Java 面试怪圈内卷手册

公众号：面试怪圈

前序

大家好，我是可爱猪猪，非常有幸能够和大家在这份资料里面相见。

正如大家所知道的一样，我是一个程序员。可能你看到“可爱猪猪”的时候，觉得可能很 Q，

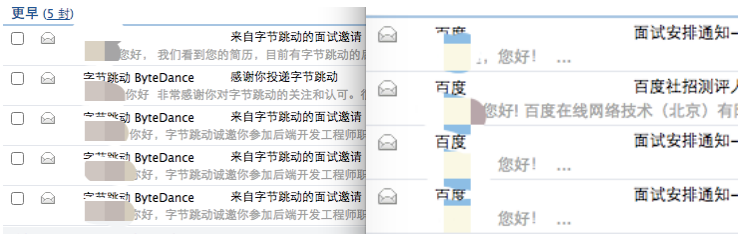
但是我是一个老程序员了，一个普普通通的老程序员。

可能大家好奇老程序员都去了哪里？其实我也挺好奇的。

反正，我在一个非 BAT 的厂里继续码，曾经也试图去挤进一些大厂，可是始终没有找到大厂的入口。些许有些遗憾吧。

我在 2 年前，创建了一个公众号“面试怪圈”，一直不温不火。文章也写过，个人网站也建过，曾经也有过很多想法。但是由于时间、家庭、工作各种算借口的理由，也从没有为面试怪圈做过什么贡献。

自己也三面过很多大厂，比如下面两个 B 字辈，也做过很多所谓大厂的算法题。



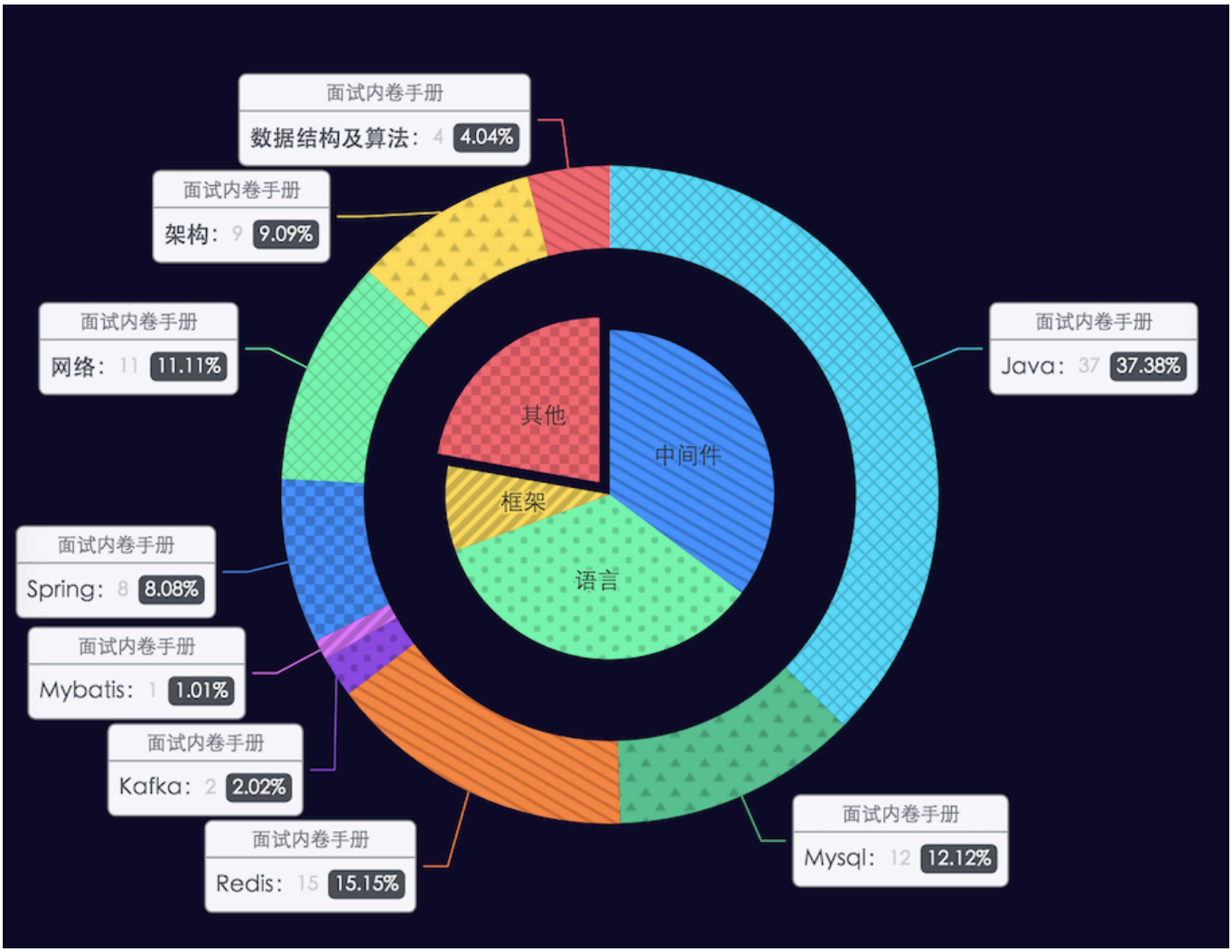
虽然没有然后，但却留下了宝贵的过程，本文档将 Java 面试题也开源了。

面试现状太卷了，我们只能不辞辛劳地卷下去。

资料倾心总结，关注公众号，感谢大家❤！

此外，定期会发布资料到 www.mianshiguaiquan.com。欢迎大家收藏

开箱清单


面试内卷手册
数据结构及算法：4 4.04%
面试内卷手册
架构：9 9.09%
面试内卷手册
网络：11 11.11%
面试内卷手册
Java：37 37.38%
其他
中间件
框架
语言
面试内卷手册
Spring：8 8.08%
面试内卷手册
Mybatis：1 1.01%
面试内卷手册
Kafka：2 2.02%
面试内卷手册
Redis：15 15.15%
面试内卷手册
Mysql：12 12.12%

版权说明

不惧大厂

百度、阿里、腾讯、字节、京东、美团....

对，大厂！

考算法、考逻辑、考架构、考基础....

总之，听上去要的是全能性造火箭人才

我去面试，却发现是过来者吓怕了后来者

比如说手撕算法，据说要刷 LeeCode、牛客才敢尝试

然后我并没有刻意去刷，百度、字节、美团的算法题我都做过，并且也都编码成功...

其实，并没有那么难，解题，更注重的思考问题的方法和逻辑

毕竟，一般一道题只有 20 分钟的编码时间

如果太难，短时间一般人很难完成的

话虽如此，但是我们还是应具备最基础数据结构的了解

以及常见问题的解题方法

所以，不要把考题想的太难，冷静思考。

比如，字节有二面有两个算法题，其中一道是：

第一次走一步，第二走两步，第三次走三步，每一次都可以向前或者向后走。

问：走多少次，可以走到 N 的位置。

思考一下吧?

题外话

不管是什么厂

只要你面试的 Java 岗位或者 Go 岗位

其实有经验的面试官从项目经验问起

然后迁出你对中间件、基础知识、架构等的理解

以下分享的内容，主要以广度和思路为主，若需要获取详细的答案，基本可以百度到。

JAVA 篇

LinkedList 和 ArrayList 的区别

数据结构:linkedlist 是基于双向链表,arraylist 是基于数组.

读写性能:尾部追加无性能差异,如果中间位置插入,linkedlist 要快,arraylist 随机查找性能更快

扩容机制:arraylist 空间不足时,需要进行扩容.

空间占用:linked 空间占用要稍微大一些.

LinkedHashMap 和 TreeMap 有序是一样的吗?

LinkedHashMap 是按照插入顺序排序

TreeMap 按照 key 的值排序

HashMap 原理

计算 key 对应的 HashCode,该 HashCode 与自身右移 16 位进行异或运算,对 Hash 值进行扰动,减少 Hash 冲突

根据 HashCode 计算取 mod 计算该 key 所在数组中的位置(n-1)&hash,

如果对应的位置无元素,则新建一个节点放到该位置

如果该位置上存在元素,则判断该元素的类型如果是树节点,则遍历树在树中追加元素

如果是非树节点则追加到链表的尾部,如果链表超过 8 个,则转换为红黑树

如果 put 元素的个数,超过数组大小*负载因子的话,则进行扩容,

扩容的时候生成原数组大小两倍的空间,然后将原数组的元素依次拷贝到新的数组.

采用红黑树的好处是如果 hash 冲突比较严重时,提升查询的性能.避免 hash 攻击,造成安全问题

concurrentHashMap

JDK1.7 基于 HashMap 的基础上增加了分段锁,通过减小锁的力度来提升并发能力,默认有 16 个分段锁,每个分段锁又是一个数组+链表的结果,segment 继承了 reentrylock 实现的,一个线程只能锁一个 segment.

而 JDK1.8 是基于 CAS+Synchronize 实现的.数据结构也是数组+链表+红黑树,当某个数组的位置为空时,采用 CAS 的乐观锁方式来进行 put 值,当数组的位置不为空时,会对节点进行锁定.因此 JDK1.8 的锁的力度更小,相对 JDK1.7 来说并发能力更好.

AQS 的实现

AQS 是一个抽象的 FIFO 队列同步器,维护了一个 volitile 语义的共享资源变量 state,和一个 FIFO 的线程等待队列.例如 ReentrantLock、信号量、CountDownLatch 就是基于 AQS 实现的.

AQS 的核心思想是,如果被请求的资源空闲,则将当前请求资源的线程设置为 owner 线程,并将资源设定为锁定状态,如果资源被占有,将线程添加到消息队列中,然后通过自旋的方式,不断尝试获取锁.

CountDownLatch、CyclicBarrier、Semaphore

CountDownLatch:维护计数器,当计数器大于 0 时,await 的线程处于阻塞状态,当计数器为 0 时,处于 await 的线程被激活

CyclicBarrier:多个线程在栅栏处等待,等到足够数量后,释放栅栏,栅栏可反复使用

Semaphore:计数器信号量,通过 acquire 和 release 控制并发运行的线程数

volatile 的作用

volatile 的作用主要有两个,一个是防止指令重排序/一个是保持内存的可见性.

- 指令重排序

JVM 编译器会对指令进行重排序以提升并发性能.会让指令按照非程序语义的顺序执行.有编译重排序,处理器指令重排序、内存重排序

而指令重排序在多线程情况下可能存在问题,比如单例模式,多线程下可能获取未初始化的对象.

创建对象的过程

1.申请内存地址.

2.在内存地址上初始化对象

3.将引用指向内存地址

假设按照 1->3->2 的顺序执行.就会存在问题.

我们通过增加 volatile 关键字来禁用重排序

- 内存可见

可见性就是一个线程修改了某个变量的值,其他线程就能立即看到修改的值.

Jvm 内存模型规定了所有的变量都是存储在主存中的,每个线程还有自己的工作线程,工作线程保存了主存变量的副本,不同线程间通讯需要先同步到主存.

使用 volatile 修饰能在每次变量发生变化后,强制刷新到主存.

- 内存屏障

阻止屏障两侧的指令重排序；即屏障下面的代码不能跟屏障上面的代码交换执行顺序

强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。

对于 Load Barrier 来说，在指令前插入 Load Barrier，可以让 CPU 高速缓存中 j 的数据失效，强制从新从主内存加载新数据；

对于 Store Barrier 来说，在指令后插入 Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

对象的组成

- 对象头

运行时数据(8 字节,Hash 码、分代年龄、锁状态标志、偏向锁的线程 ID,指向 monitor 的指针)

类指针(8 字节,压缩 4 字节)

数组长度(4 字节)

- 对象体

对齐填充(8 字节的整数倍)

Wait 和 sleep 的区别

sleep 是让出 CPU 的时间片,但是并不释放他所持有的对象锁.

wait 是放弃对象锁,进入 ObjetMonitor 的 waitSet 中,notify 后重现尝试去竞争锁

synchronize 关键字

synchronize 是解决 Java 并发问题的最常用、最简单的办法.

- 从线程安全的角度考虑主要作用有三个

1.原子性:确保线程互斥的访问同步代码块

2.可见性:保证共享变量的修改能及时可见,通过 JMM 模型的,当解锁的时候,将共享变量刷新到主存,线程加锁的时候,清空工作线程的内存,从主存中重新读取变量值.(反编译可以看到在 synchronized 进入之前执行 store,即将线程内存副本同步)

3.有序性:解决重排序问题,即一个 unlock 先行发生于后面的对同一个锁的 lock 操作

- synchronize 的实现原理

反编译得到 java 的汇编指令,synchronized 的代码块可以发现有两个关键字,monitorenter 和 monitorexit 包围代码块.

每一个对象都是一个监视锁,当线程执行 monitorenter 时,会尝试获取监视锁的所有权,一旦线程获取到了,则 moniter 的进入数+1,同一线程再次进入的时候,进入数累加,如果其他线程进入则会阻塞,当执行 monitorexit 的时候,monitor 的进入数会-1,进入数为 0 时,释放 monitor 监控锁.

monitor 对象由对象的头部指针指向,每个 monitorObject 有几个重要属性:counter(记录重入的次数)、EntryList(阻塞状态的线程会进入该列表)、WaitSet(处于等待的线程会加入到这个列表)

JDK6 对锁优化

自适应自旋锁:

线程的阻塞和唤醒需要从用户态到核心态的转换,频繁的阻塞和唤醒对 CPU 来说是一件负担很重的事情,因此,为了短时间的阻塞和唤醒 CPU 是很不值得.

自旋锁是一个线程试图获取锁的时候,如果这个锁被其他线程占有,就一直在循环检测锁是否释放,但是他占用了 CPU 的处理时间,因此自旋锁只适合短时间锁获取.因此,自旋时间太长或者次数太多,会转为阻塞.

自适应自旋锁如果本次自旋成功了,下次自旋的次数会更多,虚拟机认为既然上次自旋成功了,那么此次自旋很有可能成功.反之,如果很少能自旋成功,那么下次自旋次数会减少.以免浪费处理器资源.

锁消除:

如果虚拟机检测同步代码块不存在多线程竞争,则消除锁.比如在单线程下使用 HashTable 和 Vector 等.

锁粗化:

如果存在多个连续的同步加锁可能造成的不必要的性能损耗,因此可以合并成一个范围更大的锁.

比如循环往 vector add 数据的时候,锁会粗化到循环外

偏向锁:

Hotspot 的作者研究发现大多数场景不存在线程竞争的.而只是由单个线程多次获得.为了让锁的代价更低引入了偏向锁.偏向锁的通过对象头的偏向锁的线程 ID 和锁状态标志来判断对象锁是否被占用,这样设计的主要目的是减少 CAS 操作以及较少 Cache 一致性带来的开销.

锁升级过程

轻量级锁:

相对传统重量级锁的互斥量带来的性能消耗,轻量级锁采用 CAS+自旋的方式,如果仍未获得锁,则转换为重量级锁.

在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的 Mark Word 的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图：

拷贝对象头中的 Mark Word 复制到锁记录（Lock Record）中；

拷贝成功后，虚拟机将使用 CAS 操作尝试将锁对象的 Mark Word 更新为指向 Lock Record 的指针，并将线程栈帧中的 Lock Record 里的 owner 指针指向 Object 的 Mark Word。

如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象 Mark Word 的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，这时候线程堆栈与对象头的状态如图所示。

LockRecord(栈中存储,其中 owner 指向 marketword) MarkWord(其中 stackpointer 指向 LockRecord)

重量级锁:

监视器锁对接底层操作系统中的互斥量（mutex），这种同步成本非常高，包括操作系统调用引起的内核态与用户态之间的切换。线程阻塞造成的线程切换等

Object 的方法

toString wait notify notifyAll hashCode equals clone finalize getClass

浅 clone 和深 clone

浅 clone:指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝对象包含的引用指向的对象。

深 clone:不仅拷贝对象本身，而且拷贝对象包含的引用指向的所有对象。

hashCode 和 equals

两个都是用来判断对象是否相等的依据.hashCode 性能较好,根据对象生成的 hashCode,直接比较判断,而 equals 相对较慢,需要有一定的逻辑判断.

重写 equals 一定要重写 hashCode,hashCode 相等不一定 equals,equals 相等一定 hashCode 相等.

HashMap、Set 等就先根据 hashCode 做判断以提升性能

悲观锁和乐观锁

悲观锁:总是假设最坏的情况,每次去拿数据的时候,都认为别人会修改.例如关系型数据库的库锁、表锁、行锁等,以及 Java 中的 synchronize 和 reentrantLock 等独占锁等.

乐观锁:总是假设最好的情况,每次去拿数据的时候,都认为别人不会修改.如果存在锁竞争,可使用 CAS 算法,乐观锁适用于多读的情况.Atomic 下就是乐观锁的实现.

JVM 内存结构

程序计数器:线程私有,当前线程所执行字节码的行号指示器,解释器根据通过该计数器选取下一行字节码指令

虚拟机栈:线程私有,每个方法被执行的时候都会创建一个栈帧(局部变量表(局部变量槽)、操作数据栈、动态链接、方法的出口).

本地方法栈

堆:线程共享.存放对象实例.新生代(eden->8、survivor1->1、survivor2->1)->1、老年代->2

方法区(jdk7 永久代 jdk8 方法区):类信息、常量、类变量、即时编译器的缓存代码

直接内存:NIO 利用本机内存

对象创建过程

类加载:常量池中定位类的符号引用,并检查这个类是否被加载,如果没有加载,执行类的加载过程.

为对象分配空间:一般根据垃圾回收器决定使用指针碰撞的方式(serial、parView 使用复制算法,空间是连续的)还是空闲列表(CMS-使用标记-清除算法,空间不连续)的方式分配内存.

由于在分配空间时,存在并发的情况导致线程不安全,一般有两种方式解决:CAS+重试或者 TLAB 的方式分配

设置对象头

执行构造函数

类加载的过程

加载:

通过一个类的全限定名来获取定义此类的二进制字节流

将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

连接:

验证:验证字节码是否符合 Class 文件的规范.

准备:为类变量分配空间并初始化.

解析:将常量池中字符引用替换为直接引用

初始化:静态变量的赋值和静态代码块的执行

反射中 Class.forName()和 ClassLoader.loadClass()的区别

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

第 2 个 boolean 参数表示类是否需要初始化， Class.forName(className)默认是需要初始化。一旦初始化，就会触发目标对象的 static 块代码执行，static 参数也也会被再次初始化。

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

ClassLoader.loadClass(className,false);

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

类加载器&双亲委派机制

启动类加载器:加载 JAVA_HOME/lib 下的类

扩展类加载器:加载 JAVA_HOME/lib/ext 下的类

应用程序类加载器:加载用户类路径(ClassPath)下的类

一个类加载器收到了类加载的请求,它首先不会自己先尝试去加载这个类,而是把这个请求委派给父类加载器去完成.直到传送到顶层的启动类加载器加载.只有父加载器无法完成类加载请求,才由子类加载器完成.

作用:

防止同一个类重复加载

出于安全和稳定性考虑,保证了核心的类不能被加载替换

如何判断对象是否已死?

一般对象存活的分析方法有两种:引用计数法(无法解决循环依赖的问题)

可达性分析算法:

通过 GCRoot 作为起始节点,根据引用关系向下搜索,如果某个对象到 GCRoot 之间没有引用链,则这个对象可以回收.但是如果 finalize 中对 this 对象重新引用则对象就不一定可以回收.

哪些对象可以作为 GCRoot 呢?

虚拟机栈中的引用对象

方法区静态属性引用的对象

方法区常量引用的对象

被同步锁持有的对象等等

垃圾回收算法及其优缺点

- 标记清除算法

执行效率不稳定,如果大量对象需要回收需要执行大量的标记清除

内存碎片化问题

- 标记复制算法

优点：只需要复制存活的对象；解决了内存碎片化问题

缺点:由于预留空间进行复制,造成了空间的浪费

- 标记整理算法

解决了内存碎片化的问题

如果存活的对象很多,处理时间变慢

- 分代垃圾回收

不同年龄代对象的存活周期不一样分代处理更有优势

垃圾回收器

JDK7\JDK8 默认垃圾收集器:Parallel Scavenge+Parallel Old

JDK9 默认垃圾收集器:G1

Serial New:单线程垃圾收集器,采用标记-复制算法

ParNew:是 Serial New 的多线程版本,采用标记-复制算法

Parallel Scavenge:跟 ParNew 非常相似,它关注的是一个可控的吞吐量.可通过参数设置

垃圾回收的最大停顿时间 -XX:MaxGCPauseMillis

设置垃圾收集时间占总时间的比率 -XX:GCTimeRatio

Serial Old:单线程垃圾收集器,采用标记-整理算法

Parallel Old:跟 Parallel Scavenge 类似,采用标记整理算法.

CMS: -|低停顿的垃圾回收器,注重响应速度.标记清除算法

-|执行步骤-|初始标记:标记跟 GCRoot 直接关联的对象,时间比较短,停顿用户线程

|并发标记:标记从 GCRoot 遍历整个对象图

|重新标记:修正并发标记期间因用户线程运行产生变动的标记,(多线程)

|并发清除:清除标记为死亡的对象

-|缺点:对处理器资源非常敏感,默认(处理器+3)/4 用于垃圾回收.

最终并发标记产生的浮动垃圾,导致无法分配对象,会出现并发失败的问题,会启动 serialOld 垃圾回收,STW,重新垃圾回收.

内存碎片化,如果无内存可分配,还需要进行内存整理.

G1:-|将内存分为大小相等 region 的区域,同样具有老年代和新生代的概念,也是一款并发垃圾收集器,低响应时间,优先回收收益价值最大的对象.标记整理+复制

-|执行步骤:-|初始标记

|并发标记(只有这一个是并发的)

|最终标记(多线程)

|筛选回收:根据用户要求的响应时间,优先清除收益最大的 region,将存活对象复制到空的 region,清理旧的 region 的全部空间.(多线程)

ThreadLocal 的实现原理

当向 ThreadLocal 中 set 一个变量的时候,实际上先获取当前线程的 threadlocals,它是一个 Map 结构,key 为 ThreadLocal 对象的弱引用,Value 为设置的值.

Entry 中 key 为弱引用的目的就是当 ThreadLocal 置空时,Entry 中的 key 与 threadlocal 只剩下一弱引用,当垃圾回收时,就可以回收 threadlocal 对象.

使用线程池存在什么问题,因为线程池每次获取的可能不是一个线程,有的时候甚至线程会消亡,有的线程可能永驻线程池,因此会导致 threadlocal 的 set 和 get 不一致,或者内存溢出等问题.

强软弱虚

强引用:垃圾回收即使报 OOM,也不会回收这种对象

软引用:内存空间不足时,也就是 OOM 之前,只被软引用的对象,会回收这种对象.

弱引用:垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存

虚引用:跟没有任何引用一样,随时都有可能被回收,用来跟踪对象被垃圾回收的活动

设计模式

比如创建型的:单例、工厂、构建者

行为:策略、模版、观察者

结构:适配器、代理、享元、外观、装饰者

JDK8\9 特征

stream

lamda 表达式

接口支持默认方法

Optional

动态代理和 Cglib 有什么区别

java 动态代理-|利用 Java 的反射机制生成了代理接口的匿名类

|代理的类必须要实现接口

cglib-|利用 ASM 开源包,对代理对象的 class 文件加载进来,通过修改字节码生成的子类来实现的

|代理的类不需要实现接口

synchronized 与 ReentrantLock 的区别

synchronized 是 Java 语言的关键字,是 JVM 实现的.支持同步代码块和同步方法

ReentrantLock 是 JDK1.5 以后提供的 API 层面的基于 AQS 的互斥锁.

性能上经过优化没有太大差异,官方更建议使用 synchronized,如果需要锁的更精细化的操作,可以考虑 ReentrantLock,比如:

公平锁和非公平锁

锁超时

锁中断

tryLock 等.

死锁

为了保证线程的安全,我们会加锁,但是如果锁运用的不合理会产生死锁.

如何避免死锁:

保证锁的顺序一致性

引入锁超时机制

可用下载静态代码检测工具 findBugs

减少锁的嵌套和粒度

线程的状态

New

Runnable

Wait

Time_Wait

Blocked

Terminated

jvm 监控命令

jps 打印进程信息

jinfo 输出 java 进程环境变量及启动参数

jinfo -flags 打印所有的启动参数

jstat 查看内存的使用情况

jstack 输出线程堆栈日志

jmap 导出堆内存的转储文件

jhat 分析转储文件

线程池

内容太多参考链接：https://www.jianshu.com/p/7726c70cdc40

1、corePoolSize（线程池基本大小）：当向线程池提交一个任务时，若线程池已创建的线程数小于 corePoolSize，即便此时存在空闲线程，也会通过创建一个新线程来执行该任务，直到已创建的线程数大于或等于 corePoolSize 时，（除了利用提交新任务来创建和启动线程（按需构造），也可以通过 prestartCoreThread() 或 prestartAllCoreThreads() 方法来提前启动线程池中的基本线程。）

2、maximumPoolSize（线程池最大大小）：线程池所允许的最大线程个数。当队列满了，且已创建的线程数小于 maximumPoolSize，则线程池会创建新的线程来执行任务。另外，对于无界队列，可忽略该参数。

3、keepAliveTime（线程存活保持时间）当线程池中线程数大于核心线程数时，线程的空闲时间如果超过线程存活时间，那么这个线程就会被销毁，直到线程池中的线程数小于等于核心线程数。

4、workQueue（任务队列）：用于传输和保存等待执行任务的阻塞队列。

5、threadFactory（线程工厂）：用于创建新线程。threadFactory 创建的线程也是采用 new Thread()方式，threadFactory 创建的线程名都具有统一的风格：pool-m-thread-n（m 为线程池的编号，n 为线程池内的线程编号）。

5、handler（线程饱和策略）：当线程池和队列都满了，再加入线程会执行此策略。

5 中 io 模型

阻塞 IO 模型（BIO）

等待数据准备和数据从内核态到用户进程的拷贝是阻塞的。一般改进方案是一个链接一个线程，

缺点：多线程会成为瓶颈。

非阻塞(NIO)

用户发起 read 操作，如果内核没有准备好并不会阻塞用户进程，而是返回一个 error。

一个用户进程不断询问数据准备好没有，如果准备好阻塞拷贝数据到用户进程。

缺点：检测时，cpu 占用率比较高。

多路复用 IO（阻塞）

事件驱动 IO，采用 select/poll 不断轮询注册的文件描述符，基于内核态的，只知道有事件来了，并不知道是哪一个有事件。

epoll 采用事件驱动替换轮询，同时 epoll 采用 mmap 的方式，减少内核态到用户态的拷贝。

信号驱动 IO

异步 IO(AIO)

用户进程发起 read 操作，就可以去做其他事情，不用产生任何的阻塞。等数据接收完，拷贝到用户态。向用户进程发送一个信号。

优先级队列

内部维护的了一个二叉堆数据，每次插入数据的时候，先检查是否需要扩容，增加原来的 50%,然后进行堆排序

延时队列

内部包装了一个优先级队列，take(),会自旋，获取队列第一个元素的延时时间，如果小于延时时间则取出，否则 await 一个延时时间。

数据结构及算法

红黑树和(AVL)平衡二叉树的区别

平衡二叉树:

用平衡因子差值判断是否平衡并通过旋转来实现平衡,左右子树高度不超过 1,是严格平衡的二叉树,因为旋转是非常耗时的,因此适合多读少些的情况.

红黑树:

从根结点到叶子结点的最长路径不能多于最短路径的 2 倍,是一种弱平衡二叉树.在插入和删除比较多的时候,适合用红黑树.

红黑树的特点

是一颗含有红黑节点能自动平衡的二叉树

每个节点要么是红色要么是黑色的

根节点是黑色的

每个叶子结点也是黑色的

每个红色节点的两个子节点一定是黑色的

任意节点到每个叶子结点的路径都包含相同的黑节点.

依靠左旋、右旋以及变色来保持树的平衡

进程调度算法

先来先服务的算法

最短进程优先算法

最短剩余时间算法(最短进程的抢占版本)

时间片轮循算法

最高优先级算法

快速排序算法

从 b 站搜索视频半小时内学会

就是一种挖坑填坑算法

先从数列中取出第一个数作为基准数,作为第一个坑. 并把基准数放到临时变量

然后从右侧和左侧开始扫描大于和小于该基准数,填入空缺的坑中,直到扫描位置重合就将数列分成两个区,然后每个区分别重复该步骤.

堆排序

从 b 站搜索视频半小时内学会

Redis 篇

redis 的持久化机制

rdb:是基于二进制快照的持久化方案,是一种物理日志,

可以通过手动保存 save 或者 bgsave, save 是阻塞的, bgsave 是 fork 出一个子进程来保存

在 redis.conf 配置文件中配置了自动保存的策略, 当其中一个策略触发了就会通过 bgsave 保存.

aof:记录是逻辑日志, 备份的是数据库的变更命令, 以 redis 协议格式来保存的. Redis 中提供了 3 中同步策略, 即每秒同步、每修改同步和不同步, 默认是每秒保存一次, 如果 aof 文件过大, 会 fork 出一个子进程, 根据 redis 的 key 重写命令

如果配置了 aof, 则 redis 启动时, 会以 aof 来加载. 否则以 rdb 来加载

两个相比较:rdb 文件更小, 安全性相对较差, aof 文件更大, 也更完整.

Redis 分布式方案

Master+Slave:读写分离的一种方案, 主节点负责数据写入, slave 负责同步, 可提供读取能力. 如果主节点发生故障, 需要手动切换到 Slave

sentinel 哨兵模式:master+slave 的高可用模式, 哨兵是一个单独的进程, 通过发送命令监控 redis 服务的运行状态, 一旦 master 宕机, 会自动将 slave 切换为 master, 并通过发布订阅的模式通知其他从服务器, 修改配置文件, 让他们切换到新的 master.

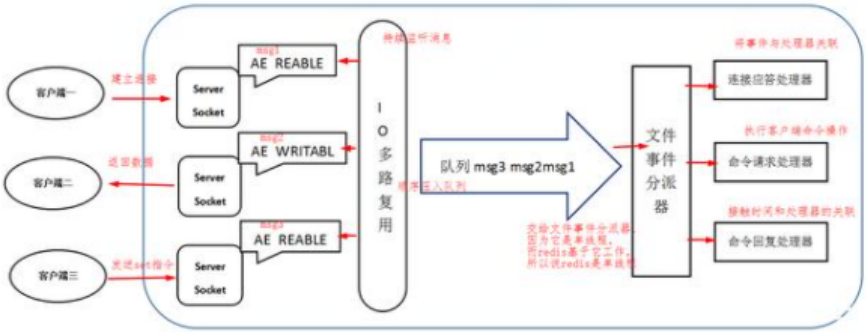
redis cluster:cluster 模式是预设虚拟槽, 一共分为 2^14 个虚拟槽, 通过对 key 进行 Hash 取 mod 计算对应的虚拟槽, 每个节点均分虚拟槽,

该集群模式是无中心化的, 节点之间相互通讯, 可获取槽所在的节点, 因此, 客户端可以请求任意节点, 如果 key 对应的 slot 不在该节点则通知客户端 key 所在的目标节点, 客户端重新请求目标节点.

该方案的问题, 由于节点之间需要相互通讯, 势必对带宽造成影响, 建议节点数量不要过多

twemproxy\codis 是基于中间代理的方式. 为 redis 节点创建代理层, 负责 key 的分发.

Redis 是单线程的吗？



> Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处理器。它的组成结构为4部分：多个套接字、IO多路复用程序、文件事件分派器、事件处理器。因为文件事件分派器队列的消费是单线程的，所以Redis才叫单线程模型。

Redis 数据结构及使用场景

	String	list	set	sortset	map
数据结构	value 为数字和字符串	双向链表	value 为空的 HashMap	HashMap+跳表	hashmap
使用场景	计数器\缓存 json 数据	队列或者列表	去重,取交集、并集等	排序,排名	缓存对象
常用命令	get、set、incr、decr、mget	lpush,rpush,lpop,rpop,lrange,BLPOP	sadd,srem,spop,sdiff ,smembers,sunion	zadd,zrange,zrem,zcard	hget、hset、hgetall

Redis 为什么可以用做分布式锁

1、Redis 为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对 Redis 的连接并不存在竞争关系。

2、Redis 的 SETNX 命令可以方便的实现分布式锁。

setNX（SET if Not eXists）

语法：SETNX key value

返回值：设置成功，返回 1 ；设置失败，返回 0

当且仅当 key 不存在时将 key 的值设为 value，并返回 1；若给定的 key 已经存在，则 SETNX 不做任何动作，并返回 0。

综上所述，可以通过 setnx 的返回值来判断是否获取到锁，并且不用担心并发访问的问题，因为 Redis 是单线程的，所以如果返回 1 则获取到锁，返回 0 则没获取到。当业务操作执行完后，一定要释放锁，释放锁的逻辑很简单，就是把之前设置的 key 删除掉即可，这样下次又可以通过 setnx 该 key 获取到锁了。

redis 快的原因

基于内存的.

数据结构简单

单线程避免了线程上下文切换

采用 IO 的多路复用技术,非阻塞 io 内部采用了 epoll,连接、读、写、关闭等都转化为事件.

为什么 redis 是单线程的?

redis 性能瓶颈主要是内存和网络带宽上.

单线程的效率已经能达到百万 tps,能够满足需要了.

Redis 过期策略

-|惰性删除:在进行 get 操作时,检查 key 是否已经过期

|定期删除:每隔一段时间执行删除过期 key 的操作,减少删除操作对 cpu 的占用.

惰性删除流程

在进行 get 或 setnx 等操作时，先检查 key 是否过期，

若过期，删除 key，然后执行相应操作；

若没过期，直接执行相应操作

定期删除流程（简单而言，对指定个数个库的每一个库随机删除小于等于指定个数个过期 key）

遍历每个数据库（就是 redis.conf 中配置的"database"数量，默认为 16）

检查当前库中的指定个数个 key（默认是每个库检查 20 个 key，注意相当于该循环执行 20 次，循环体时下边的描述）

如果当前库中没有一个 key 设置了过期时间，直接执行下一个库的遍历

随机获取一个设置了过期时间的 key，检查该 key 是否过期，如果过期，删除 key

判断定期删除操作是否已经达到指定时长，若已经达到，直接退出定期删除。

Redis 内存淘汰机制

volatile-lru:设置了过期时间的 Key 中,移除最近最少使用的.

volatile-ttl:设置了过期时间的 Key 中,移除更早过期的.

volatile-random:设置了过期时间的 key 中,随机移除

allkey-random:所有 key 随机删除.

allkey-lru:所有的 key 移除最近最少使用的.

noeviction:禁止驱除,申请内存会报错.

Redis 事务

Redis 是一组命令的集合,由 multi 和 exec 命令组成,multi 标记一个事务块的开始,在执行 exec 命令之前,先让入队列缓存,并不会执行,执行 exec 才会执行.

可以使用 discard 命令取消执行.

缓存一致性

缓存超时机制,最终一致性

延时双删+定时刷新

缓存穿透和缓存雪崩

缓存穿透:

如果 key 对应缓存不存在,每次请求不会从缓存获取,而是直接查询到数据源,从而压垮数据源.比如,相同的超时时间或者 key 在内存和数据源都不存在.

如果没有数据,也要将空的结果返回.布隆过滤器

缓存击穿:

如果缓存过期,大量的并发请求过来,大量的请求从数据库加载,压垮数据库.主要针对是同一个 key 的并发.

增加互斥锁

缓存雪崩:

服务器重启或者大量缓存在同一时间失效,会给数据库带来很大的压力.多个 key 的瞬时流量.

设置随机过期时间.

Redis 分布式锁的实现

加锁

加锁实际上就是在 redis 中，给 Key 键设置一个值，为避免死锁，并给定一个过期时间。

```
SET lock_key random_value NX PX 5000
```

值得注意的是：

random_value 是客户端生成的唯一的字符串。

NX 代表只在键不存在时，才对键进行设置操作。

PX 5000 设置键的过期时间为 5000 毫秒。

这样，如果上面的命令执行成功，则证明客户端获取到了锁。

解锁

解锁的过程就是将 Key 键删除。但也不能乱删，不能说客户端 1 的请求将客户端 2 的锁给删除掉。这时候 random_value 的作用就体现出来。

redis 锁要注意什么问题

1.需要设置过期时间，避免锁无法释放,如果设置了过期时间也有问题，有可能没释放锁，就过期了。

2.每个线程拿到锁，给 key 设置唯一标识，防止错误的自己的锁由其他线程释放，

比如：a 线程拿到锁，超时锁被释放，b 线程拿到锁，a 线程执行了 key 的删除操作。导致 b 拿到的锁也被释放

partition

Mysql 篇

事务的特性 ACID

原子性:事务执行的过程中,要么全部成功,要么全部失败

一致性:事务前后状态保持一致,比如 A 转账给 B,A 账户少 10 元,B 账户多 10 元,不能存在中间态

隔离性:多个事务并发执行的时候,事务之间是相互隔离的.

持久性:事务一旦提交,对数据的改变是永久的.

讲一下事务的隔离性

隔离性是事务的一个特征,多个事务执行,事务之间是隔离的.

- 那事务隔离存在哪些问题?

脏读:一个事务可以读取到另外一个事务未提交的内容.

不可重复读:同一个事务读取相同的数据,前后不一致

幻读:同一个事务使用相同的查询条件,莫名的多出或少了一部分数据.主要针对当前读.

- 事务的隔离级别

读取未提交:读取数据没有任何的锁,可以读取到未提交的数据,是最低的隔离级别.有脏读的问题.

读取已提交:事务只能读取到已经提交的数据,通过 MVCC 机制来实现的,每次读数据的时候,都会建立 ReadView 快照,通过该快照获取已经提交的数据.

可重复读:同一个事务前后读取的数据是一致的,同样也是通过 MVCC 机制来实现的,事务开启的时候建立 ReadView 快照.事务期间通过该 ReadView 获取快照数据.

串行化:事务读的时候加的共享锁,其他事务只能读不能写.

MVCC 的实现原理

MVCC 为多版本并发控制,是 InnoDB 提高并发性能和并发控制解决读写冲突的一种手段.能够在非阻塞的情况下实现并发读.

MVCC 主要是基于 Undolog 和 ReadView 实现的.

Undolog 每一条数据上有 DB_TRX_ID 和 DB_ROLL_PTR,分别代表的时候数据修改或新增的事务 ID 和回滚指针.

当对数据做修改时,先对数据加行锁,将修改后的数据的 DB_ROLL_PTR 指向历史数据的地址,形成链表结构,就是 undolog,该日志用于数据的回滚和快照读的实现.

ReadView 是在读取已提交和可重复读事务隔离级别下的快照读视图,该视图通过空间占用比较小的一个数据结构,支持了当前事务可见的数据.

ReadView 主要包含了三部分:

1.当前活跃的事务 Id 列表

2.当前活跃事务最小 ID

3.下一待分配的事务 ID

InnoDB 和 MyISAM 区别

两个都是 B+树,但是 InnoDB 是聚簇索引(叶子结点存数据),MyISAM 是非聚簇索引.(索引和数据单独存储)

InnoDB 支持行锁和事务,而 MyISAM 不支持

MyISAM 查询性能要好一点

B+树与 B 树的区别

B+树是 B-树的变体，也是一种多路搜索树，它与 B- 树的不同之处在于:

1.所有关键字存储在叶子节点出现,内部节点(非叶子节点并不存储真正的 data)

2.为所有叶子结点增加了一个链指针

b+树的中间节点不保存数据，所以磁盘页能容纳更多节点元素，更“矮胖”；

b+树查询必须查找到叶子节点，b 树只要匹配到即可不用管元素位置，因此 b+树查找更稳定（并不慢）；

对于范围查找来说，b+树只需遍历叶子节点链表即可，b 树却需要重复地中序遍历，如下两图：

有 k 个子树的中间节点包含有 k 个元素（B 树中是 k-1 个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。

所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素

索引设计的原则

索引尽量复用,不要重复创建索引

不要过度索引,索引过多会导致写性能下降

最左原则

索引使用的字段缩短长度,比如字符串,数字类型等.

千万大表的优化手段

1.优化现有 Mysql

- 软件调优

索引优化 explain 的索引使用情况 、覆盖索引、避免回表、合理设计索引等

增加统计表

增加缓存

读写分离

分库分表：水平拆分:范围分、日期、某个字段 hash 取模;垂直拆分

- 硬件调优:

硬盘、内存、CPU、带宽

2.换一种 100%兼容的 Mysql 数据库,云数据库等.

3.换 Nosql 存储或者大数据解决方案.

explain 的字段

id:选择标识符

sql 的执行顺序.id 相同从上到下执行,id 不同从下向上执行

select_type:select 的类型

SIMPLE\PRIMARY\SUBQUERY\UNION\UNION RESULT

table:查询的表

type:

ALL:全表扫描

const:主键和唯一索引扫描

ref:非唯一索引

eq_ref:关联使用唯一性索引

index:没有使用索引,但是使用了索引的特性,比如覆盖索引和根据索引排序等.

possiable_keys:可能用到的索引

key:实际用到的索引.

key_len:使用的索引长度

rows:扫描的行数

extra:

using index:使用了覆盖索引

using index condition:使用索引下推(like 减少回表次数)

using filesort:进行了排序

using where:服务端进行了条件筛选

using temporary:使用了临时表

分库分表需要考虑的问题

全局主键的问题:雪花算法,64 位也就是 8 字节的整数,由时间戳+工作机器 ID+序列号

事务的问题:分布式事务

join:各库存储小表或者服务层处理

分页、聚集等:各自处理然后统一聚集

数据迁移:平滑升级需要数据双写

为什么 ID 要求有序?

防止页分裂及页挪动

Mysql 怎么保证原子性的?

利用 Innodb 的 undo log。undo log 名为回滚日志，是实现原子性的关键，当事务回滚时能够撤销所有已经成功执行的 sql 语句，他需要记录你要回滚的相应日志信息。

例如:(1)当你 delete 一条数据的时候，就需要记录这条数据的信息，回滚的时候，insert 这条旧数据(2)当你 update 一条数据的时候，就需要记录之前的旧值，回滚的时候，根据旧值执行 update 操作(3)当年 insert 一条数据的时候，就需要这条记录的主键，回滚的时候，根据主键执行 delete 操 undo log 记录了这些回滚需要的信息，当事务执行失败或调用了 rollback，导致事务需要回滚，便可以利用 undo log 中的信息将数据回滚到修改之前的样子。

mysql 是如何保证持久性的？

是利用了 redo log。Mysql 是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再刷回磁盘上。如果此时突然宕机，内存中的数据就会丢失。redo log 包括两部分：一是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；二是磁盘上的重做日志文件(redo log file)，该部分日志是持久的。

innodb 通过 force log at commit 机制实现事务的持久性，即在事务提交的时候，必须先将该事务的所有事务日志写入到磁盘上的 redo log file 和 undo log file 中进行持久化。也就是说提交了两个日志文件。

架构篇

数据分区方案

范围分区:比如时间、业务字段（比如品牌 ID）、优点方便扩容,缺点容易造成数据热点

Hash 求余分区:数据均匀分布,但是扩容、缩容不方便

Hash 一致性分区:扩容、缩容只需要少部分数据迁移,数据分布均匀

虚拟槽分区:redis 分区策略,将数据分为 2^14 个槽,数据均匀分布,容易扩缩容、

一致性的理解

ACID:事务前后状态一致

CAP:某个时刻,各个节点的数据是一致的

BASE:最终一致性

微服务的服务调用一致性.

强一致、线性一致性、最终一致性

dubbo 和 springCloud 区别

dubbo 是基于 RPC(dubbo,java 原生 rmi,http,thrift),springCloud 是基于 RestApi 的

dubbo 注册中心是 zk,springCloud 是 eureka

Eureka 和 ZooKeeper 区别

ZooKeeper 有 Leader 和 Follower 角色,Eureka 各个节点平等

ZooKeeper 保证的是 CP,Eureka 保证的是 AP

ZooKeeper 在选举期间注册服务瘫痪,虽然服务最终会恢复,但是选举期间不可用的

Eureka 各个节点是平等关系,只要有一台 Eureka 就可以保证服务可用,而查询到的数据并不是最新的

BASE

BASE 理论是对 CAP 中一致性和可用性权衡的结果

BA:系统出现了不可预知的故障,业务基本可用,允许损失部分可用性,比如:响应时间的损失,功能上的损失.

S:软状态,允许数据存在中间态,但是不影响系统的使用,允许副本存在延时

E:最终数据是一致,允许实时不一致.

CAP

一致性:同一时刻数据用户看到的数据是一致

可用性:用户请求,系统必须给出响应

分区容错性:能够容忍网络故障,如果网络或者结点出现故障,仍能对外提供服务.

分布式事务&一致性保证

2PC、3PC、TCC、基于本地消息表 最大努力通知、基于消息队列的最终一致性

2PC-|协调者向所有参与者发起提交事务请求,参与者开启事务并向协调者汇报是否能处理

|协调者根据参与者的反馈决定通知所有参与者是提交还是会滚事务

-缺点:

-|同步阻塞

-|协调者单点故障

-|提交网络中断导致数据不一致

3PC-|协调者向所有参与者询问是否可以提交事务 CanCommit

|如果所有参与者都可以提交事务,则进行预提交.PrepareCommit

|如果存在参与者反馈无法提交,则中断,否则则执行提交 commit/rollback

-|解决的问题:-|如果在预提交节点超时则提交本地事务

|解决了协调者的单点故障问题

|减少了阻塞

TCC(基于事务补偿)-|基于业务层面的 2PC,对业务侵入性强,接口需要实现幂等,try-commit-cancel

try:校验并预留资源(相当于草稿)

commit:提交资源

cancel:删除预留的资源

最大努力通知机制:重试+定期查询

基于消息队列的最终一致性:基于本地消息表或者独立消息服务

限流算法

计数器算法

漏桶算法

令牌桶算法

微服务划分原则参考

按业务

性能

数据量

组织架构粒度

不能过细： 太细会出现事务问题引入了复杂度 引入更高的成本

通用的角度

发布频率

维护成本

zk 与 redis 分布式锁的区别

redis 基于内存操作并发能力更强

redis 阻塞锁需要轮询检测，消耗 CPU

redis 如果应用需要等待超时时间释放锁

zk 采用有序临时节点，如果高并发创建节点过多压力大

zk 采用监听比它小的节点，基于事件

如果客户端挂了，由于是临时节点能够立马释放锁

mybatis 篇

mybatis 的缓存实现

一级缓存是基于 sqlSession 的,是默认的缓存.内部存储 HashMap,用来缓存查询对象,当进行增删改 commit\close 等操作缓存失效.默认开启的

二级缓存是基于 Mapper 级别,多 sqlSession 共享的.缓存的对象需要序列化.缓存的是数据不是对象.

二级缓存->一级缓存->查询数据库

网络篇

TCP 三次握手

客户端发送 SYN 标记及发送 seq=x,建立链接

服务端发送 ACK 响应标记,确认号 ack=x+1,并发送 SYN 建立链接标记及 seq=y

客户端响应 ACK,并发送 ack=y+1,链接建立成功.

为什么要三次握手？

确认双方通讯是 OK 的

并同步双方会话的 seq

四次挥手

客户端发送请求释放链接 FIN 标记,并发送 seq=x

服务端响应 ACK,并发送 ack=x+1,通知客户端已收到释放链接的请求

服务端发送 FIN 请求关闭链接,并发送 seq=y

客户端响应 ACK,并发送 ack=y+1,链接关闭.

为什么是四次挥手？

服务端收到客户端关闭链接请求后,报文并未发送完毕.

select\epoll\poll 都是多路复用的 IO 技术

select 只知道有流事件,将消息传递从内核空间拷贝到用户空间,有最大连接数的限制,1000

poll 跟 select 类似,但是没有最大连接数的限制

而 epoll 知道具体是哪个流有事件发生.用户空间和内核空间共享内存.

HTTPS

客户端将支持的加密协议和版本发送给服务端

服务端将协商好的协议及版本还有证书,其中证书含公钥,将这些信息发送给客户端

客户端验证证书的合法性

客户端生成随机对称加密算法密钥,并用公钥加密

加密后的密钥发送给服务端

服务端用私钥解密

双发利用密钥进行对称加密通讯

cookie&session

Session 是另一种记录客户状态的机制，不同的是 Cookie 保存在客户端浏览器中，而 Session 保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是 Session。客户端浏览器再次访问时只需要从该 Session 中查找该客户的状态就可以了。

由于 HTTP 是一种无状态的协议，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是 Cookie 的工作原理。

http1.0/http1.1/http2.0

http1.0 每次请求都要进行三次握手

需要等待服务器的响应才能发送下次请求是阻塞的

http1.1

一个 TCP 链接可以传送多个 http 请求和响应，减少了 TCP 建立链接和关闭链接的消耗

客户端不用等待上一次 请求结果返回，就可以发出下一次请求

扩充了请求 / 响应头

http2.0

二进制分帧传输，

允许同时通过单一的 HTTP/2 连接发起多重的请求-响应消息

常见 http 请求码

- HTTP 400 错误 - 请求无效 (Bad request)
- 200 - 请求成功

- 301 - 资源（网页等）被永久转移到其它 URL
- 404 - 请求的资源（网页等）不存在
- 500 - 内部服务器错误

PUT 和 POST

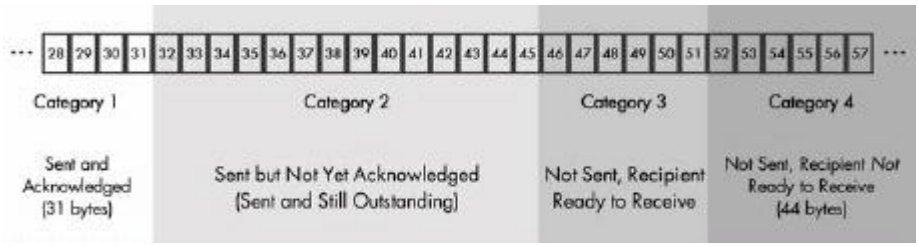
PUT 和 POS 都有更改指定 URI 的语义.但 PUT 被定义为 idempotent 的方法，POST 则不是.idempotent 的方法:如果一个方法重复执行

多次，产生的效果是一样的，那就是 idempotent 的。也就是说：

PUT 请求：如果两个请求相同，后一个请求会把第一个请求覆盖掉。（所以 PUT 用来改资源）

Post 请求：后一个请求不会把第一个请求覆盖掉。（所以 Post 用来增资源）

滑动窗口，流量控制



如图所示，发送方的窗口快照可以将窗口分为四个部分：

Category 1:已发送且已收到 ACK 确认的部分：1-31 字节

Category 2:已发送但未收到 ACK 确认的部分：32-45 字节

Category 3:未发送且在接收方范围内的部分：46-51 字节（这部分数据仍在对方系统处理能力之内）

Category 4:未发送但数据大小超过对方系统处理范围之外的部分：52 以后的字节

Spring 篇

Spring 中用到了哪些设计

工厂:根据 bean 的名称获取

动态代理:aop

观察者:applicationListener

模版:jdbctemplate

单例:scopere

spring 的事务管理

- 事务的隔离级别

读取未提交

读取已提交

可重复读

串行化

- 事务的传播机制

PROPAGATION_REQUIRED：如果有事务,那么加入该事务,没有则创建一个事务

PROPAGATION_REQUIRES_NEW:不管存不存在都创建一个新的事务

PROPAGATION_SUPPORTS：A 如果有事务，B 将使用该事务；如果 A 没有事务，B 将以非事务执行

PROPAGATION_NOT_SUPPORTS：容器不为这个方法开启事务

PROPAGATION_MANDATORY：必须在一个已有的事务中执行,否则抛出异常

PROPAGATION_NEVER：必须在一个没有的事务中执行,否则抛出异常

- 只读事务
- rollbackFor 机制

SpringMVC

1、 用户发送请求至前端控制器 DispatcherServlet。

2、 DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3、 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4、 DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5、 HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller 执行完成返回 ModelAndView。

7、 HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8、 DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9、 ViewReslover 解析后返回具体 View。

10、DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet 响应用户。

SpringIOC 的理解

控制反转,以前对象都是自己创建把控,现在交给 spring 容器来管理.Spring 容器通过配置文件、注解以及 Configuration 等来创建和管理容器的依赖关系.也有利于对象的复用和生命周期的管理.

SpringAOP

利用 java 动态代理或者 cglib,将公共行为封装成可重用模块,减少系统的重复代码,降低模块间的耦合度.比如日志记录、事务、监控、权限控制等等.

Spring Bean 的生命周期

实例化 Bean

为 Bean 注入属性

调用 BeanPostProcesser 的 postProcessBeforeInitialization

调用 InitializingBean 的 afterPropertySet 方法

调用 bean 的 init-method

调用 BeanPostProcesser 的 postProcessAfterInitialization

destory-method

SpringBoot 的理解

简化了配置和开发,大大提高了工作效率

内嵌了 Web 容器,比如 tomcat 和 jetty,独立运行

以 starter 的形式引入依赖包,简化了包依赖

不需要一系列的配置文件

增加了健康检查等各项指标信息

约定优于配置

微服务的认识以及存在的问题

微服务是一种架构风格,一个大型复杂软件应用由一个或者多个微服务组成.服务是高内聚,服务之间是低耦合的.每个服务只关心自己的任务.

优点:

每个服务职责单一、清晰

服务之间松耦合的

易于修改和维护

服务可以以不同的语言开发.

引入关注点:

服务的治理-|配置中心(Spring cloud config、zookeeper)

|服务之间的数据一致性(事务一致性保证\幂等)

|日志的规范(ELK)

|链路追踪(pinpoint)

|服务的监控(zabbix\grafa)

|服务的熔断、降级、限流(Hystrix)

|运维部署管理(docker)

|服务的注册发现、负载均衡(euraka\ribbon)

|服务的安全(api 网关)

eureka 的自我保护机制

客户端与服务端一般 30 秒钟进行一次心跳,如果 90 秒钟服务端收不到心跳,则将当前实例移除.

如果 15 分钟内,发现 85%的节点都失去心跳,则进入自我保护状态,不移除实例.

可能原因是:eureka 本身有问题或者网络出现问题

自我保护机制是对网络波动的安全保证措施.

Kafka 篇

Kafka 快的原因

1.Kafka 基于磁盘顺序写的.机械硬盘每次读写都会先寻址是很耗时的.Kafka 每个分区都是一个文件,每次写入的时候把数据追加到文件尾部.这种设计有一种缺陷,就是没办法删除数据.

2.Kafka 并没有调用 write 直接写磁盘,而是通过内存映射文件的方式,它是利用操作系统的页实现文件到物理内存的直接映射.完成映射之后,对物理内存的操作会同步到磁盘.producer.type 来配置是同步还是异步刷新内存.

3.分段快速检索

4.实现零拷贝,从内核空间到网卡直接拷贝不需要经过用户空间.

5.分区支持多消费者同时读取

Kafka 可靠性和一致性

1.分区的副本同步机制,从分区的所有副本中,选取一个作为 master,其他的作为 followers,leader 提供读写,follower 从 leader 拉取数据进行同步,当 Leader 挂了,从 ISR 列表中选取一个作为 leader

2.request.required.acks 有三个值

0:不需要 server 响应

1:leader 成功后,作出响应

-1:所有的 ISR 同步成功后,作出响应

可以将 min.insync.replicas:配置为 2 在吞吐量和数据可靠性之间做一个衡量

3.leader 新写入的消息,并不是立马就能读取到,Leader 会等 ISR 中的所有副本同步后,更新到高水位才会被 consumer 消费到,以免 leader 挂了之后,导致数据不一致.