**day06-通用Mapper和RESTful Web Service**

# 课程大纲

1. 通用Mapper
2. Restful概述
   1. 什么是REST
   2. 什么是RESTful
   3. REST标准规范
3. REST最佳实践
4. REST实现CRUD
5. 改造综合练习为REST风格

# 反馈意见

http://ifedorenko.github.com/m2e-extras/

# 教学笔记

# 通用Mapper

## 通用Mapper概述

### 什么是通用Mapper？

开源中国的介绍页面：<https://www.oschina.net/p/mybatis-mapper>

GitHub介绍页面：<https://github.com/abel533/Mapper>



作者：<http://blog.csdn.net/isea533>



### 为什么要学习通用Mapper

在我们的日常开发中，单表的增删改查是一个非常频繁的操作，而通常，我们都需要自己来定义每一张表的Mapper.xml文件，并且定义增删改查的sql语句，这些工作具有大量的重复性，意义不大，很浪费时间。

通用Mapper可以通过Mybatis的拦截器原理，动态的帮我们实现单表的增删改查功能，大大降低了我们的开发成本，减少了我们的工作量。

## 导入依赖

<!-- 通用Mapper -->

<dependency>

<groupId>tk.mybatis</groupId>

<artifactId>mapper</artifactId>

<version>4.0.1</version>

</dependency>

## 配置通用Mapper与Spring整合

通用mapper可以直接与Spring整合：

在applicationContext-mybatis.xml中原来的mapper包扫描修改为：

<!-- mapper扫描包 -->

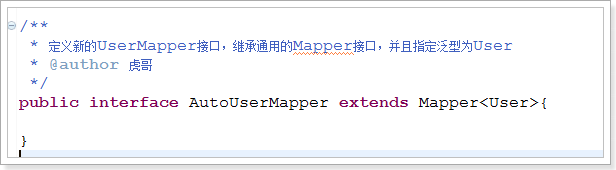
<bean class=*"tk.mybatis.spring.mapper.MapperScannerConfigurer"*>

<!-- mapper包的位置 -->

<property name=*"basePackage"* value=*"cn.itcast.usermanage.mapper"*/>

</bean>

## 使用通用Mapper接口来定义UserMapper



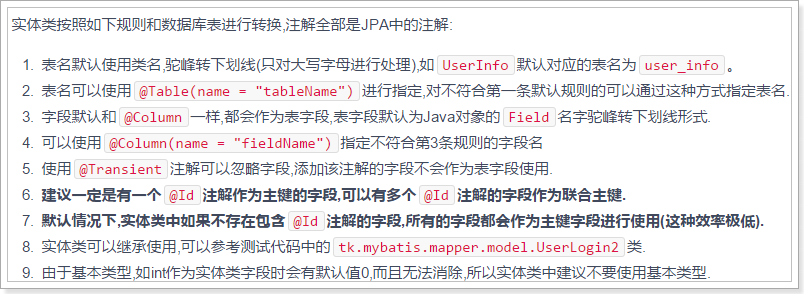
此时，我们的Mapper就已经具备了通用Mapper中提前定义好的方法：



现在，我们的UserMapper接口已经具备了单标的增删改查方法，但是并没有对应的Mapper.xml文件，我们不需要写。因为通用Mapper会帮我们动态生成这些方法对应的Sql和statement

那么问题来了：通用Mapper如何知道User对应数据库中那张表？如何知道对应数据库哪些字段？

## 设置实体类的字段





## 测试

### 查询数据

#### 查询1个用户

/\*

\* 查找1个用户，接收一个User对象。

\* 注意：

\* 1）会把User对象中的非空属性作为查询的条件，返回匹配到的一个用户！多个查询条件之间是AND关系

\* 2）如果查询的结果不止一个，会抛出异常。

\*/

@Test

**public** **void** testSelectOne() {

// 创建用户，里面的属性就是查询条件

User = **new** User();

// 根据用户名查找

user.setUserName("liuyan");

// 并且性别为女

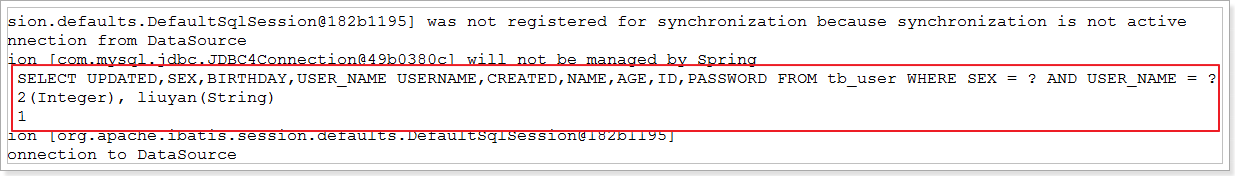
user.setSex(2);

User result = **this**.uesrMapper.selectOne(user);

System.***out***.println(result);

}

通用Mapper动态生成的SQL：



#### 查询用户集合

/\*

\* 查找用户，接收一个User对象。

\* 注意：

\* 1）会把User对象中的非空属性作为查询的条件，返回匹配到的多个用户的List集合！多个查询条件之间是AND关系

\* 2）如果想查询所有，可以传null

\*/

@Test

**public** **void** testSelect() {

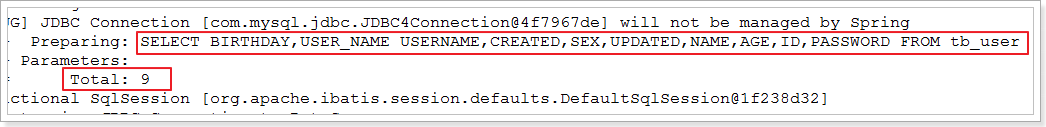
List<User> users = **this**.uesrMapper.select(**null**);

**for** (User user : users) {

System.***out***.println(user);

}

}



#### 查询总条数信息

/\*

\* 根据条件查询总条数信息,也是接收User对象作为查询条件

\* 返回符合条件的数据的条数

\*/

@Test

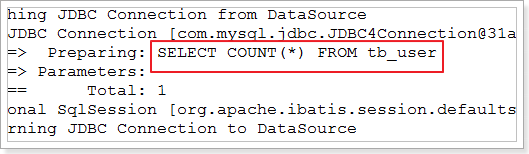
**public** **void** testSelectCount() {

// 根据条件查询总条数信息

**int** count = **this**.userMapper.selectCount(**null**);

System.***out***.println(count);

}



#### 根据主键查找

/\*

\* 根据主键查找，注意这里的主键是指在Bean当中加了@ID注解的字段

\*/

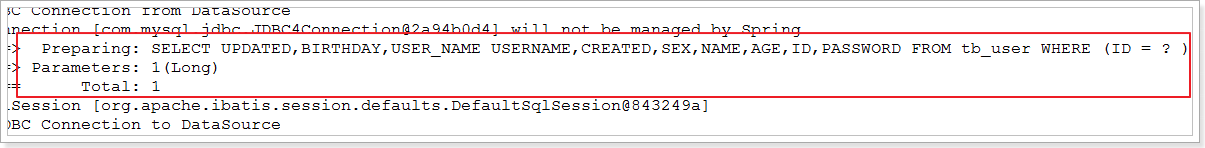
@Test

**public** **void** testSelectByPrimaryKey() {

User user = **this**.userMapper.selectByPrimaryKey(1L);

System.***out***.println(user);

}



### 添加数据

1）插入数据，不管字段是否为null

/\*

\* 添加一条数据。把接收到的对象插入数据库中，不管用户的字段是否为null。

\*/

@Test

**public** **void** testInsert() {

User user = **new** User();

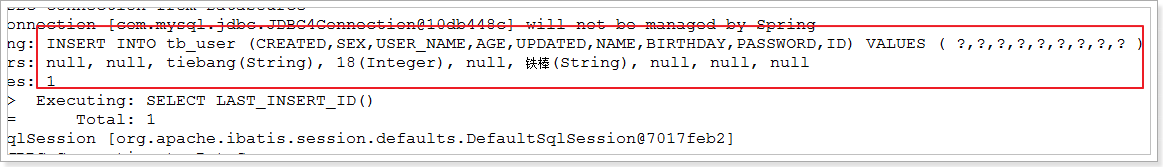
user.setUserName("tiebang");

user.setName("铁棒");

user.setAge(18);

**this**.userMapper.insert(user);

}



2）插入数据，只插入非null字段

/\*

\* 添加一条数据。把接收到的对象插入数据库中，但是仅仅插入的是非null字段

\*/

@Test

**public** **void** testInsertSelective() {

User user = **new** User();

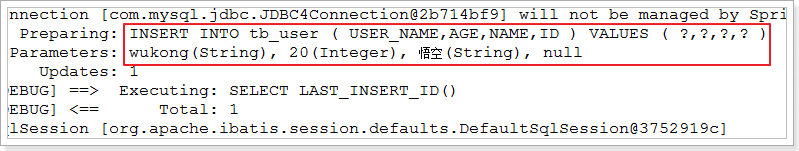
user.setUserName("wukong");

user.setName("悟空");

user.setAge(20);

**this**.userMapper.insertSelective(user);

}



### 删除数据

#### 根据组合条件删除

/\*

\* 根据条件删除。接收一个Use对象，把User的非空字段作为条件

\*/

@Test

**public** **void** testDelete() {

// 设置要删除的条件

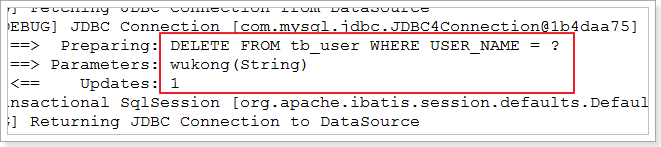
User user = **new** User();

user.setUserName("wukong");

// 删除用户

**this**.userMapper.delete(user);

}



#### 根据主键删除

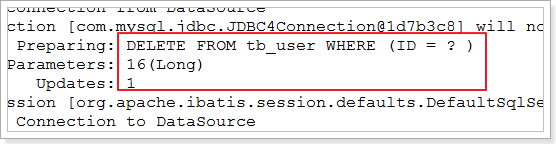
@Test

**public** **void** testDeleteByPrimaryKey() {

// 根据主键删除

**this**.userMapper.deleteByPrimaryKey(16L);

}



### 修改用户

/\*

\* 根据主键进行修改数据。

\* 1）接收一个User对象，其中的的主键ID作为修改的条件

\* 2）User的其它非null属性都会被修改

\*/

@Test

**public** **void** testUpdateByPrimaryKeySelective() {

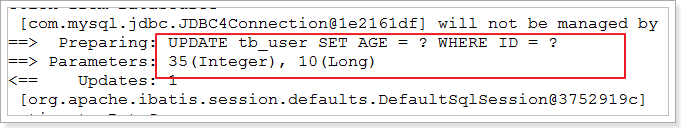
User user = **new** User();

user.setId(10L);// 用主键来确定要修改哪条数据

user.setAge(35);// 修改年龄属性

**this**.userMapper.updateByPrimaryKeySelective(user);

}

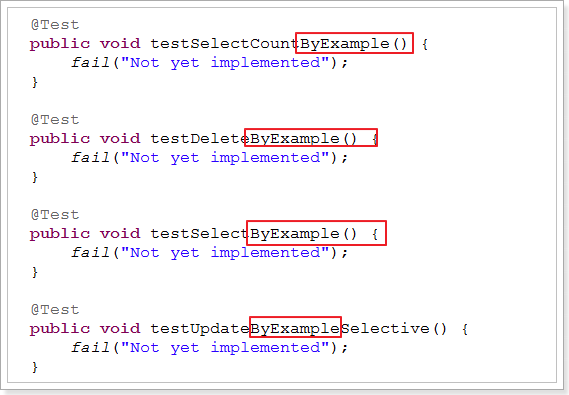


### 复杂条件的查询、删除和修改

我们发现，上面在进行删除、修改、查询时，where后面跟的条件，都是AND关系，一些复杂的，例如：OR 、IN、ORDER BY 等等都无法实现

这时就要用到通用Mapper中提供的 Example类了

删、改、查都有对应的 xxxByExample()方法



我们以查询为例，来介绍这个Example的使用

// 通过Example实现特殊条件的查询

@Test

**public** **void** testSelectByExample() {

// 创建Example对象，并且指定要操作的实体类的Class对象

Example example = **new** Example(User.**class**);

// 创建查询条件对象,默认是and关系

example.createCriteria().andEqualTo("sex", 2).andBetween("age", 16, 24);// 查询女性,并且年龄在16到24

// 添加查询条件，or关系

example.or(example.createCriteria().andEqualTo("userName", "lisi"));// 或者用户名是lisi的

// 实现排序，多个排序规则以','隔开

example.setOrderByClause("age desc");

List<User> users = **this**.userMapper.selectByExample(example);

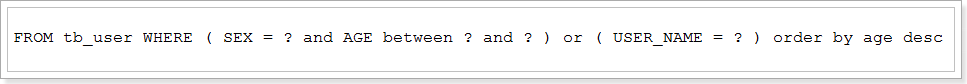
**for** (User user : users) {

System.***out***.println(user);

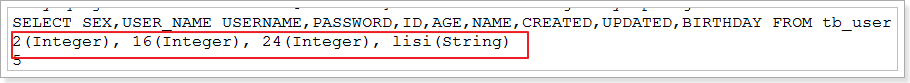
}

}

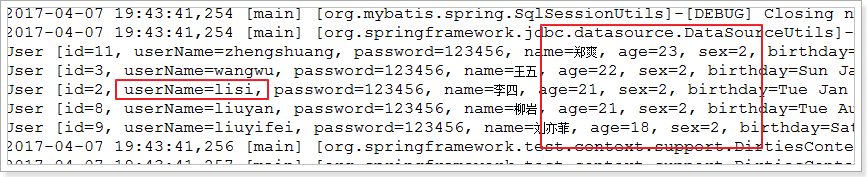
生成的SQL语句：



其中的参数：



查询的结果：



## 使用通用Mapper改造综合练习

* 改造UserService

/\*\*

\* 使用通用Mapper来实现UserService的功能

\* **@author** 虎哥

\*/

@Service

**public** **class** AutoUserService {

@Autowired

**private** AutoUserMapper userMapper;

/\*\*

\* 分页查询方法

\* **@param** page 当前页码

\* **@param** rows 每页条数

\* **@return**

\*/

**public** DataGridResult<User> queryUserListByPage(Integer page, Integer rows) {

// 使用分页助手开始分页,指定两个参数：当前页码，每页条数

PageHelper.*startPage*(page, rows);

// 然后分页拦截器会自动对接下来的查询进行分页

List<User> users = **this**.userMapper.select(**null**);// 不传查询条件

// 封装分页信息对象

PageInfo<User> pageInfo = **new** PageInfo<>(users);

// 封装页面数据对象

DataGridResult<User> result = **new** DataGridResult<>(pageInfo.getTotal(), users);

**return** result;

}

/\*\*

\* 新增用户的功能：

\* **@param** user

\*/

**public** **void** saveUser(User user) {

user.setCreated(**new** Date);

user.setUpdated(user.getCreated());

**this**.userMapper.insertSelective(user);

}

/\*\*

\* 删除用户功能

\* **@param** ids

\*/

**public** **void** deleteUserByIds(Long[] ids) {

// 把数组变为集合

List<Object> list = **new** ArrayList<>();

// 添加元素

Collections.*addAll*(list, ids);

// 因为是删除多条，需要用到in,因此我们使用Example

Example example = **new** Example(User.**class**);

// 创建条件，使用in

example.createCriteria().andIn("id", list);

**this**.userMapper.deleteByExample(example);

}

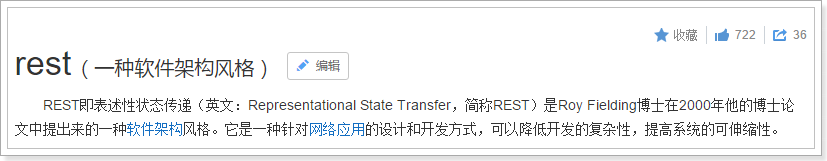
}

* 在Controller中注入新的Service

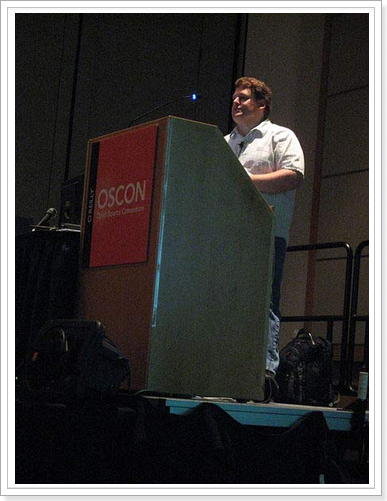


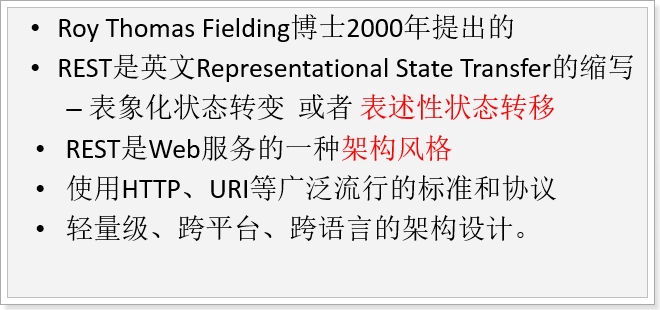
# RESTful概述

## 什么是REST



作者：





REST 仅仅是一种架构的风格，并不是真正的架构，也不是一个软件，而是一种思想。

我们可以基于现有的HTTP、URI、XML、等现有技术来实现REST的风格。而不用去学习任何新的技术。

而学习REST的关键，不是任何的API或者实现方式，而是这种思想。

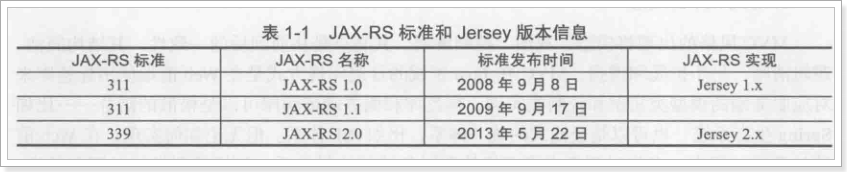
## 什么是RESTful



## REST标准规范

JAX-WS

在REST提出后，到了2008年，Java才具有了完善的官方REST式的WEB服务标准规范：JAX-RS标准：



Jersey是Java的JAX-RS标准的一种实现框架，就是一种RESTful 的框架，基于Jersey开发的WEB应用自然就是Restful的WEB服务

SpringMVC天生也是支持REST标准的。

### REST风格的WEB设计原则

从上面的定义中，我们可以发现REST其实是一种组织Web服务的架构风格，而并不是我们想象的那样是实现Web服务的一种新的技术，更没有要求一定要使用HTTP。其目标是为了创建具有良好扩展性的分布式系统。它提出了一系列架构级约束。这些约束有：

* **使用客户/服务器模型**。

客户和服务器之间通过一个统一的接口来互相通讯。

* **层次化的系统**

在一个REST系统中，客户端并不会固定地与一个服务器打交道。

* **无状态**

在一个REST系统中，服务端并不会保存有关客户的任何状态。也就是说，客户端自身负责用户状态的维持，并在每次发送请求时都需要提供足够的信息。

Http协议本身就是无状态的协议。

* **可缓存**

REST系统需要能够恰当地缓存请求，以尽量减少服务端和客户端之间的信息传输，以提高性能。

* **统一的接口**

一个REST系统需要使用一个统一的接口来完成子系统之间以及服务与用户之间的交互。这使得REST系统中的各个子系统可以独自完成演化。

如果一个系统满足了上面所列出的五条约束，那么该系统就被称为是RESTful的。

### 如何统一接口规则

要实现统一接口，必须满足一下条件：

* 每个资源都拥有一个资源标识。每个资源的资源标识可以用来唯一地标明该资源

REST是围绕资源为核心的，任何接口的设计都是围绕着资源进行，而不再是围绕着业务功能

一个典型的URI：protocol://host/resourceName

一般资源的URI中，用名词来表示一个资源，不允许出现动词

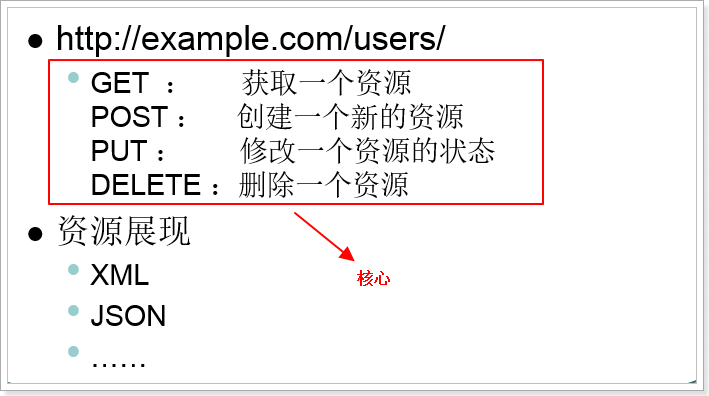
* 消息的自描述性

在REST系统中所传递的消息需要能够提供自身如何被处理的足够信息。例如该消息所使用的MIME类型，是否可以被缓存，返回结果包含哪些字段等

* 资源操作动作

不使用动词来描述要对资源进行的操作，那么REST中该如何表示CRUD呢？

一般我们会借助于HTTP协议中的请求方法来表明对资源的操作：



举例，

以前非REST时，我们的URI

查询用户： <http://localhost/user/query?id=1> - GET

添加用户： <http://localhost/user/insert> - POST

修改用户： <http://localhost/user/update> - POST

删除用户： <http://localhost/user/delete?id=1> - GET

遵循REST规范的URI定义：

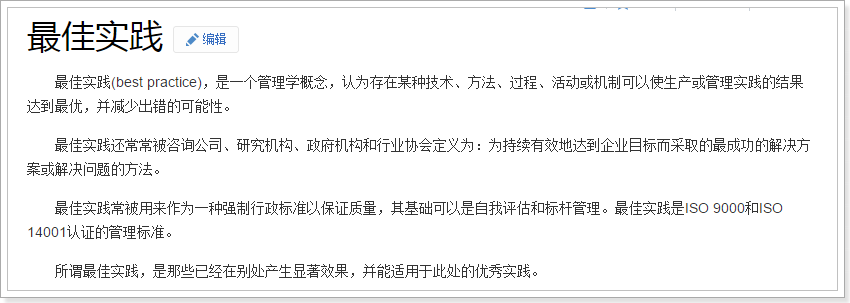
查询用户： [http://localhost/user](http://localhost/user/query?id=1)/{id} - GET

添加用户： [http://localhost/user](http://localhost/user/insert) - POST

修改用户： [http://localhost/user](http://localhost/user/update) - PUT

删除用户： [http://localhost/user](http://localhost/user/delete?id=1) - DELETE

# REST最佳实践



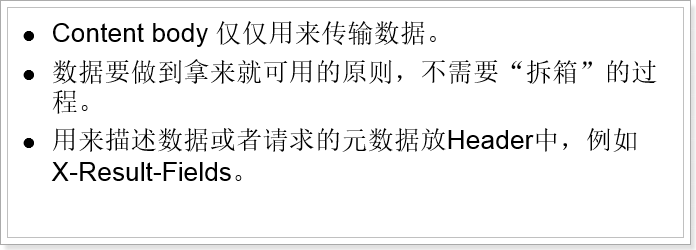
## 接口设计

REST中的最佳实践：



## 响应设计

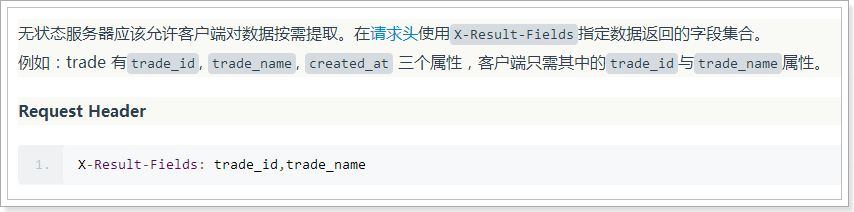
### 响应规则



示例：



### 响应字段



### 响应状态码



# REST实现用户CRUD

## 根据ID查询用户

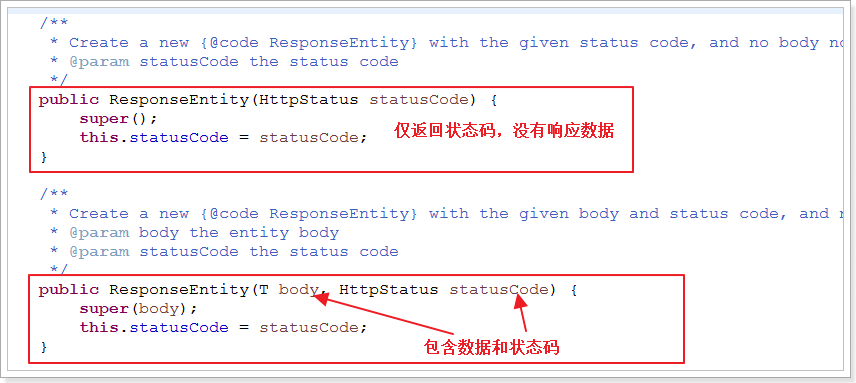
根据REST的原则，我们查询后需要根据结果返回不同的状态码：

资源存在：200

资源不存在：404

内部异常：500

这里通过ResponseEntity来封装响应的结果，包含两部分内容：数据和状态码



* 定义Controller

/\*\*

\* 根据ID查询用户

\* **@param** id

\* **@return**

\*/

@RequestMapping(value="{id}",method=RequestMethod.***GET***)

**public** ResponseEntity<User> queryUserById(@PathVariable("id") Long id){

**try** {

// 查询用户

User user = **this**.userService.queryUserById(id);

**if**(user != **null**){

// 资源存在，返回200

**return** **new** ResponseEntity<>(user, HttpStatus.***OK***);

}

// 资源不存在，返回404

**return** **new** ResponseEntity<>(HttpStatus.***NOT\_FOUND***);

} **catch** (Exception e) {

e.printStackTrace();

// 出现异常，返回500

**return** **new** ResponseEntity<>(HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

}

## 新增用户

在原来的新增用户功能中，我们返回的结果包含了status，也就是状态，这不符合REST风格。响应状态应该包含在header中

并且新增成功，应返回201状态码



/\*\*

\* 新增用户

\* **@param** user

\* **@return**

\*/

@RequestMapping(method=RequestMethod.***POST***)

**public** ResponseEntity<Void> saveUser(User user){

**try** {

// 新增用户

**this**.userService.saveUser(user);

// 新增成功，返回201

**return** **new** ResponseEntity<Void>(HttpStatus.***CREATED***);

} **catch** (Exception e) {

e.printStackTrace();

// 出现异常，返回500

**return** **new** ResponseEntity<>(HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

}

## 修改用户

### 编写Controller

修改成功，但是没有结果返回，根据REST原则，返回204：



/\*\*

\* 修改用户

\* **@param** user

\* **@return**

\*/

@RequestMapping(method=RequestMethod.***PUT***)

**public** ResponseEntity<Void> updateUser(User user){

**try** {

// 修改用户

**this**.userService.updateUser(user);

// 成功，返回204 ,操作成功，但是不返回数据

**return** **new** ResponseEntity<Void>(HttpStatus.***NO\_CONTENT***);

} **catch** (Exception e) {

e.printStackTrace();

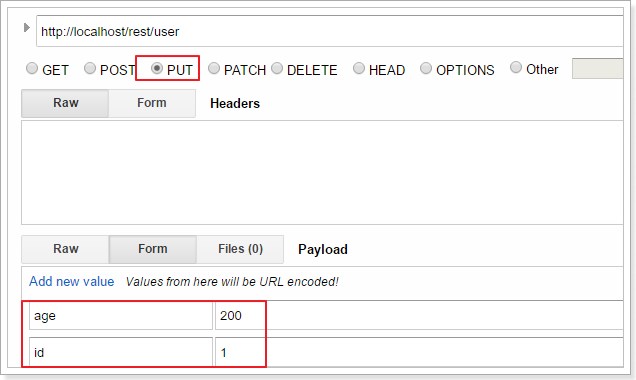
// 出现异常，返回500

**return** **new** ResponseEntity<>(HttpStatus.***INTERNAL\_SERVER\_ERROR***);

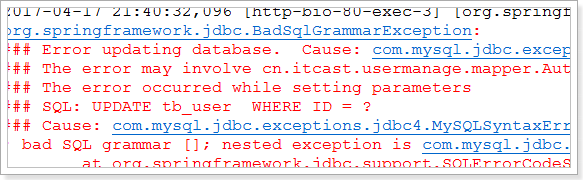
}

}

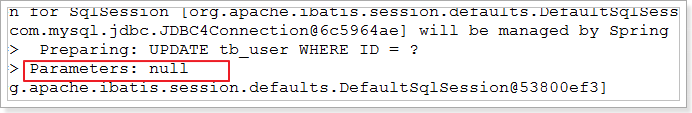
### 模拟Put请求进行测试



发生异常：



发现我们的参数都是null：



原因：

Java本身是不支持PUT和DELTE请求，当发起这些请求时，可以接收，但是请求参数接收不到！

### 解决PUT请求参数为null

#### 4.3.3.1方案一：

SpringMVC提供了一个过滤器，可以帮我们解析PUT请求的参数：

<!-- 解决PUT请求的参数为null问题 -->

<filter>

<filter-name>HttpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>

</filter>

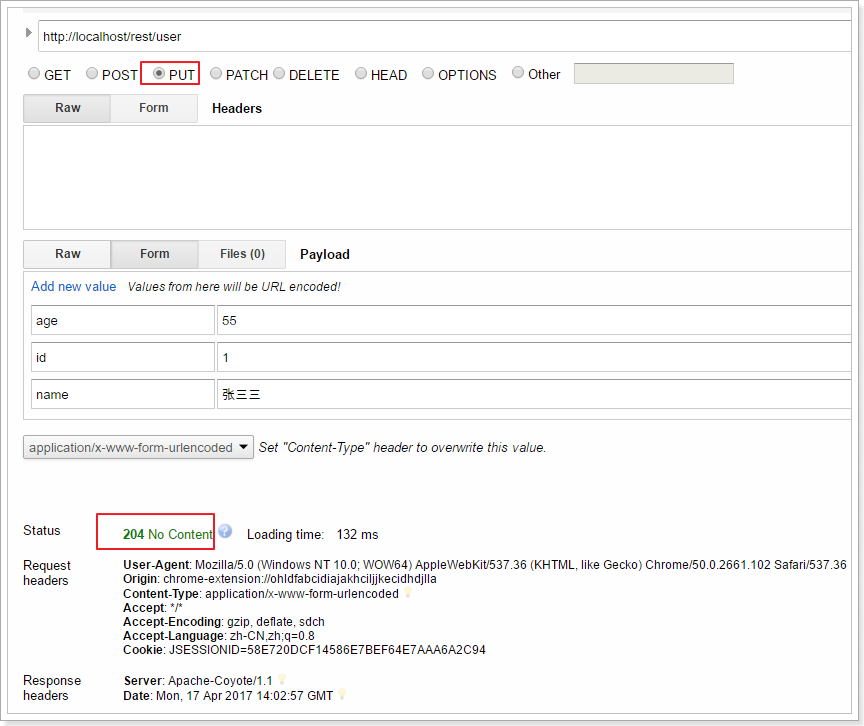
<filter-mapping>

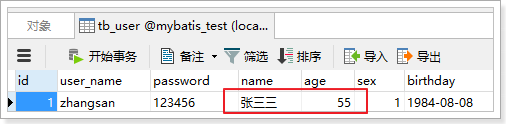
<filter-name>HttpMethodFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

再次测试：





#### 4.3.3.2方案二：

Put请求只接受json数据，请求的参数就用json

### 删除用户

#### 4.3.4.1.方案一：

/\*\*

\* 删除用户

\* **@param** user

\* **@return**

\*/

@RequestMapping(method=RequestMethod.***DELETE***)

**public** ResponseEntity<Void> deleteUser(@RequestParam("ids")Long[] ids){

**try** {

// 删除用户

**this**.userService.deleteUserByIds(ids);

// 成功，返回204 ,操作成功，但是不返回数据

**return** **new** ResponseEntity<Void>(HttpStatus.***NO\_CONTENT***);

} **catch** (Exception e) {

e.printStackTrace();

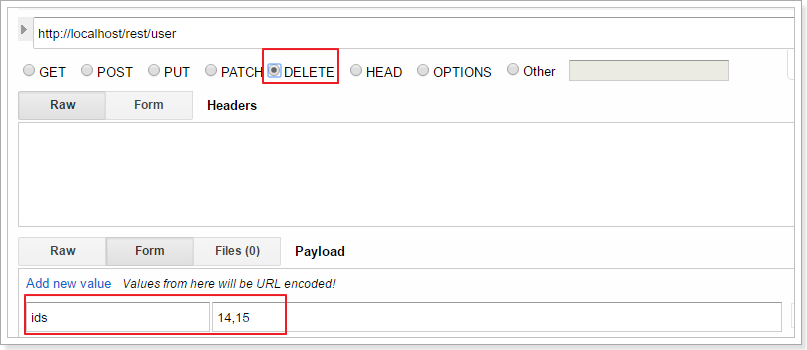
// 出现异常，返回500

**return** **new** ResponseEntity<>(HttpStatus.***INTERNAL\_SERVER\_ERROR***);

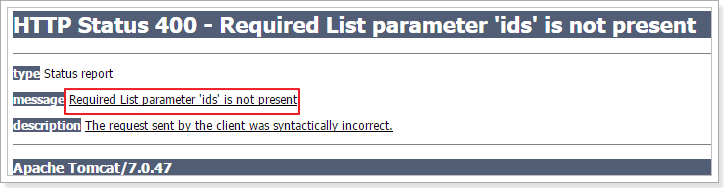
}

}

测试：



再次出错：



参数依然没有传递过来。也就是说，上面解决PUT请求的过滤器，对DELETE请求没用。

需要引入一个新的过滤器：

<!-- 通过POST请求传递参数，再 用\_method指定要转化的请求方式（DELETE或PUT），最后转为DELETE或PUT请求 -->

<filter>

<filter-name>HiddenHttpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

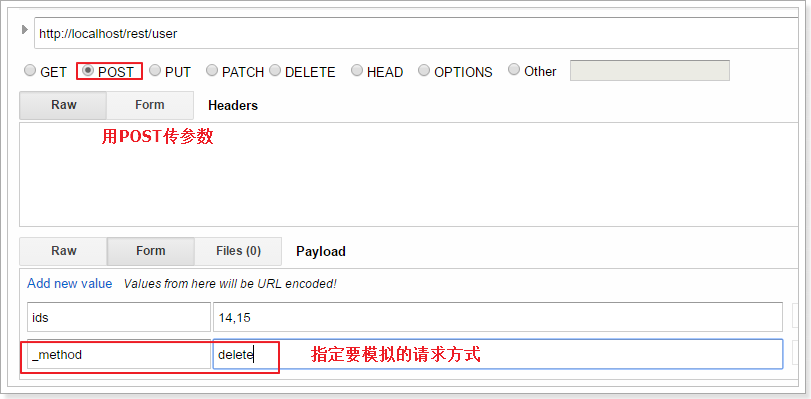
<url-pattern>/\*</url-pattern>

</filter-mapping>

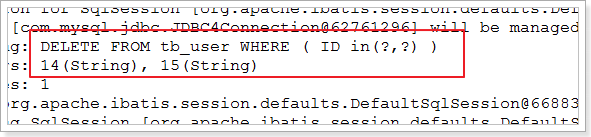
这个过滤器并不能真正的去解决DELETE请求问题，而是需要在页面发请求时，满足两点：

1. 依然发POST请求，
2. 同时传递\_method参数，指定要模拟的请求方式

然后过滤器通过POST接收参数，然后再把请求方式改为DELETE



成功接收：



#### 4.3.4.2方案二：

delete请求只接受json数据，请求的参数就用json

# 改造前端JS

## 查询用户列表首页：

### 定义分页查询controller

/\*\*

\* 查询全部用户

\* **@return**

\*/

@RequestMapping(method=RequestMethod.***GET***)

@ResponseBody

**public** ResponseEntity<DataGridResult<User>> queryUserByPage(

@RequestParam("page")Integer page,

@RequestParam("rows")Integer rows){

**try** {

// 查询用户

DataGridResult<User> result = **this**.userService.queryUserListByPage(page, rows);

**if**(result != **null**){

// 资源存在，返回200

**return** **new** ResponseEntity<>(result, HttpStatus.***OK***);

}

// 资源不存在，返回404

**return** **new** ResponseEntity<>(HttpStatus.***NOT\_FOUND***);

} **catch** (Exception e) {

e.printStackTrace();

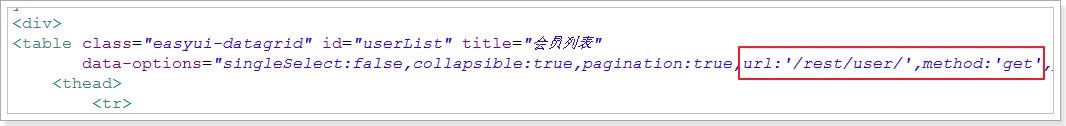
// 出现异常，返回500

**return** **new** ResponseEntity<>(HttpStatus.***INTERNAL\_SERVER\_ERROR***);

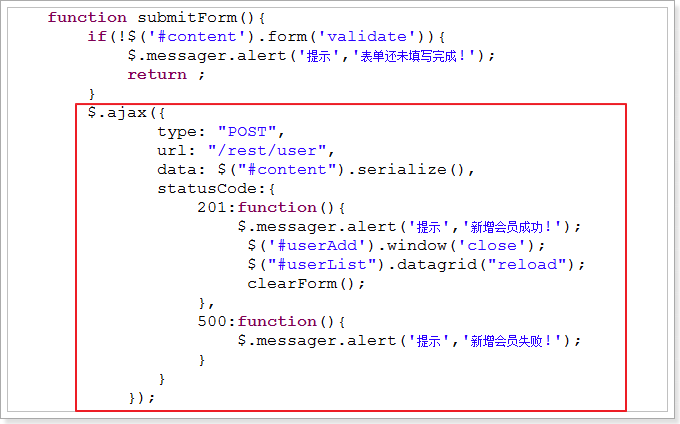
}

}

### 查询数据页面JS



## 新增用户JS



作业1：配置mybatis插件时，通用Mapper和分页助手的顺序是否可以调整？为什么？

作业2：实现修改和删除的JS改造为REST风格

## 删除用户JS

