# 一些其他的很源解读

## 1.田小波-勿在浮沙筑高台

<http://www.tianxiaobo.com/2018/09/09/MyBatis-%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E7%B3%BB%E5%88%97%E6%96%87%E7%AB%A0%E5%90%88%E9%9B%86/>



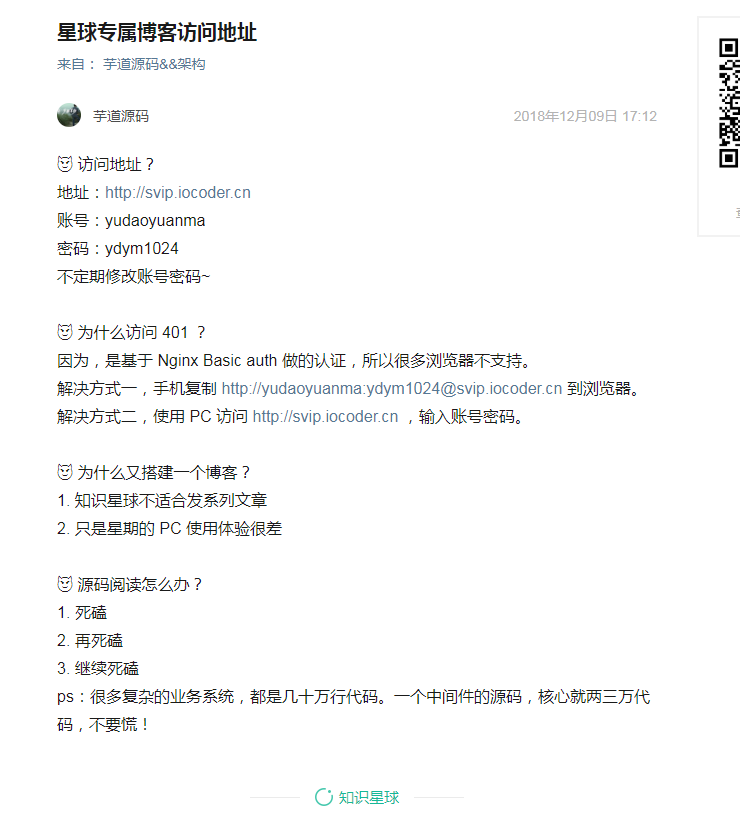
## 2.祖大俊-码读百遍，其义自见。

<https://my.oschina.net/zudajun?tab=newest&catalogId=3532897>

# 1.Start

需要付费加入知识星球：尼玛

<https://articles.zsxq.com/id_wsmvd3iubzvg.html>



<http://svip.iocoder.cn/categories/MyBatis/>

## 1.1 start

<http://www.iocoder.cn/MyBatis/good-collection/>

官方文档：<https://mybatis.org/mybatis-3/zh/index.html>

## 1.2文章目录

MyBatis 的分层非常干净，主要分成三层：

API 接口层，暴露给开发者使用。

核心处理层，实现 MyBatis 内部流程。

基础模块层，提供通用的模块功能，例如缓存、反射等等功能。

可能有点绕，我们来将 MyBatis 各个包 package 拆分到这三层中。如下图所示：



1.API 接口层

[《精尽 MyBatis 源码分析 —— 会话 SqlSession》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

2.核心处理层

配置初始化

[《精尽 MyBatis 源码解析 —— MyBatis 初始化（一）之加载 mybatis-config.md》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— MyBatis 初始化（二）之加载 Mapper 映射配置文件》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— MyBatis 初始化（三）之加载 Statement 配置》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— MyBatis 初始化（四）之加载注解配置》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

SQL 初始化

[《精尽 MyBatis 源码解析 —— SQL 初始化（上）之 SqlNode》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— SQL 初始化（下）之 SqlSource》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

SQL 执行

[《精尽 MyBatis 源码解析 —— SQL 执行（一）之 Executor.md》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

参数设置

[《精尽 MyBatis 源码解析 —— SQL 执行（二）之 StatementHandler》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

主键生成

[《精尽 MyBatis 源码解析 —— SQL 执行（三）之 KeyGenerator》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

结果集映射

[《精尽 MyBatis 源码解析 —— SQL 执行（四）之 ResultSetHandler》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

延迟加载

[《精尽 MyBatis 源码解析 —— SQL 执行（五）之 延迟加载》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

插件体系

[《精尽 MyBatis 源码解析 —— 插件体系（一）之原理》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 插件体系（二）之 PageHelper》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

**3.基础模块层**

[《精尽 MyBatis 源码解析 —— 解析器模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 反射模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 异常模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 数据源模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 事务模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 缓存模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 类型模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— IO 模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 日志模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 注解模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— Binding 模块》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

为了让大家更好的让大家更愉快的精尽 MyBatis 的源码，艿艿也提供了两篇文章：

[《精尽 MyBatis 源码解析 —— 调试环境搭建》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— 项目结构一览》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

可能胖友对 Spring 如何集成 MyBatis 非常感兴趣，艿艿还是写完了所有的文章：

[《精尽 MyBatis 源码解析 —— Spring 集成（一）之调试环境搭建》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— Spring 集成（二）之初始化》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— Spring 集成（三）之 SqlSession》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— Spring 集成（四）之事务》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

[《精尽 MyBatis 源码解析 —— Spring 集成（五）之批处理》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

## 1.3 福利

除非了源码解析，这个系列目前里面会包含：

[《精尽 MyBatis 面试题》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank) 解决面试

[《精尽 MyBatis 资料合集》](https://t.zsxq.com/NFuv3jq" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

包括很多，各种 MyBatis 的资料，不仅仅是源码，也可以是入门、实战、最佳实践等等。

[《芋道 Spring Boot 数据库连接池入门》](http://www.iocoder.cn/Spring-Boot/datasource-pool/?self)

[《芋道 Spring Boot MyBatis 入门》](http://www.iocoder.cn/Spring-Boot/MyBatis/?self) 。

[《芋道 Spring Boot 多数据源（读写分离）入门》](http://www.iocoder.cn/Spring-Boot/dynamic-datasource/?self)

[《芋道 Spring Boot 分库分表入门》](http://www.iocoder.cn/Spring-Boot/sharding-datasource/?self)

[《芋道 Spring Boot 数据库版本管理入门》](http://www.iocoder.cn/Spring-Boot/database-version-control/?self)

**【udbwcso】MyBatis 源码解析**

作者：udbwcso

博客：[https://my.oschina.net/u/657390/blog?catalog=3516796](https://my.oschina.net/u/657390/blog?catalog=3516796" \t "http://www.iocoder.cn/MyBatis/good-collection/_blank)

目录：

[《MyBatis 源码解析 —— SqlSessionFactory》](http://www.iocoder.cn/MyBatis/udbwcso/SqlSessionFactory)

[《MyBatis 源码解析 —— Configuration》](http://www.iocoder.cn/MyBatis/udbwcso/Configuration)

[《MyBatis 源码解析 —— 事务管理器》](http://www.iocoder.cn/MyBatis/udbwcso/Transaction-Manager)

[《MyBatis 源码解析 —— SqlSession 的创建过程》](http://www.iocoder.cn/MyBatis/udbwcso/SqlSession)

[《MyBatis 源码解析 —— mapper 动态代理》](http://www.iocoder.cn/MyBatis/udbwcso/mapper)

[《MyBatis 源码解析 —— MapperMethod》](http://www.iocoder.cn/MyBatis/udbwcso/MapperMethod)

[《MyBatis 源码解析 —— TypeHandler》](http://www.iocoder.cn/MyBatis/udbwcso/TypeHandler)

# 2.调试环境搭建

## 1. 依赖工具

Maven

Git

JDK

IntelliJ IDEA

## 2. 源码拉取

从官方仓库 https://github.com/mybatis/mybatis-3 Fork 出属于自己的仓库。为什么要 Fork ？既然开始阅读、调试源码，我们可能会写一些注释，有了自己的仓库，可以进行自由的提交。😈

使用 IntelliJ IDEA 从 Fork 出来的仓库拉取代码。

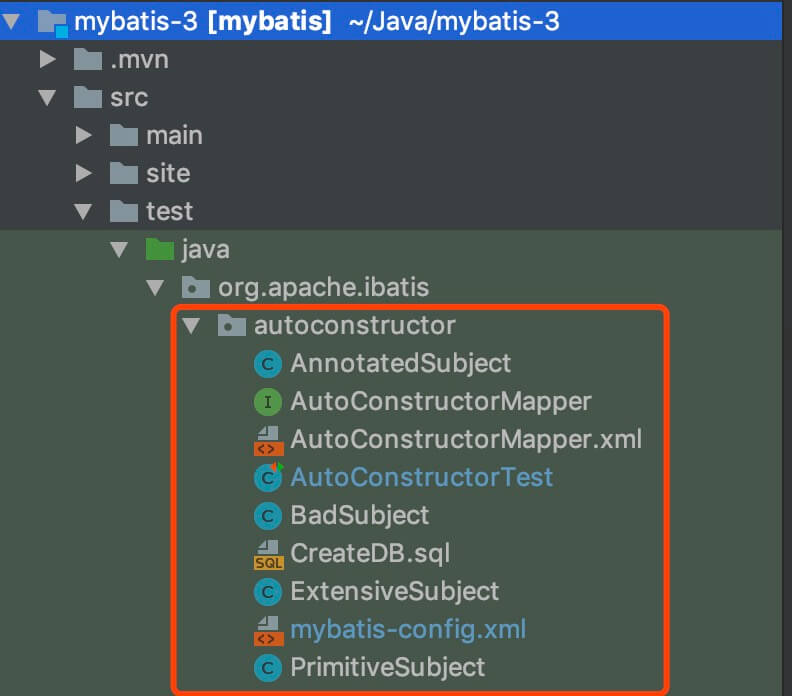
本文使用的 MyBatis 版本为 3.5.0-SNAPSHOT 。

因为 MyBatis 是有些年份的框架，所以克隆的过程可能比较慢，耐心等待。

## 3. 调试

MyBatis 想要调试，非常方便，只需要打开 org.apache.ibatis.autoconstructor.AutoConstructorTest 单元测试类，任意一个单元测试方法，右键，开始调试即可。

当然，考虑到让大家更好的理解 AutoConstructorTest 这个类，下面，我们还是详细解析下。AutoConstructorTest 所在在 autoconstructor 包下，整体结构如下：



## 4. mybatis-config.xml

mybatis-config.xml ，MyBatis 配置文件。XML 如下：

<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
  
<configuration>  
  
 <!-- autoMappingBehavior should be set in each test case -->  
  
 <environments default="development">  
 <environment id="development">  
 <transactionManager type="JDBC">  
 <property name="" value=""/>  
 </transactionManager>  
 <dataSource type="UNPOOLED">  
 <property name="driver" value="org.hsqldb.jdbcDriver"/>  
 <property name="url" value="jdbc:hsqldb:mem:automapping"/>  
 <property name="username" value="root"/>  
 </dataSource>  
 </environment>  
 </environments>  
  
 <mappers>  
 <mapper resource="org/apache/ibatis/autoconstructor/AutoConstructorMapper.xml"/>  
 </mappers>  
  
</configuration>

在 <environments /> 标签中，配置了事务管理和数据源。考虑到减少外部依赖，所以使用了 HSQLDB 。

在 <mappers /> 标签中，配置了需要扫描的 Mapper 文件。目前，仅仅扫描 AutoConstructorMapper.xml 文件。

## 5. AutoConstructorMapper.xml

AutoConstructorMapper.xml ，Mapper 文件。代码如下：

<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
  
<mapper namespace="org.apache.ibatis.autoconstructor.AutoConstructorMapper">  
</mapper>

对应的接口为 org.apache.ibatis.autoconstructor.AutoConstructorMapper 。

AutoConstructorMapper：

public interface AutoConstructorMapper {  
  
 // ========== PrimitiveSubject ==========  
  
 @Select("SELECT \* FROM subject WHERE id = #{id}")  
 PrimitiveSubject getSubject(final int id);  
 @Select("SELECT \* FROM subject")  
 List<PrimitiveSubject> getSubjects();  
  
 // ========== AnnotatedSubject ==========  
  
 @Select("SELECT \* FROM subject")  
 List<AnnotatedSubject> getAnnotatedSubjects();  
  
 // ========== BadSubject ==========  
  
 @Select("SELECT \* FROM subject")  
 List<BadSubject> getBadSubjects();  
  
 // ========== ExtensiveSubject ==========  
  
 @Select("SELECT \* FROM extensive\_subject")  
 List<ExtensiveSubject> getExtensiveSubject();  
  
}

使用注解的方法，编写 SQL 。

## 6. CreateDB.sql

CreateDB.sql 文件，用于单元测试里，初始化数据库的数据。如下：

**DROP TABLE** extensive\_subject  
**IF EXISTS**;  
  
**CREATE TABLE subject** (  
 id **INT NOT NULL**,  
 **name VARCHAR**(20),  
 age **INT NOT NULL**,  
 height **INT**,  
 weight **INT**,  
 active **BIT**,  
 dt **TIMESTAMP**);  
  
**CREATE TABLE** extensive\_subject (  
 aByte **TINYINT**,  
 aShort **SMALLINT**,  
 aChar **CHAR**,  
 anInt **INT**,  
 aLong **BIGINT**,  
 aFloat **FLOAT**,  
 aDouble **DOUBLE**,  
 aBoolean **BIT**,  
 aString **VARCHAR**(255),  
 anEnum **VARCHAR**(50),  
 aClob LONGVARCHAR,  
 aBlob LONGVARBINARY,  
 aTimestamp **TIMESTAMP**);  
  
**INSERT INTO subject VALUES** (1, 'a', 10, 100, 45, 1, *CURRENT\_TIMESTAMP*),  
 (2, 'b', 10, **NULL**, 45, 1, *CURRENT\_TIMESTAMP*),  
 (2, 'c', 10, **NULL**, **NULL**, 0, *CURRENT\_TIMESTAMP*);  
  
**INSERT INTO** extensive\_subject  
**VALUES** (1, 1, 'a', 1, 1, 1, 1.0, 1, 'a', 'AVALUE', 'ACLOB', 'aaaaaabbbbbb', *CURRENT\_TIMESTAMP*),  
 (2, 2, 'b', 2, 2, 2, 2.0, 2, 'b', 'BVALUE', 'BCLOB', '010101010101', *CURRENT\_TIMESTAMP*),  
 (3, 3, 'c', 3, 3, 3, 3.0, 3, 'c', 'CVALUE', 'CCLOB', '777d010078da', *CURRENT\_TIMESTAMP*);

创建了 subject 表，并初始化三条数据。

创建了 extensive\_subject 表，并初始化三条数据。

## 7. POJO

在 AutoConstructorMapper 中，我们可以看到有四个 POJO 类。但是，从 CreateDB.sql 中，实际只有两个表。这个是为什么呢？继续往下看噢。

### 7.1 AnnotatedSubject

package org.apache.ibatis.autoconstructor;  
  
import org.apache.ibatis.annotations.AutomapConstructor;  
  
public class AnnotatedSubject {  
  
 private final int id;  
 private final String name;  
 private final int age;  
 private final int height;  
 private final int weight;  
  
 public AnnotatedSubject(final int id, final String name, final int age, final int height, final int weight) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.height = height;  
 this.weight = weight;  
 }  
  
 @AutomapConstructor  
 public AnnotatedSubject(final int id, final String name, final int age, final Integer height, final Integer weight) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.height = height == null ? 0 : height;  
 this.weight = weight == null ? 0 : weight;  
 }  
  
}

对应 subject 表。

@AutomapConstructor 注解，表示 MyBatis 查询后，在创建 AnnotatedSubject 对象，使用该构造方法。

😈 实际场景下，非常少使用这个注解，甚至 Google 上资料也不多。

#### 7.1.1 PrimitiveSubject

package org.apache.ibatis.autoconstructor;  
  
import java.util.Date;  
  
public class PrimitiveSubject {  
  
 private final int id;  
 private final String name;  
 private final int age;  
 private final int height;  
 private final int weight;  
 private final boolean active;  
 private final Date dt;  
  
 public PrimitiveSubject(final int id, final String name, final int age, final int height, final int weight, final boolean active, final Date dt) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.height = height;  
 this.weight = weight;  
 this.active = active;  
 this.dt = dt;  
 }  
  
}

对应的也是 subject 表。

和 AnnotatedSubject 不同，在其构造方法上，weight 和 height 方法参数的类型是 int ，而不是 Integer 。那么，如果 subject 表中的记录，这两个字段为 NULL 时，会创建 PrimitiveSubject 对象报错。

#### 7.1.2 BadSubject

package org.apache.ibatis.autoconstructor;  
  
public class BadSubject {  
  
 private final int id;  
 private final String name;  
 private final int age;  
 private final Height height;  
 private final Double weight;  
  
 public BadSubject(final int id, final String name, final int age, final Height height, final Double weight) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 this.height = height;  
 this.weight = weight == null ? 0 : weight;  
 }  
  
 private class Height {  
  
 }  
  
}

对应的也是 subject 表。

和 AnnotatedSubject 不同，在其构造方法上，height 方法参数的类型是 Height ，而不是 Integer 。因为 MyBatis 无法识别 Height 类，所以会创建 BadSubject 对象报错。

老艿艿：一般情况下，**POJO 对象里，不使用基本类型**。

### 7.2 ExtensiveSubject

|  |
| --- |
| package org.apache.ibatis.autoconstructor;  public class ExtensiveSubject {  private final byte aByte;  private final short aShort;  private final char aChar;  private final int anInt;  private final long aLong;  private final float aFloat;  private final double aDouble;  private final boolean aBoolean;  private final String aString;   // enum types  private final TestEnum anEnum;   // array types   // string to lob types:  private final String aClob;  private final String aBlob;   public ExtensiveSubject(final byte aByte,  final short aShort,  final char aChar,  final int anInt,  final long aLong,  final float aFloat,  final double aDouble,  final boolean aBoolean,  final String aString,  final TestEnum anEnum,  final String aClob,  final String aBlob) {  this.aByte = aByte;  this.aShort = aShort;  this.aChar = aChar;  this.anInt = anInt;  this.aLong = aLong;  this.aFloat = aFloat;  this.aDouble = aDouble;  this.aBoolean = aBoolean;  this.aString = aString;  this.anEnum = anEnum;  this.aClob = aClob;  this.aBlob = aBlob;  }   public enum TestEnum {  AVALUE, BVALUE, CVALUE;  }  } |

对应的也是 extensive\_subject 表。

这是个复杂对象，基本涵盖了各种类型的数据。

## 8. AutoConstructorTest

org.apache.ibatis.autoconstructor.AutoConstructorTest ，单元测试类。

### 8.1 setUp

|  |
| --- |
| private static SqlSessionFactory sqlSessionFactory;  @BeforeClass public static void setUp() throws Exception {  // create a SqlSessionFactory  // 创建 SqlSessionFactory 对象，基于 mybatis-config.xml 配置文件。  try (Reader reader = Resources.getResourceAsReader("org/apache/ibatis/autoconstructor/mybatis-config.xml")) {  sqlSessionFactory = new SqlSessionFactoryBuilder().build(reader);  }   // populate in-memory database  // 初始化数据到内存数据库，基于 CreateDB.sql SQL 文件。  BaseDataTest.runScript(sqlSessionFactory.getConfiguration().getEnvironment().getDataSource(),  "org/apache/ibatis/autoconstructor/CreateDB.sql"); } |

创建 SqlSessionFactory 对象，基于 mybatis-config.xml 配置文件。

初始化数据到内存数据库，基于 CreateDB.sql SQL 文件。

### 8.2 测试方法

右键任一单元测试方法，愉快的调试起来把。

# 3.项目结构一览

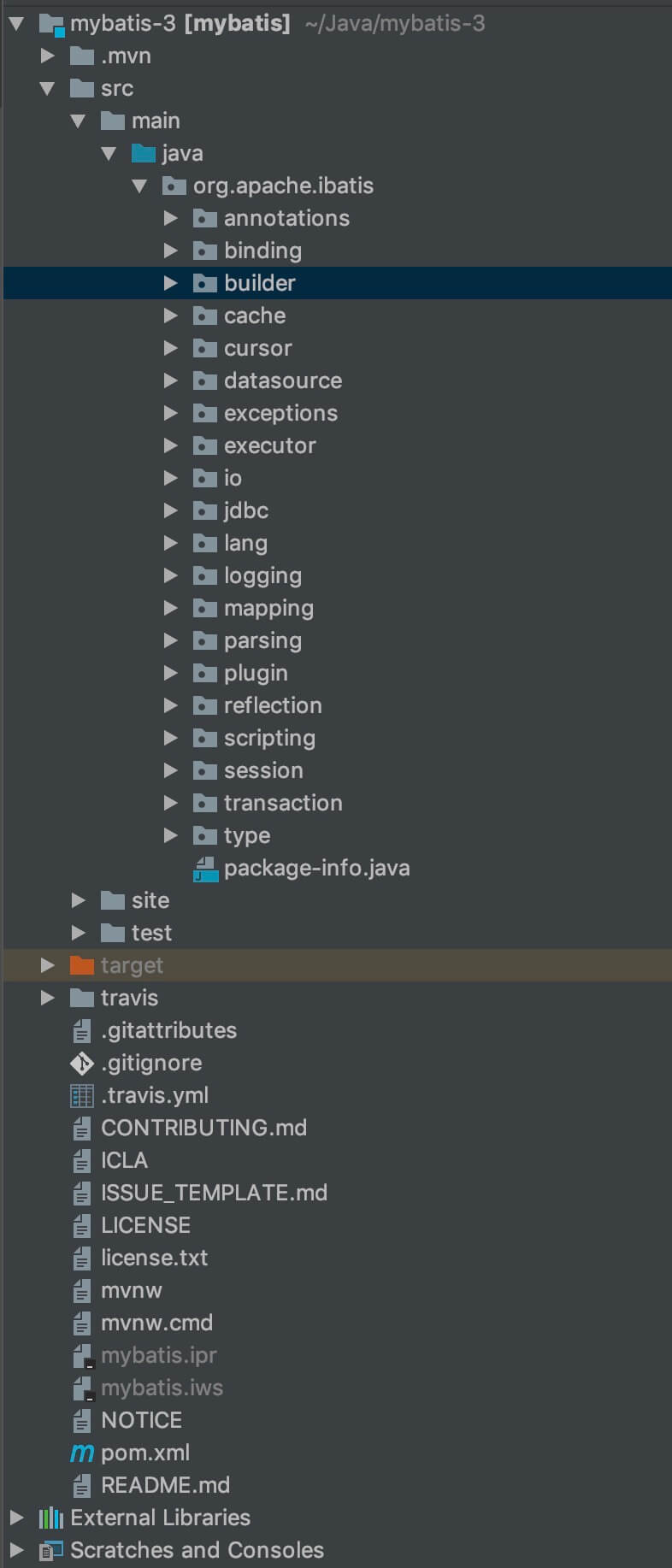
## 1. 概述

本文主要分享 MyBatis 的项目结构。

希望通过本文能让胖友对 MyBatis 的整体项目有个简单的了解。

另外，http://www.mybatis.org 提供了 MyBatis 非常不错的中文文档。如果胖友对 MyBatis 碰到一些使用上的问题，可以常常翻阅。

在拉取 MyBatis 项目后，我们会发现拆分了好多包。是不是内心一紧，产生了恐惧感？不要方，我们就是继续怼。



## 2. 代码统计

这里先分享一个小技巧。笔者在开始源码学习时，会首先了解项目的代码量。

**第一种方式：**，使用 IDEA Statistic 插件，统计整体代码量。

Statistic 统计代码量



我们可以粗略的看到，总的代码量在 21441 行。还是在可接受的范围。

**第二种方式：**，使用 Shell 脚本命令逐个 Maven 模块统计。

<https://blog.csdn.net/yhhwatl/article/details/52623879>

一般情况下，笔者使用 find . -name "\*.java"|xargs cat|grep -v -e ^$ -e ^\s\*\/\/.\*$|wc -l 。这个命令只过滤了部分注释，所以相比 IDEA Statistic 会偏多。

如下是艿艿 cd 到每个包下，然后执行脚本统计后的结果：



## 3. 整体架构

老艿艿：本小节的内容，笔者会基于 《MyBatis 技术内幕》 的 「1.4 MyBatis整体架构」 小节来编写。

这本书非常不错，笔者在死磕的源码的时候，帮助灰常大。

MyBatis 的整体架构分为三层：

1.基础支持层

2.核心处理层

3.接口层



#### 3.1 基础支持层

基础支持层，包含整个 MyBatis 的基础模块，这些模块为核心处理层的功能提供了良好的支撑。

##### 3.1.1 反射模块

对应 reflection 包。

Java 中的反射虽然功能强大，但对大多数开发人员来说，写出高质量的反射代码还是 有一定难度的。MyBatis 中专门提供了反射模块，该模块对 Java 原生的反射进行了良好的封装，提了更加简洁易用的 API，方便上层使调用，并且对反射操作进行了一系列优化，例如缓存了类的元数据，提高了反射操作的性能。

##### 3.1.2 类型模块

对应 type 包。

① MyBatis 为简化配置文件提供了别名机制，该机制是类型转换模块的主要功能之一。

② 类型转换模块的另一个功能是实现 JDBC 类型与 Java 类型之间的转换，该功能在为 SQL 语句绑定实参以及映射查询结果集时都会涉及：

在为 SQL 语句绑定实参时，会将数据由 Java 类型转换成 JDBC 类型。

而在映射结果集时，会将数据由 JDBC 类型转换成 Java 类型。

##### 3.1.3 日志模块

对应 logging 包。

无论在开发测试环境中，还是在线上生产环境中，日志在整个系统中的地位都是非常重要的。良好的日志功能可以帮助开发人员和测试人员快速定位 Bug 代码，也可以帮助运维人员快速定位性能瓶颈等问题。目前的 Java 世界中存在很多优秀的日志框架，例如 Log4j、 Log4j2、Slf4j 等。

MyBatis 作为一个设计优良的框架，除了提供详细的日志输出信息，还要能够集成多种日志框架，其日志模块的一个主要功能就是集成第三方日志框架。

##### 3.1.4 IO模块

对应 io 包。

资源加载模块，主要是对类加载器进行封装，确定类加载器的使用顺序，并提供了加载类文件以及其他资源文件的功能 。

##### 3.1.5 解析器模块

对应 parsing 包。

解析器模块，主要提供了两个功能:

一个功能，是对 XPath 进行封装，为 MyBatis 初始化时解析 mybatis-config.xml 配置文件以及映射配置文件提供支持。

另一个功能，是为处理动态 SQL 语句中的占位符提供支持。

##### 3.1.6 数据源模块

对应 datasource 包。

数据源是实际开发中常用的组件之一。现在开源的数据源都提供了比较丰富的功能，例如，连接池功能、检测连接状态等，选择性能优秀的数据源组件对于提升 ORM 框架乃至整个应用的性能都是非常重要的。

MyBatis 自身提供了相应的数据源实现，当然 MyBatis 也提供了与第三方数据源集成的接口，这些功能都位于数据源模块之中。

##### 3.1.7 事务模块

对应 transaction 包。

MyBatis 对数据库中的事务进行了抽象，其自身提供了相应的事务接口和简单实现。

在很多场景中，MyBatis 会与 Spring 框架集成，并由 Spring 框架管理事务。

##### 3.1.8 缓存模块

对应 cache 包。

在优化系统性能时，优化数据库性能是非常重要的一个环节，而添加缓存则是优化数据库时最有效的手段之一。正确、合理地使用缓存可以将一部分数据库请求拦截在缓存这一层。

MyBatis 中提供了一级缓存和二级缓存，而这两级缓存都是依赖于基础支持层中的缓 存模块实现的。这里需要读者注意的是，MyBatis 中自带的这两级缓存与 MyBatis 以及整个应用是运行在同一个 JVM 中的，共享同一块堆内存。如果这两级缓存中的数据量较大， 则可能影响系统中其他功能的运行，所以当需要缓存大量数据时，优先考虑使用 Redis、Memcache 等缓存产品。

##### 3.1.9 Binding 模块

对应 binding 包。

在调用 SqlSession 相应方法执行数据库操作时，需要指定映射文件中定义的 SQL 节点，如果出现拼写错误，我们只能在运行时才能发现相应的异常。为了尽早发现这种错误，MyBatis 通过 Binding 模块，将用户自定义的 Mapper 接口与映射配置文件关联起来，系统可以通过调用自定义 Mapper 接口中的方法执行相应的 SQL 语句完成数据库操作，从而避免上述问题。

值得读者注意的是，开发人员无须编写自定义 Mapper 接口的实现，MyBatis 会自动为其创建动态代理对象。在有些场景中，自定义 Mapper 接口可以完全代替映射配置文件，但有的映射规则和 SQL 语句的定义还是写在映射配置文件中比较方便，例如动态 SQL 语句的定义。

##### 3.1.10 注解模块

对应 annotations 包。

随着 Java 注解的慢慢流行，MyBatis 提供了注解的方式，使得我们方便的在 Mapper 接口上编写简单的数据库 SQL 操作代码，而无需像之前一样，必须编写 SQL 在 XML 格式的 Mapper 文件中。虽然说，实际场景下，大家还是喜欢在 XML 格式的 Mapper 文件中编写响应的 SQL 操作。

##### 3.1.11 异常模块

对应 exceptions 包。

定义了 MyBatis 专有的 PersistenceException 和 TooManyResultsException 异常。

#### 3.2 核心处理层

在核心处理层中，实现了 MyBatis 的核心处理流程，其中包括 MyBatis 的初始化以及完成一次数据库操作的涉及的全部流程 。

##### 3.2.1 配置解析

对应 builder 和 mapping 模块。前者为配置解析过程，后者主要为 SQL 操作解析后的映射。

在 MyBatis 初始化过程中，会加载 mybatis-config.xml 配置文件、映射配置文件以及 Mapper 接口中的注解信息，解析后的配置信息会形成相应的对象并保存到 Configuration 对象中。例如：

<resultMap>节点(即 ResultSet 的映射规则) 会被解析成 ResultMap 对象。

<result> 节点(即属性映射)会被解析成 ResultMapping 对象。

之后，利用该 Configuration 对象创建 SqlSessionFactory对象。待 MyBatis 初始化之后，开发人员可以通过初始化得到 SqlSessionFactory 创建 SqlSession 对象并完成数据库操作。

##### 3.2.2 SQL 解析

对应 scripting 模块。

拼凑 SQL 语句是一件烦琐且易出错的过程，为了将开发人员从这项枯燥无趣的工作中 解脱出来，MyBatis 实现动态 SQL 语句的功能，提供了多种动态 SQL语句对应的节点。例如<where> 节点、<if> 节点、<foreach> 节点等 。通过这些节点的组合使用， 开发人 员可以写出几乎满足所有需求的动态 SQL 语句。

MyBatis 中的 scripting 模块，会根据用户传入的实参，解析映射文件中定义的动态 SQL 节点，并形成数据库可执行的 SQL 语句。之后会处理 SQL 语句中的占位符，绑定用户传入的实参。

##### 3.2.3 SQL 执行

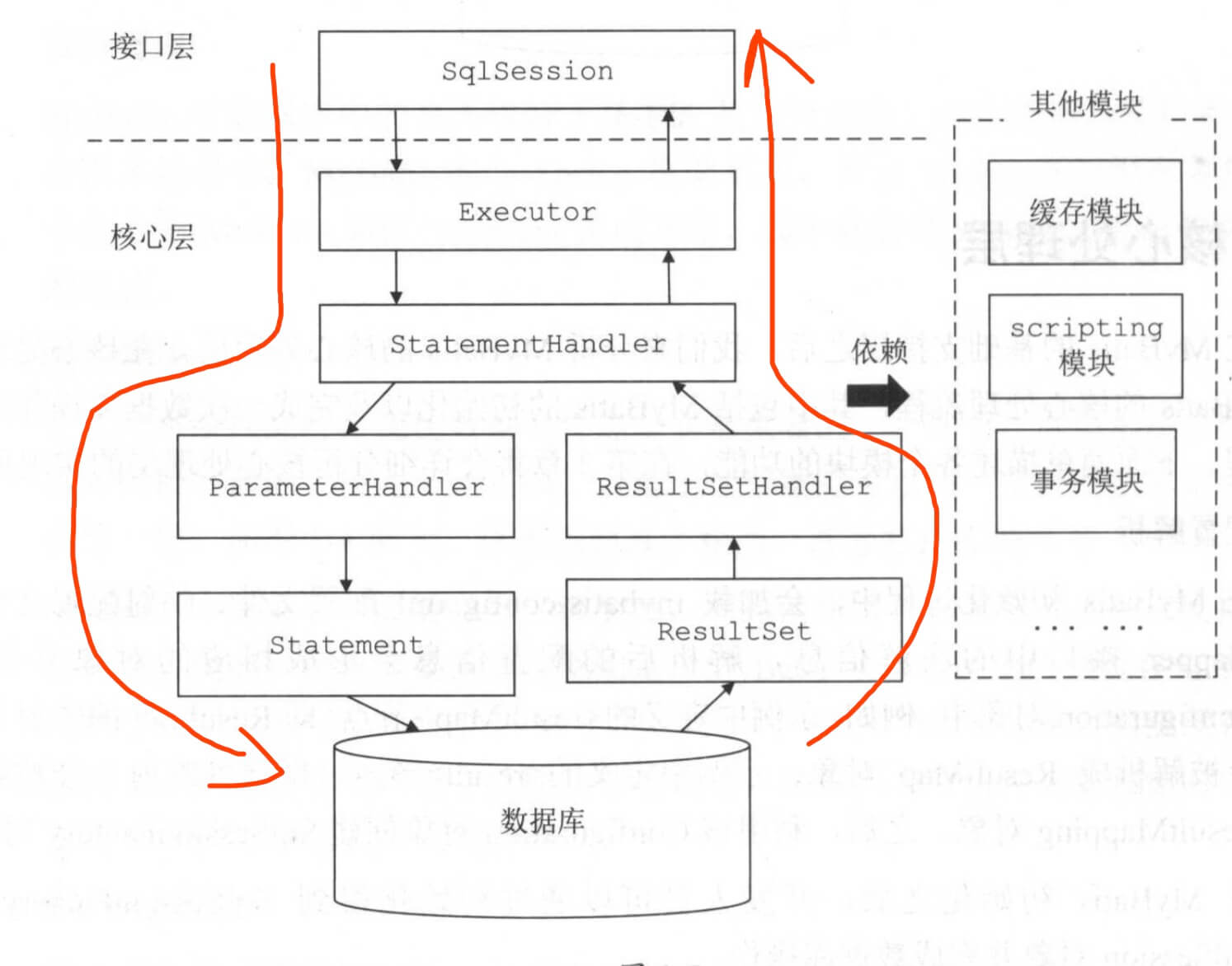
对应 executor 和 cursor 模块。前者对应执行器，后者对应执行结果的游标。

SQL 语句的执行涉及多个组件 ，其中比较重要的是 Executor、StatementHandler、ParameterHandler 和 ResultSetHandler 。

Executor 主要负责维护一级缓存和二级缓存，并提供事务管理的相关操作，它会将数据库相关操作委托给 StatementHandler完成。

StatementHandler 首先通过 ParameterHandler 完成 SQL 语句的实参绑定，然后通过 java.sql.Statement 对象执行 SQL 语句并得到结果集，最后通过 ResultSetHandler 完成结果集的映射，得到结果对象并返回。

整体过程如下图：



##### 3.2.4 插件层

对应 plugin 模块。

Mybatis 自身的功能虽然强大，但是并不能完美切合所有的应用场景，因此 MyBatis 提供了插件接口，我们可以通过添加用户自定义插件的方式对 MyBatis 进行扩展。用户自定义插件也可以改变 Mybatis 的默认行为，例如，我们可以拦截 SQL 语句并对其进行重写。

由于用户自定义插件会影响 MyBatis 的核心行为，在使用自定义插件之前，开发人员需要了解 MyBatis 内部的原理，这样才能编写出安全、高效的插件。

#### 3.3 接口层

对应 session 模块。

接口层相对简单，其核心是 SqlSession 接口，该接口中定义了 MyBatis 暴露给应用程序调用的 API，也就是上层应用与 MyBatis 交互的桥梁。接口层在接收到调用请求时，会调用核心处理层的相应模块来完成具体的数据库操作。

#### 3.4 其它层

这块，严格来说，不能叫做一个层。考虑到统一，就简单这么命名把。哈哈哈。

##### 3.4.1 JDBC 模块

对应 jdbc 包。

JDBC 单元测试工具类。所以，不感兴趣的同学，已经可以忽略 1236 行代码了。

##### 3.4.2 Lang 模块

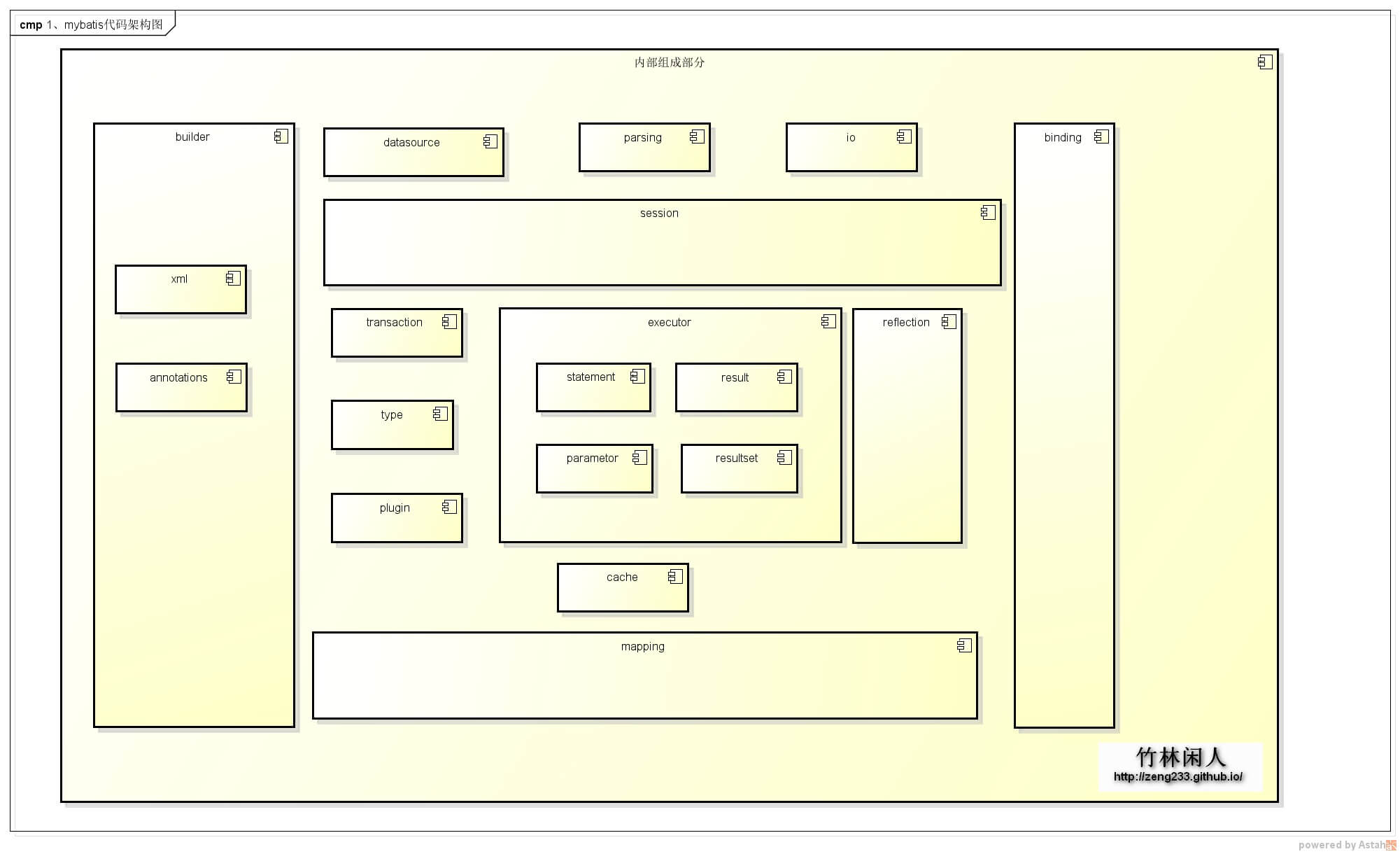
对应 lang 包。

看不懂具体用途，暂时误会。

#### 3.5 总结

总的来说，MyBatis 的代码，还是比较简单易懂。并且，看着包名，基本也能猜到大体的用途。在网络上找了一个总结的图，大家瞅瞅：

FROM 《mybatis 源码分析之整体架构分析》



#### 666. 彩蛋

有种，把 [《MyBatis 技术内幕》](https://item.jd.com/12125531.html" \t "http://svip.iocoder.cn/MyBatis/intro/_blank) 复制了一遍的感觉。不要打我的脸。

后续的文章，和 [《MyBatis 技术内幕》](https://item.jd.com/12125531.html" \t "http://svip.iocoder.cn/MyBatis/intro/_blank) 一样，按照****基础支持层 => 核心处理层 => 接口层****的顺序，逐步详细解析。虽然说，艿艿也想直接开始干核心处理层，但是考虑框架是自下而上，还是一点一点来。当然，热情的胖友，也可以从核心处理层开始看，然后看到不懂的地方，回到基础支持层。

如果对 MyBatis 不熟悉的胖友，推荐阅读下田小波的 [《MyBatis 源码分析系列文章导读》](https://www.tianxiaobo.com/2018/07/16/MyBatis-%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E7%B3%BB%E5%88%97%E6%96%87%E7%AB%A0%E5%AF%BC%E8%AF%BB/" \t "http://svip.iocoder.cn/MyBatis/intro/_blank) 。

同时，也推荐阅读无忌的 [《MyBatis 源码解读之源码结构》](https://my.oschina.net/wenjinglian/blog/1625437" \t "http://svip.iocoder.cn/MyBatis/intro/_blank) 。

# 4.解析器模块

## 1.概述



解析器模块：对应 parsing 包。

解析器模块，主要提供了两个功能:

一个功能，是对 XPath 进行封装，为 MyBatis 初始化时解析 mybatis-config.xml 配置文件以及映射配置文件提供支持。

另一个功能，是为处理动态 SQL 语句中的占位符提供支持。

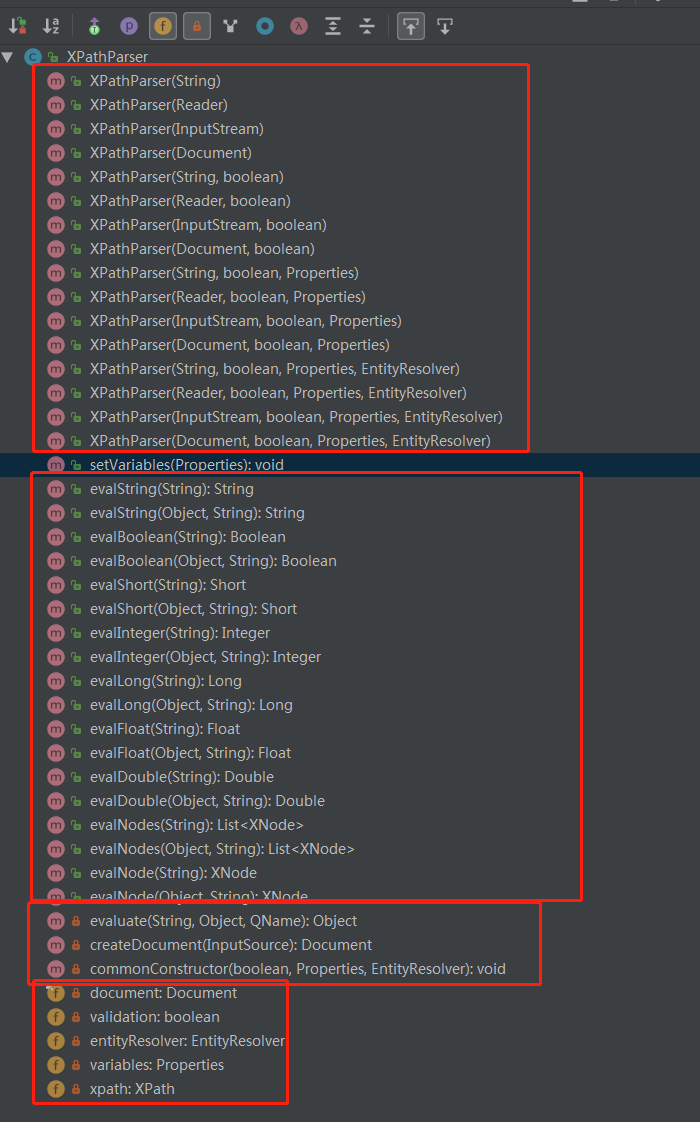
解析器模块，主要提供了两个功能:

一个功能，是对 XPath 进行封装，为 MyBatis 初始化时解析 mybatis-config.xml 配置文件以及映射配置文件提供支持。

另一个功能，是为处理动态 SQL 语句中的占位符提供支持。

下面，我们就来看看具体的源码。因为 parsing 是基础支持层，所以建议胖友在我们讲解到的类和方法中，打折断点一起来了解。

## 2.XPathParser



org.apache.ibatis.parsing.XPathParser ，基于 Java XPath 解析器，用于解析 MyBatis mybatis-config.xml 和 \*\*Mapper.xml 等 XML 配置文件。属性如下：

|  |
| --- |
| // XPathParser.java  /\*\*  \* XML Document 对象  \*/ private final Document document; /\*\*  \* 是否校验  \*/ private boolean validation; /\*\*  \* XML 实体解析器  \*/ private EntityResolver entityResolver; /\*\*  \* 变量 Properties 对象  \*/ private Properties variables; /\*\*  \* Java XPath 对象  \*/ private XPath xpath; |

**document 属性：**

XML 被解析后，生成的 org.w3c.dom.Document 对象。

**validation 属性：**

是否校验 XML 。一般情况下，值为 true 。

**entityResolver 属性：**

org.xml.sax.EntityResolver 对象，XML 实体解析器。默认情况下，对 XML 进行校验时，会基于 XML 文档开始位置指定的 DTD 文件或 XSD 文件。例如说，解析 mybatis-config.xml 配置文件时，会加载 http://mybatis.org/dtd/mybatis-3-config.dtd 这个 DTD 文件。但是，如果每个应用启动都从网络加载该 DTD 文件，势必在弱网络下体验非常下，甚至说应用部署在无网络的环境下，还会导致下载不下来，那么就会出现 XML 校验失败的情况。所以，在实际场景下，MyBatis 自定义了 EntityResolver 的实现，达到使用本地 DTD 文件，从而避免下载网络 DTD 文件的效果。详细解析，见 「3. XMLMapperEntityResolver」 。

另外，Spring 也自定义了 EntityResolver 的实现，感兴趣的胖友，可以看看 《【死磕 Spring】—— IoC 之获取验证模型》 。

**xpath 属性：**

javax.xml.xpath.XPath 对象，用于查询 XML 中的节点和元素。如果对 XPath 的使用不了解的胖友，请先跳转 《Java XPath 解析器 - 解析 XML 文档》 中，进行简单学习，灰常简单。

**variables 属性：**

变量 Properties 对象，用来替换需要动态配置的属性值。例如：

<dataSource type="POOLED">  
 <property name="driver" value="${driver}"/>  
 <property name="url" value="${url}"/>  
 <property name="username" value="${username}"/>  
 <property name="password" value="${password}"/>  
</dataSource>

variables 的来源，即可以在常用的 Java Properties 文件中配置，也可以使用 MyBatis <property /> 标签中配置。例如：

<properties resource="org/mybatis/example/config.properties">

<property name="username" value="dev\_user"/>

<property name="password" value="F2Fa3!33TYyg"/>

</properties>

这里配置的 username 和 password 属性，就可以替换上面的 ${username} 和 ${password} 这两个动态属性。

具体如何实现的，我们来看下面的 PropertyParser#parse(String string, Properties variables) 方法。

### 2.1 构造方法

XPathParser 的构造方法有 16 个之多，当然基本都非常相似，我们来挑选其中一个。代码如下：

// XPathParser.java  
  
/\*\*  
 \* 构造 XPathParser 对象  
 \*  
 \* @param xml XML 文件地址  
 \* @param validation 是否校验 XML  
 \* @param variables 变量 Properties 对象  
 \* @param entityResolver XML 实体解析器  
 \*/  
public XPathParser(String xml, boolean validation, Properties variables, EntityResolver entityResolver) {  
 commonConstructor(validation, variables, entityResolver);  
 this.document = createDocument(new InputSource(new StringReader(xml)));  
}

调用 #commonConstructor(boolean validation, Properties variables, EntityResolver entityResolver) 方法，公用的构造方法逻辑。代码如下：

// XPathParser.java  
  
private void commonConstructor(boolean validation, Properties variables, EntityResolver entityResolver) {  
 this.validation = validation;  
 this.entityResolver = entityResolver;  
 this.variables = variables;  
 // 创建 XPathFactory 对象  
 XPathFactory factory = XPathFactory.newInstance();  
 this.xpath = factory.newXPath();  
}

调用 #createDocument(InputSource inputSource) 方法，将 XML 文件解析成 Document 对象。代码如下：

*/\*\*  
 \* 创建 Document 对象  
 \*  
 \** ***@param*** *inputSource XML 的 InputSource 对象  
 \** ***@return*** *Document 对象  
 \*/*private Document createDocument(InputSource inputSource) {  
 // important: this must only be called AFTER common constructor  
 try {  
 // 1> 创建 DocumentBuilderFactory 对象  
 DocumentBuilderFactory factory = DocumentBuilderFactory.*newInstance*();  
 factory.setFeature(XMLConstants.*FEATURE\_SECURE\_PROCESSING*, true);  
 factory.setValidating(validation);// 设置是否验证 XML  
  
 factory.setNamespaceAware(false);  
 factory.setIgnoringComments(true);  
 factory.setIgnoringElementContentWhitespace(false);  
 factory.setCoalescing(false);  
 factory.setExpandEntityReferences(true);  
 // 2> 创建 DocumentBuilder 对象  
 DocumentBuilder builder = factory.newDocumentBuilder();  
 // 设置实体解析器  
 builder.setEntityResolver(entityResolver);  
 builder.setErrorHandler(new ErrorHandler() {// 实现都空的  
 @Override  
 public void error(SAXParseException exception) throws SAXException {  
 throw exception;  
 }  
  
 @Override  
 public void fatalError(SAXParseException exception) throws SAXException {  
 throw exception;  
 }  
  
 @Override  
 public void warning(SAXParseException exception) throws SAXException {  
 // NOP  
 }  
 });  
 // 3> 解析 XML 文件  
 return builder.parse(inputSource);  
 } catch (Exception e) {  
 throw new BuilderException("Error creating document instance. Cause: " + e, e);  
 }  
}

就是简单的 Java XML API 的使用，不了解的胖友，可以 Google 学习下。

### 2.2 eval 方法族

XPathParser 提供了一系列的 #eval\* 方法，用于获得 Boolean、Short、Integer、Long、Float、Double、String、Node 类型的元素或节点的“值”。当然，虽然方法很多，但是都是基于 #evaluate(String expression, Object root, QName returnType) 方法，代码如下：

// XPathParser.java  
  
/\*\*  
 \* 获得指定元素或节点的值  
 \*  
 \* @param expression 表达式  
 \* @param root 指定节点  
 \* @param returnType 返回类型  
 \* @return 值  
 \*/  
private Object evaluate(String expression, Object root, QName returnType) {  
 try {  
 return xpath.evaluate(expression, root, returnType);  
 } catch (Exception e) {  
 throw new BuilderException("Error evaluating XPath. Cause: " + e, e);  
 }  
}

调用 xpath 的 evaluate(String expression, Object root, QName returnType) 方法，获得指定元素或节点的值。

#### 2.2.1 eval 元素

eval 元素的方法，用于获得 Boolean、Short、Integer、Long、Float、Double、String 类型的元素的值。我们以 #evalString(Object root, String expression) 方法为例子，代码如下：

// XPathParser.java  
  
public String evalString(Object root, String expression) {  
 // <1> 获得值  
 String result = (String) evaluate(expression, root, XPathConstants.STRING);  
 // <2> 基于 variables 替换动态值，如果 result 为动态值  
 result = PropertyParser.parse(result, variables);  
 return result;  
}

<1> 处，调用 #evaluate(String expression, Object root, QName returnType) 方法，获得值。其中，returnType 方法传入的是 XPathConstants.STRING ，表示返回的值是 String 类型。

<2> 处，调用 PropertyParser#parse(String string, Properties variables) 方法，基于 variables 替换动态值，如果 result 为动态值。这就是 MyBatis 如何替换掉 XML 中的动态值实现的方式。关于 PropertyParser ，我们会在 「5. PropertyParser」 详细解析。

#### 2.2.2 eval 节点

eval 元素的方法，用于获得 Node 类型的节点的值。代码如下：

// XPathParser.java  
  
public List<XNode> evalNodes(String expression) { // Node 数组  
 return evalNodes(document, expression);  
}  
  
public List<XNode> evalNodes(Object root, String expression) { // Node 数组  
 // <1> 获得 Node 数组  
 NodeList nodes = (NodeList) evaluate(expression, root, XPathConstants.NODESET);  
 // <2> 封装成 XNode 数组  
 List<XNode> xnodes = new ArrayList<>();  
 for (int i = 0; i < nodes.getLength(); i++) {  
 xnodes.add(new XNode(this, nodes.item(i), variables));  
 }  
 return xnodes;  
}  
  
public XNode evalNode(String expression) { // Node 对象  
 return evalNode(document, expression);  
}  
  
public XNode evalNode(Object root, String expression) { // Node 对象  
 // <1> 获得 Node 对象  
 Node node = (Node) evaluate(expression, root, XPathConstants.NODE);  
 if (node == null) {  
 return null;  
 }  
 // <2> 封装成 XNode 对象  
 return new XNode(this, node, variables);  
}

<1> 处，返回结果有 Node 对象和数组两种情况，根据方法参数 expression 需要获取的节点不同。

<2> 处， 最终结果会将 Node 封装成 org.apache.ibatis.parsing.XNode 对象，主要为了动态值的替换。例如：

|  |
| --- |
| // XNode.java  public String evalString(String expression) {  return xpathParser.evalString(node, expression); } |

其它方法，就不详细解析。感兴趣的胖友，可以自己翻翻。

## XMLMapperEntityResolver

org.apache.ibatis.builder.xml.XMLMapperEntityResolver ，实现 EntityResolver 接口，MyBatis 自定义 EntityResolver 实现类，用于加载本地的 mybatis-3-config.dtd 和 mybatis-3-mapper.dtd 这两个 DTD 文件。代码如下：

// XMLMapperEntityResolver.java  
  
public class XMLMapperEntityResolver implements EntityResolver {  
  
 private static final String IBATIS\_CONFIG\_SYSTEM = "ibatis-3-config.dtd";  
 private static final String IBATIS\_MAPPER\_SYSTEM = "ibatis-3-mapper.dtd";  
 private static final String MYBATIS\_CONFIG\_SYSTEM = "mybatis-3-config.dtd";  
 private static final String MYBATIS\_MAPPER\_SYSTEM = "mybatis-3-mapper.dtd";  
  
 /\*\*  
 \* 本地 mybatis-config.dtd 文件  
 \*/  
 private static final String MYBATIS\_CONFIG\_DTD = "org/apache/ibatis/builder/xml/mybatis-3-config.dtd";  
 /\*\*  
 \* 本地 mybatis-mapper.dtd 文件  
 \*/  
 private static final String MYBATIS\_MAPPER\_DTD = "org/apache/ibatis/builder/xml/mybatis-3-mapper.dtd";  
  
 /\*\*  
 \* Converts a public DTD into a local one  
 \*  
 \* @param publicId The public id that is what comes after "PUBLIC"  
 \* @param systemId The system id that is what comes after the public id.  
 \* @return The InputSource for the DTD  
 \*  
 \* @throws org.xml.sax.SAXException If anything goes wrong  
 \*/  
 @Override  
 public InputSource resolveEntity(String publicId, String systemId) throws SAXException {  
 try {  
 if (systemId != null) {  
 String lowerCaseSystemId = systemId.toLowerCase(Locale.ENGLISH);  
 // 本地 mybatis-config.dtd 文件  
 if (lowerCaseSystemId.contains(MYBATIS\_CONFIG\_SYSTEM) || lowerCaseSystemId.contains(IBATIS\_CONFIG\_SYSTEM)) {  
 return getInputSource(MYBATIS\_CONFIG\_DTD, publicId, systemId);  
 // 本地 mybatis-mapper.dtd 文件  
 } else if (lowerCaseSystemId.contains(MYBATIS\_MAPPER\_SYSTEM) || lowerCaseSystemId.contains(IBATIS\_MAPPER\_SYSTEM)) {  
 return getInputSource(MYBATIS\_MAPPER\_DTD, publicId, systemId);  
 }  
 }  
 return null;  
 } catch (Exception e) {  
 throw new SAXException(e.toString());  
 }  
 }  
  
 private InputSource getInputSource(String path, String publicId, String systemId) {  
 InputSource source = null;  
 if (path != null) {  
 try {  
 // 创建 InputSource 对象  
 InputStream in = Resources.getResourceAsStream(path);  
 source = new InputSource(in);  
 // 设置 publicId、systemId 属性  
 source.setPublicId(publicId);  
 source.setSystemId(systemId);  
 } catch (IOException e) {  
 // ignore, null is ok  
 }  
 }  
 return source;  
 }  
  
}

代码比较简单，胖友自己瞅瞅哈。

## 4. GenericTokenParser

org.apache.ibatis.parsing.GenericTokenParser ，通用的 Token 解析器。代码如下：

// GenericTokenParser.java  
  
public class GenericTokenParser {  
  
 /\*\*  
 \* 开始的 Token 字符串  
 \*/  
 private final String openToken;  
 /\*\*  
 \* 结束的 Token 字符串  
 \*/  
 private final String closeToken;  
 private final TokenHandler handler;  
  
 public GenericTokenParser(String openToken, String closeToken, TokenHandler handler) {  
 this.openToken = openToken;  
 this.closeToken = closeToken;  
 this.handler = handler;  
 }  
  
 public String parse(String text) {  
 if (text == null || text.isEmpty()) {  
 return "";  
 }  
 // search open token  
 // 寻找开始的 openToken 的位置  
 int start = text.indexOf(openToken, 0);  
 if (start == -1) { // 找不到，直接返回  
 return text;  
 }  
 char[] src = text.toCharArray();  
 int offset = 0; // 起始查找位置  
 // 结果  
 final StringBuilder builder = new StringBuilder();  
 StringBuilder expression = null; // 匹配到 openToken 和 closeToken 之间的表达式  
 // 循环匹配  
 while (start > -1) {  
 // 转义字符  
 if (start > 0 && src[start - 1] == '\\') {  
 // this open token is escaped. remove the backslash and continue.  
 // 因为 openToken 前面一个位置是 \ 转义字符，所以忽略 \  
 // 添加 [offset, start - offset - 1] 和 openToken 的内容，添加到 builder 中  
 builder.append(src, offset, start - offset - 1).append(openToken);  
 // 修改 offset  
 offset = start + openToken.length();  
 // 非转义字符  
 } else {  
 // found open token. let's search close token.  
 // 创建/重置 expression 对象  
 if (expression == null) {  
 expression = new StringBuilder();  
 } else {  
 expression.setLength(0);  
 }  
 // 添加 offset 和 openToken 之间的内容，添加到 builder 中  
 builder.append(src, offset, start - offset);  
 // 修改 offset  
 offset = start + openToken.length();  
 // 寻找结束的 closeToken 的位置  
 int end = text.indexOf(closeToken, offset);  
 while (end > -1) {  
 // 转义  
 if (end > offset && src[end - 1] == '\\') {  
 // this close token is escaped. remove the backslash and continue.  
 // 因为 endToken 前面一个位置是 \ 转义字符，所以忽略 \  
 // 添加 [offset, end - offset - 1] 和 endToken 的内容，添加到 builder 中  
 expression.append(src, offset, end - offset - 1).append(closeToken);  
 // 修改 offset  
 offset = end + closeToken.length();  
 // 继续，寻找结束的 closeToken 的位置  
 end = text.indexOf(closeToken, offset);  
 // 非转义  
 } else {  
 // 添加 [offset, end - offset] 的内容，添加到 builder 中  
 expression.append(src, offset, end - offset);  
 break;  
 }  
 }  
 // 拼接内容  
 if (end == -1) {  
 // close token was not found.  
 // closeToken 未找到，直接拼接  
 builder.append(src, start, src.length - start);  
 // 修改 offset  
 offset = src.length;  
 } else {  
 // <x> closeToken 找到，将 expression 提交给 handler 处理 ，并将处理结果添加到 builder 中  
 builder.append(handler.handleToken(expression.toString()));  
 // 修改 offset  
 offset = end + closeToken.length();  
 }  
 }  
 // 继续，寻找开始的 openToken 的位置  
 start = text.indexOf(openToken, offset);  
 }  
 // 拼接剩余的部分  
 if (offset < src.length) {  
 builder.append(src, offset, src.length - offset);  
 }  
 return builder.toString();  
 }  
  
}

代码看起来好冗长，但是淡定，就一个 #parse(String text) 方法，循环( 因为可能不只一个 )，解析以 openToken 开始，以 closeToken 结束的 Token ，并提交给 handler 进行处理，即 <x> 处。

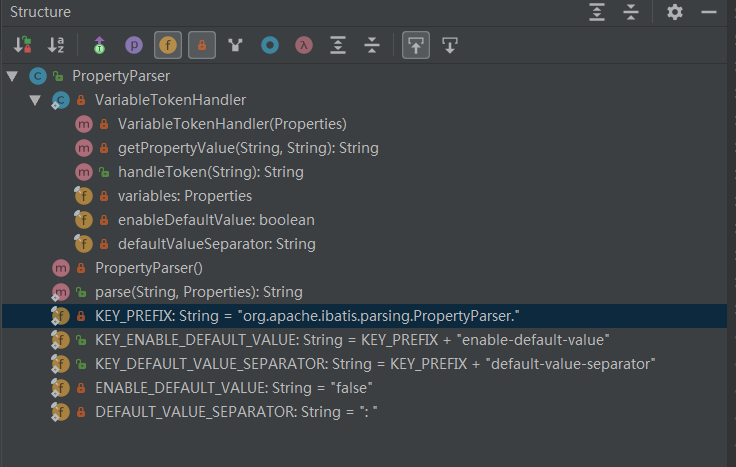
所以所以所以，胖友可以耐心看下这段逻辑，也可以忽略，大体理解就好。

关于 handler 这个 TokenHandler ，详细见 「5. TokenHandler」 。当然，这也是为什么 GenericTokenParser 叫做通用的原因，而 TokenHandler 处理特定的逻辑。

**理解结合=====>**：**GenericTokenParserTest**

## 5.PropertyParser

org.apache.ibatis.parsing.PropertyParser ，动态属性解析器。代码如下：



// PropertyParser.java  
  
public class PropertyParser {  
  
 // ... 省略部分无关的  
  
 private PropertyParser() { // <1>  
 // Prevent Instantiation  
 }  
  
 public static String parse(String string, Properties variables) { // <2>  
 // <2.1> 创建 VariableTokenHandler 对象  
 VariableTokenHandler handler = new VariableTokenHandler(variables);  
 // <2.2> 创建 GenericTokenParser 对象  
 GenericTokenParser parser = new GenericTokenParser("${", "}", handler);  
 // <2.3> 执行解析  
 return parser.parse(string);  
 }  
   
}

<1> ，构造方法，修饰符为 private ，禁止构造 PropertyParser 对象，因为它是一个静态方法的工具类。

<2> ，基于 variables 变量，替换 string 字符串中的动态属性，并返回结果。

<2.1> ，创建 VariableTokenHandler 对象。

<2.2> ，创建 GenericTokenParser 对象。

我们可以看到，openToken = { ，closeToken = } ，这不就是上面看到的 ${username} 和 {password} 的么。

同时，我们也可以看到，handler 类型为 VariableTokenHandler ，也就是说，通过它实现自定义的处理逻辑。关于它，在 「6.1 VariableTokenHandler」 中详细解析。

<2.3> ，调用 GenericTokenParser#parse(String text) 方法，执行解析。

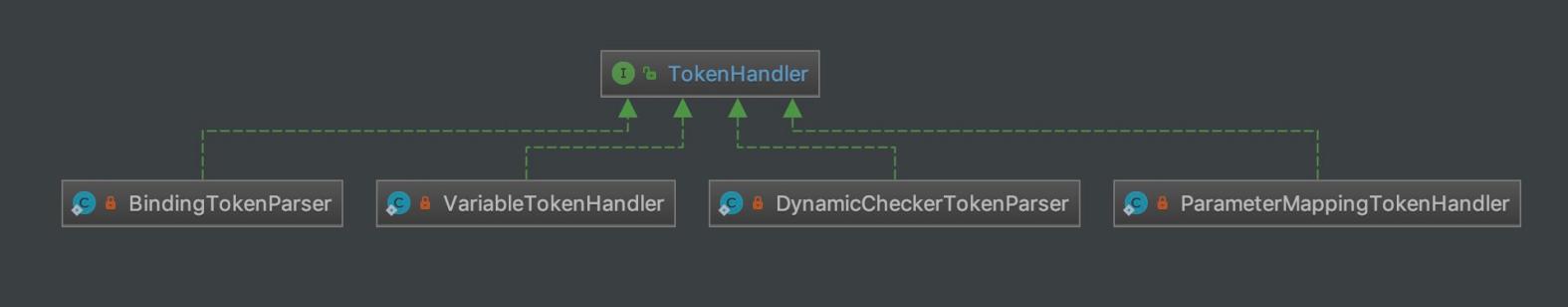
## 6. TokenHandler

org.apache.ibatis.parsing.TokenHandler ，Token 处理器接口。代码如下：

// TokenHandler.java  
  
public interface TokenHandler {  
  
 /\*\*  
 \* 处理 Token  
 \*  
 \* @param content Token 字符串  
 \* @return 处理后的结果  
 \*/  
 String handleToken(String content);  
  
}

#handleToken(String content) 方法，处理 Token ，在 「4. GenericTokenParser」 中，我们已经看到它的调用了。

TokenHandler 有四个子类实现，如下图所示：



本文暂时只解析 VariableTokenHandler 类，因为只有它在 parsing 包中，和解析器模块相关。

### 6.1 VariableTokenHandler

VariableTokenHandler ，是 PropertyParser 的内部静态类，变量 Token 处理器。具体什么用途？上面不是已经整的明明白白啦，就不重复解释啦。

#### 6.1.1 构造方法

|  |
| --- |
| // PropertyParser.java  private static final String KEY\_PREFIX = "org.apache.ibatis.parsing.PropertyParser."; /\*\*  \* The special property key that indicate whether enable a default value on placeholder.  \* <p>  \* The default value is {@code false} (indicate disable a default value on placeholder)  \* If you specify the {@code true}, you can specify key and default value on placeholder (e.g. {@code ${db.username:postgres}}).  \* </p>  \* @since 3.4.2  \*/ public static final String KEY\_ENABLE\_DEFAULT\_VALUE = KEY\_PREFIX + "enable-default-value";  /\*\*  \* The special property key that specify a separator for key and default value on placeholder.  \* <p>  \* The default separator is {@code ":"}.  \* </p>  \* @since 3.4.2  \*/ public static final String KEY\_DEFAULT\_VALUE\_SEPARATOR = KEY\_PREFIX + "default-value-separator";  private static final String ENABLE\_DEFAULT\_VALUE = "false"; private static final String DEFAULT\_VALUE\_SEPARATOR = ":";  // VariableTokenHandler 类里  /\*\*  \* 变量 Properties 对象  \*/ private final Properties variables; /\*\*  \* 是否开启默认值功能。默认为 {@link #ENABLE\_DEFAULT\_VALUE}  \*/ private final boolean enableDefaultValue; /\*\*  \* 默认值的分隔符。默认为 {@link #KEY\_DEFAULT\_VALUE\_SEPARATOR} ，即 ":" 。  \*/ private final String defaultValueSeparator;  private VariableTokenHandler(Properties variables) {  this.variables = variables;  this.enableDefaultValue = Boolean.parseBoolean(getPropertyValue(KEY\_ENABLE\_DEFAULT\_VALUE, ENABLE\_DEFAULT\_VALUE));  this.defaultValueSeparator = getPropertyValue(KEY\_DEFAULT\_VALUE\_SEPARATOR, DEFAULT\_VALUE\_SEPARATOR); }  private String getPropertyValue(String key, String defaultValue) {  return (variables == null) ? defaultValue : variables.getProperty(key, defaultValue); } |

虽然看起来有一大坨的变量，但是不要怕。

variables 属性，变量 Properties 对象。

enableDefaultValue 属性，是否开启默认值功能。默认为 ENABLE\_DEFAULT\_VALUE ，即不开启。想要开启，可以配置如下：

<properties resource="org/mybatis/example/config.properties">  
 <!-- ... -->  
 <property name="org.apache.ibatis.parsing.PropertyParser.enable-default-value" value="true"/> <!-- Enable this feature -->  
</properties>

defaultValueSeparator 属性，默认值的分隔符。默认为 KEY\_DEFAULT\_VALUE\_SEPARATOR ，即 ":" 。想要修改，可以配置如下：

|  |
| --- |
| <properties resource="org/mybatis/example/config.properties">  <!-- ... -->  <property name="org.apache.ibatis.parsing.PropertyParser.default-value-separator" value="?:"/> <!-- Change default value of separator --> </properties> |

分隔符被修改成了 ?: 。

什么意思参考这里：<https://www.cnblogs.com/lujiango/p/8630154.html>

#### 6.1.2 handleToken

// VariableTokenHandler 类里  
  
@Override  
public String handleToken(String content) {  
 if (variables != null) {  
 String key = content;  
 // 开启默认值功能  
 if (enableDefaultValue) {  
 // 查找默认值  
 final int separatorIndex = content.indexOf(defaultValueSeparator);  
 String defaultValue = null;  
 if (separatorIndex >= 0) {  
 key = content.substring(0, separatorIndex);  
 defaultValue = content.substring(separatorIndex + defaultValueSeparator.length());  
 }  
 // 有默认值，优先替换，不存在则返回默认值  
 if (defaultValue != null) {  
 return variables.getProperty(key, defaultValue);  
 }  
 }  
 // 未开启默认值功能，直接替换  
 if (variables.containsKey(key)) {  
 return variables.getProperty(key);  
 }  
 }  
 // 无 variables ，直接返回  
 return "${" + content + "}";  
}

## 666. 彩蛋

小文一篇。按照前面的代码统计，我们已经看掉了小 1000 行的代码了。继续搞起。

参考和推荐如下文章：

徐郡明 《MyBatis 技术内幕》 的 「2.1 解析器模块」 小节

祖大俊 《Mybatis3.3.x技术内幕（七）：Mybatis初始化之六个工具》

----<https://my.oschina.net/zudajun/blog/668596>

# 4.1配置文件解析过程

<http://www.tianxiaobo.com/2018/07/20/MyBatis-%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90-%E9%85%8D%E7%BD%AE%E6%96%87%E4%BB%B6%E8%A7%A3%E6%9E%90%E8%BF%87%E7%A8%8B/>

# 4.2 映射文件解析过程

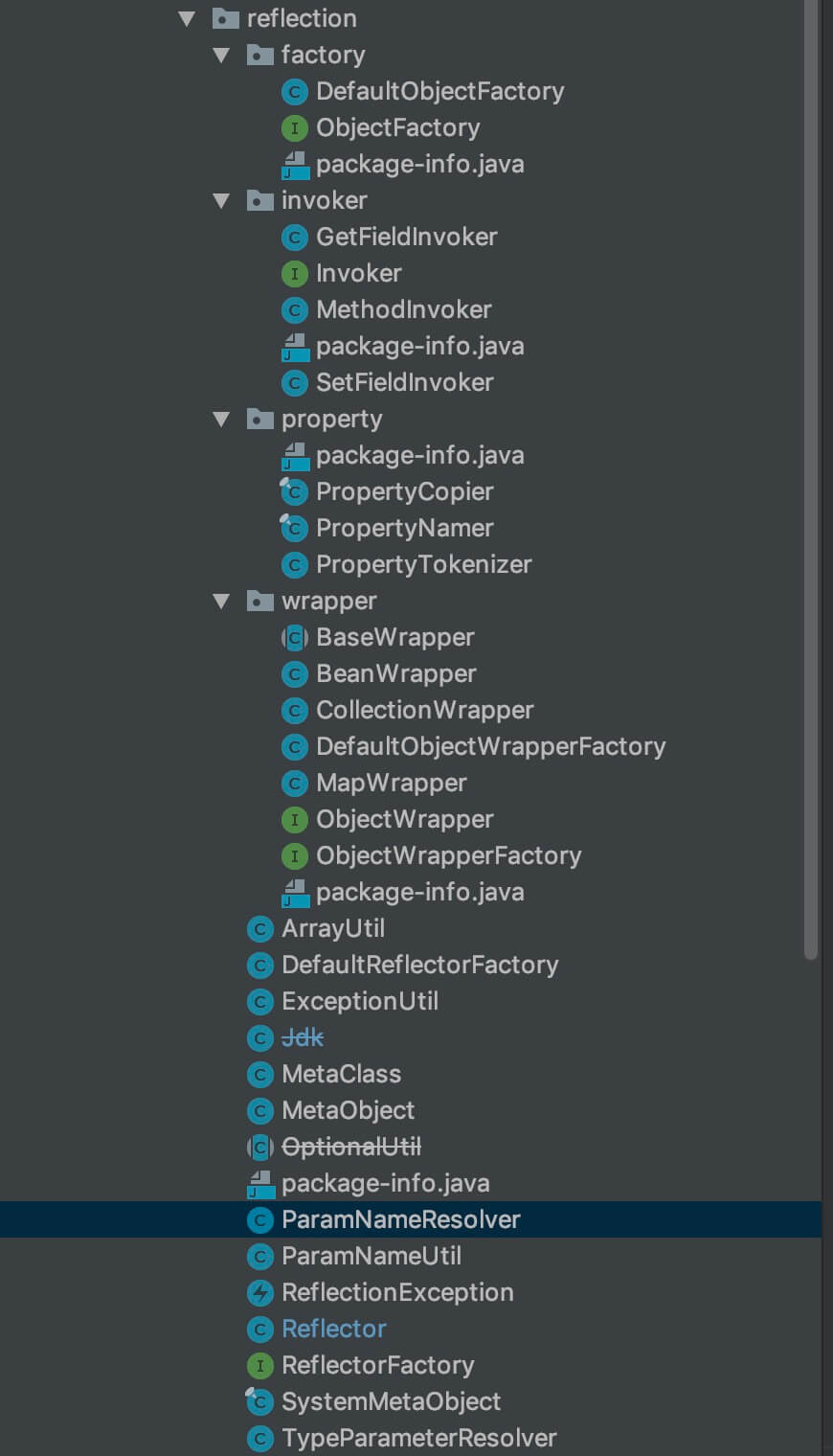
<http://www.tianxiaobo.com/2018/07/30/MyBatis-%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90-%E6%98%A0%E5%B0%84%E6%96%87%E4%BB%B6%E8%A7%A3%E6%9E%90%E8%BF%87%E7%A8%8B/>

# 5.反射模块

java 中的反射虽然功能强大，但对大多数开发人员来说，写出高质量的反射代码还是 有一定难度的。MyBatis 中专门提供了反射模块，该模块对 Java 原生的反射进行了良好的封装，提了更加简洁易用的 API，方便上层使调用，并且对反射操作进行了一系列优化，例如缓存了类的元数据，提高了反射操作的性能。

相比 parsing 包来说，reflection 包的代码量大概是 2-3 倍。当然，不要慌，都是比较简单的代码。

当然，这是一篇非常非常非常长的博客，因为笔者基本贴了这个模块所有的代码。如果想要比较好的理解这个模块，胖友一定要基于 MyBatis 提供的这个模块的单元测试，多多调试哈。



## 2. Reflector

## 3. ReflectorFactory

## 4. Invoker

## 5. ObjectFactory

## 6. Property 工具类

## 7. MetaClass

## 8. ObjectWrapper

## 9. ObjectWrapperFactory

## 10. MetaObject

## 11. SystemMetaObject

## 12. ParamNameUtil

## 13. ParamNameResolver

## 14. TypeParameterResolver

## 15. ArrayUtil

## 16. ExceptionUtil

## 666. 彩蛋

比想象中，长了超级超级超级多的文章。写的都快吐血了，鬼知道我经历了什么！！！

参考和推荐如下文章：

徐郡明 《MyBatis 技术内幕》 的 「2.2 反射工具箱」 小节

祖大俊 《Mybatis3.3.x技术内幕（七）：Mybatis初始化之六个工具》

如果胖友看到此处，还是一脸懵逼，还是那句话，多多调试！！！

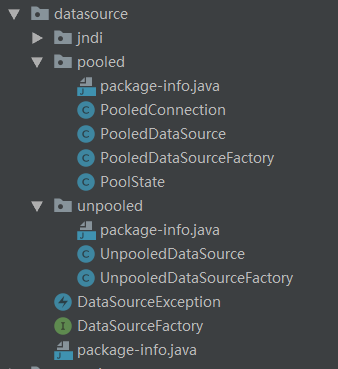
# 6.异常模块

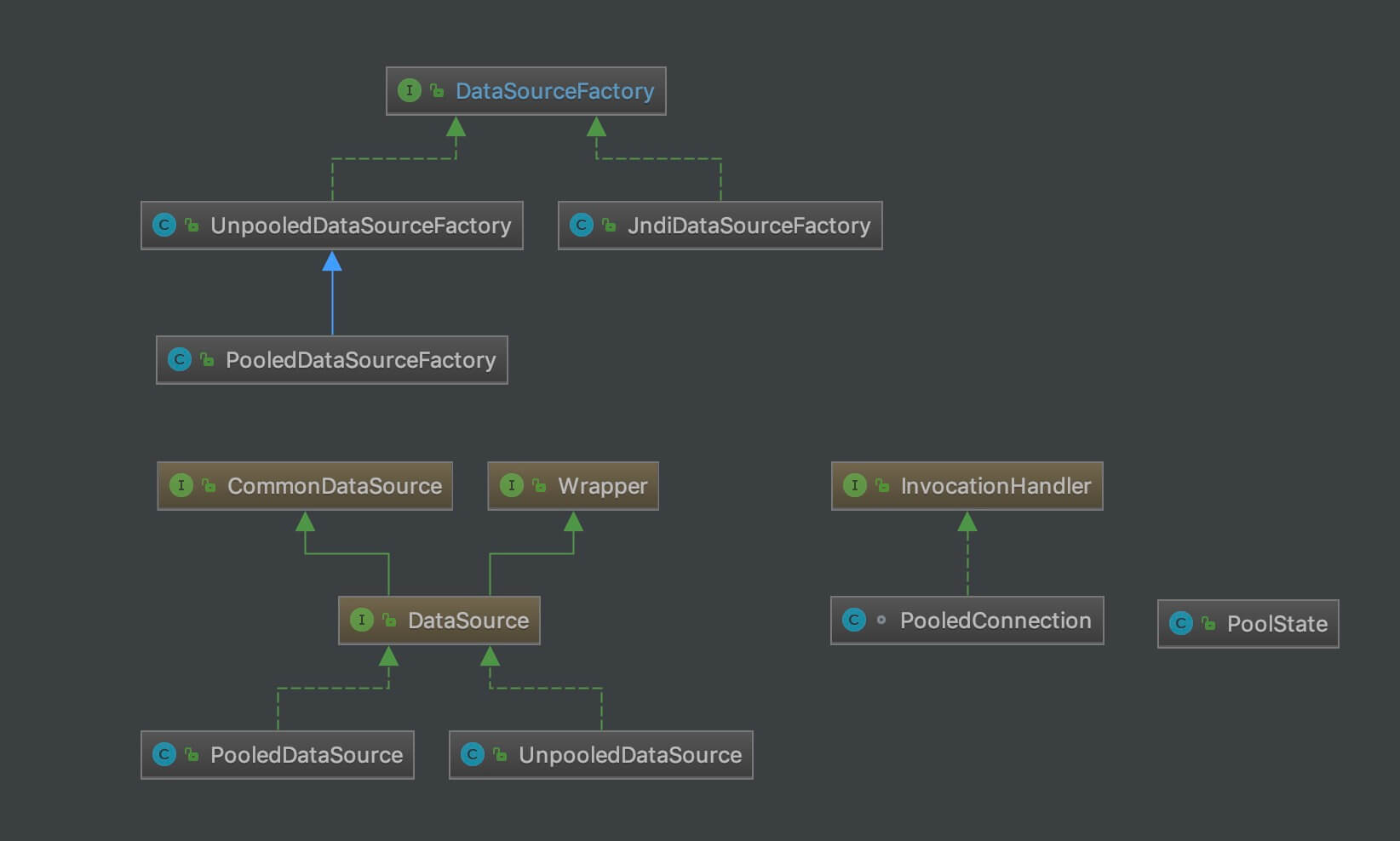
# 7.数据源模块

## 1.概述

数据源是实际开发中常用的组件之一。现在开源的数据源都提供了比较丰富的功能，例如，连接池功能、检测连接状态等，选择性能优秀的数据源组件对于提升 ORM 框架乃至整个应用的性能都是非常重要的。

MyBatis 自身提供了相应的数据源实现，当然 MyBatis 也提供了与第三方数据源集成的接口，这些功能都位于数据源模块之中。





## 2. DataSourceFactory

|  |
| --- |
| public interface DataSourceFactory {   /\*\*  \* 设置 DataSource 对象的属性  \*  \* @param props 属性  \*/  void setProperties(Properties props);   /\*\*  \* 获得 DataSource 对象  \*  \* @return DataSource 对象  \*/  DataSource getDataSource();  } |

### 2.1 UnpooledDataSourceFactory

org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory ，实现 DataSourceFactory 接口，非池化的 DataSourceFactory 实现类。

**UNPOOLED**– 这个数据源的实现只是每次被请求时打开和关闭连接。虽然有点慢，但对于在数据库连接可用性方面没有太高要求的简单应用程序来说，是一个很好的选择。 不同的数据库在性能方面的表现也是不一样的，对于某些数据库来说，使用连接池并不重要，这个配置就很适合这种情形。UNPOOLED 类型的数据源仅仅需要配置以下 5 种属性：

driver – 这是 JDBC 驱动的 Java 类的完全限定名（并不是 JDBC 驱动中可能包含的数据源类）。

url – 这是数据库的 JDBC URL 地址。

username – 登录数据库的用户名。

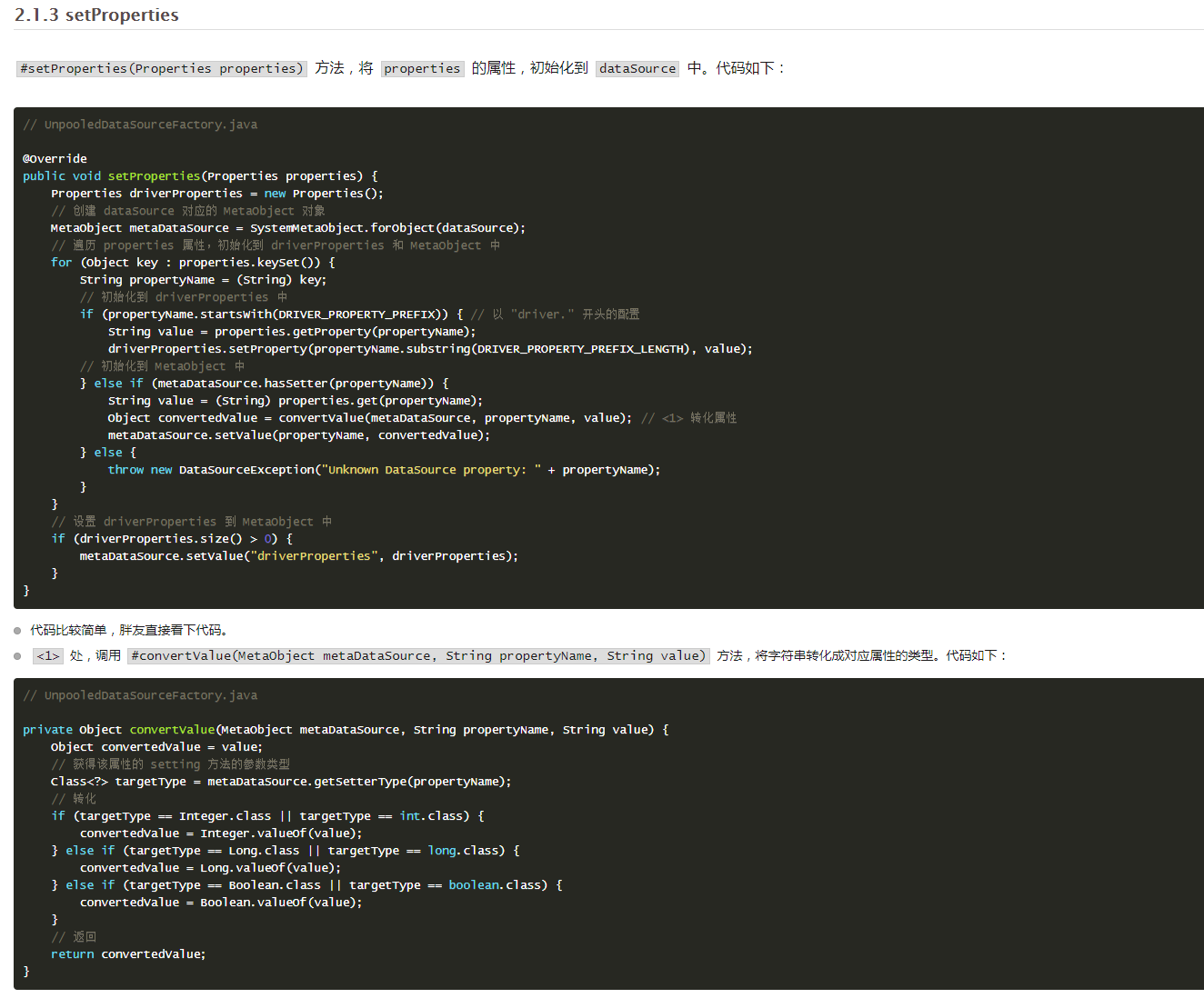
password – 登录数据库的密码。

defaultTransactionIsolationLevel – 默认的连接事务隔离级别。

作为可选项，你也可以传递属性给数据库驱动。要这样做，属性的前缀为“driver.”，例如：

driver.encoding=UTF8

这将通过 DriverManager.getConnection(url,driverProperties) 方法传递值为 UTF8 的 encoding 属性给数据库驱动。



### 2.2 PooledDataSourceFactory

org.apache.ibatis.datasource.pooled.PooledDataSourceFactory ，

继承 UnpooledDataSourceFactory 类，池化的 DataSourceFactory 实现类。

FROM 《MyBatis 文档 —— XML 映射配置文件》

**POOLED**– 这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来，避免了创建新的连接实例时所必需的初始化和认证时间。 这是一种使得并发 Web 应用快速响应请求的流行处理方式。

除了上述提到 UNPOOLED 下的属性外，还有更多属性用来配置 POOLED 的数据源：

poolMaximumActiveConnections – 在任意时间可以存在的活动（也就是正在使用）连接数量，默认值：10

poolMaximumIdleConnections – 任意时间可能存在的空闲连接数。

poolMaximumCheckoutTime – 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）

poolTimeToWait – 这是一个底层设置，如果获取连接花费了相当长的时间，连接池会打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直安静的失败），默认值：20000 毫秒（即 20 秒）。

poolMaximumLocalBadConnectionTolerance – 这是一个关于坏连接容忍度的底层设置， 作用于每一个尝试从缓存池获取连接的线程. 如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 poolMaximumIdleConnections 与 poolMaximumLocalBadConnectionTolerance 之和。 默认值：3 (新增于 3.4.5)

poolPingQuery – 发送到数据库的侦测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动失败时带有一个恰当的错误消息。

poolPingEnabled – 是否启用侦测查询。若开启，需要设置 poolPingQuery 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。

poolPingConnectionsNotUsedFor – 配置 poolPingQuery 的频率。可以被设置为和数据库连接超时时间一样，来避免不必要的侦测，默认值：0（即所有连接每一时刻都被侦测 — 当然仅当 poolPingEnabled 为 true 时适用）。

PooledDataSource 比 UnpooledDataSource 的配置项多很多。

public class PooledDataSourceFactory extends UnpooledDataSourceFactory {  
  
 public PooledDataSourceFactory() {  
 this.dataSource = new PooledDataSource();  
 }  
  
}

默认创建了 PooledDataSource 对象。

其它方法，在父类中 UnpooledDataSourceFactory 中已经实现。所以，真正的**池化**逻辑，在 **PooledDataSource** 对象中。

## 3. DataSource

# 7.1 数据源模块++

## 1.内置数据源

<https://www.tianxiaobo.com/2018/08/19/MyBatis-%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90-%E5%86%85%E7%BD%AE%E6%95%B0%E6%8D%AE%E6%BA%90/>

# 8.事务模块

# 9.缓存模块

# 10.类型模块

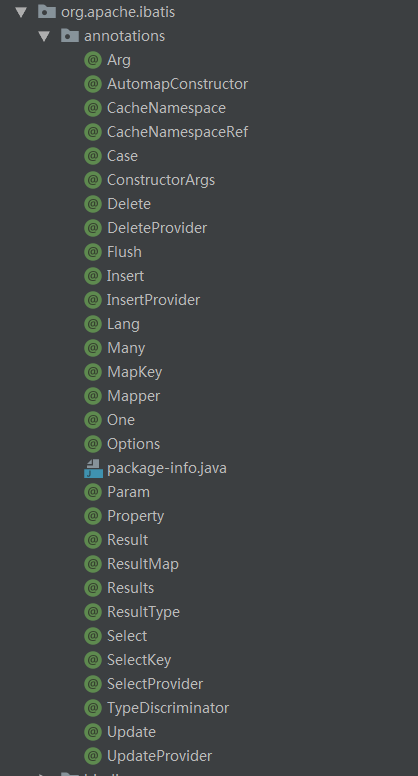
# 11.IO模块

# 12.日志模块

# 13.注解模块

## 1.使用姿势

随着 Java 注解的慢慢流行，MyBatis 提供了注解的方式，使得我们方便的在 Mapper 接口上编写简单的数据库 SQL 操作代码，而无需像之前一样，必须编写 SQL 在 XML 格式的 Mapper 文件中。虽然说，实际场景下，大家还是喜欢在 XML 格式的 Mapper 文件中编写响应的 SQL 操作。



mybatis最初配置信息是基于 XML ,映射语句(SQL)也是定义在 XML 中的。而到了 MyBatis 3提供了新的基于注解的配置。mybatis提供的注解有很多，笔者进行了分类：

**增删改查**：@Insert、@Update、@Delete、@Select、@MapKey、@Options、@SelelctKey、@Param、@InsertProvider、@UpdateProvider、@DeleteProvider、@SelectProvider

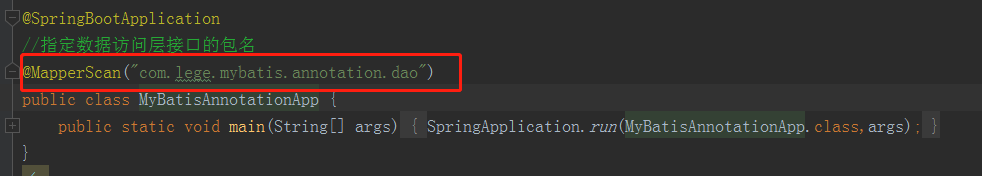
**结果集映射**：@Results、@Result、@ResultMap、@ResultType、@ConstructorArgs、@Arg、@One、@Many、@TypeDiscriminator、@Case、@AutomapConstructor

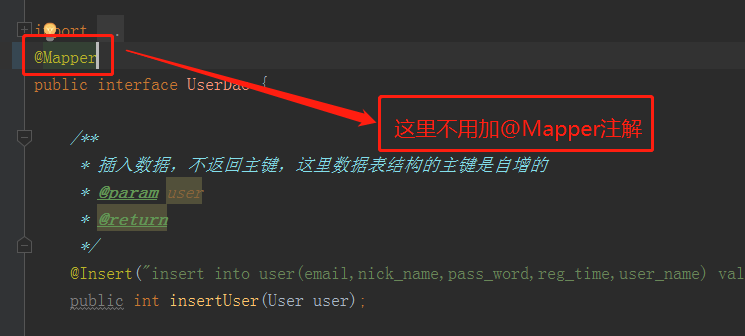
**缓存**：@CacheNamespace、@Property、@CacheNamespaceRef、@Flush

**其他**：@Mapper

绝大部分注解，在xml映射文件中都有元素与之对应，但是不是所有。此外在mybatis-spring中提供了~~@Mapper注解~~和@MapperScan注解，用于和spring进行整合。（Mapper注解我查下来是MyBatis提供的。）

首先解释下：@Mapper注解和@MapperScan注解





如果是结合spring使用，如上可以定位到Mapper接口。而不用在每一个Mapper接口上单独加@Mapper注解。

如果不结合spring使用或者是不使用上面的@MapperScan接口：那么需要在每一个Mapper接口上加上@Mapper注解用来标明这是一个MyBatis Mapper接口。

发现田守枝对注解的整理更好，引用如下：FROM 《mybatis 注解配置详解》

如下就是在说如何用，以及一些注意点，主要来源于田守枝博库

### 1.1 增删改查



**1.@Insert、@Update、@Delete、@Select**

使用对象：方法，

对应的xml：<insert><update><delete><select>

描述：

这四个注解分别代表将会被执行的 SQL 语句。它们用字符串数组（或单个字符串）作为参数。如果传递的是字符串数组，字符串之间先会被填充一个空格再连接成单个完整的字符串。这有效避免了以 Java 代码构建 SQL 语句时的“丢失空格”的问题。然而，你也可以提前手动连接好字符串。属性有：value，填入的值是用来组成单个 SQL 语句的字符串数组。

**2.@Options**

使用对象：方法，

对应的xml：映射语句的属性

描述：

这个注解提供访问大范围的交换和配置选项的入口，它们通常在映射语句上作为属性出现。Options 注解提供了通俗易懂的方式来访问它们，而不是让每条语句注解变复杂。属性有：useCache=true, flushCache=FlushCachePolicy.DEFAULT, resultSetType=FORWARD\_ONLY, statementType=PREPARED, fetchSize=-1, timeout=-1, useGeneratedKeys=false, keyProperty="id", keyColumn="", resultSets=""。值得一提的是， Java 注解无法指定 null 值。因此，一旦你使用了 Options 注解，你的语句就会被上述属性的默认值所影响。要注意避免默认值带来的预期以外的行为。

注意： keyColumn 属性只在某些数据库中有效（如 Oracle、PostgreSQL等）。请在插入语句一节查看更多关于 keyColumn 和 keyProperty 两者的有效值详情。

**3.@MapKey**

使用对象：方法，

对应的xml：

描述：

这是一个用在返回值为 Map 的方法上的注解。它能够将存放对象的 List 转化为 key 值为对象的某一属性的 Map。属性有： value，填入的是对象的属性名，作为 Map 的 key 值。

**4.@SelectKey**

使用对象：方法，

对应的xml：<selectKey>

描述：

这个注解的功能与 <selectKey> 标签完全一致，用在已经被 @Insert 或 @InsertProvider 或 @Update 或 @UpdateProvider 注解了的方法上。若在未被上述四个注解的方法上作 @SelectKey 注解则视为无效。如果你指定了 @SelectKey 注解，那么 MyBatis 就会忽略掉由 @Options 注解所设置的生成主键或设置（configuration）属性。属性有：statement 填入将会被执行的 SQL 字符串数组，keyProperty 填入将会被更新的参数对象的属性的值，before 填入 true 或 false 以指明 SQL 语句应被在插入语句的之前还是之后执行。resultType 填入 keyProperty 的 Java 类型和用 Statement、 PreparedStatement 和 CallableStatement 中的 STATEMENT、 PREPARED 或 CALLABLE 中任一值填入 statementType。默认值是 PREPARED。

**5.@Param**

使用对象：参数，

对应的xml：N/A

描述：

如果你的映射方法的形参有多个，这个注解使用在映射方法的参数上就能为它们取自定义名字。若不给出自定义名字，多参数（不包括 RowBounds 参数）则先以 "param" 作前缀，再加上它们的参数位置作为参数别名。例如 #{param1}, #{param2}，这个是默认值。如果注解是 @Param("person")，那么参数就会被命名为 #{person}。

**6.@InsertProvider、@UpdateProvider、@DeleteProvider、@SelectProvider**

使用对象：方法，

对应的xml：<insert><update><delete><select>

描述：

允许构建动态 SQL。这些备选的 SQL 注解允许你指定类名和返回在运行时执行的 SQL 语句的方法。（自从MyBatis 3.4.6开始，你可以用 CharSequence 代替 String 来返回类型返回值了。）当执行映射语句的时候，MyBatis 会实例化类并执行方法，类和方法就是填入了注解的值。你可以把已经传递给映射方法了的对象作为参数，"Mapper interface type" 和 "Mapper method" 会经过 ProviderContext （仅在MyBatis 3.4.5及以上支持）作为参数值。（MyBatis 3.4及以上的版本，支持多参数传入）属性有： type, method。type 属性需填入类。method 需填入该类定义了的方法名。注意 接下来的小节将会讨论类，能帮助你更轻松地构建动态 SQL。

|  |
| --- |
| // Select.java  @Documented @Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD) // 方法 public @interface Select {   /\*\*  \* @return 查询语句  \*/  String[] value();  } |

映射器接口示例，假设有以下UserMapper接口：

import com.lege.mybatis.annotation.po.User;  
import org.apache.ibatis.annotations.\*;  
  
import java.util.List;  
import java.util.Map;  
  
public interface UserMapper {  
 @Insert("INSERT INTO user(id,name) VALUES (#{id},#{name})")  
 @Options(useGeneratedKeys = true, keyColumn = "id", keyProperty = "id")  
 public int insert(User user);  
  
 @Update(" UPDATE user SET name=#{name} WHERE id=#{id}")  
 public int update(User user);  
  
 @Delete("DELETE FROM user WHERE id=#{id}")  
 public int delete(int id);  
  
 @Select("SELECT id,name FROM user WHERE id= #{id}")  
 public User selectById(int id);  
  
 @Select("SELECT id,name FROM user")  
 public List<User> selectAll();  
  
 @Select("SELECT id,name FROM user")  
 @MapKey("id")  
 public Map<Integer, User> selectMap();  
  
 @Select({"<script>",  
 "SELECT id,name " + "FROM user " + "WHERE id IN "  
 + "<foreach item='id' index='index' collection=‘array' open='(' separator=',' close=')'> "  
 + "#{id}"  
 + "</foreach> ",  
 "</script>"})  
 public List<User> selectByIds(int... ids);  
  
 @Select("SELECT id,name FROM user LIMIT #{offset},#{limit}")  
 public List<User> selectPage(@Param("offset") int offset, @Param("limit") int limit);  
}

说明：

**1、@Insert、@Update、@Delete、@Options、@SelectKey注解**

mybatis会根据接口方法上的@Insert、@Update、@Delete注解，分别去调用SqlSession的insert、update、delete方法。这个几个方法返回的都是一个int，表示影响的记录行数。

特别的，在UserMapper接口的insert方法上，除了添加了@Insert注解，还添加了@Options注解。在上面的案例中，@Options注解用于获取自动生成主键，并设置到User实体中。此外，@SelectKey注解也可以用于获取自动生成的主键，使用方式如下：

@Insert("INSERT INTO user(id,name) VALUES (#{id},#{name})")

@SelectKey(statement = "SELECT LAST\_INSERT\_ID()", keyProperty = "id", before = false, resultType = Integer.class)

public int insert(User user);

**2、@Select注解**

在上面的案例中，UserMapper的selectById、selectAll、selectMap、selectByIds、selectPage方法都添加了@Select注解。mybatis会根据方法的返回值类型User、List<User>、Map<Integer,User>判断是调用SqlSession的selectOne、selectList还是selectMap方法。

**3、@MapKey注解**

特别的，对于返回值是Map的情况，方法上额外添加了一个@MapKey("id")注解，表示将User实例的id属性当做Map的key（将返回结果从List转换成晨Map）。

**4 、动态sql与<script>标签**

在UserMapper的selectByIds方法中，可以看到@Select注解里填写的SQL前后分别添加了<script>、</script>，这是因为SQL中使用了动态sql标签<foreach>。不管是@Insert、@Update、@Delete、@Select注解，只要SQL里使用了mybatis的动态sql标签(包括：if、choose …when ...otherwise、trim 、where、 set、foreach、bind)等，都建议在sql前后分别加上<script>、</script>，否则可能会出现一些参数找不到的情况。

**5、@Param注解**

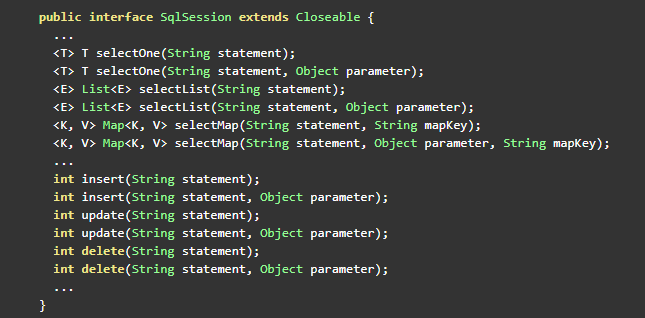
@Param注解用于给方法参数起一个名字。以下是笔者总结的使用原则：

**1.在方法只接受一个参数的情况下，可以不使用@Param**。

**2.在方法接受多个参数的情况下，建议一定要使用@Param注解给参数命名**。

例如上述案例的selectPage方法接受2个参数，所以其两个参数都使用了@Param注解。@Param注解看起来配置最简单，实际上理解确实最复杂，下面进行详细的介绍。

前面已经提到，当映射器接口定义的方法被调用时，mybatis内部根据方法上注解：@Insert、@Update、@Delete、@Select来选择调用SqlSession的insert、update、delete、selectXXX方法。以下是SqlSession接口的相关方法定义(部分省略)：



可以看到部分方法接受一个Object类型的parameter参数，另外一些方法则没有此参数。mybatis在执行前，除了会根据方法上注解：@Insert、@Update、@Delete、@Select来选择调用SqlSession的insert、update、delete、selectXXX方法；还会提前对传入映射器接口方法中的参数进行一些处理，然后再调用SqlSession的相应方法。逻辑如下：

**1)、**如果映射器接口方法不接受参数，mybatis在执行时会调用相应无parameter参数的方法重载形式。例如，如UserMapper接口的selectAll方法，其不接受参数，返回值类型为List，因此调用SqlSession不接受paramter参数的selectList方法：

<E> List<E> selectList(String statement);

**2)、**如果映射器方法只有一个参数，并且这个参数没有使用@Param注解，则直接用这个参数来调用SqlSession相应接受Object类型parameter方法参数的重载形式。例如，如UserMapper接口的selectByIds方法，其接受1个int[]数组类型参数作为查询条件，返回值类型为List<User>，因此调用SqlSession接受paramter参数的selectList方法：

<E> List<E> selectList(String statement, Object parameter);

**3)、**如果映射器方法只接受一个参数，但是使用了@Param注解，也会先封装到Map中；或者映射器方法总是接受多个参数，不管有没有使用@Param注解指定参数，总是会先封装到一个Map中。之后，调用SqlSession的相应方法把这个Map当做parameter参数传入。

例如：UserMapper接口的selectPage方法， 通过@Param("offset”)和@Param("limit”)为2个int参数指定了名字。

@Select("SELECT id,name FROM user LIMIT #{offset},#{limit}")

public List<User> selectPage(@Param("offset") int offset, @Param("limit") int limit);

假设我们传入0和10，那么参数封装后的Map结构如下：

key value

--------------------------

offset 0 //1

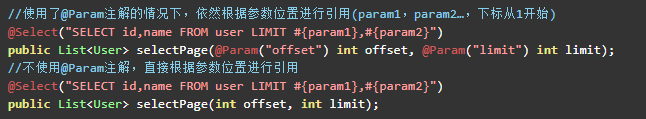
limit 10 //2

param1 0 //3

param2 10 //4

关于Map中1、2两个key-value，比较好理解，是我们通过@Param注解指定的映射关系。而3、4两个key-value，实际上是参数位置(param1、param2…，下标从1开始)和参数值的映射关系，不管我们有没有使用@Param注解，存在多个参数的情况下，我们总是可以按照位置进行引用。

因此，将UserMapper的selectPage方法定义改成以下形式也是正确的：



显然，根据参数位置进行引用不太直观，因此建议在存在多个参数的情况，总是通过@Param注解显式的指定参数名。

**6 动态sql相关注解@InsertProvider、@UpdateProvider、@DeleteProvider、@SelectProvider**

public interface UserBuilderMapper {  
 @SelectProvider(type = UserSqlBuilder.class, method = "buildSelectByIdSql")  
 public User selectById(@Param("id") int id);  
  
 @InsertProvider(type = UserSqlBuilder.class, method = "buildInsertSql")  
 @Options(useGeneratedKeys = true, keyColumn = "id", keyProperty = "id")  
 public int insert(User user);  
  
 @UpdateProvider(type = UserSqlBuilder.class, method = "buildUpdateSql")  
 public int update(User user);  
  
 @UpdateProvider(type = UserSqlBuilder.class, method = "buildUpdateSql2")  
 public int update2(User user);  
  
 @DeleteProvider(type = UserSqlBuilder.class, method = "buildDeleteSql")  
 public int delete(@Param("id") int id);  
  
 //建议将sql builder以映射器接口内部类的形式进行定义  
 public static class UserSqlBuilder {  
 public static String buildSelectByIdSql(@Param("id") int id) {  
 return new SQL() {  
 {  
 SELECT("id, name");  
 FROM("user");  
 WHERE("id=#{id}");  
 }  
 }.toString();  
 }  
 public static String buildInsertSql(User user) {  
 return new SQL() {  
 {  
 INSERT\_INTO("user");  
 VALUES("name", "#{name}");  
 }  
 }.toString();  
 }  
 public static String buildUpdateSql(User user) {  
 return new SQL() {  
 {  
 UPDATE("user");  
 SET("name=#{name}");  
 WHERE("id=#{id}");  
 }  
 }.toString();  
 }  
 public static String buildDeleteSql(@Param("id") int id) {  
 return new SQL() {  
 {  
 DELETE\_FROM("user");  
 WHERE("id=#{id}");  
 }  
 }.toString();  
 }  
 public String findUserById(User user) {  
 return new SQL(){{  
 SELECT("id,name");  
 SELECT("other");  
 FROM("user");  
 if(user.getId()!=null){  
 WHERE("id = #{id}");  
 }  
 if(user.getUserName()!=null){  
 WHERE("name = #{name}");  
 }  
 //从这个toString可以看出，其内部使用高效的StringBuilder实现SQL拼接  
 }}.toString();  
 }  
  
 */\*\*  
 \* 这是一个示例：表示使用的方式姿势  
 \** ***@param*** *user  
 \** ***@return*** *\*/* public String buildUpdateSql2(User user) {  
 SQL sql = new SQL();  
 sql.UPDATE("user");  
 if (user.getEmail() != null) {  
 sql.SET("`email`=#{email}");  
 }  
 sql.SET("`status`=#{status}");  
 if (user.getNickName() != null) {  
 sql.SET("`nickName`=#{nickName}");  
 }  
 sql.SET("`progress`=#{progress}");  
 if (user.getRegTime() != null) {  
 sql.SET("`regTime`=#{regTime}");  
 }  
 sql.WHERE("IDX=#{IDX}");  
  
 return sql.toString();  
 }  
 }  
}

### 1.2 结果集映射相关注解



### 1.3 缓存相关注解



# 13.1 Binding模块

# 14.MyBatis 初始化（一）之加载 mybatis-config

# 15.MyBatis 初始化（二）之加载 Mapper 映射配置文件

# 16.MyBatis 初始化（三）之加载 Statement 配置

# 17.MyBatis 初始化（四）之加载注解配置

# 18.SQL 初始化（上）之 SqlNode

# 19.SQL 初始化（下）之 SqlSource

# 20.SQL 执行（一）之 Executor

# 21.SQL 执行（二）之 StatementHandler

# 22.SQL 执行（三）之 KeyGenerator

# 23.SQL 执行（四）之 ResultSetHandler

# 24.SQL 执行（五）之延迟加载

# 25.会话 SqlSession

# 26.插件体系（一）之原理

# 插件体系（二）之 PageHelper

# 28.Spring 集成（一）之调试环境搭建

# 29.Spring 集成（二）之初始化

# 30.Spring 集成（三）之 SqlSession

# 31.Spring 集成（四）之事务

# 32.Spring 集成（五）之批处理