# 亿农批

卖家入驻发布商品，买家登录买货，双方成交实现共赢，包含卖家入驻，卖家买家身份一键切换，卖家买家开通银行虚户，卖家管理店铺发布商品，买家登录购买商品下单等核心功能

Sass多租户平台, 买家,卖家,小程序,运营后台

**后台设计**: 一个集团一个端,有定制化

抽象类实现默认功能,按集团分包,各集团实现继承抽象类,定义各自别名,如有定制化重写方法实现, 通过@Autowired Map<String,ITestService>注入所有实现类,通过buId匹配获取实现类,进行调用

新增集团: 新增包,赋值一份,自定义修改

后期需拆分部署: 复制项目,把冗余集团去掉,京东中台部分,打成jar,拆分后依赖jar即可

**拦截器**: 日志打印, 全局异常处理, traceId处理, 网关参数处理

**异常**: 业务异常, 系统异常, 超时异常, 为了代码中针对不同异常，做信息传递，打点，降级，日志级别等一系列处理

**日志traceId**

MDC 内部封装了ThreadLocal, log打印时从ThreadLocal中获取

编写: 请求头中获取traceId,无创建, 放入MDC.put(MDC\_TRACE\_ID, traceId);

线程池中父子线程间手动传递,

**红包系统:**

**场景一:** 个数限制

生成小红包放入redis list, 抢一个插入数据库一条

一个用户只可抢一个, 用redis set处理

二倍均值法生成所有小红包 Math.random() \* 2(max - min) + min 状态未领取

场景二: 个数不限制

**直接用数据库 ,sql只要大于余额就减 数据库单机支撑qps 200**

update table set count = count -N where count -N > 0

**微服务业务编程难点 数据一致性**

## 难点Redis缓存编写逻辑

**查询流程**：

1. reids中获取 hget(表名+userId%桶数, userId.toString()) 异常log.warn( )
2. 缓存结果为空（穿透处理）|| 异常（log.warn( )） return null
3. 缓存结果为null，查询数据库 return
4. 线程池异步添加缓存--添加失败不处理

hset(表名+userId%桶数, userId, Json串,固定时间+随机时间)) 随机时间为**防止雪崩**

数据库结果为null设置缓存数据为空“”（**穿透处理**）

**数据添加修改**：

datbus平台监控DB数据变动，发送mq，消费端删除缓存数据（mq重试保证肯定能删除）

如果无databus类似监控平台：

线程池异步多次删除未成功，则必须发送到mq成功(失败保存到表，定时发送直到成功)

月付用户表缓存实现：

**userId查询月付用户信息**

接口一：月付内部写流程中查询 直接查主库

接口二：月付外部查询 先查从库，无（未同步完成）-> 查主库

接口三：非强一致场景 查询redis缓存（运营平台）

用户表数据量在达到1亿，选用string内存消耗过大，决定使用hash

经性能测试得出单桶存储100个元素性能最佳，既约100W个桶

**数据类型选型:**

**大value**: redis单线程,value过大会造成阻塞

5种数据类型在使用前要压测,得出最优value大小

使用时根据业务数据量,选择合理数据结构

**key数量过多**：key本身占用+Category前缀，集群模式中还会建立些slot2key的映射关系。

key个数上亿内存消耗十分明显**,** 可使用hash

**缓存穿透**: **查询一个数据库不存在的数据 解決：在缓存中设置为空值**

**缓存击穿: 单个Key过期瞬间高并发 解决: 查DB时加锁**

**方案:** while( true ){

If( get(key) = null ){

加锁成功 -> 查DB return

加锁失败 -> sleep() 压测得出休眠多长时间

**缓存雪崩: 大量Key集中过期,** 缓存数据设置分散的过期时间

**布隆过滤器:**

key通过hash映射到bitmap, 大概率拦截. 如果hash冲突未拦截住, 查库无数据, 则添加缓存key=空

## MQ消费幂等方案

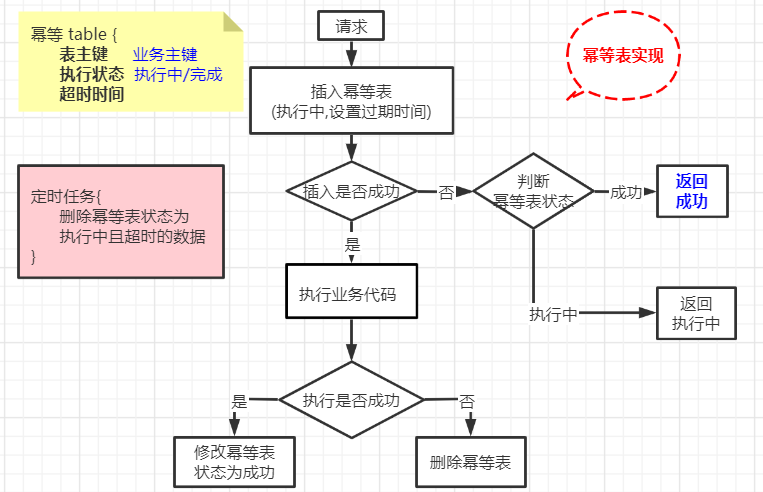
定义: 相同参数多次重复执行, 返回结果相同且与一次执行影响相同

**单insert操作 :**

设置业务主键为表字段唯一索引, 如果重复插入会报唯一异常,处理异常

**多数据源:**

数据库幂等表



## Mq发送可靠消息

**消息发起方:**

**方式一: 数据库事务 + MQ消息发送 + 记录消息表**

关系型数据库事务{

执行本地事务 失败事务回滚

try{

发送MQ消息 失败事务回滚

catch( e ){

try{

记录消息表 定时任务发送消息,保证消息成功发送

}catch( e ){

throw 事务回滚

}

}

}

**方式二: 数据库事务 + 消息服务(封装MQ消息发送 + 记录消息表)**

关系型数据库事务{

执行本地事务 失败事务回滚

发送MQ消息 到 消息服务 失败事务回滚

}

消息表失败消息处理: 定时轮询发送,保证消息发送成功

**消息消费方**: 执行失败可发送回滚消息进行回滚

**分库分表插件**

库表定位：hash，Range时间区间，混合方式定位库、表

SQL解析 => 执行器优化 => SQL路由 => SQL改写 => SQL执行 => 结果归并

**开关降级**：

通常使用配置中心配置开关

代码中判断: 降级打开返回友好提示

接口修改升级, 新逻辑有问题则继续使用老接口

页面配置关闭接口调用入口

**环境隔离:**

预发域名 注册中心加环境别名 调用隔离

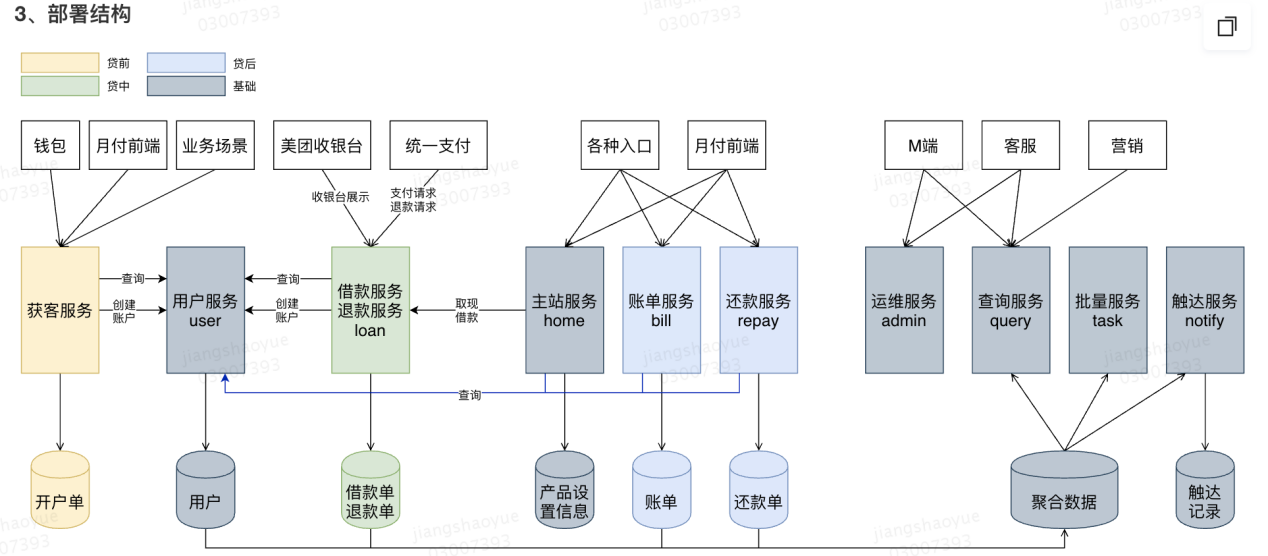
**灰度方案**:

用户风控系统白名单控制新老接口流量切入

**定期压测:**

**故障演练,故障恢复: 监控报警**

# 美团月付



先借后还的金融产品

贷前：获客、用户

贷中：借款、退款

贷后：账单、还款、代扣

基础：任务、触达、主站、通用查询、运维工具

**获客服务：**

**标准开户流程:**

1. 申请开通：

[用户画像系统] 判定，异常降级为通过

[风控系统] 白名单校验

[合同系统] 查询合同版本

[客户系统] 查询实名信息

生成流水号 (时间戳 + ip + AtomicLong%100000

开户申请流水表open\_account\_trade(记录每一步校验结果或异常)

1. [钱包认证] 二要素认证，通过则记录校验信息，异常修改申请表状态、记录异常码
2. 人脸识别校验
3. [管控系统] 校验用户是否被管控
4. [风控系统] 授信校验，返回授信金额
5. [合同系统] 合同版本查询
6. [风控系统] 提交开户审批-> mq 开户状态：审核中

通过消息日志表记录消息，持久化成功提交mq，定时任务保证消息可发送成功

前端轮询查询开户状态

风控审批mq消费： 风控通过thrift -> user服务开户

**用户服务：**

**服务职责**：开户，销户，月付用户信息查询（查主，先从后主，缓存）

**服务特点**：流量大，读多写少，功能稳定

**开户流程：**

开户流水状态： 1风控通过，待开户

2开户失败

3绑定产品关系成功，待创建账户

4创建账户成功，待创建额度

5创建额度成功，待签合同

6创建合同成功，待创建交易账户

7开户成功

redis redlock锁客户号

saga:

创建开户流水表，[客户系统]查询实名，[用户中心]查询手机号 客户未实名**回滚**

绑定客户关系 客户不存在 **回滚**

签署合同 校验不通过 **回滚**

创建财务账户

创建额度

持久化用户表

资方审批

**注销流程:**

1. 校验 ：支付中订单、未入账订单、未结清账单

2. 修改用户状态为销户中（作用：禁止用户支付操作） 失败--> 本地事务回滚

3. 持久化销户表 失败--> 本地事务回滚

3. 二次校验 ：

失败 --> throw 本地事务回滚

成功 --> 异步销户saga{ 重试 必须销户成功

账务系统销户 幂等

额度系统销户 幂等

客户系统销户 幂等

修改用户表状态为销户成功

}

return 状态为销户中。

前端轮询查询销户结果接口<-->

**借款服务：**

**服务职责**：普通、分期、延期、取现功能、各入口月付可用查询

**服务特点**：流量大，读多写少，功能稳定

借款状态：1处理中2成功 3失败4待冻结额度5待记账6待扣减额度7待资金划拨

1. 持久化借款单

2. 线程池并行校验:

@Autowired List<抽象校验器> -> Map

校验器处理不同渠道 如:标准收银台/前置收银台/极速/先享/轻付等

未开户 白名单

已开户 账户，额度，黑名单

卷是否有效

校验通过，未开户 -> 用户服务开户

3. saga 借款：

冻结卷 失败**回滚**

冻结额度 额度不足**回滚**

查询出资比例

转账资金划拨 ->借款成功 mq账单服务记账

**记账：**

mq消费

saga记账 -> 重试

分期、普通入账

绑定卷

解冻并减扣额度

发放零花

**支付故障盘复**

**收到监控报警**：支付冻结额度后，查询出资比例，转账资金划拨系统异常

**定位问题**： 联系资金系统，确认问题为DB故障

**止损**： DBA确认故障无法快速恢复，需要降级月付，手动降级关闭月付入口

**故障解除**： 资金系统故障解除，打开月付入口

**故障数据修复**：采用后台人工补偿

用户用其他方式支付完成 -> 退款

对用户补偿5元卷，用户数据通过saga日志获取

# 中信云平台

**适配器服务：**

**业务描述**：

中信云平台提供各云服务供应商产品，服务物理机操作还是由个供应商提供。适配器服务对个各供应商api，如购买、变配、续费、取消、按流量计费产品查询等

适配器服务既对接各云商http接口

1.请求参数抽象封装，定义抽象父类，公共参数：供应商、服务、行为、url相关。继承抽象类定义操作行为私有参数，如阿里云ecs购买、变配、查询等行为，实现易扩展

2.针对云商不同封装http

启动：

@Autowired

List<AbstractRequest> requestList; 注入所有子类（多例）

Map.put(供应商+服务+行为 ，.class)

调用流程：

1.请求数据处理

请求对象：Request {

流水号 (用于幂等)

供应商别名

服务别名

行为别名

业务参数: {用map接收}

AbstractRequest{封装属性，url，处理类等} }

自定义注解切面，通过 供应商+服务+行为，工厂中获取.class，spring中获取bean，通过反射把业务参数设置进实例

2.获取供应商对应http的封装，调用

线程、ThreadLocal、线程池、volatilte、CAS、synchronized、AQS

jvm类加载、区域划分 对象搜索、垃圾回收算法、垃圾回收器、GC调优常识

io模型、Netty Reactor线程模型

spring IOC、AOP源码

springboot、spring cloud openfeign+ribbon+hystrx、dubbo spi RPC流程

eureka、nacos、zk zab选举

Redis 为什么快 数据结构 回收策略 过期策略 持久化 集群 代码编写

MySQL 慢查询 事务 主从复制 分库分表 Leaf

Rocketmq nameserver

guava令牌桶限流，sentnel

幂等

分布式锁 redis zk

分布式事务解决方案 2/3PC TCC 可靠消息 rodketmq半事务 saga

红包系统

秒杀方案

月付

中信云

# 设计模式

**java三大特性:**

**封装：**实例数据私有化，对外部隐藏不可见，高内聚低耦合

**继承:** 父类代码重用,子类可重写父类方法

**多态：**编译看类型，运行找对象。父类型引用指向子类对象

面向对象设计可复用、可扩展的经验总结

**单例模式**

**双层检查锁**

public class DoubleCheckedLocking {

private static volatile Instance instance;

public static Instance getInstance(){

**if(instance == null){ //1**

synchronized (DoubleCheckedLocking.class){

**if(instance == null){ //2**

instance = new Instance();

}

为什么添加volatile禁止指令重排?

第一个线程获取锁创建对象,创建对象不是原子操作

对象初始化步骤: 1.分配对象内存空间

2.初始化对象

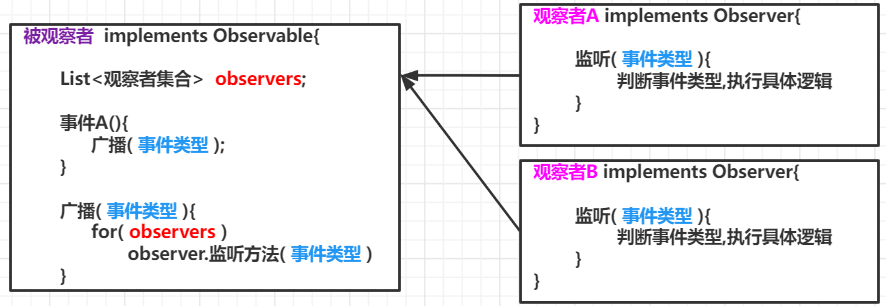
3.设置索引指向内存地址

如果3 2指令重排,内存地址赋给索引,构造器还未完成初始化,后续线程读到if(in==null)会获取一个不完整的对象

**内部类初始化 :** jvm保证一个类只有一个class实例

**枚举写法 :** 枚举实例化,会为每个属性生成一个对象,且禁止反射

**观察者模式**



Spring中为了扩展 🡪 事件, 广播器单独抽象为类

**策略模式**

中信适配器, 根据参数中供应商+服务+行为, 确定哪个类来执行

**适配器模式**

如银行卡支付, 支付接口 -> 适配器 -> 银行支付接口

**动态代理**

**JDK代理**:

代理类 实现 **InvocationHandler**{

getProxy(){ 基于目标类接口生成代理对象 }

重写invoke(){

…………………………………………

反射调用目标类方法

…………………………………………

}

}

**生成的代理对象具有与接口同名方法,内部反射**

**CGLIB** 使用的继承机制, 通过修改字节码生成子类

**责任链模式**

sentinel中 创建调用链路树->创建调用链路数来源->统计->限流计算->降级处理

项目启动 spi读取类配置顺序,按顺序设置类next属性,组成一个单链表

从链表第一个节点entry方法调用,方法最后调用next.entry()

# jdk容器

**HashMap**

**数组+链表+红黑树**

put() :

1.判断是否生成数组,未生成(调用扩容方法)

2.hash数组下标= (len-1)&(key.hashCode())^(h >>>16) >>>二进制右移几位,即除以16

3.判断下标上: 空/链表/树

🡪空 : 生成链表并插入

🡪链表: 判断链表长度>8 &元素个数>树形化阀值64转树,key.equals,有覆盖无插入

🡪树 : 判断key.equals(k)有覆盖无插入

4.判断元素个数/数组长度>=加载因子,大于扩容

**自定义初始化容量**

初始长度为16,扩容时都会是2的次方 :100(元素个数)/0.75(加载因子)=133 2幂上取 ->256

1.7头插入扩容后有可能造成链死循环 1.8改为尾插入

**LinkedHashMap**

实现：双向链表(维护key插入顺序) + extend HashMap

key按插入顺序有序的HashMap,保证了元素迭代的顺序但增加了时间和空间上的开销

**磁盘加载时为解决内存不够用采用了 分页装入**

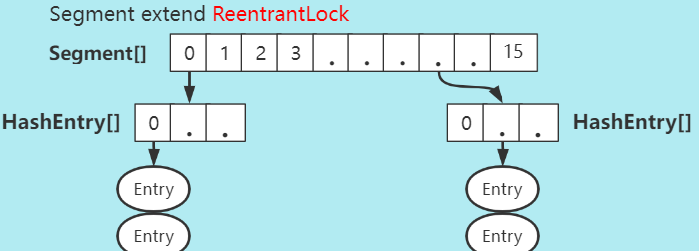
内存分成固定大小的页框(4k),把硬盘上程序分成4K大小的块,用到哪块加载那块

加载过程中,如果内存已满, 会把最不常用的一块放到swap分区(磁盘), 把最新的一块加载进来. 找**最不常用的块算法LRU(**时间复杂度O(1))

**LRU思路:**  LinkedHashMap -> 双向链表(维护块被操作顺序) + extend HashMap

ConcurrentHashMap

**1.7 分段锁**



Segment[]默认长度16即16个并发, 相当于16个HashMap

**1.8 CAS + synchronized** **红黑树**: 优化链表过⻓导致的性能问题

**Put:**

for( 死循环 ){

1. 判断是否生成数组,未生成则初始化数组**创建volatile Node[]**

2.hash获取数组下标

3.判断下标上: 空/链表/树

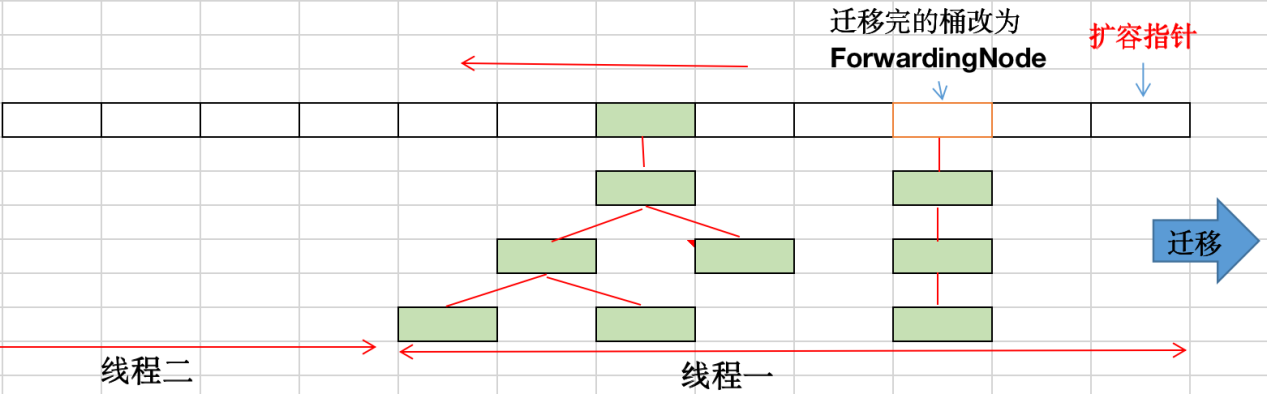
🡪空 : **Unsafe添加节点到数组**

🡪**桶头节点hash值为-1既ForwardingNode - 帮助扩容**

🡪链表: **synchronized（桶头结点**）

🡪树 : **synchronized（桶头结点**）

**多线程扩容：**



1. 根据cpu核数算出步长，最小为16

2. 扩容线程一获取数组长度，**从数组最右端开始**，cas修改扩容数组索引,**向左移步长个桶**，进行迁移，

3. 线程二put/remove，hash key的数组下标节点为ForwardingNode则表示正在迁移，参与扩容，桶被全部分配或已到最大线程数，则不参与

**Get:** 如果正在扩容，老map桶头节点为ForwardingNode则表示此桶已经迁移完成，可通过find方法从新map中获取

**ConcurrentSkipListMap**

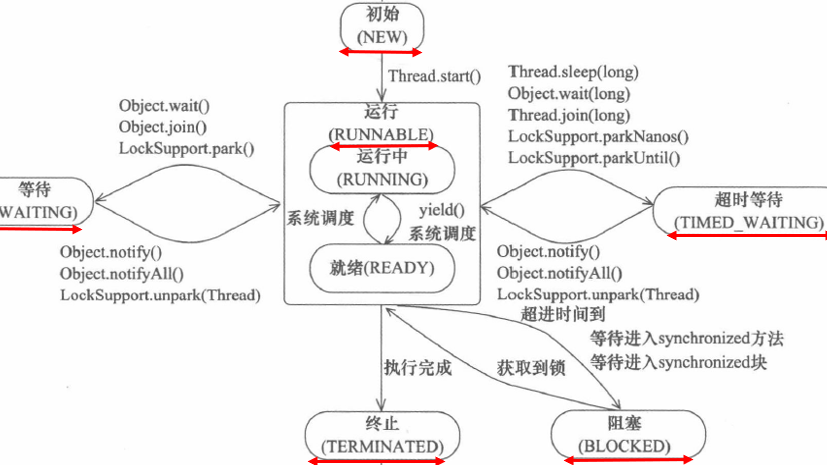
extend SortedMap(key有序接口)

用跳表（SkipList）实现

# 并发编程:

**线程**: 在linux中就是一个普通进程,共享进程内存空间,全局数据等.

**线程状态:**



**线程创建**

**继承Thread类**

**实现Runnable** 目的: 实现类可继承，实现其他类和接口

**实现Callable** 目的：获取线程返回值和异常

Callable中执行方法为call(), 通过包装接口RunnableFuture, 进行执行

Runnable Futrure

| |

RunnableFuture

|

实现FutureTask { **run**(){ **call()** } }

FutureTask{

run(){

try {

result = c.call();

正常运行完成，结果放入 Object outcome

唤醒调用线程 LockSupport.unpark(t);

} catch (Throwable ex) {

如果异常，捕获异常放入Object outcome

唤醒调用线程 LockSupport.unpark(t);

}

}

get(){ 状态非完成

LockSupport.park(t); 等待唤醒

结果正常，return v;

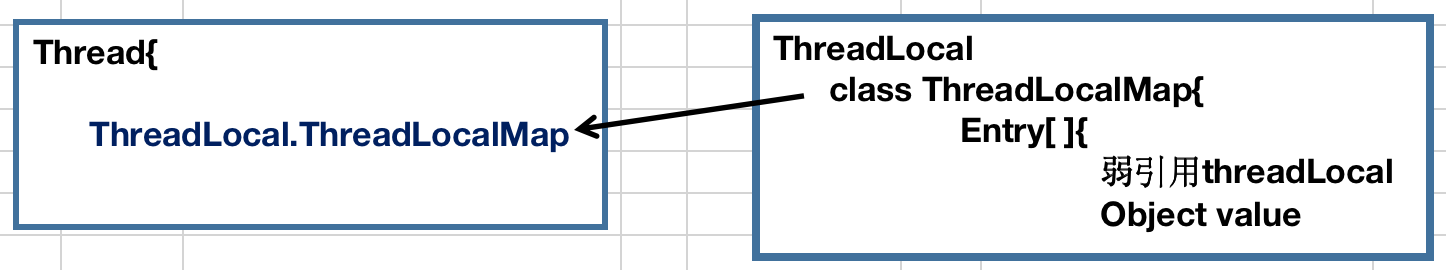
结果异常，throws

}

}

**ThreadLocal**

本质是为线程赋值局部变量, 可扩容数组+线性查找实现,步长1



**内存泄漏问题:**

ThreadLocalMap生命周期同Thread

线程池场景，线程会被复用，业务任务执行完成，需手动清理任务在ThreadLocalMap的Entry[ ]数组中对应的Entry=null Entry.value=null。如果不清理则造成内存浪费

为防止开发人员忘记清理, Entry.key使用弱引用,

**任务执行完成，**弱引用被GC回收，其他ThreadLocal set(),get() 操作ThreadLocalMap时，会Entry.key=null判断 -> 清除entry，使下次被回收

**线程池**

**线程并行目的**: 最大程度利用CPU的多核性能，缩短处理时间

**线程池目的:**  利用已有线程减少线程创建销毁时间，线程可管理

**基本属性：**

HashSet<Worker> workers 存放工作线程

**构造函数:**

public ThreadPoolExecutor(

int corePoolSize, 核心worker线程数量

int maximumPoolSize, 最大worker线程数量

long keepAliveTime, 默认worker数大于核心线程数,队列获取超时,此worker线程退出, 也可设置核心线程获取超时退出

TimeUnit unit, 时间单位

BlockingQueue<Runnable> workQueue, 阻塞队列,注意设置队列大小

ThreadFactory threadFactory, 有默认工厂,一般自己实现,自定义线程名称

RejectedExecutionHandler handler) {} 默认4种拒绝策略,可自己实现

**添加过程:**

1. 当前活动线程数worker < 核心线程数，创建worker，放入HashSet，添加成功返回
2. 当前活动线程数worker >=核心线程数，添加任务到阻塞队列。添加成功返回
3. 添加队列失败，活动线程数<最大线程数，创建worker对象，失败走拒绝策略

**worker线程获取任务**：

线程中断，线程池停止 - > 退出结束worker线程

worker线程数小于核心线程数，take()获取，直到获取任务后执行

worker线程数大于核心线程数，poll(time)超时获取，获取超时线程退出

拒绝策略:

父线程中执行

1. 抛异常
2. 丢弃任务
3. 父线程执行任务
4. 移除阻塞队列头节点,添加新任务
5. 自定义

**线程数：**

压测得出结果, 生产上动态调整

线程执行: CPU运算 + CPU切换线程 + 网络/磁盘io( DMA芯片执行,不消耗cpu)

公式: 线程数 = 内核数 \* 期望CPU利用率 \* (1 + io等待时间/cpu计算时间)

**volatile**

volatile保证可见性和有序性，对单个volatile变量的读/写具有原子性

**字节码层面**: ACC\_VOLATILE

**JVM层面**: 规定8个hanppens-before原则 和 4个内存屏障

规定在读和写前后都加屏障,使用lock指令,具体由不同的CPU实现

**OS和硬件层面**: lock 指令实现 | MESI缓存一致性协议实现

**CAS**

比较和替换，预期值=内存值 则修改,否则不修改

unsafe.compareAndSwapInt(操作对象引用, 属性, 预期值, 修改值)

使用CPU锁,比syncronized快,一般配合循环,具体实现参照atomic包

**Synchronized**

synchronized(obj){} 修饰代码块：代码编译后生成monitorenter、moniterexit指令

synchronized m{} 修饰方法时：对方法flag 上标记 ACC\_SYNCHRONIZED

通过对象头MarkWord记录 偏向标记，锁标记 来实现锁分类与升级

**偏向锁**

场景：共享对象生命周期内，只被同一个线程使用

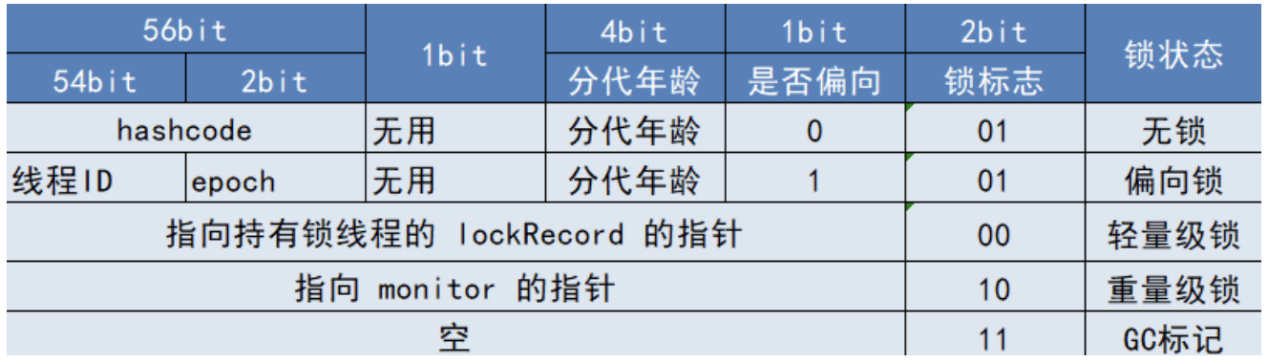
实现: cas线程id到对象头中

**轻量级锁 :**

场景：共享对象生命周期内，会被多个线程交替访问，不出现抢夺

实现: cas不失败, 失败升级

**重量级锁**: 锁对象同步



**AQS**

synchronized是jvm层面提供的非公平锁

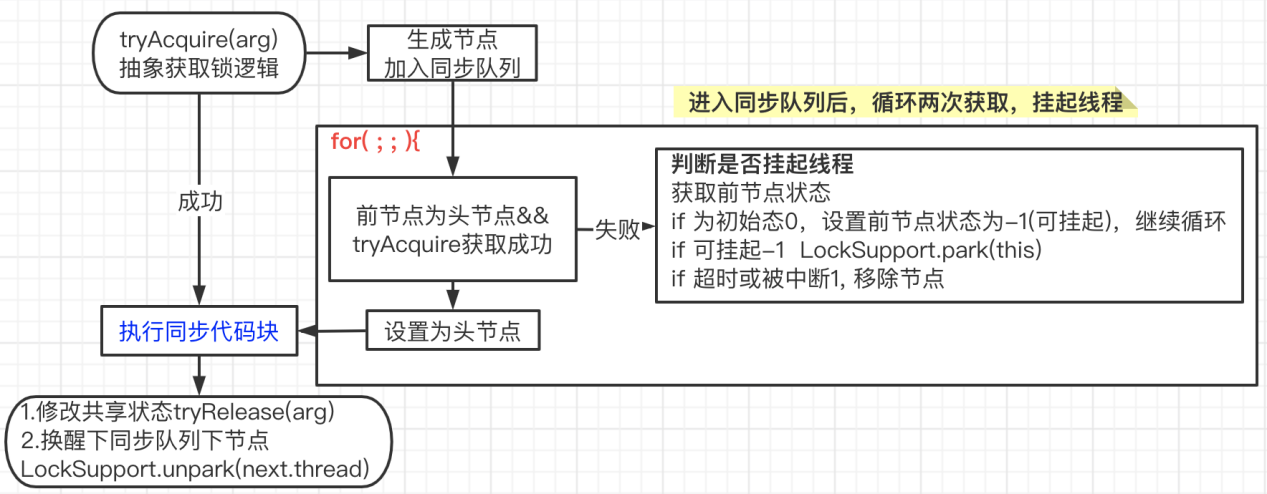
AQS是JDK封装的同步器模板, 通过**循环cas更新volatile共享变量** + **同步队列**实现锁

可实现 公/非平，可中断，可超时，可读共享功能

锁 + condition阻塞队列 实现阻塞队列

**独占模式**

同一时刻只允许一个线程获取锁，实现类需实现tryAcquire(1)，编写获取锁逻辑



**ReentrantLock**

支持公平、非公平，可重入的独占锁

**公平锁**：tryAcquire(1){

if**共享状态state==0** && **同步队列为空** && cas一次状态为1

成功->获取锁

if**是否为重入**（当前获取锁的线程同执行线程）

cas状态+1 -> 同线程重入

else

进入同步队列

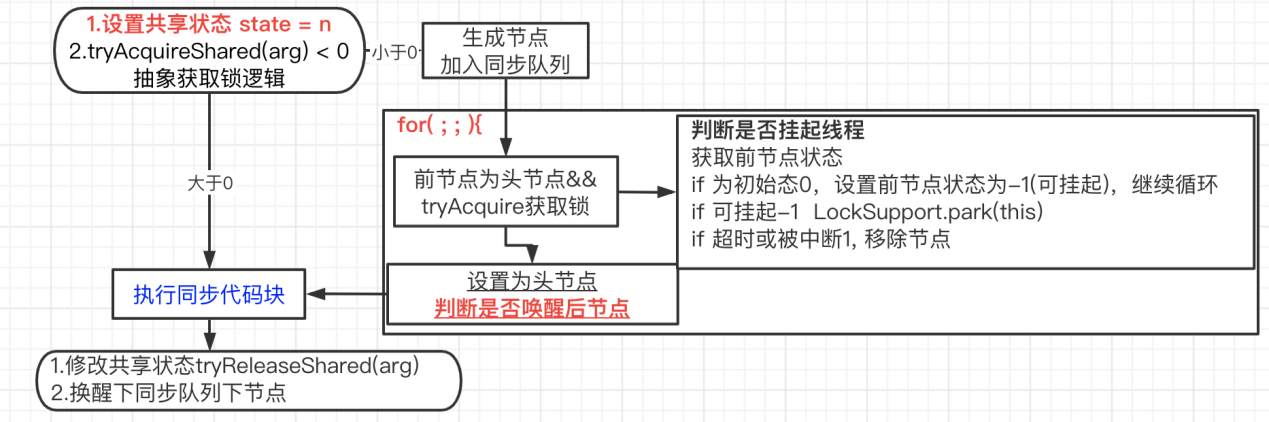
**非公平锁**：进tryAcquire前cas一次，进tryAcquire后cas一次，进入同步队列同公平锁

**共享模式:**

控制同时访问特定资源的线程数量，做流量控制，如数据库连接池。

与独占模式实现区别：1. 需先设置共享变量值，既资源数。

1. 共享模式会被多个线程同时获取。
2. tryAcquireShared返回值为数值，用于逻辑判断



**Semaphore**

**公平锁实现 非公平锁实现为:去掉了队列不为空的判断**

tryAcquireShared(int acquires) {

for ( ; ; ) {

获取共享变量state

if 同步队列不为空 return -1; 负数,加入队列

If state-资源获取数<0 return 负数,加入队列

If state-资源获取数>0 && cas成功 正数,获取锁

tryReleaseShared(int releases){

for ( ; ; ) {

获取共享变量state

cas state+资源获取数

**CountDownLatch**

**调用方式：**

CountDownLatch downLatch = new CountDownLatch(2);

new Thread(() -> {

downLatch.countDown(); //调用tryReleaseShared(1){CAS state-1}

}).start();

downLatch.await(); 主线程加入同步队列，只有这一个线程访问锁，既为头对象，也不会 被挂起，循环获取条件state==0，向下执行

tryAcquireShared(int acquires) {

return (getState() == 0) ? 1 : -1; 获取锁条件state==0

}

**响应中断**

**增加了可中断功能 interrupt()** 中断此线程

一： wait() sleep() join()时 中断状态将被清除， 抛出InterruptedException

二: LockSupport.park()时会唤醒线程

三: InterruptibleChannel上的I / O操作，则该通道将被关闭，线程的中断状态将被设置，抛出ClosedByInterruptException

以上不成立，设置中断状态,不影响线程继续执行

boolean interrupted() 中断返回true,未中断返回false,并清除中断状态既设置为false

boolean isInterrupted() 中断返回true,未中断返回false

**代码逻辑：**

if (Thread.interrupted()) throw new InterruptedException();

if (!tryAcquire(arg)){

同**acquire(1)，挂起后判断中断状态，中断->抛**InterruptedException

}

**例如: AQS中断处理**

if (Thread.interrupted())

throw new InterruptedException();

**线程池中处理**:

while (task = getTask() != null) { 获取队列中任务

Thread.interrupted() 判断线程是否被中断

**响应中断,超时**

**在中断功能上，增加超时处理**

**两次for cas 节点需挂起时，**

**剩余时间：大于1秒->按剩余时间挂起LockSupport.parkNanos(this, nanosTimeout)**

**小于1秒继续循环，剩余时间小于0，返回false**

**Condition阻塞队列**

LinkedBlockingDeque{

**数据链表**

**ReentrantLock lock**

**Condition notFull** **生产等待队列**

**Condition notEmpty** **消费等待队列**

put(E e){

获取锁后： 即同步队列头节点

If 容器未满

添加到**数据容器**，++count

唤醒**消费等待队列**头节点并移动到锁的**同步队列尾**

else 容器已满

从锁的**同步队列头**移动到**生产等待队列**尾部

lock.unlock(); ->解锁

}

take(){

获取锁后：

If **数据容器**不为null

移除容器元素, --count

唤醒**生产等待队列**头节点并移动到锁的**同步队列尾**

else

从锁的**同步队列**头移动到**消费等待队列**尾部

lock.unlock(); ->解锁

}

**Atomic原子操作类**

本质都是Unsafe和一个volatile变量的包装类,Unsafe有3种cas操作

Unsafe{

compareAndSwapObject(Object var1, long var2, Object var4, Object var5);

compareAndSwapInt(Object var1, long var2, int var4, int var5);

compareAndSwapLong(Object var1, long var2, long var4, long var6);

}

**原子更新基本类型**: volatile+循环cas

原子更新带有版本号的引用类型 解决ABA问题

# JVM

**类加载器:**

**启动类加载器** jre\lib目录下jar

**扩展类加载器** lib\ext目录下jar

**系统类加载器** \ClassPath下

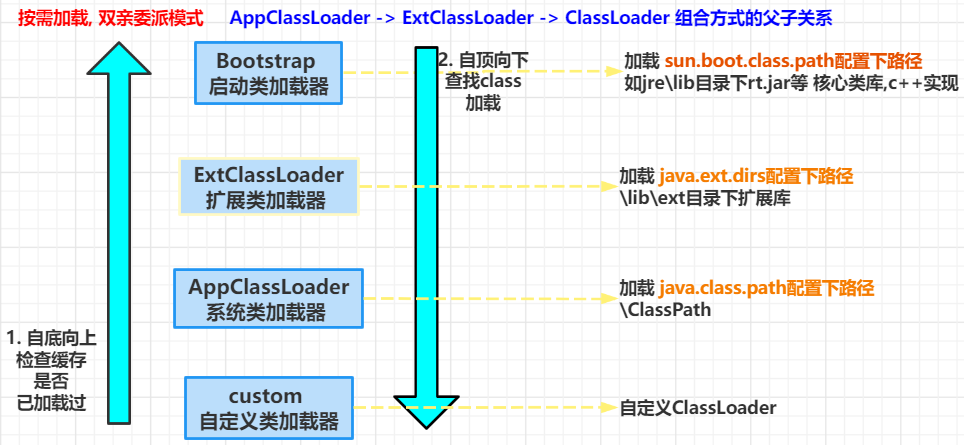
**线程上下文类加载器**

场景: 解决父加载器不可使用子类加载器的问题

原理：Thead{ 类中属性

ClassLoader contextClassLoader 默认为AppClassLoader系统类加载器

如：数据库驱动加载类DriverManager是由Bootstrap类加载器加载的，Bootstrap类加载器只能加载JAVA\_HOME下lib目录下的文件，但实现类是服务商提供的，由AppClassLoader加载。可通过线程上下文类加载器来加载



**双亲委派模式:**

从子加载器到父加载器缓存一次查找, 有返回, 到顶层加载器缓存还没有

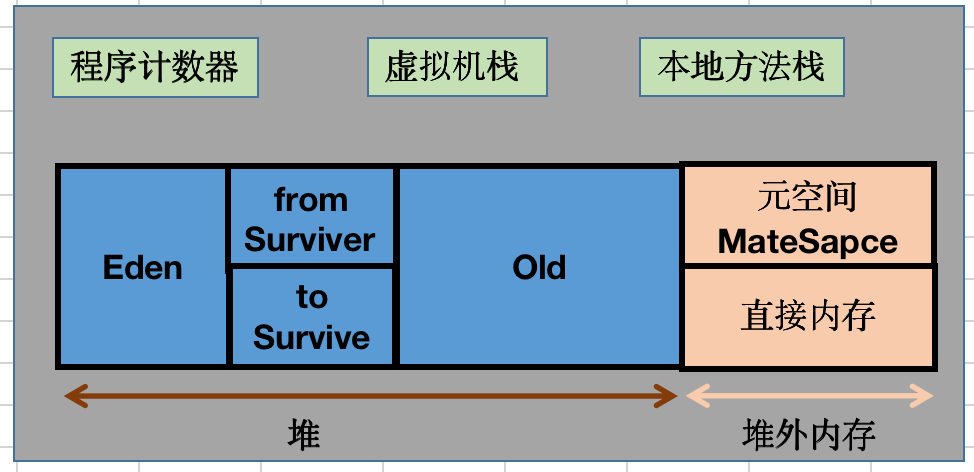
从顶层向下依次加载, 加载完成返回, 到底层加载器还未加载完成ClassNotFoundException

**为什么使用双亲委派:**

**安全性 :** 保证jdk核心类库中类不被覆盖.

如密码一般都用String类型,自己写java.lang.String.class且编写数据发邮件逻辑,则会把所有密码获取到. 通过双亲委派,先检查缓存,缓存无,再由上到下查找类并加载,则返回的是jdk中的String

**Jvm区域划分:**



**堆对象搜索算法:**

**引用计数法:**

调用一次累计1,引用完减1,解决循环引用需同步处理，效率低下（Recycler 算法）

**根可达算法:**

从GC Roots对象中引用向下搜索，三色标记法标记

**GC Roots对象**：

1. 方法区中类静态属性、常量池中引用的对象
2. 虚拟机栈中栈帧本地变量表中的引用的对象
3. 本地方法栈中引用的对象

会在安全点记录在OopMap上，以便快速定位根节点

**三色标记法：**

**标记的过程：**

1. 标记前对象全为**白色**

2. GC Roots根对象置为黑色，根对象直接引用的对象置为灰色

3. 遍历灰色对象的所有直接引用，遍历完成，灰色对象本身置为黑色，引用置为灰色。

4. 重复步骤3，直到没有灰色对象为止。

-->结束时，黑色对象存活，白色对象回收。

**并发标记问题：**

**浮动垃圾：黑色与灰色之间引用, 被用户线程删除**，灰色继续扫描下去变为黑色，既垃圾被标为黑色存活了下来，只能下次GC

**对象消失：灰色与白色之间引用, 被用户线程删除, 白色又被黑色引用,** 把原本存活对象错误的标记为已消亡。

**解决对象消失：**

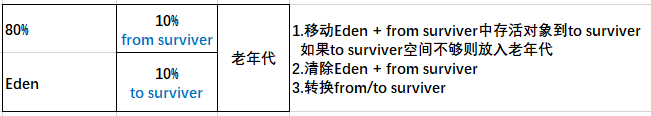
**CMS：写屏障+增量更新。**黑色引用白色时，写屏障AOP切面来记录引用关系，等扫描结 束后，再以引用关系里黑色对象为根重新扫描一次即可

**G1：原始快照。**灰色与白色引用被断开时，记录引用关系快照，等扫描结束后，再以灰色 对象为根以快照扫描

**回收算法**

**标记复制法:**

新生代98%为可回收对象,新生对象先放入eden区,当eden区无法分配内存时,触发垃圾回收



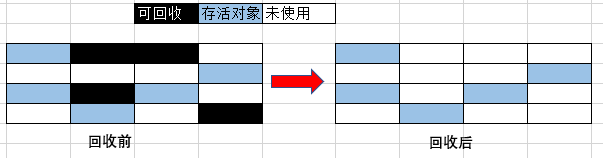
to surviver区空间不足 或 经过15次回收还存活的对象则放入老年代

为什么是15次?因为对象头中用4个bit存储对象年龄,最大15

标记复制算法会浪费空间,在对象存活率高时就要进行较多复制操作,效率会变低

**标记清除法:**

**标记出要清除的对象, 统一回收所有标记的对象**



**问题**:

1.效率低

2.产生内存碎片,分配大对象时可能无法找到足够的连续内存而频繁触发垃圾回收

**标记整理法:**

标记出要清除的对象, 移动存活对象到一端，清除端边界以外的内存

**CMS**

**并行**: 用户线程等待,并行多个垃圾收集线程

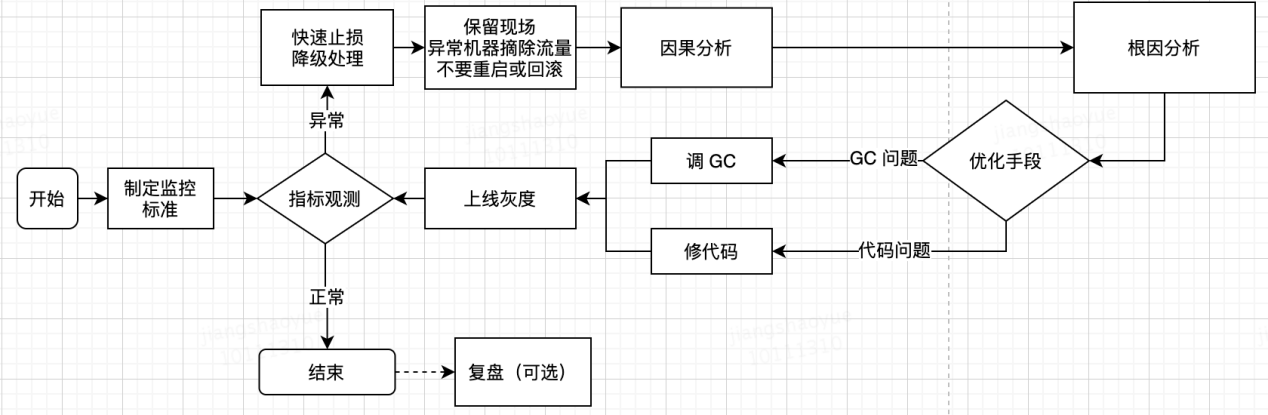
**并发**: 用户线程和垃圾收集线程同时运行

**回收过程**

以获取最短回收停顿时间为目标的收集器

1. 初始标记：**并行**标记GC Root根节点
2. **并发**标记：用户线程/cms gc线程同时运行
3. 最终标记：**并行**标记处理增量（**写屏障+增量更新** 处理对象消失）
4. **并发**清除: 标记清除法。(可配置参数，几次回收压缩一次碎片)

**问题排查流程**



**如果指标都发生异常,需找出根因**:

**时序分析**：通过指标异常发生时间点,时间最早的最大概率是根因

**概率分析**：通过历史经验,由近到远分析 如过往慢查的问题比较多

**实验分析**：通过故障演练等方式对问题现场进行模拟

**反证分析**：从整个集群角度观察,是否发生指标异常

方式一: jmap -dump生产日志文件, 用工具查看, 排查问题

方式二: **命令行或阿里arthas线上排查**

**GC 频繁,耗时增大:**

jstat -gc 动态观察gc情况/阅读GC日志发现频繁GC

**频繁Young GC**

**表象:** [GC (Allocation Failure) -> 分配对象失败

**原因:** 年轻代分配小，对象年龄太大、程序创建对象过快

**设置:** 设定年代大小、升级年龄

**CMS 三种GC类型**

**background** : CMS 并行/并发

后台线程定时扫描，若达到容量阈值，到业务线程安全点，并行+并发CMS GC

**foreground**：

晋升失败 GC日志中出现promotion failed：

Young GC对象晋升失败 --> 停止业务线程，并行CMS GC+元空间+压缩

CMS并发模式失败 GC日志中出现concurrent mode failure

CMS GC并发backgroud回收中，Young GC对象晋升失败 -->

停止业务线程，停止CMS GC并发，退化为serial old，单线程CMS GC+元空间+压缩

**建议**：调小CMS回收阈值，预留充足内存

降低压缩频率 3次Foreground gc压缩一次

设置并行处理Reference，加快处理速度，缩短耗时

最终标记前进行一次Y GC，减少停顿时间

合理设置并发并行线程数

GCLocker Initiated GC：如果线程执行在 JNI 临界区时，刚好需要进行 GC，此时 GC Locker 将会阻止 GC 的发生，同时阻止其他线程进入 JNI 临界区，直到最后一个线程退出临界区时触发一次 GC。

线程 Block增多

**jstack定位线程状态**: 重点关注：WAITING BLOCKED

waiting on <0x0000000088ca3310锁对象>很多线程都waiting

running状态的线程, 即持有锁的线程, 线程池线程工厂自定义线程名称意义所在

**元空间内存溢出**

**[Full GC (Metadata GC Threshold) backgruond 并行/并发CMS**

**表象**：MetaSpace 已使用大小持续增长，每次GC也无法释放

**原因**：不停地加载新Class到元空间，一般发生在动态类加载等情况上

案例：orika框架做对象转化，转化两个不同属性名称时，会通过javassist动态生成转化类.class，生成的类名包含时间戳和自增序列，每次调用多会生成新的类

解决：转化类对象改为静态属性初始化

内存飙高: **jmap** - histo 4655 | head -20，查找生成对象数量

**arthas重要功能:**

**jad反编译** : 动态代理生成类的问题定位

第三方的类（观察代码）

版本问题（确定自己最新提交的版本是不是被使用）

**redefine 热替换**: 动态修改线上代码

目前有些限制条件：只能改方法实现

**Reference**

**finalize()**

类加载时，发现类重写Object的finalize()方法，此方法会被标记为Finalizer类。

对象初始化，生成一个finalizer，持有对象引用。

GC时，对象只被finalizer引用，则放入Finalizer 类属性队列中，FinalizerThread守护线程从队列获取对象，执行finalize( )，守护线程优先级低，finalize()方法中一般为资源释放比较耗时，则在GC后调用，执行中可能已执行了继承GC，调用完后，下次会被回收，所以最少经历两次GC。

Finalizer 是 FinalReference 的唯一实现

**对象软引用**：

创建SoftReference soft = new SoftReference(obj); 获取对象soft.get()

回收条件：内存不足时，对象只有软引用被引用，会被回收

场景： 有用但并非必须的对象，如GuavaCache本地缓存

**对象弱引用**：WeekReference

回收条件：对象只有弱引用被引用，GC就被回收

场景： ThreadLocalMap中Enty引用ThreadLocal的弱引用

**对象虚引用**：new PhantomReference(obj，referenceQueue) 必须使用队列

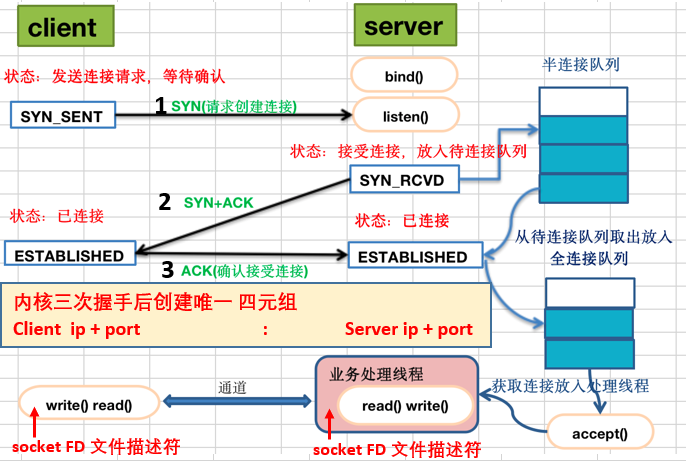
场景： 追踪gc回收对应对象的活动

直接内存的回收就用到虚引用，jvm申请一块直接内存之后，会在堆内存分配一个对象保存这个堆外内存的引用，对象被回收，引用会被放入队列，被系统回收

# IO

**网络通信**

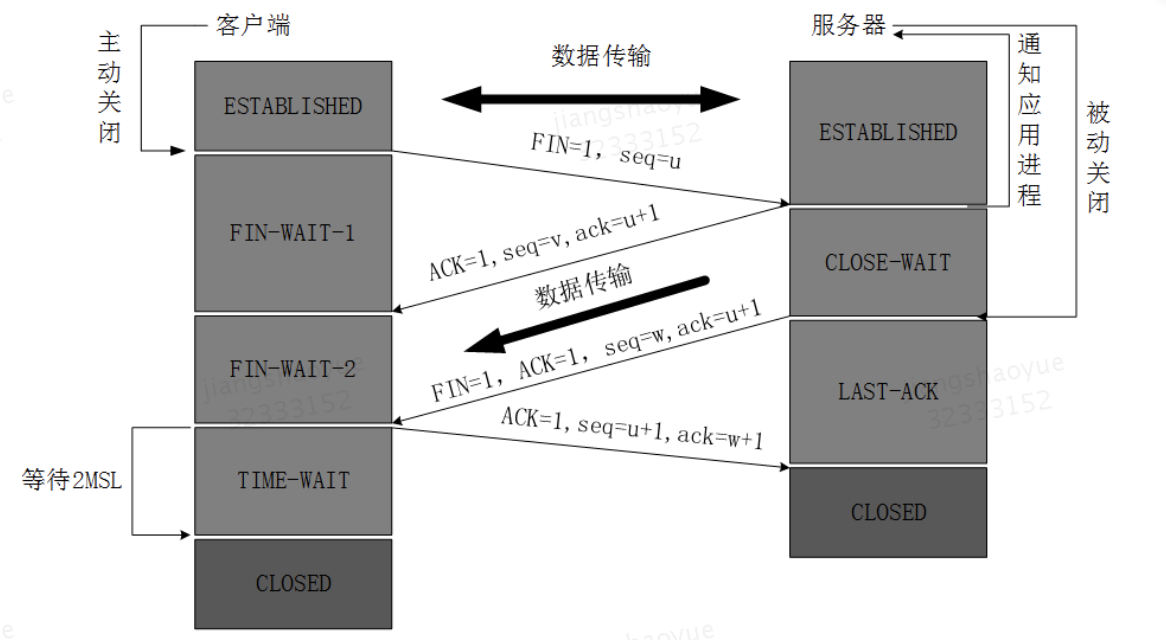
**TCP创建连接:三次握手**



三次握手后创建唯一四元组, 创建socket连接, 即双方机器资源创建,并没有物理连接

服务端业务线程调用accept()后,获取系统资源socket连接的FD文件描述符,进行读写操作

**TCP释放连接:四次分手**



主动关闭方: 发送关闭己方数据传输FIN

被动关闭方: 收到FIN,发送ACK确认

被动关闭方: 发送关闭己方数据传输FIN,状态为CLOSED

主动关闭方: 收到FIN,发送ACK确认,TIME-WAIT状态等待两倍报文时间,才会关闭

**异常情况：CLOSE\_WAIT状态数过多或者持续增长。**

**IO模型**

**技术发展方向:**

**减少业务线程数**: 线程数过多1.耗内存(线程栈约占内存1M,) 2.cpu调度增加,浪费处理时间

解决: 使用非阻塞

**减少系统调用**: IO操作需系统调用,上下文切换极其耗时

**解决**: 多路复用器 select -> poll -> epoll

**减少数据复制**: mmap共享内存, sendfile零拷贝

**同步阻塞BIO:**

同步: 由用户线程系统调用

阻塞:accept()和IO读写**阻塞**用户线程,直到有效数据返回

**业务线程数:** 一个线程处理accept(),N个线程处理N个连接读写

**同步非阻塞NIO:**

**非阻塞:** accept()和IO读写**不阻塞**用户线程,有数据返回,无数据返回-1

**业务线程数:** 一个线程处理accept(),几个线程遍历处理N个连接读写,**线程数减少**

**系统调用次数: 用户线程遍历**连接数据是否准备好**,**每次遍历N次

**多路复用器Selector:**

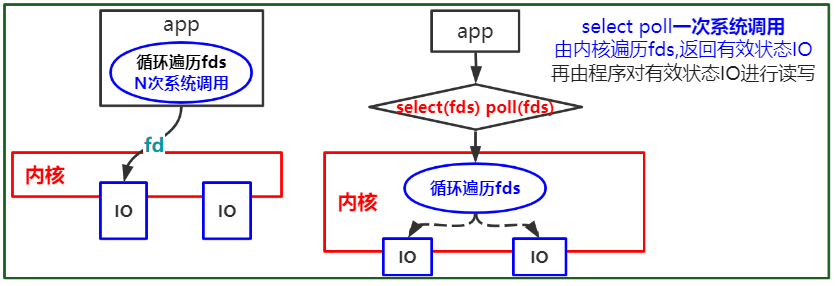
**Select :** 用户线程每次可最多传1024个连接文件描述符fds,由**内核遍历fds**, 返回已有效fd

**poll :** 一次调用fds数量不限制, 其他同select

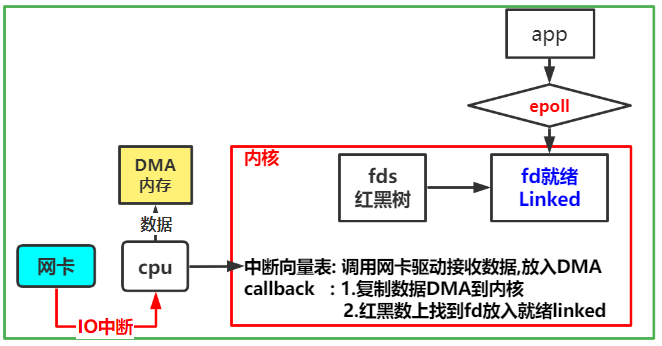
**epoll** **:** 内核创建连接fds红黑树, fd有效事件就绪, fd放入有效链表,查询直接返回

**异 步**:

数据就绪,系统直接把数据写到用户线程,不用户线程调用,linux暂不支持



**epoll** : 网卡中断,CUP复制数据到DMA,中断回调,DMA复制数据到内核,fds红黑树上找到就绪的fd放入就绪linked



**Reactor线程模型**

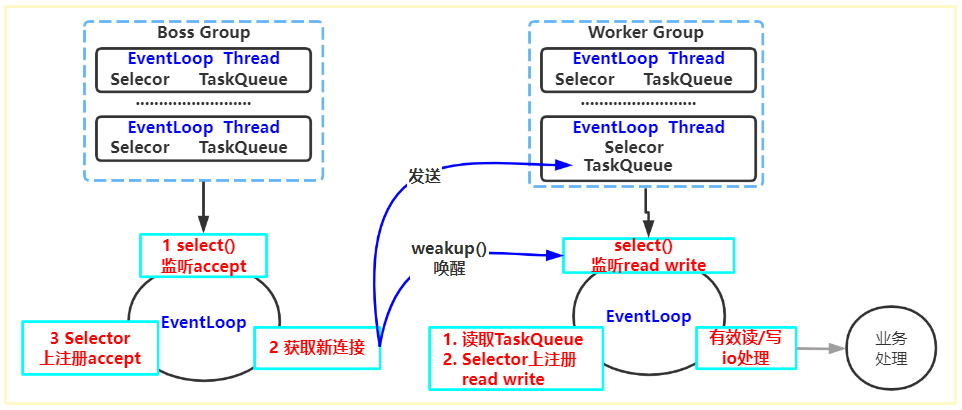
**水平触发问题**:

select()读取到有效读事件,数据全部读完,内核中有效读才变为无效.如果异步处理读,未读完就进行了下一次select(),fd可读事件会被重复监听到

解决: 每次select()中,同步读取完数据, 采用多Selector方式,实现多线程

默认创建cup核数\*2个线程,也可指定,每个线程中都有一个Selector

用一个线程监听accept,监听到后负载均衡放入其他线程注册队列,由其他线程注册连接R W



**一次解码处理拆包/粘包**

TCP是流式协议，消息无边界. 根据滑动窗口大小(变化的)决定包大小

**拆包**: 线程数据包大时,TCP会拆分包多次发送

**粘包**: 线程数据包小时,TCP会多个包合并为一个包发送

解决：找出消息的边界

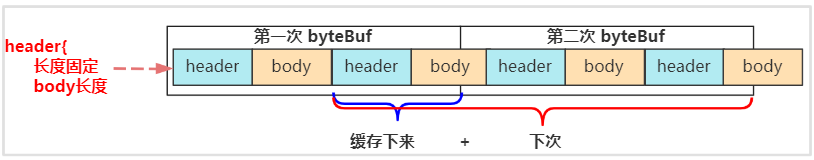
方式一: 线程消息长度固定 缺点: 浪费空间

方式二: 消息尾分隔符 缺点: 需扫描内容,处理转义字符

方式三: 固定长度记录消息长度信息 ,解析固定长度, 得到消息长度

如果buf剩余长度大于消息长度,则移动buf指针读取

如果buf剩余长度小于消息长度,则程序缓存下剩余字节,合并到下一次读buf



**Dubbo协议:**



**二次解码序列化**

二进制反序列化为对象

**零拷贝**

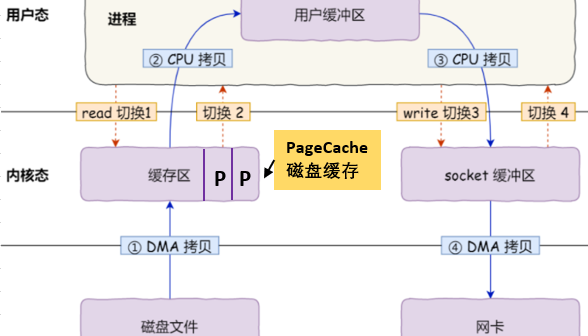
https://mp.weixin.qq.com/s/oPv1-wrhYjiOC1o0M0tjMA

**read write**

**read**(file, tmp\_buf, len); **write**(socket, tmp\_buf, len);

**4 次用户态与内核态的上下文切换**

**2次DMA拷贝, 2次CPU拷贝**



**mmap**

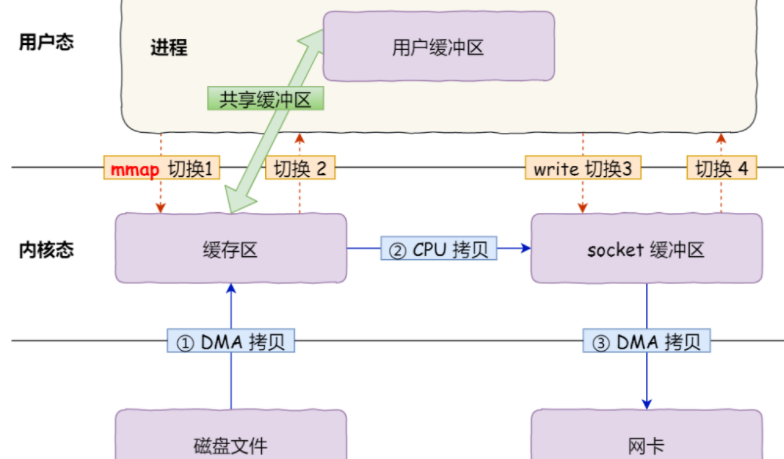
buf = **mmap**(file, len); **write**(sockfd, buf, len);

**4 次用户态与内核态的上下文切换**

**2次DMA拷贝, 1次CPU拷贝**

**mmap()** 也称共享内存,jvm堆外内存

系统调用把内核缓冲区里的数据「映射」到用户空间,用户态直接使用内核缓冲



**sendfile**

**ssize\_t sendfile**( int out\_fd, 目的端文件描述符

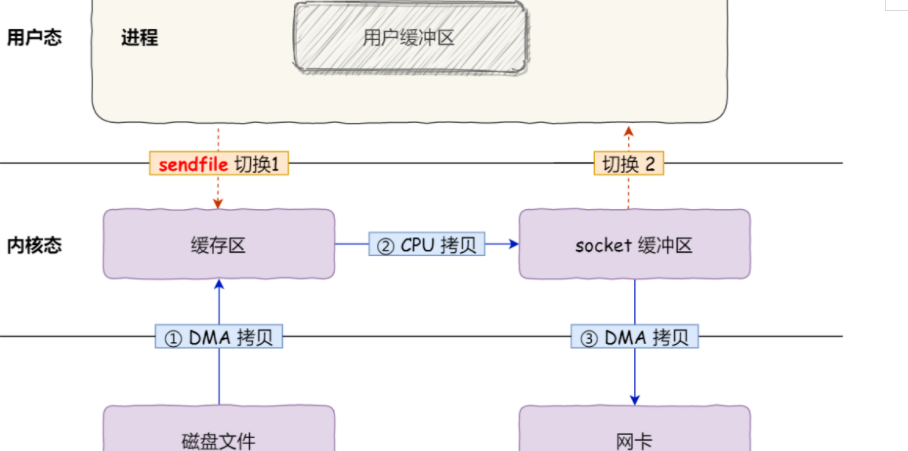
int in\_fd, 源端文件描述符

off\_t \*offset, 源端的偏移量

size\_t count); 复制数据的长度 **ssize\_t出参(实际复制数据的长度)**

**2 次用户态与内核态的上下文切换**

**2次DMA拷贝, 1次CPU拷贝**

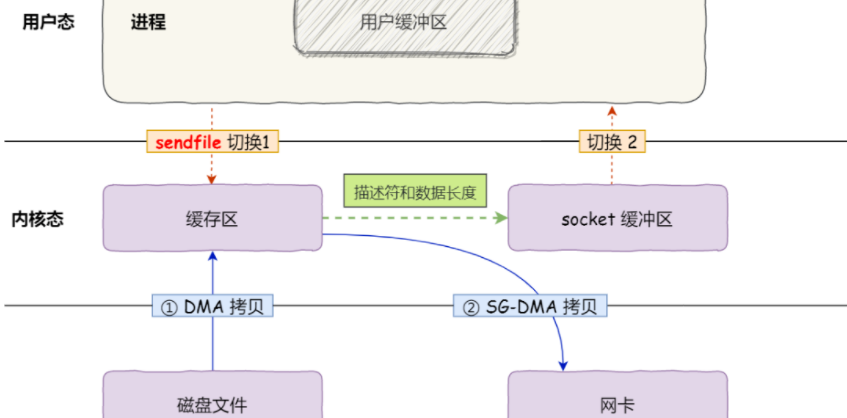


**SG-DMA网卡**

真正零拷贝: 网卡支持 SG-DMA

**2 次用户态与内核态的上下文切换**

**2次DMA拷贝**



# RPC

面向接口编程

**动态代理**生成接口代理类{

**封装信息**: 服务,接口,协议,调用方式(同步,异步,只发送)等

**注册中心**: 获取生产者组ip + 端口

**负载均衡**: 根据策略选择一个生产者

**调用方式**: 生成唯一requestId,放入消息体

同步: 发送 Future/CountDownLatch/单独阻塞队列 方式阻塞业务线程

放入缓存Map<requestId,Future>

响应 根据requestId缓存中获取Future,设置返回值结束阻塞

异步: 发送 不阻塞业务线程,放入缓存Map<requestId,Future>

响应 根据requestId缓存中获取返回值,调用回调方法

sendonly: 直接发送,无响应

}

**负载均衡策略:**

**随机:** Random.nextInt(allList.size())

**线性轮询:** AtomicInteger +1 %allList.size

**权重:** 平均连接时间越短的server,权重越大

## Dubbo

**模块与功能**

**接口层（Service）**：业务接口与实现

**配置层（Config）**：

注册beanDefinition到spring

@DubboService注解处理，服务暴露

@DubboReference注解处理，服务引用

**代理层（Proxy）**：

提供jdk和javassist 两种实现

**注册层（Registry）**：

封装服务地址的注册与发现

**监控层（Monitor）**: 统计服务调用次数，调用时间的，调用链跟踪的服务。

**集群层（Cluster）：**

失败策略，路由，负载均衡

**远程调用层(Protocol):**

封将 RPC 协议 @SPI Protocol

提供生产、消费端调用是过滤器 @SPI Filter

数据封装Invocation 和 Result

**信息交换层（Exchange）**

RpcInvocation -转-> Request

Response -转-> AsyncRpcResult

ExchangeCodec -> Request Response 编解码功能

**网络传输层（Transport）**：

@SPI Dispatcher 线程派发

抽象Client，RemotingServer，Channel 默认netty

**序列化层（Serialize）**

提供协议消息body序列化反序列化，支持fastjson,jdk,gson等 默认Hessian

**Spi**

**JDK SPI**

定义：指定路径下，文件名为接口全限定名，内容为实现类全限定名

加载：路径+接口名获取文件，解析内容获取实现类名，反射生成对象，如jdbc驱动加载

**dubbo-spi**

可默认获取，指定获取，通过URL配置动态获取，AOP，IOC功能

文件内容定义：

别名=实现类全限定名

静态代理类全限定名

**步骤一：创建扩展加载器**

ExtensionLoader{

type = 接口类型

ExtensionFactory创建依赖对象工厂加载器，动态方式获取依赖工厂实现

AdaptiveExtensionFactory 类上使用@Adaptive，重新加载方法{

循环spring和spi哪个可获取到就返回

}

SpringExtensionFactory 依赖对象从spring容器中获取

SpiExtensionFactory 返回接口动态代理类（jdk，javassist），既动态获取， 依赖对象方法需添加@Adaptive注解。代理对象中方法从URL中获取扩 展实现类别名，指定获取

}

**步骤二：加载扩展实例对象**

1. 路径+接口全限定名=文件路径

2. 读文件,获取类名,加载class。AOP和扩展点实现的分别存放

**默认获取**：从@SPI注解中获取默认名，通过.class反射生成实例

**指定获取**：通过别名获取.class反射生成实例

**动态获取**：其方法必须使用@Adaptive，参数必须有URL，会根据接口生成动态代理类

public class 接口$Adaptive implements 接口{

有@Adaptive 方法 method(URL ){

URL参数中获取实现类别名

**指定获取**

}

无@Adaptive方法 method(){ throw 抛异常 }

}

ioc注入：

编写：set方式注入。定义接口类型属性，编写set方法,方法入参为 URL

1. 从依赖对象工厂获取依赖对象
2. 反射注入

aop静态代理：

编写：静态代理类：implements接口，定义接口类型属性，构造器以接口类型为入参

1. 文件中直接配置类全限定名，解析时加载代理类

3. 反射代理类构造方法，生成代理类，返回

## 限流:

**雪崩:**高并发会造成线程资源耗尽，响应变慢。调用方线程无法释放而耗尽，引发级联反应而发生雪崩

保护高并发系统的三把利器：缓存，限流，降级

**限流方式和场景**：

限制总并发数 如数据库连接池，线程池

限制瞬时并发数 nginx的limit\_conn模块，java的Semaphore(10), hystrix ActomicLong计数

时间窗口平均速率Guava的RateLimiter，nginx的limit\_req模块，mq固定频率消费

其他 根据网络连接数、网络流量、CPU、内存负载等限流

**漏桶算法：**

固定大小队列，固定流出频率，流入超过容量则拒绝。可通过MQ实现

**令牌筒算法：**

固定频率生成令牌, 获取令牌则执行, 未获取等待或拒绝

**Guava RateLimit平滑突发限流:**

synchronized () {

**由请求时发放令牌**

当前时间 > 下次发放令牌时间

可发放令牌数 = (当前时间-下次发放令牌时间)/令牌发放间隔

存储令牌数 += 可发放令牌数,超过最大令牌数取最大令牌数

下次发放令牌时间为当前时间

下次发放令牌时间 += 预支令牌数\*发放令牌时间间隔

存储令牌数 -= 本次需获取令牌数(小于0取0)

}

线程需休眠(下次发放令牌时间-当前时间)

## 降级:

通过熔断组件 或 程序中开关,在服务故障前,使程序快速返回,结束请求

**熔断器降级**：如Sentinel，hystrix

**开关降级**： 通常使用配置中心配置开关

如：if ( 开关关闭 ) { 获取评论数据} else { 直接返回空评论数据 }

如: 接口修改升级, 新逻辑有问题则继续使用老接口

如: 页面配置关闭接口调用入口

**Sentinel**

**NodeSelectorSlot :** 构建资源调用链路树

**ClusterBuilderSlot:** 构建资源来源节点

**StatisticSlot: 资源的样本时间窗口统计**(通过数,拒绝数,异常数,成功数,执行时间)

**SystemSlot:**  控制总的入口流量

**AuthoritySlot** 来源黑白名单控制

**FlowSlot** qps/线程数限流

**DegradeSlot** 熔断降级规则校验(**慢调用,异常比,异常数**)

**固定时间窗口弊端**: 在窗口交接点,极短时间可通过单窗口两倍流量

**滑动时间窗口弊端**: 每次统计窗口数据,都需从当前时间前的一个窗口时间段内所有记录进 行统计,性能低

**固定窗口样本分段实现**:

**结构**: 把固定时间窗口分成N个样本窗口,生成N大小数组,每个样本窗口有个计量桶

**统计:**

计算当前时间样本数组下标 = 当前时间 **/** 样本窗口时间 **%** 样本窗口数组长度

计算当前样本窗口起始时间 = 当前时间 - 当前时间 % 样本窗口时间

当前样本窗口起始时间 = 数组下标上样本窗口时间 -> 直接统计

当前样本窗口起始时间 > 数组下标上样本窗口时间 -> 说明下标上样本窗口已过期重新设置起始时间和清空计量桶

**获取**:

遍历样本窗口数组,进行统计

当前时间 - 样本窗口起始时间 > 窗口时间 表示已过期,不参数计算

**限流原理:**

直接失败: 获取当前时间窗口qps或线程数,比较

排队等待: 类似guava RateLimit 令牌桶等待

**降级原理:**

**状态**: 打开,半打开,关闭

熔断器打开后, 到达下一时间窗口状态会变为半打开,试着请求,成功则修改状态为关闭

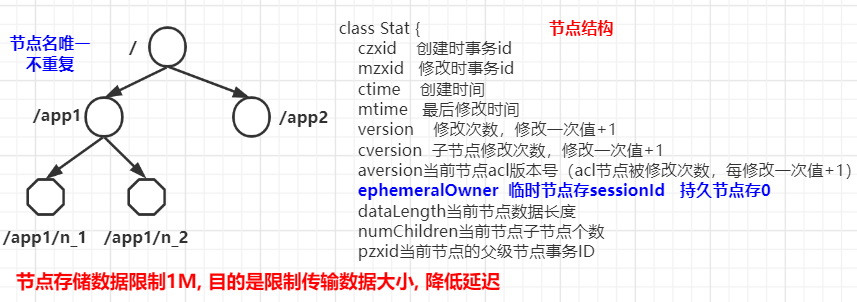
## Zk

**分布式协调服务**:

**高性能**: 内存存储, 读写分离, 读:写 10:1性能最佳

**高可用**: ZAB选举块 200ms,数据持久化

**写有序**: 服务端只有主节点处理写操作,放入阻塞队列再处理,生成唯一zxid



**持久节点** : 持久化到磁盘

**临时节点** : 节点中保存sessionId,当client断开临时节点删除

**序列属性:** 重复创建相同path,会从0递增生成新path名并返回

create -s /app/user [data] /app/user000000000

create -s /app/user [data] /app/user000000001

**Zxid**: 周期 +递增

低32位变化: 连接创建,断开,写操作 +1

高32位变化: 选举完成+1,低32位满后+1 低32位从0开始

**Session**: client与集群第一次创建连接,leader生成sessionId, client:session=1:1, 广播所有节点,

临时节点会保存sessionId, Client默认3秒发心跳,当连接断开,leader根据心跳超时,会清除所有同sessionId的临时节点

**Watch:** client可以在数据节点上设置watch,当节点变更,服务端会向客户端发送事件指令

一次性监控,如想永久监控,需监控后重新设置

**配置中心 :** 永久节点 + watch

**做幂等 :** 临时节点 + watch

**分布式锁 :** 临时有序节点 + watch

**ZAB选举**

**选举触发**: 集群启动, leader宕机

从节点心跳检测到,发起开始选举广播, 可能同时,可能有先后

while( running ){

case LOOKING: 1.设置自身状态LOOKING后执行选举

2.周期++, 本机当前投票给自身, 广播

while( LOOKING )

响应队列中获取

**周期相同** : 远程投票与本机投票的zxid myid比较

远程投票大 -> 更改投票, 广播

本机投票大 -> 不处理

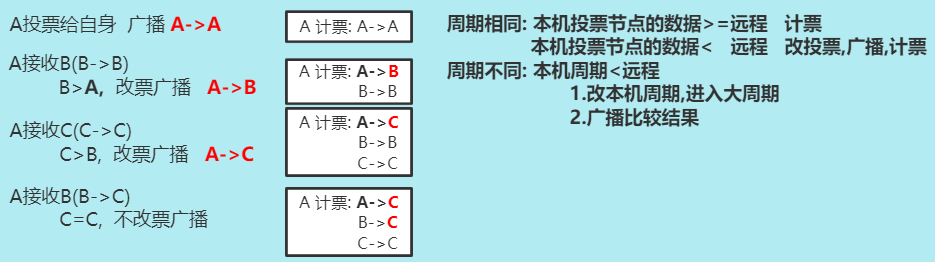
**远程周期大**: 说明本机为刚启动, 清票箱, 改周期, 进入大周期,广播比较结果

添加到投票箱: Map< myid : 机投票信息>

If( 本机当前投票节点在票箱票数过半 ){

超时从响应队列获取 -> 无 选举成功

-> 有 继续循环



**脑裂:** 集群出现网络问题, 各分区选举, 出现多个Leader

解决: 过半选举机制

**ZAB数据最终一致:**

**Leader写处理 + 二阶段提交 + 过半完成 + 定时数据同步 =** 数据最终一致

Leader Follower Observer

一阶段发送数据 接收数据写入日志,返回响应

接收响应过半,Zxid+1,二阶段提交 日志数据-生效

未过半,回滚 日志数据-失效

定时同步数据

**Sync**: 数据最终一致则有可能get旧值, 可在get()前执行sync, 从节点会从主节点拉取数据

## 分布式锁

**锁特性**:

1.同时刻只能被一个获取 zk主节点使用队列保证写有序

2.只有获取锁的人可释放锁 临时有序节点值唯一

3.释放锁后如何通知其他人 节点watch其前一个节点,前节点被删除会被回调,即得锁

4.客户端宕机,造成死锁 临时节点会被删除

5.服务端单点,宕机锁丢失 连接其他server,客户端sessionId不变,数据不丢失

**加锁流程**:

1.持久化锁节点下,创建临时有序节点

2.获取锁节点下所有子节点,从小到大排序

自身为第一个则表示获得锁 return

非第一个,watch监控前一节点删除事件(停止阻塞CountDownLatch. countDown ())

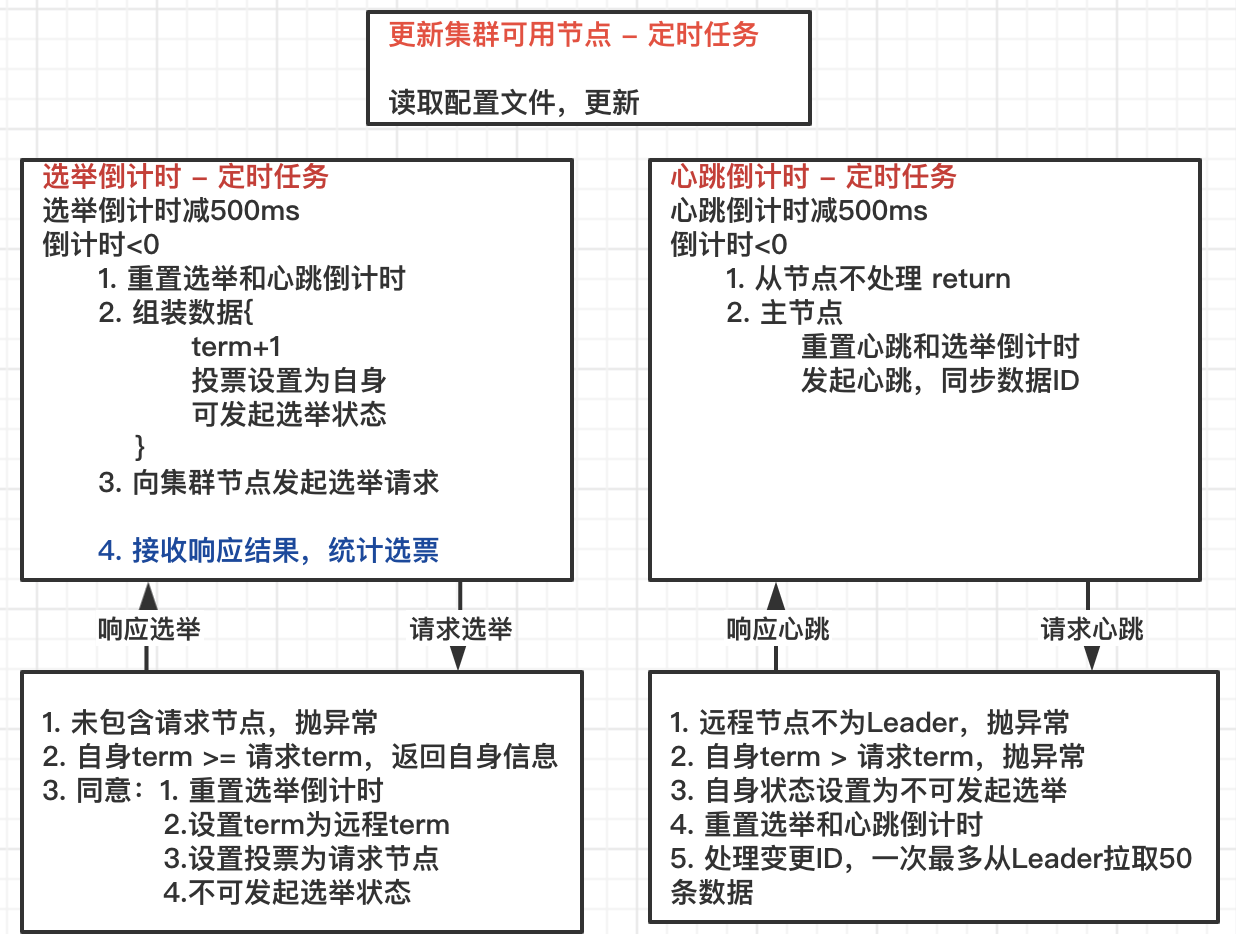
CountDownLatch.wait() 阻塞

**解锁**: 删除临时节点

## Nacos Raft

**角色3 种**：Leader主节点， Candidate可发起选举 , Follower不可以发起选举

**选举周期:** term 与zk不同处为只比较term



## eureka

**集群**: 集群无主从, 各节点存储全部数据, 客户端注册, 节点间进行数据同步, 最终一致

**心跳**: 客户端每30s向集群发送心跳, server节点90s未收到心跳, 判定客户端下线, 注销

**保护机制**:

**原因**: 客户端正常, 由于网络故障, 未接收到客户端心跳, 造成误判下线.

**规则**: 如果在15分钟内超过85%的客户端节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，Eureka Server自动进入自我保护机制

**策略**: 1. Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。

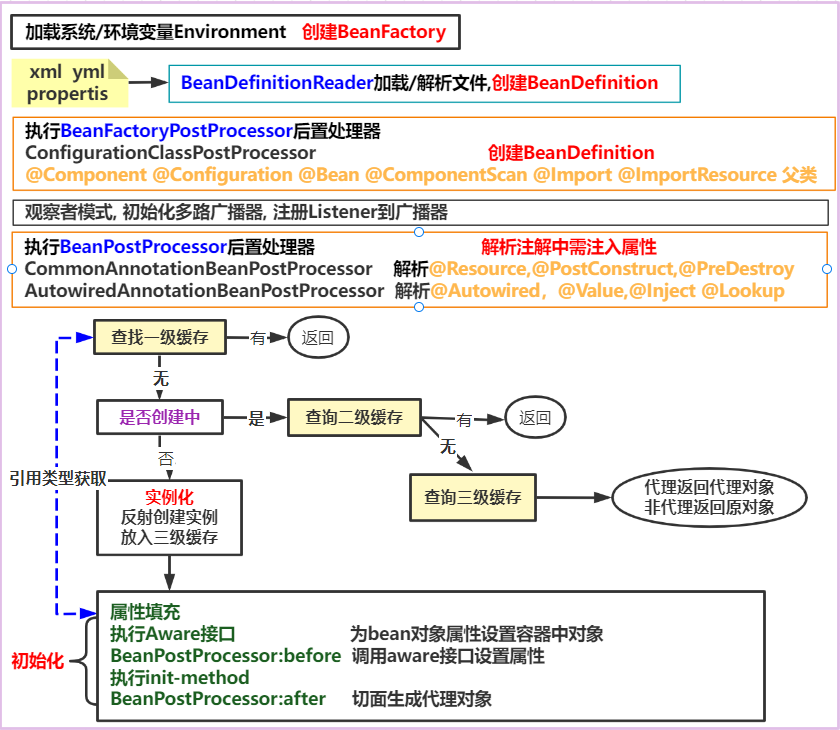
2.Server仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上，保证当前节点依然可用。

3.当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中

**缺陷**: 真正异常未发送心跳的客户端也被保留了下来, 服务间调用异常增加

# Spring

**IOC**



**自动装配:**

**springboot自动装配原理**

1. spring启动在执行BeanFactory后置处理器ConfigurationClassPostProcessor,时会处理@Import注解,实例化引入类,并调用其实现方法
2. springboot启动注解中引入@Import(AutoConfigurationImportSelector),其逻辑,会加载META-INF/spring.factories文件,生产BeanDefinition

**实例化方式:**

1. 自定义BeanPostProcessor InstantiationAwareBeanPostProcessor

2. factory-method 标签

3. supplier

**4. FactoryBean**

**5. 反射 cglib + 反射**

**构造函数选择:**

获取所有构造参数,通过参数个数,参数定义类型顺序 与 配置文件中配置参数做匹配

**循环依赖:**

**一级缓存** 完整 普通对象,代理对象

**二级缓存** 完成实例化, 但未完成初始化的对象 **解决循环依赖问题**

**三级缓存** 函数式接口的lamder表达式(一个内部类),调用时判断bean类型,代理类型返回代理类,非代理类型返回原对象 **解决代理对象生成问题**

**循环依赖原因**:

A <-> B 互相依赖,实例化A属性填充B时实例化B,B属性填充A会再创建A,形成循环创建

查询缓存顺序 1 -> 2 -> 3 或 1-> 3

**BeanFactory 与 FactoryBean区别:**

**相同点**: 都是用来创建对象

BeanFactory: 按sping标准流程创建对象的工厂

FactoryBean: 实现FactoryBean接口的类,由BeanFactory创建,通过FactoryBean接口类型单独做处理,单例缓存与普通bean区分存放,重写方法getObject()中可自定义创建对象

FactoryBean 获取方式ac.getBean( "&beanName")

自定义创建对象 获取方式 ac.getBean( "beanName")

**Spring Aop**

**@Aspect** **切面注解**

**@Component** **注册bean**

public class LogUtil {

**@Pointcut**("execution(**切点表达式**)")

**@Around**() AspectJ**Around**Advice 通知

**@Before**() AspectJ**MethodBefore**Advice

**@After**() AspectJ**After**Advice

**@AfterReturning**() AspectJ**AfterReturning**Advice

**@AfterThrowing**() AspectJ**AfterThrowing**Advice

}

**切面类**: 注册BeanDefination, 创建Advisor通知对象,

**目标类**: 在初始化最后环节BeanPostProcessor:after时, 遍历Advisor集合, 通过切点表达式, 判断是否需要被代理, 如需被代理,匹配出符合条件的Advisor, 拓扑排序,生成代理对象

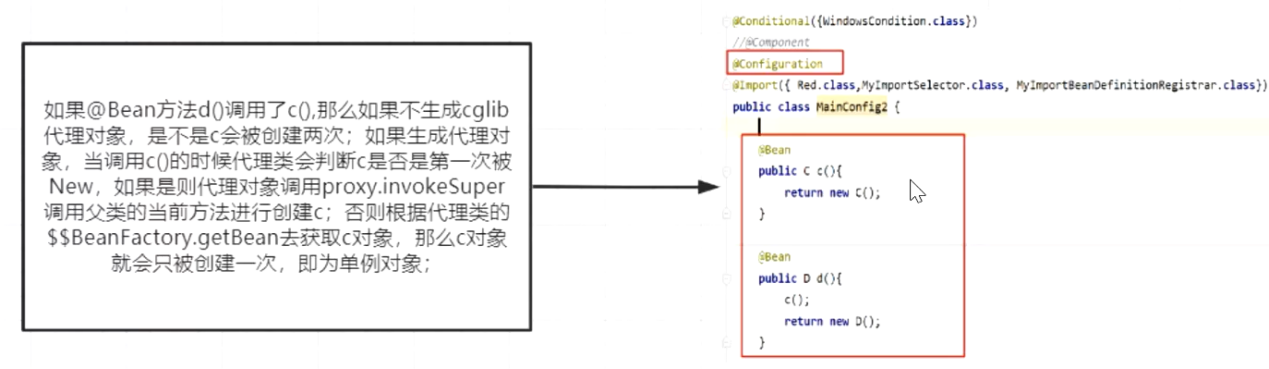
调用代理对象, 通知链转化为拦截器链执行

**动态代理**: 生成字节码文件, 加载到内存, 反射创建代理对象

调用代理对象方法, 反射调用目标类方法

**为什么配置@Configuration就会被代理?**

为保证@Bean生成对象单例

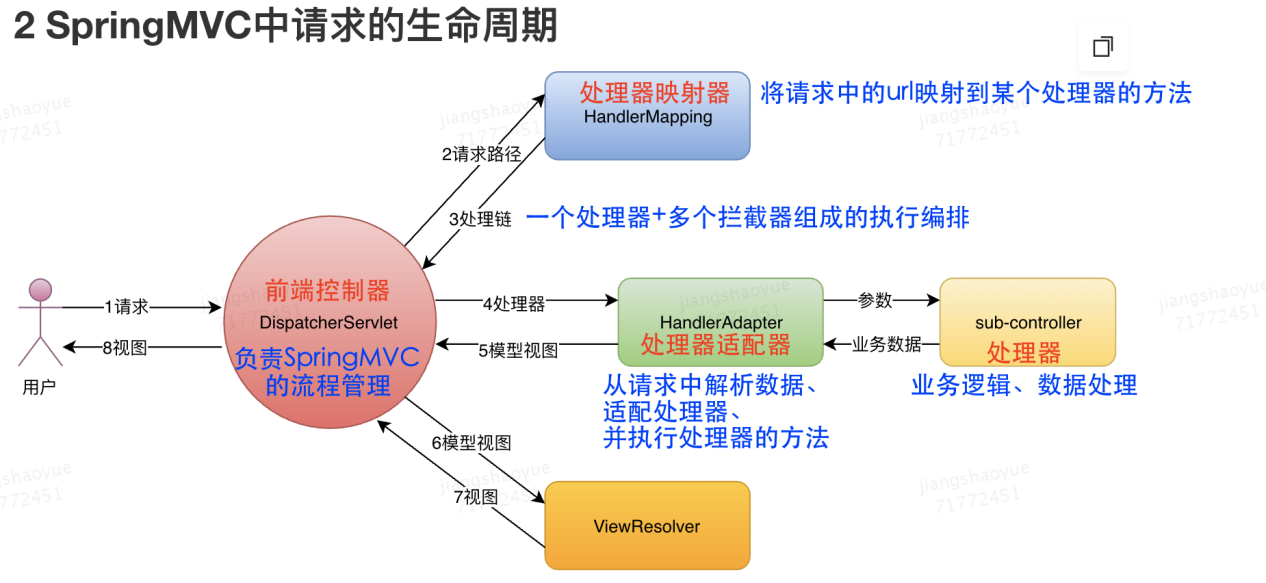


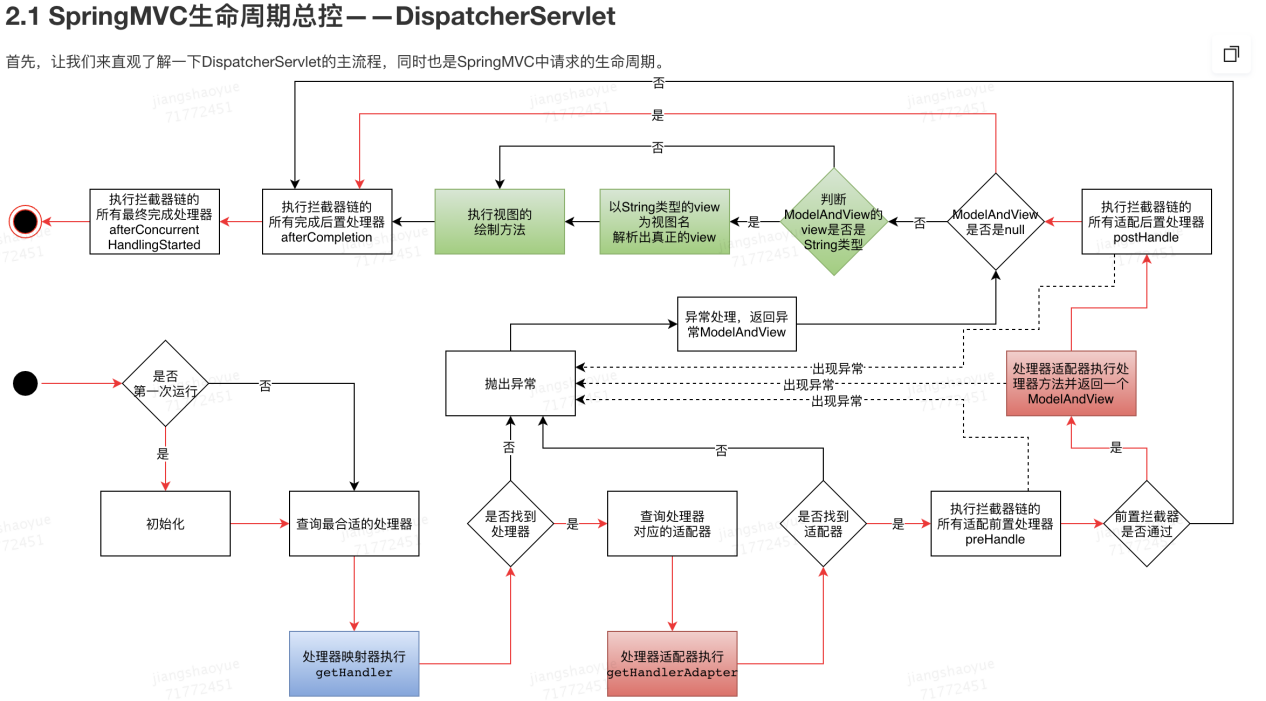
**Spring mvc**

Model（模型）：数据操作既Service层和Dao层。

View（视图）： 页面ui

Controller（控制器）：接收用户输入，操作模型，并使视图更新。既后端的web交互层。





**Springboot**

Springboot实现方式的指导思想:约定优于配置，也称作按约定编程

1. 完全注解化，内置tomcat，热部署

2. 简化 Maven 配置,通过自动装配，简化对已有框架、组件(Spring、Mybatis...)的接入

**配置优先级:**

在应用配置时，会合并它们的内容，如果有重复的配置，优先级高的配置会覆盖优先级低的

1.命令行参数(Command Line Args) (高)

2.系统环境变量(Environment Variables)

3.位于文件系统中的配置文件

4.位于classpath中的配置文件

5.固化到代码中的配置项

bootstrap.yml(引导程序) 优先级高于 application.yml

**Springcloud**

**目标:**

Spring Cloud充分利用现有开源组件,在它们之上设计一套统一规范/接口使他们能够介入Spring Cloud体系并且能够无缝切换底层实现 如只要实现DiscoveryClient[dɪˈskʌvəri]相关接口,Spring底层注册中心就可以随意更换

Springcloud是一个完整的微服务解决方案,提供分布式情况下的各种解决方案集合,而Dubbo是一款高性能Java RPC框架,主要区别是springcloud默认使用rest调用,但Dubbo将融入Spring Cloud生态,spring-cloud-dubbo,项目中引用jar则用Dubbo调用,不引入使用默认RESTful调用, 即融入后同gRPC

# Mysql

数据库选型: https://db-engines.com/en/

**字段**

|  |  |  |  |
| --- | --- | --- | --- |
| **整数** | 大小 | 有符号范围 | 无符号范围 |
| tinyint | 1字节 | (-128，127) | (0，255) |
| smallint | 2字节 | (-32 768，32 767) | (0，65 535) |
| mediumint | 3字节 | (-8 388 608，8 388 607) | (0，16 777 215) |
| int | 4字节 | (-2147483648,2147483647) | (0，4 294 967 295) |
| bigint | 8字节 | 19位 |  |
| decimal | DECIMAL(M,D) |  |  |

Tinyint(1)在java中0对应false 其他都为true

Tinyint(4)为正常整数

小数用 decimal [desiml] 不存在精度损失,如果超过则拆分为正数和小数分开存储

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | datetime | timestamp | bigint |
| 大小范围 | 4字节 | 8字节 | 8字节 |
| 是否跟时区自动转换 | 否 | 是 | 否 |
| 性能 | 通过时间范围查找 bigint > datetime > timestamp  通过时间排序 bigint > timestamp > datetime  原因：bigint是直接数值比较，项目中使用 | | |

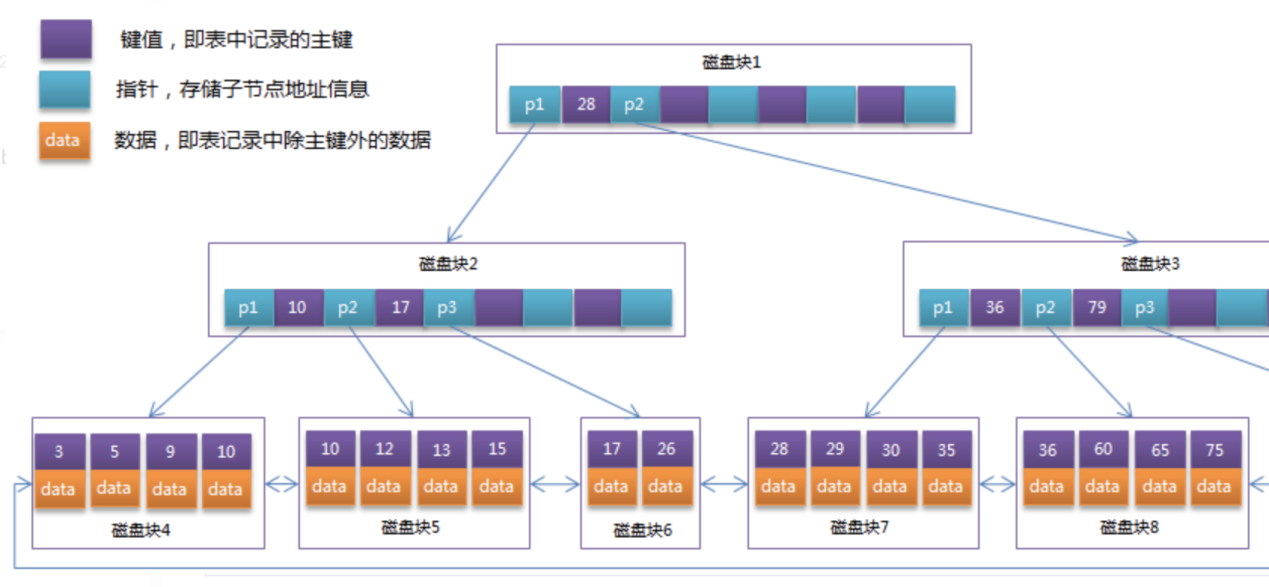
|  |  |  |
| --- | --- | --- |
| **字符** | char | varchar |
| 大小 | 0-255字节 | 0-65535字节 |
| 特性 | 固定长度,占不满补空格 | 变长字符串, |
|  | 长度相近或相等,且较短用char,如md5等 | varchar(n)表示最长放多少个字符不区分汉字数字字母 |

**索引**

磁盘: 寻址毫秒级, 带宽G/M级 秒>毫秒>微秒>纳秒

内存: 寻址纳秒级, 带宽很大

**B+tree数据结构**：



1. 页节点顺序存放索引值，增大节点存储索引值数量，降低树的高度->减少查询磁盘IO

2）叶子节点按索引值顺序存放数据，叶子节点之间关联，尽量避免排序带来的cpu消耗。

聚簇索引：默认为主键，无主键隐式创建。页子节点存放索引值和数据

辅助索引：页子节点存放索引值和聚簇索引值，需反表查询数据

唯一索引：列值唯一，可存null (只能存一个null)

普通索引：区分度越高越好，太低就不要创建索引了

组合索引：索引值为多个列值从左到右排列，查询时比较也是按顺序从左到右比较

例如：组合索引test\_user (name, phone, province)

name

name phone

name province 只会用到name

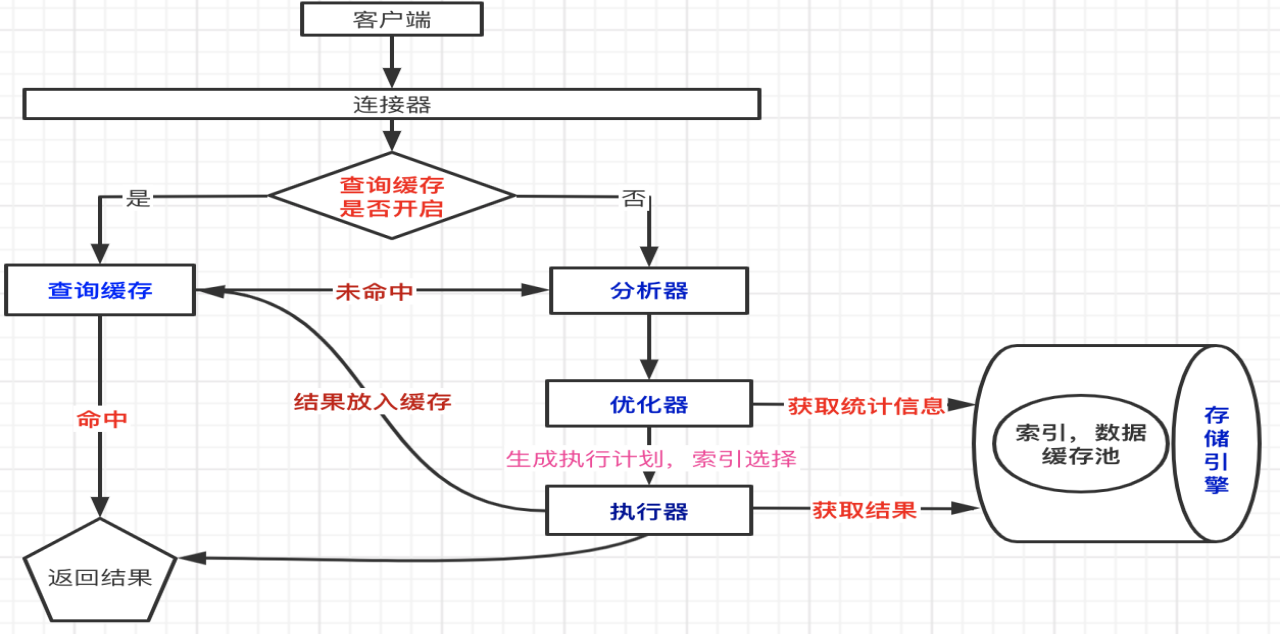
name phone province 只有这4种时才会用到索引,其他都不行

为什么随表数据量增加,性能逐渐下降?

1. 增删改需维护索引,数据量增加索引量增加

2. 查询: 并发大时会受磁盘带宽影响而变慢

**查询执行过程：**



**连接器**：负责跟客户端建立连接、获取权限、维持和管理连接。如果修改连接权限，连接断开后重新连接才生效

**查询缓存**：以(key,value)的形式存储执行过的语句和结果，默认关闭

由于对表更新操作，此表所有的查询缓存都会被清空。因此只适合更新操作极少的表开启查询缓存，其他一般不建议使用

**分析器**：检查词法分析、语法分析, 抽象语法树等。

**优化器**：引擎的统计信息和代价模型预测一个查询最优执行计划，选定索引

**执行器**：通过执行计划，调用存储引擎API，并将结果返回给客户端

如select \* from table where a = 1 and b=2; 执行计划选用b列索引idx\_b

1. 执行器从存储引擎获取满足b=2条件的第一条数据
2. 执行器判断该条记录是否满足a=1，满足放入结果集，不满足跳过
3. 执行器从存储引擎获取下一条b=2，依次到循环结束，返回客户端

**慢查询原因：**

1. 连接池连接数是否不够用

2. 网络层面：网络链路质量差，网络链路长，发送的网络数据包过大。

3. 数据库层面：未高效命中索引，查询数据量过大，查询过程存在锁等待，innodb 刷脏页，数据库的内存参数设置不合理等。

优化成本：硬件>系统配置>数据库表结构>SQL及索引

优化效果：硬件<系统配置<数据库表结构<SQL及索引

**经验**： 绝大多数的慢查都是由于索引缺失或索引不合理造成的

**执行计划**

**type**：用到的索引类型

const： 主键或唯一索引作为条件做等值匹配 where id =

eq\_ref： 表关联使用主键或唯一索引作为关联条件做等值匹配

ref： 普通索引等值匹配 where name =

range： 使用索引获取某些范围区间的记录，between、<、>、in等的查询

index： 全索引扫描

all： 全表扫描

**key**： 实际上使用的索引，如果没有则为null

**ref**： 显示索引的哪一列被用到

**rows**： 预估需要检查的行数

**extra：** 用于展示扩展信息，说明查询中的一些补充信息

Using temporary：对查询结果会使用一个临时表。 (需关注，需优化）

Using filesort： 内存排序,而不是按索引次序从表里读取行。 (需关注，需优化）

**常用原则：**

1. 使用innodb, 使用UTF-8, 表和字段加注释

2. 使用int或bigint存储金额到分, 使用varchar(20)存储手机号

3. 数据库只处理简单查询, 复杂数据逻辑放服务中处理

4. 禁止使用存储过程、视图、触发器、Event, TEXT、BLOB类型

1. 使用区分度高的列作为索引

2. 使用字段长度小、类型简单的列作为索引

3. 选用NOT NULL的列。可用0、一个特殊的值或者一个空串代替null

4. 尽量扩展索引既用复合索引，避免索引重复

5. 避免计算（+、-、\*、/等）、函数、自动/手动类型转换、not in、not exist、!=、like '%字符串' 、隐式转换 等导致索引失效

6. 尽量在order by上建索引，使用索引排序

7. 手动指定使用索引force index(索引)

8. 分页优化：优化器返回执行计划选定最优索引，从执行器一次一行获取数据，执行limit到起行数才为要获取的数据，既offset越大扫描的行数越多。优化点既直接定位到其实行。

8.1 记录上一次拉取的位置，每次拉取需要做累加逻辑where id>indexId limit10

8.2 FROM test a JOIN (SELECT id FROM test LIMIT 100,10) b ON a.id = b.id

**slow log**

**优化器追踪**

## 事务

数据库事务（原子性 一致性 隔离性 持久性ACID）保证一次操作全部成功或失败。

原子性: 事务中的所有操作要么都成功，要么都失败。

一致性: 事务前后数据库状态一致。如A-> B转账,转账前后A B账户总和不变。

隔离性: 并发时,事务之间数据相互隔离。即4种隔离级别

持久性: 事务结束后，数据永不丢失

**数据库三种并发场景：**

读--读 ：不需要并发处理

读--写 ：会出现脏读、不可重复读、幻读。**解决：隔离级别（MVCC+锁）**

写--写 ：可能更新覆盖丢失。**解决：表中添加版本version字段解决**

**脏读**： 事务可见其他事务**修改**未提交数据，其他事务回滚则为读到脏数据

**不可重复读**: 同事物多次读取同一数据，数据内容不一致（多次读间被**修改**）

**幻读：** 事物多次读取数据条数不同（**增**）

delete会修改行数据中隐藏字段“删除标记”，算修改的一种

**事务隔离级别：**事务的修改结果在什么时间能够被其他事务看到

~~Read Uncommited~~ ：读未提交。不会使用，忽略。

Read Committed (RC) ：读已提交，解决脏读，存在不可重复读。

Repeatable Read (RR) ：可重复读，存在幻读。

~~Serializable~~： 序列化，不建议使用。

RC RR级别实现： MVCC多版本并发读 + 写锁

**MVCC多版本并发控制**

**Undo log:** 引擎层，用于回滚，保证事务的一致性，记录版本链实现MVCC

**当前读**：加锁，读取最新版本。如：select lock in share mode(共享锁)

select for update 、update、insert 、delete(排他锁)

**快照读**：普通select不加锁，读取事务可见版本。



可见成立条件：读之前提交的修改版本 和 自己的修改版本 可见

可见性算法{

从最新版本开始

版本事务ID < 快照最小ID 可见（快照读前此版本已提交）

版本事务ID = 当前事务ID 可见（当前事务做的修改）

版本事务ID > 快照事务ID 不可见 -> 版本链中找上一版本新一轮判断

版本事务ID 在快照中 不可见 -> 版本链中找上一版本新一轮判断

如果回滚指针为null -> 不可见

}

RR 可重复读：只在第一次快照读时生成快照，整个事务中读返回的都为同一版本

RC 读已提交：每次快照读都生成快照，所以每次返回版本可能不同，导致不可重复度

**lock锁：**

**当前读**：加锁，读取最新版本。如：select lock in share mode(共享锁)

select for update 、update、insert 、delete(排他锁)

**按属性分**：共享锁S锁 排他锁X锁

**按粒度分**：表锁、行锁、意向锁（目的：加表锁时表中是否加有行锁未知，需要全表扫描， 为解决加表锁时不全表扫描引入意向锁。加行锁时先加意向锁， 加表锁时先判断是否有意向锁，决定加表锁还是等待）

行锁分为：

1**.单行锁**

2**.间隙锁**：锁定一个范围，不包括记录本身。防止幻读、防止间隙内插入、更新数据

3.Next-Key：：锁定一个范围，且锁定记录本身。解决幻读的问题。InnoDB默认加锁方式

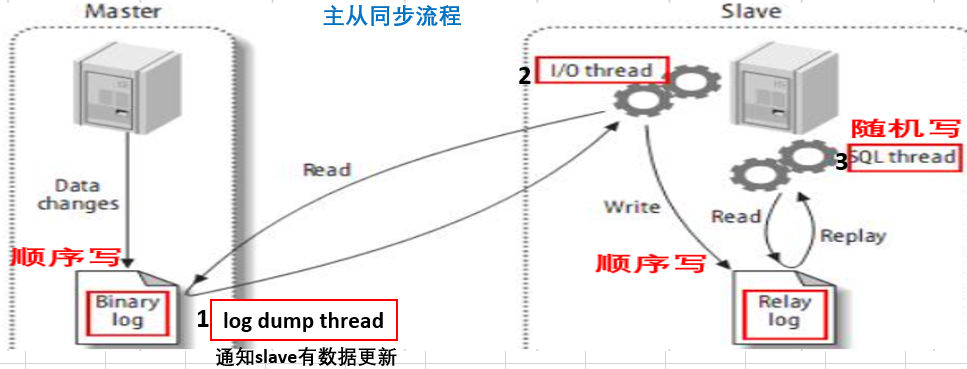
## 主从复制

**主备**: 备库不提供服务,主宕机启用备

**主从**: 读写分离

**Redo log :** 引擎层, WAL预写日志技术, 顺序写入比磁盘随机写快, 写磁盘前宕机,可通过redo log恢复数据, 保证事务持久性, 提高数据库性能

**Bin log :** server层,记录变更sql, 提供主从数据恢复、主从数据复制



**主从延时瓶颈:** 从库回放为随机写且5.7之前为单线程,耗时最长

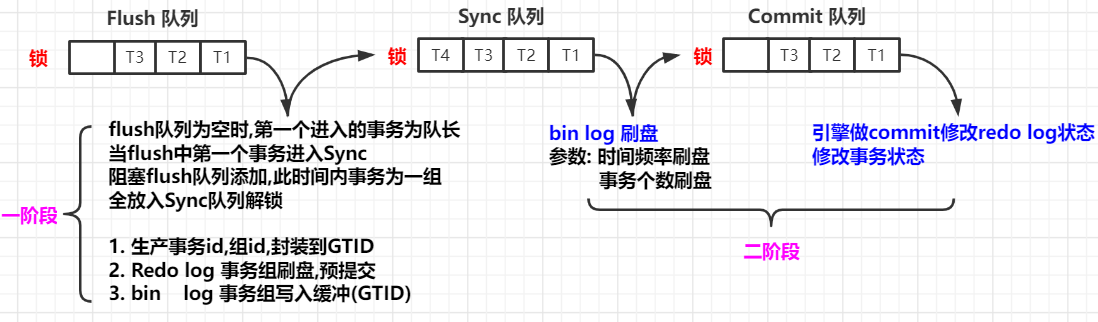
**MST并行复制**: 前提: 无数据冲突的事务可同时进入prepare预提交

用bin log组提交保证日志执行顺序,多线程回放relay Log

**组提交:** 1.减少最终刷盘IO次数

2.bin log组提交保证salve回放时多线程安全

**数据更新流程**: 二阶段提交 + redo log组提交 + bin log组提交



**结论:** 延迟是解决不了的,通过MST并行复制可缩短延时

如必须时时: 修改后查询 -> 查询主库

添加返回后查询 -> 先查从,没有再查主

同步: 所有从节点同步完成

半同步: 任意一节点完成relay log写入,返回响应

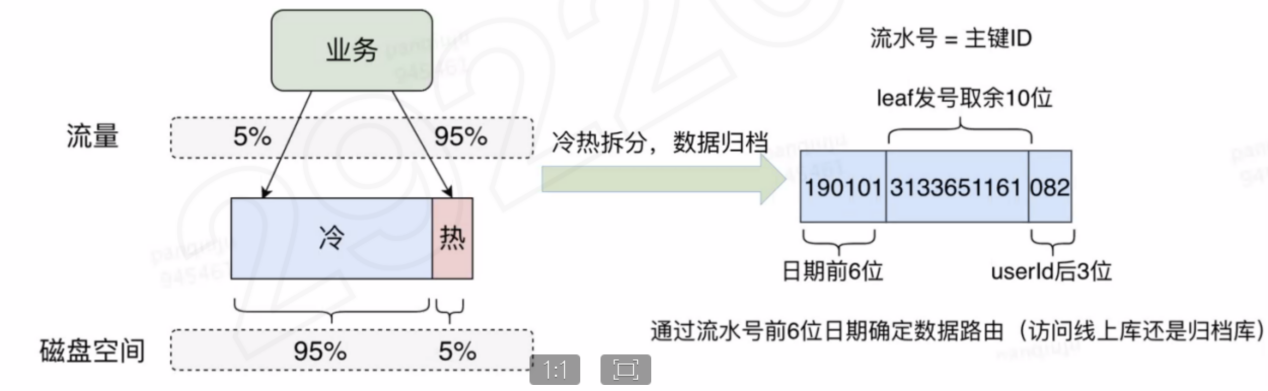
异步: 主事务提交就返回

## 分库分表

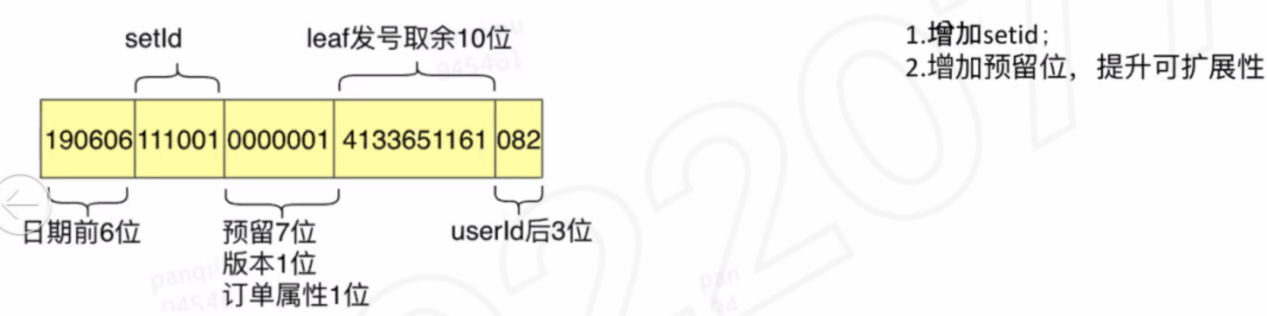
**垂直拆分**：如商品表、库存表分别单独使用一个库

**水平拆分**：单表横向拆到多个库

**冷热拆分**：热数据线上库, 冷数据归档库



**SET化：**异地多活支持识别SET标识



**分片键与分片算法**

C用户端: userId%库数 -> 库, userId%表数 -> 表 同用户数据在同库中

B商户端: 可用mapping表做商户id和用户id映射

可通过cannl类似数据同步中间件, 通过商户id分片同步到商户库

**分库分表中间件:**

**Shardingsphere** <https://shardingsphere.apache.org/>

**Mycat** <http://www.mycat.org.cn/>

**美团zabra**

库表定位：hash，Range时间区间，混合方式定位库、表

SQL解析 => 执行器优化 => SQL路由 => SQL改写 => SQL执行 => 结果归并

多表join：把不常变动的表如字典表，在每个库中各放一份

**月付分表计算**

A：预估3到5年的总记录数

B：单条记录大小(Byte)

C：单表建议记录数（小于1KW，不要超出2亿 美团集群稳定数据）

D：单库建议容量（小于300GB 美团集群稳定数据)

**分库数：**A \* B / D /1024/1024/1024（取最接近2的N次方的数，建议向上取）

**分表数**：A / C（取最接近2的N次方的数，建议向上取）

**单库分表数**：分表数/分库数

**备注**：单表最大行数，单库最大容量数是根据现有数据库最佳性能得出

此算出为基本可用分库分表数，生产冗余最少3倍

**数据迁移**

读旧库，新旧库同步双写

读新库，新旧库同步双写

全量校验，抽样校验等

读写新库

一般都用平台，手动代码方式风险太大

**数据同步中间件**

**Canal 工作原理**

1. MySQL slave 的交互协议, 伪装自己为 MySQL slave, 向 MySQL master 发送dump 协议

2. MySQL master 收到 dump 请求, 开始推送 binary log 给 slave (即 canal )

3. canal 解析 binary log 对象, 对接mq es等

## Leaf

**Id趋势递增**: 减少innodb页分裂

Innodb以块为单位存储数据,块之间为链表结构,逻辑顺序排列, 当某页写满后, 插入数据id刚好在此页, 则会把此页拆分为两页, 会造成页数据不满, 即空间碎片, 读取时需读多个不满的页才能返回需要的数据, 造成io次数增加

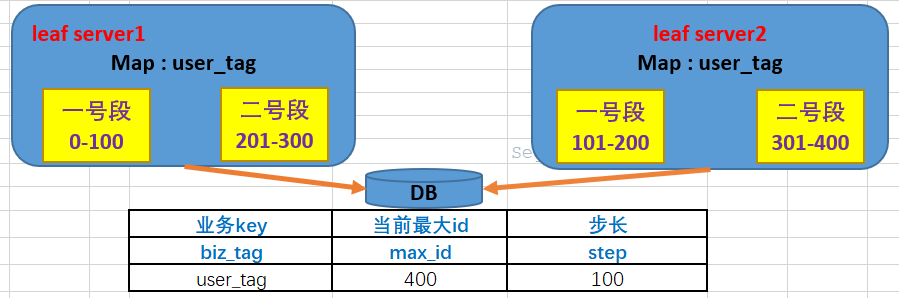
**使用数据库单调递增**:

缺点: 1. 单台mysql读写是性能瓶颈

2. 强依赖DB, DB挂掉整体不可用, 即使造成用主从, 也有可能数据非强一致重复

**Leaf-segment分段实现:**

1. 通过step步长减少数据库读写
2. 通过内存双号段切换,减少数据库io阻塞概率
3. 内存动态调整step步长,应对流量突发



**雪花算法:**

**符号位1 + 时间戳41 + 机器ID10 + 序号12(4096)**

时间回拨:

原因: 机器时钟并不稳定,忽快忽慢,会通过网络进行校验,如果快了时间就会回拨

处理: 用当前毫秒数与上次使用记录下来的毫秒数比较,当前毫秒数小则表示时间回拨了,如果小于5ms,等待两倍时间,再次判断还小,则返回报错,让客户端从其他服务器获取

机器ID: 通过ip+port在ZK上注册永久节点,从ZK上获取机器ID,保证集群中机器ID不重复

序号: 每毫秒内从一个随机数开始,进行递增

从随机数开始原因: 大多数情况在单位毫秒内获取id为几个,如从1开始则获取的ID尾数都会有1,2 通过id做分片,会照成数据不均匀

# Reids:

**为什么快**

1.内存操作,寻址纳秒级,比磁盘快10万倍

2.高效数据结构

3.多路复用机制epoll

4.单线程selector + 读写io, 协议解析, 指令处理 -> 数据安全,不切换上下文

6.0只后改为: 单线程selector, 多线程io读写/协议解析 ,单线程指令处理 同游戏模型

**数据结构:**

memcached value无类型,不能直接操作value中元素

**String:**

**value**: 可变长字节数组

**字符串**:

**数值** : 可递增/减1, 可增/减自定义值 -> 可做库存

**bitmap**: 二进制位操作, 为bit创建连续的正负向索引

1. 记录用户登录日期(同天只记录一次),统计随机时间段内用户登录次数

思路: key 用户id\_日期

: value使用365个bit位, 每bit表示一天, 登录就设置1

设置: setbit userId\_date 日期映射索引位 1登录0未登录

统计: bitcount key 索引开始位 索引结束位

1. 记录每日登录用户,统计随机时间段内活跃用户

思路: key 日期

value每bit映射一个用户,当日登录就设置bit为1

设置: setbit date 用户映射bit位 1登录0未登录

统计: bitop or(或运算) destkey key1 key2 ......

bitcount destkey 0 -1

**list： 双向链表** **提供 数组, 栈, 队列 功能**

**hash： 数组+链表**

**set： 可变长数组** **无序,去重集合**

随机抽奖:

SRANDMEMBER key [count] 返回集合中一个或多个随机数

集合中5个用户, 3个奖品, 抽3个用户 SRANDMEMBER key **3**

6个奖品, 抽重复用户 SRANDMEMBER key **-6**

SPOP key 移除并返回集合中的一个随机元素

**zset: 跳跃表SkipList scop有序,去重集合 可做排行榜**

**发布订阅**: 广播发布 , 打开监听前的数据不可获取 **可做及时信息发布**

**回收策略**

当Redis达到最大内存限制时，触发回收机制。六种可配置

1. 写操作时,直接返回错误

2、所有key中，**随机**部分key

3、所有key中，**最少使用**的key

4、设置了过期时间的key中，**随机**部分key

5、设置了过期时间的key中，**最少使用**的key

6、设置了过期时间的key中，**剩余时间短**的key

Redis做缓存时: 选最少使用方案

内存可配置:一般1-10G, 再大会造成持久化,迁移性能问题

**过期策略**

Key设置过期时间后, redis会对过期时间做倒计时

当key被删除或覆盖，会移除过期时间

当value被修改, 不会移除过期时间

**被动方式**: 每次访问时检查是否过期，过期则删除

**主动方式**: 每100毫秒

1.随机20个keys进行相关过期检测

2.删除已经过期的keys

3.如果有多于25%的keys过期，重复步奏1

## 持久化

**RDB二进制全量持久化**

**Save** : 阻塞客户端, 主进程持久化内存到磁盘 场景: 关机维护

**Bgsave** : 根据配置参数 多长时间 || 发生多少次修改,出发持久化

1.生成主进程完全相同的**子进程**, 共享物理内存

2.由子进程持久化,持久化完成阻塞客户端,把期间增量数据持久化,文件覆盖

配置：save m n多长时间 || 发生多少次修改 触发存盘(需根据数据量 ,磁盘写速度等因素)

优点：存储二进制,重启数据恢复快

缺点：最后一次写盘到宕机数据都会丢失 场景: 对数据恢复的完整性要求不高

**AOF Append Only File文件追加持久化**

主进程**按策略**同步追加方式**存储有效写操作指令**

写入策略: 1.每个写指令都同步, 数据完整但性能差

2.每秒同步一次, 宕机会丢失一秒内数据（推荐）

3.由操作系统控制

**重写:** AOF文件逐渐增大,恢复时间增大,一些key的删除造成无用指令->重写

触发机制: 配置的AOF文件初次重写大小

超过与上次文件增长百分比阈值

重写流程:

1.子进程创建新文件,主进程文件指令写入新文件

2.新文件完成, 主线程阻塞, 重写期间主进程接收的写指令, 写入新文件

3.覆盖旧文件

**恢复:** Redis创建一个无网络连接的伪客户端, 读取AOF文件, 写入redis

**优点:** 数据完整性高 场景: 对数据恢复的完整性要求高

**缺点:** 文件大,恢复慢 写盘频率高,磁盘IO高,影响性能

**混合持久化**

文件格式: bgsave二进制数据 + AOF写指令

结合了RDB二进制数据量小,恢复块 + AOF 数据完整性高

**单点问题:**

**单点故障**: 使用主从,从节点备份数据. 使用哨兵集群保证故障转移.

**单点压力**: 主写从读, 读写分离

**单点扩容**: 集群使用16384个槽, 每个master映射一部分槽, 新增或移除master节点, 手动执行指令, 进行数据迁移(已有数据,增量,校验),迁移完成切换新master

## 高可用

**哨兵获取master, slaver, 其他哨兵动态信息**:

slaver注册master, 所有哨兵监控master, 哨兵订阅master, master定时发布消息同步数据

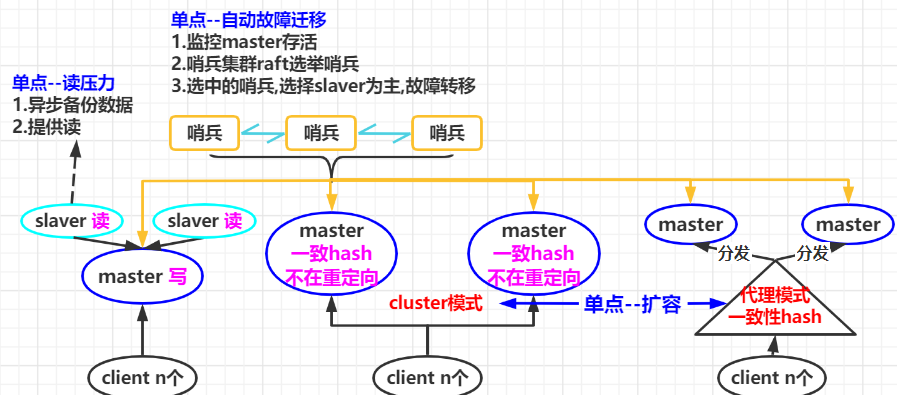
1.哨兵每秒向master发送PING消息,在设置的时间内未有效返回,认为起主观下线

2.向其他哨兵发送确认请求,超过配置数赞同则标记为客观下线

3.哨兵集群raft选举

4.选中的哨兵根据以数据完成性的多个规则挑选一个slaver转master, 向其他哨兵发布结果

5.其他slaver复制新master数据



## 数据一致性:

**CAP**:

在一个允许网络发生故障的系统中，低延迟和一致性不可能同时被满足, 该选择一致性还是可用性？应根据自身的业务特点,权衡一致性和可用性,达到最终一致

**C一致性:**

强一致: 写操作,集群所有节点同步完成,才返回响应

最终一致: 如使用mq,一定时间内所有节点数据最终会达到一致

弱一致: 异步,数据可能不一致

**A可用性 :** 响应越快即延迟越低,可用性越高,一致性越低

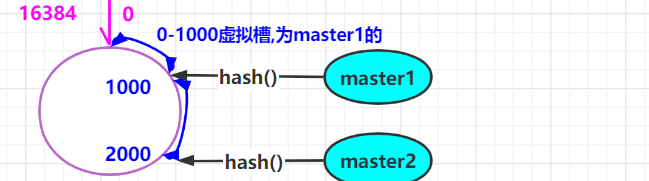
**P分区容错性:** 网络分区不可避免,允许网络分区发生时节点间数据不一致

Redis作为缓存,目的是为了快速响应, 采用了节点间异步复制数据，master宕机数据丢失,最终一致性都达不到

## 数据分片:

为解决扩容,引入master集群, 数据分片存储.

节点一致性hash值映射到16384个数(槽)组成的环上



**Cluster: 服务端分片**

多个master组成一个无主集群,通过Gossip流言协议更新元数据

使用相同一致性hash crc16算法算key, 如果槽刚好在此master上则返回, 未在则返回槽所在的master信息,客户端重定向

**代理:** 减少客户端与server连接.Predixy twemproxy codis

**分片弊端**:

不在同一节点,无法使用统计,同mysql拆分

提供 hash {tag}key 可放到同一节点上

迁移过程是否提供服务: <https://blog.csdn.net/qq_41453285/article/details/103395224>

## 分布式锁

**锁特性**:

1.同时刻只能被一个获取 redis单线程执行,保证请求同步执行

2.只有获取锁的人可释放锁 value值唯一

3.释放锁后如何通知其他人 其他人循环休眠即定时获取锁

4.客户端宕机,造成死锁 设置key到期时间

问题: 业务执行时间>到期时间 起线程,在业务未执行完前,检测锁过期,续期

5.服务端单点,宕机锁丢失 使用多master集群

**加锁：set( key, value, NX**仅当其不存在时生效**, PX, expireInSeconds**超时时间**)**

**解锁：**lua脚本原子性，指令: 比较相同则删除

**单点故障：**

Redlock方案：向N个master实例加锁，最少N/2+1个成功视为加锁成功

1. 记录当前时间戳，使用相同的key和value以串行方式从master超时获取锁

2. 获取至少N/2+1个锁，且获取锁时间 < 锁到期时间 -> 获得锁

3. 获锁成功，锁到期时间=原到期时间-获取锁时间，设置

4. 获锁失败，对所有master进行解锁，随机延迟后重试

总结: 等待锁的线程定时请求redis,压力大

需多起一个线程检测业务执行时间与锁到期时间为其续期,编程复杂

解决单点问题,加解锁逻辑更复杂

分布式锁,效率要求不强,准确度和一致性要求高, 不建议使用

## 本地缓存:

Guava Cache 使用弱引用,内存不足是回收

# MQ

消息队列是暂时存储数据的容器，是一个平衡低速系统和高速系统处理任务时间差的工具

**削峰:** 防止突发流量, 消费端匀速消费, 即限流漏桶算法实现, 允许时间差如短信,物流

**异步:** 缩短请求同步调用时长, 如耗时请求,先响应状态为处理中, 前端轮训查处理状态

**解耦:** 1.不同语言服务对接

2.依赖方接口修改或增加其他依赖接口调用，服务代码不用修改

1. **如何保证消息一定发送成功**

解决一: 本地事务 + 消息表 + 定时失败消息发送任务

解决二: Rocket mq半消息

1. **mq消息堆积**

解决：优化消费代码提升消费速度

增加消费者的数量

## Rocketmq

### NameServer

功能 类似注册中心:

1. 接收所有broker 30s长连接心跳,携带topic queue broker filterserver数据,做缓存

2. 10s定时检测 (当前时间 - broker最新心跳时间 > 2min),关闭channel, 更新数据

3. Netty监听channel 连接/关闭/异常/idle事件, 更新数据

4. 为生产,消费端提供路由数据, 即Topic下的MessageQueue分别都在哪台 Broker上

集群: Nameserver之间无通信, broker心跳到所有Namesrv节点

部分宕机不影响其他Nameserver

全部宕机生产/消费端有缓存

### Producer

**启动:** 一个进程启动一个producer

创建发送线程池

创建客户端,注册请求类型code : 处理器 映射 ClientRemotingProcessor

定时任务 从nameService更新Topic路由信息

定时任务 清理异常broker连接,向所有broker发送心跳信息

**发送公用逻辑:**

生产随机消息id(即不唯一)

确定重试次数: 同步3次,异步/oneway 1次

获取topic要发送的MessageQueue和Queue的Broker地址

nameserver中获取topic queue broker信息

选择发送目标队列 = 自增%queue

本地缓存中获取broker channel地址

封装协议信息

同步发送: 生成唯一requestId,构建响应体ResponseFuture,放入ConcurrentMap

CountDownLatch 阻塞线程

异步发送: 生成唯一requestId,构建响应体ResponseFuture,构建回调方法,

放入ConcurrentMap, 线程继续执行

ONEWAY: 发送后直接返回

响应处理:

通过requestId 从ConcurrentMap获取 ResponseFuture,并移除

有回调方法则为异步发送: 执行回调方法

无回到方法则为同步发送: 释放countDownLatch 线程继续执行

**延时消息**:

开源版本只提供了时间级别,不支持具体时间设置. 云版本提供

生产端会设置消息类型Delay\_Msg

Broker处理: 1.将延时消息放入topic SCHEDULE\_TOPIC\_XXXX存储

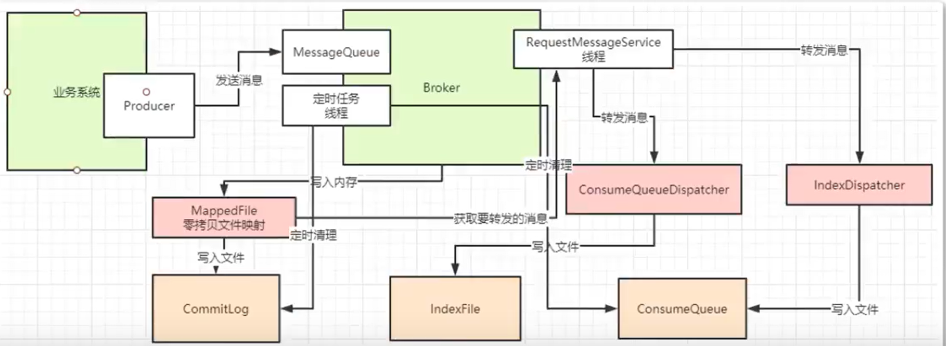
2.ScheduleMessageService 服务线程检查到时消息,写入发送时topic

**顺序消息**:

1.选择器获取topic队列集合 2.选择queue 3.把消息放入同一个队列里

**半事务消息: 参见分布式事务**

### broker



commitLog 存储消息

consumeQueue 存储消费索引

IndexFile 消息索引文件,存储消息 Key 与 Offset 的对应关系

数据同步: raft算法选举

为什么快?

1. nio mmap共享内存, 顺序写盘

2. sendfile零拷贝

### cosumer

广播模式: 订阅topic下所有queue

集群模式: 通过所有queue和所有consumer端,算出本机订阅的queue

定时任务:

1. 消息本地缓存ProcessQueue中消息条数大于1000, 延迟拉取

2. 消息本地缓存ProcessQueue中消息大小大于100M, 延迟拉取

3. 封装拉取消息请求消息头, 响应成功回调函数

4. 异步调用

5. 执行消息过滤

6. 响应结构加入本地缓存ProcessQueue,调用回调函数消费

7. 向broker发送消费结果

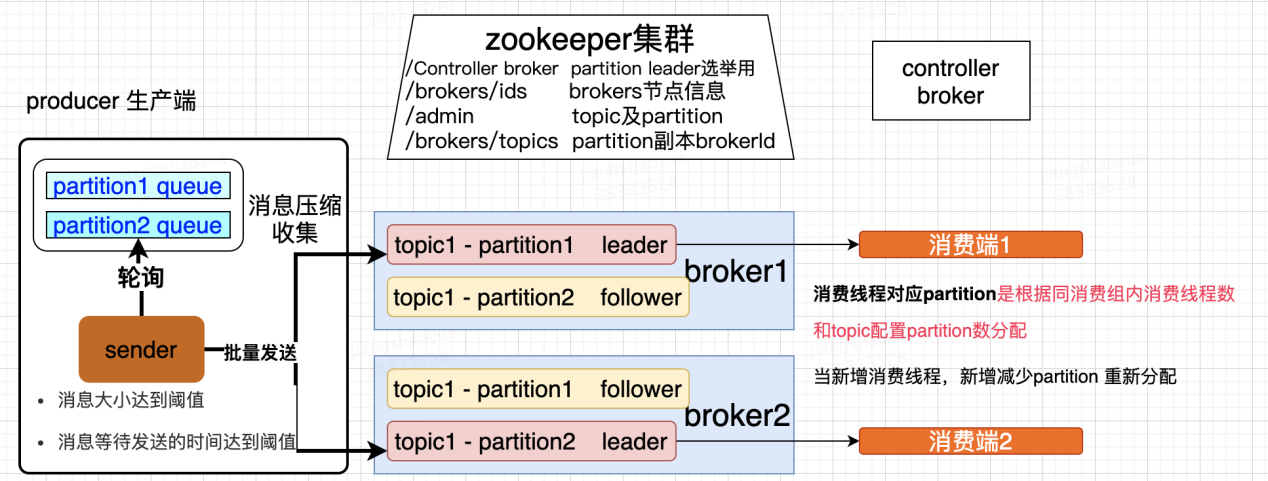
**Push** 拉取无消息会被broker挂起, 有消息后由broker推回来

**Pull** 拉取有无消息都会返回,如无消息则消耗性能

**消息过滤:** Tag / sql两方式都是本地过滤

**消费失败:** 响应broker,broker把失败消息放入重试队列

## KAFKA



属性：

Broker : 消息存储机器节点

Topic : 逻辑队列，被分为多个partition切片（个数可配置），保证水平扩展

Partition：一个leader，多个副本，副本存放在不同broker上，保证高可用

### 消息发送：

消息被压缩后追加到本地记录收集器，sender轮询收集器，当消息大小达到阈值或消息等待发送时间达到阈值，将消息发送到partition leader

### 消息存储：

1. 消息响应的三种配置
2. 消息写入broker缓存区就返回成功响应 会丢数据
3. 消息持久化后就返回成功响应 leader宕机，副本并未同步，会丢数据
4. 消息持久化，并同步副本后返回

1. 消息持久化

采用顺序写的方式来做消息持久化，各partition独立日志文件存储，日志文件起始既基准偏移量，当日志文件达到阈值（默认1G）生成新的文件，批量消息带有顺序，追加到日志文件，生成绝对偏移量。



索引文件查询：用查询消息绝对偏移量-最新日志文件基准偏移量->日志文件中相对偏移量，在索引文件中找到与相对偏移量相近的索引对应的磁盘地址，再在磁盘中查找

索引文件中存储相对偏移量，存储部分消息索引而不是全部是为了减少内存

1. 零拷贝：Linux 2.4+内核通过sendfile系统调用，提供了零拷贝

### 消息消费：

1. 消费线程与partition切片对应关系：

通过同消费组消费线程数（一个消费服务中可有1到多个消费线程）与topic切片数计算出消费线程与切片对应关系。

1. 消息消费状态

kafka是拉取方式获取消息，由消费组维护消息偏移量，之前存储在zk中，0.9后可存储在特定的topic中

### 故障恢复：

**ZK**

**/controller broker 选举用**

**/brokers/ids broker节点信息**

**/admin topic和partitions**

**/brokers/topics partition副本信息**

**controller与zookepper断开连接**

1. controller临时节点被删除，触发在其注册watcher的所有brokers

2. brokers去创建ZK上controller临时节点，成功者成为选举节点

3. 在ZK的brokers注册Watcher

**broker与zookeeper断开连接**

1. broker临时节点被删除，触发在其注册watcher的controller选举节点
2. cotroller从/brokers/ids节点上获取宕机broker的所有partition

从/brokers/topics获取宕机broker中partition的所有副本

对宕机broker中Leader partition，从其副本中选择一个做为新leader

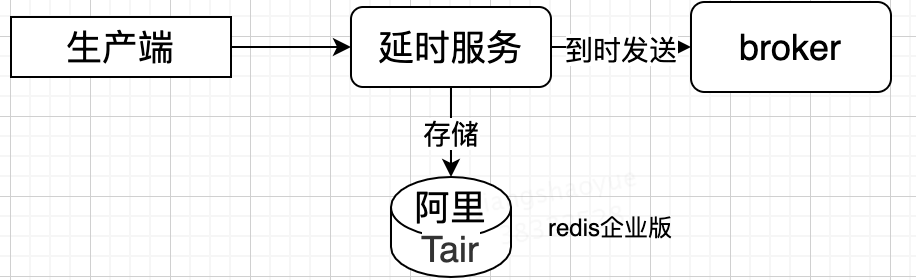
同步结果到所有涉及到的副本broker，并接收响应

整个过程涉及到的副本不接收请求，保证数据一致性

### 美团mafka封装与补充：

1. 集成cat对消息发送，消费整流程打点监控
2. 增加延时消息功能
3. 增加消息回放，重复消费功能
4. 死信重试功能
5. 增加push，消费端数量不再受topic切片限制

延时消息方案:



核心为两部分：存储 ，到期拉取

存储：使用hashmap结构存储 注意：key不可重复，防止key覆盖

key = topic+到期时间戳向下100ms取整，

副key = uudi+消息id

# 分布式事务方案:

## 2/3PC

基于数据库2PC两阶段事务提交XA规范 **+** 事务管理器TM(腾讯的TDSQL)

**两阶段提交2PC**

**一阶段预提交**

1. App发送 开启全局事务 到TM，TM生成全局事务ID，xid

2. App发送 全局事务语句 到TM，TM分解出分支事务,发送到各个mysql节点。

3. App发送 commit 语句 到TM，TM发送预提交, mysql打开本地事务, 写入redo log undo log, 返回成功或失败, 预提交后阻塞等待

**二阶段提交或回滚**

TM接收所有响应，全部OK，则向每个mysql节点发送提交，

有一个响应失败, 则像每个mysql节点发送回滚

**三阶段提交3PC:**

1. 在预提交前 ,执行sql但不开启事务, 提前了预提交结果, 减少预提交失败后的阻塞等待

2. 在mysql端增加了等待超时处理

**XA 优点**:

1.无代码侵入

2.尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域

**XA 缺点:**

1. 必须要求数据库支持XA

2. 预提交后阻塞等待, 不适合高并发高性能场景

3. 已经落地方案，都依托于重量级的应用服务器（Tuxedo/WebLogic/WebSphere)，不适用于微服务架构

4. 事务管理器TM, mysql宕机, 网络异常, 都会照成数据不一致, 并不能完全解决数据一致

**使用场景: 依赖关系型数据库,一致性强,延时高. 适用于同服务跨库, 并发低**

## TCC

应用层面代码侵入业务的两阶段提交

**第一阶段:** try对业务系统做检测及资源预留 (加锁，锁住资源)

**第二阶段:**根据第一阶段结果决定执行或取消预留资源

如: 减库存 -> 创建订单

Try阶段 : 减扣库存, 冻结预留库存+2 生产订单,状态为待确认

二阶段 : 根据try结果 执行 -> 冻结库存-2 订单状态为确认

取消 -> 库存+2,冻结库存-2 订单状态为失败

**允许空回滚**: 由于网络原因分支事务还未接到try, 全局事务已经回滚, 接收到回滚, 此时应响应回滚成功

**方悬挂控制**: 分支事务已空回滚后, 才接收到try, 则需处理try不执行

**使用场景: 时间短,一致性要求高,如交易/支付/账务**

## RocketMq 半事务方案

RocketMQ 4.3 之后提供了二阶段提交方案

发送消息 🡪 MQ 状态为待确认

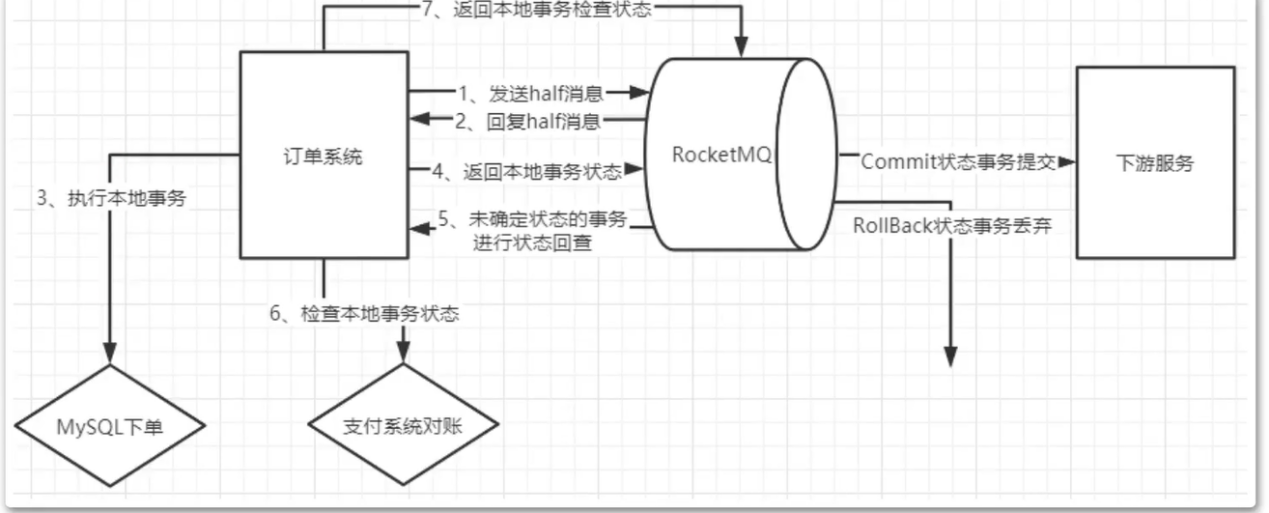
响应成功 🡨 MQ

执行本地事务

二次提交 🡪 MQ 修改消息状态待确认/确认/取消 保证本地事务与消息原子性

**二次提交超时或宕机**: MQ会对未确认的消息定时回查数据状态, 业务服务需实现回调逻辑,再次发起二次提交

**缺点:** 业务耦合强, 不可公用, 必须使用关系型数据库事务, 微服务节点多后代码编写复杂



## 美团Saga

**场景**：下游服务不支持事务代码接入，不引入协调者

**概况**: 直接提交, 如果失败通过配置 可重试,可回滚

**设计实现**:

技术方案: spring AOP + mysql日志事务表 + 反射 + 下游接口幂等

恢复： 定时任务 + mq

日志清理: 定时任务

### 注解定义：

**主事务注解 主事务成功回调注解 主事务失败回调注解 主事务回滚注解**

**子事务注解 子事务成功回调注解 子事务失败回调注解 子事务回滚注解**

**业务流水号注解 :** 用于获取参数中业务流水号

**分库分表健注解 :** 用于获取参数中分片值

**主事务注解属性**：主事务名称

事务失败后策略：全局重试，全局回滚，通过子事务配置异常判断

补偿重试次数

补偿重试时间间隔

**子事务注解属性**：子事务名称

子事务需重试异常集合

子事务需回滚异常集合

**事务5种状态**：0-初始态, 1-执行成功, 2-执行失败,3-回滚成功,4回滚失败

### 正向执行流程:

**主事务切面 ------------------------------------------------**

解析注解, 类, 方法, 入参 信息放入InheritableThreadLocal

try{

持久化数据到主事务日志表，**初始态**

执行主事务业务方法

**子事务切面 ----------------------------------------**

解析注解, 类, 方法, 入参 信息放入InheritableThreadLocal

子事务状态 **初始态**

try{

执行子事务业务方法: 方法返回值json放入上下文

子事务状态 **执行成功**: **反射调用子事务成功后方法**

}catch{

全局重试 或 子事务异常判定为重试：按重试次数，时间间隔执行

**成功**: 子事务状态 **执行成功** ,**反射执行子事务成功后方法**

失败: 子事务状态 **执行失败** ,**反射执行子事务失败后方法 throw e**

}

**-----------------------------------------------------**

反射执行主事务成功后方法

主事务状态 **执行成功**

}catch(){

**全局重试 或 子事务异常判定为重试：**主事务状态 **执行失败** **throw e**

**全局回滚 或 子事务异常判定为回滚**： 同步、异步执行回滚

try{

逆序上下文中已执行子事务

if(子事务执行成功) 🡪 反射调用子事务回滚方法

反射执行主事务回滚方法

主事务状态 **回滚成功**

}catch{

反射执行主事务失败后方法

主事务状态 **回滚失败**

**throw e**

}

} finally {

清除上下文ThreadLocal

}

### 事务补偿：

**定时任务**: 查找非终态事务id: 初始态,回滚失败,执行失败 🡪 MQ

**服务消费**: 通过事务id查询事务信息,从spring中获取bean,反射执行

**去掉子事务日志表分析:**

在事务补偿时,可从数据库中获取自事务执行数据,快速恢复上线文

**重试**: 可直接定位到执行失败的子任务,继续执行

**回滚**: 可直接定位未回滚子事务直接执行

但大前提为事务日志表异常,不可影响业务,会造成事务日志表状态不为最终

而且会增加DB操作,业务成功比例极高,成功子事务日志会占用mysql磁盘

**框架兜底**: 如果补偿为回滚,失败子事务执行成功了,则需人工处理, 如支付有对账

### 事务传播：

上下文实体添加 ‘层数’ 属性

1.主事务上下文初始化前，判断上下文有无数据，不为null表示有外层主事务 -> 不初始化

2.事务嵌套层数+1，判断嵌套层数，大于1则有外层主事务 ->不入库

3.业务执行成功 ，嵌套层数不为1 -> 不入库

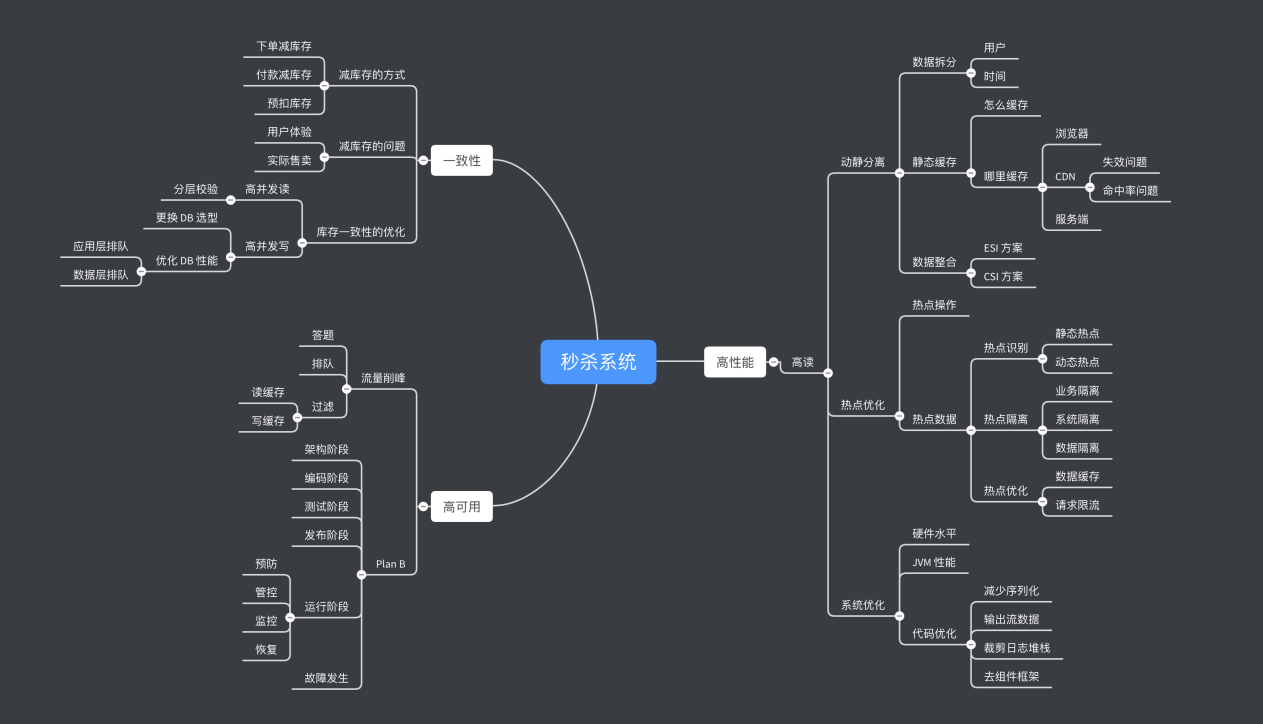
4.业务执行异常 -> 直接返回

5.finally -> 事务嵌套层数-1 ，若等于0 则释放上下文

### 使用既编写规范：

1. 能不用尽量不用分布式事务
2. 事务环节尽量要减少（减少运行损耗，减少复杂度，减少维护与故障处理复杂度）
3. 第三方写接口必须保证幂等，不能则自实现：要先查后写
4. 事务前先做校验，事务中不随时间变化的业务异常才需回滚
5. 子事务中为处理重试减少幂等接口调用，先检查

# 秒杀方案



**秒杀实质:** 以库存管理为核心的高并发读写

**通用准则:** 越靠近后端, 请求数越来越少, 单次请求数据量越来越小

热点发现: 提前预热热点数据

热点缓存：静态数据走cdn、浏览器等，动态数据走Redis、客户端本地缓存等

系统隔离: 秒杀链路与普通业务做到逻辑隔离,最好做到物理隔离,如单独机器部署

数据隔离: 下单、减库存等核心逻辑依赖数据独立存储, 划分单独mysql或redis等集群

分库分表打散热点库存

流量过滤: 答题,黑白名单,ip限制

**何时减库存:**

1. 下单减库存
2. 付款减库存
3. 预扣库存(下单后留库存一定时间，超时恢复库存)

* 秒杀场景绝大多数会购买, 下单减库存逻辑简单, 性能好

**Redis减库存:**

lua脚本（判断库存数，返回或减redis库存） -> 异步mq减扣数据库库存

**库存超卖处理：**

事务+ for update 阻塞其他事务for update读，事务结束释放锁

事务+ for update nowait 拒绝其他事务for update读

事务+ for update skip locked 跳过已加锁的数据行

事务+ for update查询判断库存数 > 减库存数 才执行修改

update table set count = count -N where sku\_id = x and count > 0

**兜底**: 使用全链路压测预估容量, 限流, 降级

**Mysql配置**:

Redolog刷盘 innodb\_flush\_log\_at\_trx\_commit = 2

0每秒刷一次，宕机丢1s数据

1提交事务就刷

2由系统控制刷磁盘

binlog刷盘

sync\_binlog = 0

0表示由系统控制，性能最高，宕机丢数据也多

1提交事务就刷，最安全性能最低

binlog\_group\_commit\_sync\_delay = 0 加速事务提交

binlog\_group\_commit\_sync\_no\_delay\_count = 0

打开自适应hash索引 innodb\_adaptive\_hash\_index = 0

关闭死锁检测 innodb\_deadlock\_detect = 0

# 监控及分析方法

**监控目标**

**提前预警**：通过观察CPU指标预知系统瓶颈、查看JVM指标来进行对应的参数调优。

**及时报警**：有问题及时报出来，失败数增多、GC频繁等。

**追盘问题**：有很多监控指标之间是具备包含关系的，一个总指标可能是很多子指标经由过程 计较得出来的，所以呈现问题我们就需要详实的数据来定位问题、追盘问题。

**容量规划**：好比DB存储空间，从它的增加速度可以预估读写性能，实时合理规划，提前扩 容或者分库分表。

**服务器基础监控：**

CPU：单个CPU以及整体的使用情况

内存：已用内存、可用内存

磁盘：磁盘使用率、磁盘读写的吞吐量

网络：出口流量、入口流量、TCP连接状态

**数据库监控**

数据库连接数、QPS、TPS、并行处理的会话数、缓存命中率、主从延时、锁状态、慢查询

**中间件监控**

Nginx： 活跃连接数、等待连接数、丢弃连接数、请求量、耗时、5XX错误率

Tomcat：最大线程数、当前线程数、请求量、耗时、错误量、堆内存使用、GC次数,耗时

缓存 ： 成功连接数、阻塞连接数、已使用内存、内存碎片率、请求量、耗时、缓存命中率

消息队列：连接数、队列数、生产速率、消费速率、消息堆积量

日志监控：访问日志、错误日志

**应用监控**

HTTP接口： URL存活、请求量、耗时、异常量

RPC接口： 请求量、耗时、超时量、拒绝量

JVM： GC次数、GC耗时、各个内存区域的大小

线程: 当前线程数、死锁线程数

线程池： 活跃线程数、任务队列大小、任务执行耗时、拒绝任务数

连接池： 总连接数、活跃连接数

链路追踪: Skywalling

业务指标： 视业务来定，比如PV、订单量等

业界开源：小米Open-falcon [falkən] 普罗米修饰

流程： 数据采集 -> 数据传输 -> 数据存储 -> 数据展示 -> 监控告警

# 架构演进

**分布式系统:**

定义: 由一组通过网络进行通信,为了完成共同的任务而协调工作的计算机节点组成的系统

目的: 利用更多的机器，处理更多的数据。

场景: 单节点处理能力, 通过硬件提升达到高昂到得不偿失的时候, 应用程序也不能进一步优化, 才考虑分布式系统

例: 大数据MapReduce, 通过网络把数据分片, 发送到任务节点, 执行完成后汇总

**分布式系统特性与衡量标准**

高性能: 更大流量的处理能力，更低的响应延迟

易扩展: 系统在迭代新功能时，能以最小的代价去扩展，

系统遇到流量压力时，可以在不改动代码的前提下，去扩容系统

高可用: 平均故障间隔/(平均故障间隔 + 故障恢复时间)\*100% = 可用性

**单体架构:**

小型应用优势

易于开发,测试

易于部署: 只需要将该WAR部署在合适的运行环境中即可。

易于扩展: 在负载均衡器后面运行多个应用副本实现扩展。

应用变大弊端

开发: 代码量巨大, 臃肿, 耦合度高, 熟悉代码成本高, 修改易影响其他模块

模块边界不清晰, 团队协作责任划分不明确, 分支管理繁琐

无法对CPU / 内存密集型组件单独伸缩

技术栈更换风险大, 无法对特定模块使用最适用框架、编程语言

部署: 任何修改必须整体部署,会中断如后台任务, 风险大, 失败必须整体回滚, 启动时间长

生产: 某部分流量增大 或 故障会导致整个系统瘫痪

**微服务架构:**

优势

开发: 各微服务代码隔离, 量小, 易阅读, 耦合度低, 利于团队划分

可针对微服务特点使用最合适的框架、编程语言

技术栈更换不会影响其他微服务

部署: 通过docker部署便捷, 符合敏捷开发

生产: 故障不会照成应用整体不可用

问题:

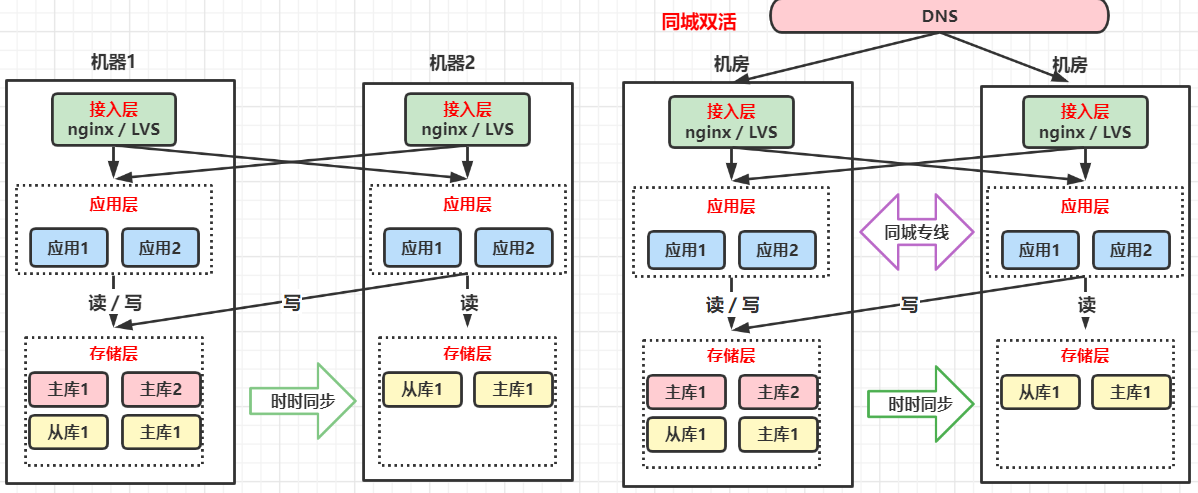
开发: 跨服务功能开发团队沟通成本增加, 测试难度增加, 服务间通信、分布式锁、事务

部署: 复杂度增高, 硬件资源成本增加

生产: 日志收集, 链路长排查问题难度增加

**异地多活**

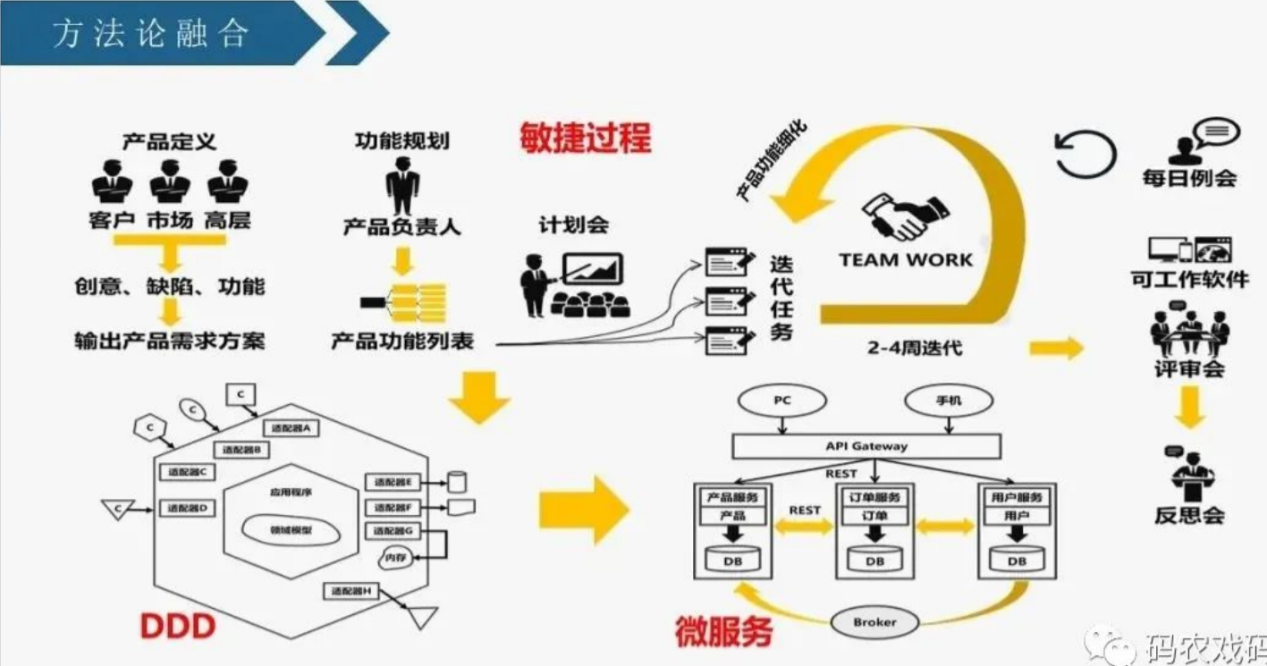
故障解决: 机器冗余 机柜冗余 机房冗余



**同城双活**: 使用同城专线机房间数据同步延时,可忽略

**伪异地双活**: 即同城双活架构机房在不同城市,上海和北京使用跨城专线,网络延时不可忽略

# DDD:



领域:      业务范围、边界, 也称业务问题域

子域:       领域业务范围进一步划分, 对应领域部分业务问题阈。

即核心思想就是将问题域逐步分解，降低业务理解和系统实现的复杂度。

子域根据自身重要性和功能属性可划分为三类子域：核心域、通用域和支撑域。

核心域:  决定产品和公司核心竞争力的子域

通用域:  被多个子域使用, 无太多个性化的诉求的子域。 如: 认证、权限

支撑域:  如:

通用语言:   通过团队交流达成共识的，能够简单、清晰、准确描述业务涵义和规则的语言, 用来确保业务需求的正确表达和理解

限界上下文: 用来确定通用语言语义所在的领域边界

聚合:

聚合根:

实体:

值对象

业务对象-领域对象-代码对象

短信,邮件,图片鉴权下载且被规定只能在后台服务器下载 需单独拆分出来,物理分离,更好维护宕机不影响主业务

**美团支付分层：**

api 提供对外接口

facade 对外接口实现 开关，参数值校验

应用服务层 Factory创建domain对象，Repository[r ɪˈpɒ zə tri]持久化，支付校验，支付

domain

repository

基础设施层 mq rpc reids utils等

启动配置层