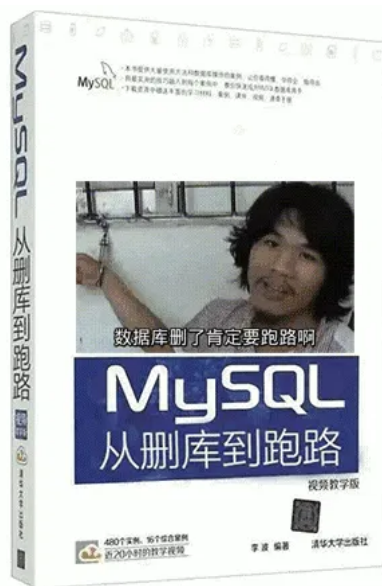


数据库

[mysql下载地址](#)



1.数据库介绍

数据库 (database) 是用来组织、存储和管理数据的仓库

当今世界是一个充满着数据的互联网世界，充斥着大量的数据。数据的来源有很多，比如出行记录、消费记录、浏览的网页、发送的消息等等。除了文本类型的数据，图像、音乐、声音都是数据

为了方便管理互联网世界中的数据，就有了数据库管理系统的概念（简称：数据库）用户可以对数据库中的数据进行新增、查询、更新、删除等操作

对于后端的开发来说，接触数据库自然是不可或缺的组成部分，几乎所有的数据流转都是靠后端直接或间接从数据库拿数据进行对数据的逻辑处理和流转让整个系统活起来，从而实现目标业务功能

本节课的目标在于了解数据库及其分类，直接通过控制台对数据库进行基础操作，数据库可视化工具，`golang` 操控数据库

2.数据库分类

数据库主要分为**关系型数据库**或者 `SQL 数据库(RDBMS)`和

非关系型数据库或者 `NoSQL 数据库(Non-RDBMS)`

关系型数据库，例如 `mysql`，`Oracle`，`SQL Server` 他们的设计理念相同，用法比较类似

非关系型数据库，例如 `redis`，非关系型数据库在一定程度上弥补了关系型数据库的缺陷

还有一些数据库例如 `Mongodb`，是一个基于分布式文件存储的数据库，是介于关系型和非关系型之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的

为什么说非关系型数据库在一定程度上弥补了关系型数据库的缺陷？

关系型数据库的最大优点就是**事务的一致性**，这个特性，使得关系型数据库中可以适用于一切要求一致性比较高的系统中。比如：银行系统。

但是在网页应用中，对这种一致性的要求不是那么的严格，允许有一定的时间间隔，所以关系型数据库这个特点不是那么的重要了。相反，关系型数据库为了维护一致性所付出的巨大代价就是读写性能比较差。而像微博、facebook 这类应用，对于**并发读写能力要求极高**，关系型数据库已经无法应付。所以必须用一种新的数据结构存储来替代关系型数据库。所以**非关系型数据库**应用而生。

目前许多大型互联网都会选用 MySQL+NoSql 的组合方案，因为 SQL 和 NoSql 都有各自的优缺点。

3. RDBMS

既然有了关系数据库这一概念，我们就需要一种系统去管理他，这就是**关系数据库管理系统**(简称 RDBMS)

RDBMS 的**特点**:

1.数据以表单的形式出现，若干的表单组成database

Database
information_schema
mysql
performance_schema
sys

上面是一个初始MySQL包含的database (**初始自带**)

2.每为各种记录名称，每列为记录名称所对应的数据域，许多的行和列组成一张表单

Field	Type	Null	Key	Default	Extra
username	varchar(20)	NO		NULL	
password	varchar(100)	NO		NULL	
money	bigint	YES		0	
age	int	YES		0	
sex	varchar(20)	YES			
true_name	varchar(20)	YES			
career	varchar(20)	YES			

上图是一个购物网站服务器数据库中user表的信息，我们先简单看一下各个字段的含义

Field：在数据库中，大多数时，表的列称为**字段**，每个字段包含某一专题的信息。就像通讯录数据库中，姓名、联系电话，这些都是表中所有行共有的属性，所以把这些列称为姓名字段和联系电话字段。在此表中的字段是指Type、Null等

Type：**数据类型**，上表中 username 的 Type 为 varchar(20)，即变长字符串（括号内的数字表示该字段最多能存储的字符数量）

Null：是否能为**空值**（注意：空值与零长度字符串不一样）

Key：数据库的物理结构，**键值**，分为 primary key、unique key、foreign key

primary key：主键约束是一个单独的列或者多个列的组合，其值能唯一地标识表中的一行数据

foreign key：外键相对于主键而言，**用于引用其他表**。外键通过子表的一个或多个列对应父表的主键或唯一键值，将子表的行和父表行建立起关联关系

unique key：唯一键也是一个常用的约束，用来保证表中的一列或几列的中的**值是唯一的**

Default：**字段默认值**，在表中插入一条新记录时，如果没有为某个字段赋值，系统就会自动为这个字段插入默认值

Extra：包含不适合在其他列中显示但十分重要的额外信息

数据类型

MySQL 支持所有标准 SQL 数值数据类型。主要包括数值、日期/时间和字符串(字符)类型

数值类型

类型	大小	范围（有符号）	范围（无符号）	用途
TINYINT	1 Bytes	(-128, 127)	(0, 255)	小整数值
SMALLINT	2 Bytes	(-32 768, 32 767)	(0, 65 535)	大整数值
MEDIUMINT	3 Bytes	(-8 388 608, 8 388 607)	(0, 16 777 215)	大整数值
INT或 INTEGER	4 Bytes	(-2 147 483 648, 2 147 483 647)	(0, 4 294 967 295)	大整数值
BIGINT	8 Bytes	(-9,223,372,036,854,775,808, 9 223 372 036 854 775 807)	(0, 18 446 744 073 709 551 615)	极大整数值
FLOAT	4 Bytes	(-3.402 823 466 E+38, -1.175 494 351 E-38), 0, (1.175 494 351 E-38, 3.402 823 466 351 E+38)	0, (1.175 494 351 E-38, 3.402 823 466 E+38)	单精度浮点数值

类型	大小	范围（有符号）	范围（无符号）	用途
DOUBLE	8 Bytes	(-1.797 693 134 862 315 7 E+308, -2.225 073 858 507 201 4 E-308), 0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	双精度浮点数值
DECIMAL	对 DECIMAL(M,D) , 如果M>D, 为 M+2否则为D+2	依赖于M和D的值	依赖于M 和D的值	小数值

日期和时间类型

类型	大小 (bytes)	范围	格式	用途
DATE	3	1000-01-01/9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59'/'838:59:59'	HH:MM:SS	时间值 或持续时间
YEAR	1	1901/2155	YYYY	年份值
DATETIME	8	'1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'	YYYY-MM-DD hh:mm:ss	混合日期和时间值
TIMESTAMP	4	'1970-01-01 00:00:01' UTC 到 '2038-01-19 03:14:07' UTC结束时间是第 2147483647 秒，北京时间 2038-1-19 11:14:07 ，格林尼治时间 2038年1月19日凌晨 03:14:07	YYYY-MM-DD hh:mm:ss	混合日期和时间值， 时间戳

字符串类型

类型	大小	用途
CHAR	0-255 bytes	定长字符串
VARCHAR	0-65535 bytes	变长字符串

类型	大小	用途
TINYTEXT	0-255 bytes	短文本字符串
TEXT	0-65 535 bytes	长文本数据
MEDIUMTEXT	0-16 777 215 bytes	中等长度文本数据
LONGTEXT	0-4 294 967 295 bytes	极大文本数据
TINYBLOB	0-255 bytes	不超过 255 个字符的二进制字符串
BLOB	0-65 535 bytes	二进制形式的长文本数据
MEDIUMBLOB	0-16 777 215 bytes	二进制形式的中等长度文本数据
LOBLOB	0-4 294 967 295 bytes	二进制形式的 极大文本数据

这也太多了罢



其实不用全部记得 🤖

终端数据库操作

登录

```
mysql -u root(你安装时设置的用户名) -p
```

⚠️ 注意：在安装好mysql时不一定会把mysql目录加入环境变量，若要实现上述登录操作请先将mysql目录加入环境变量中

或者在终端打开mysql安装目录下的bin文件夹执行 `.\mysql -u root -p`

这里的root指的是你在安装时设置的用户名，回车后会让你输入数据库密码进行验证

在登录之后就可以操作 MySQL。前文说过 RDBMS 是用来管理数据库的，那怎样实现用 RDBMS 来管理数据库，就是结构化查询语言 (Structured Query Language 简称 SQL)。通过使用 SQL 操作 RDBMS 来实现管理数据库。

SQL

与关系型数据库有关的 SQL 命令包括 CREATE、SELECT、INSERT、UPDATE、DELETE、DROP 等，根据其特性，可以将它们分为以下几个类别。

DDL - Data Definition Language，数据定义语言

对数据的结构和形式进行定义，一般用于数据库和表的创建、删除、修改等。

命令	说明
CREATE	用于在数据库中创建一个新表、一个视图或者其它对象。
ALTER	用于修改现有的数据库，比如表、记录。
DROP	用于删除整个表、视图或者数据库中的其它对象

查看数据库

```
show databases;
```

创建数据库

```
create database dbname;
```

删除数据库

```
drop database <数据库名>;
```

创建表

```
CREATE TABLE tbl_name
  (create_definition,...);
```

- 创建定义 (create_definition)
col_name data_type
data_type 表示列定义中的数据类型，以及该类型的约束的信息
常见约束：
 - NOT NULL - 指示某列不能存储 NULL 值。
 - UNIQUE - 保证某列的每行必须有唯一的值。
 - PRIMARY KEY - NOT NULL 和 UNIQUE 的结合。确保某列（或两个列多个列的结合）有唯一标识，有助于更容易更快速地找到表中的一个特定的记录。
 - FOREIGN KEY - 保证一个表中的数据匹配另一个表中的值的参照完整性。
 - CHECK - 保证列中的值符合指定的条件。
 - DEFAULT - 规定没有给列赋值时的默认值。

```
CREATE TABLE `student_score` (  
  `id` BIGINT(20) NOT NULL AUTO_INCREMENT,  
  `name` VARCHAR(20) DEFAULT '',  
  `math` INT(11) DEFAULT 0,  
  `english` INT(11) DEFAULT 0,  
  PRIMARY KEY(`id`)  
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

这里直接执行会存在问题

```
ERROR 1046 (3D000): No database selected
```

show databases; 先查看有哪些数据库 use dbname; 然后**选中数据库**（可以理解为进入此数据库，dbname为之前创建的数据库），再执行就不会报错了

修改表名

```
alter table <旧表名> rename <新表名>;
```

添加列

```
alter table tablename add <列名> <类型>;
```

修改列

```
alter table tablename change column <旧列名> <新列名> <新列类型>;  
--更改  
alter table tablename modify column <列名> <新列类型>;
```

删除列

```
alter table tablename drop column <列名>;
```

查看所有表

```
show tables;
```

查看数据表的信息

```
desc <tablename>;
```

此处的信息不是指具体数据，而是之前创建表时设置的字段

DML - Data Manipulation Language，数据处理语言

对数据库中的数据进行处理，一般用于数据项（记录）的插入、删除、修改和查询。

插入

```
insert into <表名> values (值,...,...);
insert into <表名> (列名,...) values (值,...,...);
eg:
insert into score (name,math,gender) values ("雷雨昊",50,"fm");
```

删除

```
delete from <表名> [where <表达式>] -- 不带where 语句的是全部删除
```

修改

```
update <表名> set <字段> =<值> [, <字段> =<值>...] [where 表达式]
```

查询

```
select * from <表名>; -- 查找所有
select <字段> [, <字段>...] from <表名>; -- 查找指定字段
select distinct [<字段>... 或者 *] from <表名>; -- distinct关键字用于查询不重复的记录

eg:
select * from score; -- 查找所有在score表中的数据
select name from score; -- 查找在score表中name的所有数据
select name,math from score; -- 查找在score表中name和math的所有数据
select distinct gender from score; -- 只查找在score表中的gender的类别但不输出各占有多少数量
```

Where 子句运算符	描述
=	等于
<>	不等于。在 SQL 的一些版本中，该操作符可被写成 !=
>	大于
<	小于
>=	大于等于
<=	小于等于
BETWEEN	在某个范围内
LIKE	搜索某种模式（模糊搜索）
IN	指定针对某个列的多个可能值

逻辑运算符	描述
And	与 同时满足两个条件的值
Or	或 满足其中一个条件的值(⚠ And运算级别大于Or)
Not	非 满足不包含该条件的值

ORDER BY 关键字

对检索的数据进行排序

`order by <列名...>;` 按照列名升序排序

`order by <列名...> desc;` 按照列名降序排序

SQL 函数	描述
AVG()	返回平均值
COUNT()	返回行数
FIRST()	返回第一个记录的值
LAST()	返回最后一个记录的值
MAX()	返回最大值
MIN()	返回最小值
SUM()	返回总和

MVCC

事务

事务是由单独单元的一个或多个sql语句组成，在这个单元中，每个sql语句都是相互依赖的。而整个单独单元是作为一个不可分割的整体存在，类似于物理当中的原子（一种不可分割的最小单位）。

往通俗的讲就是，事务就是一个整体，里面的内容要么都执行成功，要么都不成功。不可能存在部分执行成功而部分执行不成功的情况。

就是说如果单元中某条sql语句一旦执行失败或者产生错误，那么整个单元将会回滚(返回最初状态)。所有受到影响的数据将返回到事务开始之前的状态，但是如果单元中的所有sql语句都执行成功的话，那么该事务也就会被顺利执行。

事务的四个特性

- 原子性 (Atomicity)：指事务是一个不可分割的最小工作单位，事务中的操作只有都发生和都不发生两种情况
- 一致性 (Consistency)：事务必须使数据库从一个一致状态变换到另外一个一致状态，举一个栗子，李二给王五转账50元，其事务就是让李二账户上减去50元，王五账户上加上50元；一致性是指其他事务看到的情况是要么李二还没有给王五转账的状态，要么王五已经成功接收到李二的50元转账。而对于李二少了50元，王五还没加上50元这个中间状态是不可见的。
- 隔离性 (Isolation)：一个事务的执行不能被其他事务干扰，即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性 (Durability)：一个事务一旦提交成功，它对数据库中数据的改变将是永久性的，接下来的其他操作或故障不应对其有任何影响。

正式开启

首先需要禁用自动提交功能，也就是设置 `autocommit` 变量值为0（0:禁用 1:开启）

先查看当前状态：select @@autocommit;

```
+-----+
| @@autocommit |
+-----+
|             1 |
+-----+
```

禁用自动提交事务的功能并查看当前状态：set autocommit = 0;

```
+-----+
| @@autocommit |
+-----+
|             0 |
+-----+
```

开启事务

```
#步骤一：开启事务（可选）
start transaction;
#步骤二：编写事务中的sql语句（insert、update、delete）
update stu set math = 100 where name = "张三";
update stu set math = 60 where name = "王五";
#步骤三：结束事务
commit; #提交事务
```

我们可以发现，在当前事务还未提交的时候，此事务内部可以看到更新后的结果，但另一个事务却无法看到修改后的值

```
mysql> select * from stu;
+----+-----+-----+-----+
| id | name | math | english |
+----+-----+-----+-----+
|  1 | 张三 | 100  |      80 |
|  2 | 李四 |  80  |      90 |
|  3 | 王五 |  60  |      60 |
+----+-----+-----+-----+
3 rows in set (0.00 sec)
#另一个事务提交
mysql> select * from stu;
+----+-----+-----+-----+
| id | name | math | english |
+----+-----+-----+-----+
|  1 | 张三 |  90  |      80 |
|  2 | 李四 |  80  |      90 |
+----+-----+-----+-----+
```

事务并发问题

因为某一刻不可能总只有一个事务在运行，可能出现A在操作stu表中的数据，B也同样在操作stu表，那么就会出现并发问题，对于同时运行的多个事务，当这些事务访问数据库中相同的数据时，如果没有采用必要的隔离机制，就会发生以下各种并发问题

1. 脏读：对于两个事务T1，T2，T1读取了已经被T2更新但还没有被提交的字段之后，若T2回滚，T1读取的内容就是临时且无效的
2. 不可重复读：对于两个事务T1，T2，T1读取了一个字段，然后T2更新了该字段之后，T1在读取同一个字段，值就不同了
3. 幻读：对于两个事务T1，T2，T1在A表中读取了一个字段，然后T2又在A表中插入了一些新的数据时，T1再读取该表时，就会发现莫名多出几行了

所以，为了避免以上出现的各种并发问题，我们就必然要采取一些手段。mysql数据库系统提供了四种事务的隔离级别，用来隔离并发运行各个事务，使得它们相互不受影响，这就是数据库事务的隔离性。

隔离级别

1. read uncommitted（读未提交数据）：允许事务读取未被其他事务提交的变更。（脏读、不可重复读和幻读的问题都会出现）。
2. read committed（读已提交数据）：只允许事务读取已经被其他事务提交的变更。（可以避免脏读，但不可重复读和幻读的问题仍然可能出现）
3. repeatable read（）：确保事务可以多次从一个字段中读取相同的值，在这个事务持续期间，禁止其他事务对这个字段进行更新(update)。（可以避免脏读和不可重复读，但幻读仍然存在，也就是可能多几行）
4. serializable（串行化）：确保事务可以从一个表中读取相同的行，在这个事务持续期间，禁止其他事务对该表执行插入、更新和删除操作，所有并发问题都可避免，但性能十分低下（因为你未完成就都不可以弄，效率太低）

mysql 的默认事务隔离级别是：repeatable read

一个事务与其他事务隔离的程度称为隔离级别。数据库规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但并发性就越差

可重复读：对某字段进行操作时，其他事务禁止操作该字段。它总能保持你读取的数据是一致的

程序1操作 stu 表时，程序2是无权对 stu 表进行任何操作，如果强行操作的话，就会发生error，这里是因为update语句会默认添加独占锁，如果其他事务对持有独占锁的记录进行修改时，就会被阻塞。

“ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction”;

其中文意思是：“超过锁定等待超时；尝试重新启动事务”

只有当程序1对 stu 表操作完成后（结束事务后），程序2才可以对 stu 表进行操作

多版本并发控制

MVCC 是一种并发控制方法，一般在数据库管理系统中，实现数据库的并发访问，在编程语言中实现事务内存。

总结：主要为了提升并发性能

数据库原生锁

最原生的锁，锁住一个资源后会禁止其他任何线程访问同一个资源。但是很多应用的一个特点都是读多写少的场景，很多数据的读取次数远大于修改的次数，而读取数据间互相排斥显得不是很必要。

读写锁

读锁和读锁之间不互斥，而写锁和写锁、读锁都互斥。这样就很大提升了系统的并发能力。之后人们发现并发读还是不够

mvcc 概念

能不能让读写之间也不冲突的方法，就是读取数据时通过一种类似快照的方式将数据保存下来，这样读锁就和写锁不冲突了，不同的事务session会看到自己特定版本的数据。当然快照是一种概念模型，不同的数据库可能用不同的方式来实现这种功能

Go语言操作 MySQL

准备工作

首先下载

```
go get github.com/go-sql-driver/mysql
```

导入驱动

请匿名加载驱动程序，将其包限定符别名为 `_`，以便其导出的名称对我们的代码不可见。在后台，驱动程序将自身注册为对包可用，但通常除了运行`init`函数之外，不会发生任何其他情况

```
import (  
    "database/sql"  
    _ "github.com/go-sql-driver/mysql"  
)
```

[golang语法中下划线](#)

访问数据库

```
var dns = "[用户名[:密码]@][协议[(地址:端口)]]/dbname[?  
param1=value1&...&paramN=valueN]" //DSN（数据源名称）  
db, err := sql.Open("mysql", dns) // "mysql"指定目标数据库类型  
  
//dns示例: "root:root@tcp(127.0.0.1:3306)/test"  
// "test"为数据库名称  
//本地的mysql默认地址为127.0.0.1:3306 //3306为默认端口  
//root为用户名，后一个root为密码示例
```

```
//下面是一个较完整配置的连接
//其中后面的参数用于定制连接行为
//charset: 设置字符集, 例如utf8mb4
//parseTime: 如果设为true, 则MySQL中的DATE和DATETIME类型会被解析为Go的time.Time类型
//loc: 用于解析时间的位置信息, 如Asia/Shanghai
```

```
dsn := "user:password@tcp(127.0.0.1:3306)/dbname?
charset=utf8mb4&parseTime=True&loc=Local"
```

```
//可选参数有:
//timeout: 连接超时时间, 例如30s。
//readTimeout 和 writeTimeout: 读写操作的超时时间。
//collation: 指定字符序。
//maxAllowedPacket: 允许的最大数据包大小。
//tls: 是否启用TLS/SSL加密, 可选值包括skip-verify, preferred, required等
```

这里会返回一个 `*sql.DB`, 它不会建立与数据库的任何连接, 也不会验证驱动程序连接参数。相反, 它只是准备数据库抽象以供以后使用。与基础数据存储的第一个实际连接将在首次需要时延迟建立。如果要立即检查数据库是否可用且可访问 (例如, 检查是否可以建立网络连接并登录), 请使用 `db.Ping()` 执行此操作, 并记住检查错误:

```
err = db.Ping()
if err != nil {
    // do something here
    //log.Println("open database error: ",err)
}
```

检索数据

多行检索

`db.Query` Query 执行返回行 (通常为 SELECT) 的查询, 需要通过 `row.Next` 迭代查询, 同时 `Scan` 需要引用

```
rows, err := db.Query("select * from stu where id= ?", 1)
//在 Query 语句中 ? 表示占位符, 在这里同样的效果就是
// select * from stu where id= 1
if err != nil {
    log.Println(err)
    return
}
// 延迟调用关闭rows释放持有的数据库链接
defer rows.Close()
var user struct {
    name    string
    id      int
    math    int
    english int
}
// 迭代查询获取数据, 必须调用
for rows.Next() {
    // row.scan 必须按照先后顺序 &获取数据
```

```

err := rows.Scan(&user.id, &user.name, &user.math, &user.english)
if err != nil {
    log.Println(err)
    return
}
fmt.Println(user)
}

```

单行检索

单行查询 `db.QueryRow()` 执行一次查询，并期望返回最多一行结果（即Row）。`QueryRow`总是返回非nil的值，直到返回值的`Scan`方法被调用时，才会返回被延迟的错误。（如：未找到结果）

```

row := db.QueryRow("select * from stu where id=?", 20)
var user struct {
    name    string
    id      int
    math    int
    english int
}
// 需要注意在执行Scan的时候需要按照获取的元素个数获取
// 同时需要加上&符号(取地址) 按照先后顺序查询
err = row.Scan(&user.id, &user.name, &user.math, &user.english)
if err != nil {
    log.Println(err)
    return
}
fmt.Println(user)

```

不会真有人用Scan输入数据库罢(手动滑稽)



插入数据

我们使用 `Exec()` 实现 `INSERT UPDATE DELETE`，其源代码定义如下

```

func (db *DB) Exec(query string, args ...interface{}) (Result, error)

```

`Exec`执行一次命令（包括 删除、更新、插入等），返回的`Result`是对已执行的SQL命令的总结

```

    result, err := db.Exec("insert into stu (name ,math ,english) value (?,?,?)",
"小j", 100, 120)
    if err != nil {
        log.Println(err)
        return
    }
    // 返回新插入数据的id
    result.LastInsertId()
    // 返回影响的行数
    result.RowsAffected()

```

删除数据

```

result, err := db.Exec("delete from stu where id=?", 8)
if err != nil {
    log.Println(err)
    return
}
// 返回新插入数据的id
result.LastInsertId()
// 返回影响的行数
result.RowsAffected()

```

更新数据

```

result, err := db.Exec("update stu set math=? where name=?", 144, "小红")
if err != nil {
    log.Println(err)
    return
}
// 返回新插入数据的id
result.LastInsertId()
// 返回影响的行数
result.RowsAffected()

```

预处理

预处理是 MySQL 为了防止客户端频繁请求的一种技术，是对相同处理语句进行预先加载在 MySQL 中，将操作变量数据用占位符来代替，减少对 MySQL 的频繁请求，使得服务器高效运行。

- 即时SQL 一条 SQL 直接是流程处理，一次编译，单次运行
- 预处理SQL 一次编译、多次运行，省去了解析优化等过程；此外某种程度上能防止 SQL 注入

database/sql 中使用下面的 Prepare 方法来实现预处理操作

```

func (db *DB) Prepare(query string) (*Stmt, error)

```

Prepare 方法会先将sql语句发送给 MySQL 服务端，返回一个准备好的状态用于之后的查询和命令。返回值可以同时执行多个查询和命令。

查询

```
rows, err := db.Query("select * from stu where id= ?", 1)
```

等价于

```
var u User
stmt, err := db.Prepare("select * from score where id=?")
if err != nil {
    log.Println(err)
    return
}
row, err := stmt.Query(1)
if err != nil {
    log.Println(err)
    return
}

//前面部分
// 必须调用next
for row.Next() {
    //不要忘记& 符号和顺序
    err := row.Scan(&u.id, &u.name, &u.math, &u.english)
    if err != nil {
        log.Println(err)
        return
    }
}
fmt.Println(u)
row, err = stmt.Query(2)
if err != nil {
    log.Println(err)
    return
}
for row.Next() {
    err = row.Scan(&u.id, &u.name, &u.math, &u.english)
    if err != nil {
        log.Println(err)
        return
    }
}


fmt.Println(u)
```

虽然看起来多了几行，但是上面的 stmt 对象可以**多次使用**，而且查询**效率更高**

最后希望同学们要完成作业哦

课后作业

- lv0 复习课上所学操作，多少得复现一遍
- lv1 学生管理系统接入 `mysql`

据说上次课让你们手搓了个学生管理系统(手动滑稽) 

这次请根据自己学生的结构体进行数据库表结构创建并修改添加学生接口，通过postman等测试工具上传几个学生数据并到数据库查看

新增删除学生信息接口： `Delete/deleteStudent`

⚠ 需要提交数据库数据截图，添加后和删除后

例如：

```
//假设你的学生结构体为
type Student Struct{
    ID      int
    Name    string
    Gender  bool
    Magor   string
    .
    .
    .
}
//根据数据库数据类型进行设置表属性(通过可视化数据库操作或者终端命令操作)
//推荐可视化操作，当然你要终端命令操作我也拦不住你：)
```

- lv2 新增登录接口和注册接口

要求：使用数据库存储用户数据和密码(明文)，登录时校验用户名和密码是否正确并且合理，注册时将用户数据和密码存入数据库

⚠ 需要考虑用户数据和密码的整体性

Post/login 登录


Post/register 注册

- [可选]lv3 学有余力的同学可以把上述学生管理系统添加：

指定修改某学生的所有信息

指定修改某学生某一项信息

指定删除(初始化)某学生的某一项信息

密码总有忘记的时候，设置一个密保问题当二重保险罢 

- [可选]lv4 可以去了解一下Redis缓存机制