

我的学习日志

目录

我的学习日志

目录

获取验证码 easy-captcha

Maven依赖

使用方法

MYSQL详解

注意规范

修改表

修改表名

增加表字段

修改表字段

删除表字段

MySql数据管理

外键(了解即可)

物理外键(不建议使用)

DML语言(全部记住)

添加

修改

删除

DQL查询语言

联表查询JoinON详解

自连接

分页和排序

子查询

常用函数

聚合函数及分组过滤

事务

1.事务简介

2.事务四大特征/原则

3.MYSQL 事务处理

索引

索引分类

索引使用

索引原则

权限管理和备份

用户管理

Mysql备份

数据库三大范式

Mybatis XML映射文件详解

基本SQL XML语句

Select语句 - 查询

Insert、Update、Delete 语句

Foreach语句

Sql语句

#{xxx}/\${xxx} - 字符串替换 语句

ResultMap - 结果映射

id & result

Association - 关联

Collection - 集合

Discriminator - 鉴定器

动态 SQL

if语句

choose、when、otherwise语句

trim、where、set语句

TK.mybatis框架使用

BaseMapper方法:

IdsMapper方法:

ConditionMapper方法:

JWT (JSON Web Token)跨域身份验证

结构解析

使用JWT

1.Maven依赖

2.利用数据,生成对应Token

3.使用对应Token,获取数据

SpringBoot+Mybatis整合

Maven依赖

操作方法

1.编写实体类

2.编写mapper接口

3.编写mapper xml映射文件

4.编写service接口

5.编写service实现类

6.编写controller文件

7.配置property文件(或者yaml文件)

Mybatis逆向工程

操作方法

一.Maven依赖

二.配置generatorConfig.xml文件

三.编写生成器

PageHelper 分页

Maven依赖

操作方法

- 1.配置文件
- 2.编写对应需要分页controller层方法

Redis基础学习

Linu下安装

服务器

打开服务器

访问服务器

基础知识

数据类型

1.String

SET命令

GET命令

GETSET命令

KEYS命令

STRLEN命令

MSET命令

2.Hash

HMSET命令

HGETALL命令

HGET命令

HDEL命令

HLEN命令

3.List

LPUSH命令

LRANGE命令

LSET命令

LLEN命令

LIINDEX命令

LPOP/RPOP命令

RPOPLPUSH命令

4.Set

SADD命令

SMEMBER命令

SCARD命令

SISMEMBER命令

SREM命令

SRANDMEMBER命令

对比命令

SDIFF命令

SINTER命令

SUNION命令

5.Zset

ZADD命令

ZSCARD命令

ZRANGE/ZREVRANGE命令

三种特殊数据类型

Geospatial – 地理位置

Geoadd – 添加地理位置

Geopos – 查询地理位置

Geodist – 计算地理之间距离

Georadius – 寻找附近(通过经纬度)

Georadiusbymember – 寻找附近(通过指定元素)

Hyperloglog数据结构 – 基数统计

Bitmaps – 位图(位存储)

发布订阅

SUBSCRIBE命令

PUBLISH命令

UNSUBSCRIBE命令

事务

事务执行步骤

事务监控

Redis的持久化

RDB(Redis DataBase)

Jedis

Maven依赖

编码测试

SpringBoot+Redis

Maven依赖

配置方法

1.配置properties文件(yaml文件)中的redis环境参数

2.配置RedisConfig类

3.配置RedisUtil类

4.创建实体类

5.在Controller中使用（读写）

MD5+Salt+Hash散列进行数据加密

主要方法

注册用户时

Shiro安全框架

主要功能

细分功能

Maven依赖

快速入门语句

SpringBoot继承Shiro

Maven依赖

创建Realm类

创建ShiroConfig配置类

Controller类

Shiro+Thymeleaf页面整合

常用标签：

The has Permission tag

The authenticated tag
The hasRole tag
权限、角色访问控制
方法一：直接在页面控制（以Thymeleaf为例）
方法二：Controller代码层中控制
方法三：代码注释控制
JAVA设计模式
OOP七大原则
单例模式
工厂模式

获取验证码 easy-captcha

Maven依赖

```
<!-- 验证码easy-captcha -->
<dependency>
    <groupId>com.github.whvcse</groupId>
    <artifactId>easy-captcha</artifactId>
    <version>${easy.captcha.version}</version>
</dependency>
```

使用方法

```
// 算数类型验证码
ArithmeticCaptcha captcha = new ArithmeticCaptcha(130, 48);
// 中文类型
ChineseCaptcha captcha = new ChineseCaptcha(130, 48);
```

```
// 几位数运算，默认为两位
captcha.setLen(2);
// 获取运算的公式：3+2=?
captcha.getArithmetixcString();
// 获取运算的结果：5
String value = captcha.text();
```

```
String key = UuidUtil.createUuid();
// 存入redis并设置过期时间为5分钟
RedisUtil.set(key, value, 600);
HashMap<String, String> captchaMap = new HashMap<String, String>(2);
captchaMap.put("captchaKey", key);
captchaMap.put("image", captcha.toBase64());
// 将key和验证码base64返回给前端
return Result.success(captchaMap);
```

MYSQL详解

注意规范

注意:所有的创建和删除操作尽量添加 **IF EXISTS** 语句进行判断,以免报错.

- ``:反引号,字段名必须使用它包裹;
- -- info :单行注释,注意其--后必须空出一格才可以.
- /* info */:多行注释.
- SQL关键语句大小写不敏感,但为了快速阅读以及排错,建议写小写.
- "" :引号,Default 默认语句和Comment 备注使用.

修改表

修改表名

```
ALTER TABLE 表名 RENAME AS 新表名 ;  
ALTER TABLE teacher RENAME AS student ;
```

增加表字段

```
ALTER TABLE 表名 ADD 字段名 列属性 ;  
ALTER TABLE teacher ADD age int(10) ;
```

修改表字段

```
ALTER TABLE 表名 MODIFY 字段名 新的列属性[] ; -- 只能修改字段列的属性以及约束,不能修改字段名  
ALTER TABLE teacher MODIFY age int(12) ;
```

```
ALTER TABLE 表名 CHANGE 字段名 新字段名 新的列属性[] ; -- 字段名及列属性都能修改  
ALTER TABLE teacher CHANGE age age1 int(13) ;
```

删除表字段

```
ALTER TABLE 表名 DROP 字段名 ;  
ALTER TABLE teacher DROP age ;
```

MySql数据管理

外键(了解即可)

物理外键(不建议使用)

方法一:创建表时,增加约束.

不能单独删除被外键关系的表.

方法二:ALTER TABEL 表名 ADD CONSTRAINT '约束名' FOREIGN KEY('列名') REFERENCES '表名'('列名')

DML语言(全部记住)

数据库意义:数据存储,数据管理

DML语言:数据操作语言

添加

```
-- 插入单行数据
INSERT INTO 表名 (字段1, 字段2, 字段3, ...) VALUES (值1, 值2, 值3, ...);
INSERT INTO 表名 VALUES (值1, 值2, 值3, ...); -- 必须输入表所有字段值, 并且位置一一
对应, 否则报错.
-- 插入多行数据
INSERT INTO 表名 (字段1, 字段2, 字段3, ...) VALUES (值1, 值2, 值3, ...), (值1, 值2, 值
3, ...), ...
```

修改

```
UPDATE `表名` SET `字段` = `值` WHERE 条件...
UPDATE `表名` SET `字段` = `值`; -- 无条件时默认修改所有列数据
UPDATE `表名` SET `字段` = `值`, `字段` = `值` WHERE 条件... -- 修改多个字段
值
```

删除

```
DELETE FROM 表名 WHERE 条件...
TRUNCATE 表名 -- 清空表所有数据
```

TRUNCATE删除所有数据时会将 自增字段 计数归零,而**DELETE** 则不会.

TRUNCATE删除所有数据不会影响事务.

DELETE删除问题: 重启数据库,在INNODB中,自增列从1开始(内存丢失).

在MyISAM中,自增列不会丢失计数.

DQL查询语言

DQL:Data query language -数据查询语言

Select完整的语法：

顺序很重要：

select 去重 要查询的字段 from 表 （注意：表和字段可以取别名）
xxx join 要连接的表 on 等值判断
where (具体的值，子查询语句)
Group By （通过哪个字段来分组）
Having （过滤分组后的信息，条件和 where是一样的，位置不同）
Order By .. （通过哪个字段排序） [升序/降序]
Limit startIndex, pagesize

SELECT语法

```
1 SELECT [ALL | DISTINCT]
2 { * | table.* | [table.field1[as alias1][,table.field2[as alias2]][,...]] }
3 FROM table_name [as table_alias]
4 [left | right | inner join table_name2] -- 联合查询
5 [WHERE ...] -- 指定结果需满足的条件
6 [GROUP BY ...] -- 指定结果按照哪几个字段来分组
7 [HAVING] -- 过滤分组的记录必须满足的次要条件
8 [ORDER BY ...] -- 指定查询记录按一个或多个条件排序
9 [LIMIT {[offset,]row_count | row_countOFFSET offset}];
10 -- 指定查询的记录从哪条至哪条
```

注意：[] 括号代表可选的，{} 括号代表必选得

SELECT * FROM 表名; -- 查询表中所有字段数据

SELECT 字段1, 字段2, 字段3, ... FROM 表名;

SELECT `字段1` AS 别名1, 字段2 AS 别名2, 字段3 AS 别名3, ... FROM 表名; -- 以别名查询出来字段数据

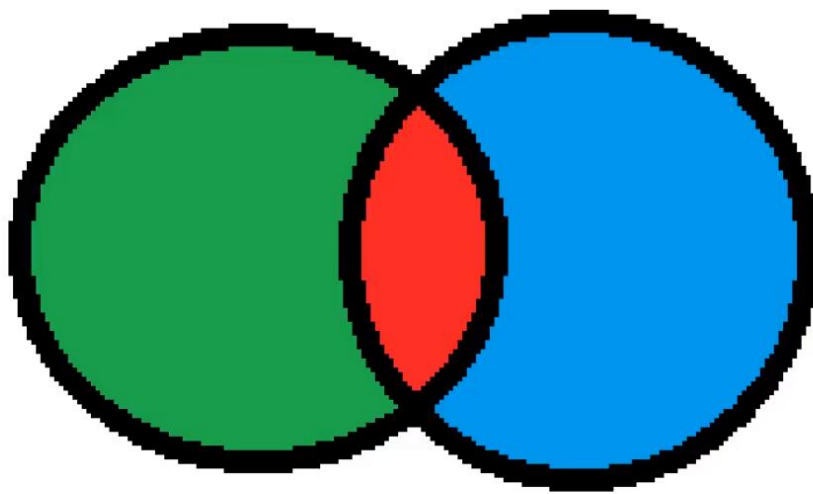
-- 函数Concat(a,b) 将a与b两结果想拼接

SELECT CONCAT('姓名', StudentName) AS 新名字 FROM student;

-- Distinct去重

SELECT DISTINCT 字段 FROM 表名;

联表查询JoinON详解



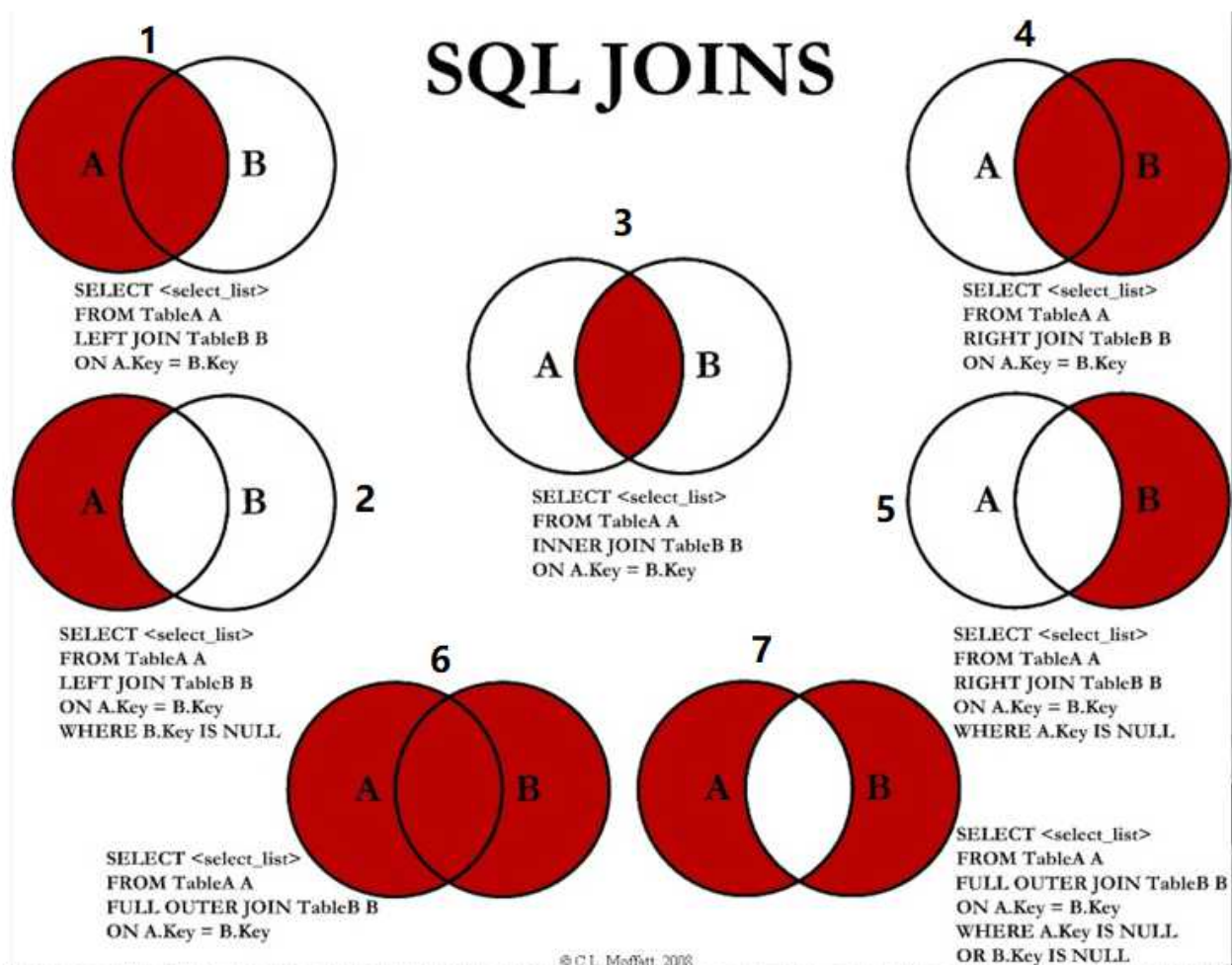
LeftJoin

左表

inner join

右表

Right Join



由于mysql中没有 full outer join 操作, 所以

mysql中6为1 union 4;

7为2 union 5;

思路:

1.分析需求,分析查询的字段来自那些表,(连接查询)

2.确定使用那种连接查询? 7种

确定交叉点(这两表那些数据是相同的)

判断的条件: 表1 字段1=表2 字段2

```
-- join on 连接查询
-- where 等值查询
SELECT s.studentNo,studentName,subjectNo,studentResult
FROM student (AS)s -- AS 可以省略
INNER JOIN/ LEFT JOIN/ RIGHT JOIN result (AS)r
WHERE/ ON s.studentNo=r.studentNo

SELECT s.studentNo,studentName,subjectName,studentResult
FROM student (AS)s -- AS 可以省略
RIGHT JOIN result (AS)r
ON s.studentNo=r.studentNo
INNER JOIN subject sub
ON r.subjectNO=sub.subjectNO
-- 连接查询可以重叠查询
```

操作	描述
inner join	如果表中至少有一个匹配,返回所有值
left join	即使右表中没有匹配的数据,也会从左表中返回
right join	即使左表中没有匹配的数据,也会从右表中返回

自连接

自己的表和自己的表连接, 核心:一张表拆为两条一样的表

<input type="checkbox"/>	categoryid	pid	▲ categoryName
<input type="checkbox"/>	7	5	ps技术
<input type="checkbox"/>	6	3	web开发
<input type="checkbox"/>	2	1	信息技术
<input type="checkbox"/>	8	2	办公信息
<input type="checkbox"/>	4	3	数据库
<input type="checkbox"/>	5	1	美术设计
<input type="checkbox"/>	3	1	软件开发

-- 把一张表看出两张一模一样的表

```
SELECT a.categoryName AS '父栏目', b.categoryName AS '子栏目'
FROM category AS a, category AS b
WHERE a.categoryid=b.pid
```

父栏目	子栏目
软件开发	数据库
软件开发	web开发
美术设计	ps技术
信息技术	办公信息

分页和排序

分页:limit 排序:order by

Order by:通过字段排序:升序 ASC ,降序 DESC

```
SELECT 字段1, 字段2, ...
FROM 表名
WHERE 条件
ORDER BY 字段 (ASC/ DESC)
```

Limit 起始值,显示个数(起始值首项为0)

```
SELECT 字段1, 字段2, ...
FROM 表名
WHERE 条件
Limit 0,5 -- 从第一条数据开始,显示5条数据
```

-- 设定每页显示5条数据

-- 第一页 limit 0,5 (1-1)*5

-- 第二页 limit 5,5 (2-1)*5

-- 第三页 limit 10,5 (3-1)*5

-- 第N页 limit (N-1)*PageSize, PageSize

-- [PageSize:页面大小, (N-1)*PageSize:起始值, N:当前页]

子查询

常用函数

ABS(-8) --绝对值 CEILING(9.4) --向上取整 FLOOR(9.4) --向下取整

RAND() --返回一个0-1之间的随机数 CHAR_LENGTH('scarf') --返回字符串的长度
CONCAT('2','3') --拼接字符串

REPLACE('2333','23','41') --替换指定字符串内容 SUBSTR('safer',1,3) --返回指定位置字符串(字符串,截取位置,截取长度)

聚合函数及分组过滤

GROUP BY 字段:通过字段来分组

COUNT():查询表中记录条数

COUNT(字段) -- 会忽略所有的NULL值

COUNT(*) -- 不会忽略NULL值,本质 是计算行数

COUNT(1) -- 不会忽略NULL值,本质是计算行数

SUM(字段):计算所有行总和

AVG(字段):计算所有行平均分

MAX(字段):查询所有行中最高分

MIN(字段):查询所有行中最低分

WHERE 条件中不能包含聚合函数.聚合函数过滤 需要使用 HAVING

```
SELECT SubjectName,AVG(studentResult) AS '平均分'
FROM result
GROUP BY SubjectNo
HAVING 平均分>80
```

事务

1.事务简介

- (1)在 MySQL 中只有使用了 InnoDB 数据库引擎的数据库或表才支持事务。
- (2)事务处理可以用来维护数据库的完整性,保证成批的 SQL 语句要么全部执行,要么全部不执行。
- (3)事务用来管理 insert,update,delete 语句。
- (4)Mysql自动默认开启事务自动提交。

```
SET autocommit = 0 /* 关闭 */
SET autocommit = 1 /* 开启(默认) */
```

2.事务四大特征/原则

一般来说，事务是必须满足4个条件（ACID）：：原子性（Atomicity，或称不可分割性）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。

原子性：一个事务（transaction）中的所有操作，要么全部完成，要么全部失败,不会只发生其中一个动作。

一致性：在事务开始结束前后数据结果要保证一定一致。

隔离性：数据库允许多个并发事务同时对其数据进行读写和修改的能力，多个并发事务不会互相影响。

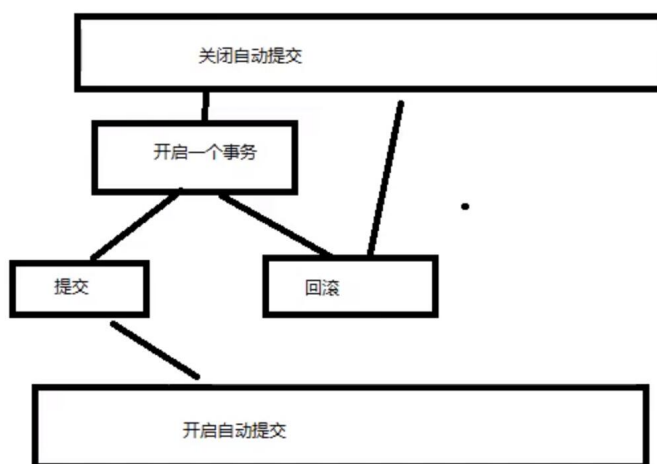
事务隔离分为不同级别：

- 脏读:指一个事务读取了另一个事务未提交的数据.
- 不可重复读:在一个事务读取数据时,多次读取结果不同.(不一定错误,只是场合不对)
- 幻读(虚读):在一个事务读取了别的事务插入的数据,导致前后读取不一致.

持久性：事务处理结束后的数据不会随外界原因而导致数据丢失,一旦事务提交不可逆。

3.MYSQL 事务处理

手动处理事务



0.关闭事务自动提交

```
SET autocommit = 0
```

1.事务开启

```
START TRANSACTION -- 标记一个事务的开始,从这之后的SQL语句都在一个事务内.
```

2.提交事务 :持久化 (成功的话)

```
COMMIT
```

3.回滚事务 :回到之前的样子(失败的话)

```
ROLLBACK
```

3.事务结束

```
SET TRANSACTION = 1
```

4.保存点

```
SAVEPOINT xxx1 (保存点名)      -- 设置一个叫xxx1的事务保存点
ROLLBACK TO SAVEPOINT xxx1 (保存点名)      -- 回滚到xxx1保存点

RELEASE SAVEPOINT xxx1 (保存点名)      -- 撤销xxx1保存点
```

索引

索引是帮助Mysql高效获取数据的数据结构.

但索引的缺点: 会降低更新表的速度, 如对表进行INSERT、UPDATE和DELETE。因为更新表时, MySQL不仅要保存数据, 还要保存一下索引文件。

索引分类

- 主键索引 (PRIMARY KEY)
 - 唯一标识,主键不可重复,只能有一个列为主键
- 唯一索引 (UNIQUE KEY)
 - 避免重复的列出现,唯一索引可以重复,多个列可以标识为 唯一索引
- 常规索引 (KEY/INDEX)
 - 默认的index,key关键字来设置
- 全文索引(FULL TEXT)
 - 在特定的数据库才有,MyISAM
- 快速定位数据

索引使用

-- 在创建表时给字段添加索引

-- 创建完毕后,添加索引

```
SHOW INDEX FROM 表名 -- 显示表中的所有索引
```

```
ALTER TABLE 表名 ADD FULLTEXT/UNIQUE 索引名 (字段名) -- 修改表结构,添加一个全文索引
```

```

CREATE INDEX/UNIQUE INDEX 索引名 ON 表名 (字段名) -- 在一个表中添加一个常规索引

/*在创建表时添加索引*/
CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

INDEX/UNIQUE [索引名] (username(length)) -- 如果是CHAR, VARCHAR类型, length
可以小于字段实际长度; 如果是BLOB和TEXT类型, 必须指定 length。

);

DROP INDEX [索引名] ON 表名; -- 删除索引

```

```

-- 插入100万条数据
CREATE FUNCTION mock_date()
Returns INT
DELIMITER $$ -- 写函数之前必须写, 标志
BEGIN
    DECLARE num INT DEFAULT 1000000;
    DECLARE i INT DEFAULT 0;
    WHILE i<num DO
        -- 插入语句
        INSERT INTO 表名(字段1, 字段2, ...) VALUES (CONCAT('值1', i), CONCAT('值
2', i), ...);
        SET i=i+1;
    END WHILE;
    RETURN i;
END;

```

索引原则

- 索引不是越多越好, 表中数据非常的时才考虑.
- 不要对进程变动数据加索引
- 小数据量的表不需要加索引
- 索引一般加载常用来查询的字段上!

索引的数据结构:

Hash 类型的索引

Btree :InnoDB默认默认索引类型

权限管理和备份

用户管理

用户表: mysql.user表

本质:对这张表进行增删改查

```
CREATE USER 用户名 IDENTIFIED BY '密码' -- 创建一个默认用户
SET PASSWORD= PASSWORD('密码') -- 修改当前用户密码
SET PASSWORD FROM 用户名= PASSWORD('密码') -- 修改指定用户密码

RENAME USER 用户名 TO 新用户名 -- 修改指定用户名
GRANT ALL PRIVILEGES ON *.* TO 用户名 -- 给指定用户授权所有权限(除了给别人授权)
在所有数据库中所有表
SHOW GRANT FOR 用户名 -- 查看指定用户权限
REVOKE ALL PRIVILEGES ON *.* FROM 用户名 -- 撤销指定用户所有权限在所有数据库中
所有表

DROP USER 用户名 -- 删除指定用户
```

Mysql备份

- 1.直接拷贝data文件夹下的物理文件
- 2.使用可视化工具手动导出
- 3.使用命令行导出 mysqldump 命令行

```
mysqldump -h`mysql地址` -u`用户名` -p`密码` 数据库名 表名 表2 表3 ... >物理磁盘
位置:/文件名.sql(导出地址)
```

数据库三大范式

为什么需要数据库范式化?

- 信息重复
- 更新异常
- 插入异常
 - 无法正常显示信息
- 删除异常
 - 丢失有效信息

第一范式(1NF)

原子性: 保证每列不可再分

第二范式(2NF)

前提:满足第一范式

每张表只描述一个事情

第三范式(3NF)

前提:满足第一范式和第二范式

确保数据表中每列数据和主键直接相关,而不能间接相关.

规范性和性能的问题:

关联查询的表不能成过三张表

- 考虑商业化需求和目标(成本,用户体验)数据库的性能更加重要
- 规范性能的问题的时候,需要适当考虑一下示范性
- 故意给某些表增加一些冗余的字段(从多表查询变单表查询)
- 故意增加一些计算列(大数据量降低为小数据量的查询:索引)

Mybatis XML映射文件详解

基本SQL XML语句

Select语句 -查询

```
<select
  id="selectPerson"
  parameterType="int"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
  ...
</select>
```

属性

描述

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterMap	将会传入这条语句的参数的类全限定名或别名。这个属性是可选的。
resultType	期望从这条语句中返回结果的类全限定名或别名。注意，如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身的类型。
resultMap	对外部 resultMap 的命名引用。 resultType 和 resultMap 之间只能同时使用一个。
flushCache	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空。
useCache	将其设置为 true 后，将会导致本条语句的结果被二级缓存缓存起来。
timeout	抛出异常之前，驱动程序等待数据库返回请求结果的秒数。

Insert、Update、Delete 语句

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数的类全限定名或别名。这个属性是可选的。
flushCache	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空，默认值：（对 insert、update 和 delete 语句）true。
useGeneratedKeys	支持自动生成主键的字段，需要再填写keyProperty 设置（仅适用于 insert 和 update）
keyProperty	指定能够唯一识别对象的属性（仅适用于 insert 和 update）

Foreach语句

```
<foreach item="item" collection="list" separator=",">
    ({item.username}, {item.password}, {item.email}, {item.bio})
</foreach>
```

- **item**: 集合中元素迭代时的别名, 该参数为必选。
- **index**: 在list和数组中,index是元素的序号, 在map中, index是元素的key, 该参数可选
- **open**: foreach代码的开始符号, 一般是(和close=")"合用。常用在in(),values()时。该参数可选
- **separator**: 元素之间的分隔符, 例如在in()的时候, separator=","会自动在元素中间用“,”隔开, 避免手动输入逗号导致sql错误, 如in(1,2,)这样。该参数可选。
- **collection**: 要做foreach的对象, 作为入参时, List对象默认用"list"代替作为键, 数组对象有"array"代替作为键, Map对象没有默认的键。当然在作为入参时可以使用@Param("keyName")来设置键, 设置keyName后, list,array将会失效。

Sql语句

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password
</sql>
```

这个 SQL 片段可以在其它语句中使用。

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t1"/>
  </include>,
    <include refid="userColumns"><property name="alias" value="t2"/>
  </include>
  from some_table t1
    cross join some_table t2
</select>
```

#{xxx}/\${xxx}-字符串替换 语句

使用 **#{}** 参数语法, 会在 SQL 语句中直接插入一个转义的字符串。更安全, 更迅速, 通常也是首选做法。

使用 **\${}** 参数语法, 直接会在 SQL 语句中直接插入一个不转义的字符串。但用这种方式接受用户的输入, 并用作语句参数是不安全的, 会导致潜在的 SQL 注入攻击。

```
#{property,javaType=int,jdbcType=NUMERIC}
```

和MyBatis 的其它部分一样, 几乎总是可以根据参数对象的类型确定 javaType, 除非该对象是一个 HashMap。这个时候, 你需要显式指定 javaType 来确保正确的类型处理器 (TypeHandler) 被使用。

JDBC 要求, 如果一个列允许使用 **null** 值, 并且会使用值为 **null** 的参数, 就必须指定 JDBC 类型 (jdbcType)。

对于数值类型，还可以设置 `numericScale` 指定小数点后保留的位数。

当 SQL 语句中的元数据（如表名或列名）是动态生成的时候，字符串替换将会非常有用。举个例子，如果你想 `select` 一个表任意一列的数据时，不需要这样写：

```
@Select("select * from user where id = #{id}")
User findById(@Param("id") long id);

@Select("select * from user where name = #{name}")
User findByName(@Param("name") String name);

@Select("select * from user where email = #{email}")
User findByEmail(@Param("email") String email);

// 其它的 "findByXxx" 方法
...
```

而是可以只写这样一个方法：

```
@Select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value") String
value);
```

其中 `${column}` 会被直接替换，而 `#{value}` 会使用 `?` 预处理。

ResultMap - 结果映射

`ResultMap` 元素是 `MyBatis` 中最重要最强大的元素。在为一些比如连接的复杂语句编写映射代码的时候，一份 `ResultMap` 能够代替实现同等功能的数千行代码。其设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。

```
<!-- 非常复杂的结果映射 -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection"
column="author_favourite_section"/>
  </association>
</resultMap>
```

```

</association>
<collection property="posts" ofType="Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <association property="author" javaType="Author"/>
  <collection property="comments" ofType="Comment">
    <id property="id" column="comment_id"/>
  </collection>
  <collection property="tags" ofType="Tag" >
    <id property="id" column="tag_id"/>
  </collection>
  <discriminator javaType="int" column="draft">
    <case value="1" resultType="DraftPost"/>
  </discriminator>
</collection>
</resultMap>

```

`constructor` - 用于在实例化类时，注入结果到构造方法中

- `idArg` - ID 参数；标记出作为 ID 的结果可以帮助提高整体性能
- `arg` - 将被注入到构造方法的一个普通结果

`id` - 一个 ID 结果；标记出作为 ID 的结果可以帮助提高整体性能

`result` - 注入到字段或 `JavaBean` 属性的普通结果

`association` - 一个复杂类型的关联；许多结果将包装成这种类型

- 嵌套结果映射 - 关联可以是 `resultMap` 元素，或是对其它结果映射的引用

`collection` - 一个复杂类型的集合

- 嵌套结果映射 - 集合可以是 `resultMap` 元素，或是对其它结果映射的引用

`discriminator` - 使用结果值来决定使用哪个 `resultMap`

- `case` - 基于某些值的结果映射
- 嵌套结果映射 - `case` 也是一个结果映射，因此具有相同的结构和元素；或者引用其它的结果映射

id & result

```

<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>

```

这些元素是结果映射的基础。`id` 和 `result` 元素都将一个列的值映射到一个简单数据类型 (`String`, `int`, `double`, `Date` 等) 的属性或字段。

属性	描述
property	映射到列结果的字段或属性。如果 <code>JavaBean</code> 有这个名称的属性（ <code>property</code> ），会先使用该属性。否则 <code>MyBatis</code> 将会寻找给定名称的字段（ <code>field</code> ）。
column	数据库中的列名，或者是查询、修改、删除时的列的别名。
javaType	一个 <code>Java</code> 类的全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。如果你映射到一个 <code>JavaBean</code> ， <code>MyBatis</code> 通常可以推断类型。
jdbcType	<code>JDBC</code> 类型，所支持的 <code>JDBC</code> 类型参见这个表格之后的“支持的 <code>JDBC</code> 类型”。只需要在可能执行插入、更新和删除的且允许空值的列上指定 <code>JDBC</code> 类型。

Association -关联

关联（`association`）元素处理“有一个”类型的关系。比如，在我们的示例中，一个博客有一个用户。关联结果映射和其它类型的映射工作方式差不多。你需要指定目标属性名以及属性的 `javaType`（很多时候 `MyBatis` 可以自己推断出来），在必要的情况下你还可以设置 `JDBC` 类型。

关联的不同之处是，你需要告诉 `MyBatis` 如何加载关联。`MyBatis` 有两种不同的方式加载关联：

1. 嵌套 `Select` 查询：通过执行另外一个 `SQL` 映射语句来加载期望的复杂类型。
 2. 嵌套结果映射：使用嵌套的结果映射来处理连接结果的重复子集。
- 关联的嵌套 `Select` 查询

属性	描述
column	数据库中的列名，或者是查询、修改、删除时的列的别名。
select	用于加载复杂类型属性的映射语句的 <code>ID</code> 。

```

<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author"
select="selectAuthor"/>
</resultMap>

<select id="selectAuthor" resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
</select>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

```

两个 **select** 查询语句：一个用来加载博客（Blog），另外一个用来加载作者（Author），而且博客的结果映射描述了应该使用 `selectAuthor` 语句加载它的 **author** 属性。

方式虽然很简单，但这个方法会导致成百上千的 SQL 语句被执行。影响SQL性能。

- 关联的嵌套结果映射

```

<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id"
javaType="Author" resultMap="authorResult"/>
</resultMap>

<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>

```

上面的示例使用了外部的结果映射元素来映射关联。这使得 **Author** 的结果映射可以被重用。博客（Blog）作者（author）的关联元素委托名为“**authorResult**”的结果映射来加载作者对象的实例。

id 元素在嵌套结果映射中扮演着非常重要的角色。你应该总是指定一个或多个可以唯一标识结果的属性。

也可以所有的结果映射放在一个具有描述性的结果映射元素中。直接将结果映射作为子元素嵌套在内。

```

<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
  </association>
</resultMap>

```

Collection -集合

```

<collection property="posts" ofType="domain.blog.Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <result property="body" column="post_body"/>
</collection>

```

一个博客 (Blog) 只有一个作者 (Author)。但一个博客有很多文章 (Post)。

```
private List<Post> posts;
```

映射嵌套结果集合到一个 List 中，可以使用集合元素。和关联元素一样，可以使用嵌套 Select 查询，或基于连接的嵌套结果映射集合。

- 集合的嵌套 Select 查询

```

<resultMap id="blogResult" type="Blog">
  <collection property="posts" javaType="ArrayList" column="id"
ofType="Post" select="selectPostsForBlog"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectPostsForBlog" resultType="Post">
  SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>

```

在一般情况下，MyBatis 可以推断 javaType 属性，因此并不需要填写。

- 集合的嵌套结果映射


```

<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post" resultMap="blogPostResult"
columnPrefix="post_" />
</resultMap>

<resultMap id="blogPostResult" type="Post">
  <id property="id" column="id"/>
  <result property="subject" column="subject"/>
  <result property="body" column="body"/>
</resultMap>

```

其中columnPrefix属性其含义是将XXX自动添加到它下面的column中。

Discriminator - 鉴定器

一个数据库查询可能会返回多个不同的结果集（但总体上还是有一定的联系的）。鉴别器（discriminator）元素就是被设计来应对这种情况的，另外也能处理其它情况，例如类的继承层次结构。鉴别器的概念很好理解——它很像 Java 语言中的 switch 语句。

```

<resultMap id="vehicleResult" type="Vehicle">
  ...
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultMap="carResult"/>
    <case value="2" resultMap="truckResult"/>
    <case value="3" resultMap="vanResult"/>
    <case value="4" resultMap="suvResult"/>
  </discriminator>
</resultMap>

<resultMap id="carResult" type="Car">
  <result property="doorCount" column="door_count" />
</resultMap>

```

动态 SQL

MyBatis 3 替换了之前的大部分元素，大大精简了元素种类，现在要学习的元素种类比原来的一半还要少。

if 语句

```

<if test="title != null">
  XXX
</if>

```

如果传入了“title”参数，那么就会额外输出XXX内容。

choose、when、otherwise语句

```
<choose>
  <when test="title != null">
    XXX
  </when>
  <when test="author != null and author.name != null">
    XXX
  </when>
  <otherwise>
    XXX
  </otherwise>
</choose>
```

MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

trim、where、set语句

```
<where>
  <if test="state != null">
    XXX
  </if>
  <if test="title != null">
    XXX
  </if>
  <if test="author != null and author.name != null">
    XXX
  </if>
</where>
```

TK.mybatis框架使用

```

<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper-spring-boot-starter</artifactId>
    <version>2.0.3-beta1</version>
</dependency>

<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
    <version>4.0.0</version>
</dependency>

```

创建BaseMapper接口继承Mapper, ConditionMapper

```

public interface BaseMapper<T> extends Mapper<T>, ConditionMapper<T> {
    ...
}

```

创建对应Dao层接口,例如类的Dao

```

public interface SysUserMapper extends BaseMapper<SysUser> {
    ....
}

```

BaseMapper方法:

```

/**
 * 保存一个实体, null属性也会保存
 *
 * @param record
 * @return
 */
int insert(T record);

/**
 * 保存一个实体, null属性不会保存
 *
 * @param record
 * @return
 */
int insertSelective(T record);

/**
 * 根据实体属性作为条件进行删除, 查询条件使用等号
 */
int delete(T record);

/**

```

```

    * 根据主键更新属性不为null的值
    */
int updateByPrimaryKeySelective(T record);

/**
    * 根据实体中的属性值进行查询，查询条件使用等号
    */
List<T> select(T record);

/**
    * 查询全部结果，select(null)方法能达到同样的效果
    */
List<T> selectAll();

/**
    * 根据实体中的属性进行查询，只能有一个返回值，有多个结果是抛出异常，查询条件使用等号
    */
T selectOne(T record);

/**
    * 根据实体中的属性查询总数，查询条件使用等号
    */
int selectCount(T record);

```

IdsMapper方法:

```

/**
    * 根据主键@Id进行查询，多个Id以逗号,分割
    * @param id
    * @return
    */
List<T> selectByIds(String ids);

/**
    * 根据主键@Id进行删除，多个Id以逗号,分割
    * @param id
    * @return
    */
int deleteByIds(String ids);

```

ConditionMapper方法:

```

/**
    * 根据Condition条件进行查询
    */
public List selectByCondition(Object condition);

```

```

/**

```

```
    * 根据Condition条件进行查询
    */
    public int selectCountByCondition(Object condition);

    /**
     * 根据Condition条件删除数据，返回删除的条数
     */
    public int deleteByCondition(Object condition);

    /**
     * 根据Condition条件更新实体`record`包含的全部属性，null值会被更新，返回更新的条数
     */
    public int updateByCondition(T record, Object condition);

    /**
     * 根据Condition条件更新实体`record`包含的全部属性，null值会被更新，返回更新的条数
     */
    public int updateByConditionSelective(T record, Object condition);
```

其中传入的Object condition应为tk.mybatis.mapper.entity.Condition

JWT (JSON Web Token)跨域身份验证

结构: *heard.payload.singnature*

标头.有效负载 签名

结构解析

- 1.标头 (heard)：包含令牌的类型以及使用的签名算法。
- 2.有效负载 (payload)：包含实体和其他数据的声明。尽管它可以防止被篡改，但任何人依然可以读取这些签名的信息。除非加密，否则不要在 JWT 的有效载荷或头部元素中放入秘密信息。
- 3.签名 (singnature)：签名用于验证消息是否在整个过程中被更改。对于使用私钥签名的令牌，它还可以验证 JWT 的发送方是否是它所说的那个发送方。

使用JWT

1.Maven依赖

```
<dependency>
    <groupId>com.auth0</groupId>
    <artifactId>java-jwt</artifactId>
    <version>3.11.0</version>
</dependency>
```

2.利用数据，生成对应Token

```
Calendar instance = Calendar.getInstance();
instance.add(Calendar.SECOND, 20);

String token = JWT.create()
    .withHeader(map) //设置header
    .withClaim("userId", 21) //设置payload
    .withClaim("userName", "xxx")
    .withExpiresAt(instance.getTime()) //指定令牌过期时间
    .sign(Algorithm.HMAC256("@QWER@")); //设置signature 其中使用
HMAC256算法
```

3.使用对应Token，获取数据

```
JWTVerifier jwtVerifier =
JWT.require(Algorithm.HMAC256("@QWER@")).build(); //获取对应Jwt算法解析器

//解析对应Token，获得解码后的Jwt
DecodedJWT decodedJWT =
jwtVerifier.verify("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9" +

    ".eyJ1c2VybmFtZSI6ImV4cCI6MTYwMzc2NTU4MCwidXNlcklkIjoyMX0" +
    ".cHJX4xoGPFBI8Qgk6me_44F1y_lunIDS9V0DJfLudw8");

//获取对应Token中数据
System.out.println(decodedJWT.getClaim("userId").asInt());
System.out.println(decodedJWT.getClaim("userName").asString());
```

SpringBoot+Mybatis整合

Maven依赖

```
<!--mysql数据库驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

<!--mybatis-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.0</version>
</dependency>
```

操作方法

application.yml文件配置

```
spring:
  datasource:
    username: root
    password: 123456
    #?serverTimezone=UTC解决时区的报错
    url: jdbc:mysql://localhost:3306/springboot?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    driver-class-name: com.mysql.cj.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource

    #Spring Boot 默认是不注入这些属性值的，需要自己绑定
    #druid 数据源专有配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true

    #配置监控统计拦截的filters，stat:监控统计、log4j: 日志记录、wall: 防御sql注入
    #如果允许时报错 java.lang.ClassNotFoundException:
org.apache.log4j.Priority
    #则导入 log4j 依赖即可，Maven 地址：
https://mvnrepository.com/artifact/log4j/log4j
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties:
druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

1.编写实体类

```
@Table(name = "SYS_ORDER")    //声明此对象映射到数据库的数据表
public class SysOrder implements Serializable {
    private static final long serialVersionUID = 2688088497753868718L;

    @Id    //主键字段
    private String merchantOrderNo;
    private String merchantNo;
    private Integer amount;
```

```

...

public String getMerchantOrderNo() {
    return merchantOrderNo;
}

public void setMerchantOrderNo(String merchantOrderNo) {
    this.merchantOrderNo = merchantOrderNo;
}

public String getMerchantNo() {
    return merchantNo;
}

public void setMerchantNo(String merchantNo) {
    this.merchantNo = merchantNo;
}

...
}

```

2.编写mapper接口

```

@Mapper //指定这是一个操作数据库的mapper
public interface SysOrderMapper extends BaseMapper<SysOrder> {

    BizDataDTO getBizDataByOrderNo(@Param("merchantOrderNo") String
merchantOrderNo);

    List<SysOrderDTO> queryOrderData(@Param("orderQueryRequest")
OrderQueryRequest orderQueryRequest);

    SysOrderDTO getOrderByOrderNo(@Param("merchantOrderNo") String
merchantOrderNo);

    ...
}

```

3.编写mapper xml映射文件

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.com.bossoftcq.ipp.mapper.SysOrderMapper">
    <resultMap id="BaseResultMap"
type="cn.com.bossoftcq.ipp.model.doo.SysOrder">
        <id column="MERCHANT_ORDER_NO" jdbcType="VARCHAR"
property="merchantOrderNo" />
        <result column="MERCHANT_NO" jdbcType="VARCHAR"
property="merchantNo" />
        <result column="AMOUNT" jdbcType="DECIMAL" property="amount" />
    </resultMap>

```



```

</resultMap>
...
<select id="getOrderByOrderNo"
        parameterType="java.lang.String"
resultMap="orderWithBizList">
    SELECT
    so.*,bi.biz_ID,bi.sys_type
    FROM
    sys_order so,sys_order_biz bi
    where
    so.merchant_order_no = bi.merchant_order_no
    and so.MERCHANT_ORDER_NO = #{merchantOrderNo,jdbcType=VARCHAR}
</select>
...
</mapper>

```

4.编写service接口

```

public interface SysOrderService extends BaseService<SysOrder>{

    PageInfoBO queryOrderData (OrderQueryRequest orderQueryRequest);

    Result cancelOrderByMerchantOrderNo (String merchantOrderNo);

    ...

}

```

5.编写service实现类

```

@Service //需要在接口实现类中使用@Service注解，才能被SpringBoot扫描
public class SysOrderServiceImpl extends BaseServiceImpl<SysOrder>
implements SysOrderService {

    private static final Logger log =
LoggerFactory.getLogger (SysOrderServiceImpl.class);

    @Autowired
    private SysOrderMapper orderMapper;
    @Autowired
    private EduBizSplitDataMapper eduBizSplitDataMapper;
    ...
    @Override
    public PageInfoBO queryOrderData (OrderQueryRequest
orderQueryRequest) {
        //如果是super则查询所有，否则上查下
        UserDTO user = UserUtil.getUser();
        if (user.isSuperAdmin()) {
            orderQueryRequest.setDeptId(null);
        }else{

```

```

        orderQueryRequest.setDeptId(user.getDeptId());
    }

    PageHelper.startPage(orderQueryRequest.getPage(), orderQueryRequest.getLimit());

    List<SysOrderDTO> orders =
orderMapper.queryOrderData(orderQueryRequest);
    if (!orders.isEmpty()) {
        PageInfo<SysOrderDTO> pageInfo = new PageInfo<>(orders);
        return new PageInfoBO(pageInfo.getTotal(),
pageInfo.getList());
    }
    return null;
}
...
}

```

6.编写controller文件

```

@RestController
public class OrderController {

    private static final Logger LOGGER =
LoggerFactory.getLogger(OrderController.class);

    @Autowired
    private SysOrderService sysOrderService;

    @GetMapping("/order/queryData")
    public Result queryOrderData(OrderQueryRequest orderQueryRequest) {
        return
Result.success(sysOrderService.queryOrderData(orderQueryRequest));
    }

}

```

7.配置property文件(或者yaml文件)

```

mybatis:
    # Mybatis配置Mapper路径
    mapper-locations: classpath:mapping/*.xml
    # Mybatis配置Model类对应
    type-aliases-package: cn.com.bossoftcq.ipp.model

```

Mybatis逆向工程

操作方法

一.Maven依赖

```
<dependency>
  <groupId>org.mybatis.generator</groupId>
  <artifactId>mybatis-generator-core</artifactId>
  <version>1.3.7</version>
</dependency>
```

二.配置generatorConfig.xml文件

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE generatorConfiguration
  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration
  1.0//EN"
  "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
  <context id="sakila" targetRuntime="MyBatis3">
    <!-- 生成的Java文件的编码 -->
    <property name="javaFileEncoding" value="UTF-8" />
    <!-- 格式化java代码 -->
    <property name="javaFormatter"
      value="org.mybatis.generator.api.dom.DefaultJavaFormatter"
    />

    <!-- 格式化XML代码 -->
    <property name="xmlFormatter"
      value="org.mybatis.generator.api.dom.DefaultXmlFormatter"
    />

    <commentGenerator>
      <!-- 是否去除自动生成的注释 true: 是 : false:否 -->
      <property name="suppressAllComments" value="true" />
      <!-- 是否生成注释代时间戳 -->
      <property name="suppressDate" value="true" />
    </commentGenerator>

    <!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->
    <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
      connectionURL="jdbc:mysql://localhost:3306/23?
      useSSL=false&useUnicode=false&characterEncoding=UTF8"
      userId="root"
      password="200428">
    </jdbcConnection>
```

```

<javaTypeResolver>
    <!-- 把jdbc对应的日期类型转成java8中的LocateDateTime类型 -->
    <property name="useJSR310Types" value="true"/>
</javaTypeResolver>

<!-- targetProject:生成PO类的位置 -->
<javaModelGenerator
targetPackage="com.example.seconddemowebrestfulcrud.entity"
    targetProject=".\\src\\main\\java">
    <!-- 从数据库返回的值被清理前后的空格 -->
    <property name="trimStrings" value="true"/>
</javaModelGenerator>

<!-- targetProject:mapper映射文件生成的位置 -->
<sqlMapGenerator targetPackage="mapper"
    targetProject=".\\src\\main\\resources"/>

<!-- targetPackage: mapper接口生成的位置 -->
<javaClientGenerator type="XMLMAPPER"

targetPackage="com.example.seconddemowebrestfulcrud.mapper"
    targetProject=".\\src\\main\\java"/>

<!-- 指定数据库表 -->
<table schema="23" tableName="user_info"
    domainObjectName="User"
    enableCountByExample="false"
    enableDeleteByExample="false" enableSelectByExample="false"
    enableUpdateByExample="false">
    <!-- 如果设置为true, 生成的entity类会直接使用column本身的名字, 而不会再
使用驼峰命名方 -->
    <property name="useActualColumnNames" value="false" />
    <!-- 生成的SQL中的表名将不会包含schema和catalog前缀 -->
    <property name="ignoreQualifiersAtRuntime" value="true" />
</table>
</context>
</generatorConfiguration>

```

三.编写生成器

```

public class Generator {

    public void generator() throws Exception {
        List<String> warnings = new ArrayList<String>();
        boolean overwrite = true;
        /**指向逆向工程配置文件*/
        File configFile = new File("D:\\first-app-demo\\second-demo-web-
restfulcrud\\src\\main\\resources\\generator" +

```

```

        "\\generatorConfig.xml");
    ConfigurationParser parser = new ConfigurationParser(warnings);
    Configuration config = parser.parseConfiguration(configFile);
    DefaultShellCallback callback = new
DefaultShellCallback(overwrite);
    MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,
        callback, warnings);
    myBatisGenerator.generate(null);
}

    public static void main(String[] args) throws Exception {
        try {
            Generator generatorSqlmap = new Generator();
            generatorSqlmap.generator();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

最后，运行其中main方法即可生成POJO实体类、Mapper接口、Xml映射文件。

PageHelper 分页

Maven依赖

```

<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.3.0</version>
</dependency>

```

操作方法

1.配置文件

```

pagehelper:
    # 指定分页插件使用哪种方言
    helperDialect: mysql
    # 分页合理化，默认值为false。当该参数设置为 true 时，pageNum<=0 时会查询第一页，
    pageNum>pages，会查询最后一页。
    reasonable: true
    ...

```

2.编写对应需要分页controller层方法

```
/**
 * 会员管理
 */
@RequestMapping("/manageMember")
public String manageMember(@RequestParam(defaultValue = "1") int
pageNum,
                           @RequestParam(defaultValue = "10") int
pageSize,
                           Model model) {

    // 设置查询分页大小
    PageHelper.startPage(pageNum, pageSize);
    // 装入PageInfo容器中
    PageInfo pageInfo=new
PageInfo(adminMemberService.selectAllUser());
    model.addAttribute("pageInfo", pageInfo);
    return "adminMemberManage";
}

public PageInfoBO queryOrderData(OrderQueryRequest orderQueryRequest)
{
    //如果是super则查询所有，否则上查下
    UserDTO user = UserUtil.getUser();
    if (user.isSuperAdmin()) {
        orderQueryRequest.setDeptId(null);
    } else {
        orderQueryRequest.setDeptId(user.getDeptId());
    }

    PageHelper.startPage(orderQueryRequest.getPage(), orderQueryRequest.getLi
mit());
    //查询数据
    List<SysOrderDTO> orders =
orderMapper.queryOrderData(orderQueryRequest);
    if (!orders.isEmpty()) {
        PageInfo<SysOrderDTO> pageInfo = new PageInfo<>(orders);
        // pageInfo.getTotal : 获取总条数
        //pageInfo.getList : 获取数据
        return new PageInfoBO(pageInfo.getTotal(),
pageInfo.getList());
    }
    return null;
}
}
```

Redis基础学习

什么是NoSql

NoSql=not only Sql

泛指非关系型数据库,web2.0互联网诞生,传统的关系型数据库难以对付Web2.0时代.尤其超大规模的高并发社区.

关系型数据库:表格,行列记录.(POI)

NoSQL的特点:

- 1.方便扩展(数据之间没有关系)
- 2.大数据量高性能(Redis 一秒写8万次,读取11万)

Redis是开源的一个内存数据库,数据保存在内存中,但是我们都知内存的数据变化是很快的,也容易发生丢失。

幸好Redis还为我们提供了持久化的机制,分别是RDB(Redis DataBase)和AOF(Append Only File)。

Linu下安装

一.将安装文件包复制到 opt 文件夹下解压

二.安装C++环境包

```
yum install gcc-c++
```

三.进入redis文件夹 运行 make 命令 ,然后在运行 make install 命令.

Redis默认安装路径 'usr/local/bin/'

四.将Redis配置文件复制到Redis默认安装路径usr/local/bin/下. 之后就使用这个作为配置文件启动.

五.Redis默认不是后台启动的.需要修改配置文件

```
daemonize yes
```

六.启动Redis服务.

七.查看redis服务打开端口状态

```
ps -ef|grep redis
```

服务器

打开服务器

```
redis-server.exe (/.../redis.conf)
```

访问服务器

```
redis-cli.exe -h 服务器IP -p 端口号 (默认6379)
```

基础知识

Redis默认有16个数据库

默认使用第0个数据库. 可以使用select进行切换数据库

```
select (2) # 切换数据库

dbsize # 查看数据库大小

flushdb # 清空当前数据库
flushall # 清空全部数据库

exists (key) #判断是否存在key, 返回0,1
exists (key) (数据库名) #移动key到指定数据库中, 返回0,1

expire (key) (秒数) #设置key的过期时间, 返回0,1 key过期后会自动为null
```

Redis是单线程(Redis6.0支持多线程).Redis是基于 内存操作,CPU不是Redis的性能瓶颈,Redis的瓶颈是根据机器的内存和网络带宽.

Redis将所有的数据放在内存中的,所有说使用单线程性能最高.

数据类型

Redis支持5类数据类型:

String (字符串)

hash (散列)

list (列表)

set (集合)

zset（有序集合）

1.String

string 是 redis 最基本的类型，一个 key 对应一个 value。

string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据。比如jpg图片或者序列化的对象。

string 类型的单个值最大能存储 **512MB**。

```
set user:1:xx {name=aa,age=32,emil=231432@qq.com}
```

SET命令

设置指定 key 的值

```
SET -key- -value-
```

GET命令

获取指定 key 的值

```
GET -key-
```

GETSET命令

将给定 key 的值设为 value ，并返回 key 的旧值

```
GETSET -key- -value-
```

KEYS命令

查询当前数据库所有key

```
KEYS *
```

STRLEN命令

返回 key 所储存的字符串值的长度

```
STRLEN -key-
```

MSET命令

同时设置一个或多个 key-value 对

```
MSET -key1- -value1- -key2- -value2- ...
```

2.Hash

hash 哈希 是一个键值(key=>value)对集合，string 类型的 field 和 value 的映射表。

hash 特别适合用于存储对象。

每个 hash 可以存储 $2^{32} - 1$ 键值对（40多亿）。

HMSET命令

为指定key设置hash表

```
HMSET -key- -field1- -value1- -field2- -value2- ...
```

HGETALL命令

遍历整个key的hash表内容

```
HGETALL -key-
```

HGET命令

获取某个key中指定hash表内容

```
HGET -key- -field-
```

HDEL命令

删除整个key中的某个field以及对应value

```
HDEL -key- -field-
```

HLEN命令

获取hash表中字段的数量

```
HLEN -key-
```

3.List

List列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

LPUSH命令

将一个或多个值插入到列表头部。

```
LPUSH -key- -value1- -value2- ...
```

LRANGE命令

根据指定长度获取对应list中值

```
LRANGE key -start index- -stop index-
```

LSET命令

通过索引来设置元素的值。

```
LSET -key- -index- -value-
```

LLEN命令

获取列表长度。

```
LLEN -key-
```

LIINDEX命令

通过索引获取列表中的元素。

```
LINDEX -key- -index-
```

LPOP/RPOP命令

删除最后一个元素/删除第一个元素

```
LPOP -key-  
RPOP -key-
```

RPOPLPUSH命令

删除第一个元素并将其添加到另一个List列表中

```
RPOPLPUSH -key- -other key-
```

4.Set

Set 是 string 类型的无序集合。

Set不允许内容重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 0或者1

SADD命令

添加一个 string 元素到 key 对应的 set 集合中，成功返回 1，如果元素已经在集合中返回 0。

```
SADD -key- -value1- -value2- ...
```

SMEMBER命令

返回集合中的所有的成员。

```
SMEMBER -key-
```

SCARD命令

命令返回集合中元素的数量。

```
SCARD -key-
```

SISMEMBER命令

判断成员元素是否是集合的成员。返回 0或者1

```
SISMEMBER -key- -value-
```

SREM命令

移除set集合中的指定元素

```
SREM -key- -value-
```

SRANDMEMBER命令

在指定集合中随机获取指定个数的成员

```
SRANDMEMBER -key- -size(默认为1)-
```

对比命令

SDIFF命令

展示出两集合中的差集

```
SDIFF -key1- -key2-
```

SINTER命令

展示出两集合中的交集

```
SINTER -key1- -key2-
```

SUNION命令

展示出两集合中的并集

```
SUNION -key1- -key2-
```

5.Zset

有序集合和集合一样也是 **string** 类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个 **double** 类型的分数。

redis 正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(**score**)却可以重复。

分数值可以是整数或双精度浮点数。

ZADD命令

向有序集合添加一个或多个成员,或者更新已存在成员的分数

```
ZADD -key- -score1- -value1- -score2- -value2- ...
```

ZSCARD命令

计算集合中元素的数量。

```
ZSCARD -key-
```

ZRANGE/ZREVRANGE命令

通过索引区间返回有序集合指定区间内的成员

```
ZRANGE/ZREVRANGE -key- -minScore- -maxScore- (-withscores-)
```

三种特殊数据类型

Geospatial - 地理位置

Geospatial 可以推算地理位置信息,两地之间距离,方圆几里的人.

GEOSPATIAL的底层原理是**ZSET**. 所以也可以使用**ZSET**的命令

Geoadd - 添加地理位置

```
# 两极无法添加,通常我们使用java程序一次性导入
Geoadd -表名- -东经- -北纬- -列名- [-东经- -北纬- -列名-]...
Geoadd china:city 106.73 31.86 bazhong
```

Geopos - 查询地理位置

```
# 获取指定的城市的经纬度
geopos -表名- -列名-
geopos china:city shanghai
```

Geodist - 计算地理之间距离

单位:

- **M** 表示 单位为米 (默认)
- **KM** 表示 单位为千米
- **MI** 表示 单位为英里
- **FT** 表示 单位为英寸

```
Geodist -表名- -列名1- -列名2- [单位]
Geodist china:city chengdu bazhong km
```

Georadius - 寻找附近(通过经纬度)

```
Georadius -表名- -经度- -纬度- -半径距离- [单位] [withcoord/withdist] [count 数量]
Georadius china:city 110 30 500 km count 3
```

Georadiusbymember - 寻找附近(通过指定元素)

```
Georadiusbymember -表名- -列名- -半径距离- [单位]
Georadiusbymember china:city bazhong 1000 km
```

Hyperloglog数据结构 - 基数统计

Redis Hyperloglog - 基数统计.

基数:不可重复的元素,可以接受误差.

网页的UV(一个人访问网站多次,依旧算一个人的数据量)

传统的方式,set保存用户的id,然后就可以统计set中的元素数量作为访问标准,会消耗大量的资源,比较麻烦.

Hyperloglog的优点:

- 占用的内存固定, 2^{64} 大小元素,只需要12KB内存存储.

```
PFADD mykey a b c d e f g h i j k # 创建第一组元素
(integer) 1
PFCOUNT mykey # 统计第一组元素基数数量
(integer) 11

PFADD mykey2 a b c d p q l # 创建第二组元素
(integer) 1
PFCOUNT mykey2 # 统计第二组元素基数数量
(integer) 7

# PFMERGE(合并log) -newkey- -key1- -key2-
PFMERGE mykey3 mykey mykey2 # 合并第一组元素和第二组元素
OK
PFCOUNT mykey3 # 统计第三组元素基数数量
(integer) 15
```

如果允许容错,那么一定可以使用Hyperloglog. 但如果不允许容错,那使用SET或者自己的数据类型即可.

Bitmaps - 位图(位存储)

统计用户信息,活跃,不活跃,登录,未登录,打卡,未打卡...两种状态所有Bitmaps!

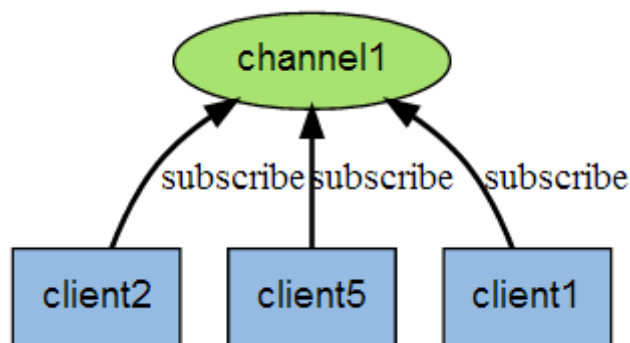
Bitmaps数据结构,二进制记录,只有0,1 两种状态.

365天的数据状态=365 bit 1字节=8bit 46个字节即可.

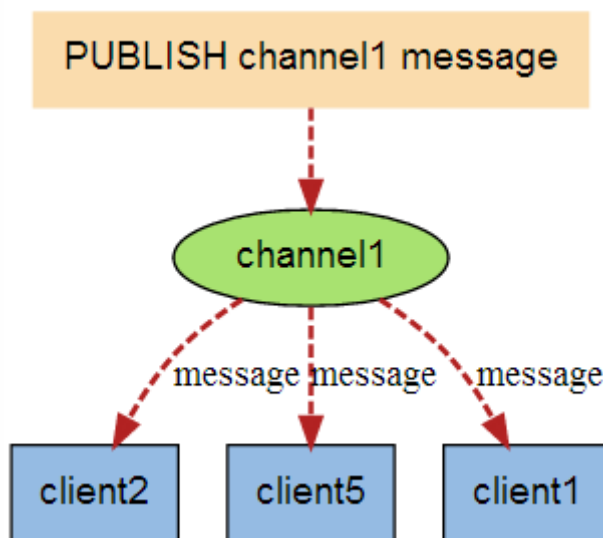
```
# Setbit -表名- -时间- -状态(0\1) -
127.0.0.1:6379> setbit sign 0 1 # 周一 打卡
(integer) 0
127.0.0.1:6379> setbit sign 1 0 # 周二 未打卡
(integer) 0
127.0.0.1:6379> setbit sign 2 1 # 周三 打卡
(integer) 0
127.0.0.1:6379> setbit sign 3 1 # 周四 打卡
(integer) 0
127.0.0.1:6379> setbit sign 4 0 # 周五 未打卡
(integer) 0
127.0.0.1:6379> getbit sign 4 # 查询周五打卡状态
(integer) 0
127.0.0.1:6379> bitcount sign # 查询所有打卡次数(只统计1)
```

发布订阅

发布订阅 (pub/sub) 是一种消息通信模式：发送者 (pub) 发送消息，订阅者 (sub) 接收消息。



当有新消息通过 PUBLISH 命令发送给频道 channel1 时，这个消息就会被发送给订阅它的三个客户端：



SUBSCRIBE命令

订阅给定的一个或多个频道的信息。

```
Subscribe -channel1- -channel2- ....
```


PUBLISH命令

将信息推送到指定的频道。

```
Publish -channel- -message-
```

UNSUBSCRIBE命令

退订指定频道。

```
Unsubscribe -channel1- -channel2-
```

事务

Redis的事务本质:一组命令的集合. 一个事务中所有的命令都会被序列化,在事务执行过程中,会按照顺序执行!

Redis单条命令保证原子性,但Redis的事务不包证原子性!

-Redis事务没有隔离级别的概念-

事务执行步骤

- 开启事务(multi)
- 命令入队(...)
- 执行事务(exec) / 撤销事务(discard)

事务监控

悲观锁:无论什么操作都认为会存在问题,随时都会上锁

乐观锁:无论什么操作都不会认为会存在问题,随时都不上锁.更新数据时会判断一下,在此期间是否修改过数据,

获取Version.更新时对比Version,没变则更新成功,否则失败.

测试多线程修改值,使用watch可以当作redis的乐观锁的操作.unwatch取消乐观锁.

Redis的持久化

面试和工作,持久化都是重点!

RDB(Redis DataBase)

指定的时间间隔内将内存中的数据快照 写入磁盘中,恢复时就将快照读取到内存中.

Redis会单独创建一个(fork)子进程来进行持久化,先将数据写入一个临时文件中,再用这个临时文件替换上场持久化好的文件.

整个过程主进程不会进行任何IO操作的,这就确保了极高的性能.

RDB缺点:最后一次持久化后的数据宕机可能丢失.

Redis默认持久化就是RDB,一般情况下不需要修改这个配置!

RDB保存的文件是dump.rdb 在工作环境我们会将该文件进行备份.

```
# The filename where to dump the DB
dbfilename dump.rdb
```

RDB触发机制:

- 1.save的规则满足情况下触发RDB持久化规则.
- 2.执行了flushall命令,也会触发RDB持久化规则.
- 3.退出Redis,也会触发RDB持久化规则

如何恢复RDB文件:

1.将需要恢复的RDB文件放置到redis启动的文件目录下就可以,redis启动时就会自动检查dump.rdb文件并恢复其中的数据.

2.查看需要存放的位置

```
127.0.0.1:6379> config get dir
1) "dir"
2) "D:\\redis" # 如果在这个目录下查找dump.rdb,启动就会自动恢复其中的数据.
```

RDB优点:

- 1.适合大规模的数据恢复!
- 2.对数据完整性要求不高!

RDB缺点:

- 1.需要一定时间间隔进行操作.最后一次持久化后的数据宕机可能丢失.
 - 2.fork进程的时候,会占用一定的内容空间.
-

Jedis

Jedis是Redis官方推荐的Java连接开发工具.使用Java操作Redis的中间键.

Maven依赖

```
<!--导入jedis包-->
<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.3.0</version>
</dependency>
<!--导入fastjson包-->
<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.75</version>
</dependency>
```

编码测试

- 连接数据库

```
//1.创建一个Jedis对象
Jedis jedis = new Jedis("127.0.0.1", 6379);
System.out.println(jedis.ping());
```

- 操作命令

```
JSONObject jsonObject=new JSONObject();
jsonObject.put("hello", "world");
jsonObject.put("name", "zssaer");
//开启redis事务
Transaction multi=jedis.multi();
String jsonString = jsonObject.toJSONString();

try {
    multi.set("user1", jsonString);
    multi.exec();
} catch (Exception e) {
    multi.discard();
    e.printStackTrace();
}finally {
    System.out.println(jedis.get("user1"));
    jedis.close();
}
```

- 断开连接

```
jedis.close(); //关闭连接
```

SpringBoot+Redis

在SpringBoot2.x后,原来使用的Jedis被替换成了lettuce.

Jedis:底层采用的是直连方式,多个线程操作是不安全的.多线程连接操作需要使用jedis-pool连接池!更像BIO模式

lettuce:采用netty,实例可以在多个线程中共享,不存在线程不安全的情况.可以减少线程数量.更像NIO模式.

Maven依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.3.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.12.1</version>
</dependency>
```

RedisAutoConfiguration源码分析:

```
@Bean
@ConditionalOnMissingBean(name = {"redisTemplate"}) //我们可以自己定义个
redisTemplate来替换这个默认的!
@ConditionalOnSingleCandidate(RedisConnectionFactory.class)
public RedisTemplate<Object, Object>
redisTemplate(RedisConnectionFactory redisConnectionFactory) {
    //默认的RedisTemplate 没有过多设置,redis对象都是需要序列化!
    //两个泛型都是object,object类型,我们后使用需要强制转换<String,object>
    RedisTemplate<Object, Object> template = new RedisTemplate();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}

@Bean
@ConditionalOnMissingBean
@ConditionalOnSingleCandidate(RedisConnectionFactory.class)
```

```
public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory
redisConnectionFactory) {
    StringRedisTemplate template = new StringRedisTemplate();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}
```

RedisTemplate使用:

```
//opsForValue 操作String
//opsForList 操作List
//opsForSet 操作Set
//opsForHash 操作Hash
//opsForZset 操作Zset
redisTemplate.opsForValue().set("mykey", "myvalue");
System.out.println(redisTemplate.opsForValue().get("mykey"));
```

配置方法

1.配置properties文件(yaml文件)中的redis环境参数

```
# Redis数据库索引（默认为0）
spring.redis.database=0
# Redis服务器地址
spring.redis.host=localhost
# Redis服务器连接端口
spring.redis.port=6379
# Redis服务器连接密码（默认为空）
spring.redis.password=123456
#连接池最大连接数（使用负值表示没有限制）
spring.redis.jedis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.jedis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.jedis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.jedis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=300
```

2.配置RedisConfig类

以下配置类直接copy使用即可

```
/**
 * Redis配置类
 */
@Configuration
@EnableCaching //开启注解
```

```

public class RedisConfig extends CachingConfigurerSupport {
    /**
     * retemplate相关配置
     * @param factory
     * @return
     */
    @Bean
    public RedisTemplate<String, Object>
redisTemplate(RedisConnectionFactory factory) {

        RedisTemplate<String, Object> template = new RedisTemplate<>();
        // 配置连接工厂
        template.setConnectionFactory(factory);

        //使用Jackson2JsonRedisSerializer来序列化和反序列化redis的value值（默
        认使用JDK的序列化方式）
        Jackson2JsonRedisSerializer jacksonSeial = new
Jackson2JsonRedisSerializer(Object.class);

        ObjectMapper om = new ObjectMapper();
        // 指定要序列化的域，field,get和set,以及修饰符范围，ANY是都有包括private
        和public
        om.setVisibility(PropertyAccessor.ALL,
JsonAutoDetect.Visibility.ANY);
        // 指定序列化输入的类型，类必须是非final修饰的，final修饰的类，比如
        String,Integer等会跑出异常
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jacksonSeial.setObjectMapper(om);

        // 值采用json序列化
        template.setValueSerializer(jacksonSeial);
        //使用StringRedisSerializer来序列化和反序列化redis的key值
        template.setKeySerializer(new StringRedisSerializer());

        // 设置hash key 和value序列化模式
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(jacksonSeial);
        template.afterPropertiesSet();

        return template;
    }

    /**
     * 对hash类型的数据操作
     *
     * @param redisTemplate
     * @return
     */
    @Bean

```

```
    public HashOperations<String, String, Object>
hashOperations(RedisTemplate<String, Object> redisTemplate) {
    return redisTemplate.opsForHash();
}

/**
 * 对redis字符串类型数据操作
 *
 * @param redisTemplate
 * @return
 */
@Bean
    public ValueOperations<String, Object>
valueOperations(RedisTemplate<String, Object> redisTemplate) {
    return redisTemplate.opsForValue();
}

/**
 * 对链表类型的数据操作
 *
 * @param redisTemplate
 * @return
 */
@Bean
    public ListOperations<String, Object>
listOperations(RedisTemplate<String, Object> redisTemplate) {
    return redisTemplate.opsForList();
}

/**
 * 对无序集合类型的数据操作
 *
 * @param redisTemplate
 * @return
 */
@Bean
    public SetOperations<String, Object>
setOperations(RedisTemplate<String, Object> redisTemplate) {
    return redisTemplate.opsForSet();
}

/**
 * 对有序集合类型的数据操作
 *
 * @param redisTemplate
 * @return
 */
@Bean
```

```

        public ZSetOperations<String, Object>
zSetOperations(RedisTemplate<String, Object> redisTemplate) {
            return redisTemplate.opsForZSet();
        }
    }
}

```

3.配置RedisUtil类

以下工具类直接copy使用即可

```

/**
 * TODO (控制RedisTemplate封装)
 */
@Component
public class RedisUtil {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    public RedisUtil(RedisTemplate<String, Object> redisTemplate) {
        this.redisTemplate = redisTemplate;
    }

    /**
     * 指定缓存失效时间
     * @param key 键
     * @param time 时间(秒)
     * @return
     */
    public boolean expire(String key, long time) {
        try {
            if (time > 0) {
                redisTemplate.expire(key, time, TimeUnit.SECONDS);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 根据key 获取过期时间
     * @param key 键 不能为null
     * @return 时间(秒) 返回0代表为永久有效
     */
    public long getExpire(String key) {
        return redisTemplate.getExpire(key, TimeUnit.SECONDS);
    }
}

```



```

/**
 * 判断key是否存在
 * @param key 键
 * @return true 存在 false不存在
 */
public boolean hasKey(String key) {
    try {
        return redisTemplate.hasKey(key);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 删除缓存
 * @param key 可以传一个值 或多个
 */
@SuppressWarnings("unchecked")
public void del(String ... key) {
    if (key != null && key.length > 0) {
        if (key.length == 1) {
            redisTemplate.delete(key[0]);
        } else {
            redisTemplate.delete(CollectionUtils.arrayToList(key));
        }
    }
}

//=====String=====
/**
 * 普通缓存获取
 * @param key 键
 * @return 值
 */
public Object get(String key) {
    return key == null ? null : redisTemplate.opsForValue().get(key);
}

/**
 * 普通缓存放入
 * @param key 键
 * @param value 值
 * @return true成功 false失败
 */
public boolean set(String key, Object value) {
    try {
        redisTemplate.opsForValue().set(key, value);
    }
}

```

```

        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 普通缓存放入并设置时间
 * @param key 键
 * @param value 值
 * @param time 时间(秒) time要大于0 如果time小于等于0 将设置无限期
 * @return true成功 false 失败
 */
public boolean set(String key, Object value, long time) {
    try {
        if (time > 0) {
            redisTemplate.opsForValue().set(key, value, time,
TimeUnit.SECONDS);
        } else {
            set(key, value);
        }
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 递增
 * @param key 键
 * @param delta 要增加几(大于0)
 * @return
 */
public long incr(String key, long delta) {
    if (delta < 0) {
        throw new RuntimeException("递增因子必须大于0");
    }
    return redisTemplate.opsForValue().increment(key, delta);
}

/**
 * 递减
 * @param key 键
 * @param delta 要减少几(小于0)
 * @return
 */
public long decr(String key, long delta) {

```

```

        if(delta<0){
            throw new RuntimeException("递减因子必须大于0");
        }
        return redisTemplate.opsForValue().increment(key, -delta);
    }

//=====Map=====
/**
 * HashGet
 * @param key 键 不能为null
 * @param item 项 不能为null
 * @return 值
 */
public Object hget(String key,String item){
    return redisTemplate.opsForHash().get(key, item);
}

/**
 * 获取hashKey对应的所有键值
 * @param key 键
 * @return 对应的多个键值
 */
public Map<Object,Object> hmget(String key){
    return redisTemplate.opsForHash().entries(key);
}

/**
 * HashSet
 * @param key 键
 * @param map 对应多个键值
 * @return true 成功 false 失败
 */
public boolean hmset(String key, Map<String,Object> map){
    try {
        redisTemplate.opsForHash().putAll(key, map);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * HashSet 并设置时间
 * @param key 键
 * @param map 对应多个键值
 * @param time 时间(秒)
 * @return true成功 false失败

```

```

    */
    public boolean hmset(String key, Map<String, Object> map, long time) {
        try {
            redisTemplate.opsForHash().putAll(key, map);
            if (time > 0) {
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 向一张hash表中放入数据,如果不存在将创建
     * @param key 键
     * @param item 项
     * @param value 值
     * @return true 成功 false失败
     */
    public boolean hset(String key, String item, Object value) {
        try {
            redisTemplate.opsForHash().put(key, item, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 向一张hash表中放入数据,如果不存在将创建
     * @param key 键
     * @param item 项
     * @param value 值
     * @param time 时间(秒) 注意:如果已存在的hash表有时间,这里将会替换原有的时间
     * @return true 成功 false失败
     */
    public boolean hset(String key, String item, Object value, long time) {
        try {
            redisTemplate.opsForHash().put(key, item, value);
            if (time > 0) {
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

```

```

    }
}

/**
 * 删除hash表中的值
 * @param key 键 不能为null
 * @param item 项 可以使多个 不能为null
 */
public void hdel(String key, Object... item){
    redisTemplate.opsForHash().delete(key,item);
}

/**
 * 判断hash表中是否有该项的值
 * @param key 键 不能为null
 * @param item 项 不能为null
 * @return true 存在 false不存在
 */
public boolean hHasKey(String key, String item){
    return redisTemplate.opsForHash().hasKey(key, item);
}

/**
 * hash递增 如果不存在,就会创建一个 并把新增后的值返回
 * @param key 键
 * @param item 项
 * @param by 要增加几(大于0)
 * @return
 */
public double hincr(String key, String item,double by){
    return redisTemplate.opsForHash().increment(key, item, by);
}

/**
 * hash递减
 * @param key 键
 * @param item 项
 * @param by 要减少记(小于0)
 * @return
 */
public double hdecr(String key, String item,double by){
    return redisTemplate.opsForHash().increment(key, item,-by);
}

//=====set=====
/**
 * 根据key获取Set中的所有值
 * @param key 键
 * @return

```

```

    */
    public Set<Object> sGet(String key) {
        try {
            return redisTemplate.opsForSet().members(key);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 根据value从一个set中查询,是否存在
     * @param key 键
     * @param value 值
     * @return true 存在 false不存在
     */
    public boolean sHasKey(String key, Object value) {
        try {
            return redisTemplate.opsForSet().isMember(key, value);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 将数据放入set缓存
     * @param key 键
     * @param values 值 可以是多个
     * @return 成功个数
     */
    public long sSet(String key, Object...values) {
        try {
            return redisTemplate.opsForSet().add(key, values);
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }

    /**
     * 将set数据放入缓存
     * @param key 键
     * @param time 时间(秒)
     * @param values 值 可以是多个
     * @return 成功个数
     */
    public long sSetAndTime(String key, long time, Object...values) {
        try {

```

```

        Long count = redisTemplate.opsForSet().add(key, values);
        if(time>0) {
            expire(key, time);
        }
        return count;
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

/**
 * 获取set缓存的长度
 * @param key 键
 * @return
 */
public long sGetSetSize(String key){
    try {
        return redisTemplate.opsForSet().size(key);
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

/**
 * 移除值为value的
 * @param key 键
 * @param values 值 可以是多个
 * @return 移除的个数
 */
public long setRemove(String key, Object ...values) {
    try {
        Long count = redisTemplate.opsForSet().remove(key, values);
        return count;
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

//=====list=====

/**
 * 获取list缓存的内容
 * @param key 键
 * @param start 开始
 * @param end 结束 0 到 -1代表所有值
 * @return
 */

```

```

    */
    public List<Object> lGet(String key, long start, long end) {
        try {
            return redisTemplate.opsForList().range(key, start, end);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 获取list缓存的长度
     * @param key 键
     * @return
     */
    public long lGetListSize(String key) {
        try {
            return redisTemplate.opsForList().size(key);
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }

    /**
     * 通过索引 获取list中的值
     * @param key 键
     * @param index 索引 index>=0时, 0 表头, 1 第二个元素, 依次类推; index<0
    时, -1, 表尾, -2倒数第二个元素, 依次类推
     * @return
     */
    public Object lGetIndex(String key, long index) {
        try {
            return redisTemplate.opsForList().index(key, index);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 将list放入缓存
     * @param key 键
     * @param value 值
     * @return
     */
    public boolean lSet(String key, Object value) {
        try {
            redisTemplate.opsForList().rightPush(key, value);

```



```

        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将list放入缓存
 * @param key 键
 * @param value 值
 * @param time 时间(秒)
 * @return
 */
public boolean lSet(String key, Object value, long time) {
    try {
        redisTemplate.opsForList().rightPush(key, value);
        if (time > 0) {
            expire(key, time);
        }
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将list放入缓存
 * @param key 键
 * @param value 值
 * @return
 */
public boolean lSet(String key, List<Object> value) {
    try {
        redisTemplate.opsForList().rightPushAll(key, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将list放入缓存
 * @param key 键
 * @param value 值
 * @param time 时间(秒)
 * @return

```

```

    */
    public boolean lSet(String key, List<Object> value, long time) {
        try {
            redisTemplate.opsForList().rightPushAll(key, value);
            if (time > 0) {
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 根据索引修改list中的某条数据
     * @param key 键
     * @param index 索引
     * @param value 值
     * @return
     */
    public boolean lUpdateIndex(String key, long index, Object value) {
        try {
            redisTemplate.opsForList().set(key, index, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 移除N个值为value
     * @param key 键
     * @param count 移除多少个
     * @param value 值
     * @return 移除的个数
     */
    public long lRemove(String key, long count, Object value) {
        try {
            Long remove = redisTemplate.opsForList().remove(key, count,
value);

            return remove;
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }
}

```

4.创建实体类

```
@Component
public class User implements Serializable {
    private Integer id;

    private String username;

    private String password;

    public Integer getId() {
        return id;
    }
    ...
}
```

其中实体类必须继承其**Serializable** 类实现其序列化

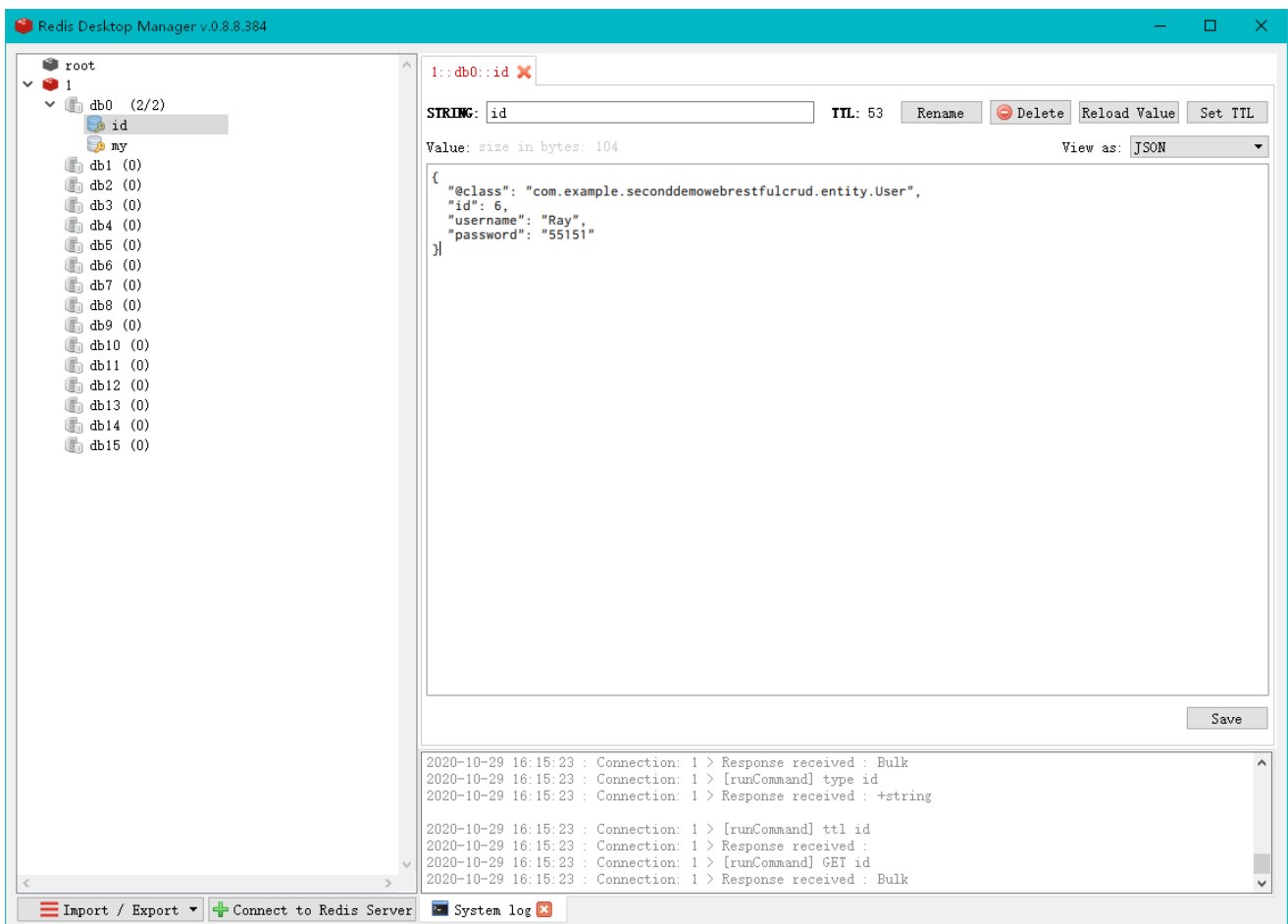
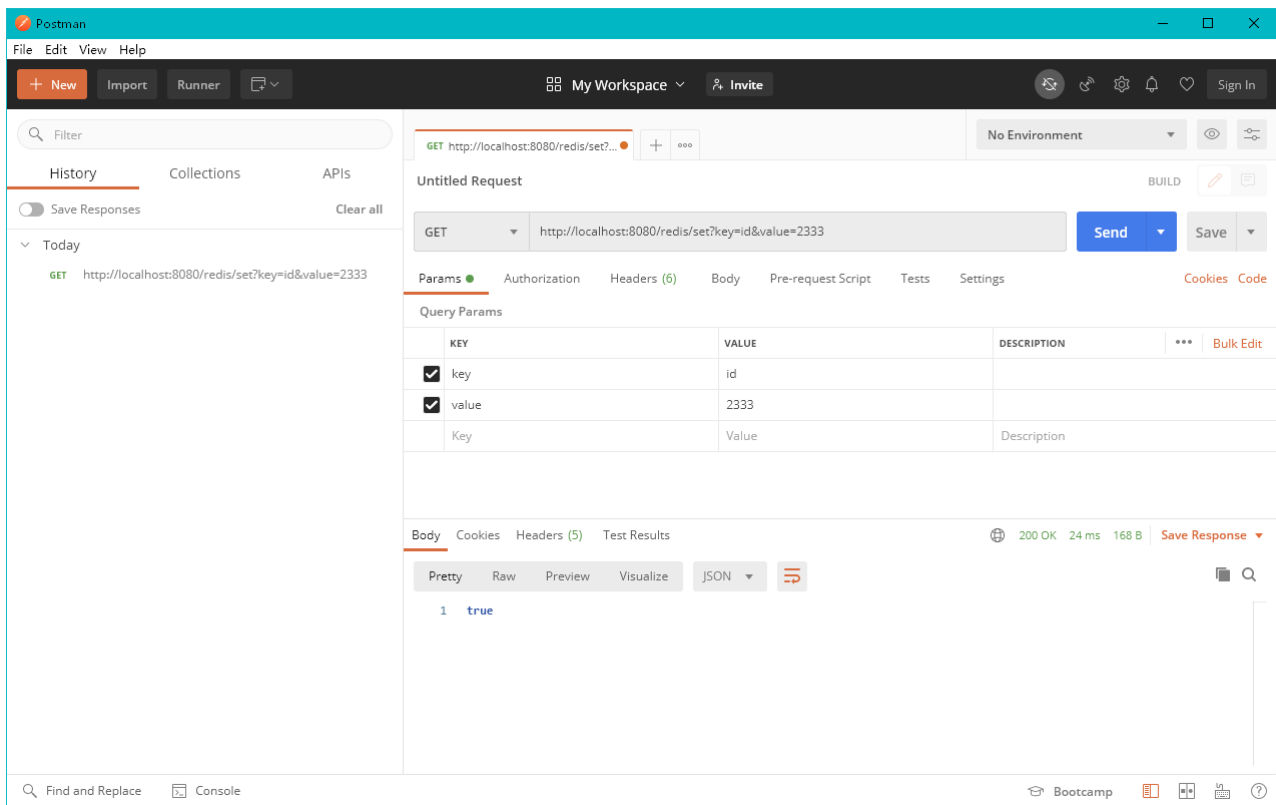
5.在Controller中使用（读写）

```
private static int ExpireTime = 60; //设置redis存储的过期时间为60s

@Resource
private RedisUtil redisUtil;

/**
 * 写入数据至Redis
 * @param key Key值
 * @param value
 * @return
 */
@RequestMapping("set")
public boolean redisSet(String key,String value) {
    User user=new User();
    user.setId(6);
    user.setUsername("Ray");
    user.setPassword("55151");
    return redisUtil.set(key,user,ExpireTime);
}

/**
 * 读取Redis中的键值
 * @param key Key值
 * @return
 */
@RequestMapping("get")
public Object redisget(String key) {
    return redisUtil.get(key);
}
```



MD5+Salt+Hash散列进行数据加密

对于存储重要信息内容（如密码、支付码）时，为了用户信息的安全性，必须使用数据加密。

其中MD5+Salt+Hash散列 加密方式比较流行且简便。

主要方法

注册用户时

```
public class User {  
    private Integer id;  
    private String username;  
    private String password;  
    //保存注册时随机盐值，以确保登录时解密  
    private String salt;  
    ...  
}
```

```
public class SaltUtils {  
    /**  
     * 生成随机Salt的静态方法，以确保Salt不固定  
     * @param n  
     * @return  
     */  
    public static String getSalt(int n) {  
        char[] chars =  
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*"  
().toCharArray();  
        StringBuffer sb=new StringBuffer();  
        for (int i = 0; i < n; i++) {  
            char c=chars[new Random().nextInt(chars.length)];  
            sb.append(c);  
        }  
        return sb.toString();  
    }  
}
```

```
/**  
 * 以下步骤多在Service层实现  
 */  
//1.生成随机盐  
//2.将随机盐保存到数据  
String salt = SaltUtils.getSalt(8);  
//明文密码进行MD5 + salt + hash散列次数  
Md5Hash md5Hash = new Md5Hash(password, salt, 1024);  
//将用户输入的密码进行16进制化  
password=md5Hash.toHex();
```

Shiro安全框架

Apache Shiro是一个强大且易用的Java安全框架,执行身份验证、授权、密码和会话管理。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

主要功能

shiro主要有三大功能模块:

1. **Subject**: 主体,一般指用户。
2. **SecurityManager**: 安全管理器,管理所有**Subject**,可以配合内部安全组件。(类似于SpringMVC中的DispatcherServlet)
3. **Realms**: 用于进行权限信息的验证,一般需要自己实现。

细分功能

1. Authentication: 身份认证/登录(账号密码验证)。
2. Authorization: 授权,即角色或者权限验证。
3. Session Manager: 会话管理,用户登录后的session相关管理。
4. Cryptography: 加密,密码加密等。
5. Web Support: Web支持,集成Web环境。
6. Caching: 缓存,用户信息、角色、权限等缓存到如redis等缓存中。
7. Run As: 允许一个用户假装为另一个用户(如果他们允许)的身份进行访问。
8. Remember Me: 记住我,登录后,下次再来的话不用登录了。

Maven依赖

```
<!--shiro-core-->
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-core</artifactId>
  <version>1.6.0</version>
</dependency>
```

快速入门语句

```
// 得到DefaultSecurityManager对象
DefaultSecurityManager defaultManager=new
DefaultSecurityManager();
// 读取ini配置文件
IniRealm iniRealm=new IniRealm("classpath:shiro.ini");
// 配置DefaultSecurityManager对象
defaultSecurityManager.setRealm(iniRealm);
// 获取SecurityUtils对象
```

```

SecurityUtils.setSecurityManager(defaultSecurityManager);

// 获取当前用户对象 Subject
Subject currentUser = SecurityUtils.getSubject();

// 通过当前用户获取Session
Session session = currentUser.getSession();

//判断用户是否被认证
currentUser.isAuthenticated()

//通过Token进行登录操作
currentUser.login(token)

//根据输入账户名和密码获取Token
UsernamePasswordToken token = new UsernamePasswordToken("lonestarr",
"vespa");

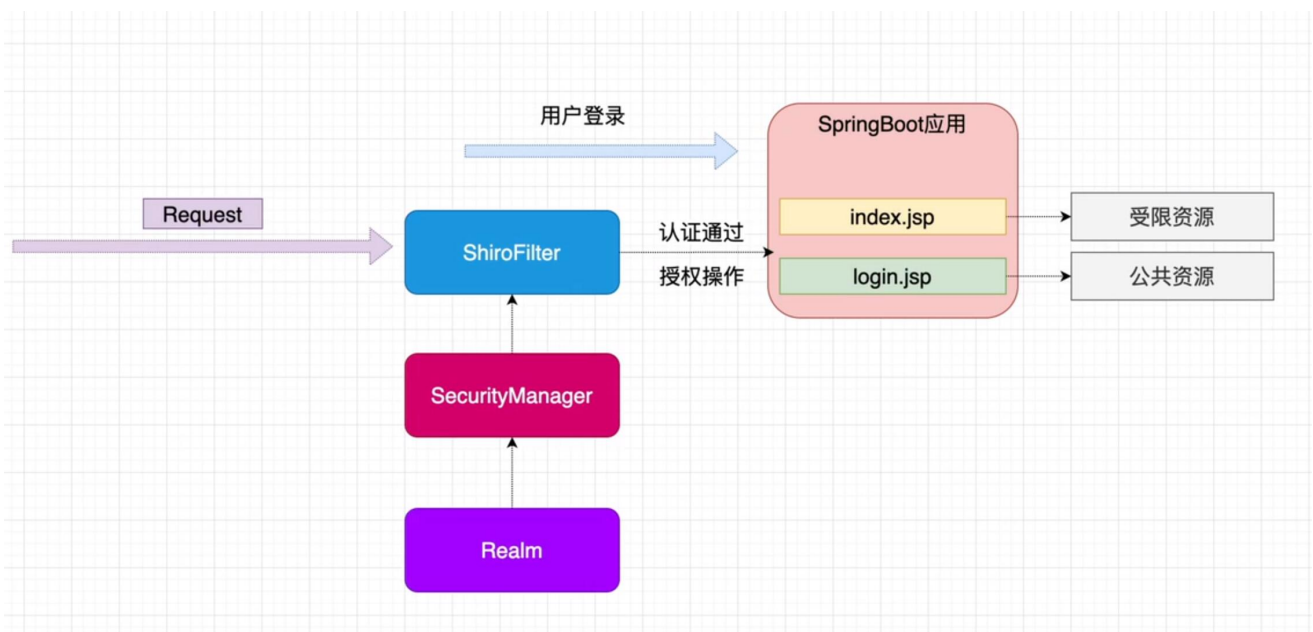
//判断用户的身份
currentUser.hasRole("schwartz")

//判断用户拥有的权限
currentUser.isPermitted("lightsaber:wield")

//注销当前用户
currentUser.logout();

```

SpringBoot继承Shiro



Maven依赖

```
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-spring-boot-starter</artifactId>
  <version>1.6.0</version>
</dependency>
```

创建Realm类

```
public class CustomRealm extends AuthorizingRealm {

    @Autowired
    private LoginService loginService;

    /**
     * @MethodName doGetAuthorizationInfo
     * @Description 权限配置类
     * @Param [principalCollection]
     * @Return AuthorizationInfo
     * @Author WangShiLin
     */
    @Override
    protected AuthorizationInfo
doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        //获取登录用户名
        String name = (String)
principalCollection.getPrimaryPrincipal();
        //查询用户名称
        User user = loginService.getUserByName(name);
        //添加角色和权限
        SimpleAuthorizationInfo simpleAuthorizationInfo = new
SimpleAuthorizationInfo();
        for (Role role : user.getRoles()) {
            //添加角色
            simpleAuthorizationInfo.addRole(role.getRoleName());
            //添加权限
            for (Permissions permissions : role.getPermissions()) {
                //将用户拥有的权限加载到获取权限中

                simpleAuthorizationInfo.addStringPermission(permissions.getPermissionsName());
            }
        }
        return simpleAuthorizationInfo;
    }

    /**
     * @MethodName doGetAuthenticationInfo
```



```

    * @Description 认证配置类
    * @Param [authenticationToken]
    * @Return AuthenticationInfo
    * @Author WangShiLin
    */
    @Override
    protected AuthenticationInfo
doGetAuthenticationInfo(AuthenticationToken authenticationToken) throws
AuthenticationException {
    if (StringUtils.isEmpty(authenticationToken.getPrincipal())) {
        return null;
    }
    //获取用户信息
    String name = authenticationToken.getPrincipal().toString();
    User user = loginService.getUserByName(name);
    if (user == null) {
        //这里返回后会报出对应异常
        return null;
    } else {
        //这里验证authenticationToken和simpleAuthenticationInfo的信息
        SimpleAuthenticationInfo simpleAuthenticationInfo = new
SimpleAuthenticationInfo(name,
user.getPassword().toString(),ByteSource.Util.bytes("x23*2d"),getName())
;

        return simpleAuthenticationInfo;
    }
}

```

创建Realm类继承AuthorizingRealm，重写doGetAuthorizationInfo（授权配置）、doGetAuthenticationInfo（认证配置）方法。

其中AuthenticationToken 用于收集用户提交的身份（如用户名）及凭据（如密码）。

其中ByteSource.Util.bytes方法为用户设置时的随机盐值。

创建ShiroConfig配置类

```

@Configuration
public class ShiroConfig {
    //将自己的验证方式加入容器
    @Bean
    public CustomRealm myShiroRealm() {
        return new CustomRealm();
    }

    //权限管理，配置主要是Realm的管理认证
    @Bean
    public DefaultWebSecurityManager securityManager() {

```

```

        DefaultWebSecurityManager securityManager = new
DefaultWebSecurityManager();
        //设置realm hash验证
        HashedCredentialsMatcher credentialsMatcher= new
HashedCredentialsMatcher();
        //使用加密方法
        credentialsMatcher.setHashAlgorithmName("md5");
        //散列次数
        credentialsMatcher.setHashIterations(1024);
        userRealm.setCredentialsMatcher(credentialsMatcher);
        //绑定Realm
        securityManager.setRealm(myShiroRealm());
        return securityManager;
    }

    //Filter工厂，设置对应的过滤条件和跳转条件
    @Bean
    public ShiroFilterFactoryBean shiroFilterFactoryBean(SecurityManager
securityManager) {
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
ShiroFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(securityManager);
        Map<String, String> map = new HashMap<>();
        //登出
        map.put("/logout", "logout");
        //对所有用户认证
        map.put("/**", "authc");
        //登录
        shiroFilterFactoryBean.setLoginUrl("/login");
        //首页
        shiroFilterFactoryBean.setSuccessUrl("/index");
        //错误页面，认证不通过跳转
        shiroFilterFactoryBean.setUnauthorizedUrl("/error");
        shiroFilterFactoryBean.setFilterChainDefinitionMap(map);
        return shiroFilterFactoryBean;
    }

    //注入权限管理
    @Bean
    public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
        AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor = new
AuthorizationAttributeSourceAdvisor();

        authorizationAttributeSourceAdvisor.setSecurityManager(securityManager)
;

        return authorizationAttributeSourceAdvisor;
    }

```

```
}
```

其中shiro内置过滤器：

```
anno: 无需认证即可访问
authc: 必须认证才可以访问
user : 不许拥有记住我功能才能访问
perms: 拥有对某个资源访问权限才能使用    ((perms认证必须放在authc认证前, 否则无效))
role: 拥有某个角色权限才能访问
```

权限限定访问：

```
map.put("/set", "perms[user:set]");    //只限定拥有'user:set'权限的用户访问
```

Controller类

```
@RestController
@Slf4j
public class LoginController {

    @GetMapping("/login")
    public String login(User user) {
        if (StringUtils.isEmpty(user.getUserName()) ||
        StringUtils.isEmpty(user.getPassword())) {
            return "请输入用户名和密码! ";
        }
        //用户认证信息
        Subject subject = SecurityUtils.getSubject();
        UsernamePasswordToken usernamePasswordToken = new
        UsernamePasswordToken(
            user.getUserName(),
            user.getPassword()
        );
        try {
            //进行验证, 这里可以捕获异常, 然后返回对应信息
            subject.login(usernamePasswordToken);
            //        subject.checkRole("admin");
            //        subject.checkPermissions("query", "add");
        } catch (UnknownAccountException e) {
            log.error("用户名不存在!", e);
            return "用户名不存在! ";
        } catch (AuthenticationException e) {
            log.error("账号或密码错误!", e);
            return "账号或密码错误! ";
        } catch (AuthorizationException e) {
            log.error("没有权限!", e);
            return "没有权限";
        }
    }
}
```

```
        return "login success";
    }

    ...
}
```

- 1.用 SecurityUtils.getSubject()获取Subject类。
- 2.将用户输入进去的账户密码信息封装入UsernamePasswordToken类。
- 3.使用Subject类的login方法判断登录结果，并捕捉相关错误异常。

登录错误异常

- UnknownAccountException: 用户名不存在
- AuthenticationException: 账户或者密码错误
- AuthorizationException: 没有权限
- Account Exception : 账号异常
 - ConcurrentAccessException: 并发访问异常（多个用户同时登录时抛出）
 - UnknownAccountException: 未知的账号
 - ExcessiveAttemptsException: 认证次数超过限制
 - DisabledAccountException: 禁用的账号
 - LockedAccountException: 账号被锁定
 - UnsupportedTokenException: 使用了不支持的Token

Shiro+Thymeleaf页面整合

Maven依赖:

```
<!--
https://mvnrepository.com/artifact/com.github.theborakompanioni/thymeleaf-extras-shiro -->
<dependency>
    <groupId>com.github.theborakompanioni</groupId>
    <artifactId>thymeleaf-extras-shiro</artifactId>
    <version>2.0.0</version>
</dependency>
```

Thymeleaf页面头部加入 xmlns:shiro="<http://www.pollix.at/thymeleaf/shiro>" 开启代码提示。

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
      xmlns:shiro="http://www.pollix.at/thymeleaf/shiro">
    ...
</html>
```

常用标签:

The has Permission tag

shiro:hasPermission="xxx" 判断当前用户是否拥有xxx权限

```
<div shiro:hasPermission="user:set"></div>
```

The authenticated tag

authenticated="" 已经用户得到认证

```
<a shiro:authenticated="" href="updateAccount.html">Update your contact
information</a>
```

The hasRole tag

shiro:hasRole="xxx" 判断当前用户为xxx权限

```
<a shiro:hasRole="administrator" href="admin.html">Administer the
system</a>
```

权限、角色访问控制

方法一：直接在页面控制（以Thymeleaf为例）

```
<!--拥有user:add权限的任何人都能看见-->
<div shiro:hasPermission="user:add:*">
    <a th:href="@{/user/add}">Add</a>
</div>
<!--拥有admin角色才能看见-->
<div shiro:hasRole="admin">
    <a th:href="@{/user/update}">Update</a>
</div>
```

方法二：Controller代码层中控制

```
//获取当前用户
Subject subject = SecurityUtils.getSubject();
if (subject.hasRole("admin")) {
    System.out.println("添加成功!");
}else{
    System.out.println("添加失败!");
}
```

方法三：代码注释控制

```
@RequestMapping("/user/add")
@RequiresRoles("admin") //判断角色
@RequiresPermissions("user:add:*") //判断权限
public String add() {
    return "user/add";
}
```

JAVA设计模式

OOP七大原则

开：开闭原则

口：接口隔离原则

里：里氏替换原则

合：合成复用原则

最：最少知道原则（迪米特原则）

单：单一职责原则

依：依赖倒置原则

单例模式

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一.这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。
-

单例模式类的构造器必须为**private**私有化,并且需要内建一个实例,必为其提供一个**Get**实例的公共方法。

获取单例模式的类时只有通过使用该**Get**方法获取当前实例.

```
public class SingleObject {

    //创建 SingleObject 的一个对象
    private static SingleObject instance = new SingleObject();

    //让构造函数为 private，这样该类就不会被实例化
    private SingleObject() {}

    //获取唯一可用的对象
    public static SingleObject getInstance(){
        return instance;
    }

    public void showMessage() {
        System.out.println("Hello World!");
    }

}
```

工厂模式

工厂模式实现了创建者和调用者的分离

工厂模式分为: 简单工厂模式

工厂方法模式

抽象工厂模式

核心本质: 实例化对象不能使用**new**,而用工厂方法代替

将选择实现类,建立统一管理和控制.

工厂模式需要创建一个工厂类,用来返回对应需求的实体类.

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }

}
```

```
public class ShapeFactory {
```

//使用 getShape 方法获取形状类型的对象

//简单工厂模式

```
public Shape getShape(String shapeType) {  
    if(shapeType == null){  
        return null;  
    }  
    if(shapeType.equalsIgnoreCase("CIRCLE")){  
        return new Circle();  
    } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
        return new Rectangle();  
    } else if(shapeType.equalsIgnoreCase("SQUARE")){  
        return new Square();  
    }  
    return null;  
}  
  
}
```