# MyBatis基础模块-日志模块

首先日志在我们开发过程中占据了一个非常重要的地位,是开发和运维管理之间的桥梁,在Java中的日志框架也非常多,Log4j,Log4j2,Apache Commons Log,java.util.logging,slf4j等,这些工具对外的接口也都不尽相同,为了统一这些工具,MyBatis定义了一套统一的日志接口供上层使用。首先大家对于适配器模式要了解下哦。

# 1, Log

Log接口中定义了四种日志级别,相比较其他的日志框架的多种日志级别显得非常的精简,但也能够满足大多数常见的使用了

```
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);
}
```

# 2. LogFactory

LogFactory工厂类负责创建日志组件适配器,

```
* Marker to be used by logging implementations that support markers.
                               28
∨ 🗎 main
                               29
                                        public static final String MARKER = "MYBATIS";
    org.apache.ibatis
annotations
binding
                                        private static Constructor<? extends Log> logConstructor;
     builder
     DII cache
                                        static {
                                          // 按序加载对应的日志组件,从上往下加载,上面的成功了,下面的就不会在加载了
                               34
      m exceptions
     m executor
                                          /**
                               35
                               36
                                           * tryImplementation(LogFactory::useSlf4jLogging); 等价于
                               37
                                           * tryImplementation(new Runnable(){
      logging
                               38
                                               void run(){
       jdbc jdk14
                               39
                                                  useSlf4jLogging();
       log4j
                               40
       □ log4j2
                                           * })
       slf4j
                               42
                                           */
       Log
                               43
                                           tryImplementation(LogFactory::useSlf4jLogging);
        LogException
       C LogFactory
                               44
                                          tryImplementation(LogFactory::useCommonsLogging);
       apackage-info.jav
                               45
                                          tryImplementation(LogFactory::useLog4J2Logging);
      m parsing
                                        tryImplementation(LogFactory::useLog4JLogging);
                               47
                                        tryImplementation(LogFactory::useJdkLogging);
      scripting
                               48
                                          tryImplementation(LogFactory::useNoLogging);
      session
      test
                               49
      🚜 package-info.java
                                        private LogFactorv() {
```

在LogFactory类加载时会执行其静态代码块,其逻辑是按序加载并实例化对应日志组件的适配器,然后使用LogFactory.logConstructor这个静态字段,记录当前使用的第三方日志组件的适配器。具体代码如下,每个方法都比较简单就不一一赘述了。

### 3、日志应用

那么在MyBatis系统启动的时候日志框架是如何选择的呢?首先我们在全局配置文件中我们可以设置对应的日志类型选择

```
<?xml version="1.0" encoding="UTF-8" ?>
                             <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "http://mybatis</pre>
1 01-ORM
                       2
                       3
                            <configuration>
03-MyBatisExample
 sro
                       4
                       5
                                 properties resource="db.properties"
    ∨ 🛅 com.boge
                       6
                                 <settings>
      > 🛅 db
                      7
                                     <!-- 打印查询语句 ctrl+al
      > 🛅 log
                             setting name="logImpl" value="STDOUT_LOGGING"
                      8
      > 🛅 mapper
                     9
    > mapper
                                     <!-- 控制全局缓存(二级缓存), 默认 true-->
      db.properties
mybatis-config.xml
                     11
                                     <setting name="cacheEnabled" value="true"/>

∨ Image test

   ✓ i java
✓ com.boge.test
                                     <!-- 延迟加载的全局开关。当开启时,所有关联对象都会延迟加载。默认 false -->
       d Test1
                                     <setting name="lazyLoadingEnabled" value="true"/>
> iii target
m pom.xml
                                     <!-- 当开启时,任何方法的调用都会加载该对象的所有属性。默认 false,可通过select标签的 j
                                     <setting name="aggressiveLazyLoading" value="false"/>
05-MyBatisExt
                     16
                              <!-- Mubatis 创建具有延迟加载能力的对象所用到的代理工具,默认JAVASSIST --> configuration ) settings ) setting
```

这个"STDOUT\_LOGGING"是怎么来的呢?在Configuration的构造方法中其实是设置的各个日志实现的 别名的

```
⊕ <u>∓</u> ÷ −
                                     × Simple 
                                       201
                                                                          typeAliasRegistry.registerAlias(alias: "DB_VENDOR", VendorDatabaseIdProvider.class);
                                                                          typeAliasRegistry.registerAlias( alias: "XML", XMLLanguageDriver.class);
                                                                         typeAliasRegistry.registerAlias( alias: "RAW", RawLanguageDriver.class);
                                                                         typeAliasRegistry.registerAlias( alias: "SLF4J", Slf4jImpl.class);
                                                                          typeAliasRegistry.registerAlias( alias: "COMMONS_LOGGING", JakartaCommonsLoggingImpl.class);
                                                                          typeAliasRegistry.registerAlias( alias: "LOG4J", Log4jImpl.class);
                                                                         typeAliasRegistry.registerAlias( alias: "LOG4J2", Log4j2Impl.class);
                                                                        typeAliasRegistry.registerAlias(alias: "JDK_LOGGING", Jdk14LoggingImpl.class);
typeAliasRegistry.registerAlias(alias: "STDOUT_LOGGING", StdOutImpl.class);
                                                                            typeAliaskegistry.registerAlias( alias: "NU_LUGGING", NoLoggingimpl.class);
                                                                          typeAliasRegistry.registerAlias( alias: "CGLIB", CglibProxyFactory.class);
                                                                          {\tt typeAliasRegistry.registerAlias(alias: "JAVASSIST", JavassistProxyFactory.class);}
                                                                          // LanguageDriver 完成SQL配置信息到SqlSource对象的转
                                                                          // XMLLanguageDriver为XML语言驱动
                               need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate
```

#### 然后在解析全局配置文件的时候就会处理日志的设置

```
🔋 SimpleExecutor.java 🗴 🚯 StatementHandler.java 🗶 🔞 BaseStatementHandler.java 🗶 🌑 PreparedStatementHandler.java 🗶 🚇 mybatis-config.xml 🗶 🌑 Configuration.java 🗶 🖫 XMLConfigBuilder.java 🔻 🐞 RoutingS 🗸 Maven
                 propertiesElement(root.evalNode("properties"));
                                                                                                                                          > Profiles
                 // 解析 settings 标签 返回的结果就是 settings 中定义的信息 自定义的属性
                                                                                                                                            Profiles
1 01-ORM
2 02-Hiber
3 03-MyBa
3 04-MyBa
3 05-MyBa
                 Properties settings = settingsAsProperties(root.evalNode("settings"));
                 loadCustomVfs(settings);
                                                                                                                                            /// MyBatis\
                 loadCustomLogImpl(settings):
                 typeAliasesElement(root.evalNode("typeAliases"));
                 pluginElement(root.evalNode("plugins"));
                 // 用于创建对象
                 objectFactoryElement(root.evalNode("objectFactory"));
                 // 用于对对象进行加工
 128
                 objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
                 // 反射工具箱
                 reflectorFactoryElement(root.evalNode("reflectorFactory"));
its passed: 1 of 1 test – 1 sec 559 m
```

#### 进入方法

```
private void loadCustomLogImpl(Properties props) {
    // 获取 logImpl设置的 日志 类型
    Class<? extends Log> logImpl = resolveClass(props.getProperty("logImpl"));
    // 设置日志
    configuration.setLogImpl(logImpl);
}
```

#### 进入setLogImpl方法中

```
public void setLogImpl(Class<? extends Log> logImpl) {
   if (logImpl != null) {
     this.logImpl = logImpl; // 记录日志的类型
     // 设置 适配选择
     LogFactory.useCustomLogging(this.logImpl);
   }
}
```

#### 再进入useCustomLogging方法

```
public static synchronized void useCustomLogging(Class<? extends Log> clazz) {
   setImplementation(clazz);
}
```

```
private static void setImplementation(Class<? extends Log> implClass) {
      // 获取指定适配器的构造方法
     Constructor<? extends Log> candidate =
implClass.getConstructor(String.class);
      // 实例化适配器
     Log log = candidate.newInstance(LogFactory.class.getName());
      if (log.isDebugEnabled()) {
       log.debug("Logging initialized using '" + implClass + "' adapter.");
      }
      // 初始化 logConstructor 字段
      logConstructor = candidate;
    } catch (Throwable t) {
      throw new LogException("Error setting Log implementation. Cause: " + t,
t);
    }
  }
```

这就关联上了我们前面在LogFactory中看到的代码,启动测试方法看到的日志也和源码中的对应上来了,还有就是我们自己设置的会覆盖掉默认的sl4i日志框架的配置

```
private static void setimplementation(Class<? extends Log> implClass) {
                             try {
                              // 获取指定适配器的构造方法
                              Constructor<? extends Log> candidate = implClass.getConstructor(String.class);
                              Log log = candidate.newInstance(LogFactory.class.getName());
                              if (log.isDebt
                              log.debug s: "Logging initialized using '" + implClass + "' adapter.");
                            }
                              // 初始化 logConstructor 字段
               119
                             logConstructor = candidate;
                 120
                            } catch (Throwable t) {
                              throw new LogException("Error setting Log implementation. Cause: " + t, t);
              125
126
D:\software\java\jdk8\bin\java.exe ...
              log4j:WARN No appenders could be found for logger (rg.apache.ibatis.logging.LogFactory).
              log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logqing.apache.org/log4j/1.2/fag.html#noconfig for more
             Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
              Class not found: org.jboss.vfs.VFS
              JBoss 6 VFS API is not available in this environment.
              Class not found: org.jboss.vfs.VirtualFile
```

## 4、JDBC 日志

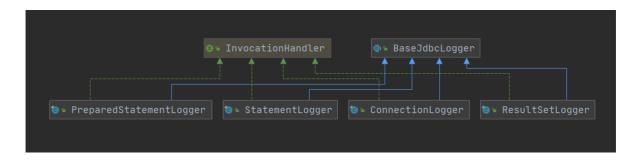
当我们开启了 STDOUT的日志管理后,当我们执行SQL操作时我们发现在控制台中可以打印出相关的日志信息

```
// 获取指定适配器的构造方法
               Constructor<? extends Log> candidate = implClass.getConstructor(String.class);
              Log log = candidate.newInstance(LogFactory.class.getName());
           if (log.isDebugEnabled()) {
                lon dahun( s. "Lonninn initializad usinn !" + imnlflass + "! adantan ").
need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server
Created connection 1995619265.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@76f2bbc1]
==> Preparing: select * from t_user
==> Parameters:
type = interface java.util.List
<== Columns: id, user_name, real_name, password, age, d_id, i_id
          Row: 1, admin, 管理员, 111, 22, 1001, null
type = class com.boge.pojo.User
<==
         Row: 666, hibernate-1, 持久层框架, null, null, null, null
type = class com.boge.pojo.User
         Row: 668, hibernate-1, 持久层框架, null, null, null, null
type = class com.boge.pojo.User
        Total: 3
list.size() = 3
list.size() = 3
```

那这些日志信息是怎么打印出来的呢?原来在MyBatis中的日志模块中包含了一个jdbc包,它并不是将日志信息通过jdbc操作保存到数据库中,而是通过JDK动态代理的方式,将JDBC操作通过指定的日志框架打印出来。下面我们就来看看它是如何实现的。

### 4.1 BaseJdbcLogger

BaseJdbcLogger是一个抽象类,它是jdbc包下其他Logger的父类。继承关系如下



从图中我们也可以看到4个实现都实现了InvocationHandler接口。属性含义如下

```
// 记录 PreparedStatement 接口中定义的常用的set*() 方法
protected static final Set<String> SET_METHODS;
// 记录了 Statement 接口和 PreparedStatement 接口中与执行SQL语句有关的方法
protected static final Set<String> EXECUTE_METHODS = new HashSet<>();

// 记录了PreparedStatement.set*() 方法设置的键值对
private final Map<Object, Object> columnMap = new HashMap<>();
// 记录了PreparedStatement.set*() 方法设置的键 key
private final List<Object> columnNames = new ArrayList<>();
// 记录了PreparedStatement.set*() 方法设置的值 Value
private final List<Object> columnValues = new ArrayList<>();

protected final Log statementLog;// 用于日志输出的Log对象
protected final int queryStack; // 记录了SQL的层数,用于格式化输出SQL
```

### 4.2 ConnectionLogger

ConnectionLogger的作用是记录数据库连接相关的日志信息,在实现中是创建了一个Connection的代理对象,在每次Connection操作的前后我们都可以实现日志的操作。

```
public final class ConnectionLogger extends BaseJdbcLogger implements
InvocationHandler {
  // 真正的Connection对象
  private final Connection connection;
  private ConnectionLogger(Connection conn, Log statementLog, int queryStack) {
    super(statementLog, queryStack);
    this.connection = conn;
  }
  @override
  public Object invoke(Object proxy, Method method, Object[] params)
      throws Throwable {
    try {
      // 如果是调用从Object继承过来的方法,就直接调用 toString,hashCode,equals等
      if (Object.class.equals(method.getDeclaringClass())) {
        return method.invoke(this, params);
      }
      // 如果调用的是 prepareStatement方法
      if ("prepareStatement".equals(method.getName())) {
        if (isDebugEnabled()) {
          debug(" Preparing: " + removeBreakingWhitespace((String) params[0]),
true);
        }
        // 创建 PreparedStatement
        PreparedStatement stmt = (PreparedStatement) method.invoke(connection,
params);
        // 然后创建 PreparedStatement 的代理对象 增强
        stmt = PreparedStatementLogger.newInstance(stmt, statementLog,
queryStack);
        return stmt;
        // 同上
      } else if ("prepareCall".equals(method.getName())) {
        if (isDebugEnabled()) {
          debug(" Preparing: " + removeBreakingWhitespace((String) params[0]),
true);
        PreparedStatement stmt = (PreparedStatement) method.invoke(connection,
params);
        stmt = PreparedStatementLogger.newInstance(stmt, statementLog,
queryStack);
        return stmt;
        // 同上
      } else if ("createStatement".equals(method.getName())) {
        Statement stmt = (Statement) method.invoke(connection, params);
        stmt = StatementLogger.newInstance(stmt, statementLog, queryStack);
        return stmt;
```

```
} else {
        return method.invoke(connection, params);
   } catch (Throwable t) {
      throw ExceptionUtil.unwrapThrowable(t);
   }
  }
  /**
  * Creates a logging version of a connection.
  * @param conn - the original connection
   * @return - the connection with logging
  */
  public static Connection newInstance(Connection conn, Log statementLog, int
queryStack) {
   InvocationHandler handler = new ConnectionLogger(conn, statementLog,
queryStack);
   ClassLoader cl = Connection.class.getClassLoader();
    // 创建了 Connection的 代理对象 目的是 增强 Connection对象 给他添加了日志功能
    return (Connection) Proxy.newProxyInstance(cl, new Class[]
{Connection.class}, handler);
  }
  /**
  * return the wrapped connection.
  * @return the connection
  */
  public Connection getConnection() {
   return connection;
  }
}
```

其他几个xxxxLogger的实现和ConnectionLogger几乎是一样的就不在次赘述了,请自行观看。

### 4.3 应用实现

```
在实际处理的时候,日志模块是如何工作的,我们来看看。
```

在我们要执行SQL语句前需要获取Statement对象,而Statement对象是通过Connection获取的,所以 我们在SimpleExecutor中就可以看到相关的代码

```
private Statement prepareStatement(StatementHandler handler, Log statementLog)
throws SQLException {
    Statement stmt;
    Connection connection = getConnection(statementLog);
    // 获取 Statement 对象
    stmt = handler.prepare(connection, transaction.getTimeout());
    // 为 Statement 设置参数
    handler.parameterize(stmt);
    return stmt;
}
```

```
protected Connection getConnection(Log statementLog) throws SQLException {
   Connection connection = transaction.getConnection();
   if (statementLog.isDebugEnabled()) {
        // 创建Connection的日志代理对象
        return ConnectionLogger.newInstance(connection, statementLog, queryStack);
    } else {
        return connection;
    }
}
```

#### 在进入到handler.prepare方法中

```
@override
  protected Statement instantiateStatement(Connection connection) throws
SQLException {
    String sql = boundSql.getSql();
   if (mappedStatement.getKeyGenerator() instanceof Jdbc3KeyGenerator) {
      String[] keyColumnNames = mappedStatement.getKeyColumns();
      if (keyColumnNames == null) {
        return connection.prepareStatement(sql,
PreparedStatement.RETURN_GENERATED_KEYS);
     } else {
        // 在执行 prepareStatement 方法的时候会进入进入到ConnectionLogger的invoker方法
中
        return connection.prepareStatement(sql, keyColumnNames);
      }
    } else if (mappedStatement.getResultSetType() == ResultSetType.DEFAULT) {
      return connection.prepareStatement(sql);
    } else {
      return connection.prepareStatement(sql,
mappedStatement.getResultSetType().getValue(), ResultSet.CONCUR_READ_ONLY);
    }
  }
```

```
| December | Code Analyze Selecture Baid Run | Code NCS Window Belly | Majesticities | December |
```

```
@Override
public <E> List<E> query(Statement statement, ResultHandler resultHandler)
throws SQLException {
    PreparedStatement ps = (PreparedStatement) statement;
    // 到了JDBC的流程
    ps.execute(); // 本质上 ps 也是 日志代理对象
    // 处理结果集
    return resultSetHandler.handleResultSets(ps);
}
```

如果是查询操作,后面的ResultSet结果集操作,其他是也通过ResultSetLogger来处理的,前面的清楚了,后面的就很容易的。