java基础

概述

1. 面向对象和面向过程的区别?
   答：面向过程的思想是自顶向下，逐步细化的实现功能，面向对象是封装事物的属性和行为，即抽象化后，协调调用事物来实现功能。
   面向过程
   优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源，比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。
   缺点：没有面向对象易维护、易复用、易扩展
   面向对象
   优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护。
   缺点：性能比面向过程低
2. 面向对象的特征有哪些?
   答:
   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,抽象只关注对象的哪些属性和行为,并不关注这此行为的细节是什么。
   封装：将对象抽象称为一个高度自治和抽象的个体，对象的状态由这个对象自己的行为（方法）来读取。
   继承：是从已有类得到继承信息创建新类的过程,继承让变化中的软件系统有了一定的延续性。
   多态：指允许不同类的对象对同一消息作出响应。
3. jdk与jre的区别与联系?
   答:
   JDK：Java Development Kit 的简称，Java 开发工具包，提供了 Java 的开发环境和运行环境。
   JRE：Java Runtime Environment 的简称，Java 运行环境，为 Java 的运行提供了所需环境。
   具体来说 JDK 其实包含了 JRE，同时还包含了编译 Java 源码的编译器 Javac，还包含了很多 Java 程序调试和分析的工具。简单来说：如果你需要运行 Java 程序，只需安装 JRE 就可以了，如果你需要编写 Java 程序，需要安装 JDK。

基本数据类型

1. 强类型语言和弱类型语言区别?
   答:
   强类型定义语言：强制数据类型定义的语言。也就是说，一旦一个变量被指定了某个数据类型，如果不经过强制转换，那么它就永远是这个数据类型了。
   弱类型定义语言：数据类型可以被忽略的语言。它与强类型定义语言相反, 一个变量可以赋不同数据类型的值。
   强类型定义语言在速度上可能略逊色于弱类型定义语言，但是强类型定义语言带来的严谨性能够有效的避免许多错误。
2. java基本数据类型有哪些?
   答:
   byte：8位，最大存储数据量是255，存放的数据范围是-128~127之间。
   short：16位，最大数据存储量是65536，数据范围是-32768~32767之间。
   int：32位，最大数据存储容量是2的32次方减1，数据范围是负的2的31次方到正的2的31次方减1。
   long：64位，最大数据存储容量是2的64次方减1，数据范围为负的2的63次方到正的2的63次方

减1。

float：32位，数据范围在3.4e-45~1.4e38，直接赋值时必须在数字后加上f或F。

double：64位，数据范围在4.9e-324~1.8e308，赋值时可以加d或D也可以不加。

boolean：只有true和false两个取值。Java虚拟机不提供操作boolean类型的字节码指令，程序在编译后boolean类型都转化成了int操作。但是Java虚拟机支持boolean类型的数组的访问和修改，共用byte类型数组的字节码指令。

char：16位，存储Unicode码，用单引号赋值。

returnAddress类型:会被Java虚拟机的jsr、ret和jsr_w指令所使用。returnAddress类型的值指向一条虚拟机指令的操作码。与前面介绍的那些数值类的原生类型不同，returnAddress类型在Java语言之中并不存在相应的类型，也无法在程序运行期间更改returnAddress类型的值。

3. String属于基本数据类型吗?

答:

属于对象。

4. 数值类型转换规则?

答:

简单类型数据间的转换，有两种方式:自动转换和强制转换

自动转换

具体地讲,当一个较"小"数据与一个较"大"的数据一起运算时,系统将自动将"小"数据转换成"大"数据,再进行运算。这些类型由"小"到"大"分别为 (byte，short，char)--int--long--float—double。这里我们所说的"大"与"小",并不是指占用字节的多少,而是指表示值的范围的大小。

①下面的语句可以在Java中直接通过:

byte b;int i=b; long l=b; float f=b; double d=b;

②如果低级类型为char型，向高级类型（整型）转换时，会转换为对应ASCII码值，例如

char c='c'; int i=c;System.out.println("output:"+i);输出：output:99;

③对于byte,short,char三种类型而言，他们是平级的，因此不能相互自动转换，可以使用下述的强制类型转换。

short i=99 ; char c=(char)i; System.out.println("output:"+c);输出：output:c;

强制转换

将"大"数据转换为"小"数据时，你可以使用强制类型转换。即你必须采用下面这种语句格式： int n=(int)3.14159/2;可以想象，这种转换肯定可能会导致溢出或精度的下降。

表达式的数据类型自动提升注意下面的规则:

①所有的byte,short,char型的值将被提升为int型;

②如果有一个操作数是long型，计算结果是long型;

③如果有一个操作数是float型，计算结果是float型;

④如果有一个操作数是double型，计算结果是double型;

例， byte b; b=3; b=(byte)(b*3);//必须声明byte。

对于short i =1;i = i+1;由于1是int类型，因此i+1运算结果也是int类型，需要强制转换类型才能赋值给short类型；而short i = 1； i+=1;可以正确编译，因为i+=1；相当于i=(short)(i+1);其中有隐含的强制类型转换。拓展：short i = 1;i++;也是正确的。需要注意的是，整数都默认为int类型，除非你将其定义为short类型，因此1+short类型的i还是int类型，所以会报错；而i++并没有经过i+1赋值语句，因此不会报错，而i+=1等价于i++，所以也不会报错。

5. unicode详解?

答:

这里引用阮一峰的一篇日志《字符编码笔记：ASCII，Unicode 和 UTF-8》，具体如下:

ASCII 码

我们知道，计算机内部，所有信息最终都是一个二进制值。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从00000000到11111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为 ASCII 码，一直沿用至今。

ASCII 码一共规定了128个字符的编码，比如空格SPACE是32（二进制00100000），大写的

字母A是65（二进制01000001）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的一位统一规定为0。

非 ASCII 编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用 ASCII 码表示。于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130（二进制10000010）。这样一来，这些欧洲国家使用的编码体系，可以表示最多256个符号。

但是，这里又出现了新的问题。不同的国家有不同的字母，因此，哪怕它们都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码中代表了é，在希伯来语编码中却代表了字母Gimel (ג)，在俄语编码中又会代表另一个符号。但是不管怎样，所有这些编码方式中，0--127表示的符号是一样的，不一样的只是128--255的这一段。

至于亚洲国家的文字，使用的符号就更多了，汉字就多达10万左右。一个字节只能表示256种符号，肯定是不够的，就必须使用多个字节表达一个符号。比如，简体中文常见的编码方式是GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示 256 x 256 = 65536 个符号。

中文编码的问题需要专文讨论，这篇笔记不涉及。这里只指出，虽然都是用多个字节表示一个符号，但是GB类的汉字编码与后文的 Unicode 和 UTF-8 是毫无关系的。

Unicode

正如上一节所说，世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。为什么电子邮件常常出现乱码？就是因为发信人和收信人使用的编码方式不一样。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是 Unicode，就像它的名字都表示的，这是一种所有符号的编码。

Unicode 当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字严。具体的符号对应表，可以查询unicode.org，或者专门的汉字对应表。

Unicode 的问题

需要注意的是，Unicode 只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字严的 Unicode 是十六进制数4E25，转换成二进制数足足有15位（100111000100101），也就是说，这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

这里就有两个严重的问题，第一个问题是，如何才能区别 Unicode 和 ASCII？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果 Unicode 统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是0，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。

它们造成的结果是：1）出现了 Unicode 的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示 Unicode。2）Unicode 在很长一段时间内无法推广，直到互联网的出现。

UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8 就是在互联网上使用最广的一种 Unicode 的实现方式。其他实现方式还包括 UTF-16（字符用两个字节或四个字节表示）和 UTF-32（字符用四个字节表示），不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8 是 Unicode 的实现方式之一。

UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8 的编码规则很简单，只有二条：

1）对于单字节的符号，字节的第一位设为0，后面7位为这个符号的 Unicode 码。因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。

2）对于n字节的符号（n > 1），第一个字节的前n位都设为1，第n + 1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的 Unicode 码。

下表总结了编码规则，字母x表示可用编码的位。

Unicode符号范围(十六进制)	UTF-8编码方式（二进制）
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

跟据上表，解读 UTF-8 编码非常简单。如果一个字节的第一位是0，则这个字节单独就是一个字符；如果第一位是1，则连续有多少个1，就表示当前字符占用多少个字节。

下面，还是以汉字严为例，演示如何实现 UTF-8 编码。

严的 Unicode 是4E25（100111000100101），根据上表，可以发现4E25处在第三行的范围内（0000 0800 - 0000 FFFF），因此严的 UTF-8 编码需要三个字节，即格式是1110xxxx 10xxxxxx 10xxxxxx。然后，从严的最后一个二进制位开始，依次从后向前填入格式中的x，多出的位补0。这样就得到了，严的 UTF-8 编码是11100100 10111000 10100101，转换成十六进制就是E4B8A5。

Unicode 与 UTF-8 之间的转换

通过上一节的例子，可以看到严的 Unicode码 是4E25，UTF-8 编码是E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

Windows平台，有一个最简单的转化方法，就是使用内置的记事本小程序notepad.exe。打开文件后，点击文件菜单中的另存为命令，会跳出一个对话框，在最底部有一个编码的下拉条。

里面有四个选项：ANSI，Unicode，Unicode big endian和UTF-8。

1）ANSI是默认的编码方式。对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对 Windows 简体中文版，如果是繁体中文版会采用 Big5 码）。

2）Unicode编码这里指的是notepad.exe使用的 UCS-2 编码方式，即直接用两个字节存入字符的 Unicode 码，这个选项用的 little endian 格式。

3）Unicode big endian编码与上一个选项相对应。我在下一节会解释 little endian 和 big endian 的涵义。

4）UTF-8编码，也就是上一节谈到的编码方法。

选择完"编码方式"后，点击"保存"按钮，文件的编码方式就立刻转换好了。

Little endian 和 Big endian

上一节已经提到，UCS-2 格式可以存储 Unicode 码（码点不超过0xFFFF）。以汉字严为例，Unicode 码是4E25，需要用两个字节存储，一个字节是4E，另一个字节是25。存储的时候，4E在前，25在后，这就是 Big endian 方式；25在前，4E在后，这是 Little endian 方式。

这两个古怪的名称来自英国作家斯威夫特的《格列佛游记》。在该书中，小人国里爆发了内战，战争起因是人们争论，吃鸡蛋时究竟是从大头(Big-endian)敲开还是从小头(Little-endian)敲开。为了这件事情，前后爆发了六次战争，一个皇帝送了命，另一个皇帝丢了王位。

第一个字节在前，就是"大头方式"（Big endian），第二个字节在前就是"小头方式"（Little endian）。

那么很自然的，就会出现一个问题：计算机怎么知道某一个文件到底采用哪一种方式编码？

Unicode 规范定义，每一个文件的最前面分别加入一个表示编码顺序的字符，这个字符的名字叫做"零宽度非换行空格"（zero width no-break space），用FEFF表示。这正好是两个字节，而且FF比FE大1。

如果一个文本文件的头两个字节是FE FF，就表示该文件采用大头方式；如果头两个字节是FF FE，就表示该文件采用小头方式。

实例

下面，举一个实例。

打开"记事本"程序notepad.exe，新建一个文本文件，内容就是一个严字，依次采用ANSI，Unicode，Unicode big endian和UTF-8编码方式保存。

然后，用文本编辑软件UltraEdit 中的"十六进制功能"，观察该文件的内部编码方式。

1）ANSI：文件的编码就是两个字节D1 CF，这正是严的 GB2312 编码，这也暗示 GB2312 是采用大头方式存储的。

2）Unicode：编码是四个字节FF FE 25 4E，其中FF FE表明是小头方式存储，真正的编码是4E25。

3）Unicode big endian：编码是四个字节FE FF 4E 25，其中FE FF表明是大头方式存储。

4）UTF-8：编码是六个字节EF BB BF E4 B8 A5，前三个字节EF BB BF表示这是UTF-8编码，后三个E4B8A5就是严的具体编码，它的存储顺序与编码顺序是一致的。

String详解

概述

String 类代表字符串。Java 程序中的所有字符串字面值（如 "abc" ）都作为此类的实例实现。

字符串是常量；它们的值在创建之后不能更改。字符串缓冲区支持可变的字符串。因为 String 对象是不可变的，所以可以共享。

java.lang.String:

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
```

源码分析

1. 成员变量

   /** String的属性值 */

   private final char value[];

   /** The offset is the first index of the storage that is used. */

   /**数组被使用的开始位置/

private final int offset;
/** The count is the number of characters in the String. */
/String中元素的个数/
private final int count;
/** Cache the hash code for the string */
/String类型的hash值/
private int hash; // Default to 0
/** use serialVersionUID from JDK 1.0.2 for interoperability */
private static final long serialVersionUID = -6849794470754667710L;
private static final ObjectStreamField[] serialPersistentFields =
new ObjectStreamField[0];

从源码看出String底层使用一个字符数组来维护的。

成员变量可以知道String类的值是final类型的，不能被改变的，所以只要一个值改变就会生成一个新的String类型对象，存储String数据也不一定从数组的第0个元素开始的，而是从offset所指的元素开始。

2. String的构造方法
String()
初始化一个新创建的 String 对象，使其表示一个空字符序列。
String(byte[] bytes)
通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String
String(byte[] bytes, Charset charset)
通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。
String(byte[] bytes, int offset, int length)
通过使用平台的默认字符集解码指定的 byte 子数组，构造一个新的 String。
String(byte[] bytes, int offset, int length, Charset charset)
通过使用指定的 charset 解码指定的 byte 子数组，构造一个新的 String。
String(byte[] bytes, int offset, int length, String charsetName)
通过使用指定的字符集解码指定的 byte 子数组，构造一个新的 String。
String(byte[] bytes, String charsetName)
通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。
String(char[] value)分配一个新的 String，使其表示字符数组参数中当前包含的字符序列。
String(char[] value, int offset, int count)
分配一个新的 String，它包含取自字符数组参数一个子数组的字符。
String(int[] codePoints, int offset, int count)
分配一个新的 String，它包含 Unicode 代码点数组参数一个子数组的字符。
String(String original)
初始化一个新创建的 String 对象，使其表示一个与参数相同的字符序列；换句话说，新创建的字符串是该参数字符串的副本。
String(StringBuffer buffer)
分配一个新的字符串，它包含字符串缓冲区参数中当前包含的字符序列。
String(StringBuilder builder)
分配一个新的字符串，它包含字符串生成器参数中当前包含的字符序列。

3. string重写了equals方法
public boolean equals(Object anObject) {
if (this == anObject) {
return true;
}
if (anObject instanceof String) {
String anotherString = (String)anObject;
int n = value.length;
if (n == anotherString.value.length) {
char v1[] = value;

```
char v2[] = anotherString.value;
int i = 0;
while (n-- != 0) {
if (v1[i] != v2[i])
return false;
i++;
}
return true;
}
}
return false;
}
```

String不可变的好处

1. 便于实现String常量池
   只有当字符串是不可变的，字符串池才有可能实现。字符串池的实现可以在运行时节约很多heap空间，因为不同的字符串变量都指向池中的同一个字符串。但如果字符串是可变的，那么String interning将不能实现(译者注：String interning是指对不同的字符串仅仅只保存一个，即不会保存多个相同的字符串。)，因为这样的话，如果变量改变了它的值，那么其它指向这个值的变量的值也会一起改变。
2. 避免网络安全问题
   如果字符串是可变的，那么会引起很严重的安全问题。譬如，数据库的用户名、密码都是以字符串的形式传入来获得数据库的连接，或者在socket编程中，主机名和端口都是以字符串的形式传入。因为字符串是不可变的，所以它的值是不可改变的，否则黑客们可以钻到空子，改变字符串指向的对象的值，造成安全漏洞。
3. 使多线程安全
   因为字符串是不可变的，所以是多线程安全的，同一个字符串实例可以被多个线程共享。这样便不用因为线程安全问题而使用同步。字符串自己便是线程安全的。
4. 避免本地安全性问题
   类加载器要用到字符串，不可变性提供了安全性，以便正确的类被加载。譬如你想加载java.sql.Connection类，而这个值被改成了myhacked.Connection，那么会对你的数据库造成不可知的破坏。
5. 加快字符串处理速度
   因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。

字符串常量池

1、**字符串常量池概述**

1）**常量池表**（Constant_Pool table）

Class文件中存储所有常量（包括字符串）的table。

这是Class文件中的内容，还不是运行时的内容，不要理解它是个池子，其实就是Class文件中的字节码指令。

2）**运行时常量池**（Runtime Constant Pool）

JVM内存中方法区的一部分，这是运行时的内容

这部分内容（绝大部分）是随着JVM运行时候，从常量池转化而来，每个Class对应一个运行时常量池

上一句中说绝大部分是因为：除了 Class中常量池内容，还可能包括动态生成并加入这里的内容

3）**字符串常量池**（String Pool）

这部分也在方法区中，但与Runtime Constant Pool不是一个概念，String Pool是JVM实例全局共

享的，全局只有一个

JVM规范要求进入这里的String实例叫“被驻留的interned string”，各个JVM可以有不同的实现，HotSpot是设置了一个哈希表StringTable来引用堆中的字符串实例，被引用就是被驻留。

2、**享元模式**

其实字符串常量池这个问题涉及到一个设计模式，叫“享元模式”，顾名思义 - - - > 共享元素模式，也就是说：一个系统中如果有多处用到了相同的一个元素，那么我们应该只存储一份此元素，而让所有地方都引用这一个元素

Java中String部分就是根据享元模式设计的，而那个存储元素的地方就叫做“字符串常量池 - String Pool”

3、**详细分析**

举例：

int x = 10;

String y = "hello";

1)首先，10和"hello"会在经过javac（或者其他编译器）编译过后变为Class文件中constant_pool table的内容

2)当我们的程序运行时，也就是说JVM运行时，每个Classconstant_pool table中的内容会被加载到JVM内存中的方法区中各自Class的Runtime Constant Pool。

3)一个没有被String Pool包含的Runtime Constant Pool中的字符串（这里是"hello"）会被加入到String Pool中（HosSpot使用hashtable引用方式），步骤如下：

一是：在Java Heap中根据"hello"字面量create一个字符串对象(直接使用双引号声明出来的String对象会直接存储在常量池中)

二是：将字面量"hello"与字符串对象的引用在hashtable中关联起来，键 - 值 形式是："hello" = 对象的引用地址。

另外来说，当一个新的字符串出现在Runtime Constant Pool中时怎么判断需不需要在Java Heap中创建新对象呢?

策略是这样：会先去根据equals来比较Runtime Constant Pool中的这个字符串是否和String Pool中某一个是相等的（也就是找是否已经存在），如果有那么就不创建，直接使用其引用；反之，如上3

如此，就实现了享元模式，提高的内存利用效率。

举例：

使用String s = new String("hello");会创建几个对象

会创建2个对象

首先，出现了字面量"hello"，那么去String Pool中查找是否有相同字符串存在，因为程序就这一行代码所以肯定没有，那么就在Java Heap中用字面量"hello"首先创建1个String对象。

接着，new String("hello")，关键字new又在Java Heap中创建了1个对象，然后调用接收String参数的构造器进行了初始化。最终s的引用是这个String对象.

String.intern()

public native String intern();

这个方法是一个 native 的方法，但注释写的非常明了。“如果常量池中存在当前字符串, 就会直接返回当前字符串. 如果常量池中没有此字符串, 会将此字符串放入常量池中后, 再返回”。

实例

一个String的题目：

```
String s1 = "ABCD";
String s2 = "A" + "B" + "C" + "D";
String s3 = "AB" + "CD";
String s4 = new String("ABCD");
String s = "AB";
String s5 = s + "CD";
String s6 = getSt() + "CD";
final String ss = "AB";
```

```
    String s7 = ss + "CD";
    public  static String getSt()
    return "AB";
}
```

其中s1,s2,s3,s7相等，其他都不等

分析：

s1,s2,s3在编译的时候都被优化为“ABCD"了，所以相等。s5,s6中有变量，都是运行时使用new StringBuilder连接（如果有变量，那么会调用StringBuilder，最后调用Sb的tostring），s7是final修饰的常量，也会在编译器被优化

Integer详解

1.介绍

关于Integer和int在面试的时候出现的频率很高。而我们所熟知的是Integer是int 的包装类型，int的初始值为0，Integer的初始值为null，这是基本都知道的。至于Integer的自动装箱和拆箱，以及Integer的缓存等小细节需要深入思考

2.自动装箱与拆箱

1. 自动装箱

```
public class CaseTest {

    public static void main(String[] args) {
        Integer i = 100;
    }
}
```

2. 自动拆箱

```
public class CaseTest {

    public static void main(String[] args) {
        Integer i = 100;
        int j = i;
    }
}
```

3.Integer的缓存

Integer定义了一个静态内部类IntegerCache作为缓存

```
private static class IntegerCache {
    //常量最小值-128
        static final int low = -128;
    //常量最大值
        static final int high;
    //Integer缓存数组
        static final Integer cache[];

        static {
            //初始化h变量为127
            int h = 127;
            String integerCacheHighPropValue =
                sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
            if (integerCacheHighPropValue != null) {
                int i = parseInt(integerCacheHighPropValue);
        //说明i最小可以取到127，规定了i的下限，所以i>=127
                i = Math.max(i, 127);
        //由于i>=127并且Integer.MAX_VALUE - (-low)>128,因此可以得到h>=127
                h = Math.min(i, Integer.MAX_VALUE - (-low));
            }
            high = h;
        //初始化Integer缓存数组
            cache = new Integer[(high - low) + 1];
        //数组中的初始值是-128
            int j = low;
         //循环为数组赋值，数组0位是-128，最大是127（临界值）
             for(int k = 0; k < cache.length; k++)
                 cache[k] = new Integer(j++);
         }

         private IntegerCache() {}
}
/**
 * 返回一个表示指定的 int 值的 Integer 实例。
 * 如果不需要新的 Integer 实例，则通常应优先使用该方法，而不是构造方法 Integer(int)
 * 因为该方法有可能通过缓存经常请求的值而显著提高空间和时间性能
 */
public static Integer valueOf(int i) {
    assert IntegerCache.high >= 127;
    //当-128<=i<=127时，直接从缓存数组中取出值
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    //否则返回新的Integer对象
    return new Integer(i);
}
```

例子：

```
public static void main(String[] args) {
        //标准的装箱过程
        Integer i =new Integer(100);
        Integer i1 =100;
        Integer i2 =100;
        Integer i3 =128;
        Integer i4 =128;
        System.out.println("i==i1 :"+(i==i1));
        System.out.println("i1==i2 :"+(i1==i2));
        System.out.println("i3==i4 :"+(i3==i4));
}
```

说明：

（1） i == i1 ： 由于i本身是new Integer的方式在堆中单独开辟空间，i1是 -128<=i1<=127，所以i1是

从缓存中取出数据的。而缓存的地址和new Integer单独开辟空间对应的地址不同，返回false。
（2） i1 == i2 ： i1是 -128<=i1<=127，所以i1是从缓存中取出数据的。i2是-128<=i2<=127，所以i2也是从缓存中取出数据的。所以i1和i2对应的是同一个缓存地址，返回true。
（3）i3 == i4 ： i3>127并且i4>127,所以i3和i4分别是new Integer这种单独开辟空间，地址不同，返回false。

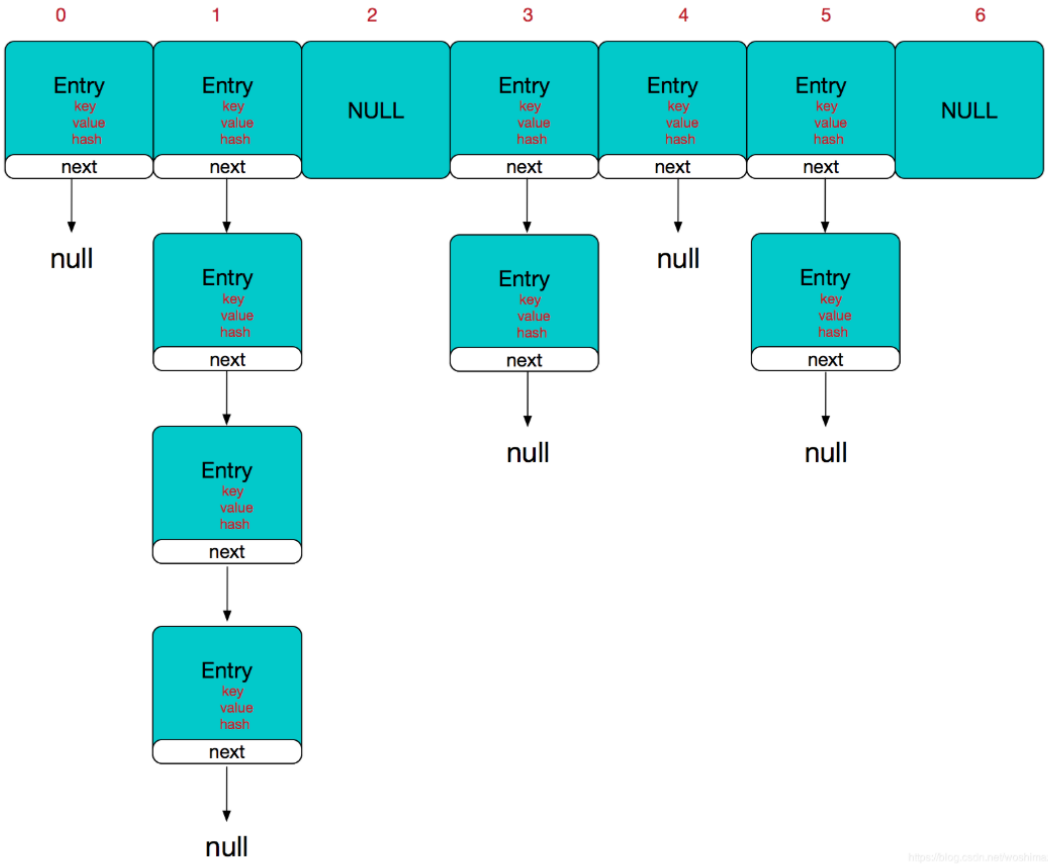
java内部类

链接：https://www.cnblogs.com/dolphin0520/p/3811445.html
##关于hashcode重写
正确书写hashCode的办法：
【原则】按照equals( )中比较两个对象是否一致的条件用到的属性来重写hashCode()。
{1}常用的办法就是利用涉及到的的属性进行线性组合。
{2}线性组合过程中涉及到的组合系数自定义即可。
注意，拼接之后的数值不能超过整形的表达范围。
{3}公式：属性1的int形式+ C1*属性2的int形式*+ *C2*属性3的int形式+ ...
【技巧】当属性是引用类型的时候，如果已经重写过hashCode()，那么这个引用属性的int形式就是直接调用属性已有的hashCode值。
最典型的就是这个属性是字符串类型的，String类型已经重写了hashCode()方法，所以直接拿来使用即可。
如：大表弟的两个字符串拼接(deviceId.hashCode() >> 2) + (courseScheduleId.hashCode() >> 2)

HashMap

结构

结构上由数组与链表构成，JDK 8后由数组+链表+红黑树构成。JDK8采用尾插法，之前采用头插法。结构图如下：



源码

1.重要字段

```
 /**
     * The number of key-value mappings contained in this map.
     * 即实际存储的键值对个数
     */
transient int size;
/**
     * The number of times this HashMap has been structurally modified
     * Structural modifications are those that change the number of mappings in
     * the HashMap or otherwise modify its internal structure (e.g.,
     * rehash).  This field is used to make iterators on Collection-views of
     * the HashMap fail-fast.  (See ConcurrentModificationException).
     *HashMap被改变的次数，由于HashMap非线程安全，在对HashMap进行迭代时，
     *如果期间其他线程的参与导致HashMap的结构发生变化了（比如put，remove等操作），
     *需要抛出异常ConcurrentModificationException
     */
transient int modCount;
/**
     * The next size value at which to resize (capacity * load factor).
     * 当size到达此值时，将触发扩容
     * @serial
     */
int threshold;
  /**
      * The load factor for the hash table.
      *
      * 加载因子
      */
final float loadFactor;
/**
      * The table, initialized on first use, and resized as
      * necessary. When allocated, length is always a power of two.
      * (We also tolerate length zero in some operations to allow
      * bootstrapping mechanics that are currently not needed.)
      * 内部数组，数组的长度就是initiaCapacity.且initiaCapacity会取最接近的那个2的幂的数
      * 通过tableSizeFor(int cap)方法。比如new HashMap<>(10);其实初始的table的长度是16
      */
transient Node<K,V>[] table;
```

2.重要内部类

```
static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
```

```
        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }

        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }
```

3.重要构造方法

```
public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }
```

注意：构造方法并没有初始化table，其实table初始化在put操作中。

4.重要方法

- get(Object key):其中通过k的hashCode高16位与hashCode低16位与或得到hash。而数组下标是通过数组长度减去1后和hash进行&运算得到。即getNode里的（n-1）& hash

```
static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
    }

public V get(Object key) {
        Node<K,V> e;
```

```
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }

final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            if ((e = first.next) != null) {
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }
```

- put方法

```
public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
```

```
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }


final Node<K,V>[] resize() {
        Node<K,V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        int oldThr = threshold;
        int newCap, newThr = 0;
        if (oldCap > 0) {
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
                newThr = oldThr << 1; // double threshold
        }
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else {               // zero initial threshold signifies using defaults
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
```

```
                        Node<K,V> next;
                        do {
                            next = e.next;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

扩容机制

线程不安全分析

List

泛型

反射

JDK 8

新特性概述

Lambda

Stream

新日期API

概述

JDK8中增加了一套全新的日期时间API，位于 java.time 包中，下面是一些关键类.不可变的线程安全类

- Instant：代表的是时间戳
- LocalDate：不包含具体时间的日期，格式为 2020-01-03
- LocalTime：不含日期的时间，格式为 09:53:56.749
- LocalDateTime：包含了日期及时间，格式为 2020-01-03T09:53:56.749

代码块示例

```
1         Instant instant = Instant.now();
2         System.out.println("当前时间戳是：" + instant);//当前时间戳是：2018-09-
06T10:14:29.460Z
3         Date date = Date.from(instant);
4         System.out.println("当前时间戳是：" + date);//当前时间戳是：Thu Sep 06
18:14:29 CST 2018
5         instant = date.toInstant();
```

```
 1         LocalDate nowDate = LocalDate.now();
 2         System.out.println("今天的日期：" + nowDate);//今天的日期：2018-09-06
 3         int year = nowDate.getYear();//年：一般用这个方法获取年
 4         System.out.println("year：" + year);//year：2018
 5         int month = nowDate.getMonthValue();//月：一般用这个方法获取月
 6         System.out.println("month：" + month);//month：9
 7         int day = nowDate.getDayOfMonth();//日：当月的第几天，一般用这个方法获取日
 8         System.out.println("day：" + day);//day：6
 9
10         int dayOfYear = nowDate.getDayOfYear();//日：当年的第几天
11         System.out.println("dayOfYear：" + dayOfYear);//dayOfYear：249
12
13         //星期
14         System.out.println(nowDate.getDayOfWeek());//THURSDAY
15         System.out.println(nowDate.getDayOfWeek().getValue());//4
16         //月份
17         System.out.println(nowDate.getMonth());//SEPTEMBER
18         System.out.println(nowDate.getMonth().getValue());//9

19         System.out.println(LocalDate.of(1991, 11, 11));//直接传入对应的年月日
20         System.out.println(LocalDate.of(1991, Month.NOVEMBER, 11));//相对上面只
是把月换成了枚举
21         LocalDate birDay = LocalDate.of(1991, 11, 11);
22         System.out.println(LocalDate.ofYearDay(1991,
birDay.getDayOfYear()));//第一个参数为年，第二                个参数为当年的第多少天
23         System.out.println(LocalDate.ofEpochDay(birDay.toEpochDay()));//参数为
距离1970-01-01的天数

24         System.out.println(LocalDate.parse("1991-11-11"));
25
System.out.println(LocalDate.parse("19911111",DateTimeFormatter.ofPattern("yyyyM
Mdd")));
```

```
        LocalTime nowTime = LocalTime.now();
        System.out.println("今天的时间：" + nowTime);//今天的时间：15:33:56.749
        int hour = nowTime.getHour();//时
        System.out.println("hour：" + hour);//hour：15
        int minute = nowTime.getMinute();//分
        System.out.println("minute：" + minute);//minute：33
        int second = nowTime.getSecond();//秒
        System.out.println("second：" + second);//second：56
        int nano = nowTime.getNano();//纳秒
        System.out.println("nano：" + nano);//nano：749000000
```

```
        LocalDateTime nowDateTime = LocalDateTime.now();
        System.out.println("今天是：" + nowDateTime);//今天是：2018-09-06T15:33:56.750
        System.out.println(nowDateTime.getYear());//年
        System.out.println(nowDateTime.getMonthValue());//月
        System.out.println(nowDateTime.getDayOfMonth());//日
        System.out.println(nowDateTime.getHour());//时
        System.out.println(nowDateTime.getMinute());//分
        System.out.println(nowDateTime.getSecond());//秒
        System.out.println(nowDateTime.getNano());//纳秒
        //日：当年的第几天
        System.out.println("dayOfYear：" + nowDateTime.getDayOfYear());//dayOfYear：249
        //星期
        System.out.println(nowDateTime.getDayOfWeek());//THURSDAY
        System.out.println(nowDateTime.getDayOfWeek().getValue());//4
        //月份
        System.out.println(nowDateTime.getMonth());//SEPTEMBER
        System.out.println(nowDateTime.getMonth().getValue());//9
```

日期时间比较

在JDK8中，LocalDate类中使用isBefore()、isAfter()、equals()方法来比较两个日期，可直接进行比较

```
        LocalDate myDate = LocalDate.of(2018, 9, 5);
        LocalDate nowDate = LocalDate.now();
        System.out.println("今天是2018-09-06吗？  " + nowDate.equals(myDate));
        System.out.println(myDate + "是否在"+nowDate+"之前?"
        myDate.isBefore(nowDate));//2018-09-05是否在2018-09-06之前？  true
        System.out.println(myDate + "是否在"+nowDate+"之后?"
        myDate.isAfter(nowDate));//2018-09-05是否在2018-09-06之后？  false
```

日期时间格式化

在JDK8之前，时间日期的格式化非常麻烦，经常使用SimpleDateFormat来进行格式化，但是SimpleDateFormat并不是线程安全的，要结合ThreadLocal使用。在JDK8中，引入了一个全新的线程安全的日期与时间格式器DateTimeFormatter

```
        LocalDateTime ldt = LocalDateTime.now();
        System.out.println(ldt);//2018-09-06T18:22:47.366
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss");
        String ldtStr = ldt.format(dtf);
        System.out.println(ldtStr);//2018-09-06 18:22:47
        String ldtStr1 = dtf.format(ldt);
        System.out.println(ldtStr1);//2018-09-06 18:22:47
```

java 并发编程

java IO

IDEA

Linux

计算机网络基础

ip

ipv4

ipv6

TCP

UDP

HTTP

HTTPS

设计模式

简述

建造者模式

命令模式

单例模式

装饰者模式

观察者模式

代理模式

策略模式

工厂方法模式

简单工厂模式

抽象工厂模式

数据结构

常见算法

分布式及WEB相关

幂等性

概念

HTTP/1.1中对幂等性的定义是：一次和多次请求某一个资源**对于资源本身**应该具有同样的结果（网络超时等问题除外）。也就是说，**其任意多次执行对资源本身所产生的影响均与一次执行的影响相同。**其实幂等性是系统服务对外一种承诺（而不是实现），承诺只要调用接口成功，外部多次调用对系统的影响是一致的。声明为幂等的服务会认为外部调用失败是常态，并且失败之后必然会有重试。

为何需要幂等性

原因很简单，在系统调用没有达到期望的结果后，会重试。那重试就会面临问题，重试之后不能给业务逻辑带来影响，例如创建订单，第一次调用超时了，但是调用的系统不知道超时了是成功了还是失败了，然后他就重试，但是实际上第一次调用订单创建是成功了的，这时候重试了，显然不能再创建订单了。

什么情况使用幂等性

以SQL为例，有下面三种场景，只有第三种场景需要开发人员使用其他策略保证幂等性：

1. `SELECT col1 FROM tab1 WHER col2=2`，无论执行多少次都不会改变状态，是天然的幂等。
2. `UPDATE tab1 SET col1=1 WHERE col2=2`，无论执行**成功**多少次**状态**都是一致的，因此也是幂等操作。
3. `UPDATE tab1 SET col1=col1+1 WHERE col2=2`，每次执行的结果都会发生变化，这种不是幂等的。

如何保证幂等性

幂等需要通过**唯一的业务单号**来保证。也就是说相同的业务单号，认为是同一笔业务。使用这个唯一的业务单号来确保，后面多次的相同的业务单号的处理逻辑和执行效果是一致的。 下面以支付为例，在不考虑并发的情况下，实现幂等很简单：①先查询一下订单是否已经支付过，②如果已经支付过，则返回支付成功；如果没有支付，进行支付流程，修改订单状态为'已支付'。

防重复提交

上述的保证幂等方案是分成两步的，第②步依赖第①步的查询结果，无法保证原子性的。在高并发下就会出现下面的情况：第二次请求在第一次请求第②步订单状态还没有修改为'已支付状态'的情况下到来。既然得出了这个结论，余下的问题也就变得简单：把查询和变更状态操作加锁，将并行操作改为串行操作。

乐观锁

如果只是更新**已有**的数据，没有必要对业务进行加锁，设计表结构时使用乐观锁，一般通过version来做乐观锁，这样既能保证执行效率，又能保证幂等。例如：`UPDATE tab1 SET col1=1,version=version+1 WHERE version=#version#` 不过，乐观锁存在失效的情况，就是常说的ABA问题，不过如果version版本一直是自增的就不会出现ABA的情况。（从网上找了一张图片很能说明乐观锁，引用过来，出自Mybatis对乐观锁的支持）

防重表

使用订单号orderNo做为去重表的唯一索引，每次请求都根据订单号向去重表中插入一条数据。第一次请求查询订单支付状态，当然订单没有支付，进行支付操作，无论成功与否，执行完后更新订单状态为成功或失败，删除去重表中的数据。后续的订单因为表中唯一索引而插入失败，则返回操作失败，直到第一次的请求完成（成功或失败）。**可以看出防重表作用是加锁的功能。**

分布式锁

这里使用的防重表可以使用分布式锁代替，比如Redis。订单发起支付请求，支付系统会去Redis缓存中查询是否存在该订单号的Key，如果不存在，则向Redis增加Key为订单号。查询订单支付已经支付，如果没有则进行支付，支付完成后删除该订单号的Key。通过Redis做到了分布式锁，只有这次订单订单支付请求完成，下次请求才能进来。相比去重表，将放并发做到了缓存中，较为高效。思路相同，**同一时间只能完成一次支付请求。**

token令牌

这种方式分成两个阶段：申请token阶段和支付阶段。 第一阶段，在进入到提交订单页面之前，需要订单系统根据用户信息向支付系统发起一次申请token的请求，支付系统将token保存到Redis缓存中，为第二阶段支付使用。 第二阶段，订单系统拿着申请到的token发起支付请求，支付系统会检查Redis中是否存在该token，如果存在，表示第一次发起支付请求，删除缓存中token后开始支付逻辑处理；如果缓存中不存在，表示非法请求。 实际上这里的token是一个信物，支付系统根据token确认，你是你妈的孩子。不足是需要系统间交互两次，流程较上述方法复杂。

支付缓冲区

把订单的支付请求都快速地接下来，一个快速接单的缓冲管道。后续使用异步任务处理管道中的数据，过滤掉重复的待支付订单。优点是同步转异步，高吞吐。不足是不能及时地返回支付结果，需要后续监听支付结果的异步返回。

分布式一致性

分类

强一致：当更新操作完成之后，任何多个后续进程或者线程的访问都会返回最新的更新过的值。这种是对用户最友好的，就是用户上一次写什么，下一次就保证能读到什么。根据 CAP 理论，这种实现需要牺牲可用性。

弱一致：系统并不保证续进程或者线程的访问都会返回最新的更新过的值。系统在数据写入成功之后，不承诺立即可以读到最新写入的值，也不会具体的承诺多久之后可以读到。

最终一致：弱一致性的特定形式。系统保证在没有后续更新的前提下，系统最终返回上一次更新操作的值。

CAP理论

CAP是Consistency、Availablity和Partition-tolerance的缩写。分别是指：

1.一致性（Consistency）：每次读操作都能保证返回的是最新数据，在分布式系统中，如果能针对一个数据项的更新执行成功后，所有的用户都可以读到其最新的值，这样的系统就被认为具有严格的一致性。

2.可用性（Availablity）：任何一个没有发生故障的节点，会在合理的时间内返回一个正常的结果，也就是对于用户的每一个请求总是能够在有限的时间内返回结果；

3.分区容忍性（Partition-torlerance）：当节点间出现网络分区（不同节点处于不同的子网络，子网络之间应该是联通的，但是子网络之间无法联通了，也就是被切分成了孤立的集群网络），照样可以提供满足一致性和可用性的服务，除非整个网络环境都发生了故障。

CAP理论指出：CAP三者只能取其二，不可兼得。

首先，如果我们要使网络分区不存在，就必须将系统部署在单个节点上，因为网络总是会出现故障，分区总是存在的，**所以当部署在单节点上，可以同时保证CP**，但是这时候，就没什么意义了，这都不是分布式了，同时单点故障可能会发生，就不会保证A可用性。

所以我们必须明确一点：对于分布式系统而言，分区容错性是必须要满足的，因为分区的出现时必然，也是必须要解决的问题。所以，P必须要保证，那么我们就要在C和A之间做权衡。

- 有两个或以上节点时，当网络分区发生时，集群中两个节点不能相互通信。此时如果保证数据的一致性C，那么必然会有一**个节点被标记为不可用的状态，违反了可用性A的要求，只能保证CP。**
- 反之，如果保证可用性A，**即两个节点可以继续各自处理请求，那么由于网络不通不能同步数据，必然又会导致数据的不一致**，只能保证AP。

BASE理论

在上边，我们谈到，因为P总是存在的，放弃不了。另外，可用性、一致性也是我们一般系统必须要满足的，如何在可用性和一致性进行权衡，所以就出现了各种一致性的理论与算法

1.基本可用（Basically Available）：基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性——但请注意，这绝不等价于系统不可用。如

- 响应时间上的损失：正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1～2秒。
- 功能上的损失：正常情况下，在一个电子商务网站上进行购物，消费者几乎能够顺利地完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面。

2.软状态（ Soft State）:软状态也称为弱状态，和硬状态相对，是指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时。

3.最终一致性（ Eventual Consistency）:最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

一致性变种

因果一致性：如果进程A在更新完某个数据项后通知了进程B，那么进程B之后对该数据项的访问都应该能够获取到进程A更新后的最新值，并且如果进程B要对该数据项进行更新操作的话，务必基于进程A更新后的最新值，即不能发生丢失更新情况。与此同时，与进程A无因果关系的进程C的数据访问则没有这样的限制。

读己之所写：进程A更新一个数据项之后，它自己总是能够访问到更新过的最新值，而不会看到旧值。也就是说，对于单个数据获取者来说，其读取到的数据，一定不会比自己上次写入的值旧。因此，读己之所写也可以看作是一种特殊的因果一致性。

会话一致性：将对系统数据的访问过程框定在了一个会话当中：系统能保证在同一个有效的会话中实现“读己之所写”的一致性，也就是说，执行更能操作之后，客户端能够在同一个会话中始终读取到该数据项的最新值。

分布式一致性机制整理

JVM

Spring框架

Spring MVC

Spring Boot

Spring Cloud

Shiro

CAS 单点登录

Mysql

事务

MVCC多并发版本控制

概念

MVCC（multi version concurrency control）即多版本并发控制，其作用就是在让特定隔离级别的事务在并发时，保证在事务中能实现一致性读，虽然锁可以控制并发操作，但是锁的系统开销比较大，而MVCC可以在大多数情况下替代行级锁，同时可以降低系统开销。

分布式事务

索引详解

定义

索引(Index)是帮助MySQL高效获取数据的数据结构.可以得出索引的本质就是数据结构。在数据之外,数据库还维护着满足特定查找算法的数据结构,这些数据结构以某种方式引用(指向)数据,这样就可以在这些数据结构的基础上实现高级查找算法,这种数据结构就是索引。

一般来说索引本身很大,不适合全部存储在内存中,因此索引往往以索引文件的形式存储在磁盘上。

我们平常所说的索引,如果没有特别指明,都是指B树(多路搜索树,并不一定是二叉的)结构组织的索引,其中聚集索引、次要索引、覆盖索引、复合索引、前缀索引,唯一索引默认都是使用B+树索引,统称索引.当然,除了B+树这种类型的索引之外,还有哈稀索引(hash index)等

优点

类似大学图书馆建书目录索引,提高数据检索的效率,降低数据库的IO成本。通过索引列对数据进行排序,降低数据排序成本,降低了CPU的消耗同时也可以加速表和表之间的连接。

缺点

实际上索引也可以看成一张表,该表保存了主键与索引字段,并指向实体表的记录,所以索引也是要占内存空间的。
虽然索引大大提高了查询速度,但同时也会降低更新表的速度,比如对表进行insert,update和delete的操作，因为更新表时,MySQL不仅要保存数据,还要保存索引文件的每次更新，比如因为更新所带来的键值变化后的索引信息。
索引只是高效的一个因素,如果你的MySQL有大数据量的表,就需要花时间研究建立最优秀的索引,或优化查询方法。

分类

聚集索引（聚簇索引）

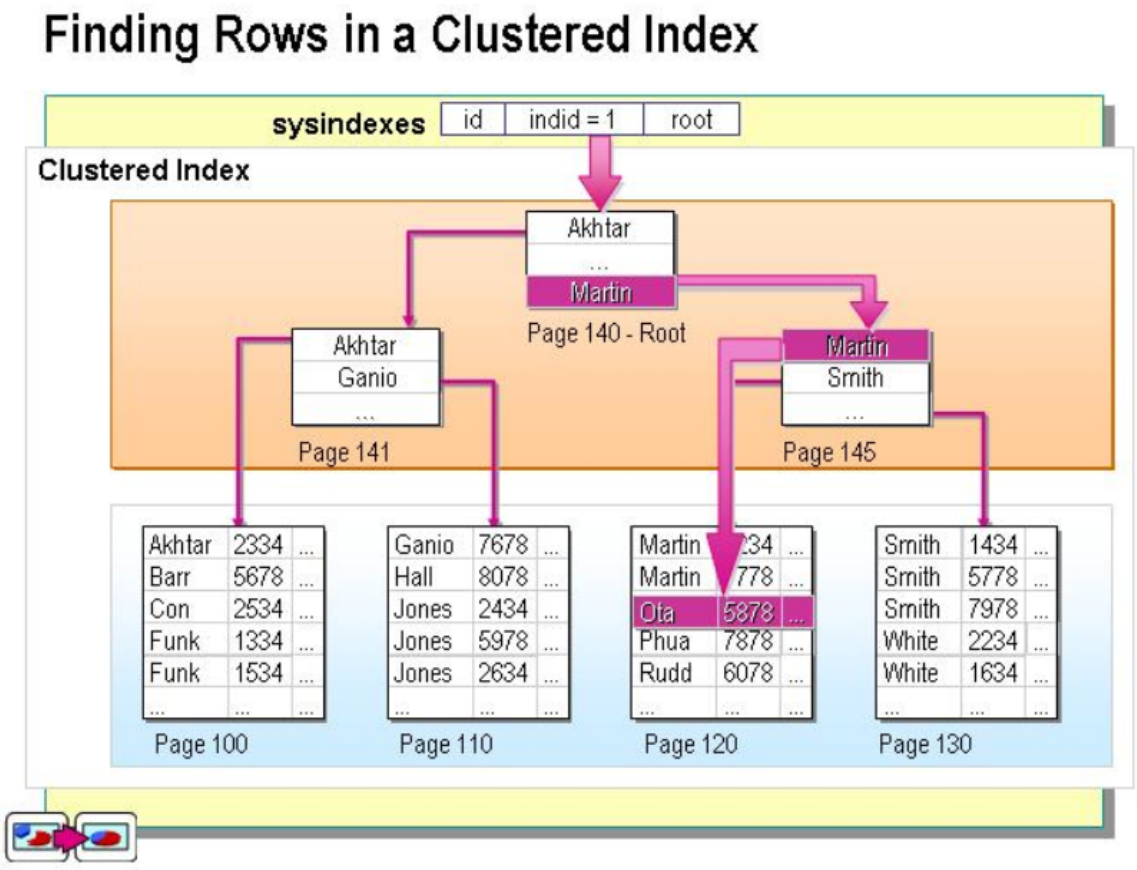
定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。

打个比方，一个表就像是我们以前用的新华字典，聚集索引就像是拼音目录，而每个字存放的页码就是我们的数据物理地址，我们如果要查询一个“哇”字，我们只需要查询“哇”字对应在新华字典拼音目录对应的页码，就可以查询到对应的“哇”字所在的位置，而拼音目录对应的A-Z的字顺序，和新华字典实际存储的字的顺序A-Z也是一样的，如果我们中文新出了一个字，拼音开头第一个是B，那么他插入的时候也要按照拼音目录顺序插入到A字的后面，现在用一个简单的示意图来大概说明一下在数据库中的样子：

地址	id	username	score
0x01	1	小明	90
0x02	2	小红	80
0x03	3	小华	92
..	..	..	..
0xff	256	小英	70

注：第一列的地址表示该行数据在磁盘中的物理地址，后面三列才是我们SQL里面用的表里的列，其中id是主键，建立了聚集索引。

结合上面的表格就可以理解这句话了吧：数据行的物理顺序与列值的顺序相同，如果我们查询id比较靠后的数据，那么这行数据的地址在磁盘中的物理地址也会比较靠后。而且由于物理排列方式与聚集索引的顺序相同，所以也就只能建立一个聚集索引了。下图是实际存放示意图



从上图可以看出聚集索引的好处了，索引的叶子节点就是对应的数据节点（MySQL的MyISAM除外，此存储引擎的聚集索引和非聚集索引只多了个唯一约束，其他没什么区别），可以直接获取到对应的**全部列**的数据，而非聚集索引在索引没有覆盖到对应的列的时候需要进行**二次查询**，后面会详细讲。因此在查询方面，聚集索引的速度往往会更占优势。

创建聚集索引

如果不创建索引，系统会自动创建一个隐含列作为表的聚集索引。

创建表的时候指定主键（注意：SQL Sever默认主键为聚集索引，也可以指定为非聚集索引，而MySQL里主键就是聚集索引）

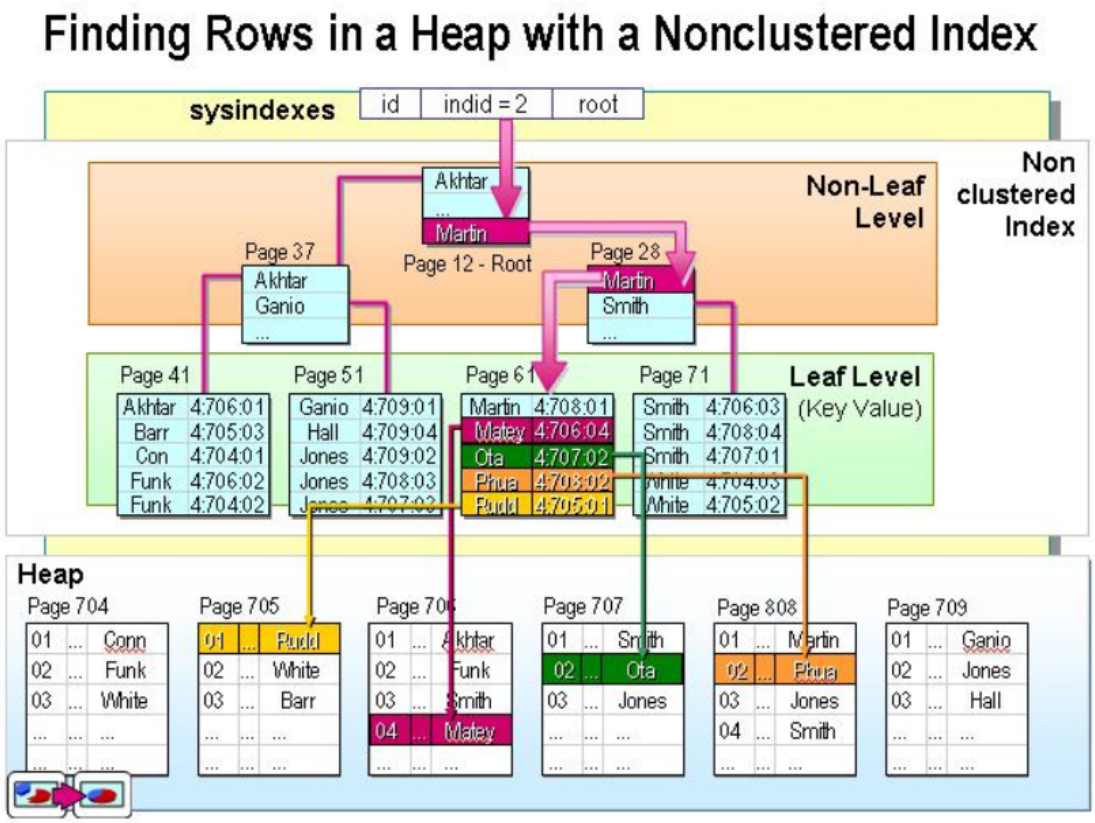
```
create table t1(
    id int primary key,
    name nvarchar(255)
)
```

值得注意的是，最好还是在创建表的时候添加聚集索引，由于聚集索引的物理顺序上的特殊性，因此如果再在上面创建索引的时候会根据索引列的排序移动全部数据行上面的顺序，会非常地耗费时间以及性能.所以主键一般使用自增整形。

非聚集索引

定义：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。

按照定义，除了聚集索引以外的索引都是非聚集索引，只是人们想细分一下非聚集索引，分成**普通索引，唯一索引，全文索引**。如果非要把非聚集索引类比成现实生活中的东西，那么非聚集索引就像新华字典的偏旁字典，他结构顺序与实际存放顺序不一定一致。



非聚集索引叶子节点仍然是索引节点，只是有一个指针指向对应的数据块，此如果使用**非聚集索引查询**，而查询**列**中包含了其他**该索引没有覆盖的列**，那么他还要进行第二次的查询，查询节点上对应的数据行的数据。

如有以下表t1：

id	username	score
1	小明	90
2	小红	80
3	小华	92
..	..	..
256	小英	70

该表有聚集索引clustered index(id), 非聚集索引index(username)

使用以下语句进行查询，不需要进行二次查询，直接就可以从非聚集索引的节点里面就可以获取到查询列的数据。

```
select id, username from t1 where username = '小明'
select username from t1 where username = '小明'
```

但是使用以下语句进行查询，就需要二次的查询去获取原数据行的score：

```
select username, score from t1 where username = '小明'
```

所以要注意的是非聚集索引其实叶子节点除了会存储索引覆盖列的数据，也会存放聚集索引所覆盖的列数据。

如何解决二次查询问题

使用复合索引（覆盖索引），建立两列以上的索引，即可查询复合索引里的列的数据而不需要进行回表二次查询，如index(col1, col2)，执行下面的语句：

```
select col1, col2 from t1 where col1 = '213';
```

要注意使用复合索引需要满足最左侧索引的原则，也就是查询的时候如果where条件里面没有最左边的一到多列，索引就不会起作用。

小结：

使用聚集索引的查询效率要比非聚集索引的效率要高，但是如果需要频繁去改变聚集索引的值，写入性能并不高，因为需要移动对应数据的物理位置。

非聚集索引在查询的时候可以的话就避免二次查询，这样性能会大幅提升。

不是所有的表都适合建立索引，只有数据量大表才适合建立索引，且建立在选择性高的列上面性能会更好。

基本语法

创建：create [unique] index indexName on tb_name(column_name)

添加：alter table tb_name add [unique] index [indexName] on (columnname)

删除：drop index [indexName] on tb_name;

创建索引场景

- 主键自动建立唯一索引
- 频繁作为查询条件的字段应该创建索引
- 查询中与其他表关联的字段,外键关系建立索引
- 频繁更新的字段不适合创建索引,因为每次更新不单单是更新数据还会更新索引
- Where条件里用得到的字段适合创建索引
- 单键/组合索引的选择问题,在高并发下倾向创建组合索引
- 查询中排序的字段,排序字段若通过索引去访问将大大提高排序速度
- 查询中统计或者分组字段

不适合创建索引场景

- 表记录太少(一般生产环境下,三百万条记录性能就可能开始下降,官方说的是五百万到八百万)
- 经常增删改的表
- 某个数据列的值包含许多重复的内容

复合索引场景分析

组合索引查询的各种场景 兹有 Index (A,B,C) ——组合索引多字段是有序的，并且是个完整的BTree 索引。

可以用上该组合索引	不能用上组合索引	用上部分组合索引
A>5	B>5	A>5 AND B=2
A=5 AND B>6	B=6 AND C=7	A=5 AND B>6 AND C=2
A=5 AND B=6 AND C=7		
A=5 AND B IN (2,3) AND C>5		

索引注意事项

- 建什么索引用什么索引,顺序也最好保持一致
- 最佳左前缀索引名称命名（如字段name,age,city,则索引命名应该是nameAgeCity或者xxx_nameAgeCity,顺序很重要）
- 不在索引列上做任何操作（计算，函数，or，类型转换），会导致索引失效而转向全表扫描
- 存储引擎不能使用索引中范围条件右边的列（如name='lin' and age>25 and city='qingdao',则age后面的索引会实效
- 尽量使用覆盖索引（只访问索引的查询（索引列和要查询的列一致）），减少select *
- MySQL在使用不等于（!=或者<>）的时候无法使用索引会导致全表扫描
- is null，is not null 也无法使用索引
- like以通配符在这（'%abc','%abc%'）两种情况会索引实效变成全表扫描，'abc%'则不会，若要'%abc','%abc%'不失效，建议使用覆盖索引，且查询的字段要少于索引或者与索引一致，不使用select *。如为name，age，city建了索引，请这么使用:select name或者select age,或者select city或者select name,age,city。如果select name,age,city,email则会全表扫描
- 字符串不加引号索引失效
- 少用or，用他来连接时索引会失效
- select * from A where exists (select 1 from where b.id=A.id)#当A表的数据系小于B表时，用exists优于in
- 使用join代替子查询

linux相关配置