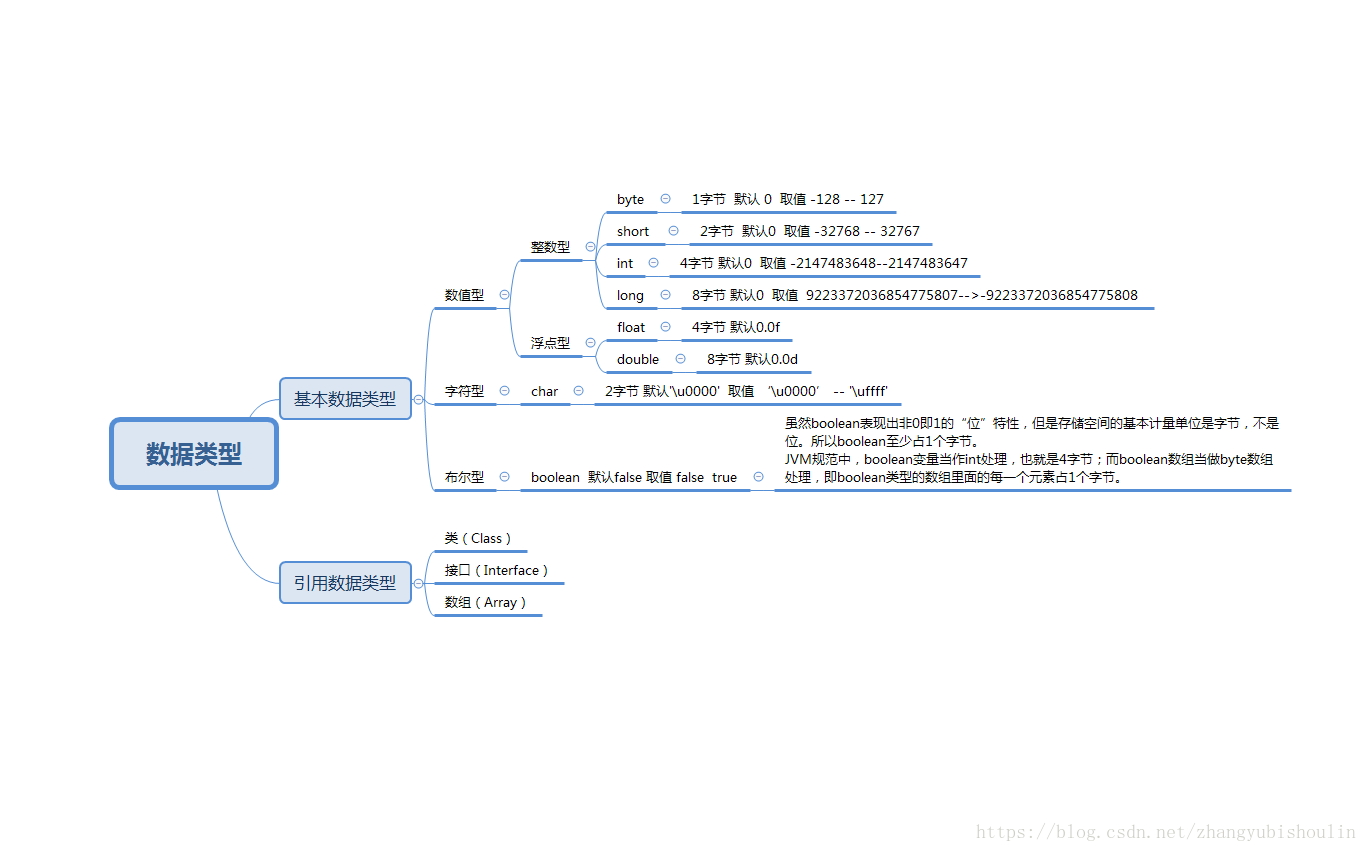
# JAVA基础

1. JAVA中的几种基本数据类型是什么，各自占用多少字节。



1. String类能被继承吗，为什么。

不能, final类型

1. String，Stringbuffer，StringBuilder的区别。

运行速度:StringBuilder >StringBuffer >String

线程安全:StringBuilder是线程不安全的，而StringBuffer是线程安全的

1. ArrayList和LinkedList有什么区别。

ArrayList : 基于数组实现的非线程安全的集合。查询元素快，插入，删除中间元素慢。

LinkedList : 基于链表实现的非线程安全的集合。查询元素慢，插入，删除中间元素快。

1. 讲讲类的实例化顺序，比如父类静态数据，构造函数，字段，子类静态数据，构造函数，字段，当new的时候，他们的执行顺序。

此题考察的是类加载器实例化时进行的操作步骤（加载–>连接->初始化）。

父类静态变量、

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。

1. 用过哪些Map类，都有什么区别，HashMap是线程安全的吗,并发下使用的Map是什么，他们内部原理分别是什么，比如存储方式，hashcode，扩容，默认容量等。

HashMap线程不安全，数组+链表+红黑树

Hashtable线程安全，锁住整个对象，数组+链表

ConccurentHashMap线程安全，CAS+同步锁，数组+链表+红黑树

HashMap的key，value均可为null，其他两个不行。

1. JAVA8的ConcurrentHashMap为什么放弃了分段锁，有什么问题吗，如果你来设计，你如何设计。

弃用原因：

1)加入多个分段锁浪费内存空间。

2)生产环境中， map 在放入时竞争同一个锁的概率非常小，分段锁反而会造成更新等操作的长时间等待。

3)为了提高 GC 的效率

新的同步方案

既然弃用了分段锁， 那么一定由新的线程安全方案， 我们来看看源码是怎么解决线程安全的呢？（源码保留了segment 代码， 但并没有使用）

首先通过 hash 找到对应链表过后， 查看是否是第一个object， 如果是， 直接用cas原则插入，无需加锁。然后如果不是链表第一个object， 则直接用链表第一个object加锁，这里加的锁是synchronized，虽然效率不如 ReentrantLock， 但节约了空间，这里会一直用第一个object为锁， 直到重新计算map大小， 比如扩容或者操作了第一个object为止。

1. 有没有有顺序的Map实现类，如果有，他们是怎么保证有序的。

Hashmap和Hashtable 都不是有序的。

TreeMap和LinkedHashmap都是有序的。（TreeMap默认是key升序，LinkedHashmap默认是数据插入顺序）

TreeMap是基于比较器Comparator来实现有序的。

LinkedHashmap是基于链表来实现数据插入有序的。

1. 抽象类和接口的区别，类可以继承多个类么，接口可以继承多个接口么,类可以实现多个接口么。

区别：

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。

6、抽象方法只能申明，不能实现。abstract void abc();不能写成abstract void abc(){}。

7、抽象类里可以没有抽象方法

8、如果一个类里有抽象方法，那么这个类只能是抽象类

9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

10、接口可继承接口，并可多继承接口，但类只能单根继承。

类不能继承多个类

接口可以继承多个接口

类可以实现多个接口

1. 继承和聚合的区别在哪。

**继承**

指的是一个类继承另外的一个类的功能，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。

**聚合**

聚合体现的是整体与部分、拥有的关系，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；比如计算机与CPU、公司与员工的关系等；

1. IO模型有哪些，讲讲你理解的nio ，他和bio，aio的区别是啥，谈谈reactor模型。

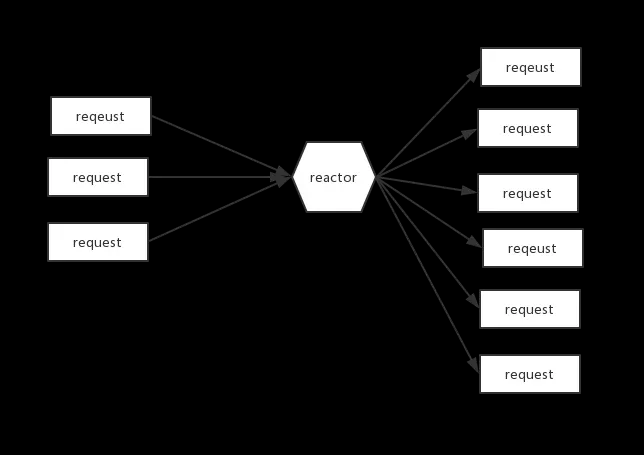


reactor是什么？

1)事件驱动

2)可以处理一个或多个输入源

3)通过Service Handle同步的将输入事件采用多路复用分发给相应的Request Handler(一个或多个）处理



1. 反射的原理，反射创建类实例的三种方式是什么。

//创建Class对象的方式一：(对象.getClass())，获取类中的字节码文件

Class class1 = p1.getClass();

//创建Class对象的方式二：(类.class:需要输入一个明确的类，任意一个类型都有一个静态的class属性)

Class class3 = Person.class;

//创建Class对象的方式三：(forName():传入时只需要以字符串的方式传入即可)

//通过Class类的一个forName（String className)静态方法返回一个Class对象，className必须是全路径名称；

//Class.forName()有异常：ClassNotFoundException

Class class4 = Class.forName("cn.xbmchina.Person");

1. 反射中，Class.forName和ClassLoader区别 。

Class.forName(className)方法，内部实际调用的方法是 Class.forName(className,true,classloader);

第2个boolean参数表示类是否需要初始化， Class.forName(className)默认是需要初始化。

一旦初始化，就会触发目标对象的 static块代码执行，static参数也也会被再次初始化。

ClassLoader.loadClass(className)方法，内部实际调用的方法是 ClassLoader.loadClass(className,false);

第2个 boolean参数，表示目标对象是否进行链接，false表示不进行链接，由上面介绍可以，不进行链接意味着不进行包括初始化等一些列步骤，那么静态块和静态对象就不会得到执行

1. 描述动态代理的几种实现方式，分别说出相应的优缺点。

原理区别：

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP

2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

如何强制使用CGLIB实现AOP？

（1）添加CGLIB库，SPRING\_HOME/cglib/\*.jar

（2）在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>

JDK动态代理和CGLIB字节码生成的区别？

（1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类

（2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法

因为是继承，所以该类或方法最好不要声明成final

1. 动态代理与cglib实现的区别。
2. 为什么CGlib方式可以对接口实现代理。

Cglib是一个优秀的动态代理框架，它的底层使用ASM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。

1. final的用途。

1、被final修饰的类不可以被继承

2、被final修饰的方法不可以被重写

3、被final修饰的变量不可以被改变（切记不可变的是变量的引用而非引用指向对象的内容。）

4、被final修饰的方法，JVM会尝试为之寻求内联，这对于提升Java的效率是非常重要的。因此，假如能确定方法不会被继承，那么尽量将方法定义为final的，具体参见运行期优化技术的方法内联部分

5、被final修饰的常量，在编译阶段会存入调用类的常量池中，具体参见类加载机制最后部分和Java内存区域

1. 写出三种单例模式实现 。

1 饿汉式

2 懒汉式

3 静态内部类实现

1. 如何在父类中为子类自动完成所有的hashcode和equals实现？这么做有何优劣。

父类的equals不一定满足子类的equals需求。比如所有的对象都继承Object，默认使用的是Object的equals方法，在比较两个对象的时候，是看他们是否指向同一个地址。

但是我们的需求是对象的某个属性相同，就相等了，而默认的equals方法满足不了当前的需求，所以我们要重写equals方法。

如果重写了equals 方法就必须重写hashcode方法，否则就会导致map等集合操作对象时出现问题。

1. 请结合OO设计理念，谈谈访问修饰符public、private、protected、default在应用设计中的作用。

OO设计理念：封装、继承、多态

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。所以我们可以通过public、private、protected、default 来进行访问控制



1. 深拷贝和浅拷贝区别。

深拷贝对引用数据类型的成员变量的对象图中所有的对象都开辟了内存空间；而浅拷贝只是传递地址指向，新的对象并没有对引用数据类型创建内存空间。

1. 数组和链表数据结构描述，各自的时间复杂度。

数组和链表的区别：

1、从逻辑结构角度来看：

数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况。当数据增加时，可能超出原先定义的元素个数；当数据减少时，造成内存浪费。

链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删除数据项时，需要移动其它数据项）

2、数组元素在栈区，链表元素在堆区；

3、从内存存储角度来看：

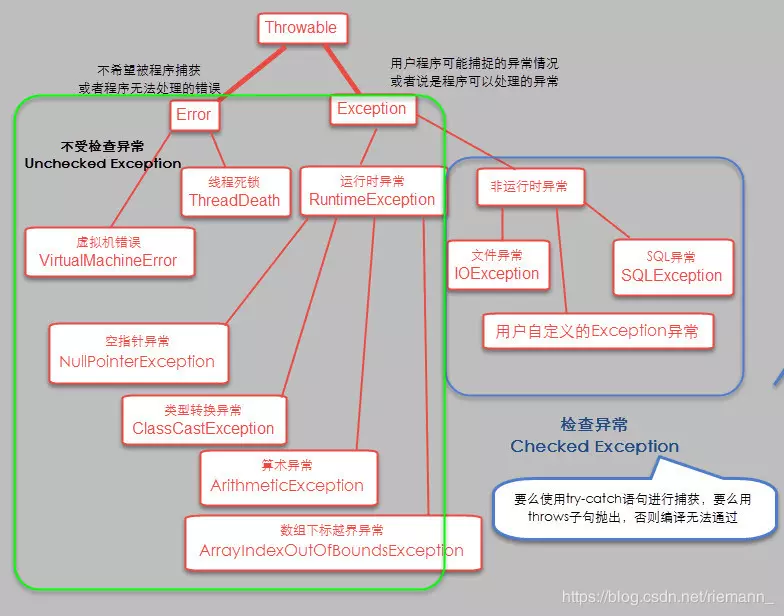
(静态)数组从栈中分配空间, 对于程序员方便快速,但自由度小。

链表从堆中分配空间, 自由度大但申请管理比较麻烦。

数组利用下标定位，时间复杂度为O(1)，链表定位元素时间复杂度O(n)；

数组插入或删除元素的时间复杂度O(n)，链表的时间复杂度O(1)。

1. error和exception的区别，CheckedException，RuntimeException的区别。



1. 请列出5个运行时异常。

NullPointerException

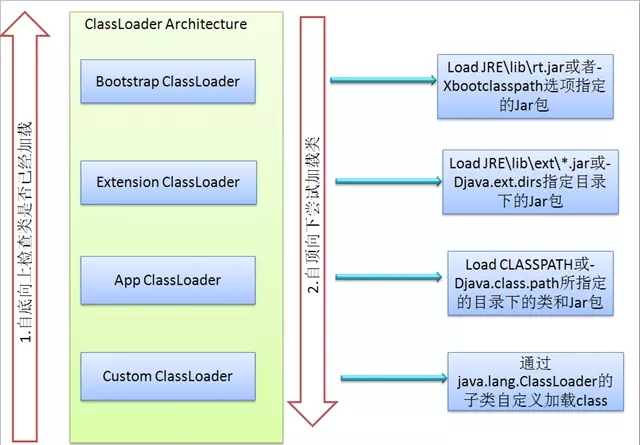
IndexOutOfBoundsException

ClassCastException

FileNotFoundException

ArithmeticException

1. 在自己的代码中，如果创建一个java.lang.String类，这个类是否可以被类加载器加载？为什么。



加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是说当发现这个类没有的时候会先去让自己的父类去加载，父类没有再让儿子去加载，那么在这个例子中我们自己写的String应该是被Bootstrap ClassLoader加载了，所以App ClassLoader就不会再去加载我们写的String类了，导致我们写的String类是没有被加载的。

1. 说一说你对java.lang.Object对象中hashCode和equals方法的理解。在什么场景下需要重新实现这两个方法。

对于equals()与hashcode()，比较通用的规则：

* 1. 两个obj，如果equals()相等，hashCode()一定相等

②两个obj，如果hashCode()相等，equals()不一定相等

父类的equals不一定满足子类的equals需求。比如所有的对象都继承Object，默认使用的是Object的equals方法，在比较两个对象的时候，是看他们是否指向同一个地址。

但是我们的需求是对象的某个属性相同，就相等了，而默认的equals方法满足不了当前的需求，所以我们要重写equals方法。

如果重写了equals 方法就必须重写hashcode方法，否则就会降低map等集合的索引速度。

1. 在jdk1.5中，引入了泛型，泛型的存在是用来解决什么问题。

新增加了泛型的技术，这就将上述向下转型的问题消灭在了萌芽之中。

泛型的核心意义在于：类在进行定义的时候可以使用一个标记，此标记就表示类中属性或者方法以及参数的类型，标记在使用的时候，才会去动态的设置类型。

1. 有没有可能2个不相等的对象有相同的hashcode。

有可能，最简单的方法，百分百实现的方式就是重写hascode();

1. Java中的HashSet内部是如何工作的。

HashSet 的内部采用 HashMap来实现。由于 Map 需要 key 和 value，所以HashSet中所有 key 的都有一个默认 value。类似于 HashMap，HashSet 不允许重复的 key，只允许有一个null key，意思就是 HashSet 中只允许存储一个 null 对象。

1. 什么是序列化，怎么序列化，为什么序列化，反序列化会遇到什么问题，如何解决。

什么是序列化？

序列化：把对象转换为字节序列的过程称为对象的序列化。

反序列化：把字节序列恢复为对象的过程称为对象的反序列化

什么情况下需要序列化？

当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；

当你想用套接字在网络上传送对象的时候；

当你想通过RMI传输对象的时候；

如何实现序列化？

实现Serializable接口即可

注意事项：

transient 修饰的属性，是不会被序列化的

静态static的属性，他不序列化。

实现这个Serializable 接口的时候，一定要给这个 serialVersionUID 赋值

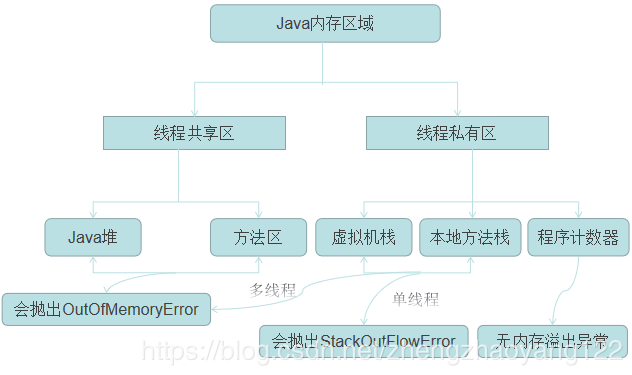
1. java8的新特性。

# JVM知识

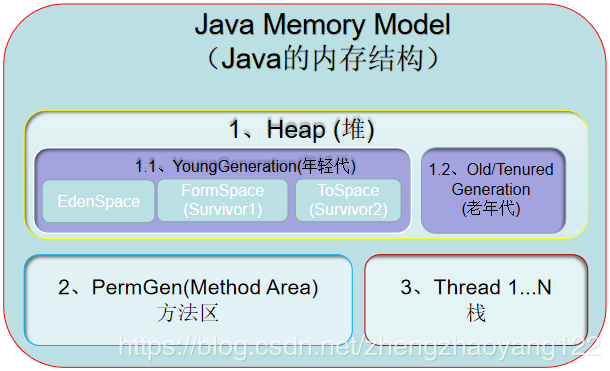
1. 什么情况下会发生栈内存溢出。

【1】、线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常。递归的调用一个简单的方法，不断累积就会抛出 StackOverflowError 异常。

【2】、如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。无限循环的创建线程，并对每个线程增加内存。则会抛出 OutOfMemoryError 异常。



1. JVM的内存结构，Eden和Survivor比例。



JVM内存结构主要由三部分组成：堆空间、方法区和栈。堆内存是 JVM虚拟机中最大的一部分，它由年轻代和老年代组成。而年轻代有被分为三部分，Eden空间、From Survivor空间和To Survivor空间。默认情况下年轻代按照8：1：1的比例分配。方法区存储类信息、常量、静态变量等数据，是线程共享的区域。为了与 Java堆区分，方法区有一个别名 Non-Heap（非堆）。栈分为Java虚拟机栈和本地方法栈主要用于方法的执行。

1. JVM内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为Eden和Survivor。

●  所谓新生代和老年代是针对分代（存活年龄）收集算法来定义的，这样就针对性的进行 GC。例如：MinorGC 回收清理新生代，频率相对较高。MajorGC 主要清理老年代，频率相对较少。FullGC 是指清理这个堆空间包括年轻代和持久代等。这样非常高效的提高了GC的效率，提交系统性能。

 ●  新生代分为 Eden和Survivor 两个分区，当新对象时首先会分配到 Eden区中（特殊情况：大对象会直接放入老年代，大对象指需要连续的大量内存空间的对象），当 Eden没有足够空间的时候就会触发 MinorGC。如果GC后，对象还活着，并且能被Survivor 空间接受，那么就会移至 Survivor中，并将其年龄设为1，对象在 Survivor中每熬过一次GC，年龄就+1，当年龄增加到一定程度（默认为15，可设置）时，就会被晋升到老年代。

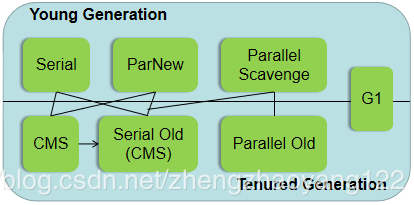
1. JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代，说说你知道的几种主要的JVM参数。

Java GC 机制主要完成3件事：确定哪些内存需要回收，确定什么时候需要执行 GC，如何执行 GC。



1. 你知道哪几种垃圾收集器，各自的优缺点，重点讲下cms和G1，包括原理，流程，优缺点。

目前比较常用的垃圾收集器和它们直接搭配使用的情况如下，上面是新生代收集器，下面则是老年代收集器，根据不同业务场景进行选取。



 ●  Serial 收集器：是一个新生代收集器，单线程执行，使用复制算法。它在进行垃圾收集时，必须暂停其他所有的工作线程(用户线程)也就是 Stop The World。是JVM Client 模式下默认的新生代收集器。对于限定单个 CPU 的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

 ●  ParNew收集器：可看作 Serial收集器的多线程版本，除使用多条线程进行垃圾收集之外，其余行为与 Serial收集器一样。 使用方式可以使用-XX:+UseConcMarkSweepGC，或者是使用-XX:+UseParNewGC来强制开启，可以通过-XX:ParallelGCThreads 来调整或者限制垃圾收集的线程数量。

 ●  Parallel Scavenge收集器：也是新生代收集器，也是使用复制算法的收集器，还是并行多线程收集器。Parallel Scavenge 收集器的特点是它关注的点与其他收集器不同，CMS 等收集器的关注点是尽可能地缩短垃圾收集的时间，从而降低用户线程的停顿时间，而 parallel Scavenge收集器的目标则是达到一个可控制的吞吐量。吞吐量= 程序运行时间/(程序运行时间 + 垃圾收集时间)，虚拟机总共运行了100分钟。其中垃圾收集花掉1分钟，那吞吐量就是99%。  
Parallel Scavenge 提供了两个参数用来精确控制，分别是控制最大垃圾收集停顿时间的 -XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的 -XX:GCTimeRatio参数。

MaxGCPauseMillis参数允许的值是一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。不过大家不要认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集300MB新生代肯定比收集500MB快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。  
GCTimeRatio参数的值应当是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。  
Parallel Scavenge 收集器也经常称为“吞吐量优先”收集器。除上述两个参数之外，Parallel Scavenge 收集器还有一个参数-XX:+UseAdaptiveSizePolicy 值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）

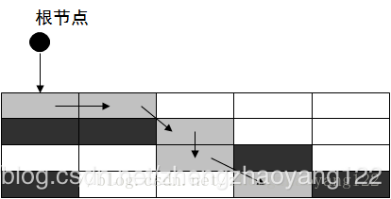
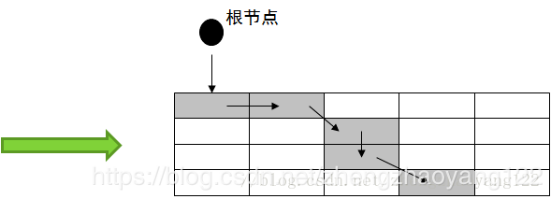
 ●  Serial Old：是 Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记－整理”算法。这个收集器的主要意义也是在于给 Client 模式下的虚拟机使用。如果在 Server模式下，那么它主要还有两大用途：一种用途是在 JDK 1.5以及之前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途就是作为 CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。这两点都将在后面的内容中详细讲解。  
 ●  Parallel Old：是Parallel Scavenge 收集器的老年代版本，使用多线程和“标记－整理”算法。这个收集器是在 JDK1.6中才开始提供的，在此之前，新生代的 Parallel Scavenge收集器一直处于比较尴尬的状态。原因是，如果新生代选择了Parallel Scavenge收集器，老年代除了 Serial Old（PS MarkSweep）收集器外别无选择（还记得上面说过 Parallel Scavenge 收集器无法与 CMS收集器配合工作吗？）。由于老年代 Serial Old收集器在服务端应用性能上的“拖累”，使用了 Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS的组合“给力”。  
直到 Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及 CPU资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。  
 ●  CMS（Concurrent Mark Sweep）收集器：是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java应用集中在互联网站或者 B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。  
从名字（包含“Mark Sweep”）上就可以看出，CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：① 初始标记（CMS initial mark）② 并发标记（CMS concurrent mark）③ 重新标记（CMS remark）④ 并发清除（CMS concurrent sweep）其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下 GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行 GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。  
CMS 收集器无法处理浮动垃圾（Floating Garbage），可能出现 “Concurrent Mode Failure” 失败而导致另一次 Full GC 的产生。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在 JDK 1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数 -XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在 JDK 1.6中，CMS 收集器的启动阈值已经提升至92%。要是 CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用 Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数 -XX:CMSInitiatingOccupancyFraction 设置得太高很容易导致大量 “Concurrent Mode Failure” 失败，性能反而降低。  
CMS是一款基于“标记—清除”算法实现的收集器，如果读者对前面这种算法介绍还有印象的话，就可能想到这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。为了解决这个问题，CMS 收集器提供了一个-XX:+UseCMSCompactAtFullCollection 开关参数（默认就是开启的），用于在 CMS收集器顶不住要进行 FullGC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的 Full GC后，跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。  
 ●  G1收集器：是一款面向服务端应用的垃圾收集器。HotSpot 开发团队赋予它的使命是（在比较长期的）未来可以替换掉 JDK 1.5中发布的 CMS收集器。与其他 GC收集器相比，G1具备如下特点。  
   ☞ 并行与并发：G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU（CPU或者CPU核心）来缩短 Stop-The-World 停顿的时间，部分其他收集器原本需要停顿 Java线程执行的 GC动作，G1 收集器仍然可以通过并发的方式让 Java 程序继续执行。  
   ☞ 分代收集：与其他收集器一样，分代概念在 G1中依然得以保留。虽然 G1可以不需要其他收集器配合就能独立管理整个 GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次 GC的旧对象以获取更好的收集效果。  
   ☞ 空间整合：与 CMS 的“标记—清理”算法不同，G1从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个 Region之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。  
   ☞ 可预测的停顿：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

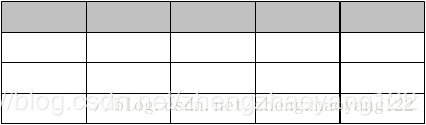
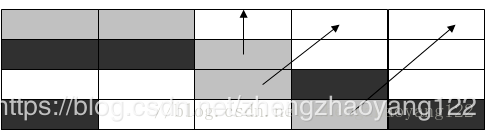
在 G1 之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java 堆的内存布局就与其他收集器有很大差别，它将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region（不需要连续）的集合。

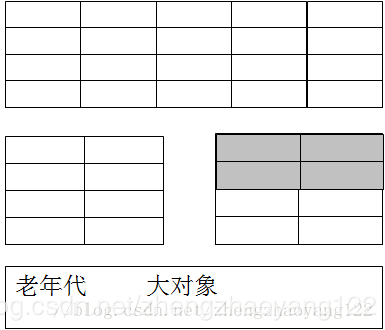
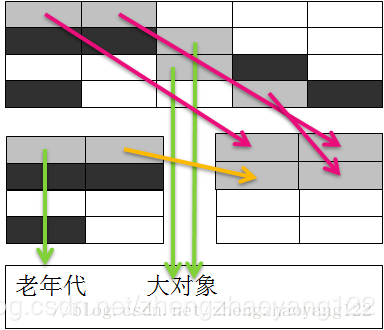
1. 垃圾回收算法的实现原理。

  1）、标记-清除：是现代垃圾回收算法的基本思想。标记-清除将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始可达的对象。因此，未标记的对象就是未被引用的对象。然后，在清除阶段，清除所有未被标记的对象。



   2）、标记-压缩：标记-压缩算法适合用于存活对象较多的场合，如老年代。在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，并不是简单的清理未标记的对象，而是将所有的存活对象压缩到内存的另一端，之后，清理边界外所有的对象。  


  3）、复制算法：①、与标记算法相比，复制算法是一种相对高效的回收方法。  
     ②、不适用存活对象较多的场合，比如老年代。        
     ③、将原有的内存空间分为两块，每次只是用一块，在垃圾回收时，将正在适用的内存存活对象复制一份到未使用的内存中，之后，清除正在适用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。（浪费空间）  


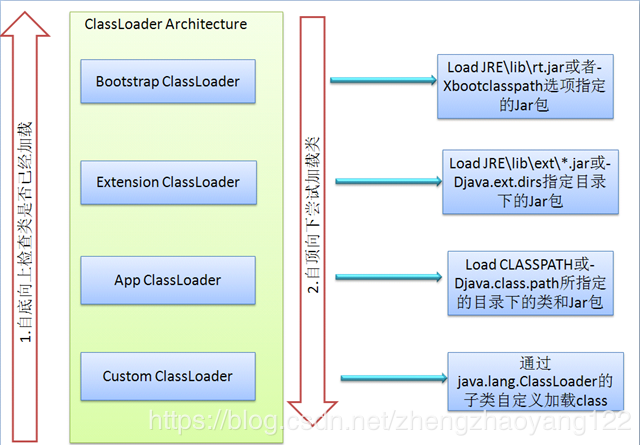
1. 当出现了内存溢出，你怎么排错。

首先分析是什么类型的内存溢出，对应的调整参数或者优化代码。

1. JVM内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存等。

【1】重排序：在执行程序时，为了提高性能，编译器和处理器会对指令做重排序。（譬如：a=1;b=2;重排后b=2;a=1;）。  
  ◀ 编译器优化重排序：编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。  
  ◀ 指令级并行的重排序：如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。  
  ◀ 内存系统的重排序：处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。  
       
【2】内存屏障（Memory Barrier）：又称内存栅栏，为了保障执行顺序和可见性的一条 cpu 指令。  
  ◀ 保障执行顺序：编译器和CPU能够重排序指令，保证最终相同的结果，尝试优化性能。插入一条Memory Barrier会告诉编译器和CPU：不管什么指令都不能和这条Memory Barrier指令重排序。  
  ◀ Memory Barrier 保证可见性是指：强制刷出各种 CPU cache，如一个 Write-Barrier（写入屏障）将刷出所有在 Barrier 之前写入cache 的数据，因此，任何 CPU上 的线程都能读取到这些数据的最新版本。例如：Volatile 就是基于 Memory Barrier 实现的。  
【3】happen-before：操作之间的内存可见性。在JMM中，如果一个操作的执行结果需要对另一个操作可见，那么这两个操作之间必须要存在 happens-before 关系，这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。  
  ◀ 程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。  
  ◀ 监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。  
  ◀ volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。  
  ◀ 传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。  
【4】主内存：共享变量存储的区域即是主内存。  
【5】工作内存：每个线程copy的本地内存，存储了该线程以读/写共享变量的副本。

1. 简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。

【1】类的加载过程：JVM 将类加载过程分为三个步骤：装载（Load），链接（Link）和初始化(Initialize)链接又分为三个步骤：  
    ● 装载：查找并加载类的二进制数据；  
    ● 链接：验证：确保被加载类的正确性；  
                  准备：为类的静态变量分配内存，并将其初始化为默认值；  
                  解析：把类中的符号引用转换为直接引用；  
    ● 初始化：为类的静态变量赋予正确的初始值；  
【2】JVM 的类加载是通过 ClassLoader 及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：  
     

【3】打破双亲委派机制则不仅要继承 ClassLoader类，还要重写 loadClass 和 findClass方法。默认的 loadClass方法是实现了双亲委派机制的逻辑，即会先让父类加载器加载，当无法加载时才由自己加载。为了破坏双亲委派机制必须重写loadClass方法。

1. 讲讲JAVA的反射机制。

JAVA反射机制是在运行状态中，对于任意一个实体类，都能够获取这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 java语言的反射机制。

1. 你们线上应用的JVM参数有哪些。

|  |  |
| --- | --- |
| -server | Server模式启动 |
| -Xms6000M | 初始堆内存6000m |
| -Xmx6000M | 最大堆内存6000m |
| -Xmn500M | 最小堆内存500m |
| -XX:PermSize=500M | 永久代500m |
| -XX:MaxPermSize=500M | 最大永久代500m |
| -XX:SurvivorRatio=65536 | 设置年轻代中Eden区与Survivor区的比值 |
| -XX:MaxTenuringThreshold=0 | 设置垃圾最大年龄（在年轻代的存活次数）。如果设置为0的话，则年轻代对象不经过Survivor区直接进入年老代 |
| -Xnoclassgc | 关闭class的垃圾回收功能，即虚拟机加载的类，即便是不使用，没有实例也不会回收 |
| -XX:+DisableExplicitGC | 不响应 System.gc() 代码 |
| -XX:+UseParNewGC | 设置年轻代为并发收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此参数 |
| -XX:+UseConcMarkSweepGC | 即CMS收集，设置老年代为并发收集 |
| -XX:+UseCMSCompactAtFullCollection | 打开内存空间的压缩和整理，在Full GC后执行。可能会影响性能，但可以消除内存碎片 |
| -XX:CMSFullGCsBeforeCompaction=0 | 由于并发收集器不对内存空间进行压缩和整理，所以运行一段时间并行收集以后会产生内存碎片，内存使用效率降低。此参数设置运行0次Full GC后对内存空间进行压缩和整理，即每次Full GC后立刻开始压缩和整理内存 |
| -XX:+CMSClassUnloadingEnabled | 允许对类元数据进行回收 |
| -XX:+CMSParallelRemarkEnabled | 开启并行收集 |
| -XX:CMSInitiatingOccupancyFraction=90 | 允许90%的Survivor区被占用（JVM默认为50%）。提高对于Survivor区的使用率 |
| -XX:SoftRefLRUPolicyMSPerMB=0 | 软引用对象在最后一次被访问后能存活0毫秒（JVM默认为1000毫秒） |
| -XX:+PrintClassHistogram | 按下 Ctrl+Break 后打印堆内存中类实例的柱状信息，同JDK的 jmap -histo 命令 |
| -XX:+PrintGCDetails | 每次GC时打印详细信息 |
| -XX:+PrintGCTimeStamps | 打印每次GC的时间戳 |
| -XX:+PrintHeapAtGC | 在进行GC的前后打印出堆的信息 |
| -Xloggc:log/gc.log | 日志文件的输出路径 |

1. g1和cms区别,吞吐量优先和响应优先的垃圾收集器选择。

CMS 是以获取最短回收停顿时间为目标的收集器。基于标记-清除算法实现。比较占用 cpu 资源，切易造成碎片。

G1 是面向服务端的垃圾收集器，是 jdk9默认的收集器，基于标记-整理算法实现。可利用多核、多cpu，保留分代，实现可预测停顿，可控。

1. 怎么打出线程栈信息。

jps获得进程号

top -Hp 25606 获取本进程中所有线程的CPU耗时性能

printf '%x\n' 25608 得到线程的16进制值

jstack 25606 | grep 6408

1. 请解释如下jvm参数的含义：

-server -Xms512m -Xmx512m -Xss1024K  
-XX:PermSize=256m -XX:MaxPermSize=512m -  
XX:MaxTenuringThreshold=20XX:CMSInitiatingOccupancyFraction=80 -  
XX:+UseCMSInitiatingOccupancyOnly。

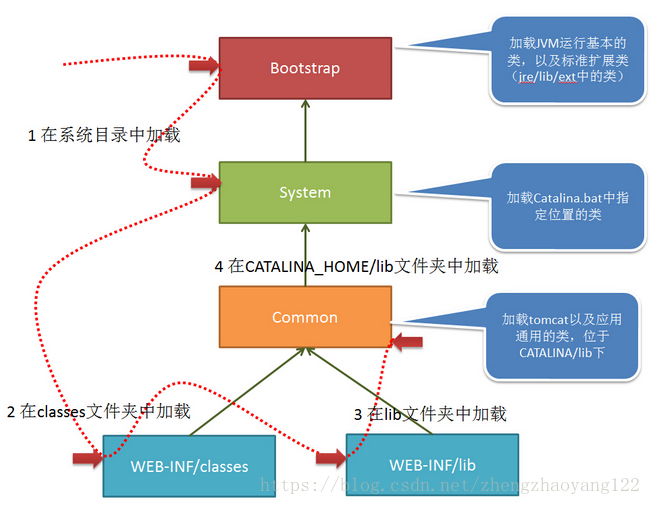
|  |  |
| --- | --- |
| **-server** | Server模式启动 |
| **-Xms512m** | 初始堆内存512m |
| **-Xmx512m** | 最大堆内存512m |
| **-Xss1024K** | 线程栈空间1m |
| **-XX:PermSize=256m** | 永久代256m |
| **-XX:MaxPermSize=512m** | 最大永久代512m |
| **-XX:MaxTenuringThreshold=20** | 最大转为老年代检查次数20 |
| **-XX:CMSInitiatingOccupancyFraction=80** | CMS回收开启时机：内存占用80% |
| **-XX:+UseCMSInitiatingOccupancyOnly** | 命令JVM不基于运行时收集的数据来启动CMS垃圾收集周期 |

# 开源框架知识

1. 简单讲讲tomcat结构，以及其类加载器流程，线程模型等。

☞ 模块组成结构：Tomcat的核心组件就Connector和Container，一个Connector+一个Container（Engine）构成一个Service，Service就是对外提供服务的组件，有了Service组件Tomcat就能对外提供服务了，但是光有服务还不行，还需要有环境让你提供服务才行，所以最外层的Server就是为Service提供了生存的土壤

 ☞  类加载器流程：Tomcat启动时，会创建以下4种类加载器



红色虚线表示：应用需要到某个类时，则会按照下面的顺序进行类加载。

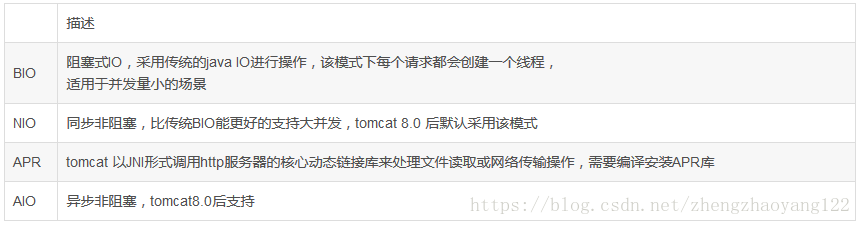
     ①、使用bootstrap引导类加载器加载

     ②、使用system系统类加载器加载

     ③、使用应用类加载器在WEB-INF/classes中加载

     ④、使用应用类加载器在WEB-INF/lib中加载

     ⑤、使用common类加载器在CATALINA\_HOME/lib中加载

 ☞  线程模型：支持以下四种线程模型。  


1. tomcat如何调优，涉及哪些参数 。

 ☞ Tomcat调优主要从四个方面考虑：1）、吞吐量。2）、Responsetime。 3）、Cpuload。 4）、MemoryUsage。

☞ 参数调优：

1）、Tomcat启动参数的优化：Tomcat 的启动参数位于tomcat的安装目录\bin目录下，如果你是Linux操作系统就是catalina.sh文件，如果你是Windows操作系统那么你需要改动的就是catalina.bat文件。

✔ Linux系统中catalina.sh文件中添加如下参数（重要参数随后说明）：

export JAVA\_OPTS="-server -Xms1400M -Xmx1400M -Xss512k -XX:+AggressiveOpts -XX:+UseBiasedLocking

-XX:PermSize=128M -XX:MaxPermSize=256M -XX:+DisableExplicitGC -XX:MaxTenuringThreshold=31

-XX:+UseConcMarkSweepGC -XX:+UseParNewGC  -XX:+CMSParallelRemarkEnabled

-XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m  -XX:+UseFastAccessorMethods

-XX:+UseCMSInitiatingOccupancyOnly -Djava.awt.headless=true "

   ✔ Windowns系统中catalina.bat文件中添加如下参数（重要参数随后说明）：

set JAVA\_OPTS=-server -Xms1400M -Xmx1400M -Xss512k -XX:+AggressiveOpts -XX:+UseBiasedLocking

-XX:PermSize=128M -XX:MaxPermSize=256M -XX:+DisableExplicitGC -XX:MaxTenuringThreshold=31

-XX:+UseConcMarkSweepGC -XX:+UseParNewGC  -XX:+CMSParallelRemarkEnabled

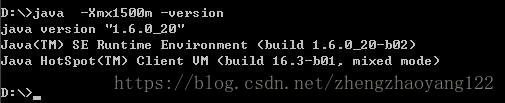
-XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m  -XX:+UseFastAccessorMethods

-XX:+UseCMSInitiatingOccupancyOnly -Djava.awt.headless=true

◀ -Server（重要）：只要Tomcat是运行在生产环境中，这个参数必须添加。因为Tomcat默认是java-client模式运行，添加server后表示以真实的production的模式运行，将拥有更大、更高的并发处理能力，更快、更强的JVM垃圾回收机制，可以获得更多的负载和吞吐量等等。

  ◀ -Xms -Xmx：既JVM内存设置了，把Xms与Xmx两个值设成一样是最优的做法。（否则当内存=Xmx向Xms变化时，CPU高速运转触发垃圾回收机制，严重时会导致系统‘卡壳’，因此一开始我们就把这两个设成一样，使得Tomcat在启动时就为最大化参数充分利用系统的效率。）

   ※在设这个最大内存即Xmx值时请先打开一个命令行：能够正常显示JDK的版本信息，说明这个值能够用。



  ◀ -Xmn：设置年轻代大小为512m。整个堆大小=年轻代 + 老年代 + 持久代。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

  ◀ -Xss：是指设定每个线程的堆栈大小。这个就要依据程序，看一个线程大约需要占用多少内存，可能会有多少线程同时运行等。一般不易设置超过1M，要不然容易出现out ofmemory。

2）、Tomcat容器内优化：打开tomcat安装目录\conf\server.xml文件。   其中如下参数的默认值远远不够我们使用，我们对其进行了更改，更改后的配置如下：

<Connector port="8080" protocol="HTTP/1.1"

URIEncoding="UTF-8"  minSpareThreads="25" maxSpareThreads="75"

enableLookups="false" disableUploadTimeout="true" connectionTimeout="20000"

acceptCount="300"  maxThreads="300" maxProcessors="1000" minProcessors="5"

useURIValidationHack="false"

compression="on" compressionMinSize="2048"

compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"

redirectPort="8443" />

▶ URIEncoding=”UTF-8”：使得tomcat可以解析含有中文名的文件的url。

  ▶ minSpareThreads：最小备用线程数，tomcat启动时的初始化的线程数。

  ▶ maxSpareThreads：如果空闲状态的线程数多于设置的数目，则将这些线程中止，减少这个池中的线程总数。

  ▶ connectionTimeout：网络连接超时时间毫秒数。

  ▶ maxThreads：Tomcat使用线程来处理接收的每个请求。这个值表示Tomcat可创建的最大的线程数，即最大并发数。

  ▶ acceptCount：当线程数达到maxThreads后，后续请求会被放入一个等待队列，这个acceptCount是这个队列的大小，如果这个队列也满了，就直接refuse connection。

  ▶ maxProcessors与minProcessors：在 Java中线程是程序运行时的路径，是在一个程序中与其它控制线程无关的、能够独立运行的代码段。它们共享相同的地址空间。多线程帮助程序员写出CPU最大利用率的高效程序，使空闲时间保持最低，从而接受更多的请求。 通常Windows是1000个左右，Linux是2000个左右。

  ▶ useURIValidationHack：设成"false"，可以减少它对一些url的不必要的检查从而减省开销。

  ▶ enableLookups：设置为"false"，主要为了消除DNS查询对性能的影响我们可以关闭DNS查询，方式是修改server.xml文件中的enableLookups参数值

  ▶ disableUploadTimeout：允许Servlet容器，正在执行使用一个较长的连接超时值，以使Servlet有较长的时间来完成它的执行，默认值为false

  ▶ 给Tomcat配置gzip压缩(HTTP压缩)功能：HTTP 压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩并浏览。相对于普通的浏览过程HTML、CSS、Javascript、Text ，它可以节省40%左右的流量。更为重要的是，它可以对动态生成的，包括CGI、PHP、JSP、ASP、 Servlet、SHTML等输出的网页也能进行压缩，压缩效率惊人。

    1)、compression="on" 打开压缩功能

    2)、compressionMinSize="2048" 启用压缩的输出内容大小，这里面默认为2KB

    3)、noCompressionUserAgents="gozilla, traviata" 对于以下的浏览器，不启用压缩

    4)、compressableMimeType="text/html,text/xml" 压缩类型

  ▶ redirectPort： 如果我们走https协议的话，我们将会用到8443端口这个段的配置。

1. 讲讲Spring加载流程。

初始化环境—>加载配置文件—>实例化Bean—>调用Bean显示信息

1. Spring AOP的实现原理。

AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

Spring提供了两种方式生成代理对象：JDKProxy和Cglib具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。

1. 讲讲Spring事务的传播属性。

   1）、PROPAGATION\_REQUIRED —— 支持当前事务，如果当前没有事务，就新建一个事务。（常见的选择）比如ServiceB.methodB的事务级别定义为PROPAGATION\_REQUIRED,那么由于执行ServiceA.methodA的时候，ServiceA.methodA已经起了事务，这时调用ServiceB.methodB，ServiceB.methodB看到自己已经运行在ServiceA.methodA的事务内部，就不再起新的事务。而假如ServiceA.methodA运行的时候发现自己没有在事务中，他就会为自己分配一个事务。这样，在ServiceA.methodA或者在ServiceB.methodB内的任何地方出现异常，事务都会被回滚。即使ServiceB.methodB的事务已经被提交，但是ServiceA.methodA在接下来fail要回滚，ServiceB.methodB也要回滚。

   2）、PROPAGATION\_SUPPORTS —— 支持当前事务，如果当前没有事务，就以非事务方式执行。

   3）、PROPAGATION\_MANDATORY ——支持当前事务，如果当前没有事务，就抛出异常。

   4）、PROPAGATION\_REQUIRES\_NEW ——支持当前事务，如果当前没有事务，就将当前事务挂起。如ServiceA.methodA的事务级别为PROPAGATION\_REQUIRED，ServiceB.methodB的事务级别为PROPAGATION\_REQUIRES\_NEW，那么当执行到ServiceB.methodB的时候，ServiceA.methodA所在的事务就会挂起，ServiceB.methodB会起一个新的事务，等待ServiceB.methodB的事务完成以后，A才继续执行。他与PROPAGATION\_REQUIRED的事务区别在于事务的回滚程度了。因为ServiceB.methodB是新起一个事务，那么就是存在两个不同的事务。如果ServiceB.methodB已经提交，那么ServiceA.methodA失败回滚，ServiceB.methodB是不会回滚的。如果ServiceB.methodB失败回滚，如果他抛出的异常被ServiceA.methodA捕获，ServiceA.methodA事务仍然可能提交。

   5）、PROPAGATION\_NOT\_SUPPORTED —— 以非事务方式执行当前操作，如果当前存在事务，就把事务挂起来。

   6）、PROPAGATION\_NEVER —— 以非事务方式执行，如果当前存在事务，则抛异常。

   7）、PROPAGATION\_NESTED—— 如果当前存在事务，则在嵌套事务内执行，关键是savepoint。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。与PROPAGATION\_REQUIRES\_NEW的区别是NESTED的事务和他的父事务是相依的，它的提交是要等父事务一块提交。也就是说，如果父事务最后回滚，它也要回滚。

1. Spring如何管理事务的。

   Spring事务管理主要包括3个接口，Spring事务主要由以下三个共同完成的：

   1）、PlatformTransactionManager：事务管理器，主要用于平台相关事务的管理。主要包括三个方法：①、commit：事务提交。②、rollback：事务回滚。③、getTransaction：获取事务状态。

   2）、TransacitonDefinition：事务定义信息，用来定义事务相关属性，给事务管理器PlatformTransactionManager使用这个接口有下面四个主要方法：①、getIsolationLevel：获取隔离级别。②、getPropagationBehavior：获取传播行为。③、getTimeout获取超时时间。④、isReadOnly：是否只读（保存、更新、删除时属性变为false--可读写，查询时为true--只读）事务管理器能够根据这个返回值进行优化，这些事务的配置信息，都可以通过配置文件进行配置。

   3）、TransationStatus：事务具体运行状态，事务管理过程中，每个时间点事务的状态信息。例如：①、hasSavepoint()：返回这个事务内部是否包含一个保存点。②、isCompleted()：返回该事务是否已完成，也就是说，是否已经提交或回滚。③、isNewTransaction()：判断当前事务是否是一个新事务。

1. Spring怎么配置事务（具体说出一些关键的xml 元素）。

☞ 配置事务的方法有两种：1）、基于XML的事务配置。2）、基于注解方式的事务配置。

 基于XML的事务配置：

<?xml version="1.0" encoding="UTF-8"?>

<!-- from the file 'context.xml' -->

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<!-- 数据元信息 -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

<property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

<!-- 管理事务的类,指定我们用谁来管理我们的事务-->

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- 首先我们要把服务对象声明成一个bean 例如HelloService -->

<bean id="helloService" class="com.yintong.service.HelloService"/>

<!-- 然后是声明一个事物建议tx:advice,spring为我们提供了事物的封装，这个就是封装在了<tx:advice/>中 -->

<!-- <tx:advice/>有一个transaction-manager属性，我们可以用它来指定我们的事物由谁来管理。

默认：事务传播设置是 REQUIRED，隔离级别是DEFAULT -->

<tx:advice id="txAdvice" transaction-manager="txManager">

<!-- 配置这个事务建议的属性 -->

<tx:attributes>

<!-- 指定所有get开头的方法执行在只读事务上下文中 -->

<tx:method name="get\*" read-only="true"/>

<!-- 其余方法执行在默认的读写上下文中 -->

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<!-- 我们定义一个切面，它匹配FooService接口定义的所有操作 -->

<aop:config>

<!-- <aop:pointcut/>元素定义AspectJ的切面表示法，这里是表示com.yintong.service.helloService包下的任意方法。 -->

<aop:pointcut id="helloServiceOperation" expression="execution(\* com.yintong.service.helloService.\*(..))"/>

<!-- 然后我们用一个通知器：<aop:advisor/>把这个切面和tx:advice绑定在一起，表示当这个切面：fooServiceOperation执行时tx:advice定义的通知逻辑将被执行 -->

<aop:advisor advice-ref="txAdvice" pointcut-ref="helloServiceOperation"/>

</aop:config>

</beans>

基于注解方式的事务配置 ：  @Transactional：直接在Java源代码中声明事务的做法让事务声明和将受其影响的代码距离更近了，而且一般来说不会有不恰当的耦合的风险，因为，使用事务性的代码几乎总是被部署在事务环境中。

♣ 注解事务，只需要在XML配置中配置一句就可以了，如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<bean id="helloService" class="com.yintong.service.HelloService"/>

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- 配置注解事务 -->

<tx:annotation-driven transaction-manager="txManager"/>

</beans>

 ♣ 主要在类中定义事务注解@Transactional，如下：

//@Transactional 注解可以声明在类上，也可以声明在方法上。在大多数情况下，方法上的事务会首先执行

@Transactional(readOnly = true)

public class HelloService{

public Foo getFoo(String fooName) {

}

//@Transactional 注解的事务设置将优先于类级别注解的事务设置 propagation:可选的传播性设置

@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW)

public void updateFoo(Hel hel) {

}

}

1. 说说你对Spring的理解？非单例注入的原理？它的生命周期？循环注入的原理？

✔ Spring的理解：1）、Spring是一个开源框架，主要是为简化企业级应用开发而生。可以实现EJB可以实现的功能，Spring是一个IOC和AOP容器框架。

 ♧ 控制反转（IOC）：Spring容器使用了工厂模式为我们创建了所需要的对象，我们使用时不需要自己去创建，直接调用Spring为我们提供的对象即可，这就是控制反转的思想。

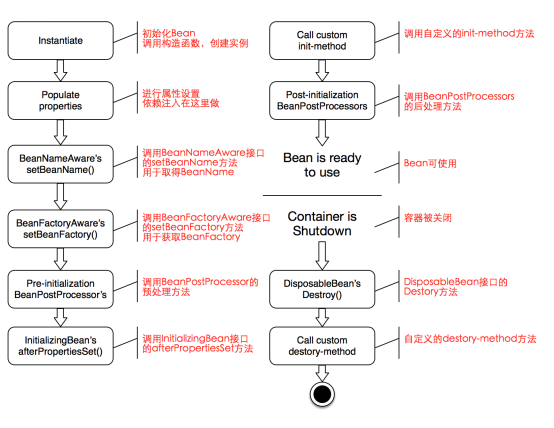
 ♧ 依赖注入（DI）：Spring使用Java Bean对象的Set方法或者带参数的构造方法为我们在创建所需对象时将其属性自动设置所需要的值的过程就是依赖注入的基本思想。

 ♧ 面向切面编程（AOP）：在面向对象编程(OOP)思想中，我们将事物纵向抽象成一个个的对象。而在面向切面编程中，我们将一个个对象某些类似的方面横向抽象成一个切面，对这个切面进行一些如权限验证，事物管理，记录日志等公用操作处理的过程就是面向切面编程的思想。

2）、在Spring中，所有管理的都是JavaBean对象，而BeanFactory和ApplicationContext就是Spring框架的那个IOC容器，现在一般使用ApplicationContext，其不但包括了BeanFactory的作用，同时还进行了更多的扩展。

✔ 非单例注入原理：在大部分情况下，容器中的bean都是singleton类型的。如果一个singleton bean要引用另外一个singleton bean或者一个非singleton bean要引用另外一个非singleton，通常情况下将一个bean定义为另一个bean的property值就可以了。不过对于具有不同生命周期的bean来说这样做就会有问题了，比如在调用一个singleton类型bean A的某个方法时，需要引用另一个非singleton（prototype）类型的bean B，对于bean A来说，容器只会创建一次，这样就没法在需要的时候每次让容器为bean A提供一个新的的bean B实例。

✔ 生命周期：



✔ 循环依赖注入：

循环依赖就是循环引用，就是两个或多个Bean相互之间的持有对方，比如A引用B，B引用C，C引用A，则它们最终反映为一个环。

spring 中循环依赖注入分三种情况

1. 构造器循环依赖 (报错)

2. setter方法循环注入

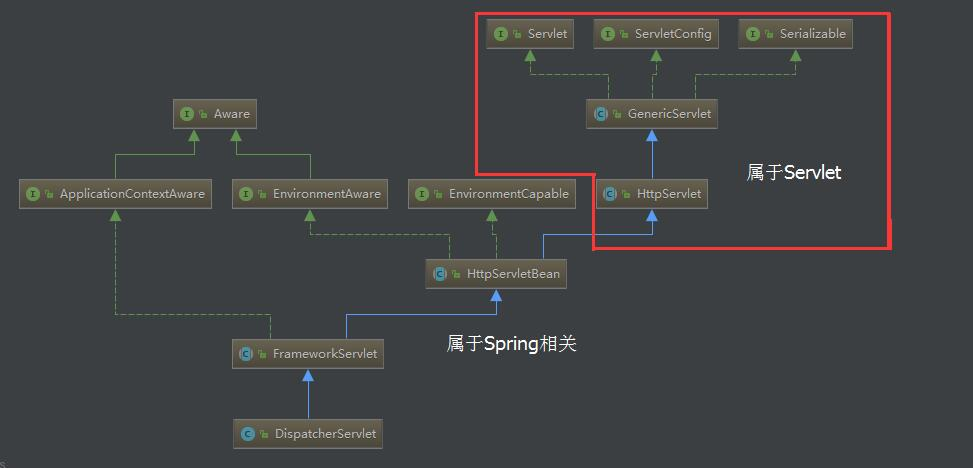
2.1 setter方法注入 单例模式(scope=singleton) (成功)

2.2 setter方法注入 非单例模式 (报错)

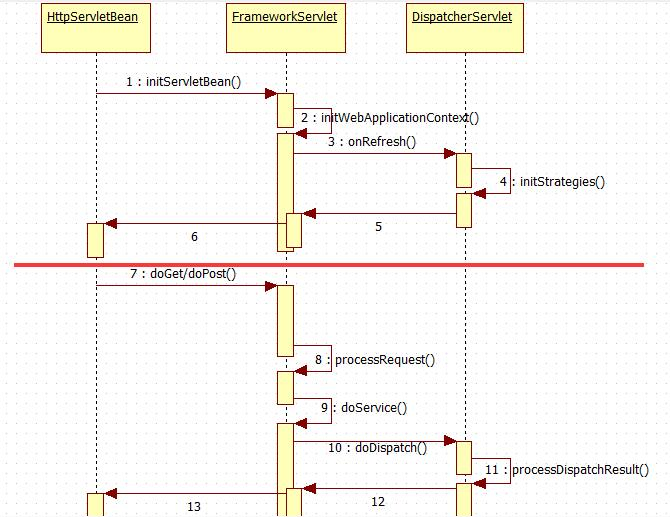
1. Springmvc 中DispatcherServlet初始化过程。

在使用springmvc框架，会在web.xml文件配置一个DispatcherServlet, 这正是web容器开始初始化，同时会在建立自己的上下文来持有SpringMVC的bean对象。

从继承关系来看，DispatcherServlet继承FrameworkServlet和HttpServletBean而继承HttpServlet,通过使用Servlet API 来对HTTP请求进行响应，成为SpringMVC的前端处理器。



时序图:



注：作为Servlet,DispatcherServlet的启动和Servlet的启动相关联的。在Servlet初始化过程中，Servlet的init方法会被调用，以进行初始化，然而DispatcherServlet的基类，所以从HttpServletBean中的初始化过程开始。

DispatcherServlet实现了Servlet接口的实现类。Servlet的生命周期分为3个阶段：初始化、运行和销毁。而其初始化阶段可分为：

Servlet容器加载Servlet类，把类的.class文件中的数据读到内存中；

Servlet容器中创建一个ServletConfig对象。该对象中包含了Servlet的初始化配置信息；

Servlet容器创建一个Servlet对象；

Servlet容器调用Servlet对象的init()方法进行初始化。

1. netty的线程模型，netty如何基于reactor模型上实现的。

Reactor 模式

是一种为处理服务请求并发,提交到一个或者多个服务处理程序的事件设计模式。当请求抵达后，服务处理程序使用解多路分配策略，然后同步地派发这些请求至相关的请求处理程序。

1. 为什么选择netty。

开发出高质量的NIO 程序并不是一件简单的事情，除去NIO 固有的复杂性和Bug不谈，作为一个NIO 服务端,需要能够处理网络的闪断、客户端的重复接入、客户端的安全认证、消息的编解码、半包读写等情况， 如果你没有足够的NIO 编程经验积累， 一个NIO 框架的稳定往往需要半年甚至更长的时间。更为糟糕的是， 一旦在生产环境中发生问题， 往往会导致跨节点的服务调用中断， 严重的可能会导致整个集群环境都不可用， 需要重启服务器，这种非正常停机会带来巨大的损失。而netty针对上述问题有了完整的解决方案。

1. 什么是TCP粘包，拆包。解决方式是什么。

粘包拆包问题是处于网络比较底层的问题，在数据链路层、网络层以及传输层都有可能发生。我们日常的网络应用开发大都在传输层进行，由于UDP有消息保护边界，不会发生粘包拆包问题，因此粘包拆包问题只发生在TCP协议中。接收端收到了两个数据包，但是这两个数据包要么是不完整的，要么就是多出来一块，这种情况即发生了拆包和粘包。

粘包、拆包解决办法:

1、发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

2、发送端将每个数据包封装为固定长度（不够的可以通过补0填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

3、可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

1. netty的fashwheeltimer的用法，实现原理，是否出现过调用不够准时，怎么解决。
2. netty的心跳处理在弱网下怎么办。

使用 Netty 实现心跳机制的关键就是利用 IdleStateHandler 来产生对应的 idle 事件.

一般是客户端负责发送心跳的 PING 消息, 因此客户端注意关注 ALL\_IDLE 事件, 在这个事件触发后, 客户端需要向服务器发送 PING 消息, 告诉服务器"我还存活着".

服务器是接收客户端的 PING 消息的, 因此服务器关注的是 READER\_IDLE 事件, 并且服务器的 READER\_IDLE 间隔需要比客户端的 ALL\_IDLE 事件间隔大(例如客户端ALL\_IDLE 是5s 没有读写时触发, 因此服务器的 READER\_IDLE 可以设置为10s)

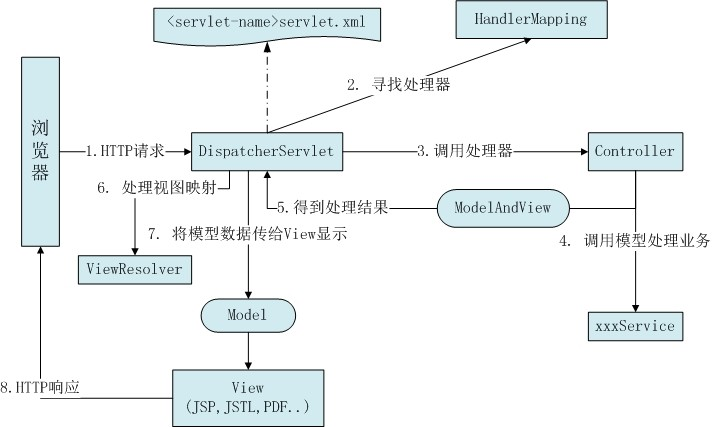
当服务器收到客户端的 PING 消息时, 会发送一个 PONG 消息作为回复. 一个 PING-PONG 消息对就是一个心跳交互.

1. netty的通讯协议是什么样的。

自定义协议的通信

1. springmvc用到的注解，作用是什么，原理。

SpringMVC原理图



1.客户端请求提交到DispatcherServlet

2.由DispatcherServlet控制器查询一个或多个HandlerMapping，找到处理请求的Controller

3.DispatcherServlet将请求提交到Controller

4.Controller调用业务逻辑处理后，返回ModelAndView

5.DispatcherServlet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图

6.视图负责将结果显示到客户端

DispatcherServlet是整个Spring MVC的核心。它负责接收HTTP请求组织协调Spring MVC的各个组成部分。其主要工作有以下三项：

1. 截获符合特定格式的URL请求。

2. 初始化DispatcherServlet上下文对应的WebApplicationContext，并将其与业务层、持久化层的WebApplicationContext建立关联。

3. 初始化Spring MVC的各个组成组件，并装配到DispatcherServlet中。

<mvc:annotation-driven /> 说明:

是一种简写形式，可以让初学者快速成应用默认的配置方案，会默认注册 DefaultAnnotationHandleMapping以及AnnotionMethodHandleAdapter 这两个 Bean, 这两个 Bean ，前者对应类级别， 后者对应到方法级别；

上在面的 DefaultAnnotationHandlerMapping和AnnotationMethodHandlerAdapter 是 Spring 为 @Controller 分发请求所必需的。

annotation-driven 扫描指定包中类上的注解，常用的注解有:

@Controller 声明Action组件

@Service 声明Service组件 @Service("myMovieLister")

@Repository 声明Dao组件

@Component 泛指组件, 当不好归类时.

@RequestMapping("/menu") 请求映射

@Resource 用于注入，( j2ee提供的 ) 默认按名称装配，@Resource(name="beanName")

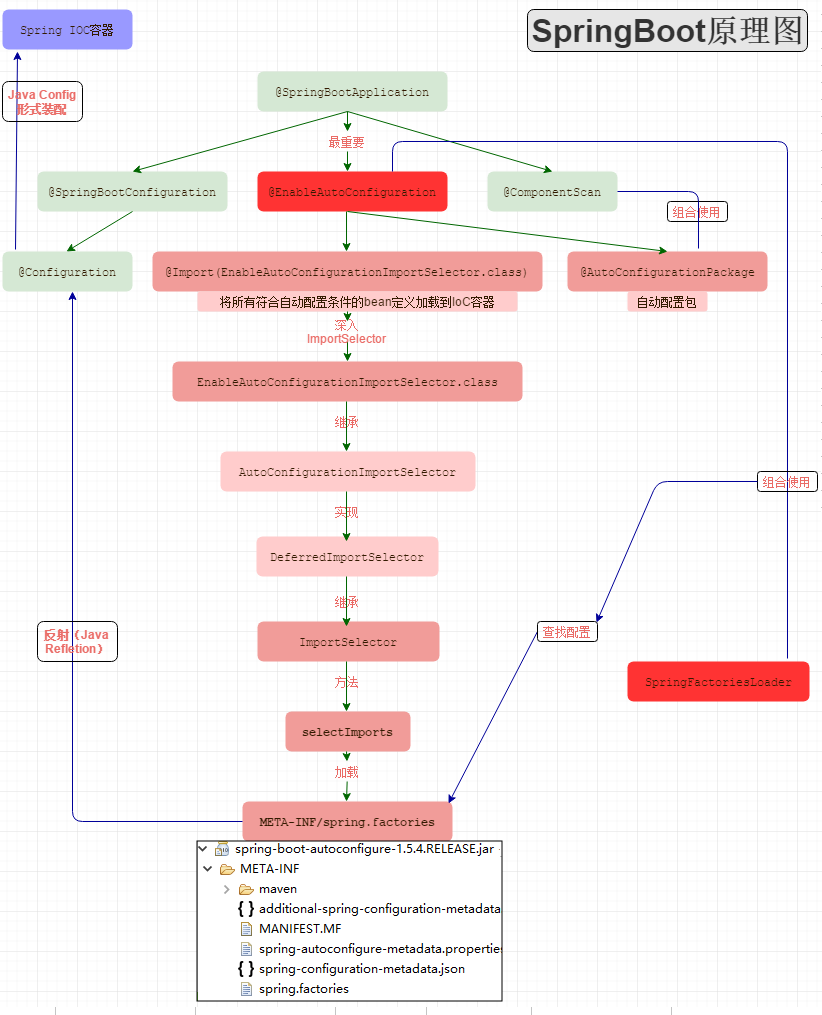
@Autowired 用于注入，(srping提供的) 默认按类型装配

@Transactional( rollbackFor={Exception.class}) 事务管理

@ResponseBody

@Scope("prototype") 设定bean的作用域

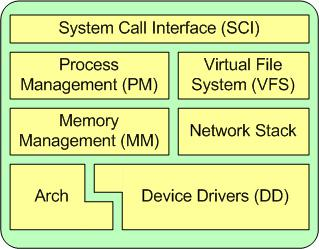
1. springboot启动机制。



SpringBoot的核心组件完成了基本的解析，综合来看，大部分都是Spring框架背后的一些概念和实践方式，SpringBoot只是在这些概念和实践上对特定的场景事先进行了固化和升华，而也恰恰是这些固化让我们开发基于Sping框架的应用更加方便高效。

# 操作系统

1. Linux系统下你关注过哪些内核参数，说说你知道的。



1. Linux下IO模型有几种，各自的含义是什么。

阻塞式io，非阻塞io，io复用模型，信号驱动io模型，异步io模型。

1. 平时用到哪些Linux命令。

Ls,find,tar,tail,cp,rm,vi，grep,ps,pkill等等

1. 用一行命令查看文件的最后五行。

Tail -n 5 filename

1. 用一行命令输出正在运行的java进程。

ps -ef|grep Java

1. 介绍下你理解的操作系统中线程切换过程。

Linux下的线程实质上是轻量级进程(light weighted process),线程生成时会生成对应的进程控制结构，只是该结构与父线程的进程控制结构共享了同一个进程内存空间。 同时新线程的进程控制结构将从父线程（进程）处复制得到同样的进程信息，如打开文件列表和信号阻塞掩码等。创建线程比创建新进程成本低，因为新创建的线程使用的是当前进程的地址空间。相对于在进程之间切换，在线程之间进行切换所需的时间更少，因为后者不包括地址空间之间的切换。

线程切换上下文切换的原理与此类似，只是线程在同一地址空间中，不需要MMU等切换，只需要切换必要的CPU寄存器，因此，线程切换比进程切换快的多。

1. 进程和线程的区别。

Linux 实现并没有区分这两个概念（进程和线程）

1. 进程：程序的一次执行

2. 线程：CPU的基本调度单位

一个进程可以包含多个线程。

1. top 命令之后有哪些内容，有什么作用。

top命令：相当于Windows下的资源管理器，能够动态实时的显示系统中进程的资源占用情况

1. 线上CPU爆高，请问你如何找到问题所在。

1、top命令：Linux命令。可以查看实时的CPU使用情况。也可以查看最近一段时间的CPU使用情况。

2、PS命令：Linux命令。强大的进程状态监控命令。可以查看进程以及进程中线程的当前CPU使用情况。属于当前状态的采样数据。

3、jstack：Java提供的命令。可以查看某个进程的当前线程栈运行情况。根据这个命令的输出可以定位某个进程的所有线程的当前运行状态、运行代码，以及是否死锁等等。

4、pstack：Linux命令。可以查看某个进程的当前线程栈运行情况。

# 多线程

1. 多线程的几种实现方式，什么是线程安全。

【1】继承Thread类，重写run()方法

【2】、实现 Runnable 接口，并实现该结构的 run() 方法

【3】、实现 Callable 接口，重写 call() 方法

线程安全:

在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况。

1. volatile的原理，作用，能代替锁么。

volatile 变量的内存可见性是基于内存屏障（Memory Barrier）实现

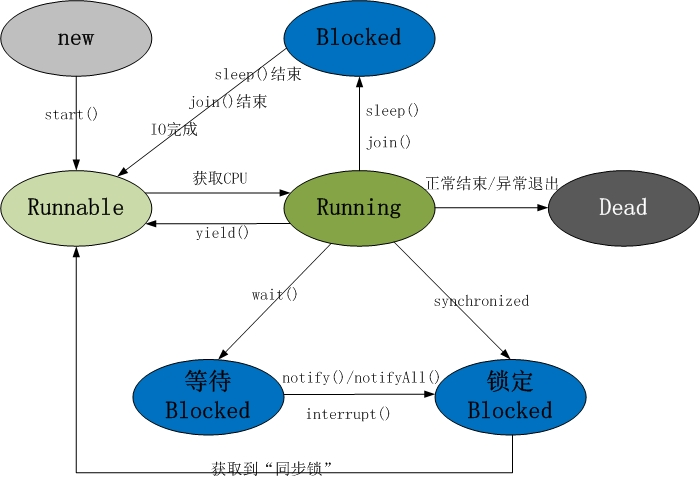
volatile 是 Java 虚拟机提供的轻量级同步机制。具有如下三种特性：

1）、保证可见性：被 volatile 修改的变量会被存储在主内存中，而不是工作内存中。可以被其他线程共享访问。

2）、不保证原子性：既对修饰的变量的操作是可分割的（举例：i++ 十次得到的结果，不一定是10），也就是线程不安全的。

3）、禁止指令重排：计算机会对编译后的 java 语言进行编译。为了提高运行效率，会对指令进行重排。单线程重排后，是没有问题的，但是在多线程中重排就会影响执行结果。

1. 画一个线程的生命周期状态图。



1. sleep和wait的区别。

它们最大本质的区别是:sleep()不释放同步锁,wait()释放同步锁.

还有用法的上的不同是:sleep(milliseconds)可以用时间指定来使他自动醒过来,如果时间不到你只能调用interreput()来强行打断;wait()可以用notify()直接唤起.

sleep和wait的区别还有：

1.这两个方法来自不同的类分别是Thread和Object

2.最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3.wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在

任何地方使用

synchronized(x){

x.notify()

//或者wait()

}

4.sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常sleep和sleep(0)的区别。

1. Lock与Synchronized的区别 。

1.首先synchronized是java内置关键字，在jvm层面，Lock是个java类；

2.synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；

3.synchronized会自动释放锁(a 线程执行完同步代码会释放锁 ；b 线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；

4.用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；

5.synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可判断、可公平（两者皆可）

6.Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

1. synchronized的原理是什么

实现原理： JVM 是通过进入、退出 对象监视器(Monitor) 来实现对方法、同步块的同步的，而对象监视器的本质依赖于底层操作系统的 互斥锁(Mutex Lock) 实现。

在同步块的入口和出口分别有monitorenter和monitorexit指令。当执行monitorenter指令时，线程试图获取锁也就是获取monitor（monitor对象存在于每个Java对象的对象头中，synchronized锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因）的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行monitorexit指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

在synchronized修饰方法时是添加ACC\_SYNCHRONIZED标识，该标识指明了该方法是一个同步方法，JVM通过该ACC\_SYNCHRONIZED访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

1. 解释以下名词：重排序，自旋锁，偏向锁，轻量级锁，可重入锁，公平锁，非公平锁，乐观锁，悲观锁。

https://www.cnblogs.com/jyroy/p/11365935.html#idx\_4

1. 用过哪些原子类，他们的原理是什么。

• 基本类型：AtomicInteger, AtomicLong, AtomicBoolean.

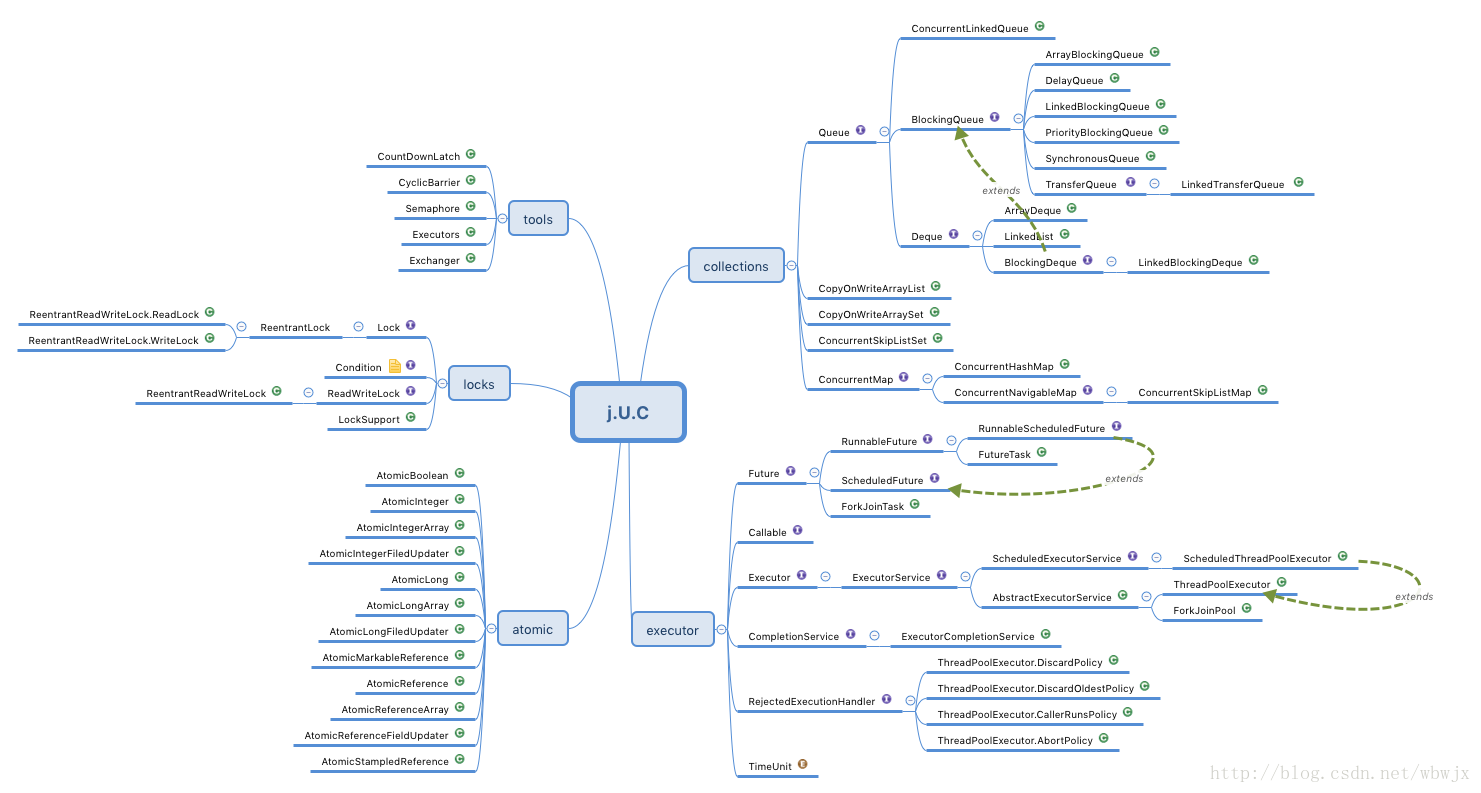
• 数组类型：AtomicIntegerArray, AtomicLongArray.

• 引用类型：AtomicReference, AtomicStampedRerence.

• 对象的属性修改类型：AtomicIntegerFieldUpdater, AtomicLongFieldUpdater, AtomicReferenceFieldUpdater.

用volatile进行声明, CAS方法操作

1. JUC下研究过哪些并发工具，讲讲原理。



1. 用过线程池吗，如果用过，请说明原理，并说说newCache和newFixed有什么区别，构造函数的各个参数的含义是什么，比如coreSize，maxsize等。

newFixedThreadPool()方法：该方法返回一个固定数量的线程池；

newSingleThreadExecutor()方法：该方法返回只有一个线程的线程池；

newCachedThreadPool()方法：该方法返回一个可根据实际情况调整线程数量的线程池；

newScheduledThreadPool()方法：该方法返回一个ScheduleExecutorService对象，可以指定线程数量；

newSingleThreadScheduledExecutor()方法：该方法返回一个ScheduleExecutorService对象，线程数量为1；

public ThreadPoolExecutor(

int corePoolSize, // 指定线程池中线程的数量（总线程量可大于等于这个值）

int maximumPoolSize, // 指定线程池中最大线程数量（总线程量不可能超越这个数值）

long keepAliveTime, // 超过corePoolSize数量的空闲线程，存活的时间

TimeUtil unit, // keepAliveTime的单位

BlockingQueue<Runnable> workQueue, // 任务队列，被提交但为执行的任务

ThreadFactory threadFactory, // 线程工厂，用于创建线程

RejectedExecutionHandler handler // 当workQueue队列满的时候的拒绝策略

)

1. 线程池的关闭方式有几种，各自的区别是什么。

线程池提供了两个关闭方法，shutdownNow和shuwdown方法。

shutdownNow方法的解释是：线程池拒接收新提交的任务，同时立马关闭线程池，线程池里的任务不再执行。

shutdown方法的解释是：线程池拒接收新提交的任务，同时等待线程池里的任务执行完毕后关闭线程池。

awaitTermination(long timeout, TimeUnit unit) 方法可以获取线程池是否已经关闭

1. 假如有一个第三方接口，有很多个线程去调用获取数据，现在规定每秒钟最多有10个线程同时调用它，如何做到。

ScheduledThreadPoolExecutor 设置定时，进行调度。

ScheduledThreadPoolExecutor scheduled = new ScheduledThreadPoolExecutor(10);

scheduled.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

System.out.println("time:" + Thread.currentThread().getName());

}

}, 0, 100, TimeUnit.MILLISECONDS);//0表示首次执行任务的延迟时间，100表示每次执行任务的间隔时间，TimeUnit.MILLISECONDS执行的时间间隔数值单位

1. spring的controller是单例还是多例，怎么保证并发的安全。

spring的controller默认是单例的, 使用成员变量不安全, 不得不使用可用@Scope("prototype")将controller设为多例

1. 用三个线程按顺序循环打印abc三个字母，比如abcabcabc。

ExecutorService excutorService = Executors.newFixedThreadPool(3);

for(int i=0;i<3;i++){

excutorService.execute(()->System.out.println("abc"));

}

1. ThreadLocal用过么，用途是什么，原理是什么，用的时候要注意什么。

1)ThreadLocal是为了解决多线程中相同变量的访问冲突问题

2)通过每个线程单独一份存储空间，牺牲空间来解决冲突,具有线程隔离的效果，只有在线程内才能获取到对应的值，线程外则不能访问到想要的值。

3)ThreadLocal使用不当造成OOM

1. 如果让你实现一个并发安全的链表，你会怎么做。

Collections.synchronizedList() ConcurrentLinkedQueue

使用交替锁（hand-over-hand locking）。只锁住链表的一部分，链表没有被锁住的部分自由访问

1. 有哪些无锁数据结构，他们实现的原理是什么。

LockFree

无锁数据结构的实现主要基于两个方面：原子性操作和内存访问控制方法。

1. 讲讲java同步机制的wait和notify。

这两个方法只能在同步代码块中调用，wait会释放掉对象锁，等待notify唤醒。

1. CAS机制是什么，如何解决ABA问题。

CAS全称 Compare And Swap（比较与交换），是一种无锁算法。在不使用锁（没有线程被阻塞）的情况下实现多线程之间的变量同步。

利用版本号比较可以有效的解决ABA问题。

1. 多线程如果线程挂住了怎么办。

根据具体情况（sleep,wait,join等），酌情选择notifyAll，notify进行线程唤醒。

1. countdowlatch和cyclicbarrier的内部原理和用法，以及相互之间的差别(比如countdownlatch的await方法和是怎么实现的)。
2. 对AbstractQueuedSynchronizer了解多少，讲讲加锁和解锁的流程，独占锁和公平所加锁有什么不同。
3. 使用synchronized修饰静态方法和非静态方法有什么区别。

学习Java的小伙伴都知道synchronized关键字是解决并发问题常用解决方案，常用的有以下三种使用方式：

1)修饰代码块，即同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象。

2)修饰普通方法，即同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象。

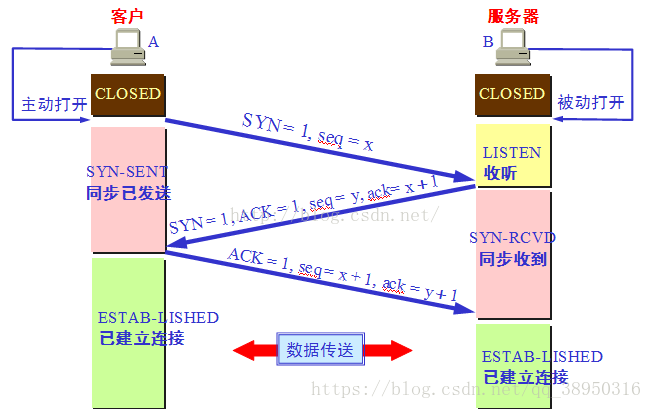
3)修饰静态方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象。

1. 简述ConcurrentLinkedQueue和LinkedBlockingQueue的用处和不同之处。
2. 导致线程死锁的原因？怎么解除线程死锁。
3. 非常多个线程（可能是不同机器），相互之间需要等待协调，才能完成某种工作，问怎么设计这种协调方案。
4. 用过读写锁吗，原理是什么，一般在什么场景下用。
5. 开启多个线程，如果保证顺序执行，有哪几种实现方式，或者如何保证多个线程都执行完再拿到结果。
6. 延迟队列的实现方式，delayQueue和时间轮算法的异同。

# TCP与HTTP

1. http1.0和http1.1有什么区别。
2. TCP三次握手和四次挥手的流程，为什么断开连接要4次,如果握手只有两次，会出现什么。

【1】、TCP三次握手：

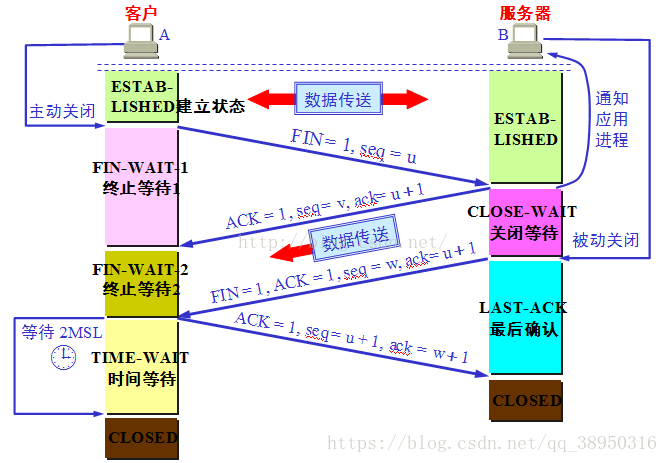


1）第一次握手：A的TCP客户进程也是首先创建传输控制块TCB，然后向B发出连接请求报文段，（首部的同步位SYN=1，初始序号seq=x），（SYN=1的报文段不能携带数据）但要消耗掉一个序号，此时TCP客户进程进入SYN-SENT（同步已发送）状态。

2）第二次握手：B收到连接请求报文段后，如同意建立连接，则向A发送确认，在确认报文段中（SYN=1，ACK=1，确认号ack=x+1，初始序号seq=y），TCP服务器进程进入SYN-RCVD（同步收到）状态；

3）第三次握手：TCP客户进程收到B的确认后，要向B给出确认报文段（ACK=1，确认号ack=y+1，序号seq=x+1）（初始为seq=x，第二个报文段所以要+1），ACK报文段可以携带数据，不携带数据则不消耗序号。TCP连接已经建立，A进入ESTABLISHED（已建立连接）。当B收到A的确认后，也进入ESTABLISHED状态。

【2】、TCP四次挥手：



1）A的应用进程先向其TCP发出连接释放报文段（FIN=1，序号seq=u），并停止再发送数据，主动关闭TCP连接，进入FIN-WAIT-1（终止等待1）状态，等待B的确认。

2）B收到连接释放报文段后即发出确认报文段，（ACK=1，确认号ack=u+1，序号seq=v），B进入CLOSE-WAIT（关闭等待）状态，此时的TCP处于半关闭状态，A到B的连接释放。

3）A收到B的确认后，进入FIN-WAIT-2（终止等待2）状态，等待B发出的连接释放报文段。

4）B发出连接释放报文段（FIN=1，ACK=1，序号seq=w，确认号ack=u+1），B进入LAST-ACK（最后确认）状态，等待A的确认。

5）A收到B的连接释放报文段后，对此发出确认报文段（ACK=1，seq=u+1，ack=w+1），A进入TIME-WAIT（时间等待）状态。此时TCP未释放掉，需要经过时间等待计时器设置的时间2MSL后，A才进入CLOSED状态。

1. TIME\_WAIT和CLOSE\_WAIT的区别。

CLOSE\_WAIT：等待关闭，是被动关闭连接形成的，也就是第二次挥手时产生的状态。也就是当对方 Close 一个 SOCKET 后发送 FIN 报文给自己，系统会回应一个 ACK 报文给对方，此时进入CLOSE\_WAIT状态。接着，我们需要考虑的事情是查看是否还有数据发送给对方，如果没有就可以 Close 这个链接，发送 FIN 给对方，也既关闭连接。所以在 CLOSE\_WAIT 状态时，需要查看自己是否需要关闭连接。

TIME\_WAIT：是主动关闭连接方形成的，表示收到了对方的 FIN 报文，并发送 ACK 报文，等待 2MSL（*Maximum Segment Lifetime：片段在网络中最大的存活时间*）后进入 CLOSE 状态。主要是防止最后一个 ACK 丢失，由于 TIME\_WAIT 等待时间较长，因此 server 端尽量减少关闭。

1. 说说你知道的几种HTTP响应码，比如200, 302, 404。

**【1】**200 OK：表示客户端请求成功。

【3】302 Moved Temporarily：所请求的页面已经临时转移至新的url。  
【3】400 Bad Request 语义有误：不能被当前服务器理解。  
【4】401 Unauthorized： 当前请求需要用户验证。  
【5】403 Forbidden： 服务器收到消息，但是拒绝提供服务。  
【6】404 Not Found ：请求资源不存在。  
【7】408 Request Timeout： 请求超时，客户端没有在服务器预备等待的时间内完成发送。  
【8】500 Internal Server Error： 服务器发生不可预期的错误。  
【9】503 Server Unavailable ：由于临时的服务器维护或过载，服务器当前不能处理请求，此状况知识临时的，可恢复。

1. 当你用浏览器打开一个链接的时候，计算机做了哪些工作步骤。

1）、解析域名。

2）、发起 TCP 的 3 次握手。

3）、建立 TCP 请求后发起 HTTP 请求。

4）、服务器响应 HTTP 请求。

5）、浏览器得到 HTML 代码，进行解析和处理 JSON 数据，并请求 HTML 代码中的静态资源（JS、CSS、图片等）。

6）、浏览器对页面进行渲染。

1. TCP/IP如何保证可靠性

1）、三次握手。

2）、将数据截断为合理的长度。应用数据被分割成 TCP 认为最合适发送的数据块。

3）、超时重发。

4）、对于收到的请求，给予确认响应。

5）、如果校验出数据包有错，则丢弃报文段，不响应。

6）、对失序数据进行重新排序，发送于客户端。

7）、能够丢弃重复数据。

8）、流量控制。TCP连接的两端都有缓存大小控制，接收端只允许发送端发送自己缓存剩余大小的数据。有效防止缓存溢出。

9）、拥塞控制。当网络阻塞时，减少数据的发送。

1. 说说TCP头的结构。



1. 16位端口号：包括了源端口号和目的端口号。进行TCP通信时，客户端通常使用系统自动选择的临时端口号(一般都很大)，而服务器则使用知服务端口号或服务器管理员自定义的端口号。
2. 32位序号：一次TCP通信过程中对一个传输方向上的字节流的每个字节的编号(从这个方向第一个报文段依次排列)。假设主机A和主机B进行TCP通信，A发送给B的第一个TCP报文段中的序号值是系统初始化的一个随机值ISN(初始序号值)。那么之后在A到B的方向上发送的TCP报文段中的序号值将会被系统设置为ISN加上该报文段所携带数据的第一个字节在整个数据字节流中的偏移。
3. 32位确认号：用作对另一方发送来的TCP报文段做出相应。其值是收到对方的报文段的序号值加1。
4. 4位头部长度：标识该TCP头部有多少个32bit字(4字节)。一共有4位，所以最大能表示TCP头部大小为60字节。
5. 6位标志位包含如下几项：

* URG标志，表示紧急指针是否有效。
* ACK标志，表示确认号是否有效。一般称携带ACK标志的报文段是确认报文段。
* PSH标志，提示接收端应用程序立即从TCP接受缓冲区读走数据。
* RST标志，表示要求对方重新建立连接。称携带RST标志的TCP报文段为复位报文段。
* SYN标志，表示请求建立一个连接。称携带SYN标志的TCP报文段为同步报文段。
* FIN标志，表示通知对方本端将关闭连接。称携带FIN标志的TCP报文段为结束报文段。

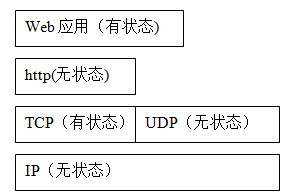
1. 16位窗口大小：是TCP流量控制的一个手段。这里说的窗口指的是接收通告窗口(RWND)。它告诉对方本端TCP接收缓冲区还能容纳多少字节的数据，以让对方控制发送数据的速度。
2. 16位校验和：由发送端填充，接收端对TCP报文段执行CRC算法以效验TCP报文段在传输过程中是否损坏(包括TCP头部和数据部分)。这也是TCP可靠传输的一个重要保障。
3. 16位紧急指针：是一个正的偏移量。它和序号字段的值相加表示最后一个紧急数据的下一字节的序号。即这个字段是紧急指针相对当前序号的偏移，为紧急偏移。以让接收端迅速接受到紧急数据。TCP的紧急指针是发送端向接收端发送紧急数据的方法。
4. 如何避免浏览器缓存。
5. 如何理解HTTP协议的无状态性。

无状态指:

* 每次的请求都是独立的，它的执行情况和结果与前面的请求和之后的请求是无直接关系的，它不会受前面的请求应答情况直接影响，也不会直接影响后面的请求应答情况
* web应用是有状态的。所以我们加入了cookies、session等机制去实现由状态的web应用。所以我们可以这么理解：

Web应用=http协议+session、cookies等状态机制+其他辅助的机制。

* TCP为一个有状态的传输层通信协议，而UDP则不是；IP是无状态的。要明白这种有状态与否的判定，是指你这一协议栈层次所要实现的功能——是否由上下文决定——来判定的（是否受之前的通信过程直接影响、是否直接影响之后的通信过程）



1. 简述Http请求get和post的区别。

【1】GET请求可被缓存，POST请求不能被缓存。

【2】GET请求被保留着浏览器历史记录中，POST请求不会被保留。

【3】GET请求能被收藏至书签中，POST请求不能被收藏至书签。

【4】GET请求不应在处理敏感数据时使用，POST可以用户处理敏感数据。

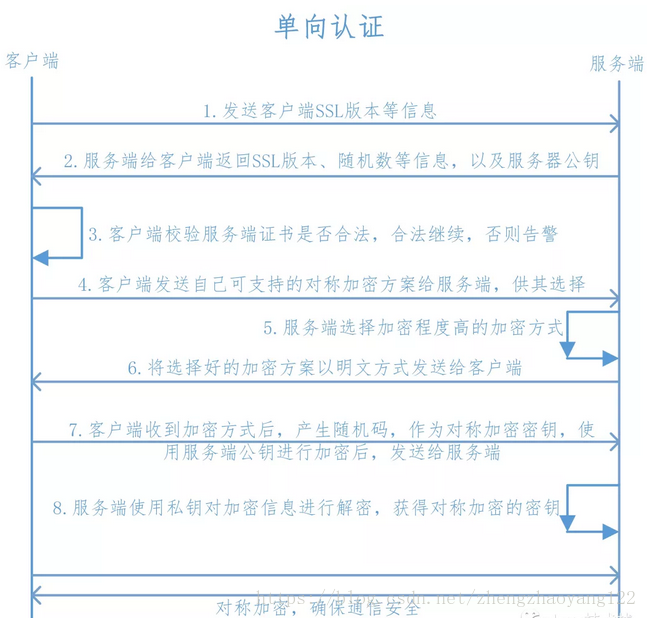
【5】 GET请求有长度限制，POST请求没有长度限制。

【6】POST不限制提交的数据类型，所以POST可以提交文件到服务器。

1. HTTP有哪些method

★ GET：获取资源。  
★ POST：表单提交。  
★ HEAD：获取报头信息，HEAD 方法与 GET 方法类似，但并不会返回响应主体。  
★ PUT 与 PATCH：更新资源，PUT 对后台来说 PUT 方法的参数是一个完整的资源对象，它包含了对象的所有字段，PATCH 对后台来说 PATCH 方法的参数只包含我们需要修改的资源对象的字段。  
★ DELETE：删除资源。  
★ OPTIONS：获取目标资源所支持的通信选项,使用 OPTIONS 方法对服务器发起请求，以检测服务器支持哪些 HTTP 方法。

1. 简述HTTP请求的报文格式。
2. HTTP的长连接是什么意思。
3. HTTPS的加密方式是什么，讲讲整个加密解密流程。



1. Http和https的三次握手有什么区别
2. 什么是分块传送。
3. Session和cookie的区别。

1）、Cookie 保存在客户端，未设置存储时间的 Cookie，关闭浏览器会话 Cookie 就会被删除；设置了存储时间的 Cookie 保存在用户设备的磁盘中直到过期，同时 Cookie 在客户端所以可以伪造，不是十分安全，敏感数据不易保存。Session 保存在服务器端，存储在 IIS 的进程开辟的内存中，而 Session 过多会消耗服务器资源，所以尽量少使用 Session。

2）、Session 是服务器用来跟踪用户的一种手段，每个Session都有一个唯一标识：session ID。当服务端生成一个 Session 时就会向客户端发送一个 Cookie 保存到客户端，这个 Cookie 保存的是 Session 的 SessionID 这样才能保证客户端发起请求后，用户能够与服务器端成千上万的 Session 进行匹配，同时也保证了不同页面之间传值的正确性.

3）、存储数据类型不同：Session 能够存储任意的 JAVA 对象，Cookie 只能存储 String 类型的对象。

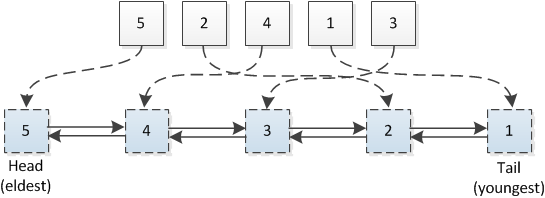
4）、大于10K 的数据，不要用到 Cookies。

# 架构设计与分布式

1. 用java自己实现一个LRU。

LRU（Least Recently Used：最近最少使用）：简单的说，就是保证基本的Cache容量，如果超过容量则必须丢掉最不常用的缓存数据，再添加最新的缓存。每次读取缓存都会改变缓存的使用时间，将缓存的存在时间重新刷新。其实，就是清理缓冲的一种策略。

我们可以通过双向链表的数据结构实现 LRU Cache，链表头（head）保存最新获取和存储的数据值，链表尾（tail）既为最不常使用的值，当需要清理时，清理链表的 tail 即可，并将前一个元素设置为tail。结构图如下：



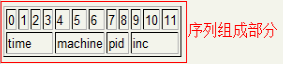
1. 分布式集群下如何做到唯一序列号。

分布式架构下，生成唯一序列号是设计系统常常会遇到的一个问题。例如，数据库使用分库分表的时候，当分成若干个 sharding 表后，如何能够快速拿到一个唯一序列号，是经常遇到的问题。实现思路如下：

【1】、数据库自增长序列或字段：全数据库唯一。  
  优点：简单，代码方便，性能可以接受。数字ID天然排序，对分页后者需要排序的结果很有帮组。适合小应用，无需分表，么有高并发性能要求。  
  缺点：不同数据库实现不同，在水平分表时，使用自增ID时可能会出现ID冲突。同时在高并发的情况下需要使用事务。在性能达不到要求的情况下，比较难于扩展。如果多个系统需要合并或者设计到数据迁移会相当痛苦。  
  优化：针对主库单点，如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。  
【2】、UUID：常见的方式。可以利用数据库也可以利用程序生成32位的16进制格式的字符串，唯一性很高。  
  优点：简单，代码方便，生产ID性能非常好且全球基本唯一，在数据迁移和系统后期合并，或数据库变更等情况下都可应对。  
  缺点：没有排序，无法保证趋势递增。UUID使用字符串存储，查询效率低。存储空间较大，如果数据海量就绪考虑存储量问题，传输数据量大。  
【3】、Redis生成ID：当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作 INCR和INCRBY来实现。可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：  
A：1,6,11,16,21  
B：2,7,12,17,22  
C：3,8,13,18,23  
D：4,9,14,19,24  
E：5,10,15,20,25  
这个，随便负载到哪个机确定好，未来很难做修改。但是3-5台服务器基本能够满足器上，都可以获得不同的ID。但是步长和初始值一定需要事先需要了。使用Redis集群也可以防止单点故障（系统中一点失效，就会让整个系统无法运作的部件）的问题。另外，比较适合使用Redis来生成每天从0开始的流水号。比如订单号=日期+当日自增长号。可以每天在Redis中生成一个Key，使用INCR进行累加。  
  优点：不依赖于数据库，灵活方便，且性能优于数据库。数字ID天然排序，对分页或者需要排序的结果很有帮助。  
  缺点：如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。需要编码和配置的工作量比较大。

【4】、Twitter（推特） 的 snowflake 算法

【5】、MongoDB 的 ObjectId：MongoDB 的 ObjectId 和 snowflake 算法类似。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求。使其在分片环境中要容易生成得多。



前4 个字节是从标准纪元开始的时间戳，单位为秒。时间戳，提供了秒级别的唯一性。由于时间戳在前，这意味着ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，如将其作为索引提高效率。这4 个字节也隐含了文档创建的时间。绝大多数客户端类库都会公开一个方法从ObjectId 获取这个信息。  
接下来的3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的ObjectId，不产生冲突。  
为了确保在同一台机器上并发的多个进程产生的ObjectId 是唯一的，接下来的两字节来自产生ObjectId 的进程标识符（PID）。  
前9 字节保证了同一秒钟不同机器不同进程产生的ObjectId 是唯一的。后3 字节就是一个自动增加的计数器，确保相同进程同一秒产生的ObjectId 也是不一样的。同一秒钟最多允许每个进程拥有不同的ObjectId。

【6】、其他一些方案：比如京东淘宝等电商的订单号生成。因为订单号和用户id 在业务上的区别，订单号尽可能要多些冗余的业务信息，比如：滴滴：时间+起点编号+车牌号 淘宝订单：时间戳+用户ID 其他电商：时间戳+下单渠道+用户ID，有的会加上订单第一个商品的ID。而用户ID，则要求含义简单明了，包含注册渠道即可，尽量短。

1. 设计一个秒杀系统，30分钟没付款就自动关闭交易。

*秒杀架构设计理念*：限流： 鉴于只有少部分用户能够秒杀成功，所以要限制大部分流量，只允许少部分流量进入服务后端。  
削峰：对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。高峰值流量是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术。  
异步处理：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。  
内存缓存：秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升。  
可拓展：当然如果我们想支持更多用户，更大的并发，最好就将系统设计成弹性可拓展的，如果流量来了，拓展机器就好了。像淘宝、京东等双十一活动时会增加大量机器应对交易高峰。

*设计思路*：将请求拦截在系统上游，降低下游压力：秒杀系统特点是并发量极大，但实际秒杀成功的请求数量却很少，所以如果不在前端拦截很可能造成数据库读写锁冲突，甚至导致死锁，最终请求超时。  
充分利用缓存：利用缓存可极大提高系统读写速度。  
消息队列：消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

*前端方案*：页面静态化：将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过CDN来抗峰值。  
禁止重复提交：用户提交之后按钮置灰，禁止重复提交。  
用户限流：在某一时间段内只允许用户提交一次请求，比如可以采取IP限流。

*后端方案*：*服务端控制器层(网关层)*：限制uid（UserID）访问频率：我们上面拦截了浏览器访问的请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问uid，限制访问频率。  
*服务层*：上面只拦截了一部分访问请求，当秒杀的用户量很大时，即使每个用户只有一个请求，到服务层的请求数量还是很大。比如我们有100W用户同时抢100台手机，服务层并发请求压力至少为100W。  
采用消息队列缓存请求：既然服务层知道库存只有100台手机，那完全没有必要把100W个请求都传递到数据库啊，那么可以先把这些请求都写到消息队列缓存一下，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。  
利用缓存应对读请求：对类似于12306等购票业务，是典型的读多写少业务，大部分请求是查询请求，所以可以利用缓存分担数据库压力。  
利用缓存应对写请求：缓存也是可以应对写请求的，比如我们就可以把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。

*数据库层*：数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。

*30分钟没付款就自动关闭交易*：要看你取消订单后需不需要恢复库存了，不需要的话可以插入个结束时间来做标识，如果需要恢复库存的话可使用 *swoole*里面的毫秒定时器 *swoole\_timer\_after*来实现，如果使用的是 *laravel*框架的话也可以使用延时队列来实现。我们公司在页面上会有倒计时刷新，其实也是通过页面定时器（setTiemout、setInterval）定时向后台发送请求。当超时后会进行订单状态的修改，达到自动关闭。

1. 如何使用redis和zookeeper实现分布式锁？有什么区别优缺点，会有什么问题，分别适用什么场景。（延伸：如果知道redlock，讲讲他的算法实现，争议在哪里）
2. 如果有人恶意创建非法连接，怎么解决。

通过Filter进行拦截处理。

1. 分布式事务的原理，优缺点，如何使用分布式事务，2pc 3pc 的区别，解决了哪些问题，还有哪些问题没解决，如何解决，你自己项目里涉及到分布式事务是怎么处理的。
2. 什么是一致性hash。
3. 什么是restful，讲讲你理解的restful。
4. 如何设计一个良好的API。
5. 如何设计建立和保持100w的长连接。
6. 解释什么是MESI协议(缓存一致性)。
7. 说说你知道的几种HASH算法，简单的也可以。
8. 什么是paxos算法， 什么是zab协议。
9. 一个在线文档系统，文档可以被编辑，如何防止多人同时对同一份文档进行编辑更新。
10. 线上系统突然变得异常缓慢，你如何查找问题。
11. 说说你平时用到的设计模式。
12. Dubbo的原理，有看过源码么，数据怎么流转的，怎么实现集群，负载均衡，服务注册和发现，重试转发，快速失败的策略是怎样的 。
13. 一次RPC请求的流程是什么。
14. 自己实现过rpc么，原理可以简单讲讲。Rpc要解决什么问题。
15. 异步模式的用途和意义。
16. 编程中自己都怎么考虑一些设计原则的，比如开闭原则，以及在工作中的应用。
17. 设计一个社交网站中的“私信”功能，要求高并发、可扩展等等。 画一下架构图。
18. MVC模式，即常见的MVC框架。
19. 聊下曾经参与设计的服务器架构并画图，谈谈遇到的问题，怎么解决的。
20. 应用服务器怎么监控性能，各种方式的区别。
21. 如何设计一套高并发支付方案，架构如何设计。
22. 如何实现负载均衡，有哪些算法可以实现。
23. Zookeeper的用途，选举的原理是什么。
24. Zookeeper watch机制原理。
25. Mybatis的底层实现原理。
26. 请思考一个方案，实现分布式环境下的countDownLatch。
27. 后台系统怎么防止请求重复提交。
28. 描述一个服务从发布到被消费的详细过程。
29. 讲讲你理解的服务治理。
30. 如何做到接口的幂等性。
31. 如何做限流策略，令牌桶和漏斗算法的使用场景。
32. 什么叫数据一致性，你怎么理解数据一致性。
33. 分布式服务调用方，不依赖服务提供方的话，怎么处理服务方挂掉后，大量无效资源请求的浪费，如果只是服务提供方吞吐不高的时候该怎么做，如果服务挂了，那么一会重启，该怎么做到最小的资源浪费，流量半开的实现机制是什么。
34. dubbo的泛化调用怎么实现的，如果是你，你会怎么做。
35. 远程调用会有超时现象，如果做到优雅的控制，JDK自带的超时机制有哪些，怎么实现的。

# 算法

1. 10亿个数字里里面找最小的10个。
2. 有1亿个数字，其中有2个是重复的，快速找到它，时间和空间要最优。
3. 2亿个随机生成的无序整数,找出中间大小的值。
4. 给一个不知道长度的（可能很大）输入字符串，设计一种方案，将重复的字符排重。
5. 遍历二叉树。
6. 有3n+1个数字，其中3n个中是重复的，只有1个是不重复的，怎么找出来。
7. 写一个字符串（如：www.javastack.cn）反转函数。
8. 常用的排序算法，快排，归并、冒泡。 快排的最优时间复杂度，最差复杂度。冒泡排序的优化方案。
9. 二分查找的时间复杂度，优势。
10. 一个已经构建好的TreeSet，怎么完成倒排序。
11. 什么是B+树，B-树，列出实际的使用场景。
12. 一个单向链表，删除倒数第N个数据。
13. 200个有序的数组，每个数组里面100个元素，找出top20的元素。
14. 单向链表，查找中间的那个元素。

# 数据库知识

1. 数据库隔离级别有哪些，各自的含义是什么，MYSQL默认的隔离级别是是什么。

READ UNCIMMITTED（未提交读）

READ COMMITTED（提交读）

REPEATABLE READ（可重复读）(mysql默认)

SERIALIZABLE（可串行化）

1. 什么是脏读，不可重复读，幻读。

1)脏读是指一个事务读取了未提交事务执行过程中的数据。

2)不可重复读是指对于数据库中的某个数据，一个事务执行过程中多次查询返回不同查询结果，这就是在事务执行过程中，数据被其他事务提交修改了。

不可重复读同脏读的区别在于，脏读是一个事务读取了另一未完成的事务执行过程中的数据，而不可重复读是一个事务执行过程中，另一事务提交并修改了当前事务正在读取的数据。

3)幻读是事务非独立执行时发生的一种现象，例如事务T1批量对一个表中某一列列值为1的数据修改为2的变更，但是在这时，事务T2对这张表插入了一条列值为1的数据，并完成提交。此时，如果事务T1查看刚刚完成操作的数据，发现还有一条列值为1的数据没有进行修改，而这条数据其实是T2刚刚提交插入的，这就是幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点同脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

1. MYSQL有哪些存储引擎，各自优缺点。

MySQL支持的多种存储引擎，可以通过“SHOW ENGINES”语句来查看。常用的存储引擎只有三种：默认的InnoDB、MyISAM和MEMORY。

1. 高并发下，如何做到安全的修改同一行数据。

1、使用悲观锁

悲观锁本质是当前只有一个线程执行操作，排斥外部请求的修改。遇到加锁的状态，就必须等待。结束了唤醒其他线程进行处理。虽然此方案的确解决了数据安全的问题，但是，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

2、FIFO（First Input First Output，先进先出）缓存队列思路

直接将请求放入队列中，就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程的感觉哈。

然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时间还是会大幅下降，系统还是陷入异常。

3、使用乐观锁

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

1. 乐观锁和悲观锁是什么，INNODB的标准行级锁有哪2种，解释其含义。

♪ 乐观锁（Optimistic Concurrency Control，缩写”OCC”）：

对表中的数据进行操作时，先给表中最新的数据加一个版本(version)字段，每操作一次，将该记录的版本号加1。也就是先查询出该记录，获取 version 字段，修改完数据后准备提交之前，先判断此刻 version 的值是否与刚刚查询出来时的 version 的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，并将 version 字段的值加1；如果更新时发现此刻的 version 值与刚刚获取出来的 version 的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。

♪ 悲观锁（Pessimistic Concurrency Control，缩写”PCC”）：

与乐观锁相对应的就是悲观锁。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟 java 中的 synchronized 很相似，所以悲观锁需要耗费较多的时间。所以悲观锁并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。

注意：要使用悲观锁，就必须关闭 Mysql 数据库的自动提交属性，因为 MySQL 默认使用 autocommit 模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。关闭设置：set autocommit=0;

InnoDB的标准行级锁有哪2种：

♬共享锁：共享锁指的就是对于多个不同的事务，对同一个资源共享同一个锁，在执行语句后面加上 lock in share mode 就代表对某些资源加上共享锁。

♬排它锁：排它锁与共享锁相对应，就是指对于多个不同的事务，对同一个资源只能有一把锁。在需要执行的语句后面加上 for update 就可以了。对于 update、insert、delete 语句会自动加排它锁。

1. SQL优化的一般步骤是什么，怎么看执行计划，如何理解其中各个字段的含义。

sql优化一般步骤概要：

1.通过 show status命令了解各种sql的执行频率

2.定位执行效率较低的sql语句

3.通过explain分析低效sql的执行计划

4.通过 show profile 分析sql

5.通过trace分析 优化器 如何选择执行计划

6.确定问题并采取相应的优化措施

1. 数据库会死锁吗，举一个死锁的例子，mysql怎么解决死锁。

♪ 产生死锁的原因:

事务之间对资源访问顺序的交替

并发修改同一记录

索引不当导致全表扫描

♪ 解决死锁:

①、重启数据库。

②、杀掉抢资源的进程：先查哪些进程在抢资源：

SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_TRX;

☎、杀掉抢资源的进程：

Kill trx\_mysql\_thread\_id;

1. MYsql的索引原理，索引的类型有哪些，如何创建合理的索引，索引如何优化。

MySql索引的原理：

1）、通过不断地缩小想要获取数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是说，有了这种索引机制，我们可以总用同一种查找方式来锁定数据。  
2）、索引是通过复杂的算法，提高数据查询性能的手段。从磁盘 io 到内存 io 的转变。

MySql索引的类型：

1）、普通索引 index：加速查找

2）、唯一索引：①、主键索引：primary key：加速查找+主键唯一约束且不为空。

  ②、唯一索引：unique index：加速查找+主键唯一约束。

3）、联合索引：①、primary key(id,name)：联合主键索引。

  ②、unique index(id,name)：联合唯一索引。

  ③、index(id,name）：联合普通索引。

4）、全文索引 fulltext index：用于搜索很长一篇文章的时候，效果最好。InnoDB 不支持

创建索引的原则:

经常用作查询条件的字段

经常用作表连接的字段

经常出现在 order by，group by 之后的字段

索引优化:

非空字段 NOT NULL，Mysql 很难对空值作查询优化

区分度高，离散度大，作为索引的字段值尽量不要有大量相同值

索引的长度不要太长（比较耗费时间）

1. 聚集索引和非聚集索引的区别。

聚集索引：就是索引和记录紧密在一起。InnoDB 数据文件就是索引文件，主键索引就是聚集索引  
非聚集索引：索引文件和数据文件分开存放，索引文件的叶子页只保存了主键值，要定位记录还要去查找相应的数据块。MyISAM 索引和数据分离，使用的是非聚集索引

1. select for update 是什么含义，会锁表还是锁行或是其他。

select for update 语句是我们经常使用手工加锁语句。借助 for update 子句，我们可以在应用程序的层面手工实现数据加锁保护操作。属于并发行锁。

1. 为什么要用Btree实现，它是怎么分裂的，什么时候分裂，为什么是平衡的。

Key 超过1024才分裂，因为随着数据的增多，一个结点的 key 满了，为了保持 B 树的特性，就会产生分裂，就向红黑树和 AVL树为了保持树的性质需要进行旋转一样！

1. 数据库的ACID是什么。

原子性（Atomicity）

一致性（Consistency）

隔离性（Isolation）

持久性（Durability）

1. 某个表有近千万数据，CRUD比较慢，如何优化。

数据千万级别之多，占用的存储空间也比较大，可想而知它不会存储在一块连续的物理空间上，而是链式存储在多个碎片的物理空间上。可能对于长字符串的比较，就用更多的时间查找与比较，这就导致用更多的时间。

1）、作为关系型数据库，是什么原因出现了这种大表？是否可以做表拆分，减少单表字段数量，优化表结构。

2）、在保证主键有效的情况下，检查主键索引的字段顺序，使得查询语句中条件的字段顺序和主键索引的字段顺序保持一致。

3）、在程序逻辑中采用手动事务控制，不要每插入一条数据就自动提交，而是定义一个计数器，进行批量手动提交，能够有效提高运行速度。

1. Mysql怎么优化table scan的。

 ■ 避免在 where 子句中对字段进行 is null 判断。

 ■ 应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将会导致引擎放弃使用索引而进行全表扫描。

 ■ 避免在 where 子句中使用 or 来连接条件。

 ■ in 和not in 也要慎用。

 ■ 避免Like 查询（like ‘abc%’ 才用到索引）。

 ■ 避免在 where 子句中使用参数, 使用 NUM=@num 参数这种。

 ■ 避免在where 子句中对字段进行表达式操作 num/2=XX。

 ■ 在where子句中对字段进行函数操作。

1. 如何写sql能够有效的使用到复合索引。

对于复合索引,在查询使用时,最好将条件顺序按找索引的顺序,这样效率最高

例如:

CREATE INDEX columnId ON table1(col1,col2,col3) ;

select \* from table1 where col1= A and col2= B and col3 = C

1. mysql中in 和exists 区别。

mysql 中的 in 语句是把外表和内表作 hash 连接，而 exists 语句是对外表作 loop 循环，每次 loop 循环再对内表进行查询。一直大家都认为 exists 比 in 语句的效率要高，这种说法其实是不准确的。这个是要区分环境的。

㊤、如果查询的两个表大小相当，那么用 in 和 exists 差别不大。

㊥、如果两个表中一个较小，一个是大表，则子查询表大的用 exists，子查询表小的用 in。

㊦、not in 和 not exists 如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而 not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用 not exists 都比 not in 要快。

EXISTS 只返回 TRUE 或 FALSE，不会返回 UNKNOWN  
IN 当遇到包含NULL的情况，那么就会返回 UNKNOWN

1. 数据库自增主键可能的问题。

【1】、使用自增主键对数据库做分库分表，可能出现一些诸如主键重复等的问题。

【2】、数据库导入的时候，可能会因为主键出现一些问题。

1. MVCC的含义，如何实现的。

MVCC：Multi-Version Concurrency Control 多版本并发控制，MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问；在编程语言中实现事务内存。MVCC 最大的好处，相信也是耳熟能详：读不加锁，读写不冲突。在读多写少的 OLTP 应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的 RDBMS，都支持了 MVCC。

1. 你做过的项目里遇到分库分表了吗，怎么做的，有用到中间件么，比如sharding jdbc等,他们的原理知道么。

垂直分表

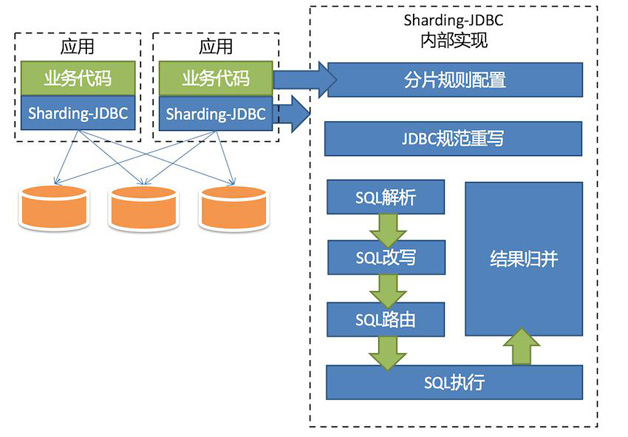
垂直分库

水平分表

水平分库分表

Sharding-JDBC是实现了JDBC协议的jar文件。基于JDBC协议的实现与基于MySQL等数据库协议实现的中间层略有差别。

无论使用哪种架构，核心逻辑均极为相似，除了协议实现层不同（JDBC或数据库协议），都会分为分片规则配置、SQL解析、SQL改写、SQL路由、SQL执行以及结果归并等模块。



1. MYSQL的主从延迟怎么解决。

原因分析:

MySQL的主从复制都是单线程的操作，主库对所有DDL和DML产生的日志写进binlog，由于binlog是顺序写，所以效率很高。Slave的SQL Thread线程将主库的DDL和DML操作事件在slave中重放。DML和DDL的IO操作是随即的，不是顺序的，成本高很多。另一方面，由于SQL Thread也是单线程的，当主库的并发较高时，产生的DML数量超过slave的SQL Thread所能处理的速度，或者当slave中有大型query语句产生了锁等待那么延时就产生了。

常见原因：Master负载过高、Slave负载过高、网络延迟、机器性能太低、MySQL配置不合理。

解决从库复制延迟的问题：

1. 优化网络

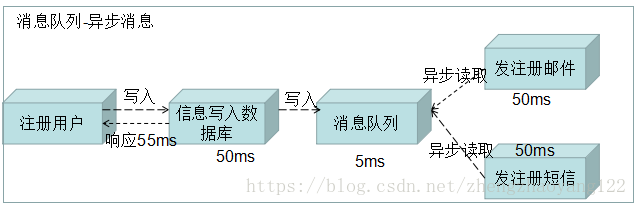
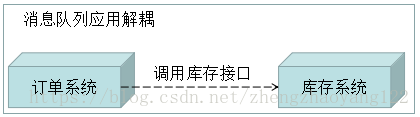
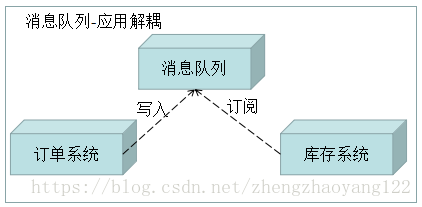
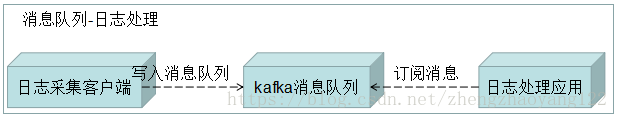
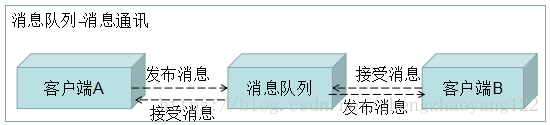
2. 升级Slave硬件配置

3. Slave调整参数，将 sync\_binlog 设置为 0 或者关闭 binlog，修改innodb\_flush\_log\_at\_trx\_commit参数值为 0

4. 增加从服务器，这个目的还是分散读的压力， 从而降低服务器负载

# 消息队列

1. 消息队列的使用场景。

 ☞ 以下介绍消息队列在实际应用常用的使用场景。异步处理、应用解耦、流量削锋和消息通讯四个场景。  
 1）、异步处理：场景说明：用户注册后，需要发注册邮件和注册短信。  
        
      引入消息队列后架构如下：用户的响应时间=注册信息写入数据库的时间，例如50毫秒。**发注册邮箱、发注册短信写入消息队列**后，直接返回客户端，因写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50毫秒。按照传统的做法①、串行方式，将注册信息写入数据库成功后，发注册邮件，再发送注册短信，以上三个成功后，返回客户端。可能需要150毫秒，这样使用消息队列提高了3倍。②、并行方式，将注册信息写入数据库成功后，发送注册邮件，同时发送注册短信。也可能需要100毫秒，这样使用消息队列提高了2倍。  
  2）、应用解耦：场景说明：用户下单后，订单系统需要通知库存系统。如下图：  
            
 传统模式的缺点：1）、库存系统无法访问时，则订单减库存业务将会失败，从而导致订单失败。  
                              2）、订单系统与库存系统耦合。  
 引入消息队列：1）、用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功。  
                          2）、库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作。  
  ☛ 当库存系统不能正常使用时，也不会影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的**解耦**。  
         
  3）、流量削锋：场景说明：秒杀或团抢活动中使用广泛。秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。一般需要在应用前端加入消息队列。  
         
       用户请求：服务器接受后，首先写入消息队列。当消息队列长度超出最大数量，则直接抛弃用户请求或跳转至错误页面。  
       秒杀业务处理：根据消息队列中的请求信息，再做后续处理。  
  ▁▂▃ 这样可以有效的控制活动人数和有效缓解短时间内的高流量冲击，防止压垮应用系统。  
  4）、日志处理：指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。  
        
   ▷ 日志采集客户端：负责日志数据采集，定时写受写入Kafka队列。  
   ▷ kafka消息队列：负责日志数据的接收，存储和转发。  
   ▷ 日志处理应用：订阅并消费kafka队列中的日志数据。  
  5）、消息通信：消息队列一般都内置了高效的通信机制，因此也可以用纯消息通信。比如实现点对点消息队列，或者聊天室。  
     【1】、点对点通讯：客户端A和客户端B使用同一队列，进行消息通讯。  
        
     【2】、聊天室通讯（发布订阅模式）：客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。  
      

1. 如何保证消息的有序性。
2. 用过哪些MQ，和其他mq比较有什么优缺点，MQ的连接是线程安全的吗，你们公司的MQ服务架构怎样的。
3. MQ系统的数据如何保证不丢失。
4. rabbitmq如何实现集群高可用。
5. kafka吞吐量高的原因。

1）、顺序读写磁盘，充分利用了操作系统的预读机制。  
2）、Linux中使用sendfile命令，减少一次数据拷贝：  
        ①、把数据从硬盘读取到内核中的页缓存。  
        ②、把数据从内核中读取到用户空间（sendfile命令跳过此步骤）。        ③、把用户空间的数据写到socket缓存区中。  
        ④、操作系统将数据从socket缓冲区中复制到网卡缓冲区，以便将数据经网络发出。  
3）、生产者客户端缓存消息批量发送，消费者批量从broker获取消息，减少IO次数，充分利用磁盘顺序读写的性能。  
4）、通常情况下kafka的瓶颈不是CPU或者磁盘，而是网络宽带，所以生产者可以对数据进行压缩。

1. kafka 和其他消息队列的区别，kafka 主从同步怎么实现。
2. 利用mq怎么实现最终一致性。
3. 使用kafka有没有遇到什么问题，怎么解决的。
4. MQ有可能发生重复消费，如何避免，如何做到幂等。
5. MQ的消息延迟了怎么处理，消息可以设置过期时间么，过期了你们一般怎么处理。

# 缓存

1. 常见的缓存策略有哪些，如何做到缓存(比如redis)与DB里的数据一致性，你们项目中用到了什么缓存系统，如何设计的。
2. 如何防止缓存击穿和雪崩。
3. 缓存数据过期后的更新如何设计。
4. redis的list结构相关的操作。
5. Redis的数据结构都有哪些。

【1】、String：可以是字符串，整数或者浮点数，对整个字符串或者字符串中的一部分执行操作，对整个整数或者浮点执行自增(increment)或者自减(decrement)操作。  
【2】、List：一个链表，链表上的每个节点都包含了一个字符串，链表的两端推入或者弹出元素，根据偏移量对链表进行修剪(trim)，读取单个或者多个元素，根据值查找或者移除元素。可参考5  
【3】、Set：包含字符串的无序收集器(unordered collection)、并且被包含的每个字符串都是独一无二的。添加，获取，移除单个元素，检查一个元素是否存在于集合中，计算交集（sinter），并集（suion），差集（sdiff），从集合里面随机获取元素。  
【4】、SortSet：是一个排好序的 Set，它在 Set 的基础上增加了一个顺序属性 score，这个属性在添加修改元素时可以指定，每次指定后，SortSet 会自动重新按新的值排序。sorted set 的内部使用 HashMap 和跳跃表(SkipList)来保证数据的存储和有序，HashMap 里放的是成员到 score 的映射，而跳跃表里存放的是所有的成员，排序依据是 HashMap 里存的 score。

【 5】、hash：Hash 是一个 String 类型的 field 和 value 之间的映射表，即 redis 的 Hash 数据类型的 key（hash表名称）对应的 value 实际的内部存储结构为一个 HashMap，因此 Hash 特别适合存储对象。相对于把一个对象的每个属性存储为 String 类型，将整个对象存储在 Hash 类型中会占用更少内存。

1. Redis的使用要注意什么，讲讲持久化方式，内存设置，集群的应用和优劣势，淘汰策略等。
2. redis2和redis3的区别，redis3内部通讯机制。
3. 当前redis集群有哪些玩法，各自优缺点，场景。
4. Memcache的原理，哪些数据适合放在缓存中。
5. redis和memcached 的内存管理的区别。
6. Redis的并发竞争问题如何解决，了解Redis事务的CAS操作吗。
7. Redis的选举算法和流程是怎样的。
8. redis的持久化的机制，aof和rdb的区别。
9. redis的集群怎么同步的数据的。
10. 知道哪些redis的优化操作。
11. Reids的主从复制机制原理。
12. Redis的线程模型是什么。
13. 请思考一个方案，设计一个可以控制缓存总体大小的自动适应的本地缓存。
14. 如何看待缓存的使用（本地缓存，集中式缓存），简述本地缓存和集中式缓存和优缺点。
15. 本地缓存在并发使用时的注意事项。

# 搜索

1. elasticsearch了解多少，说说你们公司es的集群架构，索引数据大小，分片有多少，以及一些调优手段 。elasticsearch的倒排索引是什么。
2. elasticsearch 索引数据多了怎么办，如何调优，部署。
3. elasticsearch是如何实现master选举的。
4. 详细描述一下Elasticsearch索引文档的过程。
5. 详细描述一下Elasticsearch搜索的过程。
6. Elasticsearch在部署时，对Linux的设置有哪些优化方法？
7. lucence内部结构是什么。