# 常量池（JDK1.8版）

相关链接：

https://blog.csdn.net/qq\_31615049/article/details/81611918

Java中的常量池分为三种类型：

-1）类文件中常量池（The Constant Pool）

-2）运行时常量池（The Run-Time Constant Pool）

-3）字符串常量池

## 类文件中常量池（Constant Pool）

---存在于Class文件中

诞生时间：编译时

内容概要：字面量（Literal）和符号引用（Symbolic Reference）

Class文件中存在一个常量池，存在的是编译期生成各种字面量和符号引用。

常量池指的是在编译期被确定，并被保存在已编译的.class文件中的一些数据。

字面量：包括文本字符串、被声明为final的常量值、基本数据类型的值。

符号引用：包括类和接口的全限定名、字段的名称和描述符、方法的名称和描述符。

常量池的每一项常量都是一个表，一共有11种各不相同的表结构数据，这每个表开始的第一位都是一个字节的标志位（取值1~12）。

Java中有8种基本类型和一个特殊类型String，这些类型为了使它们在运行过程中速度更快、更节省内存，都提供了一种常量池的概念。

## 运行时常量池

---存在于内存的元空间中

诞生时间：JVM运行时

内容概要：class文件元信息描述，**编译后的代码数据**，引用类型数据，类文件常量池。

运行时常量池：就是将编译后的类信息放入运行时的一个区域中，用来动态获取类信息。

运行时常量池是在类加载完成之后，将每个class常量池中的符号引用值转存到运行时常量池中，即每个class都有一个运行时常量池，类在解析之后，将符号引用替换成直接引用，与全局常量池中的引用值保持一致。

当java文件被编译成class文件之后，会生成上面的class常量池。而当类加载到内存之后，JVM就会将class常量池中的内容存放到运行时常量池中。

## 字符串常量池

--存在于堆中

字符串池中的内容是在类加载完成，经过验证、准备阶段之后，在堆中生成字符串对象实例，然后将该字符串对象实例的引用值存到String Pool中。（String Pool中存的是引用值而不是具体的实例对象）。

在HotSpot VM里实现的String Pool功能的是一个StringTable类，它是一个哈希表，里面存的是驻留字符串(也就是我们常说的用双引号括起来的)的引用（而不是驻留字符串实例本身）。

字符串各版本变化：

Jdk1.6及之前版本中，字符串常量池放在堆的Perm区，Perm区是一个类静态区域，主要存储一些加载类的信息、常量池、方法片段等内容，默认只有4m，一旦常量池使用大量intern是会直接产生java.lang.OutOfMemoryError:PermGen space错误的。

Jdk1.7版本中，字符串常量池从Perm区移动到堆Heap区，原因是Perm区大小了。

Jdk1.8版本中，已经取消Perm区，而新建立一个元数据区。

String类型的常量池主要使用方法有两种：

-1）直接使用双引号声明出来的String对象会直接存储在常量池中；

-2）若不是双引号声明的String对象，可使用intern()方法。

Intern()方法会从字符串常量池中查询当前字符串是否存在，若不存在则在堆内存创建当前字符串，再将其引用值存入常量池中并返回其引用值；若存在则直接返回其引用值。

字符串所在的内存区域：

String str1 = "helloworld"; //存在于常量池中

String str2 = new String("helloworld"); // 'helloworld'存在于常量池中，String对象存在于堆中

String str3 = str1 + "world"; // 有字符串引用时，在堆中建立一个String对象将引用返回

String ss = new String("abc"); //该语句会生成两个对象，一个是“abc”的实例对象，并且在StringTable中存储一个“abc”的引用值，还有一个是new出来的一个“abc”的实例对象，与上面那个是不同的实例。

# String类的使用

## String对象

大部分常量池中的字符串是在编译期确定的，除非很明确的调用String对象的intern方法返回（或创建）一个运行时常量池中的String对象。

(String s1 = "aa")中的这些String对象属于“interned String”。 String是Java对象，根据JVM规范它必须存在于java堆中，interned String也是。但JVM有个StringTable存着interned String的引用，保证内容相同的String对象不被重复intern。（这里便是编译器的优化）

字符串“+操作”：本质上是创建了StringBuilder对象进行append操作，然后将拼接后的stringBuilder对象用toString处理成String对象。

实例：

String s0 = "abc";

String s1 = "ab" + "c";

String s2 = "ab" + new String("c");

System.out.println(s0==s1); //true，说明：s0中的“abc”是字符串常量，在编译期就被确定了；而“ab”和“c”也是字符串常量，当一个字符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，因此s1也是字符串常量，是“abc”的一个引用

System.out.println(s0==s2); //false，说明：s2后半部分无法在编译期确定，所以该字符串不放入常量池中

## intern()方法

源码：public native String intern();

定义：intern()方法首先会从常量池中查找是否存在该常量值，若存在则直接返回它的引用，若不存在则在常量池中创建后返回它的引用。

大体实现结构：Java使用jni调用c++实现的StringTable的intern()方法，StringTable的intern()方法跟Java中的HashTable差不多，只是不能扩容。若放进的String Pool的String非常多，就会造成Hash冲突严重，从而导致链表会很长，进而导致调用String.intern时性能会大幅下降。

jdk1.6中StringTable是固定的，大小为1009。

Jdk1.7可设置参数指定：-XX:StringTableSize=99991

Jdk1.7版本的String#intern()说明：

详情：https://www.cnblogs.com/wxgblogs/p/5635099.html

-1）将字符串常量池从Perm区移到了Java Heap区

-2）String#intern方法时，若若存在堆中的String对象，字符串常量池会保存堆中String对象的引用，而不是重新创建对象。

实例一：（jdk1.7版本）

String s3 = new String("1") + new String("1");

s3.intern();

String s4 = "11";

System.out.println(s3 == s4); //true

实例二：（jdk1.7版本）

String s3 = new String("1") + new String("1");

String s4 = "11";

s3.intern();

System.out.println(s3 == s4); //false

实例三：（jdk1.7版本）

String s3 = new String("1") + new String("1");

String s4 = "11";

s3 = s3.intern();

System.out.println(s3 == s4); //true

## String、StringBuffer、StringBuilder区别

-1）String是字符串常量，StringBuffer和StringBuilder都是字符串变量。

-2）StringBuffer是线程安全的，而StringBuilder是非线程安全的。由于线程安全会带来额外的系统开销，所以StringBuilder的效率比StringBuffer高

## replace、replaceAll区别：

相同点：都是全部替换，如果仅替换一次可使用replaceFirst()方法。

1）replace参数是char和CharSequence，即支持字符替换和字符串替换两种。

2）replaceAll的参数是regex，即基于规则表达式的替换。如replaceAll("\\d", "\*")把字符串中所有数字字符换成\*号

## indexof()和lastIndexOf()区别

indexOf(str [, startIndex]) //从左到右执行查找

lastIndexOf(str [, startIndex]) //从右到左执行查找

实例：查找” bbb”最近的”/\*”和”\*/”

String str = "/\* aaa \*/ /\* bbb \*/ /\* ccc \*/";

int index = str.indexOf("bbb");

int indexOf = str.indexOf("\*/", index);

int lastIndexOf = str.lastIndexOf("/\*", index);

System.out.println(indexOf + " " + lastIndexOf); //输出: 18 11

# 抽象类/接口

抽象类：

JDK1.8以前，抽象类方法默认访问权限为protected；

JDK1.8以后，抽象类方法默认访问权限为default；

接口：

JDK1.8以前，接口中的方法必须为public；

JDK1.8时，接口中的方法也可以是default；

JDK1.9时，接口中的方法也可以是private；

抽象类和接口的区别：

-1）抽象类中可存在非抽象方法；接口中的方法必须是抽象方法。

-2）抽象类中可以有普通的成员变量；接口只有常量，没有变量，必须是static final类型的，必须初始化（原因：类加载时期加载static参数）。

-3）Java8中接口会有default方法，即方法可以被实现。

-4）每个子类的构造函数都在第一行默认调用super()方法，new子类时会调用父类的构造方法，这是接口没有的。详情https://www.cnblogs.com/anrainie/archive/2012/03/28/2420738.html

选择：

-1）优先选择接口！

-2）若存在某个具体实现的方法（如BaseDao类注入了SessionFactory），只要方法没有全部实现，则设置成抽象类。

# 异常

try-catch-finally规则：

-1）try之后添加catch块或finally块。

-2）一个 try 块可能有多个 catch 块，但只会匹配一个catch块。若匹配一个匹配块，就不会再执行其它的catch代码块了。

-3）当try块和finally块都有return返回时，该方法返回的是finally的。但不建议在finally块写return语句。

-4）当try块中有System.exit(0)，表示整个虚拟机里的内容都释放，JVM停止工作，程序正常退出。此时finally中的语句不会执行。

try块和finally块的return语句描述：

try中执行到return时，先把这个值存起来，再开辟一块内存存这个值，然后去执行finally，finally执行之后，回去执行之前没执行完的return语句，将值返回。

throw与throws区别：

-1）throw是抛出一个异常对象。（如：throw new Exception()）

-2）throws是方法上声明可能抛出的异常们，表示抛出的异常由调用该方法的调用者处理。（如：public void saveAll() throws Exception1,Exception2{ … }）

实例：

public void doA(int a) throws Exception1, Exception3 {

try {

...//省略

}catch(Exception2 e2) {

System.out.println("捕获并处理Exception2异常！");

}

if(a!=b) throw new Exception3();

}

说明：

1）如果产生Exception1异常，则捕获之后再抛出，由该方法的调用者去处理。

2）如果产生Exception2异常，则该方法自己处理了(即System.out.println("捕获并处理Exception2异常！");)。所以该方法就不会再向外抛出Exception2异常，public void doA(int a) throws Exception1,Exception3里面的Exception2也就不用写了。

3）Exception3异常是该方法的某逻辑出错，程序员自己做了处理，在该段逻辑错误的情况下抛出Exception3异常，则该方法的调用者也要处理此异常。

# 内部类

4种内部类：成员内部类、静态内部类、局部内部类和匿名内部类

为什么需要内部类：

-1）高内聚，把只和这个类相关的类型放到这个类的内部。例如链表、接口回调。

-2）设计原则中的“组合优先于继承”：若继承仅仅是为了得到父类的功能，则此时应该完全抛弃继承，而改用组合实现。

实例：比较器类MyComparator、HashMap集合中的节点Entry

## 成员内部类

格式：

public class Outer {

class Inner {...} //成员内部类

}

创建内部类对象：Outer.Inner inner = (new Outer()).new Inner();

访问外部类非静态变量：外部类名.this.变量名

访问外部类静态变量：外部类名.变量名

说明：

相当于外部类的一个成员方法，访问权限与成员方法一样，能能访问外部类的所有变量和方法，包括静态和非静态，私有和非私有。

注：成员内部类没有静态变量和静态方法！！

注：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类。编译完成后出现outer.class和outer$inner.class两类。Out.java编译后会生成两个class文件，分别是Out.class和Out$Inner.class。

好处：用内部类定义在外部类中不可访问的属性。这样就在外部类中实现比外部类的private还小的权限。

外部类访问内部类：

注：成员内部类没有静态变量和静态方法！

--外部类的非静态方法访问成员内部类的非静态方法：

public void outer\_f3() {

new Inner().inner\_f1();

}

--外部类的静态方法访问成员内部类的非静态方法：

public static void outer\_f4() {

new Outer().new Inner().inner\_f1();

}

## 静态内部类

格式：

public class Outer {

static class Inner{...} //静态内部类

}

创建静态内部类对象：Outer.Inner in = new Outer.Inner();

访问静态内部类的静态成员：String staticStr = Inner.staticStr;

访问静态内部类的非静态成员：String str = new Outer.Inner().str;

说明：

用static修饰。访问权限与外部类的静态方法一样，只能访问外部类的静态变量和静态方法。

注：Outer.java文件编译后会生成两个class文件，分别为Outer.class和Outer$Inner.class。

应用场景：

1）当类与接口（或接口与接口）发生方法名冲突时，必须使用内部类来实现。用接口不能完全地实现多继承，用接口配合内部类才能实现真正的多继承。

2）Java集合类HashMap内部就有一个静态内部类Entry。Entry用来存放元素，HashMap的主干为一个Entry数组。

静态内部类与成员内部类的区别：

-1）内部：静态内部类中可以定义静态成员，而成员内部类中不允许定义静态成员。

-2）外部：静态内部类只能访问外部类的静态变量和静态方法；而成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和非私有。

访问：

class OuterOne {

public class InsideOne {...} //成员内部类

public stitic class InsideTwo{...} //静态内部类

public static void main(String[] args) {

OuterOne.InsideOne one = new OuterOne().new InsideOne(); //成员内部类访问

OuterOne.InsideTwo two = new OuterOne.InsideTwo(); //静态内部类访问

}

}

## 局部内部类

即在方法中定义的内部类，与局部变量类似，不能用public或private修饰。

局部内部类中不可定义静态变量。

定义在实例方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的局部类只能访问外部类的静态变量和方法。

局部内部类只能访问final类型的局部变量。

格式：

public class Outer { //外部类

public void f() { //方法

class Inner{...} //局部内部类：定义在方法内

}

}

## 匿名内部类

格式：new 接口()|父类构造器(参数列表) { ...//匿名内部类的类实现 }

是一种特殊的局部内部类，使用匿名内部类时，必须继承一个类或实现一个接口。（如创建对象：Date d = new Date(){...}）

由于构造器的名字必须与类名相同，而匿名类没有类名，所以匿名类不能有构造器。

匿名内部类中不能含有静态成员变量和静态方法。final修饰的局部变量才可被匿名内部类使用。

匿名内部类不能是public、protected、private、static

应用场景：

匿名内部类使用广泛，比如我们常用的绑定监听的时候。

# 反射

## 反射机制中的类：

反射机制中的类包括：

java.lang.Class //类

java.lang.reflect.Constructor //构造器

java.lang.reflect.Field //属性

java.lang.reflect.Method //方法

java.lang.reflect.Modifier //修饰符

## 常用方法

### 获取类、父类和接口

类形式为Class，泛型形式为Class<T>

1）getName()：返回类的名字，包括包名。比如：com.test.User

2）getSimpleName()：返回类名称简称，不包括包名。比如：User

3）newInstance()：快速地创建一个类的实例

4）getSuperclass()：获取某类的父类

5）getGenericSuperclass()：获得带有泛型的父类！！

6）getInterfaces()：获取某类实现的接口

获取Class方式：

Class c1 = Class.forName("Employee"); //方法一

Class c2 = Employee.class; //方法二

Class c3 = (new Employee()).getClass(); //方法三，c3是运行时类

### getSuperclass()和getGenericSuperclass()

getSuperclass()和getGenericSuperclass()区别：

1）getSuperclass() : 获取某类的父类。由于编译擦除，没有显示泛型参数

2）getGenericSuperclass : 获得带有泛型的父类！！

测试实例：

在class Student extends Person<Student>类中：

Class clazz = getClass(); //获取本类

clazz.getSuperclass() //获得该类的父类(class com.yjy.test.Person)

Type type = clazz.getGenericSuperclass(); //得到com.yjy.test.Person<com.yjy.test.Student>

ParameterizedType p = (ParameterizedType) type;

Class<T> c = (Class<T>) p.getActualTypeArguments()[0]; //得到class com.yjy.test.Student

c.getName() //com.yjy.test.Student

### 获取属性

getFields()和getDeclaredFields()区别：

-1）getFields()：访问“类及父类”中声明为public的属性，但不能访问私有属性。

-2）getDeclaredFields()：访问“本类”中所有的属性（与private/protected/public无关），但不能访问父类的属性。

获取特定属性（get/set）：

Class clazz = Class.forName("com.yjy.domain.User"); //获取类

Object o = clazz.newInstance(); //类实例化

Field[] fields = clazz.getFields(); //获取所有“类及父类”的public属性

Field idF = clazz.getDeclaredField("id"); //获取指定属性

idF.setAccessible(true); //使用反射机制可以打破封装性，导致了java对象的属性不安全。

idF.set(o, "110"); //赋值set

获取所有属性：

Field[] fields = c.getDeclaredFields(); //获取所有的public和非public属性

for(Field field : fields){

String fieldName = field.getName(); //变量名称，例如username

Modifier.toString(field.getModifiers()) // 修饰符，例如public/public static等等

Object fieldType = field.getType(); //变量类型

field.getType().getSimpleName() // 类型,例如String

}

field.isAnnotationPresent(MyFilter.class) //该字段field是否有MyFilter注解

### 获取方法

getMethods()和getDeclaredMethods()区别：

-1）getMethods()：访问“类及父类”中声明为public的方法。但不能访问私有方法。

2）getDeclaredMethods()：访问“本类”中所有的方法，与private，protected，public无关。但不能访问父类的方法。

### 获取构造器

getConstructors()和getDeclaredConstructors()区别：

-1）getConstructors()：访问“类及父类”中声明为public的构造函数。

-2）getDeclaredConstructors()：访问“本类”中所有的构造函数,与public,private,protect无关。

获取“类及父类”中声明为public的公有构造器：

Constructor[] constructors = aClass.getConstructors();

获取“类及父类”中存在参数为String类型的构造器：

Constructor constructor = aClass.getConstructor(new Class[]{String.class});

## 私有变量和私有方法

从对象的外部访问私有变量以及方法是不允许的，但是Java反射机制可以做到这一点。

-1）获取私有变量（2种）：

aClass.getDeclaredField(String name)方法

aClass.getDeclaredFields()方法。

-2）获取私有方法（2种）：

aClass.getDeclaredMethod(String name, Class[] parameterTypes)方法

aClass.getDeclaredMethods() 方法。

# 泛型

## 应用场景

1）声明一个需要被参数化（Parameterizable）的类/接口

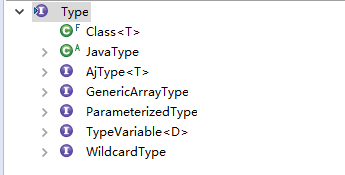
2）使用一个参数化类

java.util.List接口就是典型的例子。用泛型机制创建一个标明存储的是String类型list，这样比你创建一个Object的list要更好。

## Type类型

java.lang.reflect.Type –java语言中所有类型的公共接口

Type是Java中所有类型的公共高级接口。它们包括原始类型、参数化类型、数组类型、类型变量和基本类型



### Type的4个直接子接口

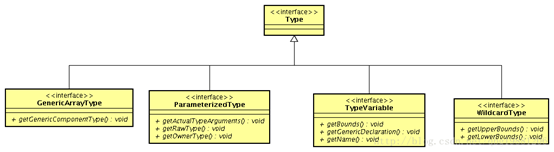
Type的4种类型的接口：

-1）ParameterizedType：参数化类型（即泛型），比如Collection

-2）GenericArrayType：数组类型

-3）TypeVariable：类型变量，是各种类型变量的公共父接口

-4）WildcardType（译为“通配符”）：代表一种通配符类型表达式，比如?,? extends Serializable等。



? extends XX描述：

XX 类是用来限定通配符的上界，XX 类是能匹配的最顶层的类，它只能匹配 XX 类以及 XX 类的子类。

? super XX描述：

XX 类是用来限定通配符的下界，XX 类是能匹配的最底层的类，它只能匹配 XX 类以及 XX 类的超类

? extends ArrayList，这样extends后面是?的上边界，这个上边界是ParameterizedType类型

? extends E，这样extends后面是?的上边界，这个上边界是TypeVariable类型

? extends E[]，这样extends后面是?的上边界，这个上边界是GenericArrayType类型

? extends Number，这样extends后面是?的上边界，这个上边界是Class类型

## 测试实例

实例：

在BaseDaoImpl抽象类中

Class<T> clazz;

public BaseDaoImpl() {

ParameterizedType pt = (ParameterizedType)getClass().getGenericSuperclass();

clazz = (Class<T>)pt.getActualTypeArguments()[0];

}

实例：

Type type = getClass().getGenericSuperclass(); //返回com.test.Person<Student>

Type type2 = ((ParameterizedType) type).getActualTypeArguments()[0]; //获取类型:Student

1）getClass().getGenericSuperclass()

获得带有泛型的父类！！

2）ParameterizedType ：参数化类型（即泛型）

被参数化：supperclass:Dao<Student>

未被参数化：superclass:Dao<T>

if(superclass instanceof ParameterizedType) { //判断是否支持泛型

// 当Dao<T>这个泛型类被参数化之后执行...

Type[] params = ((ParameterizedType) type).getActualTypeArguments();

clazz = (Class<T>) params[0];

}

3）getActualTypeArguments()

返回一个Type对象数组，这个数组代表着这个Type声明中泛型参数的实际类型

### 获取方法中的泛型返回类型：

-1）定义这个类中的方法，其返回类型是一个泛型类型

public class MyClass {

protected List<String> stringList = ...;

public List<String> getStringList(){

return this.stringList;

}

}

-2）这个类中的方法返回类型是一个泛型类型

Method method = MyClass.class.getMethod("getStringList", null);

Type returnType = method.getGenericReturnType();

if(returnType instanceof ParameterizedType){

ParameterizedType type = (ParameterizedType) returnType;

Type[] typeArguments = type.getActualTypeArguments();

for(Type typeArgument : typeArguments){

Class typeArgClass = (Class) typeArgument;

System.out.println("typeArgClass = " + typeArgClass);

}

}

结果：打印“typeArgClass = java.lang.String”。

### 获取方法中的泛型参数类型：

-1）定义方法的参数是一个被参数化的List

public class MyClass {

protected List<String> stringList = ...;

public void setStringList(List<String> list){

this.stringList = list;

}

}

-2）获取该方法的泛型参数

method = Myclass.class.getMethod("setStringList", List.class);

Type[] genericParameterTypes = method.getGenericParameterTypes();

for(Type genericParameterType : genericParameterTypes){

if(genericParameterType instanceof ParameterizedType){

ParameterizedType aType = (ParameterizedType) genericParameterType;

Type[] parameterArgTypes = aType.getActualTypeArguments();

for(Type parameterArgType : parameterArgTypes){

Class parameterArgClass = (Class) parameterArgType;

System.out.println("parameterArgClass = " + parameterArgClass);

}

}

}

结果：打印“parameterArgType = java.lang.String”。

### 获取变量中的泛型参数类型：

-1）定义泛型变量

public class MyClass {

public List<String> stringList = ...;

}

-2）获取反射参数

Field field = MyClass.class.getField("stringList");

Type genericFieldType = field.getGenericType();

if(genericFieldType instanceof ParameterizedType){

ParameterizedType aType = (ParameterizedType) genericFieldType;

Type[] fieldArgTypes = aType.getActualTypeArguments();

for(Type fieldArgType : fieldArgTypes){

Class fieldArgClass = (Class) fieldArgType;

System.out.println("fieldArgClass = " + fieldArgClass);

}

}

结果：打印“fieldArgClass = java.lang.String”。

# 注解

获取作用于类的注解：

MyAnnotation test = AnnotationTest.class.getAnnotation(MyAnnotation.class);

test.value();

获取作用于方法的注解：

Method m = AnnotationTest.class.getDeclaredMethod("method1",null);

MethodAnnotation ma = m.getAnnotation(MethodAnnotation.class);

ma.value();

## 元注解

4种类型：@Document、@Target、@Inherited、@Retention

-1）@Document：指明该元素可以被javadoc工具文档化

-2）@Target：指明注解元素的范围

如：@Target({ElementType.METHOD, ElementType.TYPE})

该作用域有：

public enum ElementType {

TYPE, //类、接口、枚举

FIELD, //字段

METHOD, //方法

PARAMETER, //参数

CONSTRUCTOR, //构造器

LOCAL\_VARIABLE, //局部变量

ANNOTATION\_TYPE, //注解类

PACKAGE //包

}

-3）@Inherited：指明该元素被自动继承

-4）@Retention：指明生命周期

如：@Retention(RetentionPolicy.RUNTIME)

生命周期有：（3种）

public enum RetentionPolicy {

SOURCE, //只在源码显示，编译时会丢弃, 在class字节码文件中不包含

CLASS, //默认，编译时会记录到class中，运行时忽略, 运行时无法获得

RUNTIME //注解会clas字节码文件中存在，在运行时可以通过反射获取到

}

## 自定义注解

1）创建自定义注解时

以@Interface标注（和创建一个接口相似）

2）java内建注解：

@Override --重写父类中的方法

@Deprecated --编译器不推荐使用该方法

@SuppressWarnings --编译器忽略特定的警告信息

## 测试实例

### 类注解测试：

-1）定义类注解

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.TYPE)

public @interface MyAnnotation {

public String name();

public String value();

}

-2）使用类注解

@MyAnnotation(name="someName", value = "Hello World")

public class TheClass {...}

-3）访问类注解

（方式一）：

Annotation[] annotations = TheClass.class.getAnnotations();

for(Annotation annotation : annotations){

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("name: " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

}

（方式二）推荐使用：

Annotation annotation = TheClass.class.getAnnotation(MyAnnotation.class);

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("name: " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

### 方法注解测试：

-1）使用方法注解

public class TheClass {

@MyAnnotation(name="someName", value = "Hello World")

public void doSomething(){}

}

-2）访问方法注解

Method method = ... //获取方法对象

Annotation annotation = method.getAnnotation(MyAnnotation.class);

…

### 参数注解测试：

-1）使用方法注解

public class TheClass {

public static void doSomethingElse(@MyAnnotation(name="aName", value="aValue") String parameter){

}

}

-2）通过Method对象来访问方法参数注解

Method method = ... //获取方法对象

Annotation[][] parameterAnnotations = method.getParameterAnnotations();

Class[] parameterTypes = method.getParameterTypes();

int i=0;

for(Annotation[] annotations : parameterAnnotations){

Class parameterType = parameterTypes[i++];

for(Annotation annotation : annotations){

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("param: " + parameterType.getName());

System.out.println("name : " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

}

}

### 变量注解测试：

-1）使用变量注解

public class TheClass {

@MyAnnotation(name="someName", value = "Hello World")

public String myField = null;

}

-2）访问变量注解(方法一)

Field field = ... //获取方法对象

Annotation[] annotations = field.getDeclaredAnnotations();

for(Annotation annotation : annotations){

if(annotation instanceof MyAnnotation){

MyAnnotation myAnnotation = (MyAnnotation) annotation;

System.out.println("name: " + myAnnotation.name());

System.out.println("value: " + myAnnotation.value());

}

}

访问变量注解(方法二)

Annotation annotation = field.getAnnotation(MyAnnotation.class);

for(){...} --同上

# 比较器Comparable和Comparator

它们都是用来做对象比较的。

## Comparable介绍

一个实现了Comparable接口的类，可以让该类的两个对象进行比较。

步骤：继承Comparable接口，并重写compareTo()方法。

实例：

class User implements Comparable<User> {

... //省略

@Override

public int compareTo(User tv) {

if (this.getSize() > tv.getSize()) return 1;

else if (this.getSize() < tv.getSize()) return -1;

else return 0;

}

}

Main方法：

ArrayList<User> al = new ArrayList<User>();

al.add(tv1);

al.add(tv2);

Collections.sort(al);

## Comparator介绍

该接口代表一个比较器，比较器具有可比性。

该方法主要是不用修改原来的类！！

在一些情况下，你不希望修改原有的类的结构，但让它可以比较，则可以使用Comparator接口。它针对其中特定的属性/字段来进行比较。

步骤：创建一个继承了Comparator接口的类，并重写compare()方法。

多数文章提到Comparator排序，是因为javase数组工具类和集合工具类中提供了sort方法：

Arrays.sort(T[], Comparator<? super T> c);

Collections.sort(List<T> list, Comparator<? super T> c);

使用场景：

-1）排序，比较两个对象排序问题，两个对象比较的结果有三种：大于，等于，小于

-2）分组，比较两个对象是否同属一组，两个对象比较的结果只有两种：等于(两个对象属于同一组)，不等于(两个对象属于不同组)

实例：

class User {... // 省略}

class SizeComparator implements Comparator<User> { //比较器

public static final SizeComparator INSTANCE = new SizeComparator();

@Override

public int compare(User tv1, User tv2) {

int tv1Size = tv1.getSize();

int tv2Size = tv2.getSize();

if (tv1Size > tv2Size) return 1;

else if (tv1Size < tv2Size) return -1;

else return 0;

}

}

Main方法：

ArrayList<User> list = new ArrayList<User>();

list.add(tv1);

list.add(tv2);

Collections.sort(al, SizeComparator.INSTANCE);

## 总结

同一个类的两个对象之间要想比较，对应的类就要实现Comparable接口，并实现compareTo()方法。使用使用Collections.sort()方法做比较。

你不希望修改一个原有的类，但是你还想让他可以比较，Comparator接口可以实现这样的功能。通过实现Comparator接口同样要重写一个方法：compare()。

# BeanUtils/Properties工具类

org.apache.commons.beanutils.BeanUtils.copyProperties(user, user2);

org.apache.commons.beanutils.PropertyUtils.copyProperties(user1, user2);

org.springframework.beans.BeanUtils.copyProperties(attach, ebankApply4);

--1）get方法

BeanUtils.getProperty(userBean,"username");

BeanUtils.getProperty(studentBean, "class.name"); //获取学生对应的班级名称

BeanUtils.getProperty(orderBean, "customers[1].name"); //获取list的属性

--2）set方法

BeanUtils.setProperty(p, "name", "张三");

--3）populate()方法

在MVC框架中，经常要从request，resultSet等对象取出值来赋入bean中。

若不想写：String a = request.getParameter(“a”); bean.setA(a); String b = …

不妨写一个Binder：

MyBean bean = ...;

HashMap map = new HashMap();

Enumeration names = request.getParameterNames();

while (names.hasMoreElements()) {

String name = (String) names.nextElement();

map.put(name, request.getParameterValues(name));

}

//支持java.sql.Date和java.sql.Timestamp两种类型

BeanUtils.populate(bean, map);

Servlet中有这样的使用：

Person person = new Person(); //这是一个JavaBean

BeanUtils.populateBean(person, request);

---->populateBean(person, request.getParameterMap()); //先将request内容转为Map类型

---->BeanUtils.populate(info, propertyMap);

-4）PropertyDescriptor类

作用：操作访问JavaBean的属性。

# java.beans.PropertyDescriptor.PropertyDescriptor(String, Class<?>)

-1）操作一个属性(get/set使用)

Person obj = new Person();

PropertyDescriptor pd = new PropertyDescriptor("username", Person.class); //得到对象的属性

Method method = pd.getWriterMethod(); //获取setter方法

method.invoke(obj, "testSetMethod");

Method method = pd.getReaderMethod(); //获取getter方法

Object value = method.invoke(obj);

-2）操作多个属性（常用！！）

Person bean = new Person();

PropertyDescriptor[] propertyDescriptors = PropertyUtils.getPropertyDescriptors(bean);

for (PropertyDescriptor propertyDescriptor : propertyDescriptors) {

String propertyName = propertyDescriptor.getName();

Object value = PropertyUtils.getProperty(bean, propertyName);

}

# 配置文件处理类(Properties)

--Properties类是HashTable的子类，该类用于处理配置文件。

实例：

Properties property = new Properties();

InputStream input = ClassLoader.getSystemResourceAsStream("db.properties");

property.load(in);

String user = property. getProperties(“user”); //使用一

Set<String> propertyNames = property.stringPropertyNames(); //遍历一

for(String name : propertyNames) {

property.getProperty(name);

}

Map<String, String> map = new HashMap<String, String>((Map)property); //遍历二

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

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

}

# Json部分

相关链接： http://blog.csdn.net/a9529lty/article/details/50621739

-1）JSON.parse(str) –将字符串转换成json对象

var data='{"name":"goatling"}';

JSON.parse(data); -- 结果是：​name:"goatling"

-2）JSON.stringify(jsonObj); --将json对象转换成字符串

var data={name:'goatling'};

JSON.stringify(data); --结果是：'{"name":"goatling"}'

或者JSON.stringify(str, null, 4) //使用四个空格缩进

-3）去除某些字段

List<HbDrugDoseMid> hbDrugDoseMids = hbSpecialMid.getHbDrugDoseMids();

List<HbResistanceMid> hbResistanceMids = hbSpecialMid.getHbResistanceMids();

JsonConfig jc=new JsonConfig();

jc.setExcludes(new String[]{"hbSpecialMid"}); //去除某些字段

JSONArray object1 = JSONArray.fromObject(hbDrugDoseMids, jc);

JSONArray object2 = JSONArray.fromObject(hbResistanceMids, jc);

JSONArray jsonThree = new JSONArray();

jsonThree.add(object1);

jsonThree.add(object2);

# 枚举(enum)

相关链接：https://www.cnblogs.com/hyl8218/p/5088287.html

相关概念：

原理：enum的语法结构尽管和class的语法不一样，但是经过编译器编译之后产生的是一个class文件。该class文件经过反编译可以看到实际上是生成了一个类，该类继承了java.lang.Enum<E>。

关键字enum，隐含了所创建的类型都是java.lang.Enum类（抽象类）的子类。

枚举类型符合通用模式 Class Enum<E extends Enum<E>>，而E表示枚举类型的名称。

枚举类型的每一个值都将映射到 protected Enum(String name, int ordinal) 构造函数中，在这里，每个值的名称都被转换成一个字符串，并且序数设置表示了此设置被创建的顺序。

常量定义：

-1）使用接口

public interface IWeek {

String MON = "Mon";

String TUE = "Tue";

String WED = "Wed";

String THU = "Thu";

String FRI = "Fri";

String SAT = "Sat";

String SUN = "Sun";

}

-2）使用枚举：

public enum WeekEnum {

MON, TUE, WED, THU, FRI, SAT, SUN;

}

--分析：这段代码实际上调用了7次Enum(String name, int ordinal)

实例：给enum自定义属性和方法

-1）自定义枚举

public enum WeekEnum {

MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6),SUN(0);

private int value;

private WeekEnum(int value) {

this.value = value;

}

public int getValue() {

return value;

}

}

-2）测试：

System.out.println("value=" + WeekEnum.SUN.getValue()); //输出：value=0

-3）EnumSet的使用

EnumSet<WeekEnum> weekSet = EnumSet.allOf(WeekEnum.class);

for (WeekEnum dayTest : weekSet) {

System.out.println("value:" + dayTest.getValue());

}