# day28基础加强

今日内容

* 泛型
* 注解
* Servlet3.0
* 动态代理
* 类加载器

## 泛型

###### 回顾泛型类

泛型类：具有一个或多个泛型变量的类被称之为泛型类。

|  |
| --- |
| public class A<T> {  private T t;  public A(T t) {  this.t = t;  }  public T get() {  return t;  }  } |

###### 2　泛型方法

泛型方法的特点：

* 方法的参数中会使用泛型变量；
* 方法的返回值中会使用泛型变量。

|  |
| --- |
| public <T> T get(T[] ts) {  return ts[ts.lengt / 2];  } |
| String[] names ={“zhangSan”, “liSo”, “wangWu”};  String name = get(names); |

　　调用泛型方法时无需指定泛型变量，编译器会通过实际参数的类型来识别泛型变量的类型，上例中传递的参数为String[]类型，那么相当于给泛型类型T赋值为String。

###### 3　继承（实现）泛型类（接口）

继承泛型类需要为父类的泛型变量赋值！就好比创建泛型类的对象时需要给泛型变量赋值一样。

|  |
| --- |
| // 创建泛型类对象  List<String> list = new ArrayList<String>(); |
| // 继承泛型类1：子类也是泛型类  public class MyList1<T> extends ArrayList<T> {  …  } |
| // 继承泛型类2：子类不是泛型类  public class MyList2 extends ArrayList<String> {  …  } |

###### 4　通配符

为了说明通配符的作用，我们先看个例子：

|  |
| --- |
| List<Object> list1 = new ArrayList<String>();  List<Object> list2 = new ArrayList<Integer>(); |

上面的调用都是编译不通过的！

这说明想写一个即可以打印list1，又可以打印list2的方法是不可能的！

|  |
| --- |
| public static void fun(List<Object> list) {…} |
| List<String> list1 = new ArrayList<String>();  List<Integer> list2 = new ArrayList<Integer>();  fun(list1);//编译不通过  fun(list2);//编译不通过 |

如果把fun()方法的泛型参数去除，那么就OK了。即不使用泛型！

|  |
| --- |
| public static void fun(List list) {…}//会有一个警告 |
| List<String> list1 = new ArrayList<String>();  List<Integer> list2 = new ArrayList<Integer>();  fun(list1);  fun(list2); |

上面代码是没有错了，但会有一个警告。警告的原因是你没有使用泛型！Java希望大家都去使用泛型。

你可能会说，这里TMD根本就不能使用泛型。

4.1　通配符概述

通配符就是专门处理这一问题的。

|  |
| --- |
| public static void fun(List<?> list) {…} |

上面代码中的“?”就是一个通配符，它只能在“<>”中使用。造成不能把它从“<>”中拿出来。

这时你可以向fun()方法传递List<String>、List<Integer>类型的参数了。当传递List<String>类型的参数时，表示给“?”赋值为String；当传递List<Integer>类型的参数给fun()方法时，表示给“?”赋值为Integer。

4.2　通配符的缺点

**带有通配符的参数不能使用与泛型相关的方法，例如：list.add(“hello”)编译不通过。**

上面的问题是处理了，但通配符也有它的缺点。

在上面例子中，List<?> list参数中的通配符可以被赋任何值，但同时你也不知道通配符被赋了什么值。

当你不知道“?”是什么时，会使你不能使用任何与泛型相关的方法。也就是说fun()方法的参数list不能再使用它的与泛型相关的方法了。例如：list.add(“hello”)是错误的，因为List类的add()方法的参数是T类型，而现在你不知道T是什么类型，你怎么去添加String的东西给list呢？如果使用者在调用fun()方法时传递的不是List<String>，而是List<Integer>时，你添加String当然是不可以的。

当然，还可以调用list的get()方法。就算你不知道“?”是什么类型，但它肯定是Object类型的。所以你可以：Object o = list.get(0);

4.3　通配符的限制

　　通配符只能出现在引用的定义中，而不能出现在创建对象中。例如：new ArrayList<?>()，这是不可以的。ArrayList<?> list = null，这是可以的。

4.4　带有下边界的通配符

List<? extends Number> list;

其中<? extends Number>表示通配符的下边界，即“?”只能被赋值为Number或其子类型。

|  |
| --- |
| public static void fun(List<? extends Number> list) {…} |
| fun(new ArrayList<Integer>());//ok  fun(new ArrayList<Double>());//ok  fun(new ArrayList<String>());//不ok |

当fun()方法的参数为List<? extends Number>后，说明你只能赋值给“?”Number或Number的子类型。

虽然这多了一个限制，但也有好处，因为你可以list的get()方法了。就算你不知道“?”是什么类型，但你知道它一定是Number或Number的子类型。所以：Number num = list.get(0)是可以的。

但是，还是不能调用list.add()方法！

4.5　带有下边界的通配符

List<? super Integer> list;

其中<? super Integer>表示通配符的下边界，即“?”只能被赋值为Integer或其父类型。

|  |
| --- |
| public static void fun(List<? super Integer> list) {…} |
| fun(new ArrayList<Integer>());//ok  fun(new ArrayList<Number>());//ok  fun(new ArrayList<Object>());//ok  fun(new ArrayList<String>());//不ok |

这时再去调用list.get()方法还是只能使用Object类型来接收：Object o = list.get(0)。因为你不知道“?”到底是Integer的哪个父类。

但是你可以调用list.add()方法了，例如：list.add(new Integer(100))是正确的。因为无论“?”是Integer、Number、Object，list.add(new Integer(100))都是正确的。

4.6　通配符小结

1. 方法参数带有通配符会更加通用；

2. 带有通配符类型的对象，被限制了与泛型相关方法的使用；

3. 下边界通配符：可以使用返回值为泛型变量的方法；

4. 上边界通配符：可以使用参数为泛型变量的方法。

###### 5　泛型父类获取子类传递的类型参数

看一个例子：

|  |
| --- |
| public class A<T> {  } |
| pubilc class B extends A<String> {  } |
| public class C extends A<Integer> {  } |

如果你需要在A类中得到子类给T赋值的类型，那么可以使用下面的方法：

|  |
| --- |
| public class A<T> {  public A() {  ParameterizedType pType = (ParameterizedType)this.getClass().getGenericSuperclass();  Class clazz = (Class)pType.getActualTypeArguments()[0];  System.out.println(clazz.getName());  }  } |

## 注解

###### 1　注解的概述

注释你还记得么？开个玩笑而已！

注释是给人看的，而注解是给程序看的！

注释是用来替代配置文件的！你回忆一下，我们以前总是要写一些配置文件，例如web.xml你还记得么？里面要写<servlet>和<servlet-mapping>！谁来读配置文件呢？当然是Tomcat！谁来写配置文件呢？当然是我们来写了！

在Servlet3.0中就可以使用使用注解来代替配置文件，开发者就不用再写配置文件了，而是写注解，然后Tomcat来读取注解。

注解也是类，需要定义了才能使用！

分别在Servlet3.0中有一个注解类为@WebServlet，然后我们就可以在Servlet中使用@WebServlet中使用这个注解了。这个注解就是用来替代<servlet>了。然后Tomcat会通过反射来读取注解中的信息！

###### 2　Java中的注解

Java中的觉注解：

* @Overrid：作用在方法上的注解。当方法不是重写父类的方法时会报错；
* @Deprecated：作用在方法上。标记该方法为作废方法（已过时）；
* @SuppressWarnings：作用在方法上，压制警告。

###### 3　定义注解类

定义注解类不能使用class、enum，也不能使用interface，而是使用@interface。

|  |
| --- |
| public @interface MyAnn{} |

###### 4　使用注解目标

　　注解可以作用在：类（接口或枚举）、属性、方法、构造器、包、参数、局部变量

|  |
| --- |
| **package** cn.itcast.annocation;  @MyAnn  **public** **class** MyClass {  @MyAnn  **private** **int** a;  @MyAnn  **public** MyClass() {}  @MyAnn  **public** **void** fun1() {}  @MyAnn  **public** **void** fun2(@MyAnn String s) {  @MyAnn  **int** n = 10;  }  } |

###### 5　注解的属性

定义注解时也可以给出属性

|  |
| --- |
| **public** **@interface** MyAnn {  String value();  **int** value1();  } |

其中value就是属性！你可能会说，它是一个方法！没错，它是一个方法，但我们非要称之为属性，因为把它当做属性更加好理解。

当为注解指定属性后，那么在使用注解时就必须要给属性赋值了：

|  |
| --- |
| @MyAnn(value1=100,value="hello")  **public** **class** MyClass {  } |

注解的属性还可以有默认值，在使用注解时就可以不给带有默认值的属性赋值了。但没有给出默认值的属性还是要赋值的。

|  |
| --- |
| **public** **@interface** MyAnn {  String value() **default** "hello world";  **int** value1();  } |
| @MyAnn(value1=100)  **public** **class** MyClass {  } |

在使用注解时，如果只给名为value的属性赋值，那么可以不给出属性的名称直接给出值。

|  |
| --- |
| **public** **@interface** MyAnn {  String value() **default** "hello world";  **int** value1() **default** 100;  } |
| @MyAnn()  **public** **class** MyClass {  } |
| @MyAnn(value="hello")  **public** **class** MyClass {  } |
| @MyAnn(value1=200)  **public** **class** MyClass {  } |
| @MyAnn(value="hello",value1=200)  **public** **class** MyClass {  } |
| @MyAnn("hello annocation")  **public** **class** MyClass {  } |
| @MyAnn(300)  **public** **class** MyClass {  } |
| @MyAnn("hello",value1=200)  **public** **class** MyClass {  } |

* 注解的属性后面要有一对圆括号，而且圆括号内不能给出东西。就像是无参的方法一样；
* 注解的属性类型只能是：基本类型、String、Enum、Class、注解类型、以上类型的一维数组类型；
* 注解的属性可以有默认值，例如：int a() default 100;
* 数组的属性默认值：int[] arr() default {1,2,3}，这里不能使用new int[]{1,2,3}
* 使用注解时，在给数组属性赋值时的格式：@MyAnn(arr={1,2,3})；

###### 6　注解的作用目标

在定义注解时可以限制注解的作用目录！例如让注解只能作用在类和方法上。

这需要使用元注解：@Target。该注解有一个属性value，类型为ElementType[]，它是枚举类型。

|  |
| --- |
| public @interface Target {  ElementType[] value();  } |
| public enum ElementType {  TYPE,FIELD,METHOD,PARAMETED,CONSTRUCTOR,LOCAL\_VARIABLE,ANNOCATION\_TYPE,PACKAGE  } |

在定义注解时，可以使用@Target注解来限制注解的作用目标：

|  |
| --- |
| @Target({ElementType.*TYPE*, ElementType.*METHOD*})  **public** **@interface** MyAnn {  } |

这样MyAnn就只能作用在类和方法上的！其中ElementType.TYPE表示类和接口。

|  |
| --- |
| @MyAnn()  **public** **class** MyClass {  @MyAnn()  **private** **int** a;    @MyAnn()  **public** **void** fun() {}  } |

###### 7　注解的保留策略

注解的保留策略是指，注解是只保留在源代码上，还是保留到class文件上，再或者是类在运行时，可以被类加载器加载到内存中。

如果希望注解被反射，那么注解就要保留到运行时，而不是源代码或类文件上。

指定注解的保留策略需要使用元注解@Retention，它有一个value属性，类型为RetentionPolicy类型，RetentionPolicy是枚举类型：

|  |
| --- |
| public @interface Retention {  RetentionPolicy value();  } |
| public enum RetentionPolicy {  SOURCE, CLASS, RUNTIME  } |

下面代码是指定注解保留到运行时

|  |
| --- |
| @Retention(RetentionPolicy.*RUNTIME*)  @Target({ElementType.*TYPE*, ElementType.*METHOD*})  **public** **@interface** MyAnn {  String value() **default** "hello";  **int** value1() **default** 100;  } |

###### 8　通过反射读取注解

读取注解需要使用反射来完成

|  |
| --- |
| @Retention(RetentionPolicy.*RUNTIME*)  @Target({ElementType.*TYPE*, ElementType.*METHOD*})  **public** **@interface** MyAnn {  String value() **default** "hello";  **int** value1() **default** 100;  } |
| @MyAnn(value="hello world", value1=200)  **public** **class** MyClass {  **private** **int** a;    @MyAnn("myMethod")  **public** **void** fun() {}  } |
| **public** **class** Demo1 {  **public** **static** **void** main(String[] args) **throws** Exception {  Class clazz = MyClass.**class**;  MyAnn myAnn = (MyAnn) clazz.getAnnotation(MyAnn.**class**);  System.*out*.println(myAnn.value());  System.*out*.println(myAnn.value1());    Method method = clazz.getMethod("fun");  MyAnn myAnn1 = method.getAnnotation(MyAnn.**class**);  System.*out*.println(myAnn1.value());  System.*out*.println(myAnn1.value1());  }  } |

## Servlet3.0新特性

###### 1　Servlet3.0新特性概述

　　Servlete3.0的主要新特性如下三部分：

* 使用@WebServlet、@WebFilter、@WebListener三个注解来替代web.xml文件中的Servlet、Filter、Listener的配置；
* Servlet异步处理：当Servlet处理比较费时的问题时，这会让客户感觉到很卡。当使用异常处理时可以把已经处理好的内容先一步响应给客户端浏览器，然后使用另一个线程来完成费时的操作，也就是把内容一部分一部分的显示出来；
* 上传组件：不用再使用fileupload等第三方的上传组件，使用Servlet3.0的上传组件会更方便。

###### 2　@WebServlet、@WebFilter、@WebListener

|  |
| --- |
| @WebServlet(  urlPatterns={"/AServlet"},  initParams={@WebInitParam(name="paramName",value="paramValue")},  loadOnStartup=1  )  **public** **class** AServlet **extends** HttpServlet {  **public** **void** init(ServletConfig config) **throws** ServletException {  System.*out*.println(config.getInitParameter("paramName"));  }  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setCharacterEncoding("utf-8");  response.setContentType("text/html;charset=utf-8");  response.getWriter().print("Hello World!");  }  } |
| @WebFilter(urlPatterns={"/\*"},  dispatcherTypes={DispatcherType.*REQUEST*, DispatcherType.*FORWARD*})  **public** **class** AFilter **implements** Filter {  **public** **void** destroy() {}  **public** **void** doFilter(ServletRequest request, ServletResponse response,  FilterChain chain) **throws** IOException, ServletException {  System.*out*.println("start filter");  chain.doFilter(request, response);  System.*out*.println("end filter");  }  **public** **void** init(FilterConfig fConfig) **throws** ServletException {}  } |
| @WebListener()  **public** **class** AListener **implements** ServletContextListener {  **public** **void** contextDestroyed(ServletContextEvent arg0) {  System.*out*.println("服务器关闭了");  }  **public** **void** contextInitialized(ServletContextEvent arg0) {  System.*out*.println("服务器启动了");  }  } |

###### 3　Servlet异步处理

Servlet异步处理就是让Servlet在处理费时的请求时不要阻塞，而是一部分一部分的显示。

也就是说，在使用Servlet异步处理之后，页面可以一部分一部分的显示数据，而不是一直卡，等到请求响应结束后一起显示。

在使用异步处理之前，一定要在@WebServlet注解中给出asyncSupported=true，不然默认Servlet是不支持异步处理的。如果存在过滤器，也要设置@WebFilter的asyncSupportedt=true。

|  |
| --- |
| @WebServlet(urlPatterns = {"/MyServlet"}, asyncSupported=**true**)  **public** **class** MyServlet **extends** HttpServlet {…} |

**注意，响应类型必须是text/html，所以：response.setContentType(“text/html;charset=utf-8”);**

使用异步处理大致可以分为两步：

* Servlet正常响应数据；
* Servlet异常响应数据。

在Servlet正常响应数据时，没什么可说的，可通知response.getWriter().print()来向客户端输出，但输出后要使用response.getWriter().flush()刷新，不然数据只是在缓冲区中，不能向客户端发送数据的。

异步响应数据需要使用request.startAsync()方法获取AsyncContext对象。然后调用AsyncContext对象的start()方法启动异步响应，start()方法需要一个Runnable类型的参数。在Runnable的run()方法中给出异步响应的代码。

|  |
| --- |
| AsyncContext ac = request.startAsyncContext(request, response);  ac.start(new Runnable() {…}); |

注意在异步处理线程中使用response做响应后，要使用response.getWriter().flush()来刷新流，不然数据是不能响应到客户端浏览器的。

|  |
| --- |
| asyncContext.start(**new** Runnable() {  **public** **void** run() {  **for**(**char** i = 'a'; i <= 'z'; i++) {  **try** {  Thread.*sleep*(100);  asyncContext.getResponse().getWriter().print(i + "&nbsp;");  asyncContext.getResponse().getWriter().flush();  } **catch** (Exception e) {  e.printStackTrace();  }  }  asyncContext.complete();  }  }); |

Tomcat需要知道异步响应是否结束，如果响应不结束，虽然客户端浏览器会看到响应的数据，但是鼠标上只是有个圈圈的不行的转啊转的，表示还没有结束响应。Tomcat会等待到超时为止，这个超时的时间可以通过AsyncContext类的getTimeout()方法获取，Tomcat默认为20000毫秒。当然也可以通过setTimeOut()方法设置，以毫秒为单位。ac.setTimeout(1000\*10)。

如果异步线程已经结束了响应，那么可以在异步线程中调用AsyncContext.complete()方法，这样Tomcat就知道异步线程已经完成了工作了。

|  |
| --- |
| @WebServlet(urlPatterns = {"/AServlet"}, asyncSupported=**true**)  **public** **class** AServlet **extends** HttpServlet {  **public** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  response.setContentType("text/html;charset=utf-8");  PrintWriter out = response.getWriter();  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.println("Servlet begin <br>");  out.flush();  **final** AsyncContext asyncContext = request.startAsync(request, response);  asyncContext.setTimeout(1000 \* 20);  asyncContext.start(**new** Runnable() {  **public** **void** run() {  **try** {  Thread.*sleep*(1000);  asyncContext.getResponse().getWriter().print("马上开始" + "<br/>");  asyncContext.getResponse().getWriter().flush();  Thread.*sleep*(2000);  } **catch** (Exception e1) {  }  **for**(**char** i = 'a'; i <= 'z'; i++) {  **try** {  Thread.*sleep*(100);  asyncContext.getResponse().getWriter().print(i + "&nbsp;");  asyncContext.getResponse().getWriter().flush();  } **catch** (Exception e) {  e.printStackTrace();  }  }  asyncContext.complete();  }  });  // asyncContext.start(businessHandleThread);  // 也可以用这种方法启动异步线程  out.println("Servlet end <br>");  }  } |

###### 4　文件上传

Servlet3.0提供了文件上传的处理方案。只需要在Servlet上添加@MultipartConfig注解即可。

|  |
| --- |
| @WebServlet(urlPatterns={"/UploadServlet"})  @MultipartConfig(maxFileSize=1024)  **public** **class** UploadServlet **extends** HttpServlet { … } |

当然也可以为@MultipartConfig注解指定属性值，它有四个属性：

* int filesizeThreshold：指定缓存的大小，当超出这个大小后，文件会保存到磁盘上；
* String location：指定临时文件的目录；
* long maxFilesize：指定上传单个文件的大小限制，如果上传的谁的超出了这个大小，那么就会抛出异常；
* long maxRequestSize：指定整个表单的大小限制。

当在Servlet上使用了@MultipartConfig注解后，那么就可以使用request.getPart(“fieldName”)来获取<input:file>的内容，其中Part表示一个文件表单项。

|  |
| --- |
| <form action=*"/a1/UploadServlet"* method=*"post"* enctype=*"multipart/form-data"*>  用户名：<input type=*"text"* name=*"username"*/><br/>  照　片：<input type=*"file"* name=*"file1"* /><br/>  <input type=*"submit"* value=*"提交"*/>  </form> |
| @WebServlet(urlPatterns={"/UploadServlet"})  @MultipartConfig(maxFileSize=1024 \* 1024)  **public** **class** UploadServlet **extends** HttpServlet {  **public** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  request.setCharacterEncoding("utf-8");  response.setContentType("text/html;charset=utf-8");    String username = request.getParameter("username");  response.getWriter().print("size: " + username + "<br/>");    Part part = request.getPart("file1");  response.getWriter().print("size: " + part.getSize() + "<br/>");  response.getWriter().print("type: " + part.getContentType() + "<br/>");  response.getWriter().print("name: " + part.getName() + "<br/>");  String name = part.getHeader("content-disposition");  String fileNameTmp = name.substring(name.indexOf("filename=")+10);  String fileName = fileNameTmp.substring(0,fileNameTmp.indexOf("\""));    System.*out*.println("fileName: " + fileName);  String savepath = **this**.getServletContext().getRealPath("/uploads");  part.write(savepath + "/" + fileName);  }  } |

## 动态代理

###### 1　学习动态代理的目的

动态代理技术都是在框架中使用，例如：Struts1、Struts2、Spring和Hibernate中都使用了动态代理技术。如果你不想自己写个框架，那么你基本上是用上不动态代理技术的。

我们学习动态代理技术的目的是为了更好的理解框架内部的原理，也就是说是为了将来我们学习框架打基础！

动态代理技术有点小难度！而且明白了动态代理技术可能一时也想不到他适合在什么情况下使用它。这些东西都会在学习框架时渐渐明白。

###### 2　运行时实现指定的接口

想实现某个接口，你需要写一个类，然后在类名字的后面给出“implements”XXX接口。这才是实现某个接口：

|  |
| --- |
| **public** **interface** MyInterface {  **void** fun1();  **void** fun2();  } |
| **public** **class** MyInterfaceImpl **implements** MyInterface {  **public** **void** fun1() {  System.*out*.println("fun1()");  }  **public** **void** fun2() {  System.*out*.println("fun2()");  }  } |

上面的代码对我们来说没有什么新鲜感，我们要说的是动态代理技术可以通过一个方法调用就可以生成一个对指定接口的实现类对象。

|  |
| --- |
| Class[] cs = {MyInterface.**class**};  MyInterface mi = (MyInterface)Proxy.*newProxyInstance*(loader, cs, h); |

上面代码中，Proxy类的静态方法newProxyInstance()方法生成了一个对象，这个对象实现了cs数组中指定的接口。没错，返回值mi是MyInterface接口的实现类。你不要问这个类是哪个类，你只需要知道mi是MyInterface接口的实现类就可以了。你现在也不用去管loader和h这两个参数是什么东东，你只需要知道，Proxy类的静态方法newProxyInstance()方法返回的方法是实现了指定接口的实现类对象，甚至你都没有看见实现类的代码。

动态代理就是在运行时生成一个类，这个类会实现你指定的一组接口，而这个类没有.java文件，是在运行时生成的，你也不用去关心它是什么类型的，你只需要知道它实现了哪些接口即可。

###### 3　newProxyInstance()方法的参数

Proxy类的newInstance()方法有三个参数：

* ClassLoader loader：它是类加载器类型，你不用去理睬它，你只需要知道怎么可以获得它就可以了：MyInterface.class.getClassLoader()就可以获取到ClassLoader对象，没错，只要你有一个Class对象就可以获取到ClassLoader对象；
* Class[] interfaces：指定newProxyInstance()方法返回的对象要实现哪些接口，没错，可以指定多个接口，例如上面例子只我们只指定了一个接口：Class[] cs = {MyInterface.class};
* InvocationHandler h：它是最重要的一个参数！它是一个接口！它的名字叫调用处理器！你想一想，上面例子中mi对象是MyInterface接口的实现类对象，那么它一定是可以调用fun1()和fun2()方法了，难道你不想调用一下fun1()和fun2()方法么，它会执行些什么东东呢？其实无论你调用代理对象的什么方法，它都是在调用InvocationHandler的invoke()方法！

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Class[] cs = {MyInterface.**class**};  ClassLoader loader = MyInterface.**class**.getClassLoader();  InvocationHandler h = **new** InvocationHandler() {  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  System.*out*.println("无论你调用代理对象的什么方法，其实都是在调用invoke()...");  **return** **null**;  }  };  MyInterface mi = (MyInterface)Proxy.*newProxyInstance*(loader, cs, h);  mi.fun1();  mi.fun2();  } |

　　InvocationHandler接口只有一个方法，即invoke()方法！它是对代理对象所有方法的唯一实现。也就是说，无论你调用代理对象上的哪个方法，其实都是在调用InvocationHandler的invoke()方法。

想象中的类：

|  |
| --- |
| **class** X **implements** MyInterface {  **private** InvocationHandler h;  **public** X(InvocationHandler h) {  **this**.h = h;  }    **public** **void** fun1() {  h.invoke();  }  **public** **void** fun2() {  h.invoke();  }  } |

　　注意，X类是我们用来理解代理对象与InvocationHandler之间的关系的，但它是不存在的类。是我们想象出来的！也就是说，它是用来说明，无论你调用代理对象的哪个方法，最终调用的都是调用处理器的invoke()方法。

###### 4　InvocationHandler的invoke()方法

InvocationHandler的invoke()方法的参数有三个：

* Object proxy：代理对象，也就是Proxy.newProxyInstance()方法返回的对象，通常我们用不上它；
* Method method：表示当前被调用方法的反射对象，例如mi.fun1()，那么method就是fun1()方法的反射对象；
* Object[] args：表示当前被调用方法的参数，当然mi.fun1()这个调用是没有参数的，所以args是一个零长数组。

　　最后要说的是invoke()方法的返回值为Object类型，它表示当前被调用的方法的返回值，当然mi.fun1()方法是没有返回值的，所以invoke()返回的就必须是null了。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Class[] cs = {MyInterface.**class**};  ClassLoader loader = MyInterface.**class**.getClassLoader();  InvocationHandler h = **new** InvocationHandler() {  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  System.*out*.println("当前调用的方法是：" + method.getName());  **return** **null**;  }  };  MyInterface mi = (MyInterface)Proxy.*newProxyInstance*(loader, cs, h);  mi.fun1();  mi.fun2();  } |
| 当前调用的方法是：fun1  当前调用的方法是：fun2 |

###### 5　动态代理的用途

动态代理的用途与装饰模式很相似，就是为了对某个对象进行增强。所有使用装饰者模式的案例都可以使用动态代理来替换。

下面我们用一个例子来说明动态代理的用途！

我们来写一个Waiter接口，它只有一个serve()方法。MyWaiter是Waiter接口的实现类：

|  |
| --- |
| **public** **interface** Waiter {  **public** **void** serve();  } |
| **public** **class** MyWaiter **implements** Waiter {  **public** **void** serve() {  System.*out*.println("服务...");  }  } |

现在我们要对MyWaiter对象进行增强，要让它在服务之前以及服务之后添加礼貌用语，即在服务之前说“您好！”，在服务之后说：“很高兴为您服务！”。

|  |
| --- |
| **public** **class** MainApp1 {  **public** **static** **void** main(String[] args) {  ClassLoader loader = MainApp1.**class**.getClassLoader();  Class[] cs = {Waiter.**class**};  Waiter target = **new** MyWaiter();  MyInvocationHandler h = **new** MyInvocationHandler(target);  Waiter waiter = (Waiter)Proxy.*newProxyInstance*(loader, cs, h);  waiter.serve();  }  }  **class** MyInvocationHandler **implements** InvocationHandler {  **public** Waiter target;  **public** MyInvocationHandler(Waiter target) {  **this**.target = target;  }  **public** Object invoke(Object proxy, Method method, Object[] args)  **throws** Throwable {  System.*out*.println("您好！");  Object result = method.invoke(target, args);  System.*out*.println("很高兴为您服务！");  **return** result;  }  } |

## 类加载器

###### 1　什么是类加载器

类加载器就是用来加载类的东西！类加载器也是一个类：ClassLoader

类加载器可以被加载到内存，是通过类加载器完成的！Java提供了三种类加载器，分别是：

* bootstrap classloader：引导类加载器，加载rt.jar中的类；
* sun.misc.Launcher$ExtClassLoader：扩展类加载器，加载lib/ext目录下的类；
* sun.misc.Launcher$AppClassLoader：系统类加载器，加载CLASSPATH下的类，即我们写的类，以及第三方提供的类。

通常情况下，Java中所有类都是通过这三个类加载器加载的。

类加载器之间存在上下级关系，系统类加载器的上级是扩展类加载器，而扩展类加载器的上级是引导类加载器。

###### 2　JVM眼中的相同的类

在JVM中，不可能存在一个类被加载两次的事情！一个类如果已经被加载了，当再次试图加载这个类时，类加载器会先去查找这个类是否已经被加载过了，如果已经被加载过了，就不会再去加载了。

但是，如果一个类使用不同的类加载器去加载是可以出现多次加载的情况的！也就是说，在JVM眼中，相同的类需要有相同的class文件，以及相同的类加载器。当一个class文件，被不同的类加载器加载了，JVM会认识这是两个不同的类，这会在JVM中出现两个相同的Class对象！甚至会出现类型转换异常！

###### 3　类加载器的代理模式

当系统类加载器去加载一个类时，它首先会让上级去加载，即让扩展类加载器去加载类，扩展类加载器也会让它的上级引导类加载器去加载类。如果上级没有加载成功，那么再由自己去加载！

例如我们自己写的Person类，一定是存放到CLASSPATH中，那么一定是由系统类加载器来加载。当系统类加载器来加载类时，它首先把加载的任务交给扩展类加载去，如果扩展类加载器加载成功了，那么系统类加载器就不会再去加载。这就是代理模式了！

相同的道理，扩展类加载器也会把加载类的任务交给它的“上级”，即引导类加载器，引导类加载器加载成功，那么扩展类加载器也就不会再去加载了。引导类加载器是用C语言写的，是JVM的一部分，它是最上层的类加载器了，所以它就没有“上级了”。它只负责去加载“内部人”，即JDK中的类，但我们知道Person类不是我们自己写的类，所以它加载失败。

当扩展类加载器发现“上级”不能加载类，它就开始加载工作了，它加载的是lib\ext目录下的jar文件，当然，它也会加载失败，所以最终还是由系统类加载器在CLASSPATH中去加载Person，最终由系统类加载器加载到了Person类。

代理模式保证了JDK中的类一定是由引导类加载加载的！这就不会出现多个版本的类，这也是代理模式的好处。

###### 3　自定义类加载器 （了解）

我们也可以通过继承ClassLoader类来完成自定义类加载器，自类加载器的目的一般是为了加载网络上的类，因为这会让class在网络中传输，为了安全，那么class一定是需要加密的，所以需要自定义的类加载器来加载（自定义的类加载器需要做解密工作）。

ClassLoader加载类都是通过loadClass()方法来完成的，loadClass()方法的工作流程如下：

* 调用findLoadedClass()方法查看该类是否已经被加载过了，如果该没有加载过，那么这个方法返回null；
* 判断findLoadedClass()方法返回的是否为null，如果不是null那么直接返回，这可以避免同一个类被加载两次；
* 如果findLoadedClass()返回的是null，那么就启动代理模式（委托机制），即调用上级的loadClass()方法，获取上级的方法是getParent()，当然上级可能还有上级，这个动作就一直向上走；
* 如果getParent().loadClass()返回的不是null，这说明上级加载成功了，那么就加载结果；
* 如果上级返回的是null，这说明需要自己出手了，这时loadClass()方法会调用本类的findClass()方法来加载类；
* 这说明我们只需要重写ClassLoader的findClass()方法，这就可以了！如果重写了loadClass()方法覆盖了代理模式！

OK，通过上面的分析，我们知道要自定义一个类加载器，只需要继承ClassLoader类，然后重写它的findClass()方法即可。那么在findClass()方法中我们要完成哪些工作呢？

* 找到class文件，把它加载到一个byte[]中；
* 调用defineClass()方法，把byte[]传递给这个方法即可。

FileSystemClassLoader

|  |
| --- |
| **public** **class** FileSystemClassLoader **extends** ClassLoader {  **private** String classpath;    **public** FileSystemClassLoader() {}    **public** FileSystemClassLoader(String classpath) {  **this**.classpath = classpath;  }    @Override  **public** Class<?> findClass(String name) **throws** ClassNotFoundException {  **try** {  **byte**[] datas = getClassData(name);  **if**(datas == **null**) {  **throw** **new** ClassNotFoundException("类没有找到：" + name);  }  **return** **this**.defineClass(name, datas, 0, datas.length);  } **catch** (IOException e) {  e.printStackTrace();  **throw** **new** ClassNotFoundException("类找不到：" + name);  }  }  **private** **byte**[] getClassData(String name) **throws** IOException {  name = name.replace(".", "\\") + ".class";  File classFile = **new** File(classpath, name);  **return** FileUtils.*readFileToByteArray*(classFile);  }  } |
| ClassLoader loader = **new** FileSystemClassLoader("F:\\classpath");  Class clazz = loader.loadClass("cn.itcast.utils.CommonUtils");  Method method = clazz.getMethod("md5", String.**class**);  String result = (String) method.invoke(**null**, "qdmmy6");  System.*out*.println(result); |

###### 4　Tomcat的类加载器

Tomcat会为每个项目提供一个类加载器，Tomcat提供的类加载器负责加载自己项目下的类，即WEB-INF\lib和WEB-INF\classes下的类。但Tomcat提供的类加载器不会使用传统的代理模式，而是自己先去加载，如果加载不到，再使用代理模式。

Tomcat提供的类加载器有这样一个好处，就是可以使自己项目下的类优先被加载！

