# 1. 回顾泛型

## 1.1 基本概念

泛型可提高集合的安全性和稳定性（限定类型），使运行时才能发现的问题变为编译时就提示。

注意：泛型是提供给javac编译器使用的，它用于限定集合的输入类型，让编译器在源代码级别上，即挡住向集合中插入非法数据。但编译器编译完带有泛形的java程序后，生成的class文件中将不再带有泛形信息，以此使程序运行效率不受到影响，这个过程称之为“擦除”。

泛型中的几个基本术语说明如下（以ArrayList举例）：

在定义泛型时：

ArrayList<E>：E叫做类型参数变量（类型的形式参数），整个ArrayList<E>叫做泛型类型，即“需要泛型的类型”；

在使用泛型时new ArrayList<Integer>()：

Integer叫做实际的参数类型；ArrayList<Integer>叫做参数化的类型（ParameterizedType）。

## 1.2定义和使用泛型

（1）如在定义方法时使用泛型，则必须要在方法返回值前声明泛型类型，例如：

|  |
| --- |
| **package** com.demo;  **public class** GenericMethodDemo {  *// <T>就是声明泛型T，然后就可在方法的任意一处使用T，  // 例如下面在返回值中、参数中、参数类型的的形式参数中使用T。* **public** <T> T m1 () {  T t = **null**;  **return** t;  }   **public** <T> **void** m2 (T t) {   }   **public** <T> **void** m3(Class<T> clazz) {   } } |

（2）如果在类上声明泛型类型，那么该类所有的实例方法都能使用该泛型类型，无需每个方法都声明一个泛型类型。但是静态方法中还是要单独声明泛型的。例如：

|  |
| --- |
| **package** com.demo;  *// 在类上声明泛型类型* **public class** GenericDemo <T> {  *// 实例方法可直接使用T* **public** T m1() {  T t = **null**;  **return** t;  }   **public void** m3(Class<T> clazz) {   }  *// 但静态方法还是要声明泛型，因为静态方法可单独调用* **public static** <T> **void** m(T t) {  }  *// Map结构,Key-Value* **public static** <K, V> V getValue(K key) {  V v = **null**;  **return** v;  } } |

关于泛型的应用这里就不再讲了（例子很多，比如基于Hibernate的DAO接口中定义的可重用的CRUD方法，就需要带有泛型）。

# 2. DAO设计模型和泛型的反射

## 2.1 DAO设计模型

在SSH框架中，为了避免在每个DAO实现中重复地写基本一致的简单的增删改查方法，我们定义了基本IBaseDao接口和BaseDaoImpl实现类，然后具体的DAO继承他们，以此解决重复编码的问题。

这里再顺便提一下abstract抽象类的使用。

我们可以把BaseDaoImpl类加上abstract关键字使之成为抽象类。为什么设置成抽象类呢？因为BaseDaoImpl不是让用户用户直接使用的，而是需要被具体的DAO实现继承使用，使用abstract关键字强制使用者继承，这样使用者也可重写其中方法，否则的话直接使用抽象类中实现的方法。

这也解释了为什么有些类中没有抽象方法的情况下，还要设置成抽象类，原因就是禁止用户直接使用，例如Tomcat实现的HttpServlet类就是抽象类，目的是让使用者覆盖其中的一些方法（例如doGet、doPost等），另外，doGet这些方法是用protected关键字修饰的，目的也是让子类覆盖使用。这些其实就是模板方法设计模式的应用，可以把一些通用的方法写在抽象类中以形成模板类，后续使用者使用该模板就能减少重复工作（也可重写某些方法）。

回到DAO设计模型上。在之前的SSH框架项目中，虽然我们使用了上述方式避免重复写增删改查的基本DAO代码，但是其中还是有些“瑕疵”的，例如find等方法是这样的：

|  |
| --- |
| */\*\*  \* 根据主键查询对象数据  \** ***@param clazz*** *类字节码文件  \** ***@param id*** *主键  \** ***@return*** *对象  \*/* T findById(Class<T> clazz, Serializable id);  */\*\*  \* 查询某实体所有的数据  \** ***@param clazz*** *类字节码文件  \** ***@return*** *对象集合  \*/* List<T> findAll(Class<T> clazz); |

如果Service层要调用该方法的话，还要这样使用：

|  |
| --- |
| **studentDao**.findAll(Student.**class**);  **studentDao**.findById(Student.**class, 1**); |

显然，我们感觉第一个参数字节码文件是多余的，因为从方法调用的对象studentDao就能判定我操作的肯定是Student，而不是其他的类。想要实现这个目标，就需要使用泛型反射的技术。

## 2.2 利用泛型反射改进BaseDaoImpl

现在我们按照2.1节最后的要求改写代码。既然实体类的字节码文件不应由方法传递过来，但我们又要使用，因此需要在BaseDaoImpl本身这个类中维护实体类的字节码，并且要在构造函数中给这个字节码赋值，这样进行使用。

最终BaseDaoImpl改进如下，详细说明见代码：

|  |
| --- |
| **package** com.ssh.dao.impl;  **import** com.ssh.dao.IBaseDao; **import** org.hibernate.Session; **import** org.hibernate.SessionFactory;  **import** javax.annotation.Resource; **import** java.io.Serializable; **import** java.lang.reflect.ParameterizedType; **import** java.util.List;  **public abstract class** BaseDaoImpl<T> **implements** IBaseDao<T> {   *// 维护的字节码对象* **private** Class **clazz**;  **public** BaseDaoImpl () {  *// 在构造方法中，要给clazz赋值，  // 这样才能让下面的方法知道操作的是哪个实体类，也就是操作哪张表  // 因为这里构造方法的代码会在具体的DAO对象创建时就执行。  // 所以如果这里能拿到正确的clazz，那么就能成功进行下去。   // 先来看目标，我们最终要给clazz赋值成什么呢？其实就是各个DAO实现类在继承本类BaseDaoImpl时所用的实际参数类型。  // 例如 class StudentDaoImpl extends BaseDaoImpl<Student>，  // 我们要拿到的就是Student。   // 注意，这里需要理解的是运行时的类型，例如我们此时在此打印this* System.***out***.println(**this**);  *// 发现该对象就是StudentDaoImpl这些DAO实现类类型，这是没有疑问的。  // 这样就好办了，只要拿到this的父类（就是带有泛型的BaseDaoImpl）的目前的泛型参数实际类型即可。  // 这里要用反射解决。   // 首先，得到当前实例的带有泛型类型的父类（即带有泛型的BaseDaoImpl）。返回的是Type类型。  // Type是一个顶级借口，既然getGenericSuperclass返回的是父类类型，因此可强转为Class类型（Class实现Type接口）  // 而且getGenericSuperclass返回的是带有泛型的父类类型，就是参数化类型，因此也可直接强转为  // ParameterizedType类型（ParameterizedType继承了Type接口）* ParameterizedType pt = (ParameterizedType) **this**.getClass().getGenericSuperclass();  *// 总之，pt就是例如BaseDaoImpl<Student>这样的类型。  // 最后利用ParameterizedType对象中提供的getActualTypeArguments方法，就能获得对应的实际参数类型。  // 这个实际的参数类型就是clazz最终想要的结果。  // 当然，getActualTypeArguments返回的是Type数组，这是因为实际类型参数可能有多个，例如Map集合的<K,V>等。  // 这里我们只要第一个，强转成Class类型即可。* **clazz** = (Class) pt.getActualTypeArguments()[0];  }   @Resource  **protected** SessionFactory **sessionFactory**;   @Override  **public void** add(T t) {  **sessionFactory**.getCurrentSession().save(t);  }   @Override  **public void** delete(T t) {  **sessionFactory**.getCurrentSession().delete(t);  }   @Override  **public void** update(T t) {  **sessionFactory**.getCurrentSession().update(t);  }   @Override  **public** T findById(Serializable id) {  *// 这里返回值也改为强转为T。* **return** (T) **sessionFactory**.getCurrentSession().get(**clazz**, id);  }   @Override  **public** List<T> findAll() {  Session session = **sessionFactory**.getCurrentSession();  **return** session.createQuery(**"from "** + **clazz**.getName(), **clazz**).list();  } } |

其他地方的代码自行修改即可。代码中说明了很多，但实际上只写了两行有效代码，需要读者好好理解。

改进之后，代码优雅了一些，并且代码能够正常执行。

## 2.3 Type接口与泛型

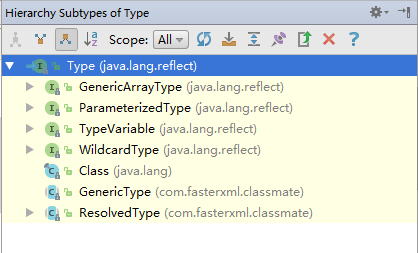
上节代码提到了Type类型，现在展开讲解。

Type接口定义在java.lang.reflect包下，它是一个“标识接口”，其中没有任何方法，

在Java中有一个接口叫“Type”（java.lang.reflect包下的），而Type只是一个“标识接口”，该接口中无方法，该接口解释如下：

|  |
| --- |
| Type is the common superinterface for all types in the Java  programming language. These include raw types, parameterized types,  array types, type variables and primitive types. |

也就是说，Type接口是Java中类型的顶级接口（公共父接口），字节码对象、参数化类型对象等都可强转成Type（但是普通对象不可以）。查看Type的继承结构会发现：



说明Class实现了此接口，ParameterizedType继承了此接口，ParameterizedType就代表泛型类型。

我们还以上一节的BaseDaoImpl和StudentDaoImpl说明（JDK只提供了根据子类对象拿到父类泛型类型的方法，也就是调用getGenericSuperclass方法）直接输出ParameterizedType对象（或者调用ParameterizedType的getName()或toString()方法）则会得到参数化类型（例如com.ssh.dao.impl.BaseDaoImpl<com.ssh.entity.Student>）。

在ParameterizedType接口中，定义了如下几个方法：

|  |
| --- |
| Type[] getActualTypeArguments();  Type getRawType();  Type getOwnerType(); |

它们的返回值都是Type类型，那么上述方法分别返回的Type都是什么呢？actualTypeArguments就是实际参数类型（Student等，是数组，因为可有多个参数）；rawType就是父类对象的类型（BaseDaoImpl）；ownerType，这个比较少用，具体可百度。例如：

|  |
| --- |
| **public** BaseDaoImpl () {  ParameterizedType pt = (ParameterizedType) **this**.getClass().getGenericSuperclass();  System.***out***.println(pt); *// 输出字符串“com.ssh.dao.impl.BaseDaoImpl<com.ssh.entity.Student>”即参数化类型* Class actualTypeArgument = (Class) pt.getActualTypeArguments()[0];  System.***out***.println(**"pt的actualTypeArgument："** + actualTypeArgument.getName()); *// 输出的是com.ssh.entity.Student实际参数类型* Class rawType = (Class) pt.getRawType();  System.***out***.println(**"pt的rawType："** + rawType.getName()); *// 输出的是com.ssh.dao.impl.BaseDaoImpl本父类类型* System.***out***.println(pt.getOwnerType()); *// 这里输出的是null* } |

# 3. Hibernate的元数据

## 3.1 使用Hibernate元数据对查询进行统一封装

### 3.1.1 简单的count使用

例如我们在BaseDao中有一个getCount()方法，能查询本表中记录的行数：

|  |
| --- |
| Long getCount(); |

我们想在BaseDaoImpl中实现一个通用的方法，任何一个持久化类对象通过getCount方法都能拿到本表的记录行数。怎么实现呢？我们想要使用select(主键)，怎么办呢？怎么得到主键字段呢？这时，就要使用Hibernate的元数据了。

实现代码：

|  |
| --- |
| **package** com.ssh.dao.impl;  **import** com.ssh.dao.IBaseDao; **import** org.hibernate.Session; **import** org.hibernate.SessionFactory; **import** org.hibernate.metadata.ClassMetadata;  **import** javax.annotation.Resource; **import** java.io.Serializable; **import** java.lang.reflect.ParameterizedType; **import** java.util.List;  **public class** BaseDaoImpl<T> **implements** IBaseDao<T> {   **private** Class **classt**;  */\*\*  \* 元数据，用来描述持久化类的  \*/* **private** ClassMetadata **classMetadata**;   @Resource  **protected** SessionFactory **sessionFactory**;    **public** BaseDaoImpl() {  ParameterizedType type = (ParameterizedType)**this**.getClass().getGenericSuperclass();  **this**.**classt** = (Class)type.getActualTypeArguments()[0];  */\*\*  \* clast就是代表持久化类的class形式  \*/* **this**.**classMetadata** = **sessionFactory**.getClassMetadata(**classt**);  }  ....   @Override  **public** Long getCount() {  Session session = **sessionFactory**.getCurrentSession();  *// getIdentifierPropertyName获得的是id属性值。* session.createQuery(**"select count( "** + **classMetadata**.getIdentifierPropertyName() + **" ) from "** + **classt**.getName() );  **return null**;  } } |

上述就成功了，所有的DAO都能用。

### 3.1.2 通用的条件查询（单表）

我们可以写一个通用的条件查询，例如一个名叫getEntriesByCondition()，方法的参数就是各种条件。如何传递各种条件呢？那么可以使用一个Map数组，其中包含了查询和条件和对应的参数值。因此这样写：

|  |
| --- |
| List<T> getEntriesByCondition(Map<String, Object> keyValues); |

现在实现它：

|  |
| --- |
| **package** com.ssh.dao.impl;  **import** com.ssh.dao.IBaseDao; **import** org.hibernate.Session; **import** org.hibernate.SessionFactory; **import** org.hibernate.metadata.ClassMetadata; **import** org.hibernate.query.Query;  **import** javax.annotation.Resource; **import** java.io.Serializable; **import** java.lang.reflect.ParameterizedType; **import** java.util.List; **import** java.util.Map;  **public class** BaseDaoImpl<T> **implements** IBaseDao<T> {   **private** Class **classt**;  */\*\*  \* 元数据，用来描述持久化类的  \*/* **private** ClassMetadata **classMetadata**;   @Resource  **protected** SessionFactory **sessionFactory**;   **public** BaseDaoImpl() {  ParameterizedType type = (ParameterizedType)**this**.getClass().getGenericSuperclass();  **this**.**classt** = (Class)type.getActualTypeArguments()[0];  */\*\*  \* clast就是代表持久化类的class形式  \*/* **this**.**classMetadata** = **sessionFactory**.getClassMetadata(**classt**);  }   @Override  **public** Long getCount() {  Session session = **sessionFactory**.getCurrentSession();  *// getIdentifierPropertyName获得的是id属性值。* session.createQuery(**"select count( "** + **classMetadata**.getIdentifierPropertyName() + **" ) from "** + **classt**.getName() );  **return null**;  }   @Override  **public** List<T> getEntriesByCondition(Map<String, Object> keyValues) {  *// 进行实现。就是字符串的拼接* StringBuffer sb = **new** StringBuffer();  *// 1. 先确定查询的表* sb.append(**"from "** + **classt**.getName());  sb.append(**" where 1=1 "**);  *// 2. 循环Map，拼接where中的条件。例如pid=:pid等。  // 为什么使用预编译的HQL而不直接拼接Map中的value到HQL中呢？因为如果不使用预编译HQL，还要特别处理字符串、数字等类型  // 因为字符串要加单引号，而数字不用加。为了简便就使用预编译HQL。* **for** (Map.Entry<String, Object> entry : keyValues.entrySet()) {  sb.append(**"and "** + entry.getKey() + **"=:"** + entry.getKey());  }  *// 3. 得到Query* Session session = **sessionFactory**.getCurrentSession();  Query query = session.createQuery(sb.toString());  *// 4. 再遍历Map，设置预编译参数值* **for** (Map.Entry<String, Object> entry : keyValues.entrySet()) {  query.setParameter(entry.getKey(), entry.getValue());  }  *// 5. 返回查询集合即可* **return** query.list();  } } |

这样使用通用查询就太简单了。因为我们弄成了动态的。