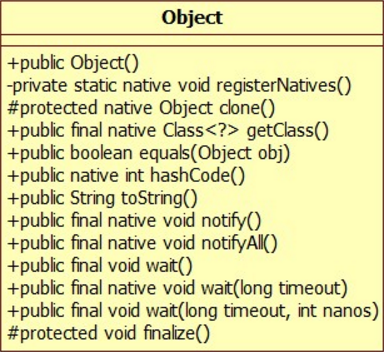
# Java

## Java Object类



## String有关的那些事儿

|  |
| --- |
| String s1 = "Hello"; // s1是保存在虚拟机栈上的符号引用，保存的是“Hello”在常量池中的地址  String s2 = "World"; // s2是保存在虚拟机栈上的符号引用，保存的是“World”在常量池中的地址  String s3 = **new** String("Hello"); // s3是保存在虚拟机栈上的符号引用，然后在堆中创建一个  // String对象，s3保存该对象在堆中的地址，而堆中保存  // 的是常量池中“Hello”的地址  String s4 = **new** String("Hello"); // 再在堆中创建一个新的对象，指向常量池中“Hello”的地  // 址，s4保存在栈上，指向堆中该对象的地址。  String s5 = s1 + s2; // 由于s1和s2都是变量，其底层是通过(new StringBuilder())  // .append(s1).append(s2).toString();实现的，会在堆中创建对  // 象，而常量池中还是保存了两个字符串常量“Hello”和“World”。  String s6 = s1 + "World"; // 由于s1是变量，处理过程同上  String s7 = "Hello" + "World"; // 相当于s7 = "HelloWorld"，直接指向常量池中的地址。  String s8 = "HelloWorld";  String s9 = s5.intern(); // 当一个String实例调用intern()方法时，JVM会查找常量池中是否  // 有相同字符串常量，如果有，则返回其的引用，如果没有，则在常量池  // 中常见一个相同的字符串常量，并返回它的引用；这样就使原本指向堆  // 中字符串对象的引用，直接指向常量池中该字符串的引用，这样可以避  // 免相同字符串常量重复在堆中申请空间。    System.***out***.println(s1 == s3); // false  System.***out***.println(s3 == s4); // false    System.***out***.println(s5 == s6); // false  System.***out***.println(s5 == s7); // false  System.***out***.println(s5 == s8); // false  System.***out***.println(s6 == s7); // false  System.***out***.println(s6 == s8); // false  System.***out***.println(s7 == s8); // true  System.***out***.println(s8 == s9); // true |

## 代码中if查询语句过多

不妨将每个查询条件放到map中，来优化if语句过多的情况

## *Integer.valueOf, Integer.parseInt, intValue*

|  |
| --- |
| Integer a = Integer.*valueOf*("1"); // 字符串或int转成Integer  **int** b = Integer.*parseInt*("2"); // 字符串转int  Integer i = **new** Integer(3);  i.intValue(); // Integer转int |

## java 创建对象的几种方式

1. 采用new
2. 通过反射
3. 采用clone
4. 通过序列化机制

前2者都需要显式地调用构造方法. 造成耦合性最高的恰好是第一种,因此你发现无论什么框架,只要涉及到解耦必先减少new的使用

## 静态变量与实例变量

1. 语法上，静态变量前要加上static关键字，而实例变量前不加；
2. 程序运行上，实例变量属于某个对象的属性，必须创建了实例对象，其中的实例对象才会被分配空间，才能使用这个实例变量。静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。标记一个方法为static，意味着这个方法，被所在类的所有实例公用，在类装载时初始化，被所有该类的实例共享，同时意味着：A、static方法内部不能引用非static变量。B、static方法不能被子类重写为非static方法。C、父类的非static方法不能被子类重写为static方法。D、static代码块可以用static {}来完成，在类被第一次装载时执行初始化，先于静态方法和其他方法的执行。标记一个变量为static，则该变量在内存中有不变的位置，相当于全局变量，所有类的实例都访问同一个存储变量区域。对其修改对于所有类的实例来说都是可见和一致的。

## JAVA内部类有哪几种

1. **静态内部类**

静态内部类虽然是外部类的成员，但是在未创建外部类的对象的情况下，可以直接创建静态内部类的对象。静态内部类可以引用外部类的静态成员变量和静态方法，但不能引用外部类的普通成员。

1. **成员内部类**

成员内部类可以调用外部类的所有成员，但只有在创建了外部类的对象后，才能调用外部的成员。成员内部类的对象必须通过外部类的对象创建

1. **局部内部类**

局部内部类只能在方法内部中使用，一旦方法执行完毕，局部内部类就会从内存中删除。必须注意：如果局部内部类中要使用他所在方法中的局部变量，那么就需要将这个局部变量定义为final的。

1. **匿名内部类**

匿名内部类的定义与对象的创建合并在一起，匿名内部类一般通过如下形式定义，并且在定义的同时进行对象的实例化。

new 类或者接口的名字() {

//匿名内部类的主体，大括号中是匿名内部类的主体，这个主体就是类或者接口的实现，如果是类，那么匿名内部类是该类的子类，如果是接口，匿名内部类需要完成接口的实现。

}

## IO中的设计模式

1. **适配器模式**

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

FileInputStream fileInput = new FileInputStream(file);   
InputStreamReader inputStreamReader = new InputStreamReader(fileInput);

FileInputStream是字节流，而InputStreamReader可以将其转为按字符读取。

1. **装饰器模式**

不改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。

BufferedReader bufferedReader=new BufferedReader(inputStreamReader);

将FileInputStream字节流包装为BufferedReader过程就是装饰的过程，使其具有一次读取一行的功能。

## 集合类



Map：HashMap、LinkedHashMap、HashTable（古老的实现类，线程安全的，不建议使用，Properties是它的子类）、TreeMap

## 接口和抽象类的区别

1. 抽象类不能被实例化；
2. 抽象类中可以有抽象方法也可以有非抽象方法；而接口中只能有抽象方法；
3. 抽象方法所在的类一定是抽象类；
4. 抽象类中的构造器不能是抽象的，因为构造器不能被重写；私有方法不能修饰抽象类，因为私有方法不能被继承，从而不能被重写；final和abstract不能同时修饰一个方法，因为一个不能被重写，一个必须被重写；
5. 抽象类中的抽象方法在非抽象子类中必须被实现；
6. 接口是一种特殊的抽象类，其中只能有抽象方法和常量（public static final）；
7. 一个类只能继承自一个类，但可以实现多个接口（单继承，多实现）；接口可以继承多个接口（多继承）；
8. 接口是没有构造器的，接口定义的是一种功能，这种功能可以被类实现；所有类都有构造器。

## java对象的序列化机制

对象的序列化机制允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上或通过网络将这种二进制流传输到另一个网络节点上。当其他程序获取了这种二进制流，就可以恢复成原来的java对象。

注意：要求类和类的属性都要可序列化，即都要继承Serializable接口。

注意：建议在类中固定private static final long serialVersionUID序列号，以区别类修改后不同版本的类的兼容性。

注意：static和transient修饰的成员变量不能序列化。

## 注解

通过@interface可以定义注解，通过以下四个元注解可以自定义其他注解。

java中的元注解主要有四个：

**@Target**：表示该注解用在什么地方

@Target(ElementType.TYPE)   //接口、类、枚举、注解  
@Target(ElementType.FIELD) //字段、枚举的常量  
@Target(ElementType.METHOD) //方法  
@Target(ElementType.PARAMETER) //方法参数  
@Target(ElementType.CONSTRUCTOR)  //构造函数  
@Target(ElementType.LOCAL\_VARIABLE) //局部变量  
@Target(ElementType.ANNOTATION\_TYPE) //注解  
@Target(ElementType.PACKAGE) //包

**@Retention**：表示在什么级别保存该注解信息

@Retention(RetentionPolicy.SOURCE)   //注解仅存在于源码中，在class字节码文件中不包含  
@Retention(RetentionPolicy.CLASS)     // 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得  
@Retention(RetentionPolicy.RUNTIME)  // 注解会在class字节码文件中存在，在运行时可以通过反射获取到

**@Document**：说明该注解将被包含在javadoc中

**@Inherited**：允许子类可以继承父类中的注解

## 进程和线程

1. 进程是资源分配的最小单位，线程时程序执行的最小单位。
2. 进程有自己独立的地址空间，每启动一个进程，系统都会为其分配地址空间，简历数据表来维护代码段、堆栈段和数据段，县城没有独立的地址空间，它们使用相同的地址空间共享数据。
3. CPU切换一个线程的花费远远小于切换进程。
4. 创建一个线程的开销小。
5. 线程占用的资源少。
6. 线程之间通信更方便，同一个进程下，线程共享全局变量、静态变量等数据，进程之间的通信需要以通信的方式进行。
7. 多进程程序更安全，生命力更强；多线程不易维护。
8. 进程对资源保护要求高，效率低。

## java中类初始化的运行顺序

1. 父类静态代码块，父类静态成员变量（二者优先级相同，先定义的先执行）
2. 子类静态代码块，子类静态成员变量（二者优先级相同，先定义的先执行）
3. 父类一般代码块，父类一般成员变量（二者优先级相同，先定义的先执行）
4. 父类构造函数
5. 子类一般代码块，子类一般成员变量（二者优先级相同，先定义的先执行）
6. 子类构造函数

## 反射的应用场景

反射就是在程序动态运行的过程中，动态加载某个类的过程。

主要的应用场景包括，对象的序列化、工厂模式、实现AOP时的动态代理以及IOC（控制反转机制）通过xml配置文件来创建对象（这其实就是简单工厂模式）等。

## 实现线程的四种方式

1. 利用继承的方式，继承自Thread类
2. 利用实现的方式，实现自Runnable接口（没有返回值）

Thread类本身就实现了Runnable接口；

第二种方式更好，因为某类通过实现Runnable接口实现线程，不会影响该类继承其他类和实现其他接口，避免了单继承的局限性；适合多个线程共同操作一份资源的情况，因为以继承的方式需要把共享变量设置为静态的才能有多个实例共享。

1. 利用实现的方式，实现自Callable接口（有返回值）
2. 通过线程池创建并管理线程

## 实现Runnable接口相对于继承Thread类的优势

1. 适合多个相同程序代码的线程去处理同一资源的情况；
2. 可以避免由于Java的单继承带来的局限；
3. 增强了程序的健壮性，代码能够被多个线程共享，代码与数据是独立的。

## Hashmap原理

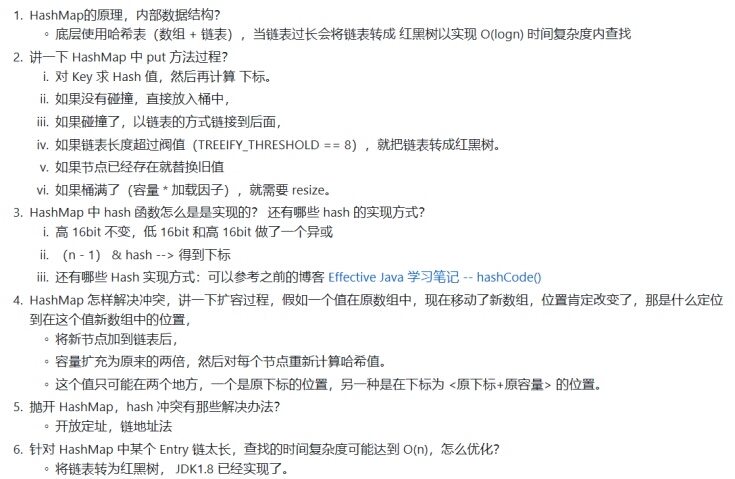
通过hash函数和hashcode计算出数据应该在数组（默认大小是16，当数据数量大于阈值=当前数组长度乘以加载因子=16\*0.75（初始情况）后，进行扩容（两倍扩容），扩容后还要重新遍历并计算每个元素的应该在新数组中的位置）中的存放位置（即，索引），当遇到冲突时采用链地址法，所以数组中的每个元素其实还是一个链表。

JDK1.7之前采用头插法，之后采用尾插法。（Why？）

确定存在数组哪个下标的方式是：利用其定制的hash算法，将hash值的前16位和后16位进行异或，这一步是保证最后得到的数组下标的索引尽可能分散到整个数组的空间。然后利用其hashcode &（n-1），（n表示数组大小（初始为16）），其值就相当于hash值对16取余。

Java1.8 之后，如果数组一个元素中的链表的数据数量大于8之后，数据结构就由单链表变成了红黑树这样就降低了查询的时间复杂度。如果结点个数小于6，就再变回单链表数据结构。

当插入的数据量较多，使得HashMap达到一定饱和度时，key映射位置发生冲突的几率会逐渐提高，这时候HashMap就需要扩展它的长度，也就是进行Resize。衡量HashMap是否需要进行Resize的条件是HashMap数组中存的数据大于等于HashMap容量\*负载因子，即HashMap.Size >= Capacity \* LoadFactor。扩容后还要对原来的数据进行ReHash。当多个线程同时对同一个HashMap对象进行rehash的时候，就有可能产生环形链表，造成下次查询时进入死循环状态。



## Vector、ArrayList、HashSet、HashMap、HashTable比较

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据结构 | 线程安全 | 底层结构 | 初始容量 | 加载因子 | 扩容增量 |
| Vector | 是 | 数组 | 10 |  | 两倍扩容 |
| ArrayList | 否 | 数组 | 10 |  | 1.5倍扩容+1 |
| HashSet | 否 | HashMap | 16 | 0.75 | 两倍扩容 |
| HashMap | 否 | 数组+链表+红黑树 | 16 | 0.75 | 两倍扩容 |
| HashTable | 是 | 数组+链表+红黑树 | 11 | 0.75 | 两倍扩容+1 |

## HashSet集合中防止键重复的原理

HashSet在新增值时，其实将值存入到一个HashMap中，map的键即为待加入到HashSet中的元素，map的值都是一个static final Object = new Object对象。而在HashMap中会先根据哈希函数（hash(key）)得到对应在数组中的位置，然后遍历数组中是否存在相同的key，判断方法是先判断hash是否相同，再通过equals()判断两个键是否相同。

**注意：hashcode不同，则equals()一定返回false；hashcode相同，则equals() 不一定返回true**。

## ConcurrentHashMap原理

为了避免HashMap的线程安全问题，可以改用HashTable和Collections. synchronizedMap()，但是二者都有着共同的性能瓶颈，无论读还是写操作，他们都会给整个集合加锁。

ConcurrentHashMap采用分段锁的设计来提高并发度。ConcurrentHashMap，它内部细分了若干个小的HashMap，称之为段(Segment)。默认情况下一个ConcurrentHashMap被进一步细分为16个段。不同的段可以同时读写，相同的段可以同时一个读一个写，但同一段不能同时写。其中大量使用final、volatile等来减少锁竞争对性能的影响，特别是Entry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性。

ConcurrentHashMap在进行size方法的时候，会先统计每个段的元素数量，然后统计修改次数，看统计元素数量操作前后的修改次数是否相等，如果不等，说明在统计元素数量过程中，又发生了对元素数量的修改操作，故重新尝试统计，并将尝试次数加1。如果最终尝试次数超过一定阈值，那么就给每个Segment加锁，再重新统计。这里用到了乐观锁和悲观锁的概念，当尝试一定次数，才无奈转为悲观锁。

## 几个并发容器

HashTable——默认给每个操作都加了锁

Collections.synchronizedXXX——传进去一个集合类，返回一个加了锁的集合类，如果传入的是map，返回的map有点等同于HashTable

ConcurrentHashMap——基于分段锁的机制来实现同步，实现了线程安全，同时提高了可并发程度

ConcurrentSkipListMap——基于跳表的同步Map，数据是有序的

## Java中的锁

**公平锁**：等待锁时间越长的那个线程就越会被分配得到锁。

**可重入锁**：也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。

**读写锁**：同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程都会被阻塞

|  |
| --- |
| ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();  lock.readLock().lock(); //上读锁，其他线程只能读不能写  lock.readLock().unlock(); //释放读锁，最好放在finally里面  lock.writeLock().lock(); //上写锁，不允许其他线程读也不允许写  lock.writeLock().unlock(); //释放写锁 |

[**其他锁**](#_乐观锁与悲观锁)

Java中的锁包括：synchronized，ReentryLock（tryLock（可尝试加锁），需要手动释放锁，可以指定为公平锁），countdownletch，condition，ReentrantReadWriteLock（读写锁）。

volatile关键字可以实现内存可见性，但不能保证原子性，并且避免指令重排序。

## ThreadLocal-线程局部变量

创建方式：**public** ThreadLocal<T> *tLocal* = **new** ThreadLocal<>();

设值：*tLocal*.get()

取值：*tLocal*.set(t);

为每个线程都维护一份自己的变量，变量存储于ThreadLocal中的内部类ThreadLocalMap对象中。

**ThreadLocal是使用空间换时间，加锁是使用时间换空间。**

## 队列Queue

ConcurrentLinkedQueue——offer()方法往里添加数据，添不进去就返回false；add()方法往里添加数据，添不进去就报错。

BlockingQueue：

LinkedBlockingQueue，ArrayBlockingQueue——offer()方法往里添加数据，添不进去就返回false；add()方法往里添加数据，添不进去就报错；put()方法往里添加数据，添不进去就阻塞。

DelayQueue——可用于执行定时任务，里面的每个任务都会有设置个延迟时间，延迟时间到了才会执行该任务。

LinkedTransferQueue——通过transfer()方法直接把任务发给消费者而不放入到队列中，如果没有可用消费者就一直阻塞等待；而通过offer、add、put方法可以将任务放到队列中。

SynchronousQueue——容量为0的队列，来的任何一个任务都要被消费者消费掉，使用add()方法会直接报错，通过put()方法会阻塞等待消费者。

## Java中的线程池

创建方式：

|  |
| --- |
| ExecutorService service = Executors.*newFixedThreadPool*(5);  // 往里面添加任务  **for**(**int** i = 0; i < 10; i ++) {  service.submit(**new** Runnable() {  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  }  });  } |

线程池用于管理线程，当需要时，线程池会帮忙创建线程，一个任务执行完成后，刚刚创建的线程不会消失，而是用于执行下一个提交的任务，避免了线程创建与回收的开销。

***execute()***方法用于执行不带返回值的任务，***submit()***方法用于执行带返回值的任务，也可执行不带返回值的任务。

service.shutdown();——等待所有任务执行完，再关闭线程池

service.shutdownNow();——立刻关闭线程池

*newSingleThreadExecutor()*：只有一个线程的线程池，LinkedBlockingQueue

*newSingleThreadScheduledExecutor()*：创建只有一个线程的定时任务线程池，内部其实调用了

*ScheduledThreadPoolExecutor(1)*

*newFixedThreadPool(****int*** *nThreads)*：有指定数量的线程的线程池，LinkedBlockingQueue

*newCachedThreadPool()*：根据需要实时创建需要数量的线程的线程池，SynchronousQueue

*newScheduledThreadPool(****int*** *corePoolSize)*：创建定时任务的线程池，DelayWorkQueue

以上线程池内部都是创建了一个ThreadPoolExecutor线程池：

|  |
| --- |
| ThreadPoolExecutor(**int** corePoolSize, // 在线程池中保留的线程数（即使线程空闲）  **int** maximumPoolSize, // 线程池中允许的线程最大数量  **long** keepAliveTime, // 当线程池中的 线程数量超过corePoolSize、  // 时，为多余的空闲线程设置销毁的等待时间  TimeUnit unit, // 上一个参数的时间单位  BlockingQueue<Runnable> workQueue , // 设置各自具体的阻塞队列  ThreadFactory threadFactory, // 创建新线程时的线程工厂  RejectedExecutionHandler handler)) // 设置线程池的拒绝策略 |

*newWorkStealingPool()*：线程池中的每个线程都维护一个自己的任务队列，当自己的任务队列为空的时候，就会去全局队列中拿任务，如果全局队列也为空，就会去其他线程的任务队列中窃取任务，这样做的目的是提高CPU利用率。里面的线程是一个守护线程（后台线程，精灵线程），主线程完成后退出，这些精灵线程就会在后台默默执行，前台看不到输出。

**new** ForkJoinPool()：采用分而治之思想，一个大任务被分成多个小任务，每个小任务都会创建一个线程。

**当一个任务欲添加到线程池的步骤：**

1. 如果线程池中线程数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务；
2. 如果线程池中的线程数量等于corePoolSize，但是缓冲队列workQueue未满，那么任务被放入缓冲队列；
3. 如果线程池中的线程数量大于等于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理被添加的任务；
4. 如果线程池中的线程数量等于maximumPoolSize，那么通过handler所指定的**拒绝策略**来处理任务。

**拒绝策略：**

1. ThreadPoolExecutor.AbortPolicy()：拒绝任务，并抛出RejectedExecutionException异常；
2. ThreadPoolExecutor.DiscardPolicy()：拒绝并直接丢弃任务；
3. ThreadPoolExecutor.DiscardOldestPolicy()：拒绝并丢弃最旧未被执行的任务，也就是队列中当前第一个任务；
4. ThreadPoolExecutor.CallerRunsPolicy()：拒绝任务，但是让被拒绝的任务直接在发起任务请求的线程中执行。
5. 自定义拒绝策略：实现RejectedExecutionHandler，并自己定义策略模式

## 进程间通信的方式

1. 共享存储器系统（共享内存）
2. 管道通信系统（即管道文件或pipe文件）
3. 消息传递系统（利用操作系统提供的通信原语）
4. 客户端-服务器系统（套接字socket）

## 40个java多线程问题总结

<https://mp.weixin.qq.com/s/9jftwCcINMlc0cq_bj2fyA>

## #{}和${}的区别

（1）#{}表示一个占位符号，可以有效防止sql注入。#{}可以接受简单类型值或pojo属性值。

（2）${}表示拼接sql串，可以接受简单类型值或pojo属性值。一般用于传入数据库表名。一般能用#的就别用$.

# JVM-Java虚拟机

## 内存中的线程共享区和线程私有区

1. 线程共享区：java堆区，方法区（运行时常量池）
2. 线程私有区：PC程序计数器（虚拟的寄存器），虚拟机栈，本地方法栈

## 永久区

永久区是一个常驻内存区域，可以说就是方法区，用于存放JDK或第三方jar包携带的类和接口等元数据，他不会被GC回收，默认64M。

## Java中GC的回收机制

判断java对象存活的算法：**可达性算法（根搜索算法）**，通过一系列的称为“GC Roots”对象作为起始点，从这些节点开始往下搜索，搜索走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连的时候（用图论的话说就是从GC Roots到这个对象不可达），则证明此对象是不可用的。

1.当eden满了，触发young GC也叫Minor GC；

2.young GC做2件事：一，回收一部分没用的对象；二，会将 Eden 区和 From 区中还被引用的对象会被移到 To区。在满足两种特殊情况下，Eden区和From区中的存活对象会被晋升到老年代里，首先是如果存活对象的分代年龄超过选项“-XX:MaxTenutingThreshold”所指定的阈值（默认是15）时，将会直接晋升到老年代中；其次当To空间的容量达到阈值时，存活对象同样是晋升到老年代中，剩下的均为垃圾对象，等待被释放。

3.当老年代也满了，利用标记-压缩算法，将所有存活对象都移动到一个规整且连续的内存空间中，然后触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

## GC Roots有哪些

1. 虚拟机栈中引用的对象；
2. 本地方法栈中native方法引用的对象；
3. 方法区中静态属性引用的对象；
4. 方法区中常量引用的对象。

## 有哪些垃圾回收器

1. **Serial收集器**，采用复制算法、串行回收和“Stop-the-World”机制的方式执行内存回收；它是新生代中单线程的垃圾收集器；
2. **Serial Old收集器**：执行老年代垃圾收集，采用了标记-压缩算法、串行回收和“Stop-the-World”机制；
3. **ParNew收集器**，采用复制算法、并行回收和“Stop-the-World”机制的方式执行内存回收；它是Serial收集器的多线程版本；
4. **Parallel收集器**：采用复制算法、并行回收和“Stop-the-World”机制的方式执行内存回收；它可以控制程序的吞吐量大小，因此他也被称作**吞吐量优先**的垃圾收集器；
5. **Parallel Old收集器**：执行老年代垃圾收集，采用标记-压缩算法、并行回收和“Stop-the-World”机制；
6. **CMS（Concurrent-Mark-Sweep）收集器**：低延迟收集器，执行老年代垃圾收集，采用标记-清除算法，回收后会产生内存碎片，只能选择空闲列表执行内存分配；
7. **G1（Garbage-First）收集器**：是一款基于并行和并发，低延迟以及暂停时间更加可控的区域化分代式垃圾收集器，它是将整个堆区划分成2048个大小相同的独立Region块。

# 设计模式

## SpringMVC中的设计模式

[见Spring部分](#_SpringMVC中的设计模式)

## 单例模式的实现方式

1. **饿汉式**

在一开始就在类变量中new出来，缺点是：当系统中这样的类较多时，会使得启动变慢，所以现在流行采用“延迟加载”的方式，在第一次使用时再创建。

1. **懒汉式**

采用“懒加载”的方式，在第一次使用该对象时再创建，但需要使用同步方法来保证线程安全。

**注意：**这里最好不要使用同步方法，即不要将synchronized直接修饰对象创建的方法，而是在判断是否是第一次创建对象的if语句里，作为同步代码块包括if语句里面的内容。

|  |
| --- |
| **class** Singleton {  **private** **static** Singleton *instance*;  **private** Singleton() {}    // public static synchronized Singleton getInstance(int a) {  **public** **static** Singleton getInstance() {  **if**(*instance* == **null**) {  **synchronized** (Singleton.**class**) {  **if**(*instance* == **null**)  *instance* = **new** Singleton();  }  }  **return** *instance*;  }  } |

**更好地：**我们采用内部类的方式，既能实现懒加载，又能实现线程安全；

|  |
| --- |
| **class** Singleton {  **private** Singleton() {}    **private** **static** **class** Inner {  **private** **static** Singleton *s* = **new** Singleton();  }    **public** **static** Singleton getSingleton() {  **return** Inner.*s*;  }  } |

**再有**：我们采用枚举类来保证单例并实现懒加载和线程安全

|  |
| --- |
| **public** **class** EnumSingleton {  **private** EnumSingleton() {}  **public** **static** EnumSingleton getInstance() {  System.***out***.println("经过这里1");  **return** Singleton.***INSTANCE***.getInstance();  }  **private** **static** **enum** Singleton {  ***INSTANCE***;  **private** EnumSingleton singleton;  // JVM会保证此方法绝对只调用一次  **private** Singleton() {  System.***out***.println("我被加载了");  singleton = **new** EnumSingleton();  }  **public** EnumSingleton getInstance() {  **return** singleton;  }  }  } |

# Spring

## SpringMVC的启动过程

<https://mp.weixin.qq.com/s/b47QP6Y_IIWb7TAxSMolbw>

第一步：发起请求到前端控制器（DispatcherServlet）

第二步：前端控制器请求HandlerMapping查找处理器Handler，可以根据xml配置、注解进行查找

第三步：处理器映射器HandlerMapping向前端控制器返回Handler

第四步：前端控制器调用处理器适配器（HandlerAdapter）去执行Handler

第五步：处理器适配器去执行Handler

第六步：Handler执行完成给适配器返回ModelAndView

第七步：处理器适配器向前端控制器返回ModelAndView

第八步：前端控制器请求视图解析器View Resolver去进行视图解析

第九步：视图解析器向前端控制器返回View

第十步：前端控制器进行视图渲染

视图渲染将模型数据（在ModelAndView对象中）填充到request域

第十一步：前端控制器向用户响应结果。

## IOC（控制反转）

就是将对象的创建工作交给框架来执行。依赖注入（DI）和依赖查找是实现控制反转的两种方式。而其中依赖注入包括设值注入（getter、setter函数），构造器注入，接口注入三种方式来实现对象创建。

**如何理解IOC?**

**Ioc—Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。**在Java开发中，**Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。**

　　●**谁控制谁，控制什么：**传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对 象的创建；**谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。**

●**为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？**因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。**

　　IoC 不是一种技术，只是一种思想，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

**所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的类的创建、销毁都由 spring来控制，也就是说控制对象生存周期的不再是引用它的对象，而是spring。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。**

## Spring中AOP的实现原理

动态代理

## SpringMVC中的设计模式

**简单工厂模式**，**工厂方法模式**（beanFactory）

**单例模式**：配置bean的时候，默认情况下是单例模式，是在一个spring bean容器中只创建一个实例，当有多个spring容器时，也可能是多个实例。可通过scope=*"prototype"*或者singleton*="false"*设置为多例模式。scope=*"request"*每一次http请求会产生一个bean，同时该bean仅在当前HTTP request内有效；scope=*"request"*每一次session会产生一个bean，同时该bean仅在当前HTTP session内有效。

**代理模式**：Springmvc中实现AOP的方式是通过动态代理的方式。

装饰者模式（动态地给一个对象添加一些额外的职责），观察者模式，策略模式，适配器模式

## 动态代理的几种实现方式

通过jdk或者cglib

# NIO

**缓存**：Buffer

**通道**：

FileChannel——本地文件io

SocketChannel——TCP网络通信

ServerSocketChannel——TCP网络通信

DatagramChannel——UDP网络通信

Pipe.SinkChannel——通过管道发送数据（管道是单向的）

Pipe.sourceChannel——通过管道读取数据（管道是单向的）

**选择器**：

Selector，接收并管理注册进来的通道以及通道对应的事件（接收事件、读事件、写事件、连接事件），迭代监听其中已就绪的事件，并判断就绪的是哪类事件，然后对相应事件进行处理。

# 数据结构&算法

## 数据结构经典排序算法比较

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **排序算法** | | **时间复杂度** | | | **空间复杂度** | **稳定性** |
| **平均情况** | **最好情况** | **最坏情况** |
| 插入排序 | 直接插入排序 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |
| 折半插入排序 | O(n2) | 比上面差 | 比上面好 | O(1) | 稳定 |
| 希尔排序 | O(nlogn)~O(n2) | O(n1.3) | O(n2) | O(1) | 不稳定 |
| 选择排序 | 简单选择排序 | O(n2) | O(n2) | O(n2) | O(1) | 不稳定 |
| 堆排序 | O(nlogn) | O(nlogn) | O(nlogn) | O(1) | 不稳定 |
| 交换排序 | 冒泡排序 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |
| 快速排序 | O(nlogn) | O(nlogn) | O(n2) | O(logn) | 不稳定 |
| 归并排序 | | O(logn) | O(logn) | O(logn) | O(n) | 稳定 |
| 基数排序 | | O(d(n+rd)) | O(d(n+rd)) | O(d(n+rd)) | O(rd) | 稳定 |
|  | | | | | | |

## 红黑树

1. 节点是红色或黑色。
2. 根节点是黑色。
3. 每个叶子节点都是黑色的空节点（NIL节点）。
4. 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

类似于平衡二叉树，插入方式与平衡二叉树相同。

## 一串数字找出连续和最大的数字子串

算法思想：定义一个临时变量max，从一个数字开始取和，当和大于max时，将该和赋值给max，当到某处和小于0时，则将该处作为起点，重新开始计算和，当和大于max时，赋值给max，依次遍历完整个数字串。

## 输入一个数字n，计算1~n之间所有数中的所有数字个数之和

算法思想：个位数有n个，十位数有n-9个，百位数有n-99个，千位数有n-999个……

## 找出n个数中前m个最大的

算法思想：采用堆排序的方式，建立一个有m个节点的小根堆，每趟排序后输出掉最小的那个数，所有数操作完成后，最后保留在堆中的m个数就是结果。时间复杂度：

## 判断一个数是否在40亿个数中

算法思想：如果将所有的数全部从外存读入map中，空间开销将是非常大的。可以考虑以下两种方法：

**方法一：**采用位图法，一个int数的取值范围为0~，所以我们可以考虑创建一个可以存储个bit位的数组，根据数组下标来指定这个数是否存在，例如，该数存在则记为1，否则记为0。在查找的时候，根据待查找的数去对应的下标下面取数，若为1，表示该数存在，否则，该数不存在。这种方法大概只需要512MB的内存，且时间复杂度为O(1)。

**方法二：**当有40亿个整数的时候，其中必定有很多连续的数，那么可以用区间将这些数保存下来。例如有1,2,3，4,6,7这几个数，那我们可以保存几个区间，即，区间[1,4]及区间[6,7]。

# 数据库

## 事务的四个特性（ACID）

原子性，一致性，隔离性，持久性

1. 原子性：事务必须是原子工作单元；对于数据修改，要么全都执行，要么全都不执行。通常与某个事务关联的操作具有共同的目标，并且是相互依赖的。如果系统只执行这些工作的一个子集，则可能破坏事务的总体目标。原子性消除了系统处理操作子集的可能性；
2. 一致性：在一个事务执行之前和执行之后数据库都必须处于一致性状态。如果数据库的状态满足所有的完整性约束，就说该数据库是一致的；
3. 隔离性：由并发事务所作的修改必须与任何其它并发事务所作的修改隔离。事务查看数据时数据所处的状态，到底是另一个事务执行之前的状态还是中间某个状态，相互之间存在什么影响，是可以通过隔离级别的设置来控制的；
4. 持久性：事务结束后，事务处理的结果必须能够得到固化，即写入数据库文件中即使机器当机数据也不会丢失，它对于系统的影响是永久性的。

## 事务的并发控制

如果不对事务进行并发控制，数据库并发操作可能有以下异常：

1. **第一类丢失更新：**(回滚丢失)两个事务同时更新count，两个事务都读取到100，事务一更新成功并提交，count=100+1=101，事务二出于某种原因更新失败，然后回滚，事务二就把count还原为它一开始读到的100，此时事务一的更新失败；
2. **脏读：**一个事务读取了另一个事务修改了但是未提交的数据；事务一更新了count，事务二此时读取count，然后事务一出于某种原因回滚了；
3. **不可重复读：**一个事务对同一行数据执行了两次或更多次查询，但是却得到了不同的结果，造成这个结果的原因是两次查询之间有别的事务对该行数据做了更新操作；（UPDATE）
4. **第二类丢失更新：**此种更新丢失是因为更新被其他事务覆盖，两个事务同时更新，都读取到100，事务一先更新成功并提交，事务二更新成功并提交，由于事务二还从100开始增加，事务一的更新就丢失了；
5. **幻读：**类似不可重复读，只是针对的不是数据的值，而是数据的数量。此种异常时一个事务在两次查询中数据的数量不同。（INSERT和DELETE）

## 事务的隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 隔离级别 | 说明 | 备注 |
| 底  ↓  高 | Read uncommitted（读未提交） | 一个事务可以读取另一个未提交事务的数据 | 可能导致脏读 |
| Read committed（读已提交）  （大多数数据库的默认隔离级别） | 一个事务要等另一个事务提交后才能读取数据 | 可避免脏读 |
| Repeatable read（可重复读）  （Mysql的默认隔离级别） | 在开始读取数据（事务开启）时，不再允许修改操作 | 可避免脏读，不可重复读 |
| Serializable（序列化） | 最高的事务隔离级别，在该级别下，事务串行化顺序执行。效率低下，比较耗数据库性能。 | 可避免一切情况 |

## 事务的开启方式

* Mysql：begin/start transaction（开始），commit（结束），savepoint（设置保存点），rollback to a可以退回到指定保存点，set transaction（设置事务的隔离级别）
* Redis：MULTI（开始），EXEC（结束），Redis没有回滚功能
* JDBC：conn.setAutoCommit(false) - 设置是否为自动提交事务，如果true（默认值为true）表示自动提交，也就是每条执行的SQL语句都是一个单独的事务，如果设置为false，那么相当于开启了事务了；conn.commit() – 提交事务；conn.rollback() – 回滚事务

## 乐观锁与悲观锁

* 乐观锁：认为读多写少，所以在对数据操作时不会加锁，而是通过**数据版本**或者**时间戳**的方式来保证读到数据的一致性；
* 悲观锁：认为写多，所以在对数据库操作的一开始就对数据加了锁

## 共享锁、排他锁和更新锁

* 共享锁(S)：事务A对对象T加S锁，其他事务也只能对T加S，多个事务可以同时读，但不能写，直到A释放S；
* 排它锁(X)：事务A对T加X后，其他事务不能对T加任何锁，只有事务A可以读写T直到A释放X；在查询语句后面增加for update即为此表增加排它锁；
* 更新锁(U)：用来预定对T加X，允许其它事务读，但不允许再加U或者X。当读取的对象被更新时，则升级为X。使用U时，修改数据的操作分为两步：首先获得一个共享锁，读取数据，然后将共享锁升级为排它锁，再执行修改操作；

## 行锁和表锁

按**作用范围**：

* 行锁：锁的作用范围是行级别，数据库能够确定哪些行需要锁的情况下使用行锁，如果不知道会影响哪些行就使用表锁。
* 表锁：作用范围是整张表。

**注意：**InnoDB引擎只有通过索引进行检索的时候才会使用行级锁，否则使用表级锁。

## 自旋锁和互斥锁

何谓**自旋锁**？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于**互斥锁**，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

## MyIsam和InnoDB的区别

InnoDB：支持外键；支持事务；聚集索引（一级索引），必须有主键，辅助索引（二级索引）需要两次索引，先查到主键，再查到数据；不保存具体行数；不支持全文索引；支持行锁。

MyIsam：不支持外键；不支持事务；非聚集索引；保存整个表的行数；支持全文索引；只支持表锁。

## 索引的原理

Mysql索引采用B+树的数据结构来存储索引。随着数据的增加，索引本身大小也随之增加，不可能全部存储在内存中，因此索引往往以索引文件的形式存储在磁盘上。这样的话，索引查找过程就要产生磁盘I/O消耗，因此每读取一个结点都要进行一次磁盘的I/O读取，由于平衡二叉树虽然性能有所提高，但是二叉树这么大的深度显然是不能够接收的，所以采用m叉树（多路平衡搜索树）——B+树作为其数据结构。

为了尽量减少磁盘I/O，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，磁盘也会顺序向后读取一定长度的数据，预读的长度一般为页的整数倍。所以MySQL中将B+树结构的索引中的每个节点的大小都设置为页的整数倍。

另外，Mysql索引对B+树的数据结构做了几点改进。~~一方面，在进行插入删除操作时，不只是保留了B+树的插入删除方式，还借鉴了平衡二叉树的插入删除方式，即向兄弟结点借位置的方式。~~另一方面，在每个叶子节点上增加一个指向相邻叶子节点的指针，就形成了带有顺序访问指针的B+树。

## 聚集索引和辅助索引（一级索引和二级索引）

聚集索引一般指primary key或unique修饰的字段，他们的键一般都具有唯一性。

而辅助索引一般指其他建表时创建的索引，一般不要求键是唯一的。

## 聚集索引与非聚集索引

聚集索引与非聚集索引的最大区别在于聚集索引的索引的逻辑顺序与实际数据中每条记录的物理顺序是一致的。聚集索引的叶子节点上存储的就是数据，而非聚集索引的叶子节点上存储的是数据的地址。

## 组合索引

在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时遵循最左前缀集合

## 全文索引

一般MySQL的全文索引只能用在MyISAM表格的char、varchar和text的字段上。关键在于分词技术的实现，他会保存文档中每个词的位置和频率。

## 不会使用索引的情况

通过explain 加上查询语句可以查看执行查询语句时是否使用索引等信息。

1. like**模糊查询**以%开头的不走索引；
2. 存在索引列的**数据类型隐形转换**的不走索引，比如列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引；
3. **组合索引**的情况下，查询条件不满足索引最左原则的不走索引；
4. Mysql估计使用索引比全表扫描慢的不走索引；
5. 用**or**分割开的条件，如果or左右两个条件中有一个没有索引，则不会使用索引。；
6. **负向查询**（not , not in, not like, <>, != ,!>,!<）不走索引；
7. 使用select \* 的**可能**不走索引，关键看第4点；
8. Where子句索引列上有**数学运算**或者**函数运算**的不走索引；

## SQL优化

[不走索引的情况最好不要用](#_不会使用索引的情况)。

<http://www.thinkphp.cn/topic/54536.html>

语句调优，建立索引，优化数据库或者服务器本身等等…

1. 开启并查看慢查询日志定位执行效率底的SQL语句；
2. 使用explain来查看具体查询语句的执行计划；
3. 确定问题，采取相应的优化措施，建立索引等。（[不会使用到索引的情况](#_不会使用索引的情况)）；
4. 在优化sql无法解决问题的情况下，可以考虑搭建缓存；
5. 缓存也搞不定话，可以搞主从复制，读写分离；
6. 还是搞不定的可以做垂直拆分，将不常用的字段单独放在一张表；
7. 万不得已才做水平拆分，很麻烦。
8. 当已知只有一行数据时使用LIMIT1 。当你查询表的时候，你已经知道结果只会有一条结果，但因为你可能需要去fetch游标，或是你也许会去检查返回的记录数。在这种情况下，加上LIMIT 1可以增加性能；
9. 为经常出现在where后的字段创建索引；
10. 在join表的时候on的判断条件的字段使用相同类型，并建立索引；
11. 千万不要ORDER BY RAND()来打乱返回的数据行；
12. 避免使用select \*，需要什么就查什么；
13. 为每张表设置一个主键ID（最好是INT UNSIGNED）；
14. 使用ENUM而不是VARCHAR；
15. 从PROCEDURE ANALYSE()取得建议；
16. 尽可能使用NOT NULL，可以设置默认值（NULL其实也是需要额外空间的）；
17. 建议把子查询改为join连接查询，因为join查询不需内建临时表处理，临时表的创建和销毁会极大地拖慢效率；
18. 使用limit时，可以通过where先查出从上次记录返回的主键开始的几条记录，然后再用limit；

例如，select \* from user order by limit 50, 5; 这会扫描55条记录。而使用select \* from user where id>50 and id<=55 limit 5; 这会降低扫描次数。

1. Mysql配置文件my.ini或my.cnf中的一些配置参数的调优

## MySQL索引优化分析

<https://mp.weixin.qq.com/s/Zt6lRKtzSwpJfRQ2kgNwug>

## where、on、having的区别

ON、WHERE、HAVING的主要差别是其子句中限制条件**起作用时机**引起的，ON是在生产临时表之前根据条件筛选记录，WHERE是从生产的临时表中筛选数据，而HAVING是对临时表中满足条件的数据，进行计算分组之后，通过HAVING限制语句筛选分组，返回结果是满足HAVING子句限制的分组。

1. 多表连接会产生一个临时表，on是在产生临时表之前对数据库记录进行过滤，然后再产生临时表；where是在产生临时表之后根据限制条件在临时表中筛选结果。返回结果：在左外（右外）连接中，ON会返回左表（右表）中的所有记录；而WHERE中，只会返回满足条件的记录（因为是从临时表中筛选，会过滤掉不满足条件的）。<https://www.cnblogs.com/poorapple/p/7199509.html>
2. HAVING和WHERE的区别也是与限制条件起作用时机有关，HAVING是在聚集函数计算**结果出来之后**筛选结果，查询结果只返回符合条件的分组，HAVING**不能单独出现**，只能出现在GROUP BY子句中；而WHERE是在**计算之前**筛选结果，如果聚集函数使用WHERE，那么聚集函数只计算满足WHERE子句限制条件的数据。<https://blog.csdn.net/bruceyang2009lzu/article/details/38846357>

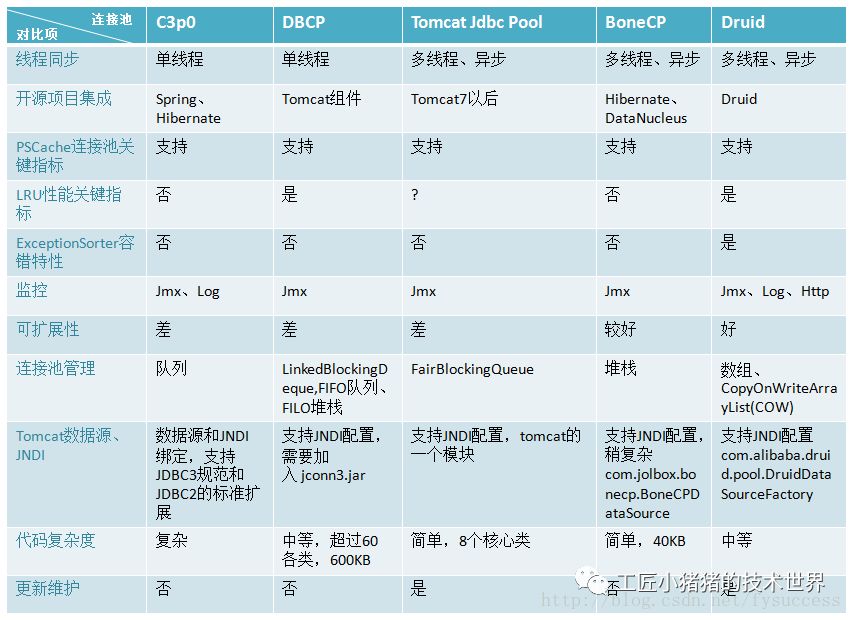
## count(\*)和count(具体某个字段，如id)的区别

前者会统计所有值的个数；后者只统计字段中非NULL值的个数。

## 数据库中in和exists的区别

1. in：先执行in后面的子查询，得出结果后，再与主查询做笛卡尔乘积，最后再判断并留下符合条件的记录；
2. exists：先执行主查询，然后再根据得到的每一条记录去判断exists后面的查询语句是否成立，留下符合条件的记录，子查询可以使用select 1替换select \*。

## 主流数据库连接池



## 简单介绍Redis

Redis是一种高性能的key-value数据库，它支持数据的备份和持久化，因为其数据存在内存中，所以性能极高，且包含丰富的数据类型（String，list，set，hash，zset等）。一般将Redis用于缓存和队列系统。

## Redis持久化的方式

1. RDB，RDB持久化就是把当前进程数据生成快照保存到硬盘的过程，可以通过手动和自动触发。

手动触发方式：save命令：会阻塞当前Redis服务器。bgsave命令会通过fork操作创建子进程，然后由子进程负责RDB持久化过程。

自动触发方式：save m n; 表示m秒内数据集存在n次修改时，自动触发bgsave。

优点：RDB是一个紧凑压缩的二进制文件，代表Redis在某个时间点的数据快照，非常适用于备份；

Redis加载RDB恢复数据远远快于AOF的方式。

缺点：RDB方式数据没办法做到实时持久化。因为bgsave每次运行都要进行fork操作创建子进程，属于重量级操作，频繁执行成本过高。

RDB文件使用特定二进制格式保存，Redis版本演进过程中有多个格式的RDB版本，存在老版本Redis服务无法兼容新版RDB格式的问题

1. AOF，AOF持久化以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式。

## 分布式锁

1. 基于数据库实现分布式锁  
   **方法一：**在数据库中创建一个表，表中包含方法名等字段，并在方法名字段上创建唯一性约束，想要执行某个方法，就是用这个方法名向表中插入数据，若插入成功则可以获得锁，反之，则未获得锁，方法执行完成后删除对应的行数据以释放锁。  
   **缺点：没有失效时间：**一旦数据未被正常删除，将永远获取不到锁；**非阻塞的：**获取不到锁则直接返回失败，需要循环多次去获取；**非可重入的：**同一线程在释放锁之前，行数据一直存在，无法再次获得锁，可以在表中新增一列，用于记录当前获取到锁的机器和线程信息，在再次获取锁的时候，可以查询这些信息是否与当前机器和线程信息匹配，若匹配，则直接获取锁。  
   **方法二：**基于数据库的排它锁来实现分布式锁。基于mysql的InnoDB引擎，在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁（[排它锁注意事项](#_共享锁、排他锁和更新锁)），当某条记录被加上排它锁之后，其它线程无法再在该行记录上增加排它锁。获得锁之后可以通过commit来释放锁。  
   **相比于方法一的优点：解决了非阻塞问题**：for update执行失败时一直处于阻塞状态，直到成功；**宕机后锁无法释放：**使用这种方式，服务宕机后数据库会自己把锁释放掉。
2. 基于缓存（Redis等）实现分布式锁  
   setnx key val；# 当key不存在时，才能set值，返回1，否则，什么也不做，直接返回0  
   expire key timeout；#设置超时时间，单位为s，超过这个时间锁会自动释放，以避免死锁  
   delete key；#用于删除key，相当于释放锁  
   **场景一：**setnx与expire命令分开写的话，不具有原子性，一旦中间宕机的话，有可能造成锁永远不会被释放；  
   解决方案：Redis 2.6.12以上版本为set指令增加了可选参数，可以同时设置val和超时时间以及nx，命令为:set(key,value,timeout,NX)，这样可以替代setnx指令。  
   **场景二：**如果节点1的线程给某个key加了锁，在线程未执行完之前，锁超时被删除。此时，节点2的线程给这个key加了锁，此时，节点1的线程执行完成，删除了该key，从而导致了误删。  
   解决方案：将该key的value值设置成线程id，删之前比较一下线程id是否匹配。
3. 基于Zookeeper实现分布式锁

## 一致性hash算法

redis集群中会采用hash算法将不同的key映射到对应的集群节点上。如果直接通过hash值取余的方法，当集群中新增或者删除节点的时候，会对所有的key进行rehash，会造成很大开销。

因此我们采用**一致性hash算法**，构建一个的环形hash空间中。先对集群中的每个机器（节点）计算出hash值（一般将机器的ip或者唯一的别名作为hash函数的输入），映射到hash环空间上。再将待存储的对象，计算出hash值，然后映射到hash环上，这样顺时针寻找离该待存储对象最近的机器，并存储到其中即可。此时，如果删除一个节点，只需要将该节点与逆时针的上一个节点之间的数据（即，原本存储在被删除节点里的全部数据）存储到该节点顺时针的下一个节点即可，其他数据无需变动。如果新增一个节点，只需要将新增节点与上一个节点之间的数据存储到新增节点中即可。采用一致性hash算法能够避免大量数据的迁移，减小了服务器的压力。

同时，为了考虑**均衡性**，我们采用**虚拟节点**的方式，将一台机器在hash环上虚拟出多个节点来，同理，将每个待存储的对象顺时针存储到离他最近的下一个虚拟节点对应的机器中，这可以一定程度上的实现负载均衡，避免因为一个机器的宕机，而导致大量的数据全部顺时针迁移到下一个机器上，造成该机器负载过重。

# 网络

## get和post的区别

1. get回退是无害的，而post会再次提交请求；
2. 浏览器中通过网址访问就是一个get请求，在form表单中，可以通过method来指定get或者post提交方式；
3. get安全性较低，因为其请求的数据会暴露在地址栏中，而post则不会，post数据放在request body中；
4. get通过url传送的参数是有长度限制的，而post没有；
5. get传递的参数类型的只是ascii字符，而post没有限制；
6. GET产生一个TCP数据包；POST产生两个TCP数据包。对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200(返回数据)；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

## Cookie和Session的区别

1. session能够存储任意的java对象，cookie只能存储String类型的对象；
2. cookie在客户端，可以编辑伪造，不安全，而session在服务端；
3. session过多时会消耗服务器资源，大型网站会有专门的Session服务器，Cookie存在客户端没问题；
4. 域的支持范围不一样。

## HTTP工作原理

1. 客户端连接到Web服务器；
2. 发送http请求；
3. 服务器接收请求并返回HTTP响应；
4. 释放TCP连接；
5. 客户端浏览器解析HTML内容。

## URI和URL的区别

URI：统一资源标志符（Uniform Resource Identifier）

URL：统一资源定位符（uniform resource location）

URN：统一资源名称 (Uniform Resource Name)

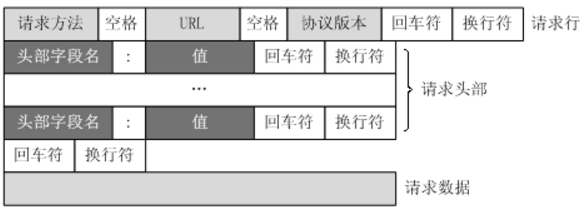
URL和URN都是URI的一种，URI是URL和URN的组合，URL告诉我们访问网络位置的方式，URN告诉我们名称，但是不包括访问方式；URI就相当于我们的身份证，URL就相当于身份证上的地址，URN就相当于我们的名字；例如：

URI：http://bitpoetry.io/posts/hello.html#intro

URL：http://bitpoetry.io/posts/hello.html

URN：bitpoetry.io/posts/hello.html#intro

## HTTP请求和响应报文



一个Http请求报文由请求行、请求头部、空行和请求数据4个部分组成；

1. 请求行

请求方法包括GET，HEAD，PUT，POST，DELETE等方法，并不是所有的浏览器都实现了所有的方法；

协议版本：HTTP/主版本号.次版本号

1. 请求头部

由多行组成，有键值对组成，每行一对，名和值之间使用冒号分隔；

|  |
| --- |
| POST /search HTTP/1.1  Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint,  application/msword, application/x-silverlight, application/x-shockwave-flash, \*/\*  Referer: <a href="http://www.google.cn/">http://www.google.cn/</a>  Accept-Language: zh-cn  Accept-Encoding: gzip, deflate  User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; TheWorld)  Host: <a href="http://www.google.cn">www.google.cn</a>  Connection: Keep-Alive  Cookie: PREF=ID=80a06da87be9ae3c:U=f7167333e2c3b714:NW=1:TM=1261551909:LM=1261551917:S=ybYcq2wpfefs4V9g;  NID=31=ojj8d-IygaEtSxLgaJmqSjVhCspkviJrB6omjamNrSm8lZhKy\_yMfO2M4QMRKcH1g0iQv9u-2hfBW7bUFwVh7pGaRUb0RnHcJU37y-FxlRugatx63JLv7CWMD6UB\_O\_r  hl=zh-CN&source=hp&q=domety |

User-Agent：产生请求的浏览器类型；发送请求的应用程序名称。

Accept：客户端可识别的内容类型列表；

Host：请求的主机名，接受请求的服务器地址。

1. 空行

空行表示请求头结束，通知服务器后面不在有请求头

1. 请求数据

若请求方法为get，则此项为空，数据包含在了请求行里；

若方法字段是POST,则通常来说此处放置的就是要提交的数据，以&符号分隔。



HTTP响应报文也由三部分组成：响应行、响应头、响应体

1. 响应行

一般由协议版本、状态码及其描述三部分组成，比如HTTP/1.1 200 OK

常见的状态码：

1\*\*—信息类，表示请求已接收，要求客户端继续提交下一步请求才能完成整个处理过程；

2\*\*—成功类，表示用户的请求被正确接收；

3\*\*—重定向，未完成请求，客户需要进一步细化请求；

4\*\*—客户端请求错误；

5\*\*—服务端错误；

1. 响应头

响应头用于描述服务器的基本信息，以及数据的描述，服务器通过这些数据的描述信息，可以通知客户端如何处理等一会儿他回送的数据；

应答头可以用来完成：设置Cookie，指定修改日期，指示浏览器按照指定的间隔刷新页面，声明文档的长度以便利用持久HTTP连接，……等等许多其他任务。

1. 响应体

响应体就是响应的消息体，如果是纯数据就是返回纯数据，如果请求的是HTML页面，那么返回的就是HTML代码，如果是JS就是JS代码，如此之类。

## TCP/IP协议中的三次握手和四次挥手：

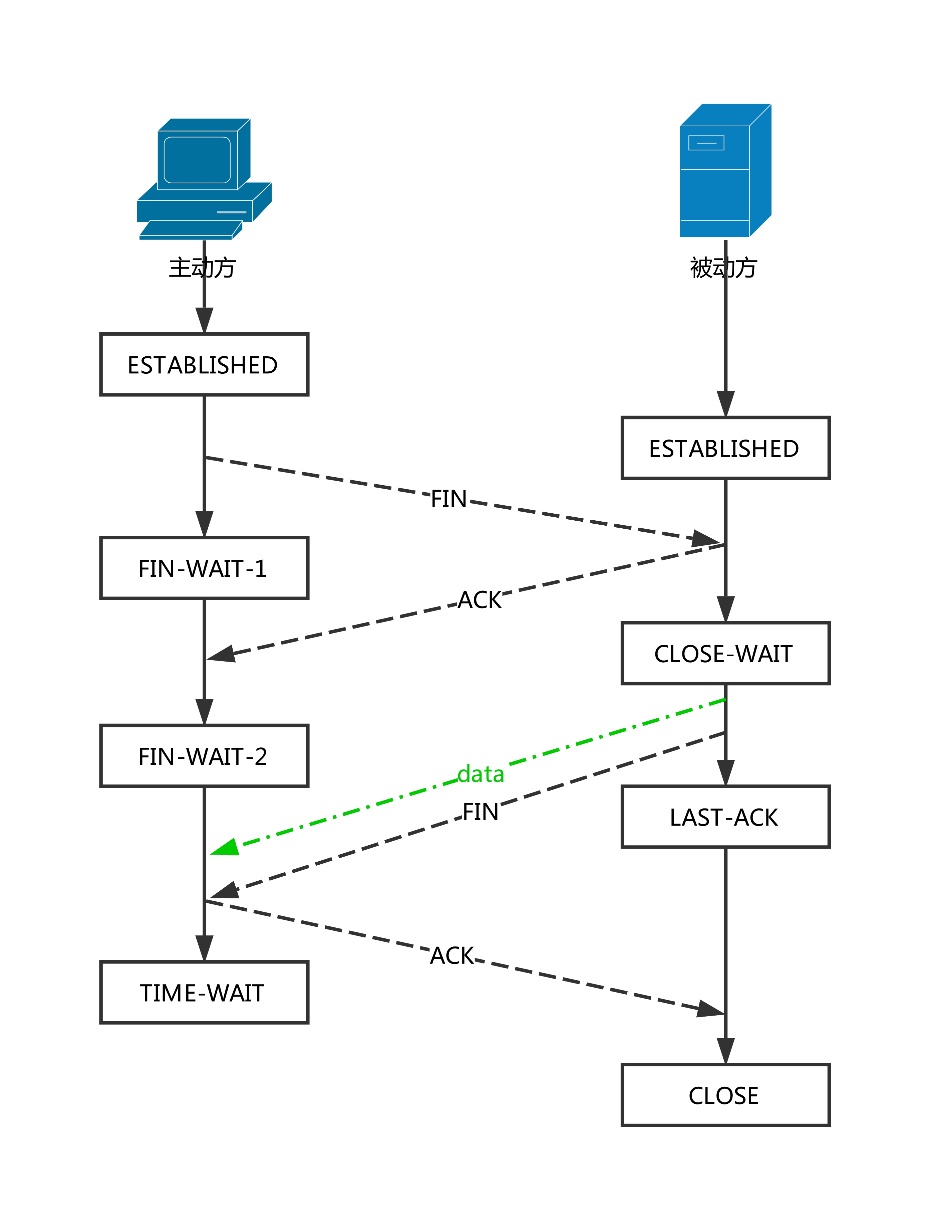
**第一次握手：**建立连接时,客户端发送syn包(syn=j)到服务器,并进入SYN\_SEND状态,等待服务器确认； SYN：同步序列编号(Synchronize Sequence Numbers)

**第二次握手：**服务器收到syn包,必须确认客户的SYN（ack=j+1）,同时自己也发送一个SYN包（syn=k）,即SYN+ACK包,此时服务器进入SYN\_RECV状态；

**第三次握手：**客户端收到服务器的SYN＋ACK包,向服务器发送确认包ACK(ack=k+1),此包发送完毕,客户端和服务器进入ESTABLISHED状态,完成三次握手.

完成三次握手,客户端与服务器开始传送数据。

四次挥手（过程）：



## UDP和TCP

1. TCP是面向连接的，UDP是无连接的；
2. TCP提供可靠的服务，也就是说TCP连接传送的数据是无差错，不丢失，不重复，且按序到达；而UDP尽最大努力交付，不保证可靠性，可能造成数据丢失；
3. UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信；
4. 每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信；
5. TCP对系统资源要求较多，UDP对系统资源要求较少；
6. 流模式（TCP）与数据报模式(UDP)。

## 网路的七层模型

(1) 减轻问题的复杂程度，一旦网络发生故障，可迅速定位所处层次；

(2) 将服务、接口、协议三个概念明确区分，服务说明某一层为上一层提供什么功能，接口说明上一层使用下层地服务，协议涉及如何实现本层地服务；

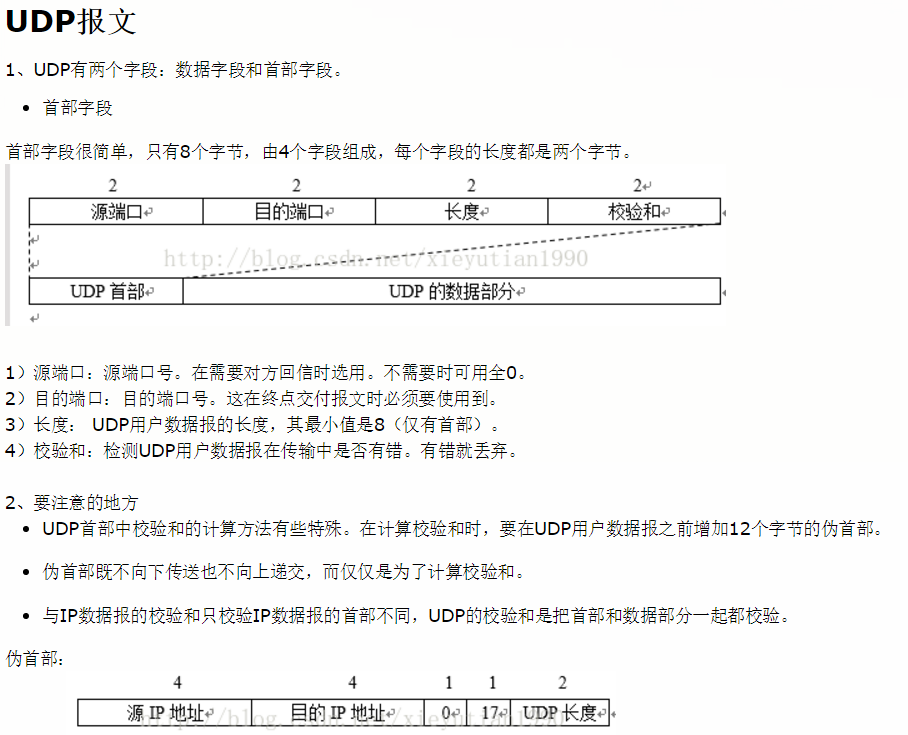
(3) 各层之间相对独立，一种高层协议可以放在多种低层协议上运行



## TCP/IP四层模型

|  |  |  |
| --- | --- | --- |
| 应用层 |  | 数据的表示及与应用程序的沟通 |
| 传输层 | TCP、UDP | 传送数据并确保数据已被有效传输 |
| 网络层 | IP | 路由功能 |
| 物理链路层 |  | 完成数据传输 |

## UDP和TCP报文



**TCP报文**



**32位端口号：**

源端口和目的端口各占16位，2的16次方等于65536，看端口的命令：netstat。

**32位序号：**

也称为顺序号（Sequence Number），简写为SEQ，

**32位确认序号**：

也称为应答号（Acknowledgment Number），简写为ACK。在握手阶段，确认序号将发送方的序号加1作为回答。

**4位首部长度**：

这个字段占4位，它的单位时32位（4个字节）。本例值为7，TCP的头长度为28字节，等于正常的长度2 0字节加上可选项8个字节。，TCP的头长度最长可为60字节（二进制1111换算为十进制为15，15\*4字节=60字节）。

**6位标志字段**：

ACK 置1时表示确认号（为合法，为0的时候表示数据段不包含确认信息，确认号被忽略。

RST 置1时重建连接。如果接收到RST位时候，通常发生了某些错误。

SYN 置1时用来发起一个连接。

FIN 置1时表示发端完成发送任务。用来释放连接，表明发送方已经没有数据发送了。

URG 紧急指针，告诉接收TCP模块紧要指针域指着紧要数据。注：一般不使用。

PSH 置1时请求的数据段在接收方得到后就可直接送到应用程序，而不必等到缓冲区满时才传送。注：一般不使用。

**16位检验和：**

检验和覆盖了整个的TCP报文段： TCP首部和TCP数据。这是一个强制性的字段，一定是由发端计算和存储，并由收端进行验证。

**16位紧急指针：**

**注：**一般不使用。

只有当U R G标志置1时紧急指针才有效。紧急指针是一个正的偏移量，和序号字段中的值相加表示紧急数据最后一个字节的序号。

**可选与变长选项**：

通常为空，可根据首部长度推算。用于发送方与接收方协商最大报文段长度（MSS），或在高速网络环境下作窗口调节因子时使用。首部字段还定义了一个时间戳选项。

最常见的可选字段是最长报文大小，又称为MSS (Maximum Segment Size)。每个连接方通常都在握手的第一步中指明这个选项。它指明本端所能接收的最大长度的报文段。1460是以太网默认的大小。

## TCP实现可靠性的方式

1. 三次握手建立连接；
2. 数据被分割成TCP认为最合适的长度发送；
3. 超时重传；
4. 数据差错检验；
5. 如果数据失序，TCP对其重新排序；
6. 丢弃重复数据；
7. 流量控制。

## 流量控制

所谓**流量控制**就是让发送方的发送速率不要太快，要让接收方来得及接收。流量控制往往指点对点通信量的控制，是接收端控制发送端发送数据的速率，以便接收端来得及接收。

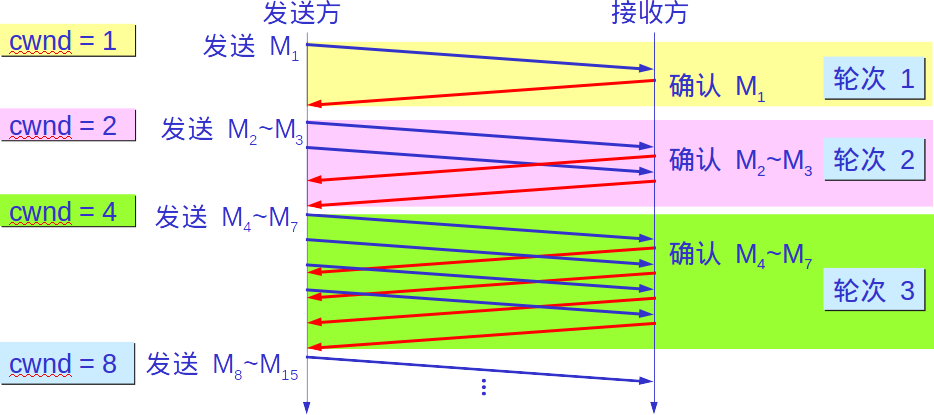
可以采用**滑动窗口**的方式，发送方的发送窗口不能超过接收方给出的接收窗口的数值。

## 拥塞控制

当对网络中某一资源的需求超过了该资源所能提供的可用部分，就会造成**拥塞**。而**拥塞控制**就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。以下是几种拥塞控制方法：

1. **慢开始**

慢开始的思路是不要一开始就把大量的数据字节注入到网络，而是由小到大逐渐增大拥塞窗口cwnd，因此发送窗口也由小到大逐渐增大，通常，先把拥塞窗口设置为一个最大的报文段MSS的数值，而在每收到一个对新的报文段的确认后，把拥塞窗口增加至多一个MSS的数值，以此来逐步增大发送方的拥塞窗口cwnd（对于每一个**传输轮次**，其实是指数增大的，因为每一轮里收到的报文段的确认都是上一轮的两倍，如下图所示），这样使得分组注入到网络的速率更加合理。



为了避免cwnd增长过大引起网络阻塞，还需设置一个慢开始门限ssthresh状态变量：

当cwnd<ssthresh时，使用上述的慢开始算法。

当cwnd>ssthresh时，改用拥塞避免算法。

当cwnd=ssthresh时，可以使用慢开始算法，也可以使用拥塞避免算法。

1. **拥塞避免**

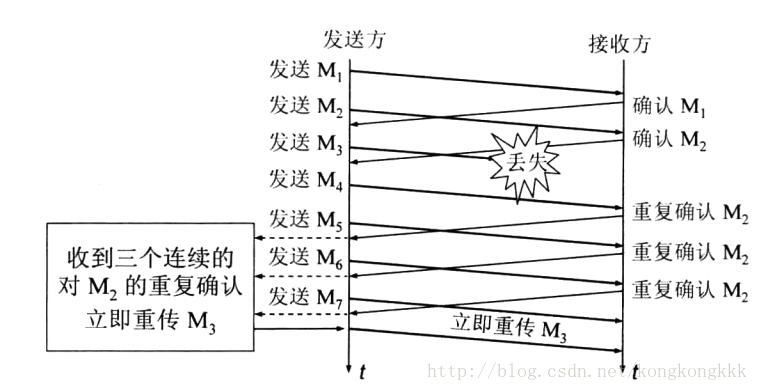
拥塞避免算法相对于慢开始算法的思路就是让每一轮次的拥塞窗口cwnd缓慢地增大，即，每一轮次后cwnd都只增大1，呈线性缓慢增长而不是指数增长。

由上可知，当cwnd>ssthresh后，改用拥塞避免算法，使得cwnd呈线性缓慢增加，当增大到使得网络出现阻塞时，则将ssthresh重新设置为此时cwnd的一半，并使得cwnd重新设置为1，执行慢开始算法。

上面两种算法合起来就是**AIMD算法（加法增大乘法减小）**。**乘法减小**指的是，不论是慢开始阶段还是拥塞避免阶段，只要出现网络拥塞（超时未确认）就将ssthresh设置为当前拥塞窗口的一半。**加法增大**指的就是拥塞避免算法的缓慢增大cwnd值。

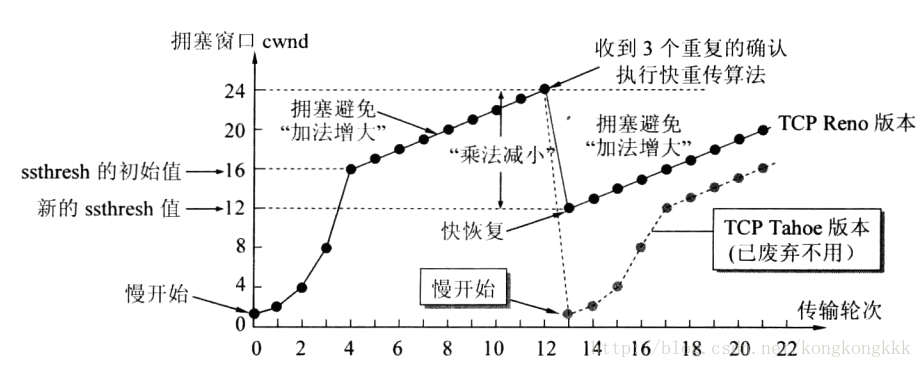
1. **快重传**

快重传算法首先要求接受方每收到一个失序的报文段后就立即发出重复确认，而不要等待自己发送数据时才稍待确认，并规定发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必等待设置的重传计时器到期。因此快重传后是整个网络的吞吐量提高约20%。



1. **快恢复**

与快重传配合使用的是快恢复算法，当发送方连续收到三个重复确认，就执行“乘法减小”算法，把慢开始门限ssthresh减半。这是为了预防网络发生拥塞。但接下来并不会执行慢开始算法。因为发送方现在认为网络很可能没有发生拥塞，因此与慢开始算法不同之处在于此时并不把拥塞窗口cwnd设置为1，而是设置为ssthresh减半后的数值，然后开始执行拥塞避免算法，是拥塞窗口缓慢线性地增大。



# Linux

## Linux文件属性

**drwxr-xr-x 2 root root 48 2013-11-27 16:34 test/**

* 第一列：文件类型和权限
* 第1个字符

"-"：普通文件；

"d"：目录,字母”d”,是dirtectory(目录)的缩写；

"l"： 符号链接。请注意,一个目录或者说一个文件夹是一个特殊文件,这个特殊文件存放的是其他文件和

文件夹的相关信息；

"b"： 块设备文件；

"c"： 字符设备文件；

"s"： 数据接口文件，套接字文件；

"p"： 管道文件。

* 第2~4个字符：文件所有者的权限
* 第5~7个字符：同用户者用户权限
* 第8~9个字符：其他用户权限  
  rwx：分别表示读、写、可执行权限
* 第二列：表示有多少文件名连接到此节点
* 第三列：文件的所有者
* 第四列：文件的所属用户组
* 第五列：文件大小，单位为B
* 第六列：文件的创建日期或者最近的修改日期
* 第七列：文件名，文件名前多一个”.”，则代表这个文件是“隐藏文件”。

## 改变文件属性和权限

* chgrp：改变文件所属用户组
* chown：改变文件所有者
* chmod：改变文件的权限

## 根目录

与开机系统相关；故以下目录不可与根目录分开：

* /etc：配置文件
* /bin：重要执行文件
* /dex：所需要的设备文件
* /lib：执行文件所需的函数库与内核所需的模块
* /sbin：重要的系统执行文件

另外：

* /usr：UNIX Software Resource，存放系统软件资源
* /var：variable，与系统运作过程有关。主要针对常态性变动的文件，包括缓存（cache）、登录文件（log file）以及某些软件运行所产生的文件，包括程序文件（lock file, run file），或者例如MySQL数据库的文件等。

## 常用命令

与开机系统相关；故以下目录不可与根目录分开：

* su：切换用户；sudo：请求root用户权限；
* /cd：切换目录；
* /pwd：显示目前所在目录；
* /mkdir：新建新目录；
* /rmdir：删除空目录；
* $PATH：执行文件路径的变量；
* cp：复制；rm：删除；mv：移动；
* basename：取得最后的文件名；dirname：取得目录名；
* cat：从第一行显示文件内容；tac：从最后一行显示；nl：显示的时候带上行号；more：一页一页地显示文件内容；less：与more类似，但可以往前翻页；head：只看头几行；tail：只看结尾几行；od：以二进制的方式读取文件内容；
* touch：创建空文件 或者 修改文件时间；
* chgrp：改变文件所属用户组；chown：改变文件所有者；chmod：改变文件的权限；
* which：寻找“执行文件”；whereis：寻找特定文件，通过数据库查找；locate：寻找特定文件，可匹配局部文件名，通过数据库查找；find，寻找文件，在硬盘中查找；
* df：列出文件系统的整体磁盘使用量；du：评估文件系统的磁盘使用量；

## 统计某个文件中某个单词个数

cat [filepath] | grep –o [word] | wc –l

cat 用于输出文件内容；

grep –o选项用于输出所有匹配的[word]，分行输出；

wc –l选项用于输出行数 –c用于输出字节数 –w用于输出单词数。

## 判断上一条指令是否执行成功

每一条基本命令执行后都有一个返回码，该返回码是用$?表示，执行成功的返回码是0：

if [ $? -ne 0 ]; then

echo "fail"

else

echo "success"

fi

-eq　　等于 | -ne　　不等于 | -gt　　大于 | -lt　　小于 | ge　　大于等于 | le　　小于等于

# 工具

## MQTT的特点

非常小的通信开销（最小的消息大小为 2 字节）；支持各种流行编程语言（包括C，Java，Ruby，Python 等等）且易于使用的客户端；支持发布 / 预定模型，简化应用程序的开发；提供三种不同消息传递等级，让消息能按需到达目的地，适应在不稳定工作的网络传输需求；使用发布/订阅消息模式，提供一对多的消息发布，解除应用程序耦合；

## 为什么用Git？其他版本控制工具？

作用：备份、协调开发

其他工具：CVS、SVN等

* SVN版本库安全性很差：1、SVN版本库服务器端历史数据被篡改，或者硬盘故障导致历史数据被篡改时， 客户端很难发现。管理员的备份也会被污染。2、SVN作为集中式版本控制系统，存在单点故障的风险。备份版本库的任务非常繁重。Git在这方面完胜SVN。首先Git是分布式版本控制系统，每个用户都相当于一份备份， 管理员无需为数据备份而担心。再有Git中包括提交、文件内容等都通过SHA1哈希保证数据的完整性， 任何恶意篡改历史数据都会被及时发现从而被挫败。
* 对合并更好的支持：1、因为Git基于对内容的追踪而非对文件名追踪，所以遇到一方或双方对文件名更改时， Git能够很好进行自动合并或提供工具辅助合并。而SVN遇到同样问题时会产生树冲突， 解决起来很麻烦。2、Git的基于DAG（有向非环图）的设计比SVN的线性提交提供更好的合并追踪， 避免不必要的冲突，提高工作效率。这是开发者选择Git、抛弃SVN的重要理由。
* 选择SVN的场景：SVN具有的悲观锁的功能，能够实现一个用户在编辑时对文件进行锁定，阻止多人同时编辑 一个文件。这一悲观锁的功能是 Git 所不具备的。对于以二进制文件 （Word文档、PPT演示稿） 为主的版本库，为避免多人同时编辑造成合并上的困难， 建议使用SVN做版本控制。

# 其他

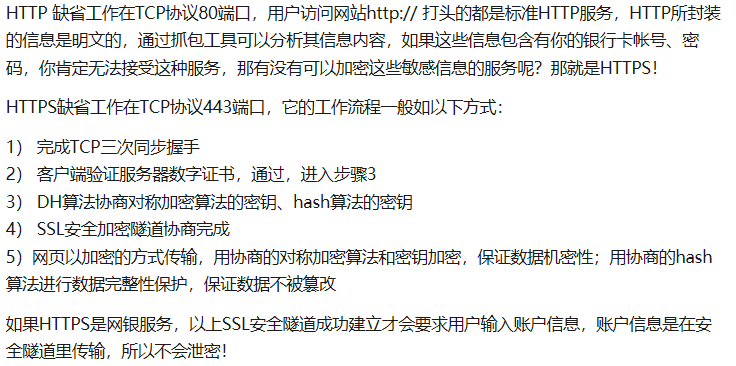
## 优缺点

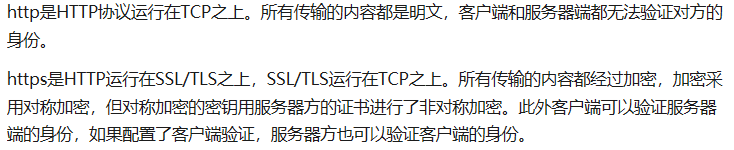
做事缺乏必要的耐心、性格有些急躁以及喜欢墨守成规、不善于与人沟通交往等缺点

## 职业目标与未来规划

作为一名全日制的学生，短期内的目标是能够快速融入工作环境，适应工作氛围，能够把学校里学到的东西，以及平时积累的东西能够运用到工作中，为公司创造价值，同时也希望能够在工作中继续学习，积累在学校积累不到的经验，提升自己，更好地为公司效力，为社会创造价值。

1. 数据库连接池的作用，主要用于解决什么问题
2. 对于简单的数据库应用，在需要访问数据库时，新创建一个连接，用完后就关闭；对于复杂数据库应用，频繁建立、关闭连接会极大降低系统性能；
3. 连接池在内部对象池中维护一定数量的数据库连接，并对外暴露数据库连接获取和返回方法。Eg:外部使用者通过getconnection方法获取连接，使用完毕后再通过releaseconnection方法将连接返回，此时连接并没有关闭，而是由连接池管理器回收，并为下一次使用做好准备；
4. 资源重用(避免频繁创建释放连接引起的大量性能开销)/更快的系统响应速度(数据库连接池的初始化工作提前完成，直接利用可用连接)/新的资源分配手段(避免某一应用独占所有数据库资源)/统一的连接管理,避免连接泄露(根据预先的连接占用超时设定，强制收回被占用连接)
5. https和http的区别





1. 设计模式、单例模式
2. 设计模式是一套被反复使用、多数人知晓的、经过分类的、代码设计经验的总结。使用设计模式的目的：为了代码可重用性，让代码更容易被他人理解、保证代码可靠性。
3. a.**创建型模式**：单例模式、抽象工厂模式、建造者模式、工厂模式、原型模式

b.**结构型模式**：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式

c.**行为型模式**：模板方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式、状态模式、策略模式、职责链模式(责任链模式)、访问者模式

1. **单例模式**：核心结构中只包含一个被称为单例的特殊类，即一个类只有一个对象实例。

**要点：**a.某个类只能有一个实例；

b.它必须自行创建这个实例；

c.它必须自行向整个系统提供这个实例；

**资源管理器常常设置成单例模式：**

**软件外部资源**：多个打印任务，只能有一个正在工作的任务；若干通信端口，避免一个通信端口同时被两个请求同时调用。

**软件内部资源**：大多数软件都有属性文件存放系统配置，这样的系统应当由一个对象来管理一个属性文件。

**实现：**a.单例模式的类只提供私有的构造函数；

b.类定义中含有一个该类的静态私有对象；

c.该类提供了一个静态的公有函数用于创建或获取它本身的静态私有对象。