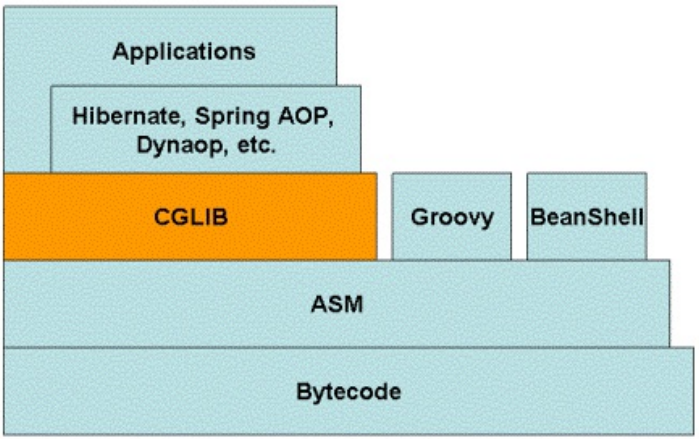
CGLIB

cglib是一个强大的、高性能的代码生成库，它的底层是通过 `ASM` 实现的；JDK动态代理虽然简单易用，但是其有一个致命缺陷是，只能对接口进行代理。

1. cglib的架构



> CGLIB底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。除了CGLIB库外，脚本语言（如Groovy何BeanShell）也使用ASM生成字节码。ASM使用类似SAX的解析器来实现高性能。我们不鼓励直接使用ASM，因为它需要对Java字节码的格式足够的了解

2. 简单动态代理用法

```
public class SampleClass {
    public void test(){
        System.out.println("hello world");
    }

    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(SampleClass.class);
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object obj, Method method, Object[] args,
                                    MethodProxy proxy) throws Throwable {
                System.out.println("before method run...");
                Object result = proxy.invokeSuper(obj, args);
                System.out.println("after method run...");
                return result;
            }
        });
        SampleClass sample = (SampleClass) enhancer.create();
        sample.test();
    }
```

```
    /* 用法二 */
    @Test
    public void testFixedValue(){
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(SampleClass.class);
        enhancer.setCallback(new FixedValue() {
            @Override
            public Object loadObject() throws Exception {
                return "Hello cglib";
            }
        });
        SampleClass proxy = (SampleClass) enhancer.create();
        System.out.println(proxy.test(null)); //拦截test，输出Hello cglib
        System.out.println(proxy.toString());
        System.out.println(proxy.getClass());
        System.out.println(proxy.hashCode());
    }
   /**
     * 输出：
     * Hello cglib
     * Hello cglib
     * class com.example.demo.multi.springBoot
        .SampleClass$$EnhancerByCGLIB$$dc4b1e6c
     *
     * java.lang.ClassCastException: java.lang.String
       cannot be cast to java.lang.Number
     */
}
```

3. 主要类

3.1 Enhancer

Enhancer创建一个被代理对象的子类并且拦截所有的方法调用（包括从Object中继承的toString和hashCode方法）。

Enhancer不能够拦截final方法，例如Object.getClass()方法，这是由于Java final方法语义决定的。基于同样的道理，Enhancer也不能对fianl类进行代理操作。这也是Hibernate为什么不能持久化final class的原因。

3.1.1 限制

- 虽然类的构造放法只是Java字节码层面的函数，但是Enhancer却不能对其进行操作。
- Enhancer同样不能操作static或者final类。

3.1.2 应用

- 如果只想对特定的方法进行拦截，对其他的方法直接放行，不做任何操作，可使用CallbackFilter：

```
@Test
public void testCallbackFilter() throws Exception{
    Enhancer enhancer = new Enhancer();
    CallbackHelper callbackHelper =
        new CallbackHelper(SampleClass.class, new Class[0]) {

        @Override
        protected Object getCallback(Method method) {
            if(method.getDeclaringClass() != Object.class
               && method.getReturnType() == String.class){
                return new FixedValue() {
                    @Override
                    public Object loadObject() throws Exception {
```

```
                    return "Hello cglib";
                }
            };
        }else{
            return NoOp.INSTANCE;
        }
    }
};
enhancer.setSuperclass(SampleClass.class);
enhancer.setCallbackFilter(callbackHelper);
enhancer.setCallbacks(callbackHelper.getCallbacks());
SampleClass proxy = (SampleClass) enhancer.create();
Assert.assertEquals("Hello cglib", proxy.test(null));
Assert.assertNotEquals("Hello cglib",proxy.toString());
System.out.println(proxy.hashCode());
}
```

3.2 ImmutableBean

通过名字就可以知道，不可变的Bean。ImmutableBean允许创建一个原来对象的包装类，这个包装类是不可变的，任何改变底层对象的包装类操作都会抛出IllegalStateException。但是我们可以通过直接操作底层对象来改变包装类对象。这有点类似于Guava中的不可变视图。上示例：

```
public class SampleBean {
    private String value;

    public SampleBean() {
    }

    public SampleBean(String value) {
        this.value = value;
    }
    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}

@Test(expected = IllegalStateException.class)
public void testImmutableBean() throws Exception{
    SampleBean bean = new SampleBean();
    bean.setValue("Hello world");
    SampleBean immutableBean =
        (SampleBean) ImmutableBean.create(bean); //创建不可变类
    Assert.assertEquals("Hello world",immutableBean.getValue());
    bean.setValue("Hello world, again"); //可以通过底层对象来进行修改
    Assert.assertEquals("Hello world, again", immutableBean.getValue());
    immutableBean.setValue("Hello cglib"); //直接修改将throw exception
}
```

3.3 Bean generator

在运行时动态的创建一个bean

```
@Test
public void testBeanGenerator() throws Exception{
    BeanGenerator beanGenerator = new BeanGenerator();
    beanGenerator.addProperty("value",String.class);
    Object myBean = beanGenerator.create();
```

```
    Method setter = myBean.getClass().getMethod("setValue",String.class);
    setter.invoke(myBean,"Hello cglib");

    Method getter = myBean.getClass().getMethod("getValue");
    Assert.assertEquals("Hello cglib",getter.invoke(myBean));
}
```

3.4 Bean Copier

cglib提供的能够从一个bean复制到另一个bean中，而且其还提供了一个转换器，用来在转换的时候对bean的属性进行操作。

```
public class OtherSampleBean {
    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}

@Test
public void testBeanCopier() throws Exception{
    /* 设置为true，则使用converter */
    /* SampleBean前面其他示例中有代码 */
    BeanCopier copier = BeanCopier.create(SampleBean.class,
                                          OtherSampleBean.class, false);
    SampleBean myBean = new SampleBean();
    myBean.setValue("Hello cglib");
    OtherSampleBean otherBean = new OtherSampleBean();
    copier.copy(myBean, otherBean, null); //设置为true，则传入converter指明怎么进行转换
   assertEquals("Hello cglib", otherBean.getValue());
}
```

3.5 BulkBean

相比于BeanCopier，BulkBean将copy的动作拆分为getPropertyValues和setPropertyValues两个方法，允许自定义处理属性

```
@Test
public void testBulkBean() throws Exception{
    BulkBean bulkBean = BulkBean.create(SampleBean.class,
            new String[]{"getValue"},
            new String[]{"setValue"},
            new Class[]{String.class});
    SampleBean bean = new SampleBean();
    bean.setValue("Hello world");
    Object[] propertyValues = bulkBean.getPropertyValues(bean);
    assertEquals(1, bulkBean.getPropertyValues(bean).length);
    assertEquals("Hello world", bulkBean.getPropertyValues(bean)[0]);
    bulkBean.setPropertyValues(bean,new Object[]{"Hello cglib"});
    assertEquals("Hello cglib", bean.getValue());
}
```

注意：

1. 避免每次进行BulkBean.create创建对象，一般将其声明为static的
2. 应用场景：针对特定属性的get,set操作，一般适用通过xml配置注入和注出的属性，运行时才确定处理的Source,Target类，只需要关注属性名即可。

3.6 BeanMap

BeanMap类实现了Java Map，将一个bean对象中的所有属性转换为一个String-to-Obejct的Java Map

```
@Test
public void testBeanMap() throws Exception{
    /* 创建一个对象 */
    BeanGenerator generator = new BeanGenerator();
    generator.addProperty("username",String.class);
    generator.addProperty("password",String.class);
    Object bean = generator.create();
    Method setUserName = bean.getClass().getMethod("setUsername", String.class);
    Method setPassword = bean.getClass().getMethod("setPassword", String.class);
    setUserName.invoke(bean, "admin");
    setPassword.invoke(bean,"password");

    /* 使用BeanMap */
    BeanMap map = BeanMap.create(bean);
    Assert.assertEquals("admin", map.get("username"));
    Assert.assertEquals("password", map.get("password"));
}
```

3.7 keyFactory

keyFactory类用来动态生成接口的实例，接口需要只包含一个newInstance方法，返回一个Object。keyFactory为构造出来的实例动态生成了Object.equals和Object.hashCode方法，能够确保相同的参数构造出的实例为单例的。

```
public interface SampleKeyFactory {
    Object newInstance(String first, int second);
}

@Test
public void testKeyFactory() throws Exception{
    SampleKeyFactory keyFactory =
        (SampleKeyFactory) KeyFactory.create(SampleKeyFactory.class);
    Object key = keyFactory.newInstance("foo", 42);
    Object key1 = keyFactory.newInstance("foo", 42);
    Assert.assertEquals(key,key1);//测试参数相同，结果是否相等
}
```

> cglib中重要的唯一标识生成器，用于cglib做cache时做map key，比较底层的基础类。

> 每个Key接口，都必须提供newInstance方法，但具体的参数可以随意定义，通过newInstance返回的为一个唯一标示，只有当传入的所有参数的equals都返回true时，生成的key才是相同的，这就相当于多key的概念。

3.8 Mixin(混合)

Mixin能够让我们将多个对象整合到一个对象中去，前提是这些对象必须是接口的实现。

```
public class MixinInterfaceTest {
    interface Interface1{
        String first();
    }
    interface Interface2{
        String second();
    }
```

```
    class Class1 implements Interface1{
        @Override
        public String first() {
            return "first";
        }
    }

    class Class2 implements Interface2{
        @Override
        public String second() {
            return "second";
        }
    }

    interface MixinInterface extends Interface1, Interface2{

    }

    @Test
    public void testMixin() throws Exception{
        Mixin mixin = Mixin.create(new Class[]{Interface1.class, Interface2.class,
                        MixinInterface.class}, new Object[]{new Class1(),
                                                            new Class2()});
        MixinInterface mixinDelegate = (MixinInterface) mixin;
        assertEquals("first", mixinDelegate.first());
        assertEquals("second", mixinDelegate.second());
    }
}
```

Mixin类比较尴尬，因为他要求Minix的类（例如MixinInterface）实现一些接口。既然被Minix的类已经实现了相应的接口，那么我就直接可以通过纯Java的方式实现，没有必要使用Minix类。

3.9 String switcher

用来模拟一个String到int类型的Map类型。如果在Java7以后的版本中，类似一个switch语句。

```
@Test
public void testStringSwitcher() throws Exception{
    String[] strings = new String[]{"one", "two"};
    int[] values = new int[]{10,20};
    StringSwitcher stringSwitcher = StringSwitcher.create(strings,values,true);
    assertEquals(10, stringSwitcher.intValue("one"));
    assertEquals(20, stringSwitcher.intValue("two"));
    assertEquals(-1, stringSwitcher.intValue("three"));
}
```

3.10 Interface Maker

Interface Maker用来创建一个新的Interface

```
@Test
public void testInterfaceMarker() throws Exception{
    Signature signature = new Signature("foo", Type.DOUBLE_TYPE,
                                       new Type[]{Type.INT_TYPE});
    InterfaceMaker interfaceMaker = new InterfaceMaker();
    interfaceMaker.add(signature, new Type[0]);
    Class iface = interfaceMaker.create();
    assertEquals(1, iface.getMethods().length);
    assertEquals("foo", iface.getMethods()[0].getName());
    assertEquals(double.class, iface.getMethods()[0].getReturnType());
}
```

- 上述的Interface Maker创建的接口中只含有一个方法，签名为double foo(int)。
- Interface Maker与上面介绍的其他类不同，它依赖ASM中的Type类型。
- 由于接口仅仅只用做在编译时期进行类型检查，因此在一个运行的应用中动态的创建接口没有什么作用。但是InterfaceMaker可以用来自动生成代码，为以后的开发做准备。

3.11 Method delegate

MethodDelegate主要用来对方法进行代理

```
interface BeanDelegate{
    String getValueFromDelegate();
}

@Test
public void testMethodDelegate()  throws Exception{
    SampleBean bean = new SampleBean();
    bean.setValue("Hello cglib");
    BeanDelegate delegate = (BeanDelegate) MethodDelegate.create(bean,"getValue", BeanDelegate.class);
    assertEquals("Hello cglib", delegate.getValueFromDelegate());
}
```

关于Method.create的参数说明：

1. 第二个参数为即将被代理的方法
2. 第一个参数必须是一个无参数构造的bean。因此MethodDelegate.create并不是你想象的那么有用
3. 第三个参数为只含有一个方法的接口。当这个接口中的方法被调用的时候，将会调用第一个参数所指向bean的第二个参数方法

3.12 MulticastDelegate

1. 多重代理和方法代理差不多，都是将代理类方法的调用委托给被代理类。使用前提是需要一个接口，以及一个类实现了该接口
2. 通过这种interface的继承关系，我们能够将接口上方法的调用分散给各个实现类上面去。
3. 多重代理的缺点是接口只能含有一个方法，如果被代理的方法拥有返回值，那么调用代理类的返回值为最后一个添加的被代理类的方法返回值

```
public interface DelegatationProvider {
    void setValue(String value);
}

public class SimpleMulticastBean implements DelegatationProvider {
    private String value;
    @Override
    public void setValue(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}

@Test
public void testMulticastDelegate() throws Exception{
    MulticastDelegate multicastDelegate =
        MulticastDelegate.create(DelegatationProvider.class);
    SimpleMulticastBean first = new SimpleMulticastBean();
    SimpleMulticastBean second = new SimpleMulticastBean();
    multicastDelegate = multicastDelegate.add(first);
```

```
    multicastDelegate  = multicastDelegate.add(second);

    DelegatationProvider provider = (DelegatationProvider) multicastDelegate;
    provider.setValue("Hello world");

    assertEquals("Hello world", first.getValue());
    assertEquals("Hello world", second.getValue());
}
```

3.13 FastClass

顾明思义，FastClass就是对Class对象进行特定的处理，比如通过数组保存method引用，因此FastClass引出了一个index下标的新概念，比如getIndex(String name, Class[] parameterTypes)就是以前的获取method的方法。通过数组存储method,constructor等class信息，从而将原先的反射调用，转化为class.index的直接调用，从而体现所谓的FastClass。

```
@Test
public void testFastClass() throws Exception{
    FastClass fastClass = FastClass.create(SampleBean.class);
    FastMethod fastMethod = fastClass.getMethod("getValue",new Class[0]);
    SampleBean bean = new SampleBean();
    bean.setValue("Hello world");
    assertEquals("Hello world",fastMethod.invoke(bean, new Object[0]));
}
```

4. 注意

- 内存问题

> 由于CGLIB的大部分类是直接对Java字节码进行操作，这样生成的类会在Java的永久堆中。如果动态代理操作过多，容易造成永久堆满，触发OutOfMemory异常。

5. CGLIB和Java动态代理的区别

1. Java动态代理只能够对接口进行代理，不能对普通的类进行代理（因为所有生成的代理类的父类为Proxy，Java类继承机制不允许多重继承）；CGLIB能够代理普通类；
2. Java动态代理使用Java原生的反射API进行操作，在生成类上比较高效；CGLIB使用ASM框架直接对字节码进行操作，在类的执行过程中比较高效

相关博客：

- CGlib 入门
- CGLIB(Code Generation Library)详解