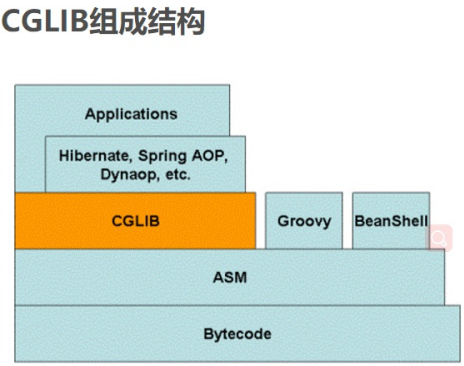
CGLIB

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；

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

cglib的API

- **ImmutableBean**

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

```
@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
}
```

value是sampleBean的一个属性字段

- **Bean generator**

Bean generator

cglib提供的一个操作bean的工具，使用它能够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));
}
```

在上面的代码中，我们使用cglib动态的创建了一个和SampleBean相同的Bean对象，包含一个属性value以及getter、setter方法

- **Bean Copier**

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{
    BeanCopier copier = BeanCopier.create(SampleBean.class, OtherSampleBean.class, false);//设置为t
    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());
}
```

- **BulkBean**

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());
}
```

使用注意：
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。

- **BeanMap**

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{      cglib的一个类
    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");
    BeanMap map = BeanMap.create(bean);
    Assert.assertEquals("admin", map.get("username"));
    Assert.assertEquals("password", map.get("password"));
}
```

- **keyFactory**

keyFactory 为接口创建实例

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);
}
```

我们首先构造一个满足条件的接口，然后进行测试

```
@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
}
```

- **Mixin(混合)**

```
    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());
    }
}
```

也许可以考虑与适配器模式结合

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。

- **String switcher → 2个数组构建map**

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"));
}
```

- **Interface Makere**

Interface Maker

正如名字所言，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。

- **Parallel Sorter(并行排序器)**

Parallel Sorter(并行排序器)

能够对多个数组同时进行排序，目前实现的算法有归并排序和快速排序

```
@Test
public void testParallelSorter() throws Exception{
    Integer[][] value = {
            {4, 3, 9, 0},
            {2, 1, 6, 0}
    };
    ParallelSorter.create(value).mergeSort(0);
    for(Integer[] row : value){
        int former = -1;
        for(int val : row){
            assertTrue(former < val);
            former = val;
        }
    }
}
```

注意

- 内存问题

注意

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。