

目录

目录	1
easymybatis开发文档	4
easymybatis开发文档	4
架构组成	4
运行流程	5
快速开始	5
创建springboot项目	5
导入项目	5
添加maven依赖	6
添加数据库配置	6
添加Java文件	6
添加测试用例	7
查询	8
分页查询	8
方式1	8
方式2	8
返回结果集和总记录数	8
完整代码	9
联表分页查询	10
根据参数字段查询	11
查询姓名为张三的用户	11
查询姓名为张三并且拥有的钱大于100块	11
查询姓名为张三并带分页	11
查询钱最多的前三名	11
将参数放在对象中查询	11
使用普通bean查询	12
@Condition注解	12
IN查询	13
排序查询	13
多表关联查询	13
使用@Select查询	14
Query类详解	14
参数介绍	15
分页参数	15
排序参数	15
条件参数	15
字段参数	16
主键策略设置	16
主键自增	16
主键使用uuid	16
自定义uuid	17
字段自动填充	18
填充器设置	18
自定义填充器	19
实战(springboot)	19
指定目标类	19
高级匹配	20
Entity中使用枚举字段	20
第一步	21

第二步	21
SQL写在xml中	22
配置说明	22
其他功能	23
指定外部模板	23
全局Dao	23
乐观锁	24
逻辑删除	24
多数据源配置	25
[附录]velocity变量说明	29
\${context}	29
\${pk}	29
\${table}	30
#foreach(\$column in \$columns)...#end	30
三分钟深入了解easymybatis	31
简单介绍	31
架构组成	31
运行流程	31
快速上手	32
意见交流	33
完整测试用例	33
使用springboot项目快速搭建	40
创建springboot项目	40
导入项目	40
添加maven依赖	40
添加数据库配置	40
添加Java文件	40
添加测试用例	41
查询	43
分页查询	43
方式1	43
方式2	43
返回结果集和总记录数	43
完整代码	44
更多查询方式可参阅：Query类详解	44
参数查询	45
根据字段查询	45
查询姓名为张三的用户	45
查询姓名为张三并且拥有的钱大于100块	45
查询姓名为张三并带分页	45
查询钱最多的前三名	45
将参数放在对象中查询	45
使用@ValueField注解查询	46
更多查询方式可参阅：Query类详解	46
多表关联查询	47
关于easymybatis	48
Query类详解	49
参数介绍	49
分页参数	49
排序参数	49

条件参数	49
字段参数	50
关于easymybatis	51
实体类主键策略设置	52
主键自增	52
主键使用uuid	52
JavaBean中使用枚举字段	53
第一步	53
第二步	53
字段自动填充	55
自定义填充器	56
实战	56
指定目标类	56
EasymybatisConfig配置项说明	58
springboot 配置方式：	58
注解使用方式：	58
xml文件注入方式：	58
camel2underline	59
mapperExecutorPoolSize	59
templateClasspath	59
commonSqlClasspath	59
mapperSaveDir	59
数据库增减字段后的SQL维护	60
easymybatis的设计初衷	61
驼峰转下划线形式（默认开启）	62
TUserDaoTest.java	63
image	69
LOGO	69
架构	69
运行流程	69
Dao层方法	70
查询条件	70
Home	72

easymybatis开发文档

easymybatis开发文档

easymybatis是一个mybatis增强类库，目的为简化mybatis的开发，让开发更高效。

- git地址：[easymybatis](#)
- demo地址：[demo with springboot](#)
- QQ交流群:328419269

easymybatis的特性如下：

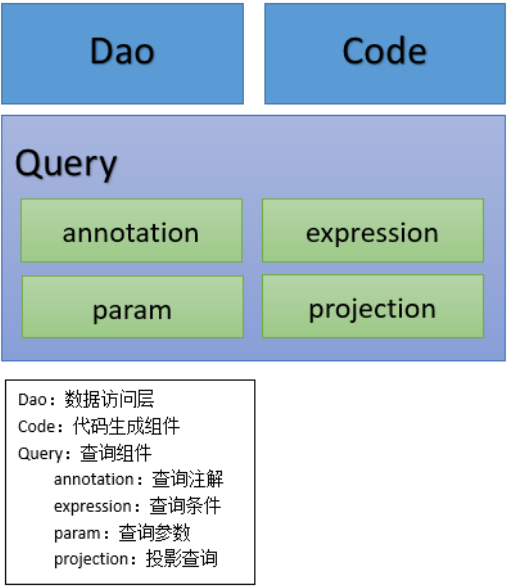
- 无需编写xml文件即可完成CRUD操作。
- 支持多表查询、聚合查询、分页查询（支持多种数据库）。
- 支持批量添加，指定字段批量添加。
- 支持Dao层访问控制，如某个dao只有查询功能，某个dao有crud功能等。
- 支持自定义sql，sql语句可以写在配置文件中，同样支持mybatis标签。
- 支持mysql，sqlserver，oracle，其它数据库扩展方便（增加一个模板文件即可）。
- 使用方式不变，与Spring集成只改了一处配置。
- 支持与springboot集成。
- mybatis参数设置灵活，继承mybatis官方设置方式。
- 轻量级，无侵入性，可与传统mybatis用法共存。
- 没有修改框架源码(无插件)，可同时使用官方提供的功能。

easymybatis支持的功能如下：

- 基本的CRUD
- 主键策略设置
- 字段填充功能
- 枚举属性
- 全局Dao
- 乐观锁
- 逻辑删除

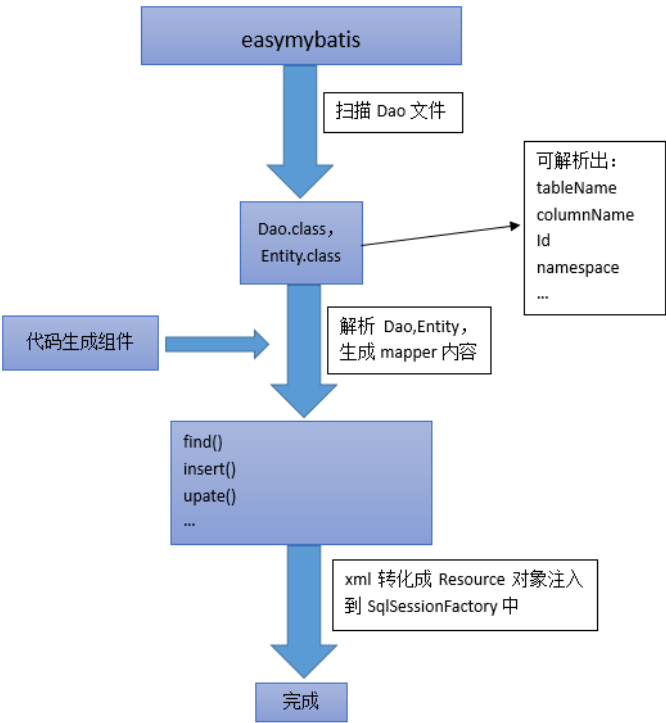
架构组成

easymybatis的架构如下：



运行流程

easymybatis的运行流程图：

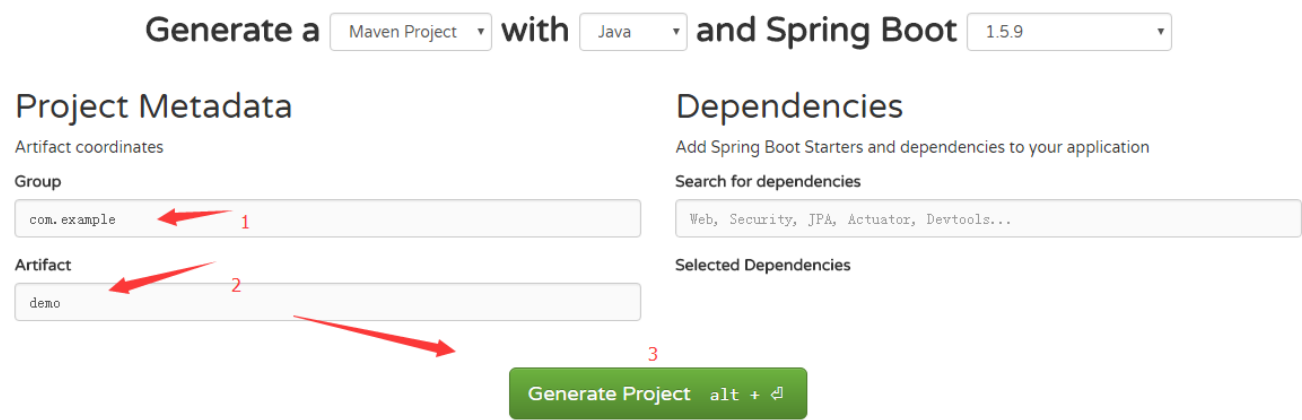


- 1. 服务器启动的时候easymybatis负责扫描Dao.java。
- 2. 扫描完成后解析出Dao.class以及实体类Entity.class。
- 3. 代码生成组件根据Dao.class和Entity.class生成mapper文件内容，生成方式由velocity模板指定。
- 4. 把mapper文件内容转化成Resource对象设置到SqlSessionFactory中。

快速开始

创建springboot项目

访问<http://start.spring.io/> 生成一个springboot空项目，输入Group，Artifact点生成即可，如图：



点击Generate Project，下载demo.zip

导入项目

将下载的demo.zip解压，然后导入项目。eclipse中右键 -> Import... -> Existing Maven Project，选择项目文件夹。导入到eclipse中后等待maven相关

jar包下载。

添加maven依赖

jar包下载完成后，打开pom.xml，添加如下依赖：

```
<!-- easymybatis的starter -->
<dependency>
  <groupId>net.oschina.durcframework</groupId>
  <artifactId>easymybatis-spring-boot-starter</artifactId>
  <version>1.8.4</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

添加数据库配置

在application.properties中添加数据库配置

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/stu?useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull
spring.datasource.username=root
spring.datasource.password=root
```

添加Java文件

假设数据库中有张t_user表，DDL如下：

```
CREATE TABLE `t_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT 'ID',
  `username` varchar(255) DEFAULT NULL COMMENT '用户名',
  `state` tinyint(4) DEFAULT NULL COMMENT '状态',
  `isdel` bit(1) DEFAULT NULL COMMENT '是否删除',
  `remark` text COMMENT '备注',
  `add_time` datetime DEFAULT NULL COMMENT '添加时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户表';
```

我们加入对应的实体类和Dao:

- TUser.java :

```
@Table(name = "t_user")
public class TUser {
    @Id
    @Column(name="id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id; // ID
    private String username; // 用户名
    private Byte state; // 状态
    private Boolean isdel; // 是否删除
    private String remark; // 备注
    private Date addTime; // 添加时间

    // 省略 getter setter
}
```

实体类文件采用和hibernate相同的方式，您可以使用我们的代码生成工具生成 <https://gitee.com/durcframework/easymybatis-generator>

- TUserDao.java :

```
public interface TUserDao extends CrudDao<TUser> {  
}
```

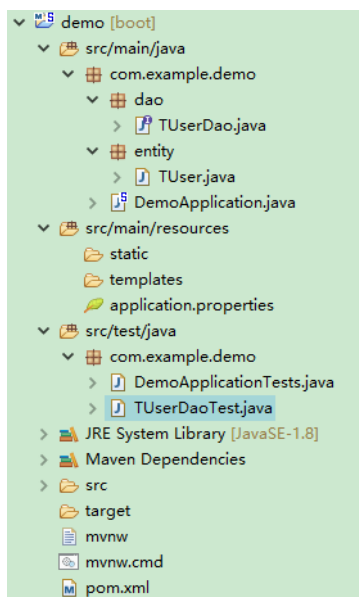
TUserDao继承CrudDao即可，这样这个Dao就拥有了CRUD功能。

添加测试用例

```
public class TUserDaoTest extends DemoApplicationTests {  
  
    @Autowired  
    TUserDao dao;  
  
    @Test  
    public void testInsert() {  
        TUser user = new TUser();  
        user.setIsdel(false);  
        user.setRemark("testInsert");  
        user.setUsername("张三");  
  
        dao.save(user);  
  
        System.out.println("添加后的主键:" + user.getId());  
    }  
  
    @Test  
    public void testGet() {  
        TUser user = dao.get(3);  
        System.out.println(user);  
    }  
  
    @Test  
    public void testUpdate() {  
        TUser user = dao.get(3);  
        user.setUsername("李四");  
        user.setIsdel(true);  
  
        int i = dao.update(user);  
        System.out.println("testUpdate --> " + i);  
    }  
  
    @Test  
    public void testDel() {  
        TUser user = new TUser();  
        user.setId(3);  
        int i = dao.del(user);  
        System.out.println("del --> " + i);  
    }  
}
```

然后运行单元测试，运行成功后表示项目已经搭建完毕了。

最后项目结构图：



查询

本小节主要讲解easymybatis的查询功能。easymybatis提供丰富的查询方式，满足日常查询所需。

分页查询

方式1

前端传递两个分页参数pageIndex，pageSize

```
// http://localhost:8080/page1?pageIndex=1&pageSize=10
@GetMapping("page1")
public List<TUser> page1(int pageIndex,int pageSize) {
    Query query = new Query().page(pageIndex, pageSize);
    List<TUser> list = dao.find(query);
    return list;
}
```

方式2

PageParam里面封装了pageIndex，pageSize参数

```
// http://localhost:8080/page2?pageIndex=1&pageSize=10
@GetMapping("page2")
public List<TUser> page2(PageParam param) {
    Query query = param.toQuery();
    List<TUser> list = dao.find(query);
    return list;
}
```

返回结果集和总记录数

方式1和方式2只能查询结果集，通常我们查询还需返回记录总数并返回给前端，easymybatis的处理方式如下：

```
// http://localhost:8080/page3?pageIndex=1&pageSize=10
@GetMapping("page3")
public Map<String,Object> page3(PageParam param) {
    Query query = param.toQuery();
    List<TUser> list = dao.find(query);
    long total = dao.countTotal(query);
}
```

```

Map<String,Object> result = new HashMap<String, Object>();
result.put("list", list);
result.put("total", total);

return result;
}

```

easymybatis提供一种更简洁的方式来处理：

```

// http://localhost:8080/page4?pageIndex=1&pageSize=10
@GetMapping("page4")
public PageInfo<TUser> page4(PageParam param) {
    PageInfo<TUser> result = QueryUtils.query(dao, param);
    return result;
}

```

PageInfo里面包含了List，total信息，还包含了一些额外信息，完整数据如下：

```

{
  "currentPageIndex": 1, // 当前页
  "firstPageIndex": 1, // 首页
  "lastPageIndex": 2, // 尾页
  "list": [ // 结果集
    {},
    {}
  ],
  "nextPageIndex": 2, // 下一页
  "pageCount": 2, // 总页数
  "pageIndex": 1, // 当前页
  "pageSize": 10, // 每页记录数
  "prePageIndex": 1, // 上一页
  "start": 0,
  "total": 20 // 总记录数
}

```

完整代码

```

@RestController
public class UserSchController {

    @Autowired
    private TUserDao dao;

    // http://localhost:8080/page1?pageIndex=1&pageSize=10
    @GetMapping("page1")
    public List<TUser> page1(int pageIndex,int pageSize) {
        Query query = new Query().page(pageIndex, pageSize);
        List<TUser> list = dao.find(query);
        return list;
    }

    // http://localhost:8080/page2?pageIndex=1&pageSize=10
    @GetMapping("page2")
    public List<TUser> page2(PageParam param) {
        Query query = param.toQuery();
        List<TUser> list = dao.find(query);
        return list;
    }

    // http://localhost:8080/page3?pageIndex=1&pageSize=10
    @GetMapping("page3")
    public Map<String,Object> page3(PageParam param) {
        Query query = param.toQuery();
        List<TUser> list = dao.find(query);
        long total = dao.countTotal(query);
    }
}

```

```

Map<String,Object> result = new HashMap<String, Object>();
result.put("list", list);
result.put("total", total);

return result;
}

// http://localhost:8080/page4?pageIndex=1&pageSize=10
@GetMapping("page4")
public PageInfo<TUser> page4(PageParam param) {
    PageInfo<TUser> result = QueryUtils.query(dao, param);
    return result;
}
}

```

联表分页查询

- 方式1：代码形式

```

/**
 * 联表分页
 * SELECT t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`
 * FROM `t_user` t
 * LEFT JOIN user_info t2 ON t.id = t2.user_id
 * WHERE t.isdel = 0 LIMIT ?,?
 */
@Test
public void testJoinPage() {
    Query query = Query.build()
        // 左连接查询,主表的alias默认为t
        .join("LEFT JOIN user_info t2 ON t.id = t2.user_id")
        .page(1, 5);

    List<TUser> list = dao.find(query);

    System.out.println("=====");
    for (TUser user : list) {
        System.out.println(
            user.getId()
            + " " + user.getUsername()
        );
    }
    System.out.println("=====");
}

```

- 方式2：xml形式：

xml:

```

<select id="findJoinPage"
parameterType="net.oschina.durcframework.easymybatis.query.Pageable"
resultMap="baseResultMap">
    SELECT t.*,t2.city,t2.address
    FROM t_user t LEFT JOIN user_info t2 ON t.id = t2.user_id
    <include refid="common.where" />
    <include refid="common.orderBy" />
    <include refid="common.limit" />
</select>

```

java :

```

@Test
public void testJoinPageXml() {
    Query query = Query.build()
        .page(1, 5);
}

```

```

List<TUser> list = dao.findJoinPage(query);

System.out.println("=====");
for (TUser user : list) {
    System.out.println(
        user.getId()
        + " " + user.getUsername()
    );
}
System.out.println("=====");
}

```

根据参数字段查询

查询姓名为张三的用户

```

// http://localhost:8080/sch?username=张三
@GetMapping("sch")
public List<TUser> sch(String username) {
    Query query = new Query();
    query.eq("username", username);
    List<TUser> list = dao.find(query);
    return list;
}

```

查询姓名为张三并且拥有的钱大于100块

```

// http://localhost:8080/sch2?username=张三
@GetMapping("sch2")
public List<TUser> sch2(String username) {
    Query query = new Query();
    query.eq("username", username).gt("money", 100);
    List<TUser> list = dao.find(query);
    return list;
}

```

查询姓名为张三并带分页

```

// http://localhost:8080/sch3?username=张三&pageIndex=1&pageSize=5
@GetMapping("sch3")
public List<TUser> sch3(String username, PageParam param) {
    Query query = param.toQuery();
    query.eq("username", username);
    List<TUser> list = dao.find(query);
    return list;
}

```

查询钱最多的前三名

```

// http://localhost:8080/sch4
@GetMapping("sch4")
public List<TUser> sch4() {
    Query query = new Query();
    query.orderby("money", Sort.DESC) // 按金额降序
        .page(1, 3);
    List<TUser> list = dao.find(query);
    return list;
}

```

将参数放在对象中查询

```
// http://localhost:8080/sch5?username=张三
@GetMapping("sch5")
public List<TUser> sch5(UserParam userParam) {
    Query query = userParam.toQuery();
    query.eq("username", userParam.getUsername());
    List<TUser> list = dao.find(query);
    return list;
}
```

UserParam继承PageSortParam类，表示支持分页和排序查询

使用普通bean查询

假设有个User类如下

```
public class User {
    private Integer id;
    private String userName;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

我们将这个类作为查询参数,那么在springmvc中可以这样写:

```
@GetMapping(path="findUserBean.do")
public List<User> findUser(User user) {
    Query query = Query.build(user);
    List<User> list = dao.find(query);
    return list;
}
```

Query query = Query.buildFromBean(user);这句是将User中的属性转换成对应条件,假设userName的值为"jim",那么会封装成一个条件where user_name='jim'

浏览器输入链接:<http://localhost:8080/easymybatis-springmvc/findUserBean.do?userName=jim> 后台将会执行如下SQL:

```
SELECT id,user_name FROM user t WHERE t.user_name = ?
```

?的值为jim

@Condition注解

@Condition注解用来强化查询，有了这个注解可以生成各种查询条件。

@Condition注解有三个属性：

- joint：表达式之间的连接符,AND|OR,默认AND
- column：数据库字段名，可选
- operator：连接符枚举，存放了等于、大于、小于等连接符

查询

如果要查询id大于2的用户只需在get方法上加上一个@Condition注解即可:

```
@Condition(operator=Operator.gt)
public Integer getId() {
    return this.id;
}
```

这样,当id有值时,会封装成一个where id>2的条件

- 需要注意的是,如果不指定column属性,系统会默认取get方法中属性名,然后转换成数据库字段名。如果需要指定数据库字段名的话,可以使用@Condition的column属性。

```
public Integer get++UserName++() { return this.userName; }
```

这种情况下会取下划线部分字段,然后转换成数据库字段名。

```
@Condition(column="username") // 显示指定字段名
public Integer getUserName() {
    return this.userName;
}
```

使用@Condition可以生产更加灵活的条件查询,比如需要查询日期为2017-12-1~2017-12-10日的记录,我们可以这样写:

```
@Condition(column="add_date",operator=Operator.ge)
public Date getStartDate() {
    return this.startDate;
}

@Condition(column="add_date",operator=Operator.lt)
public Date getEndDate() {
    return this.endDate;
}
```

转换成SQL语句:

```
t.add_date>='2017-12-1' AND t.add_date<'2017-12-10'
```

IN查询

假设前端页面传来多个值比如checkbox勾选多个id=[1,2],那么我们在User类里面可以用Integer[]或List来接收。

```
private Integer[] idArr;

public void setIdArr(Integer[] idArr) {this.idArr = idArr;}

@Condition(column="id")
public Integer[] getIdArr() {return this.idArr;}
```

这样会生成where id IN(1,2)条件。

排序查询

// 根据添加时间倒序

```
Query query = new Query();
query.addSort("create_time",Sort.DESC);
dao.find(query);
```

多表关联查询

多表关联查询使用的地方很多,比如需要关联第二张表,获取第二张表的几个字段,然后返回给前端。

easyMybatis的用法如下：假如我们需要关联第二张表，并且获取第二张表里的city，address字段。步骤如下：

- 在实体类中添加city，address字段，并标记@Transient注解。只要不是主表中的字段都要加上@Transient

```
@Transient
private String city;
@Transient
private String address;
```

```
// getter setter
```

- 接下来是查询代码：

```
Query query = new Query();
// 添加第二张表的字段,跟主表字段一起返回
query.addOtherColumns(
    "t2.city"
    ,"t2.address"
);
// 左连接查询,主表的alias默认为t
query.join("LEFT JOIN user_info t2 ON t.id = t2.user_id");
// 添加查询条件
query.eq("t.username", "张三");

List<TUser> list = dao.find(query);
```

得到的SQL语句：

```
SELECT
  t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`
, t2.city, t2.address
FROM `t_user` t LEFT JOIN user_info t2 ON t.id = t2.user_id
WHERE t.username = ?
LIMIT ?,?
```

关联了user_info表之后，还可以筛选user_info的数据，也就是针对user_info表进行查询：

```
query.eq("t2.city", "杭州");
```

使用@Select查询

@Select注解是mybatis官方提供的一个功能，easyMybatis可以理解为是官方的一种扩展，因此同样支持此功能。在Dao中添加如下代码：

```
@Select("select * from t_user where id=#{id}")
TUser selectById(@Param("id") int id);
```

编写测试用例

```
@Test
public void testSelectById() {
    TUser user = dao.selectById(3);

    System.out.println(user.getUsername());
}
```

除了@Select之外，还有@Update，@Insert，@Delete，这里就不多做演示了。

Query类详解

Query是一个查询参数类，通常配合Dao一起使用。

参数介绍

Query里面封装了一系列查询参数，主要分为以下几类：

- 分页参数：设置分页
- 排序参数：设置排序字段
- 条件参数：设置查询条件
- 字段参数：可返回指定字段

下面逐个讲解每个参数的用法。

分页参数

一般来说分页的使用比较简单，通常是两个参数， pageIndex：当前页索引， pageSize：每页几条数据。 Query类使用**page(pageIndex, pageSize)**方法来设置。 假如我们要查询第二页，每页10条数据，代码可以这样写：

```
Query query = new Query().page(2, 10);
List<User> list = dao.find(query);
```

如果要实现不规则分页，可以这样写：

```
Query query = new Query().limit(3,5);
// 对应mysql：limit 3,5
```

- 如果要查询所有数据,则可以这样写:

```
Query query = new Query();
List<User> list = dao.findAll(query);
```

排序参数

设置排序：

```
orderBy(String sortname, Sort sort)
```

其中sortname为数据库字段，非javaBean属性

- orderBy(String sortname, Sort sort)则可以指定排序方式，Sort为排序方式枚举 假如要按照添加时间倒序，可以这样写：

```
Query query = new Query().orderBy("create_time",Sort.DESC);
dao.find(query);
```

添加多个排序字段可以在后面追加：

```
query.orderby("create_time",Sort.DESC).orderBy("id",Sort.ASC);
```

条件参数

条件参数是用的最多一个，因为在查询中往往需要加入各种条件。 easymybatis在条件查询上面做了一些封装，这里不做太多讲解，只讲下基本的使用方法，以后会单独开一篇文章来介绍。感兴趣的同学可以自行查看源码，也不难理解。

条件参数使用非常简单，Query对象封装一系列常用条件查询。

- 等值查询eq(String columnName, Object value)，columnName为数据库字段名，value为查询的值 假设我们要查询姓名为张三的用户，可以这样写：

```
Query query = new Query();
query.eq("username","张三");
List<User> list = dao.find(query);
```

通过方法名即可知道eq表示等于'='，同理lt表示小于<,gt表示大于>

查询方式	说明
eq	等于=
gt	大于>
lt	小于<
ge	大于等于>=
le	小于等于<=
notEq	不等于<>
like	模糊查询
in	in()查询
notIn	not in()查询
isNull	NULL值查询
notNull	IS NOT NULL
notEmpty	字段不为空，非NULL且有内容
isEmpty	字段为NULL或者为''

如果上述方法还不能满足查询需求的话，我们可以使用自定sql的方式来编写查询条件，方法为：

```
Query query = new Query();
query.sql(" username='jim' OR username='Tom'");
```

注意：sql()方法不会处理sql注入问题，因此尽量少用。

字段参数

在某些场景下，我们只想获取表里面几个字段的信息，不想查询所有字段。此时使用方式如下：

```
Query query = new Query();
// 只返回id,username
query.setColumns(Arrays.asList("id","username"));
List<TUser> list = dao.find(query);
```

这里的"id"，"username"都为数据库字段。

主键策略设置

跟hibernate的主键生成策略一致

主键自增

数据库主键设置自增后，这样设置：

```
@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

这样在做insert后，id会自动填充自增后的值。

主键使用uuid

数据库主键是varchar类型，insert后自动填充uuid，并返回。

```
@Id
@Column(name = "id")
@GeneratedValue(generator = "system-uuid")
private String id;
```

这样在做insert后，id字段会自动填充uuid。

- 注：uuid的生成方式是调用数据库底层实现，如MySQL的实现方式为：SELECT UUID()

自定义uuid

如果不希望使用底层数据库的uuid，可自定义自己实现，实现方式如下：

- 首先entity中的id字段@GeneratedValue注解设置成AUTO

```
@Id
@Column(name = "id")
@GeneratedValue(strategy=GenerationType.AUTO)
private String id;
```

- 创建一个类UUIDFill并继承FillHandler类

```
public class UUIDFill extends FillHandler<String> {
    @Override
    public String getColumnName() {
        return "id"; // 作用在id字段上
    }
    @Override
    public FillType getFillType() {
        return FillType.INSERT; // INSERT时触发
    }
    @Override
    protected Object getFillValue(String defaultValue) {
        return UUID.randomUUID().toString(); // 自定义的uuid生成方式
    }
}
```

- 在application.properties中添加

```
# key:填充器全路径类名,value:构造函数参数值,没有可不填
mybatis.fill.com.xx.aa.UUIDFill=
```

格式为mybatis.fill.类路径=构造参数(没有可不填)

到此已经可以了，当进行insert操作后，id字段会自动插入自定义的uuid。

但是使用过程中还会有个问题，如果数据库中既有自增主键的表，也有自定义UUID主键的表，那么上面的做法就没办法区分了。因此我们要找出自定义UUID主键的表，解决办法是重写FillHandler的match(Class<?> entityClass, Field field, String columnName)方法，完整的代码如下：

```
public class UUIDFill extends FillHandler<String> {
    /* 重写方法，自定义匹配
       entityClass  实体类class
       field        字段信息
       columnName   给定的数据库字段名
    */
    @Override
    public boolean match(Class<?> entityClass, Field field, String columnName) {
        boolean isPk = field.getAnnotation(Id.class) != null; // 是否有@Id注解

        GeneratedValue gv = field.getAnnotation(GeneratedValue.class);
        boolean isAuto = gv != null && gv.strategy() == GenerationType.AUTO; // 是否有@GeneratedValue注解，并且策略是AUTO

        return isPk && isAuto;
    }
}
```

```
@Override
public String getColumnName() {
    return "id"; // 作用在id字段上
}
@Override
public FillType getFillType() {
    return FillType.INSERT; // INSERT时触发
}
@Override
protected Object getFillValue(String defaultValue) {
    return UUID.randomUUID().toString(); // 自定义的uuid生成方式
}
}
```

这样就能区分出自增主键和自定义主键了。

自定义uuid生成的配置方式采用的是easymybatis提供的字段填充功能，具体说明可参考 [字段自动填充](#) 小节。

字段自动填充

填充器设置

假设数据库表里面有两个时间字段gmt_create,gmt_update。

当进行insert操作时gmt_create，gmt_update字段需要更新。当update时，gmt_update字段需要更新。

通常的做法是通过Entity手动设置：

```
User user = new User();
user.setGmtCreate(new Date());
user.setGmtUpdate(new Date());
```

因为表设计的时候大部分都有这两个字段，所以对每张表都进行手动设置的话很容易错加、漏加。 easymybatis提供了两个辅助类DateFillInsert和DateFillUpdate，用来处理添加修改时的时间字段自动填充。配置了这两个类之后，时间字段将会自动设置。

springboot项目配置方式如下：

在application.properties中添加

```
mybatis.fill.net.oschina.durcframework.easymybatis.support.DateFillInsert=
mybatis.fill.net.oschina.durcframework.easymybatis.support.DateFillUpdate=
```

如果要指定字段名，可以写成：

```
mybatis.fill.net.oschina.durcframework.easymybatis.support.DateFillInsert=add_time
```

在springmvc的xml中配置如下：

```
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
<property name="dataSource" ref="dataSource" />
<property name="configLocation">
<value>classpath:mybatis/mybatisConfig.xml</value>
</property>
<property name="mapperLocations">
<list>
<value>classpath:mybatis/mapper/*.xml</value>
</list>
</property>

<!-- 以下是附加属性 -->

<!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
```

```

    多个用;隔开
-->
<property name="basePackage" value="com.myapp.dao" />
<property name="config">
    <bean class="net.oschina.durcframework.easymybatis.EasymybatisConfig">
        <!-- 定义填充器 -->
        <property name="fills">
            <list>
                <bean class="net.oschina.durcframework.easymybatis.support.DateFillInsert"/>
                <bean class="net.oschina.durcframework.easymybatis.support.DateFillUpdate"/>
            </list>
        </property>
    </bean>
</property>
</bean>

```

自定义填充器

除了使用easymybatis默认提供的填充之外，我们还可以自定义填充。

自定义填充类要继承FillHandler类。表示填充字段类型，如Date，String，BigDecimal，Boolean。

实战(springboot)

现在有个remark字段，需要在insert时初始化为“备注默认内容”，新建一个StringRemarkFill类如下：

```

public class StringRemarkFill extends FillHandler<String> {

    @Override
    public String getColumnName() {
        return "remark";
    }

    @Override
    public FillType getFillType() {
        return FillType.INSERT;
    }

    @Override
    protected Object getFillValue(String defaultValue) {
        return "备注默认内容";
    }

}

```

StringRemarkFill类中有三个重写方法：

- getColumnName()：指定表字段名
- getFillType()：填充方式，FillType.INSERT:仅insert时填充； FillType.UPDATE：insert，update时填充
- getFillValue(String defaultValue)：返回填充内容

然后在application.properties中添加

```
mybatis.fill.com.xx.StringRemarkFill=
```

这样就配置完毕了，调用dao.save(user);时会自动填充remark字段。

指定目标类

上面说到StringRemarkFill填充器，它作用在所有实体类上，也就是说实体类如果有remark字段都会自动填充。这样显然是不合理的，解决办法是指定特定的实体类。只要重写FillHandler类的getTargetEntityClasses()方法即可。

```

@Override
public Class<?>[] getTargetEntityClasses() {
    return new Class<?>[] { TUser.class };
}

```

```
}
```

这样就表示作用在TUser类上，多个类可以追加。最终代码如下：

```
public class StringRemarkFill extends FillHandler<String> {

    @Override
    public String getColumnName() {
        return "remark";
    }

    @Override
    public Class<?>[] getTargetEntityClasses() {
        return new Class<?>[] { TUser.class }; // 只作用在TUser类上
    }

    @Override
    public FillType getFillType() {
        return FillType.INSERT;
    }

    @Override
    protected Object getFillValue(String defaultValue) {
        return "备注默认内容"; // insert时填充的内容
    }

}
```

高级匹配

覆盖FillHandler类中的match方法可以让填充器做更高级的匹配，match方法如下

```
/**
 * 是否能够作用到指定字段
 * @param entityClass 实体类class
 * @param field 字段信息
 * @param columnName 给定的数据库字段名
 * @return
 */
public boolean match(Class<?> entityClass, Field field, String columnName)
```

这个方法返回的是一个boolean，返回true则代表作用到该属性上。例如下面的代码：

- 匹配出含有@Id和@GeneratedValue(strategy=GenerationType.AUTO)的字段

```
public boolean match(Class<?> entityClass, Field field, String columnName) {
    boolean isPk = field.getAnnotation(Id.class) != null; // 是否有@Id注解

    GeneratedValue gv = field.getAnnotation(GeneratedValue.class);
    boolean isAuto = gv != null && gv.strategy() == GenerationType.AUTO; // 是否有@GeneratedValue注解，并且策略是AUTO

    return isPk && isAuto;
}
```

关于自动填充的原理是基于mybatis的TypeHandler实现的，这里就不多做介绍了。感兴趣的同学可以查看FillHandler源码。

Entity中使用枚举字段

数据库中一些状态字段通常用0,1,2或者简单的字符串进行维护，然后JavaBean实体类中用枚举类型来保存，这样做便于使用和维护。

easymybatis上使用枚举属性很简单：枚举类实现net.oschina.durcframework.easymybatis.handler.BaseEnum接口即可。

下面是具体例子：

第一步

```
public enum UserInfoType implements BaseEnum<String> {
    INVALID("0"),VALID("1")
    ;

    private String status;

    UserInfoType(String type) {
        this.status = type;
    }

    @Override
    public String getCode() {
        return status;
    }
}
```

首先定义一个枚举类，实现BaseEnum接口，接口类型参数用String，表示保存的值是String类型，如果要保存Int类型的话改用BaseEnum。

第二步

在javaBean添加该枚举属性：

```
public class UserInfo {
    ...
    private UserInfoType status;

    // 省略getter setter
}
```

接下来就可以使用dao来进行数据操作了，下面是完整测试用例：

```
public class UserInfoDaoTest extends EasymybatisSpringbootApplicationTests {

    @Autowired
    UserInfoDao userInfoDao;

    @Test
    public void testGet() {
        UserInfo userInfo = userInfoDao.get(3);
        print("枚举字段status : " + userInfo.getStatus().getCode());
        print(userInfo);
    }

    @Test
    public void testUpdate() {
        UserInfo userInfo = userInfoDao.get(3);
        // 修改枚举值
        userInfo.setStatus(UserInfoType.INVALID);
        userInfoDao.update(userInfo);
    }

    @Test
    public void testSave() {
        UserInfo userInfo = new UserInfo();
        userInfo.setAddress("aa");
        userInfo.setCity("杭州");
        userInfo.setCreateTime(new Date());
        userInfo.setUserId(3);
        // 枚举值
        userInfo.setStatus(UserInfoType.VALID);
        userInfoDao.save(userInfo);
    }
}
```

SQL写在xml中

easymybatis提供的一些查询方式已经满足大部分的查询需求，但是有些复杂的sql语句还是需要写在xml文件中。easymybatis同样支持将sql语句写在xml中，具体配置如下：

- 在application.properties添加一句

```
mybatis.mapper-locations=classpath:/mybatis/mapper/*.xml
```

这句话用来指定xml文件的存放地。

- 在resources目录下新建一个mybatis文件夹，在mybatis文件夹下新建mapper文件夹。新建一个xml文件，名字跟Dao名字一致，如TUserDao.xml，建完后的文件路径是resources/mybatis/mapper/TUserDao.xml
- 在xml中添加sql语句，如下：

```
<?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">
<!-- 注意：文件名必须跟Dao类名字一致，因为是根据文件名做关联。 -->
<mapper>

  <select id="selectByName" parameterType="String" resultMap="baseResultMap">
    select * from t_user t where t.username = #{username} limit 1
  </select>

</mapper>
```

这个xml文件跟其它的mybatis配置文件一样，namespace可不写。这里baseResultMap没有看到定义，但是确实存在，因为这个是easymybatis提供的一个内置resultMap。

- 在TUserDao.java中添加：

```
TUser selectByName(@Param("username")String username);
```

- 编写单元测试用例

```
@Test
public void testSelectByName() {
    TUser user = dao.selectByName("张三");

    System.out.println(user.getUsername());
}
```

配置说明

easymybatis存放一些配置参数，这些参数都有默认值，一般情况下可以不用。如果使用springboot的话可以方便的对参数进行设置，只需在application.properties中设置即可，除此之外还可以设置其他mybatis官方参数，具体可参考[MyBatis-Spring-Boot-Application configuration parameters](#)

```
mybatis.camel2underline=true
mybatis.common-sql-classpath=
mybatis.mapper-save-dir=
mybatis.mapper-executor-pool-size=50
mybatis.template-classpath=
```

下面就讲解下各个属性的作用：

- camel2underline 如果为true，则开启驼峰转换下划线功能。实体类中的java字段映射成数据库字段将自动转成下划线形式。可以省略@Column注解。默认true。

- mapper-executor-pool-size mapper处理线程数，此项配置可以加快启动速度。默认值50，生产环境可以设置成较小的数，比如5
- template-classpath 指定模板文件class路径，开头结尾必须包含"/"。如果没有指定，则默认读取easymybatis/tpl/下的模板，一般情况下不做配置。
- common-sql-classpath 指定公共SQL模块class路径，如果没有指定，则默认读取easymybatis/commonSql.xml，一般情况下不做配置。
- mapper-save-dir 指定mapper文件存放路径。因为easymybatis是直接将mapper内容注入到内存当中，开发人员无感知，并且不知道mapper内容是什么样子，这个功能就是让开发人员能够查看到对应的mapper内容，方便定位和排查问题。一般情况下此项不用开启。

其他功能

指定外部模板

easymybatis依赖模板文件来生成mapper，默认的模板存放在easymybatis/tpl/下，模板文件名对应某一种数据库，如mysql.vm对应mysql数据库。

我们可以通过更改template-classpath的值来改变模板读取的位置。默认template-classpath的值为/easymybatis/tpl/。假如你想对mysql.vm做一些修改，那么可以按照如下步骤进行：

1. 使用解压工具解压easymybatis.jar
2. 在easymybatis/tpl/下找到mysql.vm，拷贝一份出来，放到你的项目中的classpath下（src/main/resources）
3. 在application.properties中添加一行

```
mybatis.template-classpath=/
```

这样在启动时会自动读取classpath根目录下的mysql.vm。控制台也会打印读取模板的信息：

```
2017-12-26 19:32:31.021 INFO 13476 --- [main] n.o.d.e.ext.MapperLocationsBuilder : 使用模板:/mysql.vm
```

如果你的项目是springmvc，采用xml配置形式，前两步不变，第三步改为：

```
<!-- 替换org.mybatis.spring.SqlSessionFactoryBean -->
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
  <property name="dataSource" ref="dataSource" />
  <property name="configLocation">
    <value>classpath:mybatis/mybatisConfig.xml</value>
  </property>
  <property name="mapperLocations">
    <list>
      <value>classpath:mybatis/mapper/*.xml</value>
    </list>
  </property>

  <!-- 以下是附加属性 -->

  <!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
  多个用;隔开
  -->
  <property name="basePackage" value="com.myapp.dao" />
  <property name="config">
    <bean class="net.oschina.durcframework.easymybatis.EasymybatisConfig">
      <!-- 指定外部模板 -->
      <property name="templateClasspath" value="/" />
    </bean>
  </property>
</bean>
```

全局Dao

easymybatis提供了全局Dao支持。

easymybatis设置全局Dao的方式如下：

1. 在resources下新建一个base.vm内容如下

```
<select id="getMySQLVersion" resultType="String">
  select version()
</select>
```

获取数据库版本号SQL.

1. 在application.properties新增一行

```
mybatis.global-vm-location=/base.vm
```

用来指定文件位置，支持file:，classpath:方式。

1. 新建一个全局Dao：GlobalDao.java

```
public interface GlobalDao {
    String getMySQLVersion();
}
```

1. 自定义的Dao继承GlobalDao.java

```
public interface AddressDao extends CrudDao<Address>,GlobalDao {

}
```

这样AddressDao就具备了getMySQLVersion()功能。base.vm中的文件编写可参考velocity变量说明

乐观锁

easymybatis提供的乐观锁使用方式跟JPA一样，使用@Version注解来实现。即：数据库增加一个int或long类型字段version，然后实体类version字段上加上@Version注解即可。实现原理是根据mysql的行锁机制(InnoDB下)，同一条记录只能被一条SQL执行，后面的SQL排队等待。这样version改变后，等待中的SQL还是老的version号，因此更新失败。

```
@Version
private Long version;
```

- 注：更新不成功不会抛出异常，而是update返回值为0

逻辑删除

从1.7版本开始支持逻辑删除功能,即更新一个字段标记为已删除。查询的时候会自动过滤掉已删除的数据。

假设数据库表中有一个字段is_deleted类型为tinyint，0表示未删除，1表示已删除。

实体类对应代码如下：

```
public class User {
    @LogicDelete
    private Byte isDeleted;
}
```

在执行dao.del(user);时会触发UPDATE语句，将is_deleted字段更新为1。

如果is_deleted类型为char(1)，f表示未删除，t表示已删除。

```
@LogicDelete(notDeleteValue = "f", deleteValue = "t")
private String isDeleted;
```

@LogicDelete提供两个属性

- notDeleteValue：指定未删除时的值,不指定默认为0
- deleteValue：指定删除后保存的值,不指定默认为1

假设1表示未删除，2表示已删除，@LogicDelete的设置方法如下：@LogicDelete(notDeleteValue = "1", deleteValue = "2")。如果每个实体类都要这样设置的话会很麻烦，easymybatis提供了全局配置

- springboot下，application.properties添加

```
# 未删除数据库保存的值，默认为0
mybatis.logic-not-delete-value=1
# 删除后数据库保存的值，默认为1
mybatis.logic-delete-value=2
```

- springmvc设置方式如下：

```
<!-- 替换org.mybatis.spring.SqlSessionFactoryBean -->
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
<property name="dataSource" ref="dataSource" />
<property name="configLocation">
<value>classpath:mybatis/mybatisConfig.xml</value>
</property>
<property name="mapperLocations">
<list>
<value>classpath:mybatis/mapper/*.xml</value>
</list>
</property>

<!-- 以下是附加属性 -->

<!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
多个用;隔开
-->
<property name="basePackage" value="com.myapp.dao" />
<property name="config">
<bean class="net.oschina.durcframework.easymybatis.EasymybatisConfig">
<property name="logicNotDeleteValue" value="1"/>
<property name="logicDeleteValue" value="2"/>
</bean>
</property>
</bean>
```

- 注：如果同时设置了@LogicDelete参数和全局配置，会优先读取注解中的配置。

多数据源配置

这里主要介绍在springboot下进行多数据源配置：

首先application.properties配置：

```
# 主数据源
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://${mysql.ip}:3306/stu?useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull
spring.datasource.username=${mysql.username}
spring.datasource.password=${mysql.password}

# 第二个数据源
spring.datasourceSecond.driver-class-name=com.mysql.jdbc.Driver
spring.datasourceSecond.url=jdbc:mysql://${mysql.ip}:3306/easydoc?useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=conv
spring.datasourceSecond.username=${mysql.username}
spring.datasourceSecond.password=${mysql.password}
```

- 禁掉springboot自带的DataSourceAutoConfiguration

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
```

- pom移除easymybatis-spring-boot-starter依赖
- pom添加:

```
<dependency>
<groupId>net.oschina.durcframework</groupId>
<artifactId>easymybatis</artifactId>
<version>1.8.4</version>
</dependency>
```

- 新建一个数据库配置文件，作为主数据源

```
/**
 * 主数据源,使用方式:
 * <pre>
 * 1. pom移除easymybatis-spring-boot-starter依赖
 * 2. pom添加:
 * <code>
 * <dependency>
 *     <groupId>net.oschina.durcframework</groupId>
 *     <artifactId>easymybatis</artifactId>
 *     <version>1.8.4</version>
 * </dependency>
 * 3. 打开注释:
 * @Configuration
 * @MapperScan(basePackages = { DbMasterConfig.basePackage }, sqlSessionRef = DbMasterConfig.sqlSessionFactoryName)
 * </code>
 * </pre>
 * @author tanghc
 */
@Configuration
@MapperScan(basePackages = { DbMasterConfig.basePackage }, sqlSessionRef = DbMasterConfig.sqlSessionFactoryName)
public class DbMasterConfig {

    /* *****只需要改这里的配置***** */
    static final String dbName = "master";
    /** 配置文件前缀 */
    public static final String prefix = "spring.datasource";
    /** 存放mapper包路径 */
    public static final String basePackage = "com.myapp.dao";
    /** mybatis的config文件路径 */
    public static final String mybatisConfigLocation = "classpath:mybatis/mybatisConfig.xml";
    /** mybatis的mapper文件路径 */
    public static final String mybatisMapperLocations = "classpath:mybatis/mapper/*.xml";
    /** 表新增时间字段名 */
    public static final String dbInsertDateColumnName = "gmt_create";
    /** 表更新时间字段名 */
    public static final String dbUpdateDateColumnName = "gmt_update";
    /* ***** */

    /** 数据源名称 */
    public static final String dataSourceName = "dataSource" + dbName;
    /** sqlSessionTemplate名称 */
    public static final String sqlSessionTemplateName = "sqlSessionTemplate" + dbName;
    /** sqlSessionFactory名称 */
    public static final String sqlSessionFactoryName = "sqlSessionFactory" + dbName;
    /** transactionManager名称 */
    public static final String transactionManagerName = "transactionManager" + dbName;
    /** transactionTemplate名称 */
    public static final String transactionTemplateName = "transactionTemplate" + dbName;

    @Bean(name = dataSourceName)
    @Primary
    @ConfigurationProperties(prefix = prefix) // application.properties中对应属性的前缀
    public DataSource dataSource() {
        return DruidDataSourceBuilder.create().build();
    }
}
```

```

@Bean
public EasymybatisConfig easymybatisConfig() {
    EasymybatisConfig config = new EasymybatisConfig();
    /*
     * 驼峰转下划线形式，默认是true 开启后java字段映射成数据库字段将自动转成下划线形式 如：userAge -> user_age
     * 如果数据库设计完全遵循下划线形式，可以启用 这样可以省略Entity中的注解，@Table，@Column都可以不用，只留
     *
     * @Id
     *
     * @GeneratedValue 参见：UserInfo.java
     */
    config.setCamel2underline(true);
    config.setFills(Arrays.asList(new DateFillInsert(dbInsertDateColumnName),
        new DateFillUpdate(dbUpdateDateColumnName)));

    return config;
}

@Bean(name = sqlSessionFactoryName)
public SqlSessionFactory sqlSessionFactory(@Autowired @Qualifier(dataSourceName) DataSource dataSource,
    EasymybatisConfig config) throws Exception {
    Assert.notNull(dataSource, "dataSource can not be null.");
    Assert.notNull(config, "EasymybatisConfig can not be null.");

    SqlSessionFactoryBeanExt bean = new SqlSessionFactoryBeanExt();

    bean.setDataSource(dataSource);
    bean.setConfigLocation(this.getResource(mybatisConfigLocation));
    bean.setMapperLocations(this.getResources(mybatisMapperLocations));

    // ====以下是附加属性====

    // dao所在的包名,跟MapperScannerConfigurer的basePackage一致,多个用;隔开
    bean.setBasePackage(basePackage);
    bean.setConfig(config);

    return bean.getObject();
}

@Bean(name = sqlSessionTemplateName)
public SqlSessionTemplate sqlSessionTemplate(
    @Autowired @Qualifier(sqlSessionFactoryName) SqlSessionFactory sessionFactory) throws Exception {
    SqlSessionTemplate template = new SqlSessionTemplate(sessionFactory); // 使用上面配置的Factory
    return template;
}

@Bean(name = transactionManagerName)
public PlatformTransactionManager annotationDrivenTransactionManager(
    @Autowired @Qualifier(dataSourceName) DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}

@Bean(name = transactionTemplateName)
public TransactionTemplate transactionTemplate(@Autowired @Qualifier(transactionManagerName) PlatformTransactionManager transactionManager) {
    return new TransactionTemplate(transactionManager);
}

private Resource[] getResources(String path) throws IOException {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    return resolver.getResources(path);
}

private Resource getResource(String path) {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    return resolver.getResource(path);
}
}

```

这个文件不用做多大的修改，只需要修改上面的配置内容即可。

- 新建第二个文件，作为第二个数据源：

```
/**
 * 第二个数据源，后续有第三个数据，复制这个文件，然后改下配置即可
 * @author tanghc
 */
@Configuration
MapperScan(basePackages = { DbSecondConfig.basePackage }, sqlSessionSessionFactoryRef = DbSecondConfig.sqlSessionFactoryName)
public class DbSecondConfig {

    /** *****只需要改这里的配置***** */
    static final String dbName = "Second";
    /** 配置文件前缀 */
    public static final String prefix = "spring.datasourceSecond";
    /** 存放mapper包路径 */
    public static final String basePackage = "com.app2.dao";
    /** mybatis的配置文件路径 */
    public static final String mybatisConfigLocation = "classpath:mybatis/mybatisConfig.xml";
    /** mybatis的mapper文件路径 */
    public static final String mybatisMapperLocations = "classpath:mybatis/mapper2/*.xml";
    /** 表新增时间字段名 */
    public static final String dbInsertDateColumnName = "gmt_create";
    /** 表更新时间字段名 */
    public static final String dbUpdateDateColumnName = "gmt_update";
    /** ***** */

    /** 数据源名称 */
    public static final String dataSourceName = "dataSource" + dbName;
    /** sqlSessionTemplate名称 */
    public static final String sqlSessionTemplateName = "sqlSessionTemplate" + dbName;
    /** sqlSessionSessionFactory名称 */
    public static final String sqlSessionSessionFactoryName = "sqlSessionFactory" + dbName;
    /** transactionManager名称 */
    public static final String transactionManagerName = "transactionManager" + dbName;
    /** transactionTemplate名称 */
    public static final String transactionTemplateName = "transactionTemplate" + dbName;

    @Bean(name = dataSourceName)
    @ConfigurationProperties(prefix = prefix) // application.properteis中对应属性的前缀
    public DataSource dataSourceMater() {
        return DruidDataSourceBuilder.create().build();
    }

    @Bean
    public EasymybatisConfig easymybatisConfig() {
        EasymybatisConfig config = new EasymybatisConfig();
        /**
         * 驼峰转下划线形式，默认是true 开启后java字段映射成数据库字段将自动转成下划线形式 如：userAge -> user_age
         * 如果数据库设计完全遵循下划线形式，可以启用 这样可以省略Entity中的注解，@Table，@Column都可以不用，只留
         *
         * @Id
         *
         * @GeneratedValue 参见：UserInfo.java
         */
        config.setCamel2underline(true);
        config.setFills(Arrays.asList(new DateFillInsert(dbInsertDateColumnName),
            new DateFillUpdate(dbUpdateDateColumnName)));

        return config;
    }

    @Bean(name = sqlSessionSessionFactoryName)
    public SqlSessionFactory sqlSessionSessionFactory(@Autowired @Qualifier(dataSourceName) DataSource dataSource,
        EasymybatisConfig config) throws Exception {
        Assert.notNull(dataSource, "dataSource can not be null.");
        Assert.notNull(config, "EasymybatisConfig can not be null.");
    }
}
```

```

    SqlSessionFactoryBeanExt bean = new SqlSessionFactoryBeanExt();

    bean.setDataSource(dataSource);
    bean.setConfigLocation(this.getResource(mybatisConfigLocation));
    bean.setMapperLocations(this.getResources(mybatisMapperLocations));

    // ====以下是附加属性====

    // dao所在的包名,跟MapperScannerConfigurer的basePackage一致,多个用;隔开
    bean.setBasePackage(basePackage);
    bean.setConfig(config);

    return bean.getObject();
}

@Bean(name = sqlSessionTemplateName)
public SqlSessionTemplate sqlSessionTemplate(
    @Autowired @Qualifier(sqlSessionFactoryName) SqlSessionFactory sessionFactory) throws Exception {
    SqlSessionTemplate template = new SqlSessionTemplate(sessionFactory); // 使用上面配置的Factory
    return template;
}

@Bean(name = transactionManagerName)
public PlatformTransactionManager annotationDrivenTransactionManager(
    @Autowired @Qualifier(dataSourceName) DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}

@Bean(name = transactionTemplateName)
public TransactionTemplate transactionTemplate(@Autowired @Qualifier(transactionManagerName) PlatformTransactionManager transactionManager) {
    return new TransactionTemplate(transactionManager);
}

private Resource[] getResources(String path) throws IOException {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    return resolver.getResources(path);
}

private Resource getResource(String path) {
    ResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();
    return resolver.getResource(path);
}
}

```

第二个数据源，后续有第三个数据，复制这个文件，然后改下配置即可

[附录]velocity变量说明

#{context}

- #{context.dbName} : 数据库名称
- #{context.packageName} : 包名
- #{context.javaBeanName} : Java类名
- #{context.javaBeanNameLF} : Java类名且首字母小写

#{pk}

- #{pk.isIdentity} : 是否自增
- #{pk.javaFieldName} : 主键java字段名
- #{pk.columnName} : 主键数据库字段名
- #{pk.isUuid} : 主键是否使用UUID策略

`${table}`

- `${table.tableName}` : 数据库表名
- `${table.comment}` : 表注释
- `${table.javaBeanNameLF}` : java类名, 并且首字母小写

`#foreach($column in $columns)...#end`

- `${column.columnName}` : 表中字段名
- `${column.type}` : java字段类型, String, Integer
- `${column.fullType}` : java字段完整类型, java.lang.String
- `${column.javaFieldName}` : java字段名
- `${column.javaFieldNameUF}` : java字段名首字母大写
- `${column.javaType}` : 字段的java类型
- `${column.javaTypeBox}` : 字段的java装箱类型,如Integer,Long
- `${column.isIdentity}` : 是否自增,返回boolean
- `${column.isPk}` : 是否自增主键,返回boolean
- `${column.isIdentityPk}` : 是否自增主键,返回boolean
- `${column.isEnum}` : 是否枚举类型,返回boolean
- `${column.mybatisJdbcType}` : 返回mybatis定义的jdbcType
- `${column.comment}` : 表字段注释

三分钟深入了解easymybatis

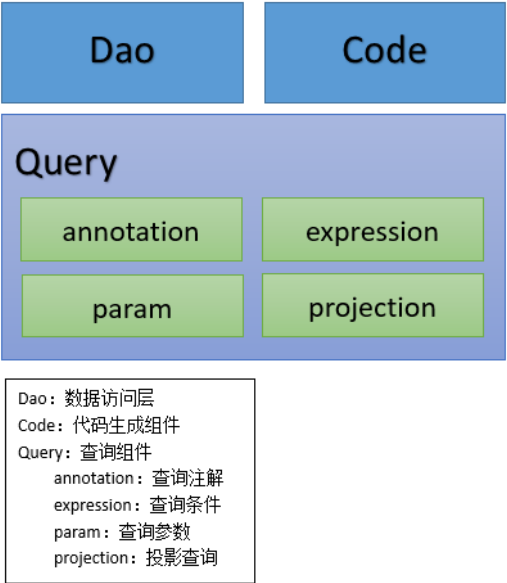
简单介绍

easymybatis是一个mybatis增强类库，目的为简化mybatis的开发，让开发更高效。easymybatis的特性如下：

- 无需编写xml文件即可完成CRUD操作。
- 支持多表查询、聚合查询、分页查询（支持多种数据库）。
- 支持批量添加，指定字段批量添加。
- 支持Dao层访问控制，如某个dao只有查询功能，某个dao有crud功能等。
- 支持自定义sql，sql语句可以写在配置文件中，同样支持mybatis标签。
- 支持mysql，sqlserver，oracle，其它数据库扩展方便（增加一个模板文件即可）。
- 使用方式不变，与Spring集成只改了一处配置。
- 轻量级，无侵入性，可与传统mybatis用法共存。
- 没有修改框架源码(无插件)，采用动态代码生成实现功能。

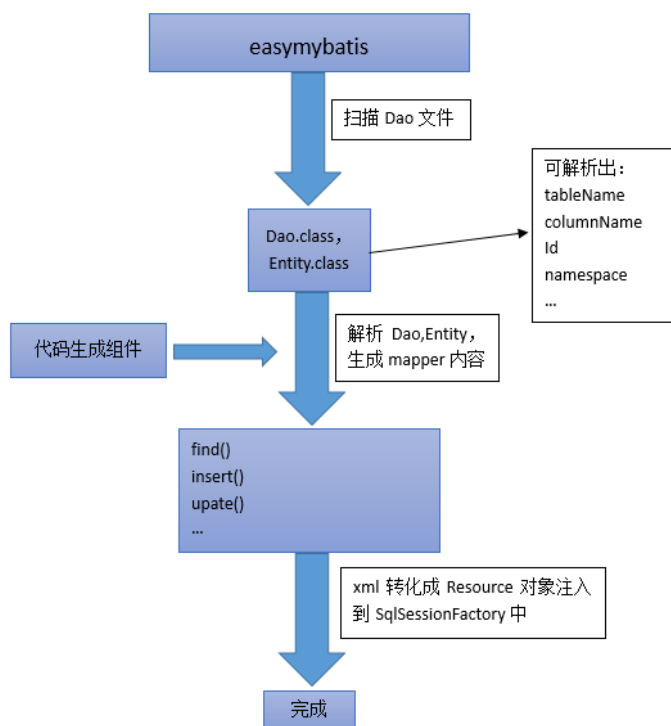
架构组成

easymybatis的架构如下：



运行流程

easymybatis的运行流程图：



1. 服务器启动的时候easymybatis负责扫描Dao.java。
2. 扫描完成后解析出Dao.class以及实体类Entity.class。
3. 代码生成组件根据Dao.class和Entity.class生成mapper文件内容，生成方式由velocity模板指定。
4. 把mapper文件内容转化成Resource对象设置到SqlSessionFactory中。

快速上手

- 一. 第一步自行搭建一个mybatis的项目，并且使用spring-mybatis插件
- 二. pom.xml加入easymybatis依赖

```
<dependency>
<groupId>net.oschina.durcframework</groupId>
<artifactId>easymybatis</artifactId>
<version>最新版本</version>
</dependency>
```

- 三. 替换org.mybatis.spring.SqlSessionFactoryBean

```
<!-- 替换SqlSessionFactoryBean -->
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
<property name="dataSource" ref="dataSource" />
<property name="configLocation">
<value>classpath:mybatis/mybatisConfig.xml</value>
</property>
<property name="mapperLocations">
<list>
<value>classpath:mybatis/mapper/*.xml</value>
</list>
</property>

<!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
多个用;隔开
-->
<property name="basePackage" value="com.myapp.dao" />

</bean>
```

- 四. 新建一个实体类，在实体类中加入JPA注解，因为需要通过注解来解析出数据库的一些信息，注解会在流程3中用到

```
@Table(name = "t_user")
public class TUser {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "username")
    private String username;

    // 省略get set
}
```

在实际项目中手动写实体类是不现实的，需要配合代码生成工具，easymybatis对应的代码生成工具链接：[代码生成器](#)。其实这个代码生成工具不是必须的，如果您有一个工具能生成hibernate实体类的话也是可行的。

- 五. 最后新建一个Dao继承CrudDao，表示这个dao具有CRUD功能

```
public interface TUserDao extends CrudDao<TUser> {
}
```

接下来就可进行编码测试了

```
@Resource
TUserDao dao;

@Test
public void testGet() {
    TUser user = dao.get(3);
    print(user);
}
```

更多例子可参考[TUserDaoTest.java](#)。重点关注Dao，Query对象即可。

如果您不想从头开始搭项目的话，这里有个搭建好的demo项目可以为您使用。

- [springboot](#)
- [springmvc](#)

两个版本，一个是基于springboot，一个是传统springmvc，推荐springboot。

意见交流

如果您在使用的过程中遇到问题的话可以加入QQ群328419269进行提问。

完整测试用例

```
```java
/**
 * dao测试
 */
public class TUserDaoTest extends EasymybatisSpringbootApplicationTests {

 @Resource
 TUserDao dao;
 @Resource
 TransactionTemplate transactionTemplate;

 // 根据主键查询
 @Test
 public void testGet() {
 TUser user = dao.get(3);
 }
}
```

```

 print(user);
}

// 根据字段查询一条记录
@Test
public void testGetProperty() {
 TUser user = dao.getByProperty("username", "王五");
 print(user);
}

// 根据条件查询一条记录
@Test
public void testGetByExpression() {
 // 查询ID=3的用户
 Query query = Query.build().eq("id", 3);

 TUser user = dao.getByExpression(query);

 print(user);
}

// 条件查询
// 查询username='张三'的用户
@Test
public void testFind() {
 Query query = new Query();
 // 添加查询条件
 query.eq("username", "张三");

 List<TUser> list = dao.find(query); // 获取结果集
 long count = dao.countTotal(query); // 获取总数
 print("count:" + count);
 for (TUser user : list) {
 System.out.println(user);
 }
}

// 分页查询
/*MySQL语句:
 *
 * SELECT t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`
 * FROM `t_user` t
 * ORDER BY id DESC
 * LIMIT ?,?
 *
 * Parameters: 2(Integer), 2(Integer)
 */
@Test
public void testPage() {
 Query query = new Query();

 query.setPage(2, 2) // 设置pageIndex, pageSize
 .addSort("id", Sort.DESC); // 添加排序

 ;

 // 查询后的结果, 包含总记录数, 结果集, 总页数等信息
 PageInfo<TUser> pageInfo = QueryUtils.query(dao, query);

 List<TUser> rows = pageInfo.getList();
 for (TUser user : rows) {
 System.out.println(user);
 }
}

// 自定义返回字段查询, 只返回两个字段
// SELECT t.id,t.username FROM `t_user` t LIMIT 0,10
@Test
public void testSelfColumns() {
 Query query = new Query();

```

```
// 只返回id,username
query.setColumns(Arrays.asList("t.id","t.username"));

List<TUser> list = dao.find(query);

for (TUser user : list) {
 System.out.println(user);
}
}

// 多表查询,left join
// 适用场景：获取两张表里面的字段信息返回给前端
/*
 * MYSQL生成如下sql:
SELECT
t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`
, t2.city, t2.address
FROM `t_user` t left join user_info t2 on t.id = t2.user_id
WHERE t2.id = ?
ORDER BY id ASC
LIMIT ?,?
*/
@Test
public void testJoin() {
 Query query = new Query();
 // 添加第二张表的字段,跟主表字段一起返回
 query.addOtherColumns(
 "t2.city"
 , "t2.address"
);
 // 左连接查询,主表的alias默认为t
 query.join("left join user_info t2 on t.id = t2.user_id");
 // 添加条件
 query.eq("t2.id", 2);

 query.addSort("t.id");

 List<TUser> list = dao.find(query);

 System.out.println("=====");
 for (TUser user : list) {
 System.out.println(
 user.getId()
 + " " + user.getUsername()
 // 下面两个字段是第二张表里面的
 + " " + user.getCity()
 + " " + user.getAddress()
);
 }
 System.out.println("=====");
}

/*
 * 聚合查询
 * 按state分组统计money和,并且按照state升序
 * 并且money大于200的state
 *
 * SELECT
 * state AS s , SUM(money) AS m
 * FROM `t_user` t
 * WHERE money > 0
 * GROUP BY state
 * HAVING m > 200
 * ORDER BY state asc
 */
@Test
public void testProjection() {

 ProjectionQuery query = new ProjectionQuery();
 // 添加列
 query.addProjection(Projections.column("state", "s"));
}
```

```

query.addProjection(Projections.sum("money", "m"));
// 添加where
query.addExpression(new ValueExpression("money", ">", 0));
// 添加group by
query.addGroupBy("state");
// 添加having
query.addHaving(new ValueExpression("SUM(money)", ">", 200));

query.addSort("state", Sort.DESC);

List<Map<String, Object>> list = dao.findProjection(query);

Assert.notEmpty(list);

print(list);
}

// 自定义sql, 见TUserDao.xml
// 注意mybatis的mapper必须跟Dao类一致
@Test
public void testSelfSql() {
 TUser user = dao.selectByName("张三");

 print(user);
}

// 添加-保存所有字段
@Test
public void testInsert() {
 TUser user = new TUser();
 user.setAddTime(new Date());
 user.setIsdel(false);
 user.setLeftMoney(22.1F);
 user.setMoney(new BigDecimal(100.5));
 user.setRemark("备注");
 user.setState((byte)0);
 user.setUsername("张三");

 dao.save(user);

 print("添加后的主键:" + user.getId());
 print(user);
}

// 添加-保存非空字段
@Test
public void testInsertIgnoreNull() {
 TUser user = new TUser();
 user.setAddTime(new Date());
 user.setIsdel(true);
 user.setMoney(new BigDecimal(100.5));
 user.setState((byte)0);
 user.setUsername("张三notnull");
 user.setLeftMoney(null);
 user.setRemark(null);

 dao.saveIgnoreNull(user);

 print("添加后的主键:" + user.getId());
 print(user);
}

// 批量添加
/*
 * 支持mysql,sqlserver2008。如需支持其它数据库使用saveMulti方法
 * INSERT INTO person (id, name, age)
VALUES
 (1, 'Kelvin', 22),
 (2, 'ini_always', 23);
 */
@Test

```

```

public void testInsertBatch() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("username" + i);
 user.setMoney(new BigDecimal(i));
 user.setRemark("remark" + i);
 user.setState((byte)0);
 user.setIsdel(false);
 user.setAddTime(new Date());
 user.setLeftMoney(200F);
 users.add(user);
 }

 int i = dao.saveBatch(users); // 返回成功数

 System.out.println("saveBatch --> " + i);
}

/**
 * 批量添加,兼容更多数据库版本,采用union all
 * INSERT INTO [t_user] ([username] , [state] , [isdel] , [remark] , [add_time] , [money] , [left_money])
 * SELECT ? , ? , ? , ? , ? , ? , ?
 * UNION ALL SELECT ? , ? , ? , ? , ? , ? , ?
 * UNION ALL SELECT ? , ? , ? , ? , ? , ? , ?
 */
@Test
public void testInsertMulti() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("username" + i);
 user.setMoney(new BigDecimal(i));
 user.setRemark("remark" + i);
 user.setState((byte)3);
 user.setIsdel(false);
 user.setAddTime(new Date());
 user.setLeftMoney(200F);
 users.add(user);
 }

 int i = dao.saveMulti(users); // 返回成功数

 System.out.println("saveMulti --> " + i);
}

// 批量添加指定字段,仅支持mysql,sqlserver2008,如需支持其它数据库使用saveMulti方法
@Test
public void testInsertBatchWithColumns() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("usernameWithColumns" + i);
 user.setMoney(new BigDecimal(i));
 user.setAddTime(new Date());

 users.add(user);
 }

 /**
 * INSERT INTO `t_user` (username , money , add_time)
 * VALUES (? , ? , ?) , (? , ? , ?) , (? , ? , ?)
 */
 int i = dao.saveBatchWithColumns(Arrays.asList(
 new Column("username", "username") // 第一个是数据库字段,第二个是java字段
 ,new Column("money", "money")
 ,new Column("add_time", "addTime")
), users);
}

```

```

System.out.println("saveBatchWithColumns --> " + i);

}

/*
 * // 批量添加指定字段,兼容
 * INSERT INTO [t_user] ([username] , [money] , [add_time])
 * SELECT ? , ? , ?
 * UNION ALL SELECT ? , ? , ?
 * UNION ALL SELECT ? , ? , ?
 */
@Test
public void testInsertMultiWithColumns() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("usernameWithColumns" + i);
 user.setMoney(new BigDecimal(i));
 user.setAddTime(new Date());

 users.add(user);
 }

 int i= dao.saveMultiWithColumns(Arrays.asList(
 new Column("username", "username") // 第一个是数据库字段,第二个是java字段
 ,new Column("money", "money")
 ,new Column("add_time", "addTime")
), users);

 System.out.println("saveMultiWithColumns --> " + i);

}

// 事务回滚
@Test
public void testUpdateTran() {
 TUser user = transactionTemplate.execute(new TransactionCallback<TUser>() {
 @Override
 public TUser doInTransaction(TransactionStatus arg0) {
 try{
 TUser user = dao.get(3);
 user.setUsername("王五1");
 user.setMoney(user.getMoney().add(new BigDecimal(0.1)));
 user.setIsdel(true);

 int i = dao.update(user);
 print("testUpdate --> " + i);
 int j = 1/0; // 模拟错误
 return user;
 }catch(Exception e) {
 e.printStackTrace();
 arg0.setRollbackOnly();
 return null;
 }
 }
 });

 print(user);

}

// 更新所有字段
@Test
public void testUpdate() {
 TUser user = dao.get(3);
 user.setUsername("李四");
 user.setMoney(user.getMoney().add(new BigDecimal(0.1)));
 user.setIsdel(true);

```

```

int i = dao.update(user);
print("testUpdate --> " + i);
}

// 更新不为null的字段
/*
 *UPDATE [t_user] SET [username]=?, [isdel]=? WHERE [id] = ?
 */
@Test
public void updateIgnoreNull() {
 TUser user = new TUser();
 user.setId(3);
 user.setUsername("王五");
 user.setIsdel(false);
 int i = dao.updateIgnoreNull(user);
 print("updateNotNull --> " + i);
}

// 根据条件更新
// UPDATE t_user SET remark = '批量修改备注' WHERE state = 0
@Test
public void testUpdateNotNullByExpression() {
 Query query = new Query();
 query.eq("state", 0);

 TUser user = new TUser();
 user.setRemark("批量修改备注");

 int i = dao.updateIgnoreNullByExpression(user, query);
 print("testUpdateNotNullByExpression --> " + i);
}

// 删除
@Test
public void testDel() {
 TUser user = new TUser();
 user.setId(14);
 int i = dao.del(user);
 print("del --> " + i);
}

// 根据条件删除
// DELETE FROM `t_user` WHERE state = ?
@Test
public void delByExpression() {
 Query query = new Query();
 query.eq("state", 3);
 int i = dao.delByExpression(query);
 print("delByExpression --> " + i);
}
}

```

# 使用springboot项目快速搭建

## 创建springboot项目

访问<http://start.spring.io/> 生成一个springboot空项目，输入Group，Artifact点生成即可，如图：

Generate a 

Maven Project

 with 

Java

 and Spring Boot 

1.5.9

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Generate Project alt + ↵

点击Generate Project，下载demo.zip

## 导入项目

将下载的demo.zip解压，然后导入项目。eclipse中右键 -> Import... -> Existing Maven Project，选择项目文件夹。导入到eclipse中后等待maven相关jar包下载。

## 添加maven依赖

jar包下载完成后，打开pom.xml，添加如下依赖：

```
<!-- easymybatis的starter -->
<dependency>
 <groupId>net.oschina.durcframework</groupId>
 <artifactId>easymybatis-spring-boot-starter</artifactId>
 <version>1.4.5</version>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <scope>runtime</scope>
</dependency>
```

## 添加数据库配置

在application.properties中添加数据库配置

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/stu?useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull
spring.datasource.username=root
spring.datasource.password=root
```

## 添加Java文件

假设数据库中有张t\_user表，DDL如下：

```
CREATE TABLE `t_user` (
40 / 73
```

```
`id` int(11) NOT NULL AUTO_INCREMENT COMMENT 'ID',
`username` varchar(255) DEFAULT NULL COMMENT '用户名',
`state` tinyint(4) DEFAULT NULL COMMENT '状态',
`isdel` bit(1) DEFAULT NULL COMMENT '是否删除',
`remark` text COMMENT '备注',
`add_time` datetime DEFAULT NULL COMMENT '添加时间',
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='用户表';
```

我们加入对应的实体类和Dao:

- TUser.java :

```
@Table(name = "t_user")
public class TUser {
 @Id
 @Column(name="id")
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private Integer id; // ID
 private String username; // 用户名
 private Byte state; // 状态
 private Boolean isdel; // 是否删除
 private String remark; // 备注
 private Date addTime; // 添加时间

 // 省略 getter setter
}
```

实体类文件采用和hibernate相同的方式，您可以使用我们的代码生成工具生成 <https://gitee.com/durcframework/easymybatis-generator>

- TUserDao.java :

```
public interface TUserDao extends CrudDao<TUser> {
}
```

TUserDao继承CrudDao即可，这样这个Dao就拥有了CRUD功能。

## 添加测试用例

```
public class TUserDaoTest extends DemoApplicationTests {

 @Autowired
 TUserDao dao;

 @Test
 public void testInsert() {
 TUser user = new TUser();
 user.setIsdel(false);
 user.setRemark("testInsert");
 user.setUsername("张三");

 dao.save(user);

 System.out.println("添加后的主键:" + user.getId());
 }

 @Test
 public void testGet() {
 TUser user = dao.get(3);
 System.out.println(user);
 }

 @Test
 public void testUpdate() {
 TUser user = dao.get(3);
```

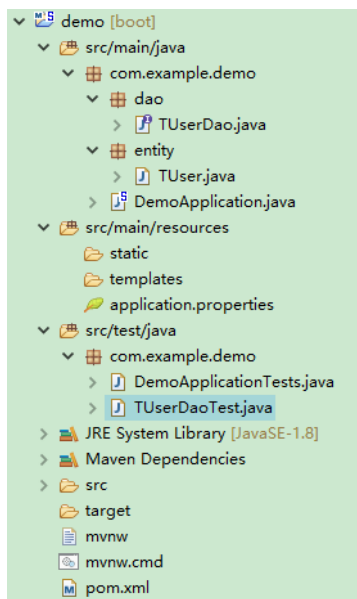
```
user.setUsername("李四");
user.setIsdel(true);

int i = dao.update(user);
System.out.println("testUpdate --> " + i);
}

@Test
public void testDel() {
 TUser user = new TUser();
 user.setId(3);
 int i = dao.del(user);
 System.out.println("del --> " + i);
}
}
```

然后运行单元测试，运行成功后表示项目已经搭建完毕了。

最后项目结构图：



## 查询

### 分页查询

#### 方式1

```
// http://localhost:8080/page1?pageIndex=1&pageSize=10
@GetMapping("page1")
public List<TUser> page1(int pageIndex,int pageSize) {
 Query query = new Query();
 query.setPage(pageIndex, pageSize);
 List<TUser> list = dao.find(query);
 return list;
}
```

#### 方式2

```
// http://localhost:8080/page2?pageIndex=1&pageSize=10
@GetMapping("page2")
public List<TUser> page2(PageParam param) {
 Query query = param.toQuery();
 List<TUser> list = dao.find(query);
 return list;
}
```

PageParam里面封装了pageIndex，pageSize参数

### 返回结果集和总记录数

```
// http://localhost:8080/page3?pageIndex=1&pageSize=10
@GetMapping("page3")
public Map<String,Object> page3(PageParam param) {
 Query query = param.toQuery();
 List<TUser> list = dao.find(query);
 long total = dao.countTotal(query);

 Map<String,Object> result = new HashMap<String, Object>();
 result.put("list", list);
 result.put("total", total);

 return result;
}
```

easymybatis提供一种更简洁的方式来处理：

```
// http://localhost:8080/page4?pageIndex=1&pageSize=10
@GetMapping("page4")
public PageInfo<TUser> page4(PageParam param) {
 PageInfo<TUser> result = QueryUtils.query(dao, param);
 return result;
}
```

PageInfo里面包含了List，total信息，还包含了一些额外信息，完整数据如下：

```
{
 "currentPageIndex": 1, // 当前页
 "firstPageIndex": 1, // 首页
 "lastPageIndex": 2, // 尾页
 "list": [// 结果集
 {},
 {}
]
}
```

```

 },
 "nextPageIndex": 2, // 下一页
 "pageCount": 2, // 总页数
 "pageIndex": 1, // 当前页
 "pageSize": 10, // 每页记录数
 "prePageIndex": 1, // 上一页
 "start": 0,
 "total": 20 // 总记录数
}

```

## 完整代码

```

@RestController
public class UserSchController {

 @Autowired
 private TUserDao dao;

 // http://localhost:8080/page1?pageIndex=1&pageSize=10
 @GetMapping("page1")
 public List<TUser> page1(int pageIndex,int pageSize) {
 Query query = new Query();
 query.setPage(pageIndex, pageSize);
 List<TUser> list = dao.find(query);
 return list;
 }

 // http://localhost:8080/page2?pageIndex=1&pageSize=10
 @GetMapping("page2")
 public List<TUser> page2(PageParam param) {
 Query query = param.toQuery();
 List<TUser> list = dao.find(query);
 return list;
 }

 // http://localhost:8080/page3?pageIndex=1&pageSize=10
 @GetMapping("page3")
 public Map<String,Object> page3(PageParam param) {
 Query query = param.toQuery();
 List<TUser> list = dao.find(query);
 long total = dao.countTotal(query);

 Map<String,Object> result = new HashMap<String, Object>();
 result.put("list", list);
 result.put("total", total);

 return result;
 }

 // http://localhost:8080/page4?pageIndex=1&pageSize=10
 @GetMapping("page4")
 public PageInfo<TUser> page4(PageParam param) {
 PageInfo<TUser> result = QueryUtils.query(dao, param);
 return result;
 }
}

```

更多查询方式可参阅：[Query类详解](#)

## 参数查询

### 根据字段查询

查询姓名为张三的用户

```
// http://localhost:8080/sch?username=张三
@GetMapping("sch")
public List<TUser> sch(String username) {
 Query query = new Query();
 query.eq("username", username);
 List<TUser> list = dao.find(query);
 return list;
}
```

查询姓名为张三并且拥有的钱大于100块

```
// http://localhost:8080/sch2?username=张三
@GetMapping("sch2")
public List<TUser> sch2(String username) {
 Query query = new Query();
 query.eq("username", username).gt("money", 100);
 List<TUser> list = dao.find(query);
 return list;
}
```

查询姓名为张三并带分页

```
// http://localhost:8080/sch3?username=张三&pageIndex=1&pageSize=5
@GetMapping("sch3")
public List<TUser> sch3(String username, PageParam param) {
 Query query = param.toQuery();
 query.eq("username", username);
 List<TUser> list = dao.find(query);
 return list;
}
```

查询钱最多的前三名

```
// http://localhost:8080/sch4
@GetMapping("sch4")
public List<TUser> sch4() {
 Query query = new Query();
 query.addSort("money", Sort.DESC) // 按金额降序
 .setPage(1, 3);
 List<TUser> list = dao.find(query);
 return list;
}
```

将参数放在对象中查询

```
// http://localhost:8080/sch5?username=张三
@GetMapping("sch5")
public List<TUser> sch5(UserParam userParam) {
 Query query = userParam.toQuery();
 query.eq("username", userParam.getUsername());
 List<TUser> list = dao.find(query);
 return list;
}
```

UserParam继承PageSortParam类，表示支持分页和排序查询

## 使用@ValueField注解查询

```
// http://localhost:8080/sch6?username=张三
@GetMapping("sch6")
public List<TUser> sch6(UserParam userParam) {
 Query query = userParam.toQuery();
 List<TUser> list = dao.find(query);
 return list;
}

public static class UserParam extends PageSortParam {
 private String username;
 // 这里定义了注解，当执行userParam.toQuery()方法的时候会自动把username值添加到条件当中。
 // 即执行了query.eq("username",username);操作
 @ValueField(column="username")
 public String getUsername() {
 return username;
 }

 public void setUsername(String username) {
 this.username = username;
 }
}
```

使用注解的好处是，只要在UserParam类中添加查询参数即可，Controller中的代码不用修改。

更多查询方式可参阅：[Query类详解](#)

## 多表关联查询

多表关联查询使用的地方很多，比如需要关联第二张表，获取第二张表的几个字段，然后返回给前端。

easymybatis的用法如下：假如我们需要关联第二张表，并且获取第二张表里的city，address字段。步骤如下：

- 在实体类中添加city，address字段，并标记@Transient注解。只要不是主表中的字段都要加上@Transient

```
@Transient
private String city;
@Transient
private String address;

// getter setter
```

- 接下来是查询代码：

```
Query query = new Query();
// 添加第二张表的字段,跟主表字段一起返回
query.addOtherColumns(
 "t2.city"
 ,"t2.address"
);
// 左连接查询,主表的alias默认为t
query.join("LEFT JOIN user_info t2 ON t.id = t2.user_id");
// 添加查询条件
query.eq("t.username", "张三");

List<TUser> list = dao.find(query);
```

得到的SQL语句：

```
SELECT
 t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`,
 t2.city, t2.address
FROM `t_user` t LEFT JOIN user_info t2 ON t.id = t2.user_id
WHERE t.username = ?
LIMIT ?,?
```

关联了user\_info表之后，还可以筛选user\_info的数据，也就是针对user\_info表进行查询：

```
query.eq("t2.city", "杭州");
```

## 关于easymybatis

easymybatis是一个mybatis增强类库，目的为简化mybatis的开发，让开发更高效。

- git地址：[easymybatis](#)
- 文档地址：[doc](#)
- demo地址：[demo with springboot](#)

## Query类详解

Query是一个查询参数类，通常配合Dao一起使用。

### 参数介绍

Query里面封装了一系列查询参数，主要分为以下几类：

- 分页参数：设置分页
- 排序参数：设置排序字段
- 条件参数：设置查询条件
- 字段参数：可返回指定字段

下面逐个讲解每个参数的用法。

### 分页参数

一般来说分页的使用比较简单，通常是两个参数，pageIndex：当前页索引，pageSize：每页几条数据。Query类使用\*\*setPage(pageIndex, pageSize)\*\*方法来设置。假如我们要查询第二页，每页10条数据，代码可以这样写：

```
Query query = new Query();
query.setPage(2, 10);
List<User> list = dao.find(query);
```

如果要实现不规则分页，可以这样写：

```
Query query = new Query();
query.setStart(3).setLimit(5);
// 对应mysql：limit 3,5
```

- 如果要查询所有数据,则可以这样写:

```
Query query = new Query();
query.setQueryAll(true);
List<User> list = dao.find(query);
```

### 排序参数

设置排序，有两个方法：

```
addSort(String sortname)
addSort(String sortname, Sort sort)
```

其中sortname为数据库字段，非javaBean属性

- addSort(String sortname) 添加排序字段，默认使用ASC
- addSort(String sortname, Sort sort)则可以指定排序方式，Sort为排序方式枚举 假如要按照添加时间倒序，可以这样写：

```
Query query = new Query();
query.addSort("create_time", Sort.DESC);
dao.find(query);
```

添加多个排序字段可以在后面追加：

```
query.addSort("create_time", Sort.DESC).addSort("id", Sort.ASC);
```

### 条件参数

条件参数是用的最多一个，因为在查询中往往需要加入各种条件。easymybatis在条件查询上面做了一些封装，这里不做太多讲解，只讲下基本的用

法，以后会单独开一篇文章来介绍。感兴趣的同学可以自行查看源码，也不难理解。

条件参数使用非常简单，Query对象封装一系列常用条件查询。

- 等值查询eq(String columnName, Object value)，columnName为数据库字段名，value为查询的值 假设我们要查询姓名为张三的用户，可以这样写：

```
Query query = new Query();
query.eq("username","张三");
List<User> list = dao.find(query);
```

通过方法名即可知道eq表示等于'='，同理lt表示小于<,gt表示大于>

查询方式	说明
eq	等于=
gt	大于>
lt	小于<
ge	大于等于>=
le	小于等于<=
notEq	不等于<>
like	模糊查询
in	in()查询
notIn	not in()查询
isNull	NULL值查询
notNull	IS NOT NULL
notEmpty	字段不为空，非NULL且有内容
isEmpty	字段为NULL或者为''

如果上述方法还不能满足查询需求的话，我们可以使用自定sql的方式来编写查询条件，方法为：

```
Query query = new Query();
query.sql(" username='Jim' OR username='Tom'");
```

注意：sql()方法不会处理sql注入问题，因此尽量少用。

## 字段参数

在某些场景下，我们只想获取表里面几个字段的信息，不想查询所有字段。此时使用方式如下：

```
Query query = new Query();
// 只返回id,username
query.setColumns(Arrays.asList("id","username"));
List<TUser> list = dao.find(query);
```

这里的"id"，"username"都为数据库字段。

查询

## 关于easymybatis

easymybatis是一个mybatis增强类库，目的为简化mybatis的开发，让开发更高效。

- git地址：[easymybatis](#)
- 文档地址：[doc](#)
- demo地址：[demo with springboot](#)

## 实体类主键策略设置

跟hibernate的主键生成策略一致

### 主键自增

数据库主键设置自增后，这样设置：

```
@Id
@Column(name = "id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;
```

这样在做insert后，id会自动填充自增后的值。

### 主键使用uuid

数据库主键是varchar类型，insert后自动填充uuid，并返回。

```
@Id
@Column(name = "id")
@GeneratedValue(generator = "system-uuid")
private String id;
```

这样在做insert后，id字段会自动填充uuid。

- 注：uuid的生成方式是调用数据库底层实现，如MySQL的实现方式为：SELECT UUID()

## JavaBean中使用枚举字段

数据库中一些状态字段通常用0,1,2或者简单的字符串进行维护，然后JavaBean实体类中用枚举类型来保存，这样做便于使用和维护。

easymybatis上使用枚举属性很简单：枚举类实现net.oschina.durcframework.easymybatis.handler.BaseEnum接口即可。

下面是具体例子：

### 第一步

```
public enum UserInfoType implements BaseEnum<String> {
 INVALID("0"),VALID("1")
 ;

 private String status;

 UserInfoType(String type) {
 this.status = type;
 }

 @Override
 public String getCode() {
 return status;
 }
}
```

首先定义一个枚举类，实现BaseEnum接口，接口类型参数用String，表示保存的值是String类型，如果要保存Int类型的话改用BaseEnum。

### 第二步

在javaBean添加该枚举属性：

```
public class UserInfo {
 ...
 private UserInfoType status;

 // 省略getter setter
}
```

接下来就可以使用dao来进行数据操作了，下面是完整测试用例：

```
public class UserInfoDaoTest extends EasymybatisSpringbootApplicationTests {

 @Autowired
 UserInfoDao userInfoDao;

 @Test
 public void testGet() {
 UserInfo userInfo = userInfoDao.get(3);
 print("枚举字段status : " + userInfo.getStatus().getCode());
 print(userInfo);
 }

 @Test
 public void testUpdate() {
 UserInfo userInfo = userInfoDao.get(3);
 // 修改枚举值
 userInfo.setStatus(UserInfoType.INVALID);
 userInfoDao.update(userInfo);
 }

 @Test
 public void testSave() {
 UserInfo userInfo = new UserInfo();
 userInfo.setAddress("aa");
 }
}
```

```
userInfo.setCity("杭州");
userInfo.setCreateTime(new Date());
userInfo.setUserId(3);
// 枚举值
userInfo.setStatus(UserInfoType.VALID);
userInfoDao.save(userInfo);
}
}
```

## 字段自动填充

假设数据库表里面有两个时间字段gmt\_create,gmt\_update。

当进行insert操作时gmt\_create，gmt\_update字段需要更新。当update时，gmt\_update字段需要更新。

通常的做法是通过Entity手动设置：

```
User user = new User();
user.setGmtCreate(new Date());
user.setGmtUpdate(new Date());
```

因为表设计的时候大部分都有这两个字段，所以对每张表都进行手动设置的话很容易错加、漏加。easymybatis提供了两个辅助类DateFillInsert和DateFillUpdate，用来处理添加修改时的时间字段自动填充。配置了这两个类之后，时间字段将会自动设置。

配置方式如下：

```
EasymybatisConfig config = new EasymybatisConfig();

config.setFills(Arrays.asList(
 new DateFillInsert()
 ,new DateFillUpdate()
));
```

在spring的xml中配置如下：

```
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
 <property name="dataSource" ref="dataSource" />
 <property name="configLocation">
 <value>classpath:mybatis/mybatisConfig.xml</value>
 </property>
 <property name="mapperLocations">
 <list>
 <value>classpath:mybatis/mapper/*.xml</value>
 </list>
 </property>

 <!-- 以下是附加属性 -->

 <!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
 多个用;隔开
 -->
 <property name="basePackage" value="com.myapp.dao" />
 <property name="config">
 <bean class="net.oschina.durcframework.easymybatis.EasymybatisConfig">
 <!-- 定义填充器 -->
 <property name="fills">
 <list>
 <bean class="com.xxx.DateFillInsert"/>
 <bean class="com.xxx.DateFillUpdate"/>
 </list>
 </property>
 </bean>
 </property>
</bean>
```

springboot中可以这样定义：

```
@Configuration
public class GlobalConfig {

 @Bean
 public ExternalConfigBean externalConfigBean() {
 ExternalConfigBean bean = new ExternalConfigBean();
 bean.setFills(Arrays.asList(new DateFillInsert()));
 }
}
```

```
 return bean;
}

}
```

如果要指定字段名，可以写成：

```
new DateFillInsert("add_time")
```

## 自定义填充器

除了使用easymybatis默认提供的填充之外，我们还可以自定义填充。

自定义填充类要继承FillHandler类。 表示填充字段类型，如Date，String，BigDecimal，Boolean。

## 实战

现在有个remark字段，需要在insert时初始化为“备注默认内容”，新建一个StringRemarkFill类如下：

```
public class StringRemarkFill extends FillHandler<String> {

 @Override
 public String getColumnName() {
 return "remark";
 }

 @Override
 public FillType getFillType() {
 return FillType.INSERT;
 }

 @Override
 protected Object getFillValue(String defaultValue) {
 return "备注默认内容";
 }

}
```

StringRemarkFill类中有三个重写方法：

- getColumnName()：指定表字段名
- getFillType()：填充方式，FillType.INSERT:仅insert时填充；FillType.UPDATE：insert，update时填充
- getFillValue(String defaultValue)：返回填充内容。

然后在easymybatisConfig中添加

```
config.setFills(Arrays.asList(
 new DateFillInsert()
 ,new DateFillUpdate()
 ,new StringRemarkFill()
));
```

这样就配置完毕了，调用dao.save(user);时会自动填充remark字段。

## 指定目标类

上面说到StringRemarkFill填充器，它作用在所有实体类上，也就是说实体类如果有remark字段都会自动填充。这样显然是不合理的，解决办法是指定特定的实体类。只要重写FillHandler类的getTargetEntityClasses()方法即可。

```
@Override
public Class<?>[] getTargetEntityClasses() {
 return new Class<?>[] { TUser.class };
}
```

```
}
```

这样就表示作用在TUser类上，多个类可以追加。最终代码如下：

```
public class StringRemarkFill extends FillHandler<String> {

 @Override
 public String getColumnName() {
 return "remark";
 }

 @Override
 public Class<?>[] getTargetEntityClasses() {
 return new Class<?>[] { TUser.class }; // 只作用在TUser类上
 }

 @Override
 public FillType getFillType() {
 return FillType.INSERT;
 }

 @Override
 protected Object getFillValue(String defaultValue) {
 return "备注默认内容"; // insert时填充的内容
 }

}
```

关于自动填充的原理是基于mybatis的TypeHandler实现的，这里就不多做介绍了。感兴趣的同学可以查看FillHandler源码。

## EasymybatisConfig配置项说明

EasymybatisConfig类里面存放一些配置参数，这些参数都有默认值，一般情况下可以不用。EasymybatisConfig的使用方法也很简单，spring普通注入一个就行了。

### springboot 配置方式：

```
mybatis.config-location=classpath:mybatis/mybatisConfig.xml
mybatis.mapper-locations=classpath:mybatis/mapper/*.xml
mybatis.camel2underline=true
mybatis.common-sql-classpath=xx
mybatis.mapper-save-dir=d:/mapper
```

### 注解使用方式：

```
@Bean(name = sqlSessionFactoryName)
public SqlSessionFactoryBean sqlSessionFactoryBean(@Autowired DataSource dataSource) throws Exception {
 SqlSessionFactoryBeanExt bean = new SqlSessionFactoryBeanExt();

 bean.setDataSource(dataSource);
 bean.setConfigLocation(this.getResource(mybatisConfigLocation));
 bean.setMapperLocations(this.getResources(mybatisMapperLocations));

 // ===以下是附加属性===

 // dao所在的包名,跟MapperScannerConfigurer的basePackage一致,多个用;隔开
 bean.setBasePackage(basePackage);
 bean.setConfig(getConfig());

 return bean;
}

public EasymybatisConfig getConfig() {
 EasymybatisConfig config = new EasymybatisConfig();

 config.setXXX();
 ...
 return config;
}
```

### xml文件注入方式：

```
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
 <property name="dataSource" ref="dataSource" />
 <property name="configLocation">
 <value>classpath:mybatis/mybatisConfig.xml</value>
 </property>
 <property name="mapperLocations">
 <list>
 <value>classpath:mybatis/mapper/*.xml</value>
 </list>
 </property>

 <!-- 以下是附加属性 -->

 <!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
 多个用;隔开
 -->
 <property name="basePackage" value="com.myapp.dao" />
 <property name="config">
 <bean class="net.oschina.durcframework.easymybatis.EasymybatisConfig">
 <property name="属性名" value="属性值"/>
 </bean>
 </property>
</bean>
```

```
</bean>
</property>
</bean>
```

下面就讲解下各个属性的作用：

#### camel2underline

如果为true，则开启驼峰转换下划线功能。实体类中的java字段映射成数据库字段将自动转成下划线形式。可以省略@Column注解。默认true。

#### mapperExecutorPoolSize

mapper处理线程数，此项配置可以加快启动速度。默认值50。

#### templateClasspath

指定模板文件class路径。如果没有指定，则默认读取easymybatis/tpl/下的模板，一般情况下不做配置。

#### commonSqlClasspath

指定公共SQL模块class路径，如果没有指定，则默认读取easymybatis/commonSql.xml，一般情况下不做配置。

#### mapperSaveDir

指定mapper文件存放路径。因为easymybatis是直接将mapper内容注入到内存当中，开发人员无感知，并且不知道mapper内容是什么样子，这个功能就是让开发人员能够查看到对应的mapper内容，方便定位和排查问题。一般情况下此项不用开启。

## 数据库增减字段后的SQL维护

传统的mybatis的开发在维护sql上比较麻烦，比如增加一个数据库字段，需要在对应的CRUD语句上添加字段信息。这样会导致错加，漏加等情况发生。

easymybatis就可以避免这种情况，因为它是围绕实体类来开发的（类似于hibernate）。easymybatis根据定义好的实体类自动生成与之对应的CRUD语句。这样在增减数据库字段的时候，只要在实体类上做增减就可以了。

## easymybatis的设计初衷

凡事都有两面性，软件也一样，优点与缺点并存。

easymybatis也同样存在问题。比如有同学说到维护性问题，按我的理解应该是SQL语句的维护。如果把所有的查询都用代码来实现的话，这样确实有影响。

但easymybatis的并不鼓励把所有查询条件都用代码实现。easymybatis同样支持手写sql到xml中，跟之前的用法是完全兼容的，甚至不用easymybatis的功能都可以。

我之前做项目是大多数是公司的IT系统，业务相对来说不是很复杂，对表的CRUD做的比较多。每次新增一张表对它进行CRUD操作时候都要手写sql，然而每次写sql基本是差不多的，无非就是表名，字段名不一样。久而久之就写烦了，所以想写个工具来帮我写。

当初的工具也挺简单的，根据数据库表生成一些通用CRUD语句，然后放在xml中。对于增删改来说，一般没什么问题。问题最多的还是查询上面。因为查询对于每个业务都不一样。今天要查询state=1的记录，明天要查询username=张三的记录。

我要做的操作就是，先在xml中写一条sql语句：

```
<select id="getByUsername" resultType="TUser">
 select * from t_user t where t.username = #{username} limit 1
</select>
```

然后在Dao中添加对应的方法：

```
TUser getByUsername(String username);
```

这很烦，真的，当中充斥这大量的复制黏贴。如果Dao里面有个getByProperty(String columnName,Object value);方法那该多好啊，意思是根据某个字段的某个值查询记录。

上面的操作只要这一句话就行了：

```
TUser user = dao.getByProperty("username", "张三");
```

因此可以事先写一段代码

```
<select id="getByProperty" resultType="TUser">
 select * from t_user t where ${columnName} = #{value} limit 1
</select>
```

这样的话就一劳永逸了。easymybatis做的就是这个工作。

当我们写了大量的相似代码时候，就应该考虑提取和封装了，尽量做抽象化，简单化，而且机器生成的sql语句肯定没问题的，不会出现把from写成form的情况。

可能有的同学在用mybatis的时候会去想hibernate，用hibernate的时候会想mybatis。既然鱼和熊掌不能兼得，为何不虚拟一个出来呢，在功能上尽量往一边靠，在CRUD上面跟hibernate一样，在自定义sql上面可以用mybatis，岂不美哉。

现在easymybatis的核心功能是根据TUser.java自动生成一些基本的CRUD语句，如此一来我们只需要维护TUser就行了，比如新增了一个字段，只需要在TUser类里面新增即可。往常的做法还要在xml中逐个添加，漏加，错加在所难免。

最后总结下吧，在使用简单CRUD语句的时候用easymybatis，复杂SQL写在xml中，做个平衡。这样在维护上面不会有太大压力。

## 驼峰转下划线形式（默认开启）

如果开启了驼峰转下划线形式，javaBean中的字段是驼峰形式，映射到数据库字段会转换成下划线形式。如：userAge -> user\_age

如果数据库设计规范全部用下划线形式的话，此项配置可以启用。

启用的好处是注解变少了，@Table,@Column不需要使用，只需要在主键字段上使用 @Id @GeneratedValue

如果数据库既有驼峰又有下划线命名，最好加上@Table,@Column注解。

配置方式：

```
<!-- 替换org.mybatis.spring.SqlSessionFactoryBean -->
<bean id="sqlSessionFactory"
class="net.oschina.durcframework.easymybatis.ext.SqlSessionFactoryBeanExt">
<property name="dataSource" ref="dataSource" />
<property name="configLocation">
<value>classpath:mybatis/mybatisConfig.xml</value>
</property>
<property name="mapperLocations">
<list>
<value>classpath:mybatis/mapper/*.xml</value>
</list>
</property>

<!-- 以下是附加属性 -->

<!-- dao所在的包名,跟MapperScannerConfigurer的basePackage一致
多个用;隔开
-->
<property name="basePackage" value="com.myapp.dao" />
<property name="config">
<bean class="net.oschina.durcframework.easymybatis.EasymybatisConfig">
<!--
/* 驼峰转下划线形式，默认是true
开启后java字段映射成数据库字段将自动转成下划线形式
如：userAge -> user_age
如果数据库设计完全遵循下划线形式，可以启用
这样可以省略Entity中的注解，@Table，@Column都可以不用，只留
@Id
@GeneratedValue
参见：UserInfo.java
*/-->
<property name="camel2underline" value="true"/>

</bean>
</property>
</bean>
```

## TUserDaoTest.java

```

/**
 * dao测试
 */
public class TUserDaoTest extends EasyMybatisSpringbootApplicationTests {

 @Resource
 TUserDao dao;
 @Resource
 TransactionTemplate transactionTemplate;

 // 根据主键查询
 @Test
 public void testGet() {
 TUser user = dao.get(3);
 print(user);
 }

 // 根据字段查询一条记录
 @Test
 public void testGetByProperty() {
 TUser user = dao.getByProperty("username", "王五");
 print(user);
 }

 // 根据条件查询一条记录
 @Test
 public void testGetByExpression() {
 // 查询ID=3的用户
 Query query = Query.build().eq("id", 3);

 TUser user = dao.getByExpression(query);

 print(user);
 }

 // 条件查询
 // 查询username='张三'的用户
 @Test
 public void testFind() {
 Query query = new Query();
 // 添加查询条件
 query.eq("username", "张三");

 List<TUser> list = dao.find(query); // 获取结果集
 long count = dao.countTotal(query); // 获取总数
 print("count:" + count);
 for (TUser user : list) {
 System.out.println(user);
 }
 }

 // 分页查询
 /**MYSQL语句:
 *
 * SELECT t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`
 * FROM `t_user` t
 * ORDER BY id DESC
 * LIMIT ?,?
 *
 * Parameters: 2(Integer), 2(Integer)
 */
 @Test
 public void testPage() {
 Query query = new Query();

 query.setPage(2, 2) // 设置pageIndex , pageSize
 .addSort("id", Sort.DESC); // 添加排序
 }
}

```

```

// 查询后的结果，包含总记录数，结果集，总页数等信息
PageInfo<TUser> pageInfo = QueryUtils.query(dao, query);

List<TUser> rows = pageInfo.getList();
for (TUser user : rows) {
 System.out.println(user);
}
}

// 自定义返回字段查询，只返回两个字段
// SELECT t.id,t.username FROM `t_user` t LIMIT 0,10
@Test
public void testSelfColumns() {
 Query query = new Query();
 // 只返回id,username
 query.setColumns(Arrays.asList("t.id","t.username"));

 List<TUser> list = dao.find(query);

 for (TUser user : list) {
 System.out.println(user);
 }
}

// 多表查询,left join
// 适用场景：获取两张表里面的字段信息返回给前端
/*
 * MYSQL生成如下sql:
SELECT
t.`id`, t.`username`, t.`state`, t.`isdel`, t.`remark`, t.`add_time`, t.`money`, t.`left_money`
, t2.city, t2.address
FROM `t_user` t left join user_info t2 on t.id = t2.user_id
WHERE t2.id = ?
ORDER BY id ASC
LIMIT ?,?
*/
@Test
public void testJoin() {
 Query query = new Query();
 // 添加第二张表的字段,跟主表字段一起返回
 query.addOtherColumns(
 "t2.city"
 , "t2.address"
);
 // 左连接查询,主表的alias默认为t
 query.join("left join user_info t2 on t.id = t2.user_id");
 // 添加条件
 query.eq("t2.id", 2);

 query.addSort("t.id");

 List<TUser> list = dao.find(query);

 System.out.println("=====");
 for (TUser user : list) {
 System.out.println(
 user.getId()
 + " " + user.getUsername()
 // 下面两个字段是第二张表里面的
 + " " + user.getCity()
 + " " + user.getAddress()
);
 }
 System.out.println("=====");
}

/*
 * 聚合查询
 * 按state分组统计money和,并且按照state升序
 * 并且money大于200的state

```

```

*
* SELECT
* state AS s , SUM(money) AS m
* FROM `t_user` t
* WHERE money > 0
* GROUP BY state
* HAVING m > 200
* ORDER BY state asc
*/
@Test
public void testProjection() {

 ProjectionQuery query = new ProjectionQuery();
 // 添加列
 query.addProjection(Projections.column("state", "s"));
 query.addProjection(Projections.sum("money", "m"));
 // 添加where
 query.addExpression(new ValueExpression("money", ">", 0));
 // 添加group by
 query.addGroupBy("state");
 // 添加having
 query.addHaving(new ValueExpression("SUM(money)", ">", 200));

 query.addSort("state", Sort.DESC);

 List<Map<String, Object>> list = dao.findProjection(query);

 Assert.notEmpty(list);

 print(list);
}

// 自定义sql，见TUserDao.xml
// 注意mybatis的mapper必须跟Dao类一致
@Test
public void testSelfSql() {
 TUser user = dao.selectByName("张三");

 print(user);
}

// 添加-保存所有字段
@Test
public void testInsert() {
 TUser user = new TUser();
 user.setAddTime(new Date());
 user.setIsdel(false);
 user.setLeftMoney(22.1F);
 user.setMoney(new BigDecimal(100.5));
 user.setRemark("备注");
 user.setState((byte)0);
 user.setUsername("张三");

 dao.save(user);

 print("添加后的主键:" + user.getId());
 print(user);
}

// 添加-保存非空字段
@Test
public void testInsertIgnoreNull() {
 TUser user = new TUser();
 user.setAddTime(new Date());
 user.setIsdel(true);
 user.setMoney(new BigDecimal(100.5));
 user.setState((byte)0);
 user.setUsername("张三notnull");
 user.setLeftMoney(null);
 user.setRemark(null);
}

```

```

dao.saveIgnoreNull(user);

print("添加后的主键:" + user.getId());
print(user);
}

// 批量添加
/*
 * 支持mysql,sqlserver2008。如需支持其它数据库使用saveMulti方法
 * INSERT INTO person (id, name, age)
VALUES
 (1, 'Kelvin', 22),
 (2, 'ini_always', 23);
 */
@Test
public void testInsertBatch() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("username" + i);
 user.setMoney(new BigDecimal(i));
 user.setRemark("remark" + i);
 user.setState((byte)0);
 user.setIsdel(false);
 user.setAddTime(new Date());
 user.setLeftMoney(200F);
 users.add(user);
 }

 int i = dao.saveBatch(users); // 返回成功数

 System.out.println("saveBatch --> " + i);
}

/**
 * 批量添加,兼容更多数据库版本,采用union all
 * INSERT INTO [t_user] ([username] , [state] , [isdel] , [remark] , [add_time] , [money] , [left_money])
 * SELECT ? , ? , ? , ? , ? , ? , ?
 * UNION ALL SELECT ? , ? , ? , ? , ? , ? , ?
 * UNION ALL SELECT ? , ? , ? , ? , ? , ? , ?
 */
@Test
public void testInsertMulti() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("username" + i);
 user.setMoney(new BigDecimal(i));
 user.setRemark("remark" + i);
 user.setState((byte)3);
 user.setIsdel(false);
 user.setAddTime(new Date());
 user.setLeftMoney(200F);
 users.add(user);
 }

 int i = dao.saveMulti(users); // 返回成功数

 System.out.println("saveMulti --> " + i);
}

// 批量添加指定字段,仅支持mysql, sqlserver2008, 如需支持其它数据库使用saveMulti方法
@Test
public void testInsertBatchWithColumns() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("usernameWithColumns" + i);
 }

```

```

 user.setMoney(new BigDecimal(i));
 user.setAddTime(new Date());

 users.add(user);
 }

 /*
 * INSERT INTO `t_user` (username , money , add_time)
 * VALUES (? , ? , ?) , (? , ? , ?) , (? , ? , ?)
 */
 int i = dao.saveBatchWithColumns(Arrays.asList(
 new Column("username", "username") // 第一个是数据库字段,第二个是java字段
 ,new Column("money", "money")
 ,new Column("add_time", "addTime")
), users);

 System.out.println("saveBatchWithColumns --> " + i);

}

/*
 * // 批量添加指定字段,兼容
 * INSERT INTO [t_user] ([username] , [money] , [add_time])
 * SELECT ? , ? , ?
 * UNION ALL SELECT ? , ? , ?
 * UNION ALL SELECT ? , ? , ?
 */
@Test
public void testInsertMultiWithColumns() {
 List<TUser> users = new ArrayList<>();

 for (int i = 0; i < 3; i++) { // 创建3个对象
 TUser user = new TUser();
 user.setUsername("usernameWithColumns" + i);
 user.setMoney(new BigDecimal(i));
 user.setAddTime(new Date());

 users.add(user);
 }

 int i = dao.saveMultiWithColumns(Arrays.asList(
 new Column("username", "username") // 第一个是数据库字段,第二个是java字段
 ,new Column("money", "money")
 ,new Column("add_time", "addTime")
), users);

 System.out.println("saveMultiWithColumns --> " + i);
}

// 事务回滚
@Test
public void testUpdateTran() {
 TUser user = transactionTemplate.execute(new TransactionCallback<TUser>() {
 @Override
 public TUser doInTransaction(TransactionStatus arg0) {
 try{
 TUser user = dao.get(3);
 user.setUsername("王五1");
 user.setMoney(user.getMoney().add(new BigDecimal(0.1)));
 user.setIsdel(true);

 int i = dao.update(user);
 print("testUpdate --> " + i);
 int j = 1/0; // 模拟错误
 return user;
 }catch(Exception e) {
 e.printStackTrace();
 arg0.setRollbackOnly();
 return null;
 }
 }
 });
}

```

```

});

print(user);
}

// 更新所有字段
@Test
public void testUpdate() {
 TUser user = dao.get(3);
 user.setUsername("李四");
 user.setMoney(user.getMoney().add(new BigDecimal(0.1)));
 user.setIsdel(true);

 int i = dao.update(user);
 print("testUpdate --> " + i);
}

// 更新不为null的字段
/*
 *UPDATE [t_user] SET [username]=?, [isdel]=? WHERE [id] = ?
 */
@Test
public void updateIgnoreNull() {
 TUser user = new TUser();
 user.setId(3);
 user.setUsername("王五");
 user.setIsdel(false);
 int i = dao.updateIgnoreNull(user);
 print("updateNotNull --> " + i);
}

// 根据条件更新
// UPDATE t_user SET remark = '批量修改备注' WHERE state = 0
@Test
public void testUpdateNotNullByExpression() {
 Query query = new Query();
 query.eq("state", 0);

 TUser user = new TUser();
 user.setRemark("批量修改备注");

 int i = dao.updateIgnoreNullByExpression(user, query);
 print("testUpdateNotNullByExpression --> " + i);
}

// 删除
@Test
public void testDel() {
 TUser user = new TUser();
 user.setId(14);
 int i = dao.del(user);
 print("del --> " + i);
}

// 根据条件删除
// DELETE FROM `t_user` WHERE state = ?
@Test
public void delByExpression() {
 Query query = new Query();
 query.eq("state", 3);
 int i = dao.delByExpression(query);
 print("delByExpression --> " + i);
}
}

```

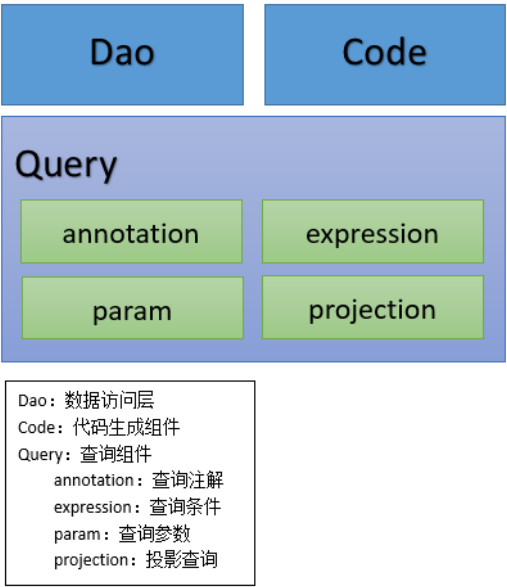
image

image

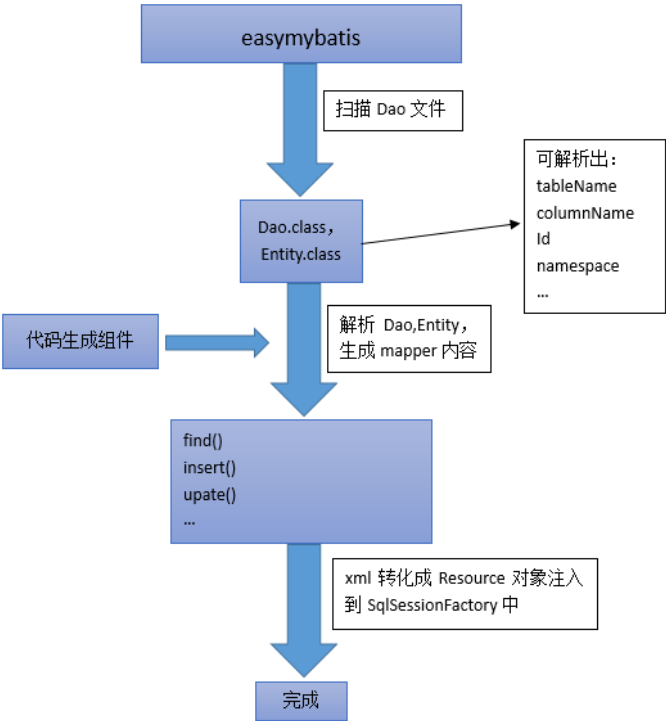
LOGO



架构



运行流程



Dao层方法

- UpdateDao<TUser>
  - update(TUser)
  - updateIgnoreNull(TUser)
  - updateIgnoreNullByExpression(TUser, Expressional)
- SchDao<TUser>
  - countTotal(Queryable)
  - find(Queryable)
  - findProjection(Queryable)
  - get(Object)
  - getByExpression(Queryable)
  - getProperty(String, Object)
  - listByProperty(String, Object, Queryable)
  - listByProperty(String, Object)
- SaveDao<TUser>
  - save(TUser)
  - saveBatch(List<TUser>)
  - saveBatchWithColumns(List<Column>, List<TUser>)
  - saveIgnoreNull(TUser)
  - saveMulti(List<TUser>)
  - saveMultiWithColumns(List<Column>, List<TUser>)
- DeleteDao<TUser>
  - del(TUser)

查询条件

- `eq(String, Object) : Query`
- `notEq(String, Object) : Query`
- `gt(String, Object) : Query`
- `gtAndEq(String, Object) : Query`
- `lt(String, Object) : Query`
- `ltAndEq(String, Object) : Query`
- `like(String, Object) : Query`
- `in(String, Collection<?>) : Query`
- `in(String, Collection<?>, ValueConvert) : Query`
- `in(String, Object[]) : Query`
- `notIn(String, Collection<?>) : Query`
- `notIn(String, Collection<?>, ValueConvert) : Query`
- `notIn(String, Object[]) : Query`
- `sql(String) : Query`
- `notNull(String) : Query`
- `isNull(String) : Query`
- `notEmpty(String) : Query`
- `isEmpty(String) : Query`
- `notEq() : Query`

# Home

welcome

