# 面试官：Mybatis 使用了哪些设计模式?

java进阶架构师 2019-09-30

来源：crazyant.net/2022.html

虽然我们都知道有20多个设计模式，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。

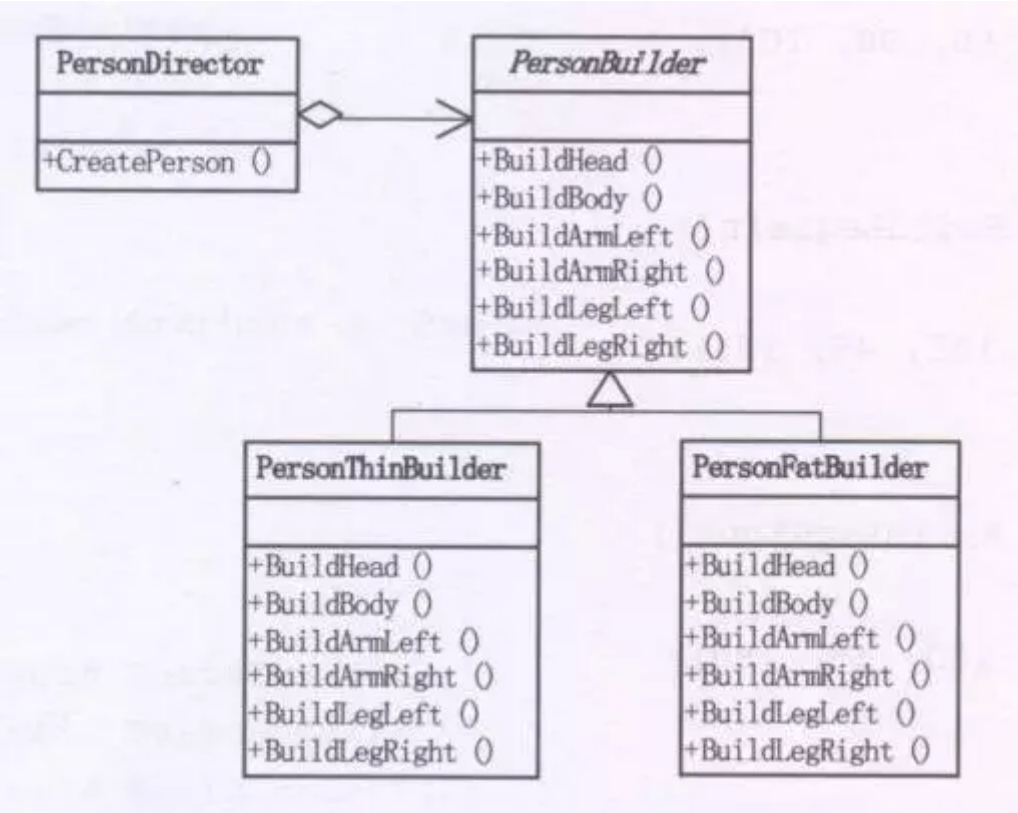
Mybatis至少遇到了以下的设计模式的使用：

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。

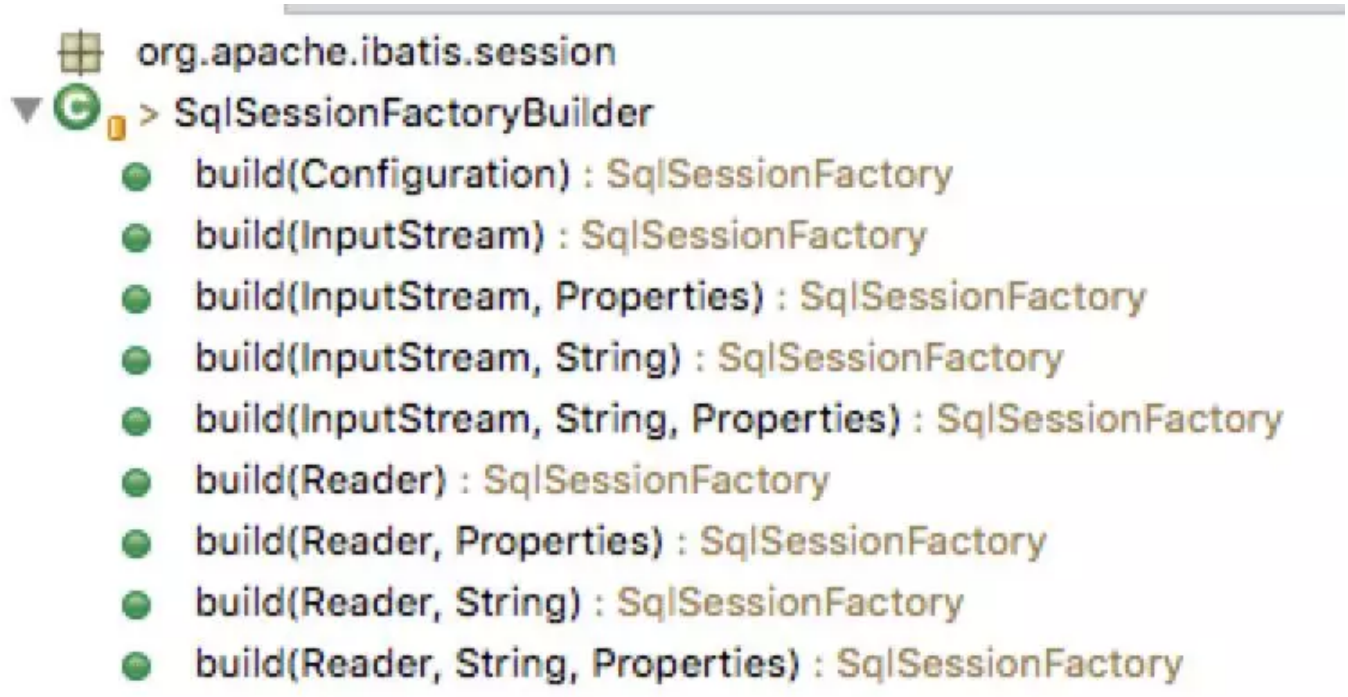



在Mybatis环境的初始化过程中，SqlSessionFactoryBuilder会调用XMLConfigBuilder读取所有的MybatisMapConfig.xml和所有的*Mapper.xml文件，构建Mybatis运行的核心对象Configuration对象，然后将该Configuration对象作为参数构建一个SqlSessionFactory对象。

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：

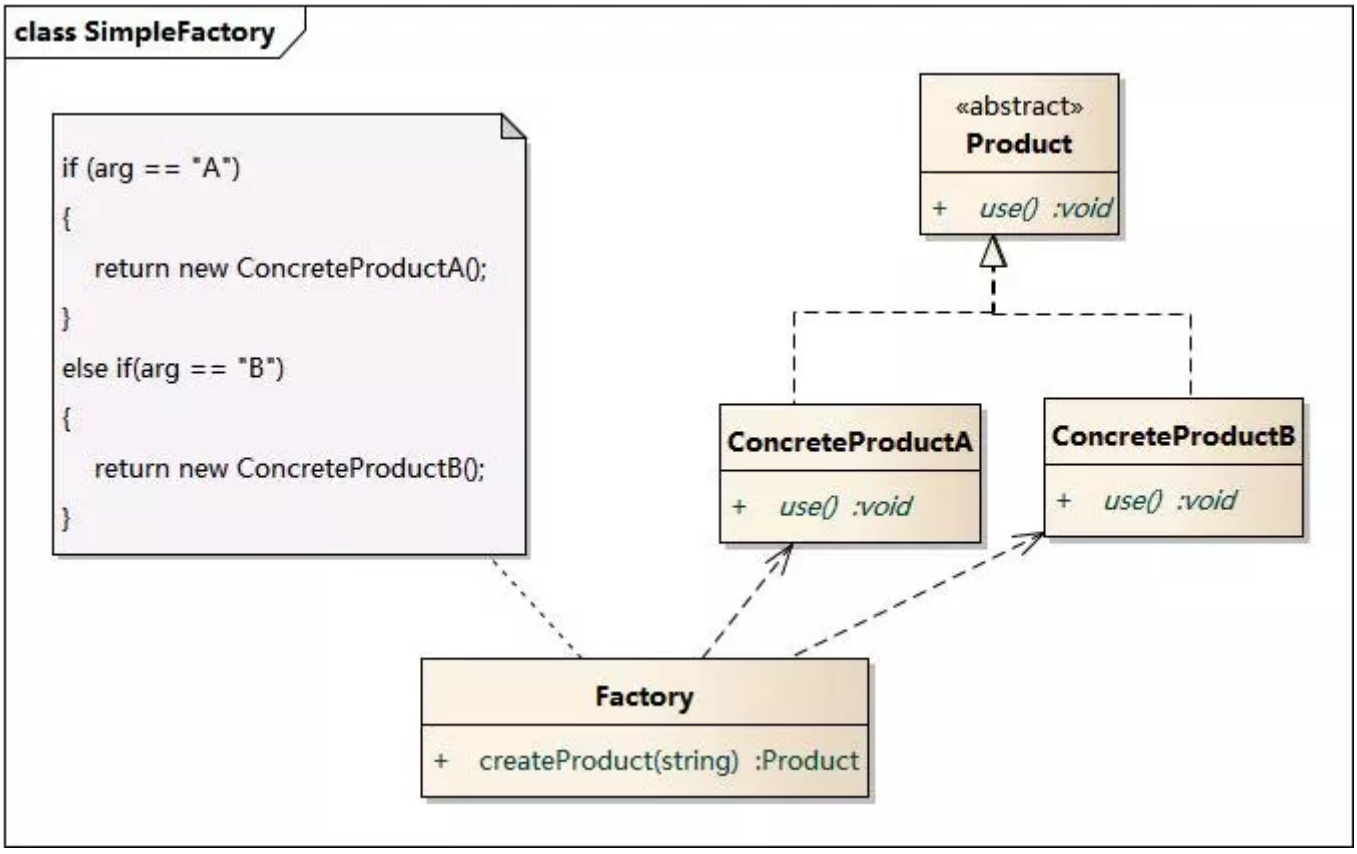



即根据不同的输入参数来构建SqlSessionFactory这个工厂对象。

## 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执行SQL语句、获取Mappers、管理事务。类似于连接MySQL的Connection对象。


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 lev
        boolean autoCommit){
    Transaction tx = null;
    try {
      final Environment environment = configuration.getEnvironment();
      final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(envirc
      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：

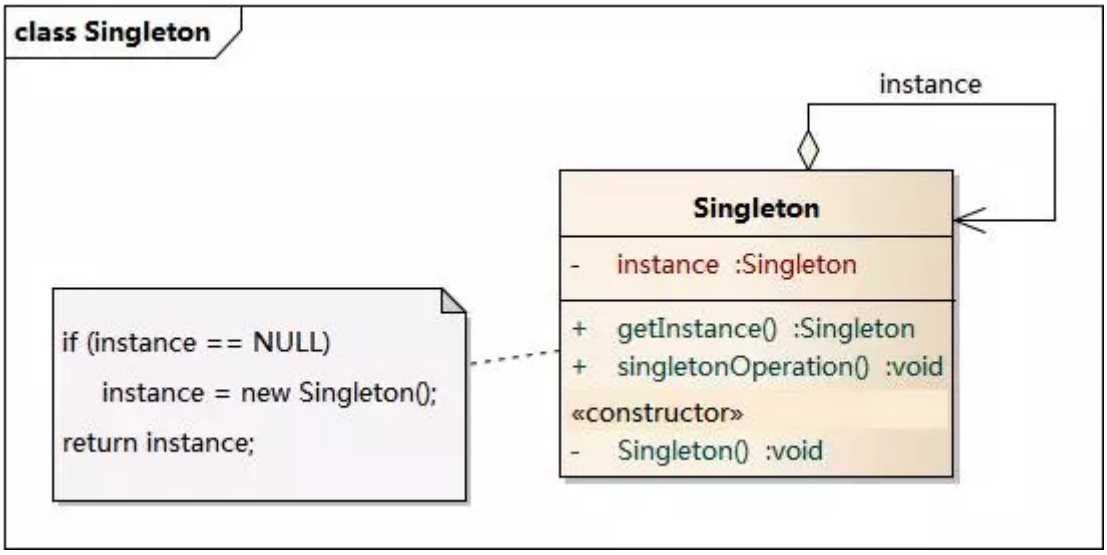
```
public final class LogFactory {
  private static Constructor<? extends Log> logConstructor;

  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变量的的类型是Constructor<? **extends** Log>，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。设计模式之单例模式实践，这篇文章推荐你看下。

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。

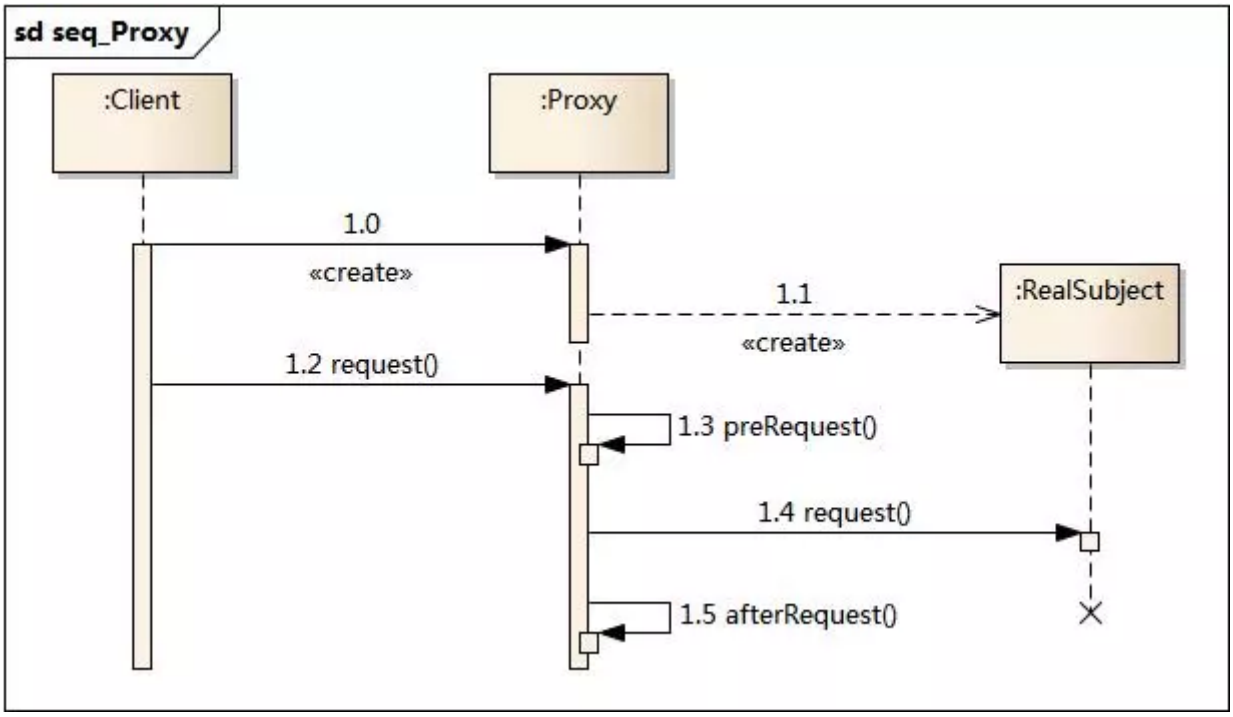
### 4、代理模式

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, MapperMethc

  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[] { mapperInte
        mapperProxy);
```

```
    }

    public T newInstance(SqlSession sqlSession) {
        final MapperProxy<T> mapperProxy = new MapperProxy<T>(sqlSession, mapperInterface, methodCac
        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。

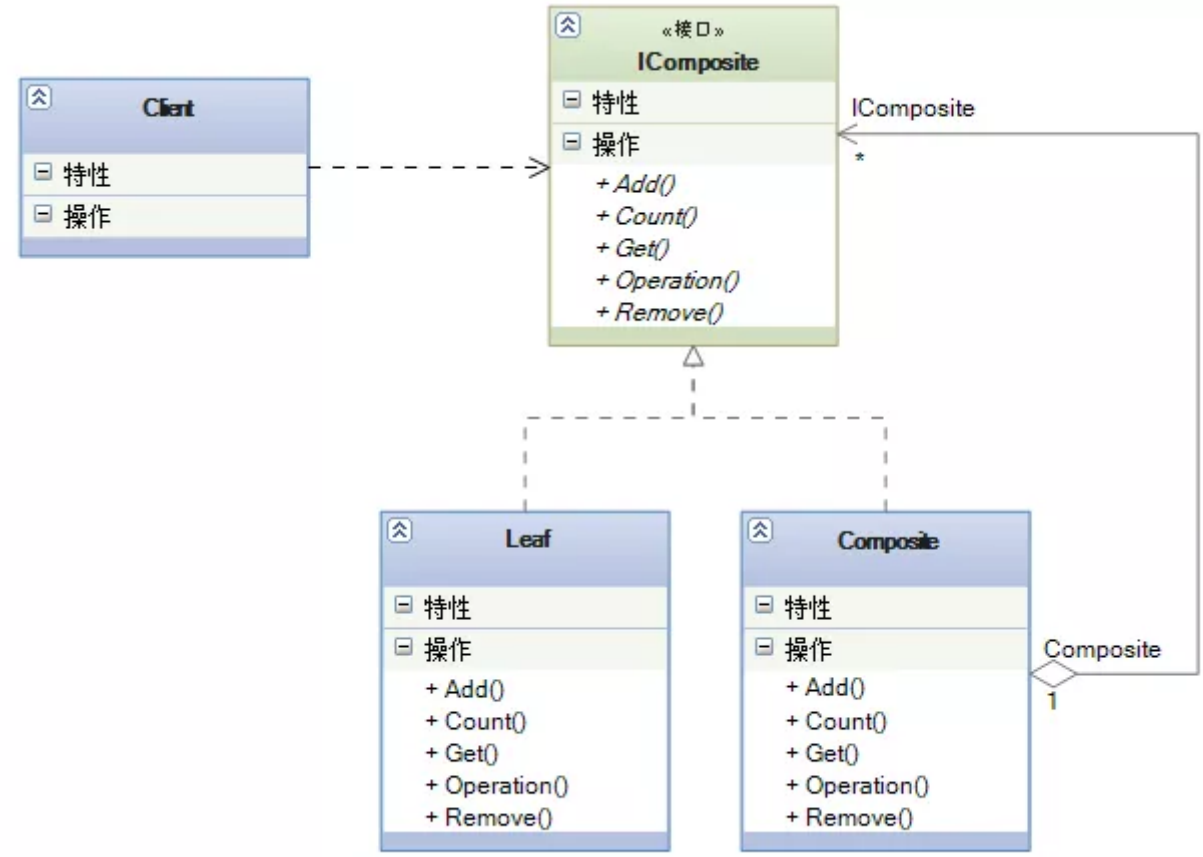
通过这种方式，我们只需要编写Mapper.java接口类，当真正执行一个Mapper接口的时候，就会转发给MapperProxy.invoke方法，而该方法则会调用后续的sqlSession.cud>executor.execute>prepareStatement等一系列方法，完成SQL的执行和返回。

## 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：

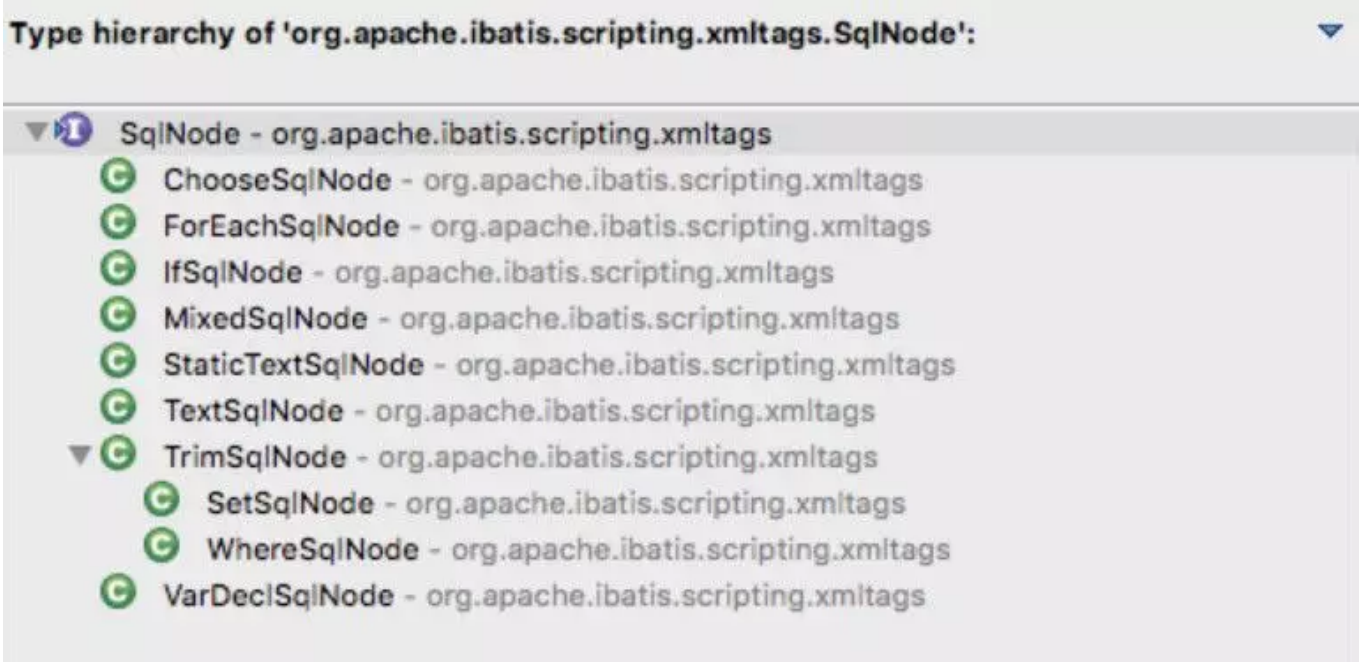
```
<update id="update" parameterType="org.format.dynamicproxy.mybatis.bean.User">
    UPDATE users
    <trim prefix="SET" prefixOverrides=",">
        <if test="name != null and name != ''">
            name = #{name}
        </if>
        <if test="age != null and age != ''">
            , age = #{age}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

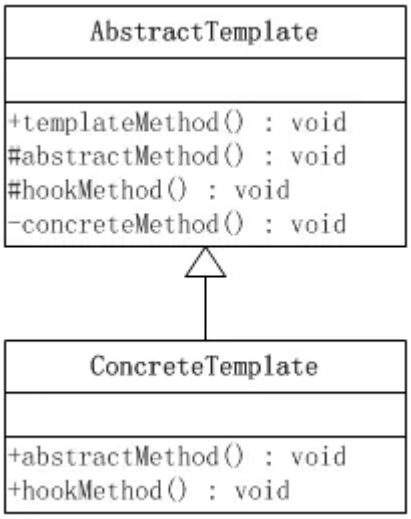
```
            return true;
        }
        return false;
    }
```

## 6、模板方法模式

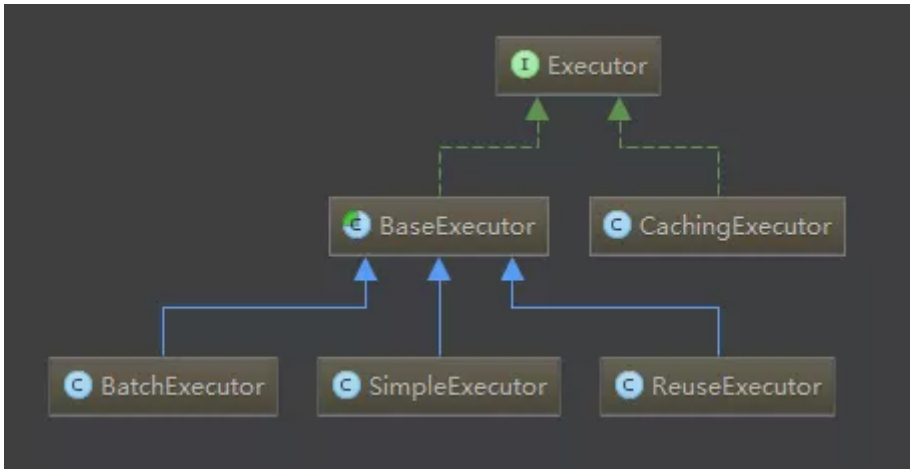
模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。关注Java技术栈微信公众号，在后台回复关键字：*架构*，可以获取更多栈长整理的架构和设计模式干货。

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：

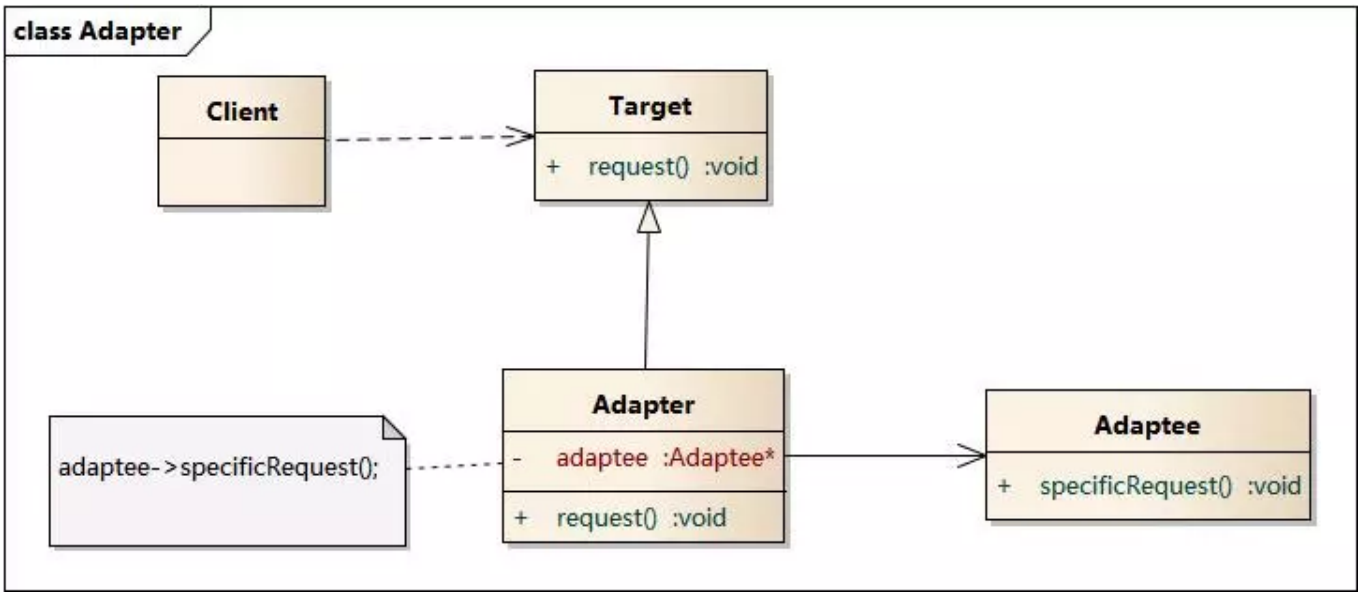
- 简单SimpleExecutor：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。（可以是Statement或PrepareStatement对象）
- 重用ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。（可以是Statement或PrepareStatement对象）
- 批量BatchExecutor：执行update（没有select，JDBC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PrepareStatement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, RowBoun
             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。

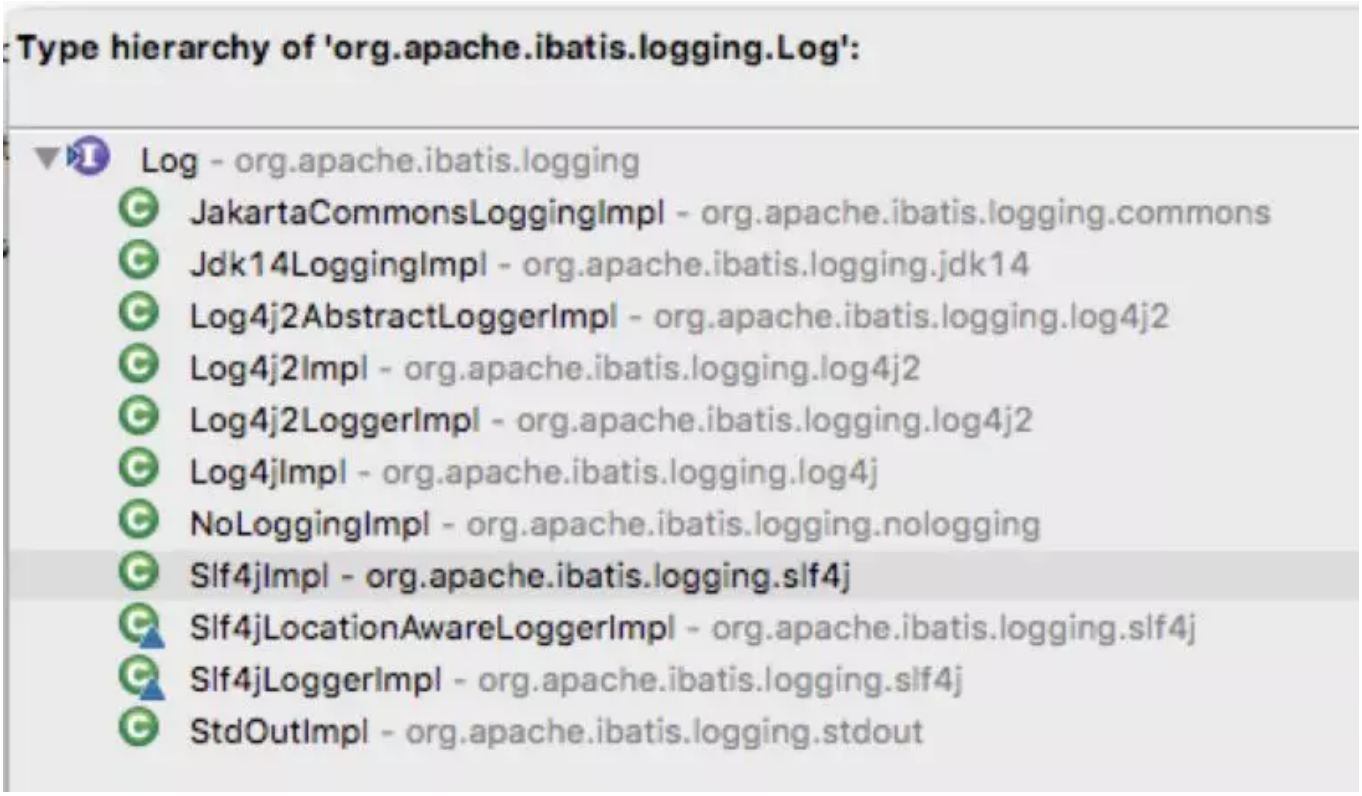



在Mybatsi的logging包中，有一个Log接口：

```
/**
* @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：



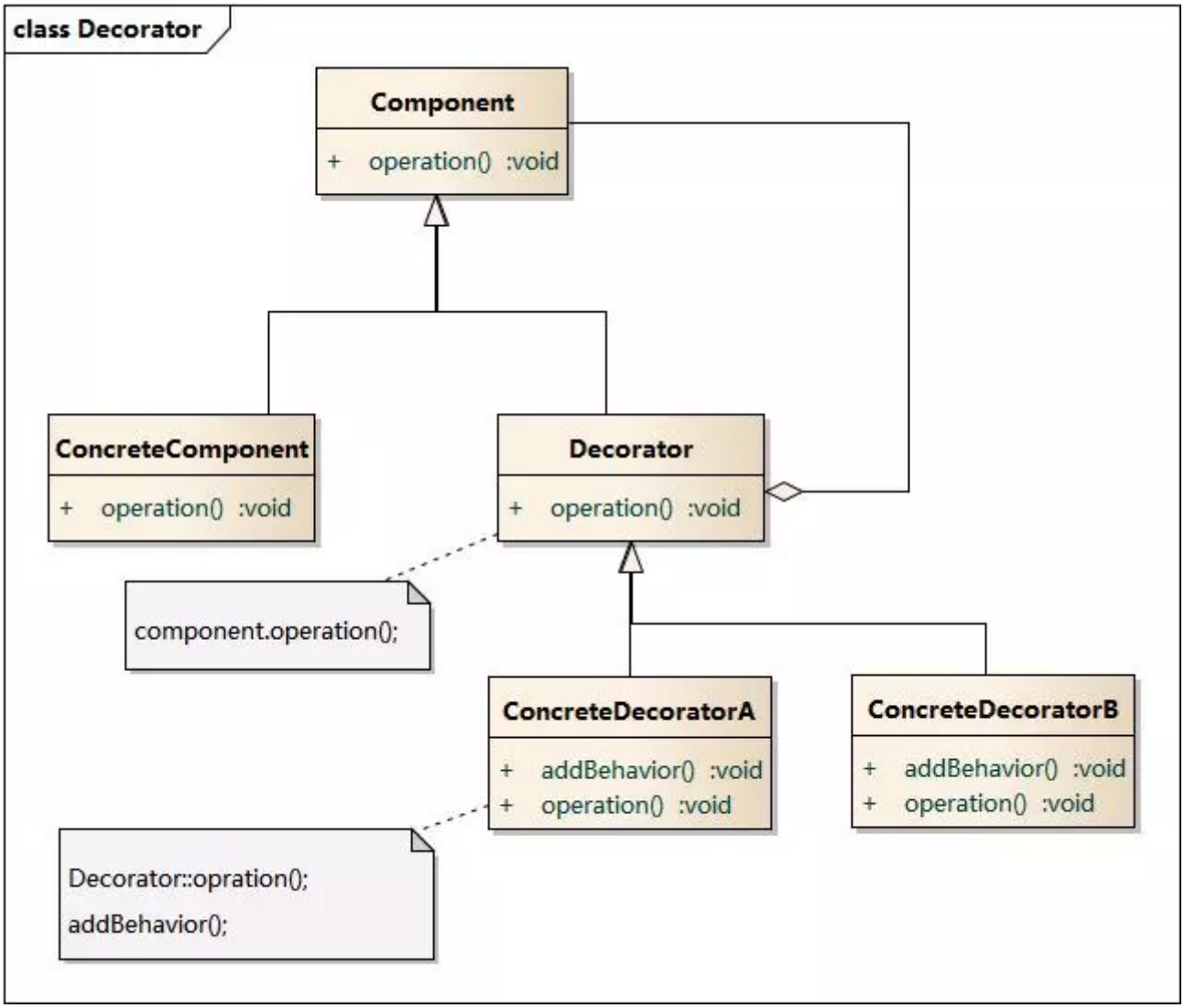

比如对于Log4jImpl的实现来说，该实现持有了org.apache.log4j.Logger的实例，然后所有的日志方法，均委托该实例来实现。

```
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);
    }

}
```

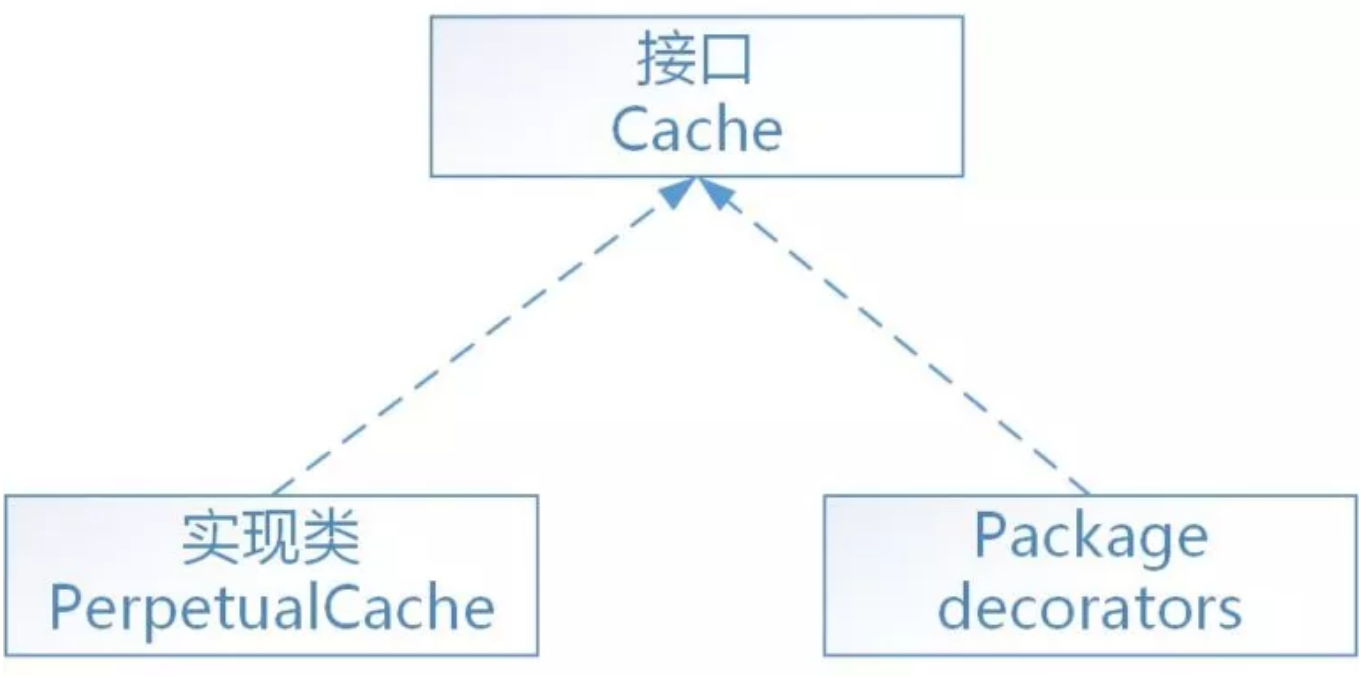
## 8、装饰者模式

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。关注Java技术栈微信公众号，在后台回复关键字：*架构*，可以获取更多栈长整理的架构和设计模式干货。

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。

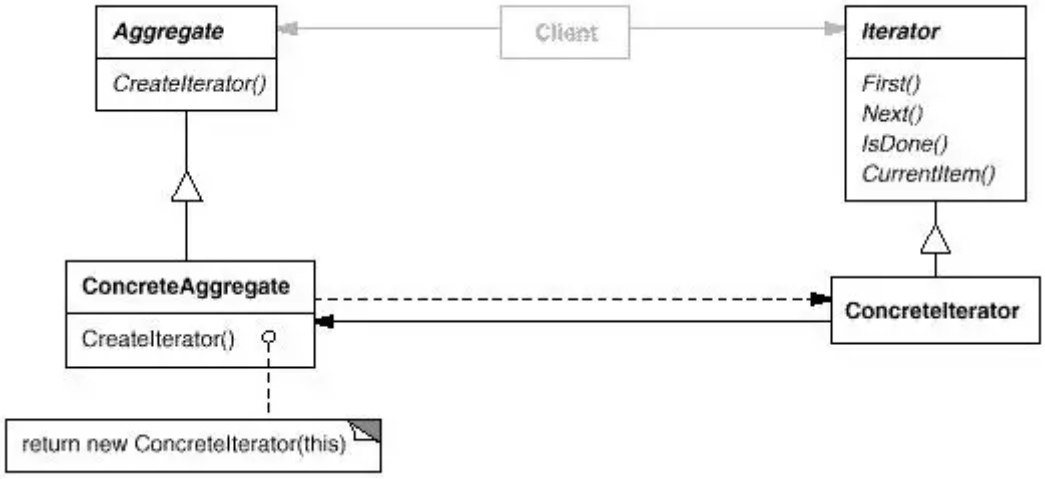
二级缓存对象的默认类型为PerpetualCache，如果配置的缓存是默认类型，则mybatis会根据配置自动追加一系列装饰器。

Cache对象之间的引用顺序为：

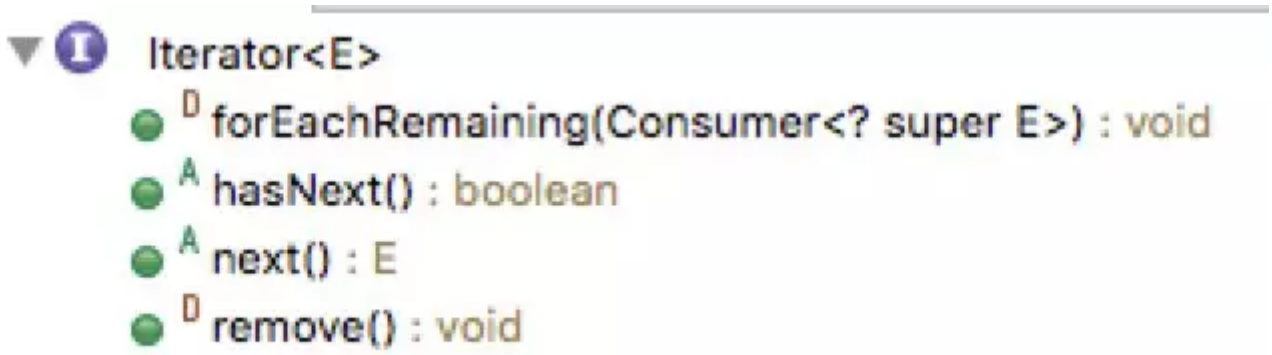
SynchronizedCache–>LoggingCache–>SerializedCache–>ScheduledCache–>LruCache–>PerpetualCache

## 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：




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。

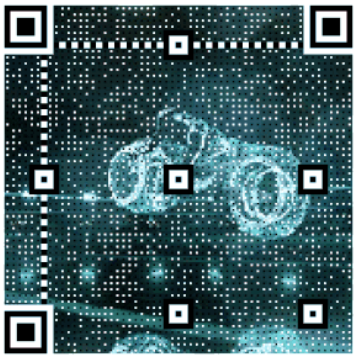
目前师长号内正在【原创】连载设计模式专题，关于设计模式的文章确实有很多，但师长发现内容雷同比较多，甚至连内容的结构也比较统一单调，连文章的风格也都相去不远。设计模式属于技术体系中比较理论的内容，比较枯燥，要是再套用书本上那种正式严肃的风格去写，会比较痛苦。

区别于网上复制粘贴，漏洞百出，陈旧不堪的水文，师长号内原创的设计模式专题，力求浅显易懂，每一篇都是认真雕琢，经过实例验证的。**专注原创高质量技术文输出！**
**设计模式专题：**

- **让设计模式飞一会儿|①开篇获奖感言**
- **让设计模式飞一会儿|②单例模式**
- **让设计模式飞一会儿|③工厂模式**
- **【原创】让设计模式飞一会儿|④原型模式**

———— e n d ————

**微服务、高并发、JVM调优、面试专栏等20大进阶架构师专题请关注公众号【Java进阶架构师】后在菜单栏查看。**



**看到这里，说明你喜欢本文**
**你的转发，是对我最大的鼓励！在看亦是支持**