

# MyBatis



## -----持久层框架详解

青岛TTC.王兴刚  
V2017



# 数据库与数据持久化

- 什么是持久化
- 数据库在持久化中的地位
- 持久层
- 理论持久层的缺陷
- ORM思想的产生
- 持久层的实现方案
  - 半自动全自动
- 事实上的工业标准
- Hibernate的药方缺陷
- 持久层方案的理性回归
- MyBatis的发展之路

# 持久化(Persistence)

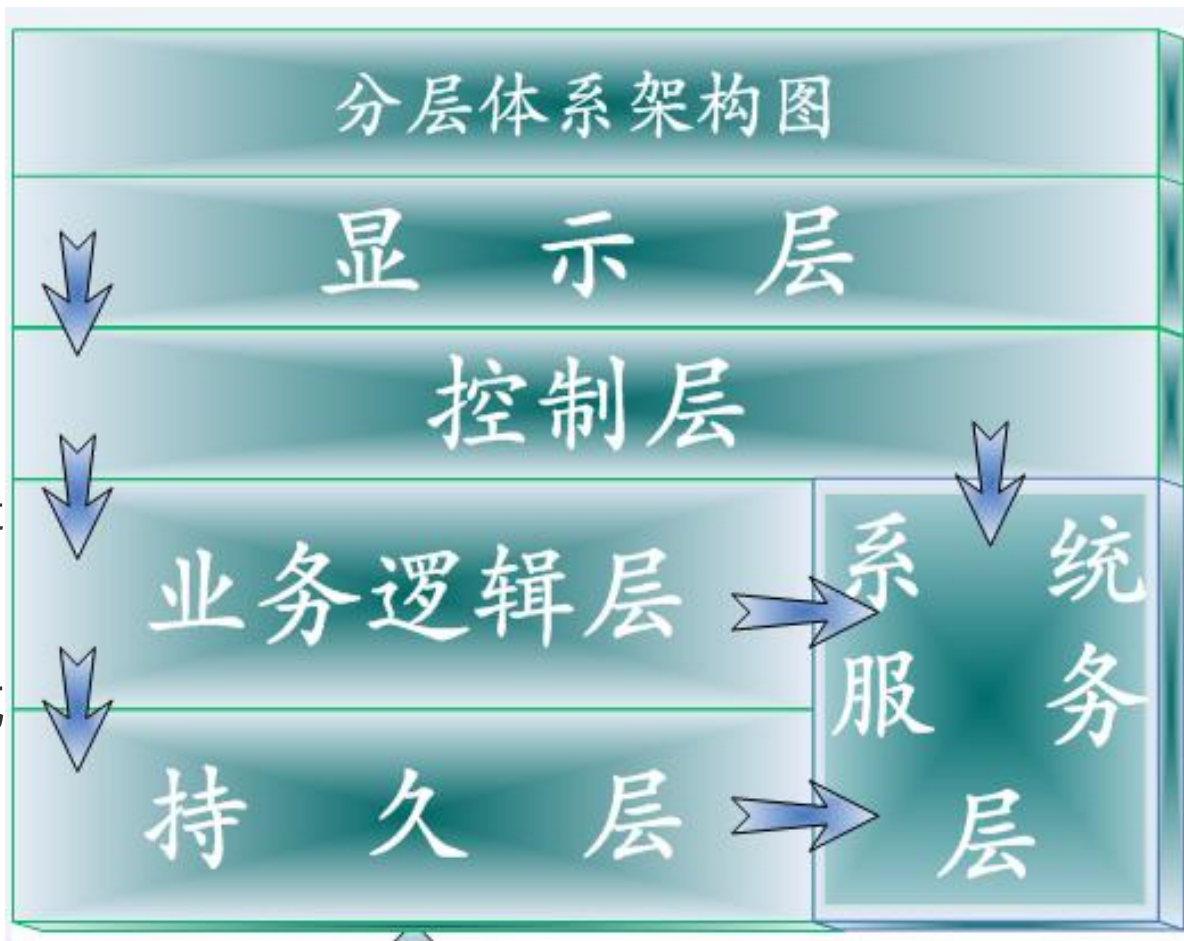
- 所谓持久化就是指将数据保存到可掉电设备中.
- 通俗而言,就是将程序执行过程中,在内存中产生的数据,保存到可以永久存储的文件中,这些文件可以是数据库,Excel,XML文件以及IO流文件等

# 数据库在持久化中的地位

- 持久化的核心是数据库
  - 信息,在人类社会的发展过程中,不断呈现爆炸增长的势态.为了面对这一现实,信息数据化是人们目前所能想到的最好手段.随着信息量的增长,数据处理的手段也在不断演进,从原来的文件时代进入到数据库时代.可以说数据库是目前信息系统建设过程中,最重要的数据(信息)载体.一切运算的终极目标都是对数据的检索和更新.
  - 基于上述,持久化的最终目的实际上就是实现对数据库检索和更新.因此,可以认为数据持久化的核心就在于以什么样的方式,操作数据库.

# 分层体系结构与持久层

- 分层体系架构,是大规模系统建设过程中普遍采用的系统架构模式,如图:



## 持久层:

实现数据持久化的系统逻辑层,就是持久层

是分层系统结构的重要组成部分

专注于数据的处理过程



# 数据库对持久层的影响(1)

- 数据库的差异性
  - SQL语言是数据库的灵魂,虽然所有的数据库都遵从了共同的SQL标准(ISO SQL标准),但是在SQL标准的本地化实现上还是存在诸多差异
- 差异化的具体表现
  - 数据类型的表述名称
  - 内置函数
  - SQL查询语法
  - 本地化特性实现上
    - rownum limit 等等

# 数据库对持久层的影响 (2)

- 持久层设计的目标.

- 持久层设计的目标就是以高效的手段进行数据处理,同时其处理手段可以在不同数据库系统之间进行无粘性的平滑移植
- 但是,由于不同数据库系统之间基于SQL层面差异性的存在,平滑移植这一目标并没有实现,无论是Hibernate,MyBatis还是EJB,都是一样,在可以预见的未来,平滑移植,可能只是个梦想

# JavaEE的正统持久层规范

- JavaEE的13项核心技术规范中,对于持久层的实现,针对不同的工作环境,给出了两种实现手段
  - EJB(Enterprise JavaBean,企业级JavaBean)
    - 分布式系统
  - JDBC(Java Data Base Connectivity,java数据库连接)
    - 非分布式系统
- 由于这些技术是由Java官方提供的,因此业界称其为“ 正统持久层规范” 或“ 正统持久化理论”



# 正统持久化理论的缺陷分析

- EJB缺陷分析(上)
  - 不可否认的一个事实是,EJB在分布式运算领域起着开先河的作用,在基于分布式运算的复杂系统建设领域有着不可或缺的地位.
  - 基于其应用领域上的定位,EJB在实现上,几乎是极尽繁琐之能事,那是相当麻烦,同时技术实现上存在着不可以回避的缺陷,这些缺陷从EJB1.X开始直到目前EJB3.X,并未完全解决,具体表现请看后继PPT

# 正统持久化理论的缺陷分析

- EJB缺陷分析(下)
  - 不能处理持久化对象间的关系
    - E-R关系描述不清楚
    - 在EJB3中基本解决
  - 使用了糟糕的查询语言
    - EJB3改善明显
  - 开发过程,测试过程及维护过程三关难越
    - EJB3改善显著
  - 重量级资源消耗
    - EJB3改善很大,但基于其应用领域上的定位,很难实现轻量级的转变

# 正统持久化理论的缺陷分析

- JDBC不符合面向对象思维
  - 依据面向对象的设计原则，组件之间，调用或传递的最小单元应该是对象而不是属性，也就是说，我们直接操作的最终元素应该是对象，并且通过对象的方法影响其属性。映射到客观世界，就是通过行为影响状态的改变。
  - 但是应用JDBC进行数据库操作时，必须将对象进行分解，直接操作每个实体（记录）的某个属性（字段）。从此，程序员的视图中，一旦操作数据库，就变成了对对象的分割，这显然是与面向对象左道而驰的。

# ORM思想的产生

- 由于EJB本身存在技术上的缺陷,而JDBC使用了一种错误的思维模型处理对象和关系(表)之间的转换.因此,在持久化技术的实践过程中,诸多大能者提出新的编程方向:
  - 用POJO映射实体(表)
  - 使用元数据表述描述对象与数据库间的映射。
- 于是，一种新的编程模型由此而生，这就是ORM  
(Object Relation Mapping, 简称ORM, 或O/RM, 或O/R mapping) 思想的产生根源

# ORM的改进之道

- 基本原理

- 现实世界的各种实体，总是可以用一个合适的对象对其进行表述。通过对象及对象之间的关系,可以模拟客观世界的复杂构型。这是面向对象理论的思想根源。而数据库则通过关系(表)演绎了客观世界的实体及其相互联系。为此，在程序设计中需要一种手段,实现内存对象到数据表的转换。这就是所谓的ORM--对象关系映射.

- 改进之道

- 依据ORM思想，将数据库中的每个表（R）映射成为一个类（O），程序只需要对映射类（O）进行检索和更新,就可以由映射机制（M）,自动转换成对数据库相应表（R）的检索和更新，而数据库中的E-R关系则通过映射类之间的对应关系进行描述。这就是ORM的改进之道

# ORM持久层实现方案分类

- ORM持久层框架，依据Mapping机制的实现过程分为两大类
  - 全自动
  - 半自动

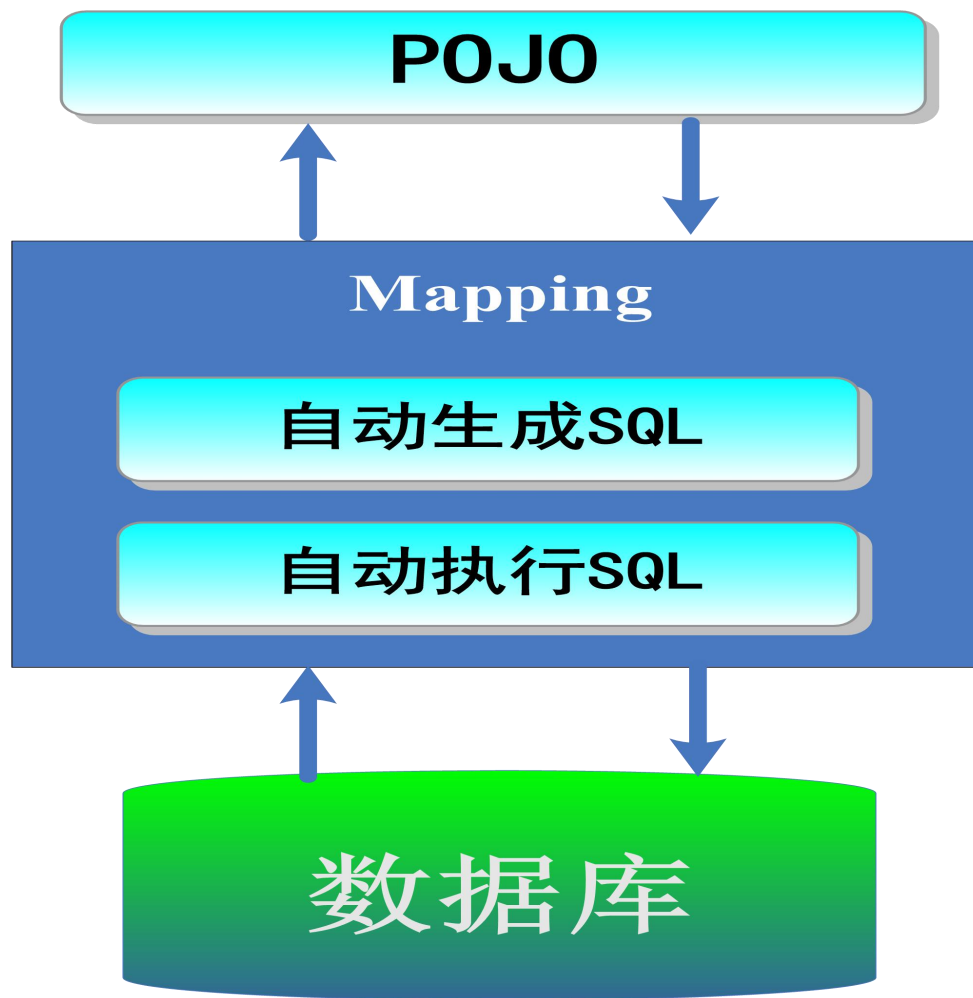


# 全自动ORM框架



目前主流的ORM持久化架构(比如Hibernate),都对数据库的表结构提供了较为完整的封装,在O与R之间,提供全套的Mapping机制.在实际的项目中,只需要定义好O与R之间的映射关系,就可以通过对O的检索和更新完成数据持久化处理.程序员无需了解SQL和JDBC规范的语法及工作原理.因为ORM框架,会根据预定的映射方案和数据存储逻辑,自动生成对应的SQL并调用JDBC接口加以执行,这就是所说的一站式服务,也可以理解为"全自动".

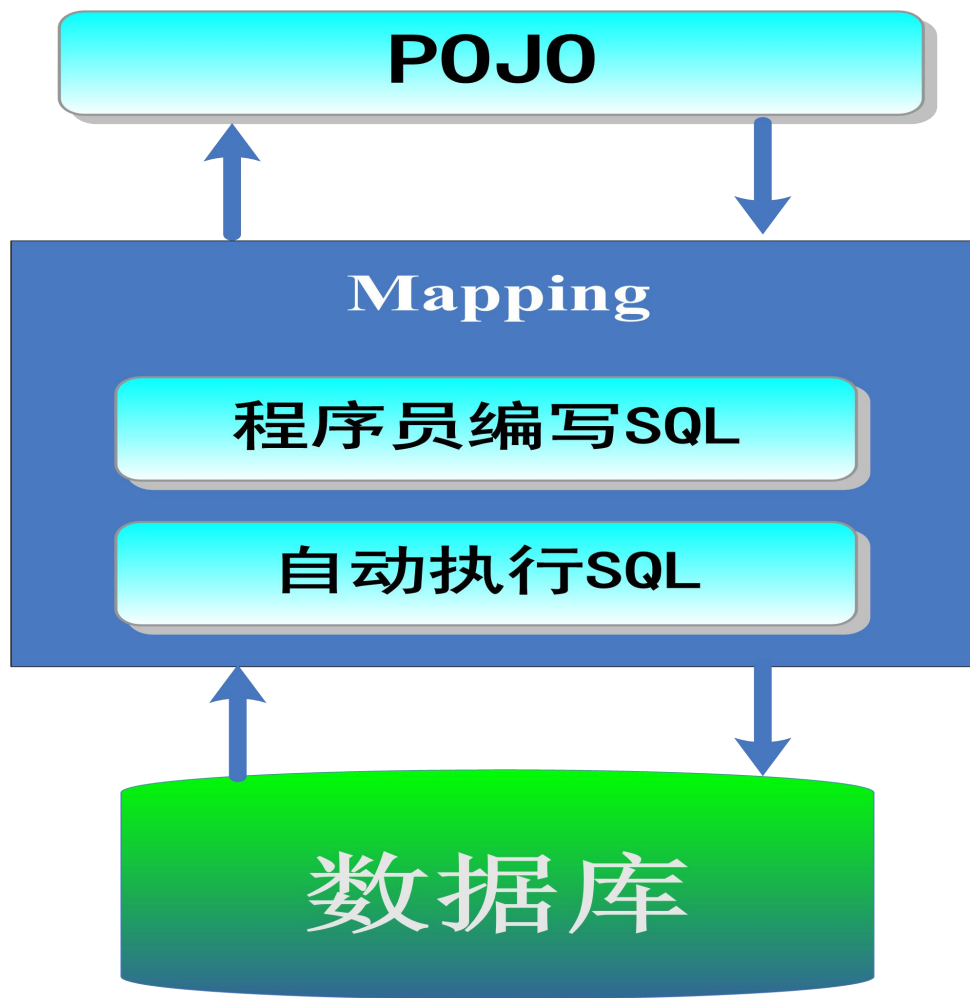
# 全自动ORM框架示意图



# 半自动ORM框架

- 所谓"半自动化",是相对于Hibernate等ORM框架的,一站式全面数据库封装机制而言的.具体实现上,就是MyBatis并不会自动生成程序运行期间需要执行的SQL语句,这些SQL语句需要由程序员自己编写,然后通过映射文件,将SQL所需的参数,以及返回的结果映射到指定POJO。

# 半自动ORM框架示意图



# 全自动的缺陷分析(1)

- 性能缺陷

- 在大规模项目中, 持久层需要处理海量数据, 对系统性能的要求也往往极为苛刻. 这样, 我们必须对系统中执行的SQL语句进行优化. 否则无法达到预定的性能指标. 由于在Hibernate框架下, 无法人为优化执行过程中生成的SQL语句, 因此在系统性能方面的表现, 是不尽如人意的.

# 全自动的缺陷分析(2)

- SQL格式的不完全支持
  - 基于上述性能考虑, 在优化SQL时候, 往往需要改变一些SQL语句的用法, 特别是对于查询语句而言, 优化的空间非常大. 往往同一个查询结果, 换了另外一种SQL句型, 性能表现上, 可能完全不同. 但是Hibernate封装后的SQL语句, 对许多特定SQL格式, 支持乏力.



# 全自动的缺陷分析 (3)

- 不利于数据安全
  - 在Hibernate下,必须将所有的表都映射成POJO,然后通过对POJO的操作反向影响底层的表.但是,出于安全考虑,在一些特殊行业中,往往一些关键表的结构,对开发人员是完全保密的.客户方能提供给开发人员的只有几条SQL而已,此时,一站式架构无法工作.

# 全自动的缺陷分析(4)

- 不适应特殊开发规范
  - 在一些特殊的开发规范中，要求所有牵涉到业务逻辑部分的数据库操作，必须在数据库层由存储过程实现，比如金融行业,工商银行、中国银行、交通银行等，都在开发规范中做出了严格指定.而Hibernate在上述工作模式下表现欠缺.

# 数据库与数据持久化



- Hibernate的药方缺陷
- 持久层方案的理性回归
- MyBatis的发展之路

# Hibernate反思

- 软件,是一个系统化的工程,整个架构设计是核心和重中之重.架构的性能最终决定了系统的整体性能.基于开源框架的二次开发,是目前架构设计中普遍采用的手段
- 所谓架构设计就是在性能和通用性之间寻求最佳配比.单纯为了通用性而牺牲性能,我认为,这是架构设计的大忌.
- Hibernate为了达到数据库移植上的通用性,在性能方面做出了很大的牺牲,这是否值得,我想,这是最值得我们反思的问题.此外,数据库间的平滑移植,是否真的存在很大的必要性?这是我们需要反思的另外一个问题!如果忽视了数据库中的若干特性语法,若干优化实现,我们用这个数据库是否有意义?

# MyBatis的理性实现

- 使用MyBatis 提供的ORM机制，对业务逻辑实现人员而言，面对的是纯粹的 Java对象，这一层与通过 Hibernate 实现 ORM 而言基本一致，而对于具体的数据操作，Hibernate会自动生成SQL 语句，而MyBatis 则要求开发者编写具体的 SQL 语句,以便优化性能。相对Hibernate 等“全自动” ORM机制而言，MyBatis 以 SQL开发的工作量和数据库移植性上的让步，为系统设计提供了更大的自由空间,同时避免了全自动的诸多缺陷,可谓是一种退而求其次的理想方案.
- **抱残守缺的哲学意境是为中而不庸**

# MyBatis简介

2004年IBATIS推出  
.NET版本,实现跨平  
台目标

2

2010年IBATIS由Apache Software 迁移到  
了google code并更名为**MyBatis**

1

IBATIS一词来源于“internet”和“abatis”的组合,是由Clinton Begin在2001年发起的开源项目,最初侧重于密码软件的开发,后来发展为Java持久层框架。当时隶属于Apache



3



# 数据模型

- 创建如下数据模型, 并导入数据库

员工信息表 (AC01)		
员工流水号	int	<pk>
姓名	varchar(4)	
身份证	varchar(20)	
性别	varchar(1)	
出生日期	date	
专业	varchar(50)	
电话	varchar(16)	
职务	varchar(20)	
备注	text	
建档日期	datetime	

FK\_Reference\_1

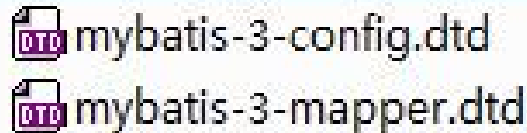
职务变更信息表 (AC02)		
变更流水号	int	<pk>
员工流水号	int	<fk>
变更日期	date	
变更原因	varchar(500)	
变更前职务	varchar(50)	
变更后职务	varchar(50)	

Name	Code	Data Type	Length	Pre	F	F	N
员工流水号	AAC101	int			<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
姓名	AAC102	varchar(4)	4		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
身份证	AAC103	varchar(20)	20		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
性别	AAC104	varchar(1)	1		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
出生日期	AAC105	date			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
专业	AAC106	varchar(50)	50		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
电话	AAC107	varchar(16)	16		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
职务	AAC108	varchar(20)	20		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
备注	AAC109	text			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
建档日期	AAC110	datetime			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Name	Code	Data Type	Length	Precisi	F	F	N
变更流水号	AAC201	int			<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
员工流水号	AAC101	int			<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
变更日期	AAC202	date			<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
变更原因	AAC203	varchar(500)	500		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
变更前职务	AAC204	varchar(50)	50		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
变更后职务	AAC205	varchar(50)	50		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

# 开发环境搭建

- Eclipse开发环境中,默认情况下没有对MyBatis开发提供支持,这给开发工作带来很多不便,比如映射及配置问的校验,提示信息的快捷支持等.因此需要对Eclipse进行扩展,以方便MyBatis的开发.
- 导入如下两个文件,对应的Public ID如下:



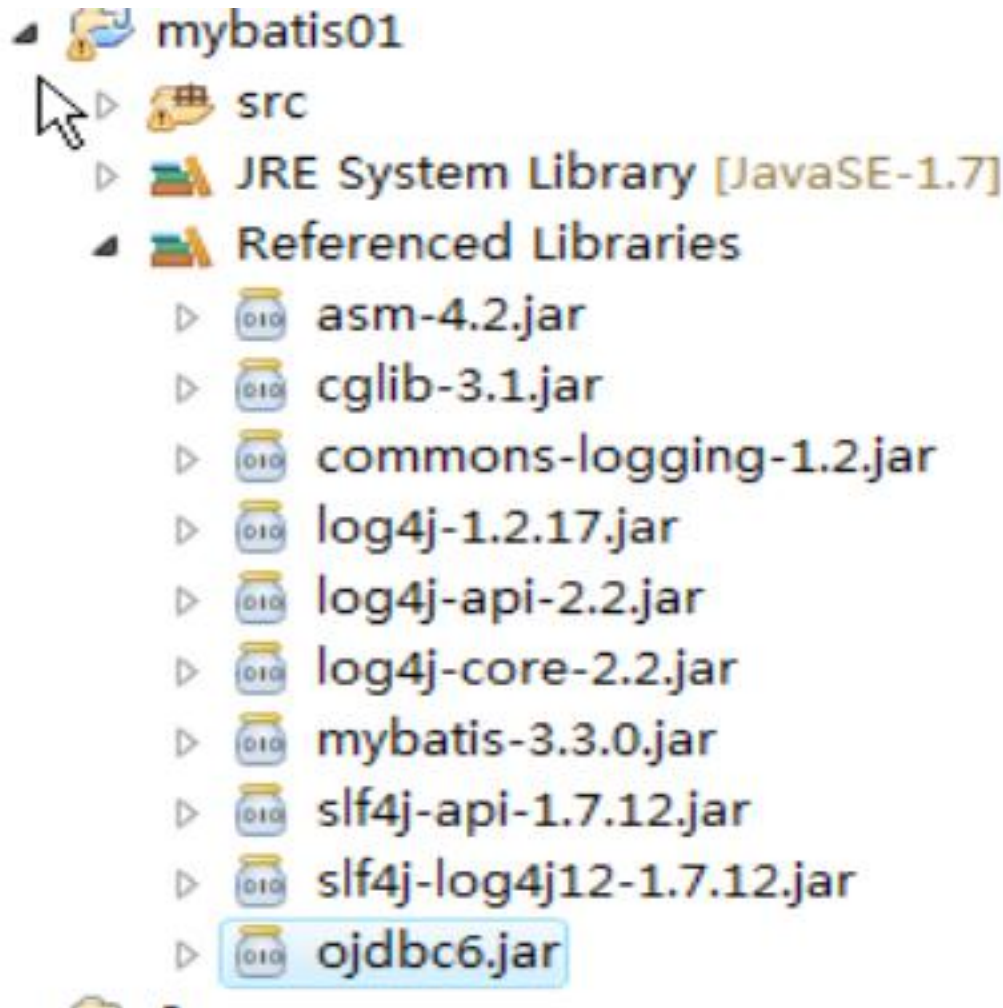
mybatis-3-config.dtd  
mybatis-3-mapper.dtd

- -//mybatis.org//DTD Mapper 3.0//EN
- <http://mybatis.org/dtd/mybatis-3-mapper.dtd>
- -//mybatis.org//DTD Config 3.0//EN
- <http://mybatis.org/dtd/mybatis-3-config.dtd>

# 创建Mybatis工程结构如下



# 为工程添加MyBatis相关jar包





# 代码开发(1)

- 编写员工信息表(AC01)的映射类, 代码如下:

```
1 package com.neusoft.mybatis.bean;
2
3 import java.io.Serializable;
4
5 public class Ac01 implements Serializable
6 {
7     private Integer aac101=null;    //流水号
8     private String aac102=null;    //姓名
9     private String aac103=null;    //身份证
10    private String aac104=null;    //性别
11    private String aac105=null;    //生日
12    private String aac106=null;    //专业
13    private String aac107=null;    //电话
14    private String aac108=null;    //职务
15    private String aac109=null;    //备注
16    //设置子和访问子及toString() 略...
17 }
```

# 代码开发(2)

- 编写映射文件, 完成数据添加

```
Ac01Mapping.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3     "mybatis-3-mapper.dtd" >
4 <mapper namespace="Ac01Mapping">
5
6     <insert id="addAc01" parameterType="com.neusoft.mybatis.bean.Ac01">
7         insert into ac01(aac102,aac103,aac104,aac105,aac106,
8             aac107,aac108,aac109,aac110)
9             values("#{aac102},#{aac103},#{aac104},#{aac105},#{aac106},
10                 #{aac107},#{aac108},#{aac109},current_timestamp)
11     </insert>
12
13 </mapper>
```



# 代码开发(3)

- 编写MyBatis配置文件

mybatis-config.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "mybatis-3-config.dtd" >
3 <configuration>
4   <environments default="mysql_datasource">
5     <environment id="mysql_datasource">
6       <transactionManager type="JDBC"/>
7       <dataSource type="POOLED">
8         <property name="driver" value="com.mysql.jdbc.Driver"/>
9         <property name="url" value="jdbc:mysql://localhost/db01?characterEncoding=GBK"/>
10        <property name="username" value="root"/>
11        <property name="password" value=""/>
12      </dataSource>
13    </environment>
14  </environments>
15  <mappers>
16    <mapper resource="com/neusoft/mybatis/xml/Ac01Mapping.xml"/>
17  </mappers>
18 </configuration>
```

# 代码开发(4)

- 编写MyBatis测试类, 代码如下

```
public static void addAc01() throws Exception{
    //创建读取配置文件的输入流
    InputStream is=Resources.getResourceAsStream("mybatis-config.xml");
    //创建SqlSessionFactory构建对象
    SqlSessionFactoryBuilder builder=new SqlSessionFactoryBuilder();
    //通过SqlSessionFactory构建对象,生成SqlSessionFactory
    SqlSessionFactory factory=builder.build(is);
    //通过SqlSessionFactory创建SqlSession对象
    SqlSession session=factory.openSession();

    //创建映射类实例,并填充数据
    Ac01 ac01=new Ac01();
    ac01.setAac102("西门吹雪");    ac01.setAac103("230101191412132451");
    ac01.setAac104("男");          ac01.setAac105("2016-11-09");
    ac01.setAac106("刺客");        ac01.setAac107("110");
    ac01.setAac108("头儿");        ac01.setAac109("这个杀手有点冷");

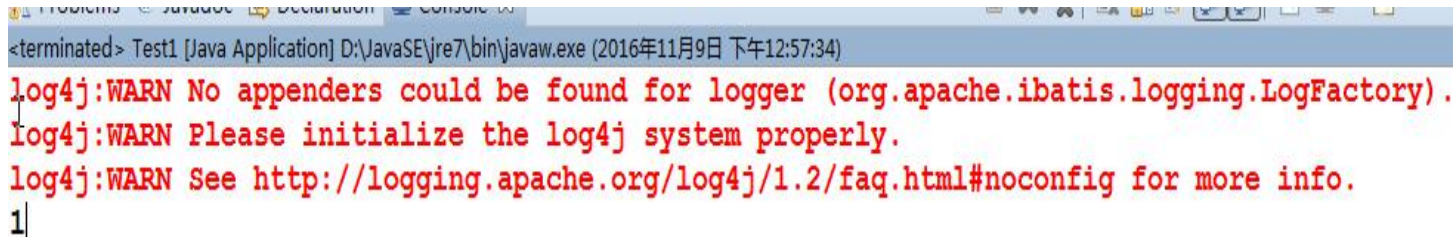
    // "Ac01Mapping.addAc01" 命名空间.id_Value
    int rs=session.insert("Ac01Mapping.addAc01",ac01);
    session.commit();
    System.out.println(rs);
    session.close();
}
```

# 代码开发 (5)

- 调用

```
public static void main(String[] args)
{
    try
    {
        Test1.addAc01();
    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
}
```

- 执行结果



<terminated> Test1 [Java Application] D:\JavaSE\jre7\bin\javaw.exe (2016年11月9日 下午12:57:34)

log4j:WARN No appenders could be found for logger (org.apache.ibatis.logging.LogFactory).

log4j:WARN Please initialize the log4j system properly.

log4j:WARN See <http://logging.apache.org/log4j/1.2/faq.html#noconfig> for more info.

1|

# 配置文件解析(1)

- MyBatis的数据源可以是如下三种之一
  - POOLED
    - 相当于数据库连接池
  - UNPOOLED
    - 相当于物理直连
  - JNDI
    - 基于应用服务器的数据源获取数据库连接



# 配置文件解析(2)

- 事务管理器TransactionManager

```
<transactionManager type="JDBC"/>
```

- MyBatis的事务管理器支持两种类型的事务
  - JDBC
    - 非分布式环境
  - MANAGED
    - 分布式计算环境(JTA)

# 迭代改进(1)

- SqlSession创建过程的改进
  - 在现有的开发模式下, SqlSession的创建过程过于繁琐, 而且每个测试方法中都编写SqlSession的创建过程, 那么代码会变得非常冗余, 因此, 接下来我们将会改进SqlSession的创建过程
- 改进思路
  - 在factory包下编写独立的类, 完成SqlSession创建及销毁过程的管理, 请看代码

# 迭代改进(2)

- SqlSession管理类代码:

```
1 package com.neusoft.mybatis.factory;
2
3+ import java.io.InputStream;
4
5
6
7
8
9 public final class MFactory
10 {
11     private static SqlSessionFactory factory=null;
12
13- static
14 {
15     try{
16         InputStream is=Resources.getResourceAsStream("mybatis-config.xml");
17         factory=new SqlSessionFactoryBuilder().build(is);
18     }
19     catch(Exception ex){
20         ex.printStackTrace();
21     }
22 }
23
24- public static SqlSession getSession(){
25     return factory.openSession();
26 }
27
28- public static void close(SqlSession session){
29     try{
30         if(session!=null){
31             session.close();
32         }
33     }catch (Exception e){
34         e.printStackTrace();
35     }
36 }
37 }
```



# 迭代改进(3)

- 改进后的测试类代码

```
29 private static void addAc01Test() throws Exception
30 {
31     Ac01 ac01=new Ac01();
32     ac01.setAac102("西门吹雪");
33     ac01.setAac103("110");
34     ac01.setAac104("1");
35     ac01.setAac105("1979-01-01");
36     ac01.setAac106("刺客");
37     ac01.setAac107("110");
38     ac01.setAac108("头儿");
39     ac01.setAac109("这个杀手有点冷");
40
41     SqlSession session=SessionFactory.getSession();
42     int rs=session.insert("Ac01Mapping.addAc01", ac01);
43     session.commit();
44     System.out.println(rs);
45     session.close();
46 }
```

# 迭代改进(4)

- 类型别名typeAliases

- 我们来看映射文件

```
<insert id="addAc01" parameterType="com.neusoft.mybatis.bean.Ac01">
```

路径越长越麻烦

SQL Code

```
</insert>
```

- 在MyBatis中,可以为映射类起别名,通过别名可以简化映射文件配置的复杂度,别名的配置需要在配置文件(**mybatis-config.xml**)中进行,同时,诸位需要注意编写位置:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
3   "mybatis-3-config.dtd" >
4 <configuration>
5
6   <typeAliases>
7     <typeAlias type="com.neusoft.mybatis.bean.Ac01" alias="ac01"/>
8   </typeAliases>
```

# 迭代改进 (5)

- 基于别名的映射文件配置

```
<insert id="addAc01" parameterType="ac01">  
    insert into ac01(aac102,aac103,aac104,aac105,aac106,  
                    aac107,aac108,aac109,aac110)  
        values({aac102},{aac103},{aac104},{aac105},{aac106},  
              {aac107},{aac108},{aac109},current_timestamp)  
</insert>
```

- 测试调用同前文

# 主键获取(1)



- 信息系统建设无法回避的一个问题就是多表级联录入,此时需要将主表主键提取出来,作为从表的外键,以保证数据的参照完整性.下面我们来提取主键,请看配置文件



# 主键获取(2)

```
<insert id="addAc01" parameterType="ac01">
  insert into ac01(aac102,aac103,aac104,aac105,aac106,
                  aac107,aac108,aac109,aac110)
    values({aac102},{aac103},{aac104},{aac105},{aac106},
          {aac107},{aac108},{aac109},current_timestamp)
  <selectKey keyProperty="aac101" order="AFTER" resultType="java.Lang.Integer">
    select last_insert_id();
  </selectKey>
</insert>
```

# 主键获取(4)

- 主键值获取完毕以后,会自动填充到映射类的主键属性中(先序获取和后序获取都一样).此时通过主键属性就可得到新增数据的主键值,请看测试代码

```
public static void addAc01Ext()  
{  
    SqlSession session=null;  
    try  
    {  
        //创建SqlSession  
        session=MyFactory.getSession();  
  
        //创建映射类实例,并填充数据  
        Ac01 ac01=new Ac01();  
        ac01.setAac102("血影狂刀");    ac01.setAac103("230101191412132451");  
        ac01.setAac104("男");          ac01.setAac105("2016-11-09");  
        ac01.setAac106("刺客");        ac01.setAac107("110");  
        ac01.setAac108("头儿");        ac01.setAac109("这个杀手有点冷");  
  
        // "Ac01Mapping.addAc01" 命名空间.id_Value  
        int rs=session.insert("Ac01Mapping.addAc01",ac01);  
        session.commit();  
        System.out.println(rs);  
        System.out.println("ac01.aac101="+ac01.getAac101());  
    }  
    finally  
    {  
        MyFactory.close(session);  
    }  
}
```

获取主键属性

# Log4j与执行期SQL显示

- 在MyBatis项目中,可以通过为系统配置日志框架的方式,显示执行期间的SQL,目前比较流行的日志框架是log4j.在本项目中,我们已经导入log4j的相关jar包,只需要配置相应属性文件即可

## log4j.properties

```
# Global logging configuration
#\u751F\u4EA7\u73AF\u5883\u914D\u7F6Einfo ERROR
log4j.rootLogger=DEBUG,stdout
# MyBatis logging configuration...
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```



# 空值的录入(1)

- 对于一个表而言,并不是所有的列,都需要录入数据,比如员工信息表中的备注.但是MyBatis在录入数据的时候,默认情况下,如果向可以为null的列输入空值时候,有时候(Oracle数据库)会报错,这个很奇怪,下面请看代码:

# 空值的录入(2)

- 测试代码

```
public static void addAc01Ext()  
{  
    SqlSession session=null;  
    try  
    {  
        //创建SqlSession  
        session=MyFactory.getSession();  
        //创建映射类实例,并填充数据  
        Ac01 ac01=new Ac01();  
        ac01.setAac102("血影狂刀");    ac01.setAac103("230101191412132451");  
        ac01.setAac104("男");          ac01.setAac105("2016-11-09");  
        ac01.setAac106("刺客");        ac01.setAac107("110");  
        ac01.setAac108("头儿");  
  
        //屏蔽备注信息,此时aac109为null,录入数据时候会报错  
        //ac01.setAac109("这个杀手有点冷");  
  
        session.insert("Ac01Mapping.addAc01",ac01);  
        session.commit();  
        System.out.println("ac01.aac101="+ac01.getAac101());  
    }  
    finally  
    {  
        MyFactory.close(session);  
    }  
}
```

# 空值的录入 (3)

- 运行信息节选

```
I  
ParameterMapping{property='aac109', mode=IN, javaType=class java.lang.String, jdbcType=null,
```

# 空值的录入(4)

- 解决方案
  - MyBatis在录入数据的时候,默认把映射类中每个属性的值都当做java的Object类型,进行转换和录入的.对于NULL值,需要将其转换成SQL可以识别的字符串类型,此时就不会报错了.(估计就是MyBatis数据类型上懵错了,嘿嘿)
  - 请看配置文件

```
<insert id="addAc01" parameterType="ac01" >
    INSERT INTO AC01 (AAC101,AAC102,AAC103,AAC104,AAC105,
        AAC106,AAC107,AAC108,AAC109,AAC110)
        VALUES (S_AC01.NEXTVAL,#{aac102},#{aac103},#{aac104},
            TO_DATE(#{aac105}, 'YYYY-MM-DD'),
            #{aac106},#{aac107},#{aac108},#{aac109,jdbcType=VARCHAR},SYSDATE)
    <selectKey keyProperty="aac101" order="AFTER" resultType="int">
        SELECT S_AC01.CURRVAL FROM DUAL
    </selectKey>
</insert>
```

# MyBatis总jdbcType类型列表

## MyBatis支持的JDBC类型

BIT	FLOAT	CHAR	TIMESTAMP	OTHER	UNDEFINED
TINYINT	REAL	VARCHAR	BINARY	BLOB	NVARCHAR
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB	NCHAR
INTEGER	NUMERIC	DATE	LONGVARBINARY	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	

上述数据类型的具体含义,大家可以参照JDK API手册中的java.sql.Types类



# 数据修改操作

```
<update id="modify" parameterType="ac01">
    update ac01 set aac103=#{aac103},aac104=#{aac104},aac105=#{aac105}
    where aac101=#{aac101}
</update>
```

```
private static void modifyAc01()
{
    SqlSession session=null;
    try{
        session=MyFactory.getSession();
        Ac01 ac01=new Ac01();
        ac01.setAac101(6);          ac01.setAac103("23010xxxxxx");
        ac01.setAac104("女");      ac01.setAac105("2010-01-01");

        String msg=session.update("Ac01Mapping.modifyAc01",ac01)>0?"修改成功":"修改失败";
        session.commit();
        System.out.println(msg);
    }finally{
        MyFactory.close(session);
    }
}
```

# 删除操作

```
<delete id="delete" parameterType="Integer">
    delete from ac01 where aac101=#{id}
</delete>
```

```
private static void deleteAc01()
{
    SqlSession session=null;
    try{
        session=MyFactory.getSession();
        String msg=session.delete("Ac01Mapping.deleteAc01", 6)>0?"删除成功":"删除失败!";
        session.commit();
        System.out.println(msg);
    }finally{
        MyFactory.close(session);
    }
}
```



# 单一实例查询(1)

```
<select id="findById" resultType="ac01" parameterType="Integer">
    select aac101,aac102,aac103,aac104,aac105,aac106
    from ac01
    where aac101=#{id}
</select>
```

```
private static void findById()
{
    SqlSession session=null;
    try
    {
        session=MyFactory.getSession();
        Ac01 ac01=session.selectOne("Ac01Mapping.findById", 7);
        System.out.println(ac01);
    }
    finally
    {
        MyFactory.close(session);
    }
}
```

# 单一实例查询的注意事项

- 如果查询字段中, 某列使用了数据库内置函数, 那么, 在没有起别名的情况下, 该列的查询结果无法填充映射类的相应属性, 此时查询结果中, 映射类的该属性为null
- 如果起了别名, 那么别名必须必须与映射类的属性名相同, 否则该列也不会填充映射类
- 如果SELECT中的某个字段在映射类中不存在对应属性, 此时不会报错, 但该字段不会封装入映射类

# 单一实例查询(2)

```
<select id="findById2" resultType="ac01" parameterType="Integer">
    select aac101,concat(aac102,aac103),aac104,aac105,aac106
    from ac01
    where aac101=#{id}
</select>
```

```
private static void findById()
{
    SqlSession session=null;
    try
    {
        session=MyFactory.getSession();
        Ac01 ac01=session.selectOne("Ac01Mapping.findById", 7);
        System.out.println(ac01);
    }
    finally
    {
        MyFactory.close(session);
    }
}
```

# 批量查询的List+Bean数据封装

```
<select id="queryAll" resultType="ac01" >
    select aac101,aac102,aac103,aac104,aac105,aac106
    from ac01
</select>
```

```
private static void queryAll()
{
    SqlSession session=null;
    try
    {
        session=MyFactory.getSession();
        List<Ac01> rows=session.selectList("Ac01Mapping.queryAll");
        System.out.println(rows);
    }
    finally
    {
        MyFactory.close(session);
    }
}
```



# 批量查询的List+Map数据封装

```
<select id="queryAllMap" resultType="java.util.Map" >
    select aac101,aac102,aac103,aac104,aac105,aac106
    from ac01
</select>
```

```
private static void queryAllMap()
{
    SqlSession session=null;
    try
    {
        session=MyFactory.getSession();
        List<Ac01> rows=session.selectList("Ac01Mapping.queryAllMap");
        System.out.println(rows);
    }
    finally
    {
        MyFactory.close(session);
    }
}
```

# 动态SQL与不定条件查询(1)

```
<select id="query" parameterType="m" resultType="khm">
  SELECT A.PID,A.PNAME,A.PSEX,A.PDATE,A.PMONEY,A.PNOTE
  FROM PERSON A
  <where>
    <if test="pname!=null">
      AND A.PNAME LIKE #{pname}
    </if>
    <if test="psex!=null and psex!=''" >
      AND A.PSEX=#{psex}
    </if>
    <if test="bdate!=null and bdate!=''">
      AND A.PDATE >=TO_DATE("#{bdate}','YYYY-MM-DD')
    </if>
    <if test="edate!=null and edate!=''">
      AND A.PDATE <=TO_DATE("#{edate}','YYYY-MM-DD')
    </if>
  </where>
</select>
```



# 动态SQL与不定条件查询(2)

```
<select id="query2" parameterType="m" resultType="kkm">
    SELECT A.PID,A.PNAME,A.PSEX,A.PDATE,A.PMONEY,A.PNOTE
    FROM PERSON A
    <trim prefix="where" prefixOverrides="AND | OR">
        <if test="pname!=null">
            AND A.PNAME LIKE #{pname}
        </if>
        <if test="psex!=null and psex!=''" >
            AND A.PSEX=#{psex}
        </if>
        <if test="bdate!=null and bdate!=''">
            AND A.PDATE >=TO_DATE("#{bdate}','YYYY-MM-DD')
        </if>
        <if test="edate!=null and edate!=''">
            AND A.PDATE <=TO_DATE("#{edate}','YYYY-MM-DD')
        </if>
    </trim>
</select>
```

# foreach与数据批量删除

- foreach 用于遍历数组进行批量条件的拼接
- 例子1:foreach与OR关联

```
<delete id="bdel1" parameterType="m">
  delete from ac01
  <if test="idlist!=null">
    <where>
      <foreach collection="idlist" item="id">
        or aac101=#{id}
      </foreach>
    </where>
  </if>
</delete>
```

# foreach与数据批量删除

- foreach 用于遍历数组进行批量条件的拼接
- 例子2: 与IN/NOT IN 关联

```
<delete id="bdel1" parameterType="m">
  delete from ac01
  <if test="idlist!=null">
    <where>
      I aac101 in
      <foreach collection="idlist" item="id" open="(" separator="," close=")">
        #{id}
      </foreach>
    </where>
  </if>
</delete>
```

# SQL段引用

```
<delete id="bdel1" parameterType="m">
  <include refid="baseDelete"/>
  <if test="idlist!=null">
    <where>
      aac101 in
      <foreach collection="idlist" item="id" open="(" separator="," close=")">
        #{id}
      </foreach>
    </where>
  </if>
</delete>
```

---

```
<sql id="baseDelete" >
  delete from ac01
</sql>
```



# 伪条件应用

```
<delete id="bdel1" parameterType="m">
  <include refid="baseDelete"/>
  <choose>
    <when test="idlist!=null and idlist!=''">
      <where>
        aac101 in
        <foreach collection="idlist" item="id" open="(" separator="," close=")">
          #{id}
        </foreach>
      </where>
    </when>
    <otherwise>
      where 1>2;
    </otherwise>
  </choose>
</delete>

<sql id="baseDelete" >
  delete from ac01|
</sql>
```

# MySQL的时间处理(1)

- Date/Time to Str

```

1 select aac110,
2       date_format(aac110, '%Y-%m-%d'),
3       date_format(aac110, '%H:%i:%S'),
4       date_format(aac110, '%Y-%m-%d %H:%i:%S')
5 from ac01;

```

信息	结果1	概况	状态
aac110	date_format(aac110,'%Y-%m-%d')	date_format(aac110,'%H:%i:%S')	date_format(aac110,'%Y-%m-%d %H:%i:%S')
2017-06-25 21:39:20	2017-06-25	21:39:20	2017-06-25 21:39:20
2017-06-25 21:50:19	2017-06-25	21:50:19	2017-06-25 21:50:19
2017-06-25 21:59:12	2017-06-25	21:59:12	2017-06-25 21:59:12
2017-06-25 22:09:07	2017-06-25	22:09:07	2017-06-25 22:09:07
2017-06-25 22:09:37	2017-06-25	22:09:37	2017-06-25 22:09:37
2017-06-25 22:11:57	2017-06-25	22:11:57	2017-06-25 22:11:57



# MySQL的时间处理(2)

- MySQL Str to Date

```
1 select str_to_date('2017-06-25', '%Y-%m-%d'),  
2         str_to_date('2017-06-25 23:44:45', '%Y-%m-%d %H:%i:%s')  
3 from ac01;  
4
```

信息 结果1 情况 状态

str_to_date('2017-06-25', '%Y-%m-%d'), str_to_date('2017-06-25 23:44:45', '%Y-%m-%d %H:%i:%s')	
2017-06-25	2017-06-25 23:44:45
2017-06-25	2017-06-25 23:44:45
2017-06-25	2017-06-25 23:44:45
2017-06-25	2017-06-25 23:44:45
2017-06-25	2017-06-25 23:44:45
2017-06-25	2017-06-25 23:44:45

# MySQL的时间处理(3)

- 日期范围查询

```
1 select *,date(aac110)
2   from ac01 a
3  where date(aac110)>='2017-06-25'
4    and date(aac110)<=current_date
```

信息	结果1	概况	状态							
AAC101	AAC102	AAC103	AAC104	AAC105	AAC106	AAC107	AAC108	AAC109	AAC110	date(aac110)
▶	1 西门吹雪	110	1	1979-01-01	刺客	110	头儿	这个杀手有点	2017-06-25 21:39:20	2017-06-25
	2 西门吹雪	110	1	1979-01-01	刺客	110	头儿	这个杀手有点	2017-06-25 21:50:19	2017-06-25
	3 西门吹雪	110	1	1979-01-01	刺客	110	头儿	这个杀手有点	2017-06-25 21:59:12	2017-06-25
	9 西门吹雪	110	1	1979-01-01	刺客	110	头儿	这个杀手有点	2017-06-25 22:09:07	2017-06-25
	11 西门1	110	1	1979-01-01	刺客	110	头儿	这个杀手有点	2017-06-25 22:09:37	2017-06-25
	12 西门1	110	1	1979-01-01	刺客	110	头儿	这个杀手有点	2017-06-25 22:11:57	2017-06-25