[JAVA知识点：](#_java知识点：)

[MYSQL知识点](#_MYSQL知识点)

[LINUX知识点](#_LINUX知识点)

[网络知识点](#_网络知识点)

[算法：](#_算法：)

[SQL优化：](#_SQL优化：)

# java知识点：

抽象类和接口的区别：

抽象类用extends被子类继承，接口用implements

抽象类可以有构造器，接口不可以有，

抽象类可以有默认的方法实现，接口不可以有方法实现

抽象类可以有public，protected，default这些修饰符，接口只有public

只能继承一个抽象类，但可以继承多个接口

抽象类和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

堆栈：先进后出

队列：先进先出

②堆是在程序运行时，而不是在程序编译时，申请某个大小的内存空间。即动态分配内存，对其访问和对一般内存的访问没有区别。

③堆是应用程序在运行的时候请求操作系统分配给自己内存，一般是申请/给予的过程。

④堆是指程序运行时申请的动态内存，而栈只是指一种使用堆的方法(即先进后出)。

栈又叫堆栈

堆是这样一种完全二叉树：根节点的值大于等于左右孩子节点的值（最大堆）或者根节点的值小于等于左右孩子节点的值（最小堆）。堆也是递归定义的，即堆的孩子节点本身也是堆。使用数组存储堆。

堆具有以下性质：

1）完全二叉树A[0:n-1]中的任意节点，索引为i的节点，其左右孩子节点是2\*i+1和2\*i+2。

2）非叶子节点最大索引是⌊n/2⌋-1，叶子节点最小索引是⌊n/2⌋。

3）最大(最小)堆的左右子树也是最大（小）堆。

synchronized修饰方法时锁定的是调用该方法的对象。它并不能使调用该方法的多个对象在执行顺序上互斥，静态修饰符很有必要。因此当不适用静态时，创建多个对象执行该方法，锁都不一样，还同步什么呢，因此当创建多个对象时用static修饰后才能实现想要的效果。

当变量的值在（-128,127）之间（也就是可以用一个字节所能表示的int值）时才会被放入常量池，否则会自动装箱生成普通Integer对象

**String.intern：**如果常量池中已经存在该字符串，则直接返回常量池中该对象的引用。否则，在常量池中加入该对象，然后 返回引用。

如何迭代Map容器：

---------------------------法1------------------------------

    Map<String, String> map = new HashMap<>();

    Set<Map.Entry<String, String>> entrySet = map.entrySet()

    for(Map.Entry<String, String> entry : entrySet){

        String key = entry.getKey();

        String value = entry.getValue();

    }

-----------------------法2-----------------------------------

    Set<String> keySet = map.keySet()

    Iterator<String> it1 = keySet.iterator();

    if(it1.hasNext())

        System.out.println(it1.next());

-----------------------法3---------------------------------------

    Collection<String> values = map.values();

    Iterator<String> it2 = values.iterator();

    if(it2.hasNext()){

        System.out.println(it2.next());

**java垃圾回收机制简单介绍**

可达性分析法：该方法的基本思想是通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。

目前Jvm使用的垃圾回收算法：它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。分为老年代（Tenured Generation）和新生代（Young Generation）。老年代的内存区域的对象一般回收频率比较低，采用了Mark-Compact算法，而新生代的内存区域由于每次需要回收大量对象，回收频率较高，所以将该区域又划分成了一个较大的Eden空间和两个较小的Suivivor空间，每次使用Eden空间和一个Survivor空间的，当需要回收垃圾时，将Eden空间和该Survivor空间的存活对象复制到另外一块Survivor空间上，然后清理到Eden和刚才使用的Survivor空间，实现垃圾回收机制.

**Mark-Compact（标记-整理）算法**   
Mark-Compact算法是在Mark-Sweep算法的基础上进行了改进，算法标记跟Mark-Sweep一样，只是在标记完之后，将标记的对象向一端移动，然后清理掉边界以外的内存区域，这样就解决了内存碎片化的问题

构造器不能被继承，因此不能被重写Override，但可以被重载Overload; 在子类构造对象时发现，访问子类构造函数时，父类构造函数也运行了； 原因是：在子类的构造函数第一行有一个默认的隐式语句：super();子类实例化过程：子类中所有的构造函数默认都会访问父类中的空参数的构造函数。

**HashTable** 是同步的(线程安全)，而**HashMap** 线程不安全，**效率上HashMap 更快**

**HashMap**允许空键值，而**Hashtable** 不允许

HashMap默认的初始大小是16，装载因子是0.75，并且扩容的大小一定是2的指数；而HashTable的默认初始大小是11，扩容的方式是 2\*old+1；

HashMap 的iterator 迭代器执行快速失败机制，也就是说在迭代过程中修改集合结构，除非调用迭代器自身的remove 方法，否则以其他任何方式的修改都将抛出并发修改异常。而Hashtable 提供了对键的列举(Enumeration)，Hashtable 返回的Enumeration 不是快速失败的。

Hashtable 和ConcurrentHashMap 都可以用于多线程的环境，但是当Hashtable 的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间；因为ConcurrentHashMap 引入了分割(segmentation)，不论它变得多么大，仅仅需要锁定map 的某个部分，而其它的线程不需要等到迭代完成才能访问map。简而言之，在迭代的过程中，ConcurrentHashMap 仅仅锁定map 的某个部分，而Hashtable 则会锁定整个map。

CurrentHashMap，是将HashMap分成了很多个片（一般默认是16片），引入了分段锁的概念，然后对每一片加锁，具体可以理解成一把大的Map分解成N个小的HashTable，根据key.hashCode（）来决定放到哪一个片上

String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，

*TreeMap*实现了*SortedMap*接口，也就是说会按照key的大小顺序对*Map*中的元素进行排序，

出于性能原因，TreeMap是非同步的, 如果需要在多线程环境使用:

SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));

TreeMap底层通过红黑树（Red-Black tree）实现，也就意味着containsKey(), get(), put(), remove()都有着log(n)的时间复杂度

**JVM知识点：**

 我们都知道Java源文件，通过编译器，能够生产相应的.Class文件，也就是字节码文件，而字节码文件又通过Java虚拟机中的解释器，编译成特定机器上的机器码 。

JVM调优主要就是优化 Heap堆 和 Method Area 方法区。

JVM内存空间包含：方法区、java堆、java栈、本地方法栈

方法区是各个线程共享的区域，存放类信息、常量、静态变量。

java堆也是线程共享的区域，我们的类的实例就放在这个区域，可以想象你的一个系统会产生很多实例，因此java堆的空间也是最大的。如果java堆空间不足了，程序会抛出OutOfMemoryError异常。

java栈是每个线程私有的区域，它的生命周期与线程相同，一个线程对应一个java栈，每执行一个方法就会往栈中压入一个元素，这个元素叫“栈帧”，而栈帧中包括了方法中的局部变量、用于存放中间状态值的操作栈，如果java栈空间不足了，程序会抛出StackOverflowError异常，想一想什么情况下会容易产生这个错误，对，递归，递归如果深度很深，就会执行大量的方法，方法越多java栈的占用空间越大。

本地方法栈角色和java栈类似，只不过它是用来表示执行本地方法的，本地方法栈存放的方法调用本地方法接口，最终调用本地方法库，实现与操作系统、硬件交互的目的。

PC寄存器，说到这里我们的类已经加载了，实例对象、方法、静态变量都去了自己改去的地方，那么问题来了，程序该怎么执行，哪个方法先执行，哪个方法后执行，这些指令执行的顺序就是PC寄存器在管，它的作用就是控制程序指令的执行顺序。

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。

类加载过程:

整个生命周期包括：加载（Loading）、验证（Verification）、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)和卸载(Unloading)7个阶段。其中准备、验证、解析3个部分统称为连接（Linking）



加载、验证、准备、初始化、卸载这5个阶段的顺序是一定的，类的加载过程必须按照这种顺序按部就班地开始，而解析过程则不一定：它在某个情况下可以在初始化阶段之后再开始，这是为了支持Java语言语言的运行时绑定（也叫动态绑定和晚期绑定）;

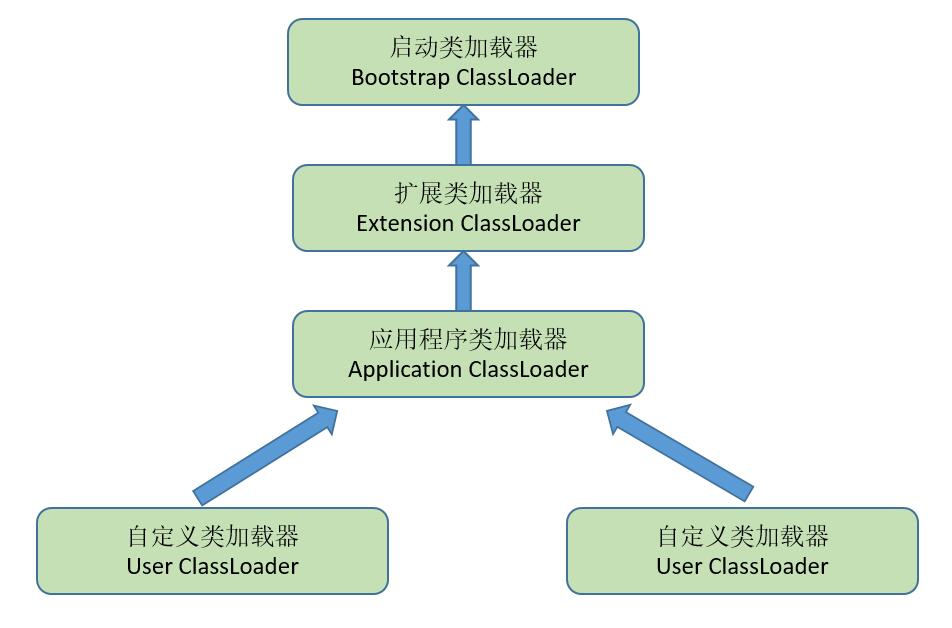
类加载阶段都是互相交叉地混合式进行的，通常是在一个阶段执行的过程中调用、激活另一阶段。

接口和类初始化的区别：当一个类在初始化时，其父类都基本上初始化过了，然而接口在初始化的时候，只有真正用到父接口的时候（如引用接口中定义的常量）才会进行初始化。

即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要类加载器不同，那么这两个类必定是不相等的。

**类加载的双亲委派模型过程**

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。



**Spring知识点：**

1：主要由以下几个模块组成：

Spring Core：核心类库，提供IOC服务；

Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；

Spring AOP：AOP服务；

Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；

Spring ORM：对现有的ORM框架的支持；

Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；

Spring MVC：提供面向Web应用的Model-View-Controller实现。

2：Spring 的优点？

（1）spring属于低侵入式设计，代码的污染极低；

（2）spring的DI机制将对象之间的依赖关系交由框架处理，减低组件的耦合性；

（3）Spring提供了AOP技术，支持将一些通用任务，如安全、事务、日志、权限等进行集中式管理，从而提供更好的复用。

（4）spring对于主流的应用框架提供了集成支持。

3:Spring:IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

Spring的IOC有三种注入方式 ：构造器注入、setter方法注入、根据注解注入;

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

4:BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

1. BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

  ②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

        ③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

5: Spring Bean的生命周期:

（1）实例化Bean：

（2）设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息 以及 通过BeanWrapper提供的设置属性的接口完成依赖注入。

（3）处理Aware接口：

Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean

（4）BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

（5）InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

（6）如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

（7）DisposableBean：

当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

（8）destroy-method：

最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

6: Spring容器中的bean的作用域可以分为5个范围，通过定义中的scope属性来定义：

（1）singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。

（2）prototype：为每一个bean请求提供一个实例。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

7： Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

8：**Spring的自动装配：**

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

（1）no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。

（2）byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。

（3）byType：通过参数的数据类型进行自动装配。

（4）constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。

（5）autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

（6）使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />开启注解装配；

9：@Autowired和@Resource之间的区别

(1) @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false），如果查询的结果不止一个，那么@Autowired会根据名称来查找；

(2) @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入

10: **Spring 框架中都用到了哪些设计模式？**

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

版权声明：本文为博主原创文章，转载请附上博文链接！

11: **Spring事务的实现方式和实现原理:**

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

spring支持编程式事务管理和声明式事务管理两种方式：

编程式事务管理使用TransactionTemplate。

声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

**12：spring的事务传播行为：**

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为：

① PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

③ PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

**13：Spring中的隔离级别：**

① ISOLATION\_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION\_READ\_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION\_READ\_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。

④ ISOLATION\_REPEATABLE\_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。

⑤ ISOLATION\_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

**14：Spring通知有哪些类型：**

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

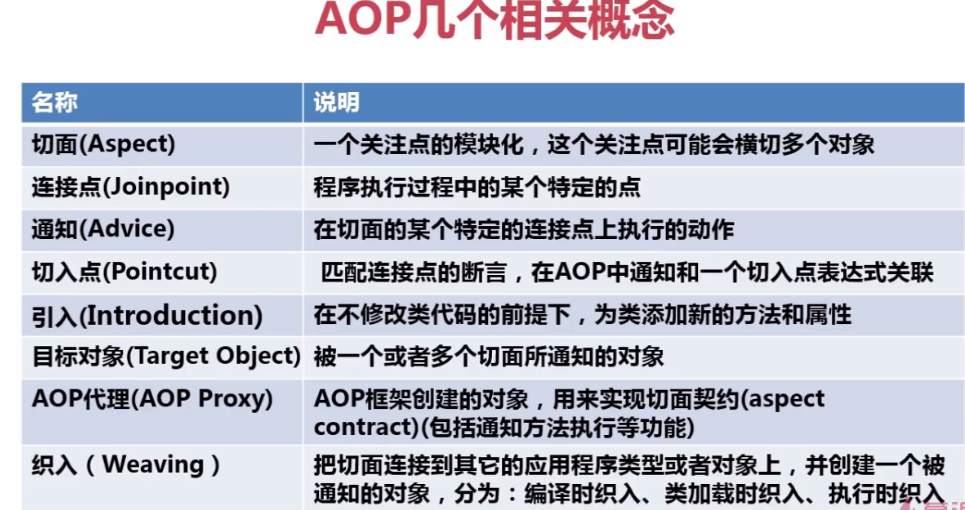
（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。 环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

15：当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用<bean/> 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

16：你可以在Spring中注入一个null 和一个空字符串吗？

可以。

面向切面的编程，或AOP， 是一种编程技术，允许程序模块化横向切割关注点，或横切典型的责任划分，如日志和事务管理。



**Mybatis知识点：**

1：它内部封装了JDBC，开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生态sql，可以严格控制sql执行性能，灵活度高。

2：**#{}和${}的区别是什么：**

#{}是预编译处理，是sql的参数占位符，${}是Properties（配置）文件中的变量占位符，属于静态文本替换；

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理${}时，就是把${}替换成变量的值。

使用#{}可以有效的防止SQL注入，提高系统安全性

SQL注入，就是通过把SQL命令插入到Web[表单](https://baike.baidu.com/item/%E8%A1%A8%E5%8D%95/5380322)提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

#### ****3：这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗****

Mapper接口（即Dao接口）里的方法，是不能重载的，因为是使用 全限名+方法名 的保存和寻找策略；

Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MapperStatement；

Mapper 接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Mapper接口生成代理对象proxy，代理对象会拦截接口方法，转而执行MapperStatement所代表的sql，然后将sql执行结果返回。

**4：Mybatis是如何进行分页的？分页插件的原理是什么？**

 Mybatis使用RowBounds（逻辑分页）对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言（各数据库有各自的方言），添加对应的物理分页语句和物理分页参数。

5：**Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式：**

第一种是使用<resultMap>标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用sql列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

6：**Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复：**

不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复；

**7：Mybatis是否支持延迟加载？如果支持，它的实现原理是什么**

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

**8：MyBatis实现一对一有几种方式?具体怎么操作的？**

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次, 通过在resultMap里面配置association节点配置一对一的类就可以完成；

嵌套查询是先查一个表，根据这个表里面的结果的 外键id，去再另外一个表里面查询数据,也是通过association配置，但另外一个表的查询通过select属性配置。

**9：MyBatis实现一对多有几种方式,怎么操作的？**

 有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在resultMap里面的collection节点配置一对多的类就可以完成；嵌套查询是先查一个表,根据这个表里面的 结果的外键id,去再另外一个表里面查询数据,也是通过配置collection,但另外一个表的查询通过select节点配置

**10：Mybatis的缓存机制:**

1）一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache/> ；

3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

11：**什么是MyBatis的接口绑定？有哪些实现方式：**

接口绑定，就是在MyBatis中任意定义接口,然后把接口里面的方法和SQL语句绑定, 我们直接调用接口方法就可以,这样比起原来了SqlSession提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定，就是在接口的方法上面加上 @Select、@Update等注解，里面包含Sql语句来绑定；另外一种就是通过xml里面写SQL来绑定, 在这种情况下,要指定xml映射文件里面的namespace必须为接口的全路径名。当Sql语句比较简单时候,用注解绑定, 当SQL语句比较复杂时候,用xml绑定,一般用xml绑定的比较多。

**12：使用MyBatis的mapper接口调用时有哪些要求？**

①  Mapper接口方法名和mapper.xml中定义的每个sql的id相同；

②  Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同；

③  Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同；

④  Mapper.xml文件中的namespace即是mapper接口的类路径。

**13：常见标签：**

select|insert|updae|delete|<resultMap>、<parameterType>、<sql>、<include>、<selectKey>|trim|where|set|foreach|if|choose|when|otherwise|bind

#### 14：Mybatis中如何执行批处理

使用BatchExecutor完成批处理；

#### 15、Mybatis都有哪些Executor执行器？它们之间的区别是什么？

答：Mybatis有三种基本的Executor执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。

SimpleExecutor：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。

ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。

BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数。

#### 16、Mybatis映射文件中，如果A标签通过include引用了B标签的内容，请问，B标签能否定义在A标签的后面，还是说必须定义在A标签的前面？

答：虽然Mybatis解析Xml映射文件是按照顺序解析的，但是，被引用的B标签依然可以定义在任何地方，Mybatis都可以正确识别。

原理是，Mybatis解析A标签，发现A标签引用了B标签，但是B标签尚未解析到，尚不存在，此时，Mybatis会将A标签标记为未解析状态，然后继续解析余下的标签，包含B标签，待所有标签解析完毕，Mybatis会重新解析那些被标记为未解析的标签，此时再解析A标签时，B标签已经存在，A标签也就可以正常解析完成了。

# MYSQL知识点

**1：Mysql的四种隔离级别：**

Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

epeatable Read（可重读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

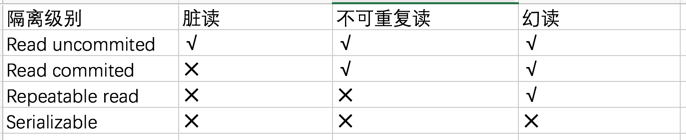
Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。



唯一索引可以为空值，但是主键索引不可以；

对于BTREE这种Mysql默认的索引类型，具有普遍的适用性

正是因为hash表在处理较小数据量时具有无可比拟的素的优势，所以hash索引很适合做缓存（内存数据库），NoSql数据库redis等，都使用了hash索引这种形式;

**2：数据库索引原理**：

加主键的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐， 跟我认知中的「表」很接近。如果给表上了主键，那么表在磁盘上的存储结构就由整齐排列的结构转变成了树状结构，也就是上面说的「平衡树」结构，换句话说，就是整个表就变成了一个索引，就是所谓的「聚集索引」；这就是为什么一个表只能有一个主键， 一个表只能有一个「聚集索引」，因为主键的作用就是把「表」的数据格式转换成「索引（平衡树）」的格式放置。

查找次数：Math.Log(100000000,10)，复杂度为O（logn）；

非聚集索引， 也就是我们平时经常提起和使用的常规索引。

非聚集索引和聚集索引的区别在于， 通过聚集索引可以查到需要查找的数据， 而通过非聚集索引可以查到记录对应的主键值 ， 再使用主键的值通过聚集索引查找到需要的数据。

不管以任何方式查询表， 最终都会利用主键通过聚集索引来定位到数据， 聚集索引（主键）是通往真实数据所在的唯一路径。

然而， 有一种例外可以不使用聚集索引就能查询出所需要的数据， 这种非主流的方法 称之为「覆盖索引」查询， 也就是平时所说的复合索引或者多字段索引查询。当为字段建立索引以后， 字段中的内容会被同步到索引之中， 如果为一个索引指定两个字段， 那么这个两个字段的内容都会被同步至索引之中。当查找的字段在这复合索引里面，所以不再需要通过主键id值来查找数据。

**B-tree（平衡多路查找树）和B+tree：**

B-tree是为磁盘等外存储设备设计的一种平衡查找树；

系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb\_page\_size将页的大小设置为4K、8K、16K，在[MySQL](http://lib.csdn.net/base/mysql)中可通过如下命令查看页的大小：show variables like 'innodb\_page\_size';

而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。



模拟查找关键字29的过程：

1. 根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
2. 比较关键字29在区间（17,35），找到磁盘块1的指针P2。
3. 根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
4. 比较关键字29在区间（26,30），找到磁盘块3的指针P2。
5. 根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
6. 在磁盘块8中的关键字列表中找到关键字29。

发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。

B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

1. 非叶子节点只存储键值信息。
2. 所有叶子节点之间都有一个链指针。
3. 数据记录都存放在叶子节点中



**说明：**

**1.通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点。**

**2. 而且所有叶子节点（即数据节点）之间是一种链式环结构。**

**3.因此可以对B+Tree进行两种查找运算：**

**3.1一种是对于主键的范围查找和分页查找**

**3.2 另一种是从根节点开始，进行随机查找。**

**3：数据库事务的四个特征及含义：**

**原子性**:整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。  
**一致性**:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。  
**隔离性**:隔离状态执行事务，使它们好像是[系统](http://www.2cto.com/os/)在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。  
**持久性**:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

**4：drop,delete与truncate的区别：**

1：一般而言，drop > truncate > delete；

 2：表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

3：drop直接删掉表 truncate删除表中数据，再插入时自增长id又从1开始 delete删除表中数据，可以加where字句

4： DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作；RUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的，执行速度快；

**5：索引类型：**

唯一索引、主键索引和聚集索引

主键索引是唯一索引的特定类型

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

**6：连接种类：**

左联接：

# LINUX知识点

文件系统命令  
Linux的文件系统采用异步的方式，对文件的修改不会马上写入磁盘，一般可以人为的使用sync命令进行手动的数据同步到磁盘。

**五种进程调度算法的总结：**

1、时间片轮转调度算法（RR）：给每个进程固定的执行时间，根据进程到达的先后顺序让进程在单位时间片内执行，执行完成后便调度下一个进程执行，时间片轮转调度不考虑进程等待时间和执行时间，属于抢占式调度。优点是兼顾长短作业；缺点是平均等待时间较长，上下文切换较费时。适用于分时系统。

2、先来先服务调度算法（FCFS）：根据进程到达的先后顺序执行进程，不考虑等待时间和执行时间，会产生饥饿现象。属于非抢占式调度，优点是公平，实现简单；缺点是不利于短作业。

3、优先级调度算法（HPF）：在进程等待队列中选择优先级最高的来执行。

4、多级反馈队列调度算法：将时间片轮转与优先级调度相结合，把进程按优先级分成不同的队列，先按优先级调度，优先级相同的，按时间片轮转。优点是兼顾长短作业，有较好的响应时间，可行性强，适用于各种作业环境。

5、高响应比优先调度算法：根据“响应比=（进程执行时间+进程等待时间）/ 进程执行时间”这个公式得到的响应比来进行调度。高响应比优先算法在等待时间相同的情况下，作业执行的时间越短，响应比越高，满足段任务优先，同时响应比会随着等待时间增加而变大，优先级会提高，能够避免饥饿现象。优点是兼顾长短作业，缺点是计算响应比开销大，适用于批处理系统。

Linux设置进程优先级：

Linux系统进程的优先级取值：-20 到 19，数越大优先级越低。  
 可以通过top命令来查看，NI那一列。

1，top命令。输入r，然后根据提示输入进程ID，再输入优先级数值。

2，renice命令。renice -n 2 -p 3432。-n，后面是优先级的值；-p，是进程号。

**Linux之uptime主机运行时间及平均负载含义**

性能监控命令；输出的信息依此为：系统现在的时间，系统从上次开机到现在运行了多长时间，系统目前有多少登陆用户，系统在1分钟内、5分钟内、15分钟内的平均负载。

注意：load average的3个值大小一般不能大于系统CPU的个数。

         本系统有2个CPU，如果load average的3个值长期大于2时，说明CPU很繁忙，负载很高，可能会影响系统性能；

         如果偶尔大于2，一般不会影响系统性能；

 如果load average的输出值小于CPU的个数，则表示CPU还有空闲的时间。

**为什么要使用多线程?**

1.防止阻塞主线程,提高吞吐量

2,提高资源的利用率

3:需要异步操作的时候使用

进程：是执行中一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。进程是表示资源分配的的基本概念，又是调度运行的基本单位，是系统中的并发执行的单位。

线程：单个进程中执行中每个任务就是一个线程。线程是进程中执行运算的最小单位。

一个线程只能属于一个进程，但是一个进程可以拥有多个线程。多线程处理就是允许一个进程中在同一时刻执行多个任务。

线程是一种轻量级的进程，与进程相比，线程给操作系统带来侧创建、维护、和管理的负担要轻，意味着线程的代价或开销比较小。

**常见命令：**

查看当前进程： ps aux 或者 ps –elf

S表示正在休眠；s表示主进程；Z表示僵尸进程  
执行退出： exit  
查看当前路径： pwd

文件权限修改： chmod

$ chmod u+x file 给 file 的属主增加执行权限  
$ chmod 751 file 给 file 的属主分配读、写、执行(7)的权限，给 file 的所在组分配读、执行(5)的权限，给其他用户分配执行(1)的权限

移动文件和改名：mv

wc 命令 - c 统计字节数 - l 统计行数 - w 统计字数。

滤出关键字附近的日志：cat -n test.log |grep "地形"  得到关键日志的行号

cat -n test.log |tail -n +92|head -n 20

tail -n +92表示查询92行之后的日志

head -n 20 则表示在前面的查询结果里再查前20条记录

一般都是使用 & 在命令结尾来让程序自动运行(后台运行);

查看后台任务：job -l

查看用过的命令列表：history

查看磁盘使用空间：df -hl

查看网络是否连通：netstat

查看 ip 地址及接口信息：ifconfig

查看各类环境变量用什么命令：查看所有 env  
查看某个，如 home： env $HOME

查找命令的可执行文件：whereis或者which

du 显示目录或文件的大小

df 显示每个<文件>所在的文件系统的信息，默认是显示所有文件系统。

一页一页地查看一个大文件的内容：# cat file\_name.txt | more

查看进程动态信息（cpu负载，内存等）：top

free命令是一个快速查看内存使用情况

查看磁盘信息：fdisk –l

查看开放的端口：netstat -lnp

查看当前硬盘使用情况（挂载目录）：df –h（以G为单位）；df –m（以M为单位）

磁盘挂载在/home/data/ 下：# mount /dev/sdb1 /home/data/

查看 ip 地址及接口信：ifconfig

buffer和cache如何区别

1. Cache：缓存区，是高速缓存，是位于CPU和主内存之间的容量较小但速度很快的存储器，因为CPU的速度远远高于主内存的速度，CPU从内存中读取数据需等待很长的时间，而  Cache保存着CPU刚用过的数据或循环使用的部分数据，这时从Cache中读取数据会更快，减少了CPU等待的时间，提高了系统的性能。

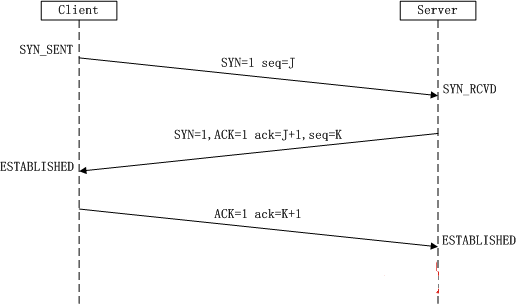
 Cache并不是缓存文件的，而是缓存块的(块是I/O读写最小的单元)；Cache一般会用在I/O请求上，如果多个进程要访问某个文件，可以把此文件读入Cache中，这样下一个进程获取CPU控制权并访问此文件直接从Cache读取，提高系统性能。

2.Buffer：缓冲区，用于存储速度不同步的设备或优先级不同的设备之间传输数据；通过buffer可以减少进程间通信需要等待的时间，当存储速度快的设备与存储速度慢的设备进行通信时，存储慢的数据先把数据存放到buffer，达到一定程度存储快的设备再读取buffer的数据，在此期间存储快的设备CPU可以干其他的事情。

Buffer：一般是用在写入磁盘的，例如：某个进程要求多个字段被读入，当所有要求的字段被读入之前已经读入的字段会先放到buffer中。

# 网络知识点

所谓三次握手（Three-Way Handshake）即建立TCP连接，是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发;



（1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。

  （2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack (number )=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。

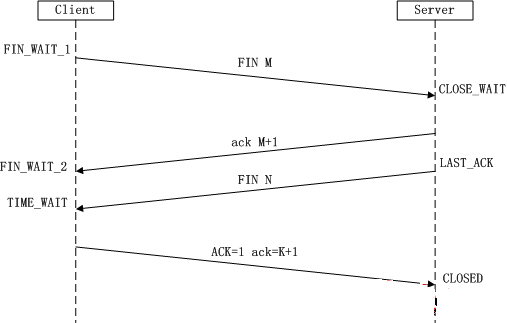
  （3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

SYN攻击：

  在三次握手过程中，Server发送SYN-ACK之后，收到Client的ACK之前的TCP连接称为半连接（half-open connect），此时Server处于SYN\_RCVD状态，当收到ACK后，Server转入ESTABLISHED状态。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将长时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击时一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了，使用如下命令可以让之现行：

  #netstat -nap | grep SYN\_RECV

所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，



由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

 （1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

  （2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

 （3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

  （4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。

为什么建立连接是三次握手，而关闭连接却是四次挥手呢:

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

**TCP/IP四层模型：**

1.链路层（数据链路层/网络接口层）：包括操作系统中的设备驱动程序、计算机中对应的网络接口卡

2.网络层（互联网层）：处理分组在网络中的活动，比如分组的选路。

3.运输层：主要为两台主机上的应用提供端到端的通信。

4.应用层：负责处理特定的应用程序细节。

**TCP和UDP有什么区别**

TCP是传输控制协议，提供的是面向连接的，可靠地字节流服务

UDP是用户数据报协议，是无连接的。因为无连接，而且没有超时重发机制，所以UDP传输速度很快。

TCP保证数据按序到达，提供流量控制和拥塞控制，在网络拥堵的时候会减慢发送字节数，而UDP不管网络是否拥堵。

TCP是连接的，所以服务是一对一服务，而UDP可以1对1，也可以1对多（多播），也可以多对多。

HTTP1.0每请求一个文档就要建立TCP连接，有几次握手的时间花销，如果一个主页上有很多链接的对象需要依次进行连接，每次连接下载都要消耗这些开销。

HTTP1.1采用持续连接。所谓持续连接就是服务器在发送响应后仍然在一段时间内保持这条连接。使得后序的请求和响应报文都在这条连接上进行。

# 分布式知识点：

Hadoop无非就是：HDFS(文件系统)，yarn(任务调配)，mapReduce(编程模型，大数据并行运算)；

**Hdfs：**HDFS的文件分布在集群机器上，同时提供副本进行容错及可靠性保证。例如客户端写入读取文件的直接操作都是分布在集群各个机器上的，没有单点性能压力。

HDFS的Block块比一般单机文件系统大得多，默认为128M。HDFS的文件被拆分成block-sized的chunk块，chunk作为独立单元存储。比Block小的文件不会占用整个Block，只会占据实际大小。

Block作为容错和高可用机制中的副本单元，即以Block为单位进行复制

整个HDFS集群由Namenode和Datanode构成master-worker（主从）模式。Namenode负责构建命名空间，管理文件的元数据等，而Datanode负责实际存储数据，负责读写工作。

HDFS 采用Master/Slave的架构来存储数据，这种架构主要由四个部分组成，分别为HDFS Client、NameNode、DataNode和Secondary NameNode：

* Client：就是客户端，文件上传 HDFS 的时候，Client 将文件切分成 一个一个的Block，然后进行存储；与 NameNode 交互，获取文件的位置信息；与 DataNode 交互，读取或者写入数据；Client 提供一些命令来管理 HDFS，比如启动或者关闭HDFS；Client 可以通过一些命令来访问 HDFS
* NameNode：就是 master，它是一个主管、管理者；管理 HDFS 的名称空间；管理数据块（Block）映射信息；配置副本策略；处理客户端读写请求；存放文件系统树及所有文件、目录的元数据；单点故障；
* DataNode：就是Slave。NameNode 下达命令，DataNode 执行实际的操作；存储实际的数据块；执行数据块的读/写操作。
* Secondary NameNode：并非 NameNode 的热备。当NameNode 挂掉的时候，它并不能马上替换 NameNode 并提供服务；辅助 NameNode，分担其工作量；在紧急情况下，可辅助恢复 NameNode；定期合并 fsimage和fsedits，并推送给NameNode

**YARN：**职能就是将资源调度和任务调度分开

YARN主要由ResourceManager、NodeManager、ApplicationMaster和Container等几个[组件](http://www.2cto.com/kf/all/zujian/)构成；

ResourceManager做的事情是负责协调集群上计算资源的分配。调度、启动每一个 Job 所属的 ApplicationMaster、另外监控 ApplicationMaster 的存在情况；

NodeManager 功能比较专一，根据要求启动和监视集群中机器的计算容器container。负责 Container 状态的维护，并向 RM 保持心跳汇报该节点资源使用情况。

ApplicationMaster 负责一个 Job 生命周期内的所有工作。注意每一个Job都有一个 ApplicationMaster。它和MapReduce任务一样在容器中运行。AM通过与RM交互获取资源，然后然后通过与NM交互，启动计算任务。

Container容器是由ResourceManager进行统一管理和分配的

Application在执行一半时ResourceManager宕机，另一个ResourceManager不能继续提供任务的执行服务，因为中间数据太多，Hadoop未实现这种任务调度的切换。

**MapReduce：**

map阶段对切分好的数据进行并行处理，处理结果传输给reduce，由reduce函数完成最后的汇总。

1、Map任务处理

　　1.1 读取HDFS中的文件。每一行解析成一个<k,v>。每一个键值对调用一次map函数。                <0,hello you>   <10,hello me>

　　1.2 覆盖map()，接收1.1产生的<k,v>，进行处理，转换为新的<k,v>输出。　　　　　　　　　　<hello,1> <you,1> <hello,1> <me,1>

　　1.3 对1.2输出的<k,v>进行分区。默认分为一个区。详见《[Partitioner](http://www.cnblogs.com/ahu-lichang/p/6657895.html)》

　　1.4 对不同分区中的数据进行排序（按照k）、分组。分组指的是相同key的value放到一个集合中。　排序后：<hello,1> <hello,1> <me,1> <you,1>  分组后：<hello,{1,1}><me,{1}><you,{1}>

　　1.5 （可选）对分组后的数据进行归约。详见《[Combiner](http://www.cnblogs.com/ahu-lichang/p/6657572.html)》

2、Reduce任务处理

　　2.1 多个map任务的输出，按照不同的分区，通过网络copy到不同的reduce节点上。（shuffle）详见《[shuffle过程分析](http://www.cnblogs.com/ahu-lichang/p/6665242.html)》

　　2.2 对多个map的输出进行合并、排序。覆盖reduce函数，接收的是分组后的数据，实现自己的业务逻辑，　<hello,2> <me,1> <you,1>

　　　　处理后，产生新的<k,v>输出。

　　2.3 对reduce输出的<k,v>写到HDFS中。

# 算法：

**二叉树:**

有左右之分，并且次序不能任意颠倒。

即使树中某结点只有一棵子树，也要区分它是左子树还是右子树。

二叉排序树又叫二叉查找树或者二叉搜索树，它首先是一个二叉树，而且必须满足下面的条件：

1）若左子树不空，则左子树上所有结点的值均小于它的根节点的值；

2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值

3）左、右子树也分别为二叉排序树

4）没有键值相等的节点（？可能是因为不好处理键值相等的节点到底是左节点还是右节点吧）

二叉树链表：

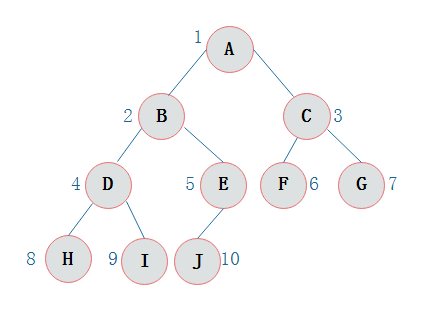
二叉树遍历：是指从二叉树的根结点出发，按照某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次，且仅被访问一次。

访问次序分为四种：前序遍历，中序遍历，后序遍历，层序遍历

前序遍历：通俗的说就是从二叉树的根结点出发，当第一次到达结点时就输出结点数据，按照先向左在向右的方向访问。

从根结点出发，则第一次到达结点A，故输出A;  
继续向左访问，第一次访问结点B，故输出B；  
按照同样规则，输出D，输出H；  
当到达叶子结点H，返回到D，此时已经是第二次到达D，故不在输出D，进而向D右子树访问，D右子树不为空，则访问至I，第一次到达I，则输出I；  
I为叶子结点，则返回到D，D左右子树已经访问完毕，则返回到B，进而到B右子树，第一次到达E，故输出E；  
向E左子树，故输出J；  
按照同样的访问规则，继续输出C、F、G；

输出顺序：**ABDHIEJCFG**



**中序遍历：**就是从二叉树的根结点出发，当第二次到达结点时就输出结点数据，按照先向左在向右的方向访问。

从根结点出发，则第一次到达结点A，不输出A，继续向左访问，第一次访问结点B，不输出B；继续到达D，H；  
到达H，H左子树为空，则返回到H，此时第二次访问H，故输出H；  
H右子树为空，则返回至D，此时第二次到达D，故输出D；  
由D返回至B，第二次到达B，故输出B；  
按照同样规则继续访问，输出J、E、A、F、C、G；

**输出顺序：HDIBJEAFCG**

**后序遍历：**就是从二叉树的根结点出发，当第三次到达结点时就输出结点数据，按照先向左在向右的方向访问。

从根结点出发，则第一次到达结点A，不输出A，继续向左访问，第一次访问结点B，不输出B；继续到达D，H；  
到达H，H左子树为空，则返回到H，此时第二次访问H，不输出H；  
H右子树为空，则返回至H，此时第三次到达H，故输出H；  
由H返回至D，第二次到达D，不输出D；  
继续访问至I，I左右子树均为空，故第三次访问I时，输出I；  
返回至D，此时第三次到达D，故输出D；  
按照同样规则继续访问，输出J、E、B、F、G、C，A；

输出顺序：**HIDJEBFGCA**

层次遍历: 层次遍历就是按照树的层次自上而下的遍历二叉树

输出顺序：**ABCDEFGHIJ**

/\*二叉树的前序遍历递归算法\*/

void PreOrderTraverse(BiTree T)

{

if(T==NULL)

return;

printf("%c", T->data); /\*显示结点数据，可以更改为其他对结点操作\*/

PreOrderTraverse(T->lchild); /\*再先序遍历左子树\*/

PreOrderTraverse(T->rchild); /\*最后先序遍历右子树\*/

}

/\*二叉树的中序遍历递归算法\*/

void InOrderTraverse(BiTree T)

{

if(T==NULL)

return;

InOrderTraverse(T->lchild); /\*中序遍历左子树\*/

printf("%c", T->data); /\*显示结点数据，可以更改为其他对结点操作\*/

InOrderTraverse(T->rchild); /\*最后中序遍历右子树\*/

}

/\*二叉树的后序遍历递归算法\*/

void PostOrderTraverse(BiTree T)

{

if(T==NULL)

return;

PostOrderTraverse(T->lchild); /\*先后序遍历左子树\*/

PostOrderTraverse(T->rchild); /\*再后续遍历右子树\*/

printf("%c", T->data); /\*显示结点数据，可以更改为其他对结点操作\*/

}

**Hash去重：**

遇到两个对象数组A,B,找到A中B不存在的对象，并保存在一个数组中：

function unique2(arr){

//i遍历arr,同时创建两个空数组result和hash

for(var i=0,result=[],hash=[];

i<arr.length;

i++){

//如果hash中以当前元素为key的元素是undefined

if(hash[arr[i]]===undefined){

//将当前元素追加到result结尾

result[result.length]=arr[i];

//在hash中添加一个新元素: key为当前元素值,值为true

hash[arr[i]]=true;

}

}//(遍历结束)

return result;//返回result

}

HashSet去重原理差不多：它的add()方法实际上调用的是HashMap中的put()方法，把要添加进HashSet中的元素当做key存入，而value则是一个固定值：一个Object类对象。

先用hashCode()方法获得传入元素的哈希值，在集合中查找是否包含哈希值相同的元素，如果相同，则继续进行比较它们地址值，一般地址值都是不相同的，所以最后会用equals()方法比较对象内的属性值。 比较结果全为false就存入，如果比较结果有true则不存.

[两个栈实现一个队列，两个队列实现一个栈](https://www.cnblogs.com/tracyhan/p/5490775.html)

1. 两个栈实现一个队列

将stack1作为存储空间，将stack2作为临时缓冲区，入队时，直接压入stack1中

 出队时，判断stack2是否为空，如果stack2为空，则将stack1中的元素倒入stack2中，否则直接弹出stack2中的元素

//入队操作

void EnQueue(stack<int> &s1,stack<int> &s2,int m)

{

s1.push(m);

}

//出队操作

void DeQueue(stack<int> &s1,stack<int> &s2,int &m)

{

if (s2.empty())

{

int p = s1.size();

for (int i=0;i<p;i++)

{

s2.push(s1.top());

s1.pop();

}

}

m = s2.top();

s2.pop();

}

1. 两个队列实现一个栈

将queue1用作进栈出栈，queue2作为一个中转站

入栈时，直接压入queue1中；

出栈时，先将queue1中的元素除最后一个元素外依次出队列，并压入队列queue2中，将留在queue1中的最后一个元素出队列即为出栈元素，最后还要把queue2中的元素再次压入queue1中

//进栈操作

void stackpush(queue<int> &q1,queue<int> &q2,int m)

{

q1.push(m);

}

//出栈操作

void stackpop(queue<int> &q1,queue<int> &q2,int &m)

{

int p = q1.size();

for (int i=0;i<p-1;i++)

{

q2.push(q1.front());

q1.pop();

}

m = q1.front();

q1.pop();

int l = q2.size();

for (int j = 0;j<l;j++)

{

q1.push(q2.front());

q2.pop();

}

}

**不使用临时变量对两个数值进行交换的方法：**

a=a+b;

b=a-b;

a=a-b;

if(a!=b){

a^=b;

b^=a;

a^=b;

}

原理：a=a^b;b=b^a=b^a^b=a^0=a;a=a^b=a^a^b=a^b=b;

**排序：**

双轴快排的算法：双轴快排是基于两个轴来进行比较，跟普通的选择一个点来作为轴点的快排是有很大的区别的，双轴排序利用了区间相邻的特性，对原本的快排进行了效率上的提高；

**冒泡排序：**

private void bubbleSort(int[] arr) {

int temp = 0;

for (int i = 0; i < arr.length - 1; i++) {

//外层循环控制排序趟数

for (int j = 0; j < arr.length - i - 1; j++) {

//内层循环控制每一趟排序多少次

if (arr[j] > arr[j + 1]) {

temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

System.out.println(Arrays.toString(arr));

//每进行一趟排序，就会少比较一次，因为每进行一趟排序都会找出一个最大值,沉到最底下不再比较

}

**直接插入排序：**

private void straightInsertSort(int[] arr) {

int tmp = -1;

int sortedIndex = 0;

//假定第一个元素被放到了正确的位置上

//这样，仅需遍历1 --- n-1次

for (int i = 1; i < arr.length; i++) {

tmp = arr[i];

sortedIndex = i - 1;

while (sortedIndex >= 0 && arr[sortedIndex] > tmp) {

arr[sortedIndex + 1] = arr[sortedIndex];//将i之前的数组向后移动

sortedIndex--;

}

arr[sortedIndex + 1] = tmp;

}

System.out.println(Arrays.toString(arr));

}

将待排序序列划分为有序区和无序区

当数据项数量不多的时候，插入排序可以很好的完成工作。

当数据项基本有序的时候，插入排序具有很高的效率。

**希尔排序：**

private void shellSort(int[] arr) {

//增量

int step = arr.length / 2;

int temp = 0;

int j = 0;

while (step > 0) {

for (int i = step; i < arr.length; i++) {//从第setp个元素开始，将其与之前的元素相比

if (arr[i] < arr[i - step]) {//使用直接插入排序

temp = arr[i];

j = i - step;

while (j >= 0 && arr[j] > temp) {

arr[j + step] = arr[j];

j = j - step;

}

arr[j + step] = temp;

}

}

step /= 2;

}

System.out.println(Arrays.toString(arr));

}

希尔排序(Shell Sort)是插入排序的一种。也称缩小增量排序，是直接插入排序算法的一种更高效的改进版本。

希尔排序是把记录按下标的一定增量分组（比如下标为1,3,5,7…的分为一组），对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

**选择排序：**

private void selectSort(int[] arr) {

int temp = 0;

int minIndex = -1;

for (int i = 0; i < arr.length; i++) {//i是无序区第一个位置

minIndex = i;

for (int j = i; j < arr.length - 1; j++) {//无序区

if (arr[minIndex] > arr[j + 1]) {

minIndex = j + 1;

}

}

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

//将无序区第一个位置于最小记录值交换，有序区就增加了一个元素，同时无序区减少了一个元素

}

System.out.println(Arrays.toString(arr));

}

初始时，有序区为空，无序区是整个待排序序列

**堆排序:**

//调整节点,arr:待排序的序列；k:待调整节点的索引值；m:堆的最大索引值

private void adjust(int[] arr, int k, int m) {

int tmp;

int i = k;//要调整的节点

int j = 2 \* k + 1;//调整节点的左孩子

while (j < m) {//最新的调整节点的左孩子索引值不能超过堆的最大索引

if (j < m && arr[j] < arr[j + 1]) {

++j;//得到左右孩子中的最大值节点

}

//如果孩子节点大于根节点则调换彼此位置

//如果arr[i]>arr[j],说明这个以节点为根的树已经满足堆的定义，算法结束

if (arr[i] > arr[j]) {

break;

} else {

tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i = j;//继续节点的调整

j = 2 \* i + 1;

}

}

}

private void heapSort(int[] arr) {

int m = arr.length;

//创建初始堆

int lastEleIndex = m - 1;//无序区的最后一个元素的索引值

int tmp;

//从最下侧的非叶子节点开始，向上创建初始堆

for (int i = m / 2 - 1; i >= 0; i--) {

this.adjust(arr, i, lastEleIndex);

}

//重建堆,每趟将堆顶节点与堆无序区最后一个元素交换，然后再调整新堆顶节点。每趟完成之后完成了一个元素的排序

for (int i = 0; i < m; i++) {

tmp = arr[0];

arr[0] = arr[lastEleIndex];

arr[lastEleIndex] = tmp;

adjust(arr, 0, --lastEleIndex);

}

System.out.println(Arrays.toString(arr));

}

堆排序是基于选择排序的改进，目的是较少比较次数。一趟选择排序中，仅保留了最小值，而堆排序排序不仅保留最小值，还把较小值保留下来，减少了比较小次数。

堆排序可以分为两个过程：构建初始堆和重建堆。

**归并排序:**

private void mergeArray(int[] arr, int left, int mid, int right, int[] temp) {

int i = left;//左序列指针

int j = mid + 1;//右序列指针

int t = 0;//临时数组指针

while (i <= mid && j <= right) {

if (arr[i] <= arr[j]) {

temp[t++] = arr[i++];

} else {

temp[t++] = arr[j++];

}

}

while (i <= mid) {//将左边剩余元素填充进temp中

temp[t++] = arr[i++];

}

while (j <= right) {//将右序列剩余元素填充进temp中

temp[t++] = arr[j++];

}

t = 0;

//将temp中的元素全部拷贝到原数组中

while (left <= right) {

arr[left++] = temp[t++];

}

}

private void mergeSort(int[] arr) {

int[] temp = new int[arr.length];

mergeSort2(arr, 0, arr.length - 1, temp);

System.out.println(Arrays.toString(arr));

}

private void mergeSort2(int[] arr, int left, int right, int[] temp) {

if (left < right) {

int mid = (left + right) / 2;

//左边归并排序，使得左子序列有序；

mergeSort2(arr, left, mid, temp);

//右边归并排序，使得右子序列有序；

mergeSort2(arr, mid + 1, right, temp);

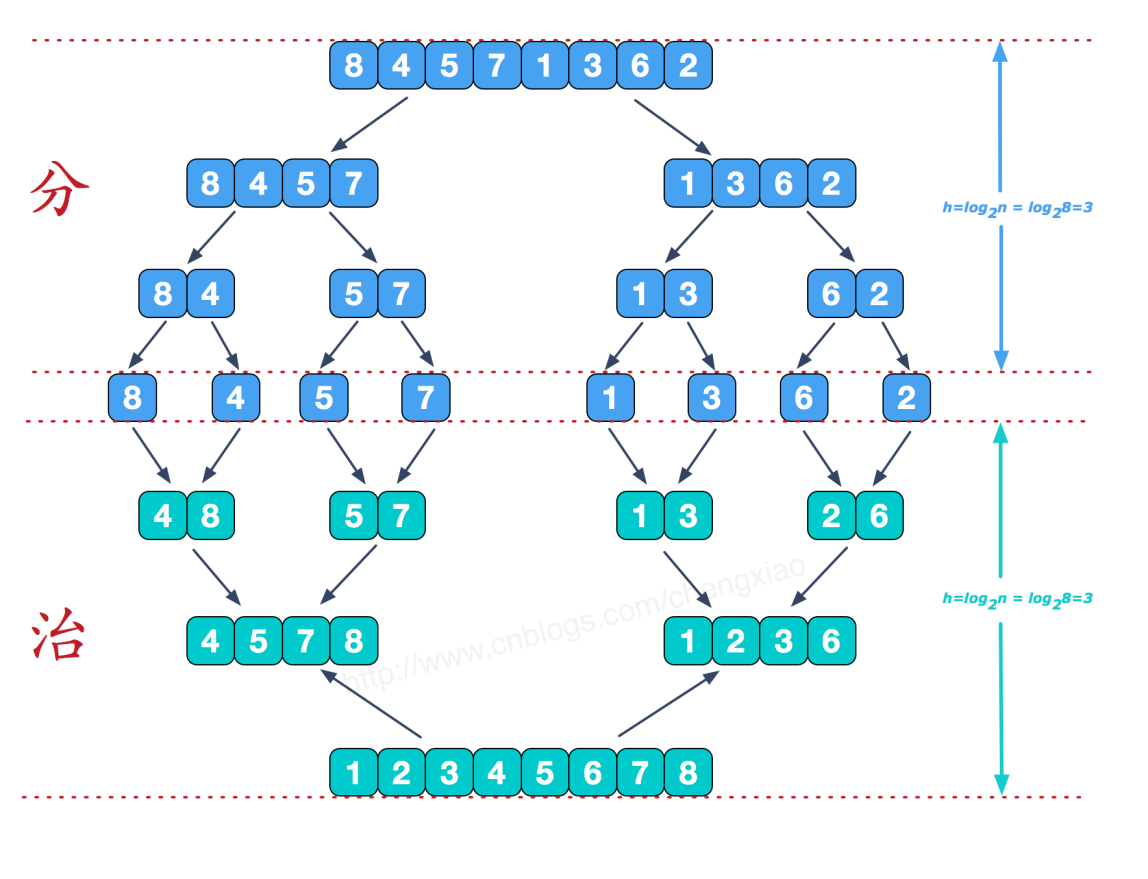
mergeArray(arr, left, mid, right, temp);

}

}

将两个或多个有序序列合并成一个有序序列的过程，归并排序就是将若干有序序列逐步归并，最终形成一个有序序列的过程。

并不是使用两个数组进行合并，而是通过数组索引的形式“描述”两个待合并的有序表



**快速排序：**

一种划分交换排序。它采用了一种分治的策略，通常称其为分治法

//分区

private int part(int[] arr, int low, int high) {

int i = low;//左游标，选择数组第一个元素作为基准值

int j = high;//右游标

//i==j表示分区过程的结束

while (i < j) {

//从右侧向左扫描，找到小于基准值arr[i]的元素,使用i<j防止数组越界(原数组可能升序排列)

while (i < j && arr[j] >= arr[i]) {

--j;

}

//上述循环结束可能是因为i==j，原数组升序排列导致。如果是这样的话，分区就可以结束了

if (i < j) {

arr[i] ^= arr[j];

arr[j] ^= arr[i];

arr[i] ^= arr[j];

++i;

}

//从左侧向右扫描，找到大于基准值arr[j]的元素，使用i<j防止数组越界(只是此时原数组可能降序排列)

while (i < j && arr[i] < arr[j]) {

i++;

}

//上述循环结束可能是因为i==j，原数组降序排列导致。如果是这样的话，分区就可以结束了

if (i < j) {

arr[i] ^= arr[j];

arr[j] ^= arr[i];

arr[i] ^= arr[j];

--j;

}

}

return i;

}

private void quickSort(int[] arr) {

this.quickSort2(arr, 0, arr.length - 1);

System.out.println(Arrays.toString(arr));

}

private void quickSort2(int[] arr, int low, int high) {

if (low < high) {

int mid = part(arr, low, high);

quickSort2(arr, low, mid - 1);

quickSort2(arr, mid + 1, high);

}

}

分区分为三个步骤：

1.在数组中，选择一个元素作为“基准”(pivot),一般选择第一个元素作为基准元素。设置两个游标i和j，初始时i指向数组首元素，j指向尾元素。  
2.从数组最右侧向前扫描，遇到小于基准值的元素停止扫描，将两者交换，然后从数组左侧开始扫描，遇到大于基准值的元素停止扫描，同样将两者交换。

3.i==j时分区完成，否则转2

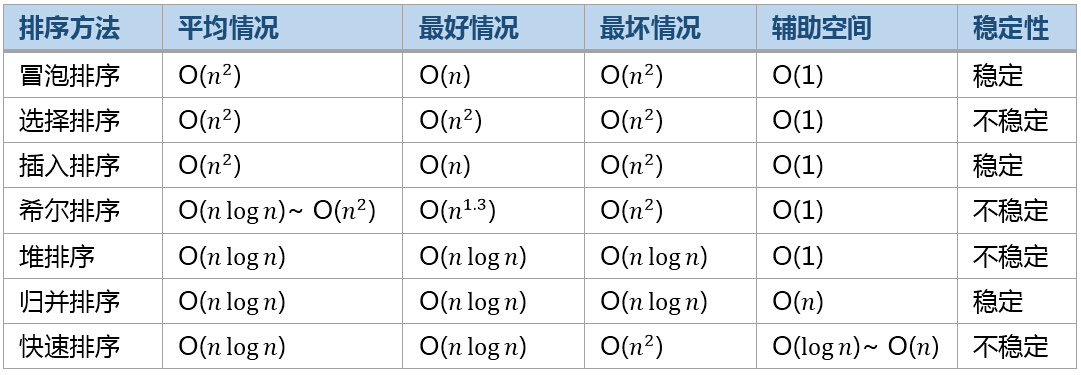
每次分区之后，基准值所处的位置(storeIndex)就是最终排序后它的位置，并且，一次分区之后，数据集一分为二，在分别对两侧的新分区进行分区，直至最后每个子数据集中只剩下一个元素

快速排序适用于待排序记录个数很多且分布随机的情况，并且快拍是目前内排序中排序算法最好的一种。

排序算法的稳定性是指排序前后具有相同关键字的记录，相对顺序保持不变。形式化的定义就是，排序之前有ri=rj,ri在rj之前，而在排序之后ri仍然在rj之前，就说这种算法是稳定的。

一般在不要求稳定性的场景下，使用快速排序就可以了但是，当我们每次进行分区划分时，选择的基准元素都是最小(排序目的是升序)/最大(排序目的是降序)，这是快速排序的最坏情况，时间复杂度上升为O(n^2)，这时可以使用堆排序和归并排序。在n较大时，相对堆排序来说，归并排序使用时间较少，但辅助空间较多(合并两个有序序列为一个有序序列)。

序列基本有序且n较小时，插入排序具有很高的排序效率。因此常将它和其他的排序方法，如快速排序、归并排序等结合在一起使用。



**斐波那契数列计算：**

**private** **long** bb(**int** i) **throws** Exception {

// if (i <= 1) {

// return i;

// }

// return bb(i - 1) + bb(i - 2);

// if (i < 3) {

// return 1;

// } else {

// return this.bb(i - 1) + this.bb(i - 2);

//

// }

**if** (i < 3) {

**return** 1;

} **else** {

**long** f1 = 0;

**long** f2 = 1;

**long** f3 = 0;

**for** (**int** j = 1; j < i; j++) {

f3 = f1 + f2;

f1 = f2;

f2 = f3;

**if** (f3 > 10000000) {

f3 -= 10000000;

}

**if** (f2 > 10000000) {

f2 -= 10000000;

}

**if** (f1 > 10000000) {

f1 -= 10000000;

}

}

**return** f3;

}

}

判断是否为2的整次幂看转化后的二进制最后一位是否为0；

**计算没有括号的算式（只有加减，没有括号，随机加括号求最小结果）：**

**private** **int** exe(String n) {

n = n.replaceAll("\\+", " + ");

n = n.replaceAll("-", " - ");

String[] b = n.split(" ");

**int** numIndex = 0;

**int** symIndex = 0;

Map<Integer, Integer> numMap = **new** HashMap<Integer, Integer>();

Map<Integer, String> symMap = **new** HashMap<Integer, String>();

**for** (**int** i = 0; i < b.length; i++) {

**try** {

**int** s = Integer.*parseInt*(b[i]);

numMap.put(numIndex, s);

numIndex++;

} **catch** (Exception e) {

symMap.put(symIndex, b[i]);

symIndex++;

}

}

**if** (numMap.size() <= 1) {

**return** numMap.get(0);

}

List<Integer> symKeyList = **new** LinkedList<Integer>(symMap.keySet());

Integer[] arr = **new** Integer[symKeyList.size()];

Integer[] symArr = symKeyList.toArray(arr);

Integer[] sortSymArr = **this**.skySort(symArr, symMap);

**return** **this**.getResult(numMap, symMap, **new** LinkedList<Integer>(Arrays.*asList*(sortSymArr)));

}

// numMap的value为算式的数值，key为数值下标

// symMap的value为算式的符号，key为算式下标

// sortList是符号下标的集合，它的顺序为符号的执行顺序

**private** **int** getResult(Map<Integer, Integer> numMap, Map<Integer, String> symMap, List<Integer> sortList) {

**if** (numMap.size() <= 1) {

**return** numMap.get(0);

}

// 第一个顺序执行的符号

**int** firstSymIndex = sortList.get(0);

String firstSym = symMap.get(firstSymIndex);

**if** (firstSym.equals("+")) {

**int** i = numMap.get(firstSymIndex) + numMap.get(firstSymIndex + 1);

numMap.put(firstSymIndex, i);

} **else** {

**int** i = numMap.get(firstSymIndex) - numMap.get(firstSymIndex + 1);

numMap.put(firstSymIndex, i);

}

numMap.remove(firstSymIndex + 1);

symMap.remove(firstSymIndex);

sortList.remove(0);

// 重排，使其重新从0下标开始

Map<Integer, Integer> midNumMap = **new** HashMap<Integer, Integer>();

Map<Integer, String> midSymMap = **new** HashMap<Integer, String>();

List<Integer> midList = **new** LinkedList<Integer>();

**int** i = 0;

**int** j = 0;

**int** k = 0;

**for** (Integer integer : numMap.values()) {

midNumMap.put(i++, integer);

}

**for** (String string : symMap.values()) {

midSymMap.put(j++, string);

}

// 第一个顺序执行的符号后面的符号下标整体前移一位，顺序未变

**for** (Integer integer : sortList) {

**if** (integer > firstSymIndex) {

midList.add(k++, integer - 1);

} **else** {

midList.add(k++, integer);

}

}

**return** **this**.getResult(midNumMap, midSymMap, midList);

}

// 保证所有加法在所有减法之前执行（加法或者减法内部之间的顺序不能变）就可以得出算式最小值

**private** Integer[] skySort(Integer[] arr, Map<Integer, String> symMap) {

**int** tmp = -1;

**int** index = 0;

**for** (**int** i = 0; i < arr.length; i++) {

**if** (symMap.get(i).equals("-")) {

index = i;

**break**;

}

}

**int** j = 0;

**for** (**int** i = 0; i < arr.length; i++) {

**if** (symMap.get(i).equals("+") && i > index) {

j = i;

**while** (j > index) {

tmp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = tmp;

j--;

}

index++;

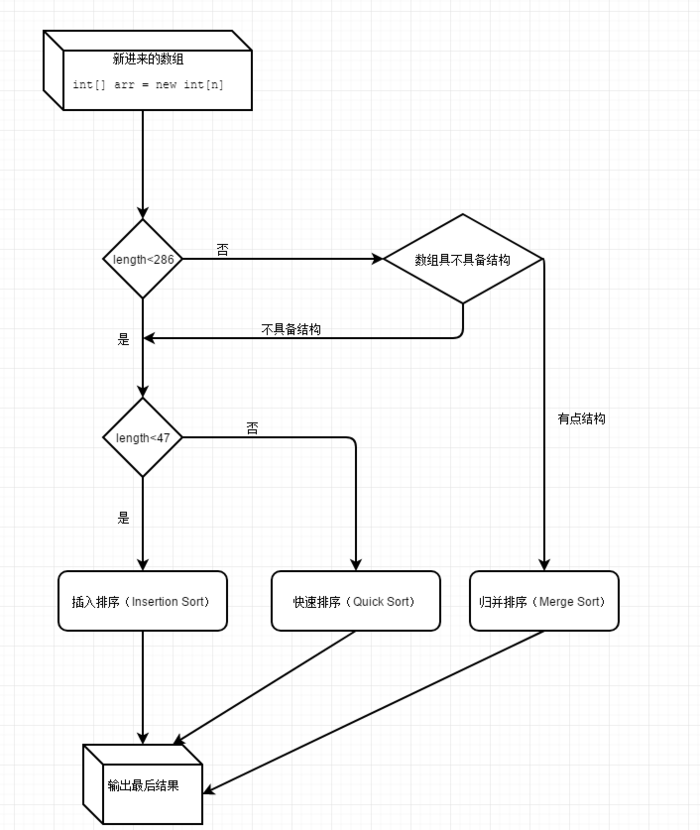
}

}

**return** arr;

}

**jdk8中的数组排序算法Arrays.sort()：**



数组具不具备结构：

**写一个函数，输入int型，返回整数逆序后的字符串。如：输入123，返回“321”。 要求必须用递归，不能用全局变量，输入必须是一个参数，必须返回字符串：**

**private** String intToString(**int** n) {

String tem = "";

**if** (n / 10 == 0) {

tem = tem + n;

} **else** {

tem = tem + n % 10 + **this**.intToString(n / 10);

}

**return** tem;

}

**红黑树：**

红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一倍(普通的二叉查找树会出现某个腿特别长的情况)。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：

每个节点要么是红色，要么是黑色。

根节点必须是黑色

每个叶子的节点都是黑色的空节点（null）

红色节点不能连续（也即是，红色节点的孩子和父亲都不能是红色）。

对于每个节点，从该点至null（树尾端）的任何路径，都含有相同个数的黑色节点。

//红黑树调整函数fixAfterInsertion()

private void fixAfterInsertion(Entry<K,V> x) {

x.color = RED;

while (x != null && x != root && x.parent.color == RED) {

if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {

Entry<K,V> y = rightOf(parentOf(parentOf(x)));

if (colorOf(y) == RED) {//如果y为null，则视为BLACK, 情况1

setColor(parentOf(x), BLACK);

setColor(y, BLACK);

setColor(parentOf(parentOf(x)), RED);

x = parentOf(parentOf(x));

} else {

if (x == rightOf(parentOf(x))) {//情况2

x = parentOf(x);

rotateLeft(x);

}

setColor(parentOf(x), BLACK); //情况3

setColor(parentOf(parentOf(x)), RED);

rotateRight(parentOf(parentOf(x)));

}

} else {

Entry<K,V> y = leftOf(parentOf(parentOf(x)));

if (colorOf(y) == RED) {//情况4

setColor(parentOf(x), BLACK);

setColor(y, BLACK);

setColor(parentOf(parentOf(x)), RED);

x = parentOf(parentOf(x));

} else {

if (x == leftOf(parentOf(x))) {//情况5

x = parentOf(x);

rotateRight(x);

}

setColor(parentOf(x), BLACK); //情况6

setColor(parentOf(parentOf(x)), RED);

rotateLeft(parentOf(parentOf(x)));

}

}

}

root.color = BLACK;

}

# SQL优化：

**in和exists区别：**

select \* from student s where s.stuid in(select stuid from score ss where ss.stuid = s.stuid)

select \* from student s where EXISTS(select stuid from score ss where ss.stuid = s.stuid)

exists()适合内表比外表数据大的情况；使用时一定要注意索引的影响，利用索引会使数据库查询有惊人的性能提升；

当内表数据与外表数据一样大时,in与exists效率差不多,可任选一个使用；

In（A为外表，B为内表）:

A表有10000条记录,B表有1000000条记录,那么最多有可能遍历10000\*1000000次,效率很差.  
A表有10000条记录,B表有100条记录,那么最多有可能遍历10000\*100次,遍历次数大大减少,效率大大提升.

Exists A为外表，B为内表）：

A表有10000条记录,B表有1000000条记录,那么exists()会执行10000次去判断A表中的id是否与B表中的id相等.

A表有10000条记录,B表有100000000条记录,那么exists()还是执行10000次,因为它只执行A.length次,可见B表数据越多,越适合exists()发挥效果.

not exists的执行顺序是：在表中查询，是根据索引查询的，如果存在就返回true，如果不存在就返回false，不会每条记录都去查询。  
之所以要多用not exists，而不用not in，也就是not exists查询的效率远远高与not in查询的效率。

Mysql的In查询会外表使用全表扫描，内表使用索引查询

# 代码题目：

1：public class Test {

    public void myMethod(String str) {

        System.err.println("string");

    }

    public void myMethod(Object obj) {

        System.err.println("object");

    }

    public static void main(String[] args) {

        Test t = new Test();

        t.myMethod(null);

    }

}

输出结果为string；null是任何引用类型的初始值，String和Object的初始值都是null，但是null会优先匹配引用类型参数为String的方法；

2：

public class Test {

    public static void main(String[] args) {

        String str1 = new String("abc");

        String str2 = new String("abc");

        System.err.println(str1.equals(str2));

        StringBuffer sb1 = new StringBuffer("abc");

        StringBuffer sb2 = new StringBuffer("abc");

        System.err.println(sb1.equals(sb2));

    }

}

第一个true，第二个false。String重写了Object中的equals方法，会将string拆分为字符数组，逐个比较各个字符

3：

public class Test {

    public static void main(String[] args) {

        System.err.println(new Test().method1());

        System.err.println(new Test().method2());

    }

    public int method1() {

        int x = 1;

        try {

            return x;

        } finally {

            ++x;

        }

    }

    public int method2() {

        int x = 1;

        try {

            return x;

        } finally {

            return ++x;

        }

    }

}

第一个返回1，第二个返回2。finally中的代码是一定会被执行的且在try中的代码执行完之后，因此若在其中return返回，会覆盖掉try中的返回值。

4：

String str1 = "a";

        String str2 = "a";

        String str3 = new String("a");

        System.err.println(str1 == str2);

        System.err.println(str1 == str3);

        str3 = str3.intern();

        System.err.println(str1 == str3);

答案为true、false、true

System.err.println(12 - 11.9 == 0.1);

结果为false。12-11.9进行运算后会转换成对象，不在是基本数据类型，因此在进行恒等判断时，就会是false。

JVM（over）

二叉树（over）

红黑树（over）

TCP/IP(over)

Linux(over)

Spring（over）

Mybatis（over）

Sql

Mysql

Hadoop（over）

Java知识

Java8新特性

Java源码

Docker

Netty

Thrift

反射

排序算法重写（over）

设计模式

Java注解原理