

这篇文章我们将讲解 ASM 在 cglib 和 fastjson 上的实际使用案例。

0x01 cglib 的简单应用

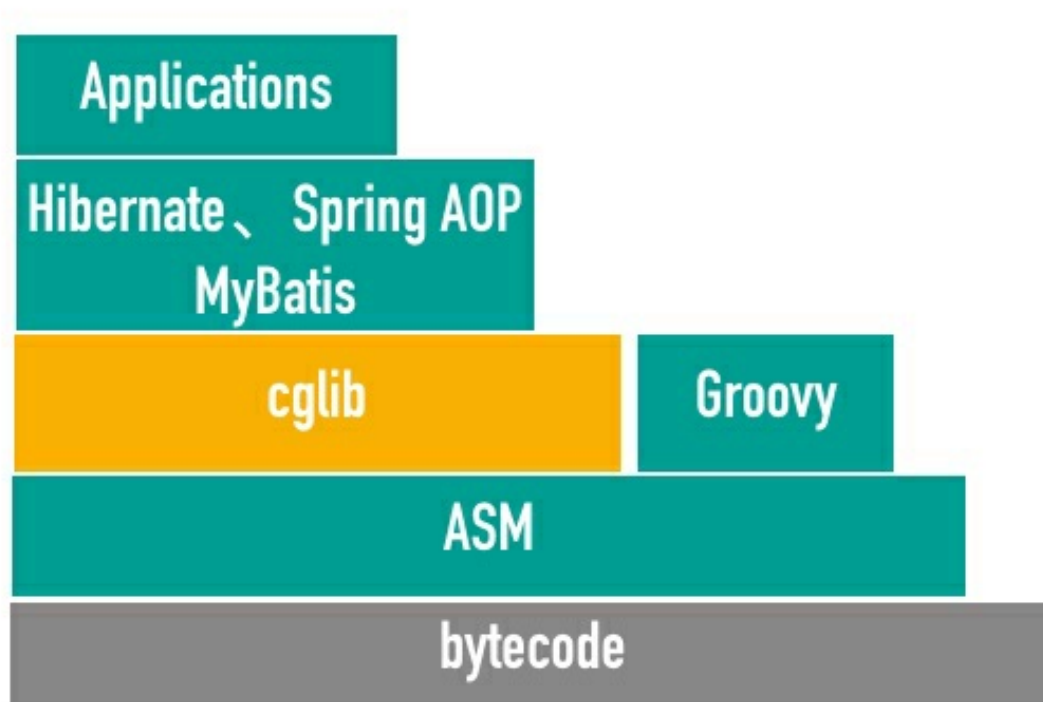


如果说 ASM 是字节码改写事实上的标准，那么可以说 cglib 则是动态代理事实上的标准。

cglib 是一个强大的、高性能的代码生成库，被大量框架使用

- Spring：为基于代理的 AOP 框架提供方法拦截
- MyBatis：用来生成 Mapper 接口的动态代理实现类
- Hibernate：用来生成持久化相关的类
- Guice、EasyMock、jMock 等

在实现内部，cglib 库使用了 ASM 字节码操作框架来转化字节码，产生新类，帮助开发者屏蔽了很多字节码相关的内部细节，不用再去关心类文件格式、指令集等



有这样一个 Person 类，想在 doJob 调用前和调用后分别记录一些日志

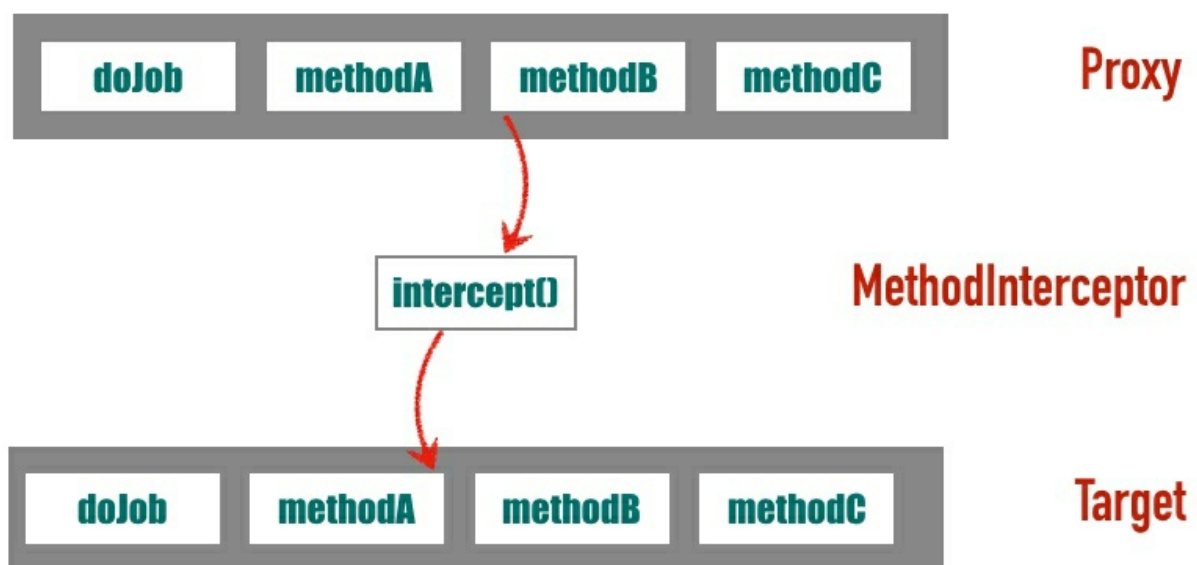
```
public class Person {
    public void doJob(String jobName) {
        System.out.println("who is this class: "
+ getClass());
        System.out.println("doing job: " +
jobName);
    }
}
```

我们可以使用 JDK 动态代理来实现，不过介于 JDK 动态代理有个明显的缺点（需要目标对象实现一个或多个接口），在这里重点介绍 cglib 的实现方案。

一个典型的实现方案是实现一个 net.sf.cglib.proxy.MethodInterceptor 接口，用来拦截方法调用。这个接口只有一个方法：public Object

`intercept(Object obj, java.lang.reflect.Method method, Object[] args, MethodProxy proxy)` throws `Throwable`;

这个方法的第一个参数 `obj` 是代理对象，第二个参数 `method` 是拦截的方法，第三个参数是方法的参数，第四个参数 `proxy` 用来调用父类的方法。`MethodInterceptor` 作为一个桥梁连接了目标对象和代理对象



cglib 代理的核心是 `net.sf.cglib.proxy.Enhancer` 类，它用于创建一个 cglib 代理。这个类有一个静态方法 `public static Object create(Class type, Callback callback)`，该方法的第一个参数 `type` 指明要代理的对象类型，第二个参数 `callback` 是要拦截的具体实现，一般都会传入一个 `MethodInterceptor` 的实现

```

public static void main(String[] _args) {
    MethodInterceptor interceptor = new
MethodInterceptor() {
        @Override
        public Object intercept(Object obj,
Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
            System.out.println(">>>>>before
intercept");
            Object o =
methodProxy.invokeSuper(obj, args);
            System.out.println(">>>>>end
intercept");
            return o;
        }
    };
    Person person = (Person)
Enhancer.create(Person.class, interceptor);
    person.doJob("coding");
}

```

运行上面的代码输出：

```

>>>>>before intercept
who is this class: class
Person$EnhancerByCGLIB$a1da8fe5
doing job: coding
>>>>>end intercept

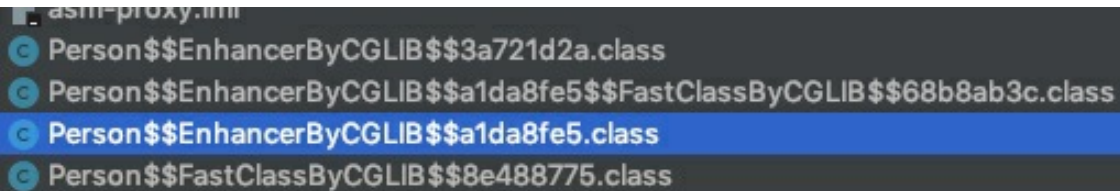
```

可以用设置系统变量让 cglib 输出生成的文件

```

System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,
"/path/to/cglib-debug-location");

```



```
asm-proxy.jar
Person$$EnhancerByCGLIB$$3a721d2a.class
Person$$EnhancerByCGLIB$$a1da8fe5$$FastClassByCGLIB$$68b8ab3c.class
Person$$EnhancerByCGLIB$$a1da8fe5.class
Person$$FastClassByCGLIB$$8e488775.class
```

核心类是 `Person$EnhancerByCGLIB$a1da8fe5.class`，这个类的反编译以后的代码如下

```
public class Person$EnhancerByCGLIB$a1da8fe5
extends Person implements Factory {
    public final void doJob(String jobName) {
        MethodInterceptor methodInterceptor =
this.CGLIB$CALLBACK_0;
        methodInterceptor.intercept(this,
CGLIB$doJob$0$Method, new Object[]{jobName},
CGLIB$doJob$0$Proxy);
    }
}
```

可以看到 cglib 生成了一个 `Person` 的子类，实现了 `doJob` 方法，此方法会调用 `MethodInterceptor` 的 `intercept` 函数，这个函数先输出 "`>>>>>before intercept`" 然后调用父类（也即真正的 `Person` 类）的 `doJob` 的方法，最后输出 "`>>>>>end intercept`"

0x02 fastjson



fastjson 是目前 java 语言中最快的 json 库，比自称最快的 jackson 速度要快。fastjson 库内置 ASM，基于 objectweb asm 3.3 改造，只保留必要的部分不到 2000 行代码，通过 ASM 自动生成序列号、反序列化字节码，减少反射开销，理论上可以提高 20% 的性能。

如果不用反射，一个 json 反序列化要怎么样来做呢？下面写了一个最简单粗暴的例子，来反序列化下面的 json 字符串

```
{  
  "id": "A10001",  
  "name": "Arthur.Zhang",  
  "score": 100  
}
```

对应 Java bean

```
public static class MyBean {  
    public String id;  
    public String name;  
    public Integer score;  
}
```

假定不考虑嵌套，特殊字符的情况，不做语法解析的情况下，可以这么来写

```
public static void main(String[] args) {
    String json = "{ \"id\": \"A10001\",
    \"name\": \"Arthur.Zhang\", \"score\": 100 }";
    // 去掉头尾的 {}
    String str = json.substring(1, json.length()
- 1);
    // 用 "," 分割字符串
    String[] fieldStrArray = str.split(",");
    MyBean bean = new MyBean();
    for (String item : fieldStrArray) {
        // 分隔 key value
        String[] parts = item.split(":");
        String key = parts[0].replaceAll("\\\"",
        "").trim();
        String value = parts[1].replaceAll("\\\"",
        "").trim();
        // 通过反射获取字段对应的 field
        Field field =
        MyBean.class.getDeclaredField(key);
        // 根据字段类型通过反射设置字段的值
        if (field.getType() == String.class) {
            field.set(bean, value);
        } else if (field.getType() ==
        Integer.class) {
            field.set(bean,
        Integer.valueOf(value));
        }
    }
    System.out.println(bean);
}
```


可以看到获取获取字段 field、设置字段值都需要通过反射的方式。那么 fastjson 是怎么解决反射低效的问题的呢？通过调试的方式，把 fastjson 生成的字节码写入到文件中。针对 MyBean，fastjson 使用 ASM 为它生成了一个反序列化的类，里面硬编码了处理序列化需要用到的所有可能场景，不再需要任何反射相关的代码。结合创新的 sort field fast match 算法，速度更上一层楼。下面是通过阅读字节码精简以后的 Java 代码。

```
public class FastjsonASMDeserializer_1_MyBean
extends JavaBeanDeserializer {
    public char[] id_asm_prefix__ =
"\id\":".toCharArray();
    public char[] name_asm_prefix__ =
"\name\":".toCharArray();
    public char[] score_asm_prefix__ =
"\score\":".toCharArray();

    @Override
    public Object deserialize(DefaultJSONParser
parser, Type type, Object fieldName, int
features) {
        JSONLexerBase lexer = (JSONLexerBase)
parser.lexer;
        MyTest.MyBean localMyBean = new
MyTest.MyBean();
        String id =
lexer.scanFieldString(this.id_asm_prefix__);
        if (lexer.matchStat > 0) {
            localMyBean.id = id;
        }
        String name =
lexer.scanFieldString(this.name_asm_prefix__);
        if (lexer.matchStat > 0) {
```

```
        localMyBean.name = name;
    }
    Integer score =
lexer.scanFieldInt(this.score_asm_prefix__);
    if (lexer.matchStat > 0) {
        localMyBean.score = score;
    }
    return localMyBean;
}
}
```

通过上面的两个例子，我们可以看到 ASM 字节码技术在底层库上的强大。可能每天写业务代码不会需要使用这些底层的优化，但是当我们想造一个轮子，想读懂开源代码背后的核心时，都不得不深入的去学习了解这部分知识，很难，但很值得。

0x03 小结

这篇文章我们主要讲解了 ASM 字节码改写技术在 cglib 和 fastjson 上的应用，一起来回顾一下要点：

- 第一，cglib 使用 ASM 生成了目标代理类的一个子类，在子类中扩展父类方法，达到代理的功能，因此要求代理的类不能是 final 的。
- 第二，fastjson 使用 ASM 生成了实例 Bean 反序列化类，彻底去掉了反射的开销，使性能更上一层楼。

0x04 思考

给你留一道作业题：大名鼎鼎的 MyBatis 也用到了 ASM，它用 ASM 实现了什么功能呢？

欢迎你在留言区留言，和我一起讨论。

