第八章 - 泛型、注解

1 泛型

1.1 问题引入

在前面学习集合时，我们都知道集合中是可以存放任意对象的，只要把对象存储集合后，那么这时他们都会被提升成Object类型。当我们在取出每一个对象，并且进行相应的操作，这时必须采用类型转换。

观察下面代码：

```
public static void main(String[] args) {
    //没有给泛型参数传值，那么泛型默认表示为Object类型
    Collection c = new ArrayList();
    c.add("hello1");
    c.add("hello2");
    c.add("hello3");
    c.add(1);

    for(Object obj : c){
        String str = (String) obj;
        System.out.println(str);
    }
}

//运行结果：
Exception in thread "main" java.lang.ClassCastException:
java.lang.Integer cannot be cast to java.lang.String

```

程序在运行时发生了问题`java.lang.ClassCastException`。

由于集合中什么类型的元素都可以存储。导致取出时强转引发运行时ClassCastException。

Collection虽然可以存储各种对象，但实际上通常Collection只存储同一类型对象。例如都是存储字符串对象。因此在JDK5之后，新增了**泛型**(Generic)语法，让你在设计API时可以指定类或方法支持泛型，这样我们使用API的时候也变得更为简洁，并得到了编译时期的语法检查。

1.2 泛型概述

泛型（Generics）的概念是在JDK1.5中引入的，它的主要目的是为了解决类型安全性和代码复用的问题。

泛型是一种强大的特性，它允许我们在定义类、接口和方法时使用**参数化类型**。

泛型基本语法为定义在`<>`中，例如下面案例：

```
//T是数据类型，但是不是确定的数据类型
//程序员在使用该类的时候，传入什么具体的类型给T，T就代表什么类型
public class MyClass<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

`MyClass`是一个泛型类，使用类型参数`T`。我们可以在创建对象时指定具体的类型，例如`MyClass<Integer>`或`MyClass<String>`。

泛型能够使我们编写出来通用的代码，提高代码的可读性和重用性。通过使用泛型，我们可以在类、接口和方法中使用类型参数，使得代码可以处理不同类型的数据，同时保持类型安全。

1.3 泛型应用

了解泛型的意思之后，接下来可以再看下之前学习过的集合中的泛型：

```
1  //Collection接口定义，其是一个泛型接口
2  public interface Collection<E> {
3      //省略...
4
5      boolean add(E e);
6  }
```

Collection是一个泛型接口，泛型参数是E，add方法的参数类型也是E类型。

在使用Collection接口的时候，给泛型参数指定了具体类型，那么就会防止出现类型转换异常的情况，因为这时候集合中添加的数据已经有了一个规定的类型，其他类型是添加不进来的。

例如下面案例中，我们指定了集合c只能存储String类型数据，则Integer类型的1就无法添加成功。

```
1  public static void main(String[] args) {
2      Collection<String> c = new ArrayList<String>();
3
4      c.add("hello1");
5      c.add("hello2");
6      c.add("hello3");
7      //编译报错，add(E e) 已经变为 add(String e)
8      //int类型的数据1，是添加不到集合中去的
9      //c.add(1);
10
11     for(String str : c) {
```

```
12          System.out.println(str);
13      }
14  }
```

> 可以看出，传入泛型参数后，add方法只能接收String类型的参数，其他类型的数据无法添加到集合中，同时在遍历集合的时候，也不需要我们做类型转换了，直接使用String类型变量接收就可以了，JVM会自动转换的

> `Collection<String> c = new ArrayList<String>();`
>
> 可简写为菱形泛型形式：
>
> `Collection<String> c = new ArrayList<>();`

菱形泛型（Diamond Operator）是JDK7中引入的一种语法糖，用于简化泛型的类型推断过程。

`Map`接口使用泛型：

```
1  //Map接口也是泛型接口
2  public interface Map<K,V> {
3      //省略...
4
5      V put(K key, V value);
6      Set<Map.Entry<K, V>> entrySet();
7  }
```

案例：

```
1   public static void main(String[] args) {
2       Map<Integer,String> map = new HashMap<>();
3
4       //根据泛型类型的指定，put方法中的key只能是Integer类型，value只能是String类型
```

```
5       map.put(1,"hello1");
6       map.put(2,"hello2");
7       map.put(3,"hello3");
8       map.put(4,"hello4");
9
10      //根据上面列出的源码可知，当前指定Map的泛型类型为：
    Map<Integer,String> map
11      //entrySet方法返回的类型就应该是Set<Map.Entry<Integer,
    String>>
12      Set<Map.Entry<Integer, String>> entrySet = map.entrySet();
13
14      for(Map.Entry entry:entrySet){
15          System.out.println(entry.getKey()+" :
    "+entry.getValue());
16      }
17  }
```

1.4 自定义泛型

Java中的泛型分三种使用情况：

- 泛型类
- 泛型接口
- 泛型方法

1）泛型类

如果泛型参数定义在类上面，那么这个类就是一个泛型类

泛型类定义格式：

```
[修饰符] class 类名<泛型类型名1,泛型类型名2,...> {
    0个或多个数据成员;
    0个或多个构造方法;
    0个或多个成员方法;
}

//注意：之前用确定数据类型的地方，现在使用自定义泛型类型名替代
```

例如：JDK中HashSet泛型类定义如下

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
{
    static final long serialVersionUID = -5024744406713321676L;

    private transient HashMap<E,Object> map;
```

泛型类实例化对象格式：

泛型类名<具体类型1,具体类型2,...> 对象名 = new 泛型类名<>(实参列表);

案例展示：

定义一个泛型类Circle，包含x，y坐标和radius半径，然后进行功能测试。

基础泛型类：

```
package com.briup.chap08.bean;

//自定义泛型类：圆
//class 类名<泛型类型1，泛型类型2,...>
//  泛型类型名字可以自行定义
public class Circle<T, E> {
    //原来具体数据类型的地方，使用泛型类型名替换即可
    private T x;
```

```
9       private T y;
10      private E radius;
11
12      //无参构造器没有任何改变
13      public Circle() {}
14      //原来具体数据类型的地方，使用泛型类型名替换即可
15      public Circle(T x, T y, E radius) {
16          this.x = x;
17          this.y = y;
18          this.radius = radius;
19      }
20
21      public T getX() {
22          return x;
23      }
24      public void setX(T x) {
25          this.x = x;
26      }
27      public T getY() {
28          return y;
29      }
30      public void setY(T y) {
31          this.y = y;
32      }
33      public E getRadius() {
34          return radius;
35      }
36      public void setRadius(E radius) {
37          this.radius = radius;
38      }
39
40      @Override
41      public String toString() {
42          return "Circle [x=" + x + ", y=" + y + ", radius=" +
    radius + "]";
43      }
```

```
}
```

测试类：

```
package com.briup.chap08.test;

import com.briup.chap08.bean.Circle;

public class Test014_GenericsClass {
    public static void main(String[] args) {
        //实例化泛型类对象：
        //  泛型类<具体类型1,具体类型2,...> 对象 = new 泛型类<>(实参s);

        //1.实例化具体类对象，2种泛型设置为Integer和Double
        // 注意，泛型类可以是任意引用类型
        Circle<Integer, Double> c1 = new Circle<>(2,3,2.5);
        int x = c1.getX();
        double r = c1.getRadius();
        System.out.println("x: " + x + " radius: " + r);

        System.out.println("------------------");

        //2.实例化具体类对象，2种泛型设置为Double和Integer
        Circle<Double, Integer> c2 = new Circle<>(2.0,3.0,2);
        double x2 = c2.getX();
        int r2 = c2.getRadius();

        System.out.println("x2: " + x2 + " r2: " + r2);
    }
}

//运行结果：
x: 2 radius: 2.5
------------------
x2: 2.0 r2: 2
```

注意：实际开发中，我们自定义泛型类的情况并不多，大家掌握定义泛型类、实例化泛型类对象的固定格式即可。

2）泛型接口

如果泛型参数定义在接口上面，那么这个接口就是一个泛型接口

定义格式：

```
[修饰符] interface 接口名<泛型类型名1,泛型类型名2,...> {  }
```

例如：JDK中Set泛型接口参考：

```
83  */
84
85 public interface Set<E> extends Collection<E> {
86     // Query Operations
87
88     /**
```

在泛型接口中，我们使用T来代表某一个类型，这个类型具体是什么将来使用的时候再传参确定。

```
1  public interface Action<T> {...}
2
3  public static void main(String[] args) {
4      //创建匿名内部类
5      Action<String> a = new Action<>() {
6          //...
7      };
8  }
9
```

泛型接口使用跟泛型类使用类似，在此不专门举例说明。

3）泛型方法

如果泛型参数定义在方法上面，那么这个方法就是一个泛型方法

泛型方法定义格式：

```
[修饰符] <泛型类型名> 返回值类型 方法名(形式参数列表){
    方法具体实现;
}
```

泛型方法调用格式：

`类对象.泛型方法(实参列表);`

`类名.static泛型方法(实参列表);`

注意：泛型方法调用时不需要额外指定泛型类型，系统自动识别泛型类型。

案例展示：

在上述Circle泛型类中，补充`泛型方法disp()和static show()`，并调用，验证上述格式。

基础类Circle：

其他代码不变，核外补充下面2个泛型方法即可！

```
public class Circle<T,E> {
    //省略...

    //泛型类中定义 泛型方法
    public <F> void disp(F f) {
        System.out.println("in 泛型方法disp, f: " + f);
    }

    //  下面写法虽然不会报错，不建议大家这样写
```

```
    //  因为泛型方法上的T 会和 泛型类上的T 产生歧义
    public static <T> void show(T t) {
        System.out.println("in 泛型static方法show, t: " + t);
    }
}
```

测试类:

```
public static void main(String[] args) {
    Circle<Integer,Integer> c = new Circle<>();

    //public <F> void disp(F f);
    //调用时系统自动识别泛型方法类型
    c.disp(1);      //Integer
    c.disp(2.3);    //Double
    c.disp("hello");//String
    c.disp('h');    //Character

    System.out.println("--------------");

    //public static <T> void show(T t);
    //通过类名可以直接调用，不需要额外指定泛型类型
    Circle.show(2.3);
    Circle.show(2);
    Circle.show("hello");
}

//运行结果:
in 泛型方法disp, f: 1
in 泛型方法disp, f: 2.3
in 泛型方法disp, f: hello
in 泛型方法disp, f: h
--------------
in 泛型static方法show, t: 2.3
in 泛型static方法show, t: 2
in 泛型static方法show, t: hello
```

1.5 注意事项

先看俩种错误的情况：

```
1  //编译通过
2  //父类型的引用，指向子类对象
3  Object o = new Integer(1);
4
5  //编译通过
6  //Object[]类型兼容所有的【引用】类型数组
7  //arr可以指向任意 引用类型 数组对象
8  Object[] arr = new Integer[1];
9
10 //编译失败
11 //注意，这个编译报错，类型不兼容
12 //int[] 是基本类型数组
13 Object[] arr = new int[1];
14
15 //编译失败
16 //错误信息：ArrayList<Integer>无法转为ArrayList<Object>
17 //在编译期间，ArrayList<Integer>和ArrayList<Object>是俩个不同的类
   型，并且没有子父类型的关系
18 ArrayList<Object> list = new ArrayList<Integer>();
```

> **注意，= 号俩边的所指定的泛型类型，必须是要一样的**
>
> 这里说的泛型类型，指的是<>中所指定的类型

虽然 `Integer` 是 `Object` 的子类型，**但是 `ArrayList<Integer>` 和 `ArrayList<Object>` 之间没有子父类型的关系**，它们就是俩个不同的类型

所以，

`Object o = new Integer(1);`编译通过

`ArrayList<Object> list = new ArrayList<Integer>();`编译报错

也就是说，俩个类型，如果是当做泛型的指定类型的时候，就没有多态的特点了

1.6 通配符

?：可以通配任意类型

观察下面代码：

```
1  public void test1(Collection<Integer> c) {
2
3  }
4
5  public void test2(Collection<String> c) {
6
7  }
8
9  public void test3(Collection<Double> c) {
10
11 }
```

> test1方法【只能】接收泛型是Integer类型的集合对象
>
> test2方法【只能】接收泛型是String类型的集合对象
>
> test3方法【只能】接收泛型是Double类型的集合对象
>
> 原因：由于泛型的类型之间没有多态，所以=号俩边的泛型类型必须一致

在这种情况下，就可以使用通配符（?）来表示泛型的父类型：

```java
public void test(Collection<?> c) {

}
```

> 注意，这时候test方法中的参数类型，使用了泛型，并且使用问号来表示这个泛型的类型，这个问号就是通配符，可以匹配所有的泛型类型
>
> test方法可以接收 **泛型是任意引用类型的** Collection集合对象

```java
public static void main(String[] args){
    Test t = new Test();
    t.test(new ArrayList<String>());
    t.test(new ArrayList<Integer>());
    t.test(new ArrayList<Double>());
    t.test(new ArrayList<任意引用类型>());
}
```

使用通配符（?）所带来的问题：

```java
Collection<?> c;
c = new ArrayList<String>();

//编译报错
//因为变量c所声明的类型是Collection，同时泛型类型是通配符（?）
//那么编译器也不知道这个?将来会是什么类型，因为这个?只是一个通配符
//所以，编译器不允许使用变量c来向集合中添加新数据。
c.add("hello");

//编译通过
//但是有一个值是可以添加到集合中的，null
//集合中一定存的是引用类型，null是所有引用类型共同的一个值，所以一定可以添加进去。
c.add(null);
```

虽然使用通配符（?）的集合，不能再往其中添加数据了，但是可以遍历集合取出数据：

```java
public static void main(String[] args) {

    ArrayList<String> list = new ArrayList<>();
    list.add("hello1");
    list.add("hello2");
    list.add("hello3");
    list.add("hello4");

    Collection<?> c = list;

    //编译报错
    //c.add("hello5");

    for(Object obj : c) {
        System.out.println(obj);
    }
```

```
17    }
```

1.7 泛型边界

在默认情况下，泛型的类型是可以任意设置的，只要是引用类型就可以。

如果在泛型中使用`extends`和`super`关键字，就可以对泛型的类型进行限制。即：规定泛型的**上限**和**下限**。

泛型的上限：

- **格式**：`类型名<? extends 类型 > 对象名称`
- **意义**：`只能接收该类型及其子类`

泛型的下限：

- **格式**：`类型名<? super 类型 > 对象名称`
- **意义**：`只能接收该类型及其父类型`

1）泛型上限

- 例如：`List<? extends Number> list`
- 将来引用list就可以接收泛型是`Number`或者`Number`子类型的List集合对象

```java
public static void main(String[] args) {

    List<? extends Number> list;
    //list可以指向泛型是Number或者Number【子】类型的集合对象
    list = new ArrayList<Number>();
    list = new ArrayList<Integer>();
    list = new ArrayList<Double>();

    //编译报错，因为String不是Number类型，也不是Number的子类型
    //list = new ArrayList<String>();
}
```

> 能表示数字的类型都是Number类型的子类型，例如Byte Short Integer Long 等

2）泛型下限

- 例如：`List<? super Number> list`
- 将来引用list就可以接收泛型是`Number`或者`Number`父类型的List集合对象

```java
public static void main(String[] args) {

    List<? super Number> list;
    //list可以指向泛型是Number或者Number【父】类型的集合对象
    list = new ArrayList<Number>();
    list = new ArrayList<Serializable>();
    list = new ArrayList<Object>();

    //编译报错，因为String不是Number类型，也不是Number的父类型
    //list = new ArrayList<String>();

    //编译报错，因为Integer不是Number类型，也不是Number的父类型
    //list = new ArrayList<Integer>();
```

```
14  }
```

泛型中`extends`和`super`对比：

- 使用extends可以定义泛型的【上限】，这个就表示将来泛型所接收的类型【最大】是什么类型。可以是这个最大类型或者它的【子类型】。
- 使用super可以定义泛型的【下限】，这个就表示将来泛型所接收的类型【最小】是什么类型。可以是这个【最小类型】或者它的【父类型】。

1.8 类型擦除

泛型类型仅存在于编译期间，编译后的字节码和运行时不包含泛型信息，所有的泛型类型映射到同一份字节码。

由于泛型是JDK1.5才加入到Java语言特性的，Java让编译器擦除掉关于泛型类型的信息，这样使得Java可以向后兼容之前没有使用泛型的类库和代码，因为在字节码（class）层面是没有泛型概念的。

例如，定义一个泛型类`Generic`是这样的：

```
1   class Generic<T> {
2       private T obj;
3
4       public Generic(T o) {
5           obj = o;
6       }
7
8       public T getObj() {
9           return obj;
10      }
11  }
```

那么，Java编译后的字节码中`Generic`相当于这样的：（类型擦除，变为原始类型Object）

```
class Generic {
    private Object obj;

    public Generic(Object o) {
        obj = o;
    }

    public Object getObj() {
        return obj;
    }
}
```

例如，

```
public static void main(String[] args) {
    //编译报错
    //ArrayList<Integer>和new ArrayList<Long>在编译期间是不同的类型
    //ArrayList<Integer> list = new ArrayList<Long>();

    //但是编译完成后，它们对应的是同一份class文件：ArrayList.class
    ArrayList<Integer> list1 = new ArrayList<Integer>();
    ArrayList<Long> list2 = new ArrayList<Long>();
    // 大家大致能看懂即可，后续反射章节会补充
    System.out.println(list1.getClass() == list2.getClass()); //true
}
```

> 注意，泛型信息被擦除后，所有的泛型类型都会统一变为原始类型：Object

例如，

```
//编译报错
//因为在编译后，泛型信息会被擦除
//所以下面两个run方法不会构成重载，本质上都是 public void run(List list)
public class Test {
    public void run(List<String> list){

    }

    public void run(List<Integer> list){

    }
}
```

> 可以看出，Java的泛型只存在于编译时期，泛型使编译器可以在编译期间对类型进行检查以提高类型安全，减少运行时由于对象类型不匹配引发的异常。
>
> 但是在编译成功后，所有泛型信息会被擦除，变为原始类型Object

1.9 泛型小结

概念	描述	示例
泛型类（Generic Class）	使用泛型参数化的类，可以在实例化时指定具体类型。	`class MyClass<T> { ... }`
泛型接口（Generic Interface）	使用泛型参数化的接口，可以在实现时指定具体类型。	`interface MyInterface<T> { ... }`
泛型方法（Generic Method）	使用泛型参数化的方法，可以在调用时指定具体类型，与所属类的泛型参数可以不同。	`public <T> void myMethod(T data) { ... }`
类型参数（Type Parameter）	在泛型中使用的占位符类型，用于指定参数化类型。	`<T>`、`<K, V>`
通配符（Wildcard）	用于表示未知类型的通配符，用于灵活处理不确定类型的泛型。	`List<?>`、`List<? extends Number>`、`List<? super Integer>`
类型限定（Type Bounds）	限定泛型的类型范围，可以是具体类型、接口或类。	`<T extends Number>`、`<T extends Comparable<T>>`
类型擦除（Type Erasure）	泛型在编译时会进行类型擦除，生成原始类型的字节码，运行时无法获取泛型类型的具体信息。	-

2 注解

本章内容不用死记硬背，了解即可，以后能看得懂代码中的注解，知道其作用即可。

2.1 概述

注解（Annotation），是JDK1.5引入的技术，用它可以对Java中的某一个段程序进行说明或标注，并且这个注解的信息可以被其他程序使用特定的方式读取到，从而完成相应的操作。

例如，`@Override`注解

```
1  public class Person {
2      @Override
3      public String toString() {
4          return super.toString();
5      }
6  }
```

`@Override`功能解析：

编译器在编译`Person类`的时候，会读取到`toString方法上的注解` `@Override`，从而帮我们检查这个方法是否是重写父类中的，如果父类中没有这个方法，则编译报错。

注解和注释的区别：

- 注解是给其他程序看的，通过参数的设置，可以在编译后class文件中【保留】注解的信息，其他程序读取后，可以完成特定的操作
- 注释是给程序员看的，无论怎么设置，编译后class文件中都是【没有】注释信息，方便程序员快速了解代码的作用或结构

2.2 格式

定义注解的格式：

- 没有属性的注解：

```
public @interface 注解名称 {

}
```

- 有属性，但没有默认值的注解：

```
public @interface 注解名称 {
    public 属性类型 属性名();

}
```

- 有属性，有默认值的注解：

```
public @interface 注解名称 {
    属性类型 属性名() default 默认值 ;

}
```

> 注意，`public`可以省去不写，默认就是`public`

2.3 范围

注解的使用范围，都定义在了一个枚举类中：

```
package java.lang.annotation;
```

```
2
3  public enum ElementType {
4      /** Class, interface (including annotation type), or enum
   declaration */
5      TYPE,
6
7      /** Field declaration (includes enum constants) */
8      FIELD,
9
10     /** Method declaration */
11     METHOD,
12
13     /** Formal parameter declaration */
14     PARAMETER,
15
16     /** Constructor declaration */
17     CONSTRUCTOR,
18
19     /** Local variable declaration */
20     LOCAL_VARIABLE,
21
22     /** Annotation type declaration */
23     ANNOTATION_TYPE,
24
25     /** Package declaration */
26     PACKAGE,
27
28     /**
29      * Type parameter declaration
30      *
31      * @since 1.8
32      */
33     TYPE_PARAMETER,
34
35     /**
36      * Use of a type
```

```
37          *
38          * @since 1.8
39          */
40         TYPE_USE
41     }
42
```

具体描述如下：

- TYPE，使用在类、接口、注解、枚举等类型上面

```
1   @Test
2   public class Hello {
3
4   }
```

- FIELD，使用在属性上面

```
1   public class Hello {
2       @Test
3       private String msg;
4   }
```

- METHOD，使用在方法上面

```
1   public class Hello {
2       @Test
3       public void say() {
4
5       }
6   }
```

- PARAMETER，使用在方法的参数前面

```
public class Hello {

    public void say(@Test String name) {

    }
}
```

- CONSTRUCTOR，使用在构造器上面 **（了解即可）**

```
public class Hello {
    @Test
    public Hello() {

    }
}
```

- LOCAL_VARIABLE，使用在局部变量上面 **（了解即可）**

```
public class Hello {
    public void say() {
        @Test
        int num = -1;
    }
}
```

- ANNOTATION_TYPE，使用在注解类型上面 **（了解即可）**

```
@Test
public @interface Hello {

}
```

- PACKAGE，使用在包上面**（了解即可）**

```
@Test
package com.briup.test;
```

> 注意，包注解只能写在package-info.java文件中

> 注意，package-info.java文件里面，只能包含package声明，并做出描述，以便将来生成doc文件，可以从API源码src.zip中，看到每个包下面都可以对应的package-info.java文件对该包做出描述

- TYPE_PARAMETER，使用在声明泛型参数前面，JDK8新增**（了解即可）**

```
public class Hello{
    public <@Test T> void sendMsg(T t) {

    }
}
```

```
interface Action<@Test T> {

}
```

- TYPE_USE，使用在代码中任何类型前面，JDK8新增**（了解即可）**

```java
public class Hello{
    public void say(@Test String name) {

        List<@Test String> list = null;

        Object obj = new @Test String("briup");

        String str = (@Test String) obj;
    }
}
```

2.4 保持

类中使用的注解，根据配置，可以**保持**到三个不同的阶段：

- SOURCE，注解只保留在源文件，当Java文件编译成class文件的时候，注解被遗弃
- CLASS，注解被保留到class文件，但jvm加载class文件时候被遗弃
- RUNTIME，注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在

注解的三种保留策略，都定义在了一个枚举类中：

```java
package java.lang.annotation;

public enum RetentionPolicy {
    /**
     * Annotations are to be discarded by the compiler.
     */
    SOURCE,

    /**
```

```
10        * Annotations are to be recorded in the class file by the
    compiler
11        * but need not be retained by the VM at run time.  This
    is the default
12        * behavior.
13        */
14       CLASS,
15
16       /**
17        * Annotations are to be recorded in the class file by the
    compiler and
18        * retained by the VM at run time, so they may be read
    reflectively.
19        *
20        * @see java.lang.reflect.AnnotatedElement
21        */
22       RUNTIME
23   }
24
```

> 注意，Retention是保留、保持的意思，Policy是政策、策略的意思

如果需要在运行时去动态获取注解信息，那只能用 RUNTIME 注解，比如@Deprecated使用RUNTIME注解。

如果要在编译时进行一些预处理操作，比如生成一些辅助代码，就用 CLASS注解；

如果只是做一些检查性的操作，比如 @Override 和 @SuppressWarnings，使用 SOURCE 注解

> 注意，因为RUNTIME的生命周期最长，所以其他俩种情况能作用到的阶段，使用RUNTIME也一定能作用到

2.5 元注解

在我们进行自定义注解的时候，一般会使用到元注解，来设置自定义注解的基本特点。所以，**元注解也就是对注解进行基本信息设置的注解。**

常用的元注解：

- @Target，用于描述注解的使用范围，例如用在类上面还是方法上面
- @Retention，用于描述注解的保存策略，是保留到源代码中、Class文件中、还是加载到内存中
- @Documented，用于描述该注解将会被javadoc生产到API文档中
- @Inherited，用于表示某个被标注的类型是被继承的，如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

例如，

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD,
PACKAGE, PARAMETER, TYPE})
public @interface Deprecated {
}
```

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {

    String[] value();
}
```

```
@Target({TYPE, FIELD, METHOD})
@Retention(RUNTIME)
public @interface Resource {

    String name() default "";


    String lookup() default "";


    Class<?> type() default java.lang.Object.class;


    enum AuthenticationType {
            CONTAINER,
            APPLICATION
    }


    AuthenticationType authenticationType() default AuthenticationType.CONTAINER;


    boolean shareable() default true;


```

```
26      String mappedName() default "";
27
28
29      String description() default "";
30  }
```

2.6 小结

1. 元数据信息：注解本身不会直接影响程序的执行，而是提供了一种用于存储和传递元数据的方式。元数据是关于程序中元素的描述信息，可以包含名称、类型、范围、约束等信息，用于辅助程序的开发、编译和执行。
2. 内置注解：Java提供了一些内置的注解，例如`@Override`、`@Deprecated`、`@SuppressWarnings`等。这些注解用于提供关于方法覆盖、过时方法、警告抑制等信息，编译器和工具可以根据注解来执行相应的处理。
3. 自定义注解：Java也允许开发者自定义注解，通过`@interface`关键字定义新的注解类型。自定义注解可以包含元素（成员变量），可以为这些元素指定默认值，并可以在程序中使用注解，并提取注解中的元素值。
4. 应用领域：注解在Java中被广泛应用于各个领域，例如框架开发、测试工具、持久化框架、Web开发等。通过注解，可以提供更丰富的配置和行为控制，简化了代码的编写和配置。
5. 本章对于注解的掌握仅限于元数据信息的认识、内置注解的认识等，在后期学习反射的时候会完善自定义注解的学习，并且使用反射机制获取注解并进行操作。