# Java基础相关

## 基本数据类型



## **String， Stringbuffer， StringBuilder 的区别**

String 字符串常量(final修饰，不可被继承)，String是常量，当创建之后即不能更改。(可以通过StringBuffer和StringBuilder创建String对象(常用的两个字符串操作类)。)

StringBuffer 字符串变量（线程安全）,其也是final类别的，不允许被继承，其中的绝大多数方法都进行了同步处理，包括常用的Append方法也做了同步处理(synchronized修饰)。其自jdk1.0起就已经出现。其toString方法会进行对象缓存，以减少元素复制开销。 StringBuilder 字符串变量（非线程安全）其自jdk1.5起开始出现。与StringBuffer一样都继承和实现了同样的接口和类，方法除了没使用synch修饰以外基本一致，不同之处在于最后toString的时候，会直接返回一个新对象。

## ArrayList 和 LinkedList 有什么区别

ArrayList和LinkedList都实现了List接口，有以下的不同点：

1、ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。

2、相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

3、LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

## **类的实例化顺序**

父类静态代变量、父类静态代码块、子类静态变量、子类静态代码块、父类非静态变量（父类实例成员变量）、父类构造函数、子类非静态变量（子类实例成员变量）、子类构造函数。

## 继承和聚合的区别在哪。

继承指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系；在Java中此类关系通过关键字extends明确标识，在设计时一般没有争议性；

聚合是关联关系的一种特例，他体现的是整体与部分、拥有的关系，即has-a的关系，此时整体与部分之间是可分离的，他们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；比如计算机与CPU、公司与员工的关系等；表现在代码层面，和关联关系是一致的，只能从语义级别来区分；

## nio

IO(BIO)是面向流的，NIO是面向缓冲区的，BIO里用户最关心“我要读”，NIO里用户最关心"我可以读了"

BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。

NIO：New IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。

AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

### 选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### 数据处理

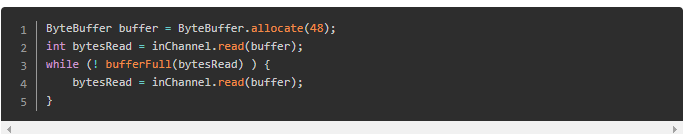
使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是仅从一个InputStream逐字节读取，而是数据必须先读入缓冲区再处理。

（Java IO: 从一个阻塞的流中读数据） 而一个NIO的实现会有所不同，下面是一个简单的例子：



注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。假设第一次read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。

所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：



bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在bufferFull()方法被调用之前状态相同。如果没有，下一个读入缓冲区的数据可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

Java NIO:从一个通道里读数据，直到所有的数据都读到缓冲区里.

## 反射的原理，反射创建类实例的三种方式是什么

Java反射是Java被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过Reflection APIs取得任何一个已知名称的class的内部信息，包括其modifiers（诸如public, static 等）、superclass（例如Object）、实现之interfaces（例如Cloneable），也包括fields和methods的所有信息，并可于运行时改变fields内容或唤起methods。

Java反射机制容许程序在运行时加载、探知、使用编译期间完全未知的classes。

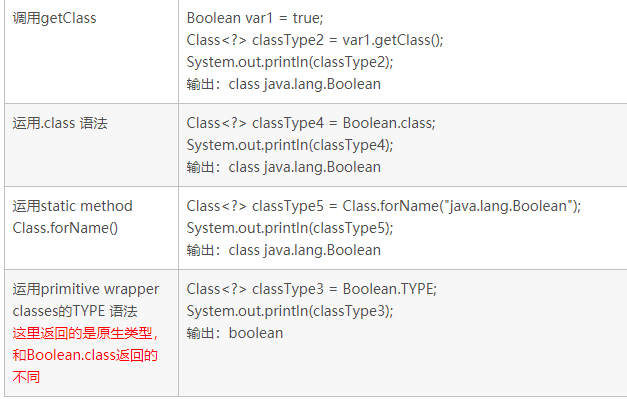
换言之，Java可以加载一个运行时才得知名称的class，获得其完整结构。

Class.forName(className)方法，其实调用的方法是Class.forName(className,true,classloader);注意看第2个boolean参数，它表示的意思，在loadClass后必须初始化。比较下我们前面准备jvm加载类的知识，我们可以清晰的看到在执行过此方法后，目标对象的 static块代码已经被执行，static参数也已经被初始化。

再看ClassLoader.loadClass(className)方法，其实他调用的方法是ClassLoader.loadClass(className,false);还是注意看第2个 boolean参数，该参数表示目标对象被装载后不进行链接，这就意味这不会去执行该类静态块中间的内容。因此2者的区别就显而易见了。

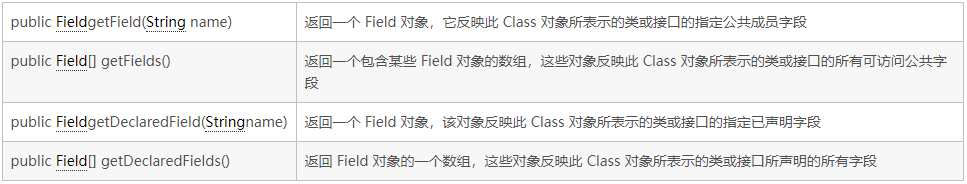
### 获取类的Class对象

Class 类的实例表示正在运行的 Java 应用程序中的类和接口。获取类的Class对象有多种方式：



### 获取类的Fields

可以通过反射机制得到某个类的某个属性，然后改变对应于这个类的某个实例的该属性值。JAVA 的Class<T>类提供了几个方法获取类的属性。



通过Java的反射机制，可以在运行期间调用对象的任何方法;

### 获取类的Method

通过反射机制得到某个类的某个方法，然后调用对应于这个类的某个实例的该方法

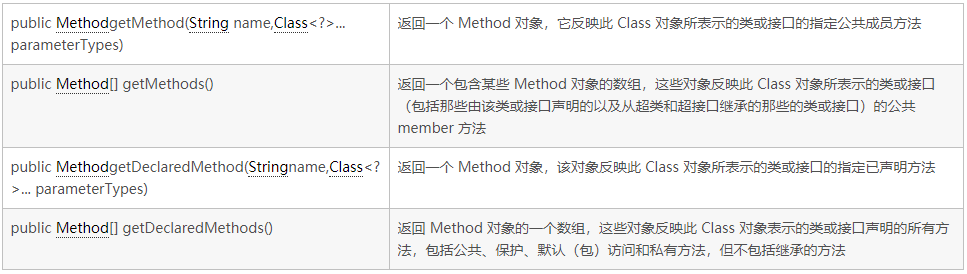
Class<T>类提供了几个方法获取类的方法。



### 获取类的Constructor

通过反射机制得到某个类的构造器，然后调用该构造器创建该类的一个实例

Class<T>类提供了几个方法获取类的构造器。



### 新建类的实例

通过反射机制创建新类的实例，有几种方法可以创建



### 调用类的函数

通过反射获取类Method对象，调用Field的Invoke方法调用函数。

### 设置/获取类的属性值

通过反射获取类的Field对象，调用Field方法设置或获取值

## 描述动态代理的几种实现方式，分别说出相应的优缺点

Jdk cglib jdk底层是利用反射机制，需要基于接口方式，这是由于



Cglib则是基于asm框架，实现了无反射机制进行代理，利用空间来换取了时间，代理效率高于jdk,

总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。由此可以看出，jdk动态代理有一定的局限性，cglib这种第三方类库实现的动态代理应用更加广泛，且在效率上更有优势。

Cglib是一个优秀的动态代理框架，它的底层使用ASM(ASM是一个通用的Java字节码操作和分析框架。 它可以用于修改现有类或直接以二进制形式动态生成类。)在内存中动态的生成被代理类的子类，使用CGLIB即使代理类没有实现任何接口也可以实现动态代理功能。CGLIB具有简单易用，它的运行速度要远远快于JDK的Proxy动态代理：

cglib有两种可选方式，继承和引用。第一种是基于继承实现的动态代理，所以可以直接通过super调用target方法，但是这种方式在spring中是不支持的，因为这样的话，这个target对象就不能被spring所管理，所以cglib还是才用类似jdk的方式，通过持有target对象来达到拦截方法的效果。

## final 的用途

类、变量、方法

final 修饰的类叫最终类，该类不能被继承。

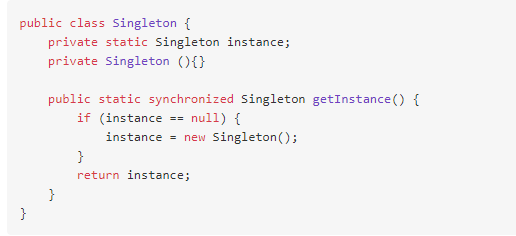
final 修饰的方法不能被重写。

final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

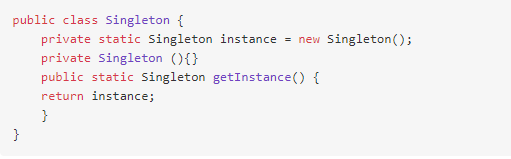
## 单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点！

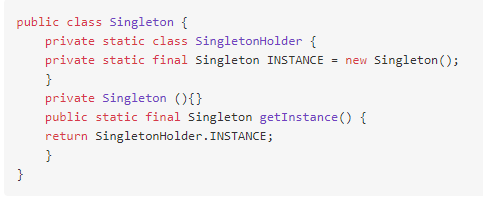
懒汉，线程不安全

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

饿汉

这种方式基于classloder机制避免了多线程的同步问题，instance在类装载时就实例化。目前java单例是指一个虚拟机的范围，因为装载类的功能是虚拟机的，所以一个虚拟机在通过自己的ClassLoader装载饿汉式实现单例类的时候就会创建一个类的实例。这就意味着一个虚拟机里面有很多ClassLoader，而这些classloader都能装载某个类的话，就算这个类是单例，也能产生很多实例。当然如果一台机器上有很多虚拟机，那么每个虚拟机中都有至少一个这个类的实例的话，那这样 就更不会是单例了。(这里讨论的单例不适合集群！)

静态内部类

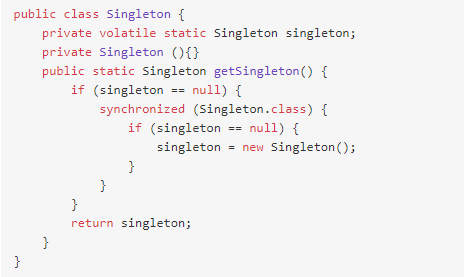
 这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载！这个时候，这种方式相比第2种方式就显得很合理。

4.枚举



这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

5.双重校验锁(jdk1.5)

这样方式实现线程安全地创建实例，而又不会对性能造成太大影响。它只是第一次创建实例的时候同步，以后就不需要同步了。

由于volatile关键字屏蔽了虚拟机中一些必要的代码优化，所以运行效率并不是很高，因此建议没有特别的需要不要使用。双重检验锁方式的单例不建议大量使用，根据情况决定。

## hashcode 和 equals 实现

hashcode

hashcode（）方法提供了对象的hashCode值，是一个native方法，返回的默认值与System.identityHashCode(obj)一致。

通常这个值是对象头部的一部分二进制位组成的数字，具有一定的标识对象的意义存在，但绝不定于地址。

作用是：用一个数字来标识对象。比如在HashMap、HashSet等类似的集合类中，如果用某个对象本身作为Key，即要基于这个对象实现Hash的写入和查找，那么对象本身如何实现这个呢？就是基于hashcode这样一个数字来完成的，只有数字才能完成计算和对比操作。

### hashcode是否唯一

hashcode只能说是标识对象，在hash算法中可以将对象相对离散开，这样就可以在查找数据的时候根据这个key快速缩小数据的范围，但hashcode不一定是唯一的，所以hash算法中定位到具体的链表后，需要循环链表，然后通过equals方法来对比Key是否是一样的。

### equals与hashcode的关系

equals相等两个对象，则hashcode一定要相等。但是hashcode相等的两个对象不一定equals相等。

### 覆盖equals时总要覆盖hashCode

一个很常见的错误根源在于没有覆盖hashCode方法。在每个覆盖了equals方法的类中，也必须覆盖hashCode方法。如果不这样做的话，就会违反Object.hashCode的通用约定，从而导致该类无法结合所有基于散列的集合一起正常运作，这样的集合包括HashMap、HashSet和Hashtable。

在应用程序的执行期间，只要对象的equals方法的比较操作所用到的信息没有被修改，那么对这同一个对象调用多次，hashCode方法都必须始终如一地返回同一个整数。在同一个应用程序的多次执行过程中，每次执行所返回的整数可以不一致。

如果两个对象根据equals()方法比较是相等的，那么调用这两个对象中任意一个对象的hashCode方法都必须产生同样的整数结果。

如果两个对象根据equals()方法比较是不相等的，那么调用这两个对象中任意一个对象的hashCode方法，则不一定要产生相同的整数结果。但是程序员应该知道，给不相等的对象产生截然不同的整数结果，有可能提高散列表的性能。

## 数组与链表的优缺点和区别

数组　是将元素在内存中**连续存放**，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要**快速访问数据，很少插入和删除元素，就应该用数组。**

　　链表　中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起，每个结点包括两个部分：一个是存储 数据元素 的　数据域，另一个是存储下一个结点地址的 指针。   
　　如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要**经常插入和删除元素你就需要用链表**。

### 内存存储区别

数组从栈中分配空间, 对于程序员方便快速,但自由度小。

链表从堆中分配空间, 自由度大但申请管理比较麻烦.

### 总结

1、存取方式上，数组可以顺序存取或者随机存取，而链表只能顺序存取；

2、存储位置上，数组逻辑上相邻的元素在物理存储位置上也相邻，而链表不一定；

3、存储空间上，链表由于带有指针域，存储密度不如数组大；

4、按序号查找时，数组可以随机访问，时间复杂度为O(1)，而链表不支持随机访问，平均需要O(n)；

5、按值查找时，若数组无序，数组和链表时间复杂度均为O(1)，但是当数组有序时，可以采用折半查找将时间复杂度降为O(logn)；

6、插入和删除时，数组平均需要移动n/2个元素，而链表只需修改指针即可；

7、空间分配方面：

　　数组在静态存储分配情形下，存储元素数量受限制，动态存储分配情形下，虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且如果内存中没有更大块连续存储空间将导致分配失败；

　　链表存储的节点空间只在需要的时候申请分配，只要内存中有空间就可以分配，操作比较灵活高效；

## 异常分类

java.lang.Throwable是所有异常的根

java.lang.Error是错误信息

java.lang.Exception是异常信息

### Checked异常

1 当前方法知道如何处理该异常，则用try...catch块来处理该异常。

2 当前方法不知道如何处理，则在定义该方法是声明抛出该异常。

我们比较熟悉的Checked异常有

Java.lang.ClassNotFoundException

Java.lang.NoSuchMetodException

java.io.IOException

### RuntimeException

Runtime如除数是0和数组下标越界等，其产生频繁，处理麻烦，若显示申明或者捕获将会对程序的可读性和运行效率影响很大。所以由系统自动检测并将它们交给缺省的异常处理程序。当然如果你有处理要求也可以显示捕获它们。

Java.lang.ClassCastException

Java.lang.IndexOutOfBoundsException

Java.lang.NullPointerException

java.lang.NumberFormatException

### Error

当程序发生不可控的错误时，通常做法是通知用户并中止程序的执行。与异常不同的是Error及其子类的对象不应被抛出。

Error是throwable的子类，代表编译时间和系统错误，用于指示合理的应用程序不应该试图捕获的严重问题。

Error由Java虚拟机生成并抛出，包括动态链接失败，虚拟机错误等。程序对其不做处理。

## 泛型

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数，泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，以提高代码的重用率

Java泛型是J2 SE1.5中引入的一个新特性，其本质是参数化类型，也就是说所操作的数据类型被指定为一个参数（type parameter）这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。

Java语言中引入泛型是一个较大的功能增强。不仅语言、类型系统和编译器有了较大的变化，以支持泛型，而且类库也进行了很大的改动，许多重要的类，比如集合框架，都已经成为泛型化的了。这带来了很多好处：

1、类型安全

泛型的主要目标是提高Java程序的类型安全。通过知道使用泛型定义的变量的类型限制，编译器可以在非常高的层次上验证类型假设。没有泛型，这些假设就只存在于系统开发人员的头脑中。

通过在变量声明中捕获这一附加的类型信息，泛型允许编译器实施这些附加的类型约束。类型错误就可以在编译时被捕获了，而不是在运行时当作ClassCastException展示出来。将类型检查从运行时挪到编译时有助于Java开发人员更早、更容易地找到错误，并可提高程序的可靠性。

2、消除强制类型转换

泛型的一个附带好处是，消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。尽管减少强制类型转换可以提高使用泛型类的代码的累赞程度，但是声明泛型变量时却会带来相应的累赞程度。在简单的程序中使用一次泛型变量不会降低代码累赞程度。但是对于多次使用泛型变量的大型程序来说，则可以累积起来降低累赞程度。所以泛型消除了强制类型转换之后，会使得代码加清晰和筒洁。

3、更高的运行效率

在非泛型编程中，将筒单类型作为Object传递时会引起Boxing（装箱）和Unboxing（拆箱）操作，这两个过程都是具有很大开销的。引入泛型后，就不必进行Boxing和Unboxing操作了，所以运行效率相对较高，特别在对集合操作非常频繁的系统中，这个特点带来的性能提升更加明显。

4、潜在的性能收益

泛型为较大的优化带来可能。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，Java系统开发人员会指定这些强制类型转换）插入生成的字节码中。但是更多类型信息可用于编译器这一事实，为未来版本的JVM的优化带来可能

# Java多线程

# Java虚拟机

## jvm 和 类的关系

当调用 java命令运行一个java程序时，必会启动一个jvm即java虚拟机。该java程序的所有线程，变量都处于jvm中，都使用该jvm的内存区

jvm终止的情况：

1.程序自然运行结束

2.遇到System.exit();Runtime.getRuntime.exit();

3.遇到未捕获异常或错误时

4.程序所在的平台强制结束了JVM进程

jvm终止，jvm内存中的数据全部丢失。

### 类的加载

类的加载 又称为 类的初始化，实际上可细分为 类的 加载、连接、初始化。下面将讲述着三个阶段的过程!

类的加载 指.class文件读入内存，并为之创建一个 java.lang.Class对象

类加载，是通过类加载器来完成的，类加载器通常由JVM提供，通常称为系统类加载器（也可以是自己写的加载器，只要继承ClassLoader基类）。

类加载无须等到“首次使用该类”时加载，jvm允许预加载某些类。。。。

### 类的连接

负责把类的二进制数据合并到JRE（java运行环境）中

1.验证 检测被加载的类是否有正确的内部结构，并和其他类协调一致

2.准备 负责为类的类变量（非对象变量）分配内存，并设置默认初始值

3.解析 将类的二进制数据中的符号引用替换成直接引用。。(static final 好像跟这个有点关系？？？？5.6.处有联系）

### 类初始化

主要对类变量（而非对象变量）的初始化

声明类变量的初始值 = 静态初始化块 他们是相同的，等效的。都会被当成类的初始化语句，JVM会按照这些语句在程序中的顺序依次执行他们

### 类初始化的时机

1.创建类的实例。new,反射，反序列化

2.使用某类的类方法--静态方法

3.访问某类的类变量，或赋值类变量

4.反射创建某类或接口的Class对象。Class.forName("Hello");---注意：loadClass调用ClassLoader.loadClass(name,false)方法,没有link，自然没有initialize

5.初始化某类的子类

6.直接使用java.exe来运行某个主类。即cmd java 程序会先初始化该类。

### 类加载器

类加载器 负责加载所有的类，为被加载如内存中的类生成一个java.lang.Class实例。一旦类被载入内存，同一个类就不会再加载第二次

如何判断是同一个类：

java中 一个类用其 全限定类名标示--包名+类名

jvm中 一个类用其 全限定类名+加载器标示---包名+类名+加载器名

加载器层次结构：

JVM启动时，姓曾的三个类加载器组成的机构

1.Bootstrap ClassLoader 根类 ------引导类加载器，加载java核心类。非java.lang.ClassLoader子类，而是JVM自身实现

2.Extension ClassLoader 扩展类-----加载JRE的扩展目录中的JAR包的类（%JAVA\_HOME%/jre/lib/ext或java.ext.dirs系统属性指定的目录）

3.System ClassLoader 系统类-----加载cmd java -cp \*\*,环境变量指定的jar包和类路径。ClassLoader.getSystemClassLoader获得 系统类加载器。

### 类的加载机制：

1.全盘负责。某类以及其所依赖的所有类，都由一个加载器负责加载。除非显示使用另外一个加载器。

2.父类委托。先父类加载器加载改Class，不行后，才尝试从自己的类路径中加载该类

3.缓存机制。缓存机制将会保证所有加载过的Class都会被缓存。。当程序需要Class时，先从缓存区中寻找Class对象，没有的话，才加载该类的.class对象。

### 加载Class9个步骤：

1.检测此Class 是否被载入过（即在缓存区中是否由此 Class），有，则进入第8步，否则执行第2步。

2.如果父类加载器不存在（要么parent 一定是根类加载器，要么本身就是根类加载器），则跳到第4步；如果父类加载器存在，则执行第3步。

3.请求使用父类加载器去载入目标类，如果成功则跳到第8步，否则执行第5步

4.请求使用 根类加载器 载入目标类，成功则跳到第8步，否则跳到第7步

5.当前类加载器 尝试寻找 Class文件（从与此ClassLoader相关的类路径中寻找），如果找到则执行第6步，否则跳到第7步。

6.从文件中载入Class，成功后跳到第8步。

7.抛出ClassNotFoundException异常。

8.返回对应的 java.lang.Class对象。

# Java集合框架

## Map

hashMap是线程不安全的，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，采用哈希表来存储的, 它根据键的hashCode值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。 HashMap最多只允许一条记录的键为null，允许多条记录的值为null。HashMap非线程安全，即任一时刻可以有多个线程同时写HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections的synchronizedMap方法使HashMap具有线程安全的能力，或者使用ConcurrentHashMap。

Hashtable：Hashtable是遗留类，很多映射的常用功能与HashMap类似，不同的是它承自Dictionary类，并且是线程安全的，任一时间只有一个线程能写Hashtable，并发性不如ConcurrentHashMap，因为ConcurrentHashMap引入了分段锁。Hashtable不建议在新代码中使用，不需要线程安全的场合可以用HashMap替换，需要线程安全的场合可以用ConcurrentHashMap替换。

LinkedHashMap：LinkedHashMap是HashMap的一个子类，保存了记录的插入顺序，在用Iterator遍历LinkedHashMap时，先得到的记录肯定是先插入的，也可以在构造时带参数，按照访问次序排序。

TreeMap：TreeMap实现SortedMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。如果使用排序的映射，建议使用TreeMap。在使用TreeMap时，key必须实现Comparable接口或者在构造TreeMap传入自定义的Comparator，否则会在运行时抛出java.lang.ClassCastException类型的异常。

对于上述四种Map类型的类，要求映射中的key是不可变对象。不可变对象是该对象在创建后它的哈希值不会被改变。如果对象的哈希值发生变化，Map对象很可能就定位不到映射的位置了。

## HashMap的内部实现

HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

### 结构实现

从结构实现来讲，HashMap是数组+链表+红黑树（JDK1.8增加了红黑树部分）实现的，(1) 从源码可知，HashMap类中有一个非常重要的字段，就是 Node[] table，即哈希桶数组，明显它是一个Node的数组。我们来看Node[]是何物。

static class Node<K,V> implements Map.Entry<K,V> {

final int hash; //用来定位数组索引位置

final K key;

V value;

Node<K,V> next; //链表的下一个node

}

Node是HashMap的一个内部类，实现了Map.Entry接口，本质是就是一个映射(键值对)。上图中的每个黑色圆点就是一个Node对象。

HashMap就是使用哈希表来存储的。哈希表为解决冲突，可以采用开放地址法和链地址法等来解决问题，Java中HashMap采用了链地址法。链地址法，简单来说，就是数组加链表的结合。在每个数组元素上都一个链表结构，当数据被Hash后，得到数组下标，把数据放在对应下标元素的链表上。

系统调用key的hashCode()方法得到其hashCode 值（该方法适用于每个Java对象），然后再通过Hash算法的后两步运算（高位运算和取模运算，下文有介绍）来定位该键值对的存储位置，有时两个key会定位到相同的位置，表示发生了Hash碰撞。当然Hash算法计算结果越分散均匀，Hash碰撞的概率就越小，map的存取效率就会越高。

如果哈希桶数组很大，即使较差的Hash算法也会比较分散，如果哈希桶数组数组很小，即使好的Hash算法也会出现较多碰撞，所以就需要在空间成本和时间成本之间权衡，其实就是在根据实际情况确定哈希桶数组的大小，并在此基础上设计好的hash算法减少Hash碰撞。那么通过什么方式来控制map使得Hash碰撞的概率又小，哈希桶数组（Node[] table）占用空间又少呢？答案就是好的Hash算法和扩容机制。

在理解Hash和扩容流程之前，我们得先了解下HashMap的几个字段。从HashMap的默认构造函数源码可知，构造函数就是对下面几个字段进行初始化，源码如下：

int threshold; // 所能容纳的key-value对极限

final float loadFactor; // 负载因子

int modCount;

int size;

首先，Node[] table的初始化长度length(默认值是16)，Load factor为负载因子(默认值是0.75)，threshold是HashMap所能容纳的最大数据量的Node(键值对)个数。threshold = length \* Load factor。也就是说，在数组定义好长度之后，负载因子越大，所能容纳的键值对个数越多。

结合负载因子的定义公式可知，threshold就是在此Load factor和length(数组长度)对应下允许的最大元素数目，超过这个数目就重新resize(扩容)，扩容后的HashMap容量是之前容量的两倍。默认的负载因子0.75是对空间和时间效率的一个平衡选择，建议大家不要修改，除非在时间和空间比较特殊的情况下，如果内存空间很多而又对时间效率要求很高，可以降低负载因子Load factor的值；相反，如果内存空间紧张而对时间效率要求不高，可以增加负载因子loadFactor的值，这个值可以大于1。

size这个字段其实很好理解，就是HashMap中实际存在的键值对数量。注意和table的长度length、容纳最大键值对数量threshold的区别。而modCount字段主要用来记录HashMap内部结构发生变化的次数，主要用于迭代的快速失败。强调一点，内部结构发生变化指的是结构发生变化，例如put新键值对，但是某个key对应的value值被覆盖不属于结构变化。

在HashMap中，哈希桶数组table的长度length大小必须为2的n次方(一定是合数)，这是一种非常规的设计，常规的设计是把桶的大小设计为素数。相对来说素数导致冲突的概率要小于合数，具体证明可以参考http://blog.csdn.net/liuqiyao\_01/article/details/14475159，Hashtable初始化桶大小为11，就是桶大小设计为素数的应用（Hashtable扩容后不能保证还是素数）。HashMap采用这种非常规设计，主要是为了在取模和扩容时做优化，同时为了减少冲突，HashMap定位哈希桶索引位置时，也加入了高位参与运算的过程。

这里存在一个问题，即使负载因子和Hash算法设计的再合理，也免不了会出现拉链过长的情况，一旦出现拉链过长，则会严重影响HashMap的性能。于是，在JDK1.8版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高HashMap的性能，其中会用到红黑树的插入、删除、查找等算法。

### 确定哈希桶数组索引位置

这里的Hash算法本质上就是三步：取key的hashCode值、高位运算、取模运算。

对于任意给定的对象，只要它的hashCode()返回值相同，那么程序调用方法一所计算得到的Hash码值总是相同的。我们首先想到的就是把hash值对数组长度取模运算，这样一来，元素的分布相对来说是比较均匀的。但是，模运算的消耗还是比较大的，在HashMap中是这样做的：调用方法二来计算该对象应该保存在table数组的哪个索引处。

方法一：

static final int hash(Object key) { //jdk1.8 & jdk1.7

int h;

// h = key.hashCode() 为第一步 取hashCode值

// h ^ (h >>> 16) 为第二步 高位参与运算

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

}

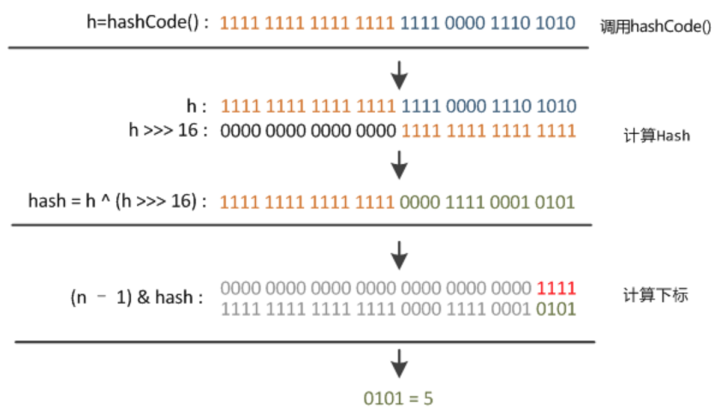
方法二：

static int indexFor(int h, int length) { //jdk1.7的源码，jdk1.8没有这个方法，但是实现原理一样的

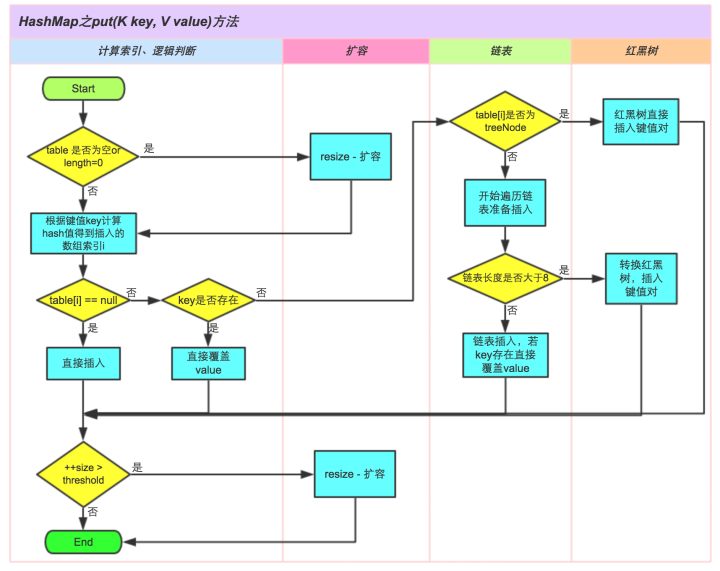
return h & (length-1); //第三步 取模运算

}

这个方法非常巧妙，它通过h & (table.length -1)来得到该对象的保存位，而HashMap底层数组的长度总是2的n次方，这是HashMap在速度上的优化。当length总是2的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率。



### 分析HashMap的put方法



①.判断键值对数组table[i]是否为空或为null，否则执行resize()进行扩容；

②.根据键值key计算hash值得到插入的数组索引i，如果table[i]==null，直接新建节点添加，转向⑥，如果table[i]不为空，转向③；

③.判断table[i]的首个元素是否和key一样，如果相同直接覆盖value，否则转向④，这里的相同指的是hashCode以及equals；

④.判断table[i] 是否为treeNode，即table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向⑤；

⑤.遍历table[i]，判断链表长度是否大于8，大于8的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现key已经存在直接覆盖value即可；

⑥.插入成功后，判断实际存在的键值对数量size是否超多了最大容量threshold，如果超过，进行扩容。

### 扩容机制

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

JDK1.7的代码，好理解一些，本质上区别不大，具体区别后文再说。

1 void resize(int newCapacity) { //传入新的容量

2 Entry[] oldTable = table; //引用扩容前的Entry数组

3 int oldCapacity = oldTable.length;

4 if (oldCapacity == MAXIMUM\_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了

5 threshold = Integer.MAX\_VALUE; //修改阈值为int的最大值(2^31-1)，这样以后就不会扩容了

6 return;

7 }

8

9 Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组

10 transfer(newTable); //！！将数据转移到新的Entry数组里

11 table = newTable; //HashMap的table属性引用新的Entry数组

12 threshold = (int)(newCapacity \* loadFactor);//修改阈值

13 }

1 void transfer(Entry[] newTable) {

2 Entry[] src = table; //src引用了旧的Entry数组

3 int newCapacity = newTable.length;

4 for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组

5 Entry<K,V> e = src[j]; //取得旧Entry数组的每个元素

6 if (e != null) {

7 src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）

8 do {

9 Entry<K,V> next = e.next;

10 int i = indexFor(e.hash, newCapacity); //！！重新计算每个元素在数组中的位置

11 e.next = newTable[i]; //标记[1]

12 newTable[i] = e; //将元素放在数组上

13 e = next; //访问下一个Entry链上的元素

14 } while (e != null);

15 }

16 }

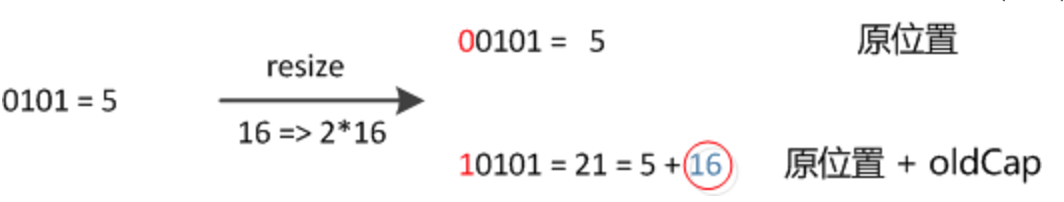
17 }

newTable[i]的引用赋给了e.next，也就是使用了单链表的头插入方式，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(如果发生了hash冲突的话），这一点和Jdk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

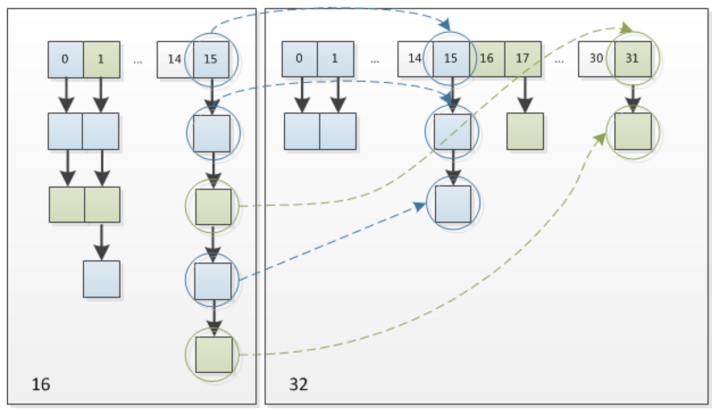
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希与高位运算结果。



元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：



这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞:

### ****线程安全性****

在多线程使用场景中，应该尽量避免使用线程不安全的HashMap，而使用线程安全的ConcurrentHashMap。那么为什么说HashMap是线程不安全的，下面举例子说明在并发的多线程使用场景中使用HashMap可能造成死循环。代码例子如下(便于理解，仍然使用JDK1.7的环境)：

### Fail-Fast 机制

java.util.HashMap 不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了 map，那么将抛出 ConcurrentModificationException，这就是所谓 fail-fast 策略。一种错误机制。 当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。某一个线程 A 通过 iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程 A 访问集合时，就会抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。

这一策略在源码中的实现是通过 modCount 域，modCount 顾名思义就是修改次数，对 HashMap 内容（当然不仅仅是 HashMap 才会有，其他例如 ArrayList 也会）的修改都将增加这个值（大家可以再回头看一下其源码，在很多操作中都有 modCount++ 这句），那么在迭代器初始化过程中会将这个值赋给迭代器的 expectedModCount。





### ****小结****

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。

(4) JDK1.8引入红黑树大程度优化了HashMap的性能。

## ConcurrentHashMap

Hashtable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下Hashtable的效率非常低下。因为当一个线程访问Hashtable的同步方法时，其他线程访问Hashtable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

JDK1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全。数据结构采用：数组+链表+红黑树。

Hashtable的任何操作都会把整个表锁住，是阻塞的。好处是总能获取最实时的更新，比如说线程A调用putAll写入大量数据，期间线程B调用get，线程B就会被阻塞，直到线程A完成putAll，因此线程B肯定能获取到线程A写入的完整数据。坏处是所有调用都要排队，效率较低。

ConcurrentHashMap 是设计为非阻塞的。在更新时会局部锁住某部分数据，但不会把整个表都锁住。同步读取操作则是完全非阻塞的。好处是在保证合理的同步前提下，效率很高。坏处是严格来说读取操作不能保证反映最近的更新。例如线程A调用putAll写入大量数据，期间线程B调用get，则只能get到目前为止已经顺利插入的部分数据。

应该根据具体的应用场景选择合适的HashMap。

## HashSet 的实现原理

默认情况下采用的是 initial capacity为16，load factor 为 0.75。

### HashSet 的实现

对于 HashSet 而言，它是基于 HashMap 实现的，HashSet 底层使用 HashMap 来保存所有元素，因此 HashSet 的实现比较简单，相关 HashSet 的操作，基本上都是直接调用底层 HashMap 的相关方法来完成，我们应该为保存到 HashSet 中的对象覆盖 hashCode() 和 equals()

### add 方法



如果此 set 中尚未包含指定元素，则添加指定元素。更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2)) 的元素 e2，则向此 set 添加指定的元素 e。如果此 set 已包含该元素，则该调用不更改 set 并返回 false。但底层实际将将该元素作为 key 放入 HashMap。

由于 HashMap 的 put() 方法添加 key-value 对时，当新放入 HashMap 的 Entry 中 key 与集合中原有 Entry 的 key 相同（hashCode()返回值相等，通过 equals 比较也返回 true），新添加的 Entry 的 value 会将覆盖原来 Entry 的 value（HashSet 中的 value 都是PRESENT），但 key 不会有任何改变，因此如果向 HashSet 中添加一个已经存在的元素时，新添加的集合元素将不会被放入 HashMap中，原来的元素也不会有任何改变，这也就满足了 Set 中元素不重复的特性。

该方法如果添加的是在 HashSet 中不存在的，则返回 true；如果添加的元素已经存在，返回 false。其原因在于我们之前提到的关于 HashMap 的 put 方法。该方法在添加 key 不重复的键值对的时候，会返回 null。



# Spring相关

# Mysql相关

# 其他拓展思维

## 使用有限内存对大型数据文件排序

切分：从目标数据文件中读取数据，读取一定数量后对读取到的数据进行排序，并生成临时排序文件，重复此过程，将原始数据文件分割为若干个已排序的数据文件

合并：根据上一阶段得到的分组文件数量，如果内存不足以一次创建所有文件的指针，则需要进行多次合并。合并时，将若干个临时数据文件合并为更大的数据文件，使用归并排序思想，使用优先级队列辅助。重复此过程直到生成的临时数据文件个数足够少。

## 不使用其他变量交换两个数

1. 加减法

x = x + y;

y = x - y;

x = x - y;//已改正

1. 异或

x = x ^ y;

y = x ^ y;

x = x ^ y;