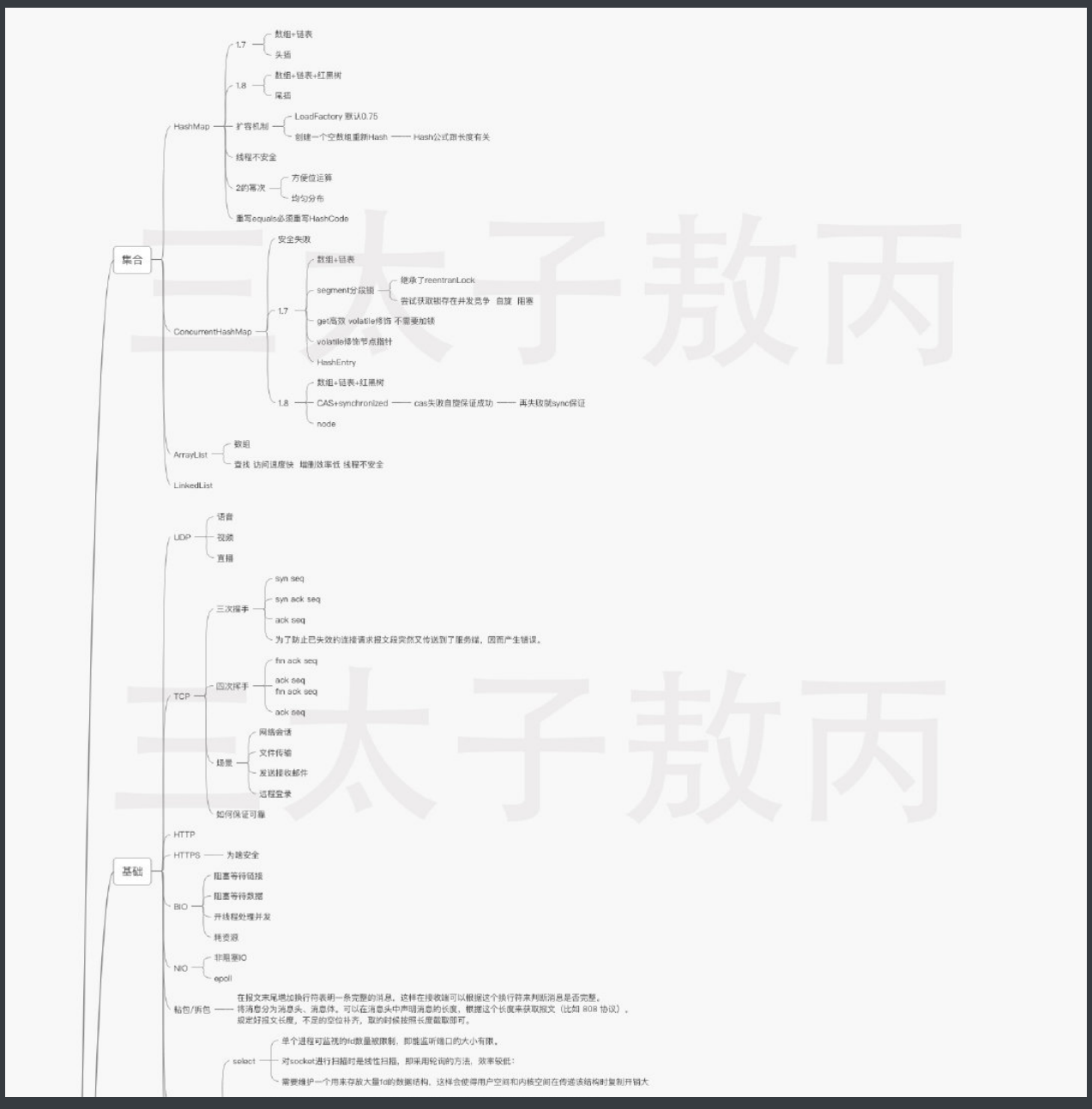
> 点赞再看，养成习惯，微信搜索【**三太子敖丙**】关注这个互联网苟且偷生的工具人。
>
> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

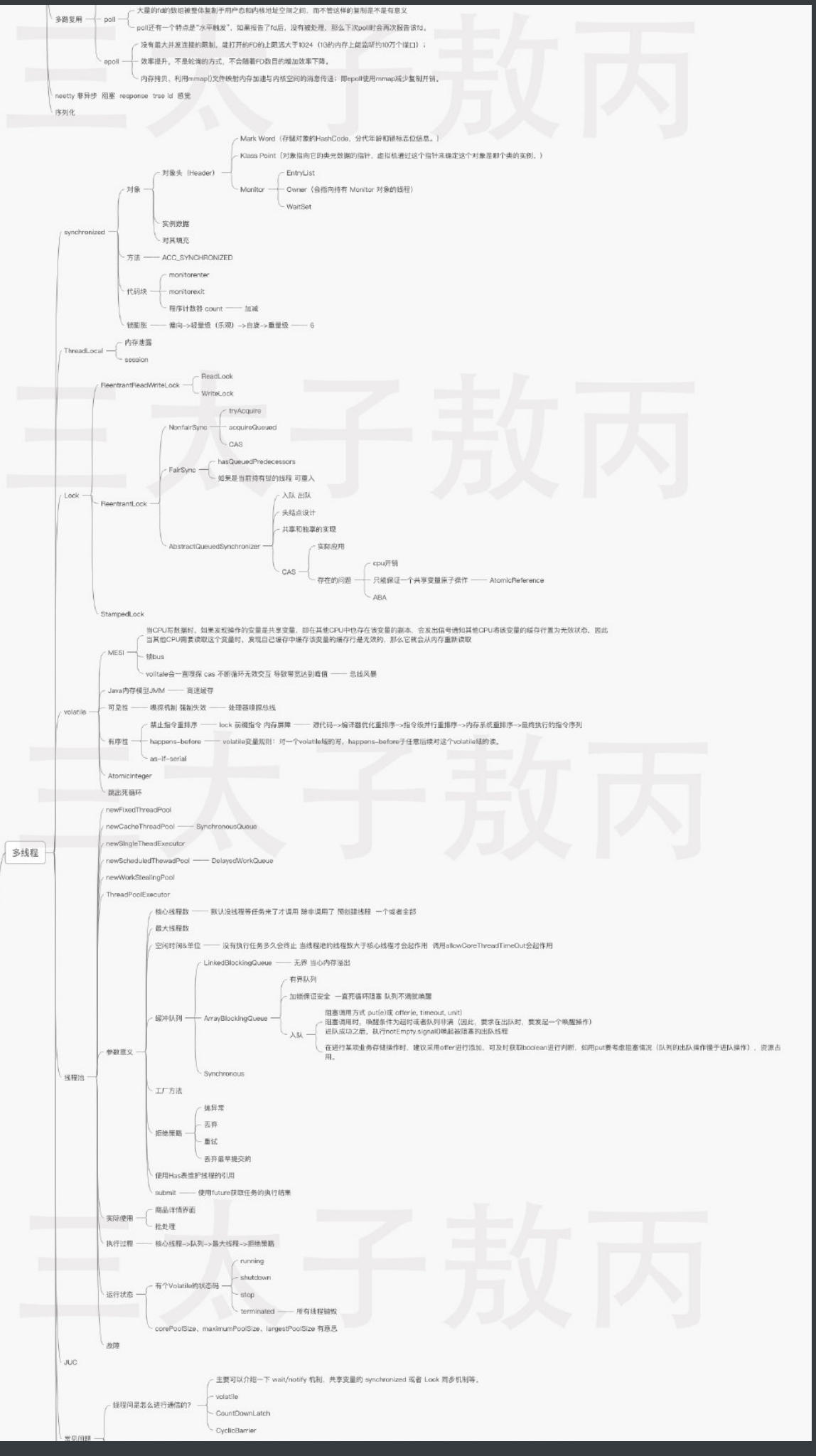
前言

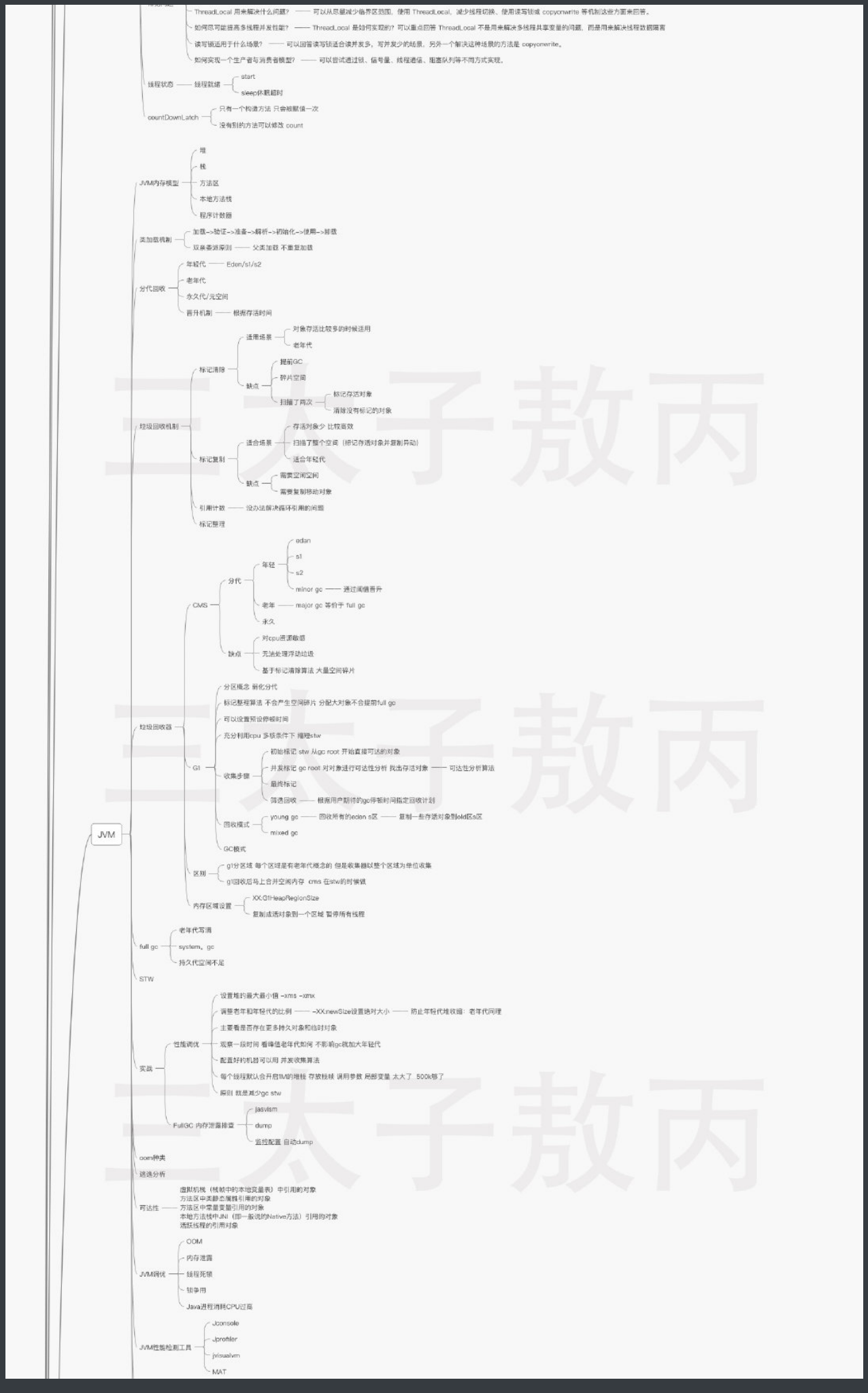
前段时间敖丙不是在复习嘛，很多小伙伴也想要我的复习路线，以及我自己笔记里面的一些知识点，好了，丙丙花了一个月的时间，整整一个月啊，给大家整理出来了。

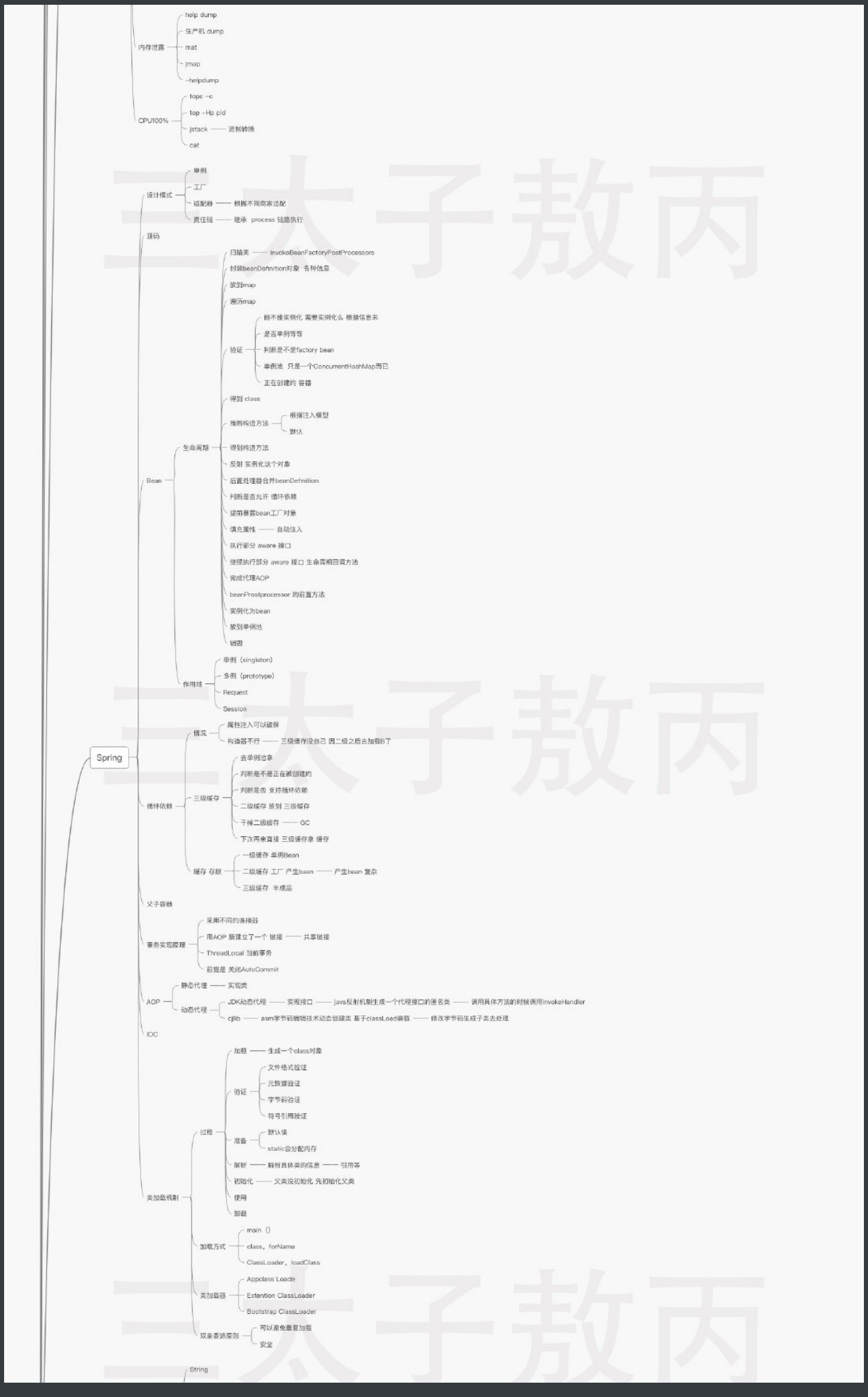
一上来我就放个大招好吧，我的复习脑图，可以说是全得不行，为了防止被盗图，我加了水印哈。

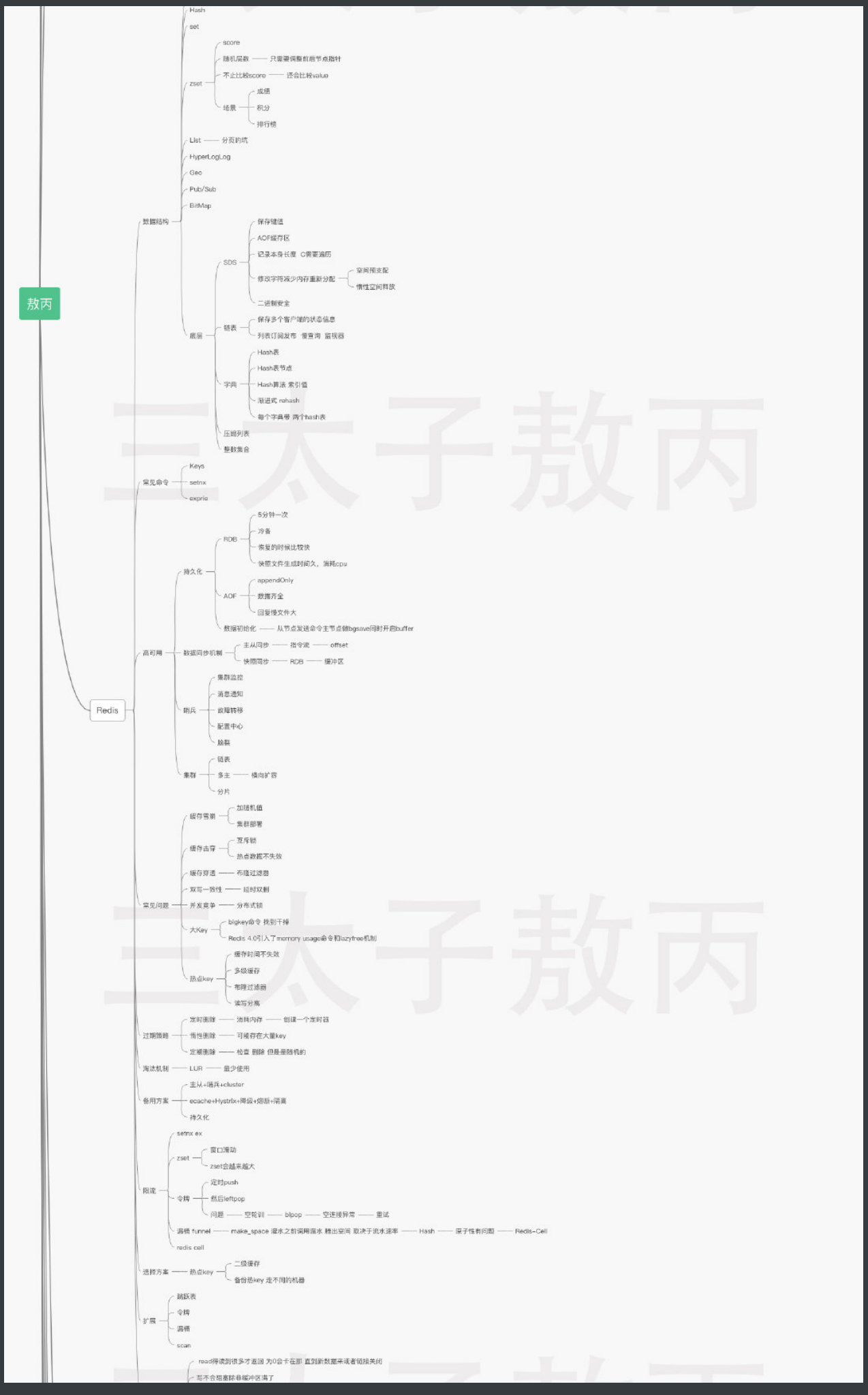
这期看下去你会发现很硬核，而且我会持续更新，啥也不说了，看在我熬夜一个月满脸痘痘的份上，你可以点赞了哈哈。

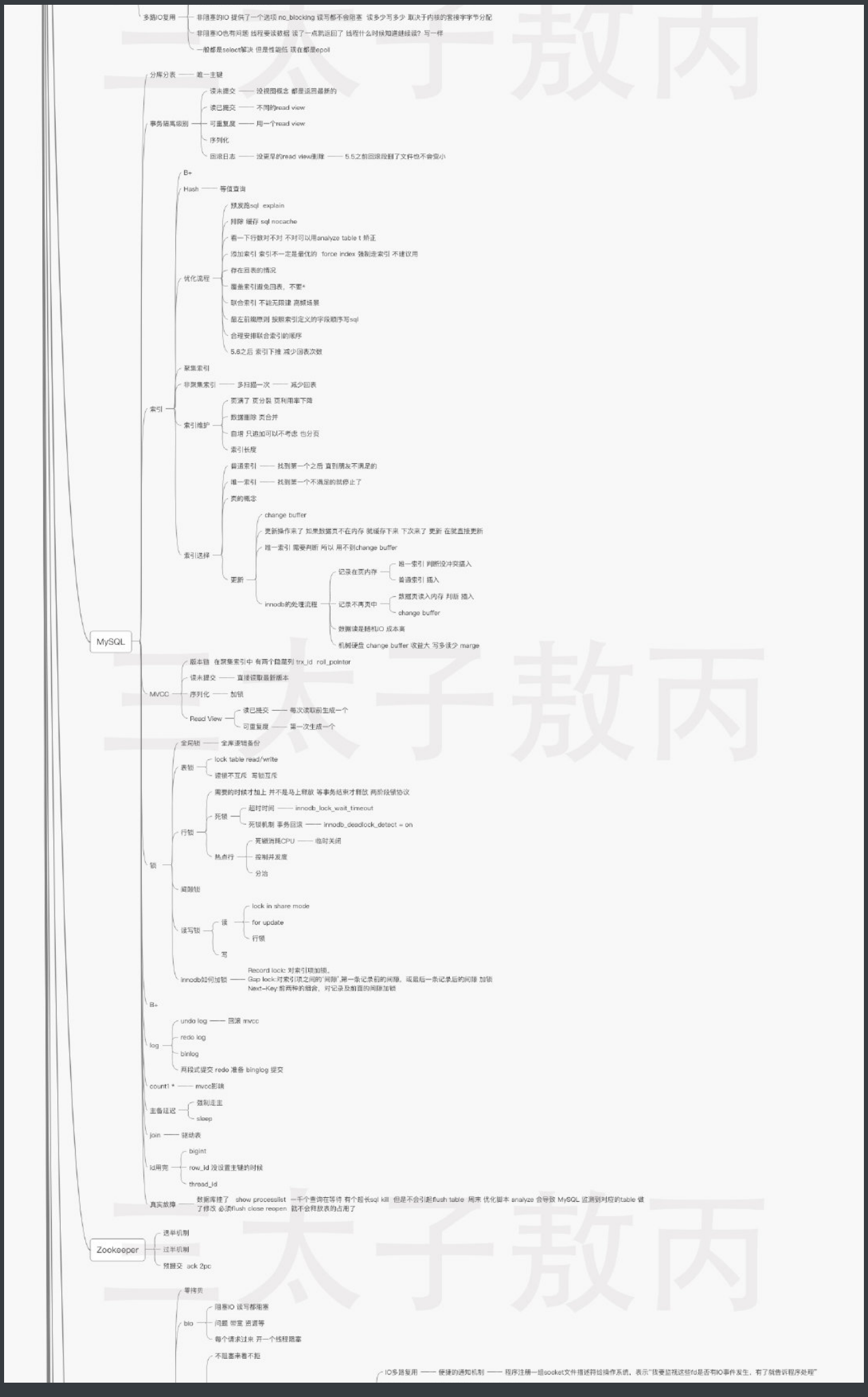


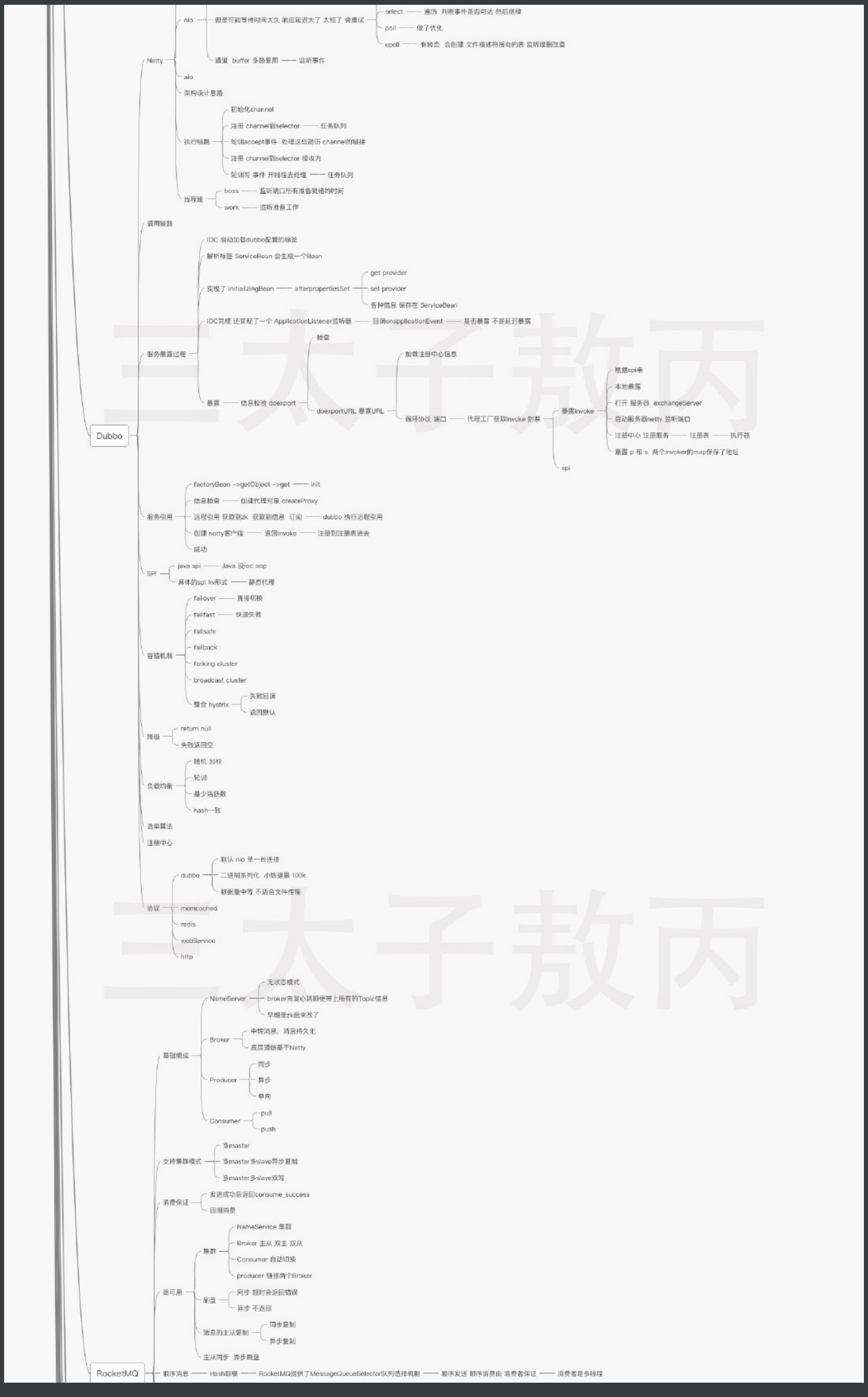

多路复用
poll
大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义
poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。
epoll
没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）；
效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。
内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。
neetty 非异步 阻塞 response trse id 感觉
序列化
synchronized
对象
对象头（Header）
Mark Word（存储对象的HashCode，分代年龄和锁标志位信息。）
Klass Point（对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例。）
Monitor
EntryList
Owner（会指向持有 Monitor 对象的线程）
WaitSet
实例数据
对其填充
方法
ACC_SYNCHRONIZED
代码块
monitorenter
monitorexit
程序计数器 count
加减
锁膨胀
难向->轻量级（乐观）->自旋->重量级
6
ThreadLocal
内存速露
session
Lock
ReentrantReadWriteLock
ReadLock
WriteLock
ReentrantLock
NonfairSync
tryAcquire
acquireQueued
CAS
FairSync
hasQueuedPredecessors
如果是当前持有锁的线程 可重入
AbstractQueuedSynchronizer
入队 出队
头结点设计
共享和独享的实现
CAS
实际应用
存在的问题
cpu开销
只能保证一个共享变量原子操作
AtomicReference
ABA
StampedLock
volatile
MESI
当CPU写数据时，如果发现操作的变量是共享变量，即在其他CPU中也存在该变量的副本，会发出信号通知其他CPU将该变量的缓存行置为无效状态，因此
当其他CPU需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取
锁bus
volitale会一直嗅探 cas 不断循环无效交互 导致带宽达到峰值
总线风暴
Java内存模型JMM
高速缓存
可见性
嗅探机制 强制失效
处理器嗅探总线
有序性
禁止指令重排序
lock 前缀指令 内存屏障
硬件级并行重排序->内存系统重排序->最终执行的指令序列
happens-before
volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。
as-if-serial
AtomicInteger
跳出死循环
多线程
线程池
newFixedThreadPool
newCacheThreadPool
SynchronousQueue
newSingleTheadExecutor
newScheduledThewadPool
DelayedWorkQueue
newWorkStealingPool
ThreadPoolExecutor
参数意义
核心线程数
默认没线程等任务来了才调用 除非调用了 预创建线程 一个或者全部
最大线程数
空闲时间&单位
没有执行任务多久会终止 当线程池的线程数大于核心线程才会起作用 调用allowCoreThreadTimeOut会起作用
缓冲队列
LinkedBlockingQueue
无界 当心内存溢出
ArrayBlockingQueue
有界队列
加锁保证安全 一直死循环阻塞 队列不满就唤醒
入队
阻塞调用方式 put(e)或 offer(e, timeout, unit)
阻塞调用时，唤醒条件为超时或者队列非满（因此，要求在出队时，要发起一个唤醒操作）
进队成功之后，执行notEmpty.signal()唤起被阻塞的出队线程
在进行某项业务存储操作时，建议采用offer进行添加，可及时获取boolean进行判断，如用put要考虑阻塞情况（队列的出队操作慢于进队操作），资源占用。
Synchronous
工厂方法
拒绝策略
抛异常
丢弃
重试
丢弃最早提交的
使用Has表维护线程的引用
submit
使用future获取任务的执行结果
实际使用
商品详情界面
批处理
执行过程
核心线程->队列->最大线程->拒绝策略
运行状态
有个Volatile的状态吗
running
shutdown
stop
terminated
所有线程销毁
corePoolSize、maximumPoolSize、largestPoolSize 有意思
故障
JUC
常见问题
线程间是怎么进行通信的?
主要可以介绍一下 wait/notify 机制，共享变量的 synchronized 或者 Lock 同步机制等。
volatile
CountDownLatch
CyclicBarrier

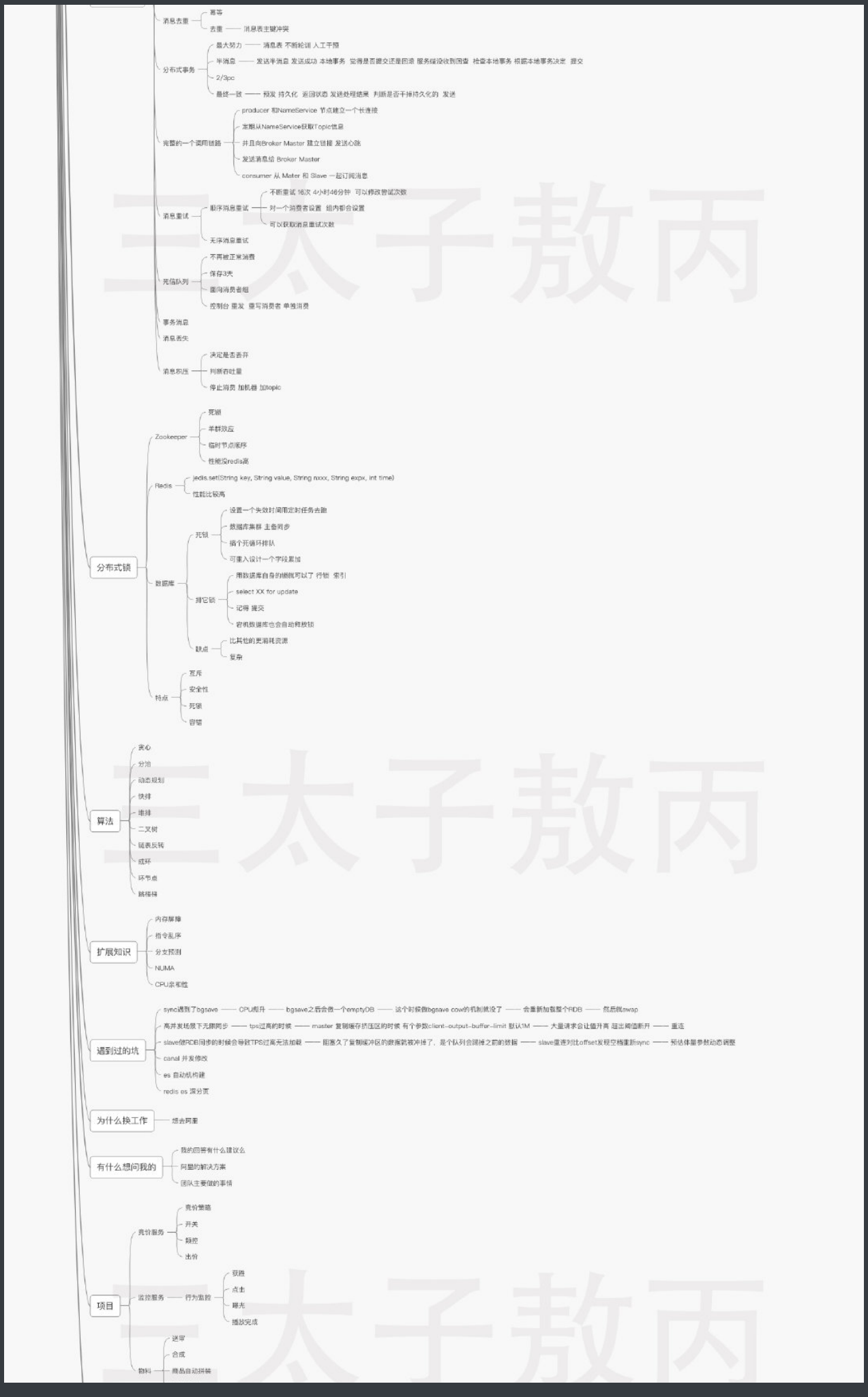

ThreadLocal 用来解决什么问题？ —— 可以从尽量减少临界区范围，使用 ThreadLocal，减少线程切换，使用读写锁或 copyonwrite 等机制这些方面来回答。
如何尽可能提高多线程并发性能？ —— ThreadLocal 最如何实现的？可以重点回答 ThreadLocal 不是用来解决多线程共享变量的问题，而是用来解决线程数据隔离
读写锁适用于什么场景？ —— 可以回答读写锁适合读并发多，写并发少的场景，另外一个解决这种场景的方法是 copyonwrite。
如何实现一个生产者与消费者模型？ —— 可以尝试通过锁、信号量、线程通信、阻塞队列等不同方式实现。
线程状态
线程就绪
start
sleep休眠超时
countDownLatch
只有一个构造方法 只会被赋值一次
没有别的方法可以修改 count
JVM内存模型
堆
栈
方法区
本地方法栈
程序计数器
类加载机制
加载->验证->准备->解析->初始化->使用->卸载
双亲委派原则 —— 父类加载 不重复加载
分代回收
年轻代 —— Eden/s1/s2
老年代
永久代/元空间
晋升机制 —— 根据存活时间
垃圾回收机制
标记清除
适用场景
对象存活比较多的时候适用
老年代
缺点
提前GC
碎片空间
扫描了两次
标记存活对象
清除没有标记的对象
标记复制
适合场景
存活对象少 比较高效
扫描了整个空间（标记存活对象并复制异动）
适合年轻代
缺点
需要空闲空间
需要复制移动对象
引用计数 —— 没办法解决循环引用的问题
标记整理
垃圾回收器
CMS
分代
年轻
eden
s1
s2
minor gc —— 通过阈值晋升
老年 —— major gc 等价于 full gc
永久
缺点
对cpu资源敏感
无法处理浮动垃圾
基于标记清除算法 大量空间碎片
G1
分区概念 弱化分代
标记整理算法 不会产生空间碎片 分配大对象不会提前full gc
可以设置预设停顿时间
充分利用cpu 多核条件下 缩短stw
收集步骤
初始标记 stw 从gc root 开始直接可达的对象
并发标记 gc root 对对象进行可达性分析 找出存活对象 —— 可达性分析算法
最终标记
筛选回收 —— 根据用户期待的gc停顿时间指定回收计划
回收模式
young gc —— 回收所有的eden s区 —— 复制一些存活对象到old区s区
mixed gc
GC模式
区别
g1分区域 每个区域是有老年代概念的 但是收集器以整个区域为单位收集
g1回收后马上合并空闲内存 cms 在stw的时候做
内存区域设置
XX:G1HeapRegionSize
复制成活对象到一个区域 暂停所有线程
JVM
full gc
老年代写满
system。gc
持久代空间不足
STW
实战
性能调优
设置堆的最大最小值 -xms -xmx
调整老年和年轻代的比例 —— -XX:newSize设置绝对大小 —— 防止年轻代堆收缩：老年代同理
主要看是否存在更多持久对象和临时对象
观察一段时间 看峰值老年代如何 不影响gc就加大年轻代
配置好的机器可以用 并发收集算法
每个线程默认会开启1M的堆栈 存放栈帧 调用参数 局部变量 太大了 500k够了
原则 就是减少gc stw
FullGC 内存泄露排查
jasvism
dump
监控配置 自动dump
oom种类
逃逸分析
可达性
虚拟机栈（栈帧中的本地变量表）中引用的对象
方法区中类静态属性引用的对象
方法区中常量变量引用的对象
本地方法栈中JNI（即一般说的Native方法）引用的对象
活跃线程的引用对象
JVM调优
OOM
内存泄露
线程死锁
锁争用
Java进程消耗CPU过高
JVM性能检测工具
Jconsole
Jprofiler
jvisualvm
MAT

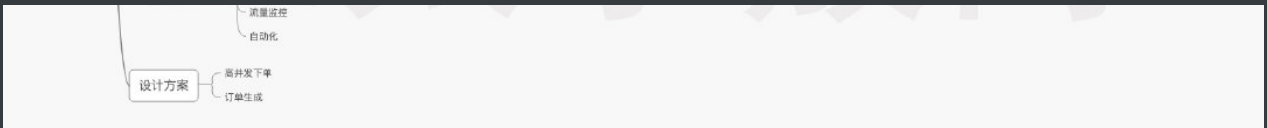

help dump
生产机 dump
内存泄露
mat
jmap
-helpdump
CPU100%
topc -c
top -Hp pid
jstack —— 进制转换
cat
设计模式
单例
工厂
适配器 —— 根据不同商家适配
责任链 —— 继承 process 链路执行
源码
Bean
生命周期
扫描类 —— invokeBeanFactoryPostProcessors
封装beanDefinition对象 各种信息
放到map
遍历map
验证
能不能实例化 需要实例化么 根据信息来
是否单例等等
判断是不是factory bean
单例池 只是一个ConcurrentHashMap而已
正在创建的 容器
得到 class
推断构造方法
根据注入模型
默认
得到构造方法
反射 实例化这个对象
后置处理器合并beanDefinition
判断是否允许 循环依赖
提前暴露bean工厂对象
填充属性 —— 自动注入
执行部分 aware 接口
继续执行部分 aware 接口 生命周期回调方法
完成代理AOP
beanPostprocessor 的前置方法
实例化为bean
放到单例池
销毁
作用域
单例（singleton）
多例（prototype）
Request
Session
Spring
循环依赖
情况
属性注入可以破解
构造器不行 —— 三级缓存没自己 因二级之后去加载B了
三级缓存
去单例池拿
判断是不是正在被创建的
判断是否 支持循环依赖
二级缓存 放到 三级缓存
干掉二级缓存 —— GC
下次再来直接 三级缓存拿 缓存
缓存 存放
一级缓存 单例Bean
二级缓存 工厂 产生baen —— 产生bean 复杂
三级缓存 半成品
父子容器
事务实现原理
采用不同的连接器
用AOP 新建立了一个 链接 —— 共享链接
ThreadLocal 当前事务
前提是 关闭AutoCommit
AOP
静态代理 —— 实现类
动态代理
JDK动态代理 —— 实现接口 —— java反射机制生成一个代理接口的匿名类 —— 调用具体方法的时候调用InvokeHandler
cjlib —— asm字节码编辑技术动态创建类 基于classLoad装载 —— 修改字节码生成子类去处理
IOC
类加载机制
过程
加载 —— 生成一个class对象
验证
文件格式验证
元数据验证
字节码验证
符号引用验证
准备
默认值
static会分配内存
解析 —— 解析具体类的信息 —— 引用等
初始化 —— 父类没初始化 先初始化父类
使用
卸载
加载方式
main ()
class。forName
ClassLoader。loadClass
类加载器
Appclass Loade
Extention ClassLoader
Bootstrap ClassLoader
双亲委派原则
可以避免重复加载
安全
String


敖丙
Redis
数据结构
Hash
set
zset
score
随机层数
只需要调整前后节点指针
不止比较score
还会比较value
场景
成绩
积分
排行榜
List
分页的坑
HyperLogLog
Geo
Pub/Sub
BitMap
底层
SDS
保存键值
AOF缓存区
记录本身长度 C需要遍历
修改字符减少内存重新分配
空间预支配
惰性空间释放
二进制安全
链表
保存多个客户端的状态信息
列表订阅发布 慢查询 监视器
字典
Hash表
Hash表节点
Hash算法 索引值
渐进式 rahash
每个字典带 两个hash表
压缩列表
整数集合
常见命令
Keys
setnx
exprie
高可用
持久化
RDB
5分钟一次
冷备
恢复的时候比较快
快照文件生成时间久，消耗cpu
AOF
appendOnly
数据齐全
回复慢文件大
数据初始化
从节点发送命令主节点做bgsave同时开启buffer
数据同步机制
主从同步
指令流
offset
快照同步
RDB
缓冲区
哨兵
集群监控
消息通知
故障转移
配置中心
脑裂
集群
链表
多主
横向扩容
分片
常见问题
缓存雪崩
加随机值
集群部署
缓存击穿
互斥锁
热点数据不失效
缓存穿透
布隆过滤器
双写一致性
延时双删
并发竞争
分布式锁
大Key
bigkey命令 找到干掉
Redis 4.0引入了memory usage命令和lazyfree机制
热点key
缓存时间不失效
多级缓存
布隆过滤器
读写分离
过期策略
定时删除
消耗内存
创建一个定时器
惰性删除
可能存在大量key
定期删除
检查 删除 但是是随机的
淘汰机制
LUR
最少使用
备用方案
主从+哨兵+cluster
ecache+Hystrix+降级+熔断+隔离
持久化
限流
setnx ex
zset
窗口滑动
zset会越来越大
令牌
定时push
然后leftpop
问题
空轮训
blpop
空连接异常
重试
漏桶 funnel
make_space 漏水之前调用漏水 腾出空间 取决于流水速率
Hash
原子性有问题
Redis-Cell
redis cell
选择方案
热点key
二级缓存
备份热key 走不同的机器
扩展
跳跃表
令牌
漏桶
scan
read得读到很多才返回 为0会卡在那 直到新数据来或者链接关闭
写不会阻塞除非缓冲区满了


多路IO复用
非阻塞的IO 提供了一个选项 no_blocking 读写都不会阻塞 读多少写多少 取决于内核的套接字字节分配
非阻塞IO也有问题 线程要读数据 读了一点就返回了 线程什么时候知道继续读? 写一样
一般都是select解决 但是性能低 现在都是epoll
MySQL
分库分表
唯一主键
事务隔离级别
读未提交
没视图概念 都是返回最新的
读已提交
不同的read view
可重复度
用一个read view
序列化
回滚日志
没更早的read view删除
5.5之前回滚段删了文件也不会变小
索引
B+
Hash
等值查询
优化流程
预发跑sql explain
排除 缓存 sql nocache
看一下行数对不对 不对可以用analyze table t 矫正
添加索引 索引不一定是最优的 force index 强制走索引 不建议用
存在回表的情况
覆盖索引避免回表，不要*
联合索引 不能无限建 高频场景
最左前缀原则 按照索引定义的字段顺序写sql
合理安排联合索引的顺序
5.6之后 索引下推 减少回表次数
聚集索引
非聚集索引
多扫描一次
减少回表
索引维护
页满了 页分裂 页利用率下降
数据删除 页合并
自增 只追加可以不考虑 也分页
索引长度
索引选择
普通索引
找到第一个之后 直到朋友不满足的
唯一索引
找到第一个不满足的就停止了
页的概念
更新
change buffer
更新操作来了 如果数据页不在内存 就缓存下来 下次来了 更新 在就直接更新
唯一索引 需要判断 所以 用不到change buffer
innodb的处理流程
记录在页内存
唯一索引 判断没冲突插入
普通索引 插入
记录不再页中
数据页读入内存 判断 插入
change buffer
数据读是随机IO 成本高
机械硬盘 change buffer 收益大 写多读少 marge
MVCC
版本链 在聚集索引中 有两个隐藏列 trx_id roll_pointer
读未提交
直接读取最新版本
序列化
加锁
Read View
读已提交
每次读取前生成一个
可重复度
第一次生成一个
锁
全局锁
全库逻辑备份
表锁
lock table read/write
读锁不互斥 写锁互斥
行锁
需要的时候才加上 并不是马上释放 等事务结束才释放 两阶段锁协议
死锁
超时时间
innodb_lock_wait_timeout
死锁机制 事务回滚
innodb_deadlock_detect = on
热点行
死锁消耗CPU
临时关闭
控制并发度
分治
间隙锁
读写锁
读
lock in share mode
for update
行锁
写
innodb如何加锁
Record lock: 对索引项加锁。
Gap lock:对索引项之间的"间隙",第一条记录前的间隙，或最后一条记录后的间隙 加锁
Next-Key:前两种的组合，对记录及前面的间隙加锁
B+
log
undo log
回滚 mvcc
redo log
binlog
两段式提交 redo 准备 binlog 提交
count(*)
mvcc影响
主备延迟
强制走主
sleep
join
驱动表
id用完
bigint
row_id 没设置主键的时候
thread_id
真实故障
数据库挂了 show processlist 一千个查询在等待 有个超长sql kill 但是不会引起flush table 周末 优化脚本 analyze 会导致 MySQL 监测到对应的table 做了修改 必须flush close reopen 就不会释放表的占用了
Zookeeper
选举机制
过半机制
预提交 ack 2pc
零拷贝
bio
阻塞IO 读写都阻塞
问题 带宽 资源等
每个请求过来 开一个线程阻塞
不阻塞来看不拒
IO多路复用
便捷的通知机制
程序注册一组socket文件描述符给操作系统，表示"我要监视这些fd是否有IO事件发生，有了就告诉程序处理"


Netty
nio
但是可能等待时间太久 响应延迟大了 太短了 会重试
select —— 遍历 判断事件是否可达 然后继续
poll —— 做了优化
epoll —— 有转态 会创建 文件描述符指向的表 监听增删改查
通道 buffer 多路复用 —— 监听事件
aio
架构设计思路
执行链路
初始化channel
注册 channel到selector —— 任务队列
轮训accept事件 处理这些循环 channel的链接
注册 channel到selector 接收方
轮训写 事件 并线程去处理 —— 任务队列
线程组
boss —— 监听端口所有准备就绪的时间
work —— 监听准备工作
Dubbo
调用链路
服务暴露过程
IOC 启动加载dubbo配置的标签
解析标签 ServiceBean 会生成一个Bean
实现了 initializingBean —— afterpropertiesSet
get provider
set provider
各种信息 保存在 ServiceBean
IOC完成 还实现了一个 ApplicationListener监听器 —— 回调onapplicationEvent —— 是否暴露 不是延迟暴露
暴露 —— 信息校验 doexport
检查
doexportURL 暴露URL
加载注册中心信息
循环协议 端口 —— 代理工厂 获取Invoke 封装
暴露Invoke
根据spi来
本地暴露
打开 服务器 exchangeServer
启动服务器netty 监听端口
注册中心 注册服务 —— 注册表 —— 执行器
暴露 p 和 s 两个invoker的map保存了地址
spi
服务引用
factoryBean ->getObject ->get —— init
信息检查 —— 创建代理对象 createProxy
远程引用 获取到zk 获取到信息 订阅 —— dubbo 执行远程引用
创建 netty客户端 —— 返回Invoke —— 注册到注册表进去
成功
SPI
java spi —— Java 没ioc aop
具体的spi kv形式 —— 静态代理
容错机制
failover —— 直接切换
failfast —— 快速失败
failsafe
failback
forking cluster
broadcast cluster
整合 hystrix
失败回调
返回默认
降级
return null
失败返回空
负载均衡
随机 加权
轮训
最少活跃数
hash一致
选举算法
注册中心
协议
dubbo
默认 nio 单一长连接
二进制系列化 小数据量 100k
数据量中等 不适合文件传输
memcached
redis
webService
http
RocketMQ
基础组成
NameServer
无状态模式
broker向发心跳顺便带上所有的Topic信息
早期是zk后来没了
Broker
中转消息，消息持久化
底层通信基于Netty
Producer
同步
异步
单向
Consumer
pull
push
支持集群模式
多master
多master多slave异步复制
多master多slave双写
消费保证
发送成功后返回consume_success
回调消费
高可用
集群
NameService 集群
Broker 主从 双主 双从
Consumer 自动切换
producer 链接两个Broker
刷盘
同步 超时会返回错误
异步 不返回
消息的主从复制
同步复制
异步复制
主从同步 异步刷盘
顺序消息 —— Hash取模 —— RocketMQ提供了MessageQueueSelector队列选择机制 —— 顺序发送 顺序消费由 消费者保证 —— 消费者是多线程


消息去重
幂等
去重
消息表主键冲突
分布式事务
最大努力
消息表 不断轮训 人工干预
半消息
发送半消息 发送成功 本地事务 监得是否提交还是回滚 服务端没收到回查 检查本地事务 根据本地事务决定 提交
2/3pc
最终一致
预发 持久化 返回状态 发送处理结果 判断是否干掉持久化的 发送
完整的一个调用链路
producer 和NameService 节点建立一个长连接
定期从NameService获取Topic信息
并且向Broker Master 建立链接 发送心跳
发送消息给 Broker Master
consumer 从 Mater 和 Slave 一起订阅消息
消息重试
顺序消息重试
不断重试 16次 4小时46分钟 可以修改重试次数
对一个消费者设置 组内都会设置
可以获取消息重试次数
无序消息重试
死信队列
不再被正常消费
保存3天
重向消费者组
控制台 重发 重写消费者 单独消费
事务消息
消息丢失
消息积压
决定是否丢弃
判断吞吐量
停止消费 加机器 加topic
分布式锁
Zookeeper
死锁
羊群效应
临时节点顺序
性能没redis高
Redis
jedis.set(String key, String value, String nxxx, String expx, int time)
性能比较高
数据库
死锁
设置一个失效时间定时任务去跑
数据库集群 主备同步
搞个死循环排队
可重入设计一个字段累加
排它锁
用数据库自身的锁就可以了 行锁 索引
select XX for update
记得 提交
宕机数据库也会自动释放锁
缺点
比其他的更消耗资源
复杂
特点
互斥
安全性
死锁
容错
算法
贪心
分治
动态规划
快排
堆排
二叉树
链表反转
成环
环节点
跳楼梯
扩展知识
内存屏障
指令乱序
分支预测
NUMA
CPU亲和性
遇到过的坑
sync遇到了bgsave
CPU飙升
bgsave之后会做一个emptyDB
这个时候做bgsave cow的机制就没了
会重新加载整个RDB
然后做swap
高并发场景下无限同步
tps过高的时候
master 复制缓存挤压区的时候 有个参数client-output-buffer-limit 默认1M
大量请求会让值升高 超出阈值断开
重连
slave做RDB同步的时候会导致TPS过高无法加载
阻塞久了复制缓冲区的数据就被冲掉了，是个队列会踢掉之前的数据
slave重连对比offset发现空档重新sync
预估体量参数动态调整
canal 并发修改
es 自动机构建
redis es 深分页
为什么换工作
想去阿里
有什么想问我的
我的回答有什么建议么
阿里的解决方案
团队主要做的事情
项目
竞价服务
竞价策略
开关
熔控
出价
监控服务
行为监控
获胜
点击
曝光
播放完成
物料
送审
合成
商品自动拼装



注：如果图被压缩了，可以去公众号【三太子敖丙】回复【复习】获取原图

Spring

Spring框架的七大模块

Spring Core：框架的最基础部分，提供 IoC 容器，对 bean 进行管理。

Spring Context：继承BeanFactory，提供上下文信息，扩展出JNDI、EJB、电子邮件、国际化等功能。

Spring DAO：提供了JDBC的抽象层，还提供了声明性事务管理方法。

Spring ORM：提供了JPA、JDO、Hibernate、MyBatis 等ORM映射层.

Spring AOP：集成了所有AOP功能

Spring Web：提供了基础的 Web 开发的上下文信息，现有的Web框架，如JSF、Tapestry、Structs 等，提供了集成

Spring Web MVC：提供了 Web 应用的 Model-View-Controller 全功能实现。

Bean定义5种作用域

singleton（单例） prototype（原型） request session global session

spring ioc初始化流程?

resource定位 即寻找用户定义的bean资源，由 ResourceLoader通过统一的接口Resource接口来完成 beanDefinition载入 BeanDefinitionReader读取、解析Resource定位的资源 成BeanDefinition 载入到ioc中（通过HashMap进行维护BD） BeanDefinition注册 即向IOC容器注册这些BeanDefinition， 通过BeanDefinitionRegistery实现

BeanDefinition加载流程?

定义BeanDefinitionReader解析xml的document BeanDefinitionDocumentReader解析document成beanDefinition

DI依赖注入流程?（实例化，处理Bean之间的依赖关系）

过程在Ioc初始化后，依赖注入的过程是用户第一次向IoC容器索要Bean时触发

- 如果设置lazy-init=true，会在第一次getBean的时候才初始化bean， lazy-init=false，会容器启动的时候直接初始化（singleton bean）；
- 调用BeanFactory.getBean（）生成bean的；
- 生成bean过程运用装饰器模式产生的bean都是beanWrapper（bean的增强）；

  依赖注入怎么处理bean之间的依赖关系?

  其实就是通过在beanDefinition载入时，如果bean有依赖关系，通过占位符来代替，在调用getbean时候，如果遇到占位符，从ioc里获取bean注入到本实例来

Bean的生命周期?

- 实例化Bean： Ioc容器通过获取BeanDefinition对象中的信息进行实例化，实例化对象被包装在BeanWrapper对象中
- 设置对象属性（DI）：通过BeanWrapper提供的设置属性的接口完成属性依赖注入；
- 注入Aware接口（BeanFactoryAware， 可以用这个方式来获取其它 Bean，ApplicationContextAware）：Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给bean
- BeanPostProcessor：自定义的处理（分前置处理和后置处理）
- InitializingBean和init-method：执行我们自己定义的初始化方法
- 使用
- destroy：bean的销毁

IOC：控制反转：将对象的创建权，由Spring管理. DI（依赖注入）：在Spring创建对象的过程中，把对象依赖的属性注入到类中。

Spring的IOC注入方式

构造器注入 setter方法注入 注解注入 接口注入

怎么检测是否存在循环依赖?

Bean在创建的时候可以给该Bean打标，如果递归调用回来发现正在创建中的话，即说明了循环依赖了。

Spring如解决Bean循环依赖问题?

Spring中循环依赖场景有：

- 构造器的循环依赖
- 属性的循环依赖
- singletonObjects：第一级缓存，里面放置的是实例化好的单例对象； earlySingletonObjects：第二级缓存，里面存放的是提前曝光的单例对象； singletonFactories：第三级缓存，里面存放的是要被实例化的对象的对象工厂
- 创建bean的时候Spring首先从一级缓存singletonObjects中获取。如果获取不到，并且对象正在创建中，就再从二级缓存earlySingletonObjects中获取，如果还是获取不到就从三级缓存

singletonFactories中取（Bean调用构造函数进行实例化后，即使属性还未填充，就可以通过三级缓存向外提前暴露依赖的引用值（提前曝光），根据对象引用能定位到堆中的对象，其原理是基于Java的引用传递），取到后从三级缓存移动到了二级缓存完全初始化之后将自己放入到一级缓存中供其他使用，

- 因为加入singletonFactories三级缓存的前提是执行了构造器，所以构造器的循环依赖没法解决。
- 构造器循环依赖解决办法：在构造函数中使用@Lazy注解延迟加载。在注入依赖时，先注入代理对象，当首次使用时再创建对象说明：一种互斥的关系而非层次递进的关系，故称为三个Map而非三级缓存的缘由 完成注入；

Spring 中使用了哪些设计模式?

- 工厂模式： spring中的BeanFactory就是简单工厂模式的体现，根据传入唯一的标识来获得bean对象；
- 单例模式： 提供了全局的访问点BeanFactory；
- 代理模式： AOP功能的原理就使用代理模式（1、JDK动态代理。2、CGLib字节码生成技术代理。）
- 装饰器模式： 依赖注入就需要使用BeanWrapper；
- 观察者模式： spring中Observer模式常用的地方是listener的实现。如ApplicationListener。
- 策略模式： Bean的实例化的时候决定采用何种方式初始化bean实例（反射或者CGLIB动态字节码生成）

AOP 核心概念

1、切面（aspect）：类是对物体特征的抽象，切面就是对横切关注点的抽象

2、横切关注点：对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点。

3、连接点（joinpoint）：被拦截到的点，因为 Spring 只支持方法类型的连接点，所以在Spring 中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器。

4、切入点（pointcut）：对连接点进行拦截的定义

5、通知（advice）：所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类。

6、目标对象：代理的目标对象

7、织入（weave）：将切面应用到目标对象并导致代理对象创建的过程

8、引入（introduction）：在不修改代码的前提下，引入可以在运行期为类动态地添加方法或字段。

解释一下AOP

传统oop开发代码逻辑自上而下的，这个过程中会产生一些横切性问题，这些问题与我们主业务逻辑关系不大，会散落在代码的各个地方，造成难以维护，aop思想就是把业务逻辑与横切的问题进行分离，达到解耦的目的，提高代码重用性和开发效率；

AOP 主要应用场景有：

- 记录日志
- 监控性能
- 权限控制
- 事务管理

AOP源码分析

- @EnableAspectJAutoProxy给容器（beanFactory）中注册一个AnnotationAwareAspectJAutoProxyCreator对象；
- AnnotationAwareAspectJAutoProxyCreator对目标对象进行代理对象的创建，对象内部，是封装JDK和CGlib两个技术，实现动态代理对象创建的（创建代理对象过程中，会先创建一个代理工厂，获取到所有的增强器（通知方法），将这些增强器和目标类注入代理工厂，再用代理工厂创建对象）；
- 代理对象执行目标方法，得到目标方法的拦截器链，利用拦截器的链式机制，依次进入每一个拦截器进行执行

  AOP应用场景

  - 日志记录
  - 事务管理
  - 线程池关闭等

AOP使用哪种动态代理?

- 当bean的是实现中存在接口或者是Proxy的子类，---jdk动态代理；不存在接口，spring会采用CGLIB来生成代理对象；
- JDK 动态代理主要涉及到 java.lang.reflect 包中的两个类：Proxy 和 InvocationHandler。
- Proxy 利用 InvocationHandler（定义横切逻辑） 接口动态创建 目标类的代理对象。

jdk动态代理

- 通过bind方法建立代理与真实对象关系，通过Proxy.newProxyInstance（target）生成代理对象
- 代理对象通过反射invoke方法实现调用真实对象的方法

动态代理与静态代理区别

- 静态代理，程序运行前代理类的.class文件就存在了；
- 动态代理：在程序运行时利用反射动态创建代理对象<复用性，易用性，更加集中都调用invoke>

CGLIB与JDK动态代理区别

- Jdk必须提供接口才能使用；
- C不需要，只要一个非抽象类就能实现动态代理

SpringMVC

springMVC流程：

（1）：用户请求发送给DispatcherServlet，DispatcherServlet调用HandlerMapping处理器映射器；

（2）：HandlerMapping根据xml或注解找到对应的处理器，生成处理器对象返回给DispatcherServlet；

（3）：DispatcherServlet会调用相应的HandlerAdapter；

（4）：HandlerAdapter经过适配调用具体的处理器去处理请求，生成ModelAndView返回给DispatcherServlet

（5）：DispatcherServlet将ModelAndView传给ViewReslover解析生成View返回给DispatcherServlet；

（6）：DispatcherServlet根据View进行渲染视图；

->DispatcherServlet->HandlerMapping->Handler ->DispatcherServlet->HandlerAdapter处理handler->ModelAndView ->DispatcherServlet->ModelAndView->ViewReslover->View ->DispatcherServlet->返回给客户

Mybatis

Mybatis原理

- sqlsessionFactoryBuilder生成sqlsessionFactory（单例）
- 工厂模式生成sqlsession执行sql以及控制事务
- Mybatis通过动态代理使Mapper（sql映射器）接口能运行起来即为接口生成代理对象将sql查询到结果映射成pojo

sqlSessionFactory构建过程

- 解析并读取配置中的xml创建Configuration对象 （单例）
- 使用Configruation类去创建sqlSessionFactory（builder模式）

Mybatis一级缓存与二级缓存

默认情况下一级缓存是开启的，而且是不能关闭的。

- 一级缓存是指 SqlSession 级别的缓存 原理：使用的数据结构是一个 map，如果两次中间出现 commit 操作 （修改、添加、删除），本 sqlsession 中的一级缓存区域全部清空
- 二级缓存是指可以跨 SqlSession 的缓存。是 mapper 级别的缓存； 原理： 是通过 CacheExecutor 实现的。CacheExecutor其实是 Executor 的代理对象

Zookeeper+eureka+springcloud

SpringBoot启动流程

- new springApplication对象，利用spi机制加载applicationContextInitializer， applicationLister接口实例（META-INF/spring.factories）；
- 调run方法准备Environment，加载应用上下文（applicationContext），发布事件 很多通过lister实现
- 创建spring容器， refreshContext（），实现starter自动化配置，spring.factories文件加载， bean实例化

SpringBoot自动配置的原理

- @EnableAutoConfiguration找到META-INF/spring.factories（需要创建的bean在里面）配置文件
- 读取每个starter中的spring.factories文件

Spring Boot 的核心注解

核心注解是@SpringBootApplication 由以下三种组成

- @SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。
- @EnableAutoConfiguration：打开自动配置的功能。
- @ComponentScan：Spring组件扫描。

SpringBoot常用starter都有哪些

spring-boot-starter-web - Web 和 RESTful 应用程序； spring-boot-starter-test - 单元测试和集成测试； spring-boot-starter-jdbc - 传统的 JDBC； spring-boot-starter-security - 使用 SpringSecurity 进行身份验证和授权； spring-boot-starter-data-jpa - 带有 Hibernate 的 Spring Data JPA； spring-boot-starter-data-rest - 使用 Spring Data REST 公布简单的 REST 服务

Spring Boot 的核心配置文件

（1）：Application.yml 一般用来定义单个应用级别的，如果搭配 spring-cloud-config 使用

（2）.Bootstrap.yml（先加载） 系统级别的一些参数配置，这些参数一般是不变的

Zuul与Gateway区别

（1）：zuul则是netflix公司的项目集成在spring-cloud中使用而已， Gateway是spring-cloud的 一个子项目；

（2）：zuul不提供异步支持流控等均由hystrix支持， gateway提供了异步支持，提供了抽象负载均衡，提供了抽象流控； 理论上gateway则更适合于提高系统吞吐量（但不一定能有更好的性能），最终性能还需要通过严密的压测来决定

（3）：两者底层实现都是servlet，但是gateway多嵌套了一层webflux框架

（4）： zuul可用至其他微服务框架中，内部没有实现限流、负载均衡；gateway只能用在springcloud中；

Zuul原理分析

（1）：请求给zuulservlet处理（HttpServlet子类） zuulservlet中有一个zuulRunner对象，该对象中初始化了RequestContext（存储请求的数据），RequestContext被所有的zuulfilter共享；

（2）： zuulRunner中有 FilterProcessor（zuulfilter的管理器），其从filterloader 中获取zuulfilter；

（3）：有了这些filter之后， zuulservelet执行的Pre-> route-> post 类型的过滤器，如果在执行这些过滤器有错误的时候则会执行error类型的过滤器，执行完后把结果返回给客户端.

Gateway原理分析

（1）：请求到达DispatcherHandler， DispatchHandler在IOC容器初始化时会在容器中实例化HandlerMapping接口

（2）：用handlerMapping根据请求URL匹配到对应的Route，然后有对应的filter做对应的请求转发最终response返回去

Zookeeper 工作原理（待查）

Zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。

zoo与eur区别

- zookeeper保证cp（一致性）
- eureka保证ap（可用性）
- zoo在选举期间注册服务瘫痪，期间不可用
- eur各个节点平等关系，只要有一台就可保证服务可用，而查询到的数据可能不是最新的，可以很好应对网络故障导致部分节点失联情况
- zoo有leader和follower角色，eur各个节点平等
- zoo采用半数存活原则（避免脑裂），eur采用自我保护机制来解决分区问题
- eur本质是个工程，zoo只是一个进程 ZooKeeper基于CP，不保证高可用，如果zookeeper正在选主，或者Zookeeper集群中半数以上机器不可用，那么将无法获得数据。 Eureka基于AP，能保证高可用，即使所有机器都挂了，也能拿到本地缓存的数据。作为注册中心，其实配置是不经常变动的，只有发版（发布新的版本）和机器出故障时会变。对于不经常变动的配置来说，CP是不合适的，而AP在遇到问题时可以用牺牲一致性来保证可用性，既返回旧数据，缓存数据。 所以理论上Eureka是更适合做注册中心。而现实环境中大部分项目可能会使用ZooKeeper，那是因为集群不够大，并且基本不会遇到用做注册中心的机器一半以上都挂了的情况。所以实际上也没什么大问题。

Hystrix原理（待查）

通过维护一个自己的线程池，当线程池达到阈值的时候，就启动服务降级，返回fallback默认值

为什么需要hystrix熔断

防止雪崩，及时释放资源，防止系统发生更多的额级联故障，需要对故障和延迟进行隔离，防止单个依赖关系的失败影响整个应用程序；

微服务优缺点

- 每个服务高内聚，松耦合，面向接口编程；
- 服务间通信成本，数据一致性，多服务运维难度增加，http传输效率不如rpc

eureka自我保护机制

- eur不移除长时间没收到心跳而应该过期的服务
- 仍然接受新服务注册和查询请求，但是不会同步到其它节点（高可用）
- 当网络稳定后，当前实例新注册信息会同步到其它节点（最终一致性）

MQ对比

ActiveMQ：Apache出品，最早使用的消息队列产品，时间比较长了，最近版本更新比较缓慢。 RabbitMQ：erlang语言开发，支持很多的协议，非常重量级，更适合于企业级的开发。性能较好，但是不利于做二次开发和维护。 RocketMQ：阿里开源的消息中间件，纯Java开发，具有高吞吐量、高可用性、适合大规模分布式系统应用的特点，分布式事务。 ZeroMQ：号称最快的消息队列系统，尤其针对大吞吐量的需求场景，采用 C 语言实现。 消息队列的选型需要根据具体应用需求而定，ZeroMQ 小而美，RabbitMQ 大而稳，Kakfa 和 RocketMQ 快而强劲

JAVA基础

AVL树与红黑树（R-B树）的区别与联系

- AVL是严格的平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；
- 红黑树是用非严格的平衡来换取增删节点时候旋转次数的降低开销；
- 所以简单说，查询多选择AVL树，查询更新次数差不多选红黑树
- AVL树顺序插入和删除时有20%左右的性能优势，红黑树随机操作15%左右优势，现实应用当然一般都是随机情况，所以红黑树得到了更广泛的应用 索引为B+树 Hashmap为红黑树

为啥redis zset使用跳跃链表而不用红黑树实现

- skiplist的复杂度和红黑树一样，而且实现起来更简单。
- 在并发环境下红黑树在插入和删除时需要rebalance，性能不如跳表。

JAVA基本数据类型

（1个字节是8个bit） 整数型：byte（1字节）、short（2字节）、int（4字节）、long（8字节） 浮点型：float（4字节）、double（8字节） 布尔型：boolean（1字节） 字符型：char（2字节）

IO与NIO

包括 类File，outputStream，inputStream，writer，readerseralizable（5类1接口）

NIO三大核心内容 selector（选择器，用于监听channel），channel（通道），buffer（缓冲区）

NIO与IO区别，IO面向流，NIO面向缓冲区；io阻塞，nio非阻塞

异常类

throwable为父类，子为error跟exception，exception分runtime（空指针，越界等）跟checkexception（sql，io，找不到类等异常）

LVS（4层与7层）原理

- 由前端虚拟负载均衡器和后端真实服务器群组成；
- 请求发送给虚拟服务器后其根据包转发策略以及负载均衡调度算法转发给真实服务器
- 所谓四层（lvs，f5）就是基于IP+端口的负载均衡；七层（nginx）就是基于URL等应用层信息的负载均衡

StringBuilder与StringBuffer

- StringBuilder 更快；
- StringBuffer是线程安全的

interrupt/isInterrupted/interrupt区别

- interrupt（） 调用该方法的线程的状态为将被置为"中断"状态（set操作）
- isinterrupted（） 是作用于调用该方法的线程对象所对应的线程的中断信号是true还是false（get操作）。例如我们可以在A线程中去调用B线程对象的isInterrupted方法，查看的是A
- interrupted（）是静态方法：内部实现是调用的当前线程的isInterrupted（），并且会重置当前线程的中断状态（getandset）

sleep与wait区别

sleep属于线程类，wait属于object类；sleep不释放锁

CountDownLatch和CyclicBarrier区别

- con用于主线程等待其他子线程任务都执行完毕后再执行，cyc用于一组线程相互等待大家都达到某个状态后，再同时执行；
- CountDownLatch是不可重用的，CyclicBarrier可重用

终止线程方法

- 使用退出标志，说线程正常退出；

- 通过判断this.interrupted（） throw new InterruptedException（）来停止 使用String常量池作为锁对象会导致两个线程持有相同的锁，另一个线程不执行，改用其他如new Object（）

ThreadLocal的原理和应用

原理：

线程中创建副本，访问自己内部的副本变量，内部实现是其内部类名叫ThreadLocalMap的成员变量threadLocals，key为本身，value为实际存值的变量副本

应用：

- 用来解决数据库连接，存放connection对象，不同线程存放各自session；
- 解决simpleDateFormat线程安全问题；
- 会出现内存泄漏，显式remove..不要与线程池配合，因为worker往往是不会退出的；

threadLocal 内存泄漏问题

如果是强引用，设置tl=null，但是key的引用依然指向ThreadLocal对象，所以会有内存泄漏，而使用弱引用则不会； 但是还是会有内存泄漏存在，ThreadLocal被回收，key的值变成null，导致整个value再也无法被访问到； 解决办法：在使用结束时，调用ThreadLocal.remove来释放其value的引用；

如果我们要获取父线程的ThreadLocal值呢

ThreadLocal是不具备继承性的，所以是无法获取到的，但是我们可以用InteritableThreadLocal来实现这个功能。InteritableThreadLocal继承来ThreadLocal，重写了createdMap方法，已经对应的get和set方法，不是在利用了threadLocals，而是interitableThreadLocals变量。

这个变量会在线程初始化的时候（调用init方法），会判断父线程的interitableThreadLocals变量是否为空，如果不为空，则把放入子线程中，但是其实这玩意没啥鸟用，当父线程创建完子线程后，如果改变父线程内容是同步不到子线程的。。。同样，如果在子线程创建完后，再去赋值，也是没啥鸟用的

线程状态

线程池有5种状态：running，showdown，stop，Tidying，TERMINATED。

- running：线程池处于运行状态，可以接受任务，执行任务，创建线程默认就是这个状态了
- showdown：调用showdown（）函数，不会接受新任务，但是会慢慢处理完堆积的任务。
- stop：调用showdownnow（）函数，不会接受新任务，不处理已有的任务，会中断现有的任务。
- Tidying：当线程池状态为showdown或者stop，任务数量为0，就会变为tidying。这个时候会调用钩子函数terminated（）。
- TERMINATED：terminated（）执行完成。

在线程池中，用了一个原子类来记录线程池的信息，用了int的高3位表示状态，后面的29位表示线程池中线程的个数。

Java中的线程池是如何实现的?

- 线程中线程被抽象为静态内部类Worker，是基于AQS实现的存放在HashSet中；
- 要被执行的线程存放在BlockingQueue中；
- 基本思想就是从workQueue中取出要执行的任务，放在worker中处理；

如果线程池中的一个线程运行时出现了异常，会发生什么

如果提交任务的时候使用了submit，则返回的feature里会存有异常信息，但是如果数execute则会打印出异常栈。但是不会给其他线程造成影响。之后线程池会删除该线程，会新增加一个worker。

线程池原理

- 提交一个任务，线程池里存活的核心线程数小于corePoolSize时，线程池会创建一个核心线程去处理提交的任务
- 如果线程池核心线程数已满，即线程数已经等于corePoolSize，一个新提交的任务，会被放进任务队列workQueue排队等待执行。
- 当线程池里面存活的线程数已经等于corePoolSize了，并且任务队列workQueue也满，判断线程数是否达到maximumPoolSize，即最大线程数是否已满，如果没到达，创建非核心线程执行提交的任务。
- 如果当前的线程数达到了maximumPoolSize，还有新的任务过来的话，直接采用拒绝策略处理。

拒绝策略

- AbortPolicy直接抛出异常阻止线程运行；
- CallerRunsPolicy如果被丢弃的线程任务未关闭，则执行该线程；
- DiscardOldestPolicy移除队列最早线程尝试提交当前任务
- DiscardPolicy丢弃当前任务，不做处理

newFixedThreadPool （固定数目线程的线程池）

- 阻塞队列为无界队列LinkedBlockingQueue
- 适用于处理CPU密集型的任务，适用执行长期的任务

newCachedThreadPool（可缓存线程的线程池）

- 阻塞队列是SynchronousQueue
- 适用于并发执行大量短期的小任务

newSingleThreadExecutor（单线程的线程池）

- 阻塞队列是LinkedBlockingQueue
- 适用于串行执行任务的场景，一个任务一个任务地执行

newScheduledThreadPool（定时及周期执行的线程池）

- 阻塞队列是DelayedWorkQueue
- 周期性执行任务的场景，需要限制线程数量的场景

java锁相关

synchronized实现原理

contentionList（请求锁线程队列） entryList（有资格的候选者队列） waitSet（wait方法后阻塞队列） onDeck（竞争候选者） ower（竞争到锁线程） !ower（执行成功释放锁后状态）； Synchronized 是非公平锁。

Synchronized 在线程进入 ContentionList 时，等待的线程会先尝试自旋获取锁，如果获取不到就进入 ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占 OnDeck 线程的锁资源。

底层是由一对monitorenter和monitorexit指令实现的（监视器锁）

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程：

- 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
- 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
- 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

ReentrantLock 是如何实现可重入性的？

内部自定义了同步器 Sync，加锁的时候通过CAS 算法 ，将线程对象放到一个双向链表 中，每次获取锁的时候 ，看下当前维 护的那个线程ID和当前请求的线程ID是否一样，一样就可重入了；

ReentrantLock如何避免死锁?

- 响应中断lockInterruptibly（）
- 可轮询锁tryLock（）
- 定时锁tryLock（long time）

tryLock 和 lock 和 lockInterruptibly 的区别

（1）：tryLock 能获得锁就返回 true，不能就立即返回 false，

（2）：tryLock（long timeout，TimeUnit unit），可以增加时间限制，如果超过该时间段还没获得锁，返回 false

（3）：lock 能获得锁就返回 true，不能的话一直等待获得锁

（4）：lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程， lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

CountDownLatch和CyclicBarrier的区别是什么

CountDownLatch是等待其他线程执行到某一个点的时候，在继续执行逻辑（子线程不会被阻塞，会继续执行），只能被使用一次。最常见的就是join形式，主线程等待子线程执行完任务，在用主线程去获取结果的方式（当然不一定），内部是用计数器相减实现的（没错，又特么是AQS），AQS的state承担了计数器的作用，初始化的时候，使用CAS赋值，主线程调用await（）则被加入共享线程等待队列里面，子线程调用countDown的时候，使用自旋的方式，减1，知道为0，就触发唤醒。

CyclicBarrier回环屏障，主要是等待一组线程到底同一个状态的时候，放闸。CyclicBarrier还可以传递一个Runnable对象，可以到放闸的时候，执行这个任务。CyclicBarrier是可循环的，当调用await的时候如果count变成0了则会重置状态，如何重置呢，CyclicBarrier新增了一个字段parties，用来保存初始值，当count变为0的时候，就重新赋值。还有一个不同点，CyclicBarrier不是基于AQS的，而是基于RentrantLock实现的。存放的等待队列是用了条件变量的方式。

synchronized与ReentrantLock区别

- 都是可重入锁； R是显示获取和释放锁，s是隐式；
- R更灵活可以知道有没有成功获取锁，可以定义读写锁，是api级别，s是JVM级别；
- R可以定义公平锁；Lock是接口，s是java中的关键字

什么是信号量Semaphore

信号量是一种固定资源的限制的一种并发工具包，基于AQS实现的，在构造的时候会设置一个值，代表着资源数量。信号量主要是应用于是用于多个共享资源的互斥使用，和用于并发线程数的控制（druid的数据库连接数，就是用这个实现的），信号量也分公平和非公平的情况，基本方式和reentrantLock差不多，在请求资源调用task时，会用自旋的方式减1，如果成功，则获取成功了，如果失败，导致资源数变为了0，就会加入队列里面去等待。调用release的时候会加一，补充资源,并唤醒等待队列。

Semaphore 应用

- acquire（） release（） 可用于对象池，资源池的构建，比如静态全局对象池，数据库连接池；
- 可创建计数为1的S，作为互斥锁（二元信号量）

可重入锁概念

（1）：可重入锁是指同一个线程可以多次获取同一把锁，不会因为之前已经获取过还没释放而阻塞；

（2）：reentrantLock和synchronized都是可重入锁

（3）：可重入锁的一个优点是可一定程度避免死锁

ReentrantLock原理（CAS+AQS）

CAS+AQS队列来实现

（1）：先通过CAS尝试获取锁， 如果此时已经有线程占据了锁，那就加入AQS队列并且被挂起；

（2）： 当锁被释放之后， 排在队首的线程会被唤醒CAS再次尝试获取锁，

（3）：如果是非公平锁， 同时还有另一个线程进来尝试获取可能会让这个线程抢到锁；

（4）：如果是公平锁， 会排到队尾，由队首的线程获取到锁。

AQS 原理

Node内部类构成的一个双向链表结构的同步队列，通过控制（volatile的int类型）state状态来判断锁的状态，对于非可重入锁状态不是0则去阻塞；

对于可重入锁如果是0则执行，非0则判断当前线程是否是获取到这个锁的线程，是的话把state状态＋1，比如重入5次，那么state=5。 而在释放锁的时候，同样需要释放5次直到state=0其他线程才有资格获得锁

AQS两种资源共享方式

- Exclusive：独占，只有一个线程能执行，如ReentrantLock
- Share：共享，多个线程可以同时执行，如Semaphore、CountDownLatch、ReadWriteLock，CyclicBarrier

CAS原理

内存值V，旧的预期值A，要修改的新值B，当A=V时，将内存值修改为B，否则什么都不做；

CAS的缺点：

（1）：ABA问题； （2）：如果CAS失败，自旋会给CPU带来压力； （3）：只能保证对一个变量的原子性操作，i++这种是不能保证的

CAS在java中的应用：

（1）：Atomic系列

公平锁与分公平锁

（1）：公平锁指在分配锁前检查是否有线程在排队等待获取该锁，优先分配排队时间最长的线程，非公平直接尝试获取锁 （2）：公平锁需多维护一个锁线程队列，效率低；默认非公平

独占锁与共享锁

（1）：ReentrantLock为独占锁（悲观加锁策略） （2）：ReentrantReadWriteLock中读锁为共享锁 （3）： JDK1.8 邮戳锁（StampedLock）， 不可重入锁 读的过程中也允许获取写锁后写入！这样一来，我们读的数据就可能不一致，所以，需要一点额外的代码来判断读的过程中是否有写入，这种读锁是一种乐观锁， 乐观锁的并发效率更高，但一旦有小概率的写入导致读取的数据不一致，需要能检测出来，再读一遍就行

4种锁状态

- 无锁
- 偏向锁 会偏向第一个访问锁的线程，当一个线程访问同步代码块获得锁时，会在对象头和栈帧记录里存储锁偏向的线程ID，当这个线程再次进入同步代码块时，就不需要CAS操作来加锁了，只要测试一下对象头里是否存储着指向当前线程的偏向锁 如果偏向锁未启动，new出的对象是普通对象（即无锁，有稍微竞争会成轻量级锁），如果启动，new出的对象是匿名偏向（偏向锁） 对象头主要包括两部分数据：Mark Word（标记字段， 存储对象自身的运行时数据）、class Pointer（类型指针， 是对象指向它的类元数据的指针）
- 轻量级锁（自旋锁） （1）：在把线程进行阻塞操作之前先让线程自旋等待一段时间，可能在等待期间其他线程已经 解锁，这时就无需再让线程执行阻塞操作，避免了用户态到内核态的切换。（自适应自旋时间为一个线程上下文切换的时间）
- （2）：在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁
- （3）：自旋锁底层是通过指向线程栈中Lock Record的指针来实现的
- 重量级锁

轻量级锁与偏向锁的区别

（1）：轻量级锁是通过CAS来避免进入开销较大的互斥操作

（2）：偏向锁是在无竞争场景下完全消除同步，连CAS也不执行

自旋锁升级到重量级锁条件

（1）：某线程自旋次数超过10次；

（2）：等待的自旋线程超过了系统core数的一半；

读写锁了解嘛，知道读写锁的实现方式嘛

常用的读写锁ReentrantReanWritelock，这个其实和reentrantLock相似，也是基于AQS的，但是这个是基于共享资源的，不是互斥，关键在于state的处理，读写锁把高16为记为读状态，低16位记为写状态，就分开了，读读情况其实就是读锁重入，读写/写读/写写都是互斥的，只要判断低16位就好了。

zookeeper实现分布式锁

（1）：利用节点名称唯一性来实现，加锁时所有客户端一起创建节点，只有一个创建成功者获得锁，解锁时删除节点。

（2）：利用临时顺序节点实现，加锁时所有客户端都创建临时顺序节点，创建节点序列号最小的获得锁，否则监视比自己序列号次小的节点进行等待

（3）：方案2比1好处是当zookeeper宕机后，临时顺序节点会自动删除释放锁，不会造成锁等待；

（4）：方案1会产生惊群效应（当有很多进程在等待锁的时候，在释放锁的时候会有很多进程就过来争夺锁）。

（5）：由于需要频繁创建和删除节点，性能上不如redis锁

volatile变量

（1）：变量可见性

（2）：防止指令重排序

（3）：保障变量单次读，写操作的原子性，但不能保证i++这种操作的原子性，因为本质是读，写两次操作

volatile如何保证线程间可见和避免指令重排

volatile可见性是有指令原子性保证的，在jmm中定义了8类原子性指令，比如write，store，read，load。而volatile就要求write-store，load-read成为一个原子性操作，这样子可以确保在读取的时候都是从主内存读入，写入的时候会同步到主内存中（准确来说也是内存屏障），指令重排则是由内存屏障来保证的，由两个内存屏障:

- 一个是编译器屏障：阻止编译器重排，保证编译程序时在优化屏障之前的指令不会在优化屏障之后执行。
- 第二个是cpu屏障：sfence保证写入，lfence保证读取，lock类似于锁的方式。java多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个lock指令，就是增加一个完全的内存屏障指令。

JVM

jre、jdk、jvm的关系：

jdk是最小的开发环境，由jre++java工具组成。

jre是java运行的最小环境，由jvm+核心类库组成。

jvm是虚拟机，是java字节码运行的容器，如果只有jvm是无法运行java的，因为缺少了核心类库。

JVM内存模型

（1）：堆<对象，静态变量，共享

（2）：方法区<存放类信息，常量池，共享>（java8移除了永久代（PermGen），替换为元空间（Metaspace））

（3）：虚拟机栈<线程执行方法的时候内部存局部变量*会存堆中对象的地址*等等数据>

（4）：本地方法栈<存放各种native方法的局部变量表之类的信息>

（5）：程序计数器<记录当前线程执行到哪一条字节码指令位置>

对象4种引用

（1）：强（内存泄露主因）

（2）：软（只有软引用的话，空间不足将被回收），适合缓存用

（3）：弱（只，GC会回收）

（4）：虚引用（用于跟踪GC状态）用于管理堆外内存

对象的构成：

一个对象分为3个区域：对象头、实例数据、对齐填充

对象头：主要是包括两部分，1.存储自身的运行时数据比如hash码，分代年龄，锁标记等（但是不是绝对哦，锁状态如果是偏向锁，轻量级锁，是没有hash码的。。。是不固定的）2.指向类的元数据指针。还有可能存在第三部分，那就是数组类型，会多一块记录数组的长度（因为数组的长度是jvm判断不出来的，jvm只有元数据信息）

实例数据：会根据虚拟机分配策略来定，分配策略中，会把相同大小的类型放在一起，并按照定义顺序排列（父类的变量也会在哦）

对齐填充：这个意义不是很大，主要在虚拟机规范中对象必须是8字节的整数，所以当对象不满足这个情况时，就会用占位符填充

如果判断一个对象是否存活：

一般判断对象是否存活有两种算法，一种是引用计数，另外一种是可达性分析。在java中主要是第二种

java是根据什么来执行可达性分析的：

根据GC ROOTS。GC ROOTS可以的对象有：虚拟机栈中的引用对象，方法区的类变量的引用，方法区中的常量引用，本地方法栈中的对象引用。

JVM 类加载顺序

（1）：加载 获取类的二进制字节流，将其静态存储结构转化为方法区的运行时数据结构

（2）：校验 文件格式验证，元数据验证，字节码验证，符号引用验证

（3）：准备 在方法区中对类的static变量分配内存并设置类变量数据类型默认的初始值，不包括实例变量，实例变量将会在对象实例化的时候随着对象一起分配在Java堆中

（4）：解析 将常量池内的符号引用替换为直接引用的过程

（5）：初始化 为类的静态变量赋予正确的初始值（Java代码中被显式地赋予的值）

JVM三种类加载器

（1）：启动类加载器（home） 加载jvm核心类库，如java.lang.*等

（2）：扩展类加载器（ext）， 父加载器为启动类加载器，从jre/lib/ext下加载类库

（3）：应用程序类加载器（用户classpath路径） 父加载器为扩展类加载器，从环境变量中加载类

双亲委派机制

（1）：类加载器收到类加载的请求

（2）：把这个请求委托给父加载器去完成，一直向上委托，直到启动类加载器

（3）：启动器加载器检查能不能加载，能就加载（结束）；否则，抛出异常，通知子加载器进行加载

（4）：保障类的唯一性和安全性以及保证JDK核心类的优先加载

双亲委派模型有啥作用：

保证java基础类在不同的环境还是同一个Class对象，避免出现了自定义类覆盖基础类的情况，导致出现安全问题。还可以避免类的重复加载。

如何打破双亲委派模型?

（1）：自定义类加载器，继承ClassLoader类重写loadClass方法；

（2）：SPI

tomcat是如何打破双亲委派模型：

tomcat有着特殊性，它需要容纳多个应用，需要做到应用级别的隔离，而且需要减少重复性加载，所以划分为：/common 容器和应用共享的类信息，/server容器本身的类信息，/share应用通用的类信息,/WEB-INF/lib应用级别的类信息。整体可以分为：boostrapClassLoader->ExtensionClassLoader->ApplicationClassLoader->CommonClassLoader->CatalinaClassLoader（容器本身的加载器）/ShareClassLoader（共享的）->WebAppClassLoader。虽然第一眼是满足双亲委派模型的，但是不是的，因为双亲委派模型是要先提交给父类装载，而tomcat是优先判断是否是自己负责的文件位置，

进行加载的。

SPI： （Service Provider interface）

（1）：服务提供接口（服务发现机制）：

（2）：通过加载ClassPath下META_INF/services，自动加载文件里所定义的类

（3）：通过ServiceLoader.load/Service.providers方法通过反射拿到实现类的实例

SPI应用?

（1）：应用于JDBC获取数据库驱动连接过程就是应用这一机制

（2）：apache最早提供的common-logging只有接口.没有实现..发现日志的提供商通过SPI来具体找到日志提供商实现类

双亲委派机制缺陷?

（1）：双亲委派核心是越基础的类由越上层的加载器进行加载， 基础的类总是作为被调用代码调用的API，无法实现基础类调用用户的代码….

（2）：JNDI服务它的代码由启动类加载器去加载，但是他需要调独立厂商实现的应用程序，如何解决?线程上下文件类加载器（Thread Context ClassLoader）， JNDI服务使用这个线程上下文类加载器去加载所需要的SPI代码，也就是父类加载器请求子类加载器去完成类加载动作Java中所有涉及SPI的加载动作基本上都采用这种方式，例如JNDI，JDBC

导致fullGC的原因

（1）：老年代空间不足

（2）：永久代（方法区）空间不足

（3）：显式调用system.gc（）

堆外内存的优缺点

Ehcache中的一些版本，各种 NIO 框架，Dubbo，Memcache 等中会用到，NIO包下ByteBuffer来创建堆外内存 堆外内存，其实就是不受JVM控制的内存。

相比于堆内内存有几个优势：

减少了垃圾回收的工作，因为垃圾回收会暂停其他的工作。 加快了复制的速度。因为堆内在 flush 到远程时，会先复制到直接内存（非堆内存），然后在发送；而堆外内存相当于省略掉了复制这项工作。 可以扩展至更大的内存空间。比如超过 1TB 甚至比主存还大的空间。

缺点总结如下：

堆外内存难以控制，如果内存泄漏，那么很难排查，通过-XX：MaxDirectMemerySize来指定，当达到阈值的时候，调用system.gc来进行一次full gc 堆外内存相对来说，不适合存储很复杂的对象。一般简单的对象或者扁平化的比较适合 jstat查看内存回收概况，实时查看各个分区的分配回收情况， jmap查看内存栈，查看内存中对象占用大小， jstack查看线程栈，死锁，性能瓶颈

JVM七种垃圾收集器

（1）： Serial 收集器 复制算法，单线程，新生代）

（2）： ParNew 收集器（复制算法，多线程，新生代）

（3）： Parallel Scavenge 收集器（多线程，复制算法，新生代，高吞吐量）

（4）：Serial Old 收集器（标记-整理算法，老年代）

（5）：Parallel Old 收集器（标记-整理算法，老年代，注重吞吐量的场景下，jdk8默认采用 Parallel Scavenge + Parallel Old 的组合）

（6）：CMS 收集器（标记-清除算法，老年代，垃圾回收线程几乎能做到与用户线程同时工作，吞吐量低，内存碎片）以牺牲吞吐量为代价来获得最短回收停顿时间-XX：+UseConcMarkSweepGC jdk1.8默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代） jdk1.9 默认垃圾收集器G1

使用场景：

（1）：应用程序对停顿比较敏感

（2）：在JVM中，有相对较多存活时间较长的对象（老年代比较大）会更适合使用CMS

cms垃圾回收过程：

（1）：初始标识<找到gcroot（stw）>

GC Roots有以下几种：

1：系统类加载器加载的对象

2：处于激活状态的线程

3：JNI栈中的对象

4：正在被用于同步的各种锁对象

5：JVM自身持有的对象，比如系统类加载器等。

（2）：并发标记（三色标记算法） 三色标记算法处理并发标记出现对象引用变化情况： 黑：自己+子对象标记完成 灰：自己完成，子对象未完成 白：未标记； 并发标记 黑->灰->白 重新标记 灰->白引用消失，黑引用指向->白，导致白漏标 cms处理办法是incremental update方案 （增量更新）把黑色变成灰色 多线程下并发标记依旧会产生漏标问题，所以cms必须remark一遍（jdk1.9以后不用cms了）

G1 处理方案：

SATB（snapshot at the begining）把白放入栈中，标记过程是和应用程序并发运行的（不需要Stop-The-World） 这种方式会造成某些是垃圾的对象也被当做是存活的，所以G1会使得占用的内存被实际需要的内存大。不过下一次就回收了 ZGC 处理方案： 颜色指针（color pointers） 2*42方=4T

（3）：重新标记（stw）

（4）并发清理

备注：重新标记是防止标记成垃圾之后，对象被引用

（5）：G1 收集器（新生代 + 老年代，在多 CPU 和大内存的场景下有很好的性能） G1在java9 便是默认的垃圾收集器，是cms 的替代者 逻辑分代，用分区（region）的思想（默认分2048份） 还是有stw 为解决CMS算法产生空间碎片HotSpot提供垃圾收集器，通过-XX：+UseG1GC来启用

G1中提供了三种模式垃圾回收模式

（1）：young gc（eden region被耗尽无法申请内存时，就会触发）

（2）：mixed gc（当老年代大小占整个堆大小百分比达到该阈值时，会触发）

（3）：full gc（对象内存分配速度过快，mixed gc来不及回收，导致老年代被填满，就会触发）

（8）：ZGC和shenandoah （oracle产收费） no stw

arthas 监控工具

（1）：dashboard命令查看总体jvm运行情况

（2）：jvm显示jvm详细信息

（3）：thread 显示jvm里面所有线程信息（类似于jstack） 查看死锁线程命令thread -b

（4）：sc * 显示所有类（search class）

（5）：trace 跟踪方法

定位频繁full GC，堆内存满 oom

第一步：jps获取进程号 第二步：jmap -histo pid | head -20 得知有个对象在不断创建 备注：jmap如果线上服务器堆内存特别大，，会卡死需堆转存（一般会说在测试环境压测，导出转存） -XX：+HeapDumpOnOutOfMemoryError或jmap -dumpLformat=b，file=xxx pid 转出文件进行分析 （arthas没有实现jmap命令）heapdump --live /xxx/xx.hprof导出文件

G1垃圾回收器（重点）

回收过程 （1）：young gc（年轻代回收）--当年轻代的Eden区用尽时--stw 第一阶段，扫描根。 根是指static变量指向的对象，正在执行的方法调用链条上的局部变量等 第二阶段，更新RS（Remembered Sets）。 处理dirty card queue中的card，更新RS。此阶段完成后，RS可以准确的反映老年代对所在的内存分段中对象的引用 第三阶段，处理RS。 识别被老年代对象指向的Eden中的对象，这些被指向的Eden中的对象被认为是存活的对象。 第四阶段，复制对象。 此阶段，对象树被遍历，Eden区内存段中存活的对象会被复制到Survivor区中空的内存分段 第五阶段，处理引用。 处理Soft，Weak，Phantom，Final，JNI Weak 等引用。

（2）：concrruent marking（老年代并发标记） 当堆内存使用达到一定值（默认45%）时，不需要Stop-The-World，在并发标记前先进行一次young gc

（3）：混合回收（mixed gc） 并发标记过程结束以后，紧跟着就会开始混合回收过程。混合回收的意思是年轻代和老年代会同时被回收

（4）：Full GC? Full GC是指上述方式不能正常工作，G1会停止应用程序的执行，使用单线程的内存回收算法进行垃圾回收，性能会非常差，应用程序停顿时间会很长。要避免Full GC的发生，一旦发生需要进行调整。

什么时候发生Full GC呢?

比如堆内存太小，当G1在复制存活对象的时候没有空的内存分段可用，则会回退到full gc，这种情况可以通过增大内存解决

尽管G1堆内存仍然是分代的，但是同一个代的内存不再采用连续的内存结构

年轻代分为Eden和Survivor两个区，老年代分为Old和Humongous两个区

新分配的对象会被分配到Eden区的内存分段上

Humongous区用于保存大对象，如果一个对象占用的空间超过内存分段Region的一半；

如果对象的大小超过一个甚至几个分段的大小，则对象会分配在物理连续的多个Humongous分段上。

Humongous对象因为占用内存较大并且连续会被优先回收

为了在回收单个内存分段的时候不必对整个堆内存的对象进行扫描（单个内存分段中的对象可能被其他内存分段中的对象引用）引入了RS数据结构。RS使得G1可以在年轻代回收的时候不必去扫描老年代的对象，从而提高了性能。每一个内存分段都对应一个RS，RS保存了来自其他分段内的对象对于此分段的引用

JVM会对应用程序的每一个引用赋值语句object.field=object进行记录和处理，把引用关系更新到RS中。但是这个RS的更新并不是实时的。G1维护了一个Dirty Card Queue

那为什么不在引用赋值语句处直接更新RS呢?

这是为了性能的需要，使用队列性能会好很多。

线程本地分配缓冲区（TLAB： Thread Local Allocation Buffer）?

栈上分配->tlab->堆上分配 由于堆内存是应用程序共享的，应用程序的多个线程在分配内存的时候需要加锁以进行同步。为了避免加锁，提高性能每一个应用程序的线程会被分配一个TLAB。TLAB中的内存来自于G1年轻代中的内存分段。当对象不是Humongous对象，TLAB也能装的下的时候，对象会被优先分配于创建此对象的线程的TLAB中。这样分配会很快，因为TLAB隶属于线程，所以不需要加锁

PLAB： Promotion Thread Local Allocation Buffer

G1会在年轻代回收过程中把Eden区中的对象复制（“提升”）到Survivor区中，Survivor区中的对象复制到Old区中。G1的回收过程是多线程执行的，为了避免多个线程往同一个内存分段进行复制，那么复制的过程也需要加锁。为了避免加锁，G1的每个线程都关联了一个PLAB，这样就不需要进行加锁了

OOM问题定位方法

（1）：jmap -heap 10765如上图，可以查看新生代，老生代堆内存的分配大小以及使用情况；

（2）：jstat 查看GC收集情况

（3）：jmap -dump：live，format=b，file=到本地

（4）：通过MAT工具打开分析

DUBBO

dubbo流程

（1）：生产者（Provider）启动，向注册中心（Register）注册

（2）：消费者（Consumer）订阅，而后注册中心通知消费者

（3）：消费者从生产者进行消费

（4）：监控中心（Monitor）统计生产者和消费者

Dubbo推荐使用什么序列化框架，还有哪些?

推荐使用Hessian序列化，还有Duddo、FastJson、Java自带序列化

Dubbo默认使用的是什么通信框架，还有哪些?

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

Dubbo有哪几种负载均衡策略，默认是哪种?

（1）：随机调用<默认>

（2）：权重轮询

（3）：最少活跃数

（4）：一致性Hash

RPC流程

（1）消费者调用需要消费的服务，

（2）：客户端存根将方法、入参等信息序列化发送给服务端存根

（3）：服务端存根反序列化操作根据解码结果调用本地的服务进行相关处理

（4）：本地服务执行具体业务逻辑并将处理结果返回给服务端存根

（5）：服务端存根序列化

（6）：客户端存根反序列化

（7）：服务消费方得到最终结果

RPC框架的实现目标PC框架的实现目标是把调用、编码/解码的过程给封装起来，让用户感觉上像调用本地服务一样的调用远程服务

服务暴露、服务引用、服务调用（TODO）

Redis

redis单线程为什么执行速度这么快?

（1）：纯内存操作，避免大量访问数据库，减少直接读取磁盘数据，redis将数据储存在内存里面，读写数据的时候都不会受到硬盘 I/O 速度的限制，所以速度快

（2）：单线程操作，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

（3）：采用了非阻塞I/O多路复用机制

Redis数据结构底层实现

String：

（1）Simple dynamic string（SDS）的数据结构

```
struct sdshdr{
 //记录buf数组中已使用字节的数量
 //等于 SDS 保存字符串的长度
 int len;
 //记录 buf 数组中未使用字节的数量
 int free;
 //字节数组，用于保存字符串
 char buf[];
}
```

它的优点： （1）不会出现字符串变更造成的内存溢出问题

（2）获取字符串长度时间复杂度为1

（3）空间预分配， 惰性空间释放free字段，会默认留够一定的空间防止多次重分配内存

应用场景： String 缓存结构体用户信息，计数

Hash：

数组+链表的基础上，进行了一些rehash优化； 1.Reids的Hash采用链地址法来处理冲突，然后它没有使用红黑树优化。

2.哈希表节点采用单链表结构。

3.rehash优化 （采用分而治之的思想，将庞大的迁移工作量划分到每一次CURD中，避免了服务繁忙）

应用场景： 保存结构体信息可部分获取不用序列化所有字段

List：

应用场景： （1）：比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现

（2）：list的实现为一个双向链表，即可以支持反向查找和遍历

Set：

内部实现是一个 value为null的HashMap，实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员 是否在集合内的原因。 应用场景： 去重的场景，交集（sinter）、并集（sunion）、差集（sdiff），实现如共同关注、共同喜好、二度好友等功能

Zset：

内部使用HashMap和跳跃表（SkipList）来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score，使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。 跳表：每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的 应用场景： 实现延时队列

redis事务

（1）： Multi开启事务

（2）： Exec执行事务块内命令

（3）： Discard 取消事务

（4）： Watch 监视一个或多个key，如果事务执行前key被改动，事务将打断

redis事务的实现特征

（1）：所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

（2）：Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行

（3）：在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行

（4）：当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。

然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。

Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了

Redis的同步机制?

（1）： 全量拷贝， 1.slave第一次启动时，连接Master，发送PSYNC命令，

2.master会执行bgsave命令来生成rdb文件，期间的所有写命令将被写入缓冲区。

3. master bgsave执行完毕，向slave发送rdb文件
4. slave收到rdb文件，丢弃所有旧数据，开始载入rdb文件
5. rdb文件同步结束之后，slave执行从master缓冲区发送过来的所以写命令。
6. 此后 master 每执行一个写命令，就向slave发送相同的写命令。

（2）：增量拷贝 如果出现网络闪断或者命令丢失等异常情况，从节点之前保存了自身已复制的偏移量和主节点的运行ID

7. 主节点根据偏移量把复制积压缓冲区里的数据发送给从节点，保证主从复制进入正常状态。

redis集群模式性能优化

（1） 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

（2） 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

（3） 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

（4） 尽量避免在压力很大的主库上增加从库

（5） 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3…这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

Redis集群方案

（1）：官方cluster方案

（2）：twemproxy

代理方案twemproxy是一个单点，很容易对其造成很大的压力，所以通常会结合keepalived来实twemproy的高可用

（3）：codis 基于客户端来进行分片

集群不可用场景

（1）：master挂掉，且当前master没有slave

（2）：集群超过半数以上master挂掉，无论是否有slave集群进入fail状态

redis 最适合的场景

（1）：会话缓存session cache

（2）：排行榜/计数器ZRANGE

（3）：发布/订阅

缓存淘汰策略

（1）：先进先出算法（FIFO）

（2）：最近使用最少Least Frequently Used（LFU）

（3）：最长时间未被使用的Least Recently Used（LRU）

当存在热点数据时，LRU的效率很好，但偶发性的、周期性的批量操作会导致LRU命中率急剧下降，缓存污染情况比较严重

redis过期key删除策略

（1）：惰性删除，cpu友好，但是浪费cpu资源

（2）：定时删除（不常用）

（3）：定期删除，cpu友好，节省空间

缓存雪崩以及处理办法

同一时刻大量缓存失效；

处理方法：

（1）：缓存数据增加过期标记

（2）：设置不同的缓存失效时间

（3）：双层缓存策略C1为短期，C2为长期

（4）：定时更新策略

缓存击穿原因以及处理办法

频繁请求查询系统中不存在的数据导致；

处理方法：

（1）：cache null策略，查询反馈结果为null仍然缓存这个null结果，设置不超过5分钟过期时间

（2）：布隆过滤器，所有可能存在的数据映射到足够大的bitmap中 google布隆过滤器：基于内存，重启失效不支持大数据量，无法在分布式场景 redis布隆过滤器：可扩展性，不存在重启失效问题，需要网络io，性能低于google

redis阻塞原因

（1）：数据结构使用不合理bigkey

（2）：CPU饱和

（3）：持久化阻塞，rdb fork子线程，aof每秒刷盘等

hot key出现造成集群访问量倾斜解决办法

（1）：使用本地缓存

（2）： 利用分片算法的特性，对key进行打散处理（给hot key加上前缀或者后缀，把一个hotkey 的数量变成 redis 实例个数N的倍数M，从而由访问一个 redis key 变成访问 N * M 个redis key）

Redis分布式锁

2.6版本以后lua脚本保证setnx跟setex进行原子性（setnx之后，未setex，服务挂了，锁不释放） a获取锁，超过过期时间，自动释放锁，b获取到锁执行，a代码执行完remove锁，a和b是一样的key，导致a释放了b的锁。 解决办法：remove之前判断value（高并发下value可能被修改，应该用lua来保证原子性）

Redis如何做持久化

bgsave做镜像全量持久化，aof做增量持久化。因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据 ，所以需要aof来配合使用。在redis实例重启时，会使用bgsave持久化文件重新构建内存，再使用aof重放近期的操作指令来 实 现完整恢复重启之前的状态。

对方追问那如果突然机器掉电会怎样?

取决于aof日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据.

redis锁续租问题?

（1）：基于redis的redission分布式可重入锁RLock，以及配合java集合中lock；

（2）：Redission 内部提供了一个监控锁的看门狗，不断延长锁的有效期，默认检查锁的超时时间是30秒

（3）：此方案的问题：如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master ，slave实例。但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。 此时就会导致多个客户端对一个分布式锁完成了加锁 解决办法：只需要将新的redis实例，在一个TTL时间内，对客户端不可用即可，在这个时间内，所有客户端锁将被失效或者自动释放.

bgsave的原理是什么?

fork和cow。fork是指redis通过创建子进程来进行bgsave操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写进的页面数据会逐渐和子进程分离开来。

RDB与AOF区别

（1）：R文件格式紧凑，方便数据恢复，保存rdb文件时父进程会fork出子进程由其完成具体持久化工作，最大化redis性能，恢复大数据集速度更快，只有手动提交save命令或关闭命令时才触发备份操作；

（2）：A记录对服务器的每次写操作（默认1s写入一次），保存数据更完整，在redis重启是会重放这些命令来恢复数据，操作效率高，故障丢失数据更少，但是文件体积更大；

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?

使用keys指令可以扫出指定模式的key列表。 如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？ 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。

如何使用Redis做异步队列?

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

可不可以不用sleep呢?

list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

能不能生产一次消费多次呢?

使用pub/sub主题订阅者模式，可以实现1：N的消息队列。

pub/sub有什么缺点?

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

redis如何实现延时队列?

使用sortedset，想要执行时间的时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理。

为啥redis zset使用跳跃链表而不用红黑树实现?

（1）：skiplist的复杂度和红黑树一样，而且实现起来更简单。

（2）：在并发环境下红黑树在插入和删除时需要rebalance，性能不如跳表。

MYSQL

数据库三范式

一： 确保每列的原子性

二：非主键列不存在对主键的部分依赖 （要求每个表只描述一件事情）

三： 满足第二范式，并且表中的列不存在对非主键列的传递依赖

数据库主从复制原理

（1）：主库db的更新事件（update、insert、delete）被写到binlog

（2）：主库创建一个binlog dump thread线程，把binlog的内容发送到从库

（3）：从库创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log.

（4）：从库还会创建一个SQL线程，从relay log里面读取内容写入到slave的db.

复制方式分类

（1）：异步复制（默认） 主库写入binlog日志后即可成功返回客户端，无须等待binlog日志传递给从库的过程，但是一旦主库宕机，就有可能出现丢失数据的情况。

（2）半同步复制：（ 5.5版本之后） （安装半同步复制插件）确保从库接收完成主库传递过来的binlog内容已经写入到自己的relay log（传送log）后才会通知主库上面的等待线程。如果等待超时，则关闭半同步复制，并自动转换为异步复制模式，直到至少有一台从库通知主库已经接收到binlog信息为止

存储引擎

（1）：Myiasm是mysql默认的存储引擎，不支持数据库事务，行级锁，外键；插入更新需锁表，效率低，查询速度快，Myisam使用的是非聚集索引

（2）：innodb 支持事务，底层为B+树实现，适合处理多重并发更新操作，普通select都是快照读，快照读不加锁。InnoDb使用的是聚集索引

聚集索引

（1）：聚集索引就是以主键创建的索引

（2）：每个表只能有一个聚簇索引，因为一个表中的记录只能以一种物理顺序存放，实际的数据页只能按照一颗 B+ 树进行排序

（3）：表记录的排列顺序和与索引的排列顺序一致

（4）：聚集索引存储记录是物理上连续存在

（5）：聚簇索引主键的插入速度要比非聚簇索引主键的插入速度慢很多

（6）：聚簇索引适合排序，非聚簇索引不适合用在排序的场合，因为聚簇索引叶节点本身就是索引和数据按相同顺序放置在一起，索引序即是数据序，数据序即是索引序，所以很快。非聚簇索引叶节点是保留了一个指向数据的指针，索引本身当然是排序的，但是数据并未排序，数据查询的时候需要消耗额外更多的I/O，所以较慢

（7）：更新聚集索引列的代价很高，因为会强制innodb将每个被更新的行移动到新的位置

非聚集索引

（1）：除了主键以外的索引

（2）：聚集索引的叶节点就是数据节点，而非聚簇索引的叶节点仍然是索引节点，并保留一个链接指向对应数据块

（3）：聚簇索引适合排序，非聚簇索引不适合用在排序的场合

（4）：聚集索引存储记录是物理上连续存在，非聚集索引是逻辑上的连续。

使用聚集索引为什么查询速度会变快?

使用聚簇索引找到包含第一个值的行后，便可以确保包含后续索引值的行在物理相邻

建立聚集索引有什么需要注意的地方吗?

在聚簇索引中不要包含经常修改的列，因为码值修改后，数据行必须移动到新的位置，索引此时会重排，会造成很大的资源浪费

InnoDB 表对主键生成策略是什么样的?

优先使用用户自定义主键作为主键，如果用户没有定义主键，则选取一个Unique键作为主键，如果表中连Unique键都没有定义的话，则InnoDB会为表默认添加一个名为row_id隐藏列作为主键。

非聚集索引最多可以有多少个?

每个表你最多可以建立249个非聚簇索引。非聚簇索引需要大量的硬盘空间和内存

BTree 与 Hash 索引有什么区别?

（1）：BTree索引可能需要多次运用折半查找来找到对应的数据块 （2）：HASH索引是通过HASH函数，计算出HASH值，在表中找出对应的数据 （3）：大量不同数据等值精确查询，HASH索引效率通常比B+TREE高 （4）：HASH索引不支持模糊查询、范围查询和联合索引中的最左匹配规则，而这些Btree索引都支持

数据库索引优缺点

（1）：需要查询，排序，分组和联合操作的字段适合建立索引

（2）：索引多，数据更新表越慢，尽量使用字段值不重复比例大的字段作为索引，联合索引比多个独立索引效率高

（3）：对数据进行频繁查询进建立索引，如果要频繁更改数据不建议使用索引

（4）：当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度。

索引的底层实现是B+树，为何不采用红黑树，B树?

（1）：B+Tree非叶子节点只存储键值信息，降低B+Tree的高度，所有叶子节点之间都有一个链指针，数据记录都存放在叶子节点中

（2）： 红黑树这种结构，h明显要深的多，效率明显比B-Tree差很多

（3）：B+树也存在劣势，由于键会重复出现，因此会占用更多的空间。但是与带来的性能优势相比，空间劣势往往可以接受，因此B+树的在数据库中的使用比B树更加广泛

索引失效条件

（1）：条件是or，如果还想让or条件生效，给or每个字段加个索引

（2）：like开头%

（3）：如果列类型是字符串，那一定要在条件中将数据使用引号引用起来，否则不会使用索引

（4）：where中索引列使用了函数或有运算

数据库事务特点

ACID 原子性，一致性，隔离性，永久性

数据库事务说是如何实现的?

（1）：通过预写日志方式实现的，redo和undo机制是数据库实现事务的基础

（2）：redo日志用来在断电/数据库崩溃等状况发生时重演一次刷数据的过程，把redo日志里的数据刷到数据库里，保证了事务 的持久性（Durability）

（3）：undo日志是在事务执行失败的时候撤销对数据库的操作，保证了事务的原子性

数据库事务隔离级别

（1）：读未提交read-uncommitted-- 脏，不可重复读--幻读 A读取了B未提交的事务，B回滚，A 出现脏读；

（2）：不可重复读read-committed-- 不可重复读--幻读 A只能读B已提交的事务，但是A还没结束，B又更新数据隐式提交，然后A又读了一次出现不可重复读；

（3）：可重复读repeatable-read<默认>-- 幻读 事务开启，不允许其他事务的UPDATE修改操作 A读取B已提交的事务，然而B在该表插入新的行，之后A在读取的时候多出一行，出现幻读；

（4）：串行化serializable--

七种事务传播行为

（1）Propagation.REQUIRED<默认> 如果当前存在事务，则加入该事务，如果当前不存在事务，则创建一个新的事务。

（2）Propagation.SUPPORTS 如果当前存在事务，则加入该事务；如果当前不存在事务，则以非事务的方式继续运行。

（3）Propagation.MANDATORY 如果当前存在事务，则加入该事务；如果当前不存在事务，则抛出异常。

（4）Propagation.REQUIRES_NEW 重新创建一个新的事务，如果当前存在事务，延缓当前的事务。

（5）Propagation.NOT_SUPPORTED 以非事务的方式运行，如果当前存在事务，暂停当前的事务。

（6）Propagation.NEVER 以非事务的方式运行，如果当前存在事务，则抛出异常。

（7）Propagation.NESTED 如果没有，就新建一个事务；如果有，就在当前事务中嵌套其他事务。

产生死锁的四个必要条件

（1）：互斥： 资源x的任意一个时刻只能被一个线程持有 （2）：占有且等待：线程1占有资源x的同时等待资源y，并不释放x （3）：不可抢占：资源x一旦被线程1占有，其他线程不能抢占x （4）：循环等待：线程1持有x，等待y，线程2持有y，等待x 当全部满足时才会死锁

@Transaction

底层实现是AOP，动态代理 （1）：实现是通过Spring代理来实现的。生成当前类的代理类，调用代理类的invoke（）方法，在invoke（）方法中调用 TransactionInterceptor拦截器的invoke（）方法；

（2）：非public方式其事务是失效的；

（3）：自调用也会失效，因为动态代理机制导致

（4）多个方法外层加入try...catch，解决办法是可以在catch里 throw new RuntimeException（）来处理

分布式事务

XA方案

有一个事务管理器的概念，负责协调多个数据库（资源管理器）的事务 不适合高并发场景，严重依赖数据库层面，同步阻塞问题；协调者故障则所有参与者会阻塞

TCC方案

严重依赖代码补偿和回滚，一般银行用，和钱相关的支付、交易等相关的场景，我们会用TCC Try，对各个服务的资源做检测，对资源进行锁定或者预留 Confirm，在各个服务中执行实际的操作 Cancel，如果任何一个服务的业务方法执行出错，那么这里就需要进行补偿，即执行已操作成功的业务逻辑的回滚操作

可靠消息最终一致性方案

1）：本地消息服务 本地消息表其实是国外的 ebay 搞出来的这么一套思想。 主动方是认证服务，有个消息异常处理系统，mq，还有消息消费端应用系统，还有采集服务；

- 在我认证返回数据中如果有发票是已经认证的，在处理认证数据的操作与发送消息在同一个本地事务中，业务执行完，消息数据也同时存在一条待确认的数据；
- 发送消息给mq，，mq发送消息给消息消费端服务，同时存一份消息数据，然后发送给采集服务，进行抵账表更新操作；
- 采集服务逻辑处理完以后反馈给消息消费端服务，其服务删除消息数据，同时通知认证服务，把消息记录改为已确认成功费状态；
- 对于异常流程，消息异常处理系统会查询认证服务中过期未确认的消息发送给mq，相当于重试

2）：独立消息最终一致性方案： A 主动方应用系统，B消息服务子系统，C消息状态确认子系统，C2 消息管理子系统 D 消息恢复子系统，mq ，消息消费端E ，被动系统F

流程：
A预发送消息给B，然后执行A业务逻辑，B存储预发送消息，A执行完业务逻辑发送业务操作结果给B，B更新预发送消息为确认并发送消息状态同时发送消息给mq，然后被E监听然后发送给F消费掉
C：对预发送消息异常的处理，去查询待确认状态超时的消息，去A中查询进行数据处理，如果A中业务处理成功了，那么C需改消息状态为确认并发送状态，然后发送消息给mq；如果A中业务处理失败了..那么C直接把消息删除即可.
C2 ： 查询消息的页面，对消息的可视化，以及批量处理死亡消息；
D： B给mq放入数据如果失败，，通过D去重试，多次重试失败，消息设置为死亡
E：确保F执行完成，发送消息给B删除消息
优化建议：
（1）数据库：如果用redis，持久化要配置成appendfsync always，确保每次新添加消息都能持久化进磁盘
（2）在被动方应用业务幂等性判断比较麻烦或者比较耗性能情况下，增加消息日志记录表.用于判断之前有无发送过；

最大努力通知性（定期校对）

（1）业务主动方完成业务处理之后，设置时间阶梯型通知规则向业务活动的被动方发送消息，允许消息丢失.

（2）被动方根据定时策略，向主动方查询，恢复丢失的业务消息

（3）被动方的处理结果不影响主动方的处理结果

（4）需增加业务查询，通知服务，校对系统服务的建设成本

（5）适用于对业务最终一致性的时间敏感度低，跨企业的业务通知活动

（6）比如银行通知，商户通知，交易业务平台间商户通知，多次通知，查询校对等

Seata（阿里）

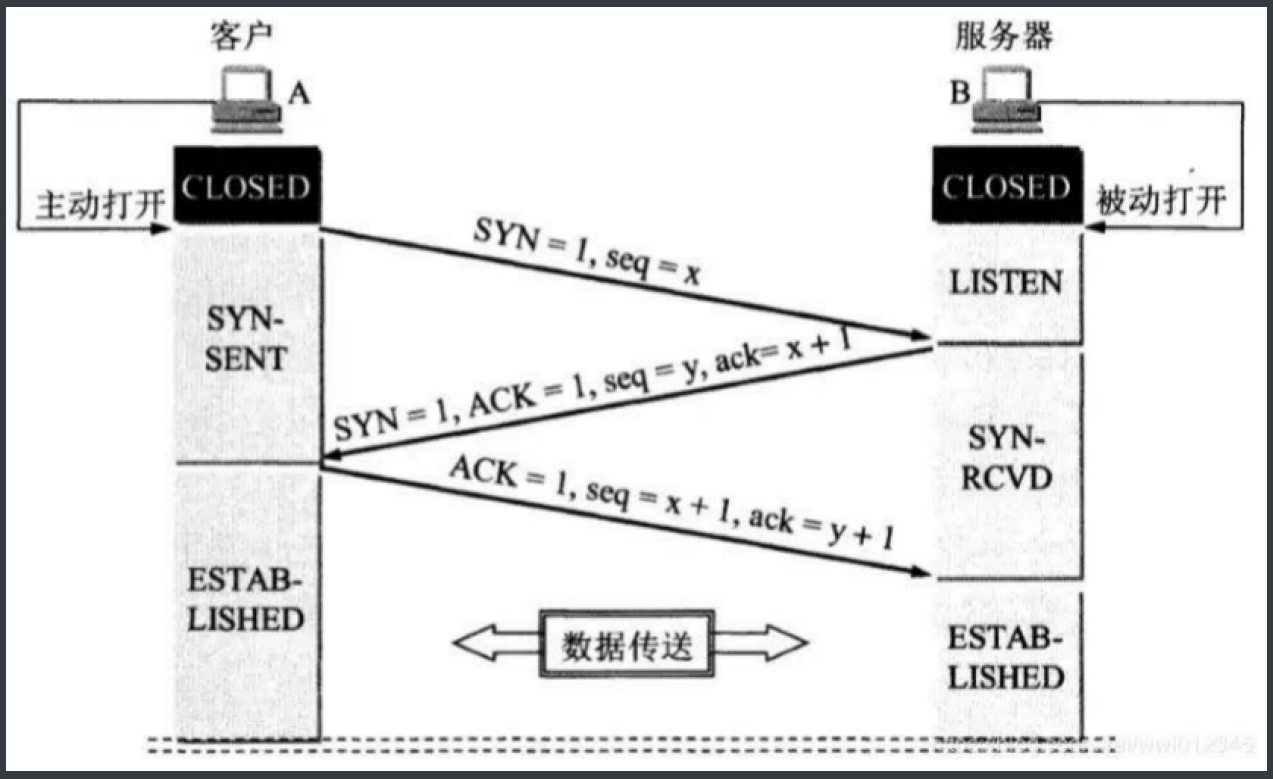
应用层基于SQL解析实现了自动补偿，从而最大程度的降低业务侵入性； 将分布式事务中TC（事务协调者）独立部署，负责事务的注册、回滚； 通过全局锁实现了写隔离与读隔离。

网络

TCP和UDP的比较

TCP向上层提供面向连接的可靠服务 ，UDP向上层提供无连接不可靠服务。 虽然 UDP 并没有 TCP 传输来的准确，但是也能在很多实时性要求高的地方有所作为 对数据准确性要求高，速度可以相对较慢的，可以选用TCP

TCP三次握手



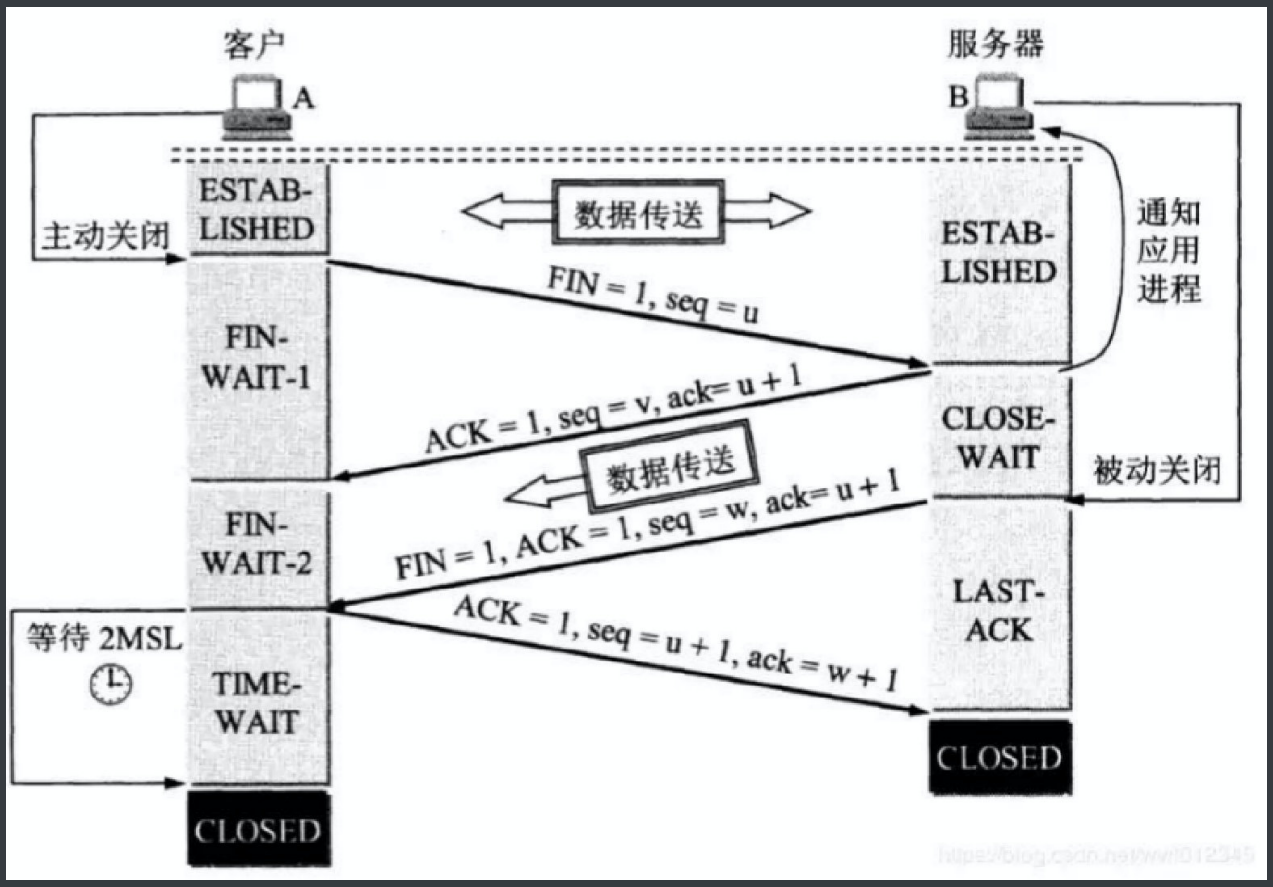
TCP四次挥手

（1）：客户端发送终止命令FIN

（2）：服务端收到后回复ACK，处于close_wait状态

（3）：服务器将关闭前需要发送信息发送给客户端后处于last_ack状态

（4）：客户端收到FIN后发送ack后处于tim-wait而后进入close状态



为什么要进行第三次握手

为了防止服务器端开启一些无用的连接增加服务器开销以及防止已失效的连接请求报文段突然又传送到了服务端

JDK1.8新特性

Lambda表达式

java也开始承认了函数式编程， 就是说函数既可以作为参数，也可以作为返回值， 大大的简化了代码的开发

default关键字

打破接口里面是只能有抽象方法，不能有任何方法的实现，接口里面也可以有方法的实现了

新时间日期APILocalDate | LocalTime | LocalDateTime

之前使用的java.util.Date月份从0开始，我们一般会+1使用，很不方便，java.time.LocalDate月份和星期都改成了enum java.util.Date和SimpleDateFormat都不是线程安全的，而LocalDate和LocalTime和最基本的String一样，是不变类型，不但线程安全，而且不能修改。 新接口更好用的原因是考虑到了日期时间的操作，经常发生往前推或往后推几天的情况。用java.util.Date配合Calendar要写好多代码，而且一般的开发人员还不一定能写对。

JDK1.7与JDK1.8 ConcurrentHashMap对比

（1）：JDK1.7版本的ReentrantLock+Segment+HashEntry（数组）

（2）：JDK1.7采用segment的分段锁机制实现线程安全

（3）：JDK1.8版本中synchronized+CAS+HashEntry（数组）+红黑树

（4）：JDK1.8采用CAS+Synchronized保证线程安全

（5）：查询时间复杂度从原来的遍历链表O（n），变成遍历红黑树O（logN）

1.8 HashMap数组+链表+红黑树来实现hashmap，当碰撞的元素个数大于8时 & 总容量大于64，会有红黑树的引入 除了添加之后，效率都比链表高，1.8之后链表新进元素加到末尾

JDK1.8使用synchronized来代替重入锁ReentrantLock?

（1）：因为粒度降低了，在相对而言的低粒度加锁方式，synchronized并不比ReentrantLock差

（2）：基于JVM的synchronized优化空间更大

（3）：在大数据量下，基于API的ReentrantLock会比基于JVM的内存压力开销更多的内存

JDK1.9新特性

模块系统：

模块是一个包的容器，Java 9 最大的变化之一是引入了模块系统（Jigsaw 项目）。

集合工厂方法

通常，您希望在代码中创建一个集合（例如，List 或 Set ），并直接用一些元素填充它。 实例化集合，几个 “add” 调用，使得代码重复。 Java 9，添加了几种集合工厂方法：

```
Set<Integer> ints = Set.of(1, 2, 3);
List<String> strings = List.of("first", "second");
```

改进的 Stream API

Stream 接口中添加了 4 个新的方法：dropWhile， takeWhile， ofNullable。还有个 iterate 方法的新重载方法

改进的 Javadoc：

Javadoc 现在支持在 API 文档中的进行搜索。另外，Javadoc 的输出现在符合兼容 HTML5 标准。

redis代理集群模式，spring有哪些注解，b+b 红黑树区别，三次握手，valitile重排序底层代码， cas 事务的4个特性，java8 java11 特性， filter和interceptor的区别 @autowired原理， dispatcherservlet，分布式事务解决方案spring都有哪些模块，fork join队列，排序算法，

集合

java的集合框架有哪几种：

两种：collection和map，其中collection分为set和List。

List你使用过哪些

ArrayList和linkedList使用的最多，也最具代表性。

你知道vector和ArrayList和linkedList的区别嘛

ArrayList实现是一个数组，可变数组，默认初始化长度为10，也可以我们设置容量，但是没有设置的时候是默认的空数组，只有在第一步add的时候会进行扩容至10（重新创建了数组），后续扩容按照3/2的大小进行扩容，是线程不安全的，适用多读取，少插入的情况

linkedList是基于双向链表的实现，使用了尾插法的方式，内部维护了链表的长度，以及头节点和尾节点，所以获取长度不需要遍历。适合一些插入/删除频繁的情况。

Vector是线程安全的，实现方式和ArrayList相似，也是基于数组，但是方法上面都有synchronized关键词修饰。其扩容方式是原来的两倍。

hashMap和hashTable和ConcurrentHashMap的区别

hashMap是map类型的一种最常用的数据结构，其底部实现是数组+链表（在1.8版本后变为了数组+链表/红黑树的方式），其key是可以为null的，默认hash值为0。扩容以2的幂等次（为什么。。。因为只有是2的幂等次的时候（n-1）&x==x%n，当然不一定只有一个原因）。是线程不安全的

hashTable的实现形式和hashMap差不多，它是线程安全的，是继承了Dictionary，也是key-value的模式，但是其key不能为null。

ConcurrentHashMap是JUC并发包的一种，在hashMap的基础上做了修改，因为hashmap其实是线程不安全的，那在并发情况下使用hashTable嘛，但是hashTable是全程加锁的，性能不好，所以采用分段的思想，把原本的一个数组分成默认16段，就可以最多容纳16个线程并发操作，16个段叫做Segment，是基于ReetrantLock来实现的

说说你了解的hashmap吧

hashMap是Map的结构，内部用了数组+链表的方式，在1.8后，当链表长度达到8的时候，会变成红黑树，这样子就可以把查询的复杂度变成O（nlogn）了，默认负载因子是0.75，为什么是0.75呢?

我们知道当负载因子太小，就很容易触发扩容，如果负载因子太大就容易出现碰撞。所以这个是空间和时间的一个均衡点，在1.8的hashmap介绍中，就有描述了，貌似是0.75的负载因子中，能让随机hash更加满足0.5的泊松分布。

除此之外，1.7的时候是头插法，1.8后就变成了尾插法，主要是为了解决rehash出现的死循环问题，而且1.7的时候是先扩容后插入，1.8则是先插入后扩容(为什么？正常来说，如果先插入，就有可能节点变为树化，那么是不是多做一次树转化，比1.7要多损耗，个人猜测，因为读写问题，因为hashmap并不是线程安全的，如果说是先扩容，后写入，那么在扩容期间，是访问不到新放入的值的，是不是不太合适，所以会先放入值，这样子在扩容期间，那个值是在的）。

1.7版本的时候用了9次扰动，5次异或，4次位移，减少hash冲突，但是1.8就只用了两次，觉得就足够了一次异或，一次位移。

concurrentHashMap呢

concurrentHashMap是线程安全的map结构，它的核心思想是分段锁。在1.7版本的时候，内部维护了segment数组，默认是16个，segment中有一个table数组（相当于一个segmeng存放着一个hashmap。。。），segment继承了reentrantlock，使用了互斥锁，map的size其实就是segment数组的count和。而在1.8的时候做了一个大改版，废除了segment，采用了cas加synchronize方式来进行分段锁（还有自旋锁的保证），而且节点对象改用了Node不是之前的HashEntity。

Node可以支持链表和红黑树的转化，比如TreeBin就是继承了Node，这样子可以直接用instanceof来区分。1.8的put就很复杂来，会先计算出hash值，然后根据hash值选出Node数组的下标（默认数组是空的，所以一开始put的时候会初始化，指定负载因子是0.75，不可变），判断是否为空，如果为空，则用cas的操作来赋值首节点，如果失败，则因为自旋，会进入非空节点的逻辑，这个时候会用synchronize加锁头节点（保证整条链路锁定）这个时候还会进行二次判断，是否是同一个首节点，在分首节点到底是链表还是树结构，进行遍历判断。

concurrentHashMap的扩容方式

1.7版本的concurrentHashMap是基于了segment的，segment内部维护了HashEntity数组，所以扩容是在这个基础上的，类比hashmap的扩容，

1.8版本的concurrentHashMap扩容方式比较复杂，利用了ForwardingNode,先会根据机器内核数来分配每个线程能分到的busket数，（最小是16），这样子可以做到多线程协助迁移，提升速度。然后根据自己分配的busket数来进行节点转移，如果为空，就放置ForwardingNode，代表已经迁移完成，如果是非空节点（判断是不是ForwardingNode，是就结束了），加锁，链路循环,进行迁移。

hashMap的put方法的过程

判断key是否是null，如果是null对应的hash值就是0，获得hash值过后则进行扰动，（1.7是9次，5次异或，4次位移，1.8是2次），获取到的新hash值找出所在的index，（n-1）&hash，根据下标找到对应的Node/entity，然后遍历链表/红黑树，如果遇到hash值相同且equals相同，则覆盖值，如果不是则新增。如果节点数大于8了，则进行树化（1.8）。完成后，判断当前的长度是否大于阀值，是就扩容（1.7是先扩容在put）。

为什么修改hashcode方法要修改equals

都是map惹的祸，我们知道在map中判断是否是同一个对象的时候，会先判断hash值，在判断equals的，如果我们只是重写了hashcode，没有顺便修改equals，比如Intger，hashcode就是value值，如果我们不改写equals，而是用了Object的equals，那么就是判断两者指针是否一致了，那就会出现valueOf和new出来的对象会对于map而言是两个对象，那就是个问题了

TreeMap了解嘛

TreeMap是Map中的一种很特殊的map，我们知道Map基本是无序的，但是TreeMap是会自动进行排序的，也就是一个有序Map(使用了红黑树来实现），如果设置了Comparator比较器，则会根据比较器来对比两者的大小，如果没有则key需要是Comparable的子类（代码中没有事先check，会直接抛出转化异常，有点坑啊）。

LinkedHashMap了解嘛

LinkedHashMap是HashMap的一种特殊分支，是某种有序的hashMap，和TreeMap是不一样的概念，是用了HashMap+链表的方式来构造的，有两者有序模式：访问有序，插入顺序，插入顺序是一直存在的，因为是调用了hashMap的put方法，并没有重载，但是重载了newNode方法，在这个方法中，会把节点插入链表中，访问有序默认是关闭的，如果打开，则在每次get的时候都会把链表的节点移除掉，放到链表的最后面。这样子就是一个LRU的一种实现方式。

数据结构+算法

TODO（未完待续）

总结

内容过于硬核了，导致很多排版细节，我没办法做得像其他期一样精致了，大家见谅。

涉及的内容和东西太多了，可能很多都是点到为止，也有很多不全的，也有很多错误的点，已经快3W字了，我校验实在困难，我会放在GitHub上面，大家可以跟我一起更新这个文章，造福后人吧。

搞不好下次我需要看的时候，我都得看着这个复习了。

我是敖丙，一个在互联网苟且偷生的工具人。

你知道的越多，你不知道的越多，人才们的 **【三连】** 就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎人才们留言，**你快说句话啊**！

> 文章持续更新，可以微信搜索「 **三太子敖丙** 」第一时间阅读，回复【**资料**】【**面试**】【**简历**】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。