# Lucky\_2.1.0的使用文档

## 简介

Lucky是一款基于java的数据持久层框架，是对JDBC高度封装后的产物，主要作用是帮助开发人员简单且高效的进行数据库开发操作。

原生的JDBC操作相对繁琐，主要的原因如下：

1. **JDBC操作数据库需要用到主要三个类：获取链接的类（Connection），预处理类（PreparedStatement）和结果集类（ResultSet），涉及到的类太多，难以记忆。**
2. **JDBC对非查询操作的处理算是比较完美的，最令人抓狂的是对查询操作的处理，当我们执行完查询操作后JDBC只返回给我们一个结果集对象（ResultSet），想要取出详细的结果我们还需要进行繁琐的遍历操作。**
3. **JDBC提供的方法不具备扩展性，就拿查询操作来举例：一般情况下，我们数据库中每多一张表时我们都会为其创建对应的实体类。那么就产生了一个很令人头疼的问题：如果想得到实体类形式的查询结果，我们就不得不一遍又一遍的去写“先遍历结果集，后封装数据”的代码**

初次使用JBDC的同学会有更深刻的体会吧！在开发之初使用JDBC也是令我非常头痛的，所以一直都在思考有没有什么好的解决方案,或者写出一个通用的一劳永逸的方法？答案是有的，因为解决问题的方法永远都比问题本身要多！所以Lucky诞生了！

Lucky的诞生有效的解决了以上的三大问题：

1. **Lucky操作数据库只需要一个类：数据控制类(SqlControl),只要获得了这个类的对象便可以轻松的操作数据库**
2. **Lucky执行非查询操作后返回的是一个布尔类型的结果，执行查询操作后返回的是Object对象或者List<?>集合，我们只需要做一次强制类型转换就能拿到实体类形式的查询结果Lucky中提供的方法几乎就是通用的，想要封装不同结果时只需要传入实体类对象对应的Class对象就好了，Lucky就会根据配置进行自动封装。**

开发Lucky主要使用到的技术有：java的反射机制，泛型，注解，配置文件，JDBC，字符串处理，基本的数据结构。

## 配置问题

俗话说的好：想要马儿跑就得让马儿吃草，所以想要Lucky稳定的工作就必须为她提供可靠的信息，具体信息如下

* **得告诉Lucky你的数据库信息(驱动包，数据库位置，用户名，密码)。**
* **构建表对应的实体类，而且实体类的书写必须符合一定的规范。**
* **建立实体类与表的映射关系（配置方式和注解方式）**

## 使用流程

1. **如何告诉Lucky我的数据库信息？**

这个部分基本是固定格式，在src下创建一个名为lucky.properties的配置文件，properties文件中的内容都是以key-value的形式体现的，所以在配置文件中key的部分大体上都是固定的（左边），有用的信息为value（右边）,这部分的类容是需要我们因人而异的进行配置的。

配置数据库信息：

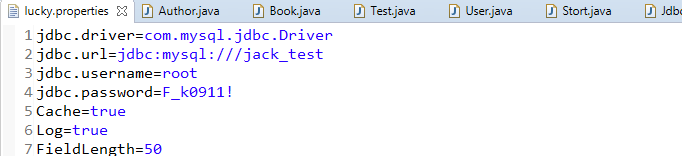
jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql:///databasename

jdbc.username=root

jdbc.password=passeord

**具体场景：**



**注意：使用之前别忘了要导入jar包**

1. **构建实体类的规范有哪些？**

**注意：想要Lucky正常工作实体类的编写一定要符合如下规范：**

***1. 实体类的属性名必须与表的字段名一致***

***2. 实体类的属性的类型必须与表字段的类型一致***

***3. 表的主键必须为int类型，且最好设置为自增长***

***4. 实体类的属性类型必须为java基本类型的包装类型（int-Integer double-Double）(重点)***

***5. 每个实体类都必须由系统生成toString()方法(重点)***

1. **如何建立实体类与表的映射关系？**

实体类创建完成后必须与对应的表建立映射关系后Lucky才能通过操作实体类来间接的操作表。

建立映射关系的实质就是通过一些方法告诉实体类表的关键信息，如：这个类对应数据库中那张表？这张表的主键是哪一个字段？

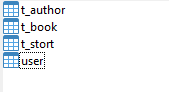
通过注解建立映射关系：

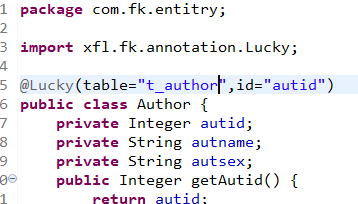
**@Lucky(table=”数据库表名”,id=”表的主键”)**

**注：@Lucky注解的必须写在在类的声明前（如图），值得一提的是如果忽略大小写后实体的类名与数据库的表名一致，这时就可以不用配置“table”属性的值。如果你和我一样，是一个怕麻烦的人，我建议你采用和我一样的偷懒方式：“在创建实体类时类名就采用表名，这样就可以减少一步配置！”**

**具体场景1(类表不同名)：**

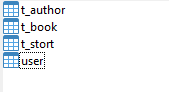
**其中表名为t\_author类名为Author**

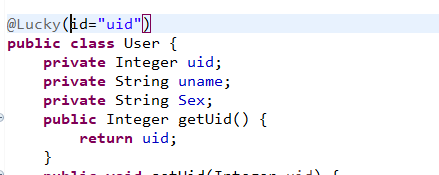
****

****

**具体场景2(类表同名)：**

**其中表名为user类名为User**

****

****

1. **如何获得操作数据库的数据控制对象（SqlControl）**

做好之前三个步骤之后我们基本上就完成了常用功能的配置，接下来就可以创建对象来操作数据库了，数据控制对象的创建需要使用静态方法**getSqlControl()**来获得：

**SqlControl sql= SqlControl. getSqlControl();**

1. **操作数据库**

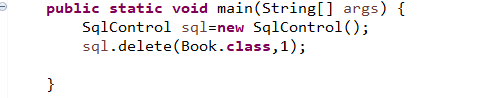
* Id操作

**--ID删除--**

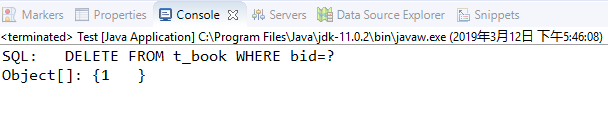
**sql.delete(实体类的class,ID)**

**具体场景：**

例如：我要删除“ID为1”的那本图书

****

**执行效果：**

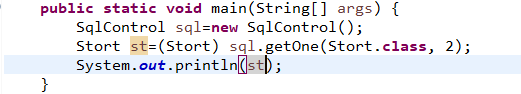
****

**--ID查询--**

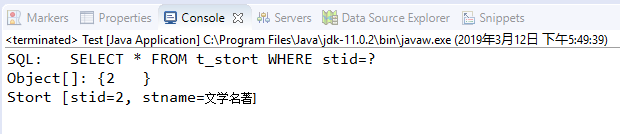
**Object obj=sql.getOne(实体类的class,ID),方法返回的而是一个Object类型的对象，想要得到实体类类型的对象就必须做一次强制类型转换**

**具体场景：**

例如：我要得到“ID为2”的图书类型信息

****

**执行效果：**

****

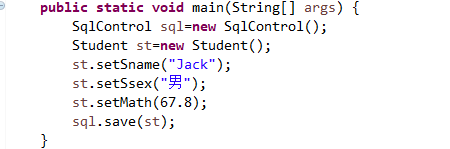
* 对象操作

**--增加操作--**

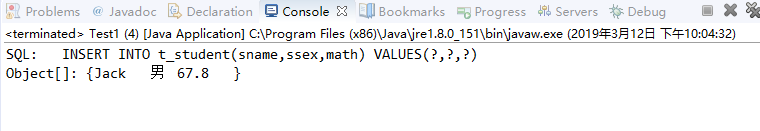
1. **将要添加的数据封装到实体类的对象中**
2. **使用sql.save(T t)方法将数据加到数据库中**

**具体场景：**

例如：我要保存”姓名”是Jack,”性别”为男，而且数学成绩是67.8的学生



**执行效果：**

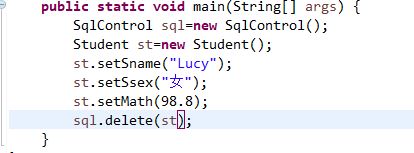


**--删除操作--**

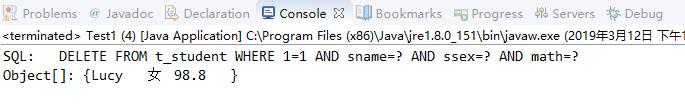
1. **将删除语句的条件封装到实体类对象中**
2. **使用sql.delete(T t)方法删除数据库中满足条件的数据**

**具体场景：**

例如：我要删除名字叫“Lucy”,且数学成绩为98.8的那个女生



**执行效果：**



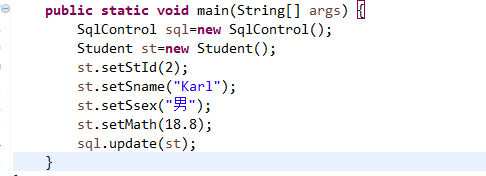
**注意：封装到对象中的所有删除条件都是由AND连接的，在删除的执行结果中就有所体现**

**--修改操作--**

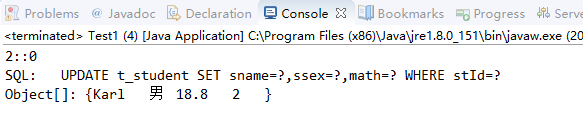
1. **将要修改后的信息封装到实体类对象中，将Id作为修改条件也封装到实体类对象中**
2. **使用sql.update(T t)方法将数据库中对应的纪录修改**

**具体场景：**

例如：我要将ID为2的那名同学的姓名改为Karl性别改为男且数学成绩改为18.8.

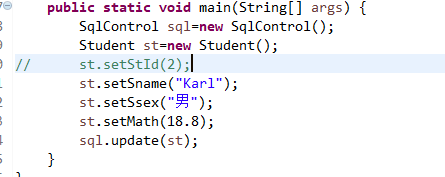


**执行效果：**

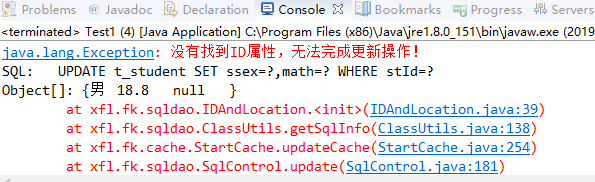


**注意：如果不设置ID值会报如下错误**

**错误场景：**



**执行结果：**

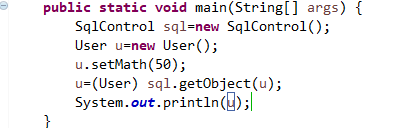


**--查询单个结果--**

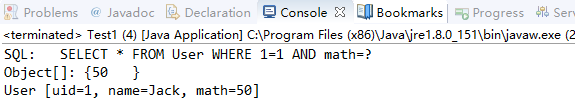
1. **如果你确定你的查询结果为单条纪录，推荐使用sql.getObject(T t)方法，因为此方法的返回值是一个Object对象。**
2. **将你的查询条件封装到实体类对象中**
3. **使用sql.getObject(T t)方法获得单条的查询记录**

**具体场景：**

例如：我很清楚我们班数学成绩为50的用户只有一个，我需要查询这个用户信息



**执行效果：**

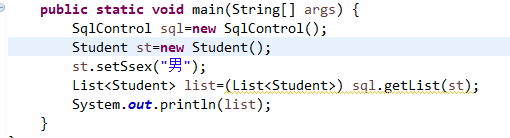


**--查询多个结果--**

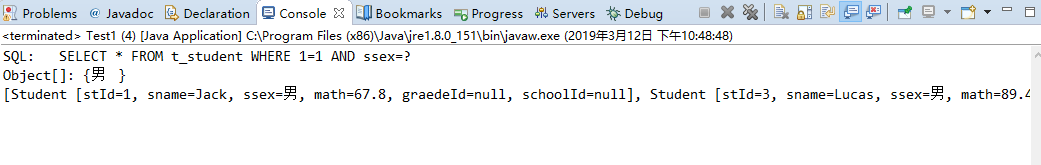
1. **如果你不确定你查询的结果是否为单条纪录，这种情况下推荐使用sql.getList()方法，此方法的返回值是一个List<?>对象。**
2. **将你的查询条件封装到实体类对象中**
3. **使用sql.getLit(T t)方法获得List集合形式的查询结果**

**具体场景：**

例如：我想得到我们班所有男生的基本信息



**执行效果：**

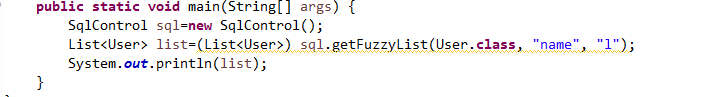


**--简单模糊查询--**

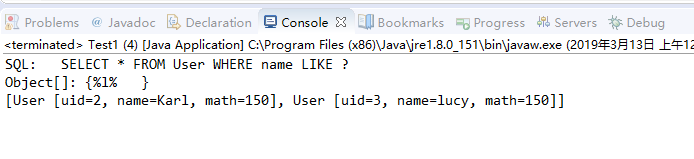
**Lucky的简单模糊查询是指模糊查询某个表的某个字段，使用sql.getFuzztListg(要查询的表对应类的Class,“需要模糊查询的字段”,“模糊查询的关键字”),该方法返回一个List<?>集合.**

**具体场景：**

例如：我想要知道用户名中带有“l”的所有用户信息



**执行效果：**

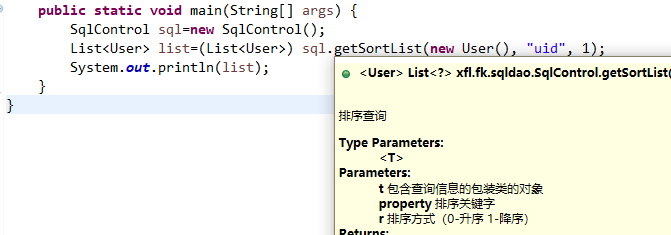


**--排序查询—**

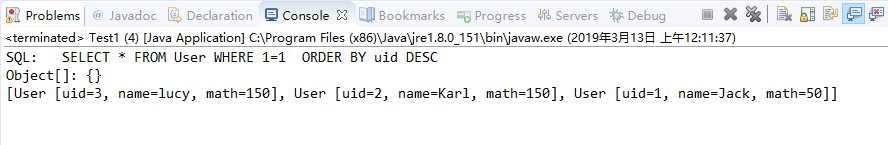
**Lucky的排序查询的方法为sql.getSortList(T t,”要排序的字段”,int(排序关键字))**

**具体场景1(降序)：**

例如：查询所有的用户，并按他们的ID降序排列

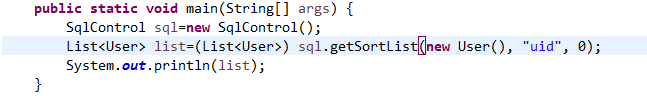


**场景1执行效果：**

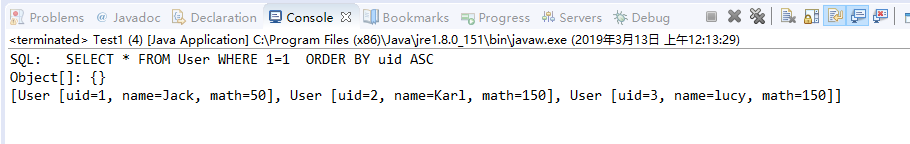


**具体场景2(升序)：**

例如：查询所有的用户，并按他们的ID升序排列



**场景2执行效果：**



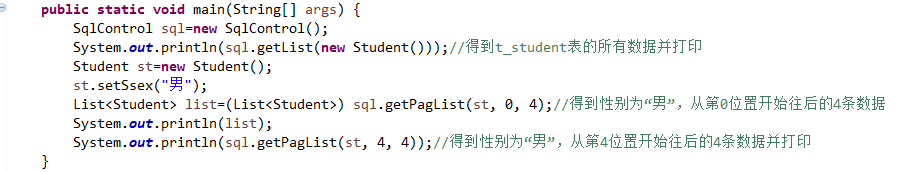
**注意：上述例子是查询user表所有的数据进行排序，其实我们也是可以加上查找条件的，如何加查询条件请参考--查询多个结果--中的查询条件封装**

**--分页查询--**

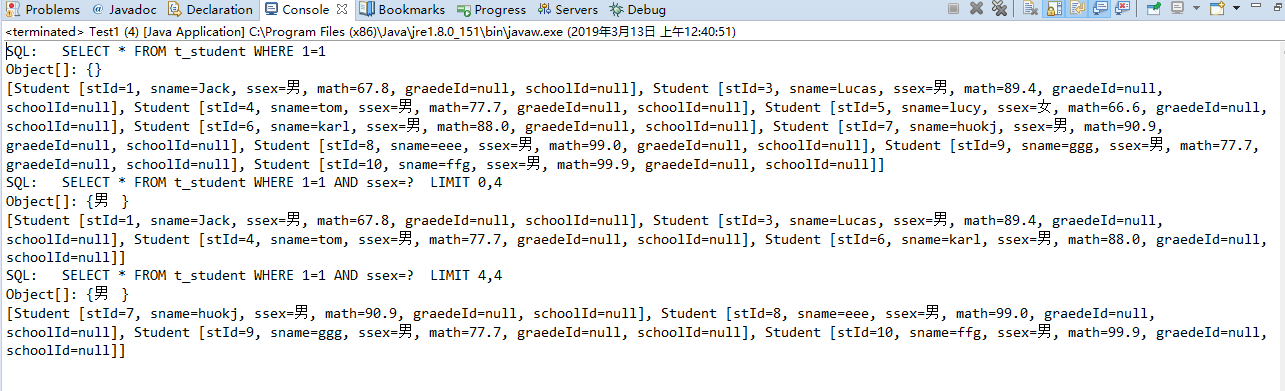
**分页查询是数据库操作中比较常用的查询方式，总所周知分页查询需要两个参数，一个是“当前页第一条记录在整张表的位置index”和“”每页需要显示的记录数size”**

**Lucky中使用sql.getPagList(T t,index,size)执行分页操作，方法返回一个List<？>集合。**

**具体场景：**



**执行效果：**



* 预编译sql操作

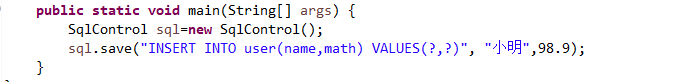
上述查询我们似乎都没有写过一条SQL语句，但是方便的背后是功能的局限，Lucky考虑到了这些问题，所以它保留了JDBC的SQL语句操作，但是也略有不同，我们不需要写完整的SQL语句，我们书写的是预编译的SQL语句。

预编译的SQL语句就是带占位符(?)的SQL语句,当然有占位符我们就要填充它，所以 “预编译的SQL语句”，和“”填充占位符的对象” 就理所应当的成了操作的参数。

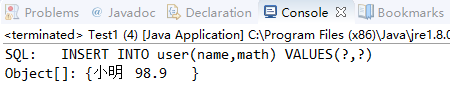
**--预编译SQL增删改--**

由于预编译SQL的增删改操作大同小异，所以就用增加操作来举例：

**具体场景：**



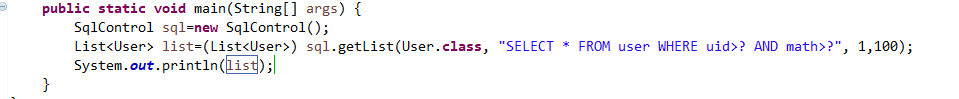
**执行效果：**



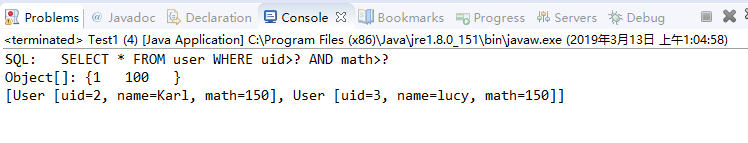
**--预编译SQL查询--**

查询操作的最大不同就是需要对结果集进行包装，所以在执行预编译SQL查询时会多出一个参数，也即 “包装类的Class”

**具体场景：**



**执行效果：**

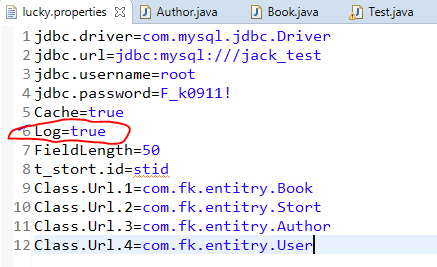


## 特色功能

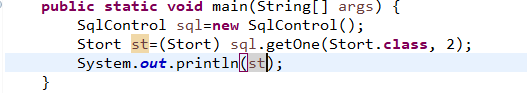
### Debug机制

这个功能算不上强大，你可以将它视作Lucky的使用小技巧。你是不是很好奇，上面所有**执行效果**的视图上怎么都有一个固定格式的输出“SQL：………和Object[ ] {……}”,我并没有打印信息，但是它却出现了？！

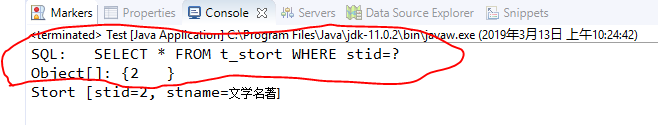
没错，接下来要介绍的小技巧就这个了，这是Lucky的Debug机制，这个小技巧是为帮助开发人员快速锁定bug而设计的，**它的使用也是需要配置的**，但配置太简单，以至于我每次都会将它遗忘！要使用这个小技巧我们只需要在lucky.properties文件中加入一个人key-value就好了：**Log=true**



**具体场景1（开启Debug）：**



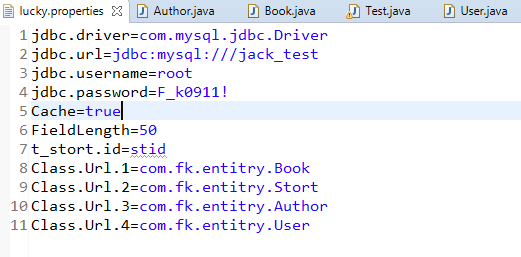
**执行效果：**



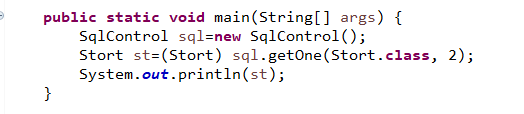
如果你是客户或者你并**不希望看到这些信息**，你只需要**删掉那行配置**就可以了

**具体场景2（关闭Debug）：**

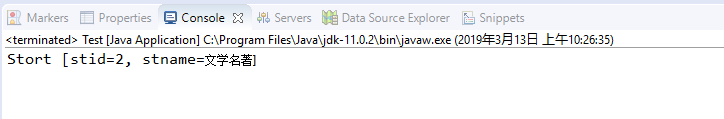
1**删除Debug配置代码：Log=true**



**2.执行相同的方法**



**执行效果（提示信息不见了）：**

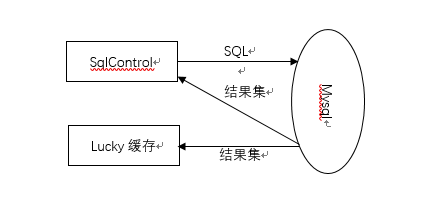


### 查询缓存

缓存机制是Lucky的一大亮点，但是它也是Lucky最不稳定的地方。

缓存机制的运行原理可以用图像来解释

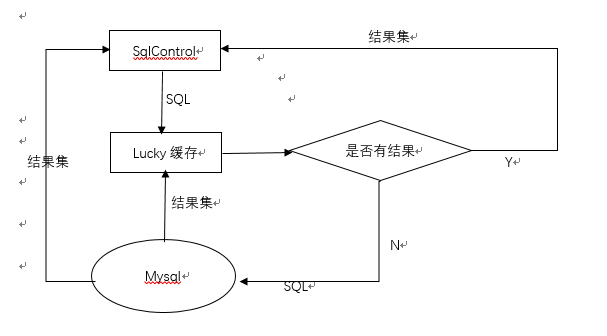
**如图1-1，为SqlControl第一次向MySql发送SQL语句的场景**



**图1-1**

当第一次发送sql语句到mysql时，结果集会被发送到两个地方：“Lucky缓存”和“SqlControl”。

**如图1-2，为之后SqlControl的查询流程场景**

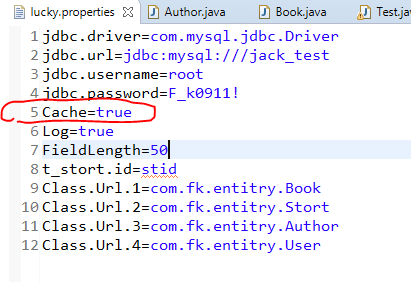
****

**图1-2**

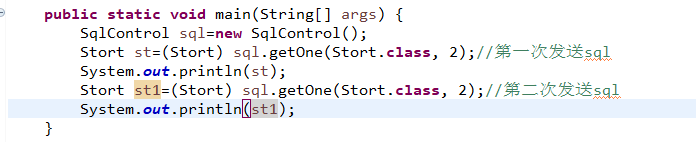
再次查询时SQL语句会首先被发送到Lucky缓存，如果在缓存中能找到对应的结果集，则直接返回结果集，如果找不到对应的结果集SQL语句才会被发送到MySql中

**--缓存机制的试用—**

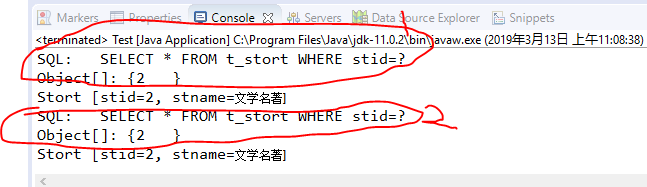
使用缓存机制需要在lucky.properties文件进行配置：**Cache=true**



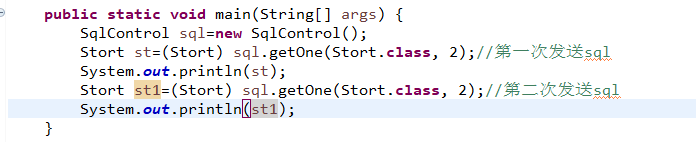
**具体场景1（不启用缓存机制）：**



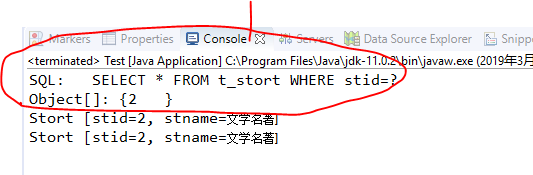
**执行效果1（两次查询同一结果，SqlControl向Mysql发送了两次Sql）：**



**具体场景2（启用缓存机制）：**



**执行效果1（两次查询同一结果，SqlControl向Mysql发送了一次Sql）：**



总结：Lucky的缓存机制可以减少对数据库的操作，所以大大降低了性能的消耗。但是Lucky的缓存机制并没有想象中的那么完美，它表现的不够稳定，所以还有待后续的优化。正因如此，Lucky的缓存机制也是可以选择使用的，**与Debug机制一样**，如果你**不想使用**只需要**删除掉配置**即可

### 自动建表机制

顾名思义，Lucky是可以帮助我们完成建表操作的，我们只需要**编写好实体类并做好相关的配置**，建表工作我们就可以放心的交给Lucky了！

Lucky建表也是需要信息的，例如：**由“我”负责创建的表有哪些**？**这些表的名字**？**这些表的主键**？**这些表的外键**？**是否需要设置级联**？建表时表**字段的长度**应该是多少才合适？

接下来的我们就来一一解决.

**--1.由“我”负责创建的表有哪些？--**

这个信息是事关重要的，而且是框架型信息，这个信息我们放到lucky.properties中Lucky就会自动的得到，我么需要这样配置：

**Class.Url.n=实体类的全路径**

**创建一张表时，其中n为任意的正整数**

**Class.Url.n+1=实体类n+1的全路径**

**Class.Url.n+2=实体类n+2的全路径**

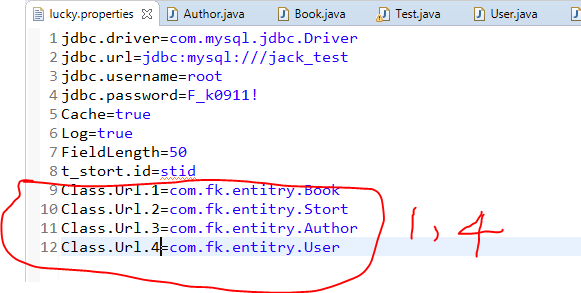
**。。。。。。。**

**Class.Url.n+m=实体类n+m的全路径**

**当需要创建m张表时：Class.Url的后缀n们必须是连续的任意正整数**

做好上述的配置后，Lucky的问题一就解决了！

**具体场景（我需要Lucky帮我创建四张表）：s**



注意：请记住这里**连续正整数**的最小值fist和最大值last，因为这个信息会在之后起到重要的作用。

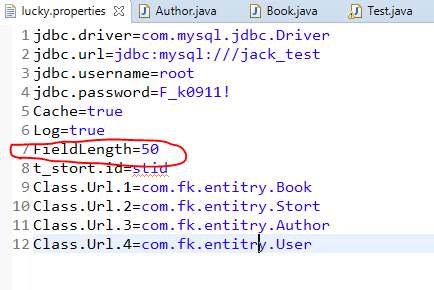
**--2.表的字段长度多少才合适？--**

这个问题的答案也是有你来决定的，也是在lucky.properties文件中配置的：

FieldLength=长度

做好上述的配置后，Lucky的问题二就解决了！

**具体场景（设置建表字段长度为50）**



注意:你不想做配置也是可以的，如果你不做配置Lucky会有一个默认的答案：**35**

**--3.这些表的名字？这些表的主键？这些表的外键？--**

这个问题的一部分答案在 **建立实体类与表的映射关系** 时就已经完成了，没错，使用到了**@Lucky()注解，**在之前我们就**已经告诉lucky表的名字和表的主键**，所以不在赘述。

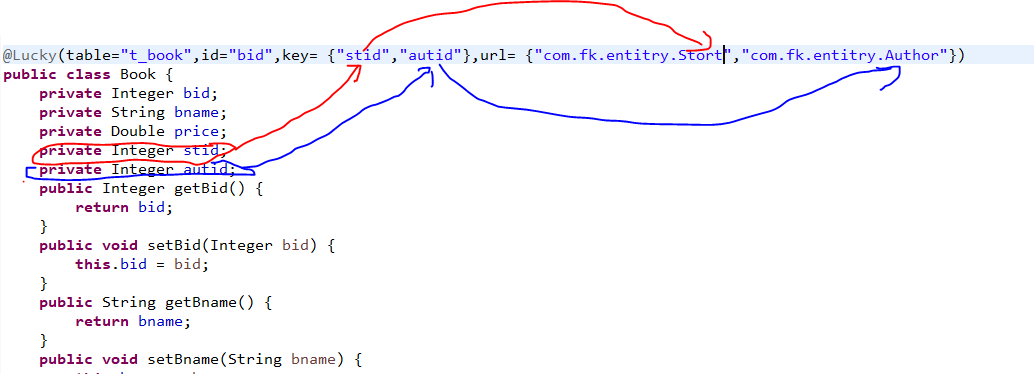
那么如何配置**外键**信息呢？答案也是**@Lucky()注解:**

**key—用来配置外键名 url—外键对应的实体类的全路径**

**@Lucky(table=”tableName”,id=”ID”,key={“外键1”,…,” 外键n”}，url={“外键1对应实体类的全路径”,…,” 外键n对应实体类的全路径”})**

做好上述的配置后，Lucky的问题三就解决了！

**具体场景（配置Book类的外键stid和autid）**



注意：key和url都是**数组形式**的，两个属性必须**成对出现**，元素也必须**一一对应**

**--4.是否需要设置级联的配置？--**

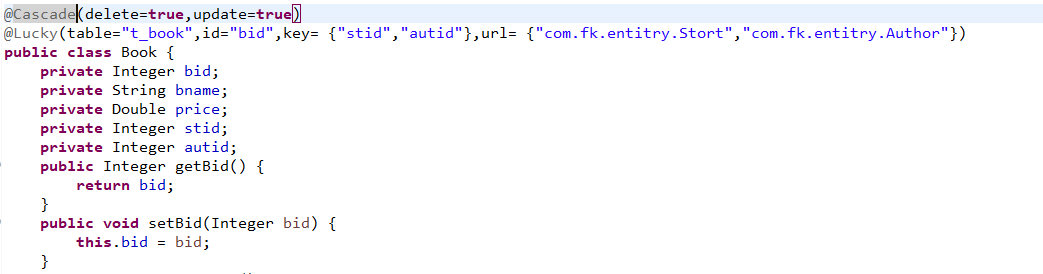
级联操作是外键的配置带来的，如果不需要级联操作可以不做配置，如果需要级联操作则需要配置，配置会引入**@Cascade()注解**

**@Cascade(delete=true,update=true)**

配置的含义显而易见，不再赘述！

做好上述的配置后，Lucky的问题四就解决了！

**具体场景（配置Book类的外键stid和autid，并设置为级联跟新和删除）：**



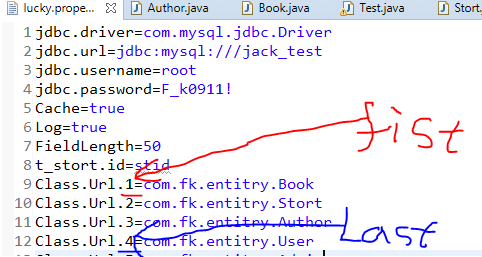
注意：@Cascaed()单独出现是没有意义的，必须配合外键的使用才会有意义，需要哪种级联就配置哪种级联，不需要则不配置

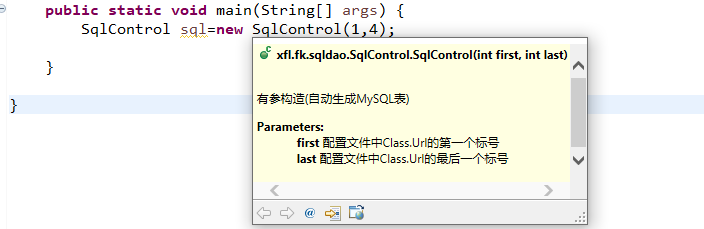
* **SqlControl有参构造建表**

在保证以上四个步骤全部完成的前提下，使用SqlControl的有参构造就可以进行建表了，那么参数是什么？哈，这时你是否还记得之前在-**-由“我”负责创建的表有哪些？--**中我提示你们记住的**最小和最大正整数fist和last呢**？，这里的参数就是这两个数.

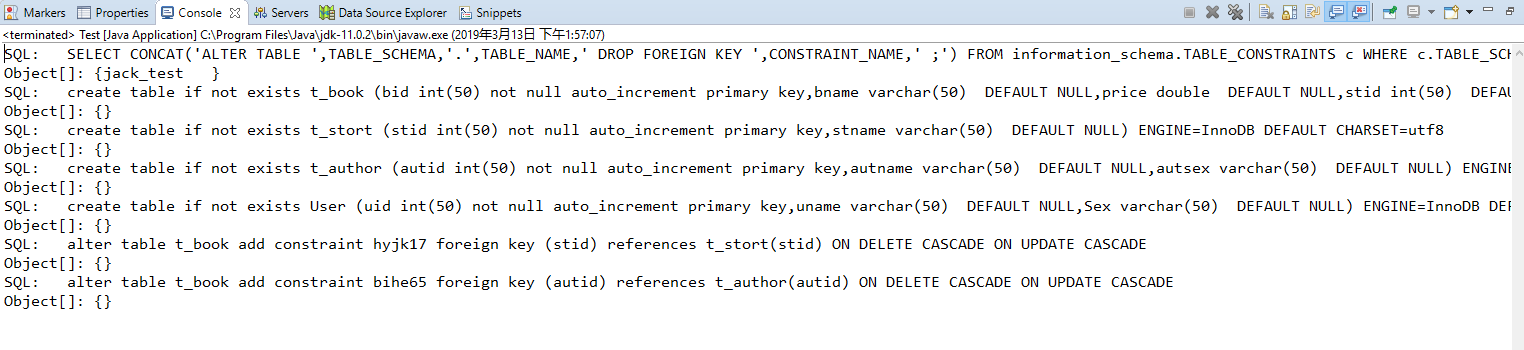
**SqlControl sql=SqlControl. getSqlControl(fist,last);**

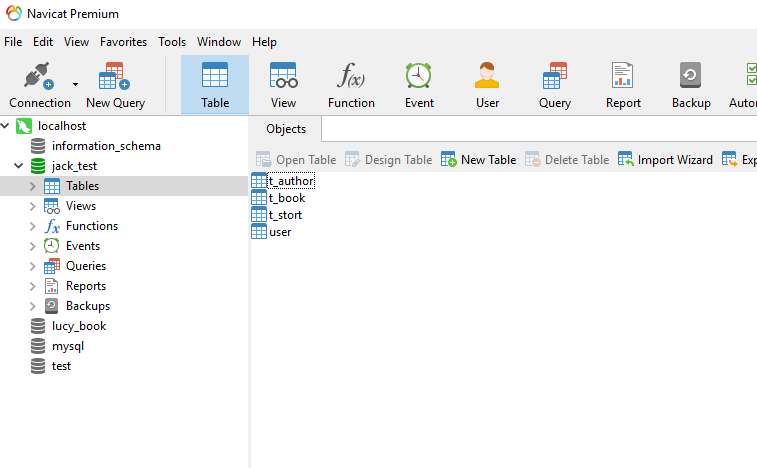
**具体场景:**



****

**执行效果:**

****

****

**注意：使用有参构造创建出的SqlControl对象也是一样可以操作数据库的，操作方法也完全一致！**

### 事务处理

数据库事务通常包含了一个序列的对数据库的读/写操作。包含有以下两个目的：

1.为数据库操作序列提供了一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常状态下仍能保持一致性的方法。

2.当多个应用程序在并发访问数据库时，可以在这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。

当事务被提交给了数据库管理系统（DBMS），则DBMS需要确保该事务中的所有操作都成功完成且其结果被永久保存在数据库中，如果事务中有的操作没有成功完成，则事务中的所有操作都需要回滚，回到事务执行前的状态；同时，该事务对数据库或者其他事务的执行无影响，所有的事务都好像在独立的运行。

**事务的性质：**

* 原子性（Atomicity）：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
* 一致性（Consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束。
* 隔离性（Isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行。
* 持久性（Durability）：已被提交的事务对数据库的修改应该永久保存在数据库中。

例子：生活中常见的转账操纵就可以看作是一个事务，我们来分析一下如下的场景：张三给李四转账1000，假设张三李四的账户初始余额都是2000。这个转账的事务其实可以看做两个实际操作：

1.张三的账户余额减少1000，

2.李四的账户余额增加1000

假设两种情况：

1. 转账成功
2. 张三的余额减少后由于网络问题转账操作被终止

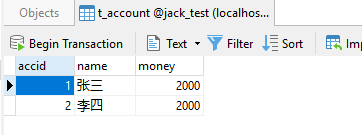
第一种情况没什么好说的，一个账户的余额增加1000，另一个减少1000即可。

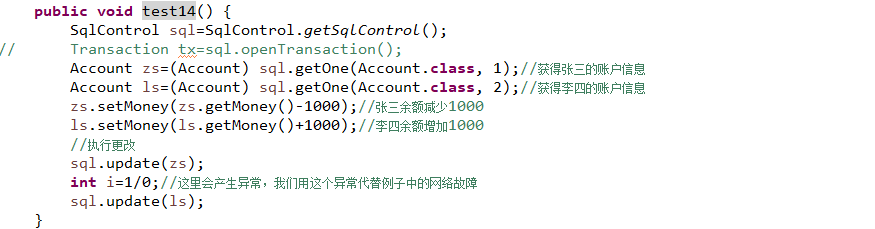
那么要是出现第二种情况，那岂不是一个余额减少了1000，另一个账户却没有增加，凭空的就少了1000元？！事务处理机制就是为了阻止这类事件发生而诞生的，Lucky的事务处理类是: **Transaction** 这个类的对象可以由**SqlControl**的**openTransaction()**获得.

**Transaction tx=sql. openTransaction();**

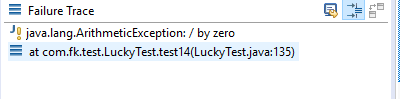
当Transaction类对象被生成时它就会记住事务的初始状态，即“张三李四的账户初始余额都是2000” 然后执行转账操作，完成操作后使用**tx.commit()**提交事务进行处理，如果出现异常就在**catch块**中使用**tx.rollback()**对事务进行回滚，方法执行后事务将回到初始状态，也即，第二种情况将不会发生

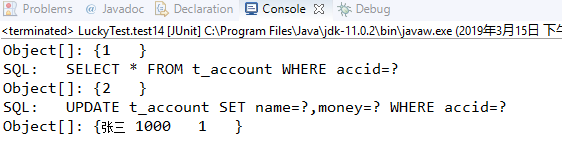
**具体场景1（不使用事务处理机制）:**

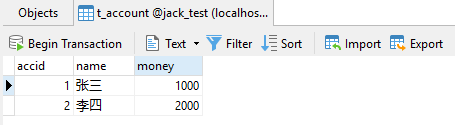




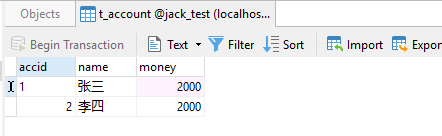
**执行效果（只减少不增加，凭空少了1000元）:**

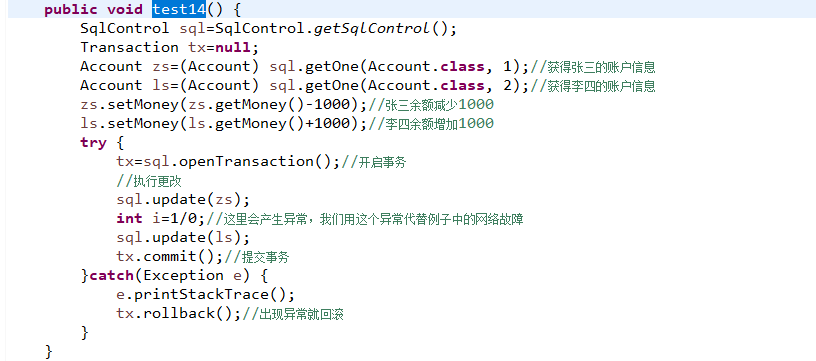






**具体场景1（使用事务处理机制）:**





**执行效果（正常回滚s’s’s）:**

