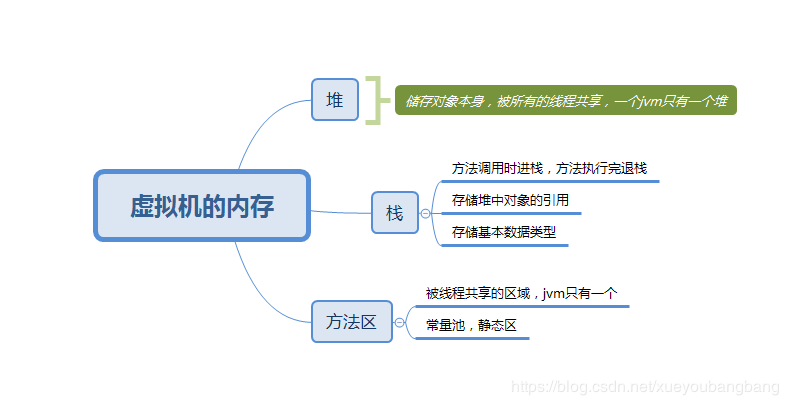
## 堆和栈

栈的效率高.由操作系统自动分配释放,在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行

按照先后定义的顺序依次压入栈中，也就是说相邻变量的地址之间不会存在其它变量。栈的内存地址生长方向与堆相反，由高到底，后定义的变量地址低于先定义的变量。栈中存储的数据的生命周期随着函数的执行完成而结束

栈是运行时单位，用来解决程序运行时的问题，堆是存储单位，解决数据存储的问题。



堆: (heap)

1.存储的全部是对象，每个对象都包含一个与之对应的class的信息。(class的目的是得到操作指令)

2.只有一个堆,伴随着JVM的启动而创建。

3.堆的申请和释放工作由程序员控制，容易产生内存泄漏--己动态分配的堆内存未释放或无法释放

栈区:

1.每个线程一个栈，栈中只保存基本数据类型和自定义对象的引用(不存放对象)

2.每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。

3.栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)

方法区:



1.又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class文件信息和static变量

2.方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。

常量池

1.常量池中包含基本类型和对象型的常量值（如String及数组）。

　　1.1 这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另 外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值小于等于127时才可使用常量池，也即对象不负责创建和管理大于127的这些类的对象。大于127的对象存放在堆中。

　　1.2 String类也实现了常量池技术

2.常量池还包含一些以文本形式出现的符号引用，比如：

　　2.1 类和接口的全限定名；

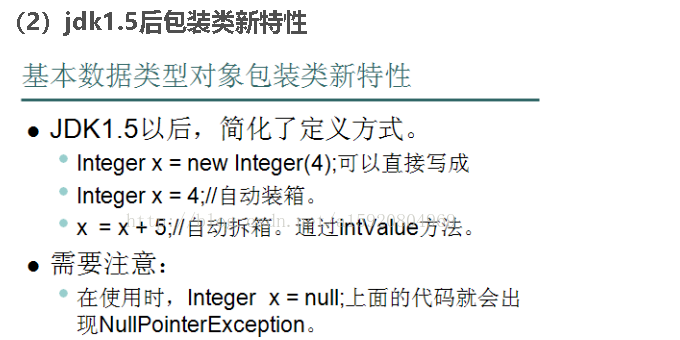
　　2.2 字段的名称和描述符；

　　2.3 方法和名称和描述符。

### 拆箱装箱JDK1.5

装箱：基本数据类型🡪包装器类型 valueOf方法

拆箱：包装器类型🡪基本数据类型 xxxValue方法



## static非 static 变量区别

1. static变量只有1份, 被该类创建的所有对象共享, 也被称为类变量

非static变量有多个, 也被称为实例变量

2. static变量存在方法区中，实例变量存在堆内存中

3.访问的方式不同

static变量 类名.变量名

实例变量： 对象名.变量名

4.在内存中分配空间的时间不同

static变量 第一次使用类

实例变量 创建对象时候

static 内部类 只能访问外部类的静态成员

## 25.final 和 abstract 关键字的作用

abstract不能修饰属性和构造方法；abstract 修饰的方法是抽象方法，需要子类被重写。

final不能修饰构造方法。final修饰的类不能被继承，方法不能被重写，使用 final 修饰的变 量的值不能被修改，所以就成了常量。

final 修饰引用类型变量，栈内存中的引用不能改变，但是所指向的堆内存中对象的属性值可以改变

final修饰类：表示该类不能被继承；

修饰方法：表示方法不能被重写但是允许重载；

修饰对象：对象的引用地址不能变，但是对象的初始化值可以变。

## final、finally、finalize 的区别

finally 在异常处理时提供 finally 块来执行任何清除操作。如果有 finally 的话，则不管是否发生异常，finally 语句都会被执行。一般情况下，都把关闭物理连接(IO流、数据库连接、Socket连接)等相关操作，放入到此代码块中

finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要清理工作。finalize() 方法是在垃圾收集器删除对象之前被调用的。它是在Object类中定义的，因此所有类都继承。

## Object 类的六个常用方法

(1) public boolean equals(java.lang.Object) 比较对象的地址值是否相等，如果子类重写，则比较对象的内容是否相等；

(2)public native int hashCode() 获取哈希码

(3)public java.lang.String toString() 把数据转变成字符串

(4)public final native java.lang.Class getClass() 获取类结构信息

(5)protected void finalize() throws java.lang.Throwable 垃圾回收前执行的方法

(6)protected native Object clone() throws java.lang.CloneNotSupportedException 克隆

(7)public final void wait() throws java.lang.InterruptedException 多线程中等待功能

(8)public final native void notify() 多线程中唤醒功能

(9)public final native void notifyAll() 多线程中唤醒所有等待线程的功能

## 重写和重载

重写的访问修饰符只能比父类大

## private/默认/protected/public 权限修饰符的区别



## 继承中构造方法的执行过程

情况 1：没有super显式调用父类有参构造，没有this显式调用自身的构造，则系统会默认先调用父类无参,再调用子类构造方法。在这种情况下，写不写“super();”语句，效果是一样 的。

情况 2：通过super显式调用父类的有参构造方法，那将执行父类相应构造方法，而不执行父类无参构造方法。

情况 3：如果子类的构造方法中通过this显式调用自身的其他构造方法，在相应构造方法中应用以上两条规则。

特别注意的是，如果存在多级继承关系，在创建一个子类对象时，以上规则会多次向更高一级父类应用，一直到执行顶级父类 Object 类的无参构造方法为止。

## ==和 equals 的区别和联系

“==”是关系运算符，equals()是方法，同时他们的结果都返回布尔值

== a) 基本类型，比较的是值 b) 引用类型，比较地址 c) 不能比较没有父子关系的两个对象

equals() a) 系统类一般已经覆盖了 equals()，比较的是内容。 b) 用户自定义类如果没有覆盖 equals()，将调用父类的 equals（Object的equals比较地址）c) 用户自定义类需要覆盖父类的 equals()

Object 的==和 equals 比较的都是地址

## 面向对象的三大基本特征，五大基本原则

　1、封装

　　封装就是隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别，将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成“类”，其中数据和函数都是类的成员。

　　2、继承

　　继承是面向对象的基本特征之一，继承机制允许创建分等级层次的类。继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。类似下面这个图：

3.多态三个条件

1、继承

2、子类重写父类的方法（向上转型）

Student person = new Student() 将一个父类的引用指向一个子类对象，自动进行类型转换。 此时通过父类引用变量调用的方法是子类覆盖或继承父类的方法

3、父类引用变量指向子类对象（向下转型）

优点：

1. 消除类型之间的耦合关系

2. 可替换性

3. 可扩充性

4. 接口性

5. 灵活性

6. 简化性

## 垃圾回收机制

传统的 C/C++语言，需要程序员负责回收已经分配内存。

显式回收垃圾回收的缺点：

1）程序忘记回收，从而导致内存泄露，降低系统性能。

2）程序错误回收程序核心类库的内存，导致系统崩溃。

Java由JRE在后台自动回收不再使用的内存

1）可以提高编程效率。

2）保护程序的完整性。

3）其开销影响性能。Java 虚拟机必须跟踪程序中有用的对象，确定哪些是无用的。

垃圾回收机制的特点

1）回收堆内存里的对象空间,不负责回收栈内存数据

2）不回收物理连接，如数据库连接、IO、Socket

3）垃圾回收发生具有不可预知性，程序无法精确控制垃圾回收机制执行

4）可以将对象的引用变量设置为 null，暗示可以回收该对象。

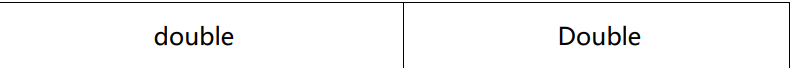
5）回收任何对象之前，总会先调用它的 finalize 方法

程序员可以通过 System.gc()或者 Runtime.getRuntime().gc()来请求回收

不要主动调用finalize ，应该交给垃圾回收机制调用

## 基本数据类型和包装类





基本数据类型不是面向对象（没有属性、方法），实际使用时存在很多的不便（比如集合的元素只能是 Object）。所以需要包装类。

## 34. Integer 与 int 的区别

Integer是int 提供的封装类

int 的默认值为 0，而 Integer 的默认值为 null即Integer 可以区分出未赋值和值为0的区别

在JSP中，Integer默认为null，el表达式在文本框中显示时，值为空白字符串，而int默认值为 0，显示为 0，所以，int 不适合作为web层的表单数据的类型。

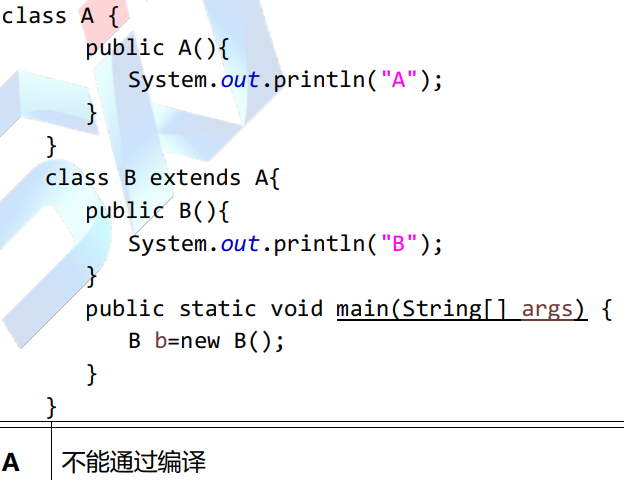
## 35. java.sql.Date 和 java.util.Date 的联系和区别

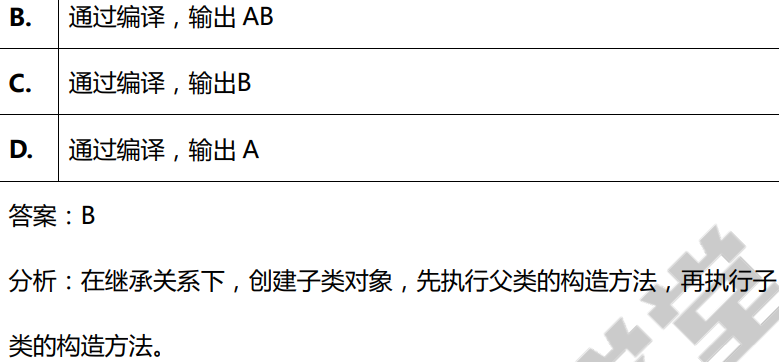
1） java.sql.Date 是 java.util.Date 的子类，是一个包装了毫秒值的瘦包装器，允许JDBC将毫秒值标识为 SQL DATE 值

2）java.sql.Date是针对 SQL 语句使用的，只包含日期而没有时间部分。

以下操作中容易出现不易被发现的 BUG：获得一个 JAVA 里的日期对象。 从数据库里读取日期 试图比较两个日期对象是否相等。如果毫秒部分丢失，本来认为相等的两个日期对象用 Equals 方法可能返回 false。sql.Timestamp 类比 java.util.Date 类精确度要高。这个类包了一个 getTime()方法，但是它不会返回额外精度部分的数据，因此必须使用...

## 40. 给定以下代码，程序将输出





## 47. 内存泄漏和溢出

内存溢出 程序在申请内存时没有足够的内存，比如申请了integer,但存了long才能存下的数

内存泄露 程序在申请内存后，无法释放已申请的内存空间

# 序列化

序列化能够将实例对象的状态信息写入到字节流,使其可以通过socket进行传输或者持久化.然后在需要的时候通过字节流中的信息来重构相同的对象。一般而言，要使得一个类可以序列化，只需简单实现 java.io.Serializable 接口即可。

#### 两种用途

很多应用需要对某些对象进行序列化，让它们离开内存空间，入住物理硬盘，以便长期保存。比如最常见的是 Web 服务器中的 Session 对象，当有 10 万用户并发访问，就有可能出现 10 万个 Session 对象，内存可能吃不消，于是 Web 容器就会把一些 seesion 先序列化到硬盘中，等要用了，再把保存在硬盘中的对象还原到内存中。当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个 Java 对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为 Java 对象。

#### serialversionUID

目的是序列化对象版本控制。如果在新版本中这个值修改了，新版本就不兼容旧版本，反序列化时会抛出InvalidClassException异常。如果修改较小，比如仅仅是增加了一个属性，我们希望向下兼容，那就不用修改；如果我们删除了一个属性，或者更改了类的继承关系，必然不兼容旧数据，这时就应该手动更新版本号

# Java 创建对象几种方式

1、new 语句

2、反射,调用 java.lang.Class 或者 java.lang.reflect.Constructor类的 newInstance()实例方法。

3、调用对象的clone()方法。

4、运用反序列化手段，调用 java.io.ObjectInputStream 对象的readObject()方法。

(1)和(2)都会显式地调用构造函数

(3)是在内存上已有对象的影印，不会调用构造函数

(4)是从文件中还原类的对象，也不会调用构造函数。

## 匿名内部类可不可以继承或实现接口？

匿名内部类是没有名字的内部类,不能继承其它类,但内部类可以作为接口,由另一个内部类实现.

1、由于匿名内部类没有名字，所以它没有构造函数,所以它必须完全借用父类的构造函数来实例化，换言之：匿名内部类完全把创建对象的任务交给了父类去完成。

2、在匿名内部类里创建新的方法没有太大意义，但它可以通过覆盖父类的方法达到神奇效果

3、匿名内部类没有名字，所以无法向下强转，持有对一个匿名内部类对象引用的变量类型一定是它的直接或间接父类类型。

## 基本类型不能做为HashMap的键值，而只能是引用类型

(1) Java使用泛型来约束HashMap<K, V>的键值对；而泛型必须是Object类型

map.put(1, “Java”)，实际上是将1进行了自动装箱操作,变为了 Integer类型

(2) 引用数据类型分为两类：系统提供的引用数据类型（如包装类、String

等）以及自定义引用数据类型。系统提供的引用数据类型中已经重写了

HashCode()和 equals()两个方法，所以能够保证key的唯一性；

但是自定义的引用数据类型需要重写HashCode()和equals()，保证key值的唯一性

# 接口和抽象类的区别

## 相同点

Ø 抽象类和接口均包含抽象方法，类必须实现所有的抽象方法，否则是抽象类

Ø 抽象类和接口都不能实例化，他们位于继承树的顶端，用来被其他类继承和实现

## 区别

接口中只能定义全局静态常量，不能定义变量。抽象类中可以定义常量和变量

接口中所有的方法都是抽象方法。抽象类中可以不全为抽象方法

接口中不能定义构造方法。抽象类中可以有构造方法，但不能用来实例化，而在子类实例化是执行，完成属于抽象类的初始化操作。

单继承多接口

接口可以继承接口

抽象类可以实现接口，抽象类可以继承实体类,可以有main方法

抽象类与普通类的唯一区别就是不能创建实例对象和允许有abstract方法。

最主要区别还是设计理念

Ø 接口 实现类仅仅是实现了接口定义的约定。接口定义了“做什么”，而实现类负责“怎么做”，体现了功能和实现分离的原则。接口和实现之间可以认为是一种“has-a 的关系”

Ø抽象类体现了一种继承关系，目的是复用代码，抽象类中定义了各个子类的相同代码，可以认为父类是一个实现了部分功能的“中间产品”，而子类是“最终产品”。父类和子类之间必须存在“is-a”的关系，即父类和子类在概念本质上应该是相同的。

### JDK8接口新特性

用默认方法与静态方法这两个新概念来扩展接口的声明

在jdk8之前，接口可以定义变量和方法，变量必须public static final，方法必须public abstract

JDK8新特性 1.允许接口中定义非抽象的方法.并且用static或default修饰

2.允许直接调用接口的静态方法

两个接口静态方法一模一样，并且一个实现类同时实现了这两个接口，此时并不会产生错误，因为jdk8只能通过接口类调用接口中的静态方法，所以对编译器来说是可以区分的。

两个接口非静态方法一模一样，并且一个实现类同时实现了这两个接口，那么必须在实现类中重写默认方法，否则编译失败。

静态方法调用 类名.方法 通过反射来区分哪个接口下的方法

非静态 对象.方法 分不出

## 57. 同步代码块和同步方法有什么区别

#### 相同点：

同步方法就是在方法前加关键字synchronized，然后被同步的方法一次只能有一个线程进入，其他线程等待。而同步代码块则是在方法内部使用大括号使得一个代码块得到同步。同步代码块会有一个同步的“目标”，使得同步块更加灵活一些（同步代码块可以通过“目标”决定需要锁定的对象）。一般情况下，如果此“目标”为 this，同步方法和代码块没有太大的区别

#### 区别：

同步方法直接在方法上加synchronized实现加锁，同步代码块则在方法内部加锁。很明显，同步方法锁的范围比较大，而同步代码块范围要小点。一般同步的范围越大，性能就越差。所以一般需要加锁进行同步时，范围越小性能更好

## 静态内部类和内部类有什么区别

静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。

静态内部类可以有静态成员(方法，属性)，而非静态内部类则不能有静态成员(方法，属性)。

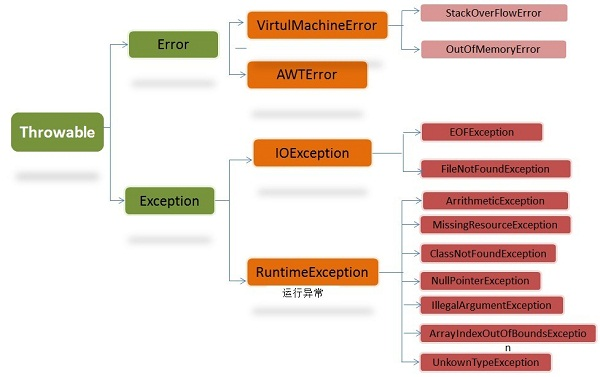
静态内部类只能访问外部类的静态成员。非静态内部类能够访问外部类的静态和非静态成员。

实例化方式不同：

1) 静态内部类：不依赖于外部类的实例，直接实例化内部类对象

2) 非静态内部类：通过外部类的对象实例生成内部类对象

## 编译时异常与运行时异常的区别



两个子类:异常,错误

异常和错误的区别：异常能被程序本身可以处理，错误是无法处理

运行时异常：都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch/throws也会编译通过。

非运行时异常 （编译异常）：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

Error（错误）:是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

。这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java中，错误通过Error的子类描述。

Exception（异常）:是程序本身可以处理的异常。Exception 类有一个重要的子类 RuntimeException。RuntimeException 类及其子类表示“JVM 常用操作”引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

通常，Java的异常(包括Exception和Error)分为可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）。

可查异常（编译器要求必须处置的异常）：正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。

除了RuntimeException及其子类以外，其他的Exception类及其子类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过。

不可查异常(编译器不要求强制处置的异常):包括运行时异常（RuntimeException与其子类）和错误（Error）。

Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

## 下列说法错误的有（）(多选)

A. 在类方法中可用 this 来调用本类的类办法

B. 在类方法中调用本类的类方法时可以直接调用

C. 在类方法中只能调用本类中的类方法

D. 在类方法中绝对不能调用实例方法

答案：ACD

A.类方法是在类加载时被加载到方法区存储的，此时还没有创建对象，所以

不能使用 this 或者 super 关键字；

C. 在类方法中还可以调用其他类的类方法；

D. 在类方法可以通过创建对象来调用实例方法。

## 下列说法错误的有（）(多选)

A. Java 面向对象语言容许单独的过栈与函数存在

B. Java 面向对象语言容许单独的方法存在

C. Java 语言中的方法属于类中的成员（member）

D. Java 语言中的方法必定隶属于某一类（对象），调用方法与过程或函

数相同

答案：ABC

B. Java 不允许单独的方法，过程或函数存在，需要隶属于某一类中；

C. 静态方法属于类的成员，非静态方法属于对象的成员。

### Math.round(11.5)和Math.round(-11.5)

round()是+0.5再向下取整

所以是12 -11

## 85. getDeclaredFields()与 getFields()的区别

getDeclaredFields(): 可以获取所有本类自己声明的方法, 不能获取继承的方法

getFields(): 只能获取所有public方法, 包括继承的方法

## &和&&的区别

#### 共同点

&和&&都可以用作逻辑与运算符，但是要看使用时的具体条件来决定。

情况 1：当上述的操作数是 boolean 类型变量时，&和&&都可以用作逻辑与运算符。

情况 2：当上述的表达式结果是 boolean 类型变量时，&和&&都可以用作逻辑与运算符。

表示逻辑与(and)，当运算符两边的表达式的结果或操作数都为 true 时，整个运算结果才为 true，否则，只要有一方为 false，结果都为 false。 &和&&的区别(不同点)： (1)、&逻辑运算符称为逻辑与运算符，&&逻辑运算符称为短路与运算 符，也可叫逻辑与运算符。 对于&：无论任何情况，&两边的操作数或表达式都会参与计算。 对于&&：当&&左边的操作数为 false 或左边表达式结果为 false 时，&& 右边的操作数或表达式将不参与计算，此时最终结果都为 false。 综上所述，如果逻辑与运算的第一个操作数是 false 或第一个表达式的结果 为 false 时，对于第二个操作数或表达式是否进行运算，对最终的结果没有 影响，结果肯定是 false。

&&效率更高

&还可以用作位运算符。当&两边操作数或两边表达式的结果不是 boolean 类型时，&用于按位与运算符的操作。

## final修饰变量，是引用还是引用的对象不能变？

final 修饰基本类型变量，其值不能改变

但是final修饰引用类型变量，栈内存中的引用不能改变，堆内存中的对象的属性值可以改变

例如class Test {

public static void main(String[] args) {

final Dog dog = new Dog("欧欧");

dog.name = "美美";//正确

dog = new Dog("亚亚");//错误

### static修饰变量,其值能改变吗?

对于静态变量在内存中只有一个拷贝（节省内存），JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。

　对于实例变量，每创建一个实例，就会为实例变量分配一次内存，实例变量可以在内存中有多个拷贝，互不影响（灵活）。

所以一般在需要实现以下两个功能时使用静态变量：

1.在对象之间共享值时

2.方便访问变量时

static final修饰的成员变量可理解为“全局常量”，常量的值才是不能更改的，才是不可修改的！

静态变量并不是说其就不能改变值，不能改变值的量叫常量。 其拥有的值是可变的 ，而且它会保持最新的值。

说其静态，是因为它不会随着函数的调用和退出而发生变化。即上次调用的时候，如果我们给静态变量赋予某个值的话，下次函数调用时，这个值保持不变。

## try/catch中的finally

在异常处理时提供 finally 块来执行任何清除操作。不管是否发生异常，finally语句都会被执行，包括 遇到return。

finally 中语句不执行的唯一情况中执行了 System.exit(0)语句。

## static变量修饰符

### 1、生命周期不同。

成员变量随着对象的创建而存在，随着对象的被回收而释放。

静态变量随着类的加载而存在，随着类的消失而消失。

### 2、调用方式不同。

成员变量只能被对象调用。

静态变量可以被对象调用，还可以被类名调用。

对象调用：p.country

类名调用 ：Person.country

### 3、别名不同。

成员变量也称为实例变量。

静态变量称为类变量。

### 4、数据存储位置不同。

成员变量数据存储在堆内存的对象中，所以也叫对象的特有数据.

静态变量数据存储在方法区(共享数据区)的静态区，所以也叫对象的共享数据

# 面向对象的特征

答：面向对象的特征主要有以下几个方面：

1、抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

2、继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段

3、封装：把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口

4、多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的（就像电动剃须刀是 A 系统，它的供电系统是 B 系统，B 系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A 系统只会通过 B 类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1. 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2. 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

### float f=3.4是否正确?

答:不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）

属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强

制类型转换 float f =(float)3.4; 或者写成 float f =3.4F;。

### short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗?

答：对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结

果也是 int 型，需要强制转换类型才能赋值给 short 型。而 short s1 = 1; s1

+= 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

### 两个值相同对象(x.equals(y) == true)， hash code可能不同?

答：不对，如果两个对象 x 和 y 满足 x.equals(y) == true，它们的哈希码应当相同。Java 对于 eqauls 方法和 hashCode 方法是这样规定的：

(1)如果两个对象相同（equals 方法返回 true），那么它们的hashCode 值一定要相同；

(2)如果两个对象的 hashCode 相同，它们并不一定相同

### 对象克隆

答：有两种方式：

1.实现 Cloneable 接口并重写 Object 类中的 clone()方法；

2.实现 Serializable 接口，通过对象的序列化和反序列化，可以实现真正的深度克隆,更重要的是通

过泛型限定，可以检查出要克隆的对象是否支持序列化，

### 创建对象时构造器的调用顺序

先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。

考点：静态代码块优先级 > 构造方法的优先级 如果再加一个普通代码块，优先顺序如下： 静态代码块>普通代码块>构造方法

构造器最后

### 数据类型之间的转换:

字符串->基本数据类型

调用基本数据类型对应的包装类中的方法 parseXXX(String)或valueOf(String)即可返回相应基本类型；

基本数据类型->字符串

一种方法是将基本数据类型与空字符串””连接（+）即可获得其所对应的字符串；另一种方法是调用 String 类中的 valueOf(…)方法返回相应字符串

# String

## 反转(递归)

public class A{

public static String reverse(String originStr) {

if(originStr == null || originStr.length() <= 1)

return originStr;

return reverse(originStr.substring(1)) +

originStr.charAt(0);

}

## 编码转换

答：代码如下所示:

String s1 = "你好";

String s2 = newString(s1.getBytes("GB2312"), "ISO-8859-1");

## 替换

s.replaceAll("原字符","替代字符");

## String在编译期的优化

字符串对象创建有两种形式，

1.字面量形式，如String str = "aaa" "aaa" 存进字符串常量池

2.new new 创建的存进了堆

对于字符串，其对象的引用都是存储在栈中的，如果是编译期已经创建好的就存储在常量池中(双引号定义的或final修饰并且能在编译期就能确定的)，如果是运行期才能确定的就存储在堆中（如：new关键字创建出来的）。

对于equals相等的字符串，在常量池中永远只有一份，在堆中有多份

String a="hello2";

String b="hello"+2;

System.out.println((a==b));

输出为：true。因为 ”hello” +2在编译期就已经被优化成 “hello2”，因此在运行期变量 a 和 b 指向的是同一个对象(在字符串常量池)

String a="hello2";

String b="hello";

String c=b+2;

System.out.println((a==c));

输出为 false。由于有符号引用的存在，所以String c=b+2不会在编译期间被优化，不会把 b+2 当做字面量处理，因此生成的对象是保存在堆上的。所以a和c不是指向同一个对象。

### String s=new String(“abc”);创建了几个 String 对象。

两个或一个，”abc”对应一个对象，这个对象放在字符串常量缓冲区，常量”abc”不管出现多少遍，都是缓冲区中的那一个。New String 每写一遍，就创建一个新的对象，它一句那个常量”abc”对象的内容来创建出一个新String 对象。如果以前就用过’abc’，直接从缓冲区拿。

### String和Builder、Buffer的区别?

相同点：

都使用 final 修饰，不能派生子类，操作的相关方法也类似例如获取字符串长度等；

不同点：

String只读，内容不能改变，StringBuffer和StringBuilder类表示的字符串对象可以直接进行修改，在修改的同时地址值不会发生改变。StringBuilder是JDK1.5中引入的，它和StringBuffer的方法完全相同，区别在于它是在单线程环境下使用的，没有被synchronized修饰，因此它的效率也比StringBuffer略高。在此重点说明一下，String、StringBuffer、StringBuilder 三者类型不一样，无法使用 equals()方法比较其字符串内容是否一样！

## String 类为什么是 final 的

1）若允许被继承，则其高度的被使用率可能会降低程序的性能

2）为了安全。JDK中提供的好多核心类比如 String，这类的类的内部好多方法的实现都不是 java 编程语言本身编写的，好多方法都是调用的操作系统本地的 API，这就是著名的“本地方法调用”，也只有这样才能做事，这种类是非常底层的，和操作系统交流频繁的，那么如果这种类可以被继承的话，如果我们再把它的方法重写了，往操作系统内部写入一段具有恶意攻击性质的代码什么的，这不就成了核心病毒了么？

## String s="Hello";s=s+"world!"是否是对前面 s指向空间内容的修改？

不是.因为 String 被设计成不可变类，所以它的所有对象都是不可变对象。在这段代码中，s 原先指向一个 String 对象，内容是 "Hello"，然后我们对s进行了+操作，这时，s 不指向原来那个对象了，而指向了另一个 String 对象，内容为"Hello world!"，原来那个对象还存在于内存之中，只是 s 这个引用变量不再指向它了。

通过上面的说明，我们很容易导出另一个结论，如果经常对字符串进行各种各样的修改，那么使用 String 来代表字符串的话会引起很大的内存开销。因为 String 对象建立之后不能再改变，所以对于每一个不同的字符串，都需要一个 String 对象来表示。这时，应该考虑使用StringBuffer 类，它允许修改，而不是每个不同的字符串都要生成一个新的对象。并且，这两种类的对象转换十分容易。

调用构造器创造string，性能低下且内存开销大，并且没有意义，因为String对象不可改变，所以对于内容相同的字符串，指向同一个String

上面的结论还基于这样一个事实：对于字符串常量，如果内容相同，Java认为它们代表同一个 String 对象。关键字 new 调用构造器，总是会创建一个新的对象，无论内容是否相同。

# 反射

## Class 类的作用？生成 Class 对象的方法有哪些？

Class 类是 Java 反射机制的起源和入口，用于获取与类相关的各种信息，提供了获取类信息的相关方法。Class 类继承自 Object 类Class 类是所有类的共同的图纸。每个类有自己的对象，好比图纸和实物的关系；每个类也可看做是一个对象，有共同的图纸 Class，存放类的 结构信息，能够通过相应方法取出相应信息：类的名字、属性、方法、构造方法、父类和接口

## 反射的使用场合和作用、及其优缺点

在编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息。通过反射可以使程序代码访问装载到 JVM 中的类的内部信息，获取已装载类的属性信息，获取已装载类的方法，获取已装载类的构造方法信息

反射提高了 Java 程序的灵活性和扩展性，降低耦合性，提高自适应能力。它允许程序创建和控制任何类的对象，无需提前硬编码目标类；

Java 反射技术应用领域很广，如软件测试等；许多流行的开源框架例如Struts、Hibernate、Spring 在实现过程中都采用了该技术

反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此 Java 反射机制主要应用在对灵活性和扩展性要求很高的系统框架上,普通程序不建议使用。使用反射会模糊程序内部逻辑：程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术，因而会带来维护问题。反射代码比相应的直接代码更复杂。

# 面向对象设计原则

面向对象设计原则是面向对象设计的基石，面向对象设计质量的依据和保障，设计模式是面向对象设计原则的经典应用

1）单一职责原则 SRP

2）开闭原则 OCP

3）里氏替代原则 LSP

4）依赖注入原则 DIP

5）接口分离原则 ISP

6）迪米特原则 LOD

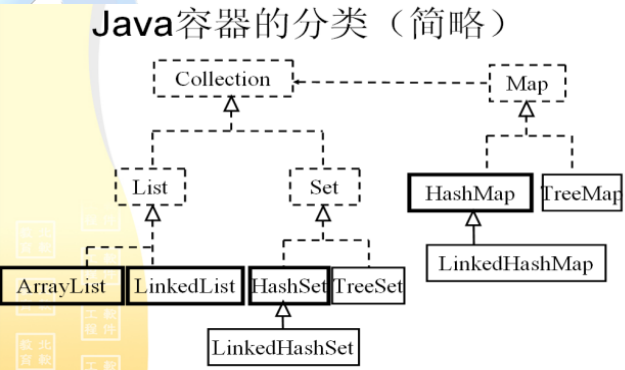
7）组合/聚合复用原则 CARP

开闭原则是面向对象设计的终极目标。其他设计原则都可以看作是开闭原则的实现手段或方法

# 集合

## Collection（List、Set、Collection、Map的区别联系）

Java 中集合主要分为两种：Collection 和 Map



1、Collection不唯一，无序

2、List不唯一，有序

3、Set唯一，无序

4、Map Key无序，唯一 value不要求有序，允许重复

## ArrayList 和 LinkedList 的区别和联系

相同点：

都实现List接口，有序、不唯一

不同点：

ArrayList数组实现,长度可变，查询效率高 虽然是数组,初始容量10,但是当满了时,就会新建一个2倍容量的数组,并把原数组复制过去,实现了扩容

LinkedList 采用双向链表存储方式。插入、删除元素时效率高

#### List list = new ArrayList() 与 ArrayList alist = new ArrayList()

List接口有多个实现类，现在你用的是ArrayList，也许哪一天需要换成LinkedList或者Vector等等，这时你只要改变一行就行 这就是面向接口编程

LinkedList和ArrayList都实现了add等方法,都实现了List接口

在List list时,并不知道实例化了Linked还是Array,但是这个list都是要去add()的

这也是多态的体现,父类引用指向子类对象

## HashSet 的使用和原理（hashCode()和 equals()）

1）哈希表的查询速度特别快，时间复杂度为 O（1）。

2）HashMap、Hashtable、HashSet 这些集合采用的是哈希表结构，需要用hashCode

3）系统类已经覆盖了hashCode 方法 自定义类放入hash类集合，必须重写 hashcode。如果不重写，调用的是 Object 的 hashcode，而Object 的 hashCode 实际上是地址。

4）向哈希表中添加数据的原理：当向集合Set中增加对象时，首先集合计算要增加对象的 hashCode 码，得到一个位置用来存放当前对象，如在该位置没有一个对象存在的话，那么集合 Set 认为该对象在集合中不存在，直接增加进去。如果在该位置有一个对象存在的话，接着将准备增加到集合中的对象与该位置上的对象进行equals方法比较，如果该 equals 方法返回 false,那么集合认为集合中不存在该对象，在进行一次散列，将该对象放到散列后计算出的新地址里。如果equals方法返回 true，那么集合认为集合中已经存在该对象了，不会再将该对象增加到集合中了。

5）在哈希表中判断两个元素是否重复要使用到hashCode()和equals()。hashCode决定数据在表中的存储位置，而equals判断是否存在相同数据。

## TreeSet 的原理和使用（Comparable和comparator）

1）TreeSet 集合，元素不允许重复且有序(自然顺序)

2）TreeSet 采用树结构存储数据，存入元素时需要和树中元素进行对比，需

要指定比较策略。

3）可以通过 Comparable(外部比较器)和 Comparator(内部比较器)来指定

比较策略，实现了 Comparable 的系统类可以顺利存入 TreeSet。自定

义类可以实现 Comparable 接口来指定比较策略。

4）可创建 Comparator 接口实现类来指定比较策略，并通过 TreeSet 构造

方法参数传入。这种方式尤其对系统类非常适用。

## 集合和数组的比较（为什么引入集合）

数组不是面向对象的，存在明显的缺陷，集合完全弥补了数组的一些缺点，比数组更灵活实用，可大大提高软件的开发效率.不同的集合框架类可适用于不同场合

1）数组的效率高于集合类.

2）数组能存放基本数据类型和对象，而集合类中只能放对象

3）数组容量固定且无法动态改变，集合类容量动态改变

4）数组无法判断实际有多少元素，length只告诉了array的容量

5）集合有多种实现方式和不同的适用场合，而不像数组仅采用顺序表方式。

6）集合以类的形式存在，具有封装、继承、多态等类的特性，通过简单的方法和属性调用即可实现各种复杂操作，提高效率

## Collection 和 Collections 的区别

1）Collection集合接口，存储一组不唯一，无序的对象。两个子接口List和Set

2）Collections类专门用来操作集合类 ，它提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

Map 的实现类中，哪些是有序的，哪些是无序的，有序的是

如何保证其有序性，你觉得哪个有序性性能更高，你有没有更好

或者更高效的实现方式？

答：

1. Map 的实现类有 HashMap,LinkedHashMap,TreeMap

2. HashMap 是有无序的，LinkedHashMap 和 TreeMap 都是有序的（LinkedHashMap记录了添加数据的顺序；TreeMap默认是自然升序）

3. LinkedHashMap 底层存储结构是哈希表+链表，链表记录了添加数据的顺序

4. TreeMap 底层存储结构是二叉树，二叉树的中序遍历保证了数据的有序性

5. LinkedHashMap 性能比较高，因为底层数据存储结构采用的哈希表

## TreeMap 和 TreeSet 在排序时如何比较元素？Collections工具类中的 sort（）方法如何比较元素？

TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元素的 compareTo()方法，当插入元素时会 回调该方法比较元素的大小。

TreeMap 要求存放的键值对映射的键必须实现 Comparable接口从而根据键对元素进行排序。Collections 工具类的 sort 方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现

Comparable 接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是 Comparator 接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实就是是通过接口注入比较元素大小的算法，也是对回调模式的应用。

## List、Map、Set 三个接口，存取元素时，各有什么特点？

List 以特定索引来存取元素，可有重复元素。

Set 不能存放重复元素（用对象的 equals()方法来区分元素是否重复）

Map 保存键值对映射，映射关系可以是一对一或多对一。

Set 和 Map 容器都有基于哈希存储和排序树（红黑树）的两种实现版本，基于哈希存储的版本理论存取时间复杂度为O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

## HashMap

hash算法用了按位与而不是取余!!!!!!

取余:xxx%16 不断在做除法,效率低,并且负数取余仍是负数,还需要转为正数

按位与: hash&(length-1)

(length-1) 1111

(hash) 1001

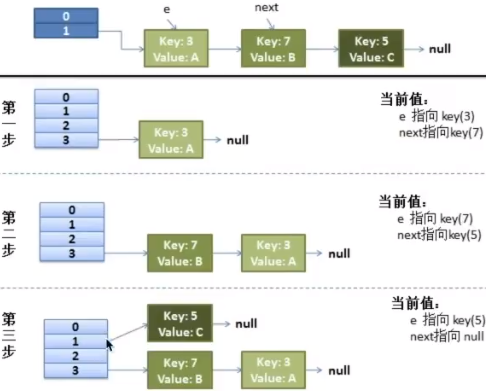
=1001

当length-1不为全1,即length不为2的幂,将出现0,而0的部分按位与永远为0

将导致0的桶永远放不进

### 7 HashMap隐患

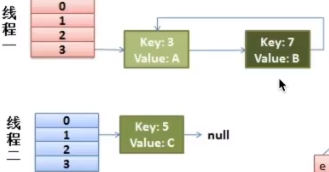
#### 容易死锁



原先: 3->5->7

resize: 7🡪3

多线程环境下,可能同时3->7 7->3,出现循环



当查询时就会出现死锁

#### 可以通过精心设计的一组object实现dos(拒绝服务攻击)

大量的object的HashCode相同,使得它们被存放在同一个桶中,使得HashMap退化为链表,而链表的查询复杂度O(n)

### 8 HashMap

#### hash方法

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); }

jdk7中,容易出现低位相同,高位不同的hash 如1101……….1111

1001………1111

将高位与地位异或(不进位的加法),能够减少碰撞的概率

#### resize方法

Node<K,V> loHead = null, loTail = null;

Node<K,V> hiHead = null, hiTail = null;

扩容时,将原链表拆为两个高低位的链表

比如16个桶,哈希码11111…….11101

1111 =1101

扩容32个桶, 11111 =11101

扩容后第一位只能是0或1,并且桶中的元素被分配在了1xxx和0xxx两个新桶中,元素保持原先的顺序.而保持了顺序就降低了多线程中,顺序调换出现的死锁问题

Map.getOrDefault((Object key, V defaultValue),取不到key时,将返回默认的value

# IO流

使用IO流往往需要多次使用try/catch,如果在一个try/catch中关闭多个流,将会导致 :关闭时其中一个流抛出异常,程序中断,之后的流将不再被关闭!!!需要一条一条的try/catch

# 多线程

## ThreadLocal 的作用和原理。列举在哪些程序中见过 ThreadLocal 的使用？

干脆不要共享资源，为每个线程创造一个资源的复本。将每一个线程存取数据的行为加以隔离，给每个线程特定空间来保管该线程所独享的资源

原理 : ThreadLocal 类中有一个Map，用于存储每一个线程的变量的副本。

## 乐观锁与悲观锁

悲观锁，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁认为别人不会修改，所以不会上锁，但是在更新时判断在此期间别人有没有去更新数据，可以使用版本号等机制,在更新数据时会提高版本号,在提交时,提交版本低于目前版本,将回滚。乐观锁适用于多读的应用类型，可以提高吞吐量

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，反倒降低性能，所以这种情况下用悲观锁就比较合适。

## 同步锁

Java 中每个对象都有一个内置锁。 当程序运行到非静态的 synchronized 同步方法上时，自动获得与正在执行代码类的当前实例（this 实例）有关的锁。获得一个对象的锁也称为获取锁、 锁定对象、在对象上锁定或在对象上同步。 当程序运行到 synchronized 同步方法或代码块时才该对象锁才起作用。 一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以 获得锁，直到第一个线程释放（或返回）锁。这也意味着任何其他线程都不 能进入该对象上的 synchronized 方法或代码块，直到该锁被释放。 释放锁是指持锁线程退出了 synchronized 同步方法或代码块。 关于锁和同步，有一下几个要点： 1）只能同步方法，而不能同步变量和类； 2）每个对象只有一个锁；当提到同步时，应该清楚在什么上同步？也就是 说，在哪个对象上同步？ 3）不必同步类中所有的方法，类可以同时拥有同步和非同步方法。 4）如果两个线程要执行一个类中的 synchronized 方法，并且两个线程使 用相同的实例来调用方法，那么一次只能有一个线程能够执行方法，另一个需要等待，直到锁被释放。也就是说：如果一个线程在对象上获得一个锁， 就没有任何其他线程可以进入（该对象的）类中的任何一个同步方法。 5）如果线程拥有同步和非同步方法，则非同步方法可以被多个线程自由访 问而不受锁的限制。 6）线程睡眠时，它所持的任何锁都不会释放。 7）线程可以获得多个锁。比如，在一个对象的同步方法里面调用另外一个 对象的同步方法，则获取了两个对象的同步锁。 8）同步损害并发性，应该尽可能缩小同步范围。同步不但可以同步整个方 法，还可以同步方法中一部分代码块。 9）在使用同步代码块时候，应该指定在哪个对象上同步，也就是说要获取哪个对象的锁。

## 方法锁和静态方法锁的区别

静态方法，需要对Class对象加锁。

非静态方法，需要对本对象(this)加锁。

## 注意哪个对象正被用于锁定：

1、调用同一个对象中非静态同步方法的线程是互斥的。如果是不同对象，则每个线程有自己的对象的锁，线程间彼此互不干预。

2、调用同一个类中的静态同步方法的线程将是互斥的，它们都是锁定在相同的Class对象上。

3、静态同步方法和非静态同步方法将永远不是互斥的，因为静态方法锁定在Class对象上，非静态方法锁定在该类的对象上。

4、对于同步代码块，要看清楚什么对象已经用于锁定（synchronized后面括号的内容）。在同一个对象上进行同步的线程将是互斥的，在不同对象上锁定的线程将永远不会互斥。

## 同步代码块与同步方法

多个线程访问同一个数据时，容易出现线程安全问题，需要某种方式来确保资源在某一时刻只被一个线程使用。需要让线程同步，保证数据安全线程同步的实现方案：

同步代码块：public void makeWithdrawal(int amt) {

synchronized (acct) { }

}

同步方法：public synchronized void makeWithdrawal(int amt) { }

同步块需要注明锁定对象，同步方法默认锁定this。

在静态方法中，都是默认锁定类对象。

同步块锁定范围小,效率高

## 继承Thread实现Runnable区别。

继承Java.lang.Thread类，并覆盖 run() 方法

优势：编写简单；

劣势：无法继承其它父类

实现 Java.lang.Runnable 接口，并实现 run()方法。

优势：可继承其它类，多线程可共享同一个Thread对象

劣势：编程方式稍微复杂，如需访问当前线程，需调用Thread.currentThread()

## start()和run()的区别

两种方法的区别

1） start 方法：

用 start 方法来启动线程，真正实现多线程运行，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到 cpu 时间片，就开始执行 run()方法，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，Run 方法运行结束，此线程随即终止。

2） run（）：

run()方法只是类的一个普通方法而已，如果直接调用 run 方法，程序中依然只有主线程这一个线程其程序执行路径还是只有一条，还是要顺序执行，还是要等待.

## 线程的6种状态

NEW，新建状态，线程被创建出来，但尚未启动时的线程状态；

RUNNABLE，就绪状态，表示可以运行的线程状态，它可能正在运行，或者是在排队等待操作系统给它分配 CPU 资源；

比如Thread.start方法就是将线程从NEW状态 转换成 RUNNABLE 状态。

BLOCKED，阻塞等待锁的线程状态，表示处于阻塞状态的线程正在等待监视器锁

比如等待执行 synchronized 代码块或者使用 synchronized 标记的方法。

WAITING，等待状态，一个处于等待状态的线程正在等待另一个线程执行某个特定的动作。

比如，一个线程调用了Object.wait()方法，那它就在等待另一个线程调用Object.notify() 或 Object.notifyAll() 方法。

TIMED\_WAITING，计时等待状态，和上者类似，只是多了一个超时时间。

比如调用了有超时时间设置的方法 Object.wait(long timeout) 和 Thread.join(long timeout) 等这些方法时，它才会进入此状态；

TERMINATED，终止状态，表示线程已经执行完成。

## volatile关键字是否能保证线程安全？

不能。虽然 volatile 提供了同步的机制，但是知识一种弱的同步机制，如需要强线程安全，还需要使用 synchronized。

Java 语言提供了一种稍弱的同步机制，即 volatile 变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为 volatile 类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile 变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取 volatile 类型的变量时总会返回最新写入的值。

一、volatile 的内存语义是：

当写一个 volatile 变量时，JMM 会把该线程对应的本地内存中的共享变量值立即刷新到主内存中。

当读一个 volatile 变量时，JMM 会把该线程对应的本地内存设置为无效，直接从主内存中读取共享变量。

二、volatile 底层的实现机制

如果把加入 volatile 关键字的代码和未加入 volatile 关键字的代码都生成汇编代码，会发现加入 volatile 关键字的代码会多出一个 lock 前缀指令。

1 、重排序时不能把后面的指令重排序到内存屏障之前的位置

2、使得本 CPU 的 Cache 写入内存

3、写入动作也会引起别的 CPU 或者别的内核无效化其 Cache，相当于让

新写入的值对别的线程可见。

## 进程和线程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程

是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能

独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中

必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程

的其他的线程共享进程所拥有的全部资源.

## 让线程同时启动

用一个 for 循环创建线程对象，同时调用 wait()方法，让所有线程等待；直到最后一个线程也准备就绪后，调用 notifyAll(), 同时启动所有线程

## 同步和异步

1.如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须同步存取。

2.当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。

## sleep()和 wait()区别

sleep是线程类(Thread)的方法；作用是导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复；调用sleep()不会释放对象锁。

wait 是Object类的方法；对此对象调用wait方法导致本线程放弃对象锁，进入等待此对象的等待锁定池。只有针对此对象发出 notify 方法(或

notifyAll)后本线程才进入对象锁定池，准备获得对象锁进行运行状态。

## sleep()和 yield()区别

① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

② 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

③ sleep()方法声明抛出InterruptedException，而 yield()方法没有声明任何异常；

④ sleep()方法比 yield()方法（跟操作系统相关）具有更好的可移植性

## 当线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法？

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。只有等待当前线程执行完毕释放锁资源之后，其他线程才有可能进行执行该同步方法！

延伸----对象锁分为三种：共享资源、this、当前类的字节码文件对象

## 线程同步相关的方法

1. wait():使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

2. sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此

方法要捕捉 InterruptedException 异常；

3. notify():唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不

能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而

且与优先级无关；

4. notityAll():唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一

个对象的锁，而是让它们竞争；

5. JDK 1.5 通过 Lock 接口提供了显式(explicit)的锁机制，增强了灵活性以

及对线程的协调。Lock 接口中定义了加锁（lock()）和解锁(unlock())的方

法，同时还提供了 newCondition()方法来产生用于线程之间通信的

Condition 对象；

6. JDK 1.5 还提供了信号量(semaphore)机制，信号量可以用来限制对某个

共享资源进行访问的线程的数量。在对资源进行访问之前，线程必须得到信

号量的许可（调用 Semaphore 对象的 acquire()方法）；在完成对资源的

访问后，线程必须向信号量归还许可（调用 Semaphore 对象的 release()

方法）。

## synchronized 关键字的用法

synchronized 关键字可以将对象或者方法标记为同步，以实现对对象

和方法的互斥访问，可以用 synchronized(对象) { … }定义同步代码块，或

者在声明方法时将 synchronized 作为方法的修饰符

## 启动一个线程是用 run()还是 start()方法?

是调用 start()，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

## 线程池（thread pool）

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在 Java 中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是"池化资源"技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

### 线程池参数

最大线程数 maximumPoolSize 性能最高的线程数

核心线程数 corePollSize 平时的流量需要的线程数

线程空闲时间

空闲的线程保留的时间 keepAliveTime

阻塞队列大小

### 队列

一个缓冲的工具，当没有足够的线程去处理任务时，可以将任务放进队列中，以队列先进先出的特性来执行工作任务

核心线程满了，进队列，队列也满了，创建新线程，直到达到最大线程数，之后再超出，会进入拒绝rejectedExecution

## synchronized和Lock的异同？

Lock是Java 5以后引入的新的API

相同点：Lock 能完成synchronized所实现的所有功能；

不同点：Lock 有比 synchronized 更精确的线程语义和更好的性能。synchronized会自动释放锁，而 Lock 一定要求程序员手工释放，并且必须在 finally 块中释放（这是释放外部资源的最好的地方）

## 线程通信的三个方法

Java 提供了 3 个方法解决线程之间的通信问题，均是 java.lang.Object 类的方法，都只能在同步方法或者同步代码块中使用，否则会抛出异常。

final void wait() 表示线程一直等待，直到其它线程通知

void wait(long timeout) 线程等待指定毫秒参数的时间

final void wait(longtimeout,int nanos)线程等待指定毫秒、微妙的时间

final void notify()唤醒一个处于等待状态的线程。注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且不是按优先级。

final void notifyAll()唤醒同一个对象上所有调用 wait()方法的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争

# Mysql

## 事务

事务主要用于处理操作量大，复杂度高的数据。比如说，在人员管理系统中，你删除一个人员，你既需要删除人员的基本资料，也要删除和该人员相关的信息，如信箱，文章等等，这样，这些数据库操作语句就构成一个事务

* 在MySQL中只有使用了Innodb引擎才支持事务。
* 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。
* 事务用来管理insert,update,delete语句

一般来说，事务是必须满足4个条件（ACID）。

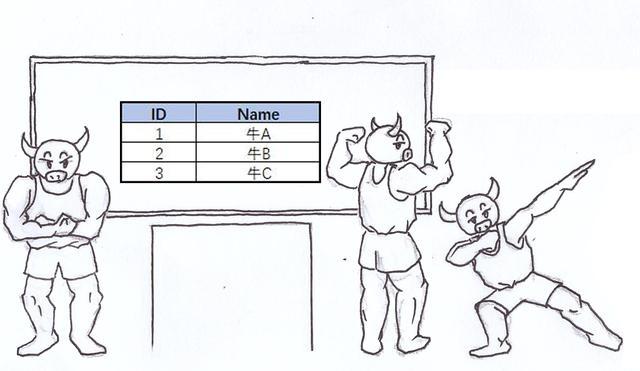
* **A原子性：**事务中的操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样。
* **C一致性：**在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
* **I隔离性：**数据库允许多个并发事务同时对数据进行操作，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。
* **D持久性：**事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

### 隔离4个级别

读未提交、读提交、可重复读和串行化。

如果没有事务隔离，会出现什么样的情况呢？

有这样一张表（T），里面记录了很多牛人的名字，不进行事务的隔离看看会发生什么呢？



第一天，事务A访问了数据库，insert into T values (4, '牛D');往数据库里加上了新来的牛人的名字，但是没有提交事务。

这时，来了另一个事务B，select Name from T;要查询所有牛人的名字。

这时，如果没有事务之间没有有效隔离，那么事务B返回的结果中就会出现“牛D”的名字。这就是“脏读（dirty read）”。

第二天，事务A访问了数据库，他要查看ID是1的牛人的名字，于是执行了

select Name from T where ID = 1;

这时，事务B来了，因为ID是1的牛人改名字了，所以要更新一下，然后提交了事务。

update T set Name = '不牛' where ID = 1;

接着，事务A还想再看看ID是1的牛人的名字，于是又执行了

select Name from T where ID = 1;

结果，两次读出来的ID是1的牛人名字竟然不相同，这就是不可重复读（unrepeatable read）。

第三天，事务A访问了数据库，select \* from T;看看数据库的牛人都有哪些

这时候，事务B来了，insert into T values(4, '牛D')往数据库加入了一个新的牛人。

这时候，事务A忘了刚才的牛人都有哪些了，于是又执行了select \* from T

结果，第一次有三个牛人，第二次有四个牛人。

相信这个时候事务A就蒙了，刚才发生了什么？这种情况就叫“幻读（phantom problem）”。

为了防止出现脏读、不可重复读、幻读等情况，我们就需要根据我们的实际需求来设置数据库的隔离级别。

读未提交（Read Uncommitted） 可以读到未提交的内容。 基本不用

查询不会加锁的，也由于查询的不加锁，所以这种隔离级别的一致性是最差的，可能会产生“脏读”、“不可重复读”、“幻读”。

读提交（Read Committed） 只能读到已经提交了的内容。Oracle 默认

这是各种系统中最常用的一种隔离级别，也是SQL Server和Oracle的默认隔离级别。这种隔离级别能够有效的避免脏读，但除非在查询中显示的加锁，如：

select \* from T where ID=2 lock in share mode;

select \* from T where ID=2 for update;

不然，普通的查询是不会加锁的。

那为什么“读提交”同“读未提交”一样，都没有查询加锁，但是却能够避免脏读呢？

这就要说道另一个机制“快照（snapshot）”，而这种既能保证一致性又不加锁的读也被称为“快照读（Snapshot Read）”可以完成高并发的查询，不过，“读提交”只能避免“脏读”，并不能避免“不可重复读”和“幻读”。

可重复读(Repeated Read)专门针对“不可重复读”这种情况而制定的隔离级别，是MySql的默认隔离级别。

在这个级别下，普通的查询同样是使用的“快照读”，但是，和“读提交”不同的是，当事务启动时，就不允许进行“修改操作（Update）”了，而“不可重复读”恰恰是因为两次读取之间进行了数据的修改，因此，“可重复读”能够有效的避免“不可重复读”，但却避免不了“幻读”，因为幻读是由于“插入或者删除操作（Insert or Delete）”而产生的。

串行化（Serializable） 基本不用

这是数据库最高的隔离级别，这种级别下，事务“串行化顺序执行”，也就是一个一个排队执行。

这种级别下，“脏读”、“不可重复读”、“幻读”都可以被避免，但是执行效率奇差，性能开销也最大，所以基本没人会用。

总结一下

为什么会出现“脏读”？因为没有“select”操作没有规矩。

为什么会出现“不可重复读”？因为“update”操作没有规矩。

为什么会出现“幻读”？因为“insert”和“delete”操作没有规矩。

“读未提（Read Uncommitted）”能预防啥？啥都预防不了。

“读提交（Read Committed）”能预防啥？使用“快照读（Snapshot Read）”，避免“脏读”，但是可能出现“不可重复读”和“幻读”。

“可重复读（Repeated Red）”能预防啥？使用“快照读（Snapshot Read）”，锁住被读取记录，避免出现“脏读”、“不可重复读”，但是可能出现“幻读”。

“串行化（Serializable）”能预防啥？有效避免“脏读”、“不可重复读”、“幻读”，不过效果谁用谁知道。

### 事务处理2种方法：

1、用 BEGIN, ROLLBACK, COMMIT来实现

BEGIN 开始

ROLLBACK 回滚

COMMIT 确认

2、直接用 SET 来改变 MySQL 的自动提交模式:

SET AUTOCOMMIT=0 禁止自动提交

SET AUTOCOMMIT=1 开启自动提交

## 索引

索引虽然能非常高效的提高查询速度，同时却会降低更新表的速度。实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也是要占用空间的。

### 索引类型

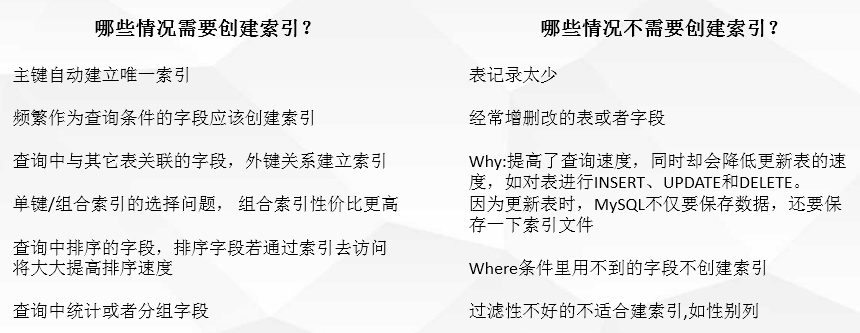
普通索引：仅加速查询

唯一索引：加速查询 + 列值唯一（可以有null）

主键索引：加速查询 + 列值唯一（不可以有null）+ 表中只有一个

组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并

全文索引：对文本的内容进行分词，进行搜索



#### 经常重复值的字段不能索引

数据库中聚集索引只有一个，默认主键。其他用户创建的索引都是非聚集索引。

非聚集索引存储了对主键的引用，即通过索引确定叶子节点之后，还需要再次根据主键去查询数据。（所以会查询两次）

如果非聚集索引重复率高（即一个同样的值有多个主键），那么首先你会从索引中取一半主键值，然后根据主键值再去查询数据，增加了IO，所以特别耗时。

## 引擎

1、MyISAM

使用这个存储引擎，每个MyISAM在磁盘上存储成三个文件。

（1）frm文件：存储表的定义数据

（2）MYD文件：存放表具体记录的数据

（3）MYI文件：存储索引

支持数据的类型也有三种：

（1）静态固定长度表

这种方式的优点在于存储速度非常快，容易发生缓存，而且表发生损坏后也容易修复。缺点是占空间。这也是默认的存储格式。

（2）动态可变长表

优点是节省空间，但是一旦出错恢复起来比较麻烦。

（3）压缩表

支持数据压缩，说明肯定也支持这个格式。在数据文件发生错误时候，可以使用check table工具来检查，而且还可以使用repair table工具来恢复。

不支持事务，但是这也意味着他的存储速度更快，如果你的读写操作允许有错误数据的话，只是追求速度，可以选择这个存储引擎。

2、InnoDB

InnoDB是默认的数据库存储引擎，他的主要特点有：

（1）可以通过自动增长列，方法是auto\_increment。

（2）支持事务。默认的事务隔离级别为可重复读，通过MVCC（并发版本控制）来实现的。

（3）使用的锁粒度为行级锁，可以支持更高的并发；

（4）支持外键约束；外键约束其实降低了表的查询速度，但是增加了表之间的耦合度。

（5）配合一些热备工具可以支持在线热备份；

（6）在InnoDB中存在着缓冲管理，通过缓冲池，将索引和数据全部缓存起来，加快查询的速度；

（7）对于InnoDB类型的表，其数据的物理组织形式是聚簇表。所有的数据按照主键来组织。数据和索引放在一块，都位于B+数的叶子节点上；

当然InnoDB的存储表和索引也有下面两种形式：

（1）使用共享表空间存储：所有的表和索引存放在同一个表空间中。

（2）使用多表空间存储：表结构放在frm文件，数据和索引放在IBD文件中。分区表的话，每个分区对应单独的IBD文件，分区表的定义可以查看我的其他文章。使用分区表的好处在于提升查询效率。

对于InnoDB来说，最大的特点在于支持事务。但是这是以损失效率来换取的。

3、Memory

将数据存在内存，为了提高数据的访问速度，每一个表实际上和一个磁盘文件关联。文件是frm。

（1）支持的数据类型有限制，比如：不支持TEXT和BLOB类型，对于字符串类型的数据，只支持固定长度的行，VARCHAR会被自动存储为CHAR类型；

（2）支持的锁粒度为表级锁。所以，在访问量比较大时，表级锁会成为MEMORY存储引擎的瓶颈；

（3）由于数据是存放在内存中，一旦服务器出现故障，数据都会丢失；

（4）查询的时候，如果有用到临时表，而且临时表中有BLOB，TEXT类型的字段，那么这个临时表就会转化为MyISAM类型的表，性能会急剧降低；

（5）默认使用hash索引。

（6）如果一个内部表很大，会转化为磁盘表。

## limit分页性能

对于小的偏移量，直接使用limit没有问题，但随着数据量的增大，越往后分页，limit语句的偏移量就会越大，速度也会明显变慢。

优化思想：避免数据量大时扫描过多的记录

解决：子查询的分页方式或者JOIN分页方式。JOIN分页和子查询分页的效率基本在一个等级上，消耗的时间也基本一致。

下面举个例子。一般MySQL的主键是自增的数字类型，这种情况下可以使用下面的方式进行优化。

以真实的生产环境的6万条数据的一张表为例，比较一下优化前后的查询耗时：

-- 传统limit，文件扫描

SELECT \* FROM tableName ORDER BY id LIMIT 50000,2;

时间: 0.171s

-- 子查询方式，索引扫描

SELECT \* FROM tableName

WHERE id >= (SELECT id FROM tableName ORDER BY id LIMIT 50000 , 1)

LIMIT 2;

时间: 0.035s

-- JOIN分页方式

SELECT \* FROM tableName AS t1

JOIN (SELECT id FROM tableName ORDER BY id LIMIT 50000, 1) AS t2

WHERE t1.id <= t2.id ORDER BY t1.id LIMIT 2;

时间: 0.036s

优化原理：

子查询在索引上完成的，而普通查询在数据文件上完成。

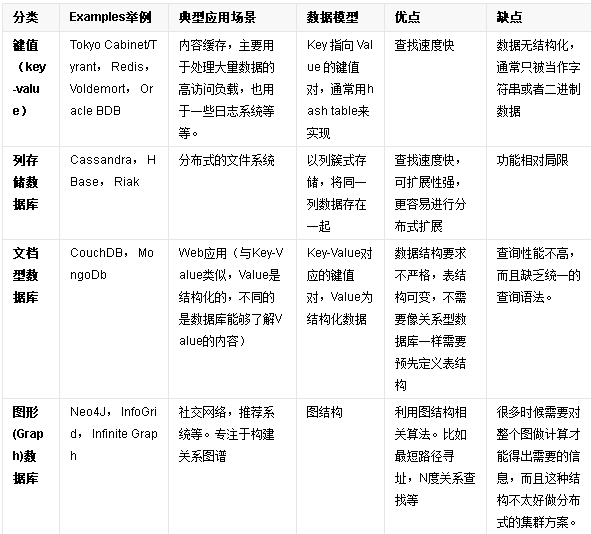
通常来说，索引文件要比数据文件小得多，所以操作起来也会更有效率。因为要取出所有字段内容，第一种需要跨越大量数据块并取出，而第二种基本通过直接根据索引字段定位后，才取出相应内容，效率自然大大提升。因此，对limit的优化，不是直接使用limit，而是首先获取到offset的id，然后直接使用limit size来获取数据。

在实际项目使用，可以利用类似策略模式的方式去处理分页。例如，每页100条数据，判断如果是100页以内，就使用最基本的分页方式；如果大于100，则使用子查询的分页方式。

# 非关系型数据库

不保证关系数据的ACID特性,去掉关系数据库的关系型特性。数据之间无关系，数据库的结构简单,这样就非常容易扩展,也在架构的层面上带来了可扩展的能力。大数据量，高读写性能。这得益于它的无关系性，

不同种类非关系型数据库



# Spring事务传播机制

## 事务的特性

* 原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
* 一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。
* 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
* 持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

## Spring事务的配置方式

Spring支持编程式事务管理以及声明式事务管理两种方式。

### 1. 编程式事务管理

编程式事务管理是侵入性事务管理，使用TransactionTemplate或者直接使用PlatformTransactionManager，对于编程式事务管理，Spring推荐使用TransactionTemplate。

### 2. 声明式事务管理

声明式事务管理建立在AOP之上，其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，执行完目标方法之后根据执行的情况提交或者回滚。  
编程式事务每次实现都要单独实现，但业务量大功能复杂时，使用编程式事务无疑是痛苦的，而声明式事务不同，声明式事务属于无侵入式，不会影响业务逻辑的实现，只需要在配置文件中做相关的事务规则声明或者通过注解的方式，便可以将事务规则应用到业务逻辑中。  
显然声明式优于编程式事务管理，这正是Spring倡导的非侵入式的编程方式。唯一不足的地方就是声明式事务管理的粒度是方法级别，而编程式事务管理是可以到代码块的，但是可以通过提取方法的方式完成声明式事务管理的配置。

## 事务的传播机制

事务的传播性一般用在事务嵌套的场景，比如一个事务方法里面调用了另外一个事务方法，那么两个方法是各自作为独立的方法提交还是内层的事务合并到外层的事务一起提交，这就是需要事务传播机制的配置来确定怎么样执行。  
常用的事务传播机制如下：

* PROPAGATION\_REQUIRED默认

外层有事务，则加入外层事务。没有则新建一个事务执行

* PROPAGATION\_REQUES\_NEW

外层事务挂起，开启新事务，当前事务执行完毕，恢复上层事务的执行。如果外层没有事务，执行当前新开的事务

* PROPAGATION\_SUPPORT  
  外层有事务，则加入外层事务，外层没有事务，以非事务方式执行。完全依赖外层的事务
* PROPAGATION\_NOT\_SUPPORT

不支持事务，如果外层有事务则挂起，执行完当前代码，则恢复外层事务，无论是否异常都不会回滚当前的代码

* PROPAGATION\_NEVER

不支持外层事务，即如果外层有事务就抛出异常

* PROPAGATION\_MANDATORY  
  与NEVER相反，如果外层没有事务则抛出异常
* PROPAGATION\_NESTED可以保存状态保存点，当前事务回滚到某一个点，从而避免所有的嵌套事务都回滚，如果子事务没有把异常吃掉，基本还是会引起全部回滚的。

## 只读

如果一个事务只对数据库执行读操作，那么该数据库就可能利用那个事务的只读特性，采取某些优化措施。通过把一个事务声明为只读，可以给后端数据库一个机会来应用那些它认为合适的优化措施。由于只读的优化措施是在一个事务启动时由后端数据库实施的， 因此，只有对于那些具有可能启动一个新事务的传播行为（REQUIRES\_NEW、EQUIRED、NESTED）的方法来说，将事务声明为只读才有意义。

## 事务超时

事务可能涉及对数据库的锁定，长时间运行事务会不必要地占用数据库资源。这时就可以声明一个事务在特定秒数后自动回滚，不必等它自己结束。

由于超时时钟在一个事务启动的时候开始的，因此，只有对于那些具有可能启动一个新事务的传播行为（REQUIRES\_NEW、REQUIRED、NESTED）的方法来说，声明事务超时才有意义。

## 回滚规则

在默认设置下，事务只在出现运行时异常（runtime exception）时回滚，而在出现受检查异常（checked exception）时不回滚。

不过，可以声明在出现特定受检查异常时像运行时异常一样回滚。同样，也可以声明一个事务在出现特定的异常时不回滚，即使特定的异常是运行时异常。

# Spring声明式事务配置参考

1. 事务的传播性：  
   @Transactional(propagation=Propagation.REQUIRED)
2. 事务的隔离级别：  
   @Transactional(isolation = Isolation.READ\_UNCOMMITTED)

读取未提交数据(会出现脏读, 不可重复读) 基本不使用

1. 只读：  
   @Transactional(readOnly=true)  
   该属性用于设置当前事务是否为只读事务，设置为true表示只读，false则表示可读写，默认值为false。
2. 事务的超时性：  
   @Transactional(timeout=30)
3. 回滚：  
   指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)  
   指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class})

# WEB

## .class 和.jar 类型的文件存放位置？

.class 文件放在 WEB-INF/classes 文件下，.jar 文件放在 WEB-INF/lib文件夹下

## char 型变量中能不能存储一个中文汉字？

java采用unicode编码，2个字节（16位）来表示一个字符， 无论是汉字还是数字，字母，或其他语言都可以存储。

## assert

assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。一般来说，assertion 用于保证程序最基本、关键的正确性。assertion 检查通常在开发和测试时开启。为了提高性能，在软件发布后， assertion 检查通常是关闭的。在实现中，断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为 true；如果表达式计算为 false，那么系统会报告一个 AssertionError。

断言用于调试目的：

assert(a > 0); // throws an AssertionError if a <= 0

断言可以有两种形式：

assert Expression1;

assert Expression1 : Expression2 ;

Expression1 应该总是产生一个布尔值。

Expression2 可以是得出一个值的任意表达式；这个值用于生成显示更多调

试信息的字符串消息。

断言在默认情况下是禁用的，要在编译时启用断言，需使用 source 1.4 标

记：

javac -source 1.4 Test.java

要在运行时启用断言，可使用-enableassertions 或者-ea 标记。

要在运行时选择禁用断言，可使用-da 或者-disableassertions 标记。

要在系统类中启用断言，可使用-esa 或者-dsa 标记。还可以在包的基础上

启用或者禁用断言。可以在预计正常情况下不会到达的任何位置上放置断言。

断言可以用于验证传递给私有方法的参数。不过，断言不应该用于验证传递

给公有方法的参数，因为不管是否启用了断言，公有方法都必须检查其参数。

不过，既可以在公有方法中，也可以在非公有方法中利用断言测试后置条件。

另外，断言不应该以任何方式改变程序的状态。

## Session

### Session的两种实现方法

#### 1、基于Cookie实现Session

服务器为客户端创建并维护Session对象，用于存放数据。同时会产生SessionID，服务器以Cookie的方式将SessionID存放在客户端。当浏览器再次访问该服务器时，会将SessionID作为Cookie信息带到服务器，服务器可以通过该SessionID检索到以前的Session对象，并对其进行访问。需要注意的是，此时的Cookie中仅仅保存了一个SessionID，而相对较多的会话数据保存在服务器端对应的Session对象中，由服务器来统一维护，这样一定程度保证了会话数据安全性，但增加了服务器端的内存开销。

存放在客户端的用于保存SessionID的Cookie会在浏览器关闭时清除。我们把用户打开一个浏览器访问某个应用开始，到关闭浏览器为止交互过程称为一个“会话”。在一个“会话”过程中，可能会向同一个应用发出了多次请求，这些请求将共享一个Session对象，因为这些请求携带了相同的SessionID信息。

#### 2、基于URL重写

Session对象的正常使用要依赖于Cookie。如果考虑到客户端浏览器出于安全的考虑禁用了Cookie，应该使用URL重写的方式使Session在客户端禁用Cookie的情况下继续生效。

### session 与 cookie 的区别

存储角度：

Session是服务器端的数据存储技术，cookie是客户端的数据存储技术

解决问题角度：

Session 解决的是同一用户不同请求的数据共享问题，cookie 解决的是不同用户不同请求的请求数据的共享问题

生命周期角度：

Session的id是依赖于cookie来进行存储的，浏览器关闭 id 就会失效

Cookie 可以单独设置其在浏览器的存储时间。

## HTTP状态码

### 2xx 成功处理了请求

### 3xx (重定向)表示要完成请求，需要进一步操作。

304 (未修改) 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。

### 4xx(请求错误)请求可能出错，妨碍服务器的处理

400 (错误请求) 服务器不理解请求的语法。

403 (禁止) 服务器拒绝请求。

### 5xx(服务器错误)

500 (服务器内部错误) 服务器遇到错误，无法完成请求。

501 (尚未实施) 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法

502 (错误网关) 服务器作为网关或代理，从上游服务器收到无效响应。

503 (服务不可用) 服务器目前无法使用(由于超载或停机维护)。 通常，这只是暂时状态。

504 (网关超时) 服务器作为网关或代理，但是没有及时从上游服务器收到请求。

505 (HTTP 版本不受支持) 服务器不支持请求中所用的 HTTP 协议版本。

### Ajax 的工作原理

异步的javascript和xml,通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用javascript来操作DOM而更新页面。从而实现向服务器提出请求和处理响应，而不阻塞用户。达到无刷新的效果

### JSON 及其作用

JSON是一种轻量级的数据交换格式，采用完全独立于语言的文本格式，是理想的数据交换格式。同时，JSON 是 JavaScript 原生格式，这意味着在 JavaScript 中处理 JSON 数据不须要任何特殊的 API 或工具包。

在 JSON 中，有两种结构：对象和数组。

{} 对象

[] 数组

, 分隔属性

: 左边为属性名，右边为属性值

属性名可用可不用引号括起，属性值为字符串一定要用引号括起

## Servlet

### Servlet生命周期

Web容器加载Servlet并将其实例化后，Servlet生命周期开始，容器运行init()方法进行Servlet的初始化；请求到达时调用service方法，service方法调用与请求对应的doGet或doPost等方法；当服务器关闭或项目被卸载时Servlet 实例被销毁，此时会调用Servlet的destroy方法

### JSP 和 Servlet关系

先有 Servlet，针对 Servlet 缺点推出 JSP。JSP 是 Servlet 的一种特殊形式

Servlet是特殊的Java程序，运行于服务器的JVM中，能够依靠服务器的支持向浏览器提供显示内容。JSP本质上是Servlet的一种简易形式，JSP被处理成类似于Servlet的Java程序，可以简化页面内容的生成。

不同点在于，Servlet 的应用逻辑在Java文件中，并完全从表示层中的HTML分离开来。而JSP是Java语言和HTML的组合。JSP侧重于视图，Servlet侧重于控制逻辑，在MVC架构模式中，JSP适合充当视图而Servlet 适合充当控制器（controller）

每个 JSP 页面就是一个 Servlet 实例——JSP 页面由系统翻译成 Servlet，

Servlet 再负责响应用户请求。

# 设计模式

### 工厂设计模式：

由一个工厂类根据传入的参数（一般是字符串参数），动态决定应该创建哪一个产品子类（这些产品子类继承自同一个父类或接口）的实例，并以父类形式返回

优点：客户端不负责对象的创建，而是由专门的工厂类完成；客户端只负责对象的调用，实现了创建和调用的分离，降低了客户端代码的难度；

缺点：如果增加和减少产品子类，需要修改简单工厂类，违背了开闭原则；如果产品子类过多，会导致工厂类非常的庞大，违反了高内聚原则，不利于后期维护

### 饿汉单例模式

\* 在类加载的时候创建实例，而不是等到第一次请求实例时创建

\* 1、私有的无参数构造方法，避免外部创建实例

\* 2、私有静态属性instance

\* 3、公有静态方法getInstance()

### 懒汉单例模式

\*在类加载的时候不创建单例实例，只有在第一次请求实例时创建

### 适配器模式

两个不兼容的接口之间的桥梁,结合了两个独立接口的功能.这种模式涉及到一个单一的类，该

类负责加入独立的或不兼容的接口功能。

属于结构型模式

### 模板模式

一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行

### 编程原则

• 单一职责原则：

高内聚:一个类只做它该做的事情。一个代码模块只完成一项功能，而不涉及与它无关的领域

模块化，每个功能模块是可以轻易的拿到其他系统中使用的，这样才能实现软件复用

• 开闭原则：软件实体应当对扩展开放，对修改关闭。要做到开闭有两个要点：

①抽象是关键，没有抽象类或接口就没有扩展点；

②封装可变性，将系统中的各种可变因素封装到一个继承结构中

• 依赖倒转原则：面向接口编程。（声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代）

• 里氏替换原则：任何时候都可以用子类型替换掉父类型。子类一定是增加父类的能力而不是减少父类的能力

• 接口隔离原则：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。

• 合成聚合复用原则：优先使用聚合或合成关系复用代码。（类与类之间简单的说有三种关系，IS-A、HAS-A、USE-A，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是 HAS-A 关系

• 迪米特法则：又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（“低耦合”，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web 开发中作为前端控制器的 Servlet 或 Filter 不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）

# 数据结构

## 堆

堆就是用数组实现的二叉树，所有它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

堆分为两种：最大堆和最小堆，两者的差别在于节点的排序方式。

在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。

### 堆和普通树的区别

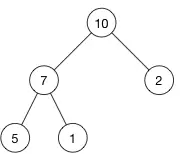
节点顺序。在二叉搜索树中，左子节点必须比父节点小，右子节点必须必比父节点大。但是在堆中并非如此。在最大堆中两个子节点都必须比父节点小，而在最小堆中，它们都必须比父节点大。

内存占用。普通树占用的内存空间比它们存储的数据要多。你必须为节点对象以及左/右子节点指针分配额为是我内存。堆仅仅使用一个数据来存储数组，且不使用指针。

平衡。二叉搜索树必须是“平衡”的情况下，其大部分操作的复杂度才能达到O(log n)。你可以按任意顺序位置插入/删除数据，或者使用 AVL 树或者红黑树，但是在堆中实际上不需要整棵树都是有序的。我们只需要满足对属性即可，所以在堆中平衡不是问题。因为堆中数据的组织方式可以保证O(log n) 的性能。

搜索。在二叉树中搜索会很快，但是在堆中搜索会很慢。在堆中搜索不是第一优先级，因为使用堆的目的是将最大（或者最小）的节点放在最前面，从而快速的进行相关插入、删除操作。

### 堆的存储方式

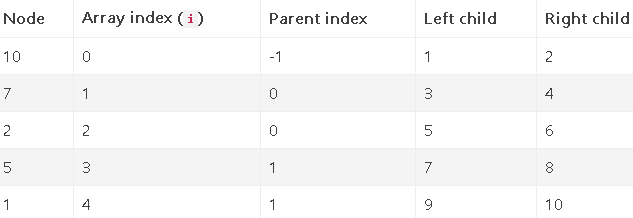
[ 10, 7, 2, 5, 1 ]

如果 i 是节点的索引，那么下面的公式就给出了它的父节点和子节点在数组中的位置

parent(i) = floor((i - 1)/2)

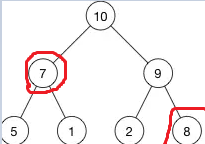
left(i) = 2i + 1

right(i) = 2i + 2



根节点(10)没有父节点，因为 -1 不是一个有效的数组索引。同样，节点 (2)，(5)和(1) 没有子节点，因为这些索引已经超过了数组的大小，所以我们在使用这些索引值的时候需要保证是有效的索引值。

堆并不一定是有序数组！要将堆转换成有序数组，需要使用堆排序。

堆中低层节点不一定小于高层节点！7<8

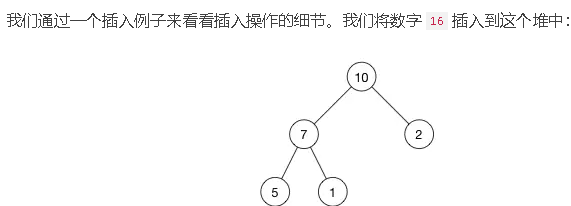
### 两个保证插入或删除节点后依然是最大/小堆的方法

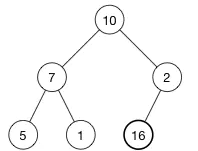
shiftUp(): 如果一个节点比它的父节点大（最大堆）或者小（最小堆），那么需要将它同父节点交换位置。这样是这个节点在数组的位置上升。

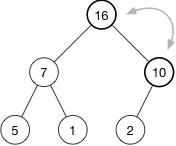
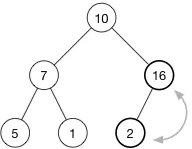
shiftDown(): 如果一个节点比它的子节点小（最大堆）或者大（最小堆），那么需要将它向下移动。这个操作也称作“堆化（heapify）”。

shiftUp 或者 shiftDown 是一个递归的过程，所以它的时间复杂度是 O(log n)

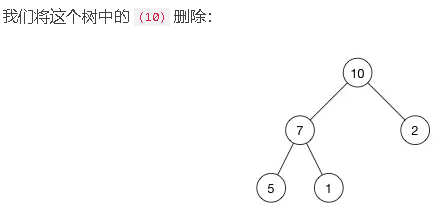
### 插入

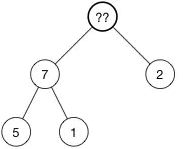


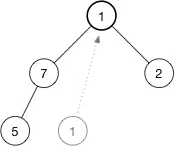
变成了[ 10, 7, 2, 5, 1, 16 ] 

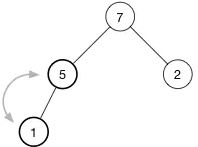
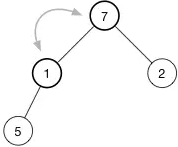
为了恢复堆属性,交换两次

### 删除根节点



此时顶部为空节点

取出最后一个节点,代替空节点

之后开始交换

### 删除子节点

remove() 的通用版本，它可能会使用到 shiftDown 和 shiftUp。

[ 10, 7, 2, 5, 1 ] 删除7

-> [ 10, 1, 2, 5, 7 ] 将最后的1与7交换,7就是需要返回的元素,用removeLast()删除,而1的位置不对,用shiftDown()调整位置

## 时间复杂度

二分查找 对数级别 logN

循环 线性级别 N

归并(分治思想) 线性对数级别 NlogN

穷举 指数级别 2^N

# 常用注解

pojo是一个统称，可以是DTO、可以是VO、可以是PO、可以是domain，这些都叫pojo  
po、do、domain三者区别很小，用处都是和数据库进行对应

DTO是数据传输对象，简单点说就是传参数用的。比如一张表30个字段，但是传参的时候只需要传5个字段，这个时候使用dto，可以避免过多的无用数据，也可以隐藏后端表结构，往往是前端传参给后端、controller、service、dao三层之间传递使用。  
VO就是view Object，专门负责给前端展示数据。VO是业务对象，业务上需要什么字段它就给什么字段，比如上面说的学生表，在给前端展示的时候需要展示名称：class\_name这个字段，那么vo里写的就不是classId，而是className或者班级类

do、vo、dto这些类之间不需要给继承关系，从设计思想来讲也不能给继承关系。互相之间的属性复制使用spring提供的BeanUtils.copyProperties。如果是集合这种数据较多的属性复制，就先转成json字符串再转成另一个类的List

## Junit注解

@BeforeClass – 表示在类中的任意public static void方法执行之前执行

@AfterClass – 表示在类中的任意public static void方法执行之后执行

@Before – 表示在任意使用@Test注解标注的public void方法执行之前执行

@After – 表示在任意使用@Test注解标注的public void方法执行之后执行

@AfterRunning: 返回通知, 在方法返回结果之后执行

@AfterThrowing: 异常通知, 在方法抛出异常之后

@Around: 环绕通知, 围绕着方法执行

日期格式处理

@JsonFormat 后台到前台

properties文件中有相同的配置spring.mvc.date-format

@DateTimeFormat 前台到后台

properties文件中有相同的配置spring.jackson.date-format

spring.jackson.time-zone

@Respostory

@Compment 把切面类加入到IOC容器中

@EnableAspectJAutoProxy //开启对AspectJ语法风格的支持

@ControllerAdvise 当Controller出现异常时,跳转页面

@RestControllerAdvise ,返回json

@bean作用在方法上

@import引入其他的配置文件

@ComponentScan(“”) 配置扫描

当ioc容器里有多个同名对象时

@Qualifier 合格者，表明哪个bean是需要的

Qualifier的参数必须是之前用@Bean注解过的

@Primary 指明优先级

@Configuration(proxyBeanMethods = false) proxyBeanMethods决定了配置类是否会被代理,如果@Bean方法间没有调用关系的话可以把 proxyBeanMethods 设置为 false。否则，方法内部引用的类生产的类和 Spring 容器中类是两个类。

@ConditionalOnBean // 当给定的在bean存在时,则实例化当前Bean @ConditionalOnMissingBean // 当给定的在bean不存在时,则实例化当前Bean @ConditionalOnClass // 当给定的类名在类路径上存在，则实例化当前Bean @ConditionalOnMissingClass // 当给定的类名在类路径上不存在，则实例化当前Bean

@ConfigurationProperties(prefix = "spring.redis") 配置类注解,prefix是配置时的前缀

@EnableConfigurationProperties(RedisProperties.class) 加载配置类

**@Lazy 懒加载**

***注入userService时,CacheAspect中自定义的切面增强还没有被加载  
导致注入进去的是还未被动态代理的,原生的userService***

此时需要运用运用懒加载

***@TableField(exist = false) 实体类中,数据库不存在的字段需要加上这个注解***

* @Transactional 是声明式[事务管理](https://blog.csdn.net/mingyundezuoan/article/details/79167332) 编程中使用的注解
* 添加位置:接口实现类或接口实现方法上，而不是接口类中
* 访问权限：public 的方法才起作用

@Transactional实现原理

1)事务开始时，通过AOP机制，生成一个代理connection对象，并将其放入DataSource实例的某个与DataSourceTransactionManager相关的某处容器中。在接下来的整个事务中，客户代码都应该使用该connection连接数据库，执行所有数据库命令[不使用该connection连接数据库执行的数据库命令，在本事务回滚的时候得不到回滚]（物理连接connection逻辑上新建一个会话session；DataSource与TransactionManager配置相同的数据源）

2)事务结束时，回滚在第1步骤中得到的代理connection对象上执行的数据库命令，然后关闭该代理connection对象（事务结束后，回滚操作不会对已执行完毕的SQL操作命令起作用）

## 五大元注解

### @Target：限制注解能修饰的对象范围

/\*\*用于描述类、接口(包括注解类型) 或enum声明\*/

    TYPE,

    /\*\* 用于描述域 Field declaration (includes enum constants) \*/

    FIELD,

    /\*\*用于描述方法 Method declaration \*/

    METHOD,

    /\*\*用于描述参数 Formal parameter declaration \*/

    PARAMETER,

    /\*\*用于描述构造器 Constructor declaration \*/

    CONSTRUCTOR,

    /\*\*用于描述局部变量 Local variable declaration \*/

    LOCAL\_VARIABLE,

    /\*\* 注解类 \*/

    ANNOTATION\_TYPE,

    /\*\*用于描述包 Package declaration \*/

    PACKAGE,

    /\*\*用来标注类型参数 Type parameter declaration     \*/

    TYPE\_PARAMETER,

    /\*\*所有类型  \*/

    TYPE\_USE

### **@Retention**

Retention注解有一个RetentionPolicy类型的枚举属性

有3个值：CLASS  RUNTIME   SOURCE标明注解应该如何去保持,这也是它的生命周期  
RetentionPolicy.SOURCE：注解只保留在源文件，当Java文件编译成class文件的时候，注解被遗弃；  
RetentionPolicy.CLASS：注解被保留到class文件，但jvm加载class文件时候被遗弃，默认RetentionPolicy.RUNTIME：注解不仅被保存到class文件中，jvm加载class文件之后，仍然存在；  
这3个生命周期分别对应于：Java源文件(.java文件) ---> .class文件 ---> 内存中的字节码。

生命周期长度 SOURCE < CLASS < RUNTIME ，所以前者能作用的地方后者一定也能作用。  
需要在运行时动态获取注解信息，那只能用RUNTIME注解，比如@Deprecated使用RUNTIME注解  
在编译时进行预处理操作，比如生成一些辅助代码（如 ButterKnife），就用 CLASS注解；  
只是检查性的操作，比如 @Override 和 @SuppressWarnings，使用SOURCE 注解。

### @Documented将注解包含在javadoc中

默认情况下，javadoc是不包括注解的，但如果使用了@Documented注解，则相关注解类型信息会被包含在生成的文档中

### @Inherited指明父类注解会被子类继承得到

### @Repeatable

指明注解为可重复注解，可以在同一个地方多次使用

## @Scheduled定时

八大参数

### Cron 定时时间

@Scheduled(cron = "0 0 5 \* \* ?") [秒] [分] [小时] [日] [月] [周] [年]

允许正则表达式,

? 不指定值

- 区间

, 指定多个值

/ 递增触发。秒”5/15” 表示从5秒开始，每增15秒触发

L 最后。对于日字段，表示当月的最后一天.对于周字段上设置”6L”这样的格式,则表示“本月最后一个星期五”

W 离指定日期的最近的工作日(周一至周五). 例如在日字段上置”15W”，表示离每月15号最近的那个工作日触发。如果15号正好是周六，则找最近的周五(14号)触发, 如果15号是周未，则找最近的下周一(16号)触发.如果15号正好在工作日(周一至周五)，则就在该天触发。如果指定格式为 “1W”,它则表示每月1号往后最近的工作日触发。如果1号正是周六，则将在3号下周一触发。(注，”W”前只能设置具体的数字,不允许区间”-“)。

# 序号(表示每月的第几个周几)，例如在周字段上设置”6#3”表示在每月的第三个周六.注意如果指定”#5”,正好第五周没有周六，则不会触发该配置

’L’和‘W’组合使用。在日字段上设置”LW”,则表示在本月的最后一个工作日触发；周字段的设置，若使用英文字母是不区分大小写的，即MON与mon相同。

### zone时区.一般留空

fixedDelay上一次执行完毕后多长时间再执行

@Scheduled(fixedDelay = 5000) //上一次执行完毕时间点之后5秒再执行

fixedDelayString 同上的字符串形式,支持占位符

@Scheduled(fixedDelayString = "5000") //上一次执行完毕时间点之后5秒再执行  
fixedRate上一次开始执行后多长时间再执行

@Scheduled(fixedRate = 5000) //上一次开始执行时间点之后5秒再执行

fixedRateString同上的字符串形式。支持占位符

initialDelay第一次延迟多长时间后再执行

@Scheduled(initialDelay=1000, fixedRate=5000) //第一次延迟1秒后执行，之后按fixedRate的规则每5秒执行一次

initialDelayString同上的字符串形式。支持占位符

# Environment类(Spring自身的类)

可以把Spring应用的运行时分成两个部分：一个是Spring应用本身，一个是Spring应用所处的环境。

定时注解@Scheduled有时需要获取当前的运行环境(用户信息,配置文件信息等)

Environment在容器中是一个抽象的集合，是指应用环境的2个方面：profiles和properties。

1. Profile

  不管是XML还是注解，Beans都有可能指派给profile配置。Environment环境对象的作用，对于profiles配置来说，它能决定当前激活的是哪个profile配置，和哪个profile是默认。

2. Properties

  properties来源于properties文件、JVM properties、system环境变量、JNDI、servlet context parameters上下文参数、专门的properties对象，Maps等等。对于properties来说，Environment对象可以提供给用户方便的服务接口、方便撰写配置、方便解析配置。  
 environment.getProperty获取配置文件中的属性

# REST

## [无状态(stateless)](https://www.cnblogs.com/dogharry/p/4305044.html)

在请求中传递SessionID被普遍认为是unRESTful的，而将用户的credentials(认证信息)包含在每个请求里又是一种非常RESTful的做法。这样一个问题经常会造成困扰。本文就REST的一些概念进行了探讨，解释了REST架构中的状态，无状态（stateless），以及两种状态的区别

“状态”通常指的是为两个相互关联的用户交互操作保留的某种公共信息，它们常常被用来存储工作流或用户状态信息等数据。这些信息可以被指定不同的作用域如page，request，session或全局作用域，而存储他们的责任也同样可以由Client端或Server端负责。虽然存储状态为企业软件开发带来了诸多便利，但是它也给分布式系统的其他方面带来了许多限制，比如在负载均衡方面，在有状态的模式下，一个用户的请求必须被提交到保存有其相关状态信息的服务器上，否则这些请求可能无法被理解，这也就意味着在此模式下服务器端无法对用户请求进行自由调度。于此相关的另一个问题是容错性，倘若保有用户信息的服务器宕机，那么该用户最近的所有交互操作将无法被透明地移送至备用服务器上，除非该服务器时刻与主服务器同步全部用户的状态信息。此外，由于HTTP本身不是一个有状态的协议，开发人员必须通过模拟实现状态的钝化与激活等。

无状态指的是任意一个Web请求必须完全与其他请求隔离，当请求端提出请求时，请求本身包含了相应端为相应这一请求所需的全部信息。这一约束的出现改善了分布式系统的可见性、可靠性以及可伸缩性。如果一个网站期望用户以A->B->C的流程来交互，而在执行至B时回退的话，那么系统很有可能不是按照其所期望的方式运行，因为用户的状态可能被不可逆地修改了。反过来，任何用户可以在浏览器地址栏中输入http://www.google.com/search?q=RESTful&start=100来获得从第一百条开始的关于RESTful的记录，并且当Google服务器瘫痪时，相关用户请求会被透明地移送至其他服务器。

RESTful架构对于state的两个不同的解释: 应用状态(Application State)和资源状态(Resource State)。应用状态指的是与某一特定请求相关的状态信息，而资源状态则反映了某一存储在服务器端资源在某一时刻的特定状态，该状态不会因为用户请求而改变，任何用户在同一时刻对该资源的请求都会获得这一状态的表现（Representation）。RESTful架构要求服务器端不保有任何与特定HTTP请求相关的资源，所以应用状态必须由请求方在请求过程中提供。那么为什么传递一个session ID是违背REST架构风格而传递user credentials却不是。实际上“传递某种表示状态的信息”到服务器不是“有状态”的表现。其实有状态和无状态与请求本身没有多大关联，重要的是状态信息是由请求方还是响应方负责保存。在Session ID可以被认为是一个用来标识某一会话状态的Key，将其传递给服务器端意味着这样一个请求：“请帮我取出这个状态信息”，也就是说这个请求假设响应方保有着状态信息。由于与某一特定请求相关的状态属于应用状态，而RESTful架构要求任何此类状态由请求方负责提供，所以传递Session ID被认为是unRESTful的做法。反过来，user credential作为一种应用状态，是被期望由请求方提供的，所以在请求中传递user credentials是符合RESTful架构规范的。

@Deprecated，用来表示某个类或属性或方法已经过时

@SuppressWarnings用来压制程序中出来的警告，比如在没有用泛型或是方法已经过时的时候

## 注解还是XML

1、应用的基本配置用xml，比如：数据源、资源文件等；

2、业务开发用注解，比如：Service中注入bean等；

## SpringBoot的优点

1，创建独立的spring应用程序。

2，嵌入的tomcat jetty 或者undertow 不用部署WAR文件。

3，允许通过Maven来根据需要获取starter

4，尽可能的使用自动配置spring

5，提供生产就绪功能，如指标，健康检查和外部配置

6，绝对没有代码生成，对XML没有要求配置

# 【熟悉】springboot和微服务的介绍

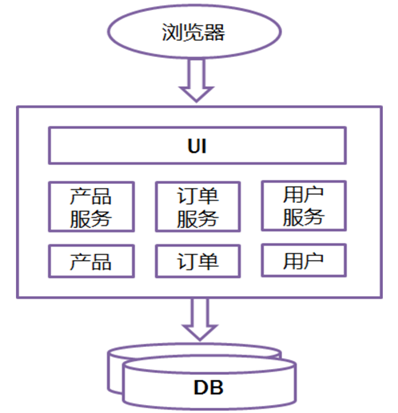
　　开箱即用，没有代码生成，也无需 XML 配置。同时也可以修改默认值来满足特定的需求。

　　提供了一些大型项目中常见的非功能特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。

　　Spring Boot 并不是对 Spring 功能上的增强，而是提供了一种快速使用 Spring 的方式。

## 传统开发模式

所有的功能打包在一个 WAR包里，基本没有外部依赖（除了容器），部署在一个JEE容器（Tomcat，JBoss，WebLogic）里，包含了 DO/DAO，Service，UI等所有逻辑。



优点：

①开发简单，集中式管理

②基本不会重复开发

③功能都在本地，没有分布式的管理和调用消耗

缺点：

1、效率低：开发都在同一个项目改代码，相互等待，冲突不断

2、维护难：代码功功能耦合在一起，新人不知道何从下手

3、不灵活：构建时间长，任何小修改都要重构整个项目，耗时

4、稳定性差：一个微小的问题，都可能导致整个应用挂掉

5、扩展性不够：无法满足高并发下的业务需求

6、对服务器的性能要求要统一，要高

# 单点登录、域用户、常规登录、AD域

1、单点登录

（1）啥是单点登录？

用户只需要登录一次就可以访问所有相互信任的应用系统。（Single Sign On，简称为 SSO）

（2）解决啥问题？

各个server拿到同一个ID，都能有办法检验出ID的有效性、并且能得到ID对应的用户信息。

（3）咋实现？

实现步骤：

登录应用1，服务器前验证 ——> Over（不通过，不通过验证失败）

                                         ——> 返回ticket（验证通过）  ——> 下次访问应用,2，发送ticket验证   ——> 以后登录应用2不需要再次登录（验证通过）

实现SSO，具备条件所有应用系统共享一个身份认证系统。

　　统一的认证系统是SSO的前提之一。认证系统的主要功能是将用户的登录信息和用户信息库相比较，对用户进行登录认证；认证成功后，认证系统应该生成统一的认证标志（ticket），返还给用户。另外，认证系统还应该对ticket进行效验，判断其有效性。所有应用系统均能够识别和提取ticket信息

　　要实现SSO的功能，让用户只登录一次，就必须让应用系统能够识别已经登录过的用户。应用系统应该能对ticket进行识别和提取，通过与认证系统的通讯，能自动判断当前用户是否登录过，从而完成单点登录的功能。

2、域用户

（1）何为域？何为域用户？

域，域就是一方诸侯，有自己的权限和领土范围。（自己的装逼解释）；而领地内所有管辖和被管辖的都是域用户。

（2）域，还是域用户？

看了所有能看到的解释，都不满意。自己定义下，域的主要是，域对内的规则，域内用户的权限，域和域之间的规则；同域内用户之间的关系，权限。域，例如计算机里的域用户是需要服务器，进行域的创建，

（3）域账号

域账号，修改域帐号有关数据，直接修改域帐号服务器中的帐号，其他计算机就可立即获取更新后的帐号数据；本地账号，域中有数十台计算机，而且每一台计算机都必须有相同的帐号

3、AD域

AD域是Active Directory的缩写，它是基于windows的一个组合，它可以集中控制加入域的所有计算机的权限，更高效的分配权限、提高资料的安全性、节省管理成本。

域用户（我这里指的是创建域的这个用户），在任何一台加入域的计算机上都有管理员的权限。