89-开源实战五（下）：总结MyBatis框架中用到的10种设计模式

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，自己先去学习一下，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。

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

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：

```
public class SqlSessionFactoryBuilder {
  public SqlSessionFactory build(Reader reader) {
    return build(reader, null, null);
  }

  public SqlSessionFactory build(Reader reader, String environment) {
    return build(reader, environment, null);
  }
```

```
  public SqlSessionFactory build(Reader reader, Properties properties) {
    return build(reader, null, properties);
  }

  public SqlSessionFactory build(Reader reader, String environment, Properties properties) {
    try {
      XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);
      return build(parser.parse());
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error building SqlSession.", e);
    } finally {
      ErrorContext.instance().reset();
      try {
        reader.close();
      } catch (IOException e) {
        // Intentionally ignore. Prefer previous error.
      }
    }
  }

  public SqlSessionFactory build(InputStream inputStream) {
    return build(inputStream, null, null);
  }

  public SqlSessionFactory build(InputStream inputStream, String environment) {
    return build(inputStream, environment, null);
  }

  public SqlSessionFactory build(InputStream inputStream, Properties properties) {
    return build(inputStream, null, properties);
  }

  public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
    try {
      XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
      return build(parser.parse());
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error building SqlSession.", e);
    } finally {
      ErrorContext.instance().reset();
      try {
        inputStream.close();
      } catch (IOException e) {
        // Intentionally ignore. Prefer previous error.
      }
    }
  }

  public SqlSessionFactory build(Configuration config) {
    return new DefaultSqlSessionFactory(config);
  }
}
```

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。

```
public class SqlSessionFactoryBuilder {
  public SqlSessionFactory build(Reader reader);
  public SqlSessionFactory build(Reader reader, String environment);
```

```
  public SqlSessionFactory build(Reader reader, Properties properties);
  public SqlSessionFactory build(Reader reader, String environment, Properties properties);

  public SqlSessionFactory build(InputStream inputStream);
  public SqlSessionFactory build(InputStream inputStream, String environment);
  public SqlSessionFactory build(InputStream inputStream, Properties properties);
  public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties);

  // 上面所有的方法最终都调用这个方法
  public SqlSessionFactory build(Configuration config);
}
```

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。

```
public interface SqlSessionFactory {
  SqlSession openSession();
  SqlSession openSession(boolean autoCommit);
  SqlSession openSession(Connection connection);
  SqlSession openSession(TransactionIsolationLevel level);
  SqlSession openSession(ExecutorType execType);
  SqlSession openSession(ExecutorType execType, boolean autoCommit);
  SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level);
  SqlSession openSession(ExecutorType execType, Connection connection);
  Configuration getConfiguration();
}
```

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

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {
  private final Configuration configuration;
  public DefaultSqlSessionFactory(Configuration configuration) {
    this.configuration = configuration;
  }

  @Override
  public SqlSession openSession() {
    return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, false);
  }

  @Override
  public SqlSession openSession(boolean autoCommit) {
    return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, autoCommit);
  }

  @Override
  public SqlSession openSession(ExecutorType execType) {
    return openSessionFromDataSource(execType, null, false);
  }

  @Override
  public SqlSession openSession(TransactionIsolationLevel level) {
    return openSessionFromDataSource(configuration.getDefaultExecutorType(), level, false);
  }

  @Override
  public SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level) {
    return openSessionFromDataSource(execType, level, false);
  }

  @Override
  public SqlSession openSession(ExecutorType execType, boolean autoCommit) {
    return openSessionFromDataSource(execType, null, autoCommit);
  }

  @Override
  public SqlSession openSession(Connection connection) {
    return openSessionFromConnection(configuration.getDefaultExecutorType(), connection);
  }

  @Override
  public SqlSession openSession(ExecutorType execType, Connection connection) {
    return openSessionFromConnection(execType, connection);
  }

  @Override
  public Configuration getConfiguration() {
    return configuration;
  }

  private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, bool
    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();
    }
  }

  private SqlSession openSessionFromConnection(ExecutorType execType, Connection connection) {
    try {
      boolean autoCommit;
      try {
        autoCommit = connection.getAutoCommit();
      } catch (SQLException e) {
        // Failover to true, as most poor drivers
        // or databases won't support transactions
        autoCommit = true;
      }
      final Environment environment = configuration.getEnvironment();
      final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment);
      final Transaction tx = transactionFactory.newTransaction(connection);
      final Executor executor = configuration.newExecutor(tx, execType);
      return new DefaultSqlSession(configuration, executor, autoCommit);
    } catch (Exception e) {
      throw ExceptionFactory.wrapException("Error opening session.  Cause: " + e, e);
    } finally {
      ErrorContext.instance().reset();
    }
  }
  //...省略部分代码...
}
```

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，而这里只不过是通过传递进不同的参数，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Congfiguration，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。

```
public abstract class BaseExecutor implements Executor {
  //...省略其他无关代码...

  @Override
  public int update(MappedStatement ms, Object parameter) throws SQLException {
    ErrorContext.instance().resource(ms.getResource()).activity("executing an update").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    clearLocalCache();
    return doUpdate(ms, parameter);
  }

  public List<BatchResult> flushStatements(boolean isRollBack) throws SQLException {
    if (closed) {
      throw new ExecutorException("Executor was closed.");
    }
    return doFlushStatements(isRollBack);
  }

  private <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHa
    List<E> list;
    localCache.putObject(key, EXECUTION_PLACEHOLDER);
    try {
      list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
    } finally {
      localCache.removeObject(key);
    }
    localCache.putObject(key, list);
    if (ms.getStatementType() == StatementType.CALLABLE) {
      localOutputParameterCache.putObject(key, parameter);
    }
    return list;
  }

  @Override
  public <E> Cursor<E> queryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds) throws SQLExc
    BoundSql boundSql = ms.getBoundSql(parameter);
    return doQueryCursor(ms, parameter, rowBounds, boundSql);
  }

  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, ResultH

  protected abstract <E> Cursor<E> doQueryCursor(MappedStatement ms, Object parameter, RowBounds rowBounds,
}
```

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。

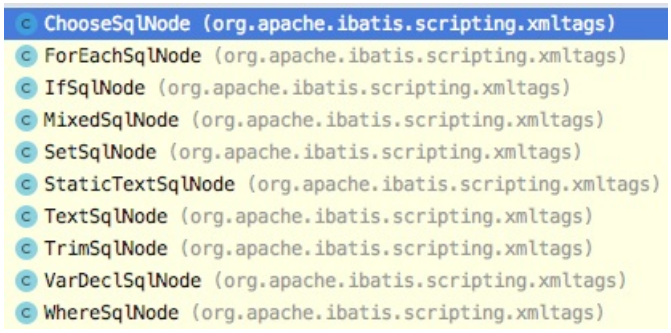
```
<update id="update" parameterType="com.xzg.cd.a89.User"
   UPDATE user
   <trim prefix="SET" prefixOverrides=",">
       <if test="name != null and name != ''">
           name = #{name}
       </if>
       <if test="age != null and age != ''">
           , age = #{age}
       </if>
       <if test="birthday != null and birthday != ''">
           , birthday = #{birthday}
       </if>
   </trim>
   where id = ${id}
</update>
```

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释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：

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

对于不同的语法小单元，MyBatis定义不同的SqlNode实现类。



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。

```
public class ErrorContext {
  private static final String LINE_SEPARATOR = System.getProperty("line.separator","\n");
  private static final ThreadLocal<ErrorContext> LOCAL = new ThreadLocal<ErrorContext>();

  private ErrorContext stored;
  private String resource;
  private String activity;
  private String object;
  private String message;
  private String sql;
  private Throwable cause;

  private ErrorContext() {
  }

  public static ErrorContext instance() {
    ErrorContext context = LOCAL.get();
    if (context == null) {
      context = new ErrorContext();
      LOCAL.set(context);
    }
    return context;
  }
}
```

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。

```
public interface Cache {
  String getId();
  void putObject(Object key, Object value);
  Object getObject(Object key);
  Object removeObject(Object key);
  void clear();
  int getSize();
  ReadWriteLock getReadWriteLock();
}
```

```
public class PerpetualCache implements Cache {
  private final String id;
  private Map<Object, Object> cache = new HashMap<Object, Object>();

  public PerpetualCache(String id) {
    this.id = id;
  }

  @Override
  public String getId() {
    return id;
  }

  @Override
  public int getSize() {
    return cache.size();
  }

  @Override
  public void putObject(Object key, Object value) {
    cache.put(key, value);
  }

  @Override
  public Object getObject(Object key) {
    return cache.get(key);
  }

  @Override
  public Object removeObject(Object key) {
    return cache.remove(key);
  }

  @Override
  public void clear() {
    cache.clear();
  }

  @Override
  public ReadWriteLock getReadWriteLock() {
    return null;
  }
  //省略部分代码...
}
```

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。

```
public class LruCache implements Cache {
  private final Cache delegate;
  private Map<Object, Object> keyMap;
  private Object eldestKey;

  public LruCache(Cache delegate) {
    this.delegate = delegate;
    setSize(1024);
  }
```

```
  @Override
  public String getId() {
    return delegate.getId();
  }

  @Override
  public int getSize() {
    return delegate.getSize();
  }

  public void setSize(final int size) {
    keyMap = new LinkedHashMap<Object, Object>(size, .75F, true) {
      private static final long serialVersionUID = 4267176411845948333L;

      @Override
      protected boolean removeEldestEntry(Map.Entry<Object, Object> eldest) {
        boolean tooBig = size() > size;
        if (tooBig) {
          eldestKey = eldest.getKey();
        }
        return tooBig;
      }
    };
  }

  @Override
  public void putObject(Object key, Object value) {
    delegate.putObject(key, value);
    cycleKeyList(key);
  }

  @Override
  public Object getObject(Object key) {
    keyMap.get(key); //touch
    return delegate.getObject(key);
  }

  @Override
  public Object removeObject(Object key) {
    return delegate.removeObject(key);
  }

  @Override
  public void clear() {
    delegate.clear();
    keyMap.clear();
  }

  @Override
  public ReadWriteLock getReadWriteLock() {
    return null;
  }

  private void cycleKeyList(Object key) {
    keyMap.put(key, key);
    if (eldestKey != null) {
      delegate.removeObject(eldestKey);
      eldestKey = null;
    }
  }
}
```

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：

```
// person[0].birthdate.year 会被分解为3个PropertyTokenizer对象。其中，第一个PropertyTokenizer对象的各个属性值如注释所
public class PropertyTokenizer implements Iterator<PropertyTokenizer> {
  private String name; // person
  private final String indexedName; // person[0]
  private String index; // 0
  private final String children; // birthdate.year

  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 o
  }
}
```

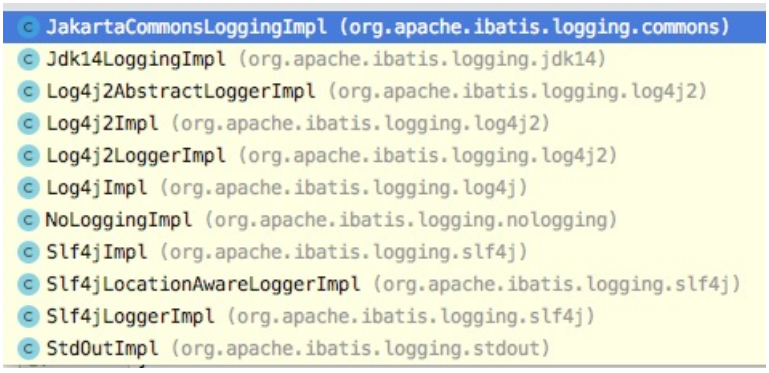
实际上，PropertyTokenizer类也并非标准的迭代器类。它将配置的解析、解析之后的元素、迭代器，这三部分本该放到三个类中的代码，都耦合在一个类中，所以看起来稍微有点难懂。不过，这样做的好处是能够做到惰性解析。我们不需要事先将整个配置，解析成多个PropertyTokenizer对象。只有当我们在调用next()函数的时候，才会解析其中部分配置。

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。

```
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);
}
```

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。



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。

```
import org.apache.ibatis.logging.Log;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class Log4jImpl implements Log {
  private static final String FQCN = Log4jImpl.class.getName();
```

```
  private final 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);
  }
}
```

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。

精选留言：

- javaadu 2020-05-27 08:38:00
  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 [9赞]

- Jxin 2020-05-27 10:03:54
  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。 [2赞]

- 小晏子 2020-05-27 09:10:44
  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。 [2赞]

- jiangjing 2020-05-28 08:39:04
  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赞]

- Mq 2020-05-28 08:06:16
  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 [1赞]

- Heaven 2020-05-27 19:54:33
  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 [1赞]

- jaryoung 2020-05-27 10:50:56
  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“Application Context”的类中。让这个类来全权负责读入配置文件，创建 Congfiguration，生成 SqlSession。

  修改前：
  public class MyBatisDemo {
  public static void main(String[] args) throws IOException {
  Reader reader = Resources.getResourceAsReader("mybatis.xml");
  SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader);
  SqlSession session = sessionFactory.openSession();
  UserMapper userMapper = session.getMapper(UserMapper.class);
  UserDo userDo = userMapper.selectById(8);

```
//...
}

}
```

修改后：

```
public class MyBatisDemo {
public static void main(String[] args) throws IOException {
ApplicationContext applicationContext = new ApplicationContext("test-config.xml");
SqlSession session = applicationContext.openSession();
UserMapper userMapper = session.getMapper(UserMapper.class);
UserDo userDo = userMapper.selectById(8);
//...
}

}
```

使用越简单，背后逻辑越复杂，也可能是封装的必要性吧。

```
public class ApplicationContext {

private Reader reader;

public ApplicationContext(String path) {
try {
reader = Resources.getResourceAsReader(path);
} catch (IOException e) {
e.printStackTrace();
}
Assert.that(reader == null, "reader can't null");
}

public ApplicationContext() {
this("mybatis-config.xml");
}
public SqlSession openSession() {
SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(reader);
return sessionFactory.openSession();
}
}
```

[1赞]

- Geek_3b1096 2020-05-29 06:01:53
  给我们读源码起很大帮助