

# ORM 思想及相关框架实现原理

该课程要求程序员必须已经基本掌握 xml 解析，反射，JDBC，Hibernate，MyBatis，Maven 等相关技术

## 课程大纲

- ORM 思想
- ORM 的经典应用：Hibernate 案例及实现原理
- ORM 的经典应用：MyBatis 案例及实现原理
- 自定义一个 ORM 框架：MiniORM

## 一. ORM 思想

目前，通过 Java 语言连接并操作数据库的技术或方式已经有很多了，例如：JDBC，Hibernate，MyBatis，TopLink 等等。其中 JDBC 是 Java 原生的 API，支持连接并操作各种关系型数据库。相信每个程序员都是从 JDBC 开始学起的，然后才接触到各种持久层框架。

JDBC 作为 Java 原生 API，有优点，也有缺点，这里主要说一下缺点：

1. 编码繁琐，效率低
2. 数据库连接的创建和释放比较重复，也造成了系统资源的浪费
3. 大量硬编码，缺乏灵活性，不利于后期维护
4. 参数的赋值和数据的封装全是手动进行

... ..

可能有些程序员还可以再列出一些 JDBC 的缺点，如果你已经很久没有使用过 JDBC 了，印象已经不深刻了，那么相信下面的代码能勾引起你的些许回忆。

```
public List<Book> findAll() {  
    Connection connection = null;  
    PreparedStatement preparedStatement = null;  
    ResultSet resultSet = null;  
    List<Book> bookList = null;  
  
    try {  
        //加载数据库驱动  
        Class.forName("com.mysql.jdbc.Driver");  
        //通过驱动管理类获取数据库链接  
        connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123");  
        //定义 sql 语句 ?表示占位符  
        String sql = "select * from t_book where author = ?";  
        //获取预处理 statement  
        preparedStatement = connection.prepareStatement(sql);  
        //设置参数，第一个参数为 sql 语句中参数的序号（从 1 开始），第二个参数为设置的参数值  
        preparedStatement.setString(1, "张三");  
    }  
}
```

```
//向数据库发出 sql 执行查询，查询出结果集
resultSet = preparedStatement.executeQuery();
//遍历查询结果集
bookList = new ArrayList<>();
while(resultSet.next()){
    Book book=new Book();
    book.setId(resultSet.getInt("id"));
    book.setName(resultSet.getString("bname"));
    book.setAuthor(resultSet.getString("author"));
    book.setPrice(resultSet.getDouble("price"));
    bookList.add(book);
}
return bookList;
} catch (Exception e) {
    e.printStackTrace();
    return null;
}finally{
    //释放资源
    if(resultSet!=null){
        try {
            resultSet.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    if(preparedStatement!=null){
        try {
            preparedStatement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    if(connection!=null){
        try {
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
}
```

正是因为 JDBC 存在着各种问题，所以才导致很多持久层框架应运而生，例如：Hibernate 和 MyBatis，这两个都是目前比较流行的持久层框架，都对 JDBC 进行了更高级的封装和优化，相信大家对这两个框架都比较熟悉。

很多程序员其实都亲自尝试过自己对 JDBC 进行封装和优化，设计并编写过一些 API，每个程序员在做这个事情时，可能设计以及实现的思想都是不一样的，这些思想各有特点，各有千秋，可以分为两大类：

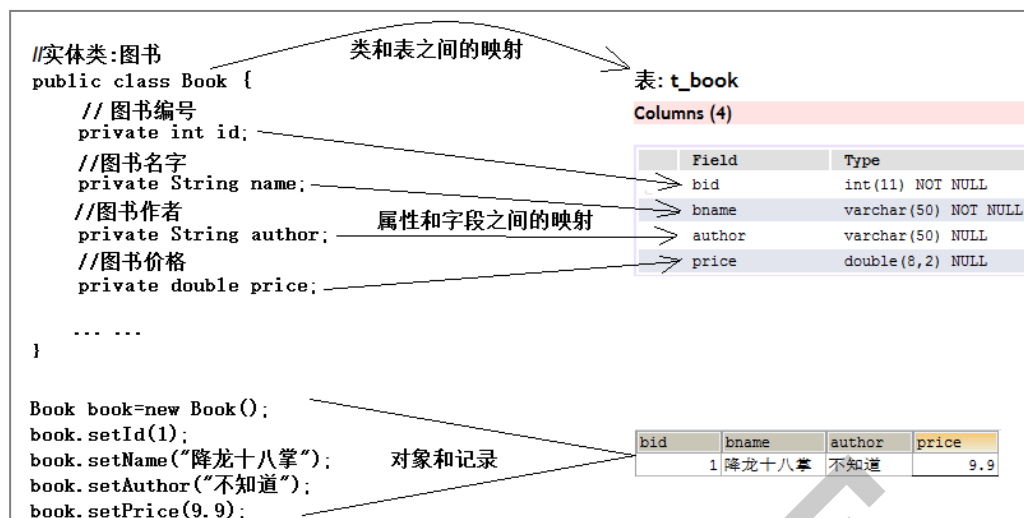
第一类：着重对 JDBC 进行 API 层的抽取和封装，以及功能的增强，典型代表是 Apache 的 DbUtils。



程序员在使用 DbUtils 时仍然需要编写 sql 语句并手动进行数据封装，但是 API 的使用比 JDBC 方便了很多，下面是使用 DbUtils 的代码片段：

```
@Test
public void testQuery(){
    //创建 queryRunner 对象，用来操作 sql 语句
    QueryRunner qr = new QueryRunner(JDBCUtils.getDataSource());
    //编写 sql 语句
    String sql = "select * from user";
    //执行 sql 语句
    try {
        List<User> list = qr.query(sql, new BeanListHandler<User>(User.class));
        System.out.println(list);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

第二类：借鉴面向对象的思想，让程序员以操作对象的方式操作数据库，无需编写 sql 语句，典型代表是 ORM。ORM(Object Relational Mapping)吸收了面向对象的思想，把对 sql 的操作转换为对象的操作，从而让程序员使用起来更加方便和易于接受。这种转换是通过对象和表之间的元数据映射实现的，这是实现 ORM 的关键，如下图所示：



由于类和表之间以及属性和字段之间建立起了映射关系，所以，通过 sql 对表的操作就可以转换为对象的操作，程序员从此无需编写 sql 语句，由框架根据映射关系自动生成，这就是 ORM 思想。

目前比较流行的 Hibernate 和 MyBatis 都采用了 ORM 思想，一般我们把 Hibernate 称之为全自动的 ORM 框架，把 MyBatis 称之为半自动的 ORM 框架。使用过这两个框架的程序员，对于 ORM 一定不会陌生。同时，ORM 也是 JPA(SUN 推出的持久层规范)的核心内容，如下图所示：

## Java Persistence API

### Java Persistence API

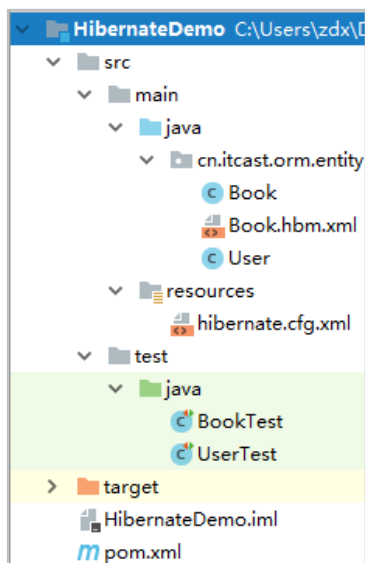
The Java Persistence API provides a POJO persistence model for **object-relational mapping**. The Java Persistence API was developed by the EJB 3.0 software expert group as part of JSR 220, but its use is not limited to EJB software components. It can also be used directly by web applications and application clients, and even outside the Java EE platform, for example, in Java SE applications. See [JSR 220](#).

## 二. ORM 的经典应用：Hibernate

Hibernate 就是应用 ORM 思想建立的一个框架，一般我们把它称之为全自动的 ORM 框架，程序员在使用 Hibernate 时几乎不用编写 sql 语句，而是通过操作对象即可完成对数据库的增删改查。

### 2.1 Hibernate 案例

本次课程不是为了讲解 Hibernate 框架是如何使用的，而是想通过 Hibernate 框架让大家对 ORM 思想有一个更加深入的理解，接下来我们从一个案例开始：



## 1. pom.xml

```
<groupId>cn.itcast.orm</groupId>
<artifactId>HibernateDemo</artifactId>
<version>1.0-SNAPSHOT</version>
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.6.Final</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.36</version>
  </dependency>
</dependencies>
<build>
  <resources>
    <resource>
      <directory>src/main/java</directory>
      <includes> <!--为了编译时能加载包中的 xml 文件-->
        <include>/**/*.xml</include>
      </includes>
      <filtering>false</filtering>
    </resource>
```



```
<resource> <!--为了编译时能加载 resources 中的 xml 文件-->
  <directory>src/main/resources</directory>
  <includes>
    <include>/**/*.xml</include>
  </includes>
  <filtering>false</filtering>
</resource>
</resources>
</build>
```

## 2. Hibernate 核心配置文件 hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.url">jdbc:mysql://localhost:3306/test</property>
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.username">root</property>
    <property name="connection.password">123</property>

    <property name="hbm2ddl.auto">update</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="show_sql">>true</property>

    <mapping class="cn.itcast.orm.entity.Book"/> <!--带有映射注解的实体类-->
    <mapping resource="cn/itcast/orm/entity/Book.hbm.xml"/> <!--映射配置文件-->
  </session-factory>
</hibernate-configuration>
```

该配置文件主要设置了数据库连接信息和映射配置文件的位置信息

## 3. 实体类 Book.java

```
public class Book {
  private int id; //主键
  private String name; //图书名字
  private String author; //图书作者
  private double price; //图书价格
  ... ..
}
```

这就是一个普通的实体类，描述和存储图书信息

## 4. 映射配置文件 Book.hbm.xml

```
<hibernate-mapping>
```

```
<class name="cn.itcast.orm.entity.Book" table="t_book">
    <id name="id">
        <column name="bid"/>
        <generator class="identity"/> <!-- 主键的值采用自增方式-->
    </id>
    <property name="name">
        <column name="bname"/>
    </property>
    <property name="author">
        <column name="author"/>
    </property>
    <property name="price">
        <column name="price"/>
    </property>
</class>
</hibernate-mapping>
```

该配置文件非常关键，重点体现了 ORM 思想，类和表之间，属性和字段之间的映射关系清晰明了，当然现在也非常流行注解的方式，就是把映射关系以注解的方式放在实体类中，如下所示：

```
@Entity
@Table(name = "t_book")
public class Book {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY) //数据库自增
    @Column(name = "bid")
    private Integer id; //主键

    @Column(name="bname")
    private String name; //图书名字

    @Column(name="author")
    private String author; //图书作者

    @Column(name="price")
    private double price; //图书价格

    ... ..
}
```

不管你用 xml 配置的方式，还是注解的方式，其本质都是一样的，都是为了通过 ORM 思想把类和表之间，属性和字段之间的映射关系设置好。

## 5. 测试类

```
public class BookTest {
```



```
private SessionFactory factory;

@Before
public void init(){
    //1. 创建一个 Configuration 对象，解析 hibernate 的核心配置文件
    Configuration cfg=new Configuration().configure();
    //2. 创建 SessinFactory 对象，解析映射信息并生成基本的 sql
    factory=cfg.buildSessionFactory();
}

@Test
public void testSave(){
    //3. 得到 Session 对象，该对象具有增删改查的方法
    Session session=factory.openSession();
    //4. 开启事务
    Transaction tx=session.beginTransaction();

    //5. 保存数据
    Book book=new Book();
    book.setName("java 从入门到精通");
    book.setAuthor("传智播客");
    book.setPrice(9.9);
    session.save(book);

    //6. 提交事务
    tx.commit();

    //7. 释放资源
    session.close();
}

@Test
public void testGet(){
    //3. 得到 Session 对象，该对象具有增删改查的方法
    Session session=factory.openSession();

    //4. 通过 Session 对象进行查询
    Book book=session.get(Book.class,2);
    System.out.println(book);

    //5. 释放资源
    session.close();
}
```



```
@Test
public void testDelete(){
    //3. 得到 Session 对象，该对象具有增删改查的方法
    Session session=factory.openSession();

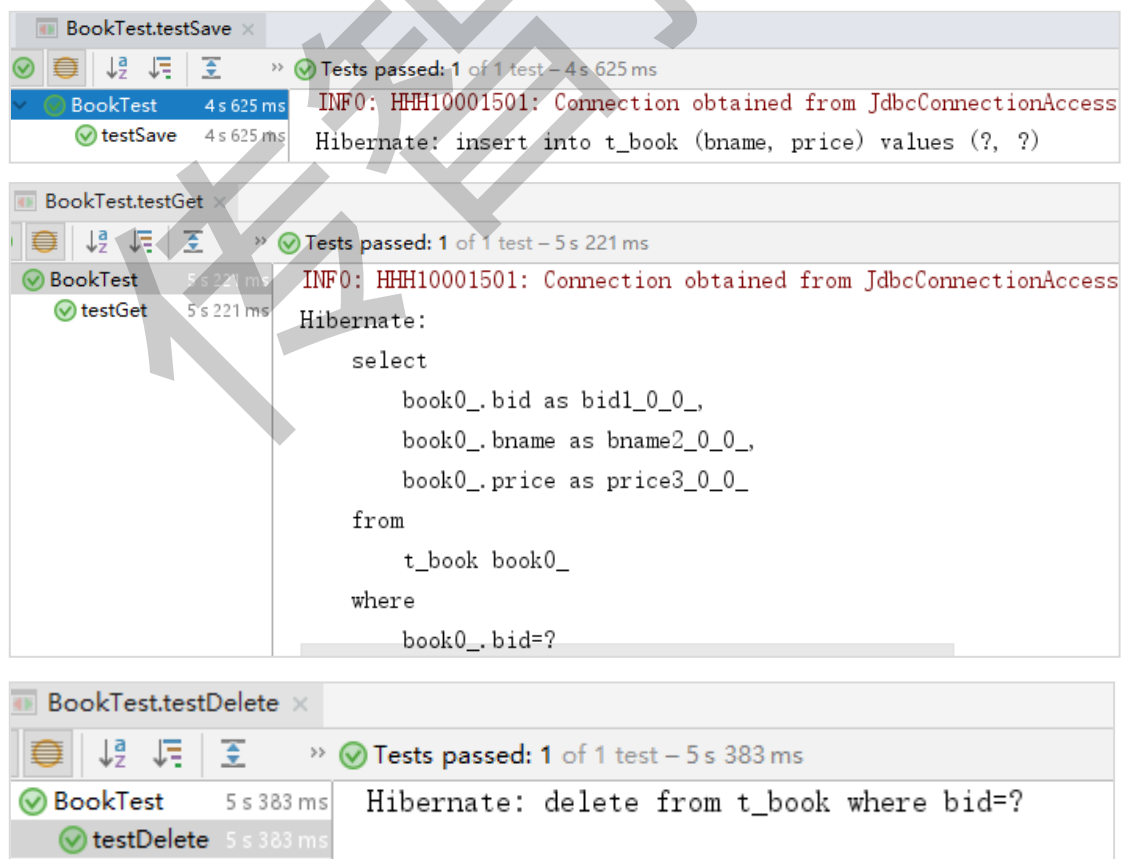
    //4. 开启事务管理
    Transaction tx=session.beginTransaction();

    //5. 删除数据
    Book book=new Book();
    book.setId(2);
    session.delete(book);

    //6. 提交事务
    tx.commit();

    //7. 释放资源
    session.close();
}
```

该测试类使用 Hibernate 的 API 实现了图书的添加，查询和删除功能，程序员无需编写 sql 语句，只需要像平时一样操作对象即可，然后由 Hibernate 框架自动生成 sql 语句，如下图所示：





## 2.2 Hibernate 的 ORM 实现原理

接下来我们通过上述案例来讲解一下 Hibernate 框架是如何应用 ORM 思想的，一起剖析一下 Hibernate 的内部实现原理。

其实不管使用什么框架，最终都需要生成 sql 语句，因为数据库需要的就是 sql 语句，而我们在使用 Hibernate 编码时没有编写 sql 语句，只是提供了对象，那么 Hibernate 是如何根据对象生成 sql 语句的呢？接下来我们一起跟踪并分析一下 Hibernate 5.x 的源码。

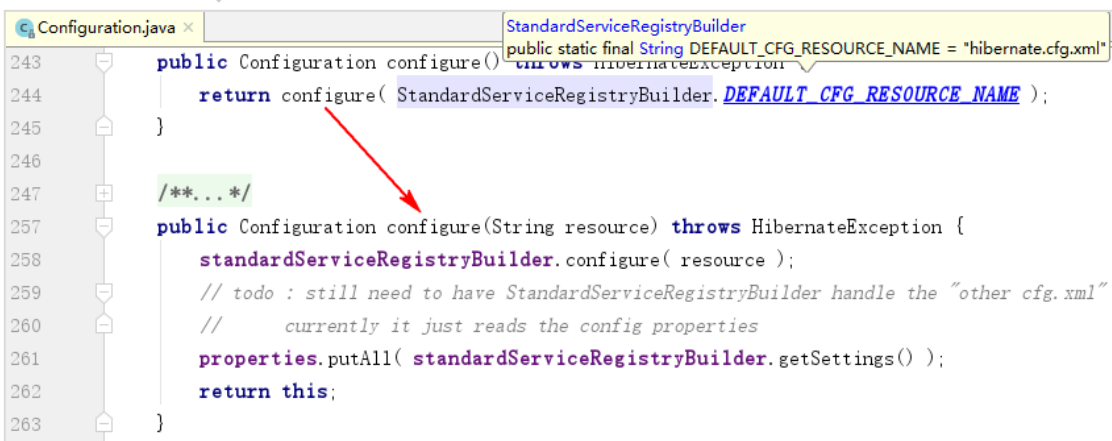
1. Configuration cfg = new Configuration().configure();

1.1 在 new Configuration()时，进行了 hibernate 的环境初始化工作，相关对象和容器被创建了出来，如下图所示：



红框中的两个对象大家要尤为注意，一个是第 158 行的 StandardServiceRegistryBuilder，一个是第 161 行的 Properties 对象，后面的源码中会重点用到这两个对象。

1.2 接下来跟踪调用 configure()方法，如下图所示：



第 244 行的 `configure(...)` 方法会默认加载名字为 `hibernate.cfg.xml` 的配置文件，然后去解析该配置文件并把解析到的数据存放到一个 `Properties` 中(第 258 和 261 行代码)。解析出来的数据如下图所示：

传智播客

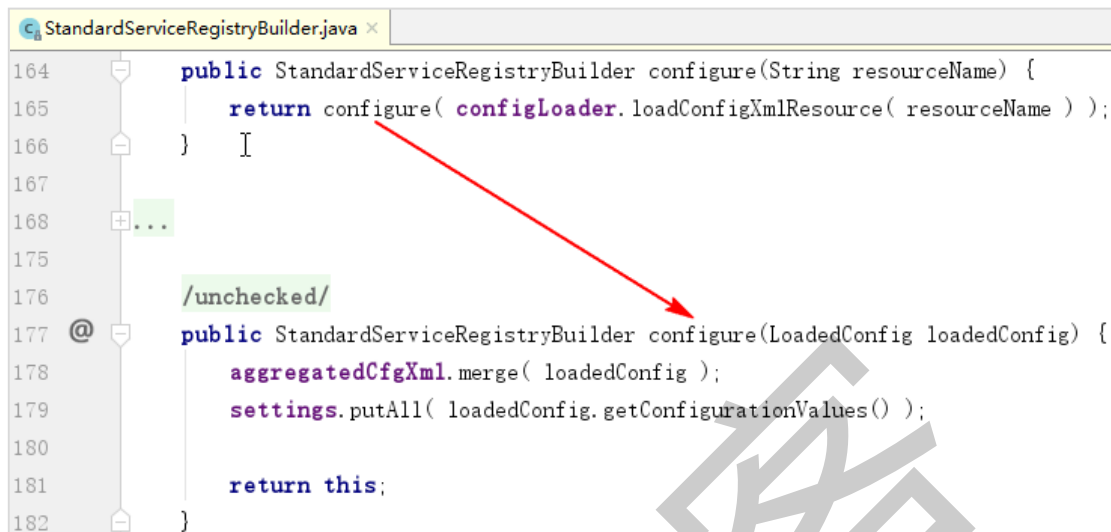


```
Variables
> this
> resource = "hibernate.cfg.xml"
> standardServiceRegistryBuilder
  properties = size = 71
    > 0 = "hibernate.connection.password" -> "123"
    > 1 = "java.runtime.name" -> "Java(TM) SE Runtime Environment"
    > 2 = "sun.boot.library.path" -> "C:\Program Files\Java\jdk1.8.0_101\jre\bin"
    > 3 = "java.vm.version" -> "25.101-b13"
    > 4 = "hibernate.connection.username" -> "root"
    > 5 = "java.vm.vendor" -> "Oracle Corporation"
    > 6 = "java.vendor.url" -> "http://java.oracle.com/"
    > 7 = "path.separator" -> ";"
    > 8 = "java.vm.name" -> "Java HotSpot(TM) 64-Bit Server VM"
    > 9 = "file.encoding.pkg" -> "sun.io"
    > 10 = "user.script" ->
    > 11 = "user.country" -> "CN"
    > 12 = "sun.java.launcher" -> "SUN_STANDARD"
    > 13 = "sun.os.patch.level" ->
    > 14 = "java.vm.specification.name" -> "Java Virtual Machine Specification"
    > 15 = "user.dir" -> "C:\Users\zdx\Desktop\ORMDemo\HibernateDemo"
    > 16 = "java.runtime.version" -> "1.8.0_101-b13"
    > 17 = "java.awt.graphicsenv" -> "sun.awt.Win32GraphicsEnvironment"
    > 18 = "hbm2ddl.auto" -> "update"
    > 45 = "hibernate.show_sql" -> "true"
    > 46 = "java.vm.specification.version" -> "1.8"
    > 47 = "sun.arch.data.model" -> "64"
    > 48 = "java.home" -> "C:\Program Files\Java\jdk1.8.0_101\jre"
    > 49 = "sun.java.command" -> "com.intellij.rt.execution.junit.JUnit4TestRunner"
    > 50 = "hibernate.dialect" -> "org.hibernate.dialect.MySQLDialect"
    > 51 = "hibernate.connection.url" -> "jdbc:mysql://localhost:3306/test"
    > 52 = "java.specification.vendor" -> "Oracle Corporation"
    > 53 = "user.language" -> "zh"
    > 54 = "awt.toolkit" -> "sun.awt.windows.WToolkit"
    > 55 = "java.vm.info" -> "mixed mode"
    > 56 = "java.version" -> "1.8.0_101"
    > 57 = "java.ext.dirs" -> "C:\Program Files\Java\jdk1.8.0_101\jre\lib\ext;"
    > 58 = "sun.boot.class.path" -> "C:\Program Files\Java\jdk1.8.0_101\jre"
    > 59 = "java.vendor" -> "Oracle Corporation"
    > 60 = "connection.driver_class" -> "com.mysql.jdbc.Driver"
    > 61 = "file.separator" -> "\"
    > 62 = "hibernate.hbm2ddl.auto" -> "update"
```

通过上图大家能很清晰得看到，hibernate.cfg.xml 中的信息被解析出来并存到了一个

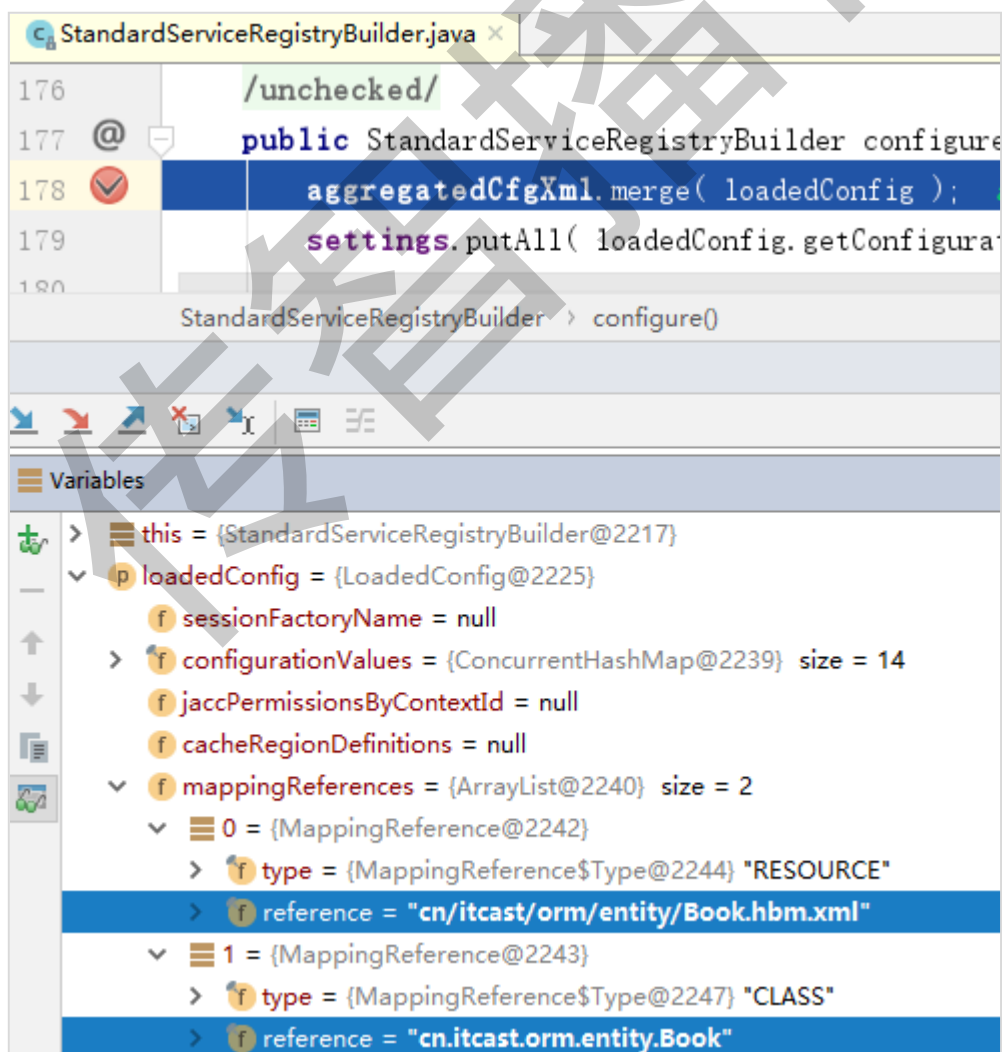
Properties 中。

1.3 我们再跟踪一下第 258 行的代码，这行代码调用了 `StandardServiceRegistryBuilder` 对象的 `config` 方法，如下图所示：



```
164 public StandardServiceRegistryBuilder configure(String resourceName) {
165     return configure( configLoader.loadConfigXmlResource( resourceName ) );
166 }
167
168 ...
175
176 /unchecked/
177 @ public StandardServiceRegistryBuilder configure(LoadedConfig loadedConfig) {
178     aggregatedCfgXml.merge( loadedConfig );
179     settings.putAll( loadedConfig.getConfigurationValues() );
180
181     return this;
182 }
```

第 165 行从 `hibernate.cfg.xml` 中解析出来映射配置文件和实体类的信息，并在第 178 行进行了合并汇总，最终存储到了 `aggregatedCfgXml` 中，如下图所示：



```
176 /unchecked/
177 @ public StandardServiceRegistryBuilder configure
178 aggregatedCfgXml.merge( loadedConfig );
179 settings.putAll( loadedConfig.getConfigurationValues() );
180
```

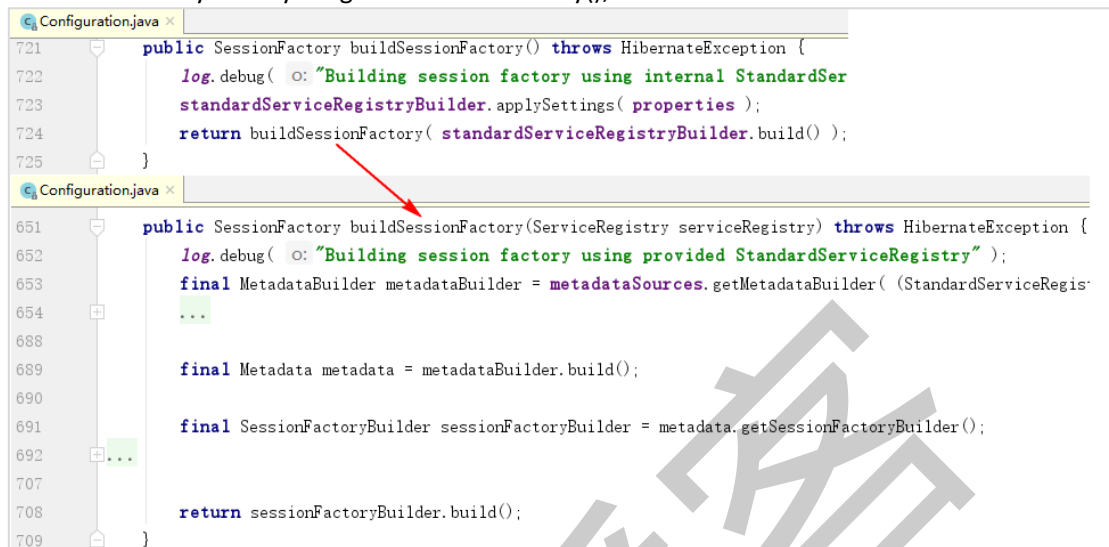
StandardServiceRegistryBuilder > configure()

Variables

- > this = {StandardServiceRegistryBuilder@2217}
- > loadedConfig = {LoadedConfig@2225}
  - sessionFactoryName = null
  - > configurationValues = {ConcurrentHashMap@2239} size = 14
  - jaccPermissionsByContextId = null
  - cacheRegionDefinitions = null
  - > mappingReferences = {ArrayList@2240} size = 2
    - > 0 = {MappingReference@2242}
      - > type = {MappingReference\$Type@2244} "RESOURCE"
      - > reference = "cn/itcast/orm/entity/Book.hbm.xml"
    - > 1 = {MappingReference@2243}
      - > type = {MappingReference\$Type@2247} "CLASS"
      - > reference = "cn.itcast.orm.entity.Book"

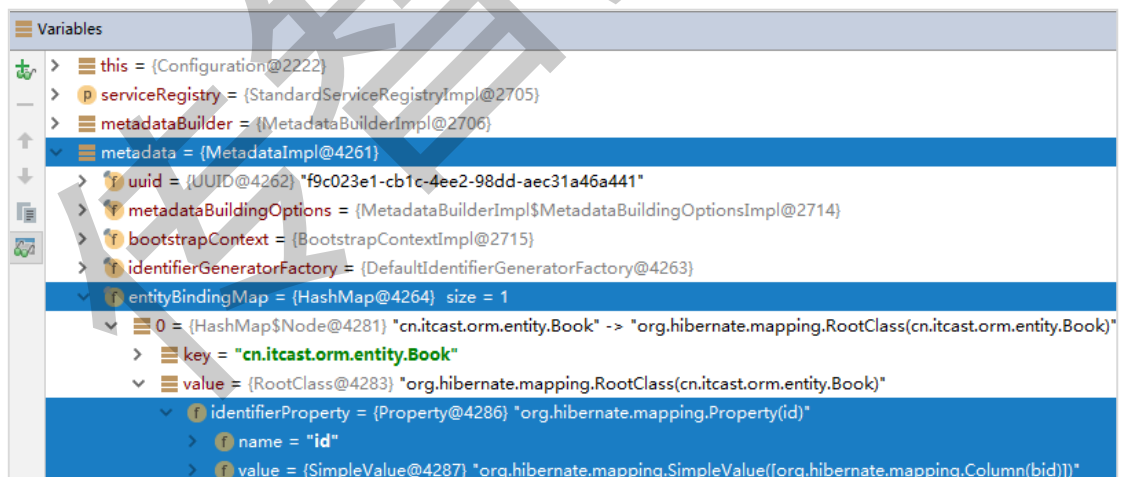
总之，第一步 `Configuration cfg = new Configuration().configure();` 已经把 `hibernate.cfg.xml` 中的信息全部解析了出来并进行了存储。

## 2. SessionFactory factory = cfg.buildSessionFactory();



由 `Configuration` 对象的 `buildSessionFactory(...)` 方法创建会话工厂（`SessionFactory`），该方法的代码非常多，这里只截取了部分关键代码：

- 第 723 行代码从 `Properties` 中得到配置文件中的数据
- 第 653 行和第 689 行分别创建 `MetadataBuilder` 对象并调用该对象的 `build()` 方法解析了映射信息，然后存储到 `Metadata` 对象中，下列截图展示了从映射配置文件或实体类中解析出来的映射数据，包含哪个类和哪个表对应，哪个属性和哪个字段对应







```
Variables
{
  f where = null
  > f table = (Table@4288) *org.hibernate.mapping.Table(t_book)*
  f discriminatorInsertable = true
  f nextSubclassId = 0
  > f declaredIdentifierProperty = (Property@4286) *org.hibernate.mapping.Property(id)*
  f declaredVersion = null
  > f metadataBuildingContext = (MetadataBuildingContextRootImpl@4289)
  f entityName = "cn.itcast.orm.entity.Book"
  f className = "cn.itcast.orm.entity.Book"
  f mappedClass = null
  > f proxyInterfaceName = "cn.itcast.orm.entity.Book"
  f proxyInterface = null
  > f jpaEntityName = "Book"
  > f discriminatorValue = "cn.itcast.orm.entity.Book"
  f lazy = true
  > f properties = (ArrayList@4291) size = 2
  > 0 = (Property@4320) *org.hibernate.mapping.Property(name)*
  > f name = "name"
  > f value = (SimpleValue@4345) *org.hibernate.mapping.SimpleValue([org.hibernate.mapping.Column(bname)])*
}

Variables
{
  f mappedClass = null
  > f proxyInterfaceName = "cn.itcast.orm.entity.Book"
  f proxyInterface = null
  > f jpaEntityName = "Book"
  > f discriminatorValue = "cn.itcast.orm.entity.Book"
  f lazy = true
  > f properties = (ArrayList@4291) size = 2
  > 0 = (Property@4320) *org.hibernate.mapping.Property(name)*
  > 1 = (Property@4321) *org.hibernate.mapping.Property(price)*
  > f name = "price"
  > f value = (SimpleValue@4350) *org.hibernate.mapping.SimpleValue([org.hibernate.mapping.Column(price)])*
}
```

- 第 708 行调用 SessionFactoryBuilder 对象的 build() 方法生成 sql 语句并返回 SessionFactory 实例，下面截图展示出了生成的 sql 语句：

```
Variables
{
  > factory = (SessionFactoryImpl@5196)
  > f name = null
  > f uuid = "a2a7b404-4cf9-49c6-95c8-2160e2cbb23e"
  f isClosed = false
  > f observer = (SessionFactoryObserverChain@5211)
  > f sessionFactoryOptions = (SessionFactoryOptionsBuilder@5212)
  > f settings = (Settings@5213)
  > f properties = (HashMap@5214) size = 72
  > f serviceRegistry = (SessionFactoryServiceRegistryImpl@5215)
  > f jdbcServices = (JdbcServicesImpl@5216)
  > f sqlFunctionRegistry = (SQLFunctionRegistry@5217)
  > f metamodel = (MetamodelImpl@5218)
  > f sessionFactory = (SessionFactoryImpl@5196)
  > f imports = (ConcurrentHashMap@5228) size = 2
  > f entityPersisterMap = (ConcurrentHashMap@5229) size = 1
  > 0 = (ConcurrentHashMap$MapEntry@5257) "cn.itcast.orm.entity.Book" -> "SingleTableEntityPersister"
  > key = "cn.itcast.orm.entity.Book"
  > value = (SingleTableEntityPersister@5258) "SingleTableEntityPersister(cn.itcast.orm.entity.Book)"
  > f loaders = (ConcurrentHashMap@5343) size = 2
  > f sqlVersionSelectString = "select bid from t_book where bid =?"
  > f sqlSnapshotSelectString = "select book_bid, book_bname as bname2_0, bo
  > f sqlLazySelectStringsByFetchGroup = (Collections$EmptyMap@5346) size = 0
  > f sqlIdentityInsertString = "insert into t_book (bname, price) values (?, ?)"
}
```

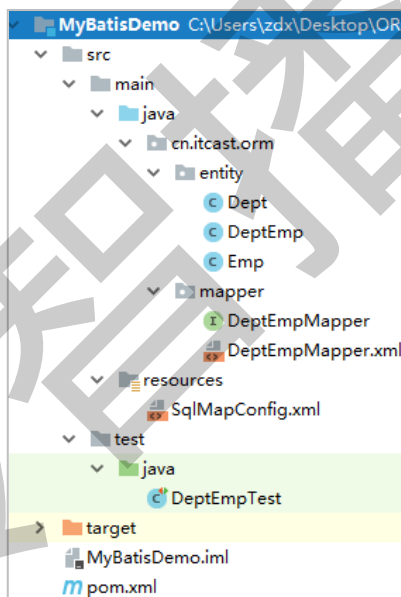
会话工厂（SessionFactory）在 Hibernate 中实际上起到了一个缓冲区的作用，它缓存了 Hibernate 自动生成的 SQL 语句和相关映射数据，只要生成了 sql 语句，那么后面实现增删改查就不在话下。

## 三. ORM 的经典应用：MyBatis

MyBatis 框架也应用了 ORM 思想，一般我们把它称之为半自动的 ORM 框架，跟 Hibernate 相比，MyBatis 更加轻量，更加灵活，为了保证这一点，程序员在使用 MyBatis 时需要自己编写 sql 语句，但是 API 的使用依然像 Hibernate 一样简单方便。

### 3.1 MyBatis 案例

本次课程不是为了讲解 MyBatis 框架是如何使用的，而是想通过 MyBatis 框架让大家对 ORM 思想有一个更加深入的理解，接下来我们从一个案例开始：



#### 1. pom.xml

```
<groupId>cn.itcast.orm</groupId>
<artifactId>MyBatisDemo</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
  <dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.0</version>
  </dependency>
  <dependency>
```





```
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.36</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
</dependencies>
<build>
    <resources>
    <resource>
        <directory>src/main/java</directory>
        <includes> <!-- 为了编译时能加载包中的 xml 文件-->
            <include>/**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>/**/*.xml</include>
        </includes>
        <filtering>false</filtering>
    </resource>
    </resources>
</build>
```

## 2. MyBatis 核心配置文件 SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <typeAliases> <!-- 配置实体类别名 -->
        <package name="cn.itcast.orm.entity"/>
    </typeAliases>
    <environments default="development">
        <environment id="development">
            <!-- 使用 jdbc 事务管理-->
            <transactionManager type="JDBC" />
            <!-- 数据库连接信息-->
            <dataSource type="POOLED">
```

```
<property name="driver" value="com.mysql.jdbc.Driver" />
<property name="url" value="jdbc:mysql://localhost:3306/test?characterEncoding=utf-8" />
<property name="username" value="root" />
<property name="password" value="123" />
</dataSource>
</environment>
</environments>
<mappers> <!-- 加载 mapper 文件 -->
    <package name="cn.itcast.orm.mapper"/>
</mappers>
```

该配置文件主要设置了数据库连接信息和映射配置文件的位置信息

### 3. 实体类 DeptEmp.java

```
// 实体类：存储各部门的员工总数
public class DeptEmp {
    private String deptName; //部门名称
    private int total; //员工总数
    ... ..
}
```

这就是一个普通的实体类，用来封装和存储查询出来的数据

### 4. Mapper 接口 DeptEmpMapper.java

```
//Mapper 接口
public interface DeptEmpMapper {
    //查询各部门的员工总数
    List<DeptEmp> getEmpTotalByDept();
}
```

这是 Mapper 接口，我们只需要定义方法，由 MyBatis 框架创建代理类(实现类)并实现功能

### 5. 映射配置文件 DeptEmpMapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.itcast.orm.mapper.DeptEmpMapper">
    <!--自定义 resultMap 配置实体类和结果集之间的映射关系-->
    <resultMap type="DeptEmp" id="dept_emp_result_map">
        <result property="deptName" column="dname"/>
        <result property="total" column="total"/>
    </resultMap>

    <!--定义查询语句-->
    <select id="getEmpTotalByDept" resultMap="dept_emp_result_map">
        select dname, count(*) as total from dept,emp where emp.dept_id=dept.did group by dname
```



```
</select>
</mapper>
```

MyBatis 其实会采用默认规则自动建立表和实体类之间，属性和字段之间的映射关系，但是由于我们这个案例中是要查询各部门的员工总数，无法自动映射，所以我们自定义了一个 `resultMap` 来配置实体类和查询结果之间的映射关系，想通过这个案例更加明显的体现一下 ORM 思想。

## 6. 测试类

```
public class DeptEmpTest {

    private SqlSessionFactory sqlSessionFactory = null;

    @Before
    public void init() throws Exception {
        // 创建 SqlSessionFactoryBuilder 对象
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
        // 加载配置文件
        InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
        // 创建 SqlSessionFactory 对象
        sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
    }

    @Test
    public void testGetEmpTotalByDept() {
        //得到 sqlSession 对象
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //得到 Mapper 代理对象
        DeptEmpMapper deptEmpMapper = sqlSession.getMapper(DeptEmpMapper.class);
        //调用方法实现查询各部门员工总数
        List<DeptEmp> list = deptEmpMapper.getEmpTotalByDept();
        for (DeptEmp de : list) {
            System.out.println(de);
        }
        //关闭 sqlSession
        sqlSession.close();
    }
}
```

该测试类通过 MyBatis 的 API 实现了查询功能，如下图所示：

DeptEmpTest.testGetEmpTotalByDept x	
Tests passed: 1 of 1 test	
DeptEmpTest 1 s 765 ms	学工部 7
testGetEmp 1 s 765 ms	就业部 4
	教学部 5

## 3.2 MyBatis 的 ORM 实现原理

接下来我们通过上述案例来讲解一下 MyBatis 框架是如何应用 ORM 思想的，一起剖析一下 MyBatis 3.x 的内部实现原理。

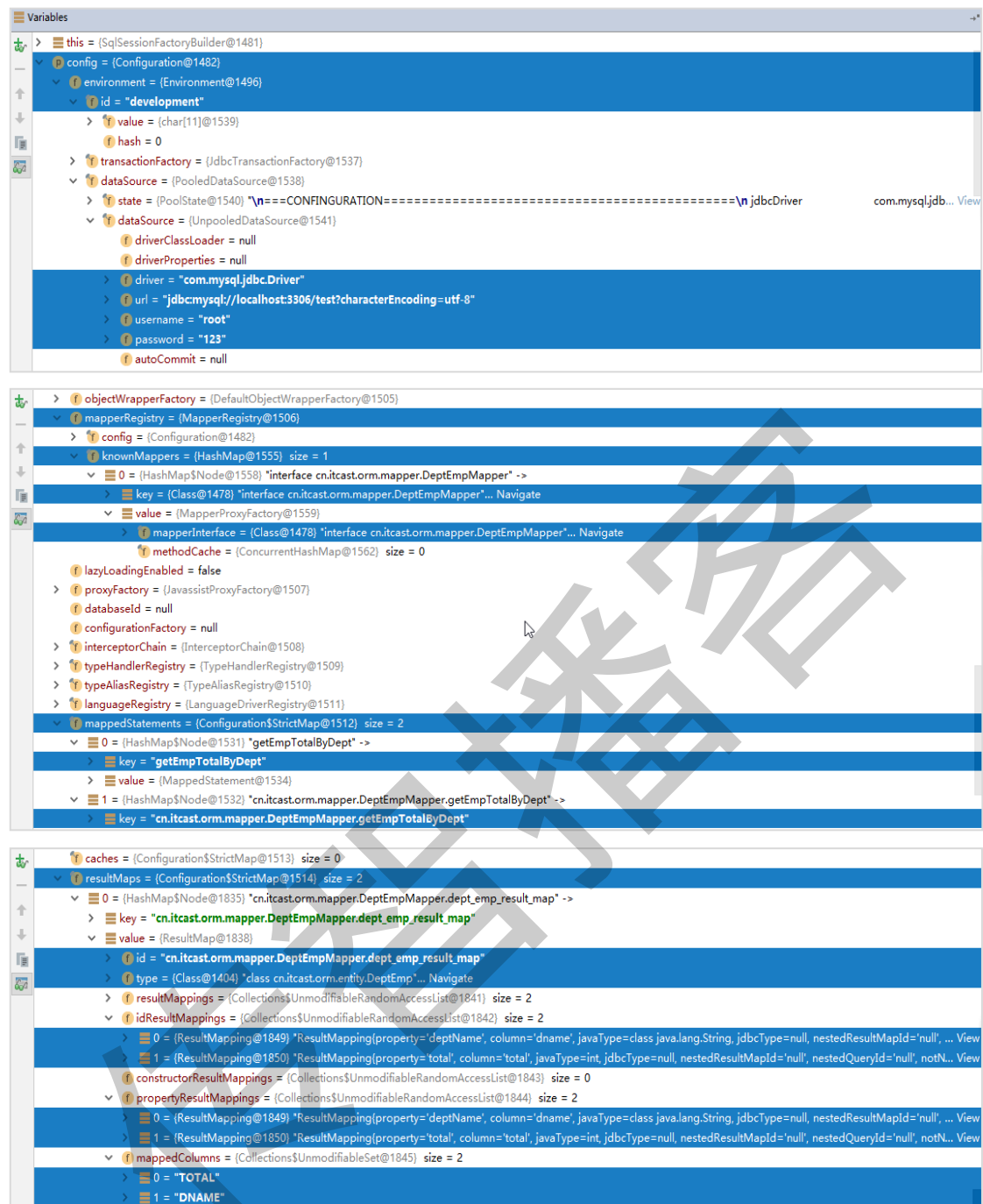
### 1. 解析 MyBatis 核心配置文件并创建 SqlSessionFactory 对象

```
InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");
SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);
```

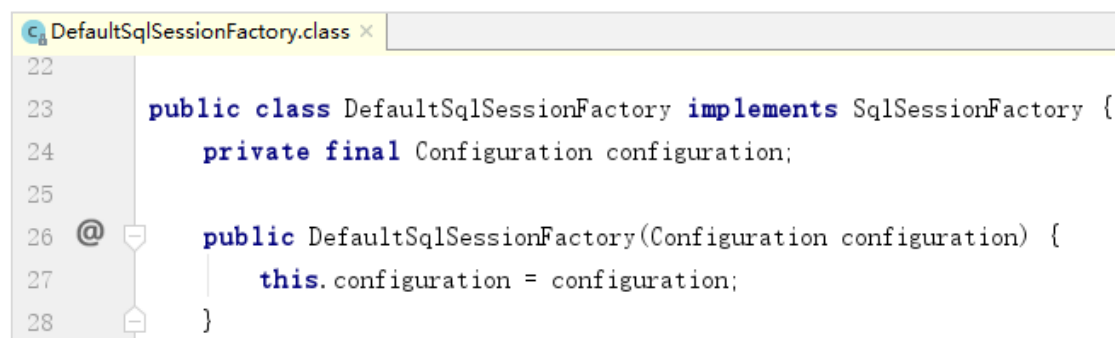
这三行代码先是创建了 `SqlSessionFactoryBuilder` 对象，然后获得了 MyBatis 核心配置文件的输入流，最后调用了 `build` 方法，我们接下来跟踪一下该方法的源码。

```
SqlSessionFactoryBuilder.class x
54 public SqlSessionFactory build(InputStream inputStream) {
55     return this.build((InputStream)inputStream, (String)null, (Properties)null);
56 }
57
58 ...
65
66 public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
67     SqlSessionFactory var5;
68     try {
69         XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
70         var5 = this.build(parser.parse());
71     } catch (Exception var14) {
72         throw ExceptionFactory.wrapException("Error building SqlSession.", var14);
73     }
74     return var5;
75 }
76
77 public SqlSessionFactory build(Configuration config) {
78     return new DefaultSqlSessionFactory(config);
79 }
```

第 69 行代码创建了一个 xml 解析器对象，并在第 70 行对 MyBatis 核心配置文件进行了解析，拿到了数据库连接数据以及映射配置文件中的数据（包括我们编写的 sql 语句和自定义的 resultMap），如下图所示：

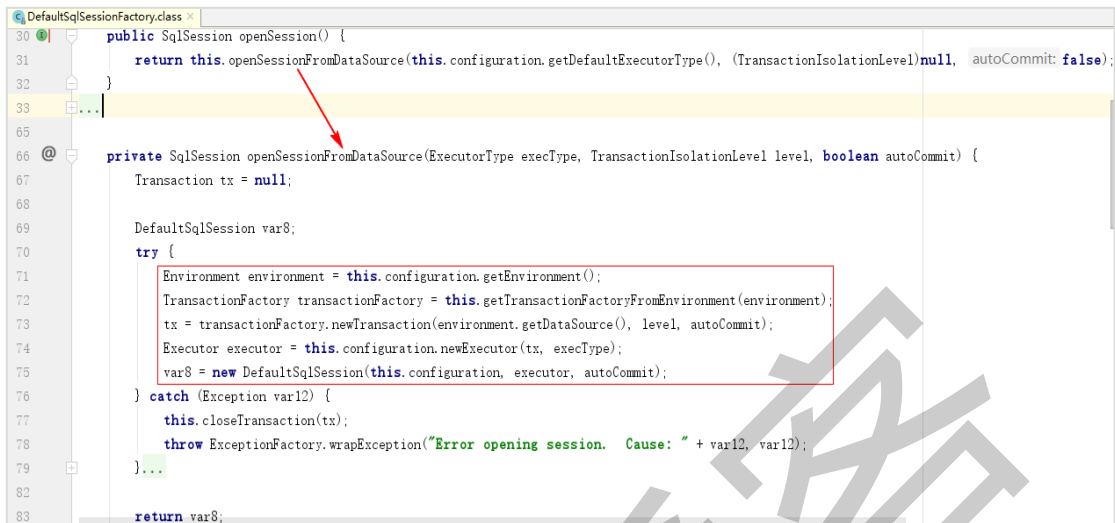


第 88 行代码创建了 DefaultSqlSessionFactory 对象，并把上图中展示的解析后拿到的数据传给了它，我们继续跟踪。



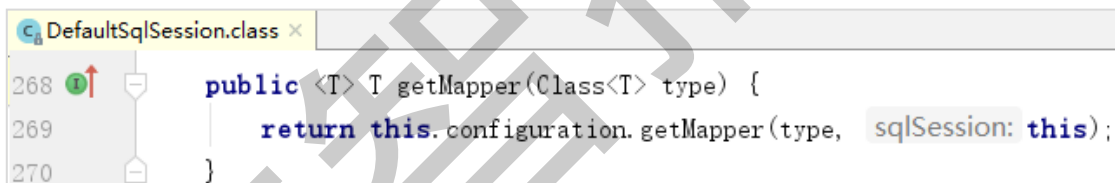
在 DefaultSqlSessionFactory 的构造方法中，我们发现，解析后拿到的数据最终被全部存入到了一个名字为 Configuration 的对象中，后续很多操作都会从该对象中获取数据。

## 2. SqlSession sqlSession = sqlSessionFactory.openSession();

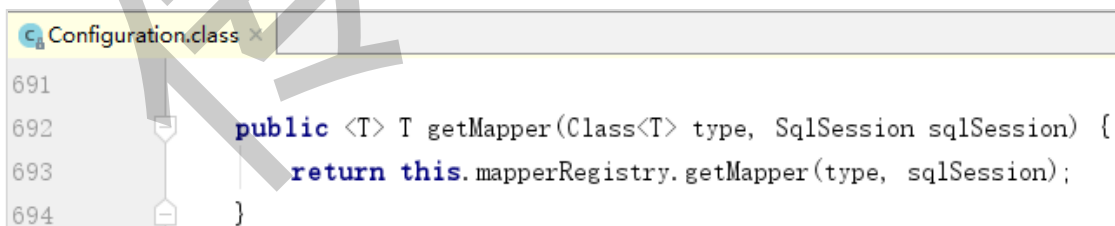


图中红框部分代码，主要通过读取 Configuration 对象中的数据分别创建了 Environment 对象，事务对象，Executor 对象等，并最终直接 new 了一个 DefaultSqlSession 对象（SqlSession 接口的实现类），该对象是 MyBatis API 的核心对象。

## 3. DeptEmpMapper deptEmpMapper = sqlSession.getMapper(DeptEmpMapper.class);



第 269 行代码调用了 Configuration 对象的 getMapper 方法，把实体类和 sqlSession 对象传了过去。



第 693 行代码调用了 MapperRegistry 对象的 getMapper 方法，把实体类和 sqlSession 对象传了过去。

```
MapperRegistry.class x
20 public class MapperRegistry {
21     private final Configuration config;
22     private final Map<Class<?>, MapperProxyFactory<?>> knownMappers = new HashMap();
23
24     @
25     public MapperRegistry(Configuration config) { this.config = config; }
26
27
28     public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
29         MapperProxyFactory<T> mapperProxyFactory = (MapperProxyFactory)<T>this.knownMappers.get(type);
30         if (mapperProxyFactory == null) {
31             throw new BindingException("Type " + type + " is not known to the MapperRegistry.");
32         } else {
33             try {
34                 return mapperProxyFactory.newInstance(sqlSession);
35             } catch (Exception var5) {
36                 throw new BindingException("Error getting mapper instance. Cause: " + var5, var5);
37             }
38         }
39     }
}
```

第 34 行代码通过代理工厂最终把我们自定义的 Mapper 接口的代理对象创建了出来。

4. List<DeptEmp> list = deptEmpMapper.getEmpTotalByDept();

```
MapperProxy.class x
27 @
28 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
29     if (Object.class.equals(method.getDeclaringClass())) {
30         try {
31             return method.invoke(obj: this, args);
32         } catch (Throwable var5) {
33             throw ExceptionUtil.unwrapThrowable(var5);
34         }
35     } else {
36         MapperMethod mapperMethod = this.cachedMapperMethod(method);
37         return mapperMethod.execute(this.sqlSession, args);
38     }
}
```

当我们调用代理对象的 getEmpTotalByDept() 方法时，框架内部会调用 MapperProxy 的 invoke 方法，我们可以观察一下这个方法三个参数的值，如下图所示：

```
Variables
> this = {MapperProxy@1514}
> proxy = {Proxy5@1498} "org.apache.ibatis.binding.MapperProxy@c8c12ac"
> method = {Method@1515} "public abstract java.util.List cn.itcast.orm.mapper.DeptEmpMapper.getEmpTotalByDept()"
> args = null
```

在 invoke 方法内锁定第 36 行代码，该行代码调用 MapperMethod 的 execute 方法实现查询功能。





```
MapperMethod.class x
43 @ public Object execute(SqlSession sqlSession, Object[] args) {
44     Object param;
45     Object result;
46     if (SqlCommandType.INSERT == this.command.getType()) {
47         param = this.method.convertArgsToSqlCommandParam(args);
48         result = this.rowCountResult(sqlSession.insert(this.command.getName(), param));
49     } else if (SqlCommandType.UPDATE == this.command.getType()) {
50         param = this.method.convertArgsToSqlCommandParam(args);
51         result = this.rowCountResult(sqlSession.update(this.command.getName(), param));
52     } else if (SqlCommandType.DELETE == this.command.getType()) {
53         param = this.method.convertArgsToSqlCommandParam(args);
54         result = this.rowCountResult(sqlSession.delete(this.command.getName(), param));
55     } else if (SqlCommandType.SELECT == this.command.getType()) {
56         if (this.method.returnsVoid() && this.method.hasResultHandler()) {
57             this.executeWithResultHandler(sqlSession, args);
58             result = null;
59         } else if (this.method.returnsMany()) {
60             result = this.executeForMany(sqlSession, args);
61         }
62     }
63     return result;
64 }
```

在 execute 方法内先进行增删改查判断，本案例进行的是查询，并且可能会查询出多条记录，所以最后锁定第 60 行代码，该行代码调用 executeForMany 方法进行查询。

```
MapperMethod.class x
121 @ private <E> Object executeForMany(SqlSession sqlSession, Object[] args) {
122     Object param = this.method.convertArgsToSqlCommandParam(args);
123     List result;
124     if (this.method.hasRowBounds()) {
125         RowBounds rowBounds = this.method.extractRowBounds(args);
126         result = sqlSession.selectList(this.command.getName(), param, rowBounds);
127     } else {
128         result = sqlSession.selectList(this.command.getName(), param);
129     }
130
131     if (!this.method.getReturnType().isAssignableFrom(result.getClass())) {
132         return this.method.getReturnType().isArray() ? this.convertToArray(result) : null;
133     } else {
134         return result;
135     }
136 }
```

第 124 行代码通过 hasRowBounds() 方法判断是否进行分页查询，本案例不需要，所以执行 else 中的代码，调用 sqlSession 的 selectList 方法（第 128 行），参数的值如下图所示：

```
Variables
param = null
this.method = {MapperMethod$MethodSignature@1529}
this.command = {MapperMethod$SqlCommand@1530}
name = "cn.itcast.orm.mapper.DeptEmpMapper.getEmpTotalByDept"
type = {SqlCommandType@1532} "SELECT"
```



```
DefaultSqlSession.class x
Decompiled .class file, bytecode version: 50.0 (Java 6)

116 public <E> List<E> selectList(String statement, Object parameter) {
117     return this.selectList(statement, parameter, RowBounds.DEFAULT);
118 }
119
120 public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
121     List var5;
122     try {
123         MappedStatement ms = this.configuration.getMappedStatement(statement);
124         var5 = this.executor.query(ms, this.wrapCollection(parameter), rowBounds, Executor.NO_RESULT_HANDLER);
125     } catch (Exception var9) {
126         throw ExceptionFactory.wrapException("Error querying database. Cause: " + var9, var9);
127     } finally {
128         ErrorContext.instance().reset();
129     }
130
131     return var5;
132 }
```

第 123 行代码从 Configuration 对象中获得 MappedStatement 对象，该对象存储了映射配置文件中的所有信息（包括我们编写的 sql 语句和自定义的 resultMap），如下图所示：

```
Variables
ms = {MappedStatement@1505}
  > resource = "cn/itcast/orm/mapper/DeptEmpMapper.xml"
  > configuration = {Configuration@1507}
  > id = "cn.itcast.orm.mapper.DeptEmpMapper.getEmpTotalByDept"
  > fetchSize = null
  > timeout = null
  > statementType = {StatementType@1525} "PREPARED"
  > resultSetType = null
  > sqlSource = {RawSqlSource@1526}
  > sqlSource = {StaticSqlSource@1535}
  > sql = "select dname, count(*) as total from dept,emp where emp.dept_id=dept.did group by dname"
  > parameterMappings = {ArrayList@1537} size = 0
  > configuration = {Configuration@1507}
  > cache = null
  > parameterMap = {ParameterMap@1527}
  > resultMaps = {Collections$UnmodifiableRandomAccessList@1528} size = 1
  > 0 = {ResultMap@1541}
  > id = "cn.itcast.orm.mapper.DeptEmpMapper.dept_emp_result_map"
  > type = {Class@1404} "class cn.itcast.orm.entity.DeptEmp"... Navigate
  > resultMappings = {Collections$UnmodifiableRandomAccessList@1543} size = 2
  > idResultMappings = {Collections$UnmodifiableRandomAccessList@1544} size = 2
  > constructorResultMappings = {Collections$UnmodifiableRandomAccessList@1545} size = 0
  > mappedColumns = {Collections$UnmodifiableSet@1547} size = 2
  > 0 = "TOTAL"
  > 1 = "DNAME"
```

第 124 行代码调用了 CachingExecutor 的 query(...)方法，继续往下跟踪。



```
CachingExecutor.class
60 public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHan
61     BoundSql boundSql = ms.getBoundSql(parameterObject);
62     CacheKey key = this.createCacheKey(ms, parameterObject, rowBounds, boundSql);
63     return this.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
64 }
65
70
71 public <E> List<E> query(MappedStatement ms, Object parameterObject, RowBounds rowBounds, ResultHandler resultHan
72     Cache cache = ms.getCache();
73     if (cache != null) {
74         this.flushCacheIfRequired(ms);
75         if (ms.isUseCache() && resultHandler == null) {
76             this.ensureNoOutParams(ms, parameterObject, boundSql);
77             List<E> list = (List<E>)this.tcm.getObject(cache, key);
78             if (list == null) {
79                 list = this.delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
80                 this.tcm.putObject(cache, key, list);
81             }
82             return list;
83         }
84     }
85 }
86
87 return this.delegate.query(ms, parameterObject, rowBounds, resultHandler, key, boundSql);
88 }
```

第 61 行通过 ms 对象从 mapper 映射配置文件中获得了 sql 语句，如下图所示：

```
Variables
boundSql = {BoundSql@1527}
sql = "select dname, count(*) as total from dept,emp where emp.dept_id=dept.did group by dname"
```

第 63 行代码调用了下面的 query 方法，该方法内部先从缓存中取数据，如果缓存中没数据，就重新查询(第 87 行)，继续往下跟踪。

```
BaseExecutor.class
118 public <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler result
119     ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId())
120     if (this.closed) {
121         throw new ExecutorException("Executor was closed.");
122     } else {
123         if (this.queryStack == 0 && ms.isFlushCacheRequired()) {
124             this.clearLocalCache();
125         }
126     }
127     List list;
128     try {
129         ++this.queryStack;
130         list = resultHandler == null ? (List<E>)this.localCache.getObject(key) : null;
131         if (list != null) {
132             this.handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
133         } else {
134             list = this.queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
135         }
136     }
```

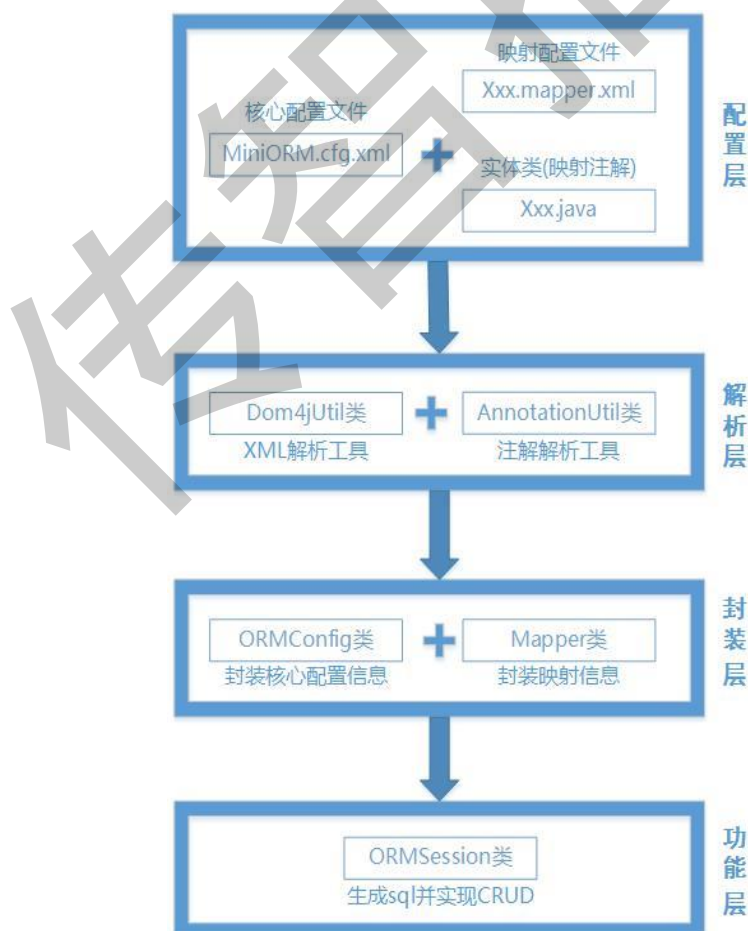
第 134 行代码通过调用 queryFromDatabase 方法最终执行了 sql 语句并将查询结果按照我们自定义的 resultMap 进行了封装，结果如下图所示：

```
Variables
└─ p boundSql = {BoundSql@1527}
   └─ f sql = "select dname, count(*) as total from dept,emp where emp.dept_id=dept.did group by dname"
      f parameterMappings = {Collections$UnmodifiableRandomAccessList@1822} size = 0
      f parameterObject = null
      f additionalParameters = {HashMap@1823} size = 0
      f metaParameters = {MetaObject@1824}
   └─ list = {ArrayList@1811} size = 3
      0 = {DeptEmp@1814} "学工部\t7"
         f deptName = "学工部"
         f total = 7
      1 = {DeptEmp@1815} "就业部\t4"
      2 = {DeptEmp@1816} "教学部\t5"
```

## 四. 自定义一个简单的 ORM 框架

前面我们通过跟踪源码的方式剖析了 Hibernate 和 MyBatis 两个框架是如何应用 ORM 思想的，接下来我们自己定义一个简单的 ORM 框架（名为 MiniORM），希望能通过这种方式让大家亲自零距离的去应用一下 ORM。

### 4.1 MiniORM 框架的结构设计



#### 1. 第一层为配置层：

- `miniORM.cfg.xml` 是框架的核心配置文件，主要用来设置数据库连接信息和映射配置文件路径信息
- `Xxx.mapper.xml` 是框架的映射配置文件，主要用来设置类和表之间以及属性和字段之间的映射关系
- `Xxx.java` 是带有映射注解的实体类，主要用来设置类和表之间以及属性和字段之间的映射关系，和 `Xxx.mapper.xml` 的作用一样，只不过采用的是注解方式，两者二选一

#### 2. 第二层为解析层：

- `Dom4jUtil` 类用来解析 `miniORM.cfg.xml` 和 `Xxx.mapper.xml` 两个配置文件的数据
- `AnnotationUtil` 类用来解析实体类中的映射注解

#### 3. 第三层为封装层：

- `ORMConfig` 类用来封装和存储从 `miniORM.cfg.xml` 文件中解析得到的数据
- `Mapper` 类用来封装和存储从 `Xxx.mapper.xml` 或实体类中解析得到的映射数据

#### 4. 第四层为功能层：

- `ORMSession` 类主要用来从 `ORMConfig` 和 `Mapper` 中获取相关数据，然后生成 `sql` 语句，最后通过对 `JDBC` 的封装最终实现增删改查功能

## 4.2 MiniORM 框架的代码实现

#### 1. pom.xml

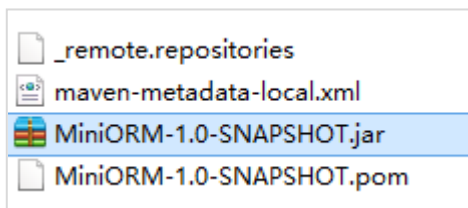
```
<groupId>cn.itcast.framework.miniorm</groupId>
<artifactId>MiniORM</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<dependencies>
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
</dependencies>

<build>
  <finalName>MiniORM</finalName>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.2</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
```

```
<source>1.8</source>
<target>1.8</target>
<encoding>UTF-8</encoding>
<compilerArguments>
    <bootclasspath>${JAVA_HOME}/jre/lib/rt.jar</bootclasspath>
</compilerArguments>
</configuration>
</plugin>
</plugins>
</pluginManagement>
</build>
```

MiniORM 框架依赖 dom4j 和 jdk1.8，编译时会打成 jar 包并 install 到本地仓库中，如下图所示：



2. miniORM.cfg.xml 是框架的核心配置文件，主要用来设置数据库连接信息和映射配置文件路径信息，源码如下所示：

```
<?xml version='1.0' encoding='utf-8'?>
<orm-factory>
    <!--数据库连接数据-->
    <property name="connection.url">jdbc:mysql://localhost:3306/test</property>
    <property name="connection.driverClass">com.mysql.jdbc.Driver</property>
    <property name="connection.username">root</property>
    <property name="connection.password">123</property>

    <!--采用 xml 配置映射数据-->
    <mapping resource="cn/itcast/orm/test/entity/Book.mapper.xml"/>
    <!--采用实体类注解配置映射数据-->
    <entity package="cn.itcast.orm.test.entity"/>
</orm-factory>
```

3. Xxx.mapper.xml 是框架的映射配置文件，主要用来设置类和表之间以及属性和字段之间的映射关系，以 Book.mapper.xml 为例，源码如下所示：

```
<?xml version='1.0' encoding='UTF-8'?>
<!--实体类和表之间的映射关系配置-->
<orm-mapping>
    <class name="cn.itcast.orm.test.entity.Book" table="t_book">
        <id name="id" column="bid"/>
        <property name="name" column="bname"/>
        <property name="author" column="author"/>
    </class>
</orm-mapping>
```

```
<property name="price" column="price"/>
</class>
</orm-mapping>
```

4. 当然 MiniORM 框架也支持在实体类上以注解方式去配置映射关系,以 Book.java 为例,源码如下所示:

```
import cn.itcast.orm.annotation.ORMColumn;
import cn.itcast.orm.annotation.ORMId;
import cn.itcast.orm.annotation.ORMTable;

//实体类:图书
@ORMTable(name = "t_book")
public class Book {

    @ORMId
    @ORMColumn(name = "bid")
    private int id; //主键

    @ORMColumn(name="bname")
    private String name; //图书名字

    @ORMColumn(name="author")
    private String author; //图书作者

    @ORMColumn(name="price")
    private double price; //图书价格

    ... ..
}
```

实体类中的@ORMTable、@ORMId、@ORMColumn 是我们自定义的三个注解,@ORMTable 用来设置当前类和哪个表对应,@ORMColumn 用来设置当前属性和表中哪个字段对应,@ORMId 用来设置哪个属性对应的字段是主键。

5. Dom4jUtil 类是一个基于 Dom4j 的工具类,主要用来解析 miniORM.cfg.xml 和 Xxx.mapper.xml,源码如下所示:

```
public class Dom4jUtil {

    /**
     * 通过文件的路径获取 xml 的 document 对象
     *
     * @param path 文件的路径
     * @return 返回文档对象
     */
    public static Document getXMLByFilePath(String path) {
        if (null == path) {
```



```
        return null;
    }
    Document document = null;
    try {
        SAXReader reader = new SAXReader();
        document = reader.read(new File(path));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return document;
}

/**
 * 获得某文档中某元素内某属性的值和元素的文本信息
 *
 * @param document    xml 文档对象
 * @param elementName 元素名
 * @param attrName    属性名
 * @return 返回一个 Map 集合
 */
public static Map<String, String> Elements2Map(Document document, String elementName,
                                                String attrName) {
    List<Element> propList = document.getRootElement().elements(elementName);
    Map<String, String> propConfig = new HashMap<>();
    for (Element element : propList) {
        String key = element.attribute(attrName).getValue();
        String value = element.getTextTrim();
        propConfig.put(key, value);
    }
    return propConfig;
}

/**
 * 针对 mapper.xml 文件，获得映射信息并存到 Map 集合中
 * @param document xml 文档对象
 * @return 返回一个 Map 集合
 */
public static Map<String, String> Elements2Map(Document document) {
    Element classElement = document.getRootElement().element("class");
    Map<String, String> mapping = new HashMap<>();

    Element idElement = classElement.element("id");
    String idKey = idElement.attribute("name").getValue();
    String idValue = idElement.attribute("column").getValue();
}
```



```
mapping.put(idKey, idValue);

List<Element> propElements = classElement.elements("property");
for (Element element : propElements) {
    String propKey = element.attribute("name").getValue();
    String propValue = element.attribute("column").getValue();
    mapping.put(propKey, propValue);
}
return mapping;
}

/**
 * 针对 mapper.xml 文件，获得主键的映射信息并存到 Map 集合中
 *
 * @param document xml 文档对象
 * @return 返回一个 Map 集合
 */
public static Map<String, String> ElementsID2Map(Document document) {
    Element classElement = document.getRootElement().element("class");
    Map<String, String> mapping = new HashMap<>();

    Element idElement = classElement.element("id");
    String idKey = idElement.attribute("name").getValue();
    String idValue = idElement.attribute("column").getValue();
    mapping.put(idKey, idValue);

    return mapping;
}

/**
 * 获得某文档中某元素内某属性的值
 *
 * @param document xml 文档对象
 * @param elementName 元素名
 * @param attrName 属性名
 * @return 返回一个 Set 集合
 */
public static Set<String> Elements2Set(Document document, String elementName, String attrName) {
    List<Element> mappingList = document.getRootElement().elements(elementName);
    Set<String> mappingSet = new HashSet<>();
    for (Element element : mappingList) {
        String value = element.attribute(attrName).getValue();
        mappingSet.add(value);
    }
}
```





```
        return mappingSet;
    }

    /**
     * 获得某文档中某元素内某属性的值
     *
     * @param document    xml 文档对象
     * @param elementName 元素名
     * @param attrName    属性名
     * @return 返回一个 Set 集合
     */
    public static String getPropValue(Document document, String elementName, String attrName) {
        Element element = (Element) document.getRootElement().elements(elementName).get(0);
        return element.attribute(attrName).getValue();
    }
}
```

6. AnnotationUtil 类主要用来通过反射技术解析实体类中的注解，从而获得映射数据，源码如下所示：

```
public class AnnotationUtil {

    /**
     得到的类名
     */
    public static String getClassName(Class clz) {
        return clz.getName();
    }

    /**
     得到 ORMTTable 注解中的表名
     */
    public static String getTableName(Class clz) {
        if (clz.isAnnotationPresent(ORMTable.class)) {
            ORMTTable ormTable = (ORMTable) clz.getAnnotation(ORMTable.class);
            return ormTable.name();
        } else {
            System.out.println("缺少 ORMTTable 注解");
            return null;
        }
    }

    /**
     得到主键属性和对应的字段
     */
}
```



```
public static Map<String, String> getIdMapper(Class clz) {
    boolean flag = true;
    Map<String, String> map = new HashMap<>();
    Field[] fields = clz.getDeclaredFields();
    for (Field field : fields) {
        if (field.isAnnotationPresent(ORMId.class)) {
            flag = false;
            String fieldName = field.getName();
            if (field.isAnnotationPresent(ORMColumn.class)) {
                ORMColumn ormColumn = field.getAnnotation(ORMColumn.class);
                String columnName = ormColumn.name();
                map.put(fieldName, columnName);
                break;
            } else {
                System.out.println("缺少 ORMColumn 注解");
            }
        }
    }
    if (flag) {
        System.out.println("缺少 ORMId 注解");
    }
    return map;
}

/*
得到类中所有属性和对应的字段
*/
public static Map<String, String> getPropMapping(Class clz) {
    Map<String, String> map = new HashMap<>();
    map.putAll(getIdMapper(clz));
    Field[] fields = clz.getDeclaredFields();
    for (Field field : fields) {
        if (field.isAnnotationPresent(ORMColumn.class)) {
            ORMColumn ormColumn = field.getAnnotation(ORMColumn.class);
            String fieldName = field.getName();
            String columnName = ormColumn.name();
            map.put(fieldName, columnName);
        }
    }
    return map;
}

/*
获得某包下面的所有类名
*/
```



```
*/  
public static Set<String> getClassNameByPackage(String packagePath) {  
    Set<String> names = new HashSet<>();  
    String packageFile = packagePath.replace(".", "/");  
    String classpath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
    if (classpath == null) {  
        classpath = Thread.currentThread().getContextClassLoader().getResource("/").getPath();  
    }  
    try {  
        classpath = java.net.URLDecoder.decode(classpath, "utf-8");  
    } catch (UnsupportedEncodingException e) {  
        e.printStackTrace();  
    }  
    File dir = new File(classpath + packageFile);  
    if (dir.exists()) {  
        File[] files = dir.listFiles();  
        for (File f : files) {  
            String name = f.getName();  
            if (f.isFile() && name.endsWith(".class")) {  
                name = packagePath + "." + name.substring(0, name.lastIndexOf("."));  
                names.add(name);  
            }  
        }  
    } else {  
        System.out.println("包路径不存在");  
    }  
    return names;  
}  
}
```

7. Mapper 类用来封装并存储从 Xxx.mapper.xml 中或从实体类中解析得到的映射信息，哪个表和哪个类映射，哪个字段和哪个属性映射等等，源码如下所示：

```
public class Mapper {  
    private String className; //类名  
    private String tableName; //表名  
    private Map<String,String> idMapper=new HashMap(); //主键字段和属性  
    private Map<String,String> propMapping=new HashMap(); //非主键字段和属性  
    ... ..  
}
```

8. ORMConfig 类主要用来存储 miniORM.cfg.xml 配置文件中的信息和 Mapper 映射信息，该类内部会使用 Dom4jUtil、AnnotationUtil 工具类去解析数据，源码如下所示：

```
public class ORMConfig {
```



```
public static String classpath; //类路径
public static File cfgFile; //核心配置文件
public static Map<String, String> propConfig; //核心配置文件数据
public static Set<String> mappingSet; //映射配置文件
public static Set<String> entitySet; //实体类
public static List<Mapper> mapperList; //解析出来的 Mapper

// 从 classpath 中加载框架的核心配置文件 miniORM.cfg.xml
static {
    classpath = Thread.currentThread().getContextClassLoader().getResource("").getPath();
    if (classpath == null) {
        classpath = Thread.currentThread().getContextClassLoader().getResource("/").getPath();
    }
    try {
        classpath = java.net.URLDecoder.decode(classpath, "utf-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    cfgFile = new File(classpath + "miniORM.cfg.xml");
    if (cfgFile.exists()) {
        Document document = Dom4jUtil.getXMLByFilePath(cfgFile.getPath());
        propConfig = Dom4jUtil.Elements2Map(document, "property", "name");
        mappingSet = Dom4jUtil.Elements2Set(document, "mapping", "resource");
        entitySet = Dom4jUtil.Elements2Set(document, "entity", "package");
    } else {
        cfgFile = null;
        System.out.println("未找到核心配置文件 miniORM.cfg.xml");
    }
}

//从 propConfig 获得信息，连接数据库
private Connection getConnection() throws ClassNotFoundException, SQLException {
    String url = propConfig.get("connection.url");
    String driverClass = propConfig.get("connection.driverClass");
    String username = propConfig.get("connection.username");
    String password = propConfig.get("connection.password");
    Class.forName(driverClass);
    Connection connection = DriverManager.getConnection(url, username, password);
    connection.setAutoCommit(true);
    return connection;
}

//从 mappingSet 中挨个解析 mapper.xml 配置文件，获得实体类和表之间的映射信息
//从 entitySet 中挨个解析实体类中的注解，获得实体类和表之间的映射信息
```



```
private void getMapping() throws ClassNotFoundException {
    mapperList = new ArrayList<>();
    for (String xmlPath : mappingSet) {
        Document document = Dom4jUtil.getXMLByFilePath(classpath + xmlPath);

        Map<String, String> mapping = Dom4jUtil.Elements2Map(document);

        String className = Dom4jUtil.getPropValue(document, "class", "name");
        String tableName = Dom4jUtil.getPropValue(document, "class", "table");
        Map<String, String> id_id = Dom4jUtil.ElementsID2Map(document);

        Mapper mapper = new Mapper();
        mapper.setClassName(className);
        mapper.setTableName(tableName);
        mapper.setIdMapper(id_id);
        mapper.setPropMapping(mapping);

        mapperList.add(mapper);
    }

    for (String packagePath : entitySet) {
        Set<String> nameSet = AnnotationUtil.getClassNameByPackage(packagePath);
        for (String name : nameSet) {
            Class clz = Class.forName(name);
            String className = AnnotationUtil.getClassName(clz);
            String tableName = AnnotationUtil.getTableName(clz);
            Map<String, String> id_id = AnnotationUtil.getIdMapper(clz);
            Map<String, String> mapping = AnnotationUtil.getPropMapping(clz);
            Mapper mapper = new Mapper();
            mapper.setClassName(className);
            mapper.setTableName(tableName);
            mapper.setIdMapper(id_id);
            mapper.setPropMapping(mapping);

            mapperList.add(mapper);
        }
    }
}

public ORMSession buildORMSession() throws Exception {
    //从 propConfig 获得信息，连接数据库
    Connection connection = getConnection();

    //从 mappingSet 中挨个解析 mapper.xml 配置文件，获得实体类和表之间的映射信息
}
```

```
getMapping();

//创建 ORMSession 对象
return new ORMSession(connection);
}
}
```

9. ORMSession 类主要用来从 ORMConfig 和 Mapper 中获取相关数据，然后生成 sql 语句，最后通过对 JDBC 的封装最终实现增删改查功能，源码如下所示：

```
public class ORMSession {

    private Connection connection;

    public ORMSession(Connection conn) {
        this.connection = conn;
    }

    //保存数据
    public void save(Object entity) throws Exception {
        String insertSQL = "";

        //1. 从 ORMConfig 中获得保存有映射信息的集合
        List<Mapper> mapperList = ORMConfig.mapperList;

        //2. 遍历集合，从集合中找到和 entity 参数相对应的 mapper 对象
        for (Mapper mapper : mapperList) {
            if (mapper.getClassName().equals(entity.getClass().getName())) {
                String tableName = mapper.getTableName();
                String insertSQL1 = "insert into " + tableName + "(" + " ";
                String insertSQL2 = " ) values ( ";

                //3. 得到当前对象所属类中的所有属性
                Field[] fields = entity.getClass().getDeclaredFields();
                for (Field field : fields) {
                    field.setAccessible(true);
                    //4. 遍历过程中根据属性得到字段名
                    String columnName = mapper.getPropMapper().get(field.getName());
                    //5. 遍历过程中根据属性得到它的值
                    String columnValue = field.get(entity).toString();
                    //6. 拼接 sql 语句
                    insertSQL1 += columnName + ",";
                    insertSQL2 += "'" + columnValue + "',";
                }
                insertSQL = insertSQL1.substring(0, insertSQL1.length() - 1) + insertSQL2.substring(0,
```



```
insertSQL2.length() - 1) + " )";
        break;
    }
}

// 把 sql 语句打印到控制台
System.out.println("MiniORM-save: " + insertSQL);

//7. 通过 JDBC 发送并执行 sql
PreparedStatement statement = connection.prepareStatement(insertSQL);
statement.executeUpdate();
statement.close();
}

//根据主键进行数据删除 delete from 表名 where 主键 = 值
public void delete(Object entity) throws Exception {
    String delSQL = "delete from ";

    //1. 从 ORMConfig 中获得保存有映射信息的集合
    List<Mapper> mapperList = ORMConfig.mapperList;

    //2. 遍历集合，从集合中找到和 entity 参数相对应的 mapper 对象
    for (Mapper mapper : mapperList) {
        if (mapper.getClassName().equals(entity.getClass().getName())) {
            // 3. 得到我们想要的 mapper 对象，并得到表名
            String tableName = mapper.getTableName();
            delSQL += tableName + " where ";

            // 4. 得到主键的字段名和属性名
            Object[] idProp = mapper.getIdMapper().keySet().toArray(); //idProp[0]
            Object[] idColumn = mapper.getIdMapper().values().toArray(); //idColumn[0]

            // 5. 得到主键的值
            Field field = entity.getClass().getDeclaredField(idProp[0].toString());
            field.setAccessible(true);
            String idVal = field.get(entity).toString();

            // 6. 拼接 sql
            delSQL += idColumn[0].toString() + " = " + idVal;
            // 把 sql 语句打印到控制台
            System.out.println("MiniORM-delete: " + delSQL);

            break;
        }
    }
}
```



```
//7. 通过 JDBC 发送并执行 sql
PreparedStatement statement = connection.prepareStatement(delSQL);
statement.executeUpdate();
statement.close();
}

// 根据主键进行查询 select * from 表名 where 主键字段 = 值
public Object findOne(Class clz, Object id) throws Exception{

    String querySQL = "select * from ";

    //1. 从 ORMConfig 中得到存有映射信息的集合
    List<Mapper> mapperList=ORMConfig.mapperList;

    //2. 遍历集合拿到我们想要的 mapper 对象
    for (Mapper mapper : mapperList) {
        if (mapper.getClassName().equals(clz.getName())) {

            // 3. 获得表名
            String tableName = mapper.getTableName();

            //4. 获得主键字段名
            Object[] idColumn = mapper.getIdMapper().values().toArray(); //idColumn[0]

            //5. 拼接 sql
            querySQL += tableName + " where " + idColumn[0].toString() + " = " + id;

            break;
        }
    }
    System.out.println("MiniORM-findOne:" +querySQL);

    //6. 通过 jdbc 发送并执行 sql, 得到结果集
    PreparedStatement statement=connection.prepareStatement(querySQL);
    ResultSet rs=statement.executeQuery();

    //7. 封装结果集, 返回对象
    if(rs.next()){
        // 查询到一行数据
        // 8.创建一个对象, 目前属性的值都是初始值
        Object obj=clz.newInstance();
        // 9. 遍历 mapperList 集合找到我们想要的 mapper 对象
        for(Mapper mapper:mapperList){
            if (mapper.getClassName().equals(clz.getName())) {
```



```
//10. 得到存有属性-字段的映射信息
Map<String,String> propMap = mapper.getPropMapper();
//11. 遍历集合分别拿到属性名和字段名
Set<String> keySet = propMap.keySet();
for(String prop:keySet){ //prop 就是属性名
    String column = propMap.get(prop); //column 就是和属性对应的字段名
    Field field = clz.getDeclaredField(prop);
    field.setAccessible(true);
    field.set(obj,rs.getObject(column));
}
break;
}
}
//12. 释放资源
statement.close();
rs.close();

//13. 返回查询出来的对象
return obj;
}else {
    // 没有查到数据
    return null;
}
}

//关闭连接，释放资源
public void close() throws Exception{
    if(connection!=null){
        connection.close();
        connection = null;
    }
}
}
```

## 4.3 MiniORM 框架的测试使用

我们自定义的 MiniORM 框架主要用来体现 ORM 思想，并不是为了开发一个成熟的持久层框架出来，因此很多逻辑并不完善，很多情况也未去考虑，请各位理解。接下来我们就测试一下该框架。

### 1. pom.xml

```
<groupId>cn.itcast.orm</groupId>
<artifactId>TestMiniORM</artifactId>
```



```
<version>1.0-SNAPSHOT</version>

<dependencies>
  <dependency>
    <groupId>cn.itcast.framework.miniorm</groupId>
    <artifactId>MiniORM</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.36</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
  </dependency>
</dependencies>
<build>
  <resources>
    <resource>
      <directory>src/main/java</directory>
      <includes> <!--为了编译时能加载包中的 xml 文件-->
        <include>/**/*.xml</include>
      </includes>
      <filtering>false</filtering>
    </resource>
    <resource> <!--为了编译时能加载 resources 中的 xml 文件-->
      <directory>src/main/resources</directory>
      <includes>
        <include>/**/*.xml</include>
      </includes>
      <filtering>false</filtering>
    </resource>
  </resources>
</build>
```

我们前面把 MiniORM 框架打成 jar 包并 install 到了本地 Maven 仓库中，因此在使用该框架时需要从本地仓库进行加载。

## 2. miniORM.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<orm-factory>
  <!--数据库连接-->
```

```
<property name="connection.url">jdbc:mysql://localhost:3306/test</property>
<property name="connection.driverClass">com.mysql.jdbc.Driver</property>
<property name="connection.username">root</property>
<property name="connection.password">123</property>

<!--采用 xml 配置映射数据-->
<mapping resource="cn/itcast/orm/test/entity/Book.mapper.xml"/>

<!--采用注解配置映射数据-->
<entity package="cn.itcast.orm.test.entity"/>
</orm-factory>
```

这是框架的核心配置文件，我们既采用了 xml 方式配置映射数据，也采用了注解方式在实体类中配置映射数据，最后可以二选一分别进行功能测试。

### 3. 实体类和映射配置文件

```
@ORMTable(name = "t_book")
public class Book {

    @ORMId
    @ORMColumn(name = "bid")
    private int id; //主键

    @ORMColumn(name="bname")
    private String name; //图书名字

    @ORMColumn(name="author")
    private String author; //图书作者

    @ORMColumn(name="price")
    private double price; //图书价格
    ... ..
}
```

```
<?xml version='1.0' encoding='UTF-8'?>
<!--实体类和表之间的映射关系配置-->
<orm-mapping>
    <class name="cn.itcast.orm.test.entity.Book" table="t_book">
        <id name="id" column="bid"/>
        <property name="name" column="bname"/>
        <property name="author" column="author"/>
        <property name="price" column="price"/>
    </class>
</orm-mapping>
```

注意：对于同一个表或实体类，不需要既进行 xml 配置，又进行注解配置，二选一即可，这



里同时进行配置只是为了测试方便。

#### 4. 测试类

```
public class BookDao {  
  
    private ORMConfig config;  
  
    @Before  
    public void init() {  
        config = new ORMConfig();  
    }  
  
    @Test  
    public void testSave() throws Exception {  
        ORMSession session = config.buildORMSession();  
        Book book = new Book();  
        book.setId(1);  
        book.setName("降龙十八掌");  
        book.setAuthor("不知道");  
        book.setPrice(9.9);  
        session.save(book);  
        session.close();  
    }  
  
    @Test  
    public void testDelete() throws Exception {  
        ORMSession session = config.buildORMSession();  
        Book book = new Book();  
        book.setId(1);  
        session.delete(book);  
        session.close();  
    }  
  
    @Test  
    public void testFindOne() throws Exception {  
        ORMSession session = config.buildORMSession();  
        Book book = (Book) session.findOne(Book.class, 1);  
        System.out.println(book);  
        session.close();  
    }  
}
```

- 我们调用 ORMSession 类的 save 方法完成了数据保存功能，由 MiniORM 框架生成 sql 语句，运行效果如下图所示：

```
BookDao.testSave x
>> Tests passed: 1 of 1 test - 671 ms
BookDao (cn.itca 671 ms) MiniORM-save: insert into t_book(bid,bname,author,price ) values ( '1','降龙十八掌','不知道','9.9' )
testSave 671 ms
```

- 我们调用 ORMSession 类的 findOne 方法完成了数据查询功能，由 MiniORM 框架生成 sql 语句，运行效果如下图所示：

```
BookDao.testFindOne x
>> Tests passed: 1 of 1 test - 806 ms
BookDao (cn.itca 806 ms) MiniORM-findOne: select * from t_book where bid = 1
testFindOne 806 ms Book {id=1, name='降龙十八掌', author='不知道', price=9.9}
```

- 我们调用 ORMSession 类的 delete 方法完成了数据删除功能，由 MiniORM 框架生成 sql 语句，运行效果如下图所示：

```
BookDao.testDelete x
>> Tests passed: 1 of 1 test - 797 ms
BookDao (cn.itca 797 ms) MiniORM-delete: delete from t_book where bid = 1
testDelete 797 ms
```