JavaSE

一.final 关键字

修饰类，类不能被继承

修饰方法，最终方法，方法不能被覆写

修饰属性：不可变属性，只能被初始化一次的属性

修饰变量：若是基本数据类型，则值不能改变，如果是引用数据类型，引用指向的对象的内容是可以改变的，但是不能让这个引用指向其他对象

二.继承（由共性衍生出特性的一种途径）

可以扩展已存在的代码模块（类），可以有效实现代码服用，避免重复代码的出现

1.java 单继承

2.执行顺序

先执行父类，再执行子类，先静态在动态

初始化定义和构造代码块按书写顺序执行

父类的构造只有一次，也就是 super()只执行一次

3.static 关键字（只出现在成员级别）

和对象解绑，不修饰局部变量，

静态属性：和对象解绑，成员之间共享的，发生在类加载时，在方法区（跟着类，保存在类所在的位置）

静态方法：无法调用普通方法，无法访问普通属性，无法使用 this

还可以修饰代码块

修饰类，做内部内部类的时候，也是和对象解绑

静态内部类和非静态内部类的区别：

前者内部可以声明 static 成员的，而后者不能声明 static 成员，因为本身是内部类，无法脱离外部

前者只能访问外部类的静态成员，后者都可以访问

后者不能脱离于外部的实体

4.this 的作用

访问自身的属性和方法

调用其他的构造方法（必须出现在构造方法的第一行）

代表对象本身

5.super　关键字作用

可以访问父类对象的属性和方法

调用父类的构造方法（有参时，一定出现在本类构造方法的第一行，无参时，不用写，系统默认调用，父类既有有参构造方法，又有无参构造方法的时候，不写的话，系统还是默认的调用无参，写有参局嗲用有参）（super() 括号里写构造方法的参数，会自动匹配找到构造方法）子类构造方法是无参的，不用写 super（）；系统默认调父类无参

子类构造方法是有参的，要调用父类无参就必须写 super（）；

6 要同时出现 this 和 super

一个构造方法内同时出现 this（18）和 super()两个都要写第一行，怎么办？

只写 this(18)因为调用的下一个构造方法里还会调用 super(),所以没有关系，父类的构造有且只有一次
7.抽象类一定要继承，因为它不能被实例化，不继承就没有存在的意义（没有抽象方法也可以是抽象类，只是他不能被实例化的）
8.可以通过继承实现了多态
9.类只加载一次，用到的时候才加载，先加载父类，再加载子类
10.初始化属性　　定义时初始化　代码块初始化　构造方法初始化
11.子类调用父类的非私有方法，直接用方法名就可以
12，抽象类：
.抽象类一定要继承，因为它不能被实例化，不继承就没有存在的意义（没有抽象方法也可以是抽象类，只是他不能被实例化的）
抽象类可以有构造方法，但不能实例化，子类对父类数据进行初始化，
子类不重写抽象方法，子类也是抽象类，重写全部的抽象方法，子类就称为具体的类，
13.抽象类与接口：
抽象类可以有普通属性，静态属性
但是接口所有的变量都是静态常量，所有的方法都是抽象方法，也没有构造器，　全为 public，接口中无静态方法，成员变量全部都是
public　static　final,实现类必须实现所有的方法
抽象类是用来捕捉子类特性的，他只是一个模板，
接口比抽象类更加抽象，自由度更高，
什么时候使用抽象类和接口：
如果有用一些方法并且想让他们中的一些具有默认实现
多实现使用接口
抽象了是实现行为，
而接口是定义行为，
抽象类是对类的整体进行抽象，希望某个行为有不同的实现方式，
而接口是对类局部行为进行抽象，比如飞机和鸟都会飞，可以将飞机和鸟设计成类，但是不能讲飞行这个特性设计成为类，它并不是对一类事物的抽象描述，你可以把它设定为一个接口 Fly，包含方法 fly;，飞机类和鸟类根据自己的需要去实现这个接口，
继承是一个　是不是的关系，子类一定是父类的种类
接口的实现规则是有没有，具不具备的关系，能飞就能实现　这个接口，不能飞就不行

HashMap

HashMap 底层实现原理
1.HashMap 继承了 abstractMap,abstractMap 实现了 Map 接口
2.它的底层是由数组+链表来实现的，
3.主体躯干是数组，数组中的每一项都是一个链表

4.HashMap 的基本功能与 HashTable 的相同，都允许 key 和 value 为空，但是 hashMap 是非线程安全的，同时他不能保证 Entry 的顺序，
5.当用 put 操作的时候，首先会根据内部定义的算法(哈希函数)找到数组下标，将数据直接放入此数组元素中，如果该下标下已经有值了，（俗称 hash 冲突），将会把这个数组元素上的链表进行遍历，将新的数据放到链表末尾
6.哈希函数作用是将一个不固定长度的二进制值进行运算，转化为一个固定长度的二进制值。将插入元素的 key 进行运算后，得到一个 index 值，从而确定在数组中的存储位置，但是不同的 key 可能得到相同的 index，就出现了哈希冲突，再好的哈希函数在优先长的哈希表上，都会造成哈希冲突
7.三种解决冲突的方式
开放地址法：放弃本次寻找的 Index,往后继续寻找下一块违背占用的空间
再散列法:没看
链地址法：采用数组（哈希表的主体）+链表（为了解决哈希冲突而存在）的形式，哈希表中存储数组，数组中的元素空间存储链表的头结点。当发生冲突的时候，将相应的 index 位置上旧元素的指针 next 指向新元素即可。
8.特点：无顺序的存储，允许存放 null 键和 null 值，而 hashTable 不允许
9.HashMap 的主干是一个 Entry 数组，里面存放 Entry 对象。每一个 Entet 对象包含键值对和 next 指针以及对应的 hash 值　　[jdk1.8 之前用的是 Entry,后者用的是 Node,两者基本等价]
10.put 操作　由 key 计算哈希值，由哈希值计算在数组中的索引
如果索引处为 null 的话，则直接在此处插入元素。如果索引处的 Entry 不为空，则遍历链表看有没有相同的 key 值，如果哈希值和 key 值都相同，则认为是同一个 Entry 对象，覆盖原来的 value 值，当哈希值相同而 key 值不同时，则认为是一个不同的 Entry 对象，那么在链表尾部新增元素
（jdk1.8 之前插入头部链表，jdk1.8 之后插入链表尾部）
链表长度大于 8 个的时候，jdk1.8 会将此链表转化为红黑树，如果是普通二叉树可能会退化为链表
HashMap 会根据 key 的 hashCode 值来保存 value，hashMap 允许一条记录的 key 为空
HashTable 是线程安全的，他使用 synchronize 来做线程安全，它全局都同步了

HashMap 为啥线程不安全
它的默认的容量是 16，随着元素不断添加到 HashMap 里，出现冲突的几率就会很大，会影响查询的性能，所以要扩容
在多并发的情况下进行扩容，有一个线程执行到。。。另一个线程已经执行完扩容，在等这个线程执行完就会出现环路，并且也会丢失一些节点
在扩容阶段
1.HashMap 在扩容阶段容易出现线程不安全，重新分桶，两个线程操作，容易形成依赖循环（1.7 出现，1.8 会用红黑树优化，不会出现死循环）
hashMap 中的 Entry 链表产生环形数据结构，next 往下找的时候，永远找不到最后一个，所以导致死循环
2.一个成功扩容，另一个数据丢失
3.在 put 阶段，先计算出 hash 的位置，在赋值，两步操作，容易出现线程安全问题，覆盖前值

4.两个线程一个想 get，一个想 remove， 本来想先 get，在 remove,但是线程不安全就可能出现先 remove 后 get

异常

异常分类（异常是通知错误的机制（运行时））
顶层类 Throwable 派生出两个重要的子类，Error 和 Exception
其中 Error 指的是 Java 运行时内部错误和资源耗尽错误. 应用程序不抛出此类异常. 这种内部错误一旦出
现，除了告知用户并使程序终止之外，再无能无力. 这种情况很少出现.不可恢复的错误，CPU 烧了
Exception 是程序猿所使用的异常类的父类.
其中 Exception 有一个子类称为 RuntimeException ，这里面又派生出很多我们常见的异常类
NullPointerException ， IndexOutOfBoundsException 等.
受查异常和非受查异常
Java 语言规范将派生于 Error 类或 RuntimeException 类的所有异常称为 **非受查异常**，所有的其他异常称为**受查异常**
受查异常一定要处理或者在方法签名里进行异常说明（让上级去处理），否则编译不通过
非受查异常可以捕获也可以不补货
Try catch finally
try 代码块中放的是可能出现异常的代码.
catch 代码块中放的是出现异常后的处理行为.
catch（异常 1|异常 2 e）{}两种异常一种处理方式，异常是从上往下匹配，一旦匹配了，就不匹配

finally 代码块中的代码用于处理善后工作，会在最后执行.
其中 catch 和 finally 都可以根据情况选择加或者不加.
有关 try 的 return 问题
finally 执行的时机是在方法返回之前(try 或者 catch 中如果有 return 会在这个 return 之前执行 finally). 但是如果
finally 中也存在 return 语句，那么就会执行 finally 中的 return，从而不会执行到 try 中原有的 return.
一般我们不建议在 finally 中写 return (被编译器当做一个警告).
关于调用栈的问题
方法之间是存在相互调用关系的，这种调用关系我们可以用 "调用栈" 来描述. 在 JVM 中有一块内存空间称为
"虚拟机栈" 专门存储方法之间的调用关系. 当代码中出现异常的时候，我们就可以使用
e.printStackTrace(); 的方式查看出现异常代码的调用栈.
异常处理流程
程序先执行 try 中的代码

如果 try 中的代码出现异常，就会结束 try 中的代码，看和 catch 中的异常类型是否匹配.
如果找到匹配的异常类型，就会执行 catch 中的代码
如果没有找到匹配的异常类型，就会将异常向上传递到上层调用者.
无论是否找到匹配的异常类型，finally 中的代码都会被执行到(在该方法结束之前执行).
如果上层调用者也没有处理的了异常，就继续向上传递.
一直到 main 方法也没有合适的代码处理异常，就会交给 JVM 来进行处理，此时程序就会异常终止.
由于 Exception 类是所有异常类的父类. 因此可以用这个类型表示捕捉所有异常.
备注: catch 进行类型匹配的时候，不光会匹配相同类型的异常对象，也会捕捉目标异常的子对象
finally 表示最后的善后工作,不管有没有异常都会执行的部分　比如说释放资源,
如果本方法中没有合适的处理异常的方式，就会沿着调用栈向上传递
如果向上一直传递都没有合适的方法处理异常，最终就会交给 JVM 处理，程序就会异常终止(和我们最开始未使用 try catch 时是一样的).
抛出异常：
throw new ArithmeticException("抛出除 0 异常");
异常说明：
public static int divide(int x, int y) throws ArithmeticException {...}
提醒调用者要捕获这个异常，告诉调用者调用这个方法是与风险的
自定义异常：
自定义异常通常会继承自 Exception 或者 RuntimeException
继承自 Exception 的异常默认是受查异常
继承自 RuntimeException 的异常默认是非受查异常.

数据库基础

1.展示当前所有数据库 show databases;
2.创建数据库　create database [if no exists] name;
3.删除当前数据库　　drop database[if exist] name;
4.使用数据库　use name;

1.展示当前数据库内所有的表 show tables;
2.创建表

```
create   table     student(
         id int primary key  comment'用户 id',
         name varchar(20) defalut'不知道'
);
comment  是注释
```

3.删除表　　drop table [if exist] student;

4.查看表的结构　字段大小，是否有默认值，是否有约束类型，有没有主键 desc
5.使用表，对表进行操作　use　student（准备增删查改的时候 CRUD）
CRUD
1.新增 C
1.单行数据+全列插入 insert into　表名 values();
2.多行数据+指定列插入 insert　into　表名(字段名，括号里的字段名可有可无，但要写全写)　values(),(),();
没指定的字段名有默认值,字段名中如果有主键约束的,则该语句不能重复使用,主键不为空且唯一，反正注意字段名的约束
3.表之间赋值数据（部分赋值）
insert into table2(f1,f2) select a1,a2 from table1
前提是 table2 和 f1,f2 必须存在　　两个表结构一致
全复制　insert into table2 select *from table1　　两个表结构不一致
2.查询 R
1.全查询　select * from　表名;
2.指定列查询　select　字段 1，字段 2　别名，字段 3　from　表名　where...;　字段可以加别名　可以是表达式（背后是实际计算）
3.去重查询 select distinct　字段名 from　表名;
4.排序　order by 升序 asc　降序 desc　默认升序
SELECT name, qq_mail FROM student ORDER BY qq_mail;
SELECT name, qq_mail FROM student　ORDER BY qq_mail DESC;
加上别名
SELECT name, chinese + english + math total FROM exam_result　ORDER BY total DESC;
对多个字段排序,按一个字段排序之后，还有相同的值，再按下一个字段进行排序
SELECT name, math, english, chinese FROM exam_result　ORDER BY math DESC, english, chinese;
5.between a to b　范围匹配[a,b]
6.in(, ,);
查询数学成绩是 58 或者 59 或者 98 或者 99 分的同学及数学成绩
SELECT name, math FROM exam_result WHERE math IN (58, 59, 98, 99);
7.模糊查询 like　　%匹配任意多个（包括 0 个）字符　_匹配一个任意字符
SELECT name FROM exam_result WHERE name LIKE '孙%';-- 匹配到孙悟空、孙权
8.字段名 is null　is not null
9.分页查询　　一般在排序之后用
从 0 开始，筛选 n 条结果　(包含第 0 个)
　SELECT ... FROM table_name [WHERE ...] [ORDER BY ...] LIMIT n;
从 s 开始，筛选 n 条结果，比第二种用法更明确，建议使用
　SELECT ... FROM table_name [WHERE ...] [ORDER BY ...] LIMIT n OFFSET s;　　// limit s,n;
三.修改 U
update　表名　set　字段名=...where/order by　条件

将总成绩倒数前三的 3 位同学的数学成绩加上 30 分
UPDATE exam_result SET math = math + 30 ORDER BY chinese + math + english LIMIT3;

四.删除 D

delete from 表名 where 条件
删除孙悟空同学的考试成绩
DELETE FROM exam_result WHERE name = '孙悟空'
delete from 表名 删除整张表的数据，但是没有把这个表删掉，

SQL 查询中各个关键字的执行先后顺序 from > on> join > where > group by > with >having > select > distinct > order by > limit

数据库约束 约束类型

1.NOT NULL - 指示某列不能存储 NULL 值。
2.UNIQUE - 保证某列的每行必须有唯一的值。
3.DEFAULT - 规定没有给列赋值时的默认值。
4.主键 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。
5.外键 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。
6.CHECK - 保证列中的值符合指定的条件。对于 MySQL 数据库，对 CHECK 子句进行分析，但是忽略
CHECK 子句。

表的设计

一对多

一个班级对应许多同学

```
//一个同学属于一个班级，外键写在同学里
create table student(
      id int primary key auto_increment;
      sn int unique,
      name varchar(20) default'不知道'
      classes_id int,//所属班级
      foreign key(classes_id) references classes(id)
);
```

多对多

图书,学生具有一个借阅场景(借阅记录表作为中间表)

```
create table student(
          id int primary key auto_increment,
         name varchar(20)
);
create table book(
          id int primary key auto_increment,
         name varchar(20)
);
create table record(
          id int primary key auto_increment,
```

```
        拿着书 student_id 属性去同学表里找找有哪些同学借过书
        student_id,
        foreign key(student_id) references student(id),
        拿着同学的 student_id 去找书的表里面查找该同学借过哪些书
        book_id,
        foreign key(book_id) references student(id),
);
```

聚合查询（聚合函数）

1.count(0)或者 count(1)返回查询到的数据总量多少行

count(字段名) 若系列的一个元素为 null，则不计入总行数

统计班级收集的 qq_mail 有多少个，qq_mail 为 NULL 的数据不会计入结果

```
SELECT COUNT(qq_mail) FROM student;
```

2.sum 统计总和

```
SELECT SUM(math) FROM exam_result WHERE math < 60;
```

3.avg 统计平均成绩

统计平均总分

```
SELECT AVG(chinese + math + english) 平均总分 FROM exam_result;
```

4.max 返回最高

返回英语最高分

```
SELECT MAX(english) FROM exam_result;
```

5.min 返回最低

group by 子句

对指定列进行分组查询

查询每个角色的最大值，最小值和平均值

```
select role,max(salary),min(salary),avg(salary) from emp group by role;
```

在 使用 group by 之后的 select 子句中的列名必须为分组后的列或列函数

例如

```
select emp_no,count(emp_no) as t
from salaries
group by emp_no
having t>15;
```

//分组之后的实质上这些行并没有消失，只是不显示，只显示一次

//分组之后的 count(emp_no)是一个组的行数，并不是整个数据表的总行数，是分组之后的，这时的 count()是列函数

```
select *  from emp group by name;//这调命令是错的，我们 select  选出所有列但是分组只有 name 列，每个 name 只出现一次
```

having

分组后过滤用 having 不用 where

联合查询

笛卡尔积 a 表中的每一个元素都会去遍历 b 表中的所有元素

```
select * from  表 1  表 2
```

```
select * from score,student
```

相当于两个表的成绩，每一个 score 都去关联了一整张表

自连接（查询不同行同一列的值）

是指同一张表连接自身进行查询（条件是跟自身有关）
先查询“计算机原理”和“Java”课程的 id
select id,name from course where name='Java' or name='计算机原理';
内连接
select 字段 from 表1 别名1 [inner] join 表2 别名2 on 连接条件 and 其他条件;
select 字段 from 表1 别名1,表2 别名2 where 连接条件 and 其他条件;
select sco.score from student stu inner join score sco on
stu.id=sco.student_id and stu.name='许仙';
多张表关联 就是表名1 join 表2 on 连接条件 join 表三 on 连接条件
//如果查询时两个表有重复字段，记得要用表.字段来标明是哪个字段
外连接（左外连接（左侧完全显示）和右外连接（右侧完全显示））
select 字段名 from 表名1 left join 表名2 on 连接条件;
左表完全显示，左表元素在右边中没有值，也要显示，显示为空，右表的列也会出现，但是值会根据左表的值补 null
select 字段 from 表名1 right join 表名2 on 连接条件;
子查询（嵌套查询）
单行子查询
select * from student where classes_id=(select classes_id from student where name='不想毕业');
多行子查询
1.[not] in 关键字
select * from score where course_id in (select id from course where name='语文' or name='英文');
select * from score where course_id not in (select id from course where name!='语文' or name!='英文');
2.[not] exists 关键字
select * from score sco where exists (select sco.id from course cou where (name='语文' or name='英文') and cou.id = sco.course_id);
select * from score sco where not exists (select sco.id from course cou where (name!='语文' or name!='英文') and cou.id = sco.course_id);
3.在 from 子句中使用子查询：子查询语句出现在 from 子句中。这里要用到数据查询的技巧，把一个子查询当做一个临时表使用
合并查询
1.union(去重)
该操作符用于取得两个结果集的并集。当使用该操作符时，会自动去掉结果集中的重复行
select * from course where id<3
union
select * from course where name='英文';
select * from course where id<3 or name='英文';
2.union all(不去重)
select * from course where id<3

```
union all
select * from course where name='英文';
```

时间日期借阅记录查询

查询某个时间段的借阅记录

开始日期小于要求的截止日期

结束日期大于要求的开始时间

```
select b.name book_name, b.id book_id ,c.id c_id ,
c.name c_name,bi.start_time,bi.end_time from book b
join category c on b.category_id=c.id
join borrow_info bi on bi.book_id=b.id
where bi.start_time<'2019-09-13 00:00:00' and bi.end_time>'2019-07-01
00:00:00';
```

另一种错误想法是因为保证全在里面

实质上语义端在里面就算是此时的借阅记录，并不是开始和结束都要处于这个时间段

MySQL 索引事务

字段的索引类似于书记的目录，可以快速找查找到数据

索引是数据结构

索引是一种特殊的文件，包含着对数据表里所有记录的引用指针　相当于记录的这些数据的地址

作用：快速定位，检索数据，索引对于提高数据库的性能有很大的帮助

使用场景：要对数据库表的某列或某激烈建立索引，需要考虑以下几点：

数据量大，并且经常要对这些列进行查询

数据表的插入和对这些列的修改的操作频率较低

索引会占用额外的磁盘空间

创建主键约束，唯一约束，和外键约束时，会自动的创建索引

索引查起来很快，但是创建索引的过程比较慢

1.查看索引

```
show index from  表名
```

2.创建索引（非主键，非唯一约束，非外键的字段）创建普通索引

```
create index  索引名 on  表名（字段名）
```

3.删除索引

```
drop index  索引名 on  表名;
```

事务

事务是指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部失败

（1）开启事务：start transaction;

（2）执行多条 SQL 语句

（3）回滚或提交：rollback/commit;

说明：rollback 即是全部失败，commit 即是全部成功。

mysql 语句的执行顺序

```
1 FROM <left_table>
2 ON <join_condition>
3 <join_type> JOIN <right_table>
```

4 WHERE <where_condition>
5 GROUP BY <group_by_list>
6 HAVING <having_condition>
7 SELECT
8 DISTINCT <column>
9 ORDER BY <order_by_condition>
10 LIMIT <limit_number>

sqlite_master(sqlite 数据库的一个内置表，存储数据库中所有表的相关信息)

```
SELECT
        'select count(*) from ' || NAME || ';' AS cnts
FROM
        sqlite_master
WHERE
        TYPE = 'table';
```

||理解为+，from 与引号之间有空格，那么是表名