# 面试题总结

目录

[面试题总结 1](#_Toc14254107)

[1.1. 基础 5](#_Toc14254108)

[1.1.1. 设计模式 5](#_Toc14254109)

[1.1.2. 1. JDK8新特性 5](#_Toc14254110)

[1.2. 并发 6](#_Toc14254111)

[1.2.1. Volatile 6](#_Toc14254112)

[1.2.2. 原子类保证线程安全实现原理? CAS算法？ 7](#_Toc14254113)

[1.2.3. 线程的状态及之间的转换？ 7](#_Toc14254114)

[1.2.4. 使用ExecutorService、Callable、Future实现有返回结果的线程 8](#_Toc14254115)

[1.2.5. 线程池有哪些关键属性 8](#_Toc14254116)

[1.2.6. Java中CyclicBarrier 和 CountDownLatch有什么不同、运用场景 9](#_Toc14254117)

[1.2.7. 什么是ThreadLocal变量？实现原理？使用场景？ 9](#_Toc14254118)

[1.2.8. Java中wait和sleep方法的区别 ？ 9](#_Toc14254119)

[1.2.9. Synchronized和lock区别？ 10](#_Toc14254120)

[1.2.10. Synchronized可以修饰哪些地方？ 10](#_Toc14254121)

[1.2.11. 锁的状态？ 10](#_Toc14254122)

[1.2.12. Notify和wait的使用 10](#_Toc14254123)

[1.2.13. AQS源码解析 11](#_Toc14254124)

[1.2.14. 线程池原理 11](#_Toc14254125)

[1.3. 基本数据结构 11](#_Toc14254126)

[1.3.1. Hashmap工作原理? 11](#_Toc14254127)

[1.3.2. 你知道 hash 的实现吗？为什么要这样实现？ 11](#_Toc14254128)

[1.3.3. HashMap 的遍历方式，哪种遍历方式效率高？ 12](#_Toc14254129)

[1.3.4. Hashmap什么时候扩容，扩容算法？ 12](#_Toc14254130)

[1.3.5. ConcurrentHashMap实现原理？ 12](#_Toc14254131)

[1.3.6. Linkedblockqueue实现原理？ 12](#_Toc14254132)

[1.3.7. Treemap、TreeSet 12](#_Toc14254133)

[1.3.8. 红黑树？ 13](#_Toc14254134)

[1.4. WEB 13](#_Toc14254135)

[1.4.1. tomcat参数配置和 线程模型、调优？ 13](#_Toc14254136)

[1.4.2. 如何解决跨域问题？ 14](#_Toc14254137)

[1.4.3. 单点登录（SSO） 14](#_Toc14254138)

[1.4.4. 分布式会话 14](#_Toc14254139)

[1.4.5. Ngin负载均衡 15](#_Toc14254140)

[1.4.6. HTTP请求过程 15](#_Toc14254141)

[1.4.7. HTTP与HTTPS有什么区别？ 16](#_Toc14254142)

[1.4.8. https是怎么保证数据的加密传输的呢？ 16](#_Toc14254143)

[1.4.9. Websocket 17](#_Toc14254144)

[1.4.10. RESTful API 17](#_Toc14254145)

[1.5. Linux 17](#_Toc14254146)

[1.5.1. 说出linux常用命令 17](#_Toc14254147)

[1.5.2. 如何查看所有java进程 18](#_Toc14254148)

[1.5.3. LINUX中如何查看某个端口是否被占用 18](#_Toc14254149)

[1.5.4. find查找文件 18](#_Toc14254150)

[1.5.5. Linux vi中查找字符内容的方法 18](#_Toc14254151)

[1.5.6. Top命令 18](#_Toc14254152)

[1.6. Git 19](#_Toc14254153)

[1.7. Spring 框架相关 19](#_Toc14254154)

[1.7.1. Bean的生命周期 19](#_Toc14254155)

[1.7.2. Spring aop原理 20](#_Toc14254156)

[1.7.3. Spring IOC原理 20](#_Toc14254157)

[1.7.4. spring与springboot的关系 20](#_Toc14254158)

[1.7.5. Spring Boot有哪些优点？ 21](#_Toc14254159)

[1.7.6. springboot自动配置原理 21](#_Toc14254160)

[1.7.7. 约定优于配置 21](#_Toc14254161)

[1.7.8. Springmvc中controller是线程安全的吗？ 21](#_Toc14254162)

[1.7.9. Springmvc中在controller注入request（HttpServletRequest）会有线程安全问题吗？ 21](#_Toc14254163)

[1.7.10. Spring事务传播行为 22](#_Toc14254164)

[1.7.11. mybatis一级缓存二级缓存 22](#_Toc14254165)

[1.7.12. Springmvc执行流程 22](#_Toc14254166)

[8.1. JVM 相关 23](#_Toc14254167)

[8.1.1. 类的双亲委派机制及其作用 23](#_Toc14254168)

[8.1.2. Jvm内存模型 23](#_Toc14254169)

[8.1.3. 判断对象是否可回收算法、GC算法 24](#_Toc14254170)

[8.1.4. 内存溢出解决方案 24](#_Toc14254171)

[8.1.5. 查找CPU占用率高的线程 25](#_Toc14254172)

[8.1.6. Jvm常用调参 27](#_Toc14254173)

[8.2. mysql数据库 27](#_Toc14254174)

[8.2.1. mysql有哪几种索引类型？索引结构（BTREE）？ 27](#_Toc14254175)

[8.2.2. 索引结构（BTREE）？ 28](#_Toc14254176)

[8.2.3. Mysql索引优化流程？ 28](#_Toc14254177)

[8.2.4. Union和union all区别？ 28](#_Toc14254178)

[8.2.5. mysql悲观锁与乐观锁？悲观锁乐观锁选择？ 28](#_Toc14254179)

[8.2.6. Mysql如何查看sql是否使用索引？ 28](#_Toc14254180)

[8.2.7. 数据库引擎：innodb和myisam、memory区别 ？场景下选择？ 29](#_Toc14254181)

[8.2.8. Mysql部署？ 29](#_Toc14254182)

[8.2.9. 数据库分库分表？如何扩容？ 30](#_Toc14254183)

[8.2.10. 关系型数据库与nosql数据库的优缺点？ 30](#_Toc14254184)

[8.2.11. Mysql主从同步？读写分离？主从一数据致性问题？ 31](#_Toc14254185)

[8.2.12. 分布式系统唯一ID生成方法？分库分表后主键id是如何保证唯一的？订单号规则？ 31](#_Toc14254186)

[8.3. Redis 31](#_Toc14254187)

[8.3.1. Redis基本数据结构？项目当中的应用场景？ 31](#_Toc14254188)

[8.3.2. Redis内存占用满了怎么办？Redis有哪几种数据淘汰策略？ 32](#_Toc14254189)

[8.3.3. Redis到底是多线程还是单线程，为什么？线程安全吗？ 32](#_Toc14254190)

[8.3.4. Redis单线程下为什么能执行这么快？ 32](#_Toc14254191)

[8.3.5. Redis中的管道？Redis事务？ 33](#_Toc14254192)

[8.3.6. Redis分布式锁实现 33](#_Toc14254193)

[8.3.7. Redis持久化方式？及其优缺点？ 33](#_Toc14254194)

[8.3.8. Redis架构模式 34](#_Toc14254195)

[8.3.9. 你们公司是如何Redis部署？ 34](#_Toc14254196)

[8.3.10. 缓存击穿、缓存雪崩、缓存穿透？ 34](#_Toc14254197)

[8.3.11. Redis缓存一致性问题？ 35](#_Toc14254198)

[8.3.12. 什么是一致性哈希算法？什么是哈希槽？ 35](#_Toc14254199)

[8.4. 消息中间件 35](#_Toc14254200)

[8.4.1. 消息中间件应用场景 35](#_Toc14254201)

[8.4.2. JMS和AMQP 35](#_Toc14254202)

[8.4.3. Rabbbitmq几种工作模式 36](#_Toc14254203)

[8.4.4. 如何保证RabbitMQ的高可用性？ 36](#_Toc14254204)

[8.4.5. 如何保证RabbitMQ不丢失消息？在使用过程中有没有遇到消息丢失的情况？ 36](#_Toc14254205)

[8.4.6. RabbitMQ如何保证消息顺序消费？ 37](#_Toc14254206)

[8.4.7. RabbitMQ如何避免消息重复的被消费？ 37](#_Toc14254207)

[8.4.8. RabbitMQ是如何部署的？ 38](#_Toc14254208)

[8.5. 搜索引擎专题 38](#_Toc14254209)

[8.5.1. Solr部署？ 38](#_Toc14254210)

[8.5.2. Solr query 和filter query 39](#_Toc14254211)

[8.5.3. 倒排索引 39](#_Toc14254212)

[8.5.4. TF-IDF算法/相似度算法/评分算法 39](#_Toc14254213)

[8.6. springcloud微服务 39](#_Toc14254214)

[8.6.1. 不同微服务交互注意事项 39](#_Toc14254215)

[8.6.2. 微服务架构与传统的单体应用架构有什么优缺点？ 40](#_Toc14254216)

[8.6.3. 服务降级、熔断、限流、资源隔离？ 40](#_Toc14254217)

[8.6.4. Spring Cloud Eureka（注册中心） 41](#_Toc14254218)

[8.6.5. Spring Cloud Ribbon（负载均衡） 41](#_Toc14254219)

[8.6.6. Spring Cloud Feign（负载均衡） 42](#_Toc14254220)

[8.6.7. Spring Cloud Hystrix（熔断器） 43](#_Toc14254221)

[8.6.8. Hystrix隔离策略及其选择 43](#_Toc14254222)

[8.6.9. Spring Cloud Zuul(网关) 44](#_Toc14254223)

[Dubbo专题 45](#_Toc14254224)

[3.1.1. dubbo 负载均衡策略？ 45](#_Toc14254225)

[3.1.2. dubbo 集群容错策略 45](#_Toc14254226)

[3.1.3. dubbo动态代理策略 46](#_Toc14254227)

[3.1.4. 注册中心挂了Dubbo服务可以继续通信吗？ 46](#_Toc14254228)

[3.1.5. dubbo 支持不同的通信协议？ 46](#_Toc14254229)

[3.1.6. dubbo 支持的序列化协议 46](#_Toc14254230)

[3.1.7. Dubbo默认使用的是什么通信框架？ 47](#_Toc14254231)

[3.1.8. dubbo 和 dubbox 之间的区别？ 47](#_Toc14254232)

[3.1.9. dubbo注册了多个一样的服务实例，如何调用指定的某一个服务呢？ 47](#_Toc14254233)

[3.1.10. Dubbo需要Web容器(tomcat/jetty)吗？ 48](#_Toc14254234)

[3.1.11. Dubbo服务运行过程 48](#_Toc14254235)

[3.1.12. Dubbo RPC实现原理 48](#_Toc14254236)

[3.1.13. Zookeeper服务列表获取更新 49](#_Toc14254237)

[3.1.14. 分布式一致性协议 Raft 49](#_Toc14254238)

[3.2. 工具专题 49](#_Toc14254239)

[3.2.1. Maven 冲突解决方法 49](#_Toc14254240)

[3.2.2. 压力测试Jmeter 49](#_Toc14254241)

[3.3. 实践专题 50](#_Toc14254242)

[3.3.1. 服务器配置 50](#_Toc14254243)

[3.3.2. 服务部署需要多少台服务器及其配置？ 50](#_Toc14254244)

[3.3.3. Solr部署？ 51](#_Toc14254245)

[3.3.4. RabbitMQ是如何部署的？ 51](#_Toc14254246)

[3.3.5. Mysql部署 51](#_Toc14254247)

[8.6.10. Redis部署 52](#_Toc14254248)

[8.6.11. 缓存击穿、缓存雪崩、缓存穿透？ 52](#_Toc14254249)

[1.1. 基本类型包装类 54](#_Toc14254250)

## 基础

### 设计模式

重点掌握：工厂、单例、装饰器、模板模式、代理模式

参考文档：<https://www.runoob.com/design-pattern/design-pattern-tutorial.html>

### 1. JDK8新特性

简答：主要特性

* Lambda表达式和函数式接口
* 接口的默认方法和静态方法
* 方法引用
* 重复注解
* Optional
* StreamsAPI

参考博客：<https://www.cnblogs.com/xingzc/p/6002873.html>

## 并发

### Volatile

1. volatile 能使得一个非原子操作变成原子操作吗？

简答：不能保证原子性。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的，但 volatile 类型的 double 和 long 就是原子的。

详解参考博文：<https://blog.csdn.net/xdzhouxin/article/details/81236356>

1. volatile运用场景或者有过什么实践？

简答：修饰状态标志变量，如控制线程是否运行的boolean变量标志

1. volatile 类型变量提供什么保证或者有什么用？

简答：volatile 变量提供程序顺序执行和变量可见性保证。可见性：可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。程序顺序执行：即程序执行的顺序按照代码的先后顺序执行。

1. 为什么会造成变量不可见性和程序执行乱序？

简答：不可见性：Cpu计算速度与内存读写速度差距较大，为了提高性能，中间会引入高速缓存。多个线程在计算时拥有自己独立的高速缓存空间，但多线程同时操作内存当中的一个变量时，会各自复制一份至高速缓存中。如果此时有一个线程修改了该变量值，由于缓存原因，其它线程无法及时获取该变量的最新值。

程序执行乱序：处理器为了提高程序运行效率，可能会对输入代码进行优化，进行指令重排，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

参考博文：<https://www.cnblogs.com/dolphin0520/p/3920373.html>

Volatile 是如何实现变量可见性和程序顺序执行或者volatile实现原理？

简答：底层通过LOCK指令锁定缓存行，使其它线程的缓存行无效，等待锁的释放，重新读取最新值。

详细原理：参考<https://www.cnblogs.com/xrq730/p/7048693.html>

### 原子类保证线程安全实现原理? CAS算法？

简答：原子类相关方法操作通过调用Unsafe类相关的native方法保证线程安全，它是基于CPU CAS指令完成。CAS算法是由硬件直接支持来保证原子性的，有三个操作数：内存位置V、旧的预期值A和新值B，当且仅当V符合预期值A时，CAS用新值B原子化地更新V的值，否则，它什么都不做。

CAS，在Java并发应用中通常指CompareAndSwap或CompareAndSet，即比较并交换

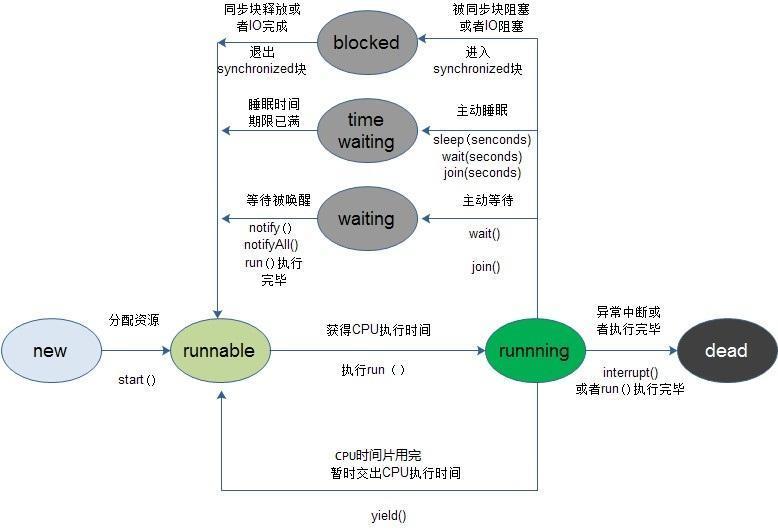
1、CAS是一个原子操作，它比较一个内存位置的值并且只有相等时修改这个内存位置的值为新的值，保证了新的值总是基于最新的信息计算的，如果有其他线程在这期间修改了这个值则CAS失败。CAS返回是否成功或者内存位置原来的值用于判断是否CAS成功。

2、JVM中的CAS操作是利用了处理器提供的CMPXCHG指令实现的。

参考博文：<https://www.cnblogs.com/chengxiao/p/6789109.html>

### 线程的状态及之间的转换？

* 新建状态（New）   
  用new语句创建的线程处于新建状态，此时它和其他Java对象一样，仅仅在堆区中被分配了内存。
* 就绪状态（Runnable）   
  当一个线程对象创建后，其他线程调用它的start()方法，该线程就进入就绪状态，Java虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得CPU的使用权。
* 运行状态（Running）   
  处于这个状态的线程占用CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。
* 阻塞状态（Blocked）   
  阻塞状态是指线程因为某些原因放弃CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。
* 死亡状态（Dead）   
  当线程退出run()方法时，就进入死亡状态，该线程结束生命周期。



竞态条件（race condition）是指设备或系统出现不恰当的执行时序，而得到不正确的结果。同步加锁

### 使用ExecutorService、Callable、Future实现有返回结果的线程

要求：会多线程编程，具体查项目

参考博文：<https://www.cnblogs.com/fengsehng/p/6048609.html>

<https://www.jianshu.com/p/78905e0e8190>

### 线程池有哪些关键属性

corePoolSize：最小工作线程

maximumPoolSize：最大工作线程

keepAliveTime：线程保活时间

workQueue：任务队列

### Java中CyclicBarrier 和 CountDownLatch有什么不同、运用场景

答：CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

CyclicBarrier 参考博文<https://www.jianshu.com/p/4ef4bbf01811>

### 什么是ThreadLocal变量？实现原理？使用场景？

答：ThreadLocal一般称为线程本地变量，它将变量与线程绑定在一起，为每一个线程维护一个独立的变量副本。实现原理可以参考博文分析。

运用场景：aop日志记录时通过ThreadLocal变量获取用户HttpServletRequest对象。

|  |
| --- |
| HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder  .getRequestAttributes()).getRequest(); |

RequestContextHolder对象中封装了存放HttpServletRequest t 的hreadlocal变量。

要求：有能力可以看看实现原理

实现原理参考博文：<https://www.cnblogs.com/dolphin0520/p/3920407.html>

### Java中wait和sleep方法的区别 ？

简答：

* sleep方法没有释放锁，而wait方法释放了锁，
* wait方法必须正在同步环境下使用，比如synchronized方法或者同步代码块。如果你不在同步条件下使用，会抛出IllegalMonitorStateException异常。另外，sleep方法不需要再同步条件下调用，你可以任意正常的使用。
* sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常
* 对象调用了wait方法，必须要采用notify()和notifyAll()方法唤醒该线程才能继续执行。调用Sleep方法后线程只有被异常中断或睡眠到期才能继续执行

参考博文：<https://www.cnblogs.com/loren-Yang/p/7538482.html>

### Synchronized和lock区别？

**简答：能够讲出前面几种**

|  |  |  |
| --- | --- | --- |
| **类别** | **synchronized** | **Lock** |
| 存在层次 | Java的关键字，在jvm层面上 | 是一个类 |
| 锁的释放 | 1、以获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm会让线程释放锁 | 在finally中必须释放锁，不然容易造成线程死锁 |
| 锁的获取 | 线程获取锁时需要等到其它线程释放锁并得到锁后程序才能继续执行 | 可以调用tryLock获取锁，并可指定等待获取时间。超时后可继续执行。 |
| 是否可中断 | Synchronized是非中断锁，必须等待线程执行完成释放锁 | Lock是可以中断锁 |
| 读锁实现 | 无法实现 | Lock可以使用读锁提高多线程读效率。 |

### Synchronized可以修饰哪些地方？

简答：

* + 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
  + 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
  + 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

### 锁的状态？

参考博文锁的状态部分：<https://blog.csdn.net/javazejian/article/details/72828483>

### Notify和wait的使用

要求:会使用它们编程及注意事项

参考博文：<https://www.cnblogs.com/hapjin/p/5492645.html>

### AQS源码解析

要求：有能力可以看看。AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它。

参考博文<https://www.cnblogs.com/waterystone/p/4920797.html>

### 线程池原理

要求：有能力可以看源码实现

参考博文：<https://blog.csdn.net/u013332124/article/details/79587436>

## 基本数据结构

### Hashmap工作原理?

简答：**存储对象时**，将 K/V 键值传给 put() 方法：①、调用 hash(K) 方法计算 K 的 hash 值，然后结合数组长度，计算得数组下标；②、调整数组大小（当容器中的元素个数大于 capacity \* loadfactor 时，容器会进行扩容resize 为 2n）；

③、i.如果 K 的 hash 值在 HashMap 中不存在，则执行插入，若存在，则发生碰撞；

ii.如果 K 的 hash 值在 HashMap 中存在，且它们两者 equals 返回 true，则更新键值对；

iii. 如果 K 的 hash 值在 HashMap 中存在，且它们两者 equals 返回 false，则插入链表的尾部或者红黑树中。

获取对象时，将 K 传给 get() 方法：①、调用 hash(K) 方法（计算 K 的 hash 值）从而获取该键值所在链表的数组下标；②、顺序遍历链表，equals()方法查找相同 Node 链表中 K 值对应的 V 值。

注意：hashCode 是定位的，**存储位置**；equals是定性的，**比较两者是否相等**

注意: HashMap可以接受null键值和值，而HashTable和ConcurrentHashmap则不能

### 你知道 hash 的实现吗？为什么要这样实现？

简答：JDK 1.8 中，是通过 hashCode() 的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度，功效和质量来考虑的，减少系统的开销，也不会造成因为高位没有参与下标的计算，从而引起的碰撞。

### HashMap 的遍历方式，哪种遍历方式效率高？

参考博文：<http://www.importnew.com/26298.html>

### Hashmap什么时候扩容，扩容算法？

简答：HashMap在元素达到负载因子对应数的时候就发生扩容。其实还有一种情况也可能会发生扩容：树形化的时候。在JDK1.8源码中，执行树形化之前，会先检查数组长度，如果长度小于64，则对数组进行扩容，而不是进行树形化。Hashmap扩容是2的整数次幂进行容量扩展。

注意：Hahsmap默认容量是16，且容量是2的n次方，不会出现非2的n次幂容量，如17、18。

### ConcurrentHashMap实现原理？

简答：至少说出分段锁优化

参考博文：<http://www.javastack.cn/article/2018/hashmap-concurrenthashmap-details/>

注意：jdk1.8之后已经是用CAS+synchronized完成的同步了，而不是segement分段锁

### Linkedblockqueue实现原理？

要求：有能力看源码实现，内部使用了Lock、Conditioin实现它阻塞等特性。

### Treemap、TreeSet

简答：TreeMap采用一种被称为“红黑树”的排序二叉树来保存 Map 中每个 Entry —— 每个 Entry 都被当成“红黑树”的一个节点对待。TreeSet 底层实际使用的存储容器就是 TreeMap，TreeSet 里绝大部分方法都是直接调用 TreeMap 的方法来实现的。

### 红黑树？

简答：红黑树是一种自平衡排序二叉树，树中每个节点的值，都大于或等于在它的左子树中的所有节点的值，并且小于或等于在它的右子树中的所有节点的值，这确保红黑树运行时可以快速地在树中查找和定位的所需节点（因为有序的）。

1. **动态代理实现原理及使用**

要求：之前讲过jdk动态代理相关知识及实现原理，已录制，可以观看视频。

参考博客：jdk <http://www.cnblogs.com/zuidongfeng/p/8735241.html>

Cglib：<https://www.cnblogs.com/monkey0307/p/8328821.html>

## WEB

### tomcat参数配置和 线程模型、调优？

简答：可配参数可以参考博文<https://blog.csdn.net/ljj_9/article/details/79145324>，

Tomcat四种线程模型

|  |  |
| --- | --- |
|  | 描述 |
| BIO | 阻塞式IO，采用传统的java IO进行操作，该模式下每个请求都会创建一个线程，适用于并发量小的场景 |
| NIO | 同步非阻塞，比传统BIO能更好的支持大并发，tomcat 8.0 后默认采用该模式 |
| APR | tomcat 以JNI形式调用http服务器的核心动态链接库来处理文件读取或网络传输操作，需要编译安装APR库 |
| AIO | 异步非阻塞，tomcat8.0后支持 |

调优参考博文：<https://blog.csdn.net/wojiao228925661/article/details/81459345>

注意：调优这里内容较多，记住几个，至少记住线程模型修改，线程池大小修改、jvm内存调整参数、超时时间，其它不做要求。

### 如何解决跨域问题？

简答：解决跨域问题有服务器端方法和前端方法，这里你必须要答出服务器端方法。

服务器端方法：设置CORS（跨资源共享），即让服务器端允许浏览器跨域访问。（必须掌握）

前端方法：JSONP请求， <script>是可以跨域的，而且在跨域脚本中可以直接回调当前脚本的函数。（了解）

服务器端java设置（代码不要死记）

|  |
| --- |
| @Configuration  public class CorsConfig extends WebMvcConfigurerAdapter {  @Override  public void addCorsMappings(CorsRegistry registry) {  registry.addMapping("/\*\*")  .allowedOrigins("\*")  .allowedMethods("GET", "HEAD", "POST", "PUT", "PATCH", "DELETE", "OPTIONS", "TRACE");  }  } |

什么是跨域问题：浏览器同源策略的限制，只允许访问同源的资源。同源，即协议相同，域名相同，端口相同。协议、端口、和域名有任意一个不同就会造成跨域。跨域问题只出现在浏览器。

参考博文：<http://www.cnblogs.com/morethink/p/6525216.html>

### 单点登录（SSO）

简答：说出单点登录原理，参考博文：<https://yq.aliyun.com/articles/636281>

什么是单点登录?:在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

### 分布式会话

简答：使用Spring Session解决session共享问题。Spring session

为何需要session共享？：默认的会话是通过Servlet容器创建和管理的，像Tomcat/Jetty都是保存在内存中的。如果我们将Web应用横向扩展搭建成分布式的集群，然后利用Nginx做负载均衡，那么来自同一用户的Http请求将有可能被负载分发到两个不同的实例中去， 当用户在其中一台做登录后，生成的session只会在本地，下次请求被nginx转发至另外一台时，由于session没有共享导致需要用户重新做登录。

解决方案：只要session实现共享，即各个服务器都能访问的到其它服务器生成的session即可。比如登录验证通过后，程序自己生成session放置redis中,所有服务器在查询sessionid时从redis获取即可。可使用现成框架spring session解决。

### Ngin负载均衡

要求：掌握配置

参考博文 <https://www.cnblogs.com/qlqwjy/p/8536779.html>

### HTTP请求过程

简答：

1、域名解析，得到对应的IP地址。

2、发起TCP的3次握手

3、建立TCP连接后发起http请求

4、服务器获取请求参数，进行业务逻辑处理，响应http请求，浏览器得到html代码

5、浏览器解析html代码，并请求html代码中的资源（如js、css、图片等）

6、浏览器对页面进行渲染呈现给用户。

注：

1.DNS域名解析采用的是递归查询的方式，过程是，先去找DNS缓存->缓存找不到就去找根域名服务器->根域名又会去找下一级，这样递归查找之后，找到了，给我们的web浏览器

2.为什么HTTP协议要基于TCP来实现？ TCP是一个端到端的可靠的面相连接的协议，HTTP基于传输层TCP协议不用担心数据传输的各种问题（当发生错误时，会重传）

3.最后一步浏览器是如何对页面进行渲染的？ a）解析html文件构成 DOM树，b）解析CSS文件构成渲染树， c）边解析，边渲染 ， d）JS 单线程运行，JS有可能修改DOM结构，意味着JS执行完成前，后续所有资源的下载是没有必要的，所以JS是单线程，会阻塞后续资源下载

### HTTP与HTTPS有什么区别？

简答：https是http协议的加密版。

说明：HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。

### https是怎么保证数据的加密传输的呢？

首先再正式发送数据前，浏览器如果发现是https传输要进行必要的准备工作才开始发送，大致分为以下几步。



1， 浏览器把协议版本号，支持的加密算法 生成的随机数A发送给服务器。

2， 服务器接到客户端发送过来的加密算法后，查看自身支持的加密算法，确定双方后边使用哪种加密算法，然后把该算法和数字证书以及生成的随机数B发给客户端。

3， 客户端接到服务器发送过来的证数后，得到证书里的非对称加密的公钥D，然后又生成一个随机数C,把C和D加密后得到的一个字符串E发给服务器。

4， 服务器接到这个字符串E后，用自身的私钥解密得到C，这样服务器就集齐了 A,B,C，然后利用ABC生成对话加密密钥 , 因为客户端也得到了A,B,C 这样双方都能进行加密解密了，至于加密解密的算法就是前面双方交换加密方法表时约定的加密算法。

所以可以看出ABC中，只有C是极难被别人得到的，这样保证了安全性，同时这个过程前面用到了非对称加密，一旦加密密钥确定，后面其实用的都是对称加密了。

核心思想：为了使传输的数据安全，我们需要对其加密。加密算法分为对称加密和非对称加密。对称加密需要双方拥有一样的密钥，如果采用对称加密，必然涉及到密钥交换，密钥在传输的过程中有可能被截取泄露，所有单纯采用对称加密是不安全的。而非对称加密密钥分私钥和公钥，只有私钥才能解密公钥加密的数据，只要密钥生成这保留私钥不公开，把公钥发送给对方，对方通过公钥加密，就能保证加密的安全性。即使公钥被人截取，没有私钥是无法解密的。

参考博文：<http://www.ruanyifeng.com/blog/2014/02/ssl_tls.html>

### Websocket

参考博文：<https://www.cnblogs.com/xdp-gacl/p/5193279.html>

### RESTful API

理解RESTful架构:<http://www.ruanyifeng.com/blog/2011/09/restful.html>

RESTful API 设计指南:<http://www.ruanyifeng.com/blog/2014/05/restful_api.html>

## Linux

### 说出linux常用命令

简答：建议实际操作，记忆更深

ls 查看目录中的文件

cd /home 进入 ‘/ home’ 目录；cd .. 返回上一级目录；cd ../.. 返回上两级目录

mkdir dir1 创建一个叫做 ‘dir1’ 的目录

rmdir dir1 删除一个叫做 ‘dir1’ 的目录 （只能删除空目录）

rm -f file1 删除一个叫做 ‘file1’ 的文件’，-f 参数，忽略不存在的文件，从不给出提示。

rm -rf /mulu 目录下面文件以及子目录下文件

cp /test1/file1 /test3/file2 如将/test1目录下的file1复制到/test3目录，并将文件名改为file2

mv /test1/file1 /test3/file2 如将/test1目录下的file1移动到/test3 目录，并将文件名改为file2

mv \* ../ Linux当前目录所有文件移动到上一级目录

ps -ef|grep xxx 显示进程pid

kill 使用kill命令来终结进程。先使用ps命令找到进程id，使用kill -9命令，终止进程。

tar –zxvf file.tar 解压 tar包

unzip file.zip 解压zip

unrar e file.rar 解压rar

free -m 查看服务器内存使用情况

### 如何查看所有java进程

简答：

ps -ef | grep java // grep是搜索关键字

ps -aux | grep java //-aux 显示所有状态

jps //java 自带的命令

### LINUX中如何查看某个端口是否被占用

简答：netstat -anp | grep 端口号

### find查找文件

find / -name httpd.conf　　#在根目录下查找文件httpd.conf，表示在整个硬盘查找

find /etc -name httpd.conf　　#在/etc目录下文件httpd.conf

find /etc -name ‘srm‘　　#使用通配符(0或者任意多个)。表示在/etc目录下查找文件名中含有字符串‘srm’的文件

find . -name ‘srm‘ 　　#表示当前目录下查找文件名开头是字符串‘srm’的文件

### Linux vi中查找字符内容的方法

简答：

1、命令模式下输入“/字符串”，例如“/Section 3”。

2、如果查找下一个，按“n”即可。

### Top命令

 top 命令则用于动态地持续监听进程的运行状态,而且可以查看系统的健康状态。

参考博文：<http://www.zixuephp.net/article-328.html>

## Git

要求：实操

教程：<https://www.liaoxuefeng.com/wiki/896043488029600>

自己搭建git远程仓库gitlab

## Spring 框架相关

### Bean的生命周期

要求：说出红色部分，黑色有助于理解。

1：Bean的实例化：

容器寻找Bean的定义信息并将其实例化。

2：属性注入：

使用依赖注入，Spring按照Bean定义信息配置Bean所有属性

3：BeanNameAware的setBeanName()：

如果Bean类有实现org.springframework.beans.BeanNameAware接口，工厂调用Bean的setBeanName()方法传递Bean的ID。

4：BeanFactoryAware的setBeanFactory()：

如果Bean类有实现org.springframework.beans.factory.BeanFactoryAware接口，工厂调用setBeanFactory()方法传入工厂自身。

5：BeanPostProcessors的ProcessBeforeInitialization()

如果有org.springframework.beans.factory.config.BeanPostProcessors和Bean关联，那么其postProcessBeforeInitialization()方法将被将被调用。

6：initializingBean的afterPropertiesSet()：

如果Bean类已实现org.springframework.beans.factory.InitializingBean接口，则执行他的afterProPertiesSet()方法

7：Bean定义文件中定义init-method：

可以在Bean定义文件中使用"init-method"属性设定方法名称例如：

如果有以上设置的话，则执行到这个阶段，就会执行initBean()方法

8：BeanPostProcessors的ProcessaAfterInitialization()

如果有任何的BeanPostProcessors实例与Bean实例关联，则执行BeanPostProcessors实例的ProcessaAfterInitialization()方法

9：此时bean已经准备就绪,可以被应用程序使用了,他们将一直驻留在应用上下文中,直到该应用上下文被销毁  
10：若bean实现了DisposableBean接口,spring将调用它的distroy()接口方法。同样的,如果bean使用了destroy-method属性声明了销毁方法,则该方法被调用

### Spring aop原理

 答： Spring AOP采用的是动态代理，在运行期间对业务方法进行增强，对于动态代理技术，Spring AOP提供了对JDK动态代理的支持以及CGLib的支持。

       JDK动态代理只能为接口创建动态代理实例，而不能对类创建动态代理。需要获得被目标类的接口信息（应用Java的反射技术），生成一个实现了代理接口的动态代理类（字节码），再通过反射机制获得动态代理类的构造函数，利用构造函数生成动态代理类的实例对象，在调用具体方法前调用invokeHandler方法来处理。

       CGLib动态代理需要依赖asm包，把被代理对象类的class文件加载进来，修改其字节码生成子类。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。

如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

### Spring IOC原理

什么是IOC：控制反转（Inversion of Control，缩写为IOC）：把创建对象的权利交给框架，是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。比如说A对象依赖B对象，不用spring的情况下，需要在A对象中new一个B对象，有了spring，它会帮助我们new一个B对象并注入到A对象中，这就是控制反转。所谓控制就是new 对象的控制权，由于控制权由a对象转交给了spring容器，所以控制权反转了。IOC也称依赖注入。

简答：

原理：spring框架解析xml文件，生成bean类的信息，根据bean类的信息利用java反射new出对象，如果这个对象又依赖了另外一个对象，先生成另外一个对象，并通过java反射赋值给原来的对象。

### spring与springboot的关系

简答：Spring Boot 并不是用来替代 Spring 的解决方案，它是基于Spring 框架用于快速构建spring应用的工具。通过自动化配置减少了很多重复性的配置信息。

### Spring Boot有哪些优点？

简答：

* 快速创建独立运行的spring项目与主流框架集成
* 项目可独立运行，无需外部依赖 Servlet 容器
* 使用了Starter（起步依赖包）管理依赖并版本控制
* 大量的自动配置，简化开发，方便集成第三方
* 提供准生产环境运行时的监控，如指标，健康，外部配置等

### springboot自动配置原理

简答：@EnableAutoConfiguration注解 通过@Import会导入一个EnableAutoConfigurationImportSelector的类，并执行其selectImports方法。selectImports方法最终调用SpringFactoriesLoader.loadFactoryNames，加载META-INF/spring.factories文件，将配置文件载入到内存中。然后过滤出key为org.springframework.boot.autoconfigure.EnableAutoConfiguration全限定名对应的配置类,spring最终将解析出的类注册到spring容器中。

### 约定优于配置

简答：约定优于配置（Convention Over Configuration）,是一种软件设计范式。目的在于减少软件开发人员所需要做出的决定的数量，从而获得简单的好处，而又不失去其中的灵活性。开发人员仅仅需要规定应用中不符合约定的部分

### Springmvc中controller是线程安全的吗？

简答：Spring生成对象默认是单例的（scope=singleton），这样就会多个线程共用一个controller，就会有线程安全问题。在没有共享变量的情况下，线程是安全的。

解决方法：

1、在Controller中使用ThreadLocal变量

2、在spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller

### Springmvc中在controller注入request（HttpServletRequest）会有线程安全问题吗？

简答：在spring框架初始IOC的时候，创建了一个Request对象的代理类，从而完成了初始注入，代理类负责从ThreadLocal中获取真正的Request对象并调用相应的方法，每次调用代理request的方法都相当于调用了该次请求真正的request对象的方法，因此不产生线程安全的问题。

### Spring事务传播行为

参考博文：<https://blog.csdn.net/weixin_39625809/article/details/80707695>

### mybatis一级缓存二级缓存

答： Mybatis的一级缓存是指SqlSession。一级缓存的作用域是一个SqlSession。Mybatis默认开启一级缓存。

在同一个SqlSession中，执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则SqlSession的缓存清空。

Mybatis的二级缓存是指mapper映射文件。二级缓存的作用域是同一个namespace下的mapper映射文件内容，多个SqlSession共享。Mybatis需要手动设置启动二级缓存。

在同一个namespace下的mapper文件中，执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则二级缓存清空。

参考文档：<https://www.cnblogs.com/happyflyingpig/p/7739749.html>

### Springmvc执行流程

1. 用户向服务器发送请求，请求被SpringMvc的前端控制器DispatcherServlet拦截；
2. DispatcherServlet对请求的URL(统一资源定位符)进行解析,得到URI(请求资源标识符)然后根据URI,调用HandlerMapping获得Handler配置的所有相关对象,包括Handler对象以及Handler对象对应的拦截器,这些对象都会封装到一个HandlerExecutionChain对象当中返回；
3. DispatcherServlet根据获取的Handler，选择一个合适的HandlerAdapter.HandlerAdapter的设计,符合面向对象的单一职责原则,代码结构清晰,便于维护,最为重要的是,代码的可复用性高,HandlerAdapter会被用于处理多种Handler,调用Handler实际处理请求的方法；
4. 提取请求中的模型数据,开始执行Handler(Controller).在填充Handler的入参过程中,根据配置,spring将帮助做一些额外的工作:

消息转换:将请求消息,如json/xml等数据转换成一个对象,将对象展缓为指定的响应信息；

数据转换:对请求信息进行数据转换,如string转换成integer,double等;

数据格式换:对请求的信息进行数据格式转换,如将字符串转换为格式化数字和格式化日期;

数据验证:验证数据的有效性,比如长度/格式等,验证结果存储到BindingResult或者ERROR中；

1. Handler执行完成后,向DispatcherServlet返回一个ModelAndView对象,ModelAndView对象中包含视图或视图模型；
2. 根据返回的ModelAndView对象,选择一个合适的ViewResolver(视图解析器)返回给DispatcherServlet；
3. ViewResolver结合Model和View来渲染视图；
4. 将视图渲染结果返回给客户端。

总结：以上八个步骤,DispatcherServlet/HandlerMapping/HandlerAdapter/ViewResolver等对象协同完成工作,完成SpringMVC请求–>响应的整个工作流程,这些对象完成的工作对于开发者而言都是不可见的,开发者不需要关心这些对象是如何完成工作的,开发者只需要在Handler(Controller)当中完成对请求的业务处理.

## JVM 相关

### 类的双亲委派机制及其作用

答：1. 当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

2. 当前classLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrp ClassLoader.

3. 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

### Jvm内存模型

答：程序计数器、java虚拟机栈、本地方法栈、堆、方法区。

说出各个区域存储的内容，具体查看附件《JVM面试相关》----内存模型----部分

### 判断对象是否可回收算法、GC算法

答：

判断对象是否可回收算法：引用计数法、可达性分析

* 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。
* 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

Gc算法：标记清除、复制、标记整理

掌握优缺点

**标记清除**

优点：实现简单

缺点：内存碎片化

**复制**

优点：对象存活时间短时，效率比较高。无内存碎片化

缺点：对象存放率比较高时，复制次数多，效率低。内存可以空间缩小一半。

**标记整理**

优点：相较于前两种，无内存碎片化问题及空间缩小一半的问题

缺点：效率相对前两种要低

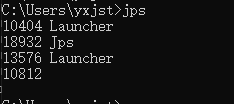
### 内存溢出解决方案

答：

jmap命令可以获得运行中的jvm的堆的快照，从而可以离线分析堆，以检查内存泄漏，检查一些严重影响性能的大对象的创建，检查系统中什么对象最多，各种对象所占内存的大小等等。可以使用jmap生成Heap Dump。 使用jmap -histo[:live] pid查看堆内存中的对象数目、大小统计直方图，如果带上live则只统计活对象。

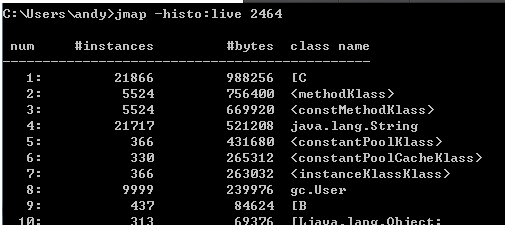
查找过程：

1、Jsp 查找java进程的pid



2、jmap -histo pid 展示class的内存情况

说明：instances（实例数）、bytes（大小）、classs name（类名）。它基本是按照使用使用大小逆序排列的。



如果查看某个你自定义的类的对象实例特别多，很有可能就是这个类的对象过多导致的内存溢出。再查找程序当中那个地方使用该类，内存溢出很有可能就出现在这些地方。

参考博文：<https://www.cnblogs.com/kongzhongqijing/articles/3621163.html>

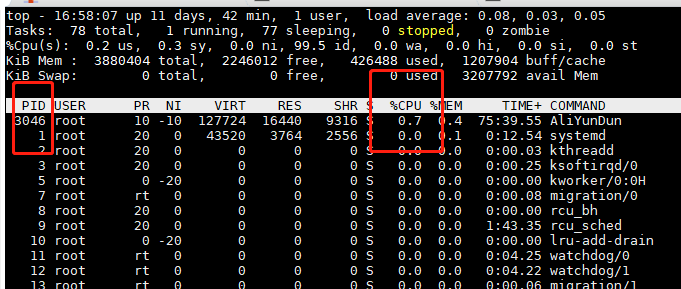
### 查找CPU占用率高的线程

jstack

观察jvm中当前所有线程的运行情况和线程当前状态

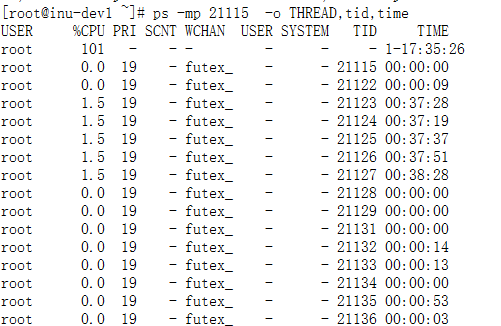
答：

1、执行top命令查看哪个java进程cpu占有率高，记录它的pid（注意这里展示出来的是整个系统的进程，bin不是只有java进程）



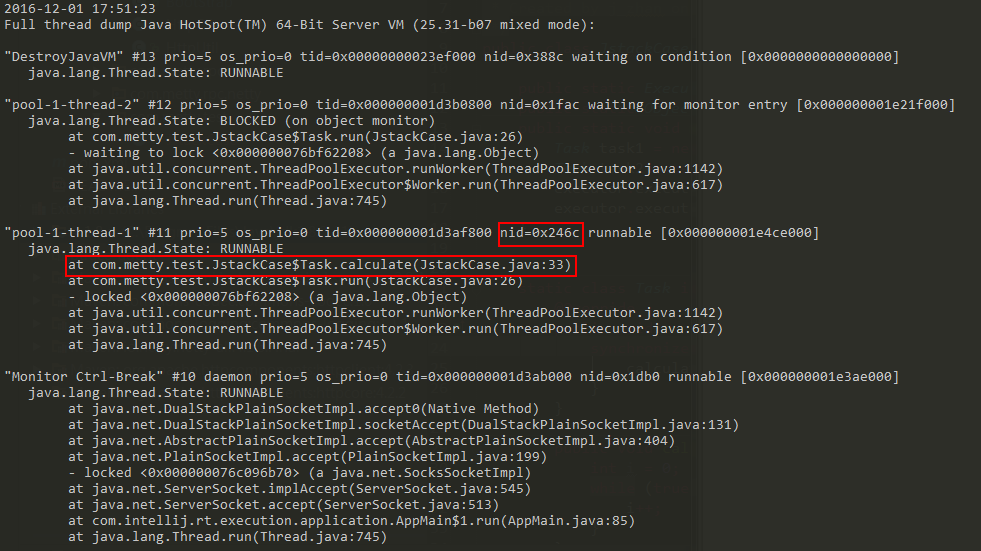
2、利用pid及相关命令，查找出该java进程中哪个线程cpu占有率 。

执行命令：ps -mp pid -o THREAD,tid,time



%CPU即 线程的使用率，tid即线程id

3、执行jps命令，查找java进程的pid，并执行jstack pid > ps.txt 导出线程信息



这里的nid就是该线程的线程id，但这里的线程id是十六进制的。我们可以把步骤2查找出的线程id转换成十六进制，并与jstack dump出来的线程信做比较，线程id相等的即是该线程了，然后根据线程信息找到cpu占有率高的代码。

### Jvm常用调参

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX:+PrintGCDetails(打印GC的详细信息)

-Xloggc:log/ge.log(指定GC.log的位置，以文件形式输出)

-XX:+HeapDumpOnOutOfMemoryError(将OOM时的堆信息导出到文件)

## mysql数据库

### mysql有哪几种索引类型？索引结构（BTREE）？

简答：

primary：唯一索引，不允许为null。

key：普通非唯一索引。

unique：表示唯一的，不允许重复的索引，可以为null。

fulltext: 表示全文搜索的索引。 FULLTEXT用于搜索很长一篇文章的时候，效果最好。用在比较短的文本，如果就一两行字的，普通的INDEX 也可以。

spatial：空间索引。

### 索引结构（BTREE）？

详细请查看附件《Mysql索引》-----------B+Tree------------部分

### Mysql索引优化流程？

简答：

1、通过show status 命令了解各种sql的执行效率

2、定位执行效率较低的SQL语句（通过慢查询记录+show processlist命令查看当前线程）

3、通过explain分析低效率的SQL语句的执行情况

4、确定问题并采取相应的优化措施。

参考博文：<https://www.cnblogs.com/jian0110/p/9356347.html>

### Union和union all区别？

简答：

1. Union，对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；
2. Union All，对两个结果集进行并集操作，包括重复行，不进行排序；
3. 从效率上说，UNION ALL 要比UNION快很多，所以，如果可以确认合并的两个结果集中不包含重复的数据的话，那么就使用UNION ALL。

### mysql悲观锁与乐观锁？悲观锁乐观锁选择？

简答：请查看附件《mysql锁详解》中的---悲观锁与乐观锁---部分。

### Mysql如何查看sql是否使用索引？

简答：使用explain 命令查看sql执行情况。

请查看《mysql索引》中的---Mysql查看是否使用到索引---部分。

### 数据库引擎：innodb和myisam、memory区别 ？场景下选择？

要求：重点掌握myisam和innodb

**InnoDB：**支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择InnoDB，因为支持事务的提交（commit）和回滚（rollback）。

**MyISAM：**插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比 较低，也可以使用。

**MEMORY：**所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择MEMOEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。

引擎区别可以参考博文：<https://www.jianshu.com/p/dc60346d55a2>

### Mysql部署？

要求:先了解一下mysql架构，请看《Mysql架构及原理》

说明：我们学习时安装的mysql不具有高可用性，如果这个节点挂了是无法对外提供服务的，在企业中需要保证高可用。

简答：数据量不大时，可以按mysql主从架构方式部署两个实例。一个实例作为master节点，master节点对外提供读写服务。一个作为slave节点，从master节点复制数据，主要用于备灾。这种架构下，如果主节点出问题，只能手动升级slave节点为master继续对外提供服务。这里需要两台服务器部署这两个节点，使用阿里云服务器，每台服务器配置为：4核8G。

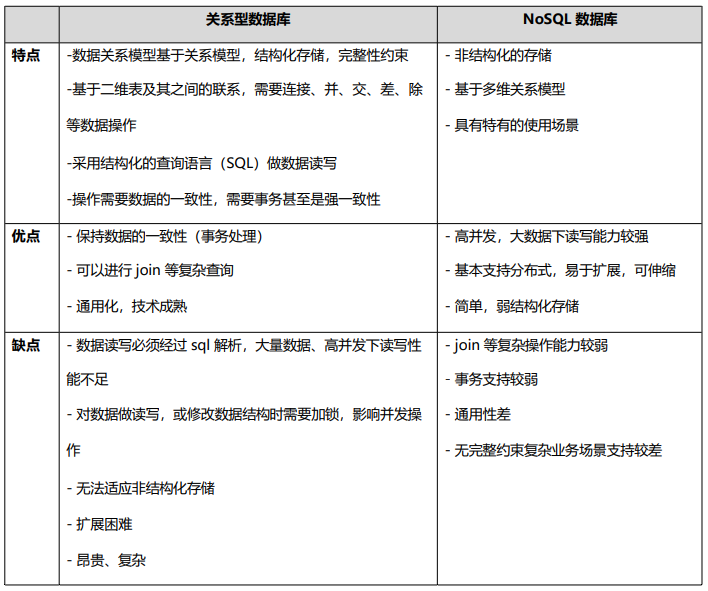
注意：4核cpu基本能够满足mysql的计算要求无须再大，内存是越大越好。根据应用大小及访问量适当调低调高配置。

### 数据库分库分表？如何扩容？

要求: 分库分表的目的是为了解决单表单库的性能瓶颈问题。具体方案原理请查看附件《Mysql分库分表》。

扩容方案请查看附件《Mysql分库分表》---扩容---部分

### 关系型数据库与nosql数据库的优缺点？



### Mysql主从同步？读写分离？主从一数据致性问题？

简答：

主从同步：主从同步使得数据可以从一个数据库服务器复制到其他服务器上，在复制数据时，一个服务器充当主服务器（master），其余的服务器充当从服务器（slave）。

读写分离：读写分离就是只在主服务器上写，只在从服务器上读， 把读压力分摊到从服务器上，提高整体的服务能力。

数据一致性：使用半同步复制模式实现主从同步（这里只有一台从服务器能及时同步数据）。其它方式请查看附件《Mysql架构及部署》---数据不一致解决方案---部分

主从同步和读写分离详情，请看附件《Mysql架构及部署》

### 分布式系统唯一ID生成方法？分库分表后主键id是如何保证唯一的？订单号规则？

简答：

1. 数据库自增长id
2. Java uuid
3. Redis INCR和INCRBY来实现
4. Twitter的snowflake算法

参考博文：<https://www.cnblogs.com/haoxinyue/p/5208136.html>

## Redis

### Redis基本数据结构？项目当中的应用场景？

简答：Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

应用场景：主要用于数据缓存及临时数据存储，如:session存储、数据库缓存、验证码存储。（主要讲一下你的项目当中的运用）

### Redis内存占用满了怎么办？Redis有哪几种数据淘汰策略？

简答：当redis内存到达最大可用大小时，会对数据进行淘汰，淘汰选择算法有：LRU、RANDOM、TTL、NOEVICATION（详细解释看下面淘汰策略）。默认的策略为noeviction策略。

redis设置配置文件的maxmemory参数，可以控制其最大可用内存大小

redis 提供 6种数据淘汰策略通过maxmemory-policy设置策略：

LRU：最近最少使用

淘汰策略：

|  |  |
| --- | --- |
| 规则名称 | 规则说明 |
| volatile-lru | 在设置了过期时间的键空间中，优先移除最近未使用的key。 |
| allkeys-lru | 在主键空间中，优先移除最近未使用的key。 |
| volatile-random | 在设置了过期时间的键空间中，随机移除某个key。 |
| allkeys-random | 在主键空间中，随机移除某个key。 |
| volatile-ttl | 在设置了过期时间的键空间中，具有更早过期时间的key优先移除。 |
| noeviction | 当内存使用达到阈值的时候，所有引起申请内存的命令会报错。 |

### Redis到底是多线程还是单线程，为什么？线程安全吗？

简答：redis是单线程的，线程安全。

因为CPU不是Redis的瓶颈。Redis的瓶颈最有可能是机器内存或者网络带宽。单线程容易实现，而且CPU不会成为瓶颈，所以更适合单线程。如果想利用cpu多核，可以在该机器上部署多个redis实例。

### Redis单线程下为什么能执行这么快？

简答：

(1) 绝大部分请求是纯粹的内存操作（非常快速）

(2) 采用单线程,避免了不必要的上下文切换和竞争条件

(3) 非阻塞IO - IO多路复用

### Redis中的管道？Redis事务？

简答：

管道模式：（pipeline）可以一次性发送多条命令并在执行完后一次性将结果返回，pipeline通过减少客户端与redis的通信次数来实现降低往返延时时间，而且Pipeline 实现的原理是队列，而队列的原理是时先进先出，这样就保证数据的顺序性。

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务操作的相关命令：MULTI、EXEC、DISCARD、WATCH

### Redis分布式锁实现

要求：需要掌握redis分布式锁的实现，具体可以查看录制的分布式锁实现视频。

或者参考博文：<https://www.cnblogs.com/fixzd/p/9479970.html>

Redis锁实现有开源的软件：Redisson

Redisson类似于jedis，也是一个java实现的redis客户端，它里面提供分布式锁实现，可以直接调用。你可以说项目当中用的就是这个分布式锁的实现。

Redisson 与jedis对比了解：<https://blog.csdn.net/niuzaiwenjie/article/details/79875543>

### Redis持久化方式？及其优缺点？

简答：Redis提供两种持久化机制RDB和AOF机制。

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。

AOF持久化方式记录每次对服务器写的操作，以redis命令请求协议的格式(完全持久化存储)保存为aof文件。

优缺点（主要从效率和安全性两个方面来讲）

1. RDB方式直接将内存数据写入快照文件，没有额外多余的数据，而AOF记录的是客户端的写操作命令，所以同样大小的数据量时，RDB要比AOF文件小，且恢复速度也更快。
2. RDB在指定时间间隔才执行一次快照存储，而AOF可以设置高频率的执行写入操作，在系统奔溃时，AOF相较于RDB，丢失更少的数据，安全性相对较高。由于高频率的写入数据到硬盘，AOF写入性能会逊于RDB。

具体持久化原理及详细情况，请查看附件《Redis》---redis持久化---部分

### Redis架构模式

简答：单机、主从、主从+哨兵、集群。

**主从**：一个主节点接受外部的读写请求，多个从节点负责从主节点复制复制数据，正常情况下从节点不接受写请求，当然也可以设置参数使其接受。这种架构下可以在主节点奔溃时，由从节点升级为主节点，继续对外提供服务。

**主从+哨兵**：该模式下相较于主从模式多了个哨兵，哨兵可以监控故障节点，当主节点出现故障时自动将从库转为主库。

**集群**：主从模式下无法扩展redis集群的容量，从节点只是对主节点数据的复制。在集群架构下，多个主节点分别负责集群的一部分数据，可以通过增加集群节点数提升整体的存储容量。

各种架构模式详细介绍请看附件《Redis》---redis架构模式---部分

### 你们公司是如何Redis部署？

简答：我们公司redis存储数据还不是很大（大概8G左右），我们采用主从加哨兵模式保证redis服务的高可用。整个集群1个主redis，1个从redis，3个哨兵（至少要3个哨兵才能保证高可用，否则无法选举leader，无法实现主从切换）。可允许一台哨兵或redis实例挂掉。使用阿里云服务器，每台机器配置：16G 内存+ 4 核 CPU（没问配置可不必说机器具体配置）

注意：其中两台哨兵可以跟redis一起部署，另外一台可部署在其他服务器上，哨兵对服务器要求不高。

### 缓存击穿、缓存雪崩、缓存穿透？

### Redis缓存一致性问题？

简答：

* 1. 先更新数据库，再更新缓存
  2. 先删除缓存，再更新数据库
  3. 先更新数据库，再删除缓存

这几种方式在高并发下都会有缓存不一致性问题，可以在数据库和缓存操作前，设置缓存的过期时间，即使后面缓存或数据库失败了，也能保证数据的最终一致性。

详情请查看附件《缓存不一致性》

### 什么是一致性哈希算法？什么是哈希槽？

要求：做了解

参考博文：<https://www.cnblogs.com/lpfuture/p/5796398.html>

## 消息中间件

### 消息中间件应用场景

简答：任务异步处理、应用解耦、流量削峰（最好能答出你在项目当中是如何运用的）。

### JMS和AMQP

简答：JMS是java的接口规范，旨在为java应用提供统一的消息操作。

JMS是接口规范，具有接口开发的特点 ：1、有利于程序设计的规范化。2、实现了程序的可插拔性，降低了耦合度。

AMQP是一种协议规范，凡是遵循同一协议规范的不同系统能够互相通讯（了解）。

### Rabbbitmq几种工作模式

要求：了解几种常用的几种工作模式，worker queue、发布订阅模式、路由模式、主题模式。

一个队列可以绑定多个消费者，同一个队列里的消费者不能重复消费一条消息，不同队列里的消费者是可以重复消费消息的。

不同工作模式的最大区别在于使用不同的交换机，不同的交换机有不同的消息转发规则。

work模式：只有默认的交换机，不能指定交换机，一个队列可以绑定多个消费者，每个队列的消息只能被其中一个消费者消费。

订阅模式：交换机会把消息转发给所有与之绑定的queue。

路由模式：交换机会把消息转发给所有与之绑定的且routingkey与该消息routingkey相等的queue。

topic模式：该模式与路由模式类似，在routingkey匹配上使用通配符匹配。

通配符：\*可以匹配一个标识符，#可以匹配0个或多个标识符

header模式：了解

### 如何保证RabbitMQ的高可用性？

简答：可以搭建镜像集群模式， 镜像模式下queue被同步到多个节点，即使某个节点挂了，还可以通过其它节点对外提供服务。

详情请查看附件《RabbitMq》--- 如何保证RabbitMQ 的高可用性？----部分

### 如何保证RabbitMQ不丢失消息？在使用过程中有没有遇到消息丢失的情况？

简答：

1. 消息发送可以通过事务模式或confirm模式确认机制来确保生产者不丢消息。
2. 消息消费通过手动确认方式来确保消息一定会被消费。
3. 对queue及消息设置为持久化确保服务重启后消息不会丢失。
4. 搭建集群模式确保消息不会因为某个节点顺坏（如硬盘损坏）而导致消息丢失。

详情请查看附件《RabbitMQ》--消息的可靠传输----部分

### RabbitMQ如何保证消息顺序消费？

简答：确保需要顺序消费的一组消息进入同一队列，且队列只能绑定一个消费者。

发布订阅模式、路由模式、主题模式都能实现消息顺序消费。

以路由模式为例，将一组需要顺序消费的消息设置相同的routingkey，且只能有一个queue与这个routingkey做绑定，这样这一组消息就会按发送顺讯进入到同一队列。同时为了保证在消费端能够顺序消费，每个queue只能绑定一个消费者。

详情请查看附件《RabbitMQ》---如何保证消息顺序消费---部分

### RabbitMQ如何避免消息重复的被消费？

简答：

什么时候会重复消费？

处理业务逻辑后，向rabbitmq发送ack消息，由于网络等原因，造成rabbitmq无法接受到ack消息，导致rabbitmq会重复发送该消息给消费者。

解决：

消费端处理消息的业务逻辑保持幂等性。

**方案1：**

在消息处理前进行去重判断，但每个消息需要需有一个消息的唯一ID作为去重的依据，如果消息已处理这不再进行消费，直接ack回复mq。这里我们可以使用redis存储已处理的消息id来进行去重。

步骤

1、查询redis是否已存在该消息id，如果存在则该消息已处理过，直接跳到第5步骤

2、如果redis中不存在该消息的唯一id，则进行业务逻辑处理

3、处理的业务结果保存至数据库（支持事务的数据库）

4、将消息的id存储至redis中。

5、发送ack至rabbitmq。

注意：这个方案业务结果保存一定要放在支持事务的数据库中，否则在业务成功情况下，redis保存失败就会导致重复消费的问题。如果数据库支持事务，在redis操作失败的情况下可以回滚数据库保存的处理结果信息，也就无所谓再处理一次消息了。

**方案2：**

在方案1中数据库需要支持事务，假如我们使用的是mongodb，就会有重复消费的问题。

此时我们可以把消息唯一id和处理结果一起保存到一个数据库表中，这样一起成功一起失败，也就不会有重复消费的问题了。

前提：保证消息有一个唯一id

步骤：

1. 如果数据库中不存在该消息的唯一id，则进行业务处理
2. 如果数据库中不存在该消息id，则进行业务处理

3、处理的业务结果与消息唯一id保存至数据库

4、发送ack至rabbitmq.

### RabbitMQ是如何部署的？

简答：为了节约资源确保rabbitmq的高可用，我们公司使用两台阿里云服务器部署rabbitmq，服务器配置为：4核，8GB。我们采用镜像集群模式部署rabbitmq，为了使数据不丢失，两个节点都设置为磁盘节点。

节点说明：

内存节点将所有的队列，交换器，绑定关系，用户，权限，和vhost的元数据信息保存在内存中。磁盘节点将这些信息保存在磁盘中，但是内存节点的性能更高，为了保证集群的高可用性，必须保证集群中有两个以上的磁盘节点，来保证当有一个磁盘节点崩溃了，集群还能对外提供访问服务。

## 搜索引擎专题

### Solr部署？

简答：我们公司采用solrCould模式部署solr集群，solrCloud支持故障自动切换，由于数据还不是很大，集群部署3个zookeeper节点和两个solr节点，两个solr节点互备，允许一台节点挂机后，另外一台还能继续对外提供服务。两台solr节点服务器硬件配置：4核，16GB。

参考文档:

主从模式和solrCloud区别：<https://blog.csdn.net/jiangchao858/article/details/53363310>

solrcloud 架构：<https://blog.csdn.net/zengxiaosen/article/details/78790253>

<https://blog.csdn.net/xiatianba/article/details/84630025>

部署方案：<https://www.jianshu.com/p/751129f6b4ed>

solrCloud集群搭建：<https://blog.csdn.net/yougoule/article/details/78445759>

### Solr query 和filter query

简答: query是普通的检索查询，具有复杂的评分机制来衡量搜索结果的相关性。filter query 是过滤查询，可以对某些查询域进行过滤查询，这些过滤域不参与评分，而且filter query还对查询结果进行缓存，能够提高后续的查询效率。

Filter query 应用场景：比如商品搜索用户可以对商品品牌、关键属性、价格进筛选

### 倒排索引

这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。

参考文档: <https://www.jianshu.com/p/b3f987b0fbf1>

### TF-IDF算法/相似度算法/评分算法

简答：TF:词频,IDF：逆向文档频率，TF-IDF是一种统计方法

1、某个词或短语在一篇文章中出现的次数越多，越相关

2、整个文档集合中包含某个词的文档数量越少，这个词越重要

参考文档：<https://blog.csdn.net/starzhou/article/details/51568262>

## springcloud微服务

### 不同微服务交互注意事项

简答：分布式事务、并发问题、接口幂等性、接口兼容性

### 微服务架构与传统的单体应用架构有什么优缺点？

**单体应用不足：**

1、不利于敏捷开发和部署

2、随着应用越来越大，启动慢、降低开发效率

3、可靠性差，某个模块出问题可能会影响整个应用的运行

4、阻碍技术创新：单体应用往往使用统一的技术平台或方案解决所有的问题，团队中的每个成员必须使用相同的开发语言和框架，要想引入新框架或新技术平台会非常困难。

5、系统愈加复杂，模块多，模块的边界模糊，依赖关系不清晰、代码难以理解、不利于维护。

**微服务架构**

优点：

1、每个服务足够内聚、足够小，代码易于理解，降低每个服务开发的复杂度。

2、服务独立测试、部署、升级、发布；服务之间相互影响较小。

3、技术栈不受限，在微服务架构中，可以结合项目业务及团队的特点，合理选择技术栈。新技术的应用，系统不会被长期限制在某个技术栈上；

4、 按需收缩：可根据需求，实现细粒度的扩展。例如，系统中的某个微服务遇到了瓶颈，可以结合这个微服务的业务特点，增加内存、升级CPU或者增加节点。

缺点：

1、增大了系统运维部署的复杂度

2、分布式开发提高了技术门槛，例如：微服务有各自的数据库，很难在不采用分布式事务的情况下跨服务实现功能。

3、服务和服务之间通过接口来“联系”，当某一个服务更改接口格式时，可能涉及到此接口的所有服务都需要做调整。

### 服务降级、熔断、限流、资源隔离？

简答：

1. **限流机制**：限流机制主要是提前对各个类型的请求设置最高的QPS阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。
2. **熔断机制**：当失败率达到阀值自动触发降级（如因网络故障、超时造成的失败率真高），熔断器触发的快速失败会进行快速恢复。
3. **服务降级：**当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心交易正常运作或高效运作。

**熔断与降级的区别**

熔断的目的是当A服务模块中的某块程序出现故障后为了不影响其他客户端的请求而做出的及时回应。

降级的目的是为了解决整体项目的压力，而牺牲掉某一服务模块而采取的措施。

### Spring Cloud Eureka（注册中心）

简答：eureka作为服务的注册中心，提供服务的注册与发现、服务检查等功能。

**说明：**

**服务注册和服务发现**：服务注册是指微服务在启动的时，将自己的信息注册到服务发现组件上的过程。服务发现是指查询可用微服务列表及其网络地址的机制。

**服务检查：**服务发现组件使用一定机制定时检测已注册的服务，如发现某实例长时间无法访问，就会在服务注册表中移除该实例。

**Eureka 服务列表获取更新**

1、Eureka client从注册中心更新服务列表，然后自身会做缓存；

2、作为服务消费者，就是从这些缓存信息中获取的服务提供者的信息；

3、增量更新的服务以30秒为周期循环调用；

4、增量更新数据在服务端保存时间为3分钟，因此Eureka client取得的数据虽然被称为"增量更新"，仍然可能和30秒前取的数据一样，所以Eureka client要自己来处理重复信息；

5、由3、4两点可以推断出，Eureka client的增量更新，其实获取的是Eureka server最近三分钟内的变更，因此，如果Eureka client有超过三分钟没有做增量更新的话（例如网络问题），那么再调用增量更新接口时，那三分钟内Eureka server的变更就可能获取不到了，这就造成了Eureka server和Eureka client之间的数据不一致，需要有个方案来及时发现这个问题；

6、正常情况下，Eureka client多次增量更新后，最终的服务列表数据应该Eureka server保持一致，但如果期间发生异常，可能导致和Eureka server的数据不一致，为了暴露这个问题，Eureka server每次返回的增量更新数据中，会带有一致性哈希码，Eureka client用本地服务列表数据算出的一致性哈希码应该和Eureka server返回的一致，若不一致就证明增量更新出了问题导致Eureka client和Eureka server上的服务列表信息不一致了，此时需要全量更新；

详细使用请参考附件《微服务讲义.pdf》-----eureka-----部分

### Spring Cloud Ribbon（负载均衡）

简答：实现服务调用的负载均衡，主要提供客户端的软件负载均衡算法。记住几个负载均衡算法：RandomRule、RoundRobinRule、WeightedResponseTimeRule等

说明：假设服务A调用服务B，服务B有可能会部署多个应用实例。此时服务A调用服务B就得选择其中一个实例进行调用，而具体选择哪个实例就是负载均衡算法决定。

**Ribbon内置的负载均衡算法**

**AvailabilityFilteringRule**： 这条规则将跳过被认为是“电路跳闸”的服务器或具有高并发连接数的服务器。

BestAvailableRule：这个规则会跳过“电路跳闸”的服务器，然后选取一个并发请求数最低的服务。

**RandomRule**：一种随机分配流量的负载平衡策略。

RetryRule：在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的server。

**RoundRobinRule**：这条规则简单地通过轮询来选择服务器。它通常被用作更高级规则的默认规则或后退。

**WeightedResponseTimeRule**：对于这个规则，每个服务器根据其平均响应时间给定一个权重。响应时间越长，得到的权重就越小。规则随机选择一个服务器，其中的可能性由服务器的权重决定。

ZoneAvoidanceRule：使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。

PredicateBasedRule：将服务器过滤逻辑委托给{@link AbstractServerPredicate}实例的规则。过滤后，服务器会以循环的方式从过滤列表中返回。

详细使用请参考附件《微服务讲义.pdf》-----ribbon-----部分

### Spring Cloud Feign（负载均衡）

简答：Feign是一个声明式的Web服务客户端。使用Feign能让编写Web服务客户端更加简单。

Feign是一个声明式的Web服务客户端。使用Feign能让编写Web服务客户端更加简单，它的使用方法是定义一个接口，然后在接口上添加注解，同时也支持JAX-RS标准的注解。Feign也支持可插拔式的编码器和解码器。SpringCloud对Feign进行了封装，使其支持SpringMVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

详细使用请参考附件《微服务讲义.pdf》-----feign-----部分

### Spring Cloud Hystrix（熔断器）

简答：Hystrix是Netflix公司开源的一个项目,它提供了熔断器功能,能够阻止分布式系统中的联动故障.

说明：

分布式系统中服务与服务之间相互依赖,一种不可避免的情况就是当某些服务出现故障时,依赖于它们的其他服务出现远程调度的线程阻塞,在高并发的情况下,可能在几秒钟内就会使整个服务处于线程负载饱和状态,从而从一个服务不可用扩散到整个服务不可用,既雪崩效应.Hystrix是Netflix公司开源的一个项目,它提供了熔断器功能,能够阻止分布式系统中的联动故障.

**hystrix的出现即为解决雪崩效应，它通过四个方面的机制来解决这个问题**

**降级机制**：当请求超时、资源不足（线程资源）、或服务器压力过大时，我们直接返回简单或默认的数据，不再继续等待处理该请求，而是直接返回客户端。

**资源隔离（线程池隔离和信号量隔离）：**限制调用分布式服务的资源使用，某一个调用的服务出现问题不会影响其他服务调用。

**熔断**：当失败率达到阀值自动触发降级(如因网络故障/超时造成的失败率高)，熔断器触发的快速失败会进行快速恢复。

**限流机制**：限流机制主要是提前对各个类型的请求设置最高的QPS阈值，若高于设置的阈值则对该请求直接返回，不再调用后续资源。

详细使用请参考附件《微服务讲义.pdf》-----hystrix-----部分

### Hystrix隔离策略及其选择

Hystrix的资源隔离策略有两种，分别为：**线程池和信号量**。

Hystrix的隔离主要是为每个依赖组件提供一个隔离的线程环境，提供两种模式的隔离：

**线程池隔离模式**：使用一个线程池来处理当前的请求，线程池对请求作处理，设置任务返回处理超时时间，堆积的请求堆积进入线程池队列。这种方式需要为每个依赖的服务申请线程池，有一定的资源消耗，好处是可以应对突发流量（流量洪峰来临时，处理不完可将数据存储到线程池队里慢慢处理）

**信号量隔离模式**：使用一个原子计数器（或信号量）来记录当前有多少个线程在运行，请求来先判断计数器的数值，若超过设置的最大线程个数则丢弃改类型的新请求，若不超过则执行计数操作请求来计数器+1，请求返回计数器-1。这种方式是严格的控制线程且立即返回模式，无法应对突发流量（流量洪峰来临时，处理的线程超过数量，其他的请求会直接返回，不继续去请求依赖的服务）

比较

|  |  |  |
| --- | --- | --- |
|  | 线程池隔离 | 信号量隔离 |
| 线程 | 与调用线程不相同线程 | 与调用线程相同 |
| 开销 | 排队、调度、上下文开销等 | 无线程切换，开销低 |
| 异步 | 支持 | 不支持 |
| 并发支持 | 支持（最大线程池大小） | 支持（最大信号量上限） |

**如何在线程池和信号量之间做选择？**

当请求的服务网络开销比较大的时候，或者是请求比较耗时的时候，我们最好是使用线程池隔离策略，这样的话，可以保证大量的容器(tomcat)线程可用，不会由于服务原因，一直处于阻塞或等待状态，快速失败返回。针对纯内存操作，比如比较复杂的耗时的逻辑，我们可以使用信号量隔离策略，因为这类服务的返回通常会非常的快，不会占用容器线程太长时间，而且也减少了线程切换的一些开销。

### Spring Cloud Zuul(网关)

简答：网关是外部调用系统内部服务的入口，可以实现屏蔽内部细节、服务路由、安全认证、限流熔断、日志监控等功能。

spring cloud zuul包含4种类型的过滤器。

pre过滤器：在请求被路由之前调用。Zuul请求微服务之前。比如请求身份验证，选择微服务实例，记录调试信息等。

route过滤器：负责转发请求到微服务。原始请求在此构建，并使用Apache HttpClient或Netflix Ribbon发送原始请求。

post过滤器：在route和error过滤器之后被调用。可以在响应添加标准HTTP Header、收集统计信息和指标，以及将响应发送给客户端等。

error过滤器：在处理请求发生错误时被调用。

详细使用请参考附件《微服务讲义.pdf》-----zuul-----部分

## Dubbo专题

### dubbo 负载均衡策略？

简答：

**1、random loadbalance**

默认情况下，dubbo 是 random load balance ，即随机调用实现负载均衡，可以对 provider 不同实例设置不同的权重，会按照权重来负载均衡，权重越大分配流量越高，一般就用这个默认的就可以了。

**2、Roundrobin loadbalance**

这个的话默认就是均匀地将流量打到各个机器上去，但是如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。

**3、leastactive loadbalance**

这个就是自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求。

**4、consistanthash loadbalance**

一致性 Hash 算法，相同参数的请求一定分发到一个 provider 上去，provider 挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大。如果你需要的不是随机负载均衡，是要一类请求都到一个节点，那就走这个一致性 Hash 策略。

### dubbo 集群容错策略

简答：

**1、failover cluster 模式**

失败自动切换，自动重试其他机器，默认就是这个，常见于读操作。

**2、failfast cluster模式**

一次调用失败就立即失败，常见于写操作。（调用失败就立即失败）

**3、failsafe cluster 模式**

出现异常时忽略掉，常用于不重要的接口调用，比如记录日志。

**4、failback cluster 模式**

失败了后台自动记录请求，然后定时重发，比较适合于写消息队列这种。

**5、forking cluster 模式**

并行调用多个 provider，只要一个成功就立即返回。

**6、broadcacst cluster**

逐个调用所有的 provider。

### dubbo动态代理策略

简答：在Dubbo中，没有使用CGLib进行代理，默认使用 javassist 动态字节码生成，创建代理类。但是可以通过 spi 扩展机制配置自己的动态代理策略。

### 注册中心挂了Dubbo服务可以继续通信吗？

简答:可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息拉取到本地缓存，所以注册中心挂了可以继续通信。

### dubbo 支持不同的通信协议？

简答：dubbo 协议、rmi 协议、hessian 协议、http 协议、webservice

**dubbo 协议**

默认就是走 dubbo 协议，单一长连接，进行的是 NIO 异步通信，基于 hessian 作为序列化协议。使用的场景是：传输数据量小（每次请求在 100kb 以内），但是并发量很高。

为了要支持高并发场景，一般是服务提供者就几台机器，但是服务消费者有上百台，可能每天调用量达到上亿次！此时用长连接是最合适的，就是跟每个服务消费者维持一个长连接就可以，可能总共就 100 个连接。然后后面直接基于长连接 NIO 异步通信，可以支撑高并发请求。

长连接，通俗点说，就是建立连接过后可以持续发送请求，无须再建立连接。而短连接，每次要发送请求之前，需要先重新建立一次连接。

**rmi 协议**

走 Java 二进制序列化，多个短连接，适合消费者和提供者数量差不多的情况，适用于文件的传输，一般较少用。

**hessian 协议**

走 hessian 序列化协议，多个短连接，适用于提供者数量比消费者数量还多的情况，适用于文件的传输，一般较少用。

**http 协议**

走 json 序列化。

**webservice**

走 SOAP 文本序列化。

### dubbo 支持的序列化协议

简答：dubbo 支持 hession、Java 二进制序列化、json、SOAP 文本序列化多种序列化协议。hessian 是其默认的序列化协议。

### Dubbo默认使用的是什么通信框架？

Dubbo 默认使用 Netty 框架，也是推荐的选择，另外内容还集成有Mina、Grizzly。

### dubbo 和 dubbox 之间的区别？

简答：dubbox 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。

### dubbo注册了多个一样的服务实例，如何调用指定的某一个服务呢？

简答：开发、测试环境可通过指定Url方式绕过注册中心直连指定的服务地址。

<dubbo:reference id="providerService" interface="org.jstudioframework.dubbo.demo.service.ProviderService" url="dubbo://127.0.0.1:29014"/>

说明：

可以配置环境点对点直连，绕过注册中心，将以服务接口为单位，忽略注册中心的提供者列表。

开发、测试环境可通过指定Url方式绕过注册中心直连指定的服务地址，避免注册中心中服务过多，启动建立连接时间过长，如

<dubbo:reference id="providerService" interface="org.jstudioframework.dubbo.demo.service.ProviderService" url="dubbo://127.0.0.1:29014"/>

### Dubbo需要Web容器(tomcat/jetty)吗？

简答：不需要，如果硬要用 Web 容器，只会增加复杂性，也浪费资源。

运行dubbo应用可以不需要部署tomcat等容器，直接Main方法运行。

### Dubbo服务运行过程

1. 服务容器负责启动，加载，运行服务提供者。  
   1. 服务提供者在启动时，向注册中心注册自己提供的服务。  
   2. 服务消费者在启动时，向注册中心订阅自己所需的服务。  
   3. 注册中心返回服务**提供者地址列表**给消费者，如果有变更，注册中心将**基于长连接推送变更数据**给消费者。  
   4. **服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用**。  
   5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。  
   6. 注册中心，服务提供者，服务消费者三者之间均为**长连接**，监控中心除外  
   7. **注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者**  
   8. **注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表**

### Dubbo RPC实现原理

a) 假设两台服务器A，B,一个应用部署在A服务器上,想要调用B服务器上应用提供的函数/方法,由于不在一个内存空间,不能直接调用,需要通过网络来表达调用的语义和传达调用的数据

b) 首先A和B建立TCP链接,并且确定好RPC框架的网路端口,能够进行网络通信

c) 然后A服务器将需要调用B服务器的方法和参数进行序列化（Serialize）

d) 通过第一步建立的链接,将序列化后的二进制流发送给B

e) B服务器收到请求后,需要对参数进行反序列化,恢复为内存中的表达方式

f) 然后B服务器找到对应的方法（寻址的一部分）进行本地调用,然后得到返回值

g) B服务器对返回值再次进行序列化,并且通过相同的途径发送给A

h) A对B服务器返回的信息再进行反序列化,得到返回结果

### Zookeeper服务列表获取更新

1. 服务容器负责启动，加载，运行服务提供者。

2. 服务提供者在启动时，向注册中心注册自己提供的服务。

3. 服务消费者在启动时，向注册中心订阅自己所需的服务。

4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心

### 分布式一致性协议 Raft

Raft协议核心：leader选举算法和数据一致性策略，详细参考下面的博文

Zookeeper实现就是采用的raft协议。

Raft协议 参考博文：<https://www.jianshu.com/p/aa77c8f4cb5c>

Zookeeper raft 参考博文：<https://blog.csdn.net/lxlmycsdnfree/article/details/78984752>

## 工具专题

### Maven 冲突解决方法

简答：使用idea maven helper插件查找出冲突jar包，再排除掉exclude不要的jar包。

Idea 插件解决参考文档：<https://blog.csdn.net/fnwibwj/article/details/81709733>

Maven 命令解决参考文档：<https://www.cnblogs.com/zdd-java/p/9760555.html>

### 压力测试Jmeter

Apache jmeter ------------后续补充

参考博文：<https://mp.weixin.qq.com/s/HhVItvBfmuUtDyfwbDcwpA>

## 实践专题

### 服务器配置



### 服务部署需要多少台服务器及其配置？

假设我们把一个系统划分为n个服务，此时需要部署n个应用程序。为了保证高可用，每个服务至少部署两个实例，防止其中一个节点挂了无法对外提供服务的问题。这时系统需要部署2n个应用程序。如下：

|  |
| --- |
| 总服务数=n  每个服务实例数=2（注：这里考虑高可用部署2台，实际还要根据负载决定部署多少台）  系统总应用个数=2\*n  每个应用程序运行内存=2G |

**服务器硬件配置**

内存=8G

CPU=4核

硬盘100G

如果每个应用需要2g来运行，一台8G的服务器大概可以部署2-3个应用，多余的内存需要供操作系统使用。

假如一个系统有11个服务，则所需物理服务器台数：

11（服务数）\*2（每个服务的实例数）/3（每台机器可部署应用）=8台

这里8台只是应用部署的台数，实际上zookeeper、mysql、redis都没算进去。

Zookeeper占用很少的资源可以与其他应用一起部署。

### Solr部署？

简答：我们公司采用solrCould模式部署solr集群，solrCloud支持故障自动切换，由于数据还不是很大，集群部署3个zookeeper节点和两个solr节点，两个solr节点互备，允许一台节点挂机后，另外一台还能继续对外提供服务。两台solr节点服务器硬件配置：4核，16GB。

参考文档:

主从模式和solrCloud区别：<https://blog.csdn.net/jiangchao858/article/details/53363310>

solrcloud 架构：<https://blog.csdn.net/zengxiaosen/article/details/78790253>

<https://blog.csdn.net/xiatianba/article/details/84630025>

部署方案：<https://www.jianshu.com/p/751129f6b4ed>

solrCloud集群搭建：<https://blog.csdn.net/yougoule/article/details/78445759>

### RabbitMQ是如何部署的？

简答：为了节约资源确保rabbitmq的高可用，我们公司使用两台阿里云服务器部署rabbitmq，服务器配置为：4核，8GB。我们采用镜像集群模式部署rabbitmq，为了使数据不丢失，两个节点都设置为磁盘节点。

节点说明：

内存节点将所有的队列，交换器，绑定关系，用户，权限，和vhost的元数据信息保存在内存中。磁盘节点将这些信息保存在磁盘中，但是内存节点的性能更高，为了保证集群的高可用性，必须保证集群中有两个以上的磁盘节点，来保证当有一个磁盘节点崩溃了，集群还能对外提供访问服务。

### Mysql部署

要求:先了解一下mysql架构，请看《Mysql架构及原理》

说明：我们学习时安装的mysql不具有高可用性，如果这个节点挂了是无法对外提供服务的，在企业中需要保证高可用。

简答：数据量不大时，可以按mysql主从架构方式部署两个实例。一个实例作为master节点，master节点对外提供读写服务。一个作为slave节点，从master节点复制数据，主要用于备灾。这种架构下，如果主节点出问题，只能手动升级slave节点为master继续对外提供服务。这里需要两台服务器部署这两个节点，使用阿里云服务器，每台服务器配置为：4核8G。

注意：4核cpu基本能够满足mysql的计算要求无须再大，内存是越大越好。根据应用大小及访问量适当调低调高配置。

### Redis部署

简答：我们公司redis存储数据还不是很大（大概8G左右），我们采用主从加哨兵模式保证redis服务的高可用。整个集群1个主redis，1个从redis，3个哨兵（至少要3个哨兵才能保证高可用，否则无法选举leader，无法实现主从切换）。可允许一台哨兵或redis实例挂掉。使用阿里云服务器，每台机器配置：16G 内存+ 4 核 CPU（没问配置可不必说机器具体配置）

注意：其中两台哨兵可以跟redis一起部署，另外一台可部署在其他服务器上，哨兵对服务器要求不高。

### 缓存击穿、缓存雪崩、缓存穿透？

参考附件《Redis》--------缓存击穿、缓存雪崩、缓存穿透----部分

草稿

如何保证接口的安全性

For 和foreach

Redis集群没有使用一致性hash,而是引入了哈希槽的概念，Redis集群有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，集群的每个节点负责一部分hash槽。

Return finally

高并发性能优化

为什么要用这个技术

解决什么问题

请求量有多少

有没有测试过

并发量多少

如何保证接口安全

Maven冲突解决

## 基本类型包装类

1. 基本类型包装类什么时候自动装箱和自动拆箱？或调用什么方法可以自动装箱和拆箱。

Integer  i = 1；

i += 1；

（1）其中Integer i =1；做了自动装箱（ 使用valueOf（）方法，int ---> Integer ）

（2）其中i +=1；先将Integer类型的 i 自动拆箱成 int（使用intValue（）方法，Integer--->int ），完成加法运行之后的 i 再装箱成Integer类型。

自动装箱：valueOf ，new Integer(1)

自动拆箱：intValue（）