redis

1. 什么是redis？

高性能 分布式 内存数据库（key/value）

丰富数据类型 string list hash set zset

支持持久化

支持事务

支持发布订阅

集群高可用（一主多从 哨兵模式）

1. Redis单线程这么快？

数据在内存中处理

采用单线程，避免多线程造成上下文的切换

1. 数据类型？

string list hash set zset

1. 常用命令？

select 0

Flushdb

Flushall

Info replication

Slave of

Slave no one

Key:

keys \* 展示所有key

exists k1 k1是否存在

type k1 k1数据类型

rename k1 k11 重命名k1

del k1 删除k1

persist k1 移除过期时间 设置为永久

expire k1 20 设置过期时间20s

ttl k1 还剩多少秒过期

String

set 设置值

setnx 当k1不存在时 设置值

setex 设置值和过期时间

mset 设置多对key-value

get 获取值

mget 获取多个值

incr 数字加1

incrby 数字加指定值

decr 数字减一

decrby 数字减指定值

1. 持久化？AOF RDB 首先能共存 加载AOF 再加载rdb

持久化：把内存数据写到磁盘中，防止服务器宕机内存数据丢失

RDB：

一定时间间隔 对内存数据进行快照 存储到dump.rdb文件中，redis重启加载该文件恢复数据

快照触发条件：

手动： 客户端save / bgsave命令 flushall shutdown

自动： 配置文件save m n （多少分钟修改了多少次） 主从复制 slave of 主库执行bgsave

save和bgsave区别？

save 会使redis阻塞，redis无法提供读写服务

bgsave Redis会单独fork一个进程持久化，将内存数据写入到临时文件中，持久化完毕后替换上一次持久化的文件 (RDB持久化的数据是fork发生的时候，之后的内存数据可能丢失)

优点：对数据完整性要求不高 适合大规模数据恢复

缺点：可能丢失数据 fork创建子进程也消耗性能

AOF：

将写操作记录到appendonlyfile.aof 文件中 启动时读取重新执行写命令

redis-check-aof 检查程序 修复文件

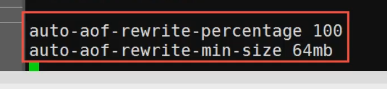
机制？

Always EverySec No

优势：数据完整性好

劣势：AOF 越来越大 需要性能去优化 但有重写

AOF重写：



1. 过期键的删除策略：

定时过期：设置定时器，清除过期key

惰性过期：访问一个key时 判断是否过期

1. 内存淘汰策略？

noeviction：写入操作会报错。

allkeys-lru：移除最近最少使用的key。

allkeys-random：随机移除某个key。

volatile-lru：在设置了过期时间的键空间中，移除最近最少使用的key。

volatile-random：在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：在设置了过期时间的键空间中，有更早过期时间的key优先移除

1. 事务？ 事务中的所有命令 一次性 顺序性 排他性执行

Multi 开启事务

命令入队 不会立即执行

Exec 执行 或者 Discard 放弃事务 队列命令都不会执行

redis单条命令具有原子性 事务不保证原子性 不支持回滚！

命令使用错误 都不会执行

命令使用正确 运行错误（比如String 加1） 其他命令会执行 这条不会执行

支持隔离？

redis单线程，不会去中断事务，总是隔离性的

watch命令是一个乐观锁，监控某一个key(check-and-set)

watch监控一个key,当其他线程修改key的时候，当前事务会感知到 事务不会执行

1. redis发布订阅

订阅频道：subscribe 频道

退订频道：unsubscribe 频道

发布信息：publish 频道 消息

1. 集群

一主多从模式

info replication 查看集群

slaveof ip port

命令行方式：重启主机从机 各自变回主机

一般在从机的配置文件中配置

主从复制原理：

全量复制

从机开启或者slaceof 的时候，发送sync同步数据命令

主机接收到sync命令，开启bgsave生成RDB并且记录写命令

主机发送rdb文件并且记录写命令

主机发送写命令

从机根据RDB替换数据

从机执行写命令



增量复制：主机将写操作发送给从机执行

问题1？

主机断开，主机回来了，依然是主机

从机断开，回来全量复制

问题2？如果从机非常多，都挂了再恢复， 一个主机会有IO宕机危险

引入联级结构

对于联级结构？ （6379 主机<-----6380<-----6381,其中6380既是主也是从）

谋朝篡位 6380 slaveof no one 当主机

主机又回来了 当从机

哨兵模式？ 一主二从三哨兵

哨兵要做的事如下：

将宕机的master下线 (master怎么才叫下线)

找一个slave作为master (怎么找)

通知所有slave连接master (修改配置后，master恢复怎么办)

启动master和slave

哨兵原理：

监控（心跳检测）：哨兵不断的检查主节点 从节点 哨兵间 是否运作正常（默认一秒）

故障转移（投票选举）：如果master挂机了，选举master，slave来连接新的master

通知： 选举结果告诉从节点 或者 配置文件中调用shell脚本通知客户端

具体过程：

每个哨兵每隔1秒向master slaver和其他哨兵 发送ping

如果master回复的时间超过设置时间（down-after-milliseconds），则被该哨兵认为主观下线

该哨兵通知其他哨兵会来确认master是否下线，当大于设置值的哨兵认为master下线，（一般是过半原则） 就认为master下线。选举新的master

哨兵向Slave发送 INFO 命令，让他们修改配置文件 重新认定master 。

哨兵选举算法？

过滤不健康的节点，（在线的，响应快的 与原来master沟通比较久的）

选择slave-priority优先级高的节点

选择复制偏移量大的节点（数据完整）

选择runid最小的节点，值越小 重启的时间越早 数据越全

1. redis做异步队列？

可以 list结构 rpush生产消息 rpop消费消息

缺点：消费者下线 消息丢失

利用发布订阅实现一个消息多个消费

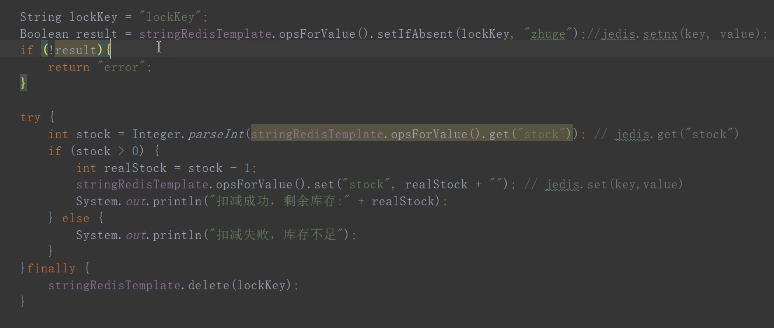
1. redis分布式锁 （针对增删改）

原理：Redis为单进程单线程模式，采用队列将并发的访问变成串行访问

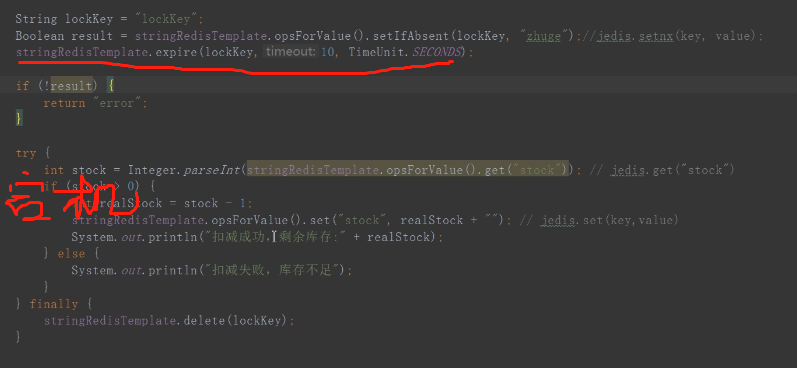
setnx如果返回true说明当前线程能操作redis==加锁（得到执行许可）执行逻辑，最后释放锁

那么setnx如果返回false说明当前线程不能操作redis,别人操作还没释放锁，直接返回错误（此时该线程是不断的去尝试获取锁）

死锁1？try catch finally 让释放锁一定执行

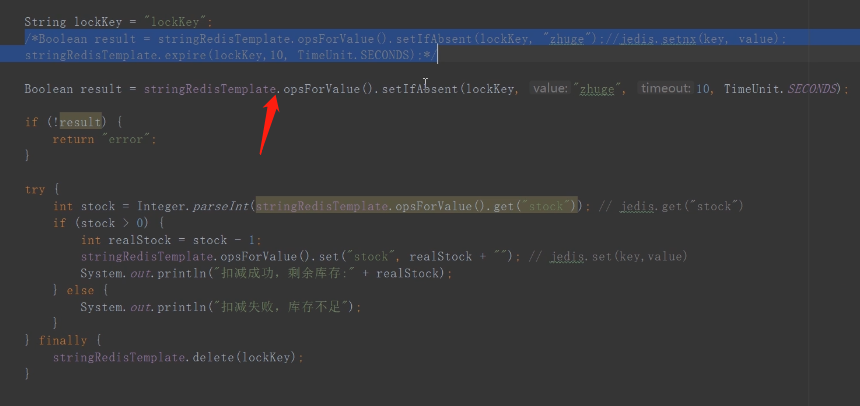


死锁2？服务器宕机 设置超时时间



死锁3？setnx与expire 不会一起执行

获取锁和设置过期时间之间宕机



死锁4？过期时间 < 程序执行时间（误删锁）

比如：过期时间设置的是10秒

A进程a线程执行总共要15秒 B进程b线程执行总共要10秒

当A执行了10秒但还没去释放锁（那么锁已过期），此时B进程b线程可以加锁，当A执行到15去删除的是B加的锁 就尼玛乱套了

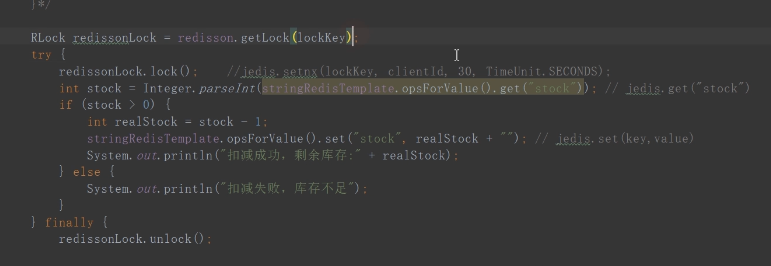
解决：谁加的锁 谁去释放，一般是把线程ID当锁的value,判断后释放锁

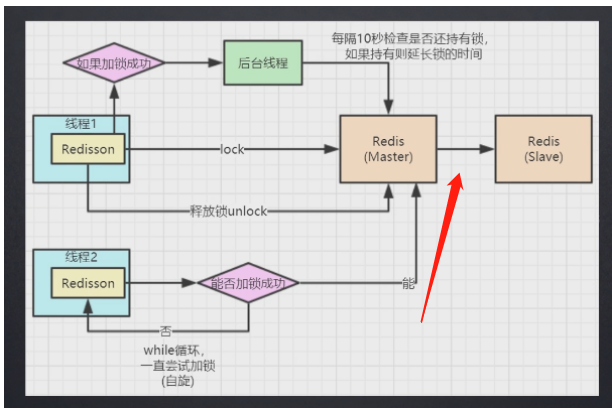


锁续命？ 过期时间的判定

对锁加时间 直到程序执行完

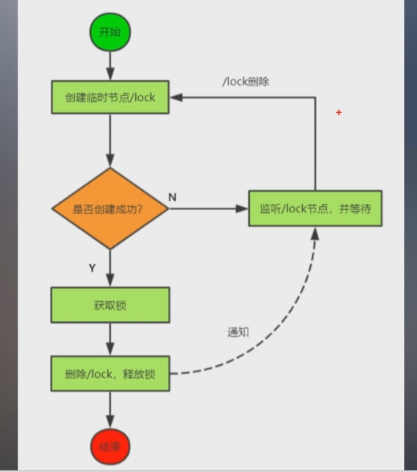
redisson



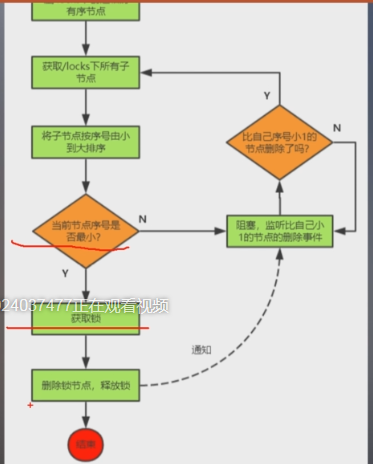


1. zookeeper分布式锁？

原理：基于临时顺序节点 + 监听



优化：解决惊群效应：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



1. 分布式事务？

2pc

3pc

activemq

2PC？

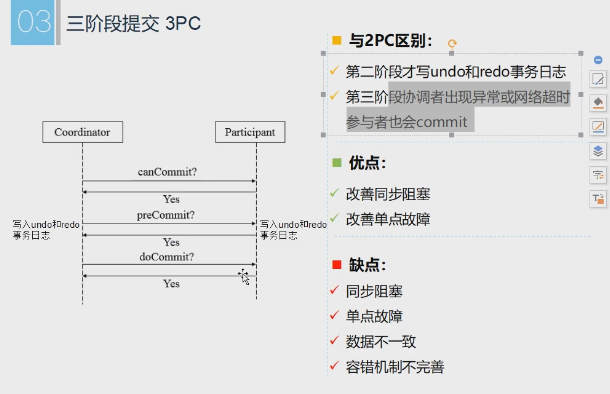


同步阻塞：节点之间等待

单点故障：事务管理器挂了

数据不一致：节点之间有的成功有的失败

3pc?



引入超时机制 优化了同步阻塞问题

1. 分布式sesson?

方案1：粘性session

Nginx根据IP进行hash算法，指定集群中某一台服务器

缺点：当服务器炸了，请求负载到其他服务器，找不到session

方案2：session持久化到redis集群

请求都冲redis集群中获取session

优点：性能高

1. 分布式缓存(针对查询)

缓存穿透是什么？（redis缓存和数据库都不存在数据）

查询一个不存在的数据（缓存和数据库都不存在），由于缓存不命中，然后查询数据库也无果，数据库也不会写入缓存，导致每次都会去请求数据库

解决方案一：缓存空值，并设置过期时间

将数据库查询为空也缓存到redis中，下次查询去访问redis

解决方案二：布隆过滤器：由0,1组成的数组，随机映射函数

布隆过滤器原理：一个值映射到布隆过滤器中，经过这些哈希函数，数组位置设置为1

当判断某个元素在其中？

如果至少有一个为0，说明一定不存在

如果没有为0的，不能说明其一定存在

误判优化：

二维数组越长误报率越小，但浪费空间。

哈希函数的个数，个数越多， 设置 1 的速度越快，起不到过滤作用。

如果个数越少，误判率越高

---------------------------------------------------------------------

对热点数据的访问！！！

缓存击穿：缓存中没有（一般是时间到期），数据库存在，高并发下没读取到缓存，去访问数据库的现象

解决方案：

1 缓存预热的时候热点数据设置永不过期，缺点：占用空间

2 增加双重检测(缓存) + 锁

缓存雪崩：高并发下 大部分key值过期或者缓存服务器炸了 导致缓存失效

解决方案：

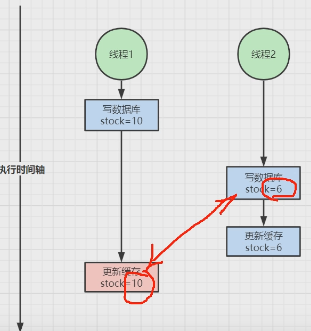
1 保证缓存服务器高可用 redis sentinel 哨兵 或者redis cluster 集群分片

2 缓存预热的时候或者保存到缓存的时候使key的过期时间均匀 （ 固定时间+随机时间）当过期时间

缓存预热：

将热点数据提前放到redis中

缓存与数据库双写一致性？



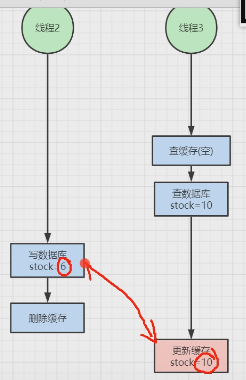
数据库写入成功 缓存写入失败

数据库写入失败 缓存写入成功

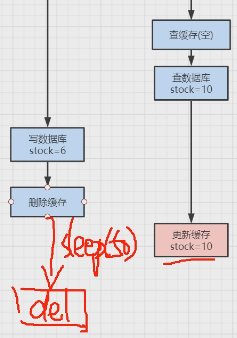
解决：

读取数据的时候 更新redis缓存

写入数据的时候 删除缓存



延时双删？



删除缓存

更新数据库

休眠 读操作的时间+几百ms

删除缓存

redis分布式锁？

使并发成为串行化--->分布式读写锁

mycat

1. 什么是mycat?(架构：keepalived + nginx + 2个mycat)

数据库中间件 连接java应用程序和数据库

作用：

mysql读写分离 ------>解耦（java与数据库）

分库（垂直拆分，按模块分库）

分表（水平拆分，分表）

1. mycat原理？

mycat对sql进行拦截，分片分析，路由分析，发到真实数据库，

对结果合并，聚合，排序，分页，返回给客户端

1. mycat三大配置文件？

schema.xml:定义逻辑库，逻辑表，分片规则

<schema> 逻辑库

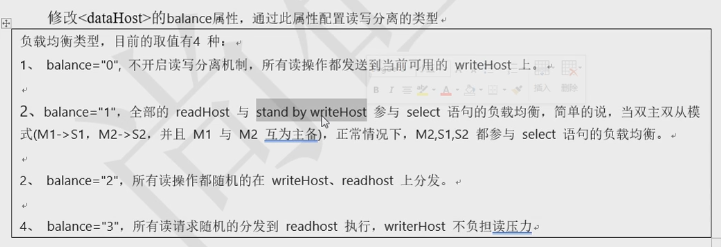
<dataNode> 具体库

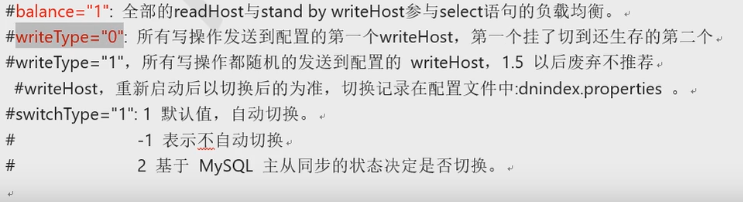
<dataHost> 读写数据库

<writebeat>写库

<readbeat> 读库

读写分离？<dataHost> balance 属性



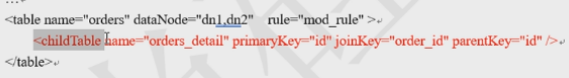


分库：<table name="customer" dataNode="dn2" />

分表：<table name="order" dataNode="dn1,dn2" rule=”PartitionByMod

” />

ER表：相当于join查询 父表和子表分片规则相同 primaryKey joinKey parentKey



全局表：具体数据库都有这数据



全局序列：

方式1：mycat 以时间戳作为id

缺点：时间戳作为id太长

方式2：序列表保存在redis中（利用incr），内存操作快

rule.xml：具体分片策略

取模分片：PartitionByMod

枚举分片：PartitionByFileType

范围分片：PartitionByLong

日期分片：PartitionByDate

server.xml：用户白名单，黑名单 对库，表的操作权限

mongodb

1. 什么是mongodb?

分布式文档数据库，

面向集合，

基于查询表达式的查询功能，

支持索引，

replication集群，

shard分片

特点：内部是JS引擎，发送JS命令，通过JS引擎，操作BSON结构的文档

1. 常用命令

show databases; 展示所有数据库

use database; 选库/创建库 有库就用 没库就创建

db 查看正在使用的数据库

db.dropdatabase() 删除当前使用数据库

db.help() 展示相关方法

show tables;或 show collections; 查看表

db.表名.insert({}) 隐式创建表 添加数据

db.表名.find() 查找表数据

db.表名.find().pretty(); 格式化查看表数据

db.表名.drop() 删除表

分页：跳过(page-1)\*N行 再取N行

db.person.find().skip(3).limit(2)

db.表名.remove(查询表达式，true/false) true:删除一行

db.person.remove({name:”msx”},true) 删除数据

db.表名.update({改谁},{怎么改表达式},{multi:true},{upsert:true})

Multi 默认为false 默认只改一行,true改多行

upsert当没有数据更新时就直接插入

$set 修改某列的值

$unset 删除某列

$rename 重命名字段名

$inc 增加某个列

$setOnInsert 额外添加属性

1. 数据类型

String

Boolean

Integer

Double

Date

1. 索引 默认btree 2.4版本引入hash

查看当前索引

db.person.getIndexes()

索引种类：

普通索引：db.person.ensureIndex({字段：1/-1}) 也会为该字段值为null添加索引，区别于稀疏索引

唯一索引：db.person.ensureIndex({字段：1/-1},{unique:true})

稀疏索引：db.person.ensureIndex({字段：1/-1},{sparse:true}) 不会为该索引字段值为null 添加索引 有值就加，null就不加

哈希索引：db.person.ensureIndex({字段：”hashed”})

优化：

重建索引：db.collection.reIndex() 目的：减少索引碎片，提高索引查找效率

1. 副本集

primary节点 负责写操作，产生oplog日志

Secondary节点 负责读操作,复制主节点oplog日志同步数据

仲裁者：不会同步数据，不能被选为primary, 参与投票

心跳检查

选举机制：

复制集成员向其他节点发送心跳，维护着其他节点的信息

secondary节点等待primary节点回复超时，slave发起选举

大多数原则：

投票成员N，大多数 = N/2 + 1(所以节点最好奇数)

每个节点投一票，达到半数则成为primary

rs.initiate({\_id:"rs",members:[{\_id:0,host:"192.168.224.204:27017"},{\_id:1,host:"192.168.224.204:27018"},{\_id:2,host:"192.168.224.204:27019"}]})

Rs.initiate() 初始化

Rs.status() 查看复制集状态

Rs.isMaster() 查看是否为主节点

Rs.slaveOk() 让secondary节点当复制本，才可以读取数据

Rs.add(host:ip) 系统自动进行了初始化:

Rs.remove(host:ip)

rs.reconfig(cfg)

Rs.help() 查看复制集帮助

Rs.conf()查看初始化配置文件

1. 分片

工作原理：

分片就是把数据分成块，存到不同的服务器上

读写数据请求的时候，先经过mongos这个路由层，mongos路由层去配置服务器（Config Server）请求分片的信息，再来判断这个请求应该去那一台服务器上读写数据。

角色：

配置服务器（Config Server）：存放meta元数据”数据在哪条分片上” 以及分片规则

Mongos:路由层

片：以chunk为单位存数据

片键：分片的规则

手动分片？ 分chuank

声明shop库允许分片

Sh.enableSharding(“shop”)

声明片键

Sh.shardCollection(“shop.goods”,{good\_id:1})

Sh.spliAt(“shop.user”,{user\_id:i\*1000})

fastdfs

1. fastDFS是什么？

Fastdfs是开源 轻量分布式文件存储系统

1. fastDFS原理？

Tracker server

负责管理和调度storage server,

每个storage在启动后会连接Tracker，告知自己所属的group等信息，并保持周期性的心跳

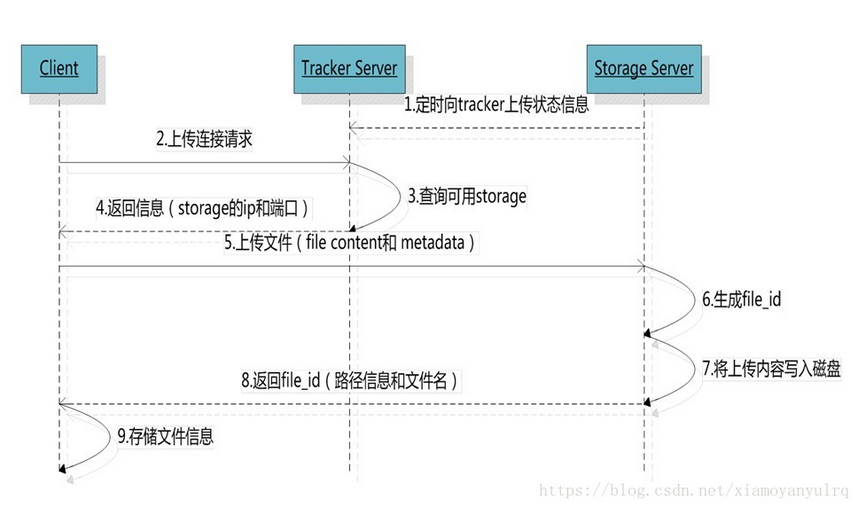
生成元数据信息来提供客户端读写服务

Storage server

存储分多个组(水平扩容=增加组)，组与组之间相互独立，组内数据冗余的方式实现备份

1. fastDFS 上传原理 下载原理？

上传原理：



选择tracker ： tracker集群间对等，任意选择

选择group : tracker按照规则选择group（规则在conf文件中配置）

Round robin 轮训

Specified group，指定某一个确定的group

Load balance，剩余存储空间多多group优先

选择storage:tracker在group 中选择storage server

Round robin 轮训

First server ordered by ip 按ip排序

First server ordered by priority 按storage优先级排序

选择path: storage将会为文件分配一个数据存储目录

Round robin 轮训

剩余存储空间最多的优先

选定存储目录后生成fileid:

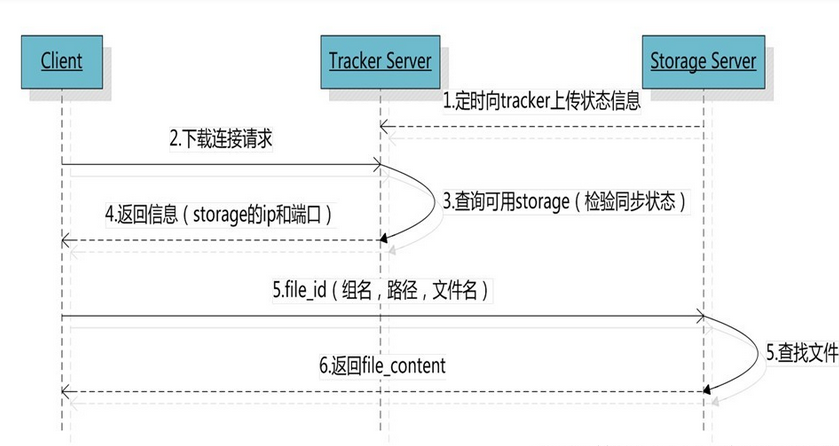
storage server ip、文件创建时间、文件大小、文件crc32的校验码和一个随机数拼接而成，然后将这个二进制串进行base64编码

选择两级目录：

Storage以fileid两次hash 路由到文件目录

文件名

下载原理：



选择任意tracker server

根据文件名解析group 大小 创建时间等信息 来选择storage提供读请求

1. 同步机制？

新增tracker的同步：

前提 tracker和storage的通讯是storage启动发起，为tracker中的每一个启动一个线程

当发现新增的tracker返回的信息少，会将storage最全的信息同步给新增的tracker

新增storage的同步:

Tracker发现新增了storage， 会将tacker所有信息返回给该storage,并把该storage返回给其他同组storage

组内数据的同步？

只在同组内的storage server之间进行同步

源数据才需要同步（以广播方式），备份数据不再同步

特例：新增storage server时，由其中一台将已有所有数据（包括源数据和备份数据）同步到新增服务器

文件同步延迟问题的提出？

文件上传后，由fastdfs 组内异步同步，当还没完成同步，就要去操作非源storage上的问题！！

优先选择源Storage server

zookeeper

1. 什么是zookeeper?

Zookeeper 分布式协调服务，分布式数据一致性解决方案

1. 文件系统（树结构）+监听机制

Znode类型：同一个znode下 节点名唯一

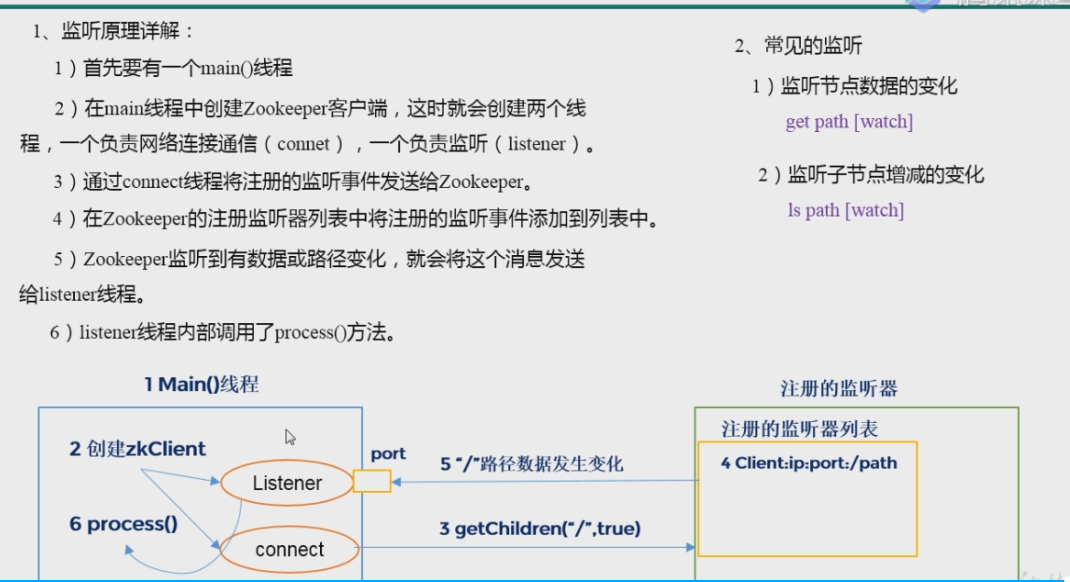
持久有序 客户端与服务端断开，节点依旧存在，顺序编号

持久无序 客户端与服务端断开，节点依旧存在，没有顺序编号

临时有序 客户端与服务端断开，节点依旧不存在，顺序编号

临时无序 客户端与服务端断开，节点依旧不存在，没有顺序编号

监听机制？



1. ZAB协议 zookeeper集群原理？



1. 集群

集群角色？



选举机制？



选举状态：leading following looking

全新集群选举

服务器1启动，给自己投票，服务器1 looking

服务器2启动，给自己投票，服务器1给2,2给1 各自都两票 但服务器2id大，2胜出，但没到达过半原则

服务器3启动，给自己投票，123各自都三票，服务器3id大 胜出 超过半数，成为leading

1和2 成为following

服务器 4 5 启动 都没有用 直接following

非全新集群选举？

* 数据 ID:数据新的 version 就大，数据每次更新都会更新 version。
* 服务器 ID:就是我们配置的 myid 中的值，每个机器一个。
* 逻辑时钟:这个值从 0 开始递增,每次选举对应一个值。 如果在同一次选举 中,这个值是一致的。 这样选举的标准就变成:
* ① 逻辑时钟小的选举结果被忽略，重新投票;

②统一逻辑时钟后，数据 id 大的胜出;

③数据 id 相同的情况下，服务器 id 大的胜出;

1. 命令

Ls /

Ls2 /

Create /.. value 持久节点

Create -e /... value 临时节点

create -e -s /a 111 临时顺序

Get /...

Get /... watch 监听

Set /... value

Delete /..

Rmr /... 递归删除

Quit 退出

1. zookeeper 可以做什么？

duboo+zookeeper zk做注册中心 基于发布订阅

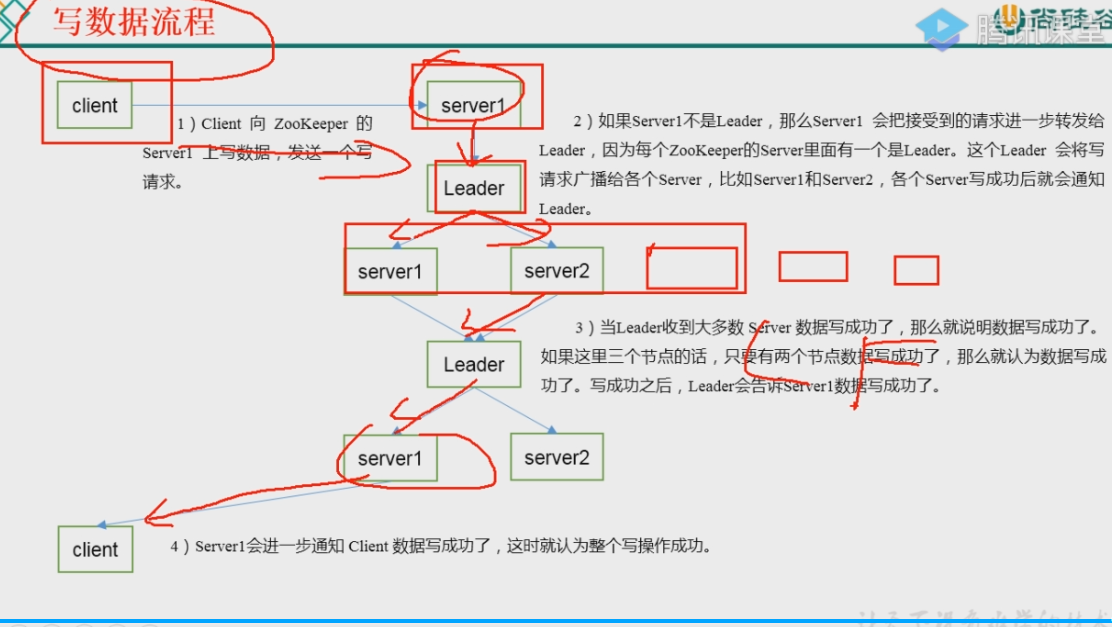
统一配置 基于监听 同一服务的不同提供者配置一致，修改后 通知同步到其他节点

统一命名 利用 zk 创建一个全局的路径，即是唯一的路径，这个路径就可以作为一个名字，指向集群中的集群

一致性：zookeeper 采用了全局递增的事务 Id 来标识

分布式锁：临时顺序节点 + 监听机制

1. zookeeper写数剧流程？



dubbo

1. Dubbo是什么？

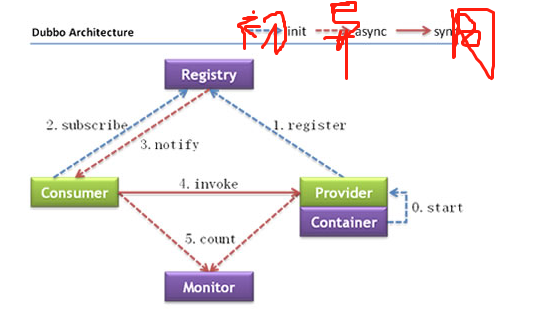
RPC远程服务调用方案 服务治理

面向接口远程调用：consumer只需要拥有provider的接口

负载均衡：软负载均衡（consumer调用哪一个实现，dubbo提供策略）

服务自动注册和发现：默认zookeeper

1. Dubbo运行原理 ？



1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

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

1. RPC速度？

序列化和反序列化 的速度

服务器之间的通讯速度

1. dubbo序列化

hessian fastjson

1. dubbo协议？

dubbo

http

redis

hessian

1. dubbo注册中心

zookeeper redis

1. dubbo配置？



1. dubbo负载均衡策略？



1. 一些配置？

timeout 超时

retries 重试

loadbalance 负载

check 消费者启动不依赖提供者

url dubbo直连

version 灰度发布

虚拟机>dubbo.xml >application.properties>dubbo.properties 覆盖策略

stub 本地存根（消费者自己实现远程接口，利用代理模式，可以做一些判断）

cluster 集群容错策略

failover 自动重试其他服务器

failfast 快速失败

failsafe 直接忽略

mock 服务降级 直接友好返回 或者 调用失败后友好返回

1. dubbo服务提供者失效原理？

zookeeper 剔除临时节点

nginx

1. 什么是nginx?

高性能的http和反向代理服务器 高负载（并发能力强） 动静分离 负载均衡

1. nginx怎么处理请求？

listen监听端口和server\_name 域名匹配server模块，

再匹配server模块里的location(根据不同的url匹配不同的代理proxy\_pass)

再匹配location里面的proxy\_pass

1. 正向代理和反向代理

正向代理：客户端知道代理地址和目标地址 比如 :我-->香港-->google

反向代理：客户端只知道代理地址(nginx),由代理地址去反射目标地址，比如： 我-->百度-->ip

1. 动静分离？

将静态资源(html,css，js)放到nginx,因为nginx是http服务器，资源可以得到访问

1. 负载均衡策略？

轮训（默认）：请求按照时间顺序分配到不同主机 主机挂了 自动剔除

权重: 越大分配越多

Ip\_hash 根据hash结果分配，固定访问一个后端 可以解决session

Fair 根据后台响应时间分配 越短越先

1. nginx 配置文件？

全局块 nginx整体运行配置

worker\_processes 1; nginx处理并发的数量 越大处理并发越多

Events nginx事件块

events {

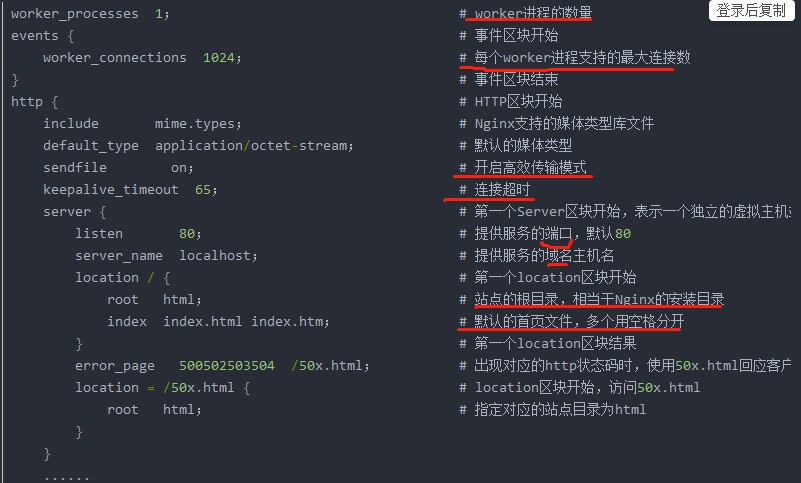
worker\_connections 1024; 与客户端最大连接数

}

http

http全局块

Server块



1. 集群：

Keepalived + nginx

Keepalived+Nginx+Tomcat搭建高可用的Web服务（一）：主备模式

<https://blog.csdn.net/apei830/article/details/78685855>

Keepalived+Nginx+Tomcat搭建高可用的Web服务（二）：双主模式

<https://blog.csdn.net/apei830/article/details/78685885>

1. Keepalived?

VRRP = 虚拟路由冗余

虚拟路由器 = 主路由器+所有备份路由器

选举一个路由做主路由器，对外提供服务，同时ping备份路由

备份路由不对外提供服务，仅仅接收主路由器信号，如果没接收到信号，则认为主路由器崩了 重新选举主路由器

抢占模式 ： MASTER从故障中恢复后，会将VIP从BACKUP节点中抢占过来。

非抢占模式： MASTER恢复后不抢占BACKUP升级为MASTER后的VI

选举机制？

Ip拥有者，如果虚拟IP地址与路由器IP地址相同，则此路由器为IP地址拥有者

比较优先级，优先级大的作为主路由器

比较IP地址，IP地址大的作为主路由器

linux

1. 虚拟网络模式？

桥连接，虚拟机可以跟同网段的设备直接通信 但会ip冲突

Nat: 虚拟机网络地址转换，可访问外网，不会造成ip冲突

主机模式：独立的主机 不能访问外网

1. linux目录结构 : 层级式的树状结构

Bin 存放执行命令

Sbin root管理员执行命令

Etc 配置文件

Usr 用户应用程序

Tmp 临时文件

Opt 安装软件

Var 日志文件

Home 普通用户目录

Root 超级管理员用户目录

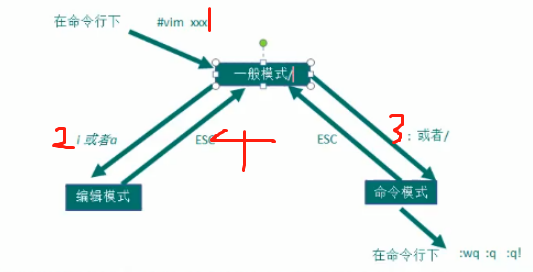
Lib 系统的基本库

Dev 设备管理器

Media 光驱等

Lost+found 系统非法关机会存一些文件

1. vim



1. 开机 关机 重启

shutdown -h now 立即关机

shutdown -h 1 1分钟关机

halt 关机

shutdown -r now 立即重启

reboot 重启

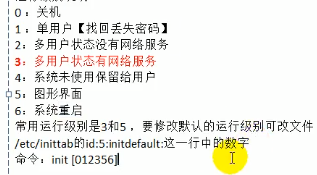
sync 内存数据写到磁盘上

logout 退出远程连接

1. linux启动级别？

配置文件 /etc/inittab

Init 0



1. 帮助命令？

Man 命令 enter往下拉 q 退出 比如： man ls

命令 --help ls --help

1. 用户？

用户目录在/home

用户配置文件在/etc/passwd

组配置文件 /etc/group

密码文件 /etc/shadow

添加用户： useradd 用户名 创建用户名组 用户名

Useradd -d 目录 用户名 创建用户名组 用户名 到指定目录

Useradd -g 组名 用户名 创建某组下的用户

添加密码 passwd用户名

删除用户 userdel 用户名 删除用户但不会删除/home 下面家目录

Userdel -r 用户名 删除用户也删除/home下面家目录

查询用户信息 id 用户名

切换用户 su 用户名

Exit 退出

创建用户组 groupadd 组名

删除用户组 groupdel 组名

修改用户的组 usermod -g 新组 用户名

Chown 用户名 文件名 修改某文件的所有者

Chown 用户名：组名 -R 目录 递归修改目录用户和组

Chgrp 组名 文件名 修改某文件的组

权限？

第1个字母 文件类型 - 普通 d 文件

第2 3 4个字母 用户权限 该用户 r 读 w 写 x 可执行

第4 5 6个字母 组权限 同组的用户 r 读 w 写 x 可执行

第7 8 9 个字母 其他组权限 不同组的用户r 读 w 写 x 可执行

Chmod 修改用户组权限

r=4 w=2 x=1

chmod 777 文件目录 ========== chmod u=rwx g=rwx o=rwx 文件目录

1. 文件目录？

Pwd 查看当前路径

Ls 查看文件

Cd cd ~ 回到家目录 cd / 回到根目录

Mkdir mkdir -p /usr/... 创建多级目录

Rmdir rmdir /usr/... 删除空目录

Rm -rf 删除非空目录 r 递归 f 强制

Touch touch a.txt b.txt 创建文件

Cp /usr/local/a.txt /usr 拷贝文件

\cp /usr/local/a.txt /usr 拷贝文件强制覆盖

Cp -r 源目录 目标目录 拷贝目录

Mv oldname newname 重命名

Mv /usr/a.txt /usr/local 移动文件

Cat -n /usr/a.txt | more 显示行号浏览文件（只读）并以全屏分页

More /etc/profile 文件全部加载 space下一页 enter下一行 q 退出 ctrl + b 上一页

Less /etc/profile 查看大文件 文件一页一页加载 /字符 匹配查找 q退出

Ls -l > a.txt 将查出来的内容 覆盖的写到 a.txt 中

Cat b.txt >> a.txt 将查出来的内容 追加的写到 a.txt 中

Echo “hello” > a.txt 覆盖加入

Echo “hello” >> a.txt 追加加入

Echo + 内容 输出到控制台

Echo $PATH

Head -n 5 a.txt 查看前5行

Tail -n 5 查看后5行

Tail -f 文件 查看文件的更新

Ln 软连接 相当于window 桌面

Ln -s /root hha 在当前目录下创建hha 指向 /root

Rm -rf hha 删除软连接

History 查看历史命令

！编号 执行历史命令

Date

date “+%Y-%m-%d ”

date “+%Y-%m-%d %H:%M:%S” + 不能少

date -s “2021-04-18 13:05:40” 设置时间

cal 2021 显示日历

搜索查找

Find /home -name hello.txt 查找home目录下hello,txt文件

Find /home -name \*.txt 查找home目录下\*.txt文件

Find /home -user xiaohong 查找home目录下用户为xiaohong的文件

Find /home -size +20M 查找home目录 下超过20M的文件 +大于 -小于

yum install mlocate 安装locate

updatedb

locate hello.txt 先创建数据库 快速定位到hello.txt

Grep 筛选

Cat -n hello.txt |grep yes 查看hello.txt中的yes -n显示行号

解压指令？

Gzip hello.txt -----> 压缩 hello.txt.gz 且 hello.txt 不会保留了

Gunzip hello.txt.gz -----> 解压 hello.txt.gz 且 hello.txt.gz 不会保留了

Zip -r my.zip /home 将/home下所有文件递归压缩 成myzip.zip

Unzip my.zip -d /home 将myzip.zip解压到/home文件夹下

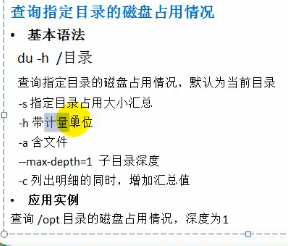
Tar -zcvf a.tar.gz a.txt b.txt 将a b 压缩成a.tar.gz

Tar -zxvf a.tar.gz -C /home 解压到/home目录

Linux 磁盘查询？

Df -lh 查看系统磁盘

Du -h 目录 查看某目录磁盘占用



du -ach --max-depth=1 zookeeper/

进程管理？

Ps -aux | more



Ps -ef|grep tomcat

Top 监控进程情况

杀掉进程？

Kill -9 进程号 强制杀掉

Killall 进程名 把该进程下的所有干掉

服务管理？

Systemctl start stop restart reload status

shell

Shell 完成一个任务的所有命令依次执行

Vim myshell.sh

#!/bin/bash 定义解析器--> 由/bin/bash 去解析 cat /etc/shells 查看解析器

# 注释

#Author A 作者

#Create 2021/05/02 时间

#version 版本

#执行代码

Echo “hello”

#给权限

Chmod 700 myshell.sh

#执行

./myshell.sh 或者 sh + 脚本名字 或者 bash + 脚本名字

变量：

设置变量

Set

echo "hello world"

echo "PATH=$PATH"

#定义变量

A=100

#使用变量

echo "A=$A"

#注销变量

unset A

echo "A=$A"

#定义静态变量 unset 静态变量会报错

readonly B=100

echo "B=$B"

#将命令的返回值给变量

C=$(ls -l /home)

D=`ls -l /home`

echo "C=$C"

echo "D=$D"

#环境变量

E=/usr/local

export E

#获取参数

echo "$0 $1 $2"

#获取所有参数

echo "$\*"

#获取所有参数

echo "$@"

#获取参数个数

echo "$#"

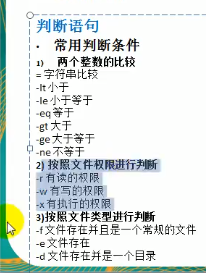
#shell运算符

E=$((2+3\*4))

F=$[2+3\*4]

G=$[$E+$F]

#判断



#判断字符串

if [ "ok" = "ok" ]

then

echo "相等"

fi

#if判断

if [ $1 -ge 60 ]

then

echo "及格了"

elif [ $1 -lt 60 ]

then

echo "不及格"

fi

#case判断

case $1 in

"1")

echo "周一"

;;

"2")

echo "周二"

;;

\*)

echo "哈哈"

;;

esac

#for循环

for i in "$\*"

do

echo "the num is $i"

done

for j in "$@"

do

echo "the num is $j"

done

SUM=0

for((i=1;i<=100;i++))

do

SUM=$[$SUM+$i]

done

echo "$SUM"

#while循环

SUM=0

i=0

while [ $i -le $1 ]

do

SUM=$[$SUM+$i]

i=$[$i+1]

done

echo "SUM=$SUM"

#读取控制台输入

read -t 10 -p "请输入一个值num1=" NUM1

echo "你输入的值=$NUM1"

#自定义函数

SUM=0

function getSum(){

SUM=$[$N1+$N2]

echo "和是$SUM"

}

read -t 10 -p "请输入一个值N1=" N1

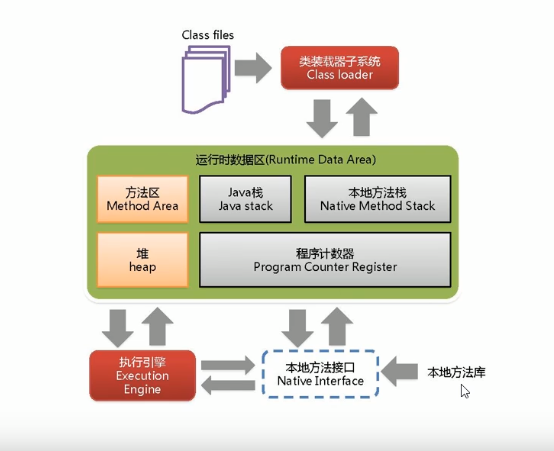
read -t 10 -p "请输入一个值N2=" N2

#调用

getSum $N1 $N2

jvm

1. jvm架构



1. 类加载机制？

加载class文件到内存：

加载：导入class文件

验证：检查class文件的正确性

准备：为类的静态变量分配内存

解析：

初始化：对静态变量和静态代码初始化

1. 类加载方式

显示加载类： class.forname

隐式加载类: new 对象的时候

1. 类加载器

启动 类加载器（bootstrap） jre/lib/rt.jar

扩展 类加载器 (extension) jre/lib/ext/\*.jar

应用程序 类加载器（appclassloader）

自定义 类加载器 (继承java.lang.ClassLoader)

1. 双亲委派机制？ 防止污染jkd源码 (虽然你写了 但我不加载)

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载

1. naive 本地方法接口 本地方法库 本地方法栈

在本地方法栈中运行native方法 本地方法库作支持

实现jvm虚拟机 与 底层系统的交互

比如：thread.start() --->start0() java运行在jvm上，无法创建线程

1. pc寄存器？

每个线程都有自己的pc寄存器 标记线程执行到哪

1. 方法区

存储类的模板信息（字段 构造方法 普通方法）

存储常量池

1. 栈

运行方法 每执行一个方法就产生一个栈针 先进后出 栈顶部就是当前执行方法 执行完方法 就出栈

存储基本数据类型 对象引用 局部变量 实例方法

void A(){

A()

}

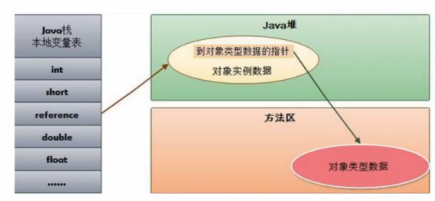
java.lang.stackOverFlowError

1. 栈 堆 方法区关系

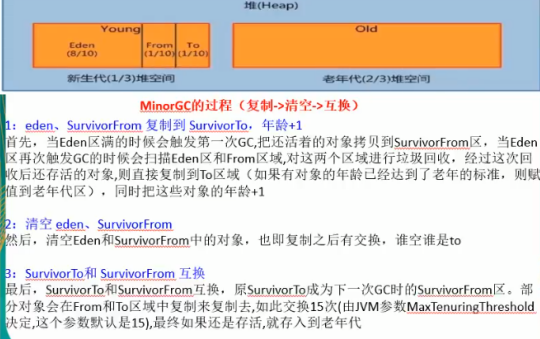
栈存储对象地址

堆存储方法区元数据的地址

方法区存储模板元数据



1. 堆



新生区(1/3)

伊甸区(1/3 的 8/10)

幸存0区（from 1/3 的 1/10）

幸存1区（to 1/3的1/10）

老年区(2/3)

新生对象进入eden区，当eden区满了，触发Minor GC（扫描eden区）,存活的对象移动到幸存from区,清空eden区

当eden区再次满了，触发Minor GC（扫描eden区和幸存from区），存活的对象复制到幸存to区，清空eden和from区

from 区变为to区 to区变为from区

当多次MinorGC仍然对象存在(默认15次交换)，进入老年区

当老年区满了，发生fullGC,

当实在清理不了老年区，OOM（堆内存溢出）

注意：对象一般是在eden区生死，一般扩大eden区调优

注意：gc会停止所有线程，尽量不要发生gc()

永久代存放JDK自带的Class元数据，不会被GC，关闭jvm才会被释放

1. 垃圾回收算法：system.gc（） 不会立即执行

引用计数法

多一次引用 计数器加一 少一次引用 计数器减一 减到0 就回收

缺点： 维护计数器消耗性能 难以处理循环引用

复制算法： 年轻代 整个from区都是垃圾 to区为空

发生YGC的时候 from区到to区

不会产生 内存碎片 （复制过程本身浪费空间）

标记清除：老年代，先标记 后清除 （有碎片）

标记整理：老年代，先标记，后压缩（没碎片）

1. JVM调优

| **参数** | **描述** |
| --- | --- |
| -Xms | 堆内存初始大小 |
| -Xmx（MaxHeapSize） | 堆内存最大允许大小 |
| -XX:PermSize | 非堆内存初始大小，一般应用设置初始化200m，最大1024m就够了 |
| -XX:MaxPermSize | 非堆内存最大允许大小 |
| -XX:NewSize（-Xns） | 年轻代内存初始大小 |
| -XX:MaxNewSize（-Xmn） | 年轻代内存最大允许大小，也可以缩写 |
| -XX:SurvivorRatio=8 | 年轻代中Eden区与Survivor区的容量比例值，默认为8，即8:1 |
| -XX:NewRatio=4 | 设置年轻的和老年代的内存比例为 1:4 |

并发编程

1. 并发编程

充分利用多核cpu，但会有 线程安全，死锁，上下文切换等问题

解决：

JDK Atomic开头的原子类、

synchronized

LOCK

Volatile

JUC

1. 多线程三大特性？

原子性：多个操作同时成功或同时失败 (i++)--->JDK Atomic开头的原子类、synchronized、LOCK

可见性：对共享资源的修改，其他线程立刻看到 (synchronized、volatile)

有序性：程序执行的顺序按照我们自己代码的先后顺序执行（volatile）

1. 并行 并发 串行

并发：同一个cpu, 多个线程按时间分片交替执行

并行：多个cpu处理多个线程

串行：多个任务依次执行

1. 进程与线程

进程：一个应用程序

线程：进程中的一个执行单元，一个进程可以运行多个线程，同一进程多个线程可共享数据

区别：

进程是操作系统分配资源的单位，资源开销大，进程之间不会影响

线程是处理器执行任务的单位，资源开销小，线程死掉 进程死掉，线程依赖进程存在，共享进程资源

1. 上下文切换

多个线程获取cpu时间片执行，时间片用完 切换线程

1. 守护线程 用户线程

守护线程：是后台线程，所有用户线程结束，守护线程结束(所以不能再finally中停止) 比如gc

方式：setDaemon(true)必须在start()方法前执行

用户线程：执行具体任务 比如main

应用程序至少两个线程：mian gc

1. 创建线程四种方式

继承thread类，重写run(),调用start()启动线程

实现runnable接口 重写run()，调用start()启动线程

实现callable接口

callable作为构造参数传给futuretask futuretask继承runnable接口

线程池

1. 继承thread 和 实现runnable接口？

采用实现runnable接口方式：

避免java的单继承 可扩展性

线程池放入runnable callable 不放入thread

1. run()和start()

Start() 启动线程并让线程进入就绪状态，当分配到时间片就开始运行执行run()，只能调用一次

Run() 就是普通方法，不会去启动线程，可重复调用

1. runnable 和 callable

Runnable 接口 run 方法无返回值；

Callable 接口 call 方法有返回值，配合FutureTask可以用来获取异步执行的结果

Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；

Callable 接口 call 方法允许抛出异常，可以获取异常信息

Callable 可以监控线程 FutureTask.get() FutureTask.isdone() FutureTask.cancel()

注：Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

1. 线程状态

新建(new)：新创建了一个线程对象。

可运行(runnable)：当调用start()方法，线程就绪状态，等待获取cpu的使用权执行。

运行(running)：可运行状态(runnable)的线程获得了cpu时间片（timeslice），执行程序代码。

阻塞(block)：放弃对 CPU的使用权，停止执行，

阻塞的情况分三种：

等待阻塞： wait()方法

同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)

其他阻塞: 通过调用线程的 sleep()或 join()

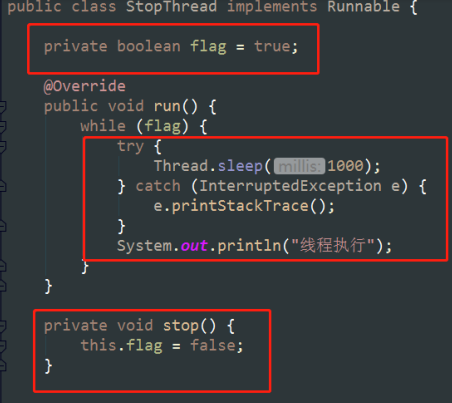
死亡(dead)：线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

1. 如何停止一个线程

stop() 可能造成死锁

interruot 打断线程

标记退出



1. 线程的一些方法

Wait（object） 使一个线程处于等待（阻塞）状态，释放所持有的对象的锁

线程不会自动唤醒，notify() notifyall()唤醒

Sleep（thread）使一个线程处于等待（阻塞）状态，不释放所持有的对象的锁

线程自动唤醒

notify 唤醒一个等待线程 不确定唤醒谁 由jvm调度

Notifyall 唤醒所有等待线程 竞争锁才能进入就绪状态

Yield: 是一个运行线程处于就绪状态，运不允许行看系统分配

Join(): 插队，让线程有序执行 main 里面 线程A A.join() A先执行 main后执行

Interrrupt: 中断线程，不会去停止线程，一旦终止 抛出异常 做出处理

1. sleep() 和 wait()区别？

类不同 sleep() thread 类的静态方法 wait() Object类的方法

是否释放锁 sleep() 不释放锁 wait()释放锁

是否自动唤醒 sleep()自动唤醒 wait() 通过notify() notifyall()

1. wait notify notifyall 必须放在同步方法，同步代码块中？

释放锁 唤醒锁 首先前提是 自己必须拥有对象的锁 进入到syschornized 肯定是拿到了对象锁

1. thread类的sleep() 和yield()为什么是静态的？

Thread 类的 sleep()和 yield()方法将在当前正在执行的线程上运行，不可能去new一个新线程 然后调sleep() yield()

1. 死锁？

A线程持有资源1 B线程持有资源2

A线程试图去持有资源2 B线程试图去持有资源1

A B线程相互阻塞

1. 线程通讯？

生产者与消费者：判断 等待 执行 通知

生产者 判断-->等待-->生产-->通知

消费者 判断-->等待-->消费-->通知

通讯方式：

syncrhoized同步块**Object类**的wait()/notify()/notifyAll()

虚假唤醒：生产者生产，所有消费者都唤醒了 无论哪一个消费者被唤醒了 其他消费者的唤醒是没有必要的

解决： 判断 if -->while，虽然所有消费者被唤醒了 但是会循环判断条件，不成立还是不会执行业务

reentrantLock **Condition类的**await()/signal()/signalAll()

精准唤醒：多个c**ondition实例 达到精准唤醒的效果 控制程序顺序**

1. 线程安全？

Synchronized? 重锁 可重入锁

// 修饰静态方法 锁定的是当前类的 Class 对象  
// 修饰实例方法 锁定的是当前实例对象

// 代码块加锁 锁定指定对象

原理：

字节码指令Monitorenter ：加锁

JVM会创建monitor对象

Owner:拥有锁的线程

Recursions:计数器

指令Monitorexit:释放锁

计数器减一 减到0的时候释放锁

指令Monitorexit:释放锁

防止同步块代码异常导致锁没被释放

1. synchronized可重入的原理

重入锁是指一个线程获取到该锁之后，该线程可以继续获得该锁。

底层原理

维护一个计数器，当线程获取该锁时，计数器加一，再次获得该锁时继续加一，释放锁时，计数器减一，当计数器值为0时，释放锁

1. 自旋锁？

Synchronized里面代码比较少 很快执行，没必要去线程切换

While循环不断尝试获取锁

如果Synchronized里面代码比较多，这样自旋会很消耗cpu

1. volatile：修饰变量 在指令前后都加一层内存屏障保证指令顺序

保证可见性

不保证原子性

禁止指令重排

1. 原子类来保证原子性

java.util.concurrent.atomic 使用CAS保证原子性

1. CAS

AtomicInteger底层实现

public final int incrementAndGet() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

1.先获取当前的value值

2.对value加一

3.调用compareAndSet方法

4.调用unsafe的compareAndSwap方法

内存位置

预期值

新值

如果内存位置 和 预期值一样 内存的值更新为新值

如果不等 线程就会发生自旋 等待下次线程执行

CAS产生问题？

ABA：1线程取出来A 2线程取出来A改成B在改成A，1线程CAS成功

自旋浪费CPU资源

1. LOCK

reentrantlock

公平锁 : 多个线程按照申请锁的顺序去获得锁，谁先申请谁先拿到锁

非公平锁： 多个线程去获取锁的时候，会直接去尝试获取锁

可重入锁：同一个线程可多次获取锁

比如：A线程5s B线程1S 先执行B 提高吞吐（先易后难）

trylock()

lock()

unlock

condition: 线程通讯 可以精确唤醒 控制执行流程

ReentrantReadWriteLock

读锁 写锁 可重入锁

对于读操作 允许并发

对于写操作 不允许并发

1. 底层原理AQS?

AbstractQueueSynchronized 抽象队列同步器

AQS内部维护着state变量 初始值为0 和 加锁线程对象 初始为null

lock()加锁，基于CAS 让线程1加锁 state=1 加锁线程对象 = 线程1

其他线程进入等待队列queue中

可重入就是让state累加1

unlock()释放锁，基于CAS 让线程1 state累减1，直到0释放锁，加锁对象=null

然后线程2从等待队列中出队 同上操作

1. lock与synchronized区别？

1 synchronized是Java内置关键字，在JVM层面，Lock是个Java类；

2 synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。

3 synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unLock()去释放锁就会造成死锁。

4通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

5 底层原理不同，synchronized jvm创建monitor对象，lock使用AQS

1. ThreadLocal

不同线程需要不同的变量，比如数据库的connection

是线程局部变量 每个线程都会有变量的一个副本，操作本地变量

原理：每个线程都会创建ThreadLocalMap，key是当前线程 value 泛型值

但是存在内存泄露的风险

Key 弱引用 value 强引用，key会被gc垃圾回收，value不会，就会出现 key为null的 ThreadLocalMap 内存泄露

解决内存泄露：调用 set()、get()、remove() 方法的时候，会清理掉 key 为 null 的记录

1. 集合类不安全？

ArrayList 不安全 add方法都没有 synchronized 并发修改异常

方案1：vector 底层采用synchronized

方案2：Collections.synchronizedList(new ArrayList()) 底层采用synchronized

方案3：CopyOnWriteArrayList 底层采用Lock锁，写的时候复制，扩容，元素放在最后

set不安全 add方法都没有 synchronized 并发修改异常

方案1：Collections.synchronizedSet(new ArrayList()) 底层采用synchronized

方案2：CopyOnWriteArraySet 底层采用Lock锁，写的时候复制，扩容，元素放在最后

map不安全

方案：concurrentHashMap

1. 工具类

CountDownLatch c = new CountDownLatch (6); 减法计数器（不可重置，线程之间不等待）

c.countDown();

c.await(); 减一

原理：底层AQS

创建计数器交给AQS的state,然后await阻塞，直到计数器为0

countDown() 计数器减一

直到为0,执行想要的线程

# CyclicBarrier 加法计数器（可重置，线程之间相互等待）

CyclicBarrier c = new CyclicBarrier(6,new runnable{})

c.await() 加1

原理：计数器加1 直到6，执行想要的线程

Semaphore 控制线程数量

Semaphore s = new Semaphore(3);

s.acquire() 来一个线程 信号量加1 直到3之前 阻塞

s.release() 走一个线程 信号量减1 直到0之前 阻塞

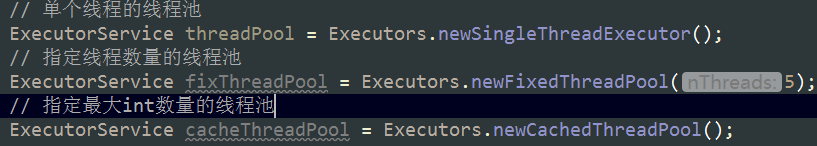
synchronizedQueue队列



往队列里面放值 阻塞直到把值从队列里面取出来

1. 线程池

减少线程的新建和销毁，管理线程 限制最大并发



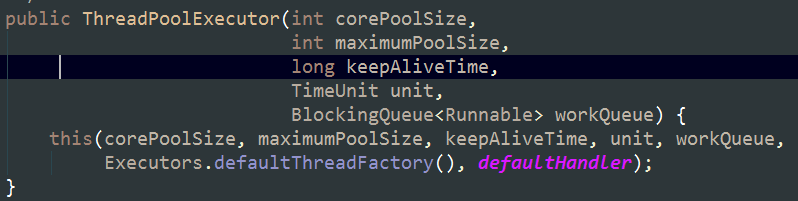
1. excute() 和 submit()区别？

excute只能运行runnable

submit运行runnable 和 callable 底层还是调用excute 可以异步得到线程执行结果

1. ThreadPoolExecutor

七大参数



corePoolSize 核心线程数

maxPoolSize 最大线程数

keepAliveTime 空闲线程存活时间

TimeUnit 控现线程存活时间的单位

BlockingQueue 阻塞队列

ThreadFactory 线程工厂

Handler 拒绝策略

超过 核心数 + 阻塞队列数--->开启最大线程

超过 最大线程数 + 阻塞队列数 -->拒绝策略

小于 核心数 + 阻塞队列数 达到keepalive时间 --->关闭最大线程

拒绝策略？

AbortPolicy：抛出 异常来拒绝新任务

CallerRunsPolicy： 哪来的 去哪处理

DiscardPolicy：直接丢弃

DiscardOldestPolicy:尝试和最早的执行线程比较 最早的执行线程还没执行完 就丢弃

java

1. JDK 与JRE

jdk 开发工具包

jre java运行环境

jvm java虚拟机

1. == 和equals

==

基本数据类型 比较值

引用数据类型 比较地址

equals

默认比较引用 很多类重写了equals 比如String 比较值

1. hashcode相同 equals不一定相同

因为 hash冲突

1. final finally finalize

final可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。

finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码方法finally代码块

finalize是一个方法，垃圾回收器调用finalize()回收垃圾

1. 8中基本数据类型

byte short int long float double boolean char

1. switch 范围

java1.5前 byte short int char

java1.5后 enum

java1.7 string

1. break continue return

break 结束当前循环

continue 跳出本次循环，继续执行下次循环

return 程序返回

1. 反射

Student stu = new Student(); Class classobj1 = stu.getClass();

Class classobj2 = Class.forName("fanshe.Student");

Class classobj3 = Student.class;

1. string stringbuilder stringbuffer

string 声明的是不可变的对象，每次操作都会生成新的 String 对象

StringBuffer 是线程安全的，性能低 Synchronized

StringBuilder 是非线程安全的，性能高

单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。

1. ArrayList LinkedList Vector

数组和list互转

list转数组 list.toArray()

数组转list Arrays.asList(array)

ArrayList 和 LinkedList区别？

ArrayList 数组 查询快 增删慢 内存占用小

LinkedList 链表 查询慢 增删快 内存占用大 数据+指针

ArrayList和Vector?

Arraylist 不安全 性能高

vector Synchronized 保证线程安全 性能低

list 和set区别？

list 有序 数据可重复 可多个null

set 无需 数据不可重复 只能一个null

1. HashSet 无序唯一 基于Hashmap,HashSet的值存放于HashMap的key上
2. HashMap底层原理？

jdk7 数组 + 链表

put

对key hash算法 再取模 得到在数组中的位置

对于hash算法一样（hash碰撞）的数据进行key的equals判断

如果equals 相同 替换

如果equals不同 以链表相连

get

对key hash算法 再取模 得到在数组中的位置

如果在数组位置上，直接返回

如果在链表上，遍历的去比较equals值

jdk8 数组+ 如果链表长度 > 8 链表变为红黑树（提高查询效率）

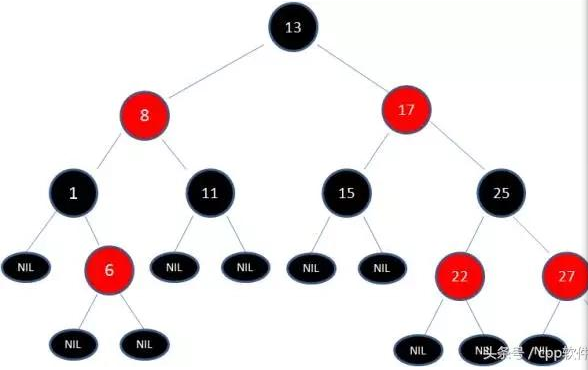
红黑树？（自平衡二叉树 降低树的高度 快速查找）

根节点黑色

叶子节点黑色null

每个红色节点的两个子节点都是黑色

从任意节点到其叶子节点包含相同的黑色节点



hash冲突?

两个值的hash算法得到相同的值，如果key的equals相同则替换 不同就链表存储

hash两个参数？

数组初始值

负载因子：当数据超过数组初始值乘以负载因子 就会扩容（双倍数组初始值）

将原来的元素重新放入新的数组

transient 修饰的属性 不会被序列化

hashtable与hashMap区别？

hashMap的key可以为null HashTable的key不允许

hashMap线程不安全 hashTable 以synchronized做了线程安全

docker

1. 什么是docker

Docker 容器虚拟化技术，对组件（程序，环境）封装，一次封装 到处运行(跨平台)

1. Docker组成？

容器（container）: 镜像创建的实例

镜像（image）: 容器模板 创建多个容器 run命令

仓库（repository）：（Ducker hub） 配置阿里镜像加速

1. Docker怎么工作？

基于 client-server结构， docker启动，开始守护线程进程，客户端通过socket连接，发送命令给服务端操作容器

对比虚拟机？

虚拟机 模拟完整的操作系统，包括（内存，处理器，声卡等。。）（资源占用大 启动慢）

Docker docker engin 替代了操作系统（轻量，启动快）

1. Docker命令？

镜像命令

Docker images -a 查看所有镜像

Docker serach tomcat 搜索镜像

Docker pull tomcat 拉取镜像

Docker pull tomcat:8 拉取指定版本

Docker rmi --force tomcat 删除

Docker rmi --force tomcat:8 删除指定版本

容器命令？

Docker run -it 镜像id 以终端方式打开容器并进入容器

Docker run -d 镜像id 后台守护运行不进入容器（但容器会自动退出）

Docker ps 显示上一次运行的容器

Exit 退出并关闭容器

Ctrl + p + q 退出不关闭容器

Docker rm 容器id 删除容器

docker rm -f 容器id 强制删除

docker start 容器id

docker restart 容器id

docker stop 容器id

docker logs -tf --tail 10 容器id

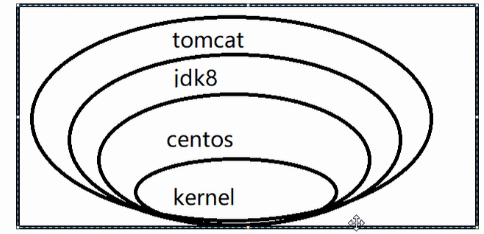
Docker exec -it 容器id /bin/bash 在宿主机上操作容器

docker attach 容器id 进入容器

Docker cp 容器id:文件路径 主机文件路径 把容器某文件拷贝到linux系统

1. UnionFS（联合加载文件）:文件系统层层叠加

每一层在宿主机上保存一份，可共享（比如：已经下载tomcat 再去下载tomcat:8就不会去下载jdk8）



1. 容器的数据卷？

数据从容器到主机：数据持久化

docker run -it -v /myhostdataV:/mycontainerdataV 容器id

1. DockerFile?



关键字：

From 基于哪个镜像 （scratch 祖先）

Maintainer 作者

Run 执行命令

Expose 暴露端口

WorkDIR 指定进入容器的目录

ENV 定义变量

Add 拷贝 压缩

Copy 拷贝 copy src源 dest目的地

VOLUME 数据卷 持久化工作

CMD 启动命令 会被docker run 的参数替换

EntryPoint 启动命令 docker run 的参数也是追加

onbuild 构建触发

使用：

docker build -f dockerfile -t 镜像名 .

docker run -it 镜像id

docker push

mysql

1. mysql binlog三种

statement 记录写操作 同时记录环境信息，一些函数无法复制

row 记录写操作 和大量日志信息

mixed 普通操作使用statement 复杂操作使用row

1. MyISAM 和 innodb对比以及选择（存储引擎修饰表）？

存储结构

myism每张表被存放在三个文件：frm-表格定义、MYD(MYData)-数据文件、MYI(MYIndex)-索引文件

innodb 数据和索引保存在 .ibd 文件中

myism采用B+树 叶子节点上存储指针 需要回表操作 非聚簇

innodb采用B+树，叶子节点上存储数据，不需要回表操作 聚簇

Myism 不支持外键 不支持事务 表级锁

Innodb 支持外键 支持事务 表级所和行级锁

1. 索引是什么，优缺点？

索引是排好序的数据结构，存储数据的引用指针

加快查询但增删改索引需要空间和时间

1. 索引的种类

主键索引：数据不重复 不为null 一个表只有一个主键索引

唯一索引：数据不重复 可以null 一个表可以有多个唯一索引

复合索引：对多个列进行索引

全文索引：

哪些情况？

主键--->唯一索引

频繁查找的字段适合

哪些不要？

表记录太少不适合

表记录经常增删改

列的值 就几个 例如：性别

一般选择Where Order by Join...on 后面的

1. B树 红黑树 B+树 hash?

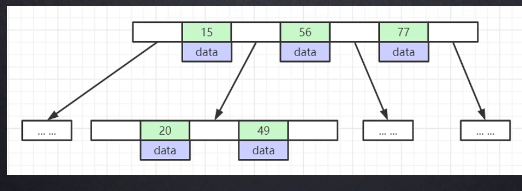
（B树 二叉树） 小的放左边 大的放右边 存储索引和指针

缺点：当递增的数据，基本上没什么用

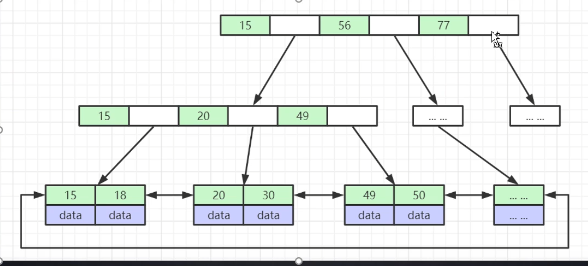
红黑树（二叉平衡树）存储索引和指针

缺点：树高度还是高 i/o操作还是多

B树



B+ 树



B树 b+树区别？

B+树维护双向指针 方便定位

B+树 冗余索引 把数据放在最底层叶子节点，树的高度变小 减少i/o操作

Hash 数组+ 链表（放指针）

1. 最左前缀？

对于联合索引，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，之后的索引就会失效

1. 事务：一组操作要么同时成功要么同时失败

ACID

原子性：事务不可分割，要么同时成功 要么同时失败

一致性： 执行事务前后，数据保持一致

隔离性： 事务之间互不干扰

持久性：数据的改变是持久的

脏读 ：A事务修改了 还没提交 B事务就读取了该数据 A事务回滚

不可重复读：一个事务读取了两次数据不一致，可能是B事务修改了

幻读：一个事务读取了两次数据行数不一致

读未提交：允许读取尚未提交的数据 导致脏读 不可重复度 幻读

读已提交：允许读取已提交的数据 导致不可重复读 幻读

可重复读：多次读取结果一致 导致幻读

串行化：事务逐个执行

1. 锁

mysql锁分读锁和写锁。

读锁是共享的，只能读不能写。

写锁是排他的，不能读和写。

表锁锁住整张表

行锁锁住行 悲观锁可以通过for update实现，乐观锁则通过版本号实现

1. sql优化

In 和 exist区别！！

SELECT   \*  FROM A WHERE id IN (SELECT id FROM B);

等价于：1、SELECT id FROM B ----->先执行in中的查询，缓存起来

              2、SELECT \*  FROM A  WHERE A.id = B.id --->遍历A表 id 在缓存的数据

结论：IN()适合B表数据比A表数据小的情况

SELECT \* FROM a WHERE EXISTS(SELECT 1 FROM b WHERE B.id  = A.id);

1、SELECT \* FROM A;

2、SELECT 1 FROM B WHERE B.id = A.id;

EXISTS()查询是将主查询的结果集放到子查询中做验证，

结论：exists()适合B表比A表数据大的情况

避免全表扫描 where order by 建立索引

不要在索引上做操作 where left(name,4)

尽量少用\*

使用！= <> 索引失效

Is null is not null 索引失效

Or 会导致索引失效

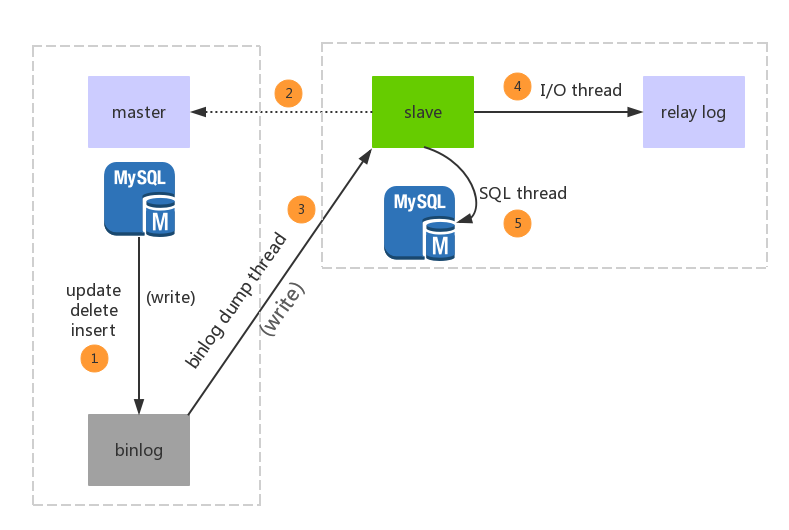
in not in 索引失效

like 索引失效

复合索引 最左前缀原则 第一个字段必须满足

varchar 代替char

1. mysql主从



1. 主库的增删改操作写到binlog文件
2. 主库创建线程把binlog发送给从库
3. 从库创建线程读取binlog写入relaylog
4. 从库创建线程读取relaylog 执行增删改操作
5. sprinboot自动装载？

Springboot启动类，SpringApplication.run（当前类传入），

其中refreshContext(context)刷新容器，解析注解的方式把BeanDefinition加入到容器

@SpringBootApplication

@Configuration 所以也会去解析它

@ComponentScan 扫描启动类目录下的注解

@EnableAutoConfiguration 下

@Import({AutoConfigurationImportSelector.class})

selectImports()从类路径下META-INF/spring.factories的自动配置类加载到容器，

selectImports()里面会去排除一些配置exclude

也会过滤一些配置,@ConditionalOnClass 如果没有引入启动器就不会被加载

springcloud

1. 注册中心eureka 服务治理 服务注册与发现

基于cs架构，EurekaClient 服务者启动，向EurekaServer发送心跳并注册自己的服务信息，EurekaClient 消费者启动定时拉取eurekaServer服务列表，调用服务

如果EurekaServer在多个心跳周期没有接收到心跳 从服务注册表移除服务

@EnableEurakaServer eureka注册中心

@EnableEurekaClient eureka 注册服务

Eureka自我保护机制？

Eureka默认多个心跳周期没有接收到eurekaclient心跳，会剔除

开启自我保护，就不会剔除

1. Ribbon 负载均衡

Nginx是服务端负载，客户端请求交给nginx来转发

ribbon是客户端负载，将注册信息列表拉取到本地进行缓存，本地负载去调用

负载策略？ @RibbonClient(name=””,configuration=MySelfRule.class)

轮训： 第N请求 % 服务集群数

轮训失败重试：轮训失败重试

随机

可用过滤：过滤掉下线的和高并发的，然后轮训

并发量最小：选择并发少的

1. feign服务接口调用

启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在接口中使用@FeignClient声明。

调用方式不同，Ribbon + restTemplate Feign采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可

1. hystrix 熔断器

服务降级 服务忙 fallback一个友好提示

服务熔断 服务达到最大访问，服务不可用 然后一个友好提示 恢复过程

服务限流 控制请求数

启动类@EnableHystrix

方法上 @HystrixCommand(fallbackMethod = “”)

1. gateway 网关
2. config 配置
3. Bus 消息总线动态刷新

Activemq

1. activemq

消息中间件，异步通信 系统解耦 流量削锋

两种方式：点对点queue模式 发布订阅topic模式

点对点：消息生产者向队列生成消息，消息消费者从队里里获取消息

消息没有消费者，消息不被丢弃（持久化）且一个消息只能被一个消费者消费，且消费者还要返回Ack信息（是否消费掉了）

发布订阅：消费者先订阅主题，发布者生成消息，消费者消费消息

消息没有订阅者，消息被丢弃（无状态），如果有多个订阅者，都会收到消息

1. 多个消费者消费不均匀？

生产消息，依次启动多个消费者，会全部被消费者1消费 默认1000条

prefetch设置为1

1. 消息的持久状态策略？

持久在线

持久离线

非持久在线（默认）

Message的DeliveMode

Message的expreation 过期时间 0 代表永久

Message的Priority 优先级 越大越急

1. 消息持久化？

KanDB(默认)基于日志文件的持久化方式

默认情况下会将消息持久化到activemq安装目录下的/data/kahadb目录下

jdbc消息存储 基于JDBC的消息存储 消息持久化到Mysql 生成三张表存储

jdbc+ journal高速缓存 减少与数据库交互

1. Activemq集群 + zookeeper 实现高可用

activemq 1主2从 当主机挂掉 利用zookeeper选举master

1. 异步投递？

默认使用同步发送，producer.send()会阻塞直到broker返回确认

采取异步发送setUseAsyncSend(true)，异步发送不会等待broker的确认

流量问题：

ProducerWindowSize：生产消息会统计消息字节，当达到最大字节即使异步发送也需要等待broker的确认，才能继续发送。

异步丢失：

异步可能造成消息丢失：异步的话，发送者会自认为发送成功 ，这个时候服务器宕机了 就会造成消息丢失

1. 数据丢失 异步签收 或者持久化？

提供者异步的AsyncCallBack(){

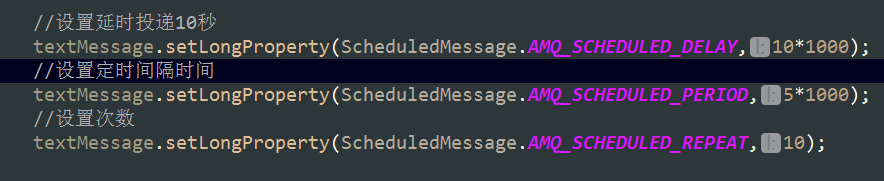
onSuccess()

OnException()

}

1. 延时投递 定时投递





1. 重试机制

 消费者收到消息，没有告诉broker确认收到该消息，broker会尝试再将该消息发送给消费者。尝试n次，如果消费者还是没有确认收到该消息，那么该消息将被放到死信队列中，之后broker不会再将该消息发送给消费者。

①　使用事务又调用了rollback

②　使用事务又没有commit

重发时间间隔：1s  重发次数：6

1. 有毒消息Poison ACK

一个消息被重发）多次（默认6次）时，消费者端会给MQ发一个“poison ack”表示这个消息有毒，告诉broker不要再发了。这个时候broker会把这个消息放到DLQ（死信队列）。

1. 重复消费

保证消息的幂等性

将消息做数据库的insert操作，消息做唯一主键,如果insert成功才会被消费

将消息做redis的set操作，消息消费完毕 set进去，每次消费前先查询，如果有 则不消费

1. 死信队列如何处理

非持久化的消息进入死信队列

持久消息不进去死信队列

过期消息也不要进去死信队列

elasticsearch

1. 什么是elasticsearch?

通过restful风格访问 基于lucene分布式全文搜索引擎

1. 数据格式

index(索引) type(类型) document(文档) field(字段)

1. 倒排索引

正排索引：由主键id 查找文档

倒排索引：首先对文档进行分词，看词条在哪些文档中存在，记录数据id,当输入词条时，找对对应记录id 找到文档

1. ik分词

ik\_smart 最少切分

ik\_max\_word 最细切分

1. RestFull操作

#创建空索引

PUT /test1

#删除索引

DELETE /test1

#查询索引

GET /test2

#查询所有索引

GET \_cat/indices?v

#创建索引

PUT /test2

{

"mappings": {

"properties": {

"name":{"type": "text"},

"age":{"type": "long"},

"birthday":{"type": "date"}

}

}

}

#插入数据

PUT /test2/\_doc/1

{

"name":"msx",

"age":28,

"birthday":"1994-09-02",

"gender":"男"

}

#查询数据

GET /test2/\_doc/1

#修改数据

PUT /test2/\_doc/1

{

"name":"msx",

"age":27,

"birthday":"1994-09-02",

"gender":"男"

}

#修改全部内容

PUT /mydb/user/3

{

"name":"李四",

"age":25,

"desc":"不知道怎么形容",

"tags":["交友","唱歌","炸男"]

}

GET /mydb/user/3

#更新部分内容

POST /mydb/user/3/\_update

{

"doc":{

"name":"李四233"

}

}

#条件查询

GET /mydb/\_search?q=name:李四

#查询单条

GET /mydb/\_search

{

"query": {

"match": {

"name": "李四"

}

}

}

#查询所有

GET /mydb/\_search

{

"query": {

"match\_all": {}

}

}

#结果过滤

GET /mydb/\_search

{

"query": {

"match\_all": {}

},

"\_source": ["age","name"]

}

#排序

GET /mydb/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"age": {

"order": "desc"

}

}

]

}

#分页

GET /mydb/\_search

{

"query": {

"match\_all": {}

},

"from": 0,

"size": 3

}

#条件查询 and

GET /mydb/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"name": "李四"

}

},

{

"match": {

"age": "25"

}

}

]

}

}

}

#条件查询 or

GET /mydb/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"name": "李四"

}

},

{

"match": {

"age": "23"

}

}

]

}

}

}

#条件查询 not

GET /mydb/\_search

{

"query": {

"bool": {

"must\_not": [

{

"match": {

"age": "23"

}

}

]

}

}

}

#过滤器

GET /mydb/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"name": "李四"

}

}

],

"filter": {

"range": {

"age": {

"gte": 10,

"lte": 30

}

}

}

}

}

}

#多条件匹配 match 使用分词器 模糊查询

GET /mydb/\_search

{

"query": {

"match": {

"tags": "男 唱"

}

}

}

#精确匹配

GET /mydb/\_search

{

"query": {

"match\_phrase": {

"tags": "男"

}

}

}

#精确查询 term

GET /mydb/\_search

{

"query": {

"bool": {

"should": [

{

"term": {

"name": {

"value": "李四"

}

}

}

]

}

}

}

#keyword 不会使用分词解析

#text 会使用分词

#高亮查询

GET /mydb/\_search

{

"query": {

"match": {

"name": "李四233"

}

},

"highlight": {

"pre\_tags": "<p style='color:red'>",

"post\_tags": "</p>",

"fields": {

"name": {}

}

}

}

#分组

GET /mydb/\_search

{

"aggs": {

"group\_name": {

"terms": {

"field": "age"

}

}

}

}

#分组 平均值

GET /mydb/\_search

{

"aggs": {

"group\_name": {

"avg": {

"field": "age"

}

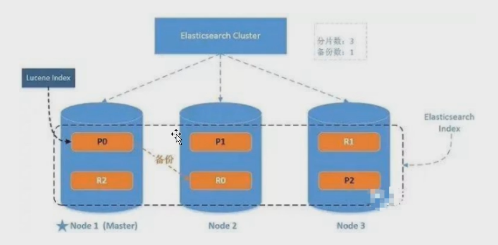
}

},

"size": 0

}

1. 选举master 3个分片 2个备份 9个服务器



心跳检测 + 选举

1节点之间心跳来发现彼此

2所有的节点(node:master:true)根据nodeId 字典排序，让第一个节点暂时成为master

3投票对某个节点达到半数规则，让节点成为master

1. es脑裂：比如网络问题 认为master挂了,实际上没有挂 重新选举master

最小主节点数（min\_master\_nodes） 超过所有（node:master:true）节点 一半以上来避免

心跳时间 延长

1. 索引分片复制：将倒排索引分开然后数据冗余复制 路由：索引从哪存 就从哪取

一致性hash hash(id) %分片数

1. es查询过程

客户端请求集群节点（协调节点）

协调节点查找数据的位置（主节点 副本集）

轮训算法请求某个节点拿到数据

将结果反馈给客户端

1. es写过程

客户端请求集群节点（协调节点）

协调节点转发请求给主节点

主节点发送数据给副本

副本保存后进行反馈给主节点 主节点反馈给客户端

1. es读写一致性？

consistency

one 主分片保存就返回

all 主分片和副本全部保存就返回

quorum 大部分保存就返回