设计模式详解：http://c.biancheng.net/design_pattern/

开闭原则：当有新功能的时候不需要修改已有的代码，只需要通过新增接口就能实现功能。

三类设计模式(https://baijiahao.baidu.com/s?id=1639156298714178350&wfr=spider&for=pc)：

- **结构性模式：主要用于定义如何使用多个对象组合出一个或多个复合对象**

1、单例模式实现方式：静态内部类、双重检查锁、枚举

- **静态内部类实现单例模式原理：根据jvm类加载原理我们可知当获取静态内部类中的单例静态对象时才会触发静态内部类的初始化（类的加载时机中的getstatic情况），这样就起到了单例对象延时加载的目的。同时类初始化阶段是执行类构造器<clinit>（）方法的过程，虚拟机会保证一个类的<clinit>（）方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>（）方法，这样就保证只有一个对象被创建。**

2、工厂模式：当对象创建过程比较复杂，初始化参数比较多的情况下，可以使用工厂模式将对象创建过程封装在工厂类中，然后将工厂类提供给客户端调用。因此使用工厂类的意义在于对调用者来说简化了对象创建过程。Executors使用了简单工厂模式（创建不同的线程池）

3、策略模式（个人理解可以和枚举搭配使用）

- **原理：策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们可以相互替换，让算法独立于使用它的客户而独立变化。**
- **使用场景：针对同一类型问题的多种处理方式，仅仅是具体行为有差别时，反应到代码中就是一个接口有多个实现类，而又需要使用 if-else 或者 switch-case**

来选择具体子类时可使用策略模式。spring的IOC就是对策略模式很好的延伸，我们可以根据不同需求注入不同的实现类，即在spring中策略模式无处不在。

- 示例：在mybatis的<environment/>标签中配置数据源类型就是策略模式的体现：

```
<!--配置environment环境 -->
<environments default="development">
    <!-- 环境配置1，每个SqlSessionFactory对应一个环境 -->
    <environment id="development">
        <transactionManager type="JDBC" />
        <dataSource type="UNPOOLED">
            <property name="driver" value="${jdbc_driver}" />
            <property name="url" value="${jdbc_url}" />
            <property name="username" value="${jdbc_username}" />
            <property name="password" value="${jdbc_password}" />
        </dataSource>
    </environment>
</environments>
```

此处可以配置UNPOOLED、POOLED、JNDI三种不同的数据源，无论配置那种数据源代码都不需要进行修改，他根据配置去找相应的实现就OK，很好的体现了策略模式的思想

- 当我们的项目不使用spring时如何使用策略模式替换大量的if else：我们可以借鉴spring的思路，将接口所有的实现类都放到一个map中，然后根据不同的tag调用不同的实现类。因为spring就是一个大的map，我们使用@Autowired注入的时候其实就是在使用策略模式去容器中拿不同的实现类。
- 策略枚举使用场景：当一个枚举的多个实例有相同行为的时候可以使用策略枚举，先定义一个私有枚举类，然后将不同的行为抽取成私有枚举类中的不同实例，不同实例中去实现不同行为（枚举类中可以定义抽象方法）。这样既避免了写重复代码，又屏蔽了具体实现。看study工程中的enumtest或者vip-jvm工程中的例子。

4、适配器模式

- 原理：适配器类实现目标接口，在适配器类中调用适配者的方法。
- 使用场景：加入系统需要调微信、支付宝的支付功能，则可在自己系统内统一定义一个支付接口，然后创建微信和支付宝的适配器类，这样可扩展性强、代码可读性强、易维护。
- 示例：mybatis的日志源码使用了该模式

5、动态代理（jdk动态代理被代理类必须实现接口，而cglib是基于字节码增强的不需要实现接口）

- 作用：无侵入的增强业务类。相反静态代理会有侵入。
- 重点：对于没有实现类的接口，JDK也可以进行动态代理。mybatis对mapper的增强就是这样做的，同时RPC框架中也是这样做的。说白了就是在调用接

口方法时候做一些其他的操作。

对于有实现类和没实现类的区别是生成代理对象的时候传参有一点区别：

- 无实现类以RPC框架和mybatis为例：

```
public class RpcClientFrame {
    public static  <T> T getRemoteProxy(final Class<?> service){
        InetSocketAddress address = new InetSocketAddress( hostname: "127.0.0.1", port: 9999);
        return (T)Proxy.newProxyInstance(service.getClassLoader(), new Class[]{service},new RpcProxy(service,address));
    }

    private static class RpcProxy implements InvocationHandler{
        private InetSocketAddress inetSocketAddress;
        private Class<?> service;
        private RpcProxy(Class<?> clazz,InetSocketAddress inetSocketAddress){
            this.service = clazz;
            this.inetSocketAddress = inetSocketAddress;
        }
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            ObjectInputStream inputStream = null;
            ObjectOutputStream outputStream = null;
            Socket socket = null;
            try{
                socket = new Socket();
                socket.connect(inetSocketAddress);
                outputStream = new ObjectOutputStream(socket.getOutputStream());
                outputStream.writeUTF(service.getName());
                outputStream.writeUTF(method.getName());
                outputStream.writeObject(method.getParameterTypes());
                outputStream.writeObject(args);
                outputStream.flush();
                System.out.println("远程服务调用成功！");
                inputStream = new ObjectInputStream(socket.getInputStream());
                Object result = inputStream.readObject();
                System.out.println("读取数成功！");
                return result;
```

获取接口本身的class对象，不需要getInterfaces()

此处传入的接口对应的class对象

进行网络增强，远程调用其他服务的实现类

```
/unchecked/
protected T newInstance(MapperProxy<T> mapperProxy) {
  //创建实现了mapper接口的动态代理对象
  return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface }, mapperProxy
}

public T newInstance(SqlSession sqlSession) {
  //每次调用都会创建新的MapperProxy对象
  final MapperProxy<T> mapperProxy = new MapperProxy<>(sqlSession, mapperInterface, methodCache);
  return newInstance(mapperProxy);
}
```

对mapper接口增强时，传入mapper接口本身对应的class对象

- 有实现类时以自写的例子为例（Mybatis的SqlSessionManager的例子也值得看）：

```
 */
public class StudyHandler implements InvocationHandler {

    private Class<? extends StudyService> clzz;

    public StudyHandler(final Class<? extends StudyService> clzz){
        this.clzz = clzz;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        StrongClass.beforStudy(args[0].toString());
        try{
            method.invoke(this.clzz.newInstance(),args);
//              throw new RuntimeException("学习困了");
        }catch (Exception e){
            StrongClass.ifException(e);
        }
        StrongClass.afterStudy();
        return null;
    }
    public StudyService getProxyInstance(){
        return (StudyService) Proxy.newProxyInstance(this.clzz.getClassLoader(),this.clzz.getInterfaces(), h: this);
    }
}
```

此处传入的是接口实现类的class对象

对于增强接口，因为没有实现类，所以不能像这里这样通过反射调用实现类方法，不过可以像RPC那样对网络进行增强或者像mybatis那样通过Mappermethod操作sqlSession进行数据库处理

此处通过反射获取实现类对象，并通过反射调用方法

此处获取接口实现类实现了那些接口

这也是和增强接口的不同之处，
增强接口此处传入的接口自己的class对象

- 代理类的.class文件需要细看。代理类继承了Proxy类并实现了被代理的接口，而Proxy类中维护了一个InvocationHandler属性。当我们调用接口的方法时其实调用的时InvocationHandler的invoke()方法：


```
public final class $BookProxy extends Proxy
  implements Book, Meet
{
  private static Method m1;
  private static Method m4;
  private static Method m2;
  private static Method m3;
  private static Method m0;

  public $BookProxy(InvocationHandler paramInvocationHandler)
    throws
  {
    super(paramInvocationHandler);
  }

  public final boolean equals(Object paramObject)
    throws
  {
    try
    {
      return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject })).booleanValue();
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }

  public final void eat()
    throws
  {
    try
    {
      this.h.invoke(this, m4, null);
      return;
    }
    catch (Error|RuntimeException localError)
    {
      throw localError;
    }
    catch (Throwable localThrowable)
    {
      throw new UndeclaredThrowableException(localThrowable);
    }
  }
```

- **CGLIB不支持final Class, 因为CGLIB是生成子类来实现AOP,所以final Class自然无法支持了。同时也无法增强被final修饰的方法，因为它是通过继承重写方法实现增强的，而final修饰的方法是不能被重写的。**
- **动态代理底层原理是直接生成二进制字节码文件（即class文件），它跳过了编写java文件这一步，在学习JVM的时候就知道只要对字节码足够熟悉那么我们可以直接写class文件交给虚拟机去执行，写java文件还需要进行编译。cglib底层是通过修改被增强类的字节码文件进行代理的。**

6、装饰器模式（和代理模式的区别）

- **作用：允许向一个现有的对象添加新的功能，是一种用于代替继承的技术，无需通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀；**
- **示例：java中的IO就是最典型的装饰器模式：**

- **mybatis中缓存模块使用装饰器模式时没有定义装饰器抽象父类。但是mybatis中的Cache可以被多层装饰，当我们选用LRU缓存淘汰策略时PerpetualCache被LruCache装饰，当设置了blocking属性时LruCache会再次被包装成BlockingCache。**
- **使用装饰父类和不使用装饰器父类的区别在于使用了装饰父类则其他每个装饰器中不需要在额外定义被装饰者的实例字段，它只是起了一个将被装饰者抽出来定**

义到父类中供具体装饰器调用的作用。

- 装饰器模式就是通过复合实现的。

7、建造者模式

- 与工厂模式区别：获取定制对象和获取普通对象的区别。建造者模式可以按需给属性赋值。
- 原理：注重于按照不同的配件加载顺序构造对象（即给属性赋值的顺序按需调整）
- 建造者模式可以使用Fluent编程风格调整属性装配组合，或者结合模板方法模式调整属性装配组合（即在Director中搭配不同组合）。Fluent相对于模板灵活一点。

8、模板方法模式

- 原理：我们可以在抽象父类中定义一个模板方法，规定方法的执行顺序，而将方法的实现推迟到子类中完成。抽象父类中包含抽象方法、具体方法、模板方法、钩子方法。
- 钩子方法：钩子方法源于模板方法模式，它由抽象父类声明并且实现，子类也可以选择加以扩展，钩子方法可能会影响模板方法的执行结果。
- 实现扩展：有了钩子方法模板方法模式才算完美，大家可以想想，由子类的一个方法返回值决定公共部分的执行结果，是不是很有吸引力呀。

9、享元模式：“享”就是分享之意，指一物被众人共享，而这也正是该模式的终旨所在。

- 原理：享元模式有点类似于单例模式，都是只生成一个对象来被共享使用。这里有个问题，那就是对共享对象的修改，为了避免出现这种情况，我们将这些对象的公共部分，或者说是不变化的部分抽取出来形成一个对象。这个对象就可以避免到修改的问题。享元的目的是为了减少不会要额内存消耗，将多个对同一对象的访问集中起来，不必为每个访问者创建一个单独的对象，以此来降低内存的消耗。

- 使用场景：JDK中的Integer、Long等包装类中使用了享元模式，将常用的对象（-128~127）缓存起来达到共享复用的目的。

10、原型模式：原型模式的主要作用是可以利用现有的类对象通过复制（克隆）的方式创建一个新的对象。当示例化一个类的对象需要耗费大量的时间和系统资源时，可是采用原型模式，将原始已存在的对象通过复制（克隆）机制创建新的对象，然后根据需要，对新对象进行修改。原型模式要求被复制的对象自身具备拷贝功能，此功能不能由外界完成。Java提供了一个Cloneable接口来标示这个对象是可拷贝的，为什么说是“标示”呢？翻开JDK的帮助看看Cloneable是一个方法都没有的，这个接口只是一个标记作用，在JVM中具有这个标记的对象才有可能被拷贝（一定要实现Cloneable接口）。

- 浅克隆：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。
- 浅克隆实现：重写Object类提供的clone()方法，并在重写的clone()方法里边必须调用Object的clone()方法（重写是因为Object类中的clone()方法是protected修饰的，我们重写时要将修饰符改为public以供其他类调用），因为Object类中的clone()方法是被native修饰的，只有通过它才能实现克隆。
- 深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。
- 深克隆两种方式：
    - 序列化：通过流进行深克隆，不需要实现Cloneable接口，但需要实现Serialable接口，因为要进行流转化。
    - 依次克隆各个引用类型属性：这种方式下属性所引用的类型也要实现浅克隆，即重写clone()方法。

11、观察者模式（Observer）：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。简单来说，如果你需要在对象状态发生改变时及时收到通知，你可以定义一个监听器，对该对象的状态进行监听，此时的监听器即为观察者（Observer），被监听对象称为主题（Subject）。

- 原理：在目标对象（被监听者）或者第三方类中保存了所有观察者对象，当目标对象发生变化时遍历所有观察者对象并调用指定方法进行业务操作。

- **最佳实践：spring中的事件监听就是观察者模式的一个很优雅的实现，并且spring在SimpleApplicationEventMulticaster类中维护了一个ConcurrentHashMap用来存放监听器和监听事件的对应关系，当我们通过容器发布一个事件的时候spring会去SimpleApplicationEventMulticaster中找到监听这个事件的监听器，然后去一一调用这些监听器的onApplicationEvent()，这样就达到了监听目的：**