**注解的好处：**

1.能够读懂别人写的代码，特别是框架相关的代码。

2.本来可能需要很多配置文件，需要很多逻辑才能实现的内容，就可以使用一个或者多个注解来替代，这样就使得编程更加简洁，代码更加清晰。

3.（重点）刮目相看。  
（但是怎么样才能让别人刮目相看呢？会用注解不是目的，最重要的是要使用自定义注解来解决问题。）

**一、Java中的常见注解**

**1）JDK注解**  
JDK注解一共分为三类：

https://upload-images.jianshu.io/upload_images/4259662-a2124c2f3c861a5d.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/684

案例：  
我们先新建一个接口people，如下：

public interface people {

public String name();

public int age();

public void work();

}

然后再建一个类Child实现类people这个接口，并实现该类的方法：

public class Child implements people {

@Override

public String name() {

return null;

}

@Override

public int age() {

return 0;

}

@Override

public void work() {

}

看到这里，我们发现这里的所有方法都会加上一个@Override标记，它告诉我们，同时也告诉编译器我们的这些方法肯定覆盖了类people里面的方法的。假如说，我现在把类people里面的某一个方法注释掉：

//public String name();

再看类Child里面的name方法就会报错。这样，以后大家看到@Override的时候就能想到这个方法是覆盖了某个接口的方法的。

然后，我们回过头来看类people里面有一个work的方法。这里我们可以理解为人是要工作的，但是并不是所有的人都在工作，那么怎么办呢？如果说这个接口正在用，我们不能删除这个方法，这个时候我们就可以这样：

@Deprecated

public void work();

@Deprecated标记就表明这个方法已经过时了，在实际中，它又有什么样的应用场景呢？我们在建一个测试类：

public class Test {

public void work() {

people people=new Child();

！ people.work();

}

}

这个时候我们会发现myeclipse会给一个警告，并且在work中间出现一个破折号，意思就是这个方法已经过时了。那么问题来了，虽然这个方法过时了，但是我们就是那么傲娇，一定要用它，怎么办呢？只需要这样：

public class Test {

@SuppressWarnings("deprecation")

public void work() {

people people=new Child();

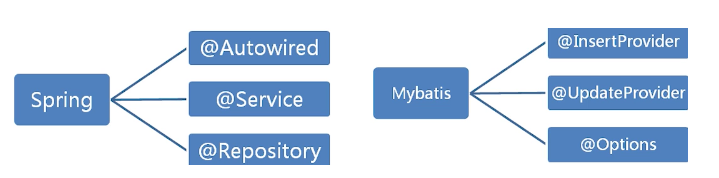
people.work();

}

}

这样我们就忽略了这个警告。@SuppressWarnings("deprecation")就表示我们忽略了deprecation这样的一个警告。

**2）Java第三方注解**



**二、注解的分类**

**1）按照运行机制划分：**  
【源码注解→编译时注解→运行时注解】

源码注解：只在源码中存在，编译成.class文件就不存在了。

编译时注解：在源码和.class文件中都存在。像前面的@Override、@Deprecated、@SuppressWarnings，他们都属于编译时注解。

运行时注解：在运行阶段还起作用，甚至会影响运行逻辑的注解。像@Autowired自动注入的这样一种注解就属于运行时注解，它会在程序运行的时候把你的成员变量自动的注入进来。

**2）按照来源划分：**  
【来自JDK的注解——来自第三方的注解——自定义注解】

**3）元注解：**  
元注解是给注解进行注解，可以理解为注解的注解就是元注解。

**三、自定义注解**

**我们分四步来解析自定义注解：**  
***自定义注解的语法要求：***

@Target({ElementType.METHOD,ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Inherited

@Documented

public @interface Description {

String desc();

String author();

int age() default 18;

}

首先我们要明确这不是一个接口，它是使用@interface关键字定义的一个注解。  
然后我们看下面的几个方法，String desc();虽然它很类似于接口里面的方法，其实它在注解里面只是一个成员变量（成员以无参无异常的方式声明），int age() default 18;（成员变量可以用default指定一个默认值的）。  
最后我们要知道：  
①.成员类型是受限制的，合法的类型包括基本的数据类型以及String，Class，Annotation,Enumeration等。  
②.如果注解只有一个成员，则成员名必须取名为value()，在使用时可以忽略成员名和赋值号（=）。  
③.注解类可以没有成员，没有成员的注解称为标识注解。

***元注解：***

有没有发现上面那段代码有一个没有说呢？没错，它们就是我们所说的元注解：

@Target({ElementType.METHOD,ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Inherited

@Documented

我们先看第一行：@Target是这个注解的作用域，ElementType.METHOD是这个注解的作用域的列表，METHOD是方法声明，除此之外，还有：  
CONSTRUCTOR（构造方法声明）,FIELD（字段声明）,LOCAL VARIABLE（局部变量声明）,METHOD（方法声明）,PACKAGE（包声明）,PARAMETER（参数声明）,TYPE（类接口）

第二行：@Retention是它的生命周期，前面不是说注解按照运行机制有一个分类嘛，RUNTIME就是在运行时存在，可以通过反射读取。除此之外，还有:  
SOURCE（只在源码显示，编译时丢弃）,CLASS（编译时记录到class中，运行时忽略）,RUNTIME（运行时存在，可以通过反射读取）

第三行：@Inherited是一个标识性的元注解，它允许子注解继承它。

第四行：@Documented，生成javadoc时会包含注解。

***使用自定义注解：***  
使用注解的语法：  
@<注解名>(<成员名1>=<成员值1>,<成员名1>=<成员值1>,...)  
案例：

@Description(desc="i am Color",author="boy",age=18)

public String Color() {

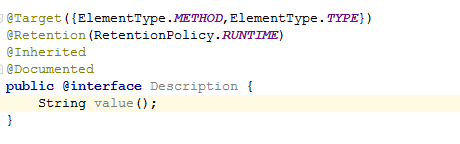
return "red";

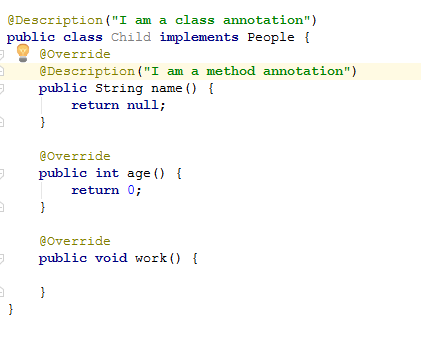
}

这里的Description是我们刚才在***自定义注解语法要求***里面定义的注解噢，然后我们可以给它的每一个成员变量赋值，注意数据类型。值得注意的是，因为我们前面定义的作用域是在方法和类接口上，所以这个注解在Color()方法上使用是没问题的。

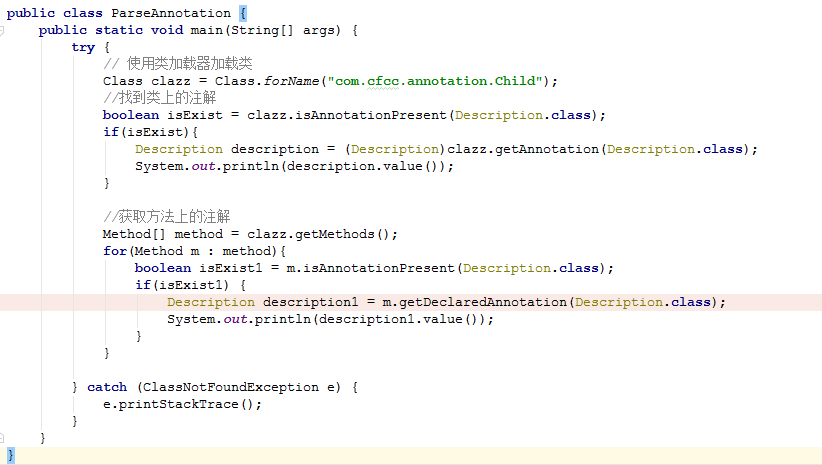
***解析注解***  
**概念：**  
通过反射获取类 、函数或成员上的运行时注解信息，从而实现动态控制程序运行的逻辑。

**准备工作：**





接下来，我们就开始测试了：

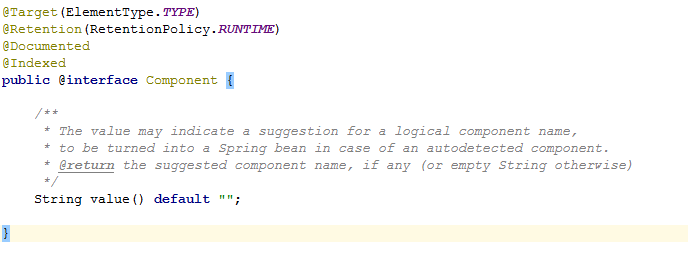


输出的结果：  
i am class annotation  
可以看到，我们成功的解析了Child类上面的注解。

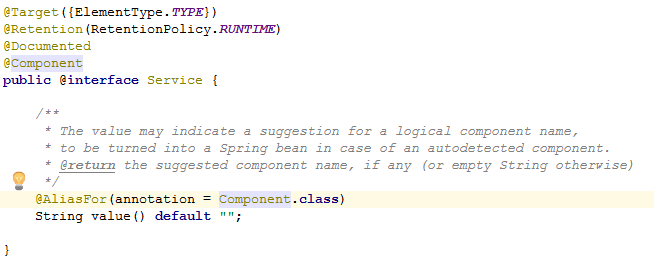
输出的结果：  
i am method annotation  
可以看到，我们成功的解析了方法上面的注解。

此时，如果把Description类里面的元注解改一下，比如:  
@Retention(RetentionPolicy.RUNTIME)→@Retention(RetentionPolicy.SOURCE)，再运行程序，结果会成怎样呢？如果改成CLASS呢？读者们要不要试一试？

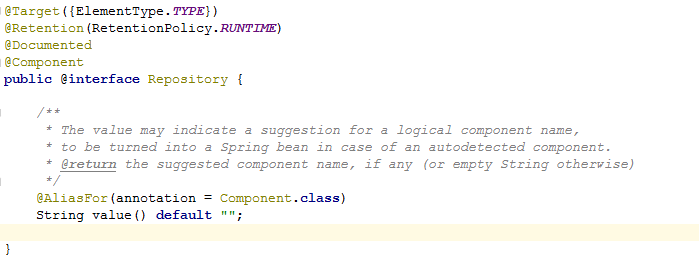
@Component



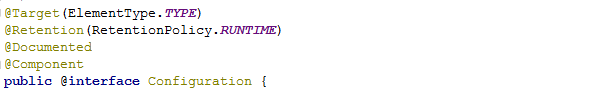
@Service



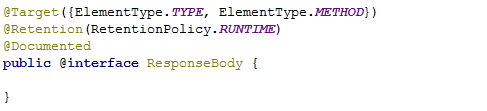
@Repository



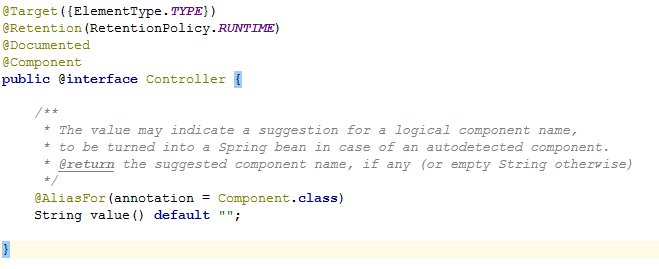
@Configuration



@ResponseBody



@Controller



@RestController

