

## 89 | 开源实战五（下）：总结MyBatis框架中用到的10种设计模式

2020-05-27 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 14:46 大小 13.53M




上节课，我带你剖析了利用职责链模式和动态代理模式实现 MyBatis Plugin。至此，我们已经学习了三种职责链常用的应用场景：过滤器（Servlet Filter）、拦截器（Spring Interceptor）、插件（MyBatis Plugin）。

今天，我们再对 MyBatis 用到的设计模式做一个总结。它用到的设计模式也不少，就我所知的不下十几种。有些我们前面已经讲到，有些比较简单。有了前面这么多讲的学习和训练，我想你现在应该已经具备了一定的研究和分析能力，能够自己做查缺补漏，把提到的所有源码都搞清楚。所以，在今天的课程中，如果有哪里有疑问，你尽可以去查阅源码，☆ 先去学习一下，有不懂的地方，再到评论区和大家一起交流。

话不多说，让我们正式开始今天的学习吧！

## SqlSessionFactoryBuilder：为什么要用建造者模式来创建 SqlSessionFactory？

在 [第 87 讲](#) 中，我们通过一个查询用户的例子展示了用 MyBatis 进行数据库编程。为了方便你查看，我把相关的代码重新摘抄到这里。


 复制代码

```
1 public class MyBatisDemo {
2     public static void main(String[] args) throws IOException {
3         Reader reader = Resources.getResourceAsReader("mybatis.xml");
4         SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader);
5         SqlSession session = sessionFactory.openSession();
6         UserMapper userMapper = session.getMapper(UserMapper.class);
7         UserDo userDo = userMapper.selectById(8);
8         //...
9     }
10 }
```

针对这段代码，请你思考一下下面这个问题。

之前讲到建造者模式的时候，我们使用 Builder 类来创建对象，一般都是先级联一组 setXXX() 方法来设置属性，然后再调用 build() 方法最终创建对象。但是，在上面这段代码中，通过 SqlSessionFactoryBuilder 来创建 SqlSessionFactory 并不符合这个套路。它既没有 setter 方法，而且 build() 方法也并非无参，需要传递参数。除此之外，从上面的代码来看，SqlSessionFactory 对象的创建过程也并不复杂。那直接通过构造函数来创建 SqlSessionFactory 不就行了吗？为什么还要借助建造者模式创建 SqlSessionFactory 呢？

要回答这个问题，我们就要先看下 SqlSessionFactoryBuilder 类的源码。我把源码摘抄到了这里，如下所示：

 复制代码

```
1 public class SqlSessionFactoryBuilder {
2     public SqlSessionFactory build(Reader reader) {
3         return build(reader, null, null);
4     }
5
6     public SqlSessionFactory build(Reader reader, String environment) {
7         return build(reader, environment, null);
8     }
9 }
```

```

9  public SqlSessionFactory build(Reader reader, Properties properties) {
10     return build(reader, null, properties);
11 }
12
13 public SqlSessionFactory build(Reader reader, String environment, Properties
14     try {
15         XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, prop
16         return build(parser.parse());
17     } catch (Exception e) {
18         throw ExceptionFactory.wrapException("Error building SqlSession.", e);
19     } finally {
20         ErrorContext.instance().reset();
21         try {
22             reader.close();
23         } catch (IOException e) {
24             // Intentionally ignore. Prefer previous error.
25         }
26     }
27 }
28
29 public SqlSessionFactory build(InputStream inputStream) {
30     return build(inputStream, null, null);
31 }
32
33 public SqlSessionFactory build(InputStream inputStream, String environment)
34     return build(inputStream, environment, null);
35 }
36
37 public SqlSessionFactory build(InputStream inputStream, Properties propertie
38     return build(inputStream, null, properties);
39 }
40
41 public SqlSessionFactory build(InputStream inputStream, String environment, l
42     try {
43         XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment,
44         return build(parser.parse());
45     } catch (Exception e) {
46         throw ExceptionFactory.wrapException("Error building SqlSession.", e);
47     } finally {
48         ErrorContext.instance().reset();
49         try {
50             inputStream.close();
51         } catch (IOException e) {
52             // Intentionally ignore. Prefer previous error.
53         }
54     }
55 }
56
57 public SqlSessionFactory build(Configuration config) {
58     return new DefaultSqlSessionFactory(config);
59 }
60

```

SqlSessionFactoryBuilder 类中有大量的 build() 重载函数。为了方便你查看，以及待会儿跟 SqlSessionFactory 类的代码作对比，我把重载函数定义抽象出来，贴到这里。

[复制代码](#)

```
1 public class SqlSessionFactoryBuilder {
2     public SqlSessionFactory build(Reader reader);
3     public SqlSessionFactory build(Reader reader, String environment);
4     public SqlSessionFactory build(Reader reader, Properties properties);
5     public SqlSessionFactory build(Reader reader, String environment, Properties
6
7     public SqlSessionFactory build(InputStream inputStream);
8     public SqlSessionFactory build(InputStream inputStream, String environment);
9     public SqlSessionFactory build(InputStream inputStream, Properties propertie:
10    public SqlSessionFactory build(InputStream inputStream, String environment, l
11
12    // 上面所有的方法最终都调用这个方法
13    public SqlSessionFactory build(Configuration config);
14 }
```

我们知道，如果一个类包含很多成员变量，而构建对象并不需要设置所有的成员变量，只需要选择性地设置其中几个就可以。为了满足这样的构建需求，我们就要定义多个包含不同参数列表的构造函数。为了避免构造函数过多、参数列表过长，我们一般通过无参构造函数加 setter 方法或者通过建造者模式来解决。

从建造者模式的设计初衷上来看，SqlSessionFactoryBuilder 虽然带有 Builder 后缀，但不要被它的名字所迷惑，它并不是标准的建造者模式。一方面，原始类 SqlSessionFactory 的构建只需要一个参数，并不复杂。另一方面，Builder 类 SqlSessionFactoryBuilder 仍然定义了 n 多包含不同参数列表的构造函数。

实际上，SqlSessionFactoryBuilder 设计的初衷只不过是为了简化开发。因为构建 SqlSessionFactory 需要先构建 Configuration，而构建 Configuration 是非常复杂的，需要做很多工作，比如配置的读取、解析、创建 n 多对象等。为了将构建 SqlSessionFactory 的过程隐藏起来，对程序员透明，MyBatis 就设计了 SqlSessionFactoryBuilder 类封装这些构建细节。

## SqlSessionFactory：到底属于工厂模式还是建造器模式？

在刚刚那段 MyBatis 示例代码中，我们通过 `SqlSessionFactoryBuilder` 创建了 `SqlSessionFactory`，然后再通过 `SqlSessionFactory` 创建了 `SqlSession`。刚刚我们讲了 `SqlSessionFactoryBuilder`，现在我们再来看下 `SqlSessionFactory`。

从名字上，你可能已经猜到，`SqlSessionFactory` 是一个工厂类，用到的设计模式是工厂模式。不过，它跟 `SqlSessionFactoryBuilder` 类似，名字有很大的迷惑性。实际上，它也并不是标准的工厂模式。为什么这么说呢？我们先来看下 `SqlSessionFactory` 类的源码。

 复制代码

```
1 public interface SqlSessionFactory {
2     SqlSession openSession();
3     SqlSession openSession(boolean autoCommit);
4     SqlSession openSession(Connection connection);
5     SqlSession openSession(TransactionIsolationLevel level);
6     SqlSession openSession(ExecutorType execType);
7     SqlSession openSession(ExecutorType execType, boolean autoCommit);
8     SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level);
9     SqlSession openSession(ExecutorType execType, Connection connection);
10    Configuration getConfiguration();
11 }
```

`SqlSessionFactory` 是一个接口，`DefaultSqlSessionFactory` 是它唯一的实现类。  
`DefaultSqlSessionFactory` 源码如下所示：

 复制代码

```
1 public class DefaultSqlSessionFactory implements SqlSessionFactory {
2     private final Configuration configuration;
3     public DefaultSqlSessionFactory(Configuration configuration) {
4         this.configuration = configuration;
5     }
6
7     @Override
8     public SqlSession openSession() {
9         return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, false);
10    }
11
12    @Override
13    public SqlSession openSession(boolean autoCommit) {
14        return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, autoCommit);
15    }
16
17    @Override
18    public SqlSession openSession(ExecutorType execType) {
19        return openSessionFromDataSource(execType, null, false);
20    }
21
22    @Override
23    public SqlSession openSession(ExecutorType execType, boolean autoCommit) {
24        return openSessionFromDataSource(execType, null, autoCommit);
25    }
26
27    @Override
28    public SqlSession openSession(Connection connection) {
29        return openSessionFromDataSource(configuration.getDefaultExecutorType(), connection, false);
30    }
31
32    @Override
33    public SqlSession openSession(TransactionIsolationLevel level) {
34        return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, false);
35    }
36
37    @Override
38    public Configuration getConfiguration() {
39        return configuration;
40    }
41}
```



```

19     return openSessionFromDataSource(execType, null, false);
20 }
21
22 @Override
23 public SqlSession openSession(TransactionIsolationLevel level) {
24     return openSessionFromDataSource(configuration.getDefaultExecutorType(), level, false);
25 }
26
27 @Override
28 public SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level) {
29     return openSessionFromDataSource(execType, level, false);
30 }
31
32 @Override
33 public SqlSession openSession(ExecutorType execType, boolean autoCommit) {
34     return openSessionFromDataSource(execType, null, autoCommit);
35 }
36
37 @Override
38 public SqlSession openSession(Connection connection) {
39     return openSessionFromConnection(configuration.getDefaultExecutorType(), connection, false);
40 }
41
42 @Override
43 public SqlSession openSession(ExecutorType execType, Connection connection) {
44     return openSessionFromConnection(execType, connection, false);
45 }
46
47 @Override
48 public Configuration getConfiguration() {
49     return configuration;
50 }
51
52 private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level,
53     Transaction tx = null;
54     try {
55         final Environment environment = configuration.getEnvironment();
56         final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
57         tx = transactionFactory.newTransaction(environment.getDataSource(), level);
58         final Executor executor = configuration.newExecutor(tx, execType);
59         return new DefaultSqlSession(configuration, executor, autoCommit);
60     } catch (Exception e) {
61         closeTransaction(tx); // may have fetched a connection so lets call close
62         throw ExceptionFactory.wrapException("Error opening session. Cause: " + e, e);
63     } finally {
64         ErrorContext.instance().reset();
65     }
66 }
67
68 private SqlSession openSessionFromConnection(ExecutorType execType, Connection connection, boolean
69     try {
70         boolean autoCommit;

```

```

71     try {
72         autoCommit = connection.getAutoCommit();
73     } catch (SQLException e) {
74         // Failover to true, as most poor drivers
75         // or databases won't support transactions
76         autoCommit = true;
77     }
78     final Environment environment = configuration.getEnvironment();
79     final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment();
80     final Transaction tx = transactionFactory.newTransaction(connection);
81     final Executor executor = configuration.newExecutor(tx, execType);
82     return new DefaultSqlSession(configuration, executor, autoCommit);
83 } catch (Exception e) {
84     throw ExceptionFactory.wrapException("Error opening session. Cause: " +
85     e);
86 } finally {
87     ErrorContext.instance().reset();
88 }
89 //...省略部分代码...
90

```

从 `SqlSessionFactory` 和 `DefaultSqlSessionFactory` 的源码来看，它的设计非常类似刚刚讲到的 `SqlSessionFactoryBuilder`，通过重载多个 `openSession()` 函数，支持通过组合 `autoCommit`、`Executor`、`Transaction` 等不同参数，来创建 `SqlSession` 对象。标准的工厂模式通过 `type` 来创建继承同一个父类的不同子类对象，而这里只不过是传递进不同的参数，来创建同一个类的对象。所以，它更像建造者模式。

虽然设计思路基本一致，但一个叫 `xxxBuilder` (`SqlSessionFactoryBuilder`)，一个叫 `xxxFactory` (`SqlSessionFactory`)。而且，叫 `xxxBuilder` 的也并非标准的建造者模式，叫 `xxxFactory` 的也并非标准的工厂模式。所以，我个人觉得，MyBatis 对这部分代码的设计还是值得优化的。

实际上，这两个类的作用只不过是为了创建 `SqlSession` 对象，没有其他作用。所以，我更建议参照 Spring 的设计思路，把 `SqlSessionFactoryBuilder` 和 `SqlSessionFactory` 的逻辑，放到一个叫 “`ApplicationContext`” 的类中。让这个类来全权负责读入配置文件，创建 `Configuration`，生成 `SqlSession`。

## BaseExecutor：模板模式跟普通的继承有什么区别？

如果去查阅 `SqlSession` 与 `DefaultSqlSession` 的源码，你会发现，`SqlSession` 执行 SQL 的业务逻辑，都是委托给了 `Executor` 来实现。`Executor` 相关的类主要是用来执行 SQL。

其中，Executor 本身是一个接口；BaseExecutor 是一个抽象类，实现了 Executor 接口；而 BatchExecutor、SimpleExecutor、ReuseExecutor 三个类继承 BaseExecutor 抽象类。

那 BatchExecutor、SimpleExecutor、ReuseExecutor 三个类跟 BaseExecutor 是简单的继承关系，还是模板模式关系呢？怎么来判断呢？我们看一下 BaseExecutor 的源码就清楚了。

 复制代码

```
1 public abstract class BaseExecutor implements Executor {
2     //...省略其他无关代码...
3
4     @Override
5     public int update(MappedStatement ms, Object parameter) throws SQLException {
6         ErrorContext.instance().resource(ms.getResource()).activity("executing an update");
7         if (closed) {
8             throw new ExecutorException("Executor was closed.");
9         }
10        clearLocalCache();
11        return doUpdate(ms, parameter);
12    }
13
14    public List<BatchResult> flushStatements(boolean isRollBack) throws SQLException {
15        if (closed) {
16            throw new ExecutorException("Executor was closed.");
17        }
18        return doFlushStatements(isRollBack);
19    }
20
21    private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, List<E> list) {
22        localCache.putObject(key, EXECUTION_PLACEHOLDER);
23        try {
24            list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
25        } finally {
26            localCache.removeObject(key);
27        }
28        localCache.putObject(key, list);
29        if (ms.getStatementType() == StatementType.CALLABLE) {
30            localOutputParameterCache.putObject(key, parameter);
31        }
32        return list;
33    }
34
35    @Override
36    public <E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds,
37        BoundSql boundSql = ms.getBoundSql(parameter);
```



```


39     return doQueryCursor(ms, parameter, rowBounds, boundSql);
40 }
41
42 protected abstract int doUpdate(MappedStatement ms, Object parameter) throws
43
44 protected abstract List<BatchResult> doFlushStatements(boolean isRollback) tl
45
46 protected abstract <E> List<E> doQuery(MappedStatement ms, Object parameter,
47
48 protected abstract <E> Cursor<E> doQueryCursor(MappedStatement ms, Object pa
49

```

模板模式基于继承来实现代码复用。如果抽象类中包含模板方法，模板方法调用有待子类实现的抽象方法，那这一般就是模板模式的代码实现。而且，在命名上，模板方法与抽象方法一般是一一对应的，抽象方法在模板方法前面多一个“do”，比如，在 BaseExecutor 类中，其中一个模板方法叫 update()，那对应的抽象方法就叫 doUpdate()。

## SqlNode：如何利用解释器模式来解析动态 SQL？

支持配置文件中编写动态 SQL，是 MyBatis 一个非常强大的功能。所谓动态 SQL，就是在 SQL 中可以包含在 trim、if、#{ }等语法标签，在运行时根据条件来生成不同的 SQL。这么说比较抽象，我举个例子解释一下。

 复制代码

```


1 <update id="update" parameterType="com.xzg.cd.a89.User"
2     UPDATE user
3     <trim prefix="SET" prefixOverrides=", ">
4         <if test="name != null and name != ''">
5             name = #{name}
6         </if>
7         <if test="age != null and age != ''">
8             , age = #{age}
9         </if>
10        <if test="birthday != null and birthday != ''">
11            , birthday = #{birthday}
12        </if>
13    </trim>
14    where id = ${id}
15 </update>

```

显然，动态 SQL 的语法规则是 MyBatis 自定义的。如果想要根据语法规则，替换掉动态 SQL 中的动态元素，生成真正可以执行的 SQL 语句，MyBatis 还需要实现对应的解释器。

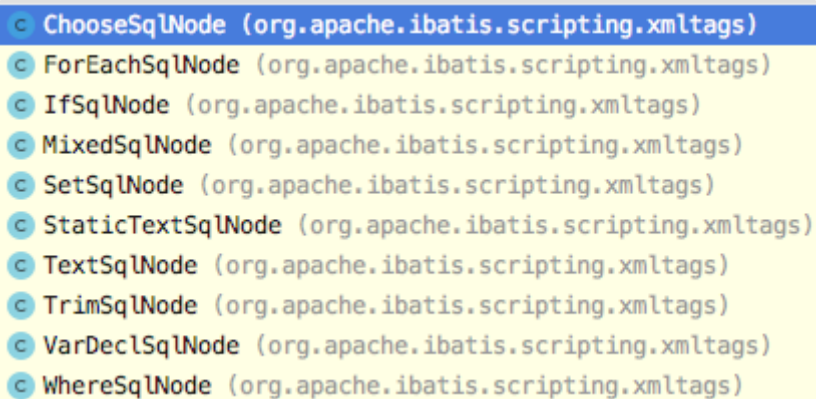
这一部分功能就可以看做是解释器模式的应用。实际上，如果你去查看它的代码实现，你会发现，它跟我们在前面讲解器模式时举的那两个例子的代码结构非常相似。

我们前面提到，解释器模式在解释语法规则的时候，一般会把规则分割成小的单元，特别是可以嵌套的小单元，针对每个小单元来解析，最终再把解析结果合并在一起。这里也不例外。MyBatis 把每个语法小单元叫 SqlNode。SqlNode 的定义如下所示：

 复制代码

```
1 public interface SqlNode {  
2     boolean apply(DynamicContext context);  
3 }
```

对于不同的语法小单元，MyBatis 定义不同的 SqlNode 实现类。



- ChooseSqlNode (org.apache.ibatis.scripting.xmltags)
- ForEachSqlNode (org.apache.ibatis.scripting.xmltags)
- IfSqlNode (org.apache.ibatis.scripting.xmltags)
- MixedSqlNode (org.apache.ibatis.scripting.xmltags)
- SetSqlNode (org.apache.ibatis.scripting.xmltags)
- StaticTextSqlNode (org.apache.ibatis.scripting.xmltags)
- TextSqlNode (org.apache.ibatis.scripting.xmltags)
- TrimSqlNode (org.apache.ibatis.scripting.xmltags)
- VarDeclSqlNode (org.apache.ibatis.scripting.xmltags)
- WhereSqlNode (org.apache.ibatis.scripting.xmltags)


整个解释器的调用入口在 DynamicSqlSource.getBoundSql 方法中，它调用了 rootSqlNode.apply(context) 方法。因为整体的代码结构跟 [第 72 讲](#) 中的例子基本一致，所以每个 SqlNode 实现类的代码，我就不带你一块阅读了，感兴趣的话你可以自己去看下。

## ErrorContext：如何实现一个线程唯一的单例模式？

在单例模式那一部分我们讲到，单例模式是进程唯一的。同时，我们还讲到单例模式的几种变形，比如线程唯一的单例、集群唯一的单例等。在 MyBatis 中，ErrorContext 这个类就是标准单例的变形：线程唯一的单例。

它的代码实现我贴到下面了。它基于 Java 中的 ThreadLocal 来实现。如果不熟悉 ThreadLocal，你可以回过头去看下 [第 43 讲](#) 中线程唯一的单例的实现方法。实际上，这

里的 ThreadLocal 就相当于那里的 ConcurrentHashMap。

 复制代码

```
1 public class ErrorContext {
2     private static final String LINE_SEPARATOR = System.getProperty("line.separator");
3     private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext>();
4
5     private ErrorContext stored;
6     private String resource;
7     private String activity;
8     private String object;
9     private String message;
10    private String sql;
11    private Throwable cause;
12
13    private ErrorContext() {}
14
15
16    public static ErrorContext instance() {
17        ErrorContext context = LOCAL.get();
18        if (context == null) {
19            context = new ErrorContext();
20            LOCAL.set(context);
21        }
22        return context;
23    }
24 }
```

## Cache：为什么要用装饰器模式而不设计成继承子类？

我们前面提到，MyBatis 是一个 ORM 框架。实际上，它不只是简单地完成了对象和数据库数据之间的互相转化，还提供了很多其他功能，比如缓存、事务等。接下来，我们再讲讲它的缓存实现。

在 MyBatis 中，缓存功能由接口 Cache 定义。PerpetualCache 类是最基础的缓存类，是一个大小无限的缓存。除此之外，MyBatis 还设计了 9 个包裹 PerpetualCache 类的装饰器类，用来实现功能增强。它们分别是：FifoCache、LoggingCache、LruCache、ScheduledCache、SerializedCache、SoftCache、SynchronizedCache、WeakCache、TransactionalCache。

 复制代码

```
1 public interface Cache {
2     String getId();
```

```
3  void putObject(Object key, Object value);
4  Object getObject(Object key);
5  Object removeObject(Object key);
6  void clear();
7  int getSize();
8  ReadWriteLock getReadWriteLock();
9  }
10
11 public class PerpetualCache implements Cache {
12     private final String id;
13     private Map<Object, Object> cache = new HashMap<Object, Object>();
14
15     public PerpetualCache(String id) {
16         this.id = id;
17     }
18
19     @Override
20     public String getId() {
21         return id;
22     }
23
24     @Override
25     public int getSize() {
26         return cache.size();
27     }
28
29     @Override
30     public void putObject(Object key, Object value) {
31         cache.put(key, value);
32     }
33
34     @Override
35     public Object getObject(Object key) {
36         return cache.get(key);
37     }
38
39     @Override
40     public Object removeObject(Object key) {
41         return cache.remove(key);
42     }
43
44     @Override
45     public void clear() {
46         cache.clear();
47     }
48
49     @Override
50     public ReadWriteLock getReadWriteLock() {
51         return null;
52     }
53     //省略部分代码...
54 }
```

这 9 个装饰器类的代码结构都类似，我只将其中的 LruCache 的源码贴到这里。从代码中我们可以看出，它是标准的装饰器模式的代码实现。

 复制代码

```
1 public class LruCache implements Cache {
2     private final Cache delegate;
3     private Map<Object, Object> keyMap;
4     private Object eldestKey;
5
6     public LruCache(Cache delegate) {
7         this.delegate = delegate;
8         setSize(1024);
9     }
10
11     @Override
12     public String getId() {
13         return delegate.getId();
14     }
15
16     @Override
17     public int getSize() {
18         return delegate.getSize();
19     }
20
21     public void setSize(final int size) {
22         keyMap = new LinkedHashMap<Object, Object>(size, .75F, true) {
23             private static final long serialVersionUID = 4267176411845948333L;
24
25             @Override
26             protected boolean removeEldestEntry(Map.Entry<Object, Object> eldest) {
27                 boolean tooBig = size() > size;
28                 if (tooBig) {
29                     eldestKey = eldest.getKey();
30                 }
31                 return tooBig;
32             }
33         };
34     }
35
36     @Override
37     public void putObject(Object key, Object value) {
38         delegate.putObject(key, value);
39         cycleKeyList(key);
40     }
41
42     @Override
43     public Object getObject(Object key) {
```

```

44     keyMap.get(key); //touch
45     return delegate.getObject(key);
46 }
47
48 @Override
49 public Object removeObject(Object key) {
50     return delegate.removeObject(key);
51 }
52
53 @Override
54 public void clear() {
55     delegate.clear();
56     keyMap.clear();
57 }
58
59 @Override
60 public ReadWriteLock getReadWriteLock() {
61     return null;
62 }
63
64 private void cycleKeyList(Object key) {
65     keyMap.put(key, key);
66     if (eldestKey != null) {
67         delegate.removeObject(eldestKey);
68         eldestKey = null;
69     }
70 }
71

```

之所以 MyBatis 采用装饰器模式来实现缓存功能，是因为装饰器模式采用了组合，而非继承，更加灵活，能够有效地避免继承关系的组合爆炸。关于这一点，你可以回过头去看下 [第 10 讲](#) 的内容。

## PropertyTokenizer：如何利用迭代器模式实现一个属性解析器？

前面我们讲到，迭代器模式常用来替代 for 循环遍历集合元素。Mybatis 的 PropertyTokenizer 类实现了 Java Iterator 接口，是一个迭代器，用来对配置属性进行解析。具体的代码如下所示：

 复制代码

```

1 // person[0].birthdate.year 会被分解为3个PropertyTokenizer对象。其中，第一个Property
2 public class PropertyTokenizer implements Iterator<PropertyTokenizer> {
3     private String name; // person
4     private final String indexedName; // person[0]
5     private String index; // 0
6     private final String children; // birthdate.year

```




```
7
8 public PropertyTokenizer(String fullname) {
9     int delim = fullname.indexOf('.');
10    if (delim > -1) {
11        name = fullname.substring(0, delim);
12        children = fullname.substring(delim + 1);
13    } else {
14        name = fullname;
15        children = null;
16    }
17    indexedName = name;
18    delim = name.indexOf('[');
19    if (delim > -1) {
20        index = name.substring(delim + 1, name.length() - 1);
21        name = name.substring(0, delim);
22    }
23 }
24
25 public String getName() {
26     return name;
27 }
28
29 public String getIndex() {
30     return index;
31 }
32
33 public String getIndexedName() {
34     return indexedName;
35 }
36
37 public String getChildren() {
38     return children;
39 }
40
41 @Override
42 public boolean hasNext() {
43     return children != null;
44 }
45
46 @Override
47 public PropertyTokenizer next() {
48     return new PropertyTokenizer(children);
49 }
50
51 @Override
52 public void remove() {
53     throw new UnsupportedOperationException("Remove is not supported, as it ha:
54 }
55 }
```

实际上，PropertyTokenizer 类也并非标准的迭代器类。它将配置的解析、解析之后的元素、迭代器，这三部分本该放到三个类中的代码，都耦合在一个类中，所以看起来稍微有点难懂。不过，这样做的好处是能够做到惰性解析。我们不需要事先将整个配置，解析成多个 PropertyTokenizer 对象。只有当我们在调用 next() 函数的时候，才会解析其中部分配置。

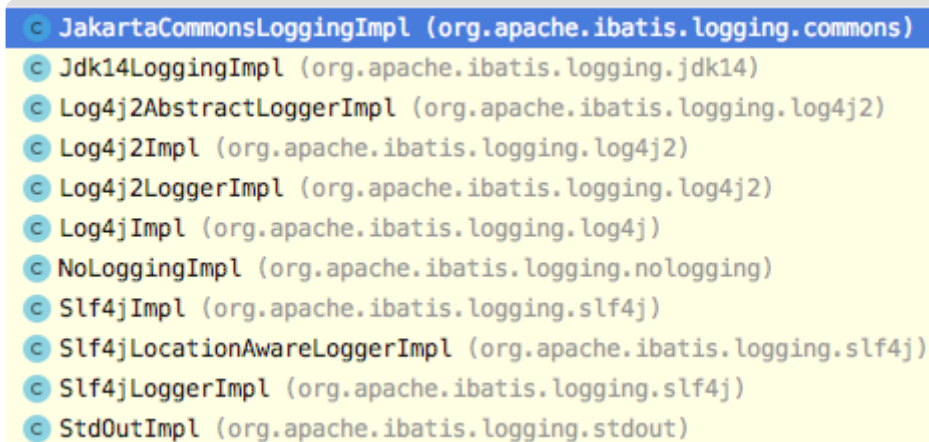
## Log：如何使用适配器模式来适配不同的日志框架？

□在适配器模式那节课中我们讲过，Slf4j 框架为了统一各个不同的日志框架（Log4j、JCL、Logback 等），提供了一套统一的日志接口。不过，MyBatis 并没有直接使用 Slf4j 提供的统一日志规范，而是自己又重复造轮子，定义了一套自己的日志访问接口。

 复制代码

```
1 public interface Log {
2     boolean isDebugEnabled();
3     boolean isTraceEnabled();
4     void error(String s, Throwable e);
5     void error(String s);
6     void debug(String s);
7     void trace(String s);
8     void warn(String s);
9 }
```

针对 Log 接口，MyBatis 还提供了各种不同的实现类，分别使用不同的日志框架来实现 Log 接口。




A screenshot of a code editor showing a list of logging implementation classes from the org.apache.ibatis.logging package. The list includes JakartaCommonsLoggingImpl, Jdk14LoggingImpl, Log4j2AbstractLoggerImpl, Log4j2Impl, Log4j2LoggerImpl, Log4jImpl, NoLoggingImpl, Slf4jImpl, Slf4jLocationAwareLoggerImpl, Slf4jLoggerImpl, and StdOutImpl. Each class name is preceded by a small circular icon.

- ◉ JakartaCommonsLoggingImpl (org.apache.ibatis.logging.commons)
- ◉ Jdk14LoggingImpl (org.apache.ibatis.logging.jdk14)
- ◉ Log4j2AbstractLoggerImpl (org.apache.ibatis.logging.log4j2)
- ◉ Log4j2Impl (org.apache.ibatis.logging.log4j2)
- ◉ Log4j2LoggerImpl (org.apache.ibatis.logging.log4j2)
- ◉ Log4jImpl (org.apache.ibatis.logging.log4j)
- ◉ NoLoggingImpl (org.apache.ibatis.logging.nologging)
- ◉ Slf4jImpl (org.apache.ibatis.logging.slf4j)
- ◉ Slf4jLocationAwareLoggerImpl (org.apache.ibatis.logging.slf4j)
- ◉ Slf4jLoggerImpl (org.apache.ibatis.logging.slf4j)
- ◉ StdOutImpl (org.apache.ibatis.logging.stdout)

这几个实现类的代码结构基本上一致。我把其中的 Log4jImpl 的源码贴到了这里。我们知道，在适配器模式中，传递给适配器构造函数的是被适配的类对象，而这里是 clazz（相当

于日志名称 name)，所以，从代码实现上来讲，它并非标准的适配器模式。但是，从应用场景上来看，这里确实又起到了适配的作用，是典型的适配器模式的应用场景。

 复制代码

```
1 import org.apache.ibatis.logging.Log;
2 import org.apache.log4j.Level;
3 import org.apache.log4j.Logger;
4
5 public class Log4jImpl implements Log {
6     private static final String FQCN = Log4jImpl.class.getName();
7     private final Logger log;
8
9     public Log4jImpl(String clazz) {
10         log = Logger.getLogger(clazz);
11     }
12
13     @Override
14     public boolean isDebugEnabled() {
15         return log.isDebugEnabled();
16     }
17
18     @Override
19     public boolean isTraceEnabled() {
20         return log.isTraceEnabled();
21     }
22
23     @Override
24     public void error(String s, Throwable e) {
25         log.log(FQCN, Level.ERROR, s, e);
26     }
27
28     @Override
29     public void error(String s) {
30         log.log(FQCN, Level.ERROR, s, null);
31     }
32
33     @Override
34     public void debug(String s) {
35         log.log(FQCN, Level.DEBUG, s, null);
36     }
37
38     @Override
39     public void trace(String s) {
40         log.log(FQCN, Level.TRACE, s, null);
41     }
42
43     @Override
44     public void warn(String s) {
45         log.log(FQCN, Level.WARN, s, null);
46     }
```

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

今天，我们讲到了 MyBatis 中用到的 8 种设计模式，它们分别是：建造者模式、工厂模式、模板模式、解释器模式、单例模式、装饰器模式、迭代器模式、适配器模式。加上上一节课中讲到的职责链和动态代理，我们总共讲了 10 种设计模式。

还是那句老话，你不需要记忆哪个类用到了哪个模式，因为不管你看多少遍，甚至记住并没有什么用。我希望你不仅仅只是把文章看了，更希望你能动手把 MyBatis 源码下载下来，自己去阅读一下相关的源码，锻炼自己阅读源码的能力。这比单纯看文章效果要好很多倍。

除此之外，从这两节课的讲解中，不知道你有没有发现，MyBatis 对很多设计模式的实现，都并非标准的代码实现，都做了比较多的自我改进。实际上，这就是所谓的灵活应用，只借鉴不照搬，根据具体问题针对性地去解决。

## 课堂讨论

今天我们提到，SqlSessionFactoryBuilder 跟 SqlSessionFactory 虽然名字后缀不同，但是设计思路一致，都是为了隐藏 SqlSession 的创建细节。从这一点上来看，命名有点不够统一。而且，我们还提到，SqlSessionFactoryBuilder 并非标准的建造者模式，SqlSessionFactory 也并非标准的工厂模式。对此你有什么看法呢？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

# 6月-7月课表抢先看

## 充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 88 | 开源实战五(中): 如何利用职责链与代理模式实现MyBatis Plugin?

下一篇 90 | 项目实战一: 设计实现一个支持各种算法的限流框架(分析)

### 精选留言 (9)

写留言



javaadu

2020-05-27

课后思考: 我理解这就是mybatis的代码写得烂, 不符合最小惊奇原则

1

9



Heaven

2020-05-27

设计思想比设计模式更重要,只要符合其设计的本意,没什么大不了的

...

2



Jxin

2020-05-27

前者隐藏的是初始化的细节，后者隐藏的选择的回话类型的细节。前者感觉建造者模式有点牵强，更像是初始化的配置类。后者工厂模式倒是没什么毛病，虽然不是标准的工厂模式。但我确实通过不同的选择，拿到了不同功能的对象。至于这些对象是同个父类的子类的对象，还是同个类不同参数的对象，我觉得只是实现方式的问题，场景上这个工厂模式并无不妥。

展开 ∨



👍 2



**小晏子**

2020-05-27

我认为非典型的建造者和工厂模式挺好的，我们并不是学院派，没必要追求典型的代码实现，既然这么做也可以简化开发并满足那些设计原则，那么就可以了。

展开 ∨



👍 2



**辣么大**

2020-05-29

这两个源码倒是很容易读。在github上看了他们10年前的这两个类的代码，重载了一些函数，但结构是一样的。我想应该是命名的习惯吧。当时也没考虑那么多。

展开 ∨



👍 1



**jiangjing**

2020-05-28

软件开发是个迭代的过程，一开始是足够好用，设计没有求全求美；后面则不断优化和增强功能。然后就是大家都熟悉怎么用了，有点小瑕疵但无关大局的代码就这么保留着吧，提供确定性



👍 1



**Mq**

2020-05-28

理解设计模式适用范围跟使用方式的也能理解这个代码，不理解的，也能通过名称理解代码的意图，思想到位就行了，也不一定每个人都理解得那么多规则

展开 ∨



👍 1



**jaryoung**

2020-05-27

个人还是喜欢大而全的玩意：  
引用文章的一句话：



实际上，这两个类的作用只不过是为了创建 `SqlSession` 对象，没有其他作用。所以，我更建议参照 Spring 的设计思路，把 `SqlSessionFactoryBuilder` 和 `SqlSessionFactory` 的逻辑，放到一个叫 “`ApplicationContext`” 的类中。让这个类来全权负责读入配置文件， ...  
展开 ▾



👍 1



**Geek\_3b1096**

2020-05-29

给我们读源码起很大帮助

展开 ▾

