

Java 获取泛型类型

 sprinkle_liz 关注

 1 2019.01.21 00:33:08 字数 1,080 阅读 20,907

不太完美的泛型

大家在使用泛型开发的时候，有很多时候会遇到这样的需求——获取泛型具体的类。
比如现在在 `Service<T>`，要想得到泛型 `T` 的类型，正常情况下，我们第一个想法就是通过 `T.class` 获得，但得到的是那刺眼的下划红色波浪线。
那么如何才能拿到泛型的类型？想要得到泛型的类型需要满足哪些条件呢？

获取到泛型类型的条件

- 1. 必须具有真实类型的存在
- 2. 泛型的类型是明确的


第一个很好理解，如果连要获取的类都不存在，即未定义，那自然是获取不到的；
第二个条件，举个例子，假设存在 `User`，那么 `List<User>` 就是明确的，`List<T>` 是不明确的。



满足上面两点，就可以获取泛型的类型了。



获取泛型类型



首先看下 `java.lang.Class.java` 中与泛型相关的方法有哪些：



Class.java



☐ Inherited members (36F12) ☐ Anonymous Classes (36I) ☐ Lambdas (36L) 



  `getEnclosingMethod(): Method`



  `getEnclosingMethod0(): Object[]`



  `getEnclosingMethodInfo(): EnclosingMethodInfo`



  `getEnumConstants(): T[]`



  `getEnumConstantsShared(): T[]`



  `getExecutableTypeAnnotationBytes(Executable): byte[]`



  `getFactory(): GenericsFactory`



  `getField(String): Field`



  `getField0(String): Field`


  `getFields(): Field[]`



  `getGenericInfo(): ClassRepository`



  `getGenericInterfaces(): Type[]`



  `getGenericSignature0(): String`



  `getGenericSuperclass(): Type`



  `getInterfaces(): Class<?>[]`



  `getInterfaces0(): Class<?>[]`

  `getMethod(String, Class<?>...): Method`

  `getMethod0(String, Class<?>[], boolean): Method`

  `getMethods(): Method[]`

  `getModifiers(): int`

  `getName(): String`

与泛型相关的方法

其中 `getGenericInfo()` 和 `getGenericSignature0()` 都是私有方法，那么我们暂时就只能尝试从 `getGenericInterfaces()` 和 `getGenericSuperclass()` 这两个方法入手了。

查看源码上的注释，大致作用如下：

https://www.jianshu.com/p/6bb9b8d6ee7a

1/4

- `getGenericInterfaces`: 返回此类直接实现的所有接口的类型。接口的实现类通过该方法，可以获取所实现的接口的泛型类型，比如：接口 `Service<T>`，实现类 `UserServiceImpl<User>`，能够得到 `Service<User>`
- `getGenericSuperclass`: 返回此类所表示的实体的直接超类的类型。一个类的子类通过该方法，可以获取超类的泛型类型，比如：`SuperClass<T>`，子类 `Main<User>`，能够得到 `SuperClass<User>`

可以看出都跟继承类/实现接口有关，因为在类或接口定义的类型参数（泛型）其实是不确定的，只有子类继承或接口实现才能确定类型参数的具体类型。比如：接口 `Service<T>`，类型参数 `T` 是不确定的，说白了其实就是个占位符，而实现类 `UserServiceImpl<User>` 才能确定类型参数 `T` 是 `User`。

为了更好理解，先来看一段代码：

```

1 | package xxx.xxx.xxx;
2 |
3 | import java.lang.reflect.ParameterizedType;
4 | import java.lang.reflect.Type;
5 |
6 | abstract class SuperClass<T> {
7 | }
8 |
9 | class Main extends SuperClass<User> {
10 |
11 |     public static void main(String[] args) {
12 |         // 获取 Main 的超类 SuperClass 的签名(携带泛型)。这里为：xxx.xxx.xxx.SuperClass<xxx
13 |         Type genericSuperclass = Main.class.getGenericSuperclass();
14 |         // 强转成 参数化类型 实体。
15 |         ParameterizedType parameterizedType = (ParameterizedType) genericSuperclass;
16 |         System.out.println(parameterizedType);
17 |
18 |         // 获取超类的泛型类型数组。即SuperClass<User>的<>中的内容，因为泛型可以有多个，所以用数组
19 |         Type[] actualTypeArguments = parameterizedType.getActualTypeArguments();
20 |         Type genericType = actualTypeArguments[0];
21 |         Class<User> clazz = (Class<User>) genericType;
22 |         System.out.println(clazz);
23 |     }
24 | }
25 |
26 |
27 | class User {
28 | }
29 |

```

打印结果：

```

1 | xxx.xxx.xxx.SuperClass<xxx.xxx.xxx.User>
2 | class xxx.xxx.xxx.User

```

上面的代码，是通过继承方式，明确类型，然后获取泛型类。

总体思路是这样的：先定义一个类 `SuperClass<T>`，其中 `T` 为类型参数，即泛型，这个时候要想获取 `T` 的类型，显然是不可能的，因为本来就是一个通用类型；但是通过继承该类，并确定了泛型的类型，那么我们就能够通过 `getGenericSuperclass()` 获取到泛型的类型。

其实，上面的代码只是为了方便测试才这么写的，也比较容易理解，一般实战的写法是这样的：

```

1 | abstract class SuperClass<T> {
2 |
3 |     public Class<T> getGenericClass() {
4 |         ParameterizedType parameterizedType = (ParameterizedType) getClass().getGeneri
5 |         return (Class<T>) parameterizedType.getActualTypeArguments()[0];
6 |     }
7 |
8 | }
9 |
10 | class Main extends SuperClass<User> {

```

```

11 |
12 |     public static void main(String[] args) {
13 |         System.out.println(new Main().getGenericClass());
14 |     }
15 |
16 | }
17 |
18 | class User {
19 | }

```

不出意外的话，会打印如下结果：

```
1 | class xxx.xxx.xxx.User
```

需要注意的是，`SuperClass<T>` 是一个抽象类，因为 `getGenericClass()` 方法是为其的子类服务的，`SuperClass<T>` 本身的实例对象如果调用该方法，将会直接报错或返回错误的结果，所以干脆定义成抽象类，避免被实例化。另外，为了防止方法被覆写，可以使用 `private` 访问修饰符或者定义为 `final` 方法。

当然多级继承也是适用的，比如：

```

1 | // 这里省略 SuperClass 和 User 的定义
2 |
3 | class Main<T> extends SuperClass<T> {
4 | }
5 |
6 | class MultiInheritance extends Main<User> {
7 |     public static void main(String[] args) {
8 |         System.out.println(new MultiInheritance().getGenericClass());
9 |     }
10 | }

```

通过上面的代码，基本可以了解通过 `getGenericSuperClass()` 获取泛型类型的大概思路和原理，而 `getGenericInterfaces()` 的获取思路也是大同小异。直接上实例代码：

```

1 | package xxx.xxx.xxx;
2 |
3 | import java.lang.reflect.ParameterizedType;
4 | import java.lang.reflect.Type;
5 |
6 | interface ServiceA<T> {
7 | }
8 |
9 | interface ServiceB<T> {
10 | }
11 |
12 | class ServiceImpl implements ServiceA<EntityA>, ServiceB<EntityB> {
13 |
14 |     public static void main(String[] args) {
15 |         Type[] genericInterfaces = ServiceImpl.class.getGenericInterfaces();
16 |         if (genericInterfaces.getClass().isAssignableFrom(ParameterizedType[].class)) {
17 |             for (Type genericInterface : genericInterfaces) {
18 |                 ParameterizedType parameterizedType = (ParameterizedType) genericInterface;
19 |                 Type[] actualTypeArguments = parameterizedType.getActualTypeArguments();
20 |                 Type type = actualTypeArguments[0];
21 |                 if (type instanceof Class) {
22 |                     Class<?> clazz = (Class<?>) type;
23 |                     System.out.println(clazz);
24 |                 }
25 |             }
26 |         }
27 |     }
28 |
29 | }
30 |
31 | class EntityA {
32 | }
33 |
34 | class EntityB {
35 | }

```

控制台打印结果如下：

```
1 | class xxx.xxx.xxx.EntityA
2 | class xxx.xxx.xxx.EntityB
```

扩展

很多情况下，Class被用来当作参数，我们其实可以将带泛型的类作为参数传入，我们一般为了方便，很少去特定定义一个类，因此，我们可以借助匿名内部类的便利来达到这一目的。

首先看一段代码：

```
1 | package xxx.xxx.xxx;
2 |
3 | import java.lang.reflect.ParameterizedType;
4 |
5 | abstract class SuperClass<T> {
6 |
7 |     public Class<T> getGenericClass() {
8 |         ParameterizedType parameterizedType = (ParameterizedType) getClass().getGeneri
9 |         return (Class<T>) parameterizedType.getActualTypeArguments()[0];
10 |     }
11 |
12 |     public static void main(String[] args) {
13 |         System.out.println(new SuperClass<User>(){}.getGenericClass());
14 |     }
15 |
16 | }
```

其中 `SuperClass<User>(){}` 就是一个匿名内部类。当然上面只是为了展示匿名内部类，真正的应用可以参考 [fastjson](#) 的 [TypeRefrence](#)。



sprinkle_liz

总资产46 共写了10.2W字 获得727个赞 共388个粉丝

关注