

Java基础

Vector, ArrayList, LinkedList的区别是什么？

答：

1. Vector、ArrayList都是以类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储。
2. List中的元素有序、允许有重复的元素，Set中的元素无序、不允许有重复元素。
3. Vector线程同步，ArrayList、LinkedList线程不同步。
4. LinkedList适合指定位置插入、删除操作，不适合查找；ArrayList、Vector适合查找，不适合指定位置的插入、删除操作。
5. ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。

HashTable, HashMap, TreeMap区别？

答：

1. HashTable线程同步，HashMap非线程同步。
2. HashTable不允许<键,值>有空值，HashMap允许<键,值>有空值。
3. HashTable使用Enumeration，HashMap使用Iterator。
4. HashTable中hash数组的默认大小是11，增加方式的 $old * 2 + 1$ ，HashMap中hash数组的默认大小是16，增长方式一定是2的指数倍。
5. TreeMap能够把它保存的记录根据键排序，默认是按升序排序。

HashMap的数据结构

jdk1.8之前list + 链表

jdk1.8之后list + 链表（当链表长度到8时，转化为红黑树）

HashMap的扩容因子

默认0.75，也就是会浪费1/4的空间，达到扩容因子时，会将list扩容一倍，0.75 是时间与空间一个平衡值；

多线程修改HashMap

多线程同时写入，同时执行扩容操作，多线程扩容可能死锁、丢数据；可以对HashMap 加入同步锁Collections.synchronizedMap(hashMap)，但是效率很低，因为该锁是互斥锁，同一时刻只能有一个线程执行读写操作，这时候应该使用ConcurrentHashMap

LinkedHashMap

[Java LinkedHashMap工作原理及实现](#)

[Java集合框架：LinkedHashMap](#)

注意：在使用Iterator遍历的时候，LinkedHashMap会产生
java.util.ConcurrentModificationException。

扩展HashMap增加双向链表的实现，号称是最占内存的数据结构。支持iterator()时按Entry的插入顺序来排序(但是更新不算，如果设置accessOrder属性为true，则所有读写访问都算)。实现上是在Entry上再增加属性before/after指针，插入时把自己加到Header Entry的前面去。如果所有读写访问都要排序，还要把前后Entry的before/after拼接起来以在链表中删除掉自己。

说说你知道的几个Java集合类：list、set、queue、map实现类

描述一下ArrayList和LinkedList各自实现和区别

[Java基础篇\(四\):ArrayList和LinkedList内部实现、区别、使用场景](#)

Java中的队列都有哪些，有什么区别

1. ArrayDeque, (数组双端队列)
2. PriorityQueue, (优先级队列)
3. ConcurrentLinkedQueue, (基于链表的并发队列)
4. DelayQueue, (延期阻塞队列) (阻塞队列实现了BlockingQueue接口)
5. ArrayBlockingQueue, (基于数组的并发阻塞队列)
6. LinkedBlockingQueue, (基于链表的FIFO阻塞队列)
7. LinkedBlockingDeque, (基于链表的FIFO双端阻塞队列)
8. PriorityBlockingQueue, (带优先级的无界阻塞队列)
9. SynchronousQueue (并发同步阻塞队列)

反射中，Class.forName和classloader的区别

[java反射中，Class.forName和classloader的区别\(代码说话\)](#)

Java7、Java8的新特性

[java7,8的几个特性](#)

Java数组和链表两种结构的操作效率，在哪些情况下(从开头开始，从结尾开始，从中间开始)，哪些操作(插入，查找，删除)的效率
讲讲IO里面的常见类，字节流、字符流、接口、实现类、方法阻塞

[Java IO流详解（二）——IO流的框架体系](#)

讲讲NIO

[NIO技术概览](#)

缓冲区

虚拟内存&&内存空间的映射

三个channel使用

ServerSocketChannel || SocketChannel || FileChannel

[Java NIO系列教程（八） SocketChannel](#)

[Java NIO系列教程（九） ServerSocketChannel](#)

[Java NIO系列教程（七） FileChannel](#)

String 编码UTF-8 和GBK的区别

- GBK编码：是指中国的中文字符，其实它包含了简体中文与繁体中文字符，另外还有一种字符“gb2312”，这种字符仅能存储简体中文字符。
- UTF-8编码：它是一种全国家通过的一种编码，如果你的网站涉及到多个国家的语言，那么建议你选择UTF-8编码。

GBK和UTF8有什么区别？

UTF8编码格式很强大，支持所有国家的语言，正是因为它的强大，才会导致它占用的空间大小要比GBK大，对于网站打开速度而言，也是有一定影响的。

GBK编码格式，它的功能少，仅限于中文字符，当然它所占用的空间大小会随着它的功能而减少，打开网页的速度比较快。

什么时候使用字节流、什么时候使用字符流

[什么时候使用字节流、什么时候使用字符流，二者的区别](#)

递归读取文件夹下的文件，代码怎么实现

```
/**
 * 递归读取文件夹下的 所有文件
 *
 * @param testFileDir 文件名或目录名
 */
private static void testLoopOutAllFileName(String testFileDir) {
    if (testFileDir == null) {
        //因为new File(null)会空指针异常,所以要判断下
        return;
    }
    File[] testFile = new File(testFileDir).listFiles();
    if (testFile == null) {
        return;
    }
    for (File file : testFile) {
```

```

        if (file.isFile()) {
            System.out.println(file.getName());
        } else if (file.isDirectory()) {
            System.out.println("-----this is a directory, and its files are as follows:-----");
            testLoopOutAllFileName(file.getPath());
        } else {
            System.out.println("文件读入有误！");
        }
    }
}

```

Object.finalize

[深入分析Object.finalize方法的实现原理](#)

SynchronousQueue实现原理

[SynchronousQueue实现原理](#)

跳表SkipList

[跳表（SkipList）及ConcurrentSkipListMap源码解析](#)

Collections.sort排序算法

[深入jdk——追踪Collections.sort 引发的bug（1）mergeSort](#)

自定义类加载器

[JVM——自定义类加载器](#)

Java并发和并行

- 并发：是指两个或多个事件在同一时间间隔发生,在一台处理器上“同时”处理多个任务;
- 并行：是指两个或者多个事件在同一时刻发生,在多台处理器上同时处理多个任务。

怎么提高并发量，请列举你所知道的方案？

[高并发解决方案——提升高并发量服务器性能解决思路](#)

系统的用户量有多少？多用户并发访问时如何解决？

[大型网站是怎样解决多用户高并发访问的](#)

说说阻塞队列的实现：可以参考ArrayBlockingQueue的底层实现（锁和同步都行）

[Java阻塞队列ArrayBlockingQueue和LinkedBlockingQueue实现原理分析](#)

进程通讯的方式：消息队列，共享内存，信号量，socket通讯等

[Linux进程间通信方式--信号，管道，消息队列，信号量，共享内存](#)

用过并发包的哪些类

Excutors可以产生哪些线程池

为什么要用线程池

[为什么要使用线程池](#)

线程池的基础概念

core, maxPoolSize, keepalive

执行任务时

1. 如果线程池中线程数量 $< \text{core}$ ，新建一个线程执行任务；
2. 如果线程池中线程数量 $\geq \text{core}$ ，则将任务放入任务队列
3. 如果线程池中线程数量 $\geq \text{core}$ 且 $< \text{maxPoolSize}$ ，则创建新的线程；
4. 如果线程池中线程数量 $> \text{core}$ ，当线程空闲时间超过了keepalive时，则会销毁线程；由此可见线程池的队列如果是无界队列，那么设置线程池最大数量是无效的；

自带线程池的各种坑

1. Executors.newFixedThreadPool(10);

固定大小的线程池：

它的实现new ThreadPoolExecutor(10, 10, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());

初始化一个指定线程数的线程池，其中corePoolSize == maximumPoolSize，使用LinkedBlockingQueue作为阻塞队列，当线程池没有可执行任务时，也不会释放线程。

由于LinkedBlockingQueue的特性，这个队列是无界的，若消费不过来，会导致内存被任务队列占满，最终oom；

2. Executors.newCachedThreadPool();

缓存线程池：

它的实现new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS, new SynchronousQueue<Runnable>());

初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到

Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销，因为线程池的最大值了

Integer.MAX_VALUE，会导致无限创建线程；所以，使用该线程池时，一定要注意控制并发的任务数，否则创建大量的线程会导致严重的性能问题；

3. Executors.newSingleThreadExecutor()

单线程线程池：

同newFixedThreadPool线程池一样，队列用的是LinkedBlockingQueue无界队列，可以无限的往里面添加任务，直到内存溢出；

volatile关键字的用法：使多线程中的变量可见

[Java并发编程：volatile关键字解析](#)

线程的几种状态

线程在一定条件下，状态会发生变化。线程一共有以下几种状态：

1. 新建状态(New)：新创建了一个线程对象。
2. 就绪状态(Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取CPU的使用权。即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。
3. 运行状态(Running)：就绪状态的线程获取了CPU，执行程序代码。
4. 阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

- a. 等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，
 - b. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。
 - c. 其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。
5. 死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

常用的线程池模式以及不同线程池的使用场景

[java线程池与五种常用线程池策略使用与解析](#)

newFixedThreadPool此种线程池如果线程数达到最大值后会怎么办，底层原理。

多线程之间通信的同步问题，synchronized锁的是对象，衍伸出和synchronized相关很多的具体问题，例如同一个类不同方法都有synchronized锁，一个对象是否可以同时访问。或者一个类的static构造方法加上synchronized之后的锁的影响。

了解可重入锁的含义，以及ReentrantLock 和synchronized的区别同步的数据结构，例如concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高

atomicinteger和Volatile等线程安全操作的关键字的理解和使用CAS和volatile关键字

通过volatile修饰的变量可以保证线程之间的可见性，但并不能保证字节码指令的原子执行，在多线程并发执行下，无法做到线程安全，得到正确的结果

1. 加锁(低效率)

2. cas

引用占小狼的简书：[面试必问的CAS，要多了解](#)

线程间通信，wait和notify

[wait和notify的理解与使用](#)

定时线程的使用

场景：在一个主线程中，要求有大量(很多很多)子线程执行完之后，主线程才执行完成。多种方式，考虑效率。

[java线程池主线程等待子线程执行完成](#)

进程和线程的区别

- 进程：

是具有一定独立功能的程序、它是系统进行资源分配和调度的一个独立单位，重点在系统调度和单独的单位，也就是说进程是可以独立运行的一段程序(比如正在运行的某个java程序)。

- 线程：

他是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源(一个线程只能属于一个进程，而一个进程可以有多个线程)。

什么叫线程安全？举例说明

- java中的线程安全是什么：

就是线程同步的意思，就是当一个程序对一个线程安全的方法或者语句进行访问的时候，其他的不能再对他进行操作了，必须等到这次访问结束以后才能对这个线程安全的方法进行访问

- 什么叫线程安全：

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，

就是线程安全的。

或者说：一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性，也就是说我们不用考虑同步的问题。

线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

存在竞争的线程不安全，不存在竞争的线程就是安全的

并发、同步的接口或方法

[Java并发编程的类、接口和方法](#)

HashMap 是否线程安全，为何不安全。 ConcurrentHashMap，线程安全，为何安全。底层实现是怎么样的。

[深入浅出ConcurrentHashMap1.8](#)

[谈谈ConcurrentHashMap1.7和1.8的不同实现](#)

[ConcurrentHashMap的红黑树实现分析](#)

[深入分析ConcurrentHashMap1.8的扩容实现](#)

[老生常谈，HashMap的死循环](#)

J. U. C下的常见类的使用。 ThreadPool的深入考察； BlockingQueue的使用。（take，poll的区别，put，offer的区别）；原子类的实现。

volatile的理解

[java volatile关键字解惑](#)

[面试必问的volatile，你了解多少？](#)

Tomcat并发

有个每秒钟5k个请求，查询手机号所属地的笔试题(记得不完整，没列出)，如何设计算法?请求再多，比如5w，如何设计整个系统?

高并发情况下，我们系统是如何支撑大量的请求的

- 尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。
- 用jprofiler等工具找出性能瓶颈，减少额外的开销。
- 优化数据库查询语句，减少直接使用hibernate等工具的直接生成语句（仅耗时较长的查询做优化）。
- 优化数据库结构，多做索引，提高查询效率。
- 统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。
- 能使用静态页面的地方尽量使用，减少容器的解析（尽量将动态内容生成静态html来显示）。
- 解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

1. HTML静态化

效率最高、消耗最小的就是纯静态化的html页面，所以尽可能使网站上的页面采用静态页面来实现，这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站，无法全部手动去挨个实现，于是出现了常见的信息发布系统CMS，像常访问的各个门户站点的新闻频道，甚至他们的其他频道，都是通过信息发布系统来管理和实现的，信息发布系统可以实现最简单的信息录入自动生成静态页面，还能具备频道管理、权限管理、自动抓取等功能，对于一个大型网站来说，拥有一套高效、可管理的CMS是必不可少的。

2. 图片服务器分离

对于Web服务器来说，不管是Apache、IIS还是其他容器，图片是最消耗资源的，于是有必要将图片与页面进行分离，这是基本上大型网站都会采用的策略，他们都有独立的图片服务器，甚至很多台图片服务器。这样的架构可以降低提供页面访问请求的服务器系统压力，并且可以保证系统不会因为图片问题而崩溃，在应用服务器和图片服务器上，可以进行不同的配置优化，比如apache在配置ContentType的时候可以尽量少支持，尽可能少的LoadModule，保证更高的系统消耗和执行效率。这一实现起来是比较容易的一现，如果服务器集群操作起来更方便，如果是独立的服务器，新手可能出现上传图片只能在服务器本地的情况下，可以在令一台服务器设置的IIS采用网络路径来实现图片服务器，即不用改变程序，又能提高性能，但对于服务器本身的IO处理性能是没有任何的改变。

3. 数据库集群和库表散列

大型网站都有复杂的应用，这些应用必须使用数据库，那么在面对大量访问的时候，数据库的瓶颈很快就能显现出来，这时一台数据库将很快无法满足应用，于是需要使用数据库集群或者库表散列。

4. 缓存

缓存一词搞技术的都接触过，很多地方用到缓存。网站架构和网站开发中的缓存也是非常重要。架构方面的缓存，对Apache比较熟悉的人都能知道Apache提供了自己的缓存模块，也可以使用外加的Squid模块进行缓存，这两种方式均可以有效的提高Apache的访问响应能力。网站程序开发方面的缓存，Linux上提供的Memory Cache是常用的缓存接口，可以在web开发中使用，比如用Java开发的时候就可以调用MemoryCache对一些数据进行缓存和通讯共享，一些大型社区使用了这样的架构。另外，在使用web语言开发的时候，各种语言基本都有自己的缓存模块和方法，PHP有Pear的Cache模块，Java就更多了，.net不是很熟悉，相信也肯定有。

5. 镜像

镜像大型网站常采用的提高性能和数据安全性的方式，镜像的技术可以解决不同网络接入商和地域带来的用户访问速度差异，比如ChinaNet和EduNet之间的差异就促使了很多网站在教育网内搭建镜像站点，数据进行定时更新或者实时更新。在镜像的细节技术方面，这里不阐述太深，有很多专业的现成的解决架构和产品可选。也有廉价的通过软件实现的思路，比如Linux上的rsync等工具。

6. 负载均衡

负载均衡将是大型网站解决高负荷访问和大量并发请求采用的终极解决办法。负载均衡技术发展了多年，有很多专业的服务提供商和产品可以选择。

硬件四层交换

第四层交换使用第三层和第四层信息包的报头信息，根据应用区间识别业务流，将整个区间的业务流分配到合适的应用服务器进行处理。第四层交换功能就象是虚IP，指向物理服务器。它传输的业务服从的协议多种多样，有HTTP、FTP、NFS、Telnet或其他协议。这些业务在物理服务器基础上，需要复杂的载量平衡算法。在IP世界，业务类型由终端TCP或UDP端口地址来决定，在第四层交换中的应用区间则由源端和终端IP地址、TCP和UDP端口共同决定。

在硬件四层交换产品领域，有一些知名的产品可以选择，比如Alteon、F5等，这些产品很昂贵，但是物有所值，能够提供非常优秀的性能和很灵活的管理能力。Yahoo中国当初接近2000台服务器使用了三四台Alteon就搞定了。

集群如何同步会话状态

[集群session一致性和同步问题](#)

[基于ZooKeeper的分布式Session实现](#)

负载均衡的原理

[六大Web负载均衡原理与实现](#)

如果有一个特别大的访问量，到数据库上，怎么做优化（DB设计，DBIO，SQL优化，Java优化）

[数据库SQL优化大总结之 百万级数据库优化方案](#)

如果出现大面积并发，在不增加服务器的基础上，如何解决服务器响应不及时问题

[如何提高服务器并发处理能力](#)

假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。

如何查找 造成 性能瓶颈出现的位置，是哪个位置造成性能瓶颈。

你的项目中使用过缓存机制吗？有没用用户非本地缓存

Semaphore、CountDownLatch、CyclicBarrier、Phaser

[并发工具类（二）同步屏障CyclicBarrier](#)

[并发工具类（三）控制并发线程数的Semaphore](#)

CLH队列

[【Java并发编程实战】—— AQS\(四\)：CLH同步队列](#)

[JAVA并发编程学习笔记之CLH队列锁](#)

产生死锁的必要条件

[操作系统：死锁的产生、条件、和解锁](#)

Java内存模型

设计模式

单例模式：饱汉、饿汉。以及饿汉中的延迟加载, 双重检查

工厂模式、装饰者模式、观察者模式。

工厂方法模式的优点（低耦合、高内聚，开放封闭原则）

如何理解观察者模式？

列举出你说熟悉的设计模式，并对其中的一种的使用举一个例子。

JVM

User user = new User() 做了什么操作，申请了哪些内存？

1. new User(); 创建一个User对象，内存分配在堆上
2. User user; 创建一个引用，内存分配在栈上
3. = 将User对象地址赋值给引用

Java的内存模型以及GC算法

[JVM内存模型与GC算法](#)

jvm性能调优都做了什么

[JVM性能调优](#)

介绍JVM中7个区域，然后把每个区域可能造成内存的溢出的情况说明

介绍GC 和GC Root不正常引用

自己从classload 加载方式，加载机制说开去，从程序运行时数据区，讲到内存分配，讲到String常量池，讲到JVM垃圾回收机制，算法，hotspot。反正就是各种扩展

[java classload 机制 详解](#)

jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析

[JVM直接内存](#)

[浅谈HotSpot逃逸分析](#)

[触摸java常量池](#)

数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold，问通常多大，没做过一问便知）

[深入理解JVM（6）：Java对象内存分配策略](#)

注意:PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效，Parallel Scavenge收集器不认识这个参数，Parallel Scavenge收集器一般并不需要设置。

如果遇到必须使用此参数的场合，可以考虑ParNew加CMS的收集器组合。

老年代中数组的访问方式

GC算法，永久代对象如何GC，GC有环怎么处理

永久代GC的原因：

- 永久代空间已经满了
- 调用了System.gc()

注意： 这种GC是full GC 堆空间也会一并被GC一次

GC有环怎么处理

1. 根搜索算法

什么是根搜索算法

垃圾回收器从被称为GC Roots的点开始遍历遍历对象，凡是可以达到的点都会标记为存活，堆中不可到达的对象都会标记成垃圾，然后被清理掉。

GC Roots有哪些

- 类，由系统类加载器加载的类。这些类从不会被卸载，它们可以通过静态属性的方式持有对象的引用。

注意，一般情况下由自定义的类加载器加载的类不能成为GC Roots

- 线程，存活的线程
- Java方法栈中的局部变量或者参数
- JNI方法栈中的局部变量或者参数
- JNI全局引用
- 用做同步监控的对象

被JVM持有的对象，这些对象由于特殊的目的不被GC回收。这些对象可能是系统的类加载器，一些重要的异常处理类，一些为处理异常预留的对象，以及一些正在执行类加载的自定义的类加载器。但是具体有哪些前面提到的对象依赖于具体的JVM实现。

2. 如何处理

基于引用对象遍历的垃圾回收器可以处理循环引用，只要是涉及到的对象不能从GC Roots强引用可到达，垃圾回收器都会进行清理来释放内存。

谁会被GC，什么时候GC

[Java JVM：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）](#)

如果想不被GC怎么办

如果想在 GC 中生存 1 次怎么办

生存一次，释放掉对象的引用，但是在对象的finalize方法中重新建立引用，但是此方法只会被调用一次，所以能在GC中生存一次。

分析System.gc()方法

[JVM源码分析之SystemGC完全解读](#)

JVM 选项 -XX:+UseCompressedOops 有什么作用？为什么要使用？

当你将你的应用从 32 位的 JVM 迁移到 64 位的 JVM 时，由于对象的指针从 32 位增加到了 64 位，因此堆内存会突然增加，差不多要翻倍。这也会对 CPU 缓存（容量比内存小很多）的数据产生不利的影响。因为，迁移到 64 位的 JVM 主要动机在于可以指定最大堆大

小，通过压缩 OOP 可以节省一定的内存。通过 `-XX:+UseCompressedOops` 选项，JVM 会使用 32 位的 OOP，而不是 64 位的 OOP。

写代码分别使得JVM的堆、栈和持久代发生内存溢出(栈溢出)

[JVM内存溢出详解（栈溢出，堆溢出，持久代溢出以及无法创建本地线程）](#)

为什么jdk8用metaspace数据结构用来替代perm?

[JDK8: PermGen变更为MetaSpace详解](#)

简单谈谈堆外内存以及你的理解和认识

[JVM源码分析之堆外内存完全解读](#)

threadlocal使用场景及注意事项

[threadlocal原理及常用应用场景](#)

JVM老年代和新生代的比例？

[JVM-堆学习之新生代老年代持久带的使用关系](#)

栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗？

1. 从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。
2. 堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。
3. 栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。
4. 面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改

变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

为什么不把基本类型放堆中呢？

因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4byte的引用（堆栈分离的好处：））。

为什么不把基本类型放堆中呢？因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

Java中的参数传递时传值呢？还是传引用？

要说明这个问题，先要明确两点：

1. 不要试图与C进行类比，Java中没有指针的概念
2. 程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

明确以上两点后。Java在方法调用传递参数时，因为没有指针，所以它都是进行传值调用（这点可以参考C的传值调用）。因此，很多书里面都说Java是进行传值调用，这点没有问题，而且也简化的C中复杂性。

但是传引用的错觉是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修

改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，栈的大小通过-Xss来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现java.lang.StackOverflowError异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点

对象引用类型分为哪几类？

[java中四种引用类型（对象的强、软、弱和虚引用）](#)

讲一讲内存分代及生命周期。

[管中窥豹——从对象的生命周期梳理JVM内存结构、GC、类加载、AOP编程及性能监控](#)

什么情况下触发垃圾回收？

[Java JVM 8：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）](#)

如何选择合适的垃圾收集算法？

[深入理解垃圾收集器和收集器的选择策略](#)

JVM给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置 进行判断。

吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

```
java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -  
XX:ParallelGCThreads=20
```

-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

```
java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -
```

XX:ParallelGCThreads=20 -XX:+UseParallelOldGC

-XX:+UseParallelOldGC : 配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -

XX:MaxGCPauseMillis=100

-XX:MaxGCPauseMillis=100 : 设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -

XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy

-XX:+UseAdaptiveSizePolicy : 设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置 :

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -

XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:+UseConcMarkSweepGC : 设置年老代为并发收集。测试中配置这个以后，-

XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC : 设置年轻代为并行收集。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，所以无需再设置此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -

XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction : 由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection : 打开对年老代的压缩。可能会影响性能，但是可以消除碎片

StringTable

[探索StringTable提升YGC性能](#)

[JVM源码分析之String.intern\(\)导致的YGC不断变长](#)

JVM中最大堆大小有没有限制？

[JVM调优常用参数配置](#)

如何进行JVM调优？有哪些方法？

[如何合理的规划一次jvm性能调优](#)

如何理解内存泄漏问题？有哪些情况会导致内存泄露？如何解决？

[详解java内存泄露和如何避免内存泄漏](#)

开源框架

hibernate和ibatis的区别

[Ibatis与Hibernate的区别](#)

讲讲mybatis的连接池。

[《深入理解mybatis原理》 Mybatis数据源与连接池](#)

spring框架中需要引用哪些jar包，以及这些jar包的用途

springMVC的原理

[Spring MVC工作原理 及注解说明](#)

spring中beanFactory和ApplicationContext的联系和区别

[Spring系列之beanFactory与ApplicationContext](#)

spring注入的几种方式（循环注入）

[Spring循环依赖的三种方式](#)

spring如何实现事务管理的

springIOC

[Spring IOC原理解读 面试必读](#)

spring AOP的原理

[Spring AOP 实现原理与 CGLIB 应用](#)

spring AOP 两种代理方式

回答为什么要用什么方法这种问题的时候，通常首先要回答两个问题，第一个就是，我要做什么事情，第二个就是，不同方法的优劣是什么。

首先，我要做什么事情。

这里的回答比较简单，就是代理Java类/接口。那么，两者在完成这件事情上，有什么差别呢

JDK Proxy	Cglib Proxy
只能代理接口	以继承的方式完成代理，不

能代理被
final修饰
的类

实际上，大部分的Java类都会以接口-实现的方式来完成，因此，在这个方面上，JDK Proxy 实际上是比较Cglib Proxy要更胜一筹的。因为如果一个类被final修饰，则Cglib Proxy无法进行代理。

其次，两种方法的优劣又在什么地方呢？

我们可以参考一下来自bytebuddy的数据，这个是在代理一个实现了具有18个方法的接口的类，时间单位为ns。

	JDK Proxy	Cglib Proxy
--	-----------	-------------

生成代理类时间	1'060.766	960.527
---------	-----------	---------

方法调用时间	0.008	0.003
--------	-------	-------

来源	JDK原生代码	第三方库，更新频率较低
----	---------	-------------

不难看出，其实Cglib代理的性能是要远远好于JDK代理的。

其实从原理也能理解，直接通过类的方法调用，肯定要比通过反射调用的时间更短。但是从来源来看的话，一个是JDK原生代码，而另一个则是第三方的开源库。JDK原生代码无疑使用的人更多范围也更广，会更佳稳定，而且还有可能在未来的JDK版本中不断优化性能。

而Cglib更新频率相对来说比较低了，一方面是因为这个代码库已经渐趋稳定，另一方面也表明后续这个库可能相对来说不会有大动作的优化维护。

对比完之后，再来回看这个问题，为什么要使用两种方式呢？

在功能上讲，实际上Cglib代理并不如JDK代理（如果大家都按接口-实现的方式来设计类）。但是从效率上将，Cglib远胜JDK代理啊！所以，为了提高效率，同时又保有在未来，当JDK代理的性能也能够同样好的时候，使用更佳稳定靠谱的JDK代码，这种可能，于是采取了这种设计。

hibernate中的1级和2级缓存的使用方式以及区别原理（Lazy-Load的理解）

Hibernate的原理体系架构，五大核心接口，Hibernate对象的三种状态转换，事务管理。

Spring boot 热加载

[Spring boot 热加载](#)

Spring Boot设置有效时间和自动刷新缓存，时间支持在配置文件中配置

[Spring Boot缓存实战 Redis 设置有效时间和自动刷新缓存，时间支持在配置文件中配置](#)

Spring 如何保证 Controller 并发的安全？

[springMVC一个Controller处理所有用户请求的并发问题](#)

spring中用到哪些设计模式？

[spring中用到哪些设计模式](#)

Spring IOC 的理解，其初始化过程？

[Spring IoC容器初始化过程学习](#)

Spring的事务管理

[Spring事务管理（详解+实例）](#)

MyBatis缓存

[《深入理解mybatis原理》 MyBatis缓存机制的设计与实现](#)

MyBatis数据源与连接池

[《深入理解mybatis原理》 Mybatis数据源与连接池](#)

分布式

CAP原理和BASE理论

[CAP原则 \(CAP定理\)、BASE理论](#)

分布式事务、分布式锁

[常用的分布式事务解决方案介绍有多少种？](#)

[分布式锁的几种实现方式](#)

分布式存储系统

redis

redis和memcache的区别；

用redis做过什么；

redis是如何持久化的：rdb和aof；

Redis数据类型

[Redis五种数据类型介绍](#)

redis集群如何同步；

[Redis 复制与集群](#)

redis的数据添加过程是怎样的：哈希槽；

[redis集群实现（四） 数据的和槽位的分配](#)

redis的淘汰策略有哪些;

Redis 内存淘汰机制

- volatile-lru -> 根据LRU算法删除带有过期时间的key。
- allkeys-lru -> 根据LRU算法删除任何key。
- volatile-random -> 根据过期设置来随机删除key, 具备过期时间的key。
- allkeys-random -> 无差别随机删, 任何一个key。
- volatile-ttl -> 根据最近过期时间来删除 (辅以TTL), 这是对于有过期时间的key
- noeviction -> 谁也不删, 直接在写操作时返回错误。

redis有哪些数据结构;

redis的单线程模型

Redis 网络架构及单线程模型

redis 集群基础

- 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.
- 节点的fail是通过集群中超过半数的master节点检测失效时才生效.
- 客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可
- redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护 node<->slot<->key.
- 如果存入一个值, 按照redis cluster哈希槽的算法: $\text{CRC16}(\text{'key'}) \div 384 = 6782$ 。那么就会把这个key 的存储分配到对应的master上

redis Cluster主从模式

- 如果进群超过半数以上master挂掉, 无论是否有slave集群进入fail状态,所以集群中至少应该有奇数个节点, 所以至少要有三个节点, 每个节点至少有一个备份节点,
- redis cluster 为了保证数据的高可用性, 加入了主从模式, 一个主节点对应一个或多个从节点, 主节点提供数据存取, 从节点则是从主节点拉取数据备份, 当这个主节点挂掉后, 就会有这个从节点选取一个来充当主节点, 从而保证集群不会挂掉。

zookeeper

zookeeper是什么;

zookeeper哪里用到;

zookeeper的选主过程;

zookeeper集群之间如何通讯

[zookeeper系列之通信模型](#)

你们的zookeeper的节点加密是用的什么方式
分布式锁的实现过程；

kafka

传递保证语义：

- At most once：消息可能会丢，但绝不会重复传递。
- At least once：消息绝不会丢，但可能会重复传递。
- Exactly once：每条消息只会被传递一次。

生产者的“Exactly once”语义方案

当生产者向Kafka发送消息，且正常得到响应的时候，可以确保生产者不会产生重复的消息。但是，如果生产者发送消息后，遇到网络问题，无法获取响应，生产者就无法判断该消息是否成功提交给了Kafka。根据生产者的机制，我们知道，当出现异常时，会进行消息重传，这就可能出现“At least one”语义。为了实现“Exactly once”语义，这里提供两个可选方案：

- 每个分区只有一个生产者写入消息，当出现异常或超时的情况时，生产者就要查询此分区的最后一个消息，用来决定后续操作是消息重传还是继续发送。
- 为每个消息添加一个全局唯一主键，生产者不做其他特殊处理，按照之前分析方式进行重传，由消费者对消息进行去重，实现“Exactly once”语义。

如果业务数据产生消息可以找到合适的字段作为主键，或是有一个全局ID生成器，可以优先考虑选用第二种方案。

消费者的“Exactly once”语义方案

为了实现消费者的“Exactly once”语义，在这里提供一种方案，供读者参考：消费者将关闭自动提交offset的功能且不再手动提交offset，这样就不使用Offsets Topic这个内部Topic记录其offset，而是由消费者自己保存offset。这里利用事务的原子性来实现“Exactly once”语义，我们将offset和消息处理结果放在一个事务中，事务执行成功则认为此消息被消费，否则事务回滚需要重新消费。当出现消费者宕机重启或Rebalance操作时，消费者可以从关系型数据库中找到对应的offset，然后调用KafkaConsumer.seek()方法手动设置消费位置，从此offset处开始继续消费。

ISR集合

ISR（In-Sync Replica）集合表示的是目前“可用”（alive）且消息量与Leader相差不多的副本集合，这是整个副本集合的一个子集。“可用”和“相差不多”都是很模糊的描述，其

实际含义是ISR集合中的副本必须满足下面两个条件：

1. 副本所在节点必须维持着与ZooKeeper的连接。
2. 副本最后一条消息的offset与Leader副本的最后一条消息的offset之间的差值不能超出指定的阈值。

每个分区中的Leader副本都会维护此分区的ISR集合。写请求首先由Leader副本处理，之后Follower副本会从Leader上拉取写入的消息，这个过程会有一定的延迟，导致Follower副本中保存的消息略少于Leader副本，只要未超出阈值都是可以容忍的。如果一个Follower副本出现异常，比如：宕机，发生长时间GC而导致Kafka僵死或是网络断开连接导致长时间没有拉取消息进行同步，就会违反上面的两个条件，从而被Leader副本踢出ISR集合。当Follower副本从异常中恢复之后，会继续与Leader副本进行同步，当Follower副本“追上”（即最后一条消息的offset的差值小于指定阈值）Leader副本的时候，此Follower副本会被Leader副本重新加入到ISR中。

请说明什么是Apache Kafka？

Apache Kafka是由Apache开发的一种发布订阅消息系统，它是一个分布式的、分区的和重复的日志服务。

请说明什么是传统的消息传递方法？

传统的消息传递方法包括两种：

- 排队：在队列中，一组用户可以从服务器中读取消息，每条消息都发送给其中一个人。
- 发布-订阅：在这个模型中，消息被广播给所有的用户。

请说明Kafka相对传统技术有什么优势？

Apache Kafka与传统的消息传递技术相比优势之处在于：

- 快速:单一的Kafka代理可以处理成千上万的客户端，每秒处理数兆字节的读写操作。
- 可伸缩:在一组机器上对数据进行分区和简化，以支持更大的数据
- 持久:消息是持久性的，并在集群中进行复制，以防止数据丢失
- 设计:它提供了容错保证和持久性

在Kafka中broker的意义是什么？

在Kafka集群中，broker术语用于引用服务器。

Kafka服务器能接收到的最大信息是多少？

Kafka服务器可以接收到的消息的最大大小是1000000字节。

解释Kafka的Zookeeper是什么？我们可以在没有Zookeeper的情况下使用Kafka吗？

Zookeeper是一个开放源码的、高性能的协调服务，它用于Kafka的分布式应用。

不，不可能越过Zookeeper，直接联系Kafka broker。一旦Zookeeper停止工作，它就不能服务客户端请求。

Zookeeper主要用于在集群中不同节点之间进行通信

在Kafka中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从之前提交的偏移量中获取

除此之外，它还执行其他活动，如：leader检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

解释Kafka的用户如何消费信息？

在Kafka中传递消息是通过使用sendfile API完成的。它支持将字节从套接口转移到磁盘，通过内核空间保存副本，并在内核用户之间调用内核。

解释如何提高远程用户的吞吐量？

如果用户位于与broker不同的数据中心，则可能需要调优套接口缓冲区大小，以对长网络延迟进行摊销。

解释一下，在数据制作过程中，你如何能从Kafka得到准确的信息？

在数据中，为了精确地获得Kafka的消息，你必须遵循两件事：在数据消耗期间避免重复，在数据生产过程中避免重复。

这里有两种方法，可以在数据生成时准确地获得一个语义：

- 每个分区使用一个单独的写入器，每当你发现一个网络错误，检查该分区中的最后一条消息，以查看您的最后一次写入是否成功
- 在消息中包含一个主键(UUID或其他)，并在用户中进行复制

解释如何减少ISR中的扰动？broker什么时候离开ISR？

ISR是一组与leaders完全同步的消息副本，也就是说ISR中包含了所有提交的消息。ISR应该总是包含所有的副本，直到出现真正的故障。如果一个副本从leader中脱离出来，将会从ISR中删除。

Kafka为什么需要复制？

Kafka的信息复制确保了任何已发布的消息不会丢失，并且可以在机器错误、程序错误或更常见些的软件升级中使用。

如果副本在ISR中停留了很长时间表明什么？

如果一个副本在ISR中保留了很长一段时间，那么它就表明，跟踪器无法像在leader收集数据那样快速地获取数据。

请说明如果首选的副本不在ISR中会发生什么？

如果首选的副本不在ISR中，控制器将无法将leadership转移到首选的副本。

有可能在生产后发生消息偏移吗？

在大多数队列系统中，作为生产者的类无法做到这一点，它的作用是触发并忘记消息。

broker将完成剩下的工作，比如使用id进行适当的元数据处理、偏移量等。

作为消息的用户，你可以从Kafka broker中获得补偿。如果你注视SimpleConsumer类，你会注意到它会获取包括偏移量作为列表的MultiFetchResponse对象。此外，当你Kafka消息进行迭代时，你会拥有包括偏移量和消息发送的MessageAndOffset对象。

kafka与传统的消息中间件对比

[kafka与传统的消息中间件对比](#)

KAFKA：如何做到1秒发布百万级条消息

[KAFKA：如何做到1秒发布百万级条消息](#)

kafka文件存储

[Kafka文件存储机制那些事](#)

据kafka官网吹，如果随机写入磁盘，速度就只有100KB每秒。顺序写入的话，7200转/s的磁盘就能达到惊人的600MB每秒！

操作系统对文件访问做了优化，文件会在内核空间分页做缓存（pageCache）。写入时先写入pageCache。由操作系统来决定何时统一写入磁盘。操作系统会使用顺序写入。

[Kafka深入理解-1：Kafka高效的文件存储设计](#)

[kafka的log存储解析——topic的分区partition分段segment以及索引等](#)

dubbo

默认使用的是什么通信框架，还有别的选择吗？

默认也推荐使用netty框架，还有mina。

服务调用是阻塞的吗？

默认是阻塞的，可以异步调用，没有返回值的可以这么做。

一般使用什么注册中心？还有别的选择吗？

推荐使用zookeeper注册中心，还有redis等不推荐。

默认使用什么序列化框架，你知道的还有哪些？

默认使用Hessian序列化，还有Duddo、FastJson、Java自带序列化。

服务提供者能实现失效踢出是什么原理？

服务失效踢出基于zookeeper的临时节点原理。

服务上线怎么不影响旧版本？

采用多版本开发，不影响旧版本。

如何解决服务调用链过长的问题？

可以结合zipkin实现分布式服务追踪。

说说核心的配置有哪些？

核心配置有 dubbo:service/ dubbo:reference/ dubbo:protocol/ dubbo:registry/
dubbo:application/ dubbo:provider/ dubbo:consumer/ dubbo:method/

dubbo推荐用什么协议？

默认使用dubbo协议。

同一个服务多个注册的情况下可以直连某一个服务吗？

可以直连，修改配置即可，也可以通过telnet直接某个服务。

画一画服务注册与发现的流程图

流程图见dubbo.io。

Dubbo集群容错怎么做？

读操作建议使用Failover失败自动切换，默认重试两次其他服务器。写操作建议使用Failfast快速失败，发一次调用失败就立即报错。

在使用过程中都遇到了些什么问题？

使用过程中的问题可以百度

dubbo和dubbox之间的区别？

dubbox是当当网基于dubbo上做了一些扩展，如加了服务可restful调用，更新了开源组件等。

你还了解别的分布式框架吗？

别的还有spring的spring cloud，facebook的thrift，twitter的finagle等。

dubbo重试雪崩

[Dubbo超时机制导致的雪崩连接](#)

TCP/IP

算法

使用随机算法产生一个数，要求把1-1000W之间这些数全部生成。

（考察高效率，解决产生冲突的问题）

两个有序数组的合并排序

一个数组的倒序

计算一个正整数的正平方根

说白了就是常见的那些查找、排序算法以及各自的时间复杂度

二叉树的遍历算法

DFS, BFS算法

比较重要的数据结构，如链表，队列，栈的基本理解及大致实现。

排序算法与时空复杂度（快排为什么不稳定，为什么你的项目还在用）

逆波兰计算器

Hoffman 编码

查找树与红黑树

如何给100亿个数字排序？

[如何给100亿个数字排序？](#)

统计海量数据中出现次数最多的前10个IP

[哈希分治法 - 统计海量数据中出现次数最多的前10个IP](#)

排序算法时间复杂度

排序方法	时间复杂度 (平均)	时间复杂度 (最坏)	时间复杂度 (最好)	空间复杂度	稳定性	复杂性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n\log^2 n)$	$O(n^2)$	$O(n)$	$O(1)$	不稳定	较复杂
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n\log^2 n)$	$O(n\log^2 n)$	$O(n\log^2 n)$	$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n\log^2 n)$	$O(n^2)$	$O(n\log^2 n)$	$O(n\log^2 n)$	不稳定	较复杂
归并排序	$O(n\log^2 n)$	$O(n\log^2 n)$	$O(n\log^2 n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(n+r)$	稳定	较复杂

判断链表中是否有环

[判断链表中是否有环 ----- 有关单链表中环的问题](#)

hash算法及常用的hash算法

[常见hash算法的原理](#)

查找算法

[七大查找算法](#)

设计与思想

重构过代码没有？说说经验；

一千万的用户实时排名如何实现；

[海量积分数据实时排名处理](#)

[如何高效地获得玩家排名？](#)

[基于redis的zset实现排行榜功能](#)

五万人并发抢票怎么实现；

[Web大规模高并发请求和抢购的解决方案](#)

有个每秒钟5k个请求，查询手机号所属地的笔试题(记得不完整，没列出)，如何设计算法？请求再多，比如5w，如何设计整个系统？

高并发情况下，我们系统是如何支撑大量的请求的

集群如何同步会话状态

[集群服务器Session同步](#)

负载均衡的原理

如果有一个特别大的访问量，到数据库上，怎么做优化（DB设计，DBIO，SQL优化，Java优化）

[大数据量高并发的数据库优化](#)

如果出现大面积并发，在不增加服务器的基础上，如何解决服务器响应不及时问题“。

[如何提高服务器并发处理能力](#)

假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。

[三个方面解决性能问题的基本思路和方法](#)

[一个Web应用的性能瓶颈一般有哪些呢？通常采取什么手段解决呢？](#)

如何查找 造成 性能瓶颈出现的位置，是哪个位置造成性能瓶颈。

[五步定位性能瓶颈](#)

[性能测试如何定位瓶颈](#)

你的项目中使用过缓存机制吗？有没有用户非本地缓存

[使用缓存的9大误区（上）](#)

[使用缓存的9大误区（下）](#)

[网站缓存技术](#)

Tomcat优化

[Tomcat 调优及 JVM 参数优化](#)

网络通信

http是无状态通信，http的请求方式有哪些，可以自己定义新的请求方式么。

socket通信，以及长连接，分包，连接异常断开的处理。

[Socket TCP/IP协议数据传输过程中的粘包和分包问题](#)

[深入理解socket网络异常](#)

socket通信模型的使用，AIO和NIO。

socket框架netty的使用，以及NIO的实现原理，为什么是异步非阻塞。

同步和异步，阻塞和非阻塞。

OSI七层模型，包括TCP, IP的一些基本知识

[OSI七层协议模型和TCP/IP四层模型比较](#)

http中，get post的区别

- get: 从服务器上获取数据，也就是所谓的查，仅仅是获取服务器资源，不进行修改。
- post: 向服务器提交数据，这就涉及到了数据的更新，也就是更改服务器的数据。
- 请求方式的区别: get 请求的数据会附加在URL之后,特定的浏览器和服务器对URL的长度有限制. post 更加安全数据不会暴露在url上,而且长度没有限制.

HTTP报文内容

[HTTP请求报文和HTTP响应报文](#)

说说http, tcp, udp之间关系和区别。

说说浏览器访问<http://www.taobao.com>，经历了怎样的过程。

[访问 www.taobao.com过程](#)

HTTP协议、 HTTPS协议，SSL协议及完整交互过程；

[HTTPS协议，SSL协议及完整交互过程](#)

tcp的拥塞，快回传，ip的报文丢弃

https处理的一个过程，对称加密和非对称加密

head各个特点和区别

ping的原理

[Ping过程 原理 详解\(图\)](#)

ARP/RARP

[TCP/IP协议详解笔记——ARP协议和RARP协议](#)

DNS解析过程

[DNS域名解析的过程](#)

Http会话的四个过程

建立连接，发送请求，返回响应，关闭连接。

数据库MySQL

MySQL的存储引擎的不同

[MySQL存储引擎之Myisam和Innodb总结性梳理](#)

MySQL参数

[Mysql优化系列（1）--Innodb引擎下mysql自身配置优化](#)

单个索引、联合索引、主键索引

[Mysql主键索引、唯一索引、普通索引、全文索引、组合索引的区别](#)

Mysql怎么分表，以及分表后如果想按条件分页查询怎么办(如果不是按分表字段来查询的话，几乎效率低下，无解)

如果按时间排序查询，使用limit n （不要使用limit m, n 页数多了之后效率低）然后记录最后一条的时间，下次从最后一条的时间开始查询

[mysql 数据库 分表后 怎么进行分页查询？Mysql分库分表方案？](#)

[mysql大数据量使用limit分页，随着页码的增大，查询效率越低下。](#)

分表之后想让一个id多个表是自增的，效率实现

[MySQL分表自增ID解决方案](#)

分布式id生成算法

[理解分布式id生成算法SnowFlake](#)

MySQL的主从实时备份同步的配置，以及原理(从库读主库的binlog)，读写分离

[MySQL主从同步的实现原理](#)

MySQL索引

[浅谈算法和数据结构：十 平衡查找树之B树](#)

[MySQL索引背后的数据结构及算法原理](#)

[MySQL索引失效的几种情况](#)

事务的四个特性，以及各自的特点（原子、隔离）等等，项目怎么解决这些问题

数据库的锁：行锁，表锁；乐观锁，悲观锁

[mysql的锁—行锁，表锁，乐观锁，悲观锁](#)

[mysql for update语法实现行锁](#)

[MySQL 加锁处理分析](#)

数据库事务的几种粒度；

[理解事务 - MySQL 事务处理机制](#)

MVCC

[Mysql中的MVCC](#)

聚簇索引

[mysql索引原理之聚簇索引](#)

关系型和非关系型数据库区别

- 关系型数据库：是指采用了关系模型(二维表格模型)来组织数据的数据库。
- 非关系型数据库：以键值对存储，且结构不固定。

MySQL死锁排查

[我的Mysql死锁排查过程（案例分析）](#)

MySQL优化

[MySQL 对于千万级的大表要怎么优化？](#)

Linux

介绍一下epoll

killl的用法，某个进程杀不掉的原因（进入内核态，忽略kill信号）

硬链接和软链接的区别

grep的使用

进程间的通信，共享内存方式的优缺点

swap分区

[Swap交换分区概念](#)

[Linux系统swappiness参数在内存与交换分区之间优化作用](#)

overcommit_memory

取值为0，系统在为应用进程分配虚拟地址空间时，会判断当前申请的虚拟地址空间大小是否超过剩余内存大小，如果超过，则虚拟地址空间分配失败。因此，也就是如果进程本身占用的虚拟地址空间比较大或者剩余内存比较小时，fork、malloc等调用可能会失败。

取值为1，系统在为应用进程分配虚拟地址空间时，完全不进行限制，这种情况下，避免了fork可能产生的失败，但由于malloc是先分配虚拟地址空间，而后通过异常陷入内核分配真正的物理内存，在内存不足的情况下，这相当于完全屏蔽了应用进程对系统内存状态的感知，即malloc总是能成功，一旦内存不足，会引起系统OOM杀进程，应用程序对于这种后果是无法预测的

取值为2，则是根据系统内存状态确定了虚拟地址空间的上限，由于很多情况下，进程的虚拟地址空间占用远大于其实际占用的物理内存，这样一旦内存使用量上去以后，对于一些动态产生的进程(需要复制父进程地址空间)则很容易创建失败，如果业务过程没有过多的这种动态申请内存或者创建子进程，则影响不大，否则会产生比较大的影响

[又一次内存分配失败\(关于overcommit_memory\)](#)

linux系统下查看CPU、内存负载情况

[linux系统下查看CPU、内存负载情况](#)