Mybatis 框架源码10种设计模式分析

作者：小傅哥
博客：https://bugstack.cn

> 沉淀、分享、成长，让自己和他人都能有所收获！😄

一、前言：小镇卷码家

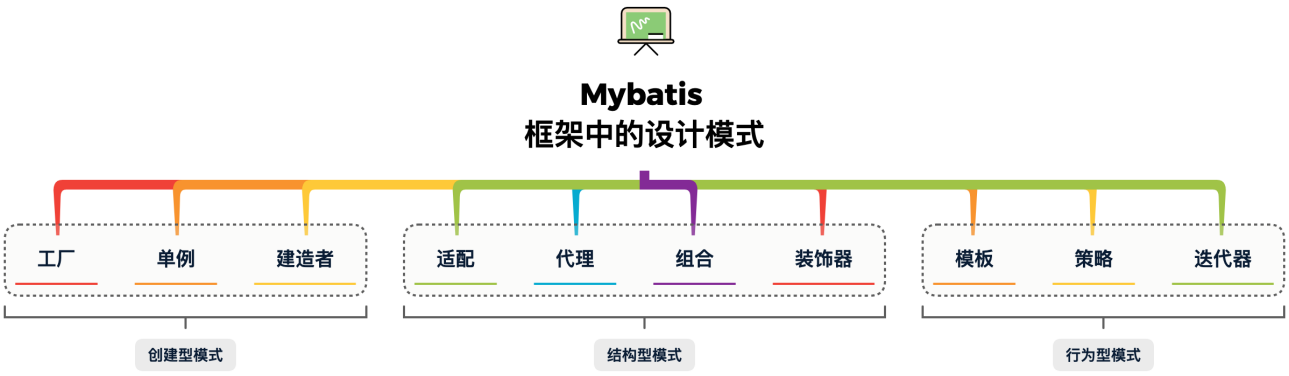
总有不少研发伙伴问小傅哥：“为什么学设计模式、看框架源码、补技术知识，就一个普通的业务项目，会造飞机不也是天天写CRUD吗？”

你说的没错，但你天天写CRUD，你觉得 **烦不？ 慌不？** 是不是既担心自己没有得到技术成长，也害怕将来没法用这些都是CRUD的项目去参加；述职、晋升、答辩，甚至可能要被迫面试时，自己手里一点干货也没有的情况。

所以你/我作为一个**小镇卷码家**，当然要扩充自己的知识储备，否则`架构，架构思维不懂`、`设计，设计模式不会`、`源码、源码学习不深`，最后就用一堆CRUD写简历吗？

二、源码：学设计模式

在 Mybatis 两万多行的框架源码实现中，使用了大量的设计模式来解耦工程架构中面对复杂场景的设计，这些是设计模式的巧妙使用才是整个框架的精华，**这也是小傅哥喜欢卷源码的重要原因**。经过小傅哥的整理有如下10种设计模式的使用，如图所示



讲道理，如果只是把这10种设计模式背下来，等着下次面试的时候拿出来说一说，虽然能有点帮助，不过这种学习方式就真的算是把路走窄了。就像你每说一个设计模式，能联想到这个设计模式在Mybatis的框架中，体现到哪个流程中的源码实现上了吗？这个源码实现的思路能不能用到你的业务流程开发里？**别总说你的流程简单，用不上设计模式！难到因为有钱、富二代，就不考试吗？**🤔

好啦，不扯淡了，接下来小傅哥就以《手写Mybatis：渐进式源码实践》的学习，给大家列举出这10种设计模式，在Mybatis框架中都体现在哪里了！

- 学习手册：`关注公众号【bugstack虫洞栈】回复【Mybatis】`
- 源码仓库：https://gitcode.net/KnowledgePlanet/doc/-/wikis/home

三、类型：创建型模式

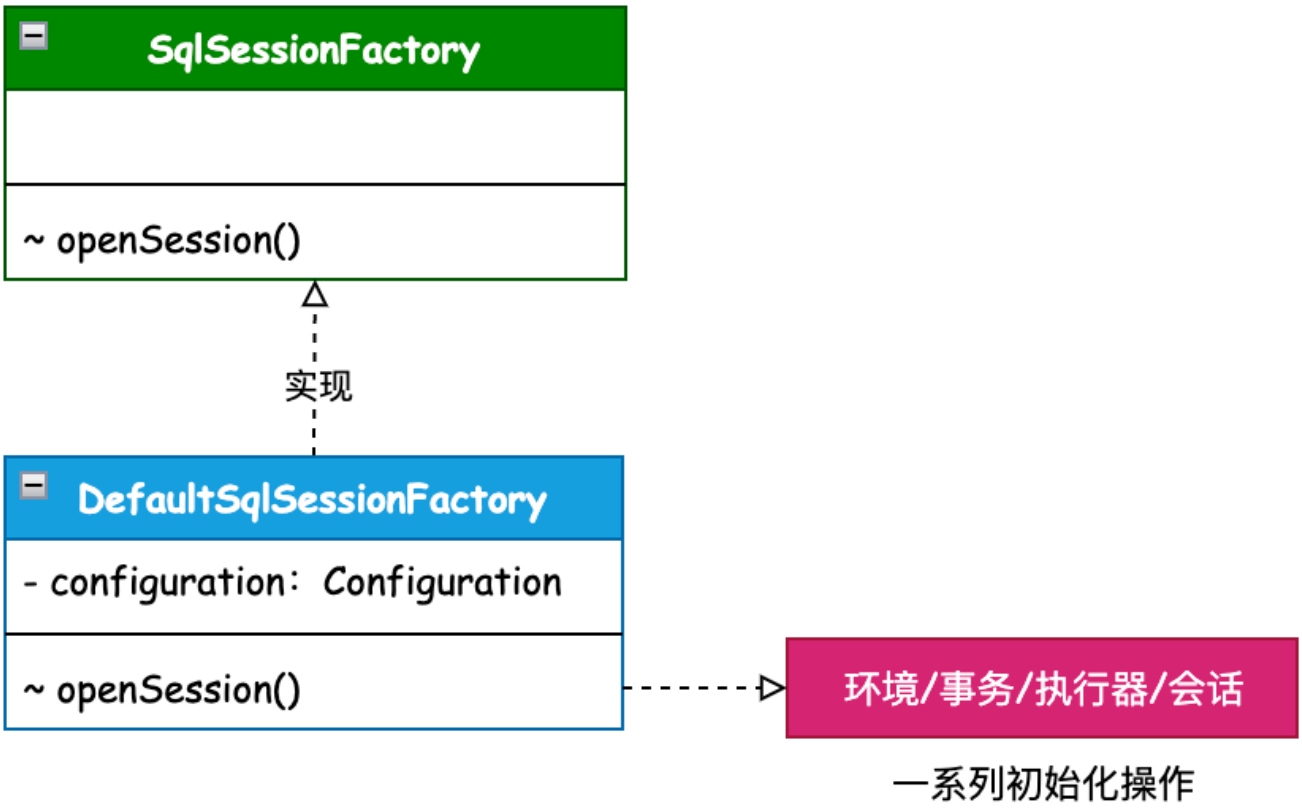
1. 工厂模式

源码详见： `cn.bugstack.mybatis.session.SqlSessionFactory`

```
public interface SqlSessionFactory {

   SqlSession openSession();

}
```

源码详见： `cn.bugstack.mybatis.session.defaults.DefaultSqlSessionFactory`

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {

    private final Configuration configuration;

    public DefaultSqlSessionFactory(Configuration configuration) {
        this.configuration = configuration;
    }

    @Override
    public SqlSession openSession() {
        Transaction tx = null;
        try {
            final Environment environment = configuration.getEnvironment();
            TransactionFactory transactionFactory =
environment.getTransactionFactory();
            tx =
transactionFactory.newTransaction(configuration.getEnvironment().getDataSource(),
TransactionIsolationLevel.READ_COMMITTED, false);
            // 创建执行器
            final Executor executor = configuration.newExecutor(tx);
            // 创建DefaultSqlSession
            return new DefaultSqlSession(configuration, executor);
        } catch (Exception e) {
            try {
                assert tx != null;
                tx.close();
            } catch (SQLException ignore) {
            }
            throw new RuntimeException("Error opening session.  Cause: " + e);
        }
    }

}
```



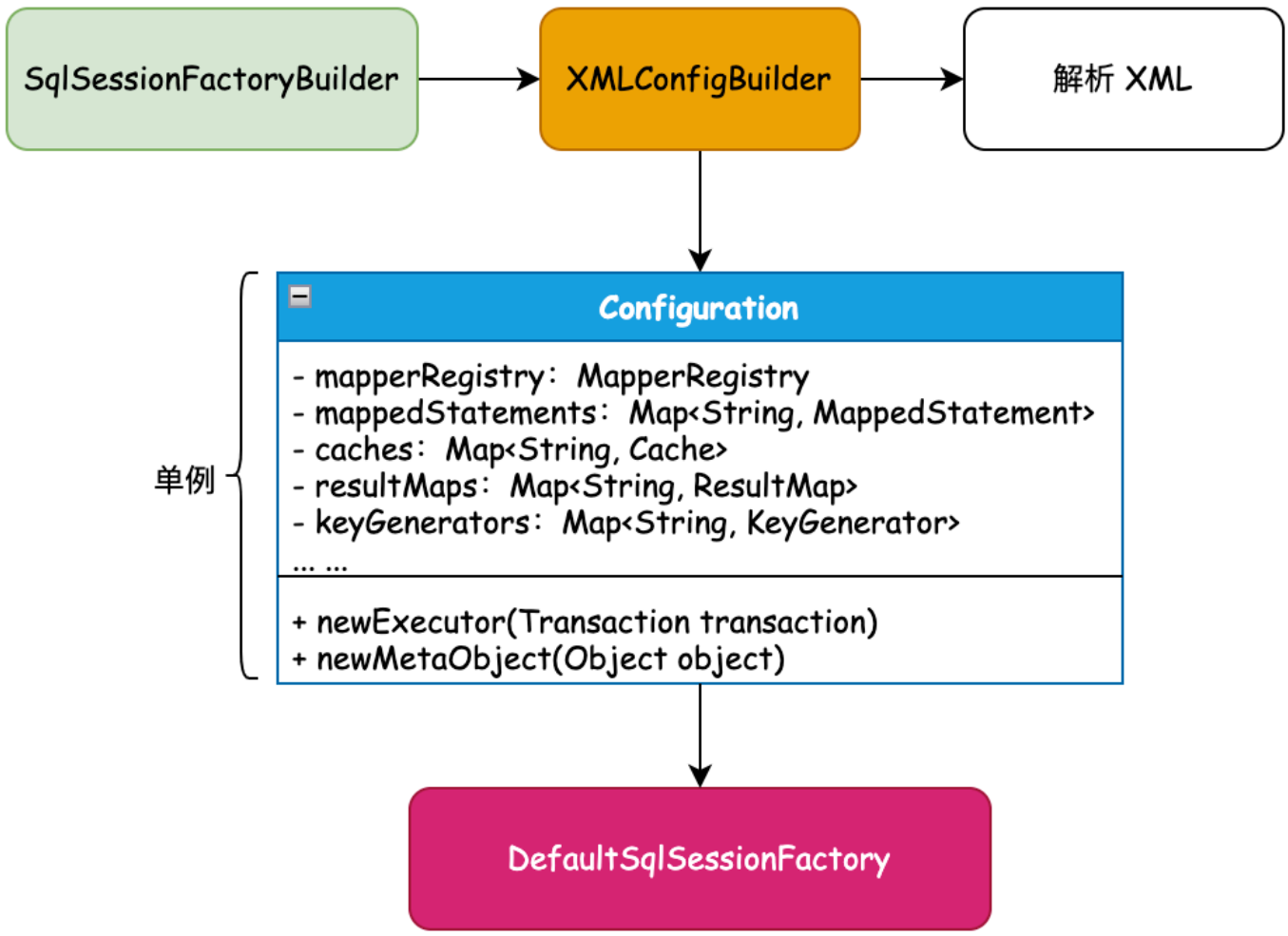
- **工厂模式**：简单工厂，是一种创建型设计模式，其在父类中提供一个创建对象的方法，允许子类决定实例对象的类型。
- **场景介绍**：`SqlSessionFactory` 是获取会话的工厂，每次我们使用 Mybatis 操作数据库的时候，都会开启一个新的会话。在会话工厂的实现中负责获取数据源环境配置信息、构建事务工厂、创建操作SQL的执行器，并最终返回会话实现类。
- **同类设计**：`SqlSessionFactory`、`ObjectFactory`、`MapperProxyFactory`、`DataSourceFactory`

2. 单例模式

源码详见：`cn.bugstack.mybatis.session.Configuration`

```
public class Configuration {

    // 缓存机制，默认不配置的情况是 SESSION
    protected LocalCacheScope localCacheScope = LocalCacheScope.SESSION;

    // 映射注册机
    protected MapperRegistry mapperRegistry = new MapperRegistry(this);

    // 映射的语句，存在Map里
    protected final Map<String, MappedStatement> mappedStatements = new HashMap<>();
    // 缓存,存在Map里
    protected final Map<String, Cache> caches = new HashMap<>();
    // 结果映射，存在Map里
    protected final Map<String, ResultMap> resultMaps = new HashMap<>();
    protected final Map<String, KeyGenerator> keyGenerators = new HashMap<>();
```

```
    // 插件拦截器链
    protected final InterceptorChain interceptorChain = new InterceptorChain();

    // 类型别名注册机
    protected final TypeAliasRegistry typeAliasRegistry = new TypeAliasRegistry();
    protected final LanguageDriverRegistry languageRegistry = new
LanguageDriverRegistry();

    // 类型处理器注册机
    protected final TypeHandlerRegistry typeHandlerRegistry = new
TypeHandlerRegistry();

    // 对象工厂和对象包装器工厂
    protected ObjectFactory objectFactory = new DefaultObjectFactory();
    protected ObjectWrapperFactory objectWrapperFactory = new
DefaultObjectWrapperFactory();

    protected final Set<String> loadedResources = new HashSet<>();

    //...
}
```



- **单例模式**：是一种创建型模式，让你能够保证一个类只有一个实例，并提供一个访问该实例的全局节点。
- **场景介绍**：Configuration 就像狗皮膏药一样大单例，贯穿整个会话的生命周期，所以的配置对象；映射、缓

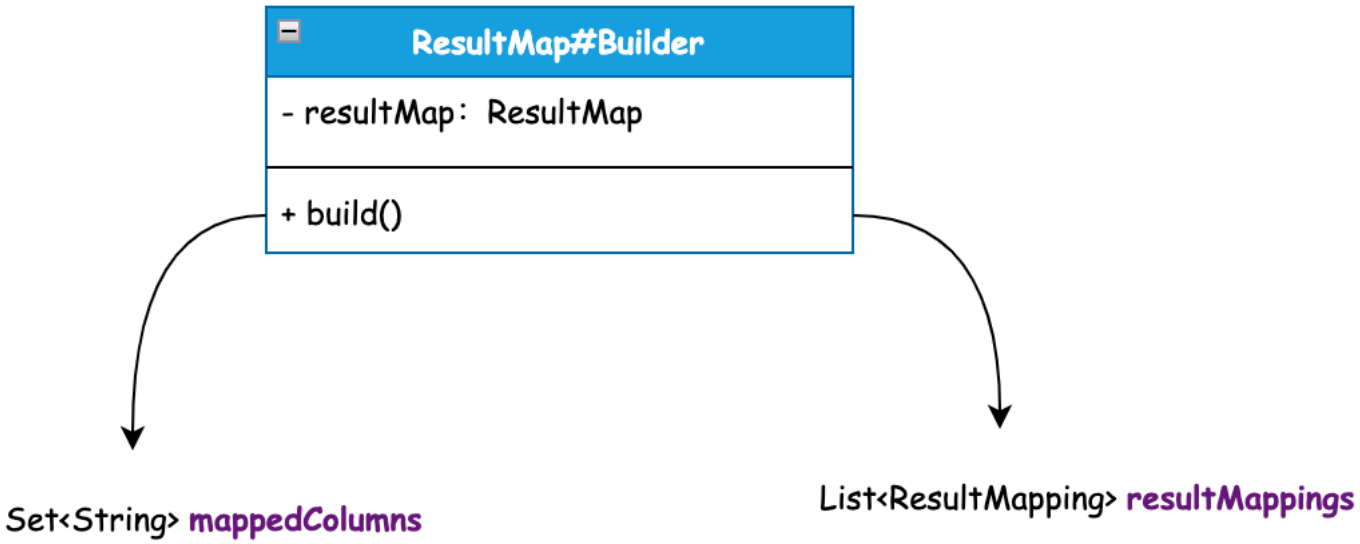
存、入参、出参、拦截器、注册机、对象工厂等，都在 Configuration 配置项中初始化。并随着 SqlSessionFactoryBuilder 构建阶段完成实例化操作。

- **同类场景**：`ErrorContext`、`LogFactory`、`Configuration`

3. 建造者模式

源码详见：`cn.bugstack.mybatis.mapping.ResultMap#Builder`

```
public class ResultMap {

    private String id;
    private Class<?> type;
    private List<ResultMapping> resultMappings;
    private Set<String> mappedColumns;

    private ResultMap() {
    }

    public static class Builder {
        private ResultMap resultMap = new ResultMap();

        public Builder(Configuration configuration, String id, Class<?> type,
List<ResultMapping> resultMappings) {
            resultMap.id = id;
            resultMap.type = type;
            resultMap.resultMappings = resultMappings;
        }

        public ResultMap build() {
            resultMap.mappedColumns = new HashSet<>();
            // step-13 新增加，添加 mappedColumns 字段
            for (ResultMapping resultMapping : resultMap.resultMappings) {
                final String column = resultMapping.getColumn();
                if (column != null) {
                    resultMap.mappedColumns.add(column.toUpperCase(Locale.ENGLISH));
                }
            }
            return resultMap;
        }

    }

    // ... get
}
```



- **建造者模式**：使用多个简单的对象一步一步构建成一个复杂的对象，这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。
- **场景介绍**：关于建造者模式在 Mybatis 框架里的使用，那真是纱窗擦屁股，给你漏了一手。到处都是 XxxxBuilder，所有关于 XML 文件的解析到各类对象的封装，都使用建造者以及建造者助手来完成对象的封装。它的核心目的就是不希望把过多的关于对象的属性设置，写到其他业务流程中，而是用建造者的方式提供最佳的边界隔离。
- **同类场景**：`SqlSessionFactoryBuilder`、`XMLConfigBuilder`、`XMLMapperBuilder`、`XMLStatementBuilder`、`CacheBuilder`

四、类型：结构型模式

1. 适配器模式

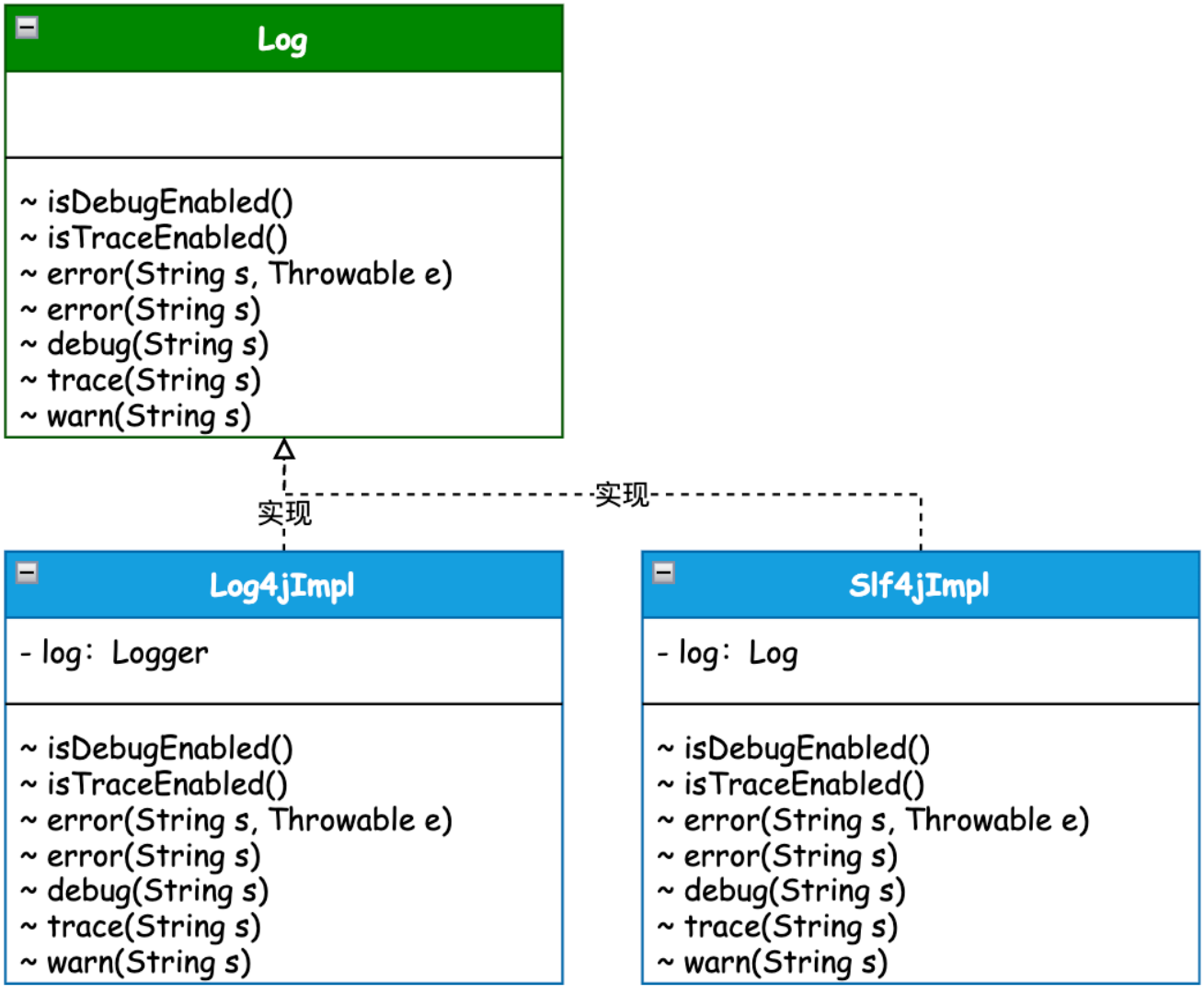
源码详见：`cn.bugstack.mybatis.logging.Log`

```
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);

}
```

源码详见： `cn.bugstack.mybatis.logging.slf4j.Slf4jImpl`

```
public class Slf4jImpl implements Log {

  private Log log;

  public Slf4jImpl(String clazz) {
    Logger logger = LoggerFactory.getLogger(clazz);

    if (logger instanceof LocationAwareLogger) {
      try {
        // check for slf4j >= 1.6 method signature
        logger.getClass().getMethod("log", Marker.class, String.class, int.class,
String.class, Object[].class, Throwable.class);
        log = new Slf4jLocationAwareLoggerImpl((LocationAwareLogger) logger);
        return;
      } catch (SecurityException e) {
        // fail-back to Slf4jLoggerImpl
      } catch (NoSuchMethodException e) {
        // fail-back to Slf4jLoggerImpl
      }
    }

    // Logger is not LocationAwareLogger or slf4j version < 1.6
    log = new Slf4jLoggerImpl(logger);
  }

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

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

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

  @Override
  public void error(String s) {
    log.error(s);
  }

  @Override
  public void debug(String s) {
```

```
    log.debug(s);
  }

  @Override
  public void trace(String s) {
    log.trace(s);
  }

  @Override
  public void warn(String s) {
    log.warn(s);
  }

}
```

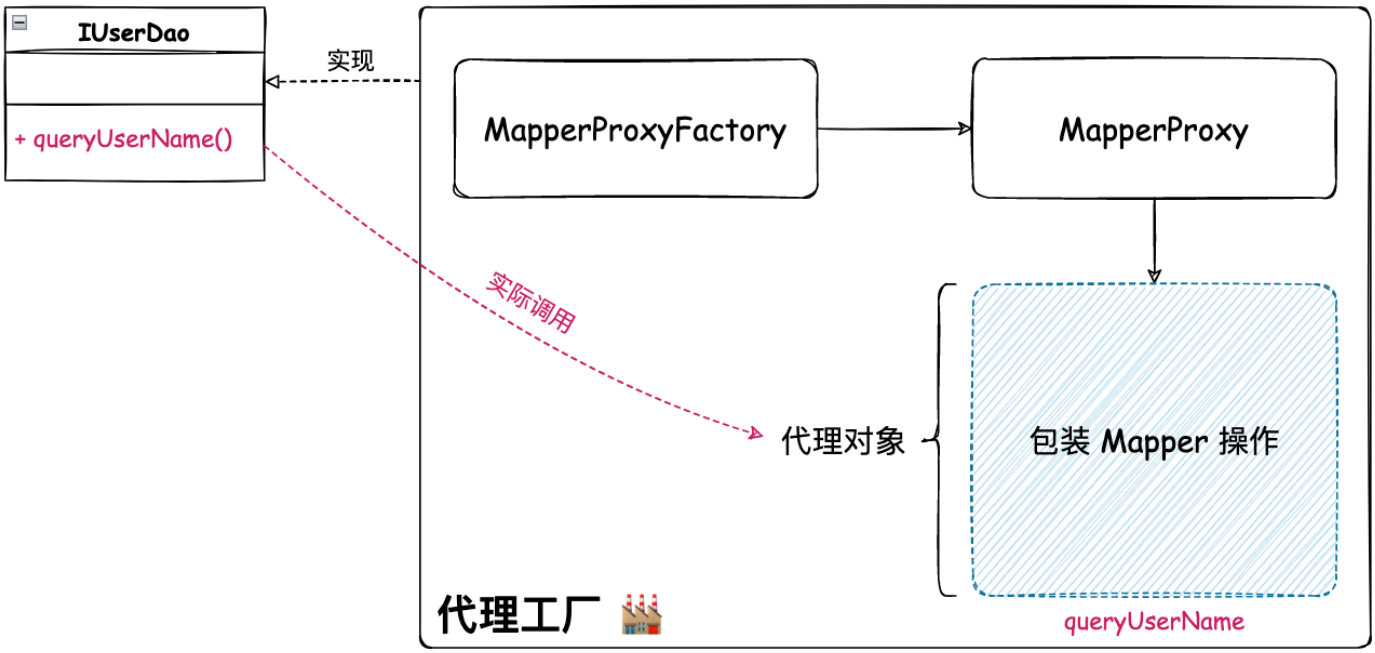


- **适配器模式**：是一种结构型设计模式，它能使接口不兼容的对象能够相互合作。
- **场景介绍**：正是因为有太多的日志框架，包括：Log4j、Log4j2、Slf4J等等，而这些日志框架的使用接口又都各有差异，为了统一这些日志工具的接口，Mybatis 定义了一套统一的日志接口，为所有的其他日志工具接口做相应的适配操作。
- **同类场景**：主要集中在对日志的适配上，Log 和 对应的实现类，以及在 LogFactory 工厂方法中进行使用。

2. 代理模式

源码详见：`cn.bugstack.mybatis.binding.MapperProxy`

```
public class MapperProxy<T> implements InvocationHandler, Serializable {

    private static final long serialVersionUID = -6424540398559729838L;

    private SqlSession sqlSession;
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethod> methodCache;

    public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method,
MapperMethod> methodCache) {
        this.sqlSession = sqlSession;
        this.mapperInterface = mapperInterface;
        this.methodCache = methodCache;
    }

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

    // ...

}
```



- **代理模式**：是一种结构型模式，让你能够提供对象的替代品或其占位符。代理控制着对原对象的访问，并允许在将请求提交给对象前进行一些处理。
- **场景介绍**：不吹牛的讲，没有代理模式，就不会有各类的框架存在。就像 Mybatis 中的 MapperProxy 映射器代理实现类，它所实现的功能就是帮助我们完成 DAO 接口的具体实现类的方法操作，你的任何一个配置的 DAO 接口所调用的 CRUD 方法，都会被 MapperProxy 接管，调用到方法执行器等一系列操作，并返回最终的数据库执行结果。
- **同类场景**：`DriverProxy`、`Plugin`、`Invoker`、`MapperProxy`

3. 组合模式

源码详见：`cn.bugstack.mybatis.scripting.xmltags.SqlNode`

```
public interface SqlNode {

    boolean apply(DynamicContext context);

}
```

源码详见：`cn.bugstack.mybatis.scripting.xmltags.IfSqlNode`

```
public class IfSqlNode implements SqlNode{

    private ExpressionEvaluator evaluator;
    private String test;
    private SqlNode contents;

    public IfSqlNode(SqlNode contents, String test) {
        this.test = test;
        this.contents = contents;
        this.evaluator = new ExpressionEvaluator();
    }
```

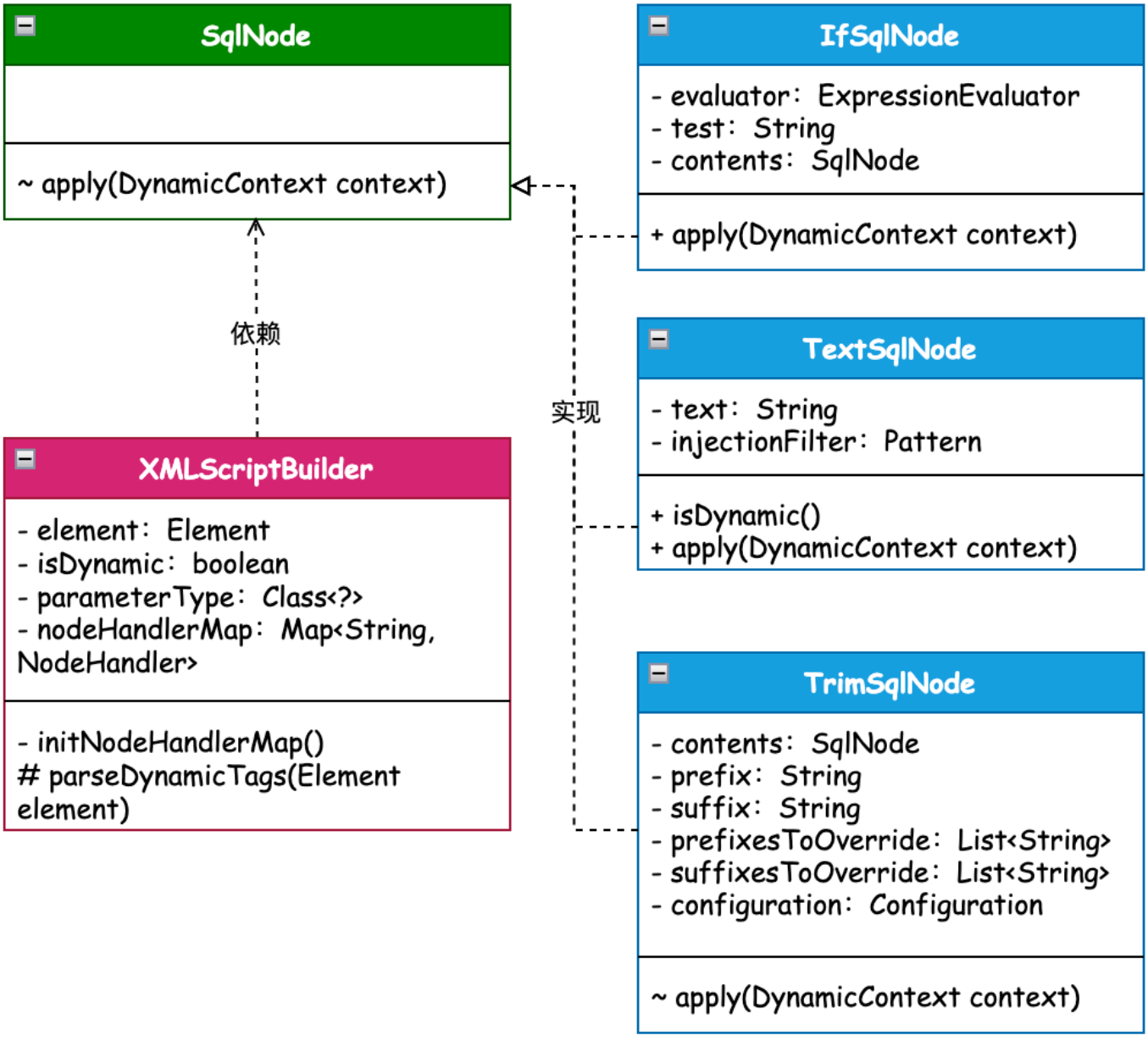
```
    @Override
    public boolean apply(DynamicContext context) {
        // 如果满足条件，则apply，并返回true
        if (evaluator.evaluateBoolean(test, context.getBindings())) {
            contents.apply(context);
            return true;
        }
        return false;
    }

}
```

源码详见： `cn.bugstack.mybatis.scripting.xmltags.XMLScriptBuilder`

```
public class XMLScriptBuilder extends BaseBuilder {

    private void initNodeHandlerMap() {
        // 9种，实现其中2种 trim/where/set/foreach/if/choose/when/otherwise/bind
        nodeHandlerMap.put("trim", new TrimHandler());
        nodeHandlerMap.put("if", new IfHandler());
    }

    List<SqlNode> parseDynamicTags(Element element) {
        List<SqlNode> contents = new ArrayList<>();
        List<Node> children = element.content();
        for (Node child : children) {
            if (child.getNodeType() == Node.TEXT_NODE || child.getNodeType() ==
Node.CDATA_SECTION_NODE) {

            } else if (child.getNodeType() == Node.ELEMENT_NODE) {
                String nodeName = child.getName();
                NodeHandler handler = nodeHandlerMap.get(nodeName);
                if (handler == null) {
                    throw new RuntimeException("Unknown element " + nodeName + " in SQL
statement.");
                }
                handler.handleNode(element.element(child.getName()), contents);
                isDynamic = true;
            }
        }
        return contents;
    }

    // ...
}
```

配置详见： `resources/mapper/Activity_Mapper.xml`

```xml
<select id="queryActivityById" parameterType="cn.bugstack.mybatis.test.po.Activity"
resultMap="activityMap" flushCache="false" useCache="true">
    SELECT activity_id, activity_name, activity_desc, create_time, update_time
    FROM activity
    <trim prefix="where" prefixOverrides="AND | OR" suffixOverrides="and">
        <if test="null != activityId">
            activity_id = #{activityId}
        </if>
    </trim>
</select>
```



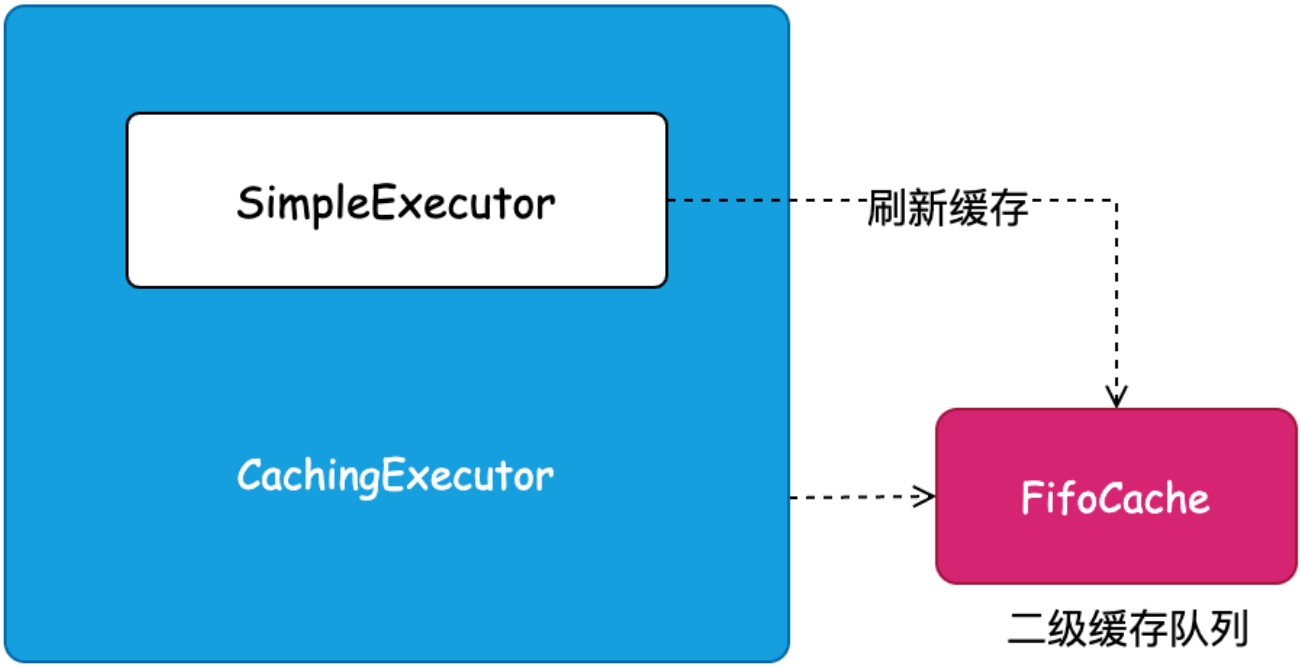
- **组合模式**：是一种结构型设计模式，你可以使用它将对象组合成树状结构，并且能独立使用对象一样使用它们。
- **场景介绍**：在 Mybatis XML 动态的 SQL 配置中，共提供了9种(trim/where/set/foreach/if/choose/when/otherwise/bind)标签的使用，让使用者可以组合出各类场景的 SQL 语句。而 SqlNode 接口的实现就是每一个组合结构中的规则节点，通过规则节点的组装完成一颗规则树组合模式的使用。*具体使用源码可以阅读《手写Mybatis：渐进式源码实践》*

- **同类场景**：主要体现在对各类SQL标签的解析上，以实现 SqlNode 接口的各个子类为主。

4. 装饰器模式

源码详见：`cn.bugstack.mybatis.session.Configuration`

```
public Executor newExecutor(Transaction transaction) {
    Executor executor = new SimpleExecutor(this, transaction);
    // 配置开启缓存，创建 CachingExecutor(默认就是有缓存)装饰者模式
    if (cacheEnabled) {
        executor = new CachingExecutor(executor);
    }
    return executor;
}
```



- **装饰器模式**：是一种结构型设计模式，允许你通过将对象放入包含行为的特殊封装对象中来为原对象绑定新的行为。
- **场景介绍**：Mybatis 的所有 SQL 操作，都是经过 SqlSession 会话调用 SimpleExecutor 简单实现的执行器完成的，而一级缓存的操作也是在简单执行器中处理。那么这里二级缓存因为是基于一级缓存刷新操作的，所以在实现上，通过创建一个缓存执行器，包装简单执行器的处理逻辑，实现二级缓存操作。那么这里用到的就是装饰器模式，也叫俄罗斯套娃模式。
- **同类场景**：主要提前在 Cache 缓存接口的实现和 CachingExecutor 执行器中。

五、类型：行为型模式

1. 模板模式

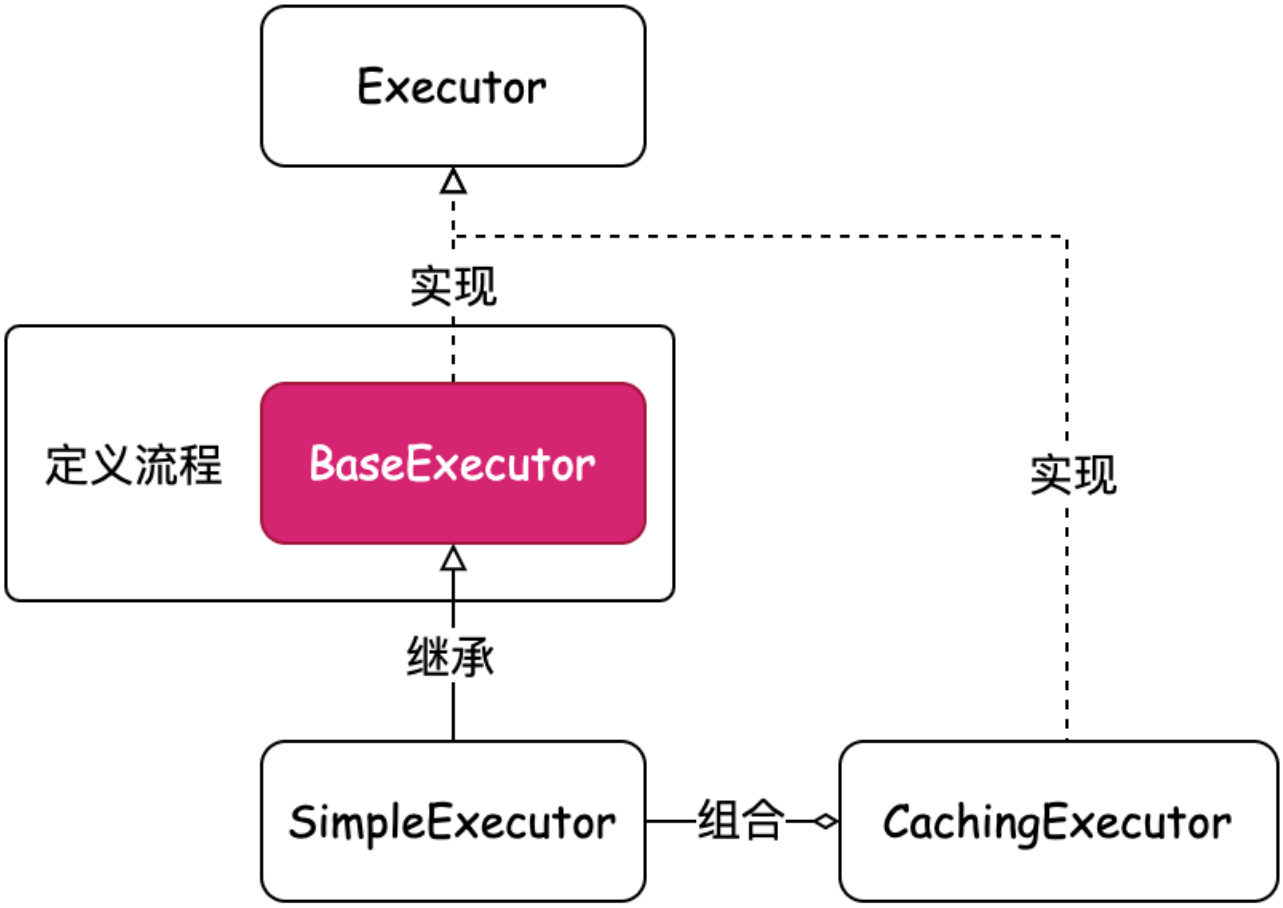
源码详见： `cn.bugstack.mybatis.executor.BaseExecutor`

```
public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler, CacheKey key, BoundSql boundSql) throws SQLException {
    if (closed) {
        throw new RuntimeException("Executor was closed.");
    }
    // 清理局部缓存，查询堆栈为0则清理。queryStack 避免递归调用清理
    if (queryStack == 0 && ms.isFlushCacheRequired()) {
        clearLocalCache();
    }
    List<E> list;
    try {
        queryStack++;
        // 根据cacheKey从localCache中查询数据
        list = resultHandler == null ? (List<E>) localCache.getObject(key) : null;
        if (list == null) {
            list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key,
boundSql);
        }
    } finally {
        queryStack--;
    }
    if (queryStack == 0) {
        if (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
            clearLocalCache();
        }
    }
    return list;
}
```

源码详见： `cn.bugstack.mybatis.executor.SimpleExecutor`

```
protected int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;
    try {
        Configuration configuration = ms.getConfiguration();
        // 新建一个 StatementHandler
        StatementHandler handler = configuration.newStatementHandler(this, ms,
parameter, RowBounds.DEFAULT, null, null);
        // 准备语句
        stmt = prepareStatement(handler);
        // StatementHandler.update
        return handler.update(stmt);
    } finally {
        closeStatement(stmt);
```

```
    }
}
```



- **模板模式**：是一种行为设计模式，它在超类中定义了一个算法的框架，允许子类在不修改结构的情况下重写算法的特定步骤。
- **场景介绍**：只要存在一系列可被标准定义的流程，在流程的步骤大部分是通用逻辑，只有一少部分是需要子类实现的，那么通常会采用模板模式来定义出这个标准的流程。就像 Mybatis 的 BaseExecutor 就是一个用于定义模板模式的抽象类，在这个类中把查询、修改的操作都定义出了一套标准的流程。
- **同类场景**：`BaseExecutor`、`SimpleExecutor`、`BaseTypeHandler`

2. 策略模式

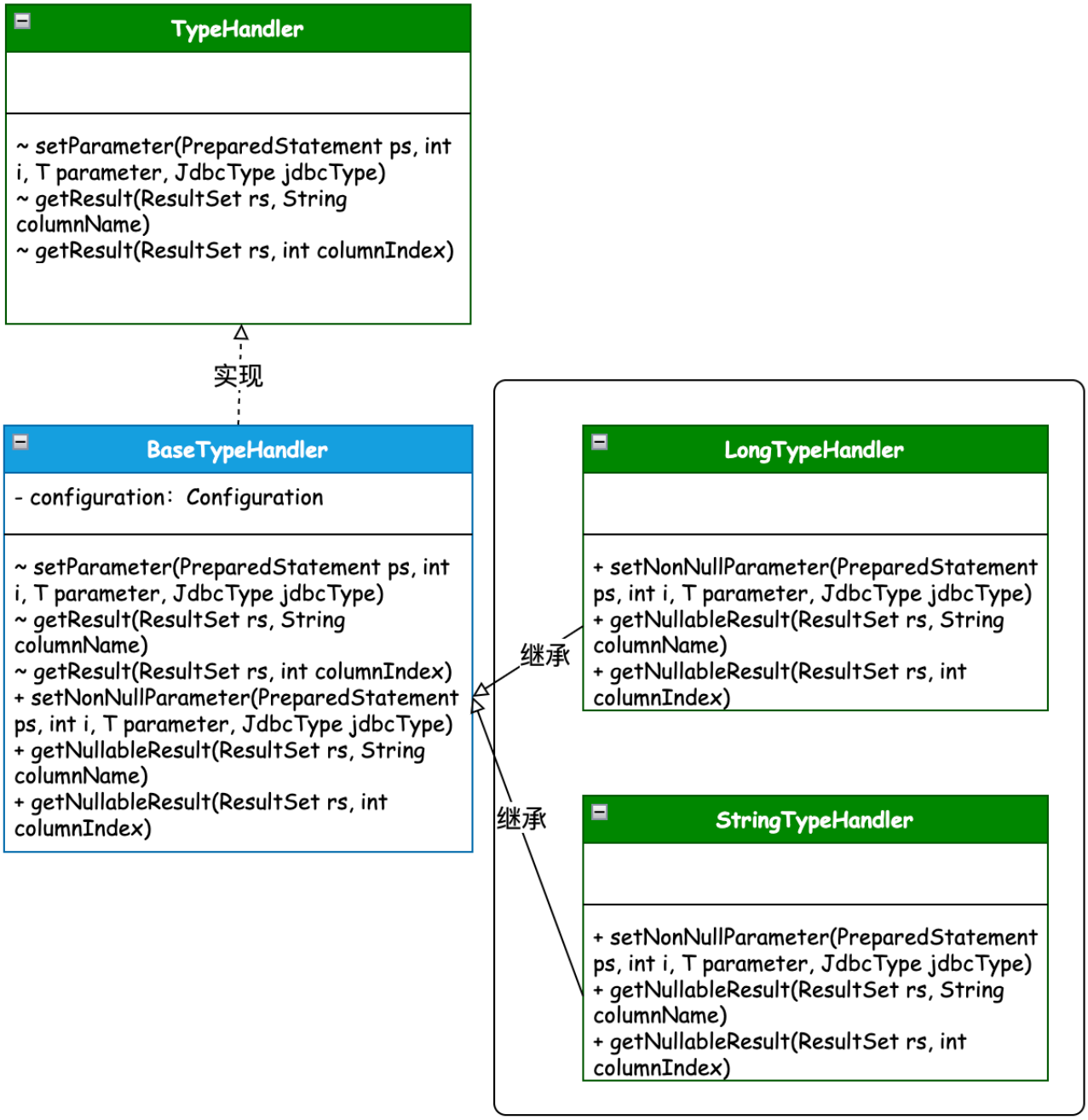
源码详见：`cn.bugstack.mybatis.type.TypeHandler`

```
public interface TypeHandler<T> {

    /**
     * 设置参数
     */
    void setParameter(PreparedStatement ps, int i, T parameter, JdbcType jdbcType)
throws SQLException;

    /**
     * 获取结果
     */
```

```
    T getResult(ResultSet rs, String columnName) throws SQLException;

    /**
     * 取得结果
     */
    T getResult(ResultSet rs, int columnIndex) throws SQLException;

}
```

源码详见： `cn.bugstack.mybatis.type.LongTypeHandler`

```
public class LongTypeHandler extends BaseTypeHandler<Long> {

    @Override
    protected void setNonNullParameter(PreparedStatement ps, int i, Long parameter,
JdbcType jdbcType) throws SQLException {
        ps.setLong(i, parameter);
    }

    @Override
    protected Long getNullableResult(ResultSet rs, String columnName) throws
SQLException {
        return rs.getLong(columnName);
    }

    @Override
    public Long getNullableResult(ResultSet rs, int columnIndex) throws SQLException {
        return rs.getLong(columnIndex);
    }

}
```



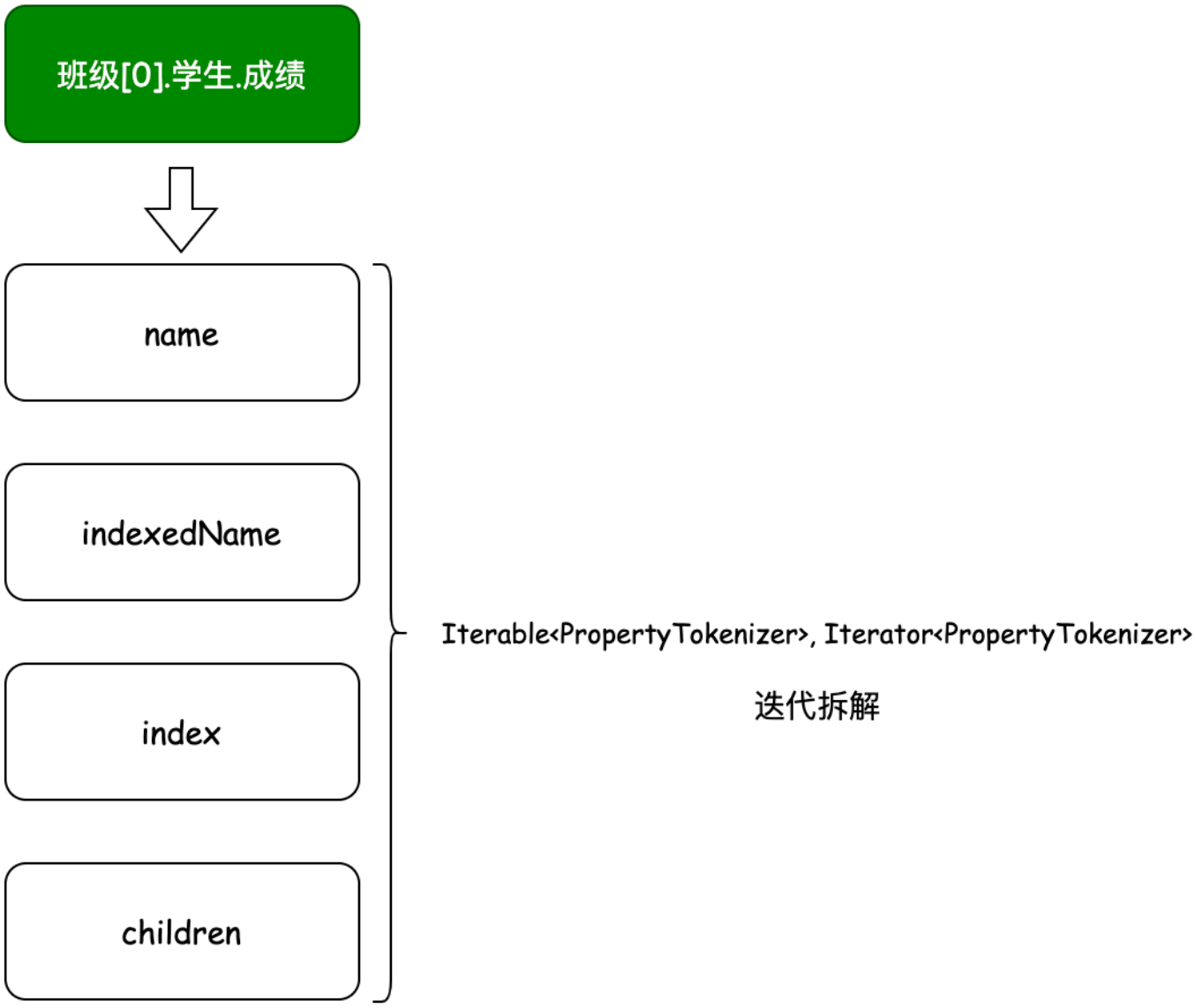
策略模式

- **策略模式**：是一种行为设计模式，它能定义一系列算法，并将每种算法分别放入独立的类中，以使算法的对象能够互相替换。
- **场景介绍**：在 Mybatis 处理 JDBC 执行后返回的结果时，需要按照不同的类型获取对应的值，这样就可以避免大量的 if 判断。所以这里基于 TypeHandler 接口对每个参数类型分别做了自己的策略实现。
- **同类场景**：`PooledDataSource\UnpooledDataSource`、`BatchExecutor\ResuseExecutor\SimpleExector\CachingExecutor`、`LongTypeHandler\StringTypeHandler\DateTypeHandler`

3. 迭代器模式

源码详见： `cn.bugstack.mybatis.reflection.property.PropertyTokenizer`

```
public class PropertyTokenizer implements Iterable<PropertyTokenizer>,
Iterator<PropertyTokenizer> {

    public PropertyTokenizer(String fullname) {
        // 班级[0].学生.成绩
        // 找这个点 .
        int delim = fullname.indexOf('.');
        if (delim > -1) {
            name = fullname.substring(0, delim);
            children = fullname.substring(delim + 1);
        } else {
            // 找不到.的话，取全部部分
            name = fullname;
            children = null;
        }
        indexedName = name;
        // 把中括号里的数字给解析出来
        delim = name.indexOf('[');
        if (delim > -1) {
            index = name.substring(delim + 1, name.length() - 1);
            name = name.substring(0, delim);
        }
    }

    // ...

}
```



- **迭代器模式**：是一种行为设计模式，让你能在不暴露集合底层表现形式的情况下遍历集合中所有的元素。
- **场景介绍**：PropertyTokenizer 是用于 Mybatis 框架 MetaObject 反射工具包下，用于解析对象关系的迭代操作。这个类在 Mybatis 框架中使用的非常频繁，包括解析数据源配置信息并填充到数据源类上，以及参数的解析、对象的设置都会使用到这个类。
- **同类场景**：`PropertyTokenizer`

六、总结：“卷王”的心得

一份源码的成体系拆解渐进式学习，可能需要1~2个月的时间，相比于爽文和疲于应试要花费更多的经历。但你总会在一个大块时间学习完后，会在自己的头脑中构建出一套完整体系关于此类知识的技术架构，无论从哪里入口你都能清楚各个分支流程的走向，这也是你成为技术专家路上的深度学习。

如果你也想有这样酣畅淋漓的学习，千万别错过傅哥为你编写的资料《手写Mybatis：渐进式源码实践》目录如图所示，共计20章

手写Mybatis：渐进式源码实践

作者：小傅哥
公众号：bugstack虫洞栈 - 回复：Mybatis