一.java的反射：

1.1 类加载和类初始化：

一般来说，类加载和类初始化是一起的，因为类加载一般情况下是在运行过程中类被使用的时候才会加载并且初始化的：

* new User(); //实例化类，形成一个对象
* User.var; //访问类的静态变量
* User.getUser(); //使用静态方法。
* Class.forName(“User”);
* 初始化User的子类
* 包含main方法的一定会初始化。

那么什么时候类加载但是类不初始化呢？

* User.var 也就是上面的访问类的静态变量，如果这个var 是被static final 修饰的就只加载类不初始化，如果是static那么必须初始化。

这说明了，加载过程中常量已经被分配到常量池了。

* clazz.getClassLoader().**loadClass**("com.mysql.jdbc.Driver");

反射中用loadClass加载类，这样不会初始化。

* Class clazz=User.class 也只会加载类而不会初始化。

注意：静态内部类实际上和外部类没有任何关系，在加载或者初始化的时候都不会对它有作用，只有在它自己使用的时候才会加载并初始化。

类加载：创建一个Class对象。

类初始化指的是，执行静态代码块并且给静态变量赋值。只加载一次。

1.2 什么是反射？

反射，指的是反射可以动态的获取类的成员，可以在不改动源代码的情况下修改类，反射技术主要有两个特点：

1. 只要知道类的全名字，就可以直接加载这个类，并得到这个类的Class对象。
2. 通过得到的Class对象，可以得到关于这个类的全部信息，甚至隐藏的信息。

1.3 反射的内容

第一步：通过Class clazz=Class.forName(类名)，加载这个类并对其进行初始化。

第二步：通过 clazz的各种静态方法，就可以完成各种操作。

二. java泛型编程

## 1.1．泛型的反射实例

案例，设置通用方法，会用到反射泛型！。

步骤：

1. 案例分析/ 实现
2. 涉及知识点
3. 优化 /用反射泛型

案例：

要从两个不同的表中查询数据，主键相同，那么通过主键查询数据，是否可以有一个通用功能呢？

反射泛型涉及的api：

Type： Type是java编程语言中所有类型的公共高级接口。包括：基本类型，引用类型，

**参数化类型**

**ParameterizedType 参数化类型的表示**

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

泛型集合： list

集合元素定义： String

参数化类型：ArrayList<String> ParameterizedType.

//基类：

/\*\*  
 \* 所有dao的公用方法，都在这里实现  
 \* Created by horse on 2017/4/22.  
 \*/  
**public class BaseDao**<T> {  
 //构造函数：1.获取当前运行类的参数话类型，2.获取参数化类型中实际类型的定义（class）  
  
 **private Class** clazz; //保存当前运行类中实际运行类的实际类型。  
  
 **private String** table; //保存当前运行类的实际名字  
  
 **public** BaseDao(){  
 Type type=**this**.getClass().getGenericSuperclass(); //当前运行类的父类。其实就是参数化类型  
 ParameterizedType pt= (ParameterizedType) type;  
  
 //获取参数化类型中，实际类型的定义  
 Type[] types= pt.getActualTypeArguments();  
 clazz=(**Class**)types[0];  
 table=clazz.getSimpleName();  
  
 **System**.out.println(table);  
 }  
 **public** T finById(**int** id){  
 */\*  
 \* 1.知道封装的对象的类型! 关键是如何知道对象。  
 \* 得到当前运行类继承的父类， BaseDao<Accout> 即得到参数化类型 ParameterizedType  
 \* 得到Account.class;  
 \*/* **String** sql="select \* from "+table + " where id= ?";  
 **ComboPooledDataSource** dataSource=**new** ComboPooledDataSource();  
 **QueryRunner** queryRunner=**new** QueryRunner(dataSource);  
 **try** {  
 **return** queryRunner.query(dataSource.getConnection(),sql,**new** BeanHandler<T>(clazz),id);  
 } **catch** (**SQLException** e) {  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
  
 */\*  
 \* 查询全部  
 \* \*/* **public** List<T> getAll(){  
 **String** sql="select \* from "+table;  
 **ComboPooledDataSource** dataSource=**new** ComboPooledDataSource();  
 **QueryRunner** queryRunner=**new** QueryRunner(dataSource);  
 **try** {  
 **return** queryRunner.query(dataSource.getConnection(),sql,**new** BeanListHandler<T>(clazz));  
 } **catch** (**SQLException** e) {  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
}

//普通类实现基类的代码：

**public class AdminDao extends BaseDao**<**Admin**> {  
 //根据主键查询  
  
 @Override  
 **public Admin** finById(**int** id) {  
 **return super**.finById(id);  
 }  
}

测试代码：

**AdminDao** adminDao=**new** AdminDao();  
**Admin** admin=adminDao.finById(1);

这几行代码的流程是：

创立一个基类，让一个实用类继承基类 并在此时就定义好 类型

**public class AdminDao extends BaseDao**<**Admin**>

这一点很重要

如果没有在这里直接定义Admin的话，那么会报错。

然后，就可以直接通过

|  |
| --- |
| Type type=**this**.getClass().getGenericSuperclass(); //当前运行类的父类。其实就是参数化类型  ParameterizedType pt= (ParameterizedType) type;   //获取参数化类型中，实际类型的定义 Type[] types= pt.getActualTypeArguments(); clazz=(**Class**)types[0]; |

clazz最后的结果是：Admin。 也就是这样最后得到的是参数化类型。

然后，我们就获取到了实际需要的类，

确立了实际的类型，就不需要那么多其他东西了。

**1.2java中泛型原理以及相关概念**

**1.2.1 原理：**

在java中程序的工作原理是：

运行

编译

运行后

.class编译后文件

.java源文件

泛型的作用主要是在编译这个阶段，在这个阶段，编译器会首先进行类型检查，然后对源码进行类型擦除，并且在类型参数出现的地方插入强制转换的相关指令用来检查错误（可能需要生成一些桥接方法（bridge method））。

**类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。同时去掉出现的类型声明，即去掉<>的内容。比如T get()方法声明就变成了Object get()；List<String>就变成了List。接下来就可能需要生成一些桥接方法（bridge method）。**

以下面代码为例子来实际讲解：

|  |
| --- |
| List<String > list=new ArrayList<String>();  list.add("abc");  list.add(1); //编译时期报错 |

当编译的时候，编译器先进行类型检查，得到String这个类型，然后进行类型擦除，用强制类型转换检查错误。就得到list.add(1);报错。 编译不通过。

那么对于下面这段代码再来分析：

|  |
| --- |
| List<String > list=new ArrayList<String>();  list.add("abc"); |

通过上面的分析，我们可以得到，这一行代码是可以编译通过的，那么编译后的代码如下：

|  |
| --- |
| List list=new ArrayList ();  list.add("abc");  <String>被擦除了，而且T add();方法变成了 Object add(); |

**1.2.2 使用泛型的注意事项**

* 泛型的类型参数不能用在Java异常处理的catch语句中。因为异常处理是由JVM在运行时刻来进行的。由于类型信息被擦除，JVM是无法区分两个异常类型MyException<String>和MyException<Integer>的。对于JVM来说，它们都是 MyException类型的。也就无法执行与异常对应的catch语句

例如：

|  |
| --- |
| try｛  Aop<String> aop=new Aop<String>();  ｝catch(Exception e){  String str; //报错！  } |

* ***泛型在静态方法和静态类中的问题***

泛型类中的静态方法和静态变量不可以使用***泛型类所声明的泛型类型参数***

**[java]** [view plain](http://blog.csdn.net/sunxianghuang/article/details/51982979) [copy](http://blog.csdn.net/sunxianghuang/article/details/51982979)

* **public** **class** Test2<T> {
* **public** **static** T one;   //编译错误
* **public** **static**  T show(T one){ //编译错误
* **return** **null**;
* }
* }

因为泛型类中的泛型参数的实例化是在定义***泛型类型对象（例如ArrayList<Integer>）***的时候指定的，而静态变量和静态方法不需要使用对象来调用。对象都没有创建，如何确定这个泛型参数是何种类型，所以当然是错误的。

泛型参数不能为基本类型。

* **对于不同传入的类型实参，生成的相应对象实例的类型是不是一样的呢？**

[复制代码](javascript:void(0);)

1 public class GenericTest {

2

3 public static void main(String[] args) {

4

5 Box<String> name = new Box<String>("corn");

6 Box<Integer> age = new Box<Integer>(712);

7

8 System.out.println("name class:" + name.getClass()); // com.qqyumidi.Box

9 System.out.println("age class:" + age.getClass()); // com.qqyumidi.Box

10 System.out.println(name.getClass() == age.getClass()); // true

11

12 }

13

14 }

[复制代码](javascript:void(0);)

由此，我们发现，在使用泛型类时，虽然传入了不同的泛型实参，但并没有真正意义上生成不同的类型，传入不同泛型实参的泛型类在内存上只有一个，即还是原来的最基本的类型（本实例中为Box），当然，在逻辑上我们可以理解成多个不同的泛型类型。

究其原因，在于Java中的泛型这一概念提出的目的，导致其只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦出，也就是说，成功编译过后的class文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

**对此总结成一句话：泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。**

## 1.3 泛型方法/泛型类/泛型接口

作用：

设计公用的类、方法，对公用的业务实现进行抽取，使程序更灵活。

泛型方法

|  |
| --- |
| public <T,K> T save(T dept ,K k){ //这样就是定义泛型。  return null;  }    //测试方法  @Test  public void test() throws Exception{  save(1.0f,1); //泛型类型确定的时间：什么时候使用泛型方法，什么时候可以确定泛型类型。  } |

泛型类/泛型接口

|  |
| --- |
| //泛型类：在创建泛型对象的时候，确定类型。或者被继承时。  GernericDmeo<String> g=new GernericDmeo<String>();  g.save("", 1);  }  // 泛型接口： 实现泛型接口的类也是抽象，那么类型在具体的实现类中确定或创建泛型类的时候确定。  记住：在实例化的时候一定要明确实际类型 |

## 1.4 泛型的关键字（了解下就好了）

泛型中 ？ extends super

关键字： ？

|  |
| --- |
| public void save(List<?> list){  //只能获取，迭代list，不能编辑list  }    public void testGeneric() throws Exception{  //? 可以接受任何泛型集合，但是不能编辑集合值，所以一般在方法参数中使用。    List<?> list=new ArrayList<String>();  } |

关键字 extends

|  |
| --- |
| /\*  \* list集合只能处理Double，float，intefer等类型  \* 限定元素范围：元素的类型要继承自Numer类 (上限)  \* \*/  public void save(List<? extends Number> list){  //这样写  }    public void testGeneric() throws Exception{  // 可以接受任何泛型集合，但是不能编辑集合值，所以一般在方法参数中使用。    List<Double> list=new ArrayList<Double>();  List<Float> list1=new ArrayList<Float>();  List<Integer> list2=new ArrayList<Integer>();      List<String> list3=new ArrayList<String>();    save(list);  // save(list3); 这个就不行，因为list3 集合中元素属于String 不是继承自 Number  } |

\*由于定义多态的时候，多是定义父类，那么extends 关键字限定只能传自己以及自己的子类的用法就很多。 而像super这样的用的就很少。

# 三.注解以及Log4j

## 概述

注解与注释，

注解，告诉编译器如何运行程序。

注释，告诉程序员阅读的，对运行什么的没有影响。

## 自定义注解

写法：

|  |
| --- |
| public @interface Author {  /\*  \* 注解属性  \* 1.修饰为默认或者public  \* 2.方法不能有主体｛｝  \* \*/  String name();  int age();  } |

使用：

|  |
| --- |
| @Author(name ="zk" , age =23 )  public class AnnotationDemo {  public void save(){  }  } |

## 元注解

注解的注解，修饰注解的注解。

@Target({TYPE, METHOD}) 用来定义注解的范围

@Retention 定义注解的生命周期。

Retention.SOURCE 注解只在源码级别有效

Retention.CLASS 注解只在字节码有效

Retention.RUNTIME 注解在运行时期有效 （范围最大）

## 获取注解信息(注解反射)

先获取代表方法的Method对象，然后通过

Author author=method.getAnnotation(Author.class); 获取到注解Author对象。

## Log4j日志jar使用

程序中为什么用日志组件

简单来说，为了项目后期部署上线后的维护、错误排查。

Log4j.JAR 开源日志组件。

使用步骤：

1. 导包

导入log4j-1.2.11.jar

1. 配置

|  |
| --- |
| **log4j.rootLogger** =debug,console, file  #2. 日志输出到控制台使用的api类。 **log4j.appender.console**=org.apache.log4j.ConsoleAppender  #日志输出格式 **log4j.appender.console.layout**=org.apache.log4j.PatternLayout #---具体格式内容 **log4j.appender.console.layout.ConversionPattern**=%d %p %c.%M()-%m%n  #--------日志输出到文件------------------------------- #---日志输出到文件采用的api， 作用是 文件大小到达一定尺寸的时候产生一个新的文件。 **log4j.appender.file**=org.apache.log4j.RollingFileAppender #指定日志文件路径 **log4j.appender.file.File**=../logs/myLog.log #指定日志文件最大大小 **log4j.appender.file.MaxFileSize**=1024KB #指定日志文件最大数目 **log4j.appender.file.MaxBackupIndex**=10 **log4j.appender.file.layout**=org.apache.log4j.PatternLayout **log4j.appender.file.layout.ConversionPattern**=%d %p %c.%M()-%m%n |

**by zk 2017/6/17**