Java注解总结（摘录）

笔记本：	java基础		
创建时间：	2018/10/2/周二 13:54	**更新时间：**	2018/10/2/周二 13:54
作者：	1634896520@qq.com		
URL：	file:///F:/GIT_resposity/Learning-Notes/java%20EE/Java%E6%B3%A8%E8%A7%A3%E6%80%BB%E7%BB%9...		

Java注解总结（摘录）

前言

注解是java引入的一项非常受欢迎的补充，它提供了一种结构化的，并且具有类型检查能力的新途径，从而使得程序员能够为代码加入元数据，而不会导致代码杂乱且难以阅读。使用注解能够帮助我们避免编写累赘的部署描述文件，以及其他生成的文件。

注解的语法比较简单，除了@符号的使用之外，它基本与java固有的语法一致。但由于java源码中提供的内置注解很少，所以大部分同学对注解都不是很了解，虽然我们都接触过，比如java内置的几种注解：

```
@Override，表示当前的方法定义将覆盖超类中的方法。
@Deprecated，表示当前方法即将废弃，不推荐使用。
@SuppressWarnings，表示忽略编译器的警告信息。
```

但这并不能让我们体会到注解的强大和便利，其实Java还另外提供了四种注解，专门负责新注解的创建。今天我们就来学习下怎么创建和使用注解。

注解类的定义

首先看看一个注解类的定义：

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
boolean primaryKey() default false;
boolean allowNull() default true;
boolean unique() default false;
}
```

除了@符号以外，注解类的定义很像一个空的接口。定义注解时，会需要一些*元注解*，如@Target和@Retention，java提供了四种元注解，定义如下：

@Target：表示该注解可以用于什么地方。

取值（ElementType）包括：
CONSTRUCTOR:用于描述构造器
FIELD:用于描述域
LOCAL_VARIABLE:用于描述局部变量
METHOD:用于描述方法
PACKAGE:用于描述包
PARAMETER:用于描述参数

TYPE:用于描述类、接口(包括注解类型) 或enum声明

@Retention：表示需要在什么级别保存该注解信息。
取值（RetentionPolicy）包括：
SOURCE:在源文件中有效（即源文件保留）
CLASS:在class文件中有效（即class保留）

RUNTIME:在运行时有效（即运行时保留），因此可以通过反射机制读取注解的信息。

@Documented:表示将此注解包含在javadoc中。

@Inherited：表示允许子类继承父类中的注解。

可以看出 定义注解格式为：

```
public @interface 注解名 {定义体}
```

注解类中定义的元素称为*注解元素*，注解元素可用的类型如下：

```
所有基本数据类型（int,float,boolean,byte,double,char,long,short)
String类型
Class类型
enum类型
Annotation类型
以上所有类型的数组
```

如果你使用了其它类型，那编译器就会报错。注意，也不允许使用任何包装类型，但由于自动打包的存在，这算不上什么限制。注解也可以作为元素的类型，比如我们再定义一个注解类：

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
int value() default 0;
String name() default "";
```

```
Constraints constraints() default @Constraints;
//@Constraints后没有括号表明使用默认值，可以加括号，在里面定义初始值，比如@Constraints（unique=true）
}
```

可以看出，所有的注解元素都有一个默认值。编译器对元素的默认值有些过分挑剔，首先，元素必须具有默认值；其次不能以null作为默认值。所以我们只能自己定义一些特殊的值，例如空字符串或负数，来表示某个元素不存在。

注解类的使用

注解类定义好了，怎么使用呢？

为了体现注解的便利和强大，在这里我们写一个demo，使用注解来自动生成一个建数据库表的SQL命令。

另外我们再提供两个注解类：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
public String name() default "";
}
```

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
String name() default "";
Constraints constraints() default @Constraints;
}
```

好，现在我们一共有4个注解类。

@DBTable 代表数据库表，注解元素name表示表名；

@Constraints 代表对数据表每一列的条件补充，有primaryKey是不是主键，默认false，allowNull是否允许为空，默认true，unique数据是否唯一，默认false；

@SQLString 代表表中的String列，注解元素value表示列长度，name表示列名，constraints表示对列的一些约束条件；

@SQLInteger 代表表中的int列，name表示列名，constraints表示对列的一些约束条件；

下面我们定义一个简单的Bean类，在其中应用以上4个注解类：

```
@DBTable(name = "MEMBER")
public class Member {
@SQLString(30) String firstName;
```

```
    @SQLString(50) String lastName;
    @SQLInteger int age;
    @SQLString(value = 30, constraints = @Constraints(primaryKey = true)) String handle;
    public String getFirstName() {
    return firstName;
    }
    public String getLastName() {
    return lastName;
    }
    public int getAge() {
    return age;
    }
    public String getHandle() {
    return handle;
    }
    public String toString() {
    return handle;
    }
    }
```

注解的元素在使用时表现为名-值对的形式，并需要置于 @注解类名 声明之后的括号内。如果没有使用括号，代表全部使用默认值。

1、类的注解@DBTable给定了值MEMBER，它将会用来作为表的名字；

2、Bean的属性firstName和lastName，都被注解为@SQLString类型，这些注解有两个有趣的地方：第一，他们都使用了嵌入的@Constraints注解的默认值；第二，它们都使用了快捷方式。何谓快捷方式？如果程序员的注解中定义了名为value的元素，并且在应用该注解的时候，如果该元素是唯一需要赋值的一个元素，那么此时无需使用名-值对的这种语法，而只需在括号内给出value元素所需的值即可。这可以应用于任何合法类型的元素。当然了，这也限制了程序员必须将此元素命名为value。

Bean属性age全部使用默认值，handle为主键。

3、Bean属性age全部使用默认值，handle为主键。

实现注解处理器

注解类使用上了，我们还需要一个注解处理器来解析我们定义的Bean，这样才能将注解转换成我们需要的操作。

以下定义解析Bean的注解处理器，我们的目的是要输出一条SQL语句。主要用到了反射技术。

```
    public class TableCreator {
    public static void main(String[] args) throws Exception{
    //传入我们定义好的Bean类名，带上包名
```

```
Class<?> cl = Class.forName("annotation.Member");
//检查我们传入的类是否带有@DBTable注解
DBTable dbTable = cl.getAnnotation(DBTable.class);
List<String> columnDefs = new ArrayList<String>();
//得到类中的所有定义的属性
for(Field filed : cl.getDeclaredFields()){
String columnName = null;
//得到属性的注解，对一个目标可以使用多个注解
Annotation[] anns = filed.getAnnotations();
if(anns.length < 1){
continue;
}
//SQLString注解走着
if(anns[0] instanceof SQLString){
SQLString sString = (SQLString)anns[0];
//name()使用的是默认值，所以这里取属性名
if(sString.name().length() < 1){
columnName = filed.getName().toUpperCase();
}else{
columnName = sString.name();
}
//构建SQL语句
columnDefs.add(columnName + " VARCHAR(" + sString.value() + ")" + getConstraints(sString.constr
aints()));
}
//SQLInteger注解走着
if(anns[0] instanceof SQLInteger){
SQLInteger sInt = (SQLInteger)anns[0];
if(sInt.name().length() < 1){
columnName = filed.getName().toUpperCase();
}else{
columnName = sInt.name();
}
columnDefs.add(columnName + " INT" + getConstraints(sInt.constraints()));
}
}
StringBuilder creator = new StringBuilder("CREATE TABLE " + dbTable.name() + "( ");
for(String c : columnDefs){
creator.append("\n" + c + ",");
}
creator.deleteCharAt(creator.length()-1);
creator.append(")");
System.out.println(creator.toString());
}
```

```
private static String getConstraints(Constraints con) {
String constraints = "";
if(!con.allowNull()){
constraints += " NOT NULL";
}
if(con.primaryKey()){
constraints += " PRIMARY KEY";
}
if(con.unique()){
constraints += " UNIQUE";
}
return constraints;
}
}
```

最后的输出：

CREATE TABLE MEMBER(

FIRSTNAME VARCHAR(30),

LASTNAME VARCHAR(50),

AGE INT,

HANDLE VARCHAR(30) PRIMARY KEY)

————————————————————————————————————--end————————————————————————————————————