元注解：声明一个注解是在什么情况下有作用并对类，或方法，或变量产生作用

自定义的注解的存活范围（生命周期）：默认是CLASS。

只能用在注解上的注解叫做元注解。（即：用于修饰注解的注解）

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

●@Retention：作用。改变自定义的注解的存活范围。

RetentionPolicy:

SOURCE

CLASS　　默认

RUNTIME

●@Target:作用，指定该注解能用在什么地方。

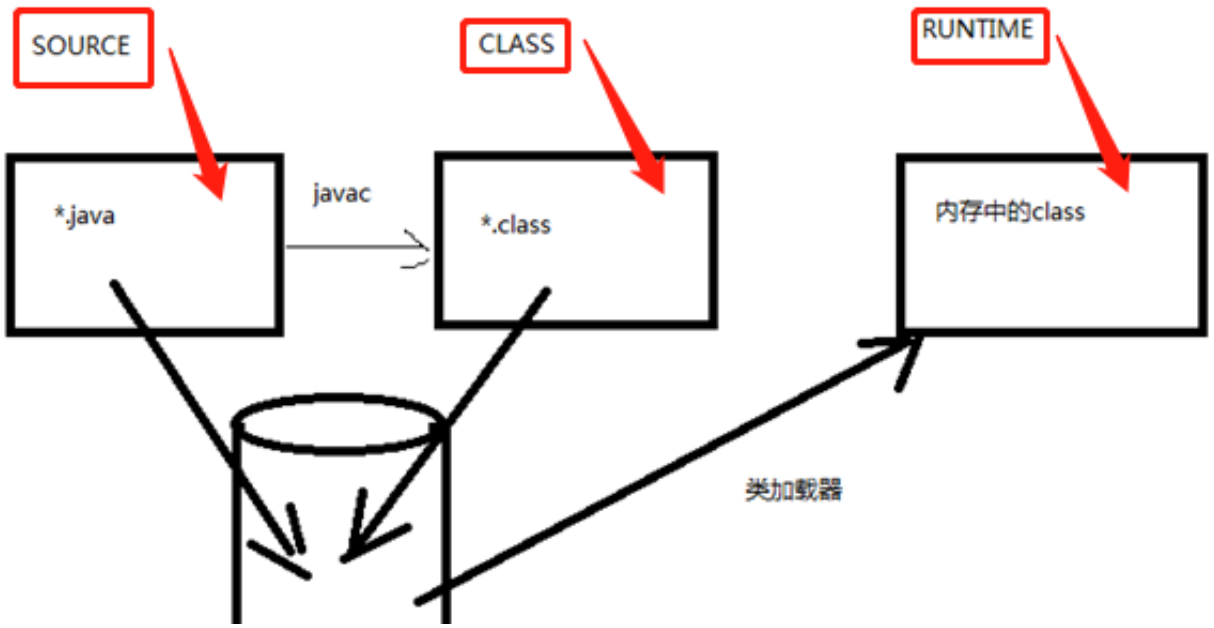
ElementType：

TYPE：类、接口、枚举

METHOD：方法

FIELD：字段

ANNOTATION_TYPE：注解



JDK提供的三个基本的注解

@Override　　描述方法的重写.

@SuppressWarnings　　压制警告.

@Deprecated　　标记过时

自定义注解 Annotation类型　　就是在你的程序代码中的某个位置加了一个标记而已

1、自定义一个类

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyTest {

}
```

元注解

对自定义注解使用元注解

2、申明注解中的成员

```
public @interface MyAnno1 {
    String name();
    int num();
    Color c();
    MyAnno2 anno();
    Class clazz();
    String[] arr();
}
```

自定义注解

注解成员声明

2.1、使用声明：

```
public class TestAnno {

    @MyAnno1(name="tom", num=18, c=Color.blue, anno=@MyAnno2, clazz=Student.class, arr="aaa")
    public void test1(){
        Color c = Color.red;
    }
```

使用自定义注解，后面的值是自定义注解中的成员

2.2、关于注解的属性类型

1. 基本类型
2. String
3. 枚举类型
4. 注解类型
5. Class类型
6. 以上类型的一维数组类型

3、定义核心运行类：

1、在核心运行类中有一个主函数：

2、获得测试类中的所有的方法.通过反射获取

3、获得每个方法,查看方法上是否有@MyTest注解.

4、如果有这个注解,让这个方法执行

```
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
/**
 * 注解功能实现类
 */
public class Anoos {

    public static void main(String[] args) throws IllegalAccessException, IllegalArgumentException, Invocat
        // 得到TestCRUD类的字节码对象
        Class clazz = Tes.class;
        // 得到类中的所有方法
        Method[] methods = clazz.getDeclaredMethods();
        //对获取的所有方法进行遍历
        for (Method method : methods) {
            //判断哪个方法上中@Annotations的注解
            boolean annotationPresent = method.isAnnotationPresent(Annotations.class);
            if (annotationPresent) {
                // 执行当前方法 如果被注解的方法没有参数那么下面第二参数 为 null  有则按数据类型写入
                method.invoke(clazz.newInstance(), "我是一个自定义的注解，想看看好不好用！");
            }
        }
```

@Test源码分析

反射注解类

java.lang.reflect.AnnotatedElement:

●· <T extends Annotation> T getAnnotation(Class<T> annotationType):得到指定类型的注解引用。没有返回null。

●· Annotation[] getAnnotations()：得到所有的注解，包含从父类继承下来的。

●· Annotation[] getDeclaredAnnotations()：得到自己身上的注解。

●· boolean isAnnotationPresent(Class<? extends Annotation> annotationType)：判断指定的注解有没有。

Class、Method、Field、Constructor等实现了AnnotatedElement接口.

如果：Class.isAnnotationPresent(MyTest.class):判断类上面有没有@MyTest注解；

Method.isAnnotationPresent(MyTest.class):判断方法上面有没有@MyTest注解。

类加载器 **ClassLoader**

1、作用：类加载器就是将.class文件加载到内存生成Class对象.

2、JVM中的类加载器：

BootStrap： 引导类加载器　老大。类加载器的祖先。

负责加载JRE/lib/rt.jar （加载JDK中绝大部分的类) runtime

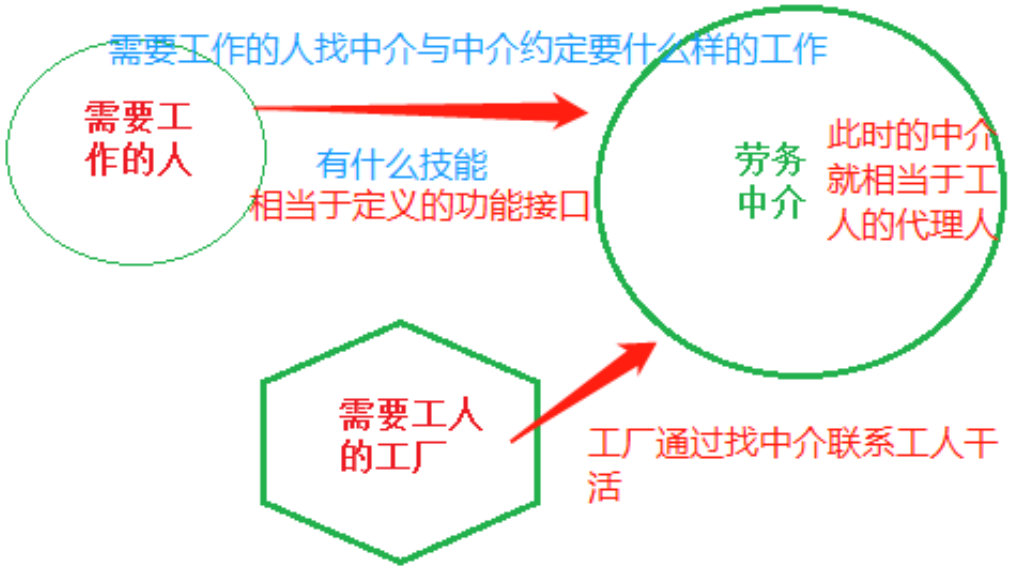
ExtClassLoader：扩展类加载器

负责加载JRE/lib/ext/*.jar

AppClassLoader：应用类加载器

负责加载在classpath环境变量中的所有类。类加载器

静态代理　　原理及实现：



1、先定义一个接口定义好功能

```
package com.rong.AnnoProxy;
**
* 代理对象与被代理对象约定的功能
*
*/
public interface IAppoint {
    public void getName();
    public void getAge(String s);
}
```

2、定义一个被代理对象类实现接口并实现接口的方法

```java
package com.rong.AnnoProxy;
/**
 * 被代理对象
 * @author LiuJinRong
 *
 */
public class PiApp implements IAppoint {

    @Override
    public void getName() {
        System.out.println("我的名字是xxxx");
    }

    @Override
    public void getAge(String s) {
        System.out.println(s+"至于年纪你自己猜猜吧！");
    }
```

3、定义一个代理类，并实现接口方法，同时代理构造方法来确定代理的是谁

```java
package com.rong.AnnoProxy;
/**
 * 代理对象
 * @author LiuJinRong
 *
 */
public class DiApp implements IAppoint {
    private IAppoint ia;
    public DiApp(IAppoint ia) {
        this.ia = ia;
    }
    @Override
    public void getName() {
        ia.getName();
    }

    @Override
    public void getAge(String s) {
        ia.getAge("去你大页的"+s);
    }
```

4、在需要调用被代理类的功能时通过 new代理类去调被代理的功能方法获取

```java
package com.rong.AnnoProxy;
/**
 * 访问代理类
 * @author LiuJinRong
 *
 */
public class XQ {

    public static void main(String[] args) {
        IAppoint appoint = new PiApp();

        DiApp diApp = new DiApp(appoint);

        diApp.getAge(diApp.getString());
    }
```

动态代理 增强一个类中的某个方法.对程序进行扩展.Spring框架中AOP.

Proxy Pattern（即：代理模式），23种常用的面向对象软件的设计模式之一

代理模式的定义：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

动态代理：

动态代理它可以直接给某一个目标对象生成一个代理对象，而不需要代理类存在。

动态代理与代理模式原理是一样的，只是它没有具体的代理类，直接通过反射生成了一个代理对象。

动态代理生成技术：

1. jdk提供一个Proxy类可以直接给实现接口类的对象直接生成代理对象。

2. cglib

Java.lang.reflect.Proxy类可以直接生成一个代理对象

Proxy.newProxyInstance():产生代理类的实例。仅能代理实现至少一个接口的类

ClassLoader：类加载器。固定写法，和被代理类使用相同的类加载器即可。

Class[] interface：代理类要实现的接口。固定写法，和被代理类使用相同的接口即可。

InvocationHandler：策略（方案）设计模式的应用。代理对象如何调用真实对象的方

法。

InvocationHandler中的**invoke**方法：调用代理类的任何方法，此方法都会执行

Object proxy:代理对象本身的引用。一般用不着。

Method method:当前调用的方法。m.invoke(p,null);

Object[] args:当前方法用到的参数

实现原理：不用去创建代理类但必须要约定接口功能方法，通过Proxy类**Proxy.newProxyInstance():**产生代理类的实例

实现步骤：

1、约定接口并定义好接口功能

```
/**
 * 代理对象与被代理对象约定的功能
 */
   public interface IAppoint {
      public void getName();
      public void getAge(String s);
   }
```

2、被代理类实现接口并实现功能方法

```
package com.rong.AnnoProxy;
/**
 * 被代理对象
 * @author LiuJinRong
 *
 */
public class PiApp implements IAppoint {

    @Override
    public void getName() {
        System.out.println("我的名字是xxxx");
    }

    @Override
    public void getAge(String s) {
        System.out.println(s+"至于年纪你自己猜猜吧！");
    }

}
```

被代理类实现接口

3、创建代理Proxy工厂获取一个实现代理的方法

实现代理方法并返回对象

```
//代理方法
public static Object geto(){
    IAppoint appoint = new PiApp();    //被代理对象
    //被代理类加载器，固定写法
    //被代理类要实现的接口。固定写法
    //代理对象如何调用真实对象的方法。
    //返回一个代理类对象appoints
    IAppoint appoints =(IAppoint) Proxy.newProxyInstance(appoint.getClass().getClassLoader(),
            appoint.getClass().getInterfaces(), new InvocationHandler() {

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            //在之前可以做一些事
            //参数二 如果被代理类方法有参数则不管
            Object invoke = method.invoke(appoint, args);
            //在之后可以做一些事
            return null;
        }
    });
    return appoints;
}
```

有返回代理对象的方法

代理对象

返回对象

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

    IAppoint appoints = (IAppoint) geto();
    appoints.getAge("来吧 我亲爱的宝贝！！！");
    appoints.getName();
}
```

获取代理对象并执行代理方法

或者可以不用返回代理对象直接在代理方法中执行代理方法

```
public static void main(String[] args) {
    geto();//执行代理方法
}
//代理方法 无返回 在方法里直接执行你理方法
public static void geto(){
    IAppoint appoint = new PiApp();    //被代理对象
    //被代理类加载器，固定写法
    //被代理类要实现的接口。固定写法
    //代理对象如何调用真实对象的方法。
    //返回一个代理类对象appoints
    IAppoint appoints =(IAppoint) Proxy.newProxyInstance(appoint.getClass().getClassLoader(),
            appoint.getClass().getInterfaces(), new InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            //在之前可以做一些事
            //参数二 如果被代理类方法有参数则不管
            Object invoke = method.invoke(appoint, args);
            //在之后可以做一些事
            return null;
        }
    });
    //无返回 直接执行代理方法
    appoints.getAge("我来了 我是你大爷 快叫爷");
}
```

在或者通过工厂在工厂中定义代理方法

```
public class ProxyFactory {
    public static Object getProxy(final Class clazz){
        Object o = Proxy.newProxyInstance(clazz.getClassLoader(),
            clazz.getInterfaces(), new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                Object b = null;
                try {
                    //开启事务
                    ConnectionManager.begin();
                    //执行真实的被代理类方法
                    b = method.invoke(clazz.newInstance(), args);
                    //提交事务
                    ConnectionManager.commit();
                } catch (Exception e) {
                    ConnectionManager.rollback();    //回滚事务
                    throw new Exception(e);
                }finally{
                    ConnectionManager.close();     //释放资源
                }
                return b;
            }
        });
        return o;
    }
```

创建代理工厂 在工厂里进行代理方法的实现

并返回 代理对象

```
IAccountService is = (IAccountService) ProxyFactory.getProxy(AccountService.class);
//分发转向
try {
    //执行代理类方法
    is.transfer(fromName, toName, money);
    request.getRequestDispatcher("/success.jsp").forward(request, response);
```

获取代理方法 返回代理对象 并执行代理方法

4、返回代理对象或执行代理方法

通过获取代理对象执行代理方法