### Java注解Annotation基础

目录

[Java注解Annotation基础 1](#_Toc444798668)

[1 定义Annotation 1](#_Toc444798669)

[1.1 成员变量 1](#_Toc444798670)

[1.2 元注解 3](#_Toc444798671)

[1.2.1 @Retention 3](#_Toc444798672)

[1.2.2 @Target 4](#_Toc444798673)

[1.2.3 @Documented 4](#_Toc444798674)

[1.2.4 @Inherited 5](#_Toc444798675)

[1.3 基本Annotation 6](#_Toc444798676)

[2 提取Annotation信息（反射） 7](#_Toc444798677)

[2.1 java.lang.annotation.Annotation 7](#_Toc444798678)

[2.2 java.lang.reflect.AnnotatedElement 7](#_Toc444798679)

[2.3 示例 8](#_Toc444798680)

[2.3.1 标记Annotation 8](#_Toc444798681)

[2.3.2 元数据Annotation 10](#_Toc444798682)

[3 注解本质 12](#_Toc444798683)

[4 注解的意义 15](#_Toc444798684)

[5 Java注解的基础知识点导图 17](#_Toc444798685)

[6奥拉星常见注解处理器解析 18](#_Toc444798686)

[6.1 Command、SimpleCommand 18](#_Toc444798687)

[6.2 BeginDate、EndDate 19](#_Toc444798688)

[6.3 EventInject 20](#_Toc444798689)

[6.4 相关的类 22](#_Toc444798690)

**Annotation（注解）就是Java提供了一种元程序中的元素关联任何信息和着任何元数据（metadata）的途径和方法。Annotion(注解)是一个接口，程序可以通过反射来获取指定程序元素的Annotion对象，然后通过Annotion对象来获取注解里面的元数据**。

Annotation提供了一种为程序元素（包、类、构造器、方法、成员变量、参数、局域变量）设置元数据的方法。**Annotation不能运行，它只有成员变量，没有方法**。Annotation跟public、final等修饰符的地位一样，都是程序元素的一部分，Annotation不能作为一个程序元素使用。

# ****1 定义Annotation****

定义新的Annotation类型使用@interface关键字（在原有interface关键字前增加@符号）。定义一个新的Annotation类型与定义一个接口很像，例如：

**public @interface Test{**

**}**

|  |  |
| --- | --- |
|  |  |

定义完该Annotation后，就可以在程序中使用该Annotation。使用Annotation，非常类似于public、final这样的修饰符，通常，会把Annotation另放一行，并且放在所有修饰符之前。例如：

**@Test**

**public class MyClass{**

**....**

**}**

|  |
| --- |
|  |

## 1.1 成员变量

Annotation只有成员变量，没有方法。Annotation的成员变量在Annotation定义中以“无形参的方法”形式来声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型。

例如：

**public @interface MyTag{**

**string name();**

**int age();**

**}**

|  |
| --- |
|  |

示例中定义了2个成员变量，这2个成员变量以方法的形式来定义。一旦在Annotation里定义了成员变量后，使用该Annotation时就应该为该Annotation的成员变量指定值。例如：

**public class Test{**

**@MyTag(name="红薯"，age=30)**

**public void info(){**

**......**

**}**

**}**

|  |
| --- |
|  |

也可以在定义Annotation的成员变量时，为其指定默认值，指定成员变量默认值使用default关键字。示例：

**public @interface MyTag{**

**string name() default "双大";**

**int age() default 18;**

**}**

|  |  |
| --- | --- |
|  |  |

如果Annotation的成员变量已经指定了默认值，使用该Annotation时可以不为这些成员变量指定值，而是直接使用默认值。例如：

**public class Test{**

**@MyTag**

**public void info(){**

**......**

**}**

**}**

Annotation类型里面的参数该怎么设定:   
　　第一,只能用public或默认(default)这两个访问权修饰.例如,String value()这里把方法设为defaul默认类型；　 　  
　　第二,参数成员只能用基本类型byte,short,char,int,long,float,double,boolean八种基本数据类型和 String,Enum,Class,annotations等数据类型,以及这一些类型的数组.例如,String value();这里的参数成员就为String;　　  
　　第三,如果只有一个参数成员,最好把参数名称设为"value",后加小括号.例:下面的例子MyAnn注解就只有一个参数成员。

**public @interface MyAnn {**

**public String value();**

**}**

如果Annotation里只有一个名为“value“的成员变量，使用该Annotation时，可以直接使用XXX(val)形式为value成员变量赋值，无须使用name=val形式

**@MyAnn("hello")**

**public class MyDemo {**

**}**

根据Annotation是否包含成员变量，可以把Annotation分为如下两类：

* 标记Annotation：没有成员变量的Annotation被称为标记。这种Annotation仅用自身的存在与否来为我们提供信息，例如@override等。
* 元数据Annotation：包含成员变量的Annotation。因为它们可以接受更多的元数据，因此被称为元数据Annotation。

## 1.2 元注解

在定义Annotation时，也可以使用JDK提供的元注解来修饰Annotation定义。JDK提供了如下4个元注解（注解的注解，不是上述的”元数据Annotation“）：

* **@Retention**
* **@Target**
* **@Documented**
* **@Inherited**

### 1.2.1 @Retention

@Retention用于指定Annotation可以保留多长时间。

@Retention包含一个名为“value”的成员变量，该value成员变量是RetentionPolicy枚举类型。使用@Retention时，必须为其value指定值。value成员变量的值只能是如下3个：

* RetentionPolicy.SOURCE：Annotation只保留在源代码中，编译器编译时，直接丢弃这种Annotation。
* RetentionPolicy.CLASS：编译器把Annotation记录在class文件中。当运行Java程序时，JVM中不再保留该Annotation。
* RetentionPolicy.RUNTIME：编译器把Annotation记录在class文件中。当运行Java程序时，JVM会保留该Annotation，程序可以通过反射获取该Annotation的信息。

示例：

**package com.demo1;**

**import java.lang.annotation.Retention;**

**import java.lang.annotation.RetentionPolicy;**

**//name=value形式**

**//@Retention(value=RetentionPolicy.RUNTIME)**

**//直接指定**

**@Retention(RetentionPolicy.RUNTIME)**

**public @interface MyTag{**

**String name() default "我兰";**

**}**

|  |  |
| --- | --- |
|  |  |

### 1.2.2 @Target

@Target指定Annotation用于修饰哪些程序元素。@Target也包含一个名为”value“的成员变量，该value成员变量类型为ElementType[ ]，ElementType为枚举类型，值有如下几个：

* ElementType.TYPE：能修饰类、接口或枚举类型
* ElementType.FIELD：能修饰成员变量
* ElementType.METHOD：能修饰方法
* ElementType.PARAMETER：能修饰参数
* ElementType.CONSTRUCTOR：能修饰构造器
* ElementType.LOCAL\_VARIABLE：能修饰局部变量
* ElementType.ANNOTATION\_TYPE：能修饰注解
* ElementType.PACKAGE：能修饰包

示例1（单个ElementType）：

**package com.demo1;**

**import java.lang.annotation.ElementType;**

**import java.lang.annotation.Target;**

**@Target(ElementType.FIELD)**

**public @interface AnnTest {**

**String name() default "sunchp";**

**}**

|  |  |
| --- | --- |
|  |  |

**示例2（多个ElementType）：**

**package com.demo1;**

**import java.lang.annotation.ElementType;**

**import java.lang.annotation.Target;**

**@Target({ ElementType.FIELD, ElementType.METHOD })**

**public @interface AnnTest {**

**String name() default "sunchp";**

**}**

### 1.2.3 @Documented

如果定义注解A时，使用了@Documented修饰定义，则在用javadoc命令生成API文档后，所有使用注解A修饰的程序元素，将会包含注解A的说明。

示例：

**@Documented**

**public @interface Testable {**

**}**

**public class Test {**

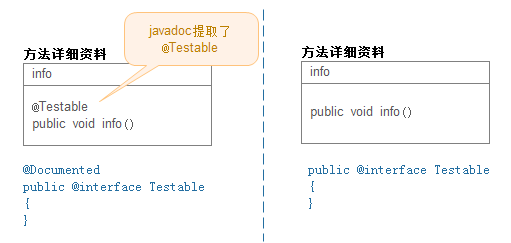
**@Testable**

**public void info() {**

**}**

**}**

|  |  |
| --- | --- |
|  |  |



### 1.2.4 @Inherited

@Inherited指定Annotation具有继承性。

示例：

|  |  |
| --- | --- |
|  |  |

**package com.demo2;**

**import java.lang.annotation.ElementType;**

**import java.lang.annotation.Inherited;**

**import java.lang.annotation.Retention;**

**import java.lang.annotation.RetentionPolicy;**

**import java.lang.annotation.Target;**

**@Target(ElementType.TYPE)**

**@Retention(RetentionPolicy.RUNTIME)**

**@Inherited**

**public @interface MyTag{**

**}**

**package com.demo2;**

**@MyTag**

**public class Base {**

**}**

**package com.demo2;**

**//SubClass只是继承了Base类**

**//并未直接使用@MyTag注解修饰**

**public class SubClass extends Base {**

**public static void main(String[] args) {**

**System.out.println(SubClass.class.isAnnotationPresent(MyTag.class));**

**}**

|  |  |
| --- | --- |
|  |  |

**}**

|  |  |
| --- | --- |
|  |  |

示例中Base使用@MyTag修饰，SubClass继承Base，而且没有直接使用@MyTag修饰，但是因为MyTag定义时，使用了@Inherited修饰，具有了继承性，所以运行结果为true。

如果MyTag注解没有被@Inherited修饰，则运行结果为：false。

## 1.3 基本Annotation

JDK默认提供了如下几个基本Annotation：

* **@Override**

限定重写父类方法。对于子类中被@Override 修饰的方法，如果存在对应的被重写的父类方法，则正确；如果不存在，则报错。@Override 只能作用于方法，不能作用于其他程序元素。

* **@Deprecated**

用于表示某个程序元素（类、方法等）已过时。如果使用被@Deprecated修饰的类或方法等，编译器会发出警告。

* **@SuppressWarning**

抑制编译器警告。指示被@SuppressWarning修饰的程序元素（以及该程序元素中的所有子元素，例如类以及该类中的方法.....）取消显示指定的编译器警告。例如，常见的@SuppressWarning（value="unchecked"）

* **@SafeVarargs**

@SafeVarargs是JDK 7 专门为抑制“堆污染”警告提供的。

# ****2 提取Annotation信息（反射）****

当开发者使用了Annotation修饰了类、方法、Field等成员之后，这些Annotation不会自己生效，必须由开发者提供相应的代码来提取并处理Annotation信息。这些处理提取和处理Annotation的代码统称为APT（Annotation Processing Tool）。

JDK主要提供了两个类，来完成Annotation的提取：

* java.lang.annotation.Annotation接口：这个接口是所有Annotation类型的父接口（后面会分析Annotation的本质，Annotation本质是接口，而java.lang.annotation.Annotation接口是这些接口的父接口）。
* java.lang.reflect.AnnotatedElement接口：该接口代表程序中可以被注解的程序元素。

## 2.1 java.lang.annotation.Annotation

java.lang.annotation.Annotation接口源码:

**package java.lang.annotation;**

**public interface Annotation {**

**boolean equals(Object obj);**

**int hashCode();**

**String toString();**

**Class<? extends Annotation> annotationType();**

**}**

java.lang.annotation.Annotation接口的主要方法是annotationType( )，用于返回该注解的java.lang.Class。

## 2.2 java.lang.reflect.AnnotatedElement

java.lang.reflect.AnnotatedElement接口源码：

**package java.lang.reflect;**

**import java.lang.annotation.Annotation;**

**public interface AnnotatedElement {**

**boolean isAnnotationPresent(Class<? extends Annotation> annotationClass);**

**<T extends Annotation> T getAnnotation(Class<T> annotationClass);**

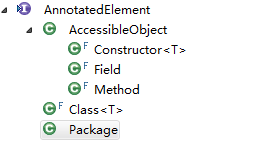
**Annotation[] getAnnotations();**

**Annotation[] getDeclaredAnnotations();**

**}**

主要方法有：

* **isAnnotationPresent(Class<? extends Annotation> annotationClass)：**判断该程序元素上是否存在指定类型的注解，如果存在则返回true，否则返回false。
* **getAnnotation(Class<T> annotationClass)：**返回该程序元素上存在的指定类型的注解，如果该类型的注解不存在，则返回null
* **Annotation[] getAnnotations()：**返回该程序元素上存在的所有注解。

java.lang.reflect.AnnotatedElement接口是所有程序元素（例如java.lang.Class、java.lang.reflect.Method、java.lang.reflect.Constructor等）的父接口。类图结构如下：

所以程序通过**反射**获取了某个类的AnnotatedElement对象（例如，A类method1()方法的java.lang.reflect.Method对象）后，就可以调用该对象的isAnnotationPresent( )、getAnnotation( )等方法来访问注解信息。

**为了获取注解信息，必须使用反射知识。**

**PS：如果想要在运行时提取注解信息，在定义注解时，该注解必须使用@Retention(RetentionPolicy.RUNTIME)修饰。**

## 2.3 示例

### 2.3.1 标记Annotation

给定一个类的全额限定名，加载类，并列出该类中被注解@MyTag修饰的方法和没被修饰的方法。

注解定义：

**package com.demo1;**

**import java.lang.annotation.ElementType;**

**import java.lang.annotation.Retention;**

**import java.lang.annotation.RetentionPolicy;**

**import java.lang.annotation.Target;**

**@Retention(RetentionPolicy.RUNTIME)**

**@Target(ElementType.METHOD)**

**public @interface MyTag {**

**}**

|  |  |
| --- | --- |
|  |  |

注解处理：

**package com.demo1;**

**import java.lang.reflect.Method;**

**public class ProcessTool {**

**public static void process(String clazz) {**

**Class targetClass = null;**

**try {**

**targetClass = Class.forName(clazz);**

**} catch (ClassNotFoundException e) {**

**e.printStackTrace();**

**}**

**for (Method m : targetClass.getMethods()) {**

**if (m.isAnnotationPresent(MyTag.class)) {**

**System.out.println("被MyTag注解修饰的方法名：" + m.getName());**

**} else {**

**System.out.println("没被MyTag注解修饰的方法名：" + m.getName());**

**}**

**}**

**}**

**}**

测试类：

**package com.demo1;**

**public class Demo {**

**public static void m1() {**

**}**

**@MyTag**

**public static void m2() {**

**}**

**}**

**package com.demo1;**

**public class Test {**

**public static void main(String[] args) {**

**ProcessTool.process("com.demo1.Demo");**

**}**

**}**

运行结果：

**没被MyTag注解修饰的方法名：m1**

**被MyTag注解修饰的方法名：m2**

**没被MyTag注解修饰的方法名：wait**

**没被MyTag注解修饰的方法名：wait**

**没被MyTag注解修饰的方法名：wait**

**没被MyTag注解修饰的方法名：equals**

**没被MyTag注解修饰的方法名：toString**

**没被MyTag注解修饰的方法名：hashCode**

**没被MyTag注解修饰的方法名：getClass**

**没被MyTag注解修饰的方法名：notify**

**没被MyTag注解修饰的方法名：notifyAll**

|  |  |
| --- | --- |
|  |  |

### 2.3.2 元数据Annotation

给定一个类的全额限定名，加载类，找出被注解MyTag修饰的方法，并输出每个方法的MyTag注解的属性。

注解定义：

**package com.demo1;**

**import java.lang.annotation.ElementType;**

**import java.lang.annotation.Retention;**

**import java.lang.annotation.RetentionPolicy;**

**import java.lang.annotation.Target;**

**@Retention(RetentionPolicy.RUNTIME)**

**@Target(ElementType.METHOD)**

**public @interface MyTag {**

**String name() default "我兰";**

**int age() default 18;**

**}**

注解处理：

**package com.demo1;**

**import java.lang.reflect.Method;**

**public class ProcessTool {**

**public static void process(String clazz) {**

**Class targetClass = null;**

**try {**

**targetClass = Class.forName(clazz);**

**} catch (ClassNotFoundException e) {**

**e.printStackTrace();**

**}**

**for (Method m : targetClass.getMethods()) {**

**if (m.isAnnotationPresent(MyTag.class)) {**

**MyTag tag = m.getAnnotation(MyTag.class);**

**System.out.println("方法" + m.getName() + "的MyTag注解内容为：" + tag.name() + "，" + tag.age());**

**}**

**}**

**}**

**}**

测试类：

|  |  |
| --- | --- |
|  |  |

**package com.demo1;**

**public class Demo {**

**public static void m1() {**

**}**

**@MyTag**

**public static void m2() {**

**}**

**@MyTag(name = "红薯")**

**public static void m3() {**

**}**

**@MyTag(name = "红薯", age = 30)**

**public static void m4() {**

**}**

**}**

**package com.demo1;**

**public class Test {**

**public static void main(String[] args) {**

**ProcessTool.process("com.demo1.Demo");**

**}**

**}**

运行结果：

**方法m2的MyTag注解内容为：我兰，18**

**方法m3的MyTag注解内容为：红薯，18**

**方法m4的MyTag注解内容为：红薯，30**

|  |  |
| --- | --- |
|  |  |

若要获取注解中的成员变量值，直接调用注解对象的"成员变量民( )"形式的方法就行，例如示例中的tag.name()等。

PS：在编译器编译注解定义时，自动在class文件中，添加与成员变量同名的抽象方法，用于反射时获取成员变量的值。

通过上面的示例可以看出，其实Annotation十分简单，它是对源代码增加的一些特殊标记，这些特殊标记可通过反射获取，当程序获取这些特殊标记后，程序可以做出相应的处理（当然也可以完全忽略这些Annotation）。

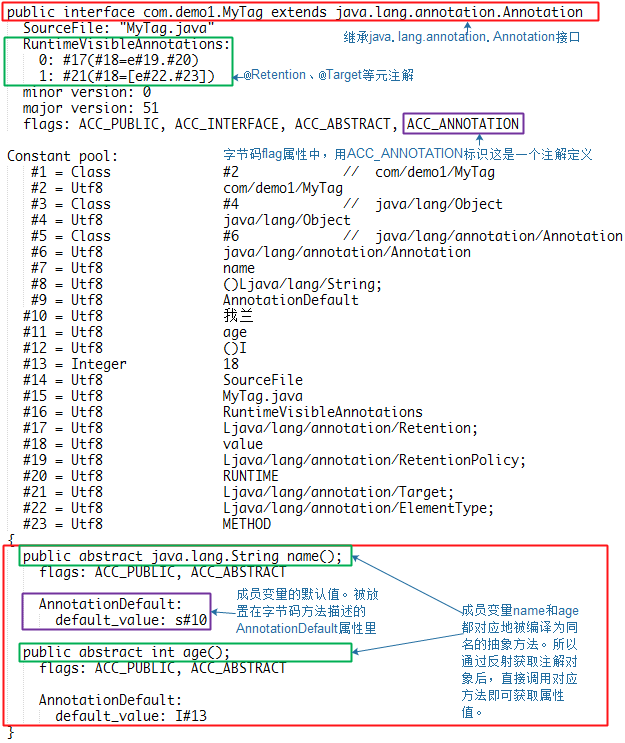
# **3 注解本质**

对于示例”2.3.2 元数据Annotation“中的MyTag注解，在编译后，生成一个MyTag.class文件。反编译该class文件：

javap -verbose -c MyTag.class > m.txt

|  |  |
| --- | --- |
|  |  |

MyTag注解的字节码为：



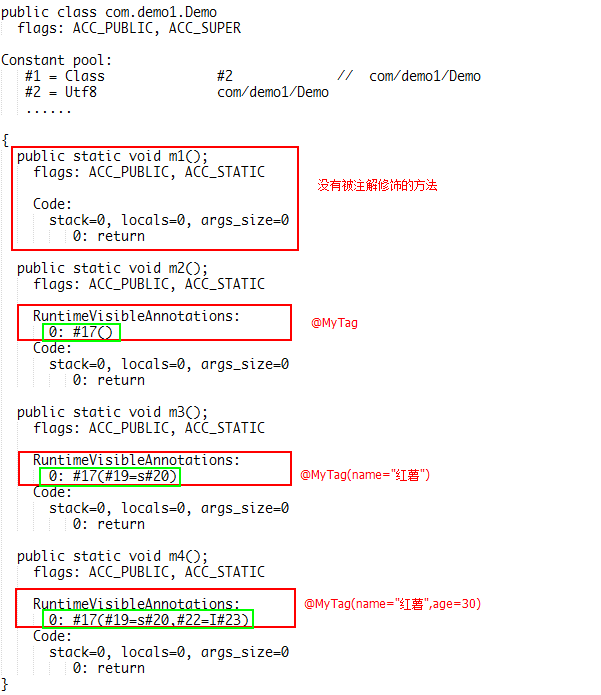
通过分析字节码可知：

* 注解实质上会被编译器编译为接口，并且继承java.lang.annotation.Annotation接口。
* 注解的成员变量会被编译器编译为同名的抽象方法。
* 根据Java的class文件规范，class文件中会在程序元素的属性位置记录注解信息。例如，RuntimeVisibleAnnotations属性位置，记录修饰该类的注解有哪些；flags属性位置，记录该类是不是注解；在方法的AnnotationDefault属性位置，记录注解的成员变量默认值是多少。

我们再反编译下示例”2.3.2 元数据Annotation“中的Demo测试类，查看下”被注解修饰的方法是怎样记录自己被注解修饰的“：

javap -verbose -c Demo.class > d.txt

|  |  |
| --- | --- |
|  |  |

反编译结果如下：

通过字节码可知：

在字节码文件中，每个方法都有RuntimeVisibleAnnotations属性位置，用来放置注解和注解的成员变量赋值。JVM在解析class文件时，会解析RuntimeVisibleAnnotations属性，并新建相应类型的注解对象，并将成员变量赋值。

如果要明白JVM对注解的运行机制，需要对class文件的格式规范有一定了解。（[资料](http://www.blogjava.net/DLevin/archive/2011/09/05/358035.html)）

# **4 注解的意义**

为编译器提供辅助信息 — Annotations可以为编译器提供而外信息，以便于检测错误，抑制警告等.

编译源代码时进行而外操作 — 软件工具可以通过处理Annotation信息来生成原代码，xml文件等等.

运行时处理 — 有一些annotation甚至可以在程序运行时被检测，使用.总之，注解是一种元数据，起到了”描述，配置“的作用

# http://images.cnitblog.com/blog/34483/201304/25200814-475cf2f3a8d24e0bb3b4c442a4b44734.jpg**5 Java注解的基础知识点导图**

# 6奥拉星常见注解处理器解析

## 6.1 Command、SimpleCommand

/\*\*

\*

\* **@author** linchengnan

\*

\*/

**public** **class** ResponderClassWrapper {

**private** **volatile** **boolean** called = **false**;

**private** **volatile** **boolean** init = **false**;

**private** **final** Class<?> clazz;

**private** **final** **boolean** initOnStartUp = **true**;

**private** **final** HashMap<HolidayExtension, HolidayResponder> responderMap = **new** HashMap<HolidayExtension, HolidayResponder>();

**private** **final** HashMap<String, Method> cmdMap = **new** HashMap<String, Method>();

/\*@Command @SimpleCommand注解处理器

\*/

**public** **ResponderClassWrapper**(Class<?> clazz) {

Command注解处理:将被Command注解的方法注册到cmcMap哈希表

**this**.clazz = clazz;

Method[] **methods** = clazz.getMethods();

**for** (Method **m** : methods) {

Command **cmd** = m.getAnnotation(Command.**class**);

**if** (cmd != **null**) {

**if** (cmdMap.put(cmd.value(), m) != **null**) {

**throw** **new** RuntimeException(String.*format*("类'%s'有重复监听的命令'%s'!", clazz.getSimpleName(), cmd.value()));

}

}

SimpleCommand **smd** = m.getAnnotation(SimpleCommand.**class**);

**if** (smd != **null**) {

String **command** = clazz.getSimpleName() + "\_" + smd.value();

**if** (cmdMap.put(command, m) != **null**) {

**throw** **new** RuntimeException(String.*format*("类'%s'有重复监听的命令'%s'!", clazz.getSimpleName(), command));

}

}

}

SimpleCommand注解处理:将被SimpleCommand注解的方法注册到cmcMap哈希表

}

/\*\*

\* 调用注册在cmcMap哈希表中的方法

\* @param cmd : 命令头

\*/

获取holidayExtension + cmd对应的方法

**public** **void** **handleCall**(HolidayExtension holidayExtension, String cmd, ActionscriptObject ao, User u, **int** fromRoom) {

HolidayResponder **responder** = responderMap.get(holidayExtension);

**if** (responder == **null** && !initOnStartUp) {

**if**(!init){

runDelayedInit();

}

responder = responderMap.get(holidayExtension);

}

Method **m** = cmdMap.get(cmd);

**try** {

m.invoke(responder, cmd, ao, u, fromRoom);//调用m方法

**if**(!called){

called = **true**;

DbManager.*getWorkDb*().executeCommand("update HolidayResponderDefine set LastCalledDate=CURDATE() WHERE SimpleClassName=?", **new** Object[]{clazz.getSimpleName()});

}

} **catch** (InvocationTargetException **e**) {

Throwable **t** = e.getCause();

Logger.*error*("handleRequest eror :" + cmd + "/" + m.getName(), t != **null** ? t : e);

} **catch** (Exception **e**) {

Logger.*error*("handleRequest eror :", e);

}

}

}

## 6.2 BeginDate、EndDate

/\*\*

\*

\* @author linchengnan

\*

\*/

public class ResponderManager {

private static final ResponderManager instance = new ResponderManager();

private final List<ResponderClassWrapper> responders = new ArrayList<ResponderClassWrapper>();

private final HashMap<String, ResponderClassWrapper> cmdReferrence = new HashMap<String, ResponderClassWrapper>();

private static void removeInvalidClasses(List<Class<? extends HolidayResponder>> subClasses) {

for (Iterator<Class<? extends HolidayResponder>> iterator = subClasses.iterator(); iterator.hasNext();) {

Class<?> clazz = iterator.next();

if(clazz.getAnnotation(Deprecated.class) != null){

iterator.remove();

continue;

}

LocationAt annotation = clazz.getAnnotation(LocationAt.class);

if(annotation != null && !LocationService.isCurrentLocationEqual(annotation.value())){

iterator.remove();

continue;

BeginDate注解处理:移除在开始时间以前的类

}

BeginDate beginDate = clazz.getAnnotation(BeginDate.class);

if(beginDate != null && System.currentTimeMillis() < DateUtil.parseYMD(beginDate.value()).getTime()){

iterator.remove();

continue;

}

EndDate endDate = clazz.getAnnotation(EndDate.class);

if(endDate != null && System.currentTimeMillis() >= DateUtil.parseYMD(endDate.value()).getTime()){

iterator.remove();

continue;

}

}

}

EndDate注解处理:移除在结束时间以后的类

}

## 6.3 EventInject

public class EventDispatcher {

private static Logger logger = Logger.getLogger(String.format("ServerLogger.%s", EventDispatcher.class.getName()));

private static HashMap<EventKeyWrap, ArrayList<MethodWrap>> eventController = new HashMap<EventKeyWrap, ArrayList<MethodWrap>>();

/\*\* EventInject处理器

\* @param clazz 事件注入的类对象

\*/

public static void listen(String zoneName, Object clazz) {

Method[] methods = clazz.getClass().getDeclaredMethods();

for (Method m : methods) {

获取eventController中key=value+zoneName的value，如果没有则插入key

EventInject ei = m.getAnnotation(EventInject.class);

if (ei != null) {

EventKeyWrap ekw = new EventKeyWrap();

ekw.eventClazz = ei.value();

ekw.zoneName = zoneName;

ArrayList<MethodWrap> array = eventController.get(ekw);

if (array == null) {

array = new ArrayList<MethodWrap>();

eventController.put(ekw, array);

}

if (!checkHadRegister(array, clazz)) {

如果未注册，把被注解的类和方法注册到eventController中

MethodWrap mw = new MethodWrap();

mw.method = m;

mw.inv = clazz;

array.add(mw);

}

}

}

}

//分发（抛出）事件

public static void dispatch(Event event) {

获取eventController中key=value+zoneName的value，如果存在则调用被注解的方法

EventKeyWrap ekw = new EventKeyWrap();

ekw.eventClazz = event.getClass();

ekw.zoneName = event.u.getZoneName();

ArrayList<MethodWrap> methodWrapArr = eventController.get(ekw);

if (methodWrapArr == null || methodWrapArr.size() == 0) {

return;

}

for (MethodWrap ivk : methodWrapArr) {

try {

ivk.method.invoke(ivk.inv, event);

} catch (IllegalArgumentException e) {

logger.error(String.format("%s.%s", ivk.inv.getClass().getName(), event.getClass().getName()), e);

} catch (IllegalAccessException e) {

logger.error(String.format("%s.%s", ivk.inv.getClass().getName(), event.getClass().getName()), e);

} catch (InvocationTargetException e) {

logger.error(String.format("%s.%s", ivk.inv.getClass().getName(), event.getClass().getName()), e);

} catch (Exception e) {

logger.error(String.format("%s.%s", ivk.inv.getClass().getName(), event.getClass().getName()), e);

}

}

}

}

## 6.4 相关的类

**ResponderManager：**

private ResponderManager() ： 获取"com.altratek.altraserver.extensions" 包下的所有继承HolidayResponder类的对象到list<> subClasses中，移除被注解为无效的的类对象（不在BeginDate、EndDate时间内等对象），遍历subClasses，把所有对象中被Command、SimpleCommand注解的方法提取到HashMap<String, ResponderClassWrapper> 中

public void handleRequest(HolidayExtension holidayExtension, String cmd, ActionscriptObject ao, User u, int fromRoom) ：处理命令请求

public synchronized void register(HolidayExtension holidayExtension) ：注册 "com.altratek.altraserver.extensions"包下的所有被EventInject注解的方法