Theory: Generics and Reflection

1 hour 0 / 5 problems solved

Type erasure process removes information about parameter types of generic classes at compile time. If you are given an object of generic class at runtime, normally it is not possible to examine which parameter is actually used. However, if you have access to an object's reference, **reflection** may provide information about the parameters of the object.

Reflection has a set of classes describing the internal implementation details of a class. There are `Class`, `Method` and `Field`. So if you have an object's reference you can access these classes and then get information about parameters of generic types.

```
void testArgument(SomeClass object) throws Exception {
    Class clazz = object.getClass();
    Field field = clazz.getDeclaredField("fieldName");
    Method method = clazz.getDeclaredMethod("methodName");
}
```

Current topic:

Generics and Reflection

Topic depends on:

✕ Reflection basics

✕ Type Erasure

§1. Print parameters

The simplest way to obtain full information about parameters is just using `toGenericString` method. It is common for `Field`, `Method` and `Class`. Let's look at the class:

```
class SomeClass<T> {
    public Map<String, Integer> map;
    public List<? extends Number> getList(T obj);
}
```

The following code snippet shows how to print details of a class at runtime:

```

System.out.println(SomeClass.class.toGenericString()); // class SomeClass<T>

    Field mapField = SomeClass.class.getDeclaredField("map");

System.out.println(mapField.toGenericString()); // public java.util.Map<java.lang.
String, java.lang.Integer> SomeClass.map

    Method method = SomeClass.class.getDeclaredMethod("getList");

System.out.println(method.toGenericString()); // public java.util.List<? extends j
ava.lang.Number> SomeClass.getList(T obj)
```


§2. Parameterized type

`toGenericString` method is quite useful, but sometimes you need to fetch parameters one by one instead of getting a summary string. For such cases reflection classes have `ParameterizedType` interface. It describes parameterized types and has the method `getActualTypeArguments` providing an array of parameter types.

To illustrate how `ParameterizedType` can be obtained, let's consider cases based on the class:

```
public class DataHolder {
    public Map<String, Integer> data;

    public void setData(Map<String, Integer> data) {
        this.data = data;
    }

    public Map<String, Integer> getData() {
        return data;

    }

}
```

Suppose you want to discover which parameter is used by the `data` field. `genericType` method of `Field` class produces an object of `ParameterizedType` interface, containing such information

```
    Field field = DataHolder.class.getDeclaredField("data");

ParameterizedType parameterizedType = (ParameterizedType) field.getGenericType();

    Type rawType = parameterizedType.getRawType(); // interface java.util.Map

Type[] argumentTypes = parameterizedType.getActualTypeArguments(); // class java.l
ang.String, class java.lang.Integer

    // or you can get type name as a String
    String arg1TypeName = argumentTypes[0].getTypeName(); // java.lang.String
    String arg2TypeName = argumentTypes[1].getTypeName(); // java.lang.Integer
```

A method may have parameterized argument and return type. For both cases `Method` class also provides `ParameterizedType`. For instance, it is possible to examine the return type of `getData` method

```
    Method method = DataHolder.class.getMethod("getData");

ParameterizedType parameterizedType = (ParameterizedType) method.getGenericReturnT
ype();
```

or arguments of `setData` method

```
    Method method = DataHolder.class.getMethod("setData", Map.class);
    Type[] parameterTypes = method.getGenericParameterTypes();

ParameterizedType parameterizedType = (ParameterizedType) parameterTypes[0]; // me
thod has a single parameter
```

§3. Wildcard type

`ParameterizedType` is good enough for extracting parameters of types like `List<String>` or `Map<K, V>`, but it lacks detailing when applied to wildcard. In other words, it has no methods for getting bounds.

Imagine there is a field of a wildcard type `List<? extends Number> wildcardField`. Using `ParameterizedType` you can obtain only `? extends Number` parameter:

```
    Field field = ...

ParameterizedType parameterizedType = (ParameterizedType) field.getGenericType();
    Type type = parameterizedType.getActualTypeArguments()
[0]; // ? extends Number
```

However, reflection allows finding bounds of the wildcard type via `WildcardType` interface:

```

WildcardType wildcardType = (WildcardType) parameterizedType.getActualTypeArgument
s()[0]; // There is a single parameter
    System.out.println(wildcardType.getLowerBounds()); // empty
    System.out.println(wildcardType.getUpperBounds()); // Number
```

§4. Type variable

We have covered how to analyze parameterized class `DataHolder`. Let's take a look at its generic version

```
class GenericDataHolder<K extends String, V extends Number> {
    public Map<K, V> data;

    public void setData(Map<K, V> data) {
        this.data = data;
    }

    public Map<K, V> getData() {
        return data;

    }

}
```

For such cases, that contain type variables, e.g. `<K extends String, V extends Number>` `TypeVariable` class is used to retrieve detailed information about types. `Class` and `Method` classes have a method `getTypeParameters`, which provides a list of type variables declared for them. `TypeVariable` class has a method for getting bounds as well as `WildcardType` provides.

```
    TypeVariable<Class<GenericDataHolder>>
[] typeVariables = GenericDataHolder.class.getTypeParameters();
    System.out.println("Type variables count " + typeVariables.length);

    System.out.println(typeVariables[0]); // K

System.out.println("First type var upper bound " + typeVariables[0].getBounds()
[0]); // java.lang.String

    System.out.println(typeVariables[1]); // V

System.out.println("Second type var upper  bound " + typeVariables[1].getBounds()
[0]); // java.lang.Number
```

§5. Generic Array type

Like wildcards, generic array type has a special interface `GenericArrayType`. Let's apply it to retrieve the type of `T[] genericArrayField` field:

```
    Field field = DataHolder.class.getDeclaredField("genericArrayField");
    GenericArrayType arrayType = (GenericArrayType) field.getGenericType();
    System.out.println(arrayType); // T[]
    System.out.println(arrayType.getGenericComponentType()); // T
```

§6. Generic ancestor and interfaces

Reflection also allows us to obtain types of generic interfaces. Let's consider the example:

```
    // Generic interface
    interface GenericInterface<T> {}

    // Class that implements generic interface with some type argument
    class SomeClass implements GenericInterface<Boolean> {}


SomeClass.class.getGenericInterfaces(); // GenericInterface<java.lang.Boolean>
```

The same approach works for getting parameters of parent generic class. The single difference is that you should use `getGenericSuperclass` method.

§7. Bridge method

Let's recall the generic class:

```
class Data<T> {
    private T data;

    public T get() {
        return data;
    }

    public void set(T data) {
        this.data = data;

    }

}
```

and it's successor:

```
public class NumberData extends Data<Number> {
    public void set(Number number) {
        System.out.println("NumberData set");
        super.set(number);
    }
}
```

As you remember the compiler generates bridge methods for **get** and **set**.
Generated bridge methods are available in runtime via reflection. Method class has `isBridge` method for checking is it bridge method or not:

```
for (Method method : NumberData.class.getMethods()) {
    if (method.isBridge()) {
        System.out.println(method.getName());
    }
}
```

The code snippet prints names of bridge methods only.

§8. Conclusion

Type erasure removes information about parameters during the compilation. An object of the generic type is not aware of what parameter is actually used. However, an object reference still holds such data. Reflection makes it possible to retrieve the data. It introduces a set of classes and interfaces providing details about parameters. Another useful application of reflection is revealing of methods which were generated by the compiler, for example, bridge methods.

Report a typo

26 users liked this theory. **5** didn't like it. **What about you?**

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion