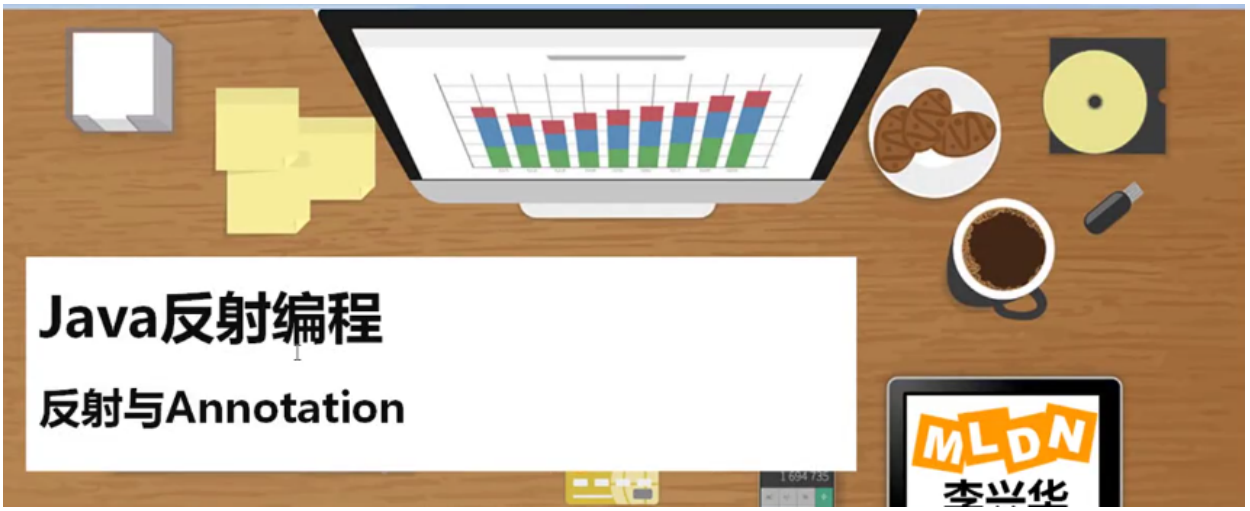


博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

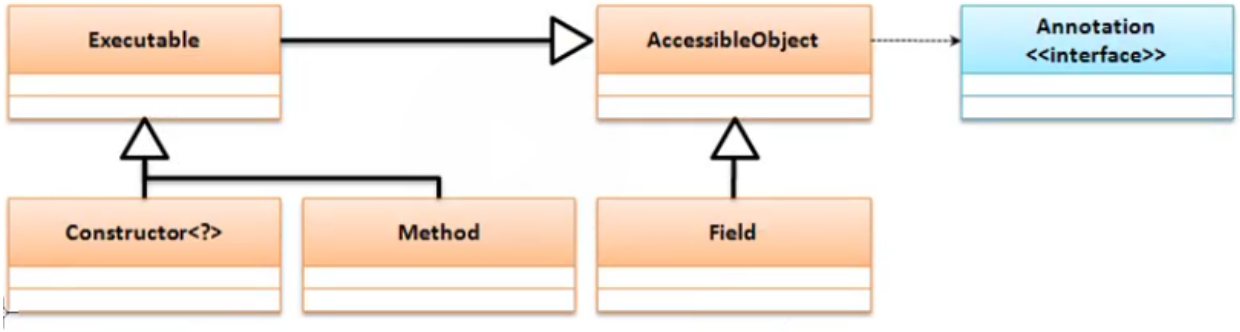
从JDK1.5之后Java开发提供了Annotation技术支持，这种技术为项目的编写带来新的模型，而后经过了十多年的发展，Annotation技术得到了非常广泛的应用，并且已经在所有的项目开发之中都会存在。

■获取Annotation信息

在进行类或方法定义的时候都可以使用一系列的Annotation进行声明，于是如果要想获得这些Annotation的信息，那么就可以直接通过反射来完成。在java.lang.reflect里面有一个AccessibleObje
ct类，在本类中提供有获取Annotation类的方法：

·获取全部Annotation：public Annotation[] getAnnotations();

·获取指定Annotation：public <T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass);



范例：定义一个接口，并且在接口上使用Annotation

```
package cn.mldn.demo;
import java.io.Serializable;
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
public class JavaAPIDemo {
```

```
	public static void main(String[] args) throws Exception {
		{	// 获取接口上的Annotation信息
			Annotation annotations [] = IMessage.class.getAnnotations() ; // 获取接口上的全部Annotation
			for (Annotation temp : annotations) {
				System.out.println(temp);
			}
		}
		System.out.println("----------------------------------");
		{	// 获取MessageImpl子类上的Annotation
			Annotation annotations [] = MessageImpl.class.getAnnotations() ; // 获取接口上的全部Annotation
			for (Annotation temp : annotations) {
				System.out.println(temp);
			}
		}
		System.out.println("----------------------------------");
		{	// 获取MessageImpl.toString()方法上的Annotation
			Method method = MessageImpl.class.getDeclaredMethod("send", String.class) ;
			Annotation annotations [] = method.getAnnotations() ; // 获取接口上的全部Annotation
			for (Annotation temp : annotations) {
				System.out.println(temp);
			}
		}
	}
}
@FunctionalInterface
@Deprecated(since="1.0")
interface IMessage {	// 有两个Annotation
	public void send(String msg) ;
}
@SuppressWarnings("serial") // 无法在程序执行的时候获取
class MessageImpl implements IMessage, Serializable {
	@Override		// 无法在程序执行的时候获取
	public void send(String msg) {
		System.out.println("【消息发送】" + msg);
	}
}
```

不同的Annotation有它的存在范围，下面对比两个Annotation：

@FunctionalInterface(运行时)：	@SuppressWarnings(源代码)：
@Documented @Retention(**RetentionPolicy.RUNTIME**) @Target(ElementType.TYPE) public @interface FunctionalInterface {}	@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE}) @Retention(**RetentionPolicy.SOURCE**) public @interface SuppressWarnings {}

现在发现“@FunctionalInterface"是在运行时生效的Annotation，所以当程序执行的时候可以获得此Annotation，而”@SuppressWarnings“是在源代码编写的时候有效。而在RetentionPolicy枚举类中还有一个class的定义，指的是定义在类定义的时候生效。

■自定义Annotation

现在已经清楚了Annotation的获取，已经Annotation的运行策略，但是最为关键性的因素是如何可以实现自定义的Annotation呢？为此在Java里面提供有新的语法，使用“@interface"来定义Annotation。

范例：自定义Annotation

```
package cn.mldn.demo;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Method method = Message.class.getMethod("send", String.class) ; // 获取指定方法
		DefaultAnnotation anno = method.getAnnotation(DefaultAnnotation.class) ; // 获取指定的Annotation
		String msg = anno.title() + "（" + anno.url() + "）";	// 消息内容
		method.invoke(Message.class.getDeclaredConstructor().newInstance(), msg) ;
	}
}
@Retention(RetentionPolicy.RUNTIME) // 定义Annotation的运行策略
@interface DefaultAnnotation {	// 自定义的Annotation
	public String title() ; // 获取数据
	public String url() default "www.mldn.cn" ;	// 获取数据，默认值
}
class Message {
	@DefaultAnnotation(title="MLDN")
	public void send(String msg) {
		System.out.println("【消息发送】" + msg);
	}
}
```

使用Annotation之后的最大特点是可以结合反射机制实现程序的处理。

■工厂设计模式与Annotation整合

现在已经清楚了Annotation的整体作用，但是Annotation到底在开发之中能做那些事情呢？为了帮助大家进一步理解Annotation的处理的下面将结合工厂模式来应用Annotation操作。

```
package cn.mldn.demo;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
```

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		MessageService messageService = new MessageService() ;
		messageService.send("www.mldn.cn");
	}
}
@Retention(RetentionPolicy.RUNTIME)
@interface UseMessage {
	public Class<?> clazz() ;
}
@UseMessage(clazz=NetMessageImpl.class)	// 利用Annotation实现了类的使用
class MessageService {
	private IMessage message ;
	public MessageService() {
		UseMessage use = MessageService.class.getAnnotation(UseMessage.class) ;
		this.message = (IMessage) Factory.getInstance(use.clazz()) ; // 直接通过Annotation
获取
	}
	public void send(String msg) {
		this.message.send(msg);
	}
}
class Factory {
	private Factory() {}
	public static <T> T getInstance(Class<T> clazz) { // 直接返回一个实例化对象
		try {
			return (T) new
MessageProxy().bind(clazz.getDeclaredConstructor().newInstance()) ;
		} catch (Exception e) {
			e.printStackTrace();
			return null ;
		}
	}
}
class MessageProxy implements InvocationHandler {
	private Object target ;
	public Object bind(Object target) {
		this.target = target ;
		return Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(), this) ;
	}
	public boolean connect() {
		System.out.println("〖代理操作〗进行消息发送通道的连接。");
		return true ;
	}
	public void close() {
		System.out.println("〖代理操作〗关闭连接通道。");
	}
```

```
	@Override
	public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
		try {
			if (this.connect()) {
				return method.invoke(this.target, args);
			} else {
				throw new Exception("【ERROR】消息无法进行发送！");
			}
		} finally {
			this.close();
		}
	}
}
interface IMessage {
	public void send(String msg) ;
}
class MessageImpl implements IMessage {
	@Override
	public void send(String msg) {
		System.out.println("【消息发送】" + msg);
	}
}
class NetMessageImpl implements IMessage {
	@Override
	public void send(String msg) {
		System.out.println("【网络消息发送】" + msg);
	}
}
```

由于Annotation的存在，所以对于面向接口的编程的配置处理将可以直接利用Annotation的属性完成控制，从而使得整体代码变得简洁。