Singleton

饿汉式

```
public class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton() {

    }
    public static Singleton getInstance() {
        return singleton;
    }
}
```

懒汉式(线程不安全)

```
public class Singleton {
    private static Singleton singleton;
    private Singleton() {

    }
    public static Singleton getInstance() {
        if (singleton == null) {
            return new Singleton();
        }
            return singleton;
    }
}
```

懒汉式(线程安全)DCL（Double Check Lock） 双重校验锁

```
public class Singleton {
    private static Singleton singleton;
    private Singleton() {

    }
    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    return new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

静态内部类单例模式

```
public class Singleton {
    private static Singleton singleton;
    private Singleton() {

    }
    public static Singleton getInstance() {
       return SingletonHolder.INSTANCE;
    }

    private static class SingletonHolder{
        private static final Singleton INSTANCE = new Singleton();
    }
}
```

Factory Method

普通工厂模式

```
public interface Sender {
    public void send();
}
```

```
public class EmailSender implements Sender {
    @Override
    public void send() {
        System.out.println("Email sender!");
    }
}
```

```
public class MsSender implements Sender {
    @Override
    public void send() {
        System.out.println("MS sender!");
    }
}
```

```
public class SenderFactory {
    public Sender produce(String name) {
        if ("ms".equals(name)) {
            return new MsSender();
        } else if ("email".equals(name)) {
            return new EmailSender();
        } else {
            System.out.println("输入有误");
            return null;
        }
    }
}
```

测试：

```
public class FactoryTest {
    public static void main(String[] args) {
        SenderFactory factory = new SenderFactory();
        Sender sender = factory.produce("email");
        sender.send();
    }
}
```

多个工厂方法模式

```
public class SenderFactory {
    public Sender produceMsSender() {
        return new MsSender();
    }
    public Sender produceMailSender() {
        return new EmailSender();
    }
}
```

测试：

```
public class FactoryTest {
    public static void main(String[] args) {
        SenderFactory factory = new SenderFactory();
        Sender sender = factory.produceMailSender();
        sender.send();
    }
}
```

静态工厂模式

```
public class SenderFactory {
    public static Sender produceMsSender() {
        return new MsSender();
    }
    public static Sender produceMailSender() {
        return new EmailSender();
    }
}
```

测试：

```
public class FactoryTest {
    public static void main(String[] args) {
        Sender sender = SenderFactory.produceMailSender();
        sender.send();
    }
}
```

抽象工厂模式

```
public interface Provider {
    public Sender produce();
}
```

```
public class MsFactory implements Provider {
    @Override
    public Sender produce() {
        return new MsSender();
    }
}
```

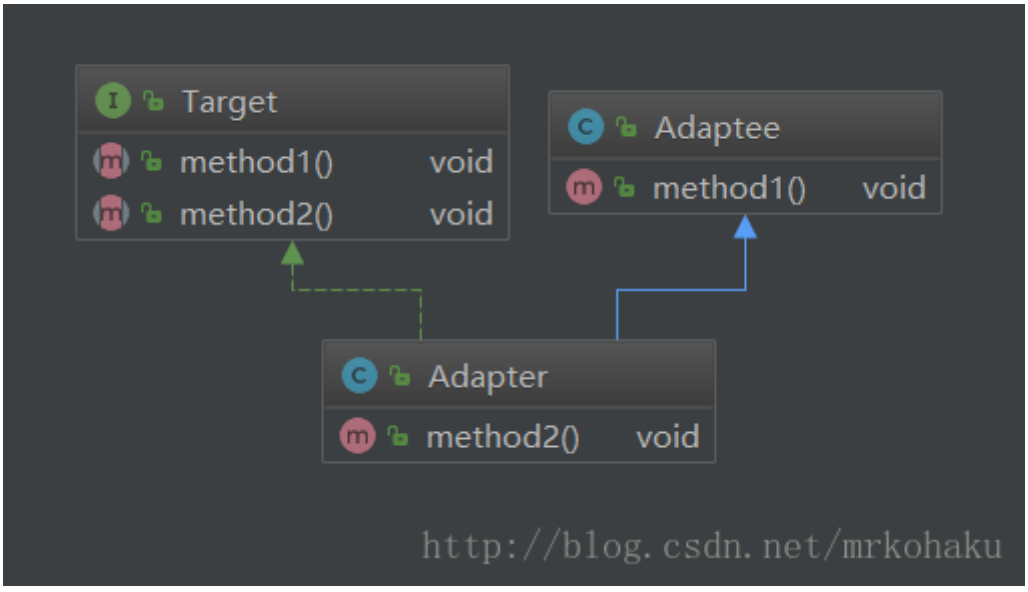
```
public class EmailSender implements Sender {
    @Override
    public void send() {
        System.out.println("Email sender!");
    }
}
```

测试:

```
public class FactoryTest {
    public static void main(String[] args) {
        EmailFactory factory = new EmailFactory();
        Sender sender = factory.produce();
        sender.send();
    }
}
```

Adapter Mode

类适配器模式



```
public interface Target {
    void f1();
    void f2();
}
```

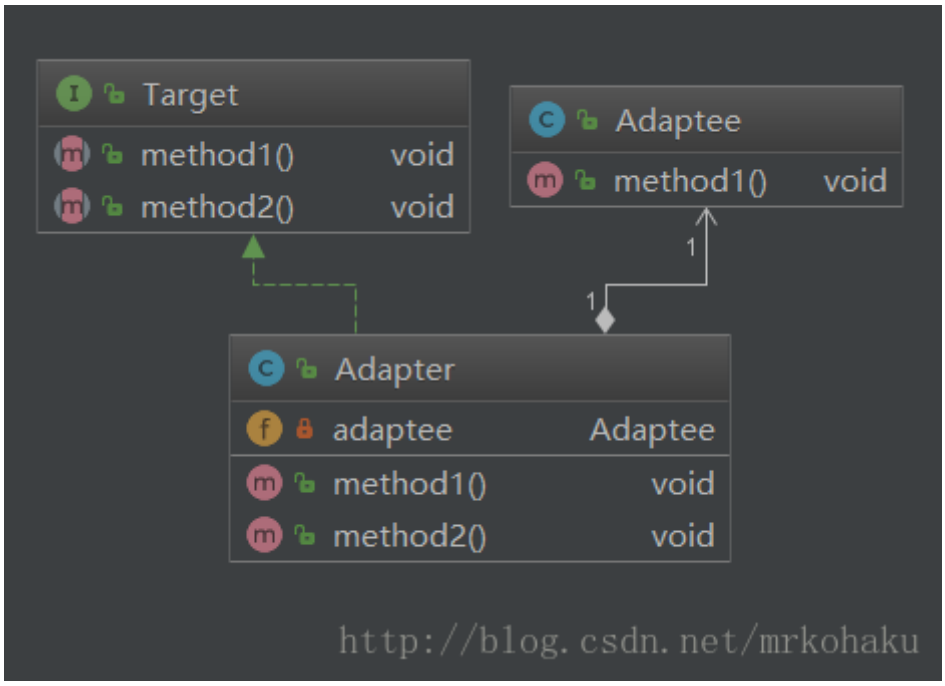
```
public class Adaptee {
    public void f1() {
        System.out.println("f1");
    }
}
```

```
public class Adapter extends Adaptee implements Target {
    @Override
    public void f2() {
        System.out.println("f2");
    }
}
```

测试:

```
public class Main {
    public static void main(String[] args) {
        Target target = new Adapter();
        target.f1();
        target.f2();
    }
}
```

对象适配器模式




```
public class Adapter implements Target {
    private Adaptee adaptee;
```

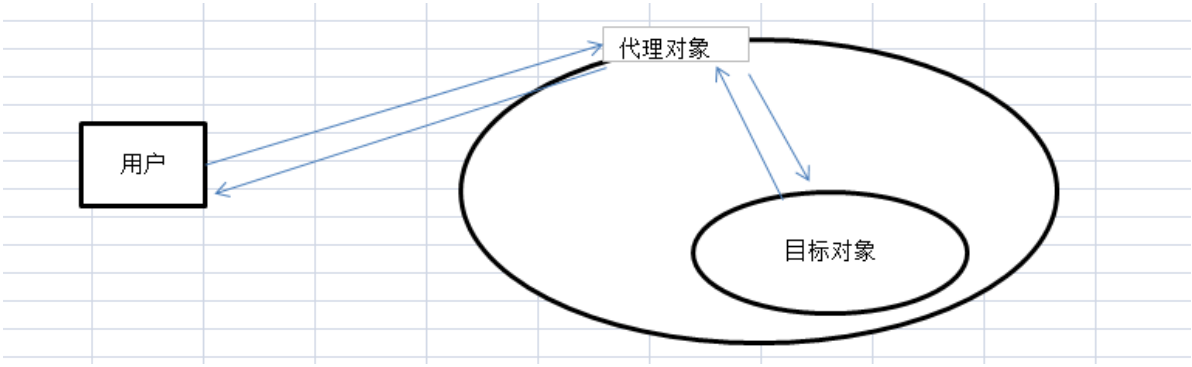
```
    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void f2() {
        System.out.println("f2");
    }

    @Override
    public void f1() {
        adaptee.f1();
    }
}
```

测试:

```
public class Main {
    public static void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        target.f1();
        target.f2();
    }
}
```

Proxy Mode

静态代理模式




```
public interface IUserDao {
    void save();
}
```

```
public class UserDao implements IUserDao {
    @Override
    public void save() {
        System.out.println("保存成功");
    }
}
```

```
public class UserDaoProxy implements IUserDao {
    private UserDao userDao;

    public UserDaoProxy(UserDao userDao) {
        this.userDao = userDao;
    }

    @Override
    public void save() {
        System.out.println("开启事务");
        userDao.save();
        System.out.println("提交事务");
    }
}
```

测试:

```
public class Main {
    public static void main(String[] args) {
        UserDaoProxy proxy = new UserDaoProxy(new UserDao());
        proxy.save();
    }
}
```

动态代理模式

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

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

```
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());
        Object invoke = method.invoke(object, args);
        System.out.println("After invoke " + method.getName());
        return invoke;
    }
}
```

测试:

```
public class Main {
    public static void main(String[] args) {
        HelloInterface hello = new Hello();
        ProxyHandler handler = new ProxyHandler(hello);
        HelloInterface h = (HelloInterface)
Proxy.newProxyInstance(hello.getClass().getClassLoader(),
hello.getClass().getInterfaces(), handler);
        h.sayHello();
    }
}
```

总结:
代理对象不需要实现接口,但是目标对象一定要实现接口,否则不能用动态代理

写一个 ArrayList 的动态代理类(变量修饰问题):

```
public class Main {
    public static void main(String[] args) {
      final List list =  new ArrayList<String>();
      List proxyInstance = (List)
Proxy.newProxyInstance(list.getClass().getClassLoader(),
list.getClass().getInterfaces(), new InvocationHandler() {
          @Override
          public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
              return method.invoke(list, args);
          }
      });
      proxyInstance.add("你好");
        System.out.println(proxyInstance);
    }
}
```

Cglib代理模式

```
/**
 * Cglib子类代理工厂
 * 对UserDao在内存中动态构建一个子类对象
 */
public class ProxyFactory implements MethodInterceptor{
    //维护目标对象
    private Object target;

    public ProxyFactory(Object target) {
        this.target = target;
    }

    //给目标对象创建一个代理对象
    public Object getProxyInstance(){
        //1.工具类
        Enhancer en = new Enhancer();
        //2.设置父类
```

```
        en.setSuperclass(target.getClass());
        //3.设置回调函数
        en.setCallback(this);
        //4.创建子类(代理对象)
        return en.create();

    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
        System.out.println("开始事务...");

        //执行目标对象的方法
        Object returnValue = method.invoke(target, args);

        System.out.println("提交事务...");

        return returnValue;
    }
}
```

测试:

```
public class Main {
    public static void main(String[] args) {
        //目标对象
        UserDao target = new UserDao();

        //代理对象
        UserDao proxy = (UserDao)new ProxyFactory(target).getProxyInstance();

        //执行代理对象的方法
        proxy.save();
    }
}
```