+": "+entry.getValue());

}

例子：

Map<String,String> map = new HashMap<String,String>(){};

map.put("a", "A");

map.put("b", "B");

map.put("c", "C");

//第一种

Set set = map.keySet();

Iterator it = set.iterator();

while(it.hasNext()){

    String str = it.next().toString();

}

//第二种

Set<Entry<String,String>> entryset = map.entrySet();

Iterator iter = entryset.iterator();

while(iter.hasNext()){

    Entry<String,String> entry = (Entry<String,String>)iter.next();

}

//第三种

for(Entry<String,String> entry : map.entrySet()){

    String strkey = entry.getKey();

    String strval = entry.getValue();

}

首先，使用entryset比使用keyset的效率要高。所以建议用后2种方法。

   第二和第三种的区别其实就是使用迭代器iteratior和使用一般的for语句的区别，和map本身倒没什么关系。最简单的判断方法，就是对要迭代的内容（本例是个map,如果是一个list同理）,在循环过程中，你需要进行add或者remove操作不。for语句的循环是不能做这样的操作的，只能使用迭代器。

   不过，个人认为，for语句虽然看上去代码少一点，但敲起来没iterator有感觉

# java泛型

## 泛型类：

**class** classType<T **extends** Number,V **extends** Object>

{

T t;

V v;

}

**class** classType2<T ,V **extends** Object>

{

T t;

V v;

}

**class** classType3<T **extends** Number,V >

{

T t;

V v;

}

## 泛型方法：

❶：

**public**<T> T getObject(Class<T>tc) **throws** InstantiationException,IllegalAccessException

{

T t= tc.newInstance();

// try {

// T t2= tc.newInstance();

// } catch (InstantiationException | IllegalAccessException e) {

// // **TODO** Auto-generated catch block

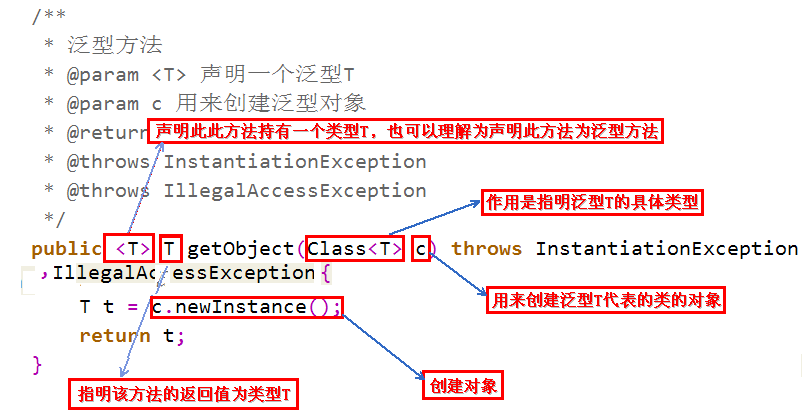
// e.printStackTrace();

// }

**return**t;

}

* 1. <T>：声明此方法有一个类型T，也可以理解为声明此方法为泛型方法
  2. Class<T>：作用是指明泛型T的具体类型，传值方法：getObject(xxxx.class);
  3. tc：用来创建泛型T代表的类的对象
  4. tc.newInstance()：创建泛型对象



❷：

**public**<T> Set<?> getT(Tt)

{

**returnnull**;

}

有通配符修饰的变量，可以遍历，不能修改，比如List<?> list,该list变量就只能够用于遍历；

# java类型之间的关系判断（instanceof、isAssignableFrom）

### instanceof

用法：

实例 instanceof 类名/接口名

作用：

判断该实例是否是（类名/接口名）或则其子类的实例；

另一个写法：

类名/接口名.class.isInstance(实例)

例：

List<String> list=new ArrayList<String>();

System.out.println(list.getClass().isInstance(new ArrayList<String>()));

System.out.println(list instanceof ArrayList);

System.out.println(ArrayList.class.isInstance(list));

System.out.println(list.getClass().isAssignableFrom(ArrayList.class));

System.out.println(ArrayList.class.isAssignableFrom(list.getClass()));

### isAssignableFrom

用法：

(类名1/接口名1).class.isAssignableFrom(类名2/接口名2)

作用：

判断（类名1/接口名1）是否是（类名2/接口名2）的父类或其自身

注：

instanceof：子-->父

isAssignableFrom：父-->子

# Java知识外延

## java.lang.Class类

### getName(),getSimpleName(),getCanonicalName ()三者之间的区别

|  |  |  |
| --- | --- | --- |
| 函数 | 非数组 | 数组 |
| getName() | 包.类名称 | [L包.类名称 |
| getSimpleName() | 类名称 | 类名称[] |
| getCanonicalName() | 包.类名称 | 包.类名称[] |

## 数据库访问部分

### javax.persistence.EntityManager;

String sql="select \* from picc\_gift\_info r where r.registerId=?registerId ";

javax.persistence.Query query = new javax.persistence.EntityManager().createNativeQuery(sql, GiftReceivedVo.class);

query.setParameter("registerId", registerId);

query.setFirstResult(start);

query.setMaxResults(limit);

List<GiftReceivedVo> list = (List<GiftReceivedVo>) query.getResultList();

1. 持久化实体persist()：

new EntityManager().persist(new GiftReceivedVo().setId(“100”))

1. 按主键查询find()：

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

1. 删除实体remove()：

new EntityManager().remove(new EntityManager().find(GiftReceivedVo.class,”100”));

1. 更新实体merge()：更新至数据库(处不处于管理模式都会更新！)

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

temp.setName(“大王哟”);

new EntityManager().merge(temp);

1. 更新实体flush()：更新至数据库（只更新处于管理模式的bean）

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

temp.setName(“大王哟”);

new EntityManager().flush (temp);

1. 从数据库中更新refresh()：与数据库同步

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

new EntityManager().refresh (temp);

1. 检测实体是否被管理contains()：好比flush()与merge()

GiftReceivedVo temp=new EntityManager().find(GiftReceivedVo.class,”100”);

Boolean b=new EntityManager().contains (temp);

b==true:处于管理模式；

b==false:未处于管理模式；

1. 分离管理的实体clear():

当处理了大量的实体后，这些实体都会处于实体管理器中，将会消耗大量的内存，使得程序运行变慢；如果要减少消耗，则可以使用clear()方法，将正在被管理的的实体从持久化内容中分离出来

new EntityManager().clear();

如果调用clear()方法，则之前所有对实体所做的任何改变都会被丢失，所以在调用clear()方法之前先调用flush()方法保存更改！

### javax.persistence.Query

关于占位符的问题：

* 如果是Query qry = new EntityManager().createNativeQuery(sql,XXXXX.class)则占位符一律是”?”;
* 如果是Query qry = new EntityManager().createQuery(sql,XXXXX.class)则占位符如果是数字，则为 “?1”；如果是字符串，则为”:XXXXXX”;

关于更新的问题：

* 如果是insert into 或者 update 则为：Query q = this.em.createNativeQuery(sql )；q.executeUpdate();

### 关于sql语句的说明

如果使用原生sql，则sql拼接语句中尽量不要出现\n等java类的转义字符，可能会出错！！！

### 注意事项

关于模糊查询：

Sql:like ?gift\_Name

query.setParameter("gift\_Name", "%"+map\_action.get("giftNameSearch").toString().trim()+"%");

关于in查询：

Sql:in (t1,t2……)

Java:使用循环赋值;