Java的动态代理(dynamic proxy)

什么是动态代理(dynamic proxy)

动态代理（以下称代理），利用Java的反射技术(Java Reflection)，在运行时创建一个实现某些给定接口的新类（也称“动态代理类”）及其实例（对象）

(Using Java Reflection to create dynamic implementations of interfaces at runtime)。

代理的是接口(Interfaces)，不是类(Class)，更不是抽象类。

动态代理有什么用

解决特定问题：一个接口的实现在编译时无法知道，需要在运行时才能实现

实现某些设计模式：适配器(Adapter)或修饰器(Decorator)

面向切面编程：如AOP in Spring

创建动态代理

利用Java的Proxy类，调用Proxy.newProxyInstance()，创建动态对象十分简单。

```
InvocationHandler handler = new MyInvocationHandler(...);
Class proxyClass = Proxy.getProxyClass(Foo.class.getClassLoader(), new Class[] {
Foo.class });

Foo f = (Foo) proxyClass.
         getConstructor(new Class[] { InvocationHandler.class }).
         newInstance(new Object[] { handler });

//或更简单
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
                                     new Class[] { Foo.class },
                                     handler);
```

Proxy.newProxyInstance()方法有三个参数：

1. 类加载器(Class Loader)

2. 需要实现的接口数组

3. InvocationHandler接口。所有动态代理类的方法调用，都会交由InvocationHandler接口实现类里的invoke()方法去处理。这是动态代理的关键所在。

InvocationHandler接口

接口里有一个invoke()方法。基本的做法是，创建一个类，实现这个方法，利用反射在invoke()方法里实现需求：

```
public class MyInvocationHandler implements InvocationHandler{

  public Object invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    //do something "dynamic"
```

```
  }
}
```

invoke()方法同样有三个参数：

1. 动态代理类的引用，通常情况下不需要它。但可以使用getClass()方法，得到proxy的Class类从而取得实例的类信息，如方法列表，annotation等。

2. 方法对象的引用，代表被动态代理类调用的方法。从中可得到方法名，参数类型，返回类型等等

3. args对象数组，代表被调用方法的参数。注意基本类型(int,long)会被装箱成对象类型(Interger, Long)

动态代理例子

1. 模拟AOP

```
public interface IVehical {

    void run();

}

//concrete implementation
public class Car implements IVehical{

    public void run() {
    System.out.println("Car is running");
    }

}

//proxy class
public class VehicalProxy {

    private IVehical vehical;

    public VehicalProxy(IVehical vehical) {
    this.vehical = vehical;
    }

    public IVehical create(){
    final Class<?>[] interfaces = new Class[]{IVehical.class};
    final VehicalInvacationHandler handler = new VehicalInvacationHandler(vehical);

    return (IVehical) Proxy.newProxyInstance(IVehical.class.getClassLoader(),
interfaces, handler);
    }

    public class VehicalInvacationHandler implements InvocationHandler{

    private final IVehical vehical;

    public VehicalInvacationHandler(IVehical vehical) {
        this.vehical = vehical;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

        System.out.println("--before running...");
        Object ret = method.invoke(vehical, args);
        System.out.println("--after running...");

        return ret;
    }

    }
}

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

```
    IVehical car = new Car();
    VehicalProxy proxy = new VehicalProxy(car);

    IVehical proxyObj = proxy.create();
    proxyObj.run();
    }
}
/*
 * output:
 * --before running...
 * Car is running
 * --after running...
 * */
```

可以看出，对IVehical接口的调用，会交由Proxy的invoke方法处理，并在不改变run()的源代码下，新增了动态的逻辑（before running/after running），这正式AOP所做的。

深入讲，Proxy.newInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)做了以下几件事.
(1)根据参数loader和interfaces调用方法 getProxyClass(loader, interfaces)创建代理类$Proxy.
$Proxy0类实现了interfaces的接口,并继承了Proxy类.
(2)实例化$Proxy0并在构造方法中把BusinessHandler传过去,接着$Proxy0调用父类Proxy的构造器,为h赋值,如下:

```
class Proxy{
  InvocationHandler h=null;
  protected Proxy(InvocationHandler h) {
   this.h = h;
  }
  ...
}
```

另外，如果将invoke()方法代码改成如下：

```
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

        System.out.println("--before running...");
//        Object ret = method.invoke(vehical, args);
        ((IVehical)proxy).run();
        System.out.println("--after running...");

        return null;
    }
```

结果会是因为run()方法会引发invoke()，而invoke()又执行run()，如此下去变成死循环，最后栈溢出

所以invoke 接口中的proxy参数不能用于调用所实现接口的某些方法（见参考**4**）。

2. 利用动态代理实现设计模式，修饰器和适配器：

见参考5

3. 在项目中，可以使用动态代理，获取配置文件，非常方便且有优势：

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Value {

    /**
     * The actual value expression: e.g. "#{systemProperties.myProp}".
     */
    String value();
}
```

```
/**
 * config interfaces, map the config properties file:
 * db.url =
 * db.validation = true
 * db.pool.size = 100
 */
public interface IConfig {

    @Value("db.url")
    String dbUrl();

    @Value("db.validation")
    boolean isValidated();

    @Value("db.pool.size")
    int poolSize();

}

//proxy class
public final class ConfigFactory {

    private ConfigFactory() {}

    public static IConfig create(final InputStream is) throws IOException{

    final Properties properties = new Properties();
    properties.load(is);

    return (IConfig) Proxy.newProxyInstance(IConfig.class.getClassLoader(),
        new Class[] { IConfig.class }, new PropertyMapper(properties));

    }

    public static final class PropertyMapper implements InvocationHandler {

    private final Properties properties;

    public PropertyMapper(Properties properties) {
        this.properties = properties;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {

        final Value value = method.getAnnotation(Value.class);

        if (value == null) return null;

          String property = properties.getProperty(value.value());
          if (property == null) return (null);

          final Class<?> returns = method.getReturnType();
          if (returns.isPrimitive())
          {
            if (returns.equals(int.class)) return (Integer.valueOf(property));
            else if (returns.equals(long.class)) return (Long.valueOf(property));
            else if (returns.equals(double.class)) return (Double.valueOf(property));
            else if (returns.equals(float.class)) return (Float.valueOf(property));
            else if (returns.equals(boolean.class)) return (Boolean.valueOf(property));
          }

        return property;
    }

    }
}

public static void main(String[] args) throws FileNotFoundException, IOException {

    IConfig config = ConfigFactory.create(new
FileInputStream("config/config.properties"));
```

```
    String dbUrl = config.dbUrl();
    boolean isLoginValidated = config.isValidated();
    int dbPoolSize = config.poolSize();

    }
```

利用动态代理载入配置文件，并将每一个配置映射成方法，方便我们使用追踪。

最后，有个小挑战就是，将动态代理类$Proxy还原出来，暂时还没做。请看参考3

参考：

http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

http://tutorials.jenkov.com/java-reflection/dynamic-proxies.html

http://hi.baidu.com/malecu/item/9e0edc115cb597a1feded5a0

http://www.cnblogs.com/duanxz/archive/2012/12/03/2799504.html

http://www.ibm.com/developerworks/cn/java/j-jtp08305.html

posted @ 2013-12-03 16:09 macemers 阅读(...) 评论(...) 编辑 收藏

刷新评论 刷新页面 返回顶部