

# Java中的责任链设计模式

Mazin [Java后端](#) 2019-12-03

点击上方 [Java后端](#), 选择 [设为星标](#)

优质文章，及时送达

作者 | Mazin

来源 | [my.oschina.net/u/3441184/blog/889552](https://my.oschina.net/u/3441184/blog/889552)

责任链设计模式的思想很简单，就是按照链的顺序执行一个个处理方法，链上的每一个任务都持有它后面那个任务的对象引用，以方便自己这段执行完成之后，调用其后面的处理逻辑。

下面是一个责任链设计模式的简单的实现：

```
public interface Task {
    public void run();
}

public class Task1 implements Task{

    private Task task;

    public Task1() {}

    public Task1(Task task){
        this.task = task;
    }

    @Override
    public void run() {
        System.out.println("task1 is run");
        if(task != null){
            task.run();
        }
    }
}

public class Task2 implements Task{

    private Task task;

    public Task2() {}

    public Task2(Task task){
        this.task = task;
    }

    @Override
    public void run() {
```

```

        System.out.println("task2 is run");

        if(task != null){
            task.run();
        }
    }
}

```

```

public class Task3 implements Task{

    private Task task;

    public Task3() {}

    public Task3(Task task){
        this.task = task;
    }

    @Override
    public void run() {
        System.out.println("task3 is run");
        if(task != null){
            task.run();
        }
    }
}

```

以上代码是模拟了三个任务类，它们都实现了统一个接口，并且它们都有一个构造函数，可以在它们初始化的时候就持有另一个任务类的对象引用，以方便该任务调用。

Tips：欢迎关注微信公众号：Java后端，每日推送 Java 项目技术博文。

这个和服务器的过滤器有点类似，过滤器的实现也都是实现了同一个接口Filter。

```

public class LiabilityChain {

    public void runChain(){
        Task task3 = new Task1();
        Task task2 = new Task2(task3);
        Task task1 = new Task3(task2);
        task1.run();
    }

}

```

上面这段代码就是一个任务链对象，它要做的事情很简单，就是将所有要执行的任务都按照指定的顺序串联起来。

```
public class ChainTest {

    public static void main(String[] args) {
        LiabilityChain chain = new LiabilityChain();
        chain.runChain();
    }

}
```

当我们获取到责任链对象之后，调用其方法，得到以下运行结果：

```
terminated: ChainTest.java:Application 0: java.lang.RuntimeException:
task1 is run
task2 is run
task3 is run
|
```

以上是一个责任链的简单的实现，如果想要深入理解其思想，建议去观察一个过滤器链的执行源码。

作者：Mazin

编辑：id: javastack

<https://my.oschina.net/u/3441184/blog/889552>

【END】

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web\_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



#### 推荐阅读

1. Spring Boot 多模块项目实践

2. MyBatis大揭秘:Plugin 插件设计原理

3. 你知道 Spring Batch 吗?

4. 3 分钟带你彻底搞懂 Java 泛型背后的秘密

5. 团队开发中 Git 最佳实践

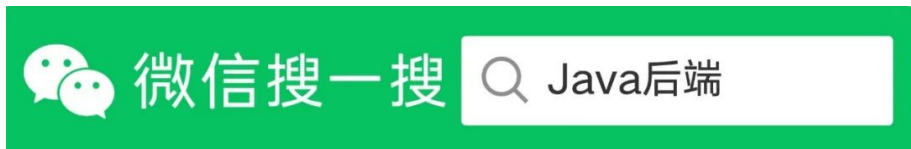


喜欢文章, 点个在看 🌟

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 必须掌握！你知道 Spring 中运用的 9 种设计模式吗？

iCoding91 Java后端 1周前



Spring中涉及的设计模式总结，在面试中也会经常问道 Spring 中设计模式的问题。本文以实现方式、实质、实现原理的结构简单介绍 Spring 中应用的 9 种设计模型，具体详细的剖析会在后面的文章发布，话不多说，来个转发、在看、收藏三连！

## 1. 简单工厂

### 实现方式：

BeanFactory。Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

### 实质：

由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

### 实现原理：

#### bean容器的启动阶段：

- 读取bean的xml配置文件,将bean元素分别转换成一个BeanDefinition对象。
- 然后通过BeanDefinitionRegistry将这些bean注册到beanFactory中，保存在它的一个ConcurrentHashMap中。
- 将BeanDefinition注册到了beanFactory之后，在这里Spring为我们提供了一个扩展的切口，允许我们通过实现接口BeanFactoryPostProcessor 在此处来插入我们定义的代码。

典型的例子就是：PropertyPlaceholderConfigurer，我们一般在配置数据库的dataSource时使用到的占位符的值，就是它注入进去的。

#### 容器中bean的实例化阶段：

实例化阶段主要是通过反射或者CGLIB对bean进行实例化，在这个阶段Spring又给我们暴露了很多的扩展点：

- **各种的Aware接口**，比如 BeanFactoryAware, 对于实现了这些Aware接口的bean, 在实例化bean时Spring会帮我们注入对应的BeanFactory的实例。
- **BeanPostProcessor接口**，实现了BeanPostProcessor接口的bean，在实例化bean时Spring会帮我们调用接口中的方法。
- **InitializingBean接口**，实现了InitializingBean接口的bean，在实例化bean时Spring会帮我们调用接口中的方法。
- **DisposableBean接口**，实现了BeanPostProcessor接口的bean，在该bean死亡时Spring会帮我们调用接口中的方法。

### 设计意义：

**松耦合**。可以将原来硬编码的依赖，通过Spring这个beanFactory这个工厂来注入依赖，也就是说原来只有依赖方和被依赖方，现在我们引入了第三方——spring这个beanFactory，由它来解决bean之间的依赖问题，达到了松耦合的效果。

**bean的额外处理。**通过Spring接口的暴露，在实例化bean的阶段我们可以进行一些额外的处理，这些额外的处理只需要让bean实现对应的接口即可，那么spring就会在bean的生命周期调用我们实现的接口来处理该bean。 **[非常重要]**

## 2. 工厂方法

### 实现方式：

FactoryBean接口。

### 实现原理：

实现了FactoryBean接口的bean是一类叫做factory的bean。其特点是，spring会在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法，所以返回的不是factory这个bean，而是这个bean.getObject()方法的返回值。

### 例子：

典型的例子有spring与mybatis的结合。

### 代码示例：

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="configLocation" value="classpath:config/mybatis-config-master.xml" />
  <property name="mapperLocations" value="classpath:config/mappers/master/**/*.xml" /> </bean>
```

### 说明：

我们看上面该bean，因为实现了FactoryBean接口，所以返回的不是 SqlSessionFactoryBean 的实例，而是它的 SqlSessionFactoryBean.getObject() 的返回值。

## 3. 单例模式

Spring依赖注入Bean实例默认是单例的。

Spring的依赖注入（包括lazy-init方式）都是发生在AbstractBeanFactory的getBean里。getBean的doGetBean方法调用getSingleton进行bean的创建。

分析getSingleton()方法

```

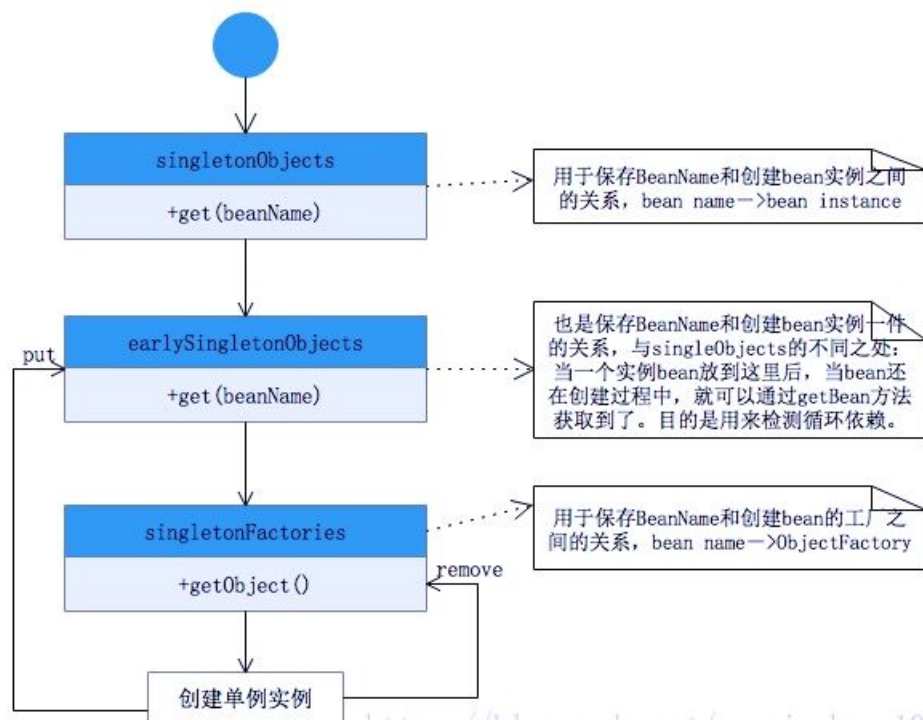
public Object getSingleton(String beanName){
    //参数true设置标识允许早期依赖
    return getSingleton(beanName,true);
}

protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    //检查缓存中是否存在实例
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        //如果为空，则锁定全局变量并处理。
        synchronized (this.singletonObjects) {
            //如果此bean正在加载，则不处理
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                //当某些方法需要提前初始化的时候则会调用addSingleFactory方法将对应的ObjectFactory初始化策略存储在singletonFactories
                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                if (singletonFactory != null) {
                    //调用预先设定的getObject方法
                    singletonObject = singletonFactory.getObject();
                    //记录在缓存中，earlySingletonObjects和singletonFactories互斥
                    this.earlySingletonObjects.put(beanName, singletonObject);
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return (singletonObject != NULL_OBJECT ? singletonObject : null);
}

```

## getSingleton()过程图

ps: spring依赖注入时, 使用了 双重判断加锁 的单例模式



<https://blog.csdn.net/caoxiaohong1005>

## 总结

**单例模式定义:** 保证一个类仅有一个实例, 并提供一个访问它的全局访问点。

**spring对单例的实现:** spring中的单例模式完成了后半句话, 即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例, 这是因为spring管理的是任意的java对象。

## 4. 适配器模式

### 实现方式：

SpringMVC中的适配器HandlerAdatper。

### 实现原理：

HandlerAdatper根据Handler规则执行不同的Handler。

### 实现过程：

DispatcherServlet根据HandlerMapping返回的handler，向HandlerAdatper发起请求，处理Handler。

HandlerAdapter根据规则找到对应的Handler并让其执行，执行完毕后Handler会向HandlerAdapter返回一个ModelAndView，最后由HandlerAdapter向DispatchServelet返回一个ModelAndView。

### 实现意义：

HandlerAdatper使得Handler的扩展变得容易，只需要增加一个新的Handler和一个对应的HandlerAdapter即可。

因此Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替controller执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC的扩展了。

## 5. 装饰器模式

### 实现方式：

Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

### 实质：

动态地给一个对象添加一些额外的职责。

就增加功能来说，Decorator模式相比生成子类更为灵活。

## 6. 代理模式

### 实现方式：

AOP底层，就是动态代理模式的实现。

### 动态代理：

在内存中构建的，不需要手动编写代理类

### 静态代理：



需要手工编写代理类，代理类引用被代理对象。

### 实现原理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象创建动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。

织入：把切面应用到目标对象并创建新的代理对象的过程。

## 7. 观察者模式

### 实现方式：

spring的事件驱动模型使用的是 观察者模式，Spring中Observer模式常用的地方是listener的实现。

### 具体实现：

事件机制的实现需要三个部分,事件源,事件,事件监听器

ApplicationEvent抽象类 **[事件]**

继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过构造器参数source得到事件源.

该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件.

代码：

```
public abstract class ApplicationEvent extends EventObject {
    private static final long serialVersionUID = 7099057708183571937L;
    private final long timestamp;
    public ApplicationEvent(Object source) {
        super(source);
        this.timestamp = System.currentTimeMillis();
    }
    public final long getTimestamp() {
        return this.timestamp;
    }
}
```

ApplicationListener接口 **[事件监听器]**

继承自jdk的EventListener,所有的监听器都要实现这个接口。

这个接口只有一个onApplicationEvent()方法,该方法接受一个ApplicationEvent或其子类对象作为参数,在方法体中,可以通过不同对Event类的判断来进行相应的处理。

当事件触发时所有的监听器都会收到消息。

代码：

```
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {
    void onApplicationEvent(E event);
}
```

ApplicationContext接口 [\[事件源\]](#)

ApplicationContext是spring中的全局容器，翻译过来是”应用上下文”。

实现了ApplicationEventPublisher接口。

### 职责：

负责读取bean的配置文档,管理bean的加载,维护bean之间的依赖关系,可以说是负责bean的整个生命周期,再通俗一点就是我们平时所说的IOC容器。

代码：

```
public interface ApplicationEventPublisher {  
    void publishEvent(ApplicationEvent event);  
}  
  
public void publishEvent(ApplicationEvent event) {  
    Assert.notNull(event, "Event must not be null");  
    if (logger.isTraceEnabled()) {  
        logger.trace("Publishing event in " + getDisplayName() + ": " + event);  
    }  
    getApplicationEventMulticaster().multicastEvent(event);  
    if (this.parent != null) {  
        this.parent.publishEvent(event);  
    }  
}
```

ApplicationEventMulticaster抽象类 [\[事件源中publishEvent方法需要调用其方法getApplicationEventMulticaster\]](#)

属于事件广播器,它的作用是把Applicationcontext发布的Event广播给所有的监听器。

代码：

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader  
    implements ConfigurableApplicationContext, DisposableBean {  
    private ApplicationEventMulticaster applicationEventMulticaster;  
    protected void registerListeners() {  
        // Register statically specified listeners first.  
        for (ApplicationListener<?> listener : getApplicationListeners()) {  
            getApplicationEventMulticaster().addApplicationListener(listener);  
        }  
        // Do not initialize FactoryBeans here: We need to leave all regular beans  
        // uninitialized to let post-processors apply to them!  
        String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);  
        for (String lisName : listenerBeanNames) {  
            getApplicationEventMulticaster().addApplicationListenerBean(lisName);  
        }  
    }  
}
```

## 8. 策略模式

### 实现方式：

Spring框架的资源访问Resource接口。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 接口来访问底层资源。

## Resource 接口介绍

source 接口是具体资源访问策略的抽象，也是所有资源访问类所实现的接口。

Resource 接口主要提供了如下几个方法：

- **getInputStream()**：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流。
- **exists()**：返回 Resource 所指向的资源是否存在。
- **isOpen()**：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束应该显式关闭，以防止资源泄漏。
- **getDescription()**：返回资源的描述信息，通常用于资源处理出错时输出该信息，通常是全限定文件名或实际 URL。
- **getFile**：返回资源对应的 File 对象。
- **getURL**：返回资源对应的 URL 对象。

最后两个方法通常无须使用，仅在通过简单方式访问无法实现时，Resource 提供传统的资源访问的功能。

Resource 接口本身没有提供访问任何底层资源的实现逻辑，**针对不同的底层资源，Spring 将会提供不同的 Resource 实现类，不同的实现类负责不同的资源访问逻辑。**

Spring 为 Resource 接口提供了如下实现类：

- **UrlResource**：访问网络资源的实现类。
- **ClassPathResource**：访问类加载路径里资源的实现类。
- **FileSystemResource**：访问文件系统里资源的实现类。
- **ServletContextResource**：访问相对于 ServletContext 路径里的资源的实现类。
- **InputStreamResource**：访问输入流资源的实现类。
- **ByteArrayResource**：访问字节数组资源的实现类。

这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。

## 9. 模版方法模式

**经典模板方法定义：**

父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现。

最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。

**所以父类模板方法中有两类方法：**

**共同的方法：**所有子类都会用到的代码

**不同的方法：**子类要覆盖的方法，分为两种：

- 抽象方法：父类中的是抽象方法，子类必须覆盖
- 钩子方法：父类中是一个空方法，子类继承了默认也是空的

注：为什么叫钩子，子类可以通过这个钩子（方法），控制父类，因为这个钩子实际是父类的方法（空方法）！

**Spring模板方法模式实质：**

是模板方法模式和回调模式的结合，是Template Method不需要继承的另一种实现方式。Spring几乎所有的外接扩展都采用这种模式。

### 具体实现：

JDBC的抽象和对Hibernate的集成，都采用了一种理念或者处理方式，那就是模板方法模式与相应的Callback接口相结合。

采用模板方法模式是为了以一种统一而集中的方式来处理资源的获取和释放，以JdbcTemplate为例：

```
public abstract class JdbcTemplate {
    public final Object execute (String sql) {
        Connection con=null;
        Statement stmt=null;
        try{
            con=getConnection () ;
            stmt=con.createStatement () ;
            Object retValue=executeWithStatement (stmt,sql) ;
            return retValue;
        }catch (SQLException e) {
            ...
        }finally{
            closeStatement (stmt) ;
            releaseConnection (con) ;
        }
    }
    protected abstract Object executeWithStatement (Statement stmt, String sql) ;
}
```

### 引入回调原因：

JdbcTemplate是抽象类，不能够独立使用，我们每次进行数据访问的时候都要给出一个相应的子类实现,这样肯定不方便，所以就引入了回调。

回调代码

```
public interface StatementCallback{
    Object doWithStatement (Statement stmt) ;
}
```

利用回调方法重写JdbcTemplate方法

```

public class JdbcTemplate {
    public final Object execute (StatementCallback callback) {
        Connection con=null;
        Statement stmt=null;
        try{
            con=getConnection () ;
            stmt=con.createStatement () ;
            Object retValue=callback.doWithStatement (stmt) ;
            return retValue;
        }catch (SQLException e) {
            ...
        }finally{
            closeStatement (stmt) ;
            releaseConnection (con) ;
        }
    }

    ...//其它方法定义
}

```

Jdbc使用方法如下：

```

JdbcTemplate jdbcTemplate=...;
final String sql=...;
StatementCallback callback=new StatementCallback(){
    public Object=doWithStatement(Statement stmt){
        return ...;
    }
}
jdbcTemplate.execute(callback);

```

## 为什么JdbcTemplate没有使用继承？

因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？

我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。但是变化的东西是一段代码，而且这段代码会用到JdbcTemplate中的变量。怎么办？

那我们就用回调对象吧。在这个回调对象中定义一个操纵JdbcTemplate中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到JdbcTemplate，从而完成了调用。

## 参考

<https://www.cnblogs.com/digdeep/p/4518571.html>  
<https://www.cnblogs.com/tongkey/p/7919401.html>  
<https://www.cnblogs.com/fingerboy/p/6393644.html>  
[https://blog.csdn.net/ovoo\\_8/article/details/51189401](https://blog.csdn.net/ovoo_8/article/details/51189401)  
<https://blog.csdn.net/z69183787/article/details/65628166>  
 《spring源码深度分析》

作者 | iCoding91

链接 | [blog.csdn.net/caoxiaohong1005](https://blog.csdn.net/caoxiaohong1005)

1. 如何降低程序员的工资？
2. 编写 Spring MVC 的 14 个小技巧
3. 技术大佬的呕心力作！
4. 详述 Spring Data JPA 的那些事儿



微信搜一搜

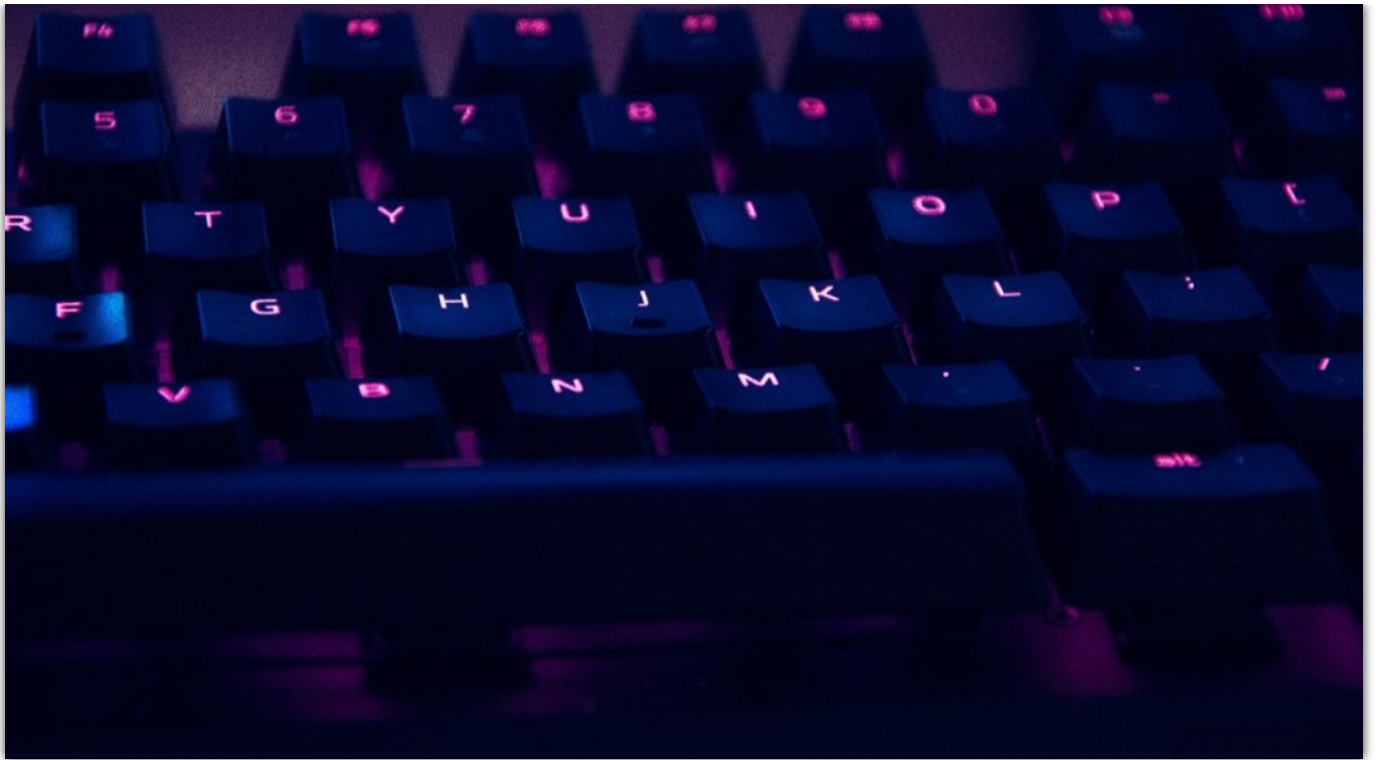
Q Java后端

[阅读原文](#)

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 腊月二十八，聊聊 MyBatis 中的设计模式

crazyant Java后端 1月22日



作者 | crazyant

链接 | [www.crazyant.net/2022.html](http://www.crazyant.net/2022.html)

虽然我们都知道有26个设计模式，但是大多停留在概念层面，真实开发中很少遇到，Mybatis源码中使用了大量的设计模式，阅读源码并观察设计模式在其中的应用，能够更深入的理解设计模式。

Mybatis至少遇到了以下的设计模式的使用：

1. Builder模式，例如SqlSessionFactoryBuilder、XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder、CacheBuilder；
2. 工厂模式，例如SqlSessionFactory、ObjectFactory、MapperProxyFactory；
3. 单例模式，例如ErrorContext和LogFactory；
4. 代理模式，Mybatis实现的核心，比如MapperProxy、ConnectionLogger，用的jdk的动态代理；还有executor.loader包使用了cglib或者javassist达到延迟加载的效果；
5. 组合模式，例如SqlNode和各个子类ChooseSqlNode等；
6. 模板方法模式，例如BaseExecutor和SimpleExecutor，还有BaseTypeHandler和所有的子类例如IntegerTypeHandler；
7. 适配器模式，例如Log的Mybatis接口和它对jdbc、log4j等各种日志框架的适配实现；
8. 装饰者模式，例如Cache包中的cache.decorators子包中等各个装饰者的实现；
9. 迭代器模式，例如迭代器模式PropertyTokenizer；

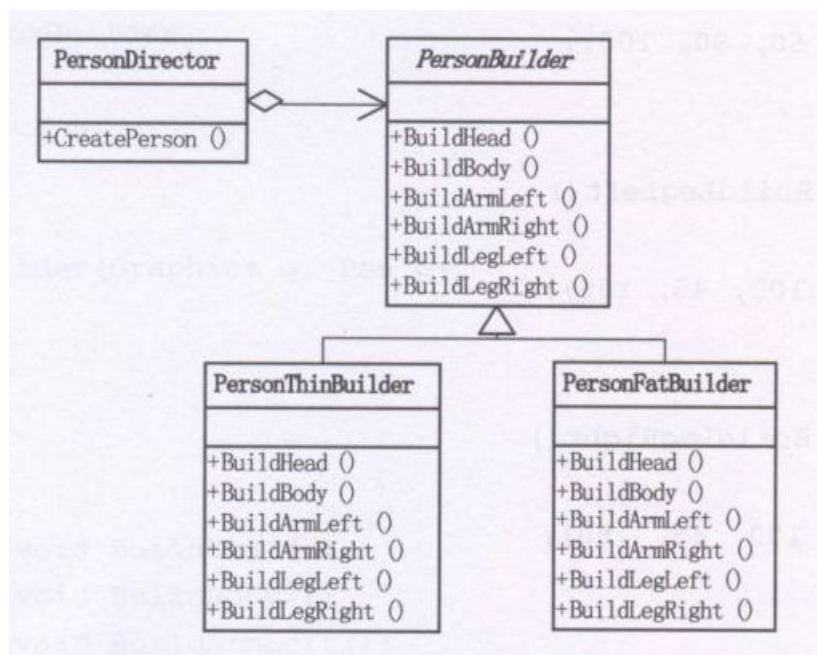
接下来挨个模式进行解读，先介绍模式自身的知识，然后解读在Mybatis中怎样应用了该模式。

## 1、Builder模式

Builder模式的定义是“将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。”，它属于创建类模



式，一般来说，如果一个对象的构建比较复杂，超出了构造函数所能包含的范围，就可以使用工厂模式和Builder模式，相对于工厂模式会产出一个完整的产品，Builder应用于更加复杂的对象的构建，甚至只会构建产品的一个部分。



在Mybatis环境的初始化过程中，`SqlSessionFactoryBuilder`会调用`XMLConfigBuilder`读取所有的`MybatisMapConfig.xml`和所有的`*Mapper.xml`文件，构建Mybatis运行的核心对象`Configuration`对象，然后将该`Configuration`对象作为参数构建一个`SqlSessionFactory`对象。

其中`XMLConfigBuilder`在构建`Configuration`对象时，也会调用`XMLMapperBuilder`用于读取`*Mapper`文件，而`XMLMapperBuilder`会使用`XMLStatementBuilder`来读取和build所有的SQL语句。

在这个过程中，有一个相似的特点，就是这些Builder会读取文件或者配置，然后做大量的XpathParser解析、配置或语法的解析、反射生成对象、存入结果缓存等步骤，这么多的工作都不是一个构造函数所能包括的，因此大量采用了Builder模式来解决。

对于builder的具体类，方法都大都用`build*`开头，比如`SqlSessionFactoryBuilder`为例，它包含以下方法：

```
org.apache.ibatis.session
▼ SqlSessionFactoryBuilder
  ● build(Configuration) : SqlSessionFactory
  ● build(InputStream) : SqlSessionFactory
  ● build(InputStream, Properties) : SqlSessionFactory
  ● build(InputStream, String) : SqlSessionFactory
  ● build(InputStream, String, Properties) : SqlSessionFactory
  ● build(Reader) : SqlSessionFactory
  ● build(Reader, Properties) : SqlSessionFactory
  ● build(Reader, String) : SqlSessionFactory
  ● build(Reader, String, Properties) : SqlSessionFactory
```

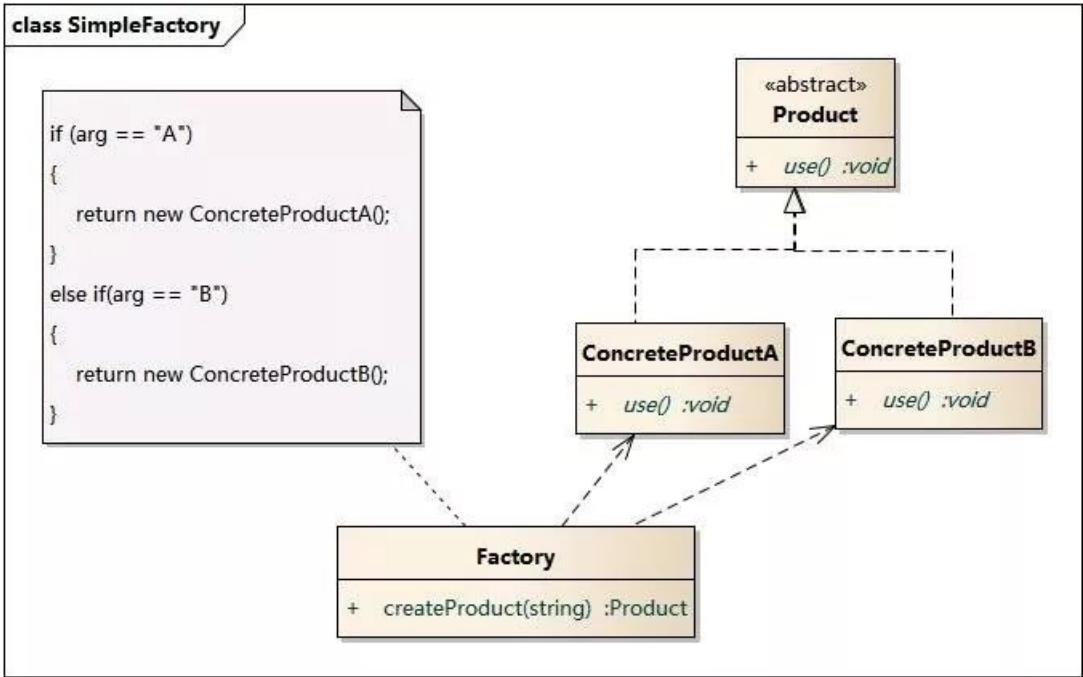
即根据不同的输入参数来构建`SqlSessionFactory`这个工厂对象。

## 2、工厂模式



在Mybatis中比如SqlSessionFactory使用的是工厂模式，该工厂没有那么复杂的逻辑，是一个简单工厂模式。

简单工厂模式(Simple Factory Pattern):又称为静态工厂方法(Static Factory Method)模式,它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。



SqlSession可以认为是一个Mybatis工作的核心的接口，通过这个接口可以执行SQL语句、获取Mappers、管理事务。类似于连接MySQL的Connection对象。

```
org.apache.ibatis.session  
▼ SqlSessionFactory  
  A getConfiguration() : Configuration  
  A openSession() : SqlSession  
  A openSession(boolean) : SqlSession  
  A openSession(Connection) : SqlSession  
  A openSession(ExecutorType) : SqlSession  
  A openSession(ExecutorType, boolean) : SqlSession  
  A openSession(ExecutorType, Connection) : SqlSession  
  A openSession(ExecutorType, TransactionIsolationLevel) : SqlSession  
  A openSession(TransactionIsolationLevel) : SqlSession
```

可以看到，该Factory的openSession方法重载了很多个，分别支持autoCommit、Executor、Transaction等参数的输入，来构建核心的SqlSession对象。

在DefaultSqlSessionFactory的默认工厂实现里，有一个方法可以看出工厂怎么产出一个产品：

```

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level,
    boolean autoCommit) {
    Transaction tx = null;
    try {
        final Environment environment = configuration.getEnvironment();
        final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
        tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
        final Executor executor = configuration.newExecutor(tx, execType);
        return new DefaultSqlSession(configuration, executor, autoCommit);
    } catch (Exception e) {
        closeTransaction(tx); // may have fetched a connection so lets call
            // close()
        throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}

```

这是一个openSession调用的底层方法，该方法先从configuration读取对应的环境配置，然后初始化TransactionFactory获得一个Transaction对象，然后通过Transaction获取一个Executor对象，最后通过configuration、Executor、是否autoCommit三个参数构建了SqlSession。

在这里其实也可以看到端倪，SqlSession的执行，其实是委托给对应的Executor来进行的。

而对于LogFactory，它的实现代码：

```

public final class LogFactory {
    private static Constructor<? extends Log> logConstructor;

    private LogFactory() {
        // disable construction
    }

    public static Log getLog(Class<?> aClass) {
        return getLog(aClass.getName());
    }
}

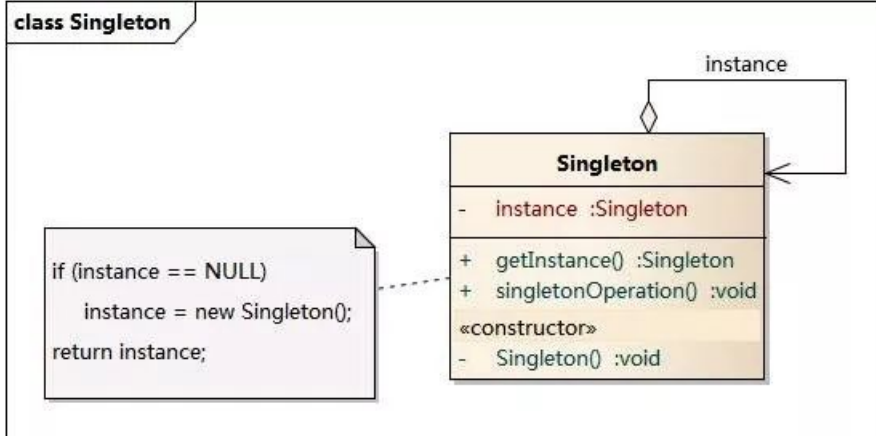
```

这里有个特别的地方，是Log变量的类型是Constructor<?extendsLog>，也就是说该工厂生产的不只是一个产品，而是具有Log公共接口的一系列产品，比如Log4jImpl、Slf4jImpl等很多具体的Log。

### 3、单例模式

单例模式(Singleton Pattern):单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。



在Mybatis中有两个地方用到单例模式，ErrorContext和LogFactory，其中ErrorContext是用在每个线程范围内的单例，用于记录该线程的执行环境错误信息，而LogFactory则是提供给整个Mybatis使用的日志工厂，用于获得针对项目配置好的日志对象。

ErrorContext的单例实现代码：

```
public class ErrorContext {  
  
    private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext>();  
  
    private ErrorContext() {  
    }  
  
    public static ErrorContext instance() {  
        ErrorContext context = LOCAL.get();  
        if (context == null) {  
            context = new ErrorContext();  
            LOCAL.set(context);  
        }  
        return context;  
    }  
}
```

构造函数是private修饰，具有一个static的局部instance变量和一个获取instance变量的方法，在获取实例的方法中，先判断是否为空如果是的话就先创建，然后返回构造好的对象。

只是这里有个有趣的地方是，LOCAL的静态实例变量使用了ThreadLocal修饰，也就是说它属于每个线程各自的数据，而在instance()方法中，先获取本线程的该实例，如果没有就创建该线程独有的ErrorContext。

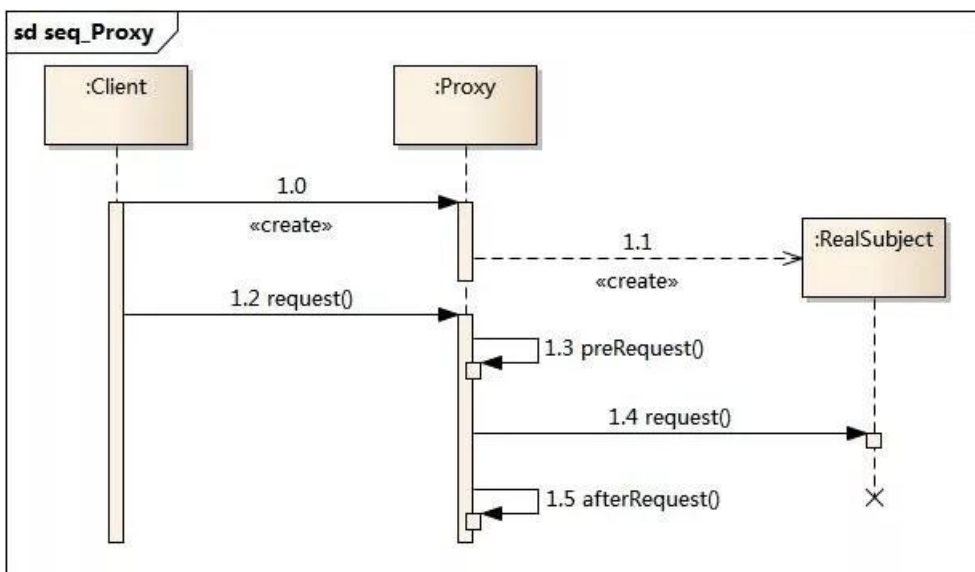
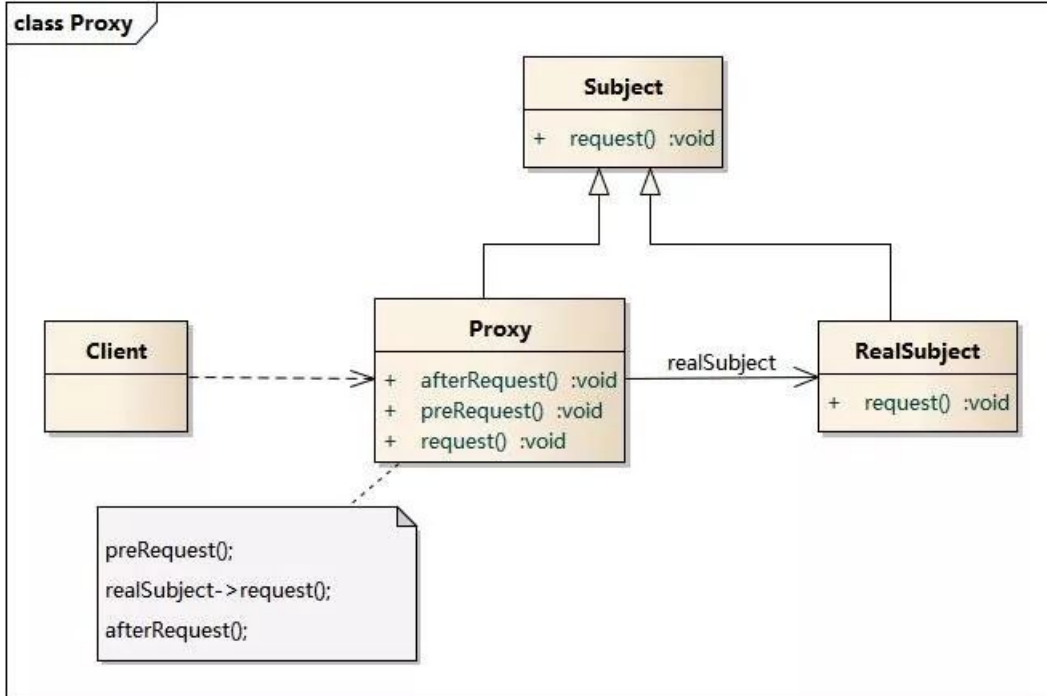
#### 4、代理模式

代理模式可以认为是Mybatis的核心使用的模式，正是由于这个模式，我们只需要编写Mapper.java接口，不需要实现，由Mybatis后台帮我们完成具体SQL的执行。

代理模式(Proxy Pattern)：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。代理模式的英文叫做Proxy或Surrogate，它是一种对象结构型模式。

代理模式包含如下角色：

- Subject: 抽象主题角色
- Proxy: 代理主题角色
- RealSubject: 真实主题角色



这里有两个步骤，第一个是提前创建一个Proxy，第二个是使用的时候会自动请求Proxy，然后由Proxy来执行具体事务；

当我们使用Configuration的getMapper方法时，会调用mapperRegistry.getMapper方法，而该方法又会调用mapperProxyFactory.newInstance(sqlSession)来生成一个具体的代理：

```

/**
 * @author Lasse Voss
 */
public class MapperProxyFactory<T> {

    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache = new ConcurrentHashMap<Method, MapperMethod>();

    public MapperProxyFactory(Class<T> mapperInterface) {
        this.mapperInterface = mapperInterface;
    }

    public Class<T> getMapperInterface() {
        return mapperInterface;
    }

    public Map<Method, MapperMethod> getMethodCache() {
        return methodCache;
    }

    @SuppressWarnings("unchecked")
    protected T newInstance(MapperProxy<T> mapperProxy) {
        return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new Class[] { mapperInterface },
            mapperProxy);
    }

    public T newInstance(SqlSession sqlSession) {
        final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCache);
        return newInstance(mapperProxy);
    }
}

```

在这里, 先通过T.newInstance(SqlSession sqlSession)方法会得到一个MapperProxy对象, 然后调用T.newInstance(MapperProxy<T> mapperProxy)生成代理对象然后返回。

而查看MapperProxy的代码, 可以看到如下内容:

```

public class MapperProxy<T> implements InvocationHandler, Serializable {

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            if (Object.class.equals(method.getDeclaringClass())) {
                return method.invoke(this, args);
            } else if (isDefaultMethod(method)) {
                return invokeDefaultMethod(proxy, method, args);
            }
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
        final MapperMethod mapperMethod = cachedMapperMethod(method);
        return mapperMethod.execute(sqlSession, args);
    }
}

```

非常典型的, 该MapperProxy类实现了InvocationHandler接口, 并且实现了该接口的invoke方法。

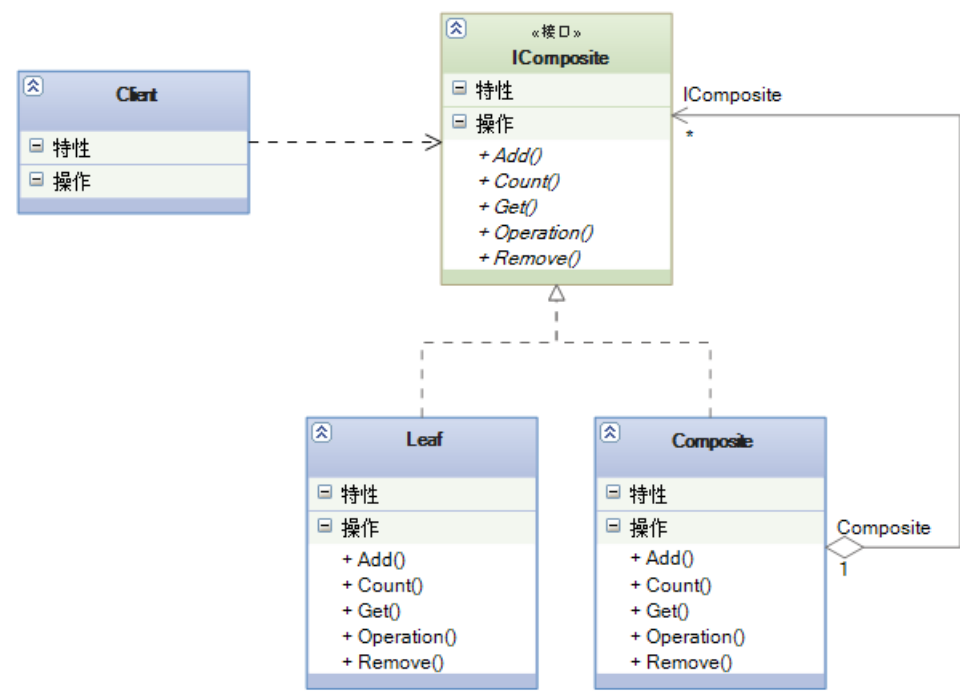
通过这种方式, 我们只需要编写Mapper.java接口类, 当真正执行一个Mapper接口的时候, 就会转发给MapperProxy.invoke方法, 而该方法则会调用后续的sqlSession.cud>executor.execute>prepareStatement等一系列方法, 完成SQL的执行和返回。

5、组合模式

组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。

组合模式对单个对象(叶子对象)和组合对象(组合对象)具有一致性，它将对象组织到树结构中，可以用来描述整体与部分的关系。同时它也模糊了简单元素(叶子对象)和复杂元素(容器对象)的概念，使得客户能够像处理简单元素一样来处理复杂元素，从而使客户程序能够与复杂元素的内部结构解耦。

在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因。



Mybatis支持动态SQL的强大功能，比如下面的这个SQL：

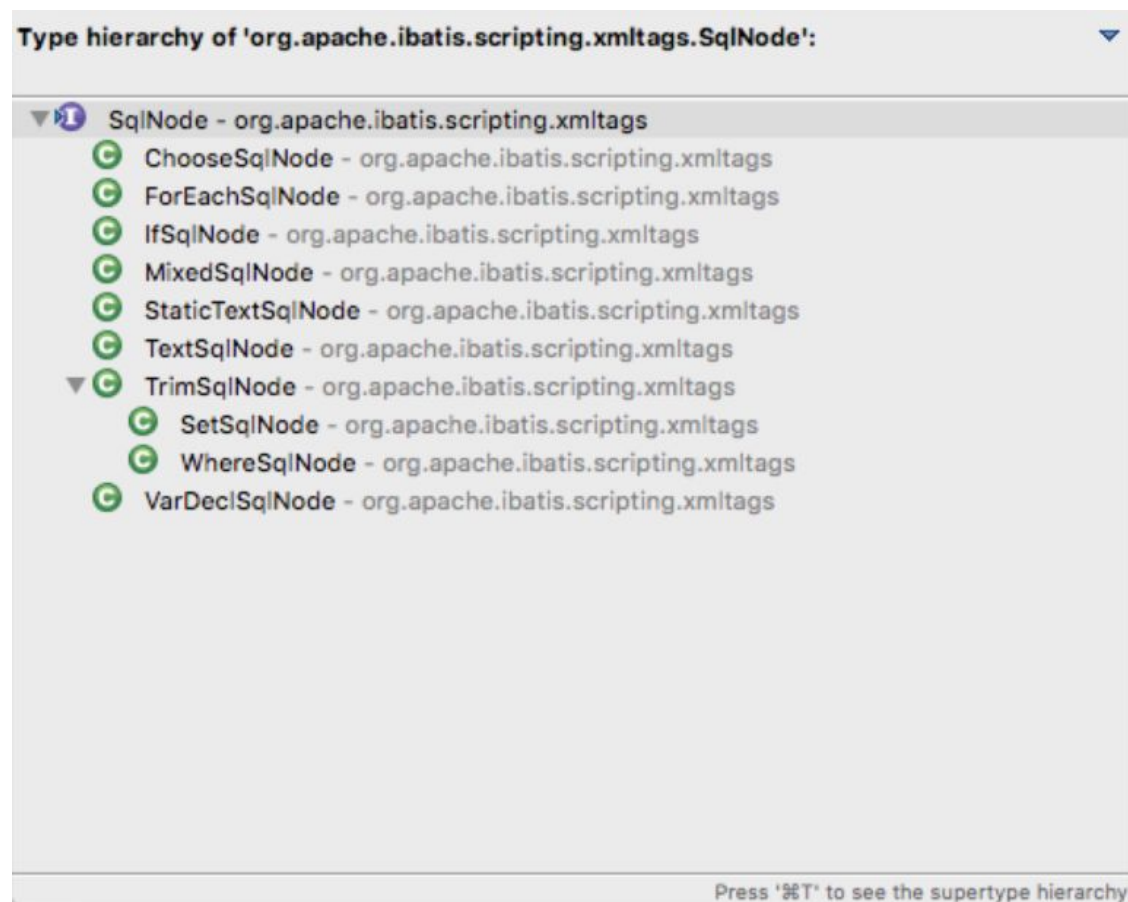
```
<update id="update" parameterType="org.format.dynamicproxy.mybatis.bean.User">
  UPDATE users
  <trim prefix="SET" prefixOverrides=", ">
    <if test="name != null and name != ''">
      name = #{name}
    </if>
    <if test="age != null and age != ''">
      , age = #{age}
    </if>
    <if test="birthday != null and birthday != ''">
      , birthday = #{birthday}
    </if>
  </trim>
  where id = ${id}
</update>
```

在这里面使用到了trim、if等动态元素，可以根据条件来生成不同情况下的SQL；

在DynamicSqlSource.getBoundSql方法里，调用了rootSqlNode.apply(context)方法，apply方法是所有的动态节点都实现的接口：

```
public interface SqlNode {
    boolean apply(DynamicContext context);
}
```

对于实现该SqlSource接口的所有节点，就是整个组合模式树的各个节点：



组合模式的简单之处在于，所有的子节点都是同一类节点，可以递归的向下执行，比如对于TextSqlNode，因为它是最底层的叶子节点，所以直接将对应的内容append到SQL语句中：

```
@Override
public boolean apply(DynamicContext context) {
    GenericTokenParser parser = createParser(new BindingTokenParser(context, injectionFilter));
    context.appendSql(parser.parse(text));
    return true;
}
```

但是对于IfSqlNode，就需要先做判断，如果判断通过，仍然会调用子元素的SqlNode，即contents.apply方法，实现递归的解析。

```
@Override
public boolean apply(DynamicContext context) {
    if (evaluator.evaluateBoolean(test, context.getBindings())) {
        contents.apply(context);
        return true;
    }
    return false;
}
```

## 6、模板方法模式

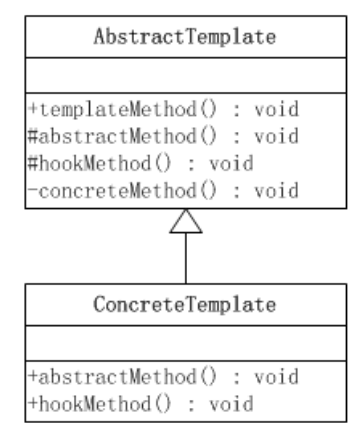
模板方法模式是所有模式中最常见的几个模式之一，是基于继承的代码复用的基本技术。

模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负

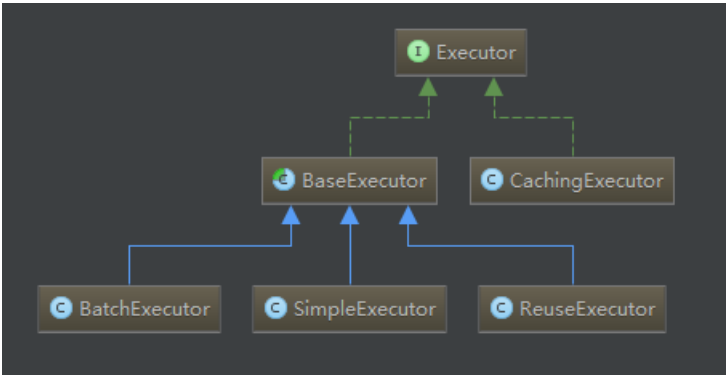


责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method);而将这些基本方法汇总起来的方法叫做模板方法(template method), 这个设计模式的名字就是从此而来。

模板类定义一个操作中的算法的骨架, 而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。



在Mybatis中, sqlSession的SQL执行, 都是委托给Executor实现的, Executor包含以下结构:



其中的BaseExecutor就采用了模板方法模式, 它实现了大部分的SQL执行逻辑, 然后把以下几个方法交给子类定制化完成:

```
protected abstract int doUpdate(MappedStatement ms, Object parameter) throws SQLException;

protected abstract List<BatchResult> doFlushStatements(boolean isRollback) throws SQLException;

protected abstract <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,
    ResultHandler resultHandler, BoundSql boundSql) throws SQLException;
```

该模板方法类有几个子类的具体实现, 使用了不同的策略:

- 简单SimpleExecutor: 每执行一次update或select, 就开启一个Statement对象, 用完立刻关闭Statement对象。(可以是Statement或PreparedStatement对象)
- 重用ReuseExecutor: 执行update或select, 以sql作为key查找Statement对象, 存在就使用, 不存在就创建, 用完后, 不关闭Statement对象, 而是放置于Map<String, Statement>内, 供下一次使用。(可以是Statement或PreparedStatement对象)
- 批量BatchExecutor: 执行update (没有select, JDBC批处理不支持select), 将所有sql都添加到批处理中 (addBatch()), 等待统一执行 (executeBatch()), 它缓存了多个Statement对象, 每个Statement对象都是



addBatch()完毕后，等待逐一执行executeBatch()批处理的；BatchExecutor相当于维护了多个桶，每个桶里都装了很多属于自己的SQL，就像苹果蓝里装了很多苹果，番茄蓝里装了很多番茄，最后，再统一倒进仓库。（可以是Statement或PreparedStatement对象）

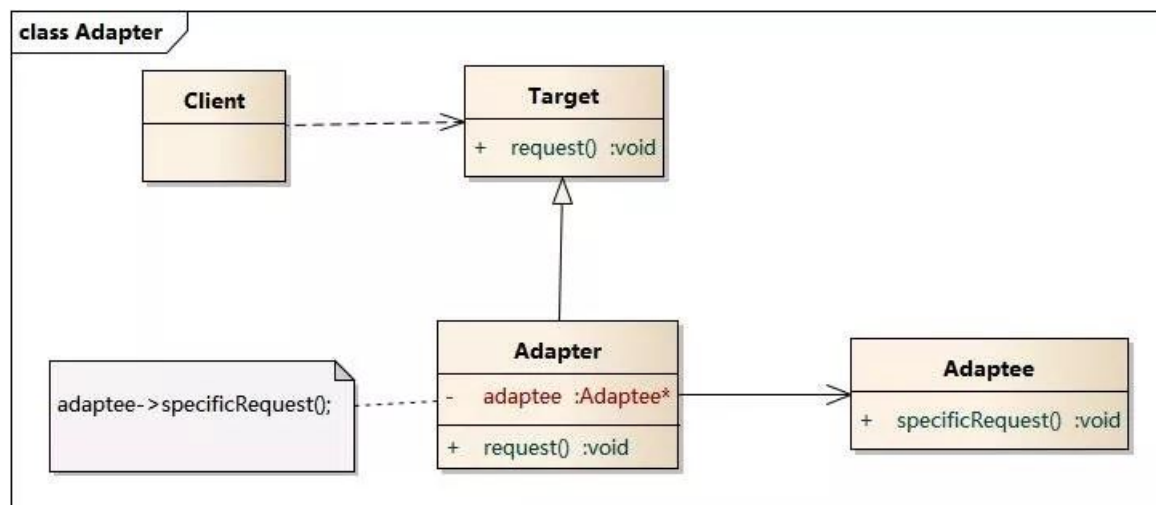
比如在SimpleExecutor中这样实现update方法：

@Override

```
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(this, ms, parameter, RowBounds.DEFAULT, null,
            null);
        stmt = prepareStatement(handler, ms.getStatementLog());
        return handler.update(stmt);
    } finally {
        closeStatement(stmt);
    }
}
```

## 7、适配器模式

适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。



在Mybatis的logging包中，有一个Log接口：

```

/**
 * @author Clinton Begin
 */
public interface Log {

    boolean isDebugEnabled();

    boolean isTraceEnabled();

    void error(String s, Throwable e);

    void error(String s);

    void debug(String s);

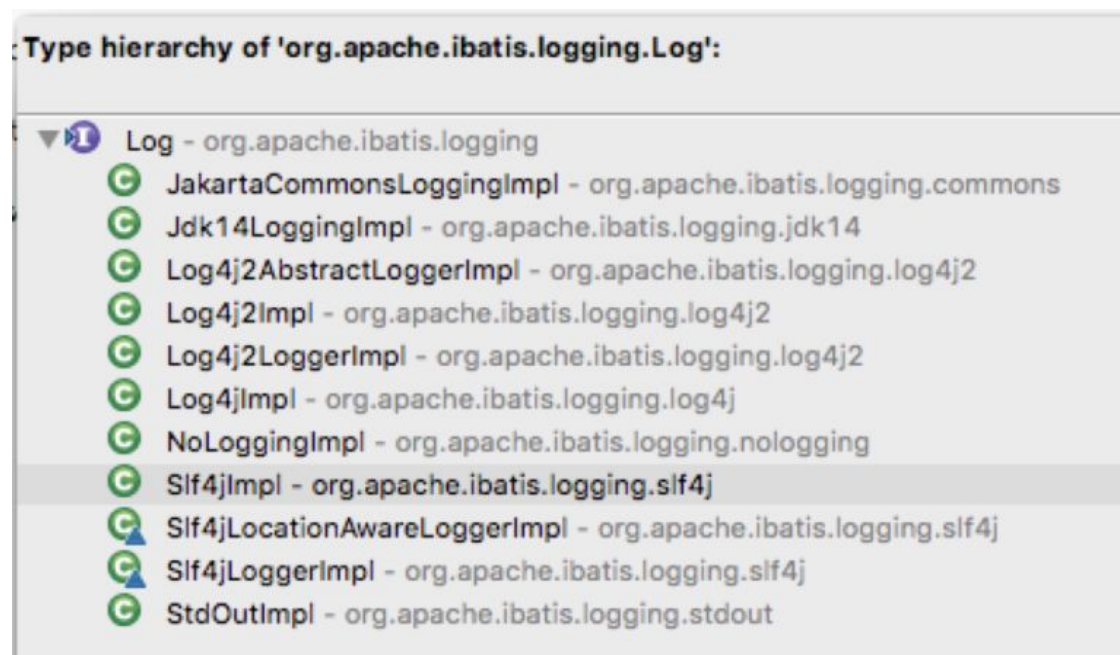
    void trace(String s);

    void warn(String s);

}

```

该接口定义了Mybatis直接使用的日志方法，而Log接口具体由谁来实现呢？Mybatis提供了多种日志框架的实现，这些实现都匹配这个Log接口所定义的接口方法，最终实现了所有外部日志框架到Mybatis日志包的适配：



比如对于Log4jImpl的实现来说，该实现持有了org.apache.log4j.Logger的实例，然后所有的日志方法，均委托该实例来实现。

```

public class Log4jImpl implements Log {

    private static final String FQCN = Log4jImpl.class.getName();

    private Logger log;

    public Log4jImpl(String clazz) {
        log = Logger.getLogger(clazz);
    }

    @Override
    public boolean isDebugEnabled() {
        return log.isDebugEnabled();
    }

    @Override
    public boolean isTraceEnabled() {
        return log.isTraceEnabled();
    }

    @Override
    public void error(String s, Throwable e) {
        log.log(FQCN, Level.ERROR, s, e);
    }

    @Override
    public void error(String s) {
        log.log(FQCN, Level.ERROR, s, null);
    }

    @Override
    public void debug(String s) {
        log.log(FQCN, Level.DEBUG, s, null);
    }

    @Override
    public void trace(String s) {
        log.log(FQCN, Level.TRACE, s, null);
    }

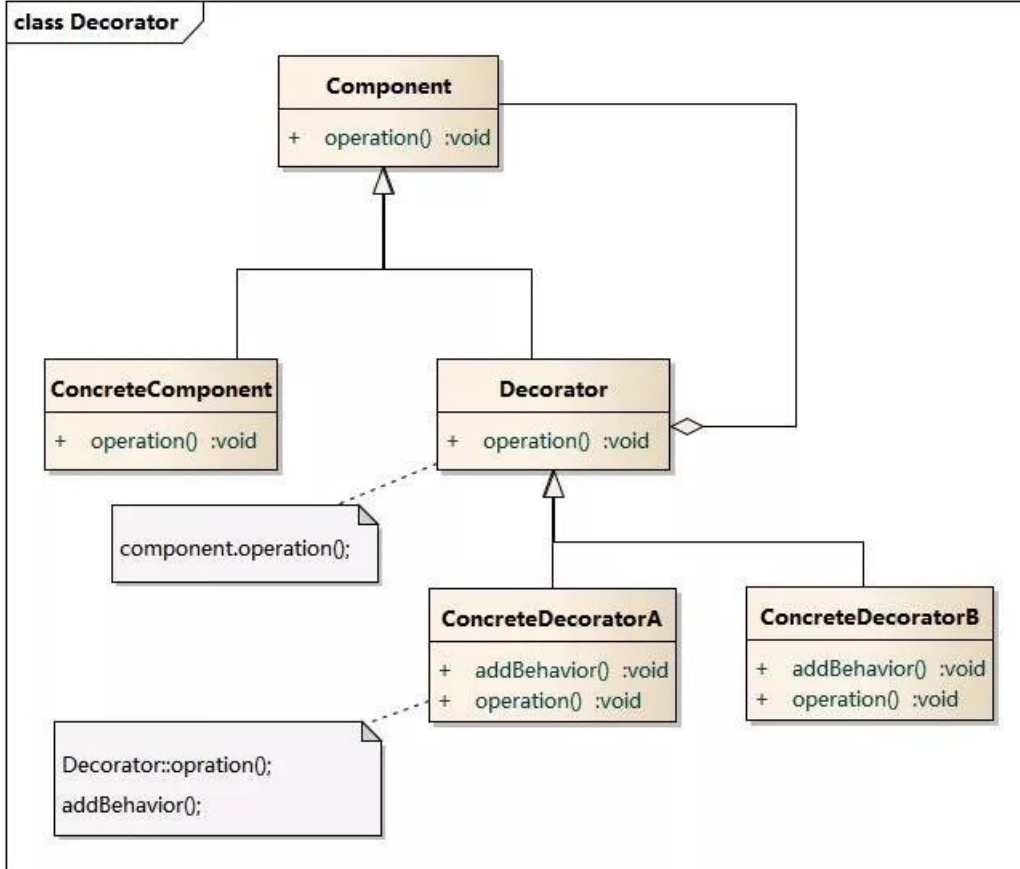
    @Override
    public void warn(String s) {
        log.log(FQCN, Level.WARN, s, null);
    }

}

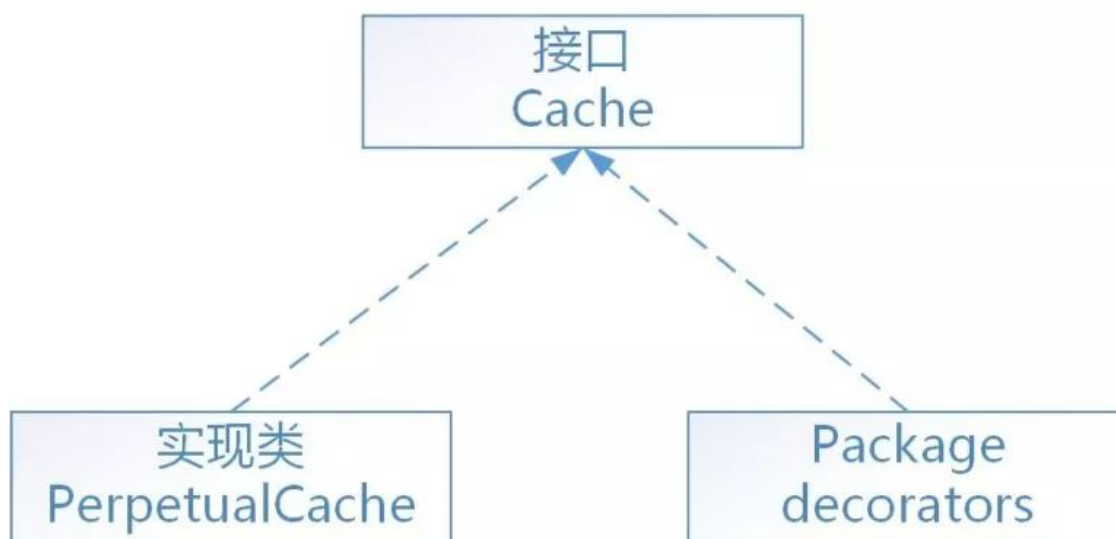
```

## 8、装饰者模式

装饰模式(Decorator Pattern) :动态地给一个对象增加一些额外的职责(Responsibility),就增加对象功能来说,装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper),与适配器模式的别名相同,但它们适用于不同的场合。根据翻译的不同,装饰模式也有人称之为“油漆工模式”,它是一种对象结构型模式。



在mybatis中，缓存的功能由根接口Cache（org.apache.ibatis.cache.Cache）定义。整个体系采用装饰器设计模式，数据存储和缓存的基本功能由PerpetualCache（org.apache.ibatis.cache.impl.PerpetualCache）永久缓存实现，然后通过一系列的装饰器来对PerpetualCache永久缓存进行缓存策略等方便的控制。如下图：



用于装饰PerpetualCache的标准装饰器共有8个（全部在org.apache.ibatis.cache.decorators包中）：

1. FifoCache：先进先出算法，缓存回收策略
2. LoggingCache：输出缓存命中的日志信息
3. LruCache：最近最少使用算法，缓存回收策略
4. ScheduledCache：调度缓存，负责定时清空缓存
5. SerializedCache：缓存序列化和反序列化存储
6. SoftCache：基于软引用实现的缓存管理策略
7. SynchronizedCache：同步的缓存装饰器，用于防止多线程并发访问
8. WeakCache：基于弱引用实现的缓存管理策略

另外，还有一个特殊的装饰器TransactionalCache：事务性的缓存

正如大多数持久层框架一样，mybatis缓存同样分为一级缓存和二级缓存

- 一级缓存，又叫本地缓存，是PerpetualCache类型的永久缓存，保存在执行器中（BaseExecutor），而执行器又在SqlSession（DefaultSqlSession）中，所以一级缓存的生命周期与SqlSession是相同的。
- 二级缓存，又叫自定义缓存，实现了Cache接口的类都可以作为二级缓存，所以可配置如encache等的第三方缓存。二级缓存以namespace名称空间为其唯一标识，被保存在Configuration核心配置对象中。

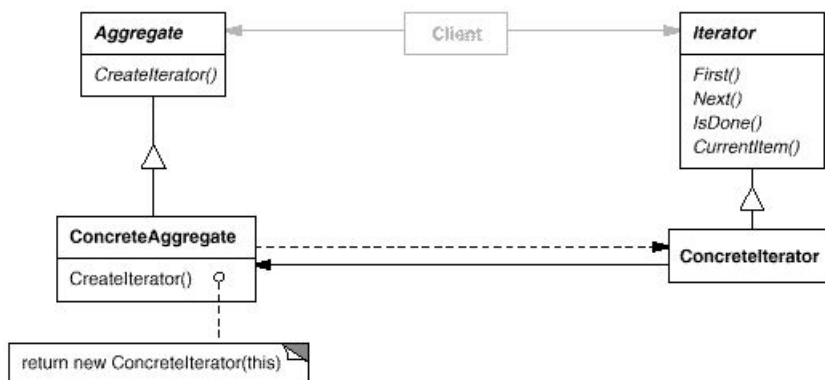
二级缓存对象的默认类型为PerpetualCache，如果配置的缓存是默认类型，则mybatis会根据配置自动追加一系列装饰器。

Cache对象之间的引用顺序为：

SynchronizedCache→LoggingCache→SerializedCache→ScheduledCache→LruCache→PerpetualCache

## 9、迭代器模式

迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF给出的定义为：提供一种方法访问一个容器（container）对象中各个元素，而又不需暴露该对象的内部细节。



Java的Iterator就是迭代器模式的接口，只要实现了该接口，就相当于应用了迭代器模式：

```
▼ I Iterator<E>
  ● D forEachRemaining(Consumer<? super E>) : void
  ● A hasNext() : boolean
  ● A next() : E
  ● D remove() : void
```

比如Mybatis的PropertyTokenizer是property包中的重量级类，该类会被reflection包中其他的类频繁的引用到。这个类实现了Iterator接口，在使用时经常被用到的是Iterator接口中的hasNext这个函数。

```

public class PropertyTokenizer implements Iterator<PropertyTokenizer> {
    private String name;
    private String indexedName;
    private String index;
    private String children;

    public PropertyTokenizer(String fullname) {
        int delim = fullname.indexOf('.');
        if (delim > -1) {
            name = fullname.substring(0, delim);
            children = fullname.substring(delim + 1);
        } else {
            name = fullname;
            children = null;
        }
        indexedName = name;
        delim = name.indexOf('[');
        if (delim > -1) {
            index = name.substring(delim + 1, name.length() - 1);
            name = name.substring(0, delim);
        }
    }

    public String getName() {
        return name;
    }

    public String getIndex() {
        return index;
    }

    public String getIndexedName() {
        return indexedName;
    }

    public String getChildren() {
        return children;
    }

    @Override
    public boolean hasNext() {
        return children != null;
    }

    @Override
    public PropertyTokenizer next() {
        return new PropertyTokenizer(children);
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException(
            "Remove is not supported, as it has no meaning in the context of properties.");
    }
}

```

可以看到，这个类传入一个字符串到构造函数，然后提供了iterator方法对解析后的子串进行遍历，是一个很常用的方法类。

参考资料：

- 图说设计模式：[http://design-patterns.readthedocs.io/zh\\_CN/latest/index.html](http://design-patterns.readthedocs.io/zh_CN/latest/index.html)

- 深入浅出Mybatis系列（十）—SQL执行流程分析（源码篇）：<http://www.cnblogs.com/dongying/p/4142476.html>
- 设计模式读书笔记——组合模式<http://www.cnblogs.com/chenssy/p/3299719.html>
- Mybatis3.3.x技术内幕（四）：五鼠闹东京之执行器Executor设计原本<http://blog.csdn.net/wagcy/article/details/32963235>
- mybatis缓存机制详解（一）——Cache <https://my.oschina.net/lixin91/blog/620068>

本文地址：<http://crazyant.net/2022.html>，转载请注明来源。



微信扫描二维码, 关注我的公众号

声明：pdf仅供学习使用，一切版权归原创公众号所有；建议持续关注原创公众号获取最新文章，学习愉快！

# 超详细：常用的设计模式汇总

Java后端 2019-11-01

点击上方 [Java后端](#), 选择 [设为星标](#)

优质文章，及时送达

## 单例模式

简单点说，就是一个应用程序中，某个类的实例对象只有一个，你没有办法去new，因为构造器是被private修饰的，一般通过getInstance()的方法来获取它们的实例。getInstance()的返回值是一个对象的引用，并不是一个新的实例，所以不要错误的理解成多个对象。单例模式实现起来也很容易，直接看demo吧

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

按照我的习惯，我恨不得写满注释，怕你们看不懂，但是这个代码实在太简单了，所以我没写任何注释，如果这几行代码你都看不明白的话，那你可以洗洗睡了，等你睡醒了再来看我的博客说不定能看懂。

上面的是最基本的写法，也叫懒汉写法（线程不安全）下面我再公布几种单例模式的写法：

### 懒汉式写法（线程安全）

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

### 饿汉式写法

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    private Singleton(){}  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```



## 静态内部类

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

## 枚举

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {
    }
}
```

这种方式是Effective Java作者Josh Bloch 提倡的方式,它不仅能避免多线程同步问题,而且还能防止反序列化重新创建新的对象,可谓是很坚强的壁垒啊,不过,个人认为由于1.5中才加入enum特性,用这种方式写不免让人感觉生疏。

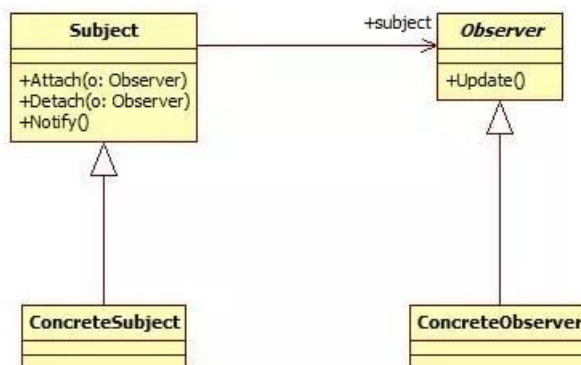
## 双重校验锁

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

总结：我个人比较喜欢静态内部类写法和饿汉式写法，其实这两种写法能够应付绝大多数情况了。其他写法也可以选择，主要还是看业务需求吧。

## 观察者模式

对象间一对多的依赖关系，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。



看不懂图的人端着小板凳到这里来，给你举个栗子：假设有三个人，小美（女，28），老王和老李。小美很漂亮，很风骚，老王和老李是两个中年男屌丝，时刻关注着小美的一举一动。有一天，小美说了一句：我老公今天不在家，一个人好无聊啊～～～，这句话被老王和老李听到了，结果乐坏了，蹭蹭蹭，没一会儿，老王就冲到小美家门口了，于是进门了……………帕～咻咻咻咻咻～ 在这里，小美是被观察者，老王和老李是观察者，被观察者发出一条信息，然后观察者们进行相应的处理，看代码：

```
public interface Person {  
  
    void getMessage(String s);  
}
```

这个接口相当于老王和老李的电话号码，小美发送通知的时候就会拨打getMessage这个电话，拨打电话就是调用接口，看不懂没关系，先往下看

```
public class LaoWang implements Person {  
  
    private String name = "老王";  
  
    public LaoWang() {  
    }  
  
    @Override  
    public void getMessage(String s) {  
        System.out.println(name + "接到了小美打过来的电话，电话内容是：" + s);  
    }  
}  
  
public class LaoLi implements Person {  
  
    private String name = "老李";  
  
    public LaoLi() {  
    }  
  
    @Override  
    public void getMessage(String s) {  
        System.out.println(name + "接到了小美打过来的电话，电话内容是：->" + s);  
    }  
}
```

代码很简单，我们再看看小美的代码：

```
public class XiaoMei {
    List<Person> list = new ArrayList<Person>();
    public XiaoMei(){
    }

    public void addPerson(Person person){
        list.add(person);
    }

    public void notifyPerson() {
        for(Person person:list){
            person.getMessage("今天家里就我一个人，你们过来吧，谁先过来谁就能得到我!");
        }
    }
}
```

我们写一个测试类来看一下结果对不对

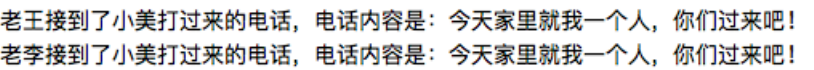
```
public class Test {
    public static void main(String[] args) {

        XiaoMei xiao_mei = new XiaoMei();
        LaoWang lao_wang = new LaoWang();
        LaoLi lao_li = new LaoLi();

        xiao_mei.addPerson(lao_wang);
        xiao_mei.addPerson(lao_li);

        xiao_mei.notifyPerson();
    }
}
```

运行结果我截图了



老王接到了小美打过来的电话，电话内容是：今天家里就我一个人，你们过来吧！  
老李接到了小美打过来的电话，电话内容是：今天家里就我一个人，你们过来吧！

运行结果

### 装饰者模式

对已有的业务逻辑进一步的封装，使其增加额外的功能，如Java中的IO流就使用了装饰者模式，用户在使用的时候，可以任意组装，达到自己想要的效果。举个栗子，我想吃三明治，首先我需要一根大大的香肠，我喜欢吃奶油，在香肠上面加一点奶油，再放一点蔬菜，最后再用两片面包夹一下，很丰盛的一顿午饭，营养又健康。（ps：不知道上海哪里有卖好吃的三明治的，求推荐～）那我们应该怎么来写代码呢？首先，我们需要写一个Food类，让其他所有食物都来继承这个类，看代码：

```
public class Food {

    private String food_name;

    public Food() {
    }

    public Food(String food_name) {
        this.food_name = food_name;
    }

    public String make() {
        return food_name;
    };
}
```

代码很简单，我就不解释了，然后我们写几个子类继承它：

```
public class Bread extends Food {

    private Food basic_food;

    public Bread(Food basic_food) {
        this.basic_food = basic_food;
    }

    public String make() {
        return basic_food.make()+"面包";
    }
}

public class Cream extends Food {

    private Food basic_food;

    public Cream(Food basic_food) {
        this.basic_food = basic_food;
    }

    public String make() {
        return basic_food.make()+"奶油";
    }
}

public class Vegetable extends Food {

    private Food basic_food;

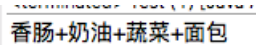
    public Vegetable(Food basic_food) {
        this.basic_food = basic_food;
    }

    public String make() {
        return basic_food.make()+"蔬菜";
    }
}
```

这几个类都是差不多的，构造方法传入一个Food类型的参数，然后在make方法中加入一些自己的逻辑，如果你还是看不懂为什么这么写，不急，你看看我的Test类是怎么写的，一看你就明白了

```
public class Test {  
    public static void main(String[] args) {  
        Food food = new Bread(new Vegetable(new Cream(new Food("香肠"))));  
        System.out.println(food.make());  
    }  
}
```

看到没有，一层一层封装，我们从里往外看：最里面我new了一个香肠，在香肠的外面我包裹了一层奶油，在奶油的外面我又加了一层蔬菜，最外面我放的是面包，是不是很形象，哈哈 这个设计模式简直跟现实生活中一摸一样，看懂了吗?我们看看运行结果



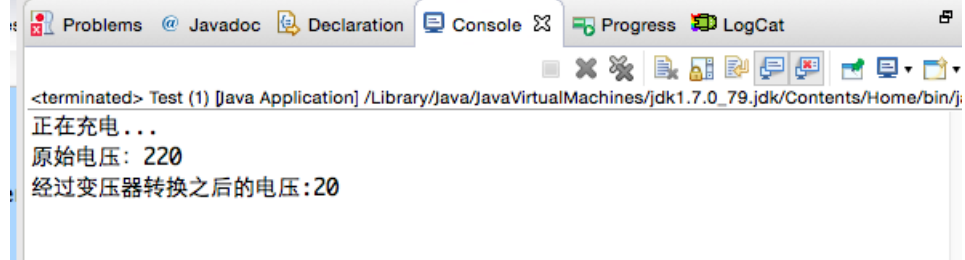
运行结果

一个三明治就做好了～～～

## 适配器模式

将两种完全不同的事物联系到一起，就像现实生活中的变压器。假设一个手机充电器需要的电压是20V，但是正常的电压是220V，这时候就需要一个变压器，将220V的电压转换成20V的电压，这样，变压器就将20V的电压和手机联系起来了。

```
public class Test {  
    public static void main(String[] args) {  
        Phone phone = new Phone();  
        VoltageAdapter adapter = new VoltageAdapter();  
        phone.setAdapter(adapter);  
        phone.charge();  
    }  
}  
  
class Phone {  
  
    public static final int V = 220;  
  
    private VoltageAdapter adapter;  
  
    public void charge() {  
        adapter.changeVoltage();  
    }  
  
    public void setAdapter(VoltageAdapter adapter) {  
        this.adapter = adapter;  
    }  
}  
  
class VoltageAdapter {  
  
    public void changeVoltage() {  
        System.out.println("正在充电...");  
        System.out.println("原始电压: " + Phone.V + "V");  
        System.out.println("经过变压器转换之后的电压:" + (Phone.V - 200) + "V");  
    }  
}
```



## 工厂模式

简单工厂模式：一个抽象的接口，多个抽象接口的实现类，一个工厂类，用来实例化抽象的接口

```

abstract class Car {
    public void run();

    public void stop();
}

class Benz implements Car {
    public void run() {
        System.out.println("Benz开始启动了。。。。");
    }

    public void stop() {
        System.out.println("Benz停车了。。。。");
    }
}

class Ford implements Car {
    public void run() {
        System.out.println("Ford开始启动了。。。");
    }

    public void stop() {
        System.out.println("Ford停车了。。。");
    }
}

class Factory {
    public static Car getCarInstance(String type) {
        Car c = null;
        if ("Benz".equals(type)) {
            c = new Benz();
        }
        if ("Ford".equals(type)) {
            c = new Ford();
        }
        return c;
    }
}

public class Test {

    public static void main(String[] args) {
        Car c = Factory.getCarInstance("Benz");
        if (c != null) {
            c.run();
            c.stop();
        } else {
            System.out.println("造不了这种汽车。。");
        }

    }

}

```

工厂方法模式：有四个角色，抽象工厂模式，具体工厂模式，抽象产品模式，具体产品模式。不再是由一个工厂类去实例化具体的产品，而是由抽象工厂的子类去实例化产品

```

public interface Moveable {
    void run();
}

public class Plane implements Moveable {
    @Override
    public void run() {
        System.out.println("plane....");
    }
}

public class Broom implements Moveable {
    @Override
    public void run() {
        System.out.println("broom.....");
    }
}

public abstract class VehicleFactory {
    abstract Moveable create();
}

public class PlaneFactory extends VehicleFactory {
    public Moveable create() {
        return new Plane();
    }
}

public class BroomFactory extends VehicleFactory {
    public Moveable create() {
        return new Broom();
    }
}

public class Test {
    public static void main(String[] args) {
        VehicleFactory factory = new BroomFactory();
        Moveable m = factory.create();
        m.run();
    }
}

```

抽象工厂模式：与工厂方法模式不同的是，工厂方法模式中的工厂只生产单一的产品，而抽象工厂模式中的工厂生产多个产品



/抽象工厂类

```
public abstract class AbstractFactory {
    public abstract Vehicle createVehicle();
    public abstract Weapon createWeapon();
    public abstract Food createFood();
}

public class DefaultFactory extends AbstractFactory{
    @Override
    public Food createFood() {
        return new Apple();
    }
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
    @Override
    public Weapon createWeapon() {
        return new AK47();
    }
}

public class Test {
    public static void main(String[] args) {
        AbstractFactory f = new DefaultFactory();
        Vehicle v = f.createVehicle();
        v.run();
        Weapon w = f.createWeapon();
        w.shoot();
        Food a = f.createFood();
        a.printName();
    }
}
```

## 代理模式（proxy）

有两种，静态代理和动态代理。先说静态代理，很多理论性的东西我不讲，我就算讲了，你们也看不懂。什么真实角色，抽象角色，代理角色，委托角色。。。乱七八糟的，我是看不懂。之前学代理模式的时候，去网上翻一下，资料一大堆，打开链接一看，基本上都是给你分析有什么什么角色，理论一大堆，看起来很费劲，不信的话你们可以去看看，我是看不懂他们在说什么。咱不来虚的，直接用生活中的例子说话。（注意：我这里并不是否定理论知识，我只是觉得有时候理论知识晦涩难懂，喜欢挑刺的人一边走，你是来学习知识的，不是来挑刺的）

到了一定的年龄，我们就要结婚，结婚是一件很麻烦的事情，（包括那些被父母催婚的）。有钱的家庭可能会找司仪来主持婚礼，显得热闹，洋气～好了，现在婚庆公司的生意来了，我们只需要给钱，婚庆公司就会帮我们安排一整套结婚的流程。整个流程大概是这样的：家里人催婚->男女双方家庭商定结婚的黄道吉日->找一家靠谱的婚庆公司->在约定的时间举行结婚仪式->结婚完毕

婚庆公司打算怎么安排婚礼的节目，在婚礼完毕以后婚庆公司会做什么，我们一概不知。。。别担心，不是黑中介，我们只要把钱给人家，人家会把事情给我们做好。所以，这里的婚庆公司相当于代理角色，现在明白什么是代理角色了吧。

代码实现请看：

```
public interface ProxyInterface {

    void marry();

}
```

Tips：欢迎关注微信公众号：Java后端，获取更多技术博文推送。

文明社会，代理吃饭，代理拉屎什么的我就不写了，有伤社会风化～～～能明白就好

好了，我们看看婚庆公司的代码：

```
public class WeddingCompany implements ProxyInterface {

    private ProxyInterface proxyInterface;

    public WeddingCompany(ProxyInterface proxyInterface) {
        this.proxyInterface = proxyInterface;
    }

    @Override
    public void marry() {
        System.out.println("我们是婚庆公司的");
        System.out.println("我们在做结婚前的准备工作");
        System.out.println("节目彩排...");
        System.out.println("礼物购买...");
        System.out.println("工作人员分工...");
        System.out.println("可以开始结婚了");
        proxyInterface.marry();
        System.out.println("结婚完毕，我们需要做后续处理，你们可以回家了，其余的事情我们公司来做");
    }

}
```

看到没有，婚庆公司需要做的事情很多，我们再看看结婚家庭的代码：

```
public class NormalHome implements ProxyInterface{

    @Override
    public void marry() {
        System.out.println("我们结婚啦～");
    }

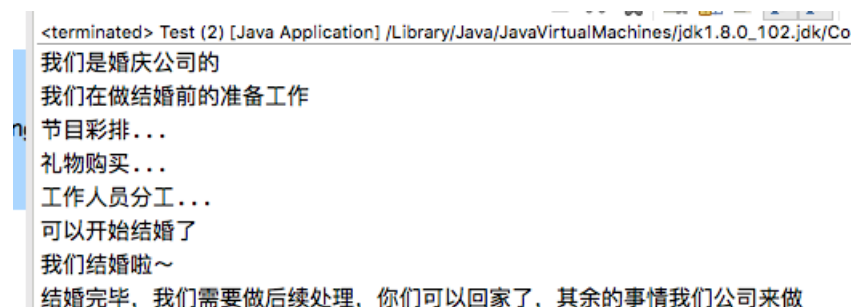
}
```

这个已经很明显了，结婚家庭只需要结婚，而婚庆公司要包揽一切，前前后后的事情都是婚庆公司来做，听说现在婚庆公司很赚钱的，这就是原因，干的活多，能不赚钱吗？

来看看测试类代码：

```
public class Test {
    public static void main(String[] args) {
        ProxyInterface proxyInterface = new WeddingCompany(new NormalHome());
        proxyInterface.marry();
    }
}
```

运行结果如下：



The screenshot shows a Java application window titled "<terminated> Test (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_102.jdk/Co". The output text is as follows:

```
<terminated> Test (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Co
我们是婚庆公司的
我们在做结婚前的准备工作
节目彩排...
礼物购买...
工作人员分工...
可以开始结婚了
我们结婚啦～
结婚完毕，我们需要做后续处理，你们可以回家了，其余的事情我们公司来做
```

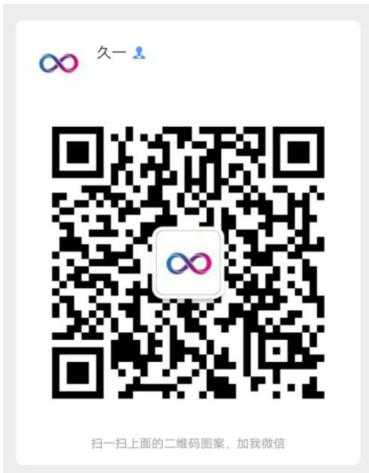
在我们预料中，结果正确，这就是静态代理，动态代理我就不想说了，跟java反射有关。

链接 | [www.jianshu.com/p/93bc5aa1f887](http://www.jianshu.com/p/93bc5aa1f887)

-END-

如果看到这里,说明你喜欢这篇文章,请**转发、点赞**。微信搜索「web\_resource」,关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓ 扫描二维码进群 ↓



推荐阅读

- 1. 盘点阿里巴巴 33 个牛逼的开源项目
- 2. RESTful 架构基础
- 3. 理解 IntelliJ IDEA 的项目配置和 Web 部署
- 4. Java 开发中常用的 4 种加密方法
- 5. 团队开发中 Git 最佳实践



学Java, 请关注公众号:Java后端

喜欢文章, 点个在看

阅读原文