JAVA动态代理

只是肿态度 关注

9 2019.05.12 23:25:39 字数 964 阅读 203,949

代理模式

为其他对象提供一个代理以控制对某个对象的访问。代理类主要负责为委托了（真实对象）预处理消息、过滤消息、传递消息给委托类，代理类不现实具体服务，而是利用委托类来完成服务，并将执行结果封装处理。

其实就是代理类为被代理类预处理消息、过滤消息并在此之后将消息转发给被代理类，之后还能进行消息的后置处理。代理类和被代理类通常会存在关联关系(即上面提到的持有的被带离对象的引用)，代理类本身不实现服务，而是通过调用被代理类中的方法来提供服务。

静态代理

创建一个接口，然后创建被代理的类实现该接口并且实现该接口中的抽象方法。之后再创建一个代理类，同时使其也实现这个接口。在代理类中持有一个被代理对象的引用，而后在代理类方法中调用该对象的方法。

接口：

```
public interface HelloInterface {
    void sayHello();
}

```

被代理类：

```
public class Hello implements HelloInterface{
    @Override
    public void sayHello() {
        System.out.println("Hello zhanghao!");
    }
}
```

代理类：

```
public class HelloProxy implements HelloInterface{
    private HelloInterface helloInterface = new Hello();
    @Override
    public void sayHello() {
        System.out.println("Before invoke sayHello" );
        helloInterface.sayHello();
        System.out.println("After invoke sayHello");
    }
}
```

代理类调用：
被代理类被传递给了代理类HelloProxy，代理类在执行具体方法时通过所持用的被代理类完成调用。

```
    public static void main(String[] args) {
        HelloProxy helloProxy = new HelloProxy();
        helloProxy.sayHello();
    }

输出：
```

```
Before invoke sayHello
Hello zhanghao!
After invoke sayHello
```

使用静态代理很容易就完成了对一个类的代理操作。但是静态代理的缺点也暴露了出来：由于代理只能为一个类服务，如果需要代理的类很多，那么就需要编写大量的代理类，比较繁琐。

动态代理

利用反射机制在运行时创建代理类。
接口、被代理类不变，我们构建一个handler类来实现InvocationHandler接口。

```
public class ProxyHandler implements InvocationHandler{
    private Object object;
    public ProxyHandler(Object object){
        this.object = object;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
        System.out.println("Before invoke "  + method.getName());
        method.invoke(object, args);
        System.out.println("After invoke " + method.getName());
        return null;
    }
}
```

执行动态代理：

```
    public static void main(String[] args) {
        System.getProperties().setProperty("sun.misc.ProxyGenerator.saveGeneratedFiles

        HelloInterface hello = new Hello();

        InvocationHandler handler = new ProxyHandler(hello);

        HelloInterface proxyHello = (HelloInterface) Proxy.newProxyInstance(hello.getC

        proxyHello.sayHello();
    }
    输出：
    Before invoke sayHello
    Hello zhanghao!
    After invoke sayHello
```

通过Proxy类的静态方法newProxyInstance返回一个接口的代理实例。针对不同的代理类，传入相应的代理程序控制器InvocationHandler。
如果新来一个被代理类Bye，如：

```
public interface ByeInterface {
    void sayBye();
}
public class Bye implements ByeInterface {
    @Override
    public void sayBye() {
        System.out.println("Bye zhanghao!");
    }
}
```

那么执行过程：

```
    public static void main(String[] args) {
        System.getProperties().setProperty("sun.misc.ProxyGenerator.saveGeneratedFiles

        HelloInterface hello = new Hello();
        ByeInterface bye = new Bye();

        InvocationHandler handler = new ProxyHandler(hello);
        InvocationHandler handler1 = new ProxyHandler(bye);

```

```
        HelloInterface proxyHello = (HelloInterface) Proxy.newProxyInstance(hello.getC

        ByeInterface proxyBye = (ByeInterface) Proxy.newProxyInstance(bye.getClass().g
        proxyHello.sayHello();
        proxyBye.sayBye();
    }
    输出:
    Before invoke sayHello
    Hello zhanghao!
    After invoke sayHello
    Before invoke sayBye
    Bye zhanghao!
    After invoke sayBye
```

动态代理底层实现

动态代理具体步骤：

1. 通过实现 InvocationHandler 接口创建自己的调用处理器；
2. 通过为 Proxy 类指定 ClassLoader 对象和一组 interface 来创建动态代理类；
3. 通过反射机制获得动态代理类的构造函数，其唯一参数类型是调用处理器接口类型；
4. 通过构造函数创建动态代理类实例，构造时调用处理器对象作为参数被传入。

既然生成代理对象是用的Proxy类的静态方newProxyInstance，那么我们就去它的源码里看一下它到底都做了些什么？

```
    public static Object newProxyInstance(ClassLoader loader,
                                          Class<?>[] interfaces,
                                          InvocationHandler h)
        throws IllegalArgumentException
    {
        Objects.requireNonNull(h);

        final Class<?>[] intfs = interfaces.clone();
        final SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            checkProxyAccess(Reflection.getCallerClass(), loader, intfs);
        }
         //生成代理类对象
        Class<?> cl = getProxyClass0(loader, intfs);

        //使用指定的调用处理程序获取代理类的构造函数对象
        try {
            if (sm != null) {
                checkNewProxyPermission(Reflection.getCallerClass(), cl);
            }

            final Constructor<?> cons = cl.getConstructor(constructorParams);
            final InvocationHandler ih = h;
            //如果Class作用域为私有，通过 setAccessible 支持访问
            if (!Modifier.isPublic(cl.getModifiers())) {
                AccessController.doPrivileged(new PrivilegedAction<Void>() {
                    public Void run() {
                        cons.setAccessible(true);
                        return null;
                    }
                });
            }
            //获取Proxy Class构造函数，创建Proxy代理实例。
            return cons.newInstance(new Object[]{h});
        } catch (IllegalAccessException|InstantiationException e) {
            throw new InternalError(e.toString(), e);
        } catch (InvocationTargetException e) {
            Throwable t = e.getCause();
            if (t instanceof RuntimeException) {
                throw (RuntimeException) t;
            } else {
                throw new InternalError(t.toString(), t);
            }
        } catch (NoSuchMethodException e) {
            throw new InternalError(e.toString(), e);
        }
    }
```

利用getProxyClass0(loader, intfs)生成代理类Proxy的Class对象。

```
    private static Class<?> getProxyClass0(ClassLoader loader,
                                           Class<?>... interfaces) {
        //如果接口数量大于65535，抛出非法参数错误
        if (interfaces.length > 65535) {
            throw new IllegalArgumentException("interface limit exceeded");
        }


        //如果指定接口的代理类已经存在与缓存中，则不用新创建，直接从缓存中取即可；
        //如果缓存中没有指定代理对象，则通过ProxyClassFactory来创建一个代理对象。
        return proxyClassCache.get(loader, interfaces);
    }
```

ProxyClassFactory内部类创建、定义代理类，返回给定ClassLoader 和interfaces的代理类。

```
        private static final class ProxyClassFactory
        implements BiFunction<ClassLoader, Class<?>[], Class<?>>{
        // 代理类的名字的前缀统一为“$Proxy”
        private static final String proxyClassNamePrefix = "$Proxy";

        // 每个代理类前缀后面都会跟着一个唯一的编号，如$Proxy0、$Proxy1、$Proxy2
        private static final AtomicLong nextUniqueNumber = new AtomicLong();

        @Override
        public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

            Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(interfaces.len
            for (Class<?> intf : interfaces) {
                //验证类加载器加载接口得到对象是否与由apply函数参数传入的对象相同
                Class<?> interfaceClass = null;
                try {
                    interfaceClass = Class.forName(intf.getName(), false, loader);
                } catch (ClassNotFoundException e) {
                }
                if (interfaceClass != intf) {
                    throw new IllegalArgumentException(
                        intf + " is not visible from class loader");
                }
                //验证这个Class对象是不是接口
                if (!interfaceClass.isInterface()) {
                    throw new IllegalArgumentException(
                        interfaceClass.getName() + " is not an interface");
                }
                if (interfaceSet.put(interfaceClass, Boolean.TRUE) != null) {
                    throw new IllegalArgumentException(
                        "repeated interface: " + interfaceClass.getName());
                }
            }

            String proxyPkg = null;     // package to define proxy class in
            int accessFlags = Modifier.PUBLIC | Modifier.FINAL;

            /*
             * Record the package of a non-public proxy interface so that the
             * proxy class will be defined in the same package.  Verify that
             * all non-public proxy interfaces are in the same package.
             */
            for (Class<?> intf : interfaces) {
                int flags = intf.getModifiers();
                if (!Modifier.isPublic(flags)) {
                    accessFlags = Modifier.FINAL;
                    String name = intf.getName();
                    int n = name.lastIndexOf('.');
                    String pkg = ((n == -1) ? "" : name.substring(0, n + 1));
                    if (proxyPkg == null) {
                        proxyPkg = pkg;
                    } else if (!pkg.equals(proxyPkg)) {
                        throw new IllegalArgumentException(
                            "non-public interfaces from different packages");
                    }
                }
            }

            if (proxyPkg == null) {
                // if no non-public proxy interfaces, use com.sun.proxy package
                proxyPkg = ReflectUtil.PROXY_PACKAGE + ".";
            }
```

```
            /*
             * Choose a name for the proxy class to generate.
             */
            long num = nextUniqueNumber.getAndIncrement();
            String proxyName = proxyPkg + proxyClassNamePrefix + num;

            /*
             *
             * 生成指定代理类的字节码文件
             */
            byte[] proxyClassFile = ProxyGenerator.generateProxyClass(
                proxyName, interfaces, accessFlags);
            try {
                return defineClass0(loader, proxyName,
                                    proxyClassFile, 0, proxyClassFile.length);
            } catch (ClassFormatError e) {
                /*
                 * A ClassFormatError here means that (barring bugs in the
                 * proxy class generation code) there was some other
                 * invalid aspect of the arguments supplied to the proxy
                 * class creation (such as virtual machine limitations
                 * exceeded).
                 */
                throw new IllegalArgumentException(e.toString());
            }
        }
    }
```

一系列检查后，调用ProxyGenerator.generateProxyClass来生成字节码文件。

```
public static byte[] generateProxyClass(final String var0, Class<?>[] var1, int var2
      ProxyGenerator var3 = new ProxyGenerator(var0, var1, var2);
      // 真正用来生成代理类字节码文件的方法在这里
      final byte[] var4 = var3.generateClassFile();
      // 保存代理类的字节码文件
      if(saveGeneratedFiles) {
          AccessController.doPrivileged(new PrivilegedAction<Void>() {
              public Void run() {
                  try {
                      int var1 = var0.lastIndexOf(46);
                      Path var2;
                      if(var1 > 0) {
                          Path var3 = Paths.get(var0.substring(0, var1).replace('.',
                          Files.createDirectories(var3, new FileAttribute[0]);
                          var2 = var3.resolve(var0.substring(var1 + 1, var0.length()
                      } else {
                          var2 = Paths.get(var0 + ".class", new String[0]);
                      }

                      Files.write(var2, var4, new OpenOption[0]);
                      return null;
                  } catch (IOException var4x) {
                      throw new InternalError("I/O exception saving generated file:
                  }
              }
          });
      }

      return var4;
  }
```

生成代理类字节码文件的generateClassFile方法:

```
  private byte[] generateClassFile() {
      //下面一系列的addProxyMethod方法是将接口中的方法和Object中的方法添加到代理方法中(proxyMe
      this.addProxyMethod(hashCodeMethod, Object.class);
      this.addProxyMethod(equalsMethod, Object.class);
      this.addProxyMethod(toStringMethod, Object.class);
      Class[] var1 = this.interfaces;
      int var2 = var1.length;

      int var3;
      Class var4;
      //获得接口中所有方法并添加到代理方法中
      for(var3 = 0; var3 < var2; ++var3) {
          var4 = var1[var3];
          Method[] var5 = var4.getMethods();
          int var6 = var5.length;

```

```
            for(int var7 = 0; var7 < var6; ++var7) {
                Method var8 = var5[var7];
                this.addProxyMethod(var8, var4);
            }
        }

        Iterator var11 = this.proxyMethods.values().iterator();

        List var12;
        while(var11.hasNext()) {
            var12 = (List)var11.next();
            checkReturnTypes(var12);
        }

        Iterator var15;
        try {
            //生成代理类的构造函数
            this.methods.add(this.generateConstructor());
            var11 = this.proxyMethods.values().iterator();

            while(var11.hasNext()) {
                var12 = (List)var11.next();
                var15 = var12.iterator();

                while(var15.hasNext()) {
                    ProxyGenerator.ProxyMethod var16 = (ProxyGenerator.ProxyMethod)var
                    this.fields.add(new ProxyGenerator.FieldInfo(var16.methodFieldName
                    this.methods.add(var16.generateMethod());
                }
            }

            this.methods.add(this.generateStaticInitializer());
        } catch (IOException var10) {
            throw new InternalError("unexpected I/O Exception", var10);
        }

        if(this.methods.size() > '\uffff') {
            throw new IllegalArgumentException("method limit exceeded");
        } else if(this.fields.size() > '\uffff') {
            throw new IllegalArgumentException("field limit exceeded");
        } else {
            this.cp.getClass(dotToSlash(this.className));
            this.cp.getClass("java/lang/reflect/Proxy");
            var1 = this.interfaces;
            var2 = var1.length;

            for(var3 = 0; var3 < var2; ++var3) {
                var4 = var1[var3];
                this.cp.getClass(dotToSlash(var4.getName()));
            }

            this.cp.setReadOnly();
            ByteArrayOutputStream var13 = new ByteArrayOutputStream();
            DataOutputStream var14 = new DataOutputStream(var13);

            try {
                var14.writeInt(-889275714);
                var14.writeShort(0);
                var14.writeShort(49);
                this.cp.write(var14);
                var14.writeShort(this.accessFlags);
                var14.writeShort(this.cp.getClass(dotToSlash(this.className)));
                var14.writeShort(this.cp.getClass("java/lang/reflect/Proxy"));
                var14.writeShort(this.interfaces.length);
                Class[] var17 = this.interfaces;
                int var18 = var17.length;

                for(int var19 = 0; var19 < var18; ++var19) {
                    Class var22 = var17[var19];
                    var14.writeShort(this.cp.getClass(dotToSlash(var22.getName())));
                }

                var14.writeShort(this.fields.size());
                var15 = this.fields.iterator();

                while(var15.hasNext()) {
                    ProxyGenerator.FieldInfo var20 = (ProxyGenerator.FieldInfo)var15.n
                    var20.write(var14);
                }

                var14.writeShort(this.methods.size());
                var15 = this.methods.iterator();

```

```
                while(var15.hasNext()) {
                    ProxyGenerator.MethodInfo var21 = (ProxyGenerator.MethodInfo)var15
                    var21.write(var14);
                }

                var14.writeShort(0);
                return var13.toByteArray();
            } catch (IOException var9) {
                throw new InternalError("unexpected I/O Exception", var9);
            }
        }
    }
```

字节码生成后，调用defineClass0来解析字节码，生成了Proxy的Class对象。在了解完代理类动态生成过程后，生产的代理类是怎样的，谁来执行这个代理类。

其中，在ProxyGenerator.generateProxyClass函数中 saveGeneratedFiles定义如下，其指代是否保存生成的代理类class文件，默认false不保存。

在前面的示例中，我们修改了此系统变量：

```
System.getProperties().setProperty("sun.misc.ProxyGenerator.saveGeneratedFiles", "true
```

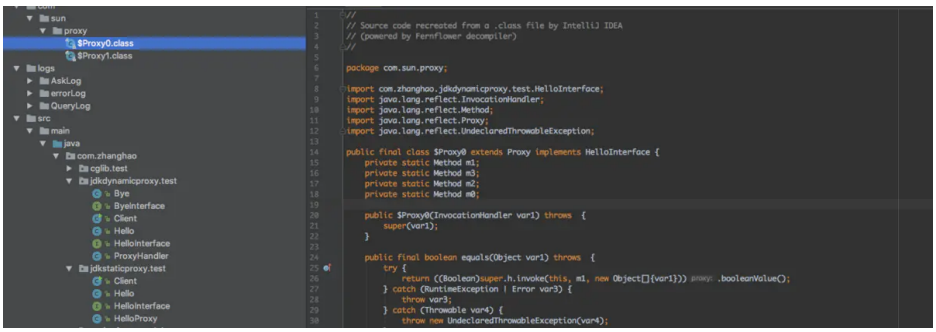

image.png

如图，生成了两个名为 ***Proxy0.class***、 Proxy1.class的class文件。

动态代理流程图：

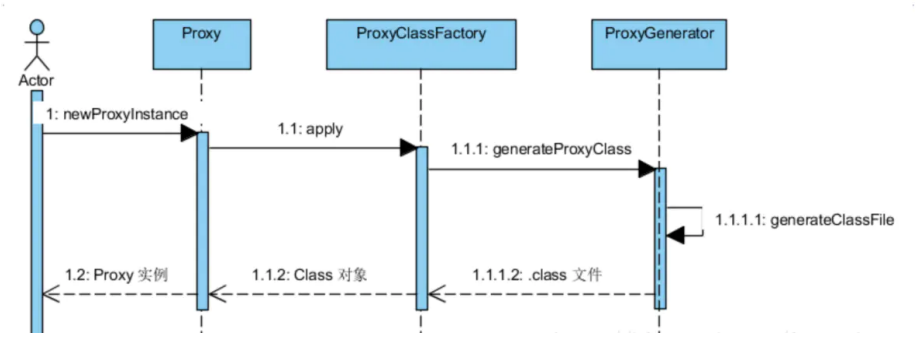

image.png