# JAVA面试高频汇总

## 第一节 BATJ一二面面试题

### MYHB1：

1. Java容器有哪些？哪些是同步容器，哪些是并发容器？
2. ArrayList和LinkedList的插入和访问时间复杂度
3. Java反射原理，注解原理？
4. 新生代分几个区？使用什么算法进行垃圾回收？为什么使用这个算法？
5. HashMap在什么情况下会扩容，或者有哪些操作会导致扩容？
6. HashMap push的执行过程？
7. HashMap检测到hash冲突后，将元素插入在链表的末尾还是开头？
8. 1.8还采用了红黑树，讲讲红黑树的特性，为什么人家一定要用红黑树而不是AVL、B树之类的？
9. https和http区别，有没有用过其他安全传输手段？
10. 线程池的工作原理，几个重要参数，然后具体几个参数分析线程池会怎么做，最后阻塞队列的作用是什么？
11. Linux怎么查看系统负载情况？
12. 请详细描述SpringMVC处理请求全流程？
13. Spring一个bean装配的过程？
14. 讲一讲AtomicInteger，为什么要用CAS而不是Synchronized？

### MYHB2：

1. 查询中哪些情况不会使用索引？

2. 数据库索引，底层是怎样实现的，为什么要用B树索引？

3. Mysql主从同步的实现原理？

4. MySQL是怎么用B+树？

5. 谈谈数据库乐观锁与悲观锁?

6. 有使用过哪些NoSQL数据库？MongoDB和Redis适用哪些场景？

7. 描述分布式事务之TCC服务设计？

8. Redis和memcache有什么区别？Redis为什么比memcache有优势？

9. 考虑redis的时候，有没有考虑容量？大概数据量会有多少？

### MYHB3：

1. 介绍项目

2. 各种聊项目，从项目的架构设计到部署流程。

3. solr和mongodb的区别，存数据为什么不用solr？

4. 还是要聊项目

5. 谈谈分布式锁、以及分布式全局唯一ID的实现比较？

6. 集群监控的时候，重点需要关注哪些技术指标？这些指标如何优化？

7. 从千万的数据到亿级的数据，会面临哪些技术挑战？你的技术解决思路？

8. 最近两年遇到的最大的挫折，从挫折中学到了什么？

9. 最近有没有学习过新技术？

10. 有啥想问我的？

### MT：

1、最近做的比较熟悉的项目是哪个？画一下项目技术架构图

2、JVM老年代和新生代的比例？

3、YGC和FGC发生的具体场景

4、jstack，jmap，jutil分别的意义？如何线上排查JVM的相关问题？

5、线程池的构造类的方法的5个参数的具体意义？

6、单机上一个线程池正在处理服务如果忽然断电该怎么办？（正在处理和阻塞队列里的请求怎么处理）？

7、使用无界阻塞队列会出现什么问题？

8、接口如何处理重复请求？

9、具体处理方案是什么？

10、如何保证共享变量修改时的原子性？

设计一个对外服务的接口实现类，在1,2,3这三个主机（对应不同IP）上实现负载均衡和顺序轮询机制（考虑并发）

### BD：

1. 介绍一个集合框架
2. HashMap HashTable底层实现区别？HashTable和ConcurrentHashTable呢？
3. HashMap和TreeMap什么区别？底层数据结构是什么？
4. 线程池用过吗？都有什么参数？底层如何实现？
5. Synchronized和Lock什么区别？Synchronized什么情况是对象锁？什么时候是全局锁？为什么？
6. ThreadLocal是什么？底层如何实现？写一个例子呗？
7. Volatile的工作原理？
8. CAS是什么，如何实现？
9. 请至少用四种写法写一个单例模式？
10. 请介绍一下JVM内存模型？用过什么垃圾回收器？描述一下回收机制
11. 线上发生频繁FullGC如何处理？CPU使用率过高怎么办？
12. 如何定位问题？如何解决？说一下解决思路和处理方法？
13. 知道字节码吗？字节码都有哪些？Integer x= 5，int y=5，比较x==y都经过哪些步骤？
14. 讲讲类加载机制呗，都有哪些类加载器，这些加载器都加载哪些文件？
15. 手写一下类加载Demo？
16. 知道OSGI吗？他是如何实现的？
17. 请问你做过哪些JVM优化？使用什么方法达到什么效果？
18. classForName(“java.lang.String”)和String classgetClassLoader() LoadClass(“java.lang.String”)什么区别呢？

### JRTT：

第一轮：

1. HashMap如果一直put元素会怎么一样？HashCode全都相同如何？equals 方法都相同如何？
2. ApplicationContext的初始化过程？初始化过程中发现循环依赖Spring是如何处理？
3. GC用什么收集器？收集过程如何？哪些部分可以作为GCRoots？
4. Volatile关键字，指令重排序有什么意义？Synchronized怎么用？
5. 并发包里的原子类有哪些？怎么实现？Cas在CPU级别用什么指令实现的？
6. Redis数据结构有哪些？如何实现sorted set？这种数据结构在极端情况下有什么问题？二叉平衡树？
7. 系统设计题：一个推送场景，50 条内容，定时推送，先推 5%用户，一段时间后再找出效果最好的几条，推给所有用户。设计相关库表，系统模块, 需要可以落地，有伪代码
8. MySql 索引是什么数据结构？B tree 有什么特点？优点是什么？
9. 慢查询怎么优化？
10. 项目：cache 设计，MQ 丢失消息，RPC 使用场景。各部分职责，有哪些优化点。
11. ThreadLocal 使用场景
12. Redis 回收和备份
13. 项目设计问题——主要是针对项目找漏洞，然后给出优化方案
14. 遇到的坑，怎么解决，后续总结。

第二轮：

1. 那些问题对你挑战最大？

2. JDK 1.8 ConcurrentHashMap 做过什么改进？HashMap 死锁？

3. 标记的时候怎么找出栈上的 GC Root？说出一种可能的方案，存在什么问题？

4. 字节码有了解过吗？

5. 本地缓存怎么优化空间？（提出 BitMap）BitMap 可行吗？怎么验证可行性？如果不可行，怎么证伪？

6. 其他语言有了解过吗？Scala 的集合有什么特性？python 有什么高级特性？

7. 怎么学习新技术？哪些是基础技术？

8. 在部门中是什么角色定位？

9. JDK 1.8 比前一个版本有哪些改进？

1. ParNew 收集过程，如何调优 ParNew？如何减少 full gc？调大 YoungGen 有什么好处坏处？

11. OpenAPI

1. 平衡二叉树

13. 业务逻辑

14. MySQL

15. 服务设计题

16. web 架构设计

17. Pack

18. http 协议

19. 爬虫

20. Java Agent

21. Zookeeper

22. LoadBalance

23. 分布式 ID

24. Nginx

25. Cookie

26. Session

27. GC

28. 手写 LRU 缓存

29. 将现有线性存储结构改成环形结构

30. Redis IO 模型

31. MQ 架构对比

32. 各种缓存 memcache，MongoDB，EhCache，Guava，Caffeine 等等，使用场景

33. 数据库乐观锁悲观锁。在 JDK 和其他中间件的体现。

34. 线程池，核心参数，扩容原理，使用注意点

35. 数据库主从同步，延迟

36. 数据库和缓存一致性问题，出现的场景，解决方案

37. Spring IOC AOP

38. 一致性哈希算法

39. RPC 服务发现与注册

1. 服务提供方不稳定，频繁变动如何提升自身稳定性

### JDJR：

一面（主要是基础面）

自我介绍，主要问了做了什么和擅长什么

@Autowired的实现原理

Bean的默认作用范围是什么？其他的作用范围？

索引是什么概念有什么作用？

MySQL里主要有哪些索引结构？哈希索引和B+树索引比较？

说说Java线程池的原理？线程池有哪些？线程池工厂有哪些线程池类型，及其线程池参数是什么？

hashmap原理，处理哈希冲突用的哪种方法？

BIO、NIO、AIO的区别和联系？

jvm内存模型jmm 知道的全讲讲

讲讲Java GC机制？

Java怎么进行垃圾回收的？什么对象会进老年代？

垃圾回收算法有哪些？为什么新生代使用复制算法？

HashMap的时间复杂度？HashMap中Hash冲突是怎么解决的？Java8中的HashMap有什么变化？

红黑树需要比较大小才能进行插入，是依据什么进行比较的？其他Hash冲突解决方式？

hash和B+树的区别？分别应用于什么场景？哪个比较好？

二面（数据库问的比较多）

自我介绍

为什么MyISAM查询性能好？

说说事务特性（ACID）？

mysql数据库默认存储引擎，有什么优点

MySQL的事务隔离级别，分别解决什么问题。

SQL慢查询的常见优化步骤是什么？

说下乐观锁，悲观锁（select for update），并写出sql实现？

讲讲TCP协议的三次握手和四次挥手过程？

用到过哪些rpc框架？

Java web过滤器的生命周期？

三面（综合面主要分布式及缓存）

自我介绍，讲下项目！

加锁有什么机制？

数据库水平切分，垂直切分的设计思路和切分顺序

Redis如何解决key冲突？

如何保证数据库与redis缓存一致的

项目中消息队列怎么用的？使用哪些具体业务场景？

JVM相关的分析工具有使用过哪些？具体的性能调优步骤吗？

MySQL的慢sql优化一般如何来做？除此外还有什么方法优化？

线上的服务器监控指标，你认为哪些指标是最需要关注的？为什么？

soa和微服务的区别？

单机系统演变为分布式系统，会涉及到哪些技术的调整？请从前面负载到后端详细描述。

设计一个秒杀系统？

### JDJR：

1. Dubbo超时重试？Dubbo超时时间设置？
2. 如何保障请求执行顺序？
3. 分布式事务与分布式锁？（扣款不要出现负数）
4. 分布式session设置
5. 执行某操作，前50次成功，第51次a全部回滚，b前50次提交，第51次抛异常，ab场景分别如何设置Spring（事务传播性）？
6. Zookeeper有哪些作用？
7. JVM内存模型？
8. 数据库垂直和水平拆分？
9. Mybatis如何分页？如何设置缓存；MySQL分页？
10. 熟悉IO吗？与NIO的区别，阻塞与非阻塞的区别？
11. 分布式session一致性
12. 分布式接口的幂等性设计「不能重复扣款」

### DD：

1、自我介绍，技术特点

2、兴趣是什么，优势是什么

3、jvm，jre以及jdk三者之间的关系？

4、Dubbo的底层原理，Zookeeper是什么

5、cincurrentMap的机制；TreeMap；Volatil关键字

6、快速排序；广度优先搜索（队列实现）

7、缓存的雪崩以及穿透的理解？

8、HashMap的key可以重复吗？

9、synchronized和lock的区别？

1. 开发一个大型网站你会考虑哪些问题?

通过面试题来看，可以看出目前互联网公司面试考点为：

1.性能调优、算法数据机构

2.高并发下数据安全、接口冪等性、原子性等

3.分布式下协同、已经锁的处理

4.数据库的分库分表、项目之间的垂直拆分

出现频率高的技术点有：

1.HashMap

2.JVM

3.Dubbo

4.Mybatis

5.Zookeeper

6.http tcp/ip

下面分享一位面试阿里P6时他简历上填写的内容，看看你离P6还有多远？

### AL：

1、开发中Java用了比较多的数据结构有哪些？

2谈谈你对HashMap的理解，底层原理的基本实现，HashMap怎么解决碰撞问题的？

这些数据结构中是线程安全的吗？假如你回答HashMap是线程安全的，接着问你有没有线程安全的map，接下来问了conurren包。

3、对JVM熟不熟悉？简单说说类加载过程，里面执行的哪些操作?问了GC和内存管理，平时在tomect里面有没有进行过相关的配置

4、然后问了http协议，get和post的基本区别，接着tcp/ip协议，三次握手，窗口滑动机制。

5、开发中用了那些数据库？回答mysql，储存引擎有哪些？然后问了我悲观锁和乐观锁问题使用场景、分布式集群实现的原理。

1. 然后问了我springmvc和mybatis的工作原理，有没有看过底层源码？

1. 简短自我介绍

2. 事务的ACID，其中把事务的隔离性详细解释一遍

3. 脏读、幻影读、不可重复读

4. 红黑树、二叉树的算法

5. 平常用到哪些集合类？ArrayList和LinkedList区别？HashMap内部数据结构？ConcurrentHashMap分段锁？

6. jdk1.8中，对hashMap和concurrentHashMap做了哪些优化

7. 如何解决hash冲突的，以及如果冲突了，怎么在hash表中找到目标值

8. synchronized 和 ReentranLock的区别？

9. ThreadLocal？应用场景？

10. Java GC机制？GC Roots有哪些？

11. MySQL行锁是否会有死锁的情况？

二面

1. 乐观锁和悲观锁了解吗？JDK中涉及到乐观锁和悲观锁的内容？

2. Nginx负载均衡策略？

3. Nginx和其他负载均衡框架对比过吗？

4. Redis是单线程？

5. Redis高并发快的原因？

6. 如何利用Redis处理热点数据

7. 谈谈Redis哨兵、复制、集群

8. 工作中技术优化过哪些？JVM、MySQL、代码等都谈谈

三面

1. Spring Cloud用到什么东西？如何实现负载均衡？服务挂了注册中心怎么判断？

2. 网络编程nio和netty相关，netty的线程模型，零拷贝实现

3. 分布式锁的实现你知道的有哪些？具体详细谈一种实现方式

4. 高并发的应用场景，技术需要涉及到哪些？怎样来架构设计？

5. 接着高并发的问题，谈到了秒杀等的技术应用：kafka、redis、mycat等

6. 最后谈谈你参与过的项目，技术含量比较高的，相关的架构设计以及你负责哪些核心编码

最新美团面试题目，技术主要是3面，重点问了：

1.Java容器的问题：hashmap、currenthashmap等，建议面试前把这几个问到最多的类的原理、到底层数据结构、再到数据扩容，以及算法复杂度，都需要重点掌握。

2.Java相关的：线程锁、以及线程流转图、线程池等。

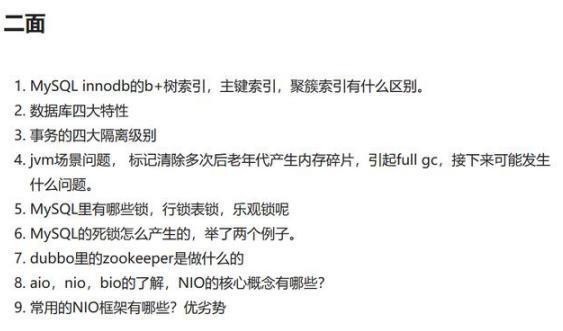
3.JVM不用说了，每次都考，如果对内存的回收，垃圾回收器的种类区别，回收算法机制这个必须要掌握。

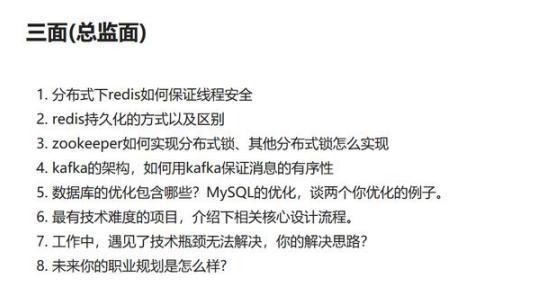
4.数据库MySQL相关的，这个也是每次必问，毕竟是平时工作中使用最多的，考察数据库基本功：存储引擎、SQL查询优化、常见索引的使用和区别、事务的使用，表范式设计，以及分库分表的策略和实际应用等。

5.分布式也是必考的系列，问得最多就是Redis、Dubbo等，面试官主要就是考察缓存、RPC的实际使用情况。

1. 如果面试前，还能对常见的高并发的场景，以及技术方案熟悉，那面试会好很多。







### PDD：

年底跳槽挂了阿里和美团，却收获拼多多Java岗offer,分享三面总结

一、项目方面

1. 首先上来简单做一下自我介绍。然后让介绍简历里的项目。

2. 说下项目里的难点，技术架构。

3. 平时开发过程中都遇到过哪些难题？

4. 平时都这么和同事沟通？

二、java基础方面。

1. 首先还是老生常谈的hashmap。hashmap的set和get的时间复杂度是多少？

2. 为什么是O(1), hashmap 在jdk1.8是线程安全的吗？

3. 为什么是线程安全的？

4. concureenthashmap了解吗？他是如何实现线程安全的？

5. 说说java泛型，为什么称java泛型为伪泛型？

6. 泛型的好处有哪些？int可以作为泛型类型吗？

7. 说一说静态代理和动态代理的区别？

8. 动态代理的底层是怎么实现的？

三、多线程方面

1. 线程run和start的区别？

2. 两次start同一个线程会怎么样？

3. 说一下java的线程模型？

4. 说一下violate关键字？他的原理是什么？violate是线程安全的吗？

5. 说一下violate和synchronized这两个关键字的区别？

6. 说下线程池的原理？有没有在项目中用过线程池？怎么用的？

7. 核心线程和非核心线程有什么区别？你在项目里怎么确定核心线程数的。

8. 对concureent包了解吗？

9. 什么是cas？cas怎么解决ABA问题？

10. 讲一下CountDownLatch和cyclicBarrier的区别？

11. 说一下java类加载器的工作机制？

12. 讲一下JVM相关。有没有做货GC调优，讲一下这么做的？

四、spring框架

1. 说一下springboot的启动过程？

2. 平时开发中都用哪些注解？

3. IOC和AOP原理讲一下

五、数据库与java中间件

1. redis常用的数据结构有哪几种，在你的项目中用过哪几种，以及在业务中使用的场景？

2. redis cluster有没有了解过，怎么做到高可用的？

3. redis集群和哨兵机制有什么区别？

4. redis的持久化机制了解吗？你们在项目中是怎么做持久化的？

5. 遇到过redis的hotkey吗？怎么处理的？

6. redis是单线程的吗？单线程为什么还这么快？

7. redis实现分布式锁以及和zk实现的性能和稳定性的对比？

8. kafka的原理？怎么保证消息不丢失？

9. 索引的常见实现方式有哪些，有哪些区别?

10. MySQL的存储引擎有哪些，有哪些区别？

11. InnoDB使用的是什么方式实现索引，怎么实现的？

12. 说下聚簇索引和非聚簇索引的区别？

六、手写算法方面

一面手写的题是leetcode 106. 从中序与后序遍历序列构造二叉树

根据一棵树的中序遍历与后序遍历构造二叉树。

注意:

你可以假设树中没有重复的元素。

例如，给出

中序遍历 inorder = [9,3,15,20,7]

后序遍历 postorder = [9,15,7,20,3]

返回如下的二叉树：

3

/\

920

/\

157

二面手写的题是 leetcode 86. 分隔链表

给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。

你应当保留两个分区中每个节点的初始相对位置。

### 示例:

输入: head = 1->4->3->2->5->2, x = 3

输出: 1->2->2->4->3->5

|  |
| --- |
| 一面(20分钟)  晚7点，因为想到下周一才面试，我刚准备出去打个羽毛球，北京的电话就来了。面试官各种抱歉，说开会拖延了。  1、自我介绍  说了很多遍了，很流畅捡重点介绍完。  2、问我数据结构算法好不好  挺好的（其实心还是有点虚，不过最近刷了很多题也只能壮着胆子充胖子了） 。  3、找到单链表的三等分点，如果单链表是有环的呢  用快慢指针，一个走一步，一个走三步。如果有环，先判断环在哪，找最后一个节点，然后用之前的无环的做法。  4、讲一下项目的架构  我重点讲了MVC。  5、说一下你熟悉的设计模式  我重点讲了单例、工厂方法、代理 。  6、有没有配置过服务器啥啥啥  这个我真不知道，都没听过呢，只能诚实说没有，毕竟都没法扯。  一面挺匆忙的，我估计面试官刚开完会还没吃饭呢。他说让我等，可能再找一个同事面我，可能就直接告诉我结果了。从一面面试官的声音和口吻，我判断他一定是个部门老大，问的设计偏多，后面hr告诉我他就是我要去的部门的老大。哈哈。  二面(60分钟)  面完一面正准备出去打羽毛球，北京的电话又来了。（注定这周五参加不了球队活动了！）  二面：跟一面比起来，二面面试官的声音听起来就像是搞技术开发的，果不其然，一个小时的纯技术电话面试！面的特别全面！  1、Spring：有没有用过Spring，Spring IOC、AOP机制与实现，Spring MVC  其实我挺不想被问到Spring的细节的，框架这些我都没有复习不太记得了。所以我对面试官说Spring里面的一些比较重要的机制我理解的还不错，然后我用一个实际的例子把我对IOC、AOP理解讲了一下，他听了说对，理解的不错（难得遇到一个边面试边能给反馈的面试官，好开心）。  Spring MVC其实我用过，我就对面试官讲了我的项目中用到的Servlet，jsp和javabean实现的MVC，以及MVC各个模块职责以及每个模块是怎么联系到一起的，最后我补充了一句我想SpringMVC的思想其实跟这个是一样的（他说对的，嘿嘿有反馈真好） 。  2、多线程：怎么实现线程安全，各个实现方法有什么区别，volatile关键字的使用，可重入锁的理解，Synchronized是不是可重入锁  这里我就主要讲了Synchronized关键字，还有并发包下面的一些锁，以及各自的优缺点和区别。volatile关键字我主要从可见性、原子性和禁止JVM指令重排序三个方面讲的，再讲了一下我在多线程的单例模式double-check中用到volatile关键字禁止JVM指令重排优化。  3、集合：HashMap底层实现，怎么实现HashMap线程安全  我讲了一下HashMap底层是数组加单链表实现，Node内部类，add的过程，Hash冲突解决办法，扩容，三种集合视图。HashMap线程安全的实现方式主要讲了HashTable、ConcurrentHashMap以及Collections中的静态方法SynchronizedMap可以对HashMap进行封装。以及这三种方式的区别，效率表现。  4、JVM内存管理，GC算法，HotSpot里面的垃圾回收器、类加载  JVM内存主要分为五个区，哪些是线程共享的，哪些是线程独享的，每个区存放什么。GC方面：怎么判断哪些对象需要被GC，GC的方法，Minor GC与Full GC。HotSpot GC算法以及7种垃圾回收期，主要讲了CMS和G1收集器。类加载：类加载的过程，Bootstrap classloader-ExtClassloader-AppClassloader，父类委托机制。  5、进程和线程的区别  从调度、并发性、拥有的资源和系统开销四个方面回答的。  6、HTTP有没有状态，我说无状态，怎么解决HTTP无状态  怎么解决HTTP无状态其实就是怎么进行会话跟踪，有四种方法：URL重写、隐藏表单域、Cookie、Session。  7、Java IO，NIO，Java中有没有实现异步IO  Java IO实现的是同步阻塞，它是怎么实现同步阻塞的。我拿了read()方法举例来讲的。NIO实现的是同步非阻塞，我详细讲了一下Selector中的select()方法轮询说明它是如何实现多路复用IO的。然后对比了一下他们的效率。面试官可能看我对这一块比较了解，又继续问我Java中有没有实现异步IO，我感觉好像没有，但面试官说有，让我想想，其实这里我并不清楚啦，所以我就对面试官讲了一下我对Unix中异步IO模型的理解，然后说至于Java里面有没有我真的不太清楚。（他居然笑了！说你理解是对的，Java里面有没有不重要！哈哈）  8、前端会不会，Ajax是什么，Ajax实现原理  前端我只是会用一些js而已，用过jquery框架，问我Ajax全称是啥，我猜是异步的js和xml。Ajax实现原理其实我也不懂，我就只简单讲了一下它通过XMLHttpRequest对象进行异步查询，Ajax引擎在客户端运行，减少了服务器工作量。  9、让我设计一个线程池  因为我简历中有写到我对多线程、并发这一块理解比较好。所以他老问这方面的题。这个问题因为我之前看过ThreadPoolExecutor的源代码，所以我就仿照那个类的设计思路来想的，详细讲了一下核心池、创建线程可以用工厂方法模式来进行设计、线程池状态、阻塞队列、拒绝策略这几个方面。设计的还算比较周全。  10、讲几个设计模式，哪些地方用到了，为什么要用  单例模式，jdk中的getRuntime()；工厂方法模式，ThreadPoolExcutor用到ThreadFactory；观察者模式：java.util包下面的Observable和Observer。最后主要讲了一下工厂方法模式的使用场景。  11、Mysql优化、索引的实现  我从数据库设计优化和查询优化两方面讲的。索引B+树实现，InnoDB和MyISAM主键索引的实现区别，一个聚集一个非聚集。  12、事务的隔离级别  四种隔离级别，可能会出现哪些异常，mysql中默认级别。  13、有没有用过Hibernate、mybatis、git  这个简单讲一下就好，分别是干什么的。  14、Linux  我说这个本科学过，但是很久没用，命令忘光了。他说没事，考你几个简单的：cd、ls、dir（真的是简单的）。  15、算法题  从10万个数中找最小的10个，时间复杂度分析（最大堆，考虑内存） 。  从一个有正有负数组中找连续子数组的最大和，时间复杂度分析（动态规划）  满二叉树第i层有多少个节点，n层的满二叉树共有多少个节点。  终于到我提问环节了：  1、你们是什么部门（他说是核心部门，大数据研发） 。  2、我对高并发和负载均衡挺有兴趣的，但是我平时在学校也没有这个环境让我在这方面有所体验，那你建议我目前可以怎么学呢（他说这确实是不太好学，只能看些理论和别人的博客，以后工作中才能慢慢学） 。  3、中间件具体是做什么的，是解决高并发和负载均衡吗（他说差不多是的，然后他说我们这个部门不是中间件，是大数据部门啊，我说恩我知道） 。  最后没啥问题了，他让我保持电话畅通。  这一面面完，口干舌燥，我一度怀疑他可能不知道我是在应聘实习生的岗位。有太多要总结的了，放在总结的地方一起讲吧。  三面（25分钟）  面试官说是他是另外一个部门的，需要进行交叉面试。  1、MySql优化 。  2、说下项目做了些什么，架构之类的。  3、在collabedit上在线写代码，题目很简单是编程之美上的原题，一个有序的整数数组，输出两个数，使它们的和为某个给定的值。之前做过很快写好，然后给他讲思路。他继续问如果数组无序怎么办，先排序。  4、两个文件，每个文件中都有若干个url，找出两个文件中相同的url（用HashMap）  这一面挺简单的，只是增加之前面试没有过的在线写代码环节，collabedit后来我才了解，像facebook一些互联网公司远程面试都会用这个在线编辑器写代码，就是文本文档写，没有提示，不能编译运行，跟白板写一样。平时练练手就好。  HR四面（30分钟）  三面面试官说他那就是终面，说我过了等hr联系我。万万没想到半小时后的hr面居然也是技术。  1、自我介绍，都四面了还自我介绍？！我还以为是单纯的hr面，所以介绍的都是我的性格和生活方面的，结果并不是。  2、问项目，问的特别特别细，技术细节，还有遇到什么问题，怎么解决的，做项目有没有人带，怎么跟别人沟通的。  3、数据库优化，如果数据库一个表特别大怎么办  数据库优化我就讲了之前讲过很多遍的点，他问一个表特别大怎么办：大表分小表，怎么实现：使用分区表。  4、问研究生的科研题目，为什么选这个题，看了一些什么论文（细到问我那些论文是发表在什么期刊上的，作者是哪所学校的），为什么要选这个算法，怎么优化的，实验结果怎么跟别人作对比的，为什么比别人的算法好（一个个问题不断砸过来，我猜我说的那些专业名词他应该不太懂，只是判断一下是不是我做的而已。。）  5、确定实习时间  这一面确实让我感到有压迫感，项目是本科做的，挺久了，一些技术细节上也没太总结，所以问细了我只能连想带编，嘿嘿。科研方面倒还好，上个学期都在弄这个，一些算法的实现和改进、对比都还记得比较清楚，回答的挺流畅的，可能这真的就是技术型的hr面吧。  好累，我不想再找实习了,不得不说美团的实习生面试跟正式校招拿offer一样隆重，被面四面下来，好累，这个部门也挺好我很满意，所以我不想再继续找了。到目前为止，Java基础、数据库、计算机网络、操作系统复习也都全面结束了。现在每天刷刷题，保持就好。接下来准备看看Tomcat源码，要进入新的阶段了，这里不是终点，加油。 |

### MYJF1：

自我介绍

项目情况：

对你来说影响最大的一个项目（该面试中有关项目问题都针对该项目展开）？

为什么会想做这个项目？这个项目的ideal是谁提出来的？

项目中如何实现的大数据的传输和存储

项目中哪一部分最难攻克？如何攻克？

基础知识考察：

模块化的好处

Htttp协议

hashmap和concurrenthashmap区别及两者的优缺点

对MySQL的了解，和oracle的区别

对设计模式的看法和认知

有哪些设计模式

如何实现分布式缓存

多线程如何避免死锁

关于树的算法题-二叉树的锯齿形层次遍历：http://www.lintcode.com/zh-cn/problem/binary-tree-zigzag-level-order-traversal/

Java的垃圾回收机制

对Runtime的了解

电面过程中非常注重基础知识的考察，面试前务必对基础知识内容进行复习和梳理。基础知识考察的内容一般会围绕项目内容进行展开，在前期对项目介绍进行准备时需适当换位，思考面试官的提问逻辑，避免给自己设下陷阱。

### MYJF2：

背景了解：

为什么要选择编程这条路

何时开始编程

如何进行自学

阅读过那些书籍

项目考察

项目运行过程中成员是否曾就某一点发生争执？作为Leader你是如何解决的？具体事例？

项目最终实现效果

项目具体部分使用的设计模式并简述选择理由

项目有何需改进之处并初拟改进方案

第一个项目：

第二个项目：

基础知识考察：

数据库的范式

JVM内存模型及调优

浏览器的缓存机制

如何解决高并发问题？是否进行过相应程序的编写？

LintCode 算法题 - 最小子串覆盖。原题链接：http://www.lintcode.com/zh-cn/problem/minimum-window-substring/

想问的问题

技术面中项目占了相当一部分时间，项目中从技术到个人团队领导能力，从完成执行能力到思想灵活度，都是面试官希望得到的信息。因此，必须对自己的项目多方面展开准备，项目回顾的过程中需不断思考改进方案和具体措施。

算法题部分由于之前练习时做过且在 www.jiuzhang.com/solution 上研究过参考答案，此次应对较为轻松，面试官反应较好。在算法题考查过程中，正确率和代码效率都是面试官关注的重点，因此在Lintcode上刷题时必须强调每一个细节的改进，不能仅安于正确。上述参考网站中的标准答案工业风较强，适合实际工作，建议在刷题过程中充分利用。此外，面试前做两道算法题练手可使面试时写算法更为娴熟，如时间富裕可尝试。

### MYJF1：

1、自我介绍、自己做的项目和技术领域

2、项目中的监控：那个监控指标常见的有哪些？

3、微服务涉及到的技术以及需要注意的问题有哪些？

4、注册中心你了解了哪些？

5、consul 的可靠性你了解吗？

6、consul 的机制你有没有具体深入过？有没有和其他的注册中心对比过？

7、项目用 Spring 比较多，有没有了解 Spring 的原理？AOP 和 IOC 的原理

8、Spring Boot除了自动配置，相比传统的 Spring 有什么其他的区别？

9、Spring Cloud 有了解多少？

10、Spring Bean 的生命周期

11、HashMap 和 hashTable 区别？

12、Object 的 hashcode 方法重写了，equals 方法要不要改？

13、Hashmap 线程不安全的出现场景

14、线上服务 CPU 很高该怎么做？有哪些措施可以找到问题

15、JDK 中有哪几个线程池？顺带把线程池讲了个遍

16、SQL 优化的常见方法有哪些

17、SQL 索引的顺序，字段的顺序

18、查看 SQL 是不是使用了索引？（有什么工具）

19、TCP 和 UDP 的区别？TCP 数据传输过程中怎么做到可靠的？

20、说下你知道的排序算法吧

21、查找一个数组的中位数？

1. 你有什么问题想问我的吗？

### MJYF2：

1、自我介绍、工作经历、技术栈

2、项目中你学到了什么技术？（把三项目具体描述了很久）

3、微服务划分的粒度

4、微服务的高可用怎么保证的？

5、常用的负载均衡，该怎么用，你能说下吗？

6、网关能够为后端服务带来哪些好处？

7、Spring Bean 的生命周期

8、xml 中配置的 init、destroy 方法怎么可以做到调用具体的方法？

9、反射的机制

10、Object 类中的方法

11、hashcode 和 equals 方法常用地方

12、对象比较是否相同

13、hashmap put 方法存放的时候怎么判断是否是重复的

14、Object toString 方法常用的地方，为什么要重写该方法

15、Set 和 List 区别？

16、ArrayList 和 LinkedList 区别

17、如果存取相同的数据，ArrayList 和 LinkedList 谁占用空间更大？

18、Set 存的顺序是有序的吗？

19、常见 Set 的实现有哪些？

20、TreeSet 对存入对数据有什么要求呢？

21、HashSet 的底层实现呢

22、TreeSet 底层源码有看过吗？

23、HashSet 是不是线程安全的？为什么不是线程安全的？

24、Java 中有哪些线程安全的 Map？

25、Concurrenthashmap 是怎么做到线程安全的？

26、HashTable 你了解过吗？

27、如何保证线程安全问题？

28、synchronized、lock

29、volatile 的原子性问题？为什么 i++ 这种不支持原子性？从计算机原理的设计来讲下不能保证原子性的原因

30、happens before 原理

31、cas 操作

32、lock 和 synchronized 的区别？

33、公平锁和非公平锁

34、Java 读写锁

35、读写锁设计主要解决什么问题？

36、你项目除了写 Java 代码，还有前端代码，那你知道前端有哪些框架吗？

37、MySQL 分页查询语句

38、MySQL 事务特性和隔离级别

39、不可重复读会出现在什么场景？

40、sql having 的使用场景

41、前端浏览器地址的一个 http 请求到后端整个流程是怎么样？能够说下吗？

42、http 默认端口，https 默认端口

43、DNS 你知道是干嘛的吗？

44、你们开发用的 ide 是啥？你能说下 idea 的常用几个快捷键吧？

45、代码版本管理你们用的是啥？

46、git rebase 和 merge 有什么区别？

1. 你们公司加班多吗？

## 类型汇总

### JAVA基础/容器

1. Java容器有哪些？哪些是同步容器，哪些是并发容器？
2. ArrayList和LinkedList的插入和访问时间复杂度？
3. Java反射原理，注解原理？
4. HashMap在什么情况下会扩容，或者有哪些操作会导致扩容？
5. HashMap push的执行过程？
6. HashMap检测到hash冲突后，将元素插入在链表的末尾还是开头？
7. 1.8还采用了红黑树，讲讲红黑树的特性，为什么人家一定要用红黑树而不是AVL、B树之类的?
8. 介绍一个集合框架
9. HashMap HashTable底层实现区别？HashTable和ConcurrentHashTable呢？HashMap 和 hashTable 区别？
10. HashMap如果一直put元素会怎么一样？HashCode全都相同如何？equals 方法都相同如何？
11. JDK 1.8 ConcurrentHashMap 做过什么改进？HashMap 死锁？
12. hashmap原理，处理哈希冲突用的哪种方法？如何解决hash冲突的，以及如果冲突了，怎么在hash表中找到目标值
13. HashMap的时间复杂度？HashMap中Hash冲突是怎么解决的？Java8中的HashMap有什么变化？
14. 红黑树需要比较大小才能进行插入，是依据什么进行比较的？其他Hash冲突解决方式？
15. jvm，jre以及jdk三者之间的关系？
16. cincurrentMap的机制；TreeMap；Volatil关键字
17. 开发中Java用了比较多的数据结构有哪些？
18. 谈谈你对HashMap的理解，底层原理的基本实现，HashMap怎么解决碰撞问题的？
19. 这些数据结构中是线程安全的吗？假如你回答HashMap是线程安全的，接着问你有没有线程安全的map，接下来问了conurren包。
20. 平常用到哪些集合类？ArrayList和LinkedList区别？HashMap内部数据结构？ConcurrentHashMap分段锁？
21. jdk1.8中，对hashMap和concurrentHashMap做了哪些优化
22. Java容器的问题：hashmap、currenthashmap等，建议面试前把这几个问到最多的类的原理、到底层数据结构、再到数据扩容，以及算法复杂度，都需要重点掌握。
23. 集合框架知识，HashMap、ArrayList、LinkedList源码，与HashTable、ConcurrentHashMap相互关联和区别？
24. 首先还是老生常谈的hashmap。hashmap的set和get的时间复杂度是多少？
25. 为什么是O(1), hashmap 在jdk1.8是线程安全的吗？为什么是线程安全的？Hashmap 线程不安全的出现场景
26. concureenthashmap了解吗？他是如何实现线程安全的？Concurrenthashmap 是怎么做到线程安全的？
27. 说说java泛型，为什么称java泛型为伪泛型？
28. 泛型的好处有哪些？int可以作为泛型类型吗？
29. 线程run和start的区别？
30. 两次start同一个线程会怎么样？
31. 说一下java的线程模型？
32. HashMap底层实现，怎么实现HashMap线程安全
33. 进程和线程的区别？
34. 模块化的好处
35. 对Runtime的了解
36. Object 的 hashcode 方法重写了，equals 方法要不要改？
37. 反射的机制
38. Object 类中的方法
39. hashcode 和 equals 方法常用地方？对象比较是否相同
40. hashmap put 方法存放的时候怎么判断是否是重复的
41. Object toString 方法常用的地方，为什么要重写该方法
42. Set 和 List 区别？
43. ArrayList 和 LinkedList 区别？如果存取相同的数据，ArrayList 和 LinkedList 谁占用空间更大？
44. Set 存的顺序是有序的吗？HashSet 的底层实现呢？
45. 常见 Set 的实现有哪些？
46. TreeSet 对存入对数据有什么要求呢？TreeSet 底层源码有看过吗？
47. HashSet 是不是线程安全的？为什么不是线程安全的？HashTable 你了解过吗？

### Spring/SpringBoot

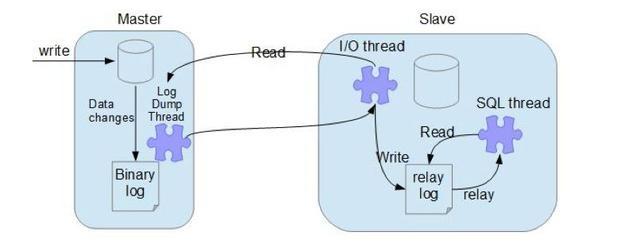
1. 详细描述SpringMVC处理请求全流程？
2. Spring一个bean装配的过程？Spring Bean 的生命周期；Spring Bean 的生命周期
3. 知道OSGI吗？他是如何实现的？SringDM动态模型
4. ApplicationContext的初始化过程？初始化过程中发现循环依赖Spring是如何处理？
5. @Autowired的实现原理
6. Bean的默认作用范围是什么？其他的作用范围？
7. Java web过滤器的生命周期？
8. 执行某操作，前50次成功，第51次a全部回滚，b前50次提交，第51次抛异常，ab场景分别如何设置Spring（事务传播性）？
9. 然后问了我springmvc和mybatis的工作原理，有没有看过底层源码？
10. 说一下springboot的启动过程？
11. 平时开发中都用哪些注解？
12. IOC和AOP原理讲一下
13. 有没有用过Spring，Spring IOC、AOP机制与实现，Spring MVC
14. Spring Boot除了自动配置，相比传统的 Spring 有什么其他的区别？
15. xml 中配置的 init、destroy 方法怎么可以做到调用具体的方法？

### MySQL数据库

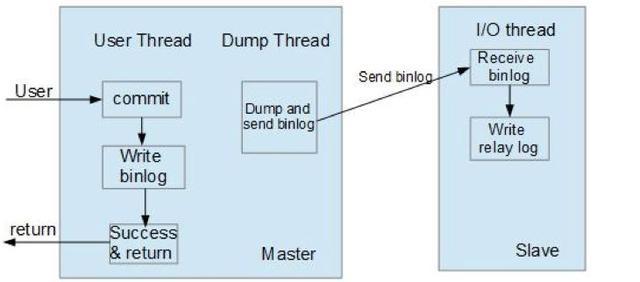
1. 查询中哪些情况不会使用索引？
2. 联合索引，最左匹配原则：A,B,C联合，A,(B+C)、A+B+C走索引，B+C不走索引；
3. Or条件两边都需要加索引；
4. 隐式转换，函数转换，不走索引；
5. Like ‘%A’不走索引；
6. 引擎优化对于连续排列，不走索引大于走索引时，不走索引；
7. Count（\*）必须匹配字段not null，否则不走索引，因为HashSet不存储键为nul；
8. 没有及时表分析，可能不走索引；
9. 不等于，not in/ not exist不走所用，可以使用join连接；
10. 数据库索引，底层是怎样实现的，为什么要用B树索引？7.MySql 索引是什么数据结构？B tree 有什么特点？优点是什么？索引是什么概念有什么作用？
11. 数据存储在磁盘上，从数据库读取数据的查询速度主要取决于磁盘IO的次数和速度；
12. 索引可以有效减少磁盘IO次数，提高读取磁盘一块相邻数据的速度，B数比二叉树同样层高的二叉树能更快查询到数据；
13. 局部性原理和磁盘预读：分页4k，内存发出缺页信号，一次IO加载局部数据到内存；
14. B+树的优势：1.单一节点存储更多元素，树变胖了，减少IO次数；2.数据存储在叶子节点，查询性能稳定；3.所有叶子节点形成有序链表，适合范围查找；

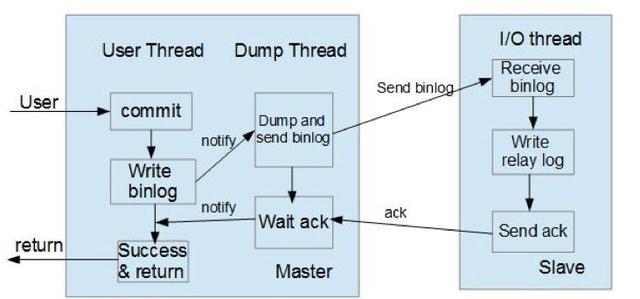
一般性情况，数据库的B+树的高度一般在2~4层，这就是说找到某一键值的行记录最多需要2到4次逻辑IO，相当于0.02到0.04s。

1. Mysql主从同步的实现原理？分布式集群实现的原理。
2. 主从复制三个线程：主节点（log dump thread）、从节点（IO thread，SQL thread）



1. 主从复制模式：异步模式：不等待从节点commit返回，直接提交事务；半同步模式：在异步的基础上，只要有一个从节点返回就提交事务；全同步：等待所有从节点返回commit，再提交事务。





1. 谈谈数据库乐观锁与悲观锁?说下乐观锁，悲观锁（select for update），并写出sql实现？乐观锁和悲观锁了解吗？JDK中涉及到乐观锁和悲观锁的内容？然后问了我悲观锁和乐观锁问题使用场景？MySQL里有哪些锁，行锁表锁，乐观锁呢？
2. 悲观锁：假定并发修改每时每刻都在发生，为了防止并发，更新前先将数据锁定起来，如果其他任务要更新，则需要等待，**synchronized**；
3. 乐观锁：假定并发修改不是每时每刻都在发生，更新前先查询版本信息，更新时要指定更新该版本对应的数据，如果版本发生变化则无法更新，**CAS**；

|  |
| --- |
| // A事务先执行前三步，然后去执行B事务  begin;  select id,name from user where id = 1 for update;  update user set phone='98765432100' WHERE id=1;  commit; |
| // B事务阻塞，直到A事务提交  BEGIN;  UPDATE `user` SET phone = '12345678999' WHERE id =1;  commit; |

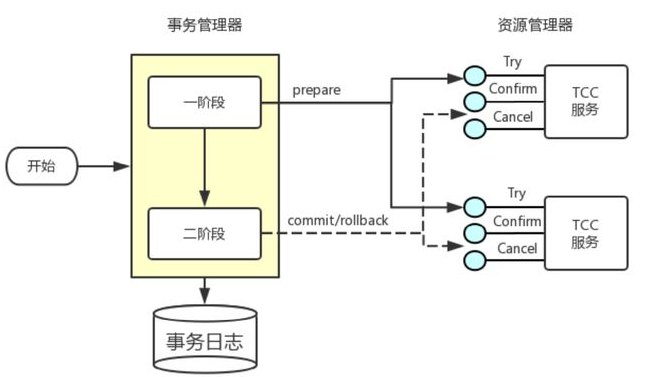
（3）CAS缺点：1.ABA问题；2.循环耗性能；3.只能保证一个共享变量的原则操作；

1. MySQL是怎么用B+树？mysql数据库默认存储引擎InnoDB，有什么优点？开发中用了那些数据库？回答mysql，储存引擎有哪些？MySQL的存储引擎有哪些，有哪些区别？innoDB的B+树索引叶子节点的Data域存储的是什么？InnoDB使用的是什么方式实现索引，怎么实现的？MySQL innoDB的B+树索引，主键索引，聚簇索引有什么区别？
2. 使用主键构建聚集索引，数据行均存放在叶子节点，能够直接访问到数据；
3. 选择自增id作为主键，不要非空UK列。避免大量分页碎片。
4. 一颗B+树可以存放多少行数据？InnoDB页的大小默认是16k，假设一行1k，可以存16k/1k=16行，非叶节点能存放ID(bigint 8字节)和指针数(6字节)16k/(8字节+6字节)=1170；树高为3的B+树能存放：1170 \* 1170 \* 16 = 21902400行数据，所以InnoDB中B+树高度一般为1-3层，就能满则千万级的数据存储；

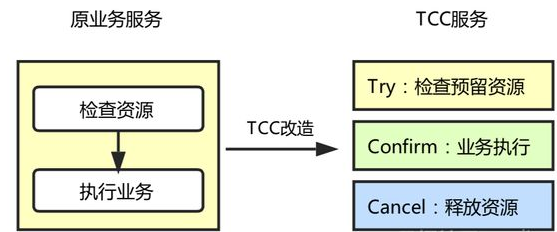
对于innodb来说，达到以下2点就会重新计算cardinality

如果表中1/16的数据发生变化如果stat\_modified\_counter>200 000 0000实际应用中，（Cardinality / 行数）应该尽量接近1。如果非常小则要考虑是否需要此索引。

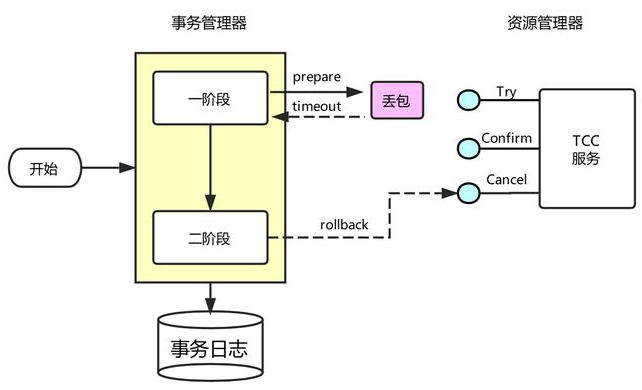
1. 描述分布式事务之TCC服务设计？
2. 解决分布式事务跨库操作一致性问题；
3. Try，Conform，Cancel；



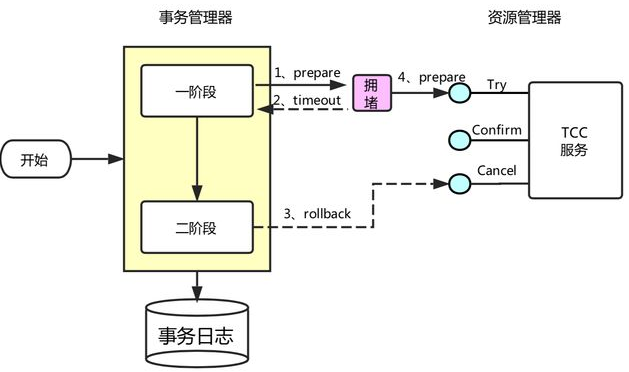
1. 业务操作分两阶段，资源的检查和预留在一阶段Try完成，业务操作的执行放在Conform操作中完成；



1. 允许空回滚：一阶段Try操作丢包而网络超时，二阶段回滚调用Cancel操作；



1. 防悬挂控制：拒绝由一阶段Try丢包网络拥堵，导致二阶段Cancel请求比Try请求多的情况，需要拒绝二阶段已经Cancel操作的一阶段Try请求；



1. 幂等控制：无论是网络包数据重传，还是异常事务补偿，都会导致TCC重复执行；实现TCC需要考虑幂等操作，即Try、Conform、Cancel执行单次和多次的业务结果一直；CAS版本控制机制实现幂等；操作本身幂等，不允许更新时有i++操作；
2. 业务数据并发访问控制：一阶段Try后，二阶段操作执行前，不会释放预留资源，如果此时有其他分布式事务修改这些业务数据就会出现并发的情况；降低锁粒度提高并发性能；
3. 蚂蚁金服的分布式事务框架DTX，三种模式：TCC、FMT、XA，用户使用没有感官上的区别；
4. 慢查询怎么优化？SQL慢查询的常见优化步骤是什么？MySQL的慢sql优化一般如何来做？除此外还有什么方法优化？18.数据库的优化包含哪些？MySQL的优化，谈两个你优化的例子。Mysql优化、索引的实现；MySql优化；SQL 优化的常见方法有哪些
5. 建立索引；
6. 避免使用隐式转换或函数使索引失效；
7. 表结构设计尽量避免Left Join和Right Join，而是使用join提高效率；
8. 必要时建立表达式索引，如lower(index\_str\_key)提高检索效率；
9. 分解复杂sql，关注lock粒度；
10. 业务经常用到的查询建立部分索引，和联合索引类似；
11. 查询字段尽量不要使用\*，业务更清晰，降低IO，带宽开销；
12. MySQL里主要有哪些索引结构？哈希索引和B+树索引比较？hash和B+树的区别？分别应用于什么场景？哪个比较好？索引的常见实现方式有哪些，有哪些区别?
13. B-Tree、 Hash、Gist、SP-Gist、Gin、BRin；
14. 哈希索引和B+树索引比较：1.等值查询使用Hash，范围/Like ‘XXX%’/排序使用B+树；2.Hash索引不支持联合索引；3.B+树稳定，Hash存在大量冲突的情况，效率降低；
15. 场景：存储数据重复低（基数大），对该列的查询以等值查询为主，没有排序，特别适合用Hash查询；其他情况如排序，范围，模糊查询，分组查询使用B+树；
16. 为什么MyISAM查询性能好？MySQL存储引擎innoDB和MyISAM的区别？
17. 聚簇索引和非聚簇索引：聚簇保证关键字的值相近的元祖存储位置相同（所以字符串不易建立聚簇，特别是随机字符串，会导致大量的系统移动操作），一个表只能一个聚簇索引；
18. MyISAM只缓存索引块，不缓存数据块，减少换进换出；
19. innoDB寻址映射到数据块，再到行，MyISAM记录的直接是文件的Offset，定位比innoDB快；
20. InnoDB需要维护MVCC，尽管没有场景使用，但还是要维护；
21. 场景：1.MyISAM适用select和insert多，不要求事务，外键；2.InnoDB适用要求事务、外键、行锁，如果有大量update和insert，尤其是针对高并发QPS较高的情况；
22. 说说事务特性（ACID）？14.事务的ACID，其中把事务的隔离性详细解释一遍；18.数据库四大特性：ACID，原子性、一致性、隔离性、持久性
23. Atomic原子性：操作不可分割，不允许加塞；
24. Consistent一致性：事务前后数据的完整性必须保持一致；
25. Isolation隔离性：多个用户并发操作，不能有相互干扰，多个并发事务之间要隔离；读未提交，读提交，可重复读
26. Durability持久性：事务一旦提交，在数据库的改变就是永久的，即使数据库出现故障也不能对其有任何影响；
27. MySQL的事务隔离级别，分别解决什么问题。事务的隔离级别；脏读、幻影读、不可重复读；18.事务的四大隔离级别：脏读、幻读、不可重复读、串行；事务传播属性；不可重复读会出现在什么场景？

两个事务之间如何控制并发访问的问题：

脏读：AB两个事务，B事务读取到A事务未提交的数据，回滚的数据读取不一致；

不可重复读：B事务读取到A前一次事务提交提交的数据；

幻读：B事务读取到C事务插入的一条数据，导致前后读取不一致；

1. 读未提交read-uncommited：脏读、不可重复读、幻读；
2. 读提交read-commited：不可重复读、幻读；
3. 可重复读repeatable-read：幻读；
4. 串行化serializable：无

Serializable(串行化)：可避免脏读、不可重复读、虚读情况的发生。

Repeatable read(可重复读)：可避免脏读、不可重复读情况的发生。

Read committed(读已提交)：可避免脏读情况发生。

Read uncommitted(读未提交)：最低级别，以上情况均无法保证。

1. 数据库水平切分，垂直切分的设计思路和切分顺序；数据库垂直和水平拆分？
2. 水平拆分避免单表过大的性能下降的缺陷；但需要考虑分割原则，降低整合数据的复杂度，存在跨节点避免join、分布式事务、排序分页查询等操作；
3. 垂直切分考虑数据源管理，连接池管理，不同业务库之间避免join；
4. Mybatis如何分页？如何设置缓存；MySQL分页？MySQL 分页查询语句？
5. 查询到内存分页；
6. Limit offset分页；
7. 拦截器分页；
8. RowBounds分页；

·不同的**SqlSession是隔离**的，为了解决这个问题，我们可以在SqlSessionFactory层面上设置二级缓存提供各个对象SqlSession

·二级缓存默认是不开启的，需要进行配置，Mybatis要求返回的POJO**必须是可序列化**的，即POJO实现Serializable接口。

　　缓存的配置只需要在XML配置<cache/>即可，或者指定**淘汰算法**，**刷新时间间隔**，**缓存状态**，**大小**等

<cache eviction="LRU" readOnly="true" flushInterval="100000" size="1024"></cache>

　　　　A. 映射语句文件中所有select语句将会被缓存；

　　　　B. 映射语句文件中所有insert、update和delete语句会被刷新缓存；

　　　　C. 缓存使用默认的LRU最近最少使用算法回收；

　　　　D. 根据时间表，缓存不会任何时间顺序刷新；

　　　　E. 缓存会存储列表集合或对象的1024个引用

1. 缓存被视为可read/write的缓存，意味着是**不可以被共享**的，而可以被安全地修改。

Mybatis生命周期

　　正确理解SqlSessionFactory、SqlSessionFactoryBuilder、SqlSession和Mapper的生命周期对于优化Mybatis尤为重要，这样可以使Mybatis高效正确完成；同为重要时Mybatis的生命周期对于理解Myabtis缓存的配置也尤为重要，我这里只做简单的文字介绍（其实也好理解）：

（1）SqlSessionFactoryBuilder：作用就是创建一个构建器，一旦创建了SqlSessionFactory，它的任务就算完成了，可以回收。

　　（2）SqlSessionFactory：作用是创建SqlSession，而SqlSession相当于JDBC的一个Connection对象，每次应用程序需要访问数据库，我们就要通过SqlSessionFactory创建一个SqlSession，所以SqlSessionFactory在整Mybatis整个生命周期中（每个数据库对应一个SqlSessionFactory，是单例产生的）。

　　（3）SqlSession：生命周期是存在于请求数据库处理事务的过程中，是一个线程不安全的对象（在多线程的情况下，需要特别注意），即存活于一个应用的请求和申请，可以执行多条SQL保证事务的一致性。

（4）Mapper：是一个接口，并没有实现类它的作用是发送SQL，返回我们需要的结果，或者发送SQL修改数据库表，所以它存活于一个SqlSession内，是一个方法级别的东西。当SqlSession销毁的时候，Mapper也会销毁。

1. MySQL行锁是否会有死锁的情况？MySQL死锁怎么产生，举两个例子。

死锁和死锁检测：



1. 事务A在等待事务B释放id=2的锁，事务B在等待事务A释放id=1的锁；
2. 发生死锁有两种方法解决：1、设置innodb\_lock\_wait\_timeout，超时释放；2、innodb\_deadlock\_detect设置为on，主动回滚死锁链条中的某一个事务；
3. 数据库MySQL相关的，这个也是每次必问，毕竟是平时工作中使用最多的，考察数据库基本功：存储引擎、SQL查询优化、常见索引的使用和区别、事务的使用，表范式设计，以及分库分表的策略和实际应用等。数据库的范式；
4. 一范式：消除重复列，但存在部分依赖；
5. 二范式：消除部分依赖，但存在传递依赖；
6. 三范式：消除传递依赖
7. BCNF范式：某一列与复合主键的某一列有关，而与其他主键无关；
8. 数据库主键索引和唯一索引区别？
9. 主键索引创建后就是包含唯一索引，但唯一索引不一定就是主键；
10. 主键不允许为空，唯一索引允许空；
11. 主键索引更适合不经常变换的列，如自增id、身份证id；
12. 主键只能有一个，唯一索引可以有多个；
13. 主键能被其他表引用为外键，唯一所有不可以；
14. RBO模式，主键执行计划默认高于唯一索引；
15. 聚集索引和非聚集索引的区别？说下聚簇索引和非聚簇索引的区别？MyISAM的B+树叶子节点的Data域存储的是主键还是物理地址？
16. 聚集索引一个表只能有一个，非聚集索引可以有多个；数据摆放完成后就定型了；
17. 聚集索引是物理上连续的，非聚集索引是逻辑上连续，物理不连续；
18. 聚集索引的叶节点上存储的是数据行，非聚集索引存放的还是索引（主键），指向数据的指针；
19. 聚集索引查询快，插入慢，非聚集索引查询慢，插入快；在有主键的表中插入数据行，由于有主键唯一性的约束，所以需要保证插入的数据没有重复。由第三条得出
20. 对MySQL的了解，和oracle的区别？
21. 事务提交：MySQL默认提交；Oracle需要手动提交；
22. 隔离级别：MySQL默认read committed；Oracle默认repeatable - read；
23. 分页查询：MySQL使用Limit offset；Oracle使用RowBounds和嵌套查询；
24. 对事务的支持：MySQL使用InnoDB才支持，而Oracle完全支持；
25. 持久化：MySQL重启丢失数据；Oracle联机日志可恢复；
26. 并发：MySQL默认表锁，索引生效时采用行锁；Oracle自锁定sql需要的资源，粒度小，不依赖索引；
27. 主从复制：MySQL配置从库简单，灾备时可能会丢失数据，需要手动切换；Oracle有推拉式复制，有dataguard的双机或多机容灾，可以自动切换，配置复杂；
28. 性能诊断：MySQL慢查询；Oracle有成熟的诊断机制，awr、sqltrace，自动表分析；
29. 成本：MySQL轻量，免费，没有服务恢复数据；Oracle重量，收费，提供服务；
30. SQL 索引的顺序，字段的顺序？20.查看 SQL 是不是使用了索引？（有什么工具）
31. tid（tid,name,age,address）组合索引。jointime单列索引：
    * 1. 索引失效：explain select name from test2 where tid=5 and name='ee' and age=13 and address='hb' order by jointime;
32. tid（tid,name,age,address,jointime）组合索引：
    * 1. 索引有效：explain select name from test2 where tid=5 and name='ee' and age=13 and address='hb' order by jointime;

字段与顺序不一致：

1. 索引失效：explain select name from test2 where tid=5  order by age;
2. 索引有效：explain select name from test2 where tid=5  order by name;
3. sql having 的使用场景

（1）where 后面要跟的是数据表里的存在的字段，如果我把ag换成avg(price)也是错误的！因为表里没有该字段。

（2）而having只是根据前面查询出来的是什么就可以后面接什么（比如一些聚合函数这样的“伪字段”）。

查询每种category\_id商品的价格平均值，获取平均价格大于100元的商品信息：

1）select category\_id , avg(price) as ag from dw\_goods group by goods\_category having ag > 100

2）select category\_id , avg(price) as ag from dw\_goods where ag>100 group by goods\_category //报错！！因为from dw\_goods 这张数据表里面没有ag这个字段

### NoSQL

1. 有使用过哪些NoSQL数据库？MongoDB和Redis适用哪些场景？
2. 数据存储：Redis是key-value，有string、hash、set、zset、GEO、bytemap、haploglog；MongoDB文档型，类似json，可以建索引，可以简单模仿关系数据库，有table和db概念；
3. 安全性：Redis全局鉴权；MongoDB分用户，RBAC；
4. 数据库有效性：Redis针对Key，不针对value中的元素；MongoDB数据仓储；
5. 持久化：Redis有RDB和AOF；MongoDB本身持久化；
6. 应用场景：Redis读写效率要求高，数据处理业务复杂、安全性要求较高；MongoDB海量数据的访问效率；
7. redis常用的数据结构有哪几种，在你的项目中用过哪几种，以及在业务中使用的场景？
8. String：存储简单的键值对，如用户信息、登录信息、配置属性等，也能做原子计数器，incr/decr；
9. Hash字典：存储、读取用户属性信息；
10. List列表：消息队列，Lpush、Rpop；
11. Set集合：唯一性统一访问IP；
12. SortedSet有序集合：带有权重的元素，设备质差排名；
13. Pub/sub和事务；
14. 漏洞修复：1.更换默认端口；2.禁止使用root用户号；3.限制访问IP；

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。

如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，

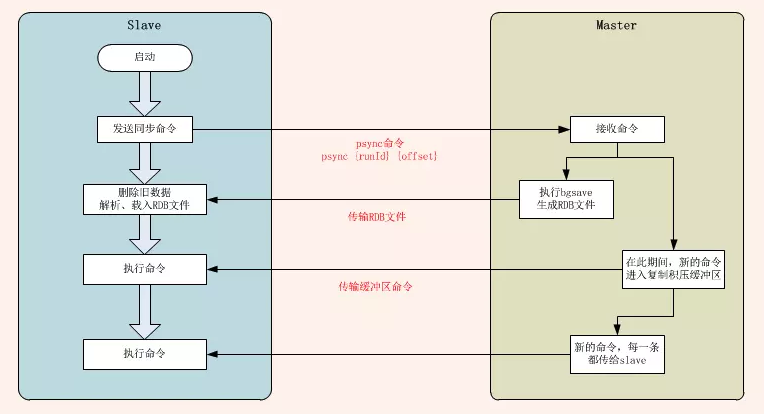
1. Redis和memcache有什么区别？Redis为什么比memcache有优势？
2. 存储方式：Redis存于内存，可定期持久化到磁盘；Memcache基于内存，掉电数据丢失；
3. 数据格式：Redis存储string、list、set、zset、GEO、bytemap、happerloglog；Memcache可存储图片、视频；
4. 过期策略：Redis可以设置有效期；Memcache永久生效；
5. 应用场景：Redis有持久化，数据处理要求高；Memcache简单的key-value；
6. 慎用：Redis单Value512M;Memcache单value 1M；
7. 考虑redis的时候，有没有考虑容量？物理内存的三分之一；大概数据量会有多少？Info memory，计算一条128Byte，\*2000w=2.5G，峰值12G/台，物理内存16G，设置3/4为redis可用内存；Redis常见问题；
8. Master/Slave模式：Master写入和AOF日志备份，crontab定期bgRewirte，出现Master5分钟不响应，slave也不响应；解决：no-appendfsync-on-rewrite的配置设为yes，AOF备份只在slave开启；
9. Master写快照：save命令调度rdbsave，会阻塞主线程工作；当快照非常大，暂停服务；
10. MasterAOF持久化：AOF文件过大，rewrite恢复速度慢；
11. Master重写AOF：bgwrite重写AOF时会占用大量CPU和内存资源，导致服务load过高，出现短暂停服；

Redis常见性能问题：14.redis cluster有没有了解过，怎么做到高可用的？

1. Connect time out：慢查询；
2. Swap和OOM：单机多实例bgrewriteaof；
3. 内存飙升：QPS较高时执行monitor，缓存了大量的命令；
4. Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件；
5. 如果数据比较重要，只开启Slave的AOF备份数据，策略设置每秒同步一次；
6. 为了保证主从复制的稳定性和速度，主从防止一个局域网，最好在一台机架或相邻机架；
7. 尽量避免在压力很大的主库上增加从库；
8. 主从复制不要采用图状结构，用单向链表更加稳定，即Master->Slave1->Slave2->....；这样解决单点故障，Master挂了可以立刻切换；
9. MySQL中有2000w数据，redis存20万数据，如何保证redis中的数据都是热点数据？Redis使用内存达到物理内存的3/4，开始执行淘汰数据策略：7.Redis 回收和备份
10. voltile-lru：从设置过期时间的key（server.db[i].expires），挑选出最近最少使用的淘汰；
11. Voltile-ttl：从设置过期时间的key（server.db[i].expires），挑选即将过期的淘汰；
12. Voltile-random：从设置过期时间的key（server.db[i].expires），随机挑选；
13. Allkeys-lru：从所有key，淘汰最近最少使用；
14. Allkeys-random：从所有key，随机淘汰数据；
15. No-enviction：禁止驱逐数据；

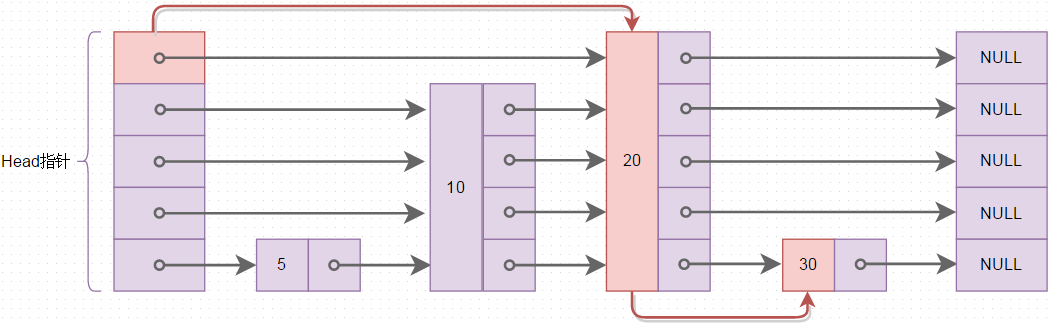
备份：Redis持久化的方式以及区别？16.redis的持久化机制了解吗？你们在项目中是怎么做持久化的？

1. RDB：快照；
2. AOF：Append-Only-File，RESP追加；
3. Redis 的同步机制了解么？12.谈谈Redis哨兵、复制、集群？redis集群和哨兵机制有什么区别？是否使用过 Redis 集群，集群的原理是什么？
4. Slave第一次注册全量同步：



（2）增量拷贝:psync {runId} {offset}如果偏移量之后的数据存在缓冲区中，则对从节点发送+continue响应，表示可以进行部分复制；否则进行全量复制。

1. Redis数据结构有哪些？如何实现sorted set？这种数据结构在极端情况下有什么问题？二叉平衡树？缺点是在某些极端情况下,二叉查找树可能变成一个线性链表。
2. Sort命令的时间复杂度为O(N+M\*Log(M)),可排序集获取一定范围内元素的时间复杂度为O(log(N) + M)



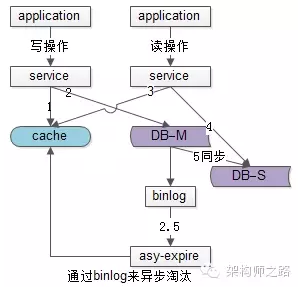
其实在redis sorted sets里面当items内容大于64的时候同时使用了hash和skiplist两种设计实现。

1. Redis如何解决key冲突？

Redis 的哈希表使用链地址法（separate chaining）来解决键冲突： 每个哈希表节点都有一个 next 指针， 多个哈希表节点可以用 next 指针构成一个单向链表， 被分配到同一个索引上的多个节点可以用这个单向链表连接起来， 这就解决了键冲突的问题。

1. Redis如何解决热key？
2. 监控热key：
   * 1. Redis4.0版本redis-faina
     2. Flink流计算框架，抓包监听redis端口，往kafka送，流式框架消费kafka里的数据，进行统计；
3. 通知系统做处理：二级缓存备份热key，EhCache/GuavaCache/HashMap；
4. 如何保证数据库与redis缓存一致的？redis 集群如何保证一致性？

reids集群是主节点通过异步复制的方式把客户的数据异步写到从节点（注意：Redis 集群可能会在将来提供同步写的方法）



1. 缓存的雪崩以及穿透的理解？

缓存穿透：

1. bitmap制作布隆过滤器，减少部分不存在的key到数据库获取连接；
2. 对不存在结果的key设置空的返回值，并缓存下来，设置超期时间；

雪崩：

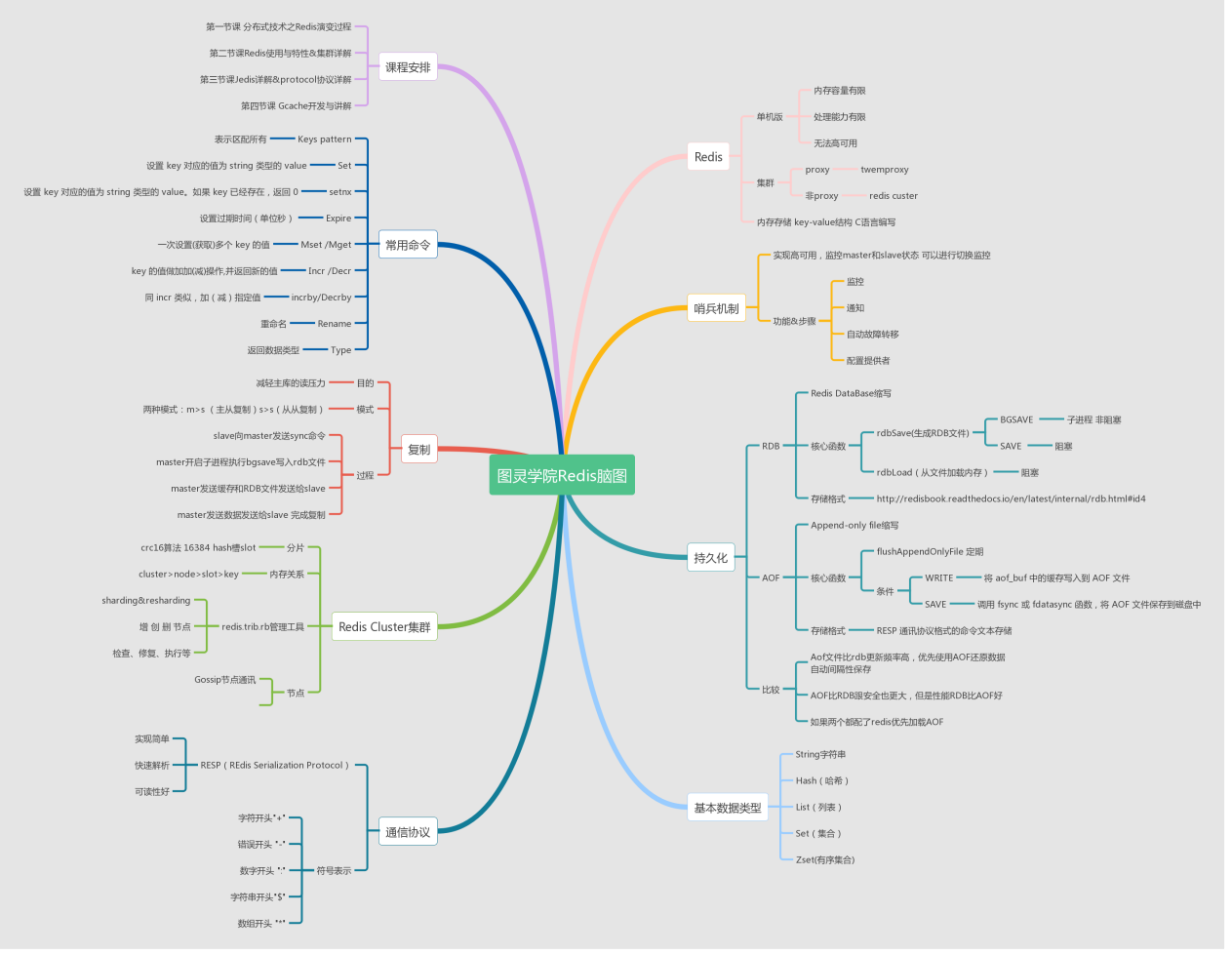
1. 事前：Redis哨兵监控实现高可用；
2. 事中：本地ehcache/guavacache，Hystrix限流，避免mysql挂；
3. 事后：利用Redis持久化机制保存的数据尽快回复；

击穿（热key）：12.如何利用Redis处理热点数据？17.遇到过redis的hotkey吗？怎么处理的？

1. Setnx
2. 流式框架，记录binlog，放队列，识别热key，回设java缓存；
3. Redis是单线程？
4. Redis高并发快的原因？
5. 分布式下Redis如何保证线程安全？
6. db数据cas；
7. 分布式锁，zookeeper；
8. Redis set nx；
9. redis是单线程的吗？单线程为什么还这么快？
10. 怎么理解 Redis 事务
11. 什么是 Redis？简述它的优缺点？
12. Redis 与 memcached 相比有哪些优势？
13. Redis 集群方案应该怎么做？都有哪些方案？
14. Redis 集群方案什么情况下会导致整个集群不可用？
15. MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据？ redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。
16. Redis 有哪些适合的场景？
17. Redis 支持的 Java 客户端都有哪些？官方推荐用哪个？
18. Redis 和 Redisson 有什么关系？
19. Jedis 与 Redisson 对比有什么优缺点？
20. 说说 Redis 哈希槽的概念？
21. Redis 集群的主从复制模型是怎样的？
22. Redis 集群会有写操作丢失吗？为什么？
23. Redis 集群之间是如何复制的？
24. Redis 集群最大节点个数是多少？
25. Redis 集群如何选择数据库？
26. Redis 中的管道有什么用？
27. 怎么理解 Redis 事务？
28. Redis 事务相关的命令有哪几个？
29. Redis key 的过期时间和永久有效分别怎么设置？
30. Redis 如何做内存优化？
31. Redis 回收进程如何工作的？
32. 加锁机制
33. 锁互斥机制
34. watch dog 自动延期机制
35. 可重入加锁机制
36. 释放锁机制
37. 上述 Redis 分布式锁的缺点
38. 使用过 Redis 分布式锁么，它是怎么实现的？
39. 使用过 Redis 做异步队列么，你是怎么用的？有什么缺点？
40. 什么是缓存穿透？如何避免？什么是缓存雪崩？何如避免？
41. 什么是 Redis？简述它的优缺点？
42. Redis 集群方案什么情况下会导致整个集群不可用？
43. MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis中的数据都是热点数据？
44. Redis 有哪些适合的场景？
45. Redis 支持的 Java 客户端都有哪些？官方推荐用哪个？
46. Redis 和 Redisson 有什么关系？
47. Jedis 与 Redisson 对比有什么优缺点？
48. Redis 如何设置密码及验证密码？
49. 说说 Redis 哈希槽的概念？
50. Redis 集群的主从复制模型是怎样的？
51. Redis 集群会有写操作丢失吗？为什么？
52. Redis 集群之间是如何复制的？
53. 怎么测试 Redis 的连通性？
54. redis 和 memcached 什么区别？为什么高并发下有时单线程的 redis 比多线程的 memcached 效率要高？
55. redis 主从复制如何实现的？redis 的集群模式如何实现？redis 的 key 是如何寻址的？
56. 使用 redis 如何设计分布式锁？说一下实现思路？使用 zk 可以吗？如何实现？这两种有什么区别？
57. 知道 redis 的持久化吗？底层如何实现的？有什么优点缺点？
58. redis 过期策略都有哪些？LRU 算法知道吗？写一下 java 代码实现？
59. 缓存穿透、缓存击穿、缓存雪崩解决方案？
60. 在选择缓存时，什么时候选择 redis，什么时候选择 memcached
61. 缓存与数据库不一致怎么办
62. 主从数据库不一致如何解决
63. Redis 常见的性能问题和解决方案
64. Redis 的数据淘汰策略有哪些
65. Redis 当中有哪些数据结构
66. 假如 Redis 里面有 1 亿个 key，其中有 10w 个 key 是以某个固定的已知的前缀开头的，如果将它们全部找出来？

redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

1. 使用 Redis 做过异步队列吗，是如何实现的
2. Redis 如何实现延时队列



1、基于Jedis Cluster开发的客户端支持Redis Cluster集群。

2、对被调用方（客户端）侵入极少，上手极快。

3、支持动态增加节点，客户端自动感知。

4、支持客户端验证与拦截。

5、异步监控调用数据，支持异步上报。

6、方便管理有效的区分业务系统。

7、支持G-Falcon协议.

项目介绍：

Enums：枚举类型

Impl：实现功能，具体实现了rediscluster和redis的命令

Jedis：封装jiedis操作

Monitor：方法拦截、数据存储、数据上传

Spring:对redis集群实列管理

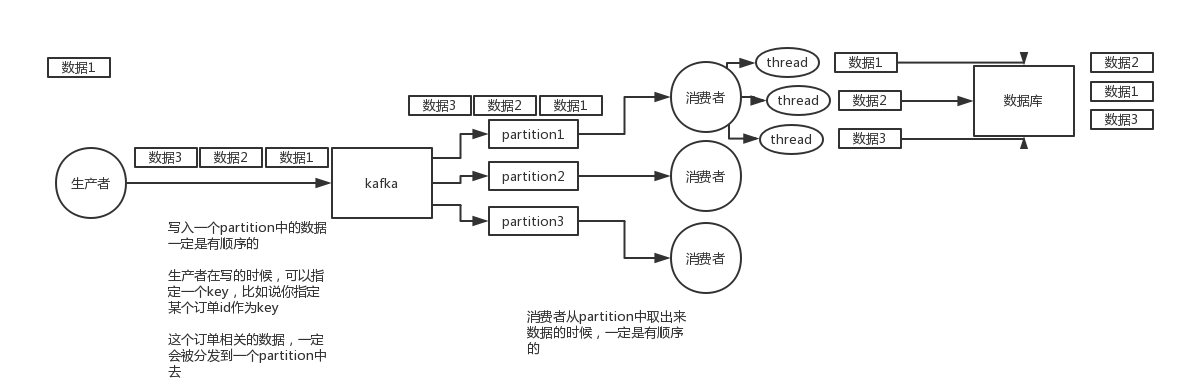
Utils：工具类

Zookeeper：动态感应集群变动

### MQ

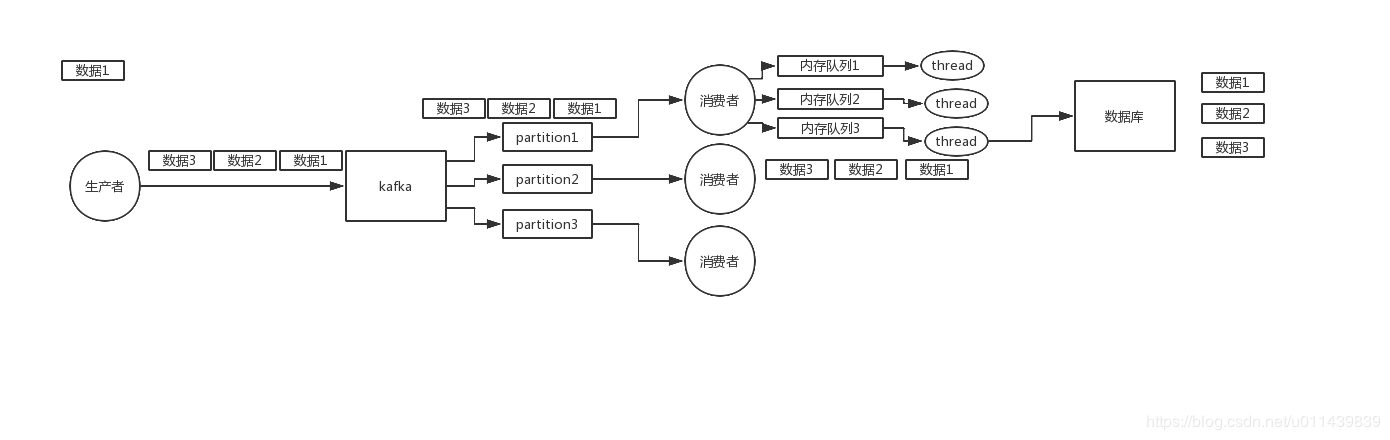
**Kafka**

1. 项目中消息队列怎么用的？使用哪些具体业务场景？
2. Kafka的架构，如何用kafka保证消息的有序性？
3. kafka的特性：1.写入一个partition中数据有序；2.一个消费者消费一个partition数据有序；
4. 有序性场景：1.数据库binlog；2.业务订单写入一个partition；
5. 为何消息错乱：



解决办法：

1. 生成者不指定分区，有指定key 则根据key的hash值与分区数进行运算后确定发送到哪个partition分区；
2. 消费者将需要保证有序的消息分发到内存队列，然后多线程去执行；ConcurrentHashMap+LinkedHashMap内存队列



1. kafka的原理？怎么保证消息不丢失？
2. Kafka 的设计时什么样的呢？
3. 数据传输的事物定义有哪三种？
4. Kafka 判断一个节点是否还活着有那两个条件？
5. producer 是否直接将数据发送到 broker 的 leader(主节点)？
6. Kafa consumer 是否可以消费指定分区消息？
7. Kafka 消息是采用 Pull 模式，还是 Push 模式？
8. Kafka 存储在硬盘上的消息格式是什么？
9. Kafka 高效文件存储设计特点
10. Kafka 与传统消息系统之间有三个关键区别？
11. Kafka 创建 Topic 时如何将分区放置到不同的 Broker 中
12. Kafka 新建的分区会在哪个目录下创建
13. partition 的数据如何保存到硬盘
14. kafka 的 ack 机制
15. Kafka 的消费者如何消费数据
16. 消费者负载均衡策略
17. 数据有序
18. kafaka 生产数据时数据的分组策略

**RabbitMQ**

1. 为什么不应该对所有的 message 都使用持久化机制？
2. 为什么 heavy RPC 的使用场景下不建议采用 disk node
3. 向不存在的 exchange 发 publish 消息会发生什么？向不存在的 queue 执行 consume 动作会发生什么？
4. 什么情况下 producer 不主动创建 queue 是安全的？
5. “dead letter”queue 的用途？
6. 为什么说保证 message 被可靠持久化的条件是 queue 和 exchange具有 durable 属性，同时 message 具有 persistent 属性才行？
7. RabbitMQ 中的 broker 是指什么？cluster 又是指什么？
8. 什么是元数据？元数据分为哪些类型？包括哪些内容？与 cluster 相关的元数据有哪些？元数据是如何保存的？元数据在 cluster 中是如何分布的？
9. RAM node 和 disk node 的区别？
10. RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？
11. RabbitMQ 概念里的 channel、exchange 和 queue 这些东东是逻辑概念，还是对应着进程实体？这些东东分别起什么作用？
12. vhost 是什么？起什么作用？

【cluster 相关】

1. 在单 node 系统和多 node 构成的 cluster 系统中声明 queue、exchange ，以及进行 binding 会有什么不同？
2. 客户端连接到 cluster 中的任意 node 上是否都能正常工作？
3. 若 cluster 中拥有某个 queue 的 owner node 失效了，且该 queue 被声明具有durable 属性，是否能够成功从其他 node 上重新声明该 queue ？
4. cluster 中 node 的失效会对 consumer 产生什么影响？若是在 cluster 中创建了mirrored queue ，这时 node 失效会对 consumer 产生什么影响？
5. 能够在地理上分开的不同数据中心使用 RabbitMQ cluster 么？
6. 为什么 heavy RPC 的使用场景下不建议采用 disk node ？
7. 向不存在的 exchange 发 publish 消息会发生什么？向不存在的 queue 执行
8. consume 动作会发生什么？
9. 什么情况下会出现 blackholed 问题？
10. 如何防止出现 blackholed 问题？
11. Consumer Cancellation Notification 机制用于什么场景？
12. Basic.Reject 的用法是什么？
13. RabbitMQ 中的 cluster、mirrored queue，以及 warrens 机制分别用于解决什么问题？存在哪些问题？

### JVM

1. 新生代分几个区？使用什么算法进行垃圾回收？为什么使用这个算法？JVM为什么要分区，为什么要有新生代和年老代；
2. JVM老年代和新生代的比例？
3. YGC和FGC发生的具体场景
4. 请介绍一下JVM内存模型？用过什么垃圾回收器？描述一下回收机制
5. 知道字节码吗？字节码都有哪些？Integer x= 5，int y=5，比较x==y都经过哪些步骤？
6. 讲讲类加载机制呗，都有哪些类加载器，这些加载器都加载哪些文件？手写一下类加载Demo？
7. 请问你做过哪些JVM优化？使用什么方法达到什么效果？
8. classForName(“java.lang.String”)和String classgetClassLoader() LoadClass(“java.lang.String”)什么区别呢？
9. GC用什么收集器？收集过程如何？哪些部分可以作为GCRoots？讲讲Java GC机制？Java GC机制？GC Roots有哪些？
10. jvm内存模型jmm 知道的全讲讲
11. Java怎么进行垃圾回收的？什么对象会进老年代？
12. 垃圾回收算法有哪些？为什么新生代使用复制算法？
13. JVM相关的分析工具有使用过哪些？具体的性能调优步骤吗？
14. JVM内存模型？
15. JVM不用说了，每次都考，如果对内存的回收，垃圾回收器的种类区别，回收算法机制这个必须要掌握。
16. 说几个垃圾收集器，cms回收有哪些过程？停顿几次？会不会产生内存碎片？老年代产生内存碎片会产生什么问题？
17. 讲讲双亲委派模型？
18. JVM场景问题，标记清除多次老年代产生内存碎片，引起Full GC，接下来可能发生什么问题；
19. 说一下java类加载器的工作机制？
20. 讲一下JVM相关。有没有做货GC调优，讲一下这么做的？
21. JVM内存管理，GC算法，HotSpot里面的垃圾回收器、类加载
22. Java的垃圾回收机制
23. JVM内存模型及调优
24. happens before 原理

### J.U.C

1. 线程池的工作原理，几个重要参数，然后具体几个参数分析线程池会怎么做，最后阻塞队列的作用是什么？线程池用过吗？都有什么参数？底层如何实现？说说Java线程池的原理？线程池有哪些？线程池工厂有哪些线程池类型，及其线程池参数是什么？说下线程池的原理？有没有在项目中用过线程池？怎么用的？核心线程和非核心线程有什么区别？你在项目里怎么确定核心线程数的。

2. 讲一讲AtomicInteger，为什么要用CAS而不是Synchronized？CAS是什么，如何实现？

3. 单机上一个线程池正在处理服务如果忽然断电该怎么办？（正在处理和阻塞队列里的请求怎么处理）？

4. 使用无界阻塞队列会出现什么问题？

5. 如何保证共享变量修改时的原子性？

6. Synchronized和Lock什么区别？Synchronized什么情况是对象锁？什么时候是全局锁？为什么？

7.ThreadLocal是什么？底层如何实现？写一个例子呗？ThreadLocal 使用场景？ThreadLocal？应用场景？

8.Volatile的工作原理？Volatile关键字，指令重排序有什么意义？Synchronized怎么用？Java同步、volatile关键字，说一下violate和synchronized这两个关键字的区别？

9. 并发包里的原子类有哪些？怎么实现？Cas在CPU级别用什么指令实现的？

10. 加锁有什么机制？

11. synchronized和lock的区别？synchronized、lock；lock 和 synchronized 的区别？

12. synchronized 和 ReentranLock的区别？

13. Java相关的：线程锁、以及线程流转图、线程池等。

14. 线程池用过哪些，线程池有哪些参数，几个常用线程池的用法和实际场景；

15. 说一下violate关键字？他的原理是什么？violate是线程安全的吗？volatile 的原子性问题？为什么 i++ 这种不支持原子性？从计算机原理的设计来讲下不能保证原子性的原因？

16. 对concureent包了解吗？

17. 什么是cas？cas怎么解决ABA问题？cas 操作

18. 讲一下CountDownLatch和cyclicBarrier的区别？

19. 怎么实现线程安全，各个实现方法有什么区别，volatile关键字的使用，可重入锁的理解，Synchronized是不是可重入锁

20.Java IO，NIO，Java中有没有实现异步IO

21. 让我设计一个线程池；JDK 中有哪几个线程池？顺带把线程池讲了个遍

22. hashmap和concurrenthashmap区别及两者的优缺点

23. 多线程如何避免死锁

24. 如何解决高并发问题？是否进行过相应程序的编写？

25. 如何保证线程安全问题？

26. 公平锁和非公平锁？

27. Java 读写锁

28. 读写锁设计主要解决什么问题？

### Netty/RPC

1. BIO、NIO、AIO的区别和联系？10.熟悉IO吗？与NIO的区别，阻塞与非阻塞的区别？Aio、nio、bio的了解，NIO的核心概念有哪些？
2. 用到过哪些rpc框架？
3. Dubbo超时重试？Dubbo超时时间设置？
4. Dubbo的底层原理，Zookeeper是什么？
5. 网络编程nio和netty相关，netty的线程模型，零拷贝实现
6. 常用的NIO框架有哪些？优劣势

### Zookeeper

1. Zookeeper有哪些作用？
2. Dubbo里的zookeeper是做什么的？
3. Zookeeper如何实现分布式锁，其他分布式锁怎么实现？
4. redis实现分布式锁以及和zk实现的性能和稳定性的对比？
5. 注册中心你了解了哪些？
6. ZooKeeper 是什么？
7. ZooKeeper 提供了什么？
8. Zookeeper 文件系统？
9. 四种类型的 znode
10. Zookeeper 通知机制？
11. Zookeeper 做了什么？
12. zk 的命名服务（文件系统）？
13. zk 的配置管理（文件系统、通知机制）？
14. Zookeeper 集群管理（文件系统、通知机制）
15. Zookeeper 分布式锁（文件系统、通知机制）
16. 获取分布式锁的流程
17. Zookeeper 队列管理（文件系统、通知机制）
18. Zookeeper 数据复制
19. Zookeeper 工作原理
20. zookeeper 是如何保证事务的顺序一致性的？
21. Zookeeper 下 Server 工作状态？
22. zookeeper 是如何选取主 leader 的？
23. Zookeeper 同步流程
24. 分布式通知和协调
25. 机器中为什么会有 leader？
26. zk 节点宕机如何处理
27. zookeeper 负载均衡和 nginx 负载均衡区别
28. zookeeper watch 机制

### 微服务

1. soa和微服务的区别？
2. Spring Cloud用到什么东西？如何实现负载均衡？服务挂了注册中心怎么判断？
3. 微服务涉及到的技术以及需要注意的问题有哪些？
4. consul 的可靠性你了解吗？
5. consul 的机制你有没有具体深入过？有没有和其他的注册中心对比过？
6. Spring Cloud 有了解多少？
7. 微服务划分的粒度
8. 网关能够为后端服务带来哪些好处？

### Nginx

1. Nginx负载均衡策略？
2. Nginx和其他负载均衡框架对比过吗？

### 设计模式

1. 请至少用四种写法写一个单例模式？
2. 说一说静态代理和动态代理的区别？
3. 动态代理的底层是怎么实现的？动态代理的底层是怎么实现的？
4. 说一下你熟悉的设计模式
5. 讲几个设计模式，哪些地方用到了，为什么要用
6. 对设计模式的看法和认知
7. 有哪些设计模式
8. 项目具体部分使用的设计模式并简述选择理由

### 算法

1. 快速排序；广度优先搜索（队列实现）；讲讲快速排序，分析时间复杂度？
2. 红黑树、二叉树的算法；
3. 从中序与后序遍历序列构造二叉树
4. 给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。
5. 找到单链表的三等分点，如果单链表是有环的呢
6. 从10万个数中找最小的10个，时间复杂度分析（最大堆，考虑内存）
7. 在collabedit上在线写代码，题目很简单是编程之美上的原题，一个有序的整数数组，输出两个数，使它们的和为某个给定的值。之前做过很快写好，然后给他讲思路。他继续问如果数组无序怎么办，先排序。
8. 两个文件，每个文件中都有若干个url，找出两个文件中相同的url（用HashMap）
9. 关于树的算法题-二叉树的锯齿形层次遍历：http://www.lintcode.com/zh-cn/problem/binary-tree-zigzag-level-order-traversal/
10. LintCode 算法题 - 最小子串覆盖。原题链接：<http://www.lintcode.com/zh-cn/problem/minimum-window-substring/>
11. 说下你知道的排序算法吧
12. 查找一个数组的中位数？

### 网络

1. https和http区别，有没有用过其他安全传输手段？
2. HTTP有没有状态，我说无状态，怎么解决HTTP无状态
3. Http协议
4. TCP 和 UDP 的区别？TCP 数据传输过程中怎么做到可靠的？
5. 前端浏览器地址的一个 http 请求到后端整个流程是怎么样？能够说下吗？
6. http 默认端口，https 默认端口
7. DNS 你知道是干嘛的吗？

### 前端

1. 前端会不会，Ajax是什么，Ajax实现原理
2. 浏览器的缓存机制
3. 你项目除了写 Java 代码，还有前端代码，那你知道前端有哪些框架吗？

### 分布式

1. 谈谈分布式锁、以及分布式全局唯一ID的实现比较？
2. 接口如何处理重复请求？具体处理方案是什么？分布式接口的幂等性设计「不能重复扣款」
3. 讲讲TCP协议的三次握手和四次挥手过程？
4. 单机系统演变为分布式系统，会涉及到哪些技术的调整？请从前面负载到后端详细描述。
5. 如何保障请求执行顺序？
6. 分布式事务与分布式锁？（扣款不要出现负数）
7. 分布式session设置？分布式session一致性？
8. 然后问了http协议，get和post的基本区别，接着tcp/ip协议，三次握手，窗口滑动机制。
9. 分布式锁的实现你知道的有哪些？具体详细谈一种实现方式
10. 分布式也是必考的系列，问得最多就是Redis、Dubbo等，面试官主要就是考察缓存、RPC的实际使用情况。
11. 如何实现分布式缓存
12. 常用的负载均衡，该怎么用，你能说下吗？

### 故障排查/优化

1. Linux怎么查看系统负载情况？
2. 集群监控的时候，重点需要关注哪些技术指标？这些指标如何优化？
3. jstack，jmap，jutil分别的意义？如何线上排查JVM的相关问题？
4. 线上发生频繁FullGC如何处理？CPU使用率过高怎么办？
5. 如何定位问题？如何解决？说一下解决思路和处理方法？
6. 遇到的坑，怎么解决，后续总结。
7. 线上的服务器监控指标，你认为哪些指标是最需要关注的？为什么？
8. 工作中技术优化过哪些？JVM、MySQL、代码等都谈谈
9. 有做过JVM优化？
10. 有没有配置过服务器啥
11. 项目中哪一部分最难攻克？如何攻克？
12. 线上服务 CPU 很高该怎么做？有哪些措施可以找到问题
13. 代码版本管理你们用的是啥？

### 项目

1. 介绍项目
2. 各种聊项目，从项目的架构设计到部署流程。
3. 从千万的数据到亿级的数据，会面临哪些技术挑战？你的技术解决思路？
4. 最近做的比较熟悉的项目是哪个？画一下项目技术架构图
5. 自我介绍，讲下项目！
6. 最后谈谈你参与过的项目，技术含量比较高的，相关的架构设计以及你负责哪些核心编码
7. 工作中，遇到了技术瓶颈无法解决，你的解决思路？
8. 首先上来简单做一下自我介绍。然后让介绍简历里的项目。
9. 平时开发过程中都遇到过哪些难题？
10. 平时都这么和同事沟通？
11. 有没有用过Hibernate、mybatis、git
12. 对你来说影响最大的一个项目（该面试中有关项目问题都针对该项目展开）
13. 为什么会想做这个项目？这个项目的ideal是谁提出来的？
14. 项目中如何实现的大数据的传输和存储
15. 项目运行过程中成员是否曾就某一点发生争执？作为Leader你是如何解决的？具体事例？
16. 项目最终实现效果
17. 项目有何需改进之处并初拟改进方案
18. 自我介绍、自己做的项目和技术领域
19. 项目中的监控：那个监控指标常见的有哪些？
20. 自我介绍、工作经历、技术栈
21. 项目中你学到了什么技术？（把三项目具体描述了很久）
22. 你们开发用的 ide 是啥？你能说下 idea 的常用几个快捷键吧？
23. git rebase 和 merge 有什么区别？

### 架构/系统设计

1. 设计一个对外服务的接口实现类，在1,2,3这三个主机（对应不同IP）上实现负载均衡和顺序轮询机制（考虑并发）；
2. 系统设计题：一个推送场景，50 条内容，定时推送，先推 5%用户，一段时间后再找出效果最好的几条，推给所有用户。设计相关库表，系统模块, 需要可以落地，有伪代码
3. 项目：cache 设计，MQ 丢失消息，RPC 使用场景。各部分职责，有哪些优化点。
4. 项目设计问题——主要是针对项目找漏洞，然后给出优化方案
5. 设计一个秒杀系统？
6. 开发一个大型网站你会考虑哪些问题?
7. 高并发的应用场景，技术需要涉及到哪些？怎样来架构设计？
8. 接着高并发的问题，谈到了秒杀等的技术应用：kafka、redis、mycat等
9. 说下项目里的难点，技术架构。
10. 讲一下项目的架构

### 个人成长

1. 最近两年遇到的最大的挫折，从挫折中学到了什么？
2. 最近有没有学习过新技术？
3. 自我介绍，主要问了做了什么和擅长什么
4. 自我介绍，技术特点
5. 兴趣是什么，优势是什么
6. 简短自我介绍
7. 未来职业规划
8. 为什么要选择编程这条路
9. 何时开始编程
10. 如何进行自学

### 杂项

1. 有啥想问我的？
2. 你有什么问题想问我的吗？
3. 你们公司加班多吗？

## 第二节Java基础

### 2.1HashMap底层执行原理

·存储结构：数组、链HashMap底层执行原理表（JDK1.8尾插法）、红黑树（JDK1.8）；

·特点：快速存储；快速查找（O(1)）；可伸缩；

·Hash算法：(hashCode) ^ (hashCode >>> 16)，^异或运算；

·数组下表计算：hash & (16 - 1) = hash % 16；

·Hash冲突：不同对象算出来数组下标是相同的；

·单向链表：用于解决Hash冲突的方案，JDK1.8采用尾插法，防止头插法并发操作引入链表环路；

·数组扩容：扩容两倍，负载因子0.75

·触发条件：数组存储比例达到75%（不论是否Hash冲突，总存储元素达到阈值75%） --0.75；

·红黑树：一种二叉树，高效的检索效率，根和叶子黑色，根到叶子路程一致，红黑交叉，同一层只有一种颜色；

·触发条件，在链表长度大于8的时候，将后边的数据存在红黑树中；

|  |
| --- |
| 桶： 就是hashmap的table数组  bin: 就是挂在数组上的链表  TreeNode: 红黑树  capacity： table总容量  MIN\_TREEIFY\_CAPACITY ：64   转化为红黑树table最小大小  TREEIFY\_THRESHOLD ：8  转化为红黑树的阈值  loadFactor:  0.75  table扩容因子，当实际length大于等于 capacity\*loadFactor时会进行扩容，并且扩容是按照2的整数次幂，原因下面解释  threshold: capacity\*loadFactor |
| IMG_256 |
| IMG_256 |
| IMG_256 |
| IMG_256 |

### 2.2HashTable和ConcurrentHashMap如何实现线程安全

·未做同步控制，代码在多线程下是线程安全的吗？否

·HashMap是线程安全的吗？否

·Map-HashTable是线程安全的吗？是，Synchronized关键字

·有了HashTable以后，为什么还要有ConcurrentHashMap？怎么在保证性能的前提下，保证线程安全；

对于操作不同数组下标对应的桶中的数据，是不存在线程安全问题的，不会同时操作共享操作，所以引入分段锁，对不同的桶使用一把锁，只有操作同一段下标的桶中的数据，才会竞争锁。JDK1.7 Table长度16，有四把锁；JDK1.8Table有多长就有多少把锁，只有操作Hash冲突的数据才会出现竞争，以此来提高读写性能。

### 2.3JVM的内存布局和垃圾回收机制

·Java虚拟机栈：每个方法在被调用时就会创建一个栈帧（局部变量表+动态链接+操作数引用+方法返回值），每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程；

·Java堆：是Java虚拟机所管理的内存中最大的一块，Java堆是被所有线程共享的一块内存区域，对象实例在这里分配内存。是垃圾收集器（GC）管理的主要区域；

·方法区：存储已经被虚拟机加载的类型西、常量、静态变量、及时编译器编译后的代码等数据，运行时常量池（Runtime Constant Pool）是方法区的一部分（JDK1.6存放方法区，JDK1.7存放于堆，JDK1.8存放于MetaSpace）；

·直接内存：（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域；

·标记-清除（Mark-Sweep）：直接回收，会产生内存碎片；

·复制（Coping）：分两个区域，任何时候总有一半数据没有利用；

·标记-整理（Mark-Compact）：边回收，边整理，存在内存区域块移动，对性能会有影响；

·分代收集：

新生代：Serial、ParNew、Parallel Scavenge

年老代：Serial Old、CMS、Parallel Old

G1：内存分块，既可能分给新生代，也有可能分给年老代，没有绝对的比例划分；

### 2.4类加载机制和双亲委派模型

加载过程：



·加载：从class文件/字节流，转换成内存实际存在的对象；

·验证：检查这个java文件/class文件，是否符合虚拟机规范；

·准备：为类变量设置初始值；

·解析：将常量池的引用替换成实际值；

·初始化：static块执行；

自定义类加载器：应用于热替换，类加密；



·Bootstrap ClassLoader：加载范围：<JAVA\_HOME>\lib目录下，并且是虚拟机识别的类库加载到虚拟机内存中；

·Extension ClassLoader：加载范围：<JAVA\_HOME>\lib\ext目录中的所有类库，开发者可以直接使用；

·Application Loader：加载用户类路径上指定的类库，开发者可以直接使用，一般情况下这个就是程序中默认的类加载器；

实现类优先级，保证类的唯一性；不同类加载器加载同一个类是不同的类。

### 2.5阐述事务的隔离级别和传播属性

7个事务传播属性：PROPAGATION

REQUIRED：支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的；

SUPPORTS：支持当前事务，如果当前没有事务，就以非事务的方式执行；

MANDATORY：委托，支持当前事务，如果当前没有事务，就抛出异常；

REQUIRES\_NEW：新建事务，如果当前存在事务，把当前事务挂起；

NOT\_SUPPORT：以非事务的方式执行，如果当前存在事务，就把当前事务挂起；

NEVER：以非事务方式执行，如果当前存在事务，则抛出异常；

NESTED：如果当前存在事务，则在嵌套事务内执行，如果当前没有事务，则进行与REQUIRED类似的操作；

重点是要有两个方法：嵌套调用（传递）

5个隔离级别：

·若干个并发的事务之间的隔离程度

ISOLATION\_DEFAULT：这是PlatformTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别，另外四个与JDBC的隔离级别相对应；

#### 2.5.1ISOLATION\_READ\_UNCOMMITTED：读未提交（脏读）

|  |
| --- |
| # 查看数据库的隔离级别  SQL>show variables like ‘%tx\_isolation%’  # 设置事务隔离级别  SQL>set SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; |

脏读示例：

|  |
| --- |
| Session A>UPDATE account SET balance = balance - 50 where id = 1；  Session A>ROLLBACK； |
| Session B>SELECT \* FROM account where id = 1；  Java memory>int ab = 950；  Session B>UPDATE account SET balance = balance - 50 where id = 1；  预计：950 - 50 = 900（实则不然，会话A操作已经回滚，当前会话B更新时，原值是1000，即仍然是950） |

#### 2.5.2ISOLATION\_READ\_COMMITTED：读已提交（不可重复读）

|  |
| --- |
| # 设置事务隔离级别  SQL>set SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED; |

不会读取未提交的数据，防止脏读现象，但有幻读现象；

|  |
| --- |
| Session A>BEGIN;  Session A>UPDATE account SET balance = balance - 50 where id = 1；  Session A>COMMIT;  Session A>UPDATE account SET balance = balance - 50 where id = 1；  Session A>COMMIT; |
| Session B>BEGIN;  Session B>SELECT \* FROM account where id = 1；// 第一次读取950  Session B>SELECT \* FROM account where id = 1；// 第二次读取900  // 在同一个事物里面，读取到同一行数据产生了两个不同的结果 |

#### 2.5.3ISOLATION\_REAPEATABLE\_READ：可重复读

|  |
| --- |
| # 设置事务隔离级别  SQL>set SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ; |

默认可重复读，防止脏读和幻读

|  |
| --- |
| Session A>BEGIN;  Session A>UPDATE account SET balance = balance - 50 where id = 1；  Session A>COMMIT; |
| Session B>BEGIN;  Session B>SELECT \* FROM account where id = 1；// 第一次读取1000  Session B>SELECT \* FROM account where id = 1；// 第二次读取1000  Session B>COMMIT;  Session B>BEGIN; // 新建事务查询  Session B>SELECT \* FROM account where id = 1；// 第三次读取950 |

#### 2.5.4ISOLATION\_SERIALIZABLE：串行化

|  |
| --- |
| # 设置事务隔离级别  SQL>set SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE; |

锁表

|  |
| --- |
| Session A>BEGIN;  Session A>SELECT \* FROM account; |
| Session B>BEGIN;  Session B>UPDATE account SET balance = balance - 50 where id = 1; // 锁住  Session B>UPDATE account SET balance = balance - 50 where id = 1; // 锁住 |

### 2.6高并发下如何做到安全修改同一行数据

JDK锁（无法解决分布式场景下的同步）、分布式锁

·Synchronized

·lock（检查异常，需要防止finally中unlock）

·Zookeeper

分布式锁类型：

·数据库：性能差

·redis：会出现死锁

·ZooKeeper：

Zookeeper数据结构：

·类似Linux文件目录；

·每一个节点都有值；

Zookeeper底层提供了两个功能：

·管理（存储、读取）用户程序提交的数据；

·为用户程序提供数据节点监听服务（触发器）；

Zookeeper四种节点类型：

·PERSISTENT--持久化目录节点 # create /aaa 重复创建报错

客户端与Zookeeper断开后，该节点依旧存在；

·PERSISTENT\_SEQUENTIAL--持久化顺序编号目录节点 # create /aaa\_000001

客户端与Zookeeper断开后，该节点依旧存在，只是与Zookeeper给该节点进行顺序编号

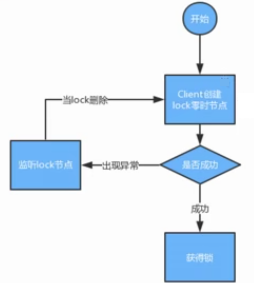
·EPHEMERAL--临时目录节点 # create /aaa

客户端与Zookeeper断开后，该节点被删除

·EPHEMERAL\_SEQUENTIAL--临时顺序编号目录节点 # create /aaa\_000001

客户端与Zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点进行顺序编号

Zookeeper基于异常的分布式锁（临时节点）



增加waitLock，CountDownLatch计数，ZK.handleDeleted(path

)监听到临时节点被删除CountDownLatch.countDown()最后删除监听。

Zookeeper基于相互监听分布式锁（性能比较高）（临时顺序编号节点）

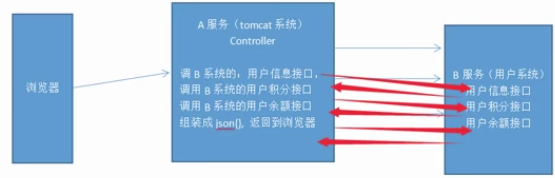
每个线程创建一个临时节点，判断当前排队是否是最小的，排在最前边的节点，是则获得锁，否则监听前边锁是否存在，使用getLock(seq-1)存在则等待，否则获取锁

释放锁：ZK.deleteTempNode

### 2.7A服务掉用B服务多个接口，响应时间最短方案

接口串行：缺点耗时

接口串行：

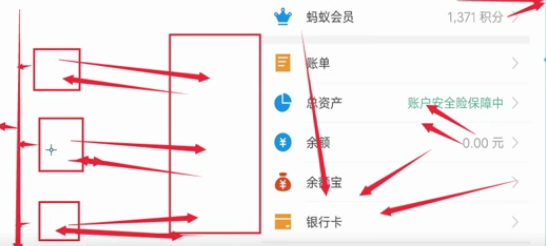


聚合接口查询：

·定义Callable任务

·创建FutureTask传入Callable任务，线程池提交FutureTask任务，并发task.get()获取结果；

·阻塞顺序获取任务结果，加入第一个接口耗时过长，将影响整体时长；取决于耗时最长的接口，其他人需要等待；



### 2.8A系统给B系统转100块钱，如何实现？

·分布式数据一致性；（事务注解@Transactional，挂住一个数据库连接不释放）

·性能优化（解决耗时事务操作占用连接）；（编程式事务@Autowired TransactionalTemplate）

·CAS锁（鼠标连点，幂等性）；（DB记录增加update \*\*\* set \*\*\*, version++; ，update影响1行，返回true，0行返回false）

### 2.9动态代理几种实现方式及优缺点

·使用反射和字节码的技术，在运行期间创建指定接口或类的子类（动态代理）以及其实例对象的技术，通过这个技术可以无侵入的为代码进行增加；

Java文件 ==》  直接从内存生成，而非Java文件

两种方式：

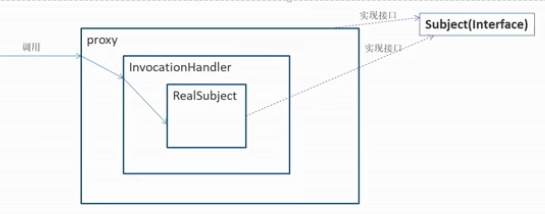
·JDK动态代理；

·CGLIB动态代理；

#### JDK动态代理：

·Proxy：是所有动态代理的父类，它提供了一个静态方法来创建动态代理的class对象和实例；

·InvocationHandler：每个动态代理实例都有一个关联的InvocationHandler，在代理实例上调用方法时，方法调用将被转发到InvocationHandler的method.invoke方法；



UserServiceInterceptor usi = new UserServiceInterceptor(); // 业务增强InvocationHandler

UserService proxy = Proxy.newProxyInstance(us.getClass().getClassLoader(), us.getClass().getInterfaces(), usi);

proxy.addUser();

·Proxy代理对象必须实现同一个被代理者接口

#### CGLIB动态代理：

Code Generation Library是一个基于ASM的字节码生成库，它允许我们在运行时，对字节码进行修改和动态生成，CGLib通过继承方式实现代理；

·Enhancer：来指定要代理的目标对象，实例处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor

·MethodInterceptor：动态代理对象的方法调用都会转到父类proxy.invokeSuper方法的真实业务代码；

Enhancer enhancer = new Enhancer();

enhancer.setSuperClass(UserService.class);

enhancer.setCallback(new UserServiceInterceptor());

UserServiceImpl usi= (UserServiceImpl ) enhancer.create();

usi.addUser();

总结：

·JDK动态代理是Java原生支持的，不需要任何外部依赖，但它只能基于接口进行代理；

·CGLib代理通过继承方式进行代理，无论目标对象有没有实现接口都可以代理，但无法增强final的方法，只能调用，不能增强

|  |
| --- |
| 因为cglib的子类是增强类，增强类没继承 fun2私有方法，所以fun1调的是父类的方法fun2，而不是增强类。  所以私有fun2 没被拦截。 |

### 2.10多线程下读概率远远大于写概率，如何解决并发问题？

·多线程下保证安全，要使用Synchronized内置锁，但加锁会有什么问题？

·对于读多写少的业务场景，我们如何解决；

·使用volatile关键字（一写多读，单例）；

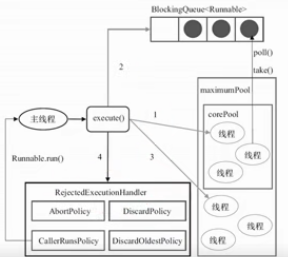
·使用读写锁（多读多写，读写锁，读写分离，订单下发队列）；

·使用写时复制容器（CopyOnWrite系列）（很少写，很多读，读不加锁，黑名单更新，仅仅保证数据一致性，不保证实时性）；

### 2.11线程池内部机制，当提交新任务是线程池是如何处理的？

·为什么要使用线程池？

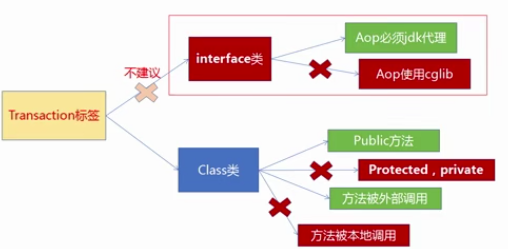
1. T1创建线程，T2提交任务，T3执行任务；并行执行
2. 资源管理，以更少的资源发挥最大的功能；



### 2.12@Transaction注解

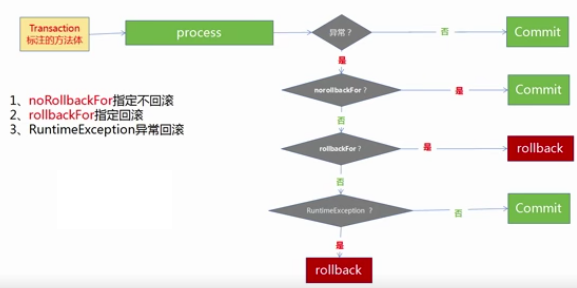
·对业务代码无侵入的事务控制；

·一般写在什么位置？



1. 写在接口上，如果项目是CGLib代理，将无法实现事务，和原调用逻辑一致；
2. 写在protected方法，事务不生效，数据被写入；
3. 写在使用this本地方法的操作，事务不生效，数据被写入；

·如何控制回滚？



### 2.13Spring的IOC容器初始化流程

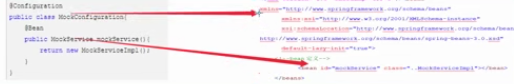
1. IOC容器是什么？HashMap对象；
2. 根据beans.xml来初始化bean，refresh（Spring3/4/5）；
3. 首先创建Bean工厂，实际上就是一个Map容器，BeanFactory.getBean(“id”)；初始化监听，加载bean，前后置处理器PostProcessorInvocationHandler，afterInitialBeanPostProcessor
4. Bean的初始化，赋值createBeanInstance(beanName, mbd, args); ==> Person person = new Person(); person.setName();
5. finishRefresh，执行监听的事件
6. Close，关闭容器，销毁bean，关闭bean工厂，map清空释放内存；

### 2.14SpringBoot启动机制

1. maven父子继承整合第三方依赖；
2. 怎么做到无配置文件集成SpringMVC？

类全路径包含：WebMVCAutoConfiguration

可以通过@EnableWebMvc集成SpringMVC的功能



**@SpringBootApplication**的组成：

·@SpringBootConfiguration：本质就是@Configuration，本身其实也是一个Spring容器的配置类；

·@ComponentScan：自动扫描并加载符合条件的组件（比如@Component和@Repository）或Bean定义，最终将这些bean定义加载到Spring容器中；

·@EnableAutoConfiguration：借助@Import的支持，手记和注册特定场景相关的bean定义，@Import(AutoConfigurationImportSelector.class)，读springboot-autoconfiguration包中的METE-INF/spring.factories中的类全路径；

1. 怎么内嵌tomcat容器？

类全路径包含：ServletWebServerFactoryAutoConfiguration

### 2.15Redis高性能的原因

1. 内存；
2. Key/value，O(1)；
3. 单线程，在内存中操作IO阻塞很少，无需多线程读写，减少线程上下文切换的开销；
4. 多路复用；
5. RESP；set length 5 ==> \*3 $3 set $6 length $1 5

### 2.16如何控制缓存更新

Tips:缓存信息的本质，就是硬盘数据的副本，归根结底是一种空间换时间的计数，数据一致性问题不可避免！

1. CacheService，SpringCache框架继承，pom中依赖spring-data-redis / jedis;
2. Bean.xml/ @Configuration中配置数据库连接池，redisTemplate，cacheManager

|  |
| --- |
| <bean id=”cacheManager” class=”org.springframework.data.redis.cache.RedisCacheManager>  <constructor-arg index=”0” ref=”redisTemplate” />  <property name=”expires”>  <map>  <entry key=”amount” value=”100” />  <entry key=”amount\_backup” value=”1000” />  </map>  </property>  </bean> |

1. @Autowired CacheService：cacheResult获取，cacheRemove删除，cachePut更新或添加，屏蔽底层缓存具体中间件不同实现；

解决缓存数据一致性问题方案：



1. 数据库实时同步失效：

·淘汰；

·更新；

·同步；

·缓存失效；

并发请求更新共享数据，设计用户粒度的锁，ConcurrentHashMap存放ReentrantLock

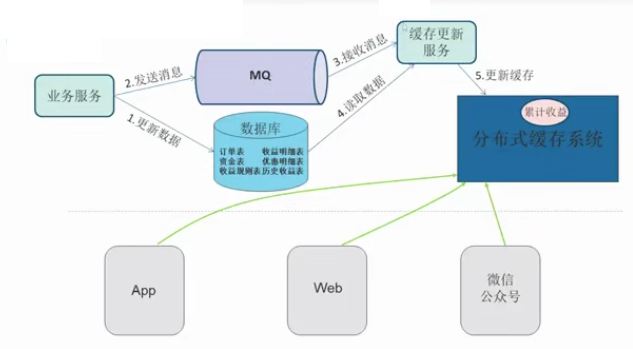


tryLock获取锁getLocalForKey获取锁：putIfAbsent成功，preLock == null ? newLock : preLock返回锁，失败则放回上次创建的锁；

releaseLock释放锁：getLocalForKey获取锁，获取到并且lock.isHeldByCurrentThread()是当前线程持有，则释放锁；

并发控制可以采用Samphone，控制同时请求的数量

（2）缓存与业务解耦，准一致性



activeMQ：web8161

rabbitMQ：web15672

（3）任务调度：报表，定时同步

### 2.17浅谈Http和Https有什么区别？

·http明文传输：数据拦截、篡改、攻击

·https密文传输：数据加密、身份验证、数据完整性

对称与非对称加密：

·对称：加解密同一个秘钥，容易泄露；

·非对称：公钥全世界都知道，私钥只有具体目标对象；

HTTPS比HTTP多出S的事情：

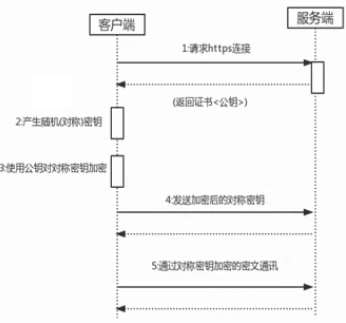
·请求https连接获取证书（公钥）；

·客户端给服务器发送（对称加密公钥）：随机数的密文；

·客户端同时给服务端发送（对称加密公钥）：随机数+私钥的密文；

·服务器根据公钥解密出随机数，同时解密出私钥；

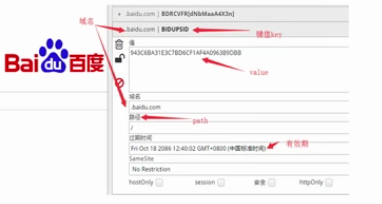
·客户端使用非对称加密进行数据传输，客户端使用公钥加密，私钥解密；



### 2.18Cookie和Session机制

**Cookie**

1. Cookie是为会话存储的键值信息；key / value
2. Cookie不可跨域名（只能拿到当前域名下的Cookie，包含父级域名）；
3. Cookie有有效期机制；
4. Path携带Cookie的请求路径（/ 表示全部请求）



**Session**

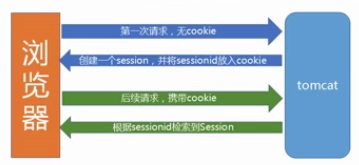
1. Session是服务器端基于内存的缓存技术，用来保存每个用户的会话技术；
2. 通过SessionID来区分用户，用户只要连接到服务器，服务器就会为之分配一个唯一的SessionID；
3. Session也有失效时间。（超过时限未得到过连接的session将被销毁）；
4. Session用法：HttpSession session = request.getSession()

**Cookie与Session的交互**

1. 服务器通过response.setCookie设置客户端Cookie属性，通常将sessionID设置进去，下次客户端访问如果存在SessionID，则识别出同一个用户再次请求的连接；

如：Set-Cookie：jsession=5645s4s6d4fs56df46sdf; Domain=dev.com; Path=/

1. 浏览器接收response Header的cookie值被保存，下次请求的request Header中，携带次cookie信息，如：Cookie: jsession=5645s4s6d4fs56df46sdf;
2. sessionID被服务器设置到Cookie中，以后浏览器的请求皆携带sessionID，服务器据此检索对应的session；
3. 当程序请求session时，首先根据cookie中携带的sessionid检索session（检索不到，会新建一个）



**总结**

1. Cookie存于客户端浏览器，session数据存放服务器；
2. Cookie不是很安全，别人分析存放在本地的Cookie并进行欺骗，考虑到安全应使用session；
3. Session存放于服务端，访问增多时，会影响服务器性能，考虑到减轻服务器性能方面，应当使用Cookie；
4. 单个Cookie保存不超过4k，很多浏览器最多缓存200个；
5. 当客户端禁止Cookie：一般使用URL重写，就是把 SessionID直接附在URL后面；

### 2.19什么是一致性Hash算法？

分布式架构缓存处理：

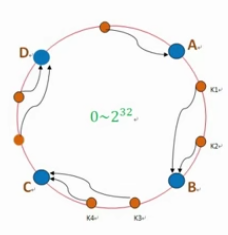
Hash算法分散数据存储：hash(n) % 4

同时也可以快速查找数据而不用遍历所有的服务器；

业务拓展缓存服务器加一台

要么缓存服务器数据全部需要重新计算存储--hash(n)%5

要么遍历所有缓存服务器；



Hash环

一致性Hash算法是对2^32取模，对服务器确定次数据在环上的位置；

数据存放

数据进来后对2^32取模，得到一个K1，在Hash环中顺时针找到服务器节点；

B服务器失效，将B的数据迁移至C即可，对于原本散列在A和D的数据，不需要做任何改变；

总结

一致性Hash算法通过减少影响范围的方式解决了增减服务器导致的数据散列问题，从而解决分布式下负载均衡问题

### 2.20MQ有可能发生重复消费，如何解决？





**为何会产生重复消息？**

超时重传/网络故障，消息生产者发送消息至队列，MQ发送异常没有给生产者应答

消费超时没有及时应答；

消费完成后，消息队列更新消息状态失败

**重复消费带来的问题和如何解决？**

重复扣款，重复下业务

幂等性操作：

Update table\_a set count = 10 where id = 1;

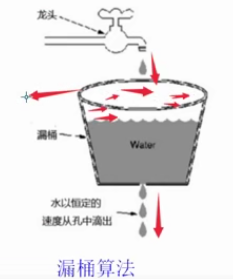
非幂等性：

Update table\_a set count = count + 1 where id =1;

CAS乐观锁：

1. 数据库新增版本号字段，带版本号去更新共享数据，更新失败重新获取版本号；
2. 去重表，数据库的唯一性索引；

### 2.21如何做限流策略，令牌桶和漏洞算法使用场景



当水请求大，会溢出，能强行限制调用的速度



突发流量的令牌保证

令牌桶算法会以一个恒定的速度往令牌池放令牌，当池子慢了以后，拒绝服务

熔断机制

## 第3节知识巩固

### 3.1使用HashMap在什么情况会出现内存泄漏？

·实现hashCode和Equals方法；

·如何确认哪些是垃圾，需要被垃圾收集器回收，GC ROOTs和引用计数；

·先释放key-value，value中的类变量引用，存放于堆中，如果直接将key置为null，或重新赋值，则原始key对应的value则可能无法释放，通常清空HashMap采用clear方法；

·事实上，HashMap中的某一个特定key很难被清理掉，因为用map.clear()会清理掉整个map。但是HashMap的内存泄漏一般不会很严重，因为只有在极少数情况下，我们才想去手动释放一个key。

## 第4节2019Java面试题

### 4.1volatile的理解（缓存命中带来的问题）

1. 可见性：读取后其他线程修改，当前线程不可知，因为缓存命中且有效使用旧的值；
2. 原子性：同时读取到，写回内存相互覆盖影响，例如i++操作；同时读取到i，都将i+1写回，覆盖；
3. 指令重排序：分配地址-->初始化-->赋值-->使用，可能会出现重排序优化；有可能拿到没有初始化完成的对象；

Q：谈谈你对volatile的理解

1. volatile是Java虚拟机提供的轻量级的同步机制；1.1保证可见性；1.2不保证原子性；1.3禁止指令重排序（初始化/赋值/使用）；
2. JMM谈谈（可见/原子/有序）；

·JMM内存模型：



·JMM关于同步的规定：

·1 线程解锁前，必须把共享变量的值刷新回主内存；

·2 线程加锁前，必须读取主内存的最新值到自己的共享内存；

·3 加锁解锁是同一把锁；

1. 你在哪些地方用到过volatile；

·单例模式

·BUS总线通知其他线程，共享数据已经被修改，使缓存失效，重新从内存加载到缓存

|  |
| --- |
| **public class** VolatileDemo {    **public static void** main(String[] args) {  MyData myData = **new** MyData();  **new** Thread(()->{  System.***out***.println(Thread.*currentThread*().getName() + **"\t come in..."**);  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  myData.set60();  System.***out***.println(Thread.*currentThread*().getName() + **"\t come out..."** + myData.**number**);  }, **"t1"**).start();  **while** (myData.**number** == 0){  *// 直到myData.number不为0* }  System.***out***.println(Thread.*currentThread*().getName() + **"\t game over..."**);  } } |

### 4.2CAS底层原理（UnSage类+CAS思想/自旋）

Q：CAS知道吗？

·CAS源码：

|  |
| --- |
| public final int incrementAndGet() {  return unsafe.getAndAddInt(this, valueOffset, 1) + 1;  }  public final int getAndAddInt(Object var1, long var2, int var4) {  int var5;  do {  var5 = this.getIntVolatile(var1, var2);  } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));  return var5;  }  字段的定位：  JAVA中对象的字段的定位可能通过staticFieldOffset方法实现，该方法返回给定field的内存地址偏移量，这个值对于给定的filed是唯一的且是固定不变的。  getIntVolatile方法获取对象中offset偏移地址对应的整型field的值,支持volatile load语义。  getLong方法获取对象中offset偏移地址对应的long型field的值  /\*\*\*  \* 获取obj对象中offset偏移地址对应的整型field的值。  \* @param obj 包含需要去读取的field的对象  \* @param obj中整型field的偏移量  \*/  public native int getIntVolatile(Object obj, long offset);  /\*\*  \* 比较obj的offset处内存位置中的值和期望的值，如果相同则更新。此更新是不可中断的。  \*  \* @param obj 需要更新的对象  \* @param offset obj中整型field的偏移量  \* @param expect 希望field中存在的值  \* @param update 如果期望值expect与field的当前值相同，设置filed的值为这个新值  \* @return 如果field的值被更改返回true  \*/  public native boolean compareAndSwapInt(Object obj, long offset, int expect, int update); |

·CAS缺点：

1. 并发修改大时，CAS循环操作对CPU消耗较大；
2. 只能保证一个共享变量的原子操作；
3. ABA问题；

### 4.3原子类/原子引用

·ABA问题：狸猫换太子/挪用公款补齐/信用卡套现，只管开头和结尾一致即可，中间有修改无所谓；

|  |
| --- |
| **public class** ABAOperation {  **public static void** main(String[] args) {  **new** Thread(()->{  AtomicInteger atomicInteger = **new** AtomicInteger(5);  System.***out***.println(Thread.*currentThread*().getName() + **"期望从5改成2019, "** +atomicInteger.compareAndSet(5, 2019) + **"\t 当前值："** + atomicInteger.get());  System.***out***.println(Thread.*currentThread*().getName() + **"从2019再改回去5, "** +atomicInteger.compareAndSet(2019, 5) + **"\t 当前值："** + atomicInteger.get());  }, **"t1"**).start();   **new** Thread(()->{  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  AtomicInteger atomicInteger = **new** AtomicInteger(5);  System.***out***.println(Thread.*currentThread*().getName() + **"期望从5改成100, "** +atomicInteger.compareAndSet(5, 100) + **"\t 当前值："** + atomicInteger.get());  }, **"t2"**).start();   **try** { TimeUnit.***SECONDS***.sleep(3); } **catch** (InterruptedException e) {e.printStackTrace();}  **new** Thread(()->{   }, **"t1"**).start();  } } |

·原子引用更新

|  |
| --- |
| **public class** AtomicReferenceOperation {  **private static** AtomicReference<User> *atomicReference* = **new** AtomicReference<>();  **static** User *bb* = **new** User(**"bb"**, 14);  **private static** AtomicStampedReference<User> *atomicReference1* = **new** AtomicStampedReference<User>(*bb*,1);    **public static void** main(String[] args) {  User zhangsan = **new** User(**"ZS"**, 14);  User lisi = **new** User(**"LS"**, 15);  */\*atomicReference.set(zhangsan);   System.out.println(atomicReference.compareAndSet(zhangsan, lisi) + "\t 修改后的值：" + atomicReference.get());  \*/* **new** Thread(()->{  **int** stamp = *atomicReference1*.getStamp();  System.***out***.println(Thread.*currentThread*().getName() + **"\t当前版本号："**+stamp);  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(*atomicReference1*.compareAndSet(*bb*, zhangsan, stamp, ++stamp));  System.***out***.println(Thread.*currentThread*().getName() + **"\t第1次修改版本号："**+*atomicReference1*.getStamp());  System.***out***.println(*atomicReference1*.compareAndSet(zhangsan, *bb*, *atomicReference1*.getStamp(), *atomicReference1*.getStamp() + 1));  System.***out***.println(Thread.*currentThread*().getName() + **"\t第2次修改版本号："**+*atomicReference1*.getStamp());  System.***out***.println(*atomicReference1*.getReference());   }, **"t1"**).start();   **new** Thread(()->{  **int** stamp = *atomicReference1*.getStamp();  System.***out***.println(Thread.*currentThread*().getName() + **"\t当前版本号："**+stamp);  **try** { TimeUnit.***SECONDS***.sleep(3); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(*atomicReference1*.compareAndSet(*bb*, lisi, stamp, ++stamp));  System.***out***.println(*atomicReference1*.getReference());  }, **"t2"**).start();  }   } @Data **class** User{  **private** String **name**;  **private** Integer **age**;   **public** User(String name, Integer age){  **this**.**name** = name;  **this**.**age** = age;  } } |

·如何规避ABA问题：

|  |
| --- |
| **public class** ABAFixOperation {   **private static** AtomicReference<User> *atomicReference* = **new** AtomicReference<>();  *// private static Integer enjoy = new Integer(100);* **private static** AtomicStampedReference<Integer> *atomicStampedReference* = **new** AtomicStampedReference<Integer>(100,1);   **public static void** main(String[] args) {  **new** Thread(()->{  AtomicInteger atomicInteger = **new** AtomicInteger(5);  System.***out***.println(Thread.*currentThread*().getName() + **"期望从5改成2019, "** +atomicInteger.compareAndSet(5, 2019) + **"\t 当前值："** + atomicInteger.get());  System.***out***.println(Thread.*currentThread*().getName() + **"从2019再改回去5, "** +atomicInteger.compareAndSet(2019, 5) + **"\t 当前值："** + atomicInteger.get());  }, **"t1"**).start();   **new** Thread(()->{  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  AtomicInteger atomicInteger = **new** AtomicInteger(5);  System.***out***.println(Thread.*currentThread*().getName() + **"期望从5改成100, "** +atomicInteger.compareAndSet(5, 100) + **"\t 当前值："** + atomicInteger.get());  }, **"t2"**).start();  **try** { TimeUnit.***SECONDS***.sleep(3); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(**"=========以下是ABA的问题解决=========="**);   **new** Thread(()->{  **int** stamp = *atomicStampedReference*.getStamp();   System.***out***.println(Thread.*currentThread*().getName() + **"\t第1次版本号："** + *atomicStampedReference*.getStamp());  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  *atomicStampedReference*.compareAndSet(100,2019,*atomicStampedReference*.getStamp(),*atomicStampedReference*.getStamp()+1);  System.***out***.println(Thread.*currentThread*().getName() + **"\t第2次版本号："** + *atomicStampedReference*.getStamp());  **boolean** result = *atomicStampedReference*.compareAndSet(2019,100,*atomicStampedReference*.getStamp(),*atomicStampedReference*.getStamp()+1);  System.***out***.println(Thread.*currentThread*().getName() + **"\t第3次版本号："** + *atomicStampedReference*.getStamp() + **"\t"** + result);  }, **"t3"**).start();   **new** Thread(()->{  **int** stamp = *atomicStampedReference*.getStamp();  System.***out***.println(Thread.*currentThread*().getName() + **"\t第1次版本号："** + stamp);  **try** { TimeUnit.***SECONDS***.sleep(3); } **catch** (InterruptedException e) {e.printStackTrace();}  **boolean** result = *atomicStampedReference*.compareAndSet(100,101,stamp,stamp+1);  System.***out***.println(Thread.*currentThread*().getName() + **"\t修改成功否："** + result + **"\t当前最新版本号："** + *atomicStampedReference*.getStamp());  System.***out***.println(**"当前值为："** +*atomicStampedReference*.getReference());   }, **"t4"**).start();   } } |

### 4.4集合不安全类

#### 4.4.1并发修改异常

|  |
| --- |
| **public class** ContainerUnSafeDemo {  **public static void** main(String[] args) {  *// List<String> list = new ArrayList<>(); // UnSafe  // List<String> list = new Vector<>(); // Safe  // List<String> list = Collections.synchronizedList(new ArrayList<>()); // Safe* List<String> list = **new** CopyOnWriteArrayList<>(); *// Safe*   **for** (**int** i = 0; i < 30; i++) {  **new** Thread(()->{  list.add(UUID.*randomUUID*().toString().substring(0,8));  System.***out***.println(list);  },String.*valueOf*(i)).start();  }   */\*\*  \* 1 故障现象  \* java.util.ConcurrentModificationException  \*  \* 2 导致原因  \* 并发修改；（串行化）  \* 共享区域；（如ConcurrentHashMap每个桶相互非共享）  \*  \*  \* 3 解决方案  \* 3.1 new Vector<>();  \* 3.2 Collections.synchronizedList(new ArrayList<>());  \* 3.3 new CopyOnWriteArrayList<>();  \*/* } } |

#### 4.4.2写时复制CopyOnWriteArrayList

·并发修改共享数据时，使用写锁，但读数据无需加锁，实现读写分离的ArrayList

|  |
| --- |
| */\*\*  \* Appends the specified element to the end of this list.  \*  \** ***@param e*** *element to be appended to this list  \** ***@return*** *{****@code*** *true} (as specified by {****@link*** *Collection#add})  \*/* **public boolean** add(E e) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  Object[] newElements = Arrays.*copyOf*(elements, len + 1);  newElements[len] = e;  setArray(newElements);  **return true**;  } **finally** {  lock.unlock();  } } |

#### 4.4.3HashSet（底层HashMap）与CopyOnWriteArraySet

·HashSet底层原理：

（1）HashMap；

（2）只使用key，value为private static final Object PRESENT = new Object();

|  |
| --- |
| Set<String> list = **new** CopyOnWriteArraySet<>(); *// Safe* **for** (**int** i = 0; i < 30; i++) {  **new** Thread(()->{  list.add(UUID.*randomUUID*().toString().substring(0,8));  System.***out***.println(list);  },String.*valueOf*(i)).start();  } |

#### 4.4.4HashMap（底层原理）与ConcurrentHashMap

|  |
| --- |
| Map<String, String> container = **new** ConcurrentHashMap<>(); *// Safe* **for** (**int** i = 0; i < 30; i++) {  **new** Thread(()->{  container.put(Thread.*currentThread*().getName(),UUID.*randomUUID*().toString().substring(0,8));  System.***out***.println(container);  },String.*valueOf*(i)).start(); } |

### 4.5PassByValue值传递/引用传递/常量池

·何为永久区？

主要是JVM在运行过程中，存放Class的静态信息，Main方法，常量，静态方法变量，共享变量等信息。一般很少被JVM进行回收。一般的动态替换Class的行为都是在这个区域来进行的。

JDK1.8

|  |
| --- |
| **public class** TestString2 {   **public static void** main(String[] args) {  *//* ***TODO Auto-generated method stub*** List<String> list = **new** ArrayList<String>();   **int** i = 0;   **while**(**true**){   list.add(String.*valueOf*(i++).intern());   }   } } |
| **-Xmx5m** |
| **java.lang.OutOfMemoryError: Java heap space** |
|  |
| **public class** OOMTest {    **public static void** main(String[] args) {  System.***out***.println(**"Let us do it now....."**);  **for**(**int** i=0;i<100000;i++){  Enhancer enhancer = **new** Enhancer();  enhancer.setSuperclass(BaseFlyer.**class**);  enhancer.setCallbackTypes(**new** Class[] {  Dispatcher.**class**, MethodInterceptor.**class** });  enhancer.setCallbackFilter(**new** CallbackFilter() {  **public int** accept(Method method) {  **return** 1;  }  });    Class clazz = enhancer.createClass();    System.***out***.println(**"Time:"** + System.*currentTimeMillis*());   }  } } |
| **JVM参数设置： -XX:MetaspaceSize=5M -XX:MaxMetaspaceSize=7M -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:\oom.dump** |
| **-XX:MinMetaspaceFreeRatio=<NNN>，<NNN>是一次GC以后，为了避免增加元数据区（高水位）的大小，空闲的类元数据区的容量的最小比例，不够就会导致垃圾回收。**  **-XX:MaxMetaspaceFreeRatio=<NNN>，<NNN>是一次GC以后，为了避免减少元数据区（高水位）的大小，空闲的类元数据区的容量的最大比例，超过就会导致垃圾回收。** |
| **java.lang.OutOfMemoryError: Metaspace** |

#### 永久代/元数据区

在JDK 7中开始进行移除永久代的努力，下面列出了JDK7中从永久带移除的东西：

·符号引用被移到了native堆

·池化string对象被移到了java堆

·Class对象、静态变量被移到了java堆

JDK 7下的永久区参数设置

|  |
| --- |
| **public class** OOMTest {    **public static void** main(String[] args) {  System.***out***.println(**"Let us do it now....."**);  **for**(**int** i=0;i<100000;i++){  Enhancer enhancer = **new** Enhancer();  enhancer.setSuperclass(BaseFlyer.**class**);  enhancer.setCallbackTypes(**new** Class[] {  Dispatcher.**class**, MethodInterceptor.**class** });  enhancer.setCallbackFilter(**new** CallbackFilter() {  **public int** accept(Method method) {  **return** 1;  }  });    Class clazz = enhancer.createClass();    System.***out***.println(**"Time:"** + System.*currentTimeMillis*());   }  } } |
| 参数设置示例： -XX:PermSize=5M -XX:MaxPermSize=7M |
| 说明：  PermSize为永久区大小， MaxPermSize为最大的永久区大小。  代码示例:  JVM参数： -XX:PermSize=5M -XX:MaxPermSize=7M -XX:  +HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=d:\oom.dump |
| java.lang.OutOfMemoryError: PermGen space |

### 4.6Java锁

·公平锁

·非公平锁

·可重入锁

·锁递归

·自旋锁

#### 4.6.1公平与非公平锁

·区别：

1. 公平锁：多个线程按照申请锁顺序获取锁，类似排队打饭，先来后到；
2. 非公平锁：不按顺序申请，来了先抢占锁，抢不到再FIFO排队；高并发情况下，有可能出现优先级反转/线程饥饿；
3. 并发包中ReentrantLock的创建可以指定构造函数的入参boolean来获得公平锁或非公平锁，默认非公平锁；

#### 4.6.2可重入锁/递归锁

·可重入锁（也叫递归锁）

1. 同一个线程外层函数获得所，内存递归函数扔能获取锁；
2. 进入内存函数自动获取锁；
3. 即，线程可以进入任何一个它已经拥有的锁，同步着的代码块；

|  |
| --- |
| **public class** ReentrantLockTest {  **private static** ReentrantLock *lock* = **new** ReentrantLock(**false**);  **public static void** main(String[] args) {  *sendSMS*();  **new** Thread(()->{  *sendSMS*();  }, **"t1"**).start();  **new** Thread(()->{  *sendSMS*();  }, **"t2"**).start();  }   **private static void** sendSMS(){  *lock*.lock();  **try** {  System.***out***.println(**"sendSMS..."**);  *sendEmail*();  } **finally** {  *lock*.unlock();  }  }   **private static void** sendEmail(){  *lock*.lock();  *lock*.lock();  **try** {  System.***out***.println(**"#####sendEmail..."**);  } **finally** {  *lock*.unlock();  *// lock.lock();* }  } } |
| **名称: t1**  **状态: java.util.concurrent.locks.ReentrantLock$NonfairSync@260536ed上的WAITING, 拥有者: main**  **总阻止数: 0, 总等待数: 1** |
| **名称: t2**  **状态: java.util.concurrent.locks.ReentrantLock$NonfairSync@260536ed上的WAITING, 拥有者: main**  **总阻止数: 1, 总等待数: 1** |
| **堆栈跟踪:**  **sun.misc.Unsafe.park(Native Method)**  **java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)**  **java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:836)**  **java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:870)**  **java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1199)**  **java.util.concurrent.locks.ReentrantLock$NonfairSync.lock(ReentrantLock.java:209)**  **java.util.concurrent.locks.ReentrantLock.lock(ReentrantLock.java:285)**  **com.byf.interview.lock.ReentrantLockTest.sendSMS(ReentrantLockTest.java:18)**  **com.byf.interview.lock.ReentrantLockTest.lambda$main$1(ReentrantLockTest.java:13)**  **com.byf.interview.lock.ReentrantLockTest$$Lambda$2/186370029.run(Unknown Source)**  **java.lang.Thread.run(Thread.java:748)** |

#### 4.6.3自旋锁SpinLock

指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，较少线程上下文切换，循环消耗CPU

|  |
| --- |
| **public final int** getAndAddInt(Object var1, **long** var2, **int** var4) {  **int** var5;  **do** {  var5 = **this**.getIntVolatile(var1, var2);  } **while**(!**this**.compareAndSwapInt(var1, var2, var5, var5 + var4));   **return** var5; } |

AtomicReference自旋锁实现：

|  |
| --- |
| **public class** SpinLockTest {  AtomicReference<Thread> **atomicReference** = **new** AtomicReference<>(**null**);   **private void** myLock(){  Thread thread = Thread.*currentThread*();  System.***out***.println(Thread.*currentThread*().getName() + **"\t come in..."**);  **while** (!**atomicReference**.compareAndSet(**null**, thread)){  *// 自旋* System.***out***.print(**"."**);  **try** { TimeUnit.***MILLISECONDS***.sleep(200); } **catch** (InterruptedException e) {e.printStackTrace();}  }  System.***out***.println();  }   **private void** myUnLock(){  Thread thread = Thread.*currentThread*();  System.***out***.println(**"\n"** + Thread.*currentThread*().getName() + **"\t come out..."**);  **atomicReference**.compareAndSet(thread, **null**);  }   **public static void** main(String[] args) {  SpinLockTest lock = **new** SpinLockTest();  **new** Thread(()->{  lock.myLock();  **try** { TimeUnit.***SECONDS***.sleep(5); } **catch** (InterruptedException e) {e.printStackTrace();}  lock.myUnLock();  }, **"t1"**).start();   **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}   **new** Thread(()->{  lock.myLock();  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  lock.myUnLock();  }, **"t2"**).start();  } } |
| **t1 come in...**  **t2 come in...**  **....................//t1持有锁，t2自旋**  **t1 come out...**  **t2 come out...** |

#### 4.6.4读写锁

独占锁：指该锁一次只能被一个线程所持有。对ReentrantLock和Synchronized而言都是独占锁

共享锁：该锁可被多个线程所持有

对ReentrantReadWriteLock其读锁是共享锁，其写锁是独占锁。

读锁的共享锁可保证并发读是非常高效的，读写，写读，写写的过程是互斥的。

Java面向对象特征：继承、多态、封装

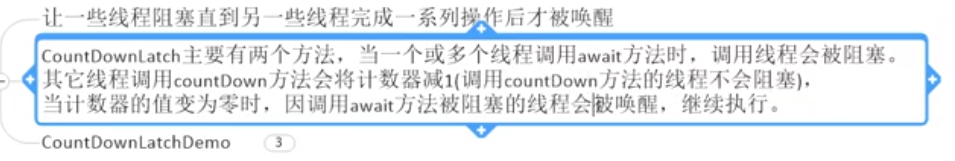
缓存特征：读、写、清空

|  |
| --- |
| **class** MyCache {  **private volatile** Map<String, String> **map** = **new** HashMap<>();  **private** ReentrantReadWriteLock **rwLock** = **new** ReentrantReadWriteLock();   **public void** put(String key, String value) {  **rwLock**.writeLock().lock();  **try** {  System.***out***.println(Thread.*currentThread*().getName() + **"\t 正在写入："** + key);  **try** {  TimeUnit.***MILLISECONDS***.sleep(300);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **map**.put(key, value);  System.***out***.println(Thread.*currentThread*().getName() + **"\t 写入完成。"**);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **rwLock**.writeLock().unlock();  }  }   **public void** get(String key) {  **rwLock**.readLock().lock();  **try** {  System.***out***.println(Thread.*currentThread*().getName() + **"\t 正在读取："** + key);  **try** {  TimeUnit.***MILLISECONDS***.sleep(300);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  String result = **map**.get(key);  System.***out***.println(Thread.*currentThread*().getName() + **"\t 读取完成："** + result);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **rwLock**.readLock().unlock();  }  } } **public class** ReadWriteLockTest {  **public static void** main(String[] args) {  MyCache myCache = **new** MyCache();  **for** (**int** i = 1; i <= 5; i++) {  **final** String j = String.*valueOf*(i);  **new** Thread(()->{  myCache.put(j,j);  },**"wt"**+String.*valueOf*(i)).start();  }   **for** (**int** i = 1; i <= 5; i++) {  **final** String j = String.*valueOf*(i);  **new** Thread(()->{  myCache.get(j);  },**"rt"**+String.*valueOf*(i)).start();  }  } } |
| **wt1 正在写入：1**  **wt1 写入完成。**  **wt2 正在写入：2**  **wt2 写入完成。**  **wt3 正在写入：3**  **wt3 写入完成。**  **wt4 正在写入：4**  **wt4 写入完成。**  **wt5 正在写入：5**  **wt5 写入完成。**  **rt1 正在读取：1**  **rt2 正在读取：2**  **rt3 正在读取：3**  **rt4 正在读取：4**  **rt5 正在读取：5**  **rt1 读取完成：1**  **rt2 读取完成：2**  **rt5 读取完成：5**  **rt3 读取完成：3**  **rt4 读取完成：4** |

### 4.7J.U.C工具方法

#### 4.7.1CountDownLatch（减法）

|  |
| --- |
| @AllArgsConstructor @ToString **public enum** CountryEnum {  ***ONE***(1,**"齐"**),  ***TWO***(2,**"楚"**),  ***THREE***(3,**"燕"**),  ***FOUR***(4,**"赵"**),  ***FIVE***(5,**"魏"**),  ***SIX***(6,**"韩"**);  @Getter **private** Integer **countryCode**;  @Getter **private** String **countryMessage**;   **public static** CountryEnum iteratorEnums(**int** i){  CountryEnum[] arrays = CountryEnum.*values*();  **for** (CountryEnum countryEnum : arrays){  **if** (i == countryEnum.**countryCode**){  **return** countryEnum;  }  }  **return null**;  } } |
| **public class** CountDownLatchTest {   **public static void** main(String[] args) **throws** InterruptedException {  CountDownLatch latch = **new** CountDownLatch(5);  */\*for (int i = 1; i <= 5; i++) {  new Thread(()->{  System.out.println(Thread.currentThread().getName() + "\t 离开教室.");  latch.countDown();  },String.valueOf(i)).start();  }  latch.await();  System.out.println(Thread.currentThread().getName() + " 班长最后离开教室，关灯.");\*/* **for** (**int** i = 1; i <= 5; i++) {  **new** Thread(()->{  System.***out***.println(Thread.*currentThread*().getName() + **"\t国被灭."**);  latch.countDown();  },CountryEnum.*iteratorEnums*(i).getCountryMessage()).start();  }  latch.await();  System.***out***.println(Thread.*currentThread*().getName() + **"秦帝国一统天下."**);  System.***out***.println();  System.***out***.println(CountryEnum.***ONE***);  System.***out***.println(CountryEnum.***ONE***.getCountryCode());  System.***out***.println(CountryEnum.***ONE***.getCountryMessage());  } } |
| 齐 国被灭.  魏 国被灭.  赵 国被灭.  楚 国被灭.  燕 国被灭.  main秦帝国一统天下.  CountryEnum(countryCode=1, countryMessage=齐)  1  齐 |



1. 让一些线程阻塞直到另一些线程完成一系列操作后才被唤醒；
2. CountDownLatch主要有两个方法，当一个或多个线程调用await方法时，调用线程会被阻塞。
3. 其他线程调用countDown方法时会将计数器减1（调用countDown方法的线程不会阻塞）；
4. 当计数器清0，因调用await方法被阻塞的线程会被唤醒，继续执行；

#### 4.7.2CyclicBarrier（加法）

|  |
| --- |
| **public class** CyclicBarrierTest {    **public static void** main(String[] args) {  CyclicBarrier cyclicBarrier = **new** CyclicBarrier(7,()->{  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*召唤神龙"**);  });  **for** (**int** i = 1; i <= 7; i++) {  **final int** tmpi = i;  **new** Thread(()->{  **try** {  System.***out***.println(Thread.*currentThread*().getName() + **"\t收集到:"** + tmpi+ **"龙珠"**);  cyclicBarrier.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (BrokenBarrierException e) {  e.printStackTrace();  }  },String.*valueOf*(i)).start();  }  } } |
| **2 收集到:2龙珠**  **4 收集到:4龙珠**  **1 收集到:1龙珠**  **3 收集到:3龙珠**  **5 收集到:5龙珠**  **6 收集到:6龙珠**  **7 收集到:7龙珠**  **\*\*\*\*\*\*\*\*\*\*\*\*召唤神龙** |

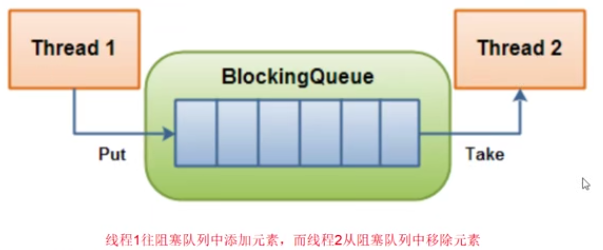
#### 4.7.3Semaphore（伸缩）

|  |
| --- |
| **public class** SemaphoreTest {  **public static void** main(String[] args) {  Semaphore semaphore = **new** Semaphore(3);  **for** (**int** i = 1; i <= 6; i++) {  **new** Thread(()->{  **try** {  semaphore.acquire();  System.***out***.println(Thread.*currentThread*().getName() + **"\t抢到车位"**);  **try** { TimeUnit.***SECONDS***.sleep(3); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(Thread.*currentThread*().getName() + **"\t停车3秒后离开车位"**);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  semaphore.release();  }  },String.*valueOf*(i)).start();  }  } } |
| **1 抢到车位**  **3 抢到车位**  **2 抢到车位**  **2 停车3秒后离开车位**  **3 停车3秒后离开车位**  **1 停车3秒后离开车位**  **4 抢到车位**  **6 抢到车位**  **5 抢到车位**  **6 停车3秒后离开车位**  **4 停车3秒后离开车位**  **5 停车3秒后离开车位** |

### 4.8阻塞队列

#### 4.8.1阻塞队列理论/接口结构和实现类

1. 当阻塞队列是空时，从队列中获取元素的操作将会被阻塞；
2. 当阻塞队列是满时，往队列中添加元素的操作将会被阻塞；



·为什么用？有什么好处？

多线程领域：所谓阻塞，在某些情况下挂起线程（即阻塞），一旦条件满足，被挂起的线程将被唤醒；

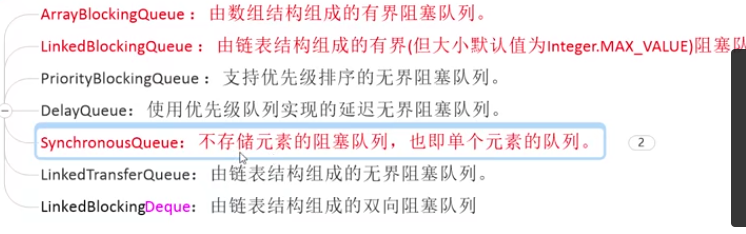
·为什么需要BlockingQueue

好处是不需要关注什么时候阻塞，什么时候唤醒，因为BlockingQueue都给你一手包办了；

在Concurrent包发布之前，多线程环境下，我们每个程序员都需要自行控制这些细节，尤其需要兼顾效率和线程安全；

·BlockQueue的9种实现种类

1. ArrayBlockingQueue：由数组结构阻塞的有界阻塞队列；
2. LinkedBlockingQueue：由链表结构组成的有界（但大小默认为Integer.MAX\_VALUE，21亿多）阻塞队列；
3. PriorityBlockingQueue：支持优先级排序的无界阻塞队列；
4. DelayQueue：使用优先级队列实现的延迟无界队列；
5. SynchronousQueue：不存储元素的阻塞队列，也即单个元素的队列；
6. LinkedTransferQueue：由链表结构组成的无界阻塞队列；
7. LinkedBlockingDeque：由链表结构组成的双向阻塞队列；



Collection<--Queue/BlockingQueue<--9种实现类

#### 4.8.2阻塞队列API之抛出异常组



|  |
| --- |
| */\*\*  \* ArrayBlockingQueue：数组结构的有界阻塞队列，FIFO排序；  \* LinkedBlockingQueue：链表结构的阻塞队列，FIFO排序，吞吐量高于ArrayBlockingQueue；  \* SynchronousQueue：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则阻塞，吞吐量通常高于LinkedBlockingQueue；  \*  \* 1 队列  \*  \* 2 阻塞队列  \* 2.1 阻塞队列有没有好的一面；  \*  \* 2.2 不得不阻塞，如何管理；  \*/* **public class** BlockingQueueTest {  **public static void** main(String[] args) {  BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);  System.***out***.println(blockingQueue.add(**"a"**));  System.***out***.println(blockingQueue.add(**"b"**));  System.***out***.println(blockingQueue.add(**"c"**));  System.***out***.println(blockingQueue.add(**"c"**));  } } |
| *true*  *true*  *true*  *Exception in thread "main" java.lang.IllegalStateException: Queue full*  *at java.util.AbstractQueue.add(AbstractQueue.java:98)*  *at java.util.concurrent.ArrayBlockingQueue.add(ArrayBlockingQueue.java:312)*  *at com.byf.interview.blockqueue.BlockingQueueTest.main(BlockingQueueTest.java:24)* |

|  |
| --- |
| **public class** BlockingQueueTest {  **public static void** main(String[] args) {  BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);  System.***out***.println(blockingQueue.add(**"a"**));  System.***out***.println(blockingQueue.add(**"b"**));  System.***out***.println(blockingQueue.add(**"c"**));  System.***out***.println(blockingQueue.remove());  System.***out***.println(blockingQueue.remove());  System.***out***.println(blockingQueue.remove());  System.***out***.println(blockingQueue.remove());  } } |
| true  true  true  a  b  c  Exception in thread "main" java.util.NoSuchElementException |

|  |
| --- |
| **public class** BlockingQueueTest {  **public static void** main(String[] args) {  BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);  System.***out***.println(blockingQueue.add(**"a"**));  System.***out***.println(blockingQueue.add(**"b"**));  System.***out***.println(blockingQueue.add(**"c"**));   System.***out***.println(**"队首元素："** + blockingQueue.element());   System.***out***.println(blockingQueue.remove());  System.***out***.println(blockingQueue.remove());  System.***out***.println(blockingQueue.remove());  } } |
| **true**  **true**  **true**  **队首元素：a**  **a**  **b**  **c** |

#### 4.8.3阻塞队列API之返回布尔类型组

|  |
| --- |
| BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3); System.***out***.println(blockingQueue.offer(**"a"**)); System.***out***.println(blockingQueue.offer(**"b"**)); System.***out***.println(blockingQueue.offer(**"c"**)); System.***out***.println(blockingQueue.offer(**"x"**)); |
| true  true  true  false |

|  |
| --- |
| BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3); System.***out***.println(blockingQueue.offer(**"a"**)); System.***out***.println(blockingQueue.offer(**"b"**)); System.***out***.println(blockingQueue.offer(**"c"**)); System.***out***.println(blockingQueue.offer(**"x"**)); System.***out***.println(blockingQueue.peek()); System.***out***.println(blockingQueue.poll()); System.***out***.println(blockingQueue.poll()); System.***out***.println(blockingQueue.poll()); System.***out***.println(blockingQueue.poll()); |
| true  true  true  false  a  a  b  c  null |

#### 4.8.4阻塞队列API之阻塞和超时控制

|  |
| --- |
| BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3); **try** {  blockingQueue.put(**"a"**);  blockingQueue.put(**"b"**);  blockingQueue.put(**"c"**);  System.***out***.println(**"========================"**);  blockingQueue.put(**"x"**); } **catch** (InterruptedException e) {  e.printStackTrace(); } |
| ========================  Process finished with exit code -1 |

|  |
| --- |
| BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3);  **try** {  blockingQueue.put(**"a"**);  blockingQueue.put(**"b"**);  blockingQueue.put(**"c"**);  System.***out***.println(**"========================"**); *// blockingQueue.put("x");* blockingQueue.take();  blockingQueue.take();  blockingQueue.take();  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  blockingQueue.take();  } **catch** (InterruptedException e) {  e.printStackTrace();  } |
| ========================  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

超时控制

|  |
| --- |
| BlockingQueue<String> blockingQueue = **new** ArrayBlockingQueue<>(3); System.***out***.println(blockingQueue.offer(**"a"**, 2, TimeUnit.***SECONDS***)); System.***out***.println(blockingQueue.offer(**"a"**, 2, TimeUnit.***SECONDS***)); System.***out***.println(blockingQueue.offer(**"a"**, 2, TimeUnit.***SECONDS***)); System.***out***.println(blockingQueue.offer(**"a"**, 2, TimeUnit.***SECONDS***)); |
| true  true  true  False// 2秒之后插入失败 |

#### 4.8.5阻塞队列之同步SynchronousQueue队列

1. SynchronousQueue没有容量；
2. 与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue；
3. 每一个put操作必须等待一个take操作，否则不能继续添加元素，反之亦然；

|  |
| --- |
| **public class** SynchronousQueueTest {   **public static void** main(String[] args) {  BlockingQueue<String> blockingQueue = **new** SynchronousQueue<>();  **new** Thread(()->{  **try** {  System.***out***.println(Thread.*currentThread*().getName() + **"\t put 1"** );  blockingQueue.put(**"1"**);  System.***out***.println(Thread.*currentThread*().getName() + **"\t put 2"** );  blockingQueue.put(**"2"**);  System.***out***.println(Thread.*currentThread*().getName() + **"\t put 3"** );  blockingQueue.put(**"3"**);   } **catch** (InterruptedException e) {  e.printStackTrace();  }  }, **"t1"**).start();   **new** Thread(()->{   **try** {  **try** { TimeUnit.***SECONDS***.sleep(5); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(Thread.*currentThread*().getName() + **"\t take "** + blockingQueue.take());  **try** { TimeUnit.***SECONDS***.sleep(5); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(Thread.*currentThread*().getName() + **"\t take "** + blockingQueue.take());  **try** { TimeUnit.***SECONDS***.sleep(5); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println(Thread.*currentThread*().getName() + **"\t take "** + blockingQueue.take());   } **catch** (InterruptedException e) {  e.printStackTrace();  }  }, **"t2"**).start();  } } |
| **t1 put 1**  **t2 take 1**  **t1 put 2**  **t2 take 2**  **t1 put 3**  **t2 take 3** |

### 4.9线程间通信

#### 4.9.1生产消费模型/交替打印

|  |
| --- |
| */\*\*  \* 题目：一个初始值为0的变量，两个线程交替操作，一个加1一个减1，来5轮  \* 1 线程 操作 资源类  \* 2 判断 干活 通知  \* 3 防止虚假唤醒  \*/* **class** ShareData{  **private int number** = 0;  **private** Lock **lock** = **new** ReentrantLock();  **private** Condition **condition** = **lock**.newCondition();   **public void** increment() **throws** InterruptedException {  **lock**.lock();  **try** {  *// 判断 if有可能虚假唤醒* **while** (**number** != 0){  *// 等待，不能执行* **condition**.await();  }   *// 干活* **number**++;  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + **number**);   *// 通知* **condition**.signalAll();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock();  }  }   **public void** decrement() **throws** InterruptedException {  **lock**.lock();  **try** {  *// 判断 if有可能虚假唤醒* **while** (**number** == 0){  *// 等待，不能执行* **condition**.await();  }   *// 干活* **number**--;  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + **number**);   *// 通知* **condition**.signalAll();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock();  }  } } **public class** ProduceConsumerOldTest {   **public static void** main(String[] args) {  ShareData shareData = **new** ShareData();    **new** Thread(()->{  **for** (**int** i = 1; i <= 5; i++) {  **try** {  shareData.increment();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  },**"t1"**).start();   **new** Thread(()->{  **for** (**int** i = 1; i <= 5; i++) {  **try** {  shareData.decrement();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  },**"t2"**).start();   **new** Thread(()->{  **for** (**int** i = 1; i <= 5; i++) {  **try** {  shareData.increment();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  },**"t3"**).start();     **new** Thread(()->{  **for** (**int** i = 1; i <= 5; i++) {  **try** {  shareData.decrement();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  },**"t4"**).start();   } } |
| *t1 1*  *t2 0*  *t1 1*  *t2 0*  *t3 1*  *t2 0*  *t3 1*  *t2 0*  *t3 1*  *t2 0*  *t3 1*  *t4 0*  *t3 1*  *t4 0*  *t1 1*  *t4 0*  *t1 1*  *t4 0*  *t1 1*  *t4 0* |

#### 4.9.2synchronized和lock区别/按序打印

|  |
| --- |
| */\*\*  \* 题目：synchronized和Lock有什么区别？用新的Lock有什么好处？举例说明  \* 1 原始构成  \* synchronized是关键字属于JVM层面  \* monitorenter（底层通过monitor对象来完成，其实wait/notify等方法也依赖monitor对象只有在同步代码块中才能调用wait/notify等方法  \* monitorexit  \* Lock是具体类（java.util.concurrent.locks.Lock）是api层面的锁  \*  \* 2 使用方法  \* synchronized 不需要手动释放锁，当synchronized代码执行完成后，系统会自动让线程释放对象锁的占用；  \* ReentrantLock 需要用户手动释放锁，若没有主动释放锁，就有可能导致死锁；  \*  \* 3 等待是否可中断  \* synchronized 不可中断，除非抛出异常或者正常运行完成  \* ReentrantLock 可中断，1.设置超时方法tryLock(long timeout, TimeUnit unit)  \* 2.lockInterruptily()代码块，调用interrupt()方法可中断  \*  \* 4 加锁是否公平  \* synchronized 非公平锁  \* ReentrantLock 两种都可以实现，默认非公平，构造方法可以传入boolean值，true为公平锁，false非公平锁  \*  \* 5 绑定多个条件Condition  \* synchronized 没有  \* ReentrantLock用来实现分组唤醒需要唤醒的线程们，可以精确唤醒；而不是像synchronized要么随机唤醒一个，要么唤醒全部；  \*  \*  \* 题目：多线程之间顺序调用，实现A->B->C三个线程启动，要求实现：  \* AA打印5次，BB打印10次，CC打印15次  \* 紧接着  \* AA打印5次，BB打印10次，CC打印15次  \* ...  \* 来10轮  \*  \*/* **class** ShareResource {  **private int number** = 1; *// A:1 B:2 C:3* **private** Lock **lock** = **new** ReentrantLock();  **private** Condition **condition1** = **lock**.newCondition();  **private** Condition **condition2** = **lock**.newCondition();  **private** Condition **condition3** = **lock**.newCondition();   **public void** print5(){  **lock**.lock();  **try** {  *// 1 盘点* **while** (**number** != 1){  **condition1**.await();  }  *// 2 干活* **for** (**int** i = 1; i <= 5; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + i);  }  *// 通知* **number** = 2;  **condition2**.signal();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock();  }  }   **public void** print10(){  **lock**.lock();  **try** {  *// 1 盘点* **while** (**number** != 2){  **condition2**.await();  }  *// 2 干活* **for** (**int** i = 1; i <= 10; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + i);  }  *// 通知* **number** = 3;  **condition3**.signal();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock();  }  }   **public void** print15(){  **lock**.lock();  **try** {  *// 1 盘点* **while** (**number** != 3){  **condition3**.await();  }  *// 2 干活* **for** (**int** i = 1; i <= 15; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + i);  }  *// 通知* **number** = 1;  **condition1**.signal();  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock();  }  } } **public class** SynchronizedAndLockTest {   **public static void** main(String[] args) {  ShareResource shareResource = **new** ShareResource();  **new** Thread(()->{  **for** (**int** i = 0; i < 10; i++) {  shareResource.print5();  }  }, **"A"**).start();   **new** Thread(()->{  **for** (**int** i = 0; i < 10; i++) {  shareResource.print10();  }  }, **"B"**).start();   **new** Thread(()->{  **for** (**int** i = 0; i < 10; i++) {  shareResource.print15();  }  }, **"C"**).start();  } } |
| *A 1*  *A 2*  *A 3*  *A 4*  *A 5*  *B 1*  *B 2*  *B 3*  *B 4*  *B 5*  *B 6*  *B 7*  *B 8*  *B 9*  *B 10*  *C 1*  *C 2*  *C 3*  *C 4*  *C 5*  *C 6*  *C 7*  *C 8*  *C 9*  *C 10*  *C 11*  *C 12*  *C 13*  *C 14*  *C 15*  *A 1*  *...* |

锁绑定多个Condition

#### 4.9.3生产消费阻塞队列实现

|  |
| --- |
| */\*\*  \* volatile/CAS/AtomicInteger/BlockingQueue/线程交互/原子引用  \*/* **class** ShareResource{  **private volatile boolean FLAG** = **true**; *//默认开启，进行生产+消费* **private** AtomicInteger **atomicInteger** = **new** AtomicInteger();   BlockingQueue<String> **blockingQueue** = **null**;   **public** ShareResource(BlockingQueue<String> blockingQueue) {  **this**.**blockingQueue** = blockingQueue;  *// 日志排查传递的具体类型* System.***out***.println(blockingQueue.getClass().getName());  }   **public void** produce() **throws** Exception{  String data = **null**;  **boolean** result = **false**;  **while** (**FLAG**){  data = **atomicInteger**.getAndIncrement() + **""**;  result = **blockingQueue**.offer(data, 2, TimeUnit.***SECONDS***);  **if**(result){  System.***out***.println(Thread.*currentThread*().getName()+**"\t插入队列"**+data+**"成功"**);  } **else** {  System.***out***.println(Thread.*currentThread*().getName()+**"\t插入队列"**+data+**"失败"**);  }  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  }  System.***out***.println(Thread.*currentThread*().getName()+**"\t大老板叫停，表示FLAG=false，生产动作结束"**);  }   **public void** consume() **throws** Exception{  String result = **null**;  **while** (**FLAG**){  result = **blockingQueue**.poll(2, TimeUnit.***SECONDS***);  **if**(result == **null** || result.equalsIgnoreCase(**""**)){  **FLAG** =**false**;  System.***out***.println(Thread.*currentThread*().getName() + **"\t超过2秒没有取到，消费退出"**);  System.***out***.println();  System.***out***.println();  System.***out***.println();  **return**;  }  System.***out***.println(Thread.*currentThread*().getName() + **"\t\t消费队列"**+result+**"成功"**);  }  }   **public void** stop(){  **this**.**FLAG** = **false**;  }  } **public class** ProductorConsumerTest {  **public static void** main(String[] args) {  ShareResource shareResource = **new** ShareResource(**new** ArrayBlockingQueue<>(10));  **new** Thread(()->{  System.***out***.println(Thread.*currentThread*().getName() + **"\t生产线程启动"**);  System.***out***.println();  **try** {  shareResource.produce();  } **catch** (Exception e) {  e.printStackTrace();  }  }, **"Productor"**).start();   **new** Thread(()->{  System.***out***.println(Thread.*currentThread*().getName() + **"\t消费者线程启动"**);  System.***out***.println();  **try** {  shareResource.consume();  } **catch** (Exception e) {  e.printStackTrace();  }  }, **"Comsumer"**).start();   **try** { TimeUnit.***SECONDS***.sleep(5); } **catch** (InterruptedException e) {e.printStackTrace();}  System.***out***.println();  System.***out***.println();  System.***out***.println(**"5秒时间到，大老板叫停"**);  shareResource.stop();  } } |
| *java.util.concurrent.ArrayBlockingQueue*  *Productor 生产线程启动*  *Comsumer 消费者线程启动*  *Productor 插入队列0成功*  *Comsumer 消费队列0成功*  *Productor 插入队列1成功*  *Comsumer 消费队列1成功*  *Productor 插入队列2成功*  *Comsumer 消费队列2成功*  *Productor 插入队列3成功*  *Comsumer 消费队列3成功*  *Productor 插入队列4成功*  *Comsumer 消费队列4成功*  *5秒时间到，大老板叫停*  *Productor 大老板叫停，表示FLAG=false，生产动作结束*  *Comsumer 超过2秒没有取到，消费退出* |

#### 4.9.4Callable线程

|  |
| --- |
| **class** MyThread1 **implements** Callable<Integer>{  @Override  **public** Integer call() **throws** Exception {  System.***out***.println(**"\*\*\*\*\*\*come in, callbale"**);  **try** { TimeUnit.***SECONDS***.sleep(3); } **catch** (InterruptedException e) {e.printStackTrace();}  **return** 1024;  } }  **public class** CallableThreadTest {   **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  FutureTask<Integer> futureTask = **new** FutureTask<>(**new** MyThread1());  FutureTask<Integer> futureTask1 = **new** FutureTask<>(**new** MyThread1());   Thread t1 = **new** Thread(futureTask, **"t1"**);  Thread t2 = **new** Thread(futureTask1, **"t2"**);  t1.start();  t2.start();    System.***out***.println(Thread.*currentThread*().getName() + **"\t主线程计算中..."**);  **int** result1 = 100;   **while** (!futureTask.isDone()){   }   **int** result = futureTask.get();  */\*System.out.println("Callable 返回值：" + futureTask.get());\*/* System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\* 汇总结果："** + (result + result1));  } } |
| **main 主线程计算中...**  **\*\*\*\*\*\*come in, callbale**  **\*\*\*\*\*\*come in, callbale**  **\*\*\*\*\*\*\*\*\*\*\* 汇总结果：1124** |

### 4.10线程池

#### 4.10.1使用及优势

·线程池做的工作主要是控制运行的线程数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量，超出数量的线程池排队等候，等其他线程执行完毕，再从队列中取出任务来执行。

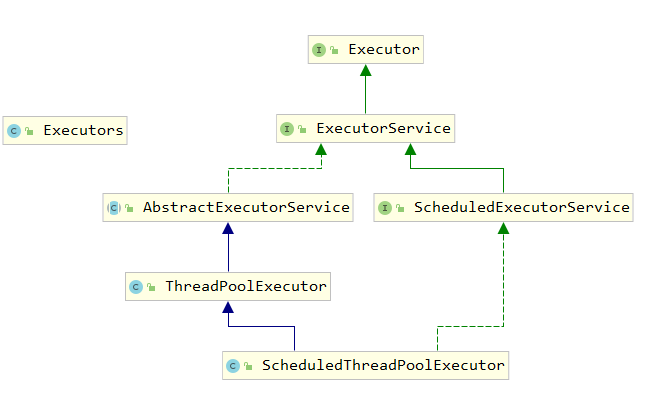
他的主要特点为：线程复用；控制最大并发数；管理线程；

1. 降低资源消耗。通过重复利用已经创建的线程降低线程创建和销毁造成的消耗；
2. 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行；
3. 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以统一分配，调优和监控。

#### 4.10.2线程池3种常用方式

**架构说明**

·Java线程池通过Executor框架实现，该框架用到了Executor，Executors（类似Array有Arrays，Collection有Collections辅助工具类），ExecutorService，ThreadPoolExecutor这几个类



**编码实现**

|  |
| --- |
| */\*\*  \* 第四种获得/使用java多线程的方式：线程池  \*/* **public class** ThreadPoolTest {   **public static void** main(String[] args) {  System.***out***.println(Runtime.*getRuntime*().availableProcessors());  ExecutorService threadPool = Executors.*newFixedThreadPool*(5);  *// 模拟10个用户来办理业务，每个用户就是一个来自外部的请求线程* **try**{  **for** (**int** i = 1; i <= 10; i++) {  threadPool.execute(()->{  System.***out***.println(Thread.*currentThread*().getName() + **"\t办理业务"**);  });  }   } **catch** (Exception e){  e.printStackTrace();  } **finally** {  threadPool.shutdown();  }  } } |
| *pool-1-thread-1 办理业务*  *pool-1-thread-2 办理业务*  *pool-1-thread-3 办理业务*  *pool-1-thread-4 办理业务*  *pool-1-thread-5 办理业务*  *pool-1-thread-4 办理业务*  *pool-1-thread-1 办理业务*  *pool-1-thread-2 办理业务*  *pool-1-thread-5 办理业务*  *pool-1-thread-3 办理业务* |

**源码解析：**

|  |
| --- |
| **public static** ExecutorService newFixedThreadPool(**int** nThreads) {  **return new** ThreadPoolExecutor(nThreads, nThreads,  0L, TimeUnit.***MILLISECONDS***,  **new** LinkedBlockingQueue<Runnable>()); } |
| **特征：**  **1 创建一个定长线程池，可控制线程最大并发数，超出的线程在队列中等待；**  **2 newFixedThreadPool创建的线程池corePoolSize和maximunPoolSize值是相等的，它使用的LinkedBlockingQueue** |

|  |
| --- |
| **public static** ExecutorService newSingleThreadExecutor() {  **return new** FinalizableDelegatedExecutorService  (**new** ThreadPoolExecutor(1, 1,  0L, TimeUnit.***MILLISECONDS***,  **new** LinkedBlockingQueue<Runnable>())); } |
| **特征：**  **1 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务，保证所有任务按照指定顺序执行；**  **2 newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，它使用LinkedBlockingQueue** |

|  |
| --- |
| **public static** ExecutorService newCachedThreadPool() {  **return new** ThreadPoolExecutor(0, Integer.***MAX\_VALUE***,  60L, TimeUnit.***SECONDS***,  **new** SynchronousQueue<Runnable>()); } |
| **特征：**  **1 创建一个可缓存线程池，如果线程池超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。**  **2 newCachedThreadPool将corePoolSize设置为0，将maximunPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue，也就是说来了任务就创建新线程，当空闲线程超过60s，就销毁线程。** |

#### 4.10.3线程池7大参数

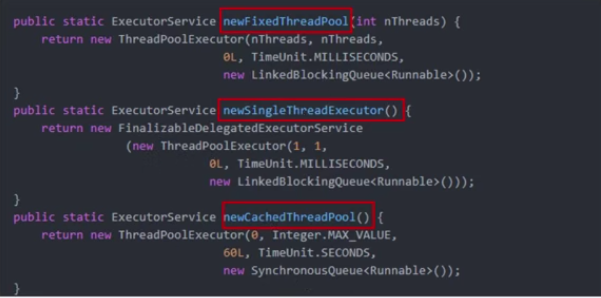
·newFixedThreadPool：执行长期的任务，性能很好；

·newSingleThreadExecutor：一个任务一个任务执行的场景；

·newCachedThreadPool：执行很多短期异步的小程序或者负载较轻的任务；

·newScheduledThreadPool：定期执行线程池

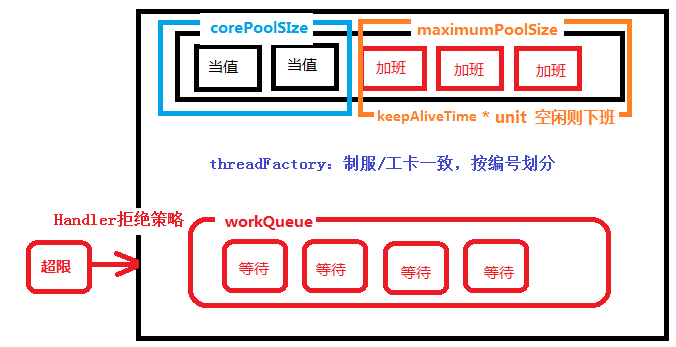
·newWorkStealingPool(int)：java8新出线程池



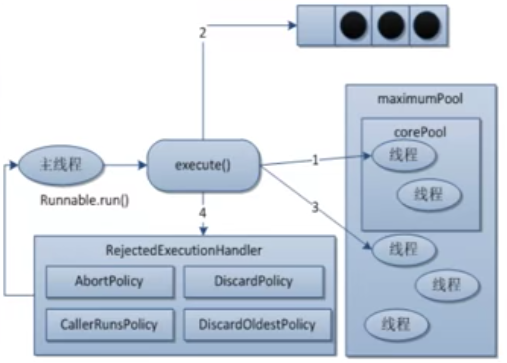
|  |
| --- |
| **public** ThreadPoolExecutor(**int** corePoolSize,  **int** maximumPoolSize,  **long** keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) |

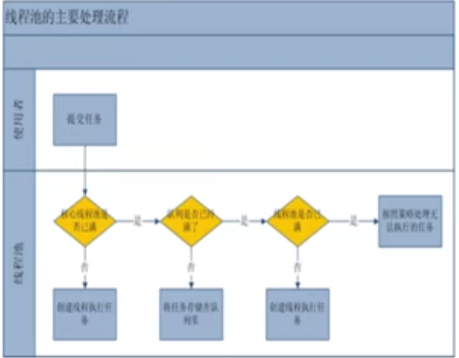
7大参数

1. corePoolSize：线程池中常驻核心线程池数；1 创建线程池后，当有请求任务到达，立刻安排当值线程执行；2 当线程池中的线程数到达corePoolSize后，就把到达的任务放到缓存队列中；
2. maximumPoolSize：线程池能够容纳同时执行的最大线程数，大于等于1，加班窗口；
3. keepAliveTime：多余的空闲线程的存活时间。当线程池数量超过corePoolSize时，当空闲时间达到keepAliveTime时，多余空闲线程会被销毁直到只剩下corePoolSize个线程为止；
4. Unit：keepAliveTime的单位；
5. workQueue：任务队列，被提交但尚未被执行的任务；
6. threadFactory：表示生成线程池中工作线程的线程工厂，用于创建线程，一般默认的即可；
7. Handler：拒绝策略，表示当队列满了并且工作线程大于等于线程池的最大线程数；



#### 4.10.4线程池底层工作原理





工作原理：

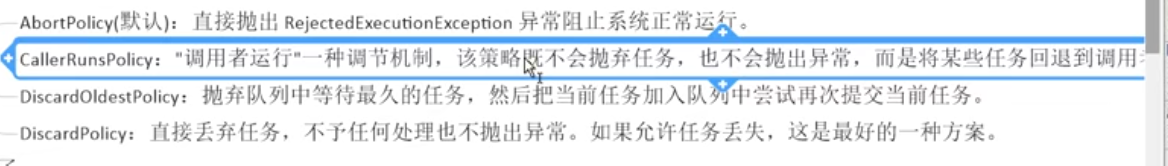
1. 在创建了线程池后，等待提交过来的任务请求；
2. 当调用execute()方法添加一个请求任务时，线程池会做如下判断：
   1. 如果正在运行的线程数量小于corePoolSize，那么马上创建线程运行这个任务；
   2. 如果正在运行的线程数量大于或等于corePoolSize，那么将这个任务**放入队列**；
   3. 如果这时候队列满了且正在运行的线程数量还小于maximunPoolSize，那么还是要创建非核心线程**立刻运行**这个任务；
   4. 如果队列满了且正在运行的线程数量大于或等于maximunPoolSize，那么线程池会**启动饱和拒绝策略来执行**；
3. 当一个线程完成任务时，它会从队列中取下一个任务来执行；
4. 当一个线程无事可做超过一定的时间(keepAliveTime\*Unit)时，线程池会判断：
   1. 如果当前运行的线程数大于corePoolSize，那么这个线程就被停掉；
   2. 所以，线程池的所有任务完成后**最终会收缩到corePoolSize的大小**；

#### 4.10.5线程池的4种拒绝策略

·是什么？

1. 等待队列也已经满了，再也塞不下新任务了；
2. 线程池中的max线程数达到最大，无法继续为新任务服务；

·JDK内置4种策略



1. AbortPolicy（默认）：直接抛出RejectedExecutionException异常阻止系统正常运行；
2. CallerRunsPolicy：“调用者运行”一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将默写任务回退到调用者，从而降低新任务的流量；
3. DiscardOldestPolicy：抛弃队列中等候最久的任务，然后把当前任务加入队列中，尝试再次提交当前任务；
4. DiscardPolicy：直接丢弃任务，不予任何处理也不抛出异常。如果允许任务丢失，这是最好的一种方案；

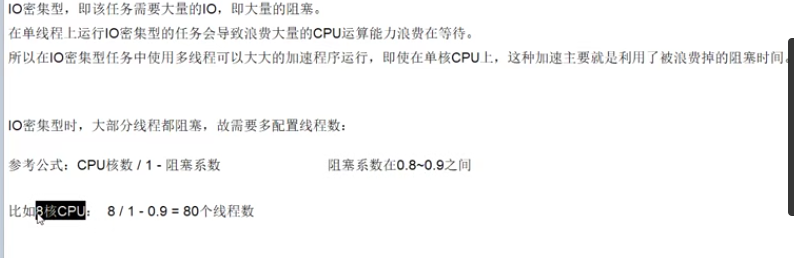
Q：你在工作中，单一/固定/可变的三种线程池创建方法，你用哪个？超级大坑！

A：一个都不用，使用ThreadPoolExecutor创建自定义线程池，防止资源耗尽；

生产实践中不允许使用Executors创建线程池：

1. FixedThreadPool和SingleThreadExecutor：允许的请求队列长度为Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致OOM；
2. CachedThreadPool和ScheduledThreadPool：允许的创建线程数量为Integer.MAX\_VALUE，可能会创建大量的线程，从而导致OOM；

#### 4.10.6如何合理配置线程池数目



·CPU密集型：需要大量运算，没有阻塞，CPU全速运行。

CPU密集型任务只有在真正的多核CPU上才能得到加速（通过多线程）；

单核不存在

CPU密集型任务配置既可能少的线程数量：CPU核数+1个线程的线程池

·IO密集型：该任务需要大量的IO，即大量阻塞。

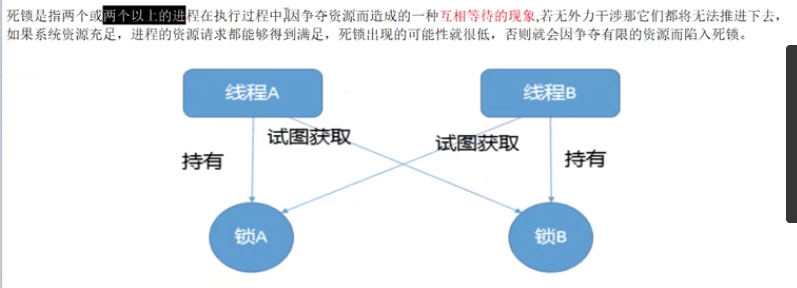
在单线程上运行IO密集型的任务会导致大量的CPU运算能力浪费在等待。

所以IO密集型任务中使用多线程可以大大的加速程序运行，即使在单核CPU上，这种加速主要是利用了被浪费掉的阻塞的试卷。

CPU核数/1-阻塞系数 阻塞系数：0.8-0.9

比如8核，8/1-0.9=80个线程数

### 4.11死锁编码及定位分析



·是什么

死锁是指两个或两个以上的进程在执行过程中，因真多资源而造成的一种互相等待的现象，若无外力干涉那它们都将无法继续推进下去，如果系统资源充足，进程的资源请求都能够得到满足，死锁出现可能性低，否则会因争夺有限的资源而进入死锁。

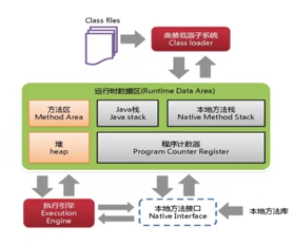
·编码实现死锁

|  |
| --- |
| **class** HoldLockThreaad **implements** Runnable{  **private** String **lockA**;  **private** String **lockB**;   **public** HoldLockThreaad(String lockA, String lockB) {  **this**.**lockA** = lockA;  **this**.**lockB** = lockB;  }   @Override  **public void** run() {  **synchronized** (**lockA**){  System.***out***.println(Thread.*currentThread*().getName() + **"\t 持有锁："** + **lockA** + **"\t等待获取锁："** +**lockB**);  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  **synchronized** (**lockB**){  System.***out***.println(Thread.*currentThread*().getName() + **"\t 持有锁："** + **lockB** + **"\t等待获取锁："** +**lockA**);  }   }  } } */\*\*  \* 死锁是指两个或两个以上的进程在执行过程中，  \* 因争夺资源而造成的一种互相等待的现象，  \* 若无外力干涉那么他们将无法继续推进下去  \*/* **public class** DeadLockTest {  **public static void** main(String[] args) {  String lockA = **"lockA"**;  String lockB = **"lockB"**;  **new** Thread(**new** HoldLockThreaad(lockA,lockB), **"threadA"**).start();  **new** Thread(**new** HoldLockThreaad(lockB,lockA), **"threadA"**).start();  } } |
|  |

·定位分析

|  |
| --- |
| G:\Effective\_Code\effective\_code>jps -l  5056  92416 sun.tools.jps.Jps  92528 org.jetbrains.jps.cmdline.Launcher  **93104 com.byf.interview.deadlock.DeadLockTest**  92100 org.jetbrains.kotlin.daemon.KotlinCompileDaemon |
| G:\Effective\_Code\effective\_code>jstack 93104  **Found one Java-level deadlock:**  =============================  "threadB":  waiting to lock monitor 0x0000000017592238 (object 0x00000000d7d8b0a8, a java.lang.String),  which is held by "threadA"  "threadA":  waiting to lock monitor 0x0000000017593578 (object 0x00000000d7d8b0e0, a java.lang.String),  which is held by "threadB"  Java stack information for the threads listed above:  ===================================================  "threadB":  at com.byf.interview.deadlock.HoldLockThreaad.run(DeadLockTest.java:20)  - waiting to lock <0x00000000d7d8b0a8> (a java.lang.String)  - locked <0x00000000d7d8b0e0> (a java.lang.String)  at java.lang.Thread.run(Thread.java:748)  "threadA":  at com.byf.interview.deadlock.HoldLockThreaad.run(DeadLockTest.java:20)  - waiting to lock <0x00000000d7d8b0e0> (a java.lang.String)  - locked <0x00000000d7d8b0a8> (a java.lang.String)  at java.lang.Thread.run(Thread.java:748)  **Found 1 deadlock.** |

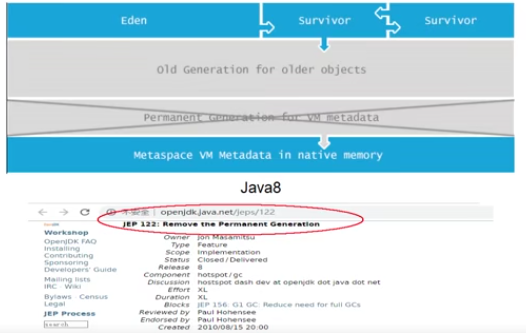
### 4.12JVM GC



根，扩展，应用，自定义

双亲委派

沙箱安排



#### 4.12.1谈谈你对GC ROOTs的理解

是否知道GC ROOTs？

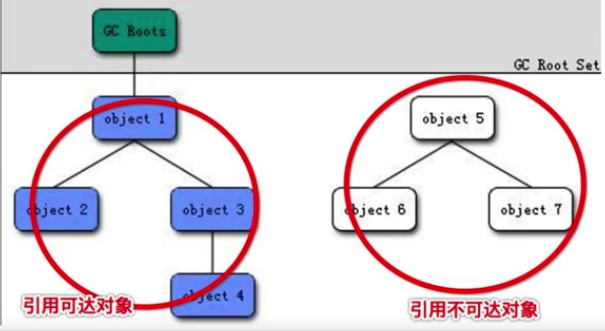
·什么是垃圾？

内存中已经不再使用到的空间就是垃圾

·如何确认一个对象是否可被回收？

1. 引用计数法：引用+1，失效-1；解决不了循环引用的问题，已经不再使用；
2. GC ROOTs可达性分析：tracing GC的“根集合”，就是一组必须活跃的引用；

基本思路就是通过一系列名为“GC ROOTs”的对象作为起始点，从这个被称为GC ROOTs的对象开始向下搜索，如果一个对象到GC ROOTs没有任何引用链相连时，则说明对象不可达。也即给定一个集合的引用作为根出发，通过引用关系遍历对象图，能被遍历到的（可到达的）对象就被判定为存活；没有被遍历到的就自然被判定死亡；



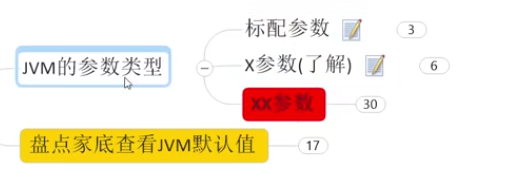
·哪些对象可以作为GC ROOTs

1. 虚拟机栈（栈帧中的局部变量区，也叫局部变量表）中引用的对象。
2. 方法区中的类静态属性引用的对象；
3. 方法区中常量引用的对象；
4. 本地方法栈JNI（Native方法）引用的对象；

|  |
| --- |
| */\*\*  \* 在java中可作为GC ROOTs的对象有：  \* （1）虚拟机栈（栈帧中的局部变量区，也叫局部变量表）中引用的对象。  \* （2）方法区中的类静态属性引用的对象；  \* （3）方法区中常量引用的对象；  \* （4）本地方法栈JNI（Native方法）引用的对象；  \*/* **public class** GCRootsTest {  **private byte**[] **byteArray** = **new byte**[100 \* 1024 \* 1024];   **private static** GCRootsTest *gcRoot1*;  **private static final** GCRootsTest ***gcRoot2*** = **new** GCRootsTest();   **public static void** m1(){  GCRootsTest gcRoot3 = **new** GCRootsTest();  **new** Thread(()->{  GCRootsTest gcRoot4 = **new** GCRootsTest();  }, **"t1"**).start();  } } |

Class 文件中存放了大量的符号引用，字节码中的方法调用指令就是以常量池中指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或第一次使用时转化为直接引用，这种转化称为静态解析。另一部分将在每一次运行期间转化为直接引用，这部分称为动态连接。

#### 4.12.2JVM标配参数和X参数



JVM参数类型：

1. 标配参数；

|  |
| --- |
| C:\Users\BYF>java -version  java version "1.8.0\_172"  Java(TM) SE Runtime Environment (build 1.8.0\_172-b11)  Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode)  C:\Users\BYF>java -help  C:\Users\BYF>java -showversion |

1. X参数（了解）；

|  |
| --- |
| -Xint：解释执行  -Xcomp：第一次使用就编译成本地代码  -Xmixed：混合模式 |

1. XX参数；

|  |
| --- |
| 题外话 |

**Boolean类型：-XX：+ 或 - 某个属性值，+表示开启，- 表示关闭**

|  |
| --- |
| **public class** HelloGC {  **public static void** main(String[] args) {  System.***out***.println(**"hello GC."**);  **try** { TimeUnit.***SECONDS***.sleep(Integer.***MAX\_VALUE***); } **catch** (InterruptedException e) {e.printStackTrace();};   } } |
| **hello GC.** |
| **G:\Effective\_Code\effective\_code>jinfo -flag PrintGCDetails 10428**  **-XX:-PrintGCDetails // 表示关闭** |
|  |
| **G:\Effective\_Code\effective\_code>jinfo -flag PrintGCDetails 7188**  **-XX:+PrintGCDetails //表示开启** |

**KV设值型**

|  |
| --- |
| **G:\Effective\_Code\effective\_code>jinfo -flag MetaspaceSize 7188**  **-XX:MetaspaceSize=21807104** |
| **G:\Effective\_Code\effective\_code>jinfo -flags 7188**  **Attaching to process ID 7188, please wait...**  **Debugger attached successfully.**  **Server compiler detected.**  **JVM version is 25.172-b11**  **Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=127926272 -XX:MaxHeapSize=2030043136 -XX:**  **MaxNewSize=676331520 -XX:MinHeapDeltaBytes=524288 -XX:NewSize=42467328 -XX:OldSize=85458944 -XX:+PrintGCD**  **etails -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-UseLar**  **gePagesIndividualAllocation -XX:+UseParallelGC**  **Command line: -XX:+PrintGCDetails -javaagent:F:\Program Files\JetBrains\IntelliJ IDEA 2019.1.1\lib\idea\_**  **rt.jar=58570:F:\Program Files\JetBrains\IntelliJ IDEA 2019.1.1\bin -Dfile.encoding=UTF-8** |
| **MaxHeapSize:操作系统默认的1/4** |

**Jinfo举例，如何查看当前运行程序的配置**

1. jinfo -flag \*\*\*\*\* pid
2. jinfo -flags pid

两个经典参数：

·-Xms2g：等价于-XX:InitailHeapSize=2g

·-Xmx2g：等价于-XX:MaxHeapSize=2g

#### 4.12.3JVM初始默认值

|  |
| --- |
| **G:\Effective\_Code\effective\_code>java -XX:+PrintFlagsInitial | findstr InitialHeapSize**  **uintx InitialHeapSize = 0 {product}**  **G:\Effective\_Code\effective\_code>java -XX:+PrintFlagsFinal | findstr InitialHeapSize**  **uintx InitialHeapSize := 127926272 {product}** |

程序运行过程中，打印JVM参数：

|  |
| --- |
| **java -XX:+PrintFlagsFinal -XX:MetasapceSize=512 MainTest**  **uintx InitialHeapSize := 127926272 {product}** |

= 表示默认值；

:= 表示修改后的值；

|  |
| --- |
| **G:\Effective\_Code\effective\_code>java -XX:+PrintCommandLineFlags -version**  **-XX:InitialHeapSize=126775168 -XX:MaxHeapSize=2028402688 -XX:+PrintCommandLineFlags -XX:+UseCompressedClassP**  **ointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC**  **java version "1.8.0\_172"**  **Java(TM) SE Runtime Environment (build 1.8.0\_172-b11)**  **Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode)** |

#### 4.12.4JVM常用基本参数

-Xms 初始大小内存，默认物理内存的1/64；等价于-XX:InitialHeapSize

-Xmx 最大分配内存，默认物理内存的1/4；等价于-XX:MaxHeapSize

-Xss 设置单个线程栈的大小，一般默认为512k~1024k；等价于-XX:ThreadStackSize

|  |
| --- |
| **G:\Effective\_Code\effective\_code>jinfo -flag ThreadStackSize 12124**  **-XX:ThreadStackSize=0 //默认使用操作系统默认** |
|  |
| **G:\Effective\_Code\effective\_code>jinfo -flag ThreadStackSize 5824**  **-XX:ThreadStackSize=128** |

-Xmn 设置新生代大小

-XX:MetaspaceSize 设置元空间大小：元空间并不在虚拟机，而是使用本地内存，默认元空间仅受本地内存限制

|  |
| --- |
|  |
| -**XX:InitialHeapSize=126775168**  **-XX:MaxHeapSize=2028402688**  **-XX:+PrintCommandLineFlags**  **-XX:+PrintGCDetails**  **-XX:ThreadStackSize=128**  **-XX:+UseCompressedClassPointers**  **-XX:+UseCompressedOops**  **-XX:-UseLargePagesIndividualAllocation**  **-XX:+UseParallelGC** |

-XX:+PrintGCDetails





|  |
| --- |
| **[GC (Allocation Failure) [PSYoungGen: 488K->472K(2560K)] 933K->917K(9728K), 0.0004613 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]**  **[Full GC (Allocation Failure) [PSYoungGen: 472K->0K(2560K)] [ParOldGen: 445K->744K(7168K)] 917K->744K(9728K), [Metaspace: 3356K->3356K(1056768K)], 0.0066895 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]** |

-XX:SurvivorRatio=x 幸存取比例，Eden: S0 : S1 = x : 1 : 1

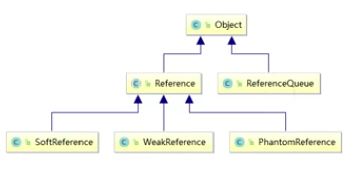
-XX:NewRatio=x Old : New = x : 1 配置新生代与老年代在堆的占比，默认2，新生代占1，老年代占2,

-XX:NewRatio=4 新生代1，老年代4，年轻代占整个堆的1/5

|  |
| --- |
| **public class** HelloGC {  **public static void** main(String[] args) {  System.***out***.println(**"hello GC."**);  *//try { TimeUnit.SECONDS.sleep(Integer.MAX\_VALUE); } catch (InterruptedException e) {e.printStackTrace();};  /\*long maxHeapSize = Runtime.getRuntime().maxMemory();  long totalMemory = Runtime.getRuntime().totalMemory();  System.out.println(maxHeapSize+"Byte\t"+maxHeapSize/(double)1024/1024+"MB");  System.out.println(totalMemory+"Byte\t"+totalMemory/(double)1024/1024+"MB"); \*/  // -Xms10m -Xmx10m -XX:+PrintGCDetails -Xss128k -XX:+PrintCommandLineFlags  // -XX:+PrintGCDetails -XX:+UseSerialGC -Xms10m -Xmx10m -XX:SurvivorRatio=4 -XX:NewRatio=1* **byte**[] bytes = **new byte**[50 \* 1024 \* 1024];   } } |
| Heap  def new generation total 4288K, used 136K [0x00000000ff600000, 0x00000000ffb00000, 0x00000000ffb00000)  eden space 3456K, 3% used [0x00000000ff600000, 0x00000000ff6221c8, 0x00000000ff960000)  from space 832K, 0% used [0x00000000ffa30000, 0x00000000ffa30000, 0x00000000ffb00000)  to space 832K, 0% used [0x00000000ff960000, 0x00000000ff960000, 0x00000000ffa30000)  tenured generation total 5120K, used 748K [0x00000000ffb00000, 0x0000000100000000, 0x0000000100000000)  the space 5120K, 14% used [0x00000000ffb00000, 0x00000000ffbbb358, 0x00000000ffbbb400, 0x0000000100000000)  Metaspace used 3446K, capacity 4496K, committed 4864K, reserved 1056768K  class space used 373K, capacity 388K, committed 512K, reserved 1048576K |

-XX:MaxTenuringThreshold 对象存活年龄，新生代S0到S1,S1到S0交换的次数，最高15，范围0-15

### 4.13强/软/弱/虚引用



**强引用：即使OOM也不回收**

|  |
| --- |
| **public class** StrongReferenceTest {  **public static void** main(String[] args) {  Object o1 = **new** Object();  Object o2 = o1;  o1 = **null**;  System.*gc*();  System.***out***.println(o2);  } } |
| **java.lang.Object@36baf30c** |

**软引用：软引用，内存足够时不收，内存不够时回收**

|  |
| --- |
| **public class** SoftReferenceTest {  **public static void** main(String[] args) {  *// memoryEnough();  notMemoryEnough*();  }   *// -Xms5m -Xmx5m -XX:+PrintGCDetails* **private static void** notMemoryEnough() {  Object o1 = **new** Object();  SoftReference<Object> softReference = **new** SoftReference<>(o1);  System.***out***.println(o1);  System.***out***.println(softReference.get());   o1 = **null**;   **try** {  **byte**[] bytes = **new byte**[30 \* 1024 \* 1024];  } **catch** (Throwable e){  e.printStackTrace();  } **finally** {  System.***out***.println();  System.***out***.println(o1);  System.***out***.println(softReference.get());  }  }   **private static void** memoryEnough() {  Object o1 = **new** Object();  SoftReference<Object> softReference = **new** SoftReference<>(o1);  System.***out***.println(o1);  System.***out***.println(softReference.get());   o1 = **null**;  System.*gc*();  System.***out***.println();  System.***out***.println(o1);  System.***out***.println(softReference.get());  } } |
| **java.lang.OutOfMemoryError: Java heap space**  **at com.byf.interview.jvm.ref.SoftReferenceTest.notMemoryEnough(SoftReferenceTest.java:21)**  **at com.byf.interview.jvm.ref.SoftReferenceTest.main(SoftReferenceTest.java:8)**  **null**  **null** |

**弱引用：内存不管够不够，只要有GC一律回收**

|  |
| --- |
| **public class** WeakReferenceTest {  **public static void** main(String[] args) {  Object o1 = **new** Object();  WeakReference<Object> weakReference = **new** WeakReference<>(o1);   System.***out***.println(o1);  System.***out***.println(weakReference.get());   o1 = **null**;  System.*gc*();  System.***out***.println(**"-------------"**);  System.***out***.println(o1);  System.***out***.println(weakReference.get());  } } |
| **java.lang.Object@36baf30c**  **java.lang.Object@36baf30c**  **-------------**  **null**  **null** |

**软引用和弱引用的适用场景：**

**·假如有一个应用需要读取大量的本地图片：**

**读取到内存影响性能；**

**一次加载又可能OOM；**

**此时适用软引用可以解决这个问题：**

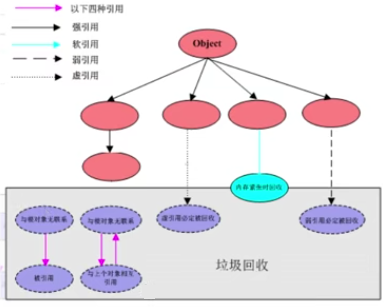
**设计思路：用一个HashMap来保存图片的路径和响应图片对象关联的软引用之间的映射关系，在内存不足时，JVM会自动回收这些缓存图片对象所占空间，从而有效避免了OOM的问题；**

**Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String, SoftReference<BitMap>>();**

**知道弱引用吗？谈谈WeakHashMap**

|  |
| --- |
| **public class** WeakHashMapTest {  **public static void** main(String[] args) {  *myHashMap*();  System.***out***.println(**"----------------"**);  *myWeakHashMap*();  }   **private static void** myHashMap(){  HashMap<Integer, String> map = **new** HashMap<>();  Integer key = **new** Integer(1);  String value = **"HashMap"**;   map.put(key,value);  System.***out***.println(map);   key = **null**;  System.***out***.println(map);   System.*gc*();  System.***out***.println(map + **"\t size:"** + map.size());   }  **private static void** myWeakHashMap(){  WeakHashMap<Integer, String> map = **new** WeakHashMap<>();  Integer key = **new** Integer(2);  String value = **"WeakHashMap"**;   map.put(key,value);  System.***out***.println(map);   key = **null**;  System.***out***.println(map);   System.*gc*();  System.***out***.println(map + **"\t size:"** + map.size());   } } |
| **{1=HashMap}**  **{1=HashMap}**  **{1=HashMap} size:1**  **----------------**  **{2=WeakHashMap}**  **{2=WeakHashMap}**  **{} size:0** |

**总结：与GCRoots结合考虑**



### 4.14OOM/SOFE内存/栈溢出

#### 4.13.1StackOverflowError

方法递归调用

|  |
| --- |
| **public class** StackOverflowErrorTest {  **public static void** main(String[] args) {  *stackOverflowError*();  }   **private static void** stackOverflowError() {  *stackOverflowError*();  } } |
| **Exception in thread "main" java.lang.StackOverflowError**  **at com.byf.interview.jvm.oom.StackOverflowErrorTest.stackOverflowError(StackOverflowErrorTest.java:9)** |

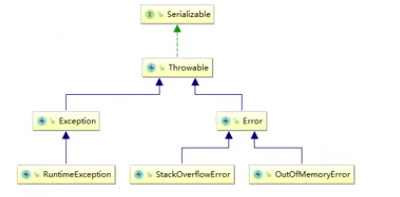
Throwable

Error, Exception

Error

VirtualMachineError

StackOverflowException



#### 4.13.2OOM: java heap space

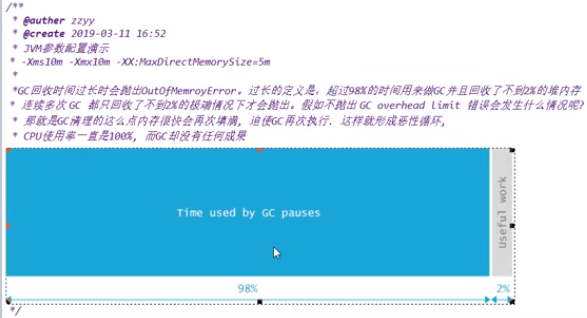
不停的new对象，在常量池中新建；

|  |
| --- |
| **-Xms10m -Xmx10m** |
| **public class** JavaHeapSpaceTest {  **public static void** main(String[] args) {  String str = **"abcdefgh"**;  **while** (**true**){  str += str + **new** Random().nextInt(1111111111) + **new** Random().nextInt(2222222);  str.intern();  }  } } |
| **Exception in thread "main" java.lang.OutOfMemoryError: Java heap space**  **at java.util.Arrays.copyOf(Arrays.java:3332)**  **at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:124)**  **at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:674)**  **at java.lang.StringBuilder.append(StringBuilder.java:208)**  **at com.byf.interview.jvm.oom.JavaHeapSpaceTest.main(JavaHeapSpaceTest.java:9)** |

|  |
| --- |
| **public class** JavaHeapSpaceTest {  **public static void** main(String[] args) {  String str = **"abcdefgh"**;  */\*while (true){  str += str + new Random().nextInt(1111111111) + new Random().nextInt(2222222);  str.intern();  }\*/* **byte**[] bytes = **new byte**[50 \* 1024 \* 1024];  } } |
| **Exception in thread "main" java.lang.OutOfMemoryError: Java heap space**  **at com.byf.interview.jvm.oom.JavaHeapSpaceTest.main(JavaHeapSpaceTest.java:12)** |

在 Java8 中，永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间的本质和永久代类似，元空间与永久代之间最大的区别在于：**元空间并不在虚拟机中，而是使用本地内存。**因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 native memory, 字符串池和类的静态变量放入 java 堆中，这样可以加载多少类的元数据就不再MaxPermSize 控制, 而由系统的实际可用空间来控制。

#### 4.13.3OOM: GC overhead limit exceeded



·GC回收时间过长会抛出OutOfMemoryError，过长的定义是，超过98%的时间用来做GC并且回收了不到2%的堆内存，连续多次GC都 只回收了不到2%的极端情况才会抛出这种异常。

·如果不抛出该异常会有什么问题？

内存不足很快再次耗光，迫使再次GC，恶性循环，CPU使用率一直100%，GC没有任何成功。

|  |
| --- |
| **-Xms20m -Xmx20m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=10m**  **// 如果内存配置的过小，则抛出OOM：Java Heap Size** |
| **public class** GCOverheadLimitExceedTest {  **public static void** main(String[] args) {  **int** i = 0;  List<String> list = **new** ArrayList<>();   **try** {  **while** (**true**){  list.add(String.*valueOf*(++i).intern());  }  } **catch** (Throwable e){  System.***out***.println(**"\*\*\*\*\*\*\*\*\*i:"** +i);  *// 不打印  //e.printStackTrace();* **throw** e;  }  } } |
| **\*\*\*\*\*\*\*\*\*i:145489**  **Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded**  **at java.lang.Integer.toString(Integer.java:401)**  **at java.lang.String.valueOf(String.java:3099)**  **at com.byf.interview.jvm.oom.GCOverheadLimitExceedTest.main(GCOverheadLimitExceedTest.java:21)** |

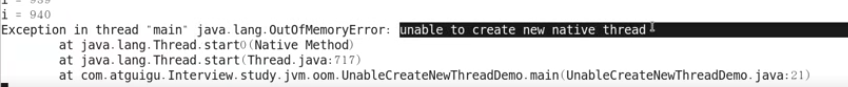
#### 4.13.4OOM:Direct buffer memory



|  |
| --- |
| */\*\*  \* JVM参数：  \* -Xms10m -Xmx10m -XX:+PrintGCDetails -XX:MaxDirectMemorySize=5m  \* 故障现象：  \* java.lang.OutOfMemoryError: Direct buffer memory  \* 导致原因：  \* 写NIO程序经常使用ByteBuffer来取或者写入数据，这是一种基于通道（Channel）与缓冲区（Buffer）的I/O方式  \* 它可以使用Native函数直接分配堆外内存，然后通过Java堆DirectByteBuffer对象为引用操作。  \* 这样能在一些场景中显著提高性能，因为避免了在Java堆和Native堆中来回拷贝  \*  \* ByteBuffer.allocate(capability) 分配JVM堆内存，属于GC管辖范围，需要用户(Java堆)/内核(Native堆)之间拷贝，速度相对较慢  \*  \* ByteBufffer.allocateDirect(capability) 分配OS本地内存，不属于GC管辖，由于不需要内存拷贝，速度相对较快  \*  \* 但如果不断分配本地内存，堆内存很少使用，那么JVM就不需要执行GC，  \*/* **public class** DirectBufferMemoryTest {  **public static void** main(String[] args) {  System.***out***.println(**"配置的maxDirectMemory："**+(sun.misc.VM.*maxDirectMemory*())/(**double**)1024/1024 + **"MB."**);  **try** { TimeUnit.***SECONDS***.sleep(1); } **catch** (InterruptedException e) {e.printStackTrace();}  ByteBuffer byteBuffer = ByteBuffer.*allocateDirect*(6 \* 1024 \* 1024);  } } |
| *[GC (Allocation Failure)*  *配置的maxDirectMemory：5.0MB.*  *[Full GC (System.gc())*  *Exception in thread "main" java.lang.OutOfMemoryError:* ***Direct buffer memory*** |

#### 4.13.5OOM: unable to create new native thread

高并发场景



|  |
| --- |
| **public synchronized void** start() {**if** (**threadStatus** != 0)  **throw new** IllegalThreadStateException(); **group**.add(**this**);   **boolean** started = **false**;  **try** {  start0();  started = **true**;  } **finally** {  **try** {  **if** (!started) {  **group**.threadStartFailed(**this**);  }  } **catch** (Throwable ignore) {}  } } |
| **// 报native thread thread无法创建的原因，start0是native方法**  **private native void** start0(); |
| */\*\*  \* 故障现象：  \* 高并发请求服务器时，经常出现：  \* java.lang.OutOfMemoryError: unable to create new native thread  \* native thread 与 OS平台有关  \* 导致原因：  \* 1.创建线程太多，超过系统承载限制；  \* 2.Linux默认1024，超过则报：  \* java.lang.OutOfMemoryError: unable to create new native thread  \*  \* 解决办法：  \* 1.降低可创建线程数，分析应用是否需要这么多线程；  \* 2.确实需要，配置Linux服务器，扩大默认限制；  \*/* **public class** UnableCreateNewThreadTest {  **public static void** main(String[] args) {  **for** (**int** i = 0; ; i++) {  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\* i = "** + i);  **new** Thread(()->{  **try** { TimeUnit.***SECONDS***.sleep(Integer.***MAX\_VALUE***); } **catch** (InterruptedException e) {e.printStackTrace();}  }, **"t1"**).start();   }  } } |

句柄数上限调整：

|  |
| --- |
| [root@localhost /root]#cat /etc/security/limits.d/20-nproc.conf  # Default limit for number of user's processes to prevent  # accidental fork bombs.  # See rhbz #432903 for reasoning.  \* soft nproc 4096  root soft nproc unlimited  [root@localhost /root]#ulimit -u  3802  [root@localhost /root]#ulimit -a  core file size (blocks, -c) 0  data seg size (kbytes, -d) unlimited  scheduling priority (-e) 0  file size (blocks, -f) unlimited  pending signals (-i) 3802  max locked memory (kbytes, -l) 64  max memory size (kbytes, -m) unlimited  open files (-n) 1024  pipe size (512 bytes, -p) 8  POSIX message queues (bytes, -q) 819200  real-time priority (-r) 0  stack size (kbytes, -s) 8192  cpu time (seconds, -t) unlimited  **max user processes (-u) 3802**  virtual memory (kbytes, -v) unlimited  file locks (-x) unlimited |

#### 4.13.6OOM: Metaspace

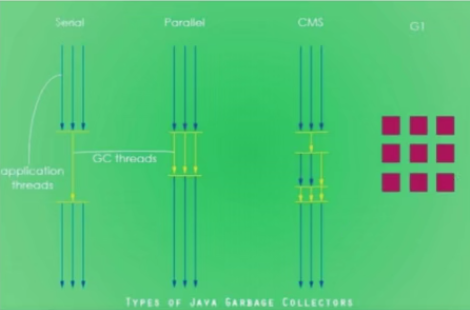
|  |
| --- |
| >java -XX:+PrintFlagsFinal | findstr MetaspaceSize  uintx MetaspaceSize = 21807104 {pd product} |

|  |
| --- |
| */\*\*  \* JVM参数：  \* -XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m  \* java8 Metaspace：并不在虚拟机，而是在本地内存，class metadata叫做Metaspace的native memory  \*  \* 元空间存放以下信息：  \* 1.虚拟机加载的类信息  \* 2.常量池  \* 3.静态变量  \* 4.即时编译后的代码  \*  \* 模拟Metaspace空间溢出，不断生成类，类占据的空间会超过Metaspace指定的大小  \*/* **public class** MetaspaceSizeTest {  **static class** OOMTest{}  **public static void** main(String[] args) {  **int** i =0 ;  **try** {  **while** (**true**){  i++;  Enhancer enhancer = **new** Enhancer();  enhancer.setSuperclass(OOMTest.**class**);  enhancer.setUseCache(**false**);  enhancer.setCallback(**new** MethodInterceptor() {  @Override  **public** Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) **throws** Throwable {  **return** methodProxy.invokeSuper(o, args);  }  });  enhancer.create();  }  } **catch** (Throwable e){  System.***out***.println(**"\*\*\*\*\*\*\* 创建了i : "** + i + **"个加强类"**);  e.printStackTrace();  }  } } |
| *\*\*\*\*\*\*\* 创建了i : 567个加强类*  ***java.lang.OutOfMemoryError: Metaspace***  *at java.lang.Class.forName0(Native Method)* |

### 4.15垃圾收集器/收集算法

·GC算法（复制/标记清理/标记整理）接口，垃圾收集器是实现落地

#### ·垃圾收集方式：Serial/Parallel/CMS/G1（串行/并行/并发/G1）



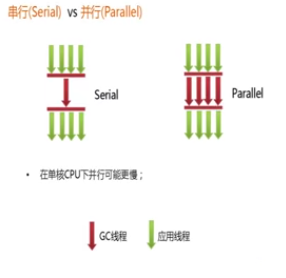
1 串行：单线程环境，且只使用一个线程进行垃圾回收，会暂停所有用户线程。所以不适合服务器环境

2 并行：多个垃圾收集线程并行工作，会暂停用户线程，适用于科学计算/大数据处理首台处理弱交互场景

3 并发：用户线程和垃圾收集线程同时执行（不一定并行，可能交替执行），不需要暂停用户线程，适用对响应时间有要求的场景

4 G1：将堆内存分割成不同区域，并发回收

总结：





#### 如何查看默认垃圾收集器？

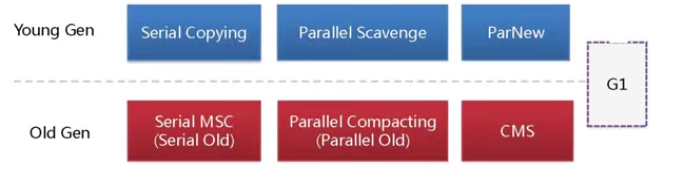
|  |
| --- |
| G:\Effective\_Code\effective\_code>**java -XX:+PrintCommandLineFlags -version**  -XX:InitialHeapSize=126775168 -XX:MaxHeapSize=2028402688 -XX:+PrintCommandLineFlags -XX:+UseCompre  ssedClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC  java version "1.8.0\_172"  Java(TM) SE Runtime Environment (build 1.8.0\_172-b11)  Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode) |
| G:\Effective\_Code\effective\_code>jinfo -flag UseSerialGC 16468  -XX:-UseSerialGC  G:\Effective\_Code\effective\_code>jinfo -flag UseParallelGC 16468  -XX:+UseParallelGC |

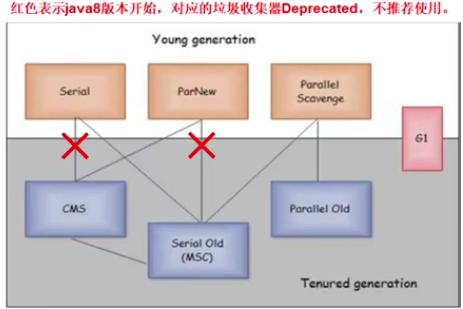
#### 生成上如何设置垃圾收集器？

-XX:+UseSerialGC、-XX:+UseParallelGC、-XX:+UseConcMarkSweepGC、-XX:+UseParNewGC、

-XX:+UseParallelOldGC、-XX:+UseG1GC、（-XX:+UseSerialOldGC）

谈谈对垃圾收集器的理解？





深透明细

思想：4大垃圾回收算法

(引用计数)

复制

标记清除

标记整理

落地实现：

串行回收 -XX:+UseSerialGC

并行回收 -XX:+UseParallelGC

并发回收

G1

#### GC约定参数



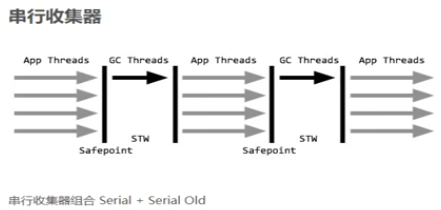
server/client

32位Windows，默认Client的JVM模式

32位其他操作系统，2G内存2个CPU以上用Server模式，低于该配置还是Client

64位，server模式

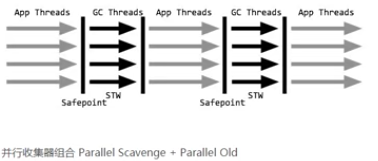
#### 4.15.1Serial收集器



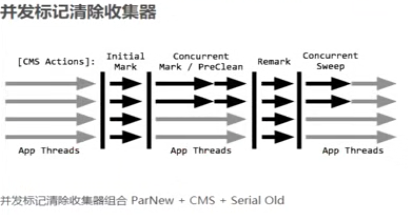
·稳定，效率高，停顿长

#### 4.15.2Paralle收集器





#### 4.15.3CMS收集器



CMS：是一种以获得最短停顿时间为目标的收集器



优点：并发收集低停顿

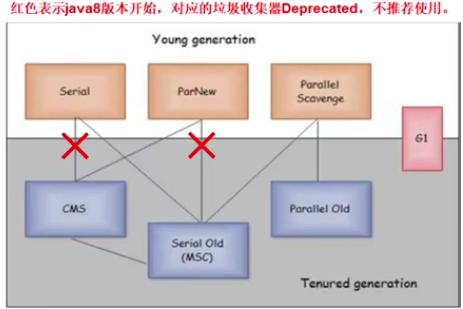
缺点：CPU压力大/内存碎片，CMS失败（堆内存耗尽之前没有完成），触发担保串行回收SerialOld；

#### 4.15.4G1收集器

·分块大小1-32MB不等，最多2048个区，支持最大内存32MB \* 2048 = 64GB

-XX:G1HeapRegionSize=32MB

·最大的好处是化整为零，避免全内存扫描，只需要按区域扫描；

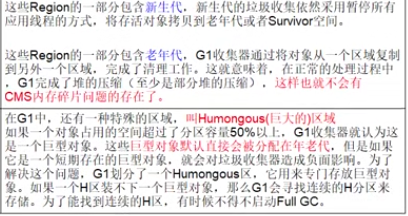


|  |
| --- |
| [GC concurrent-mark-abort]  java.lang.OutOfMemoryError: Java heap space  at java.util.Arrays.copyOf(Arrays.java:3332)  at java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:124)  at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:448)  at java.lang.StringBuilder.append(StringBuilder.java:136)  at com.byf.interview.jvm.gc.GCTest.main(GCTest.java:37)  Heap  garbage-first heap total 10240K, used 3783K [0x00000000ff600000, 0x00000000ff700050, 0x0000000100000000)  region size 1024K, 1 young (1024K), 0 survivors (0K)  Metaspace used 3493K, capacity 4496K, committed 4864K, reserved 1056768K  class space used 379K, capacity 388K, committed 512K, reserved 1048576K |





红色：大对象



G1算法将堆划分为若干区域（Region），它仍属于分代收集

·一部分新生代，收集方式仍然采用暂停所有应用线程，将存活对象拷贝到老年代或Survivor空间；

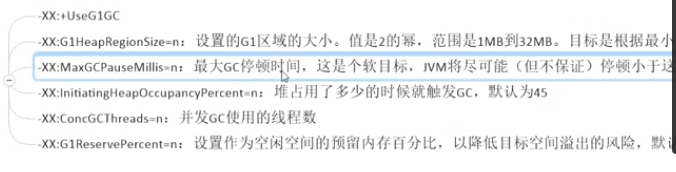
·一部分老年代，G1将对象从一个区域复制到另外一个区域，完成清理。不再像CMS内存产生碎片；

四个过程：



|  |
| --- |
| [GC pause (G1 Humongous Allocation) (young) (initial-mark), 0.0005312 secs] |
| [GC concurrent-mark-start] |
| [GC remark [Finalize Marking, 0.0001598 secs] |
| [GC cleanup 5877K->5877K(10M), 0.0162126 secs] |

G1参数配置与CMS比较



-XX:+UseG1GC

-XX:G1HeapRegionSize=32MB Region大小，值是2的幂，范围：1MB到32MB

-XX:MaxGCPauseMillis=100 最大GC挺对事件，JVM将尽可能保证小于100ms

-XX:ConcGCThreads=n 并发GC使用线程数

-XX:G1ReservePercent=n 设置最为空闲空间的预留内存百分比，以降低目标空间溢出风险

1. G1没有碎片；
2. 精确控制停顿；

#### 4.15.5如何选择合适的垃圾收集器？



组合选择：

·单CPU或小内存，单机程序

-XX:+UseSerialGC

·多CPU，需要最大吞吐量，如后台计算型应用

-XX:+UseParallelGC或者-XX:+UseParallelOldGC

·多CPU，追求低停顿，需要快速响应如互联网应用

-XX:+UseConcMarkSweepGC

-XX:+UseParNew



### 4.16JVM GC结合SpringBoot微服务优化

Tomcat / Undertow

1 IDEA开发完微服务工厂；

2 Maven进行clean package；

3 要求微服务启动的时候，同时配置我们的JVM/GC的调优参数；

3.1 内--IDEA

3.2 外--java -jar

4 公式：

java -server jvm各种参数 -jar jar包/war包

jps -l

jinfo -flags jpid

### 4.17Linux常用命令

生产环境服务变慢，谈谈你的评估？



#### ·（总）整机：top：

load average：1分钟，5分钟，15分钟系统的平均负载值，三个相加/3\*100高于60%系统负担压力重；

1查看核数，具体哪个核压力大；

Q退出

uptime：精简版系统负荷执行，运行时间

|  |
| --- |
| [root@localhost /root]#uptime  20:51:53 up 1 day, 23:08, 2 users, load average: 0.14, 0.05, 0.06 |

#### ·（分）CPU：vmstat（pmstat / pidstat）：

|  |
| --- |
| [root@localhost /root]#vmstat -n 2 3  procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----  r b swpd free buff cache si so bi bo in cs us sy id wa st  1 0 0 342320 156 544988 0 0 0 1 8 6 0 0 100 0 0  0 0 0 342328 156 544988 0 0 0 2 40 31 0 0 100 0 0  0 0 0 342328 156 544988 0 0 0 0 23 15 0 0 100 0 0 |

-- procs：

r运行，运行和等待CPU片的进程数，原则上1核的CPU的运行队列不要超过2，整个系统的运行队列不能超过总核数的2倍，否则代表系统压力过大；

b阻塞，等待资源的进程数，比如正在等待磁盘I/O、网络I/O等。

-- CPU

us用户，用户进程消耗CPU时间百分比，us值高，用户进程消耗CPU时间多，如果长期大于50%，优化程序；

sy系统，内核进程消耗的CPU时间百分比；

us+sy参考值为80%，如果us+sy大于80%，说明可能存在CPU不足；

id ：处于空闲的CPU百分比；

wa：系统等待IO的CPU时间百分比；

st：来自于一个虚拟机偷取的CPU时间的百分比；



查看额外：

|  |
| --- |
| [root@localhost /root]#yum install sysstat  [root@localhost /root]#mpstat -P ALL 2  Linux 3.10.0-862.el7.x86\_64 (localhost.localdomain) 11/15/2019 \_x86\_64\_ (4 CPU)  09:07:00 PM CPU %usr %nice %sys %iowait %irq %soft %steal %guest %gnice %idle  09:07:02 PM all 0.13 0.00 2.55 0.00 0.00 0.00 0.00 0.00 0.00 97.32 |

查看每个进程使用CPU的用量分解信息

|  |
| --- |
| [root@localhost /root]#pidstat -u 1 -p 1  Linux 3.10.0-862.el7.x86\_64 (localhost.localdomain) 11/15/2019 \_x86\_64\_ (4 CPU)  09:09:21 PM UID PID %usr %system %guest %CPU CPU Command  09:09:22 PM 0 1 0.00 0.00 0.00 0.00 0 systemd |

#### ·（分）内存：free

|  |
| --- |
| [root@localhost /root]#free -m  total used free shared buff/cache available  Mem: 974 107 265 7 601 645  Swap: 2047 0 2047 |

经验值：

1 应用程序可用内存/系统物理内存>70%内存充足

2 应用程序可用内存/系统物理内存<20%，内存不足，需要增加内存

3 20%<应用程序可用内存/系统物理内存<70%基本够用

#### ·（分）磁盘：df

|  |
| --- |
| [root@localhost /root]#df -h  Filesystem Size Used Avail Use% Mounted on  /dev/mapper/centos-root 17G 2.5G 15G 15% /  devtmpfs 476M 0 476M 0% /dev  tmpfs 488M 0 488M 0% /dev/shm  tmpfs 488M 7.7M 480M 2% /run  tmpfs 488M 0 488M 0% /sys/fs/cgroup  /dev/sda1 1014M 130M 885M 13% /boot  tmpfs 98M 0 98M 0% /run/user/0 |

#### ·（分）磁盘IO：iostat

|  |
| --- |
| [root@localhost /root]#iostat -xdk 2 3  Linux 3.10.0-862.el7.x86\_64 (localhost.localdomain) 11/15/2019 \_x86\_64\_ (4 CPU)  Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrq-sz avgqu-sz await r\_await w\_await svctm %util  sda 0.00 0.02 0.05 0.15 1.45 5.20 66.81 0.00 4.77 8.53 3.44 1.39 0.03  dm-0 0.00 0.00 0.04 0.16 1.39 5.19 64.95 0.00 6.12 10.98 4.91 1.35 0.03  dm-1 0.00 0.00 0.00 0.00 0.01 0.00 47.40 0.00 2.80 2.80 0.00 2.13 0.00 |
| [root@localhost /root]#pidstat -d 2 -p 1  Linux 3.10.0-862.el7.x86\_64 (localhost.localdomain) 11/15/2019 \_x86\_64\_ (4 CPU)  09:18:48 PM UID PID kB\_rd/s kB\_wr/s kB\_ccwr/s Command  09:18:50 PM 0 1 0.00 0.00 0.00 systemd |

MySQL大表存储，消耗磁盘I/O

磁盘块设备分布；

rkB/s每秒读取数据量KB；

wkB/s每秒写入数据kB；

Svctm：I/O请求的平均服务时间，单位毫秒；

Await：I/O请求的平均等待时间，单位毫秒，值越小，性能越好；

Util：一秒钟有百分几的时间用于I/O操作。接近100%时，表示磁盘带宽跑满，需要优化程序或者增加磁盘；

rkB/s、wkB/s根据系统应用不同会有不同的值，但有规律遵循；长期、超大数据读写，肯性不正常，需要优化程序读取；

Svctm的值与await的值很接近，表示几乎没有I/O等待，磁盘性能好；

如果await的值远高于svctm的值，则表示I/O队列等待太长，需要优化程序更换磁盘；

#### ·（分）网络IO：ifstat

|  |
| --- |
| [root@localhost /root]#ifstat -a ens33  #kernel  Interface RX Pkts/Rate TX Pkts/Rate RX Data/Rate TX Data/Rate  RX Errs/Drop TX Errs/Drop RX Over/Rate TX Coll/Rate  ens33 108678 0 60513 0 111020K 0 5198K 0  0 0 0 0 0 0 0 0 |

#### ·CPU占用过高定位思路

1 top -c | top -Hp pid

2 ps -ef|grep pid

3 ps -mp 1 -o THREAD,tid,time

|  |
| --- |
| [root@localhost /root]#ps -mp 1 -o THREAD,tid,time  USER %CPU PRI SCNT WCHAN USER SYSTEM TID TIME  root 0.0 - - - - - - 00:00:11  root 0.0 19 - ep\_pol - - 1 00:00:11 |
| -m：显示所有的线程  -p：pid进程使用CPU时间  -o：该参数后是用户自定义格式 |

4 线程ID==>16进制转换：printf “%x\n” 线程ID

5 jstack 进程ID|grep tid（第4步打印的4位16进制） -A60

### 4.18GitHub操作

#### 常用词含义：

1 watch：会持续受到该项目的动态；

2 fork：复制某个项目到自己的GitHub仓库；

3 star：点赞；

4 clone：下载项目；

5 follow：关注感兴趣的作者，受到动态；

·做一个秒杀系统，简单版？

#### 1 in

xxx关键词 in:name 项目名包含

xxx关键词 in:description项目描述包含

xxx关键词 in:readme 项目ReadMe包含

xxx关键词 int:name,readme 项目名/ReadMe包含

seckill in:name,readme

#### 2 stars/fork

springboot stars:>=50

Springcloud forks:>500

springboot forks:100..300 stars:80..100

Netty forks:100..300 stars:80..100

Netty in:name,readme forks:>100 stars:>100

#### 3 awesome增强搜索

awesome redis

框架、教程、资源

#### 4 地址#L13-#23

查看java代码第13行到23行

#### 5 T搜索

<https://github.com/codingXiaxw/seckill>

输t

<https://github.com/codingXiaxw/seckill/find/master>

#### 6区活跃用户

location:beijing language:java

## Netty

<https://blog.csdn.net/baiye_xing/article/details/76735113>

### 为什么使用Netty？

1. JavaNio框架虽然提供了多路复用的IO支持，但并没有很好的支持上层消息格式的封装，如ProtocolBuffer、JSON，而Netty具有责任链模式的编解码和解码功能；
2. 直接使用NIO需要Java多线程和编解码；
3. 要编写一个可靠、易维护、高性能的NIO应用，需要具备单点续传，半包读写，心跳，安全认证等，这些Netty提供了很好的封装；
4. JavaNIO存在一个poll/epoll的bug，空轮训即selector不能block导致CPU利用率100%；

### 为什么不用Netty5？

1. Netty5使用里ForkJoinPool，增加代码复杂度，但对性能改善不明显；
2. 多个分支的代码同步；

### 为什么Netty使用NIO而不是AIO？

1. Netty不看重Windows上使用，在Linux上，AIO的底层实现仍使用Epoll，没有很好的实现AIO，因此在性能上没有明显的优势，而且被JDK封装了一层不容易深度优化；
2. AIO接收数据需要预先分配缓存，而不是像NIO需要接收时才分配，所以对连接数量非常大，但流量小的情况浪费内存；
3. 据说Linux上AIO不够成熟，处理回调速度跟不上需求，就像快递员太少，客户太多，供不应求，造成处理速度有瓶颈；

### 5.1BIO、NIO、AIO区别

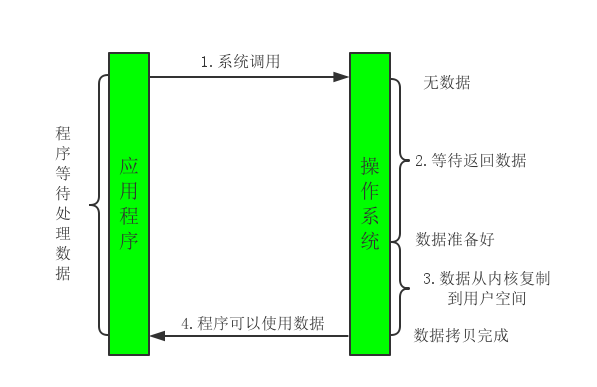
#### 阻塞/非阻塞、同步/异步

阻塞VS非阻塞：人是否坐在水壶前面一直等。

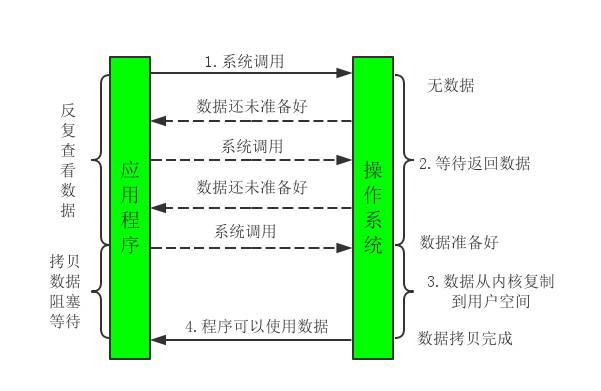
同步VS异步：水壶是不是在水烧开之后主动通知人。

#### 五种IO模型

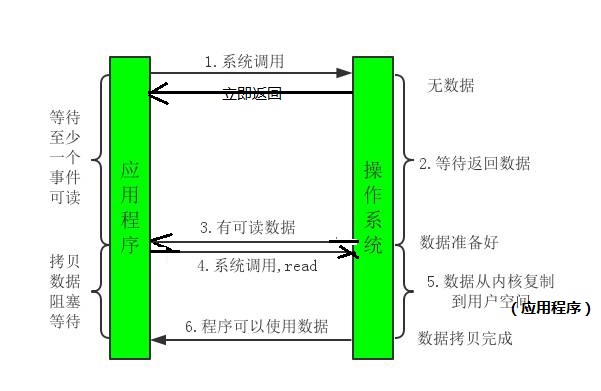
BIO （Blocking I/O）：同步阻塞I/O模式。



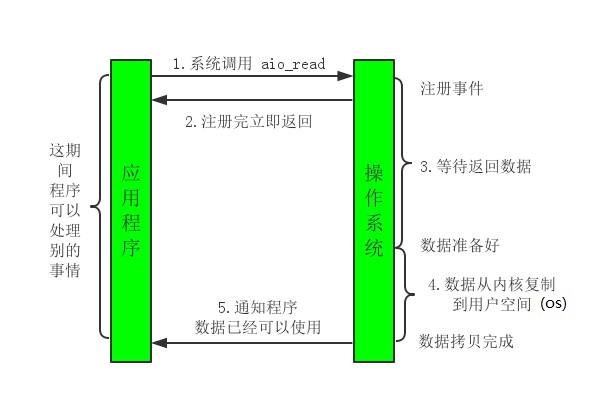
NIO （New I/O）：同步非阻塞模式。



IO多路复用（NIO）：同步非阻塞模式



AIO （Asynchronous I/O）：异步非阻塞I/O模型。



#### 经典示例

同步阻塞模式：这种模式下，我们的工作模式是先来到厨房，开始烧水，并坐在水壶面前一直等着水烧开。

同步非阻塞模式：这种模式下，我们的工作模式是先来到厨房，开始烧水，但是我们不一直坐在水壶前面等，而是回到客厅看电视，然后每隔几分钟到厨房看一下水有没有烧开。Netty的sellector同样是同步非阻塞，只不过取看水是否烧开的是由系统检查水是否烧开。

异步非阻塞I/O模型：这种模式下，我们的工作模式是先来到厨房，开始烧水，我们不一一直坐在水壶前面等，也不隔一段时间去看一下，而是在客厅看电视，水壶上面有个开关，水烧开之后他会通知我。

阻塞VS非阻塞：人是否坐在水壶前面一直等。

同步VS异步：水壶是不是在水烧开之后主动通知人。

#### NIO实现：select、pool和epoll区别

1)select:注册的socket事件由数组管理，长度有限制，轮询查找时需要遍历数组。

2)poll:注册的socket事件由链表实现，数量没有限制，遍历链表轮询查找。

3)epoll:基于事件驱动思想，采用reactor模式，通过事件回调，无需使用某种方式主动检查socket状态，被动接收就绪事件即可。

<https://vps.zzidc.com/java/2367.html>



IO：即输入/输出。通常指数据在内部存储器（内存）和外部存储器（硬盘、优盘等）或其他周边设备之间的输入和输出。

在Java中，提供了一些列API，可以供开发者来读写外部数据或文件。我们称这些API为Java IO。

IO是Java中比较重要，且比较难的知识点，主要是因为随着Java的发展，目前有三种IO共存。分别是BIO、NIO和AIO。

**Java BIO**

BIO 全称Block-IO 是一种同步且阻塞的通信模式。是一个比较传统的通信方式，模式简单，使用方便。但并发处理能力低，通信耗时，依赖网速。

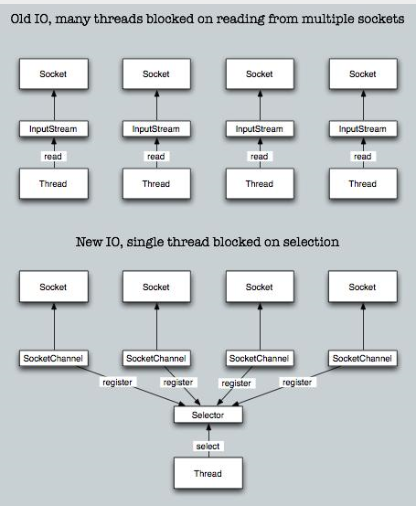
**Java NIO**

Java NIO，全程 Non-Block IO ，是Java SE 1.4版以后，针对网络传输效能优化的新功能。是一种非阻塞同步的通信模式。

NIO 与原来的 I/O 有同样的作用和目的, 他们之间最重要的区别是数据打包和传输的方式。原来的 I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

面向流的 I/O 系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。

面向块的 I/O 系统以块的形式处理数据。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按(流式的)字节处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。



#### 适用场景

BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

AIO方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

### 5.2Netty NIO的组成

#### Buffer

与Channel交互，数据从Channel读入缓冲区，从缓冲区写入Channel；

#### flip方法

反转缓冲区，将position给limit，然后将position置为0，其实就是切换读写模式；

#### clear方法

清除缓冲区，将position置为0，把capacity的值给limit

#### rewind方法

重绕此缓冲区，将positioon置为0

#### DirectByteBuffer

优点：减少一次系统空间到用户空间的拷贝。

缺点：Buffer的创建和销毁成本更高、不可控，通常会用内存池来提高性能。

直接缓冲区主要分配给容易受基础系统的本机I/O操作影响的大型、持久的缓冲区。如果数据量比较小的情况，建议使用heapBuffer，由JVM管理。

#### Channel

表示I/O源与目标打开的链接，是双向的，但不能直接访问数据，只能与Buffer进行交互，通过源码可知，FileChannel的read方法和write方法都导致数据复制了两次。

#### Selector

可使一个单独的线程管理多个Channel，open方法可创建Selector，register方法向多路复用器注册通道，可以监听的时间类型：读、写、连接、accept。注册事件后会产生一个SelectionKey：它表示SelectableChannel和Selector之间的注册关系，wakeup方法：使尚未返回的第一个选择操作立即返回，唤醒的原因是：注册了新的channel或者事件；channel关闭，取消注册；优先级粳稻的时间出发（如定时器事件），希望及时处理。

Selector在Linux的实现类是EpollSelectorImpl，委托给EPollArrayWrapper实现，其中三个native方法是对epoll的封装，而EPollSelectorImpl，implRegister方法，通过调用epoll\_ctl向epoll实例中注册时间，还将注册的文件描述符（fd）与SelectionKey的对应关系添加到fdToKey中，这个map维护了文件描述符与SelectionKey的映射。

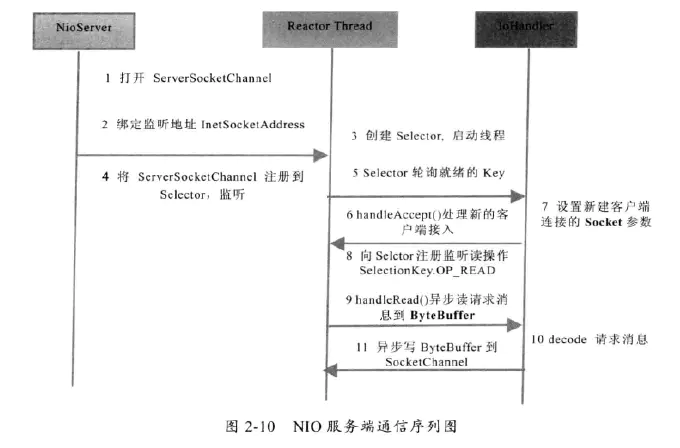
fdToKey通常很大，因为注册到Selector上的Channel非常多（百万连接）；过期或失效的Channel没有及时关闭，fdToKey总是串行读取的，而读取实在select方法中进行，该方法是非线程安全的。

#### Pipe

两个线程之间的单向数据连接，数据会被写到sink管道，从source用到读取

### 5.3NIO服务端建立过程

1. Selector.open()：打开一个Selector；
2. ServerSocketChannel.open()：创建服务端Channel；
3. Bind()：绑定到某个端口上。并配置非阻塞模式；
4. Register()：注册Channel和关注的时间到Selector上；
5. Select():轮训拿到已经就绪的事件；



<https://www.cnblogs.com/wmcoder/p/7169004.html>

|  |
| --- |
| 步骤一：打开ServerSocketChannel，用于监听客户端的连接，它是所有客户端连接的副管道：  ServerSocketChannel acceptorSvr = ServerSocketChannel.open(); |
| 步骤二：绑定监听端口，设置连接为非阻塞模式  acceptorSvr.socket().bind(new InetScoketAddress(InetAddress.getByName(IP),port));  acceptorSvr.configureBlocking(false); |
| 步骤三：创建Reactor线程，创建多路复用器并启动线程  Selector selector = Selector.open();  New Thread(new ReactorTask()).start(); |
| 步骤四：将ServerSocketChannel注册到Reactor线程的多路复用器Selector上，监听ACCEPT事件  SelectionKey key = acceptorSvr.register(selector,SelectionKey.OP\_ACCEPT,ioHandler); |
| 步骤五：多路复用器在线程run方法的无线循环体内轮询准备就绪的key  int num = selector.selector();  Set selectedKeys = selector.selectedKeys();  Iterator it = selectedKeys.iterator();  while(it.hasNext()){       SelectionKey key = (SelectionKey)it.next();       //....deal with i/o event ...  } |
| 步骤六：多路复用器监听到有新的客户端接入，处理新的接入请求，完成TCP三次握手，建立物理连接  SocketChannel channel = svrChannel.accept(); |
| 步骤七：设置客户端链路为非阻塞模式  channel.configureBlocking(false);  channel.socket().setReuseAddress(true); |
| 步骤八：将新接入的客户端连接注册到Reactor线程的多路复用器上，监听读操作，读取客户端发送的网络消息  SelectionKey key = socketChannel.register(selector,SelectionKey.OP\_READ,ioHandler); |
| 步骤九：异步读取客户端请求消息到缓冲区  int readNumber = channel.read(receivedBuffer); |
| 步骤十：对ByteBuffer进行编解码，如果有半包消息指针reset，继续读取后续的报文，将解码成功的消息封装成Task，投递到业务线程池中，进行业务逻辑编排  Object message = null;  while(buffer.hasRemain()){       byteBuffer.mark();       Object message = decode(byteBuffer);       if(message == null){            byteBuffer.reset();            break;       }       messageList.add(message);  }  if(!byteBuffer.hasRemain()){       byteBuffer.clear();  }else{       byteBuffer.compact();  }  if(messageList !=null & !messageList.isEmpty()){       for(Object messageE : nessageList){           handlerTask(messageE);       }  } |
| 步骤十一：将POJO对象encode成ByteBuffer，调用SocketChannel的异步write接口，将消息异步发送给客户端  ocketChannel.write(buffer); |

### 5.4Netty特点

1. 高性能、异步事件驱动NIO框架，它提供了对TCP、UDP和文件传输的支持；
2. 更高效的socket底层，对epoll空轮训引起的CPU占用飙升在内部处理，避免直接NIO陷阱，简化NIO的处理方式；
3. 采用多种decoder/encoder支持，对TCP粘包/半包自动化处理；
4. 可使用接收/处理线程池，提高连接效率，对重连、心跳检测简单支持；
5. 可配置IO线程数、TCP参数，TCP接收和发送缓冲区使用直接内存代替堆内存，通过内存池的方式循环利用ByteBuffer；
6. 通过引用计数及时申请释放不再引用的对象，降低GC频率；
7. 使用单线程串行化的方式，高效的Reactor线程模型；
8. 大量使用volatile、CAS和原子类，线程安全类的使用，读写锁的使用；

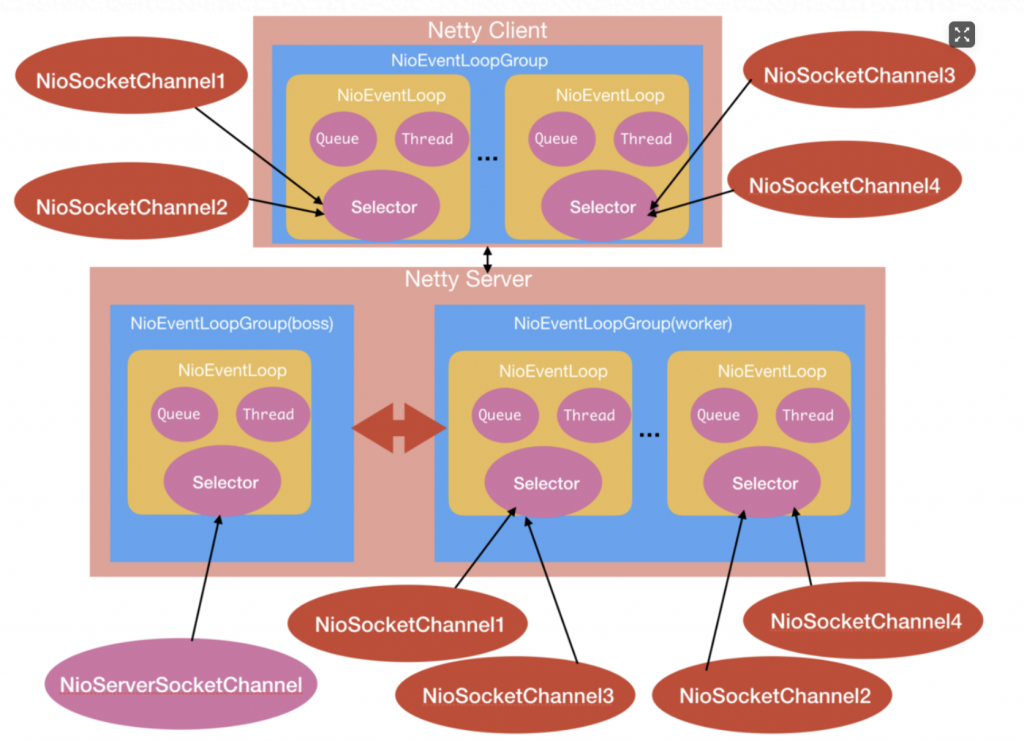
### 5.5Netty的Reactor线程模型

常见的Reactor线程模型有三种，分别如下：

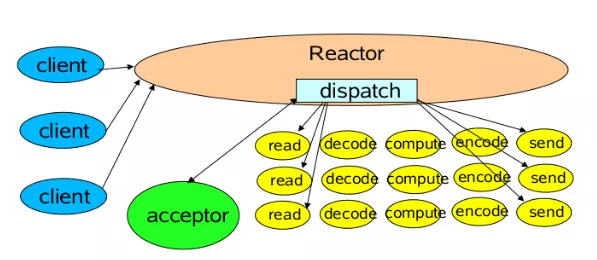
·Reactor单线程模型

·Reactor多线程模型

·主从Reactor多线程模型



#### ·Reactor单线程模型



Reactor 单线程模型使用的是一个 NIO 线程， NIO 使用的是非阻塞 I/O，所有的 I/O 操作都不会阻塞，所以一个线程可以处理多个 TCP 连接请求。

对于一些小容量应用场景，可以使用单线程模型，但是对于高负载、大并发的应用却不合适，主要原因如下：

· 一个NIO线程同时处理成百上千的链路，性能上无法支撑。即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送；

· 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往进行重发，这更加重了NIO线程的负载，最终导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈；

· 可靠性问题。一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通讯模块不可用，不能接收和处理外部信息，造成节点故障。

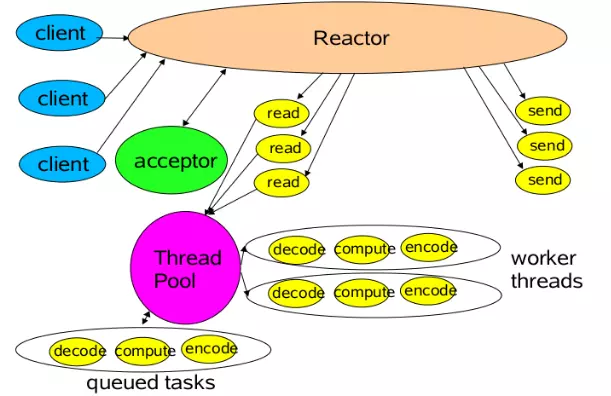
#### ·Reactor多线程模型

Reactor多线程模型与单线程模型最大区别就是有一组 NIO 线程处理 I/O 操作，它的特点如下：

·有一个专门的 NIO 线程用于监听服务端，接收客户端的TCP连接请求；

·网络I/O读、写操作等由一个NIO线程池负责，线程池可以采用标准的 JDK 线程池实现（Netty 扩展了 JDK 线程池），它包含一个任务队列和N个可用的线程，由这些 NIO 线程负责消息的读取、解码、编码和发送；

·1 个 NIO 线程可以同时处理 N 条链路，但是 1 个链路只对应 1 个 NIO 线程，防止发生并发操作问题。

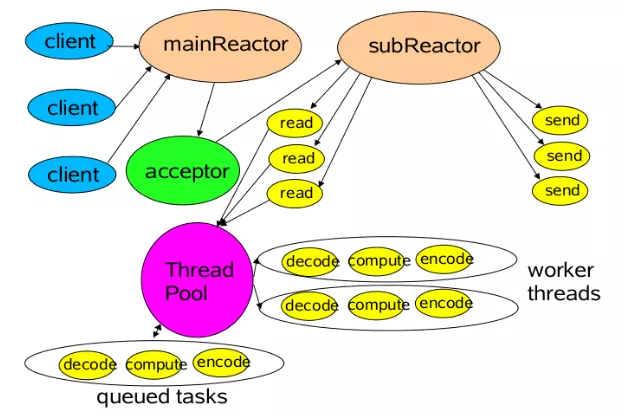


在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手信息进行安全认证，认证本身非常损耗性能。这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型--主从Reactor多线程模型。

#### ·主从Reactor多线程模型

服务端用于接收客户端连接的不再是1个单独的NIO线程，而是一个独立的NIO线程池。Acceptor接收到客户端TCP连接请求处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到I/O线程池（sub reactor线程池）的某个I/O线程上，由它负责SocketChannel的读写和编解码工作。

Acceptor线程池只用于客户端的登录、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的I/O线程上，有I/O线程负责后续的I/O操作。



#### Netty 启动示例

|  |
| --- |
|  |

每一个 EventLoopGroup 都是 Reactor 的线程池。ServerBootstrap.group 需要接受两个参数 EventLoopGroup 参数。 一个是处理接收客户端的 TCP 连接（NIO 的  
SelectionKey.OP\_CONNECT），另一个是处理 I/O 相关的操作（NIO 的  
SelectionKey.OP\_READ, SelectionKey.OP\_WRITE）。

Netty单线程模型

|  |
| --- |
|  |
| 创建只有一个线程的 Reactor 线程池。把处理接收客户端 TCP 连接的 Reactor 线程池和处理I/O相关操作的 Reactor 线程池都是使用这个只有一个线程的 Reactor 线程池。 |

Netty多线程模型

|  |
| --- |
|  |
| 创建两个 Reactor 线程池，处理客户端 TCP 连接的线程池只有1个线程，而处理 I/O 的Reactor 线程池有多个线程处理。不知道线程数则默认是 2\* CPU 个数。 |

Netty 主从多线程模型

|  |
| --- |
|  |
| 创建两个 Reactor 线程池，处理客户端 TCP 连接的线程池和处理 I/O 操作的线程池都是多个线程处理。 |

<https://www.jianshu.com/p/6ae30cf5ae9e>

Netty中的Reactor模型主要由多路复用器(Acceptor)、事件分发器(Dispatcher)、事件处理器(Handler)组成，可以分为三种。

### 5.6TCP粘包/拆包的原因及解决方法

#### 拆包和粘包是什么

正常：



粘包：



粘包和拆包：





#### 为什么会发生TCP粘包、拆包？

发生TCP粘包、拆包主要是由于下面一些原因：

1. 应用程序写入的数据大于套接字缓冲区大小，这将会发生拆包。

2.应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包。

3.进行MSS（最大报文长度）大小的TCP分段，当TCP报文长度-TCP头部长度>MSS的时候将发生拆包。

4.接收方法不及时读取套接字缓冲区数据，这将发生粘包。

#### 粘包、拆包解决办法

TCP本身是面向流的，作为网络服务器，如何从这源源不断涌来的数据流中拆分出或者合并出有意义的信息呢？通常会有以下一些常用的方法：

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。

#### 如何基于Netty处理粘包、拆包问题？

涉及到相关重要组件：

1. ByteToMessageDecoder
2. MessageToMessageDecoder

Netty中，涉及到粘包、拆包的逻辑主要在ByteToMessageDecoder及其实现中。

|  |
| --- |
| **protected void** callDecode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {  **try** {  **while** (in.isReadable()) {  **int** outSize = out.size();   **if** (outSize > 0) {  *fireChannelRead*(ctx, out, outSize);  out.clear();   *// Check if this handler was removed before continuing with decoding.  // If it was removed, it is not safe to continue to operate on the buffer.  //  // See:  // - https://github.com/netty/netty/issues/4635* **if** (ctx.isRemoved()) {  **break**;  }  outSize = 0;  }   **int** oldInputLength = in.readableBytes();  decode(ctx, in, out);   *// Check if this handler was removed before continuing the loop.  // If it was removed, it is not safe to continue to operate on the buffer.  //  // See https://github.com/netty/netty/issues/1664* **if** (ctx.isRemoved()) {  **break**;  }   **if** (outSize == out.size()) {  **if** (oldInputLength == in.readableBytes()) {  **break**;  } **else** {  **continue**;  }  }   **if** (oldInputLength == in.readableBytes()) {  **throw new** DecoderException(  StringUtil.*simpleClassName*(getClass()) +  **".decode() did not read anything but decoded a message."**);  }   **if** (isSingleDecode()) {  **break**;  }  }  } **catch** (DecoderException e) {  **throw** e;  } **catch** (Throwable cause) {  **throw new** DecoderException(cause);  } } |
| 1. 当out中有Message的时候，直接将out中的内容交给后面的channel handler去处理。 2. 当用户逻辑把当前channel handler移除的时候，立即停止对网络数据的处理。 3. 记录当前in中可读字节数。 4. decode是抽象方法，交给子类具体实现。 5. 同样判断当前channel handler移除的时候，立即停止对网络数据的处理。 6. 如果子类实现没有分理出任何message的时候，且子类实现也没有动bytebuf中的数据的时候，这里直接跳出，等待后续有数据来了再进行处理。 7. 如果子类实现没有分理出任何message的时候，且子类实现动了bytebuf中的数据，则继续循环，直到解析出message或者不在对bytebuf中数据进行处理为止。 8. 如果子类实现解析出了message但是又没有动bytebuf中的数据，那么是有问题的，抛出异常。 9. 如果标志位只解码一次，则退出。 |
| Netty已经提供了一些基于不同处理粘包、拆包规则的实现： |
| DelimiterBasedFrameDecoder是基于消息边界方式进行粘包拆包处理的。  FixedLengthFrameDecoder是基于固定长度消息进行粘包拆包处理的。  LengthFieldBasedFrameDecoder是基于消息头指定消息长度进行粘包拆包处理的。  LineBasedFrameDecoder是基于行来进行消息粘包拆包处理的。  用户可以自行选择规则然后使用Netty提供的对应的Decoder来进行具有粘包、拆包处理功能的网络应用开发。 |

<https://www.cnblogs.com/panchanggui/p/9748204.html>

出现粘包和半包现象,是因为TCP当中,只有流的概念,没有包的概念

由于网络原因 一次可能会来 0.5/1 /2/ 2.5/ 。。。。个包，当接收到时 要先看看那这个包中有多少个完整的包。把完整的包都处理了 也就是说把x都处理了。剩下的0.5留在接收区中，等待下次接收；

粘包 为x.5个包

半包 为0.5个包

重发的原因：

这回接收到的就是0.5+1.5/0.5+1.3/0.5+0.5..... 把完整的包都处理了，有残缺的扔掉 0.8的。

|  |
| --- |
| 一般情况 接收到正确的后都要给发送端一个应答。不给应答的算超时，发送端将重发。  有头没尾的不能扔  没头有尾的可以扔  有头有尾但缺东西可以扔  有头有尾不缺东西不能扔 |

可以使用UDP协议.这样可以就可以区分每个包了.但是要确保包的丢失处理.为了提到效率,可以考虑写一个滑动窗口进行收发包.

若采用TCP协议进行传输,就要将每个包区分开来.可以有三种方式.因为TCP是面向流的.流只有打开和关闭,你要用一个流传输多个包,那就要向办法区分出每个包.

·发送同样大小的包，填充’\0’

·流按字符处理,抽出一个字符做转义字符\r\n

·先将这个包的长度发送给对方(一般是4个字节表示包长),然后再将包的内容发送过去

一般处理是： 一个BUFFER，用于保存当前连接的读缓存

有数据时，Buffer = Buffer + DataIn，不停的接收

收完成后，开始解析Buffer,

根据包的协议，不停的解析Buffer，并形成一个个包进行处理，处理后，Buffer = Buffer - Data，并继续解包。

### 5.7了解哪几种序列化协议

·Java序列化：ObjectIn/Out，无法跨语言、序列化码流大、性能差；

·XML：人机可读性高，文件庞大，传输占带宽；做配置、实时数据转换SOAP；

·JSON：轻量、易读写、可扩展；不适合ms级应用，Ajax请求；

·Fastjson：假定有序快速匹配；协议交互、Web输出；

·Thrift：不仅序列化，而且是RPC框架，跨防火墙不安全，不能与HTTP公共，调试难；

·Protobuf：将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。优点：序列化后码流小，性能高、结构化数据存储格式（XML JSON等）、通过标识字段的顺序，可以实现协议的前向兼容、结构化的文档更容易管理和维护。缺点：需要依赖于工具生成代码、支持的语言相对较少，官方只支持Java 、C++ 、python。适用场景：对性能要求高的RPC调用、具有良好的跨防火墙的访问属性、适合应用层对象的持久化

#### 如何选择序列化协议？

具体场景

·对于公司间的系统调用，如果性能要求在100ms以上的服务，基于XML的SOAP协议是一个值得考虑的方案。

·基于Web browser的Ajax，以及Mobile app与服务端之间的通讯，JSON协议是首选。对于性能要求不太高，或者以动态类型语言为主，或者传输数据载荷很小的的运用场景，JSON也是非常不错的选择。

·对于调试环境比较恶劣的场景，采用JSON或XML能够极大的提高调试效率，降低系统开发成本。

·当对性能和简洁性有极高要求的场景，Protobuf，Thrift，Avro之间具有一定的竞争关系。

·对于T级别的数据的持久化应用场景，Protobuf和Avro是首要选择。如果持久化后的数据存储在hadoop子项目里，Avro会是更好的选择。

·对于持久层非Hadoop项目，以静态类型语言为主的应用场景，Protobuf会更符合静态类型语言工程师的开发习惯。由于Avro的设计理念偏向于动态类型语言，对于动态语言为主的应用场景，Avro是更好的选择。

·如果需要提供一个完整的RPC解决方案，Thrift是一个好的选择。

·如果序列化之后需要支持不同的传输层协议，或者需要跨防火墙访问的高性能场景，Protobuf可以优先考虑。

|  |
| --- |
| protobuf的数据类型有多种：bool、double、float、int32、int64、string、bytes、enum、message。protobuf的限定符：required: 必须赋值，不能为空、optional:字段可以赋值，也可以不赋值、repeated: 该字段可以重复任意次数（包括0次）、枚举；只能用指定的常量集中的一个值作为其值；  protobuf的基本规则：每个消息中必须至少留有一个required类型的字段、包含0个或多个optional类型的字段；repeated表示的字段可以包含0个或多个数据；[1,15]之内的标识号在编码的时候会占用一个字节（常用），[16,2047]之内的标识号则占用2个字节，标识号一定不能重复、使用消息类型，也可以将消息嵌套任意多层，可用嵌套消息类型来代替组。  protobuf的消息升级原则：不要更改任何已有的字段的数值标识；不能移除已经存在的required字段，optional和repeated类型的字段可以被移除，但要保留标号不能被重用。新添加的字段必须是optional或repeated。因为旧版本程序无法读取或写入新增的required限定符的字段。  编译器为每一个消息类型生成了一个.java文件，以及一个特殊的Builder类（该类是用来创建消息类接口的）。如：UserProto.User.Builder builder = UserProto.User.newBuilder();builder.build()；  Netty中的使用：ProtobufVarint32FrameDecoder 是用于处理半包消息的解码类；ProtobufDecoder(UserProto.User.getDefaultInstance())这是创建的UserProto.java文件中的解码类；ProtobufVarint32LengthFieldPrepender 对protobuf协议的消息头上加上一个长度为32的整形字段，用于标志这个消息的长度的类；ProtobufEncoder 是编码类  将StringBuilder转换为ByteBuf类型：copiedBuffer()方法 |

### 5.8Netty的零拷贝实现？

非零拷贝：file.read(bytes) + socket.send(bytes)

|  |
| --- |
| IMG_256 |
| 这种方式需要四次数据拷贝和四次上下文切换： 1. 数据从磁盘读取到内核的read buffer 2. 数据从内核缓冲区拷贝到用户缓冲区 3. 数据从用户缓冲区拷贝到内核的socket buffer 4. 数据从内核的socket buffer拷贝到网卡接口的缓冲区 |

零拷贝:FileChannel.transferTo(Byte)

|  |
| --- |
| IMG_256 |
| 明显上面的第二步和第三步是没有必要的，通过java的FileChannel.transferTo方法，可以避免上面两次多余的拷贝（当然这需要底层操作系统支持） 1. 调用transferTo,数据从文件由DMA引擎拷贝到内核read buffer 2. 接着DMA从内核read buffer将数据拷贝到网卡接口buffer |

Netty中的零拷贝

|  |
| --- |
| 关于ByteBuffer，Netty提供了两个接口: 1. ByteBuf 2. ByteBufHolder  对于ByteBuf，Netty提供了多种实现： 1. Heap ByteBuf:直接在堆内存分配 2. Direct ByteBuf：直接在内存区域分配而不是堆内存 3. CompositeByteBuf：组合Buffer |

Netty的零拷贝体现在三个方面

|  |
| --- |
| 1. Netty的接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。  2. Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便的对组合Buffer进行操作，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer。  3. Netty的文件传输采用了transferTo方法，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致的内存拷贝问题。 |

### 5.9Selector 空轮训BUG

BUG：若 Selector 的轮询结果为空，也没有 wakeup 或新消息处理，则发生空轮

询，CPU 使用率 100%。

Netty 的解决办法：对 Selector 的 select 操作周期进行统计，每完成一次空的 select 操作进

行一次计数，若在某个周期内连续发生 N 次（512）空轮询，则触发了 epoll 死循环 bug。 重建 Selector，判断是否是其他线程发起的重建请求，若不是则将原 SocketChannel 从旧的

Selector 上去除注册，重新注册到新的 Selector 上，并将原来的 Selector 关闭。

### 5.10Netty 的高性能表现在哪些方面

·心跳检测：定期清理闲置会话incative(netty5);

·串行无锁化：Reactor线程模型；

·可靠性：链路有效检测，内存保护，内存池重用ByteBuffer保护编解码，优雅停机；

·安全性：协议SSL、TSL，单向、双向和第三方CA认证；

·高效并发编程：volatile、CAS、原子类；线程安全容器；读写锁；IO通信性能三原则：传输（AIO）、协议（http）、线程（主从多线程）；

·变压器：流量整形的作用，防止由于上下游网元性能不均衡导致下游被压垮，业务流中断；防止由于通信模块接收消息过快，后端业务线程不及时处理导致撑死，丢包，延迟大问题；

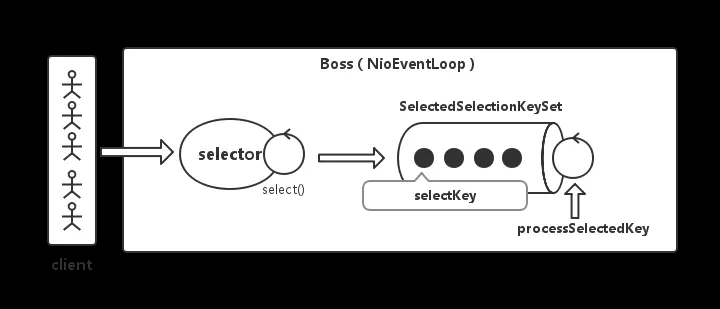
·TCP参数设置：SO\_RCVBUF和SO\_SNDBUF：建议128KB和256KB；

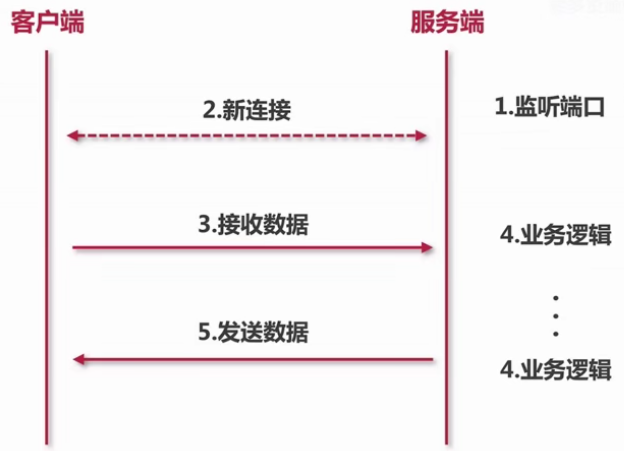
·SO\_TCPNODELAY：NAGLE算法，小包自动相连组大包，减少网络阻塞，但对于实时性敏感的应用场景关闭；

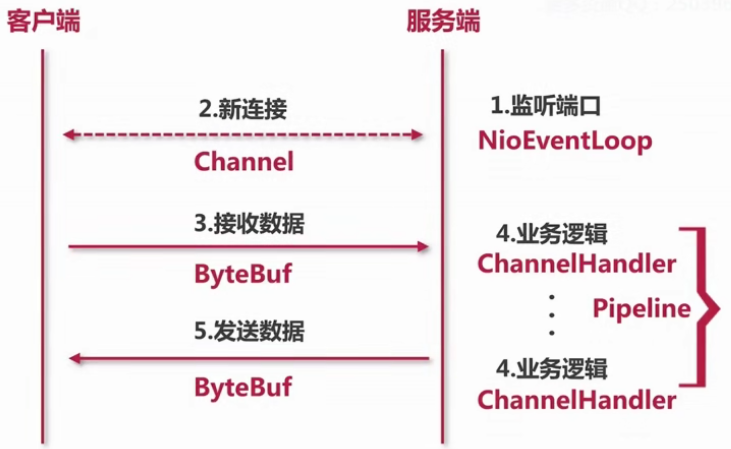
### 5.11NIOEventLoopGroup源码

|  |
| --- |
| 1. NioEventLoopGroup即MultiThreadEventExecutorGroup，内部维护一个类型为EventExecutor children[] ，默认为2\*CPUs，这样就构成了一个线程池，初始化EventExecutor时，NioEventLoopGroup重载newChild方法，所以children元素的实际类型为NioEventLoop； |
| 1. 线程启动时调用SingleThreadEventExecutor的构造方法，执行NioEventLoop类的run方法，首先会调用hasTask方法判断当前taskQueue是否有元素。如果有执行selectNow，最终执行selector.selectorNow，该方法立即返回。如果没有执行select(oldWakenUp)方法 |
| 1. Select(oldWakenUp)解决Nio空轮训Bug，selectCnt用来记录selector.select()方法的执行次数和是否执行过selector.selectNow()，如果触发epoll空轮训bug，则会反复执行selector.select(timeoutMillis),如果selectCnt达到阈值（默认512），则执行rebuildSelector犯法，进行重建，解决cpu占用100%bug； |
| 1. rebuildSelector方法先通过openSelector方法创建一个新的selector，然后将old selector的SelectionKey执行cancel，最后将old selector的channel重新注册到selector中，rebuild后，需要重新执行方法selectNow，检查受有已经ready的selectionKey； |
| 1. processSelectionKey处理I/O任务，当selectedKeys!=null时，调用processSelectionKeyOptimized方法，迭代selectedKeys获取就绪的I/O事件的selectionKey存放在数组selectedKeys中，然后为每个事件都调用processSelectedKey来处理OP\_READ、OP\_WRITE和OP\_CONNECT事件； |
| 1. runAllTasks方法（非IO任务），首先调用fetchFromScheduledTaskQueue方法，把scheduledTaskQueue中已经超过延迟执行时间的任务移到taskQueue中等待执行，然后依次从taskQueue中取到任务执行，没执行64个任务，进行耗时检查，如果已执行时间超过预先设定的执行时间，则停止执行非IO任务，避免非IO任务太多影响IO任务； |
| 1. 每个NioEventLoop对应一个线程和一个Selector，NioServerSocketChannel会主动注册到某一个NioEventLoop上的Selector上，NioEventLoop负责事件轮训； |
| 1. OutBound事件都是请求事件，发起者是Channel，处理者是unsafe，通过OutBound事件进行通知，传播方向是tail到head。InBound事件的发起者是unsafe，事件处理器者是Channel，是通知事件，传播方向是head到tail； |
| 1. 内存管理机制，首先会申请一大块内存Area，Arena由许多Chunk组成，而每个Chunk默认由2048个page组成。Chunk通过AVL树组织page，每个叶子节点表示一个Page，而中间节点表示内存矛，节点自己记录它在Arena中的偏移地址。当区域被分配出去后，中间节点上的标记会被标记，这样表示这个中间节点以下的所有节点都被分配了。大于8K的内存分配在poolChunkList中，而PoolSubpage用于分配小于8K的内存，它会把一个page分割成多段，进行内存分配。 |
| 1. ByteBuf的特点：只会自动扩容（4M），保证put方法不会抛出异常、通过内置的复合缓冲类型，实现零拷贝（Zero-Copy）；不需要调用flip来切换读写模式，读取和写入索引分开；方法链：引用计数基于AtomicIntegerFieldUpdater用于内存回收；PooledByteBuf采用二叉树实现一个内存池，集中管理内存的分配和释放，不用每次使用都新建一个缓冲区对象。UnpooledHeapByteBuf每次都会新建一个缓冲区对象。 |

Netty服务端socket在哪里创建？在哪里accpet？







#### 默认情况下，Netty服务端起多少线程？何时启动？

2\*CPU；doBind时注册Channel，调用所属eventLoop的execute方法，添加任务队列，判断inEventLoop是否是当前线程，否则startThread启动一个FastThreadLocal线程。

#### Netty是如何解决JDK的NIO空轮训Bug？

Select时截止时间没到，且任务队列为空，进行阻塞式select，即512次空轮训，重新rebuildSelector，把selectedKeys重新注册到新打开的selector上。

#### Netty如何保证异步串行无锁化？

所有操作抽象成task，存放至MPSC队列，按照截止时间顺序执行；

#### Netty在哪里检测有新连接接入？

Boss线程的第一个过程，轮训出Accet事件；Boss线程第二个过程，通过JDK底层的channel的accept方法去创建这条连接；

#### 新连接是怎样注册到NioEventLoop线程的？

Boss线程调用chooser的next方法，拿到一个NioEventLoop，然后将这条连接注册到这个NioEventLoop的selector上。

### 5.12业务中Netty的使用场景

1. 构建高性能、低时延的各种Java中间件，如MQ、分布式服务框架、ESB消息总线等，STUN；
2. Netty主要作为基础通信框架提供高性能、低时延的通信服务；
3. 共有或者私有协议栈的基础通信框架，例如基于Netty构建异步、高性能WebSocket协议栈；
4. 各领域应用，如大数据、游戏等，Netty作为高性能的通讯框架用于内部各模块的数据分发、传输和汇总等，实现模块之间高性能通信；