目录

[Java基础知识 1](#_Toc36755486)

[JAVA 多线程 2](#_Toc36755487)

[AQS 3](#_Toc36755488)

[JAVA 线程池 3](#_Toc36755489)

[JVM内存结构 4](#_Toc36755490)

[垃圾收集器与内存分配策略 5](#_Toc36755491)

[Synchronized实现原理 7](#_Toc36755492)

[Spring mvc流程 9](#_Toc36755493)

[设计模式 10](#_Toc36755494)

[Redis 11](#_Toc36755495)

[RabbitMq 14](#_Toc36755496)

[Mysql 16](#_Toc36755497)

[Spring cloud 26](#_Toc36755498)

[mybatis 28](#_Toc36755499)

[Spring 29](#_Toc36755500)

[网络通信 30](#_Toc36755501)

[分布式事务、 32](#_Toc36755502)

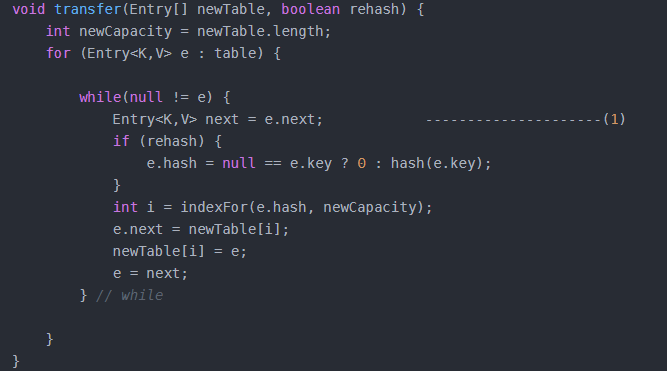
[分布式缓存 32](#_Toc36755503)

[分布式基石 32](#_Toc36755504)

[Mongodb 33](#_Toc36755505)

# Java基础知识

1. hashMap出现死锁的情况



当两个线程进行put操作的时候，假设T1和T2线程，put操作引起了扩容，假设原来的数组的链表表示为A->B，

1. 当线程T1执行完（1）处的时候挂起，T2引起了的扩容使得A和B在数组中的表示为B->A
2. T1继续执行（第一次循环），此时next=B,A.next=null,newTable[i]=A(T1线程创建的newTable与T2线程独立)
3. 第二次循环，next=A(见a),e.next=A,newTable[i]=B
4. 第三次循环，next=null,A.next=B,newTable[i]=A，此时循环结束
5. 结束后，新的数组中存在A->B,B->A，形成了回环，执行get操作时，死循环

2. 反射机制

1> 在运行时动态的获取类信息以及调用对象方法的功能

2> 反射机制的作用

1. 运行是判断任意一个对象所属的类
2. 运行时构造一个对象
3. 运行是调用任意一个对象的方法
4. 运行时判断一个类的属性和方法

3> 性能问题

反射是一系列的解释操作，通知虚拟机要做的是，比直接运行java代码慢

# JAVA 多线程

1. Java中Runable和Callable有什么不同

Runnable和Callable都代表那些要在不同的线程中执行的任务。Runnable从JDK1.0开始就有了，Callable是在 JDK1.5增加的。它们的主要区别是Callable的 call() 方法可以返回值和抛出异常，而Runnable的run()方法没有这些功能。Callable可以返回装载有计算结果的Future对象

1. 多线程的3大特性

原子性，可见性，有序性（happen-before原则）

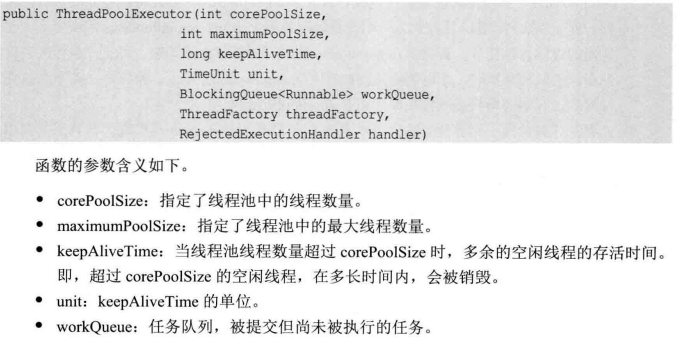
1. happen-before原则（禁止指令重排）
2. volatile规则：当操作该volatile变量时，所有前序对该变量的操作都已完成（如不存在已变更，但未写回主存的情况），所有后续对该变量的操作，都未开始
3. 程序的顺序原则，一个线程内保证程序语义的串行性
4. 传递性，A先于B，B先于C，则A 先于C
5. 可重入锁ReentrantLock
6. 可中断响应
7. 可以锁申请等待超时
8. 可设置为公平锁
9. Condition条件
   1. 类似于syn,await和singal方法用于等待和唤醒
10. 循环栅栏
11. 倒计时器
12. 信号量Semaphore(允许多个线程访问一个资源)，若信号许可被用完则需要等待其他线程释放许可

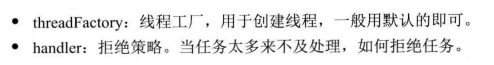
# AQS

1. AQS：java 并发包核心
2. AQS：抽象同步队列器
3. 并发包下的可重入锁，循环栅栏，计数器都是基于它实现的
4. 维护了一个核心int型变量state和一个记录当前加锁的是哪个线程
5. 维护了一个FIFO队列，没有获取锁的线程在此队列中进行等待，等待唤醒

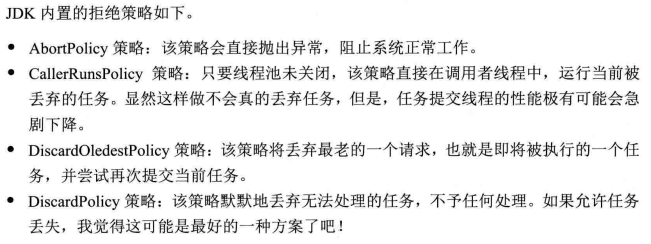
# JAVA 线程池

1. 线程池核心实现





1. workQueue
   1. 直接提交的队列，提交的任务不会被真实的保存，直接交给线程执行，一直到maximumPoolSize达到最大值后执行拒绝策略
   2. 有界的任务队列，先交给corePoolSize线程执行，然后加入队列，若队列已满，则交给maximumPoolSize执行，达到最大值后执行拒绝策略
   3. 无界的任务队列，若corePoolSize已满，则加入队列，直达系统资源耗尽
   4. 优先任务队列，一个特殊的无界队列，有优先级
2. 拒绝策略



1. 核心线程数的计算

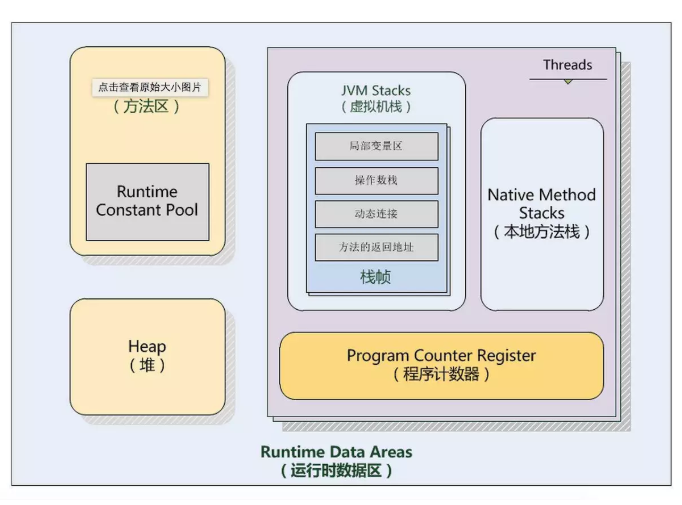
线程数 = CPU可用核心 / （1-阻塞系数）

存在大量I/O（网络，数据库交互，文件）：一般设置阻塞系数为0.5，即run的时间与等待时间相同，即2倍CPU可用核心

大量的复杂计算：就设置没有等待即阻塞系数为0，则等于CPU可用核心

# JVM内存结构

1. Java内存结构



1. 方法区

各个线程共享，存储虚拟机加载的类信息，常量，静态变量

1. 程序计数器

当前线程所执行的行号指示器，方便线程切换回来时知道执行到了哪里

1. 虚拟机栈

描述java方法执行的的内存模型。每一个方法从调用到执行完成，就对应一个栈帧在虚拟机栈中入栈到出栈的过程。有以下几部分组成

c.1 局部变量表，存放编译期可知的各种基本类型，对象引用类型

c.2 操作数栈

c.3 动态链接

c.4 方法出口信息

d. 本地方法栈

为虚拟机使用的Native方法服务

1. Java堆

存放对象的地方

1. 常量池

方法区的一部分，用于存放编译期可知的额各种字面量和符号引用

1. 对象的访问定位
2. 使用句柄访问。Java堆中分出一块内存作为句柄池，存放对象的句柄地址，这样在对象在移动地址时只会改变句柄中的实例数据指针
3. 直接指针访问。速度快，但是会经常改变栈上reference数据
4. 堆溢出

对象的内存大于堆内存，抛出outOfMemoryError，分为两种情况

1. 内存泄漏
2. 内存溢出

# 垃圾收集器与内存分配策略

1. 哪些对象可以回收
   1. 引用计数法

给对象添加一个引用计数器，每当一个地方引用它，给他加一，引用失效减一。任何时刻计数器的值为0则可以回收，但是无法解决相互循环引用的问题

* 1. 可达性分析算法

1. 从GC root对象为出发点，从这些节点向下搜索，搜索走过的路径成为引用链
2. 当一个对象没有与这个引用链相连，则这个对象不可用
3. 判定一个对象是否存活经历两次标记过程
4. objA与GC root没有引用链，进行第一次标记
5. objA重写了finalize方法，并且没有被执行过，那么对象会被插到F-Queue队列。由虚拟机创建的，低优先级的线程去执行此方法，在此方法中，若对象与GC root链上任何一个对象向链，则移出即将被回收的的集合
   1. 可作为GC root的对象
6. 本地变量表中引用的对象
7. 方法区中类静态属性引用的对象
8. 方法区常量引用的对象
9. Native方法引用的对象
10. 新生代

由一块Eden和两块Survivor（S0，S1）区域构成，比例为8：1

1. Eden：新生对象在Eden上分配，当空间不足，发生一次Minor GC
2. Survivor：发生Minor GC时，将存活的对象移到S0，清空Eden区域。再次发生时将Eden和S0存活的对象移到S1
3. 当对象GC年龄累加到超过15时，进入老年代
4. 老年代

当老年代空间不足，发生FULL GC/Minor GC

1. 收集算法
2. 标记清除算法

标记要清楚的对象，清楚标记的对象。效率问题，标记和清楚的的效率低。空间问题，造成大量的内存碎片

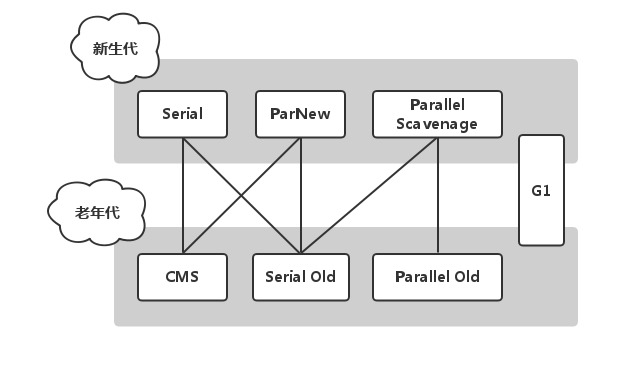
1. 复制算法（用于新生代）

将内存划分为两块相等的区域，每次只使用其中一块，当一块的对象收集完了后将这块的存活的对象复制到另一块，并清空这块的内存，提高了标记的效率同属冰面了内存碎片的问题

1. 标记整理算法（用与老年代）

将要清楚的对象标记出来，清理掉，同时将存活的对象移动到另一端，效率低，避免内存碎片

1. 对象标记过程
2. 使用一组OopMap的数据结构进行实现，在类装载完成时，HotSpot把对象内什么偏移量上是什么类型的数据计算出来存储到OopMap。GC发生时，扫描OopMap的数据就可以快速标识出存活的对象
3. 当发生GC时，活跃的线程设置中断标志，跑到安全区时查看中断标志是否为TRUE，若是中断挂起
4. 对于sleep和Blocked的线程，用安全区域解决（在此区域中对象的引用关系不会发生变化）
5. 当线程运行到Safe Region的代码时，首先标识已经进入了Safe Region，如果这段时间内发生GC，JVM会忽略标识为Safe Region状态的线程；
6. 当线程即将离开Safe Region时，会检查JVM是否已经完成GC，如果完成了，则继续运行，否则线程必须等待直到收到可以安全离开Safe Region的信号为止
7. 垃圾收集器



1. Serial收集器

暂停所有工作线程，采用复制算法收集新生代

1. Serial old收集器

采用标记整理算法收集老年代

# Synchronized实现原理

1. 对象的内存布局
   1. 对象头

1.1.1由mark\_word和class metadata address，monitor对象构成

mark\_word存储对象的hashcode，锁信息，分代年龄，GC标志

class metadata address存储对象的类信息

1.2 实例数据

1.3 填充数据

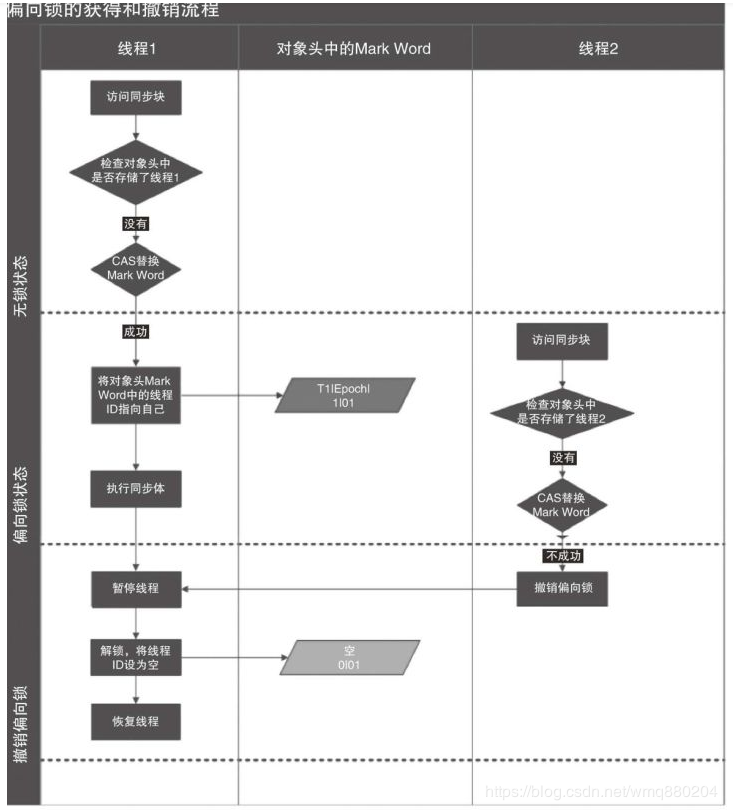
1. monitor对象

当前线程在遇到syn时试图获取monitor的持有权，获取时将monitor的进入计数器设置为1，再次重入时将计数器加一，释放锁时减一

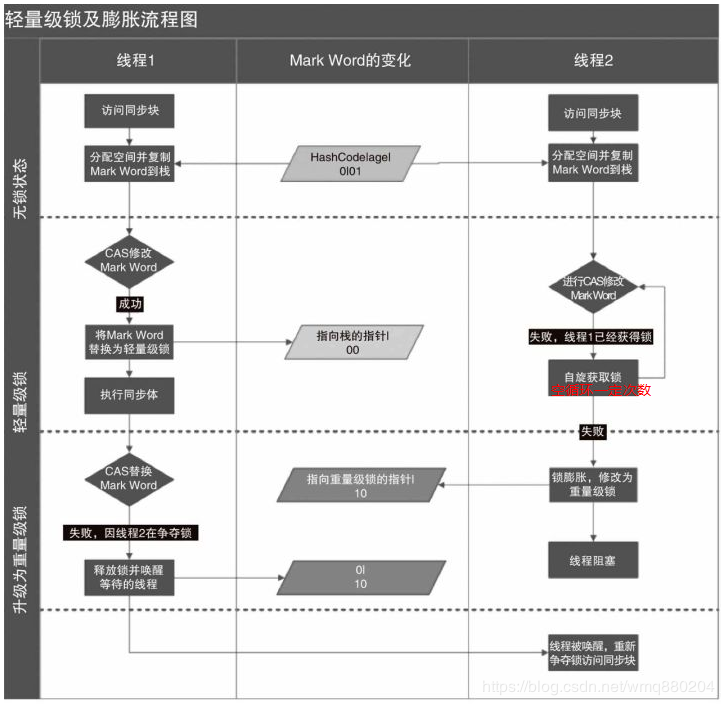
1. syn同步块底层原理
2. 同步语句块的实现使用的是monitorenter 和 monitorexit 指令
3. Monitorenter指向同步快开始的位置，monitorexit指向同步快结束的位置
4. 当执行Monitorenter的时候会尝试获取monitor对象(直接判断montor的计数器是否为0)，获取时将计数器加一
5. 假如同步块发生了异常，也会执行monitorexit指令，将monitor的计数器减一，此时编译器会自动产生一个异常处理器，用来执行monitorexit指令
6. Syn同步方法底层原理
7. 方法常量池中的方法表结构的ACC\_SYNCHRONIZED标志区分一个方法是否时同步方法
8. 当调用方法时先检查是否有ACC\_SYNCHRONIZED标志，若有，尝试获取monitor对象，回去了最后释放
9. 同样方法中发生了异常也会自动释放monitor对象
10. 由于挂起线程和恢复线程都需要转到内核态去做，所以jvm做了下面的锁优化
11. 偏向锁

当一个线程去尝试获取锁时，会有两种情况

1. 此时锁没有被占用（没有竞争），获得偏向锁，将对象mark\_word的标志位置为线程id，同步结束后将线程id置为null。此时若在同步过程中，线程不断重入时发现线程id时自己的id，则无需任何操作
2. 此时偏向锁被占用，并且不是自己的id，升级为轻量级锁



1. 轻量级锁
2. 将mark\_word复制进自己线程的私有栈（方便释放锁的时候将数据还原回去）
3. CAS将mark\_word的二进制位改为自己的私有栈地址
4. 成功则获得锁
5. 不成功则自旋一定次数后升级为重量级锁（自适应自旋锁就是线程空循环等待的自旋次数并非是固定的，而是会动态着根据实际情况来改变自旋等待的次数。简单来说就是线程如果自旋成功了，则下次自旋的次数会更多，如果自旋失败了，则自旋的次数就会减少）



# Spring mvc流程

原始得请求方式：

1. 继承httpServlet，实现doGet和doPost方法
2. 在web.xml中配置这个servlet



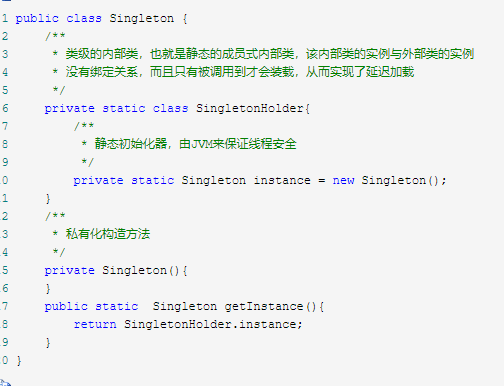
在使用spring mvc得时候就是将上述servler配置成DispatcherServlet，并且拦截所有得请求

Spring mvc流程

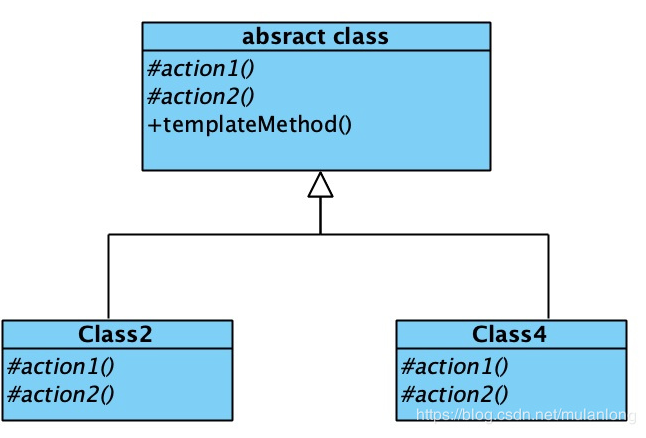
1. 发起请求到DispatcherServlet
2. DispatcherServlet根据url到handerMappering去获取相应得hander
3. 根据hander获取HanderAdapter(适配器模式)
4. HanderAdapter调用hander方法，首先执行拦截器方法，在执行controller方法获取对于得ModelAndView
5. 跳转到指定视图页面

# 设计模式

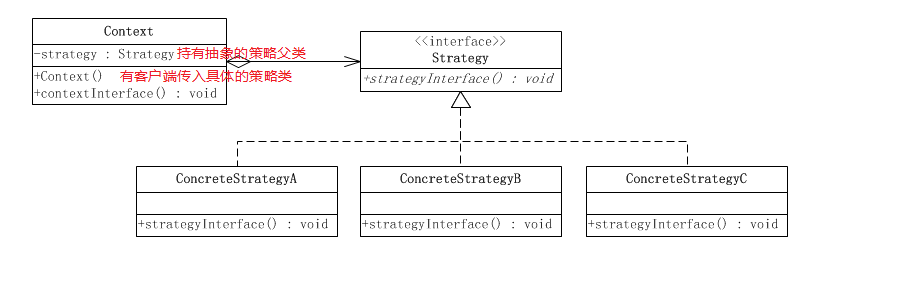
1. 单例模式



1. 模板方法模式
2. 不变的行为定义在父类中，不同的实现步骤是现在子类中

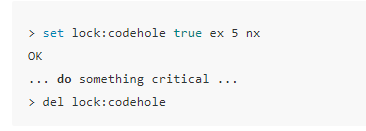


1. 策略模式



# Redis

1. 分布式锁



设置锁并且加上过期时间，但是一旦超时业务没有处理完也会出现并发问题

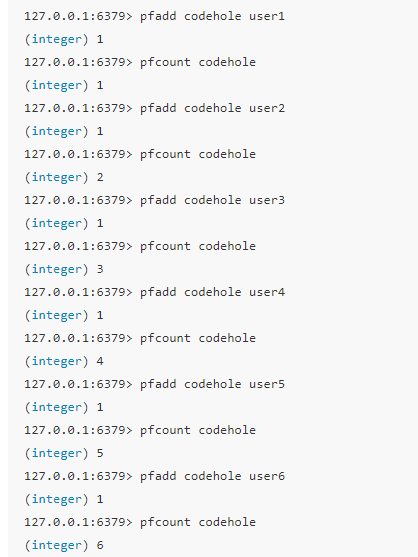
在哨兵模式集群下不安全，主节点挂掉时，从节点会取而代之，客户端上却并没有明显感知。原先第一个客户端在主节点中申请成功了一把锁，但是这把锁还没有来得及同步到从节点，主节点突然挂掉了。然后从节点变成了主节点，这个新的节点内部没有这个锁，所以当另一个客户端过来请求加锁时，立即就批准了。这样就会导致系统中同样一把锁被两个客户端同时持有，不安全性由此产生

1. 位图

场景1:记录用户365天的登录情况

位图实际就是byte数组，用一位记录用户在某天是否登录，那就是365位，总共46个字节

1. HyperLogLog数据结构进行估数



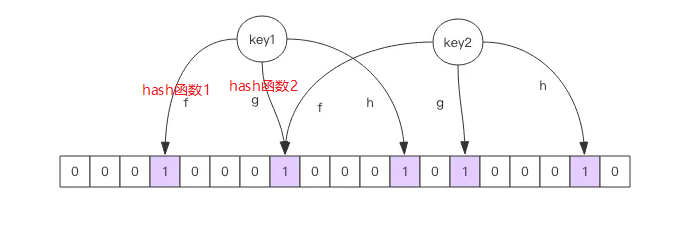
1. 布隆过滤器

当布隆过滤器说某个值存在时不一定存在

当布隆过滤器说某个值不存在时一定不存在

原理：

一个大型的位数组和几个hash函数组成，对key进行hash然后对位数组长度进行取模运算得到一个位置，每个 hash 函数都会算得一个不同的位置



1. Scan命令

scan 参数提供了三个参数，第一个是 cursor 整数值，第二个是 key 的正则模式，第三个是遍历的 limit hint, 第一次遍历时，cursor 值为 0，然后将返回结果中第一个整数值作为下一次遍历的 cursor。一直遍历到返回的 cursor 值为 0 时结束

例子：scan 0 match key99\* count 1000

1. 持久化
2. 快照，也就是全量备份

Redis使用操作系统的多进程来实现快照持久华。当redis进行快照持久化时，会fork出一个子进程，它此时与父进程共享数据，当父进程响应客户端请求的时候，父进程会复制出一个共享的页面，在复制的页面上做修改，子进程还是之前的页面进行修改

1. AOF日志，也就是增量备份

AOF 日志只记录对内存进行修改的指令记录，那么就可以通过这些指令恢复数据，当redis修改数据时，先保存指令才修改数据。这样有一个缺点就是随着redis数据集的增多，AOF日志越来越大

AOF重写不会对现有AOF日志进行改变，而是对目前数据库的状态来保存，redis会开辟一个子进程来进行重写，在重写时，服务端对所有的写命令写入AOF缓冲区，还会写入AOF重写缓冲区，写入AOF缓冲区为了同步到原有的AOF文件，写入AOF重写缓冲区是为了子进程在完成AOF重写工作时，将AOF重写缓冲区的内容写到新的AOF文件中

1. Redis事务
2. multi/exec/discard，服务器收到exec指令才开始执行整个事务队列，这个时候将不会执行其他指令
3. redis的事务不是原子性的，也就是全部成功或失败
4. watch指令，客户端在发起一个事务时，可以先监测一个key是否在事务执行时是否有修改过，若有则不执行
5. 主从复制
6. 增量同步

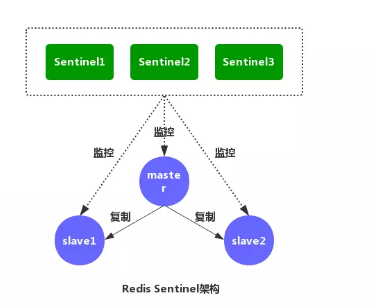
主节点会将修改命令记录在本地的buffer中，然后异步将buffer中指令同步到从节点中

由于buffer有限，redis的复制内存buffer是一个环形的定长数组，如果数组满了，就会从头开始覆盖之前的内容，如果短时间没有完成同步，需要用快照同步

1. 快照同步
2. 主节点将当前内存数据全部快照到磁盘上
3. 从节点清空内存数据，接收到快照文件进行全量加载
4. 继续进行增量同步
5. 增加从节点

从节点刚加入集群时，进行一次快照同步

1. 哨兵模式



1. 概念：Sentinel 是一个管理多个 Redis 实例的工具，它可以实现对 Redis 的 监控、通知、自动故障转移
2. Redis Sentinel的主要功能

监控：监控主节点和从节点是否正常

通知：当某个redis节点出现问题时像管理员或其他节点发送通知

自动故障转移：当主节点不能工作时，选取一个从节点作为主节点

配置提供者：客户端初始化链接时Sentinel节点集合，获取主节点信息

1. 主观下线和客观下线

默认情况下：每个Sentinel节点会以每秒一次的频率向redis节点和其他Sentinel节点发送ping命令，并通过节点的回复来判断节点是否在线

主观下线：适用于所有主节点和从节点，若在down-after-milliseconds 毫秒内，Sentinel 没有收到 目标节点 的有效回复，则会判定 该节点 为 主观下线

客观下线：只适用于主节点，如果 主节点 出现故障，Sentinel 节点会通过 sentinel is-master-down-by-addr 命令，向其它 Sentinel 节点询问对该节点的 状态判断。如果超过 <quorum> 个数的节点判定 主节点 不可达，则该 Sentinel 节点会判断 主节点 为 客观下线

1. 集群模式

去中心化，集群中每个节点都是master，并且数据保存在不同的节点上

1. 过期策略
2. 定时删除和惰性删除

过期时间的key单独放在一个字典中，定时来删除过期的key

客户端在访问这个key的时候，若过期则删除

1. 定时扫描

默认每10s进行过期扫描，从过期字典中随机20个key，删除20个中过期的key，若过期比例超过1/4，进行扫描

1. 从库的删除依靠主从同步去完成（AOF 指令）
2. Redis内存满了后LRU算法

LRU算法，淘汰最少使用的key

Redis的LRU算法

1. 给每个key加一个小字段，也就是最后访问的时间戳
2. 当redis执行写操作时，发现内存溢出，随机采样出5个key，淘汰掉最旧的5个key，直到内存没有溢出
3. maxmemory-policy 的配置可以配置时从所有key的字典中随机还是从带有过期时间key的字典中
4. 渐进式hash

redis的字典有两个hashTable，扩容步骤：

a.为ht[1]分配空间

b.在字典中维持一个索引计数器rehashIndex,并将它设置为0

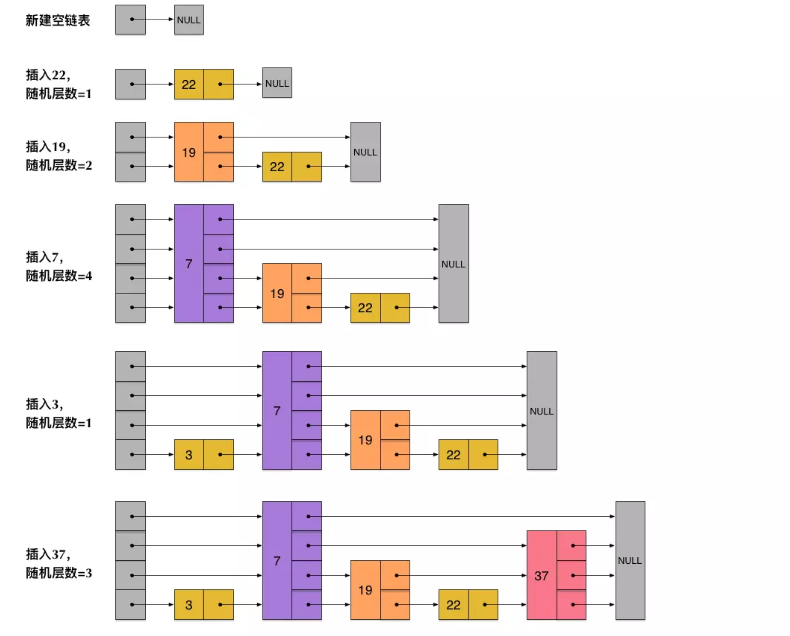
c.在reahsh期间对redis的操作会将ht[0]在rehashIndex上的数据复制到ht[1]上

完成后将rehashIndex加一

1. 当所有的数据都到ht[1]上时将rehashIndex置为-1

此外，还有定时任务将ht[0]上的数据复制到ht[1]上

1. zset实现跳表



每当有数据插入时都会随机一个层数，然后寻找插入的位置（根据score值寻找插入的位置），修改指针

# RabbitMq

1. 基本概念
2. 交换器

用来接收生产者发送的消息，并将这些消息路由给服务器中的队列

1. 绑定

用于消息队列与交换器的绑定（就是一个路由规则）

1. 消息队列

用来保存消息直到发送给消费者，消息的容器。一个消息可以投递到多个队列中，需要绑定 到交换器

1. 信道

多路复用链接中一条独立的双向数据流通道。建立在真实的TCP连接内的虚拟连接

1. 虚拟主机

表示一批交换器，消息队列和相关对象。一台服务器可以创建多个虚拟主机

1. 交换器类型
2. Direct交换器（默认交换器）

路由交换器匹配就投递到相应的队列，channel.basicPublish("", QueueName, null, message)推送direct交换器消息到对于的队列，空字符为默认的direct交换器，用队列名称当做路由键

1. Fanout交换器

它会把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中

1. Topic交换器

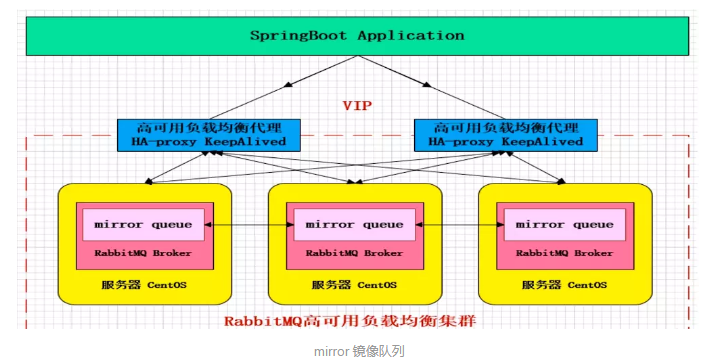
进行一些匹配，\*代表匹配一个，#代表匹配所有

1. Headers交换器

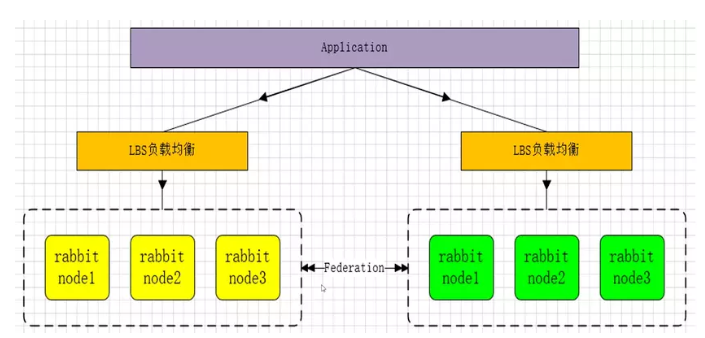
发送的消息内容中的headers属性进行匹配

1. 集群模式
2. 主从
3. 镜像模式

保证100%数据不丢失



1. 多活模式



主要介绍镜像模式：

1. 磁盘节点与内存结点

磁盘节点：保存状态到磁盘上，作为备份

内存节点：开放给客户端，最为客户端访问节点，速度更快

1. 所有节点都是对等的，即虚拟机，交换器队列以及绑定都会共享。
2. 消息的持久化取决于交换器，队列的持久化。
3. 一般配置HA proxy作为TCP反向代理

生产者消息确认模式

1. AMQP协议的事务机制

生产者需要同步等待Broker的执行结果，性能极大的降低

1. 信道设置为确认模式
   1. 发送方所有消息分配唯一ID
   2. 一旦消息投递到所匹配的队列中，信道向发送方发送确认消息（包含这个位于ID）
   3. 最大优势为异步，发送方可以继续发送，当收到确认消息调用回调处理
   4. 普通模式：每发送一个消息，就确认消息
   5. 批量模式：每发送完一批消息，再确认，有异常情况，全部重发
   6. 异步模式：为每个信道维护一个尚未确认的消息集合，每次发送加一，执行回调减一
2. 消费者应答
   1. 自动回执，Broker发送成功立即把消息删除，不用等待消费者的回执确认消息
   2. 手动回执，Broker发送成功不会把消息删除，直到收到消息者的确认消息。如果消费者没有发送ACK指令，Broker会把消息发给其他消费者（若没有其他消费者，则缓存消息，直到有新的消息消费者注册）
   3. 消费者发送拒绝消息的指令，可要求broker将消息丢弃或者重新放入队列
3. 死信队列

消息变成死信有几种情况：

消息被拒绝并且不重新入队

消息TTL过期

队列达到最大长度

延时队列

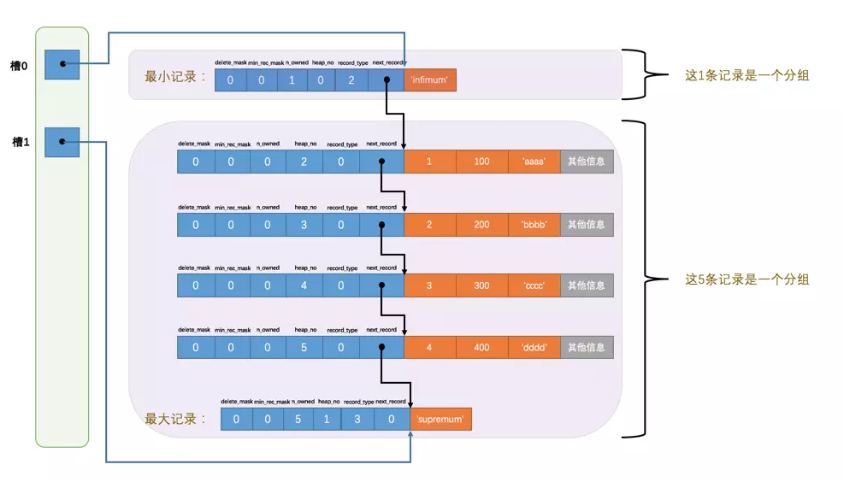
两个队列，一个正常的队列A，一个自定义的死信队列B。A上的消息过期时设置自动路由到B上，由B对应的消费者去消费消息

# Mysql

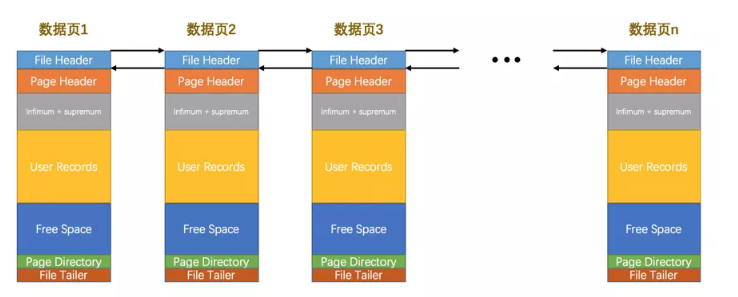
1. 数据在数据页中的存储



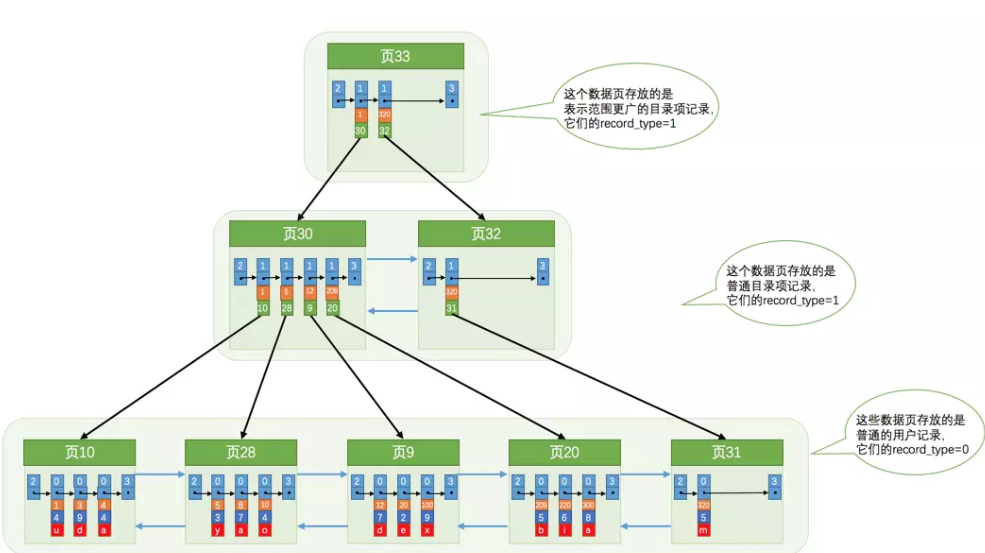
1. 删除记录会将记录从链表中移除，数据页中存在多条被删除掉的记录时，这些记录的next\_record属性将会把这些被删除掉的记录组成一个垃圾链表，以备之后重用这部分存储空间
2. 页目录



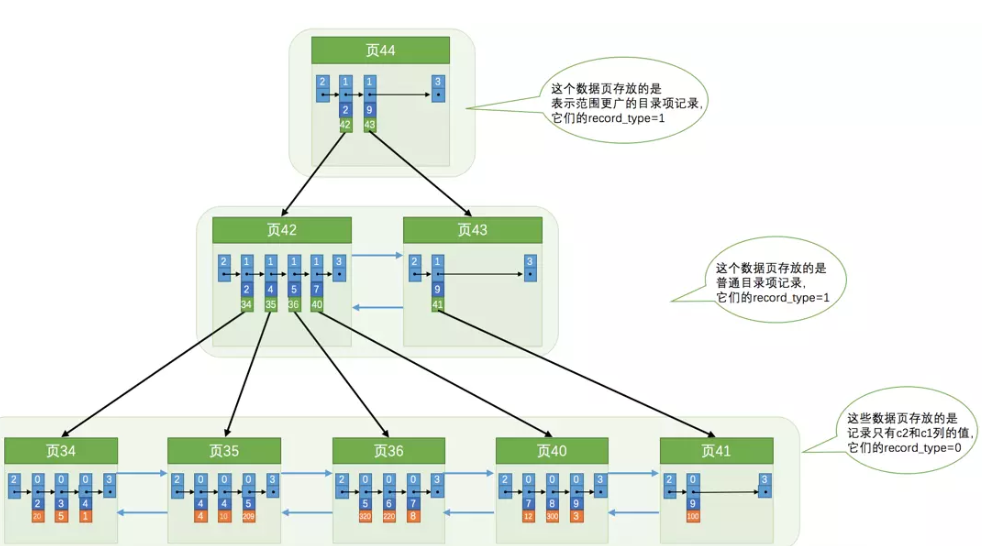
1. 将数据页里的行记录分组（包括最大记录和最小记录）
2. 将每组内最后的一条记录的地址偏移量提出出来单独存放在页目录中
3. 所以在一个数据页中查找指定主键值的记录的过程分为两步：
4. 通过二分法确定该记录所在的槽，并找到该槽所在分组中主键值最小的那条记录。
5. 通过记录的next\_record属性遍历该槽所在的组中的各个记录。
6. 数据页相连



1. File Header里FIL\_PAGE\_PREV和FIL\_PAGE\_NEXT就分别代表本页的上一个和下一个页的页号,其实就是双链表
2. 聚簇索引



1. 从最底层开始，分别提取该页的最小主键id，组成倒数第二层的目录数据（主键id+页号），然后从第二层开始循环操作，一直到最顶层
2. 非聚簇索引



1. 最底层存储的数据是主键id
2. 与聚簇索引不一样的是在非叶子节点也存储着主键id，这样索引列+主键id+页号保证唯一性
3. 对于联合索引，例如C1和C2列，先对C1列排序后在，再在C1列排序的基础上再对C2列排序，这就是为什么单独将C2列作为查询条件，不会用到索引
4. MyISAM存储引擎的索引方案



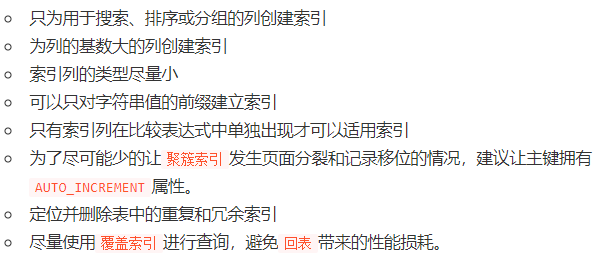
1. 将表中的记录按照插入顺序单独存储在一个叫文件中，不划分页，直接往里面塞
2. 并不会按照主键顺序排序
3. 创建一个主键的索引，叶子节点存储的是行号，通过行号查找数据
4. 对于非主键索引，叶子节点存储的是索引列+行号
5. 相当于MyISAM全部是二级索引
6. B+树索引适用条件
7. 匹配列前缀，Like %sss%不适合的原因就是索引的结构导致的
8. 匹配范围值
9. 用于排序，但是对于联合索引order by后的列的顺序也必须是索引的顺序，而且不能一个升序，一个降序。Where后面不能使用非索引列
10. 用于分组，条件跟排序的条件一致
11. 索引的代价

索引的代价就是需要回表，访问索引的时候，例如a>1000,这时候就是用顺序I/O,因为在磁盘上是连续的，但是回表的时候就是随机I/O，因为主键并不连续

1. 覆盖索引

直接查询索引列字段，不用回表

1. 创建索引原则



1. 单表访问的方法
2. Const

使用唯一索引（不包含null）或者主键查询

1. Ref

使用普通二级索引等值查询

1. Ref\_or\_null

使用普通二级索引等值查询并且把null的也查出来

1. Range

利用索引进行范围查询

1. Index

直接遍历二级索引的查询

1. All

全表扫描

1. 表连接的原理

连接查询的结果集中包含一个表中的每一条记录与另一个表中的每一条记录相互匹配的组合，像这样的结果集就可以称之为笛卡尔积

1. 首先确定第一个需要查询的表，这个表称之为驱动表
2. 从驱动表产生的结果集中的每一条记录，分别需要到t2表中查找匹配的记录，所谓匹配的记录，指的是符合过滤条件的记录。因为是根据t1表中的记录去找t2表中的记录，所以t2表也可以被称之为被驱动表

WHERE子句中的过滤条件

WHERE子句中的过滤条件就是我们平时见的那种，不论是内连接还是外连接，凡是不符合WHERE子句中的过滤条件的记录都不会被加入最后的结果集

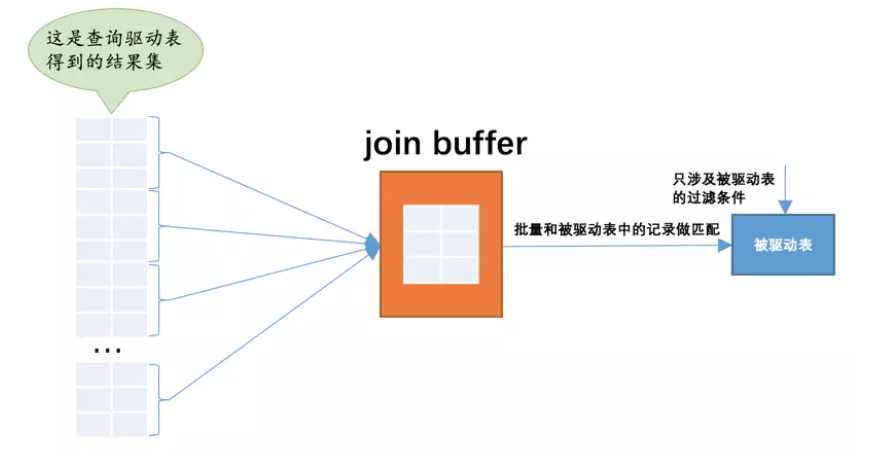
ON子句中的过滤条件

对于外连接的驱动表的记录来说，如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录，那么该记录仍然会被加入到结果集中，对应的被驱动表记录的各个字段使用NULL值填充。

需要注意的是，这个ON子句是专门为外连接驱动表中的记录在被驱动表找不到匹配记录时应不应该把该记录加入结果集这个场景下提出的，所以如果把ON子句放到内连接中，MySQL会把它和WHERE子句一样对待，也就是说：内连接中的WHERE子句和ON子句是等价的

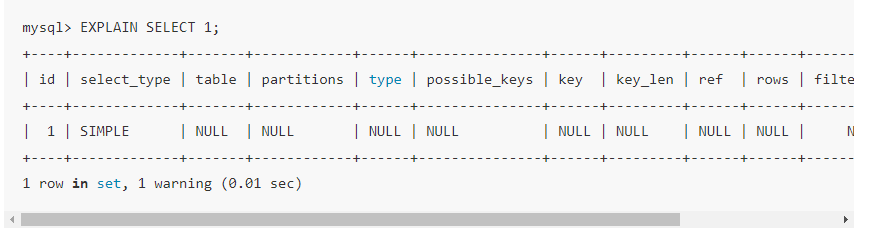
join buffer

join buffer就是执行连接查询前申请的一块固定大小的内存，先把若干条驱动表结果集中的记录装在这个join buffer中，然后开始扫描被驱动表，每一条被驱动表的记录一次性和join buffer中的多条驱动表记录做匹配，因为匹配的过程都是在内存中完成的，所以这样可以显著减少被驱动表的I/O代价



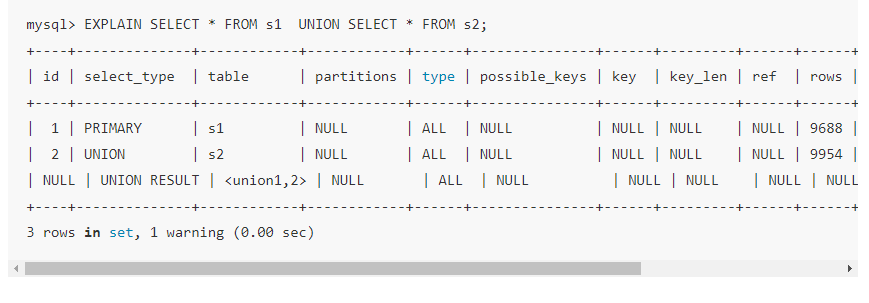
驱动表的记录并不是所有列都会被放到join buffer中，只有查询列表中的列和过滤条件中的列才会被放到join buffer中，最好不要把\*作为查询列表，只需要把我们关心的列放到查询列表就好了，这样还可以在join buffer中放置更多的记录

1. 执行计划



1. Id

在连接查询的执行计划中，每个表都会对应一条记录，这些记录的id列的值是相同的，出现在前边的表表示驱动表，出现在后边的表表示被驱动表,下面是创建了临时表



1. Type

MySQL对某个表的执行查询时的访问方法

1. Ref

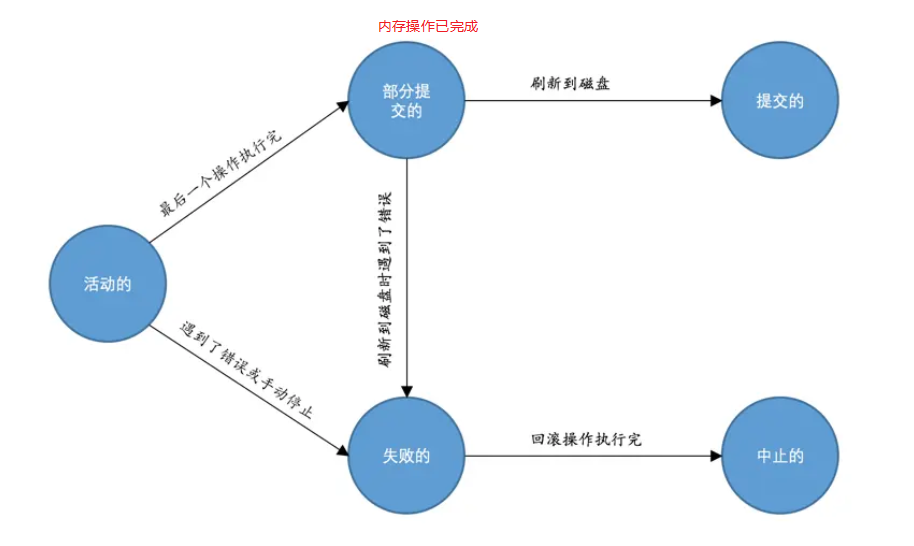
当使用索引列等值匹配的条件去执行查询时，也就是在访问方法是const、eq\_ref、ref、ref\_or\_null、unique\_subquery、index\_subquery其中之一时，ref列展示的就是与索引列作等值匹配的东东是个啥，比如只是一个常数或者是某个列

1. Rows

如果查询优化器决定使用全表扫描的方式对某个表执行查询时，执行计划的rows列就代表预计需要扫描的行数，如果使用索引来执行查询时，执行计划的rows列就代表预计扫描的索引记录行数

1. 事务的ACDI
2. 原子性（Atomicity）、隔离性（Isolation）、一致性（Consistency）和持久性（Durability）

事务的状态变化



1. 事务的自动提交

每一条语句都算是一个独立的事务，这种特性称之为事务的自动提交

1. Redo日志

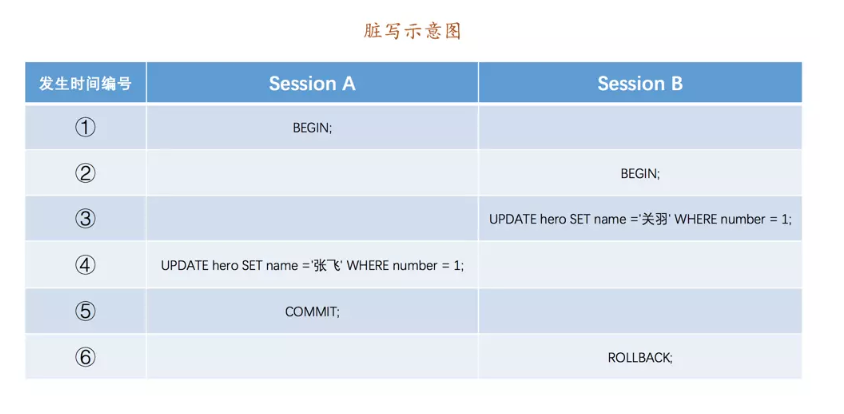
redo日志会把事务在执行过程中对数据库所做的所有修改都记录下来，在之后系统崩溃重启后可以把事务所做的任何修改都恢复出来

1. Undo日志

为了回滚而记录的这些东东称之为撤销日志

1. 事务的隔离级别

脏写：sessionA将无法看到已经修改过数据



脏读：事务A读到一个不存在的数据



不可重复读：在Session B中提交了几个隐式事务（注意是隐式事务，意味着语句结束事务就提交了），这些事务都修改了number列为1的记录的列name的值，每次事务提交之后，如果Session A中的事务都可以查看到最新的值，这种现象也被称之为不可重复读



幻读：个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，那就意味着发生了幻读



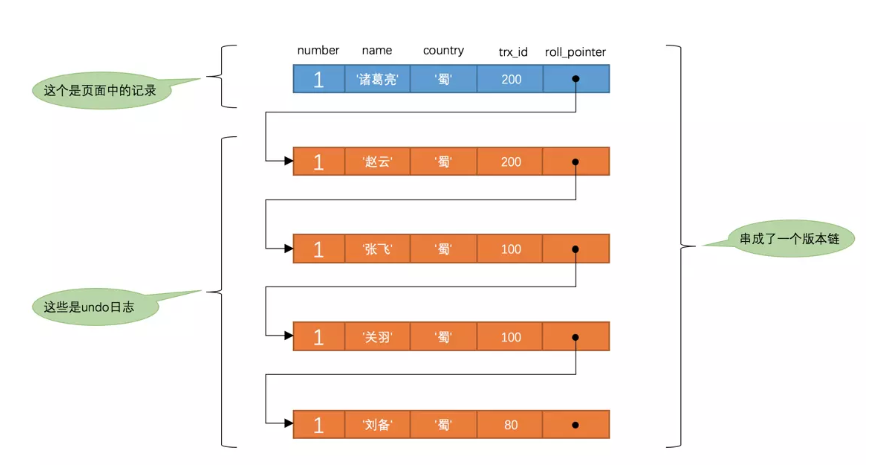
4种隔离级别：



1. 多版本并发控制（MVCC）

前提条件：

1. 如果只读事务默认事务id=0
2. 多版本并发控制只涉及读事务和写事务之间的并发，写事务之间相互阻塞（如果改动同一行记录），如果不阻塞，直接可能发生脏写
3. 事务的id是递增的
4. trx\_id：每一次事务对记录进行更改时，都会将本版本的事务id赋给trx\_id列
5. roll\_pointer：每一次事务对记录进行更改时，就上一个版本的信息写道undo日志中，roll\_pointer存储着指向上一个版本undo日志的指针，这相当于版本之间形成了链表



ReadView

1. 对于每一个select语句都会生产一个ReadView
2. m\_ids：生成readView时，当前活跃的读写事务id集合，[min\_m\_id ,max\_m\_id]
3. 如果被访问的行记录的版本trx\_id = 当前事务版本id，意味着是本次修改过的记录，可以被访问
4. 如果被访问的行记录的版本trx\_id < min\_m\_id，说明已经当前事务访问的版本已经提交，可以被访问到
5. 如果被访问的行记录的版本trx\_id > max\_m\_id，说明该版本的事务是在当前事务之后开启的，该版本不可以被访问
6. min\_m\_id < = trx\_id < = max\_m\_id，若trx\_id在这个列表中，说明事务还未提交，不可以访问，若不在，说明已经提交，可以被访问到
7. 如果该版本对当前事务不可见的话，则顺着版本链表找，直到找到一个可见的或者完全没有

read uncommitted隔离级别

1. 对于read uncommitted隔离级别事务来说，只需读取最新的版本就可以了（由于可以读到未提交的事务）

对于read committed隔离级别

1. 使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成一个独立的ReadView，也就是两次查询同样的数据，若另一个事务在两次查询的期间改变了这个数据并且已经提交，则第二次可以查到最新数据

对于REPEATABLE READ隔离级别

1. 只会在第一次执行查询语句时生成一个ReadView，之后的查询就不会重复生成了，也就是两次查询同样的数据，若另一个事务在两次查询的期间改变了这个数据并且已经提交，则第二次不可以查到最新数据。原因是第一次查询时修改数据的事务未开始或者未提交，只能查到上一个版本的数据
2. 解决了幻读的问题（插入的数据的事务id不在第一次查询的活跃事务id列表内并且大于最大的版本号）

所谓的MVCC就是通过生成一个ReadView，然后通过ReadView找到符合条件的记录版本（历史版本是由undo日志构建的），其实就像是在生成ReadView的那个时刻做了一次时间静止（就像用相机拍了一个快照），查询语句只能读到在生成ReadView之前已提交事务所做的更改，在生成ReadView之前未提交的事务或者之后才开启的事务所做的更改是看不到的。而写操作肯定针对的是最新版本的记录，读记录的历史版本和改动记录的最新版本本身并不冲突，也就是采用MVCC时，读-写操作并不冲突

# Spring cloud

1. 注册中心
   1. 采用对等复制的方式，数据在各个eureka server之间复制，不是采用主从复制
   2. 采用AP（可用性和分区容错性），保证最终一致性，而consul采用CP（一致性和分区容错性）
   3. Client下线没有通知enureka server，可以调整EvictionTask（定时任务剔除续约超时的服务）的调度频率，默认60s
   4. 应用实例下线告知enureka server，但是由于eureka server的rest api有response cache，则可以关闭readOnlyCacheMap
   5. 新服务上线，enureka client获取不到，提高client拉去server注册信息的频率，默认30s
   6. 自我保护模式：默认情况下，一段时间内没有收到服务的心跳，则会注销实例，（默认90s），由于网络抖动，短时间内丢失大量客户端的心跳，通过自我保护机制来保护实例

自我保护阈值：服务总数 \* （60s/客户端续约间隔）\* 0.85（默认），这个阈值与上一分钟收到的续约总数进行对比，若小于阈值，则进入自我保护模式，不在剔除服务

* 1. 开启HTTP Basic认证保护enureka server api
  2. 高可用原理

Region：默认情况下Redion之间数据不会复制，会选择处于用一个zone的Eureka Server列表

1. Feign
   1. 默认使用JDK原生的URLConnection发送HTTP请求，没有连接池，对每一个地址保持一个长连接，使用http client代替原始的，设置超时时间，连接池
   2. 默认集成了ribbon,Hystrix
2. Ribbon
   1. 负载均衡策略：

随机测率，轮询策略，重试策略，最低并发策略，可用并发策略，响应时间加权策略，区域权衡策略

* 1. 超时与重试

对调用进行超时配置，并且配置重试次数

* 1. 默认情况下，Ribbon客户端会从Eureka注册中心读取服务注册信息列表，来达到一种动态负载均衡的功能。

1. Hystrix
   1. 服务降级，一旦调用服务出现异常，自定义降级方法
   2. 主要分为5种异常

执行失败，抛出异常

超时

断路器打开

线程池拒绝

信号量拒绝

* 1. BAD\_REQUEST异常对象非法参数不会触发fallback且不会技术进入熔断
  2. Hystrix Request Collapser是针对多个请求调用单个后端依赖做的一种优化和节约网络开销的方法
  3. 熔断机制

Hystrix会检查一个一定时长（图中为10s）的一个时间窗（window），在这个时间窗内是否有足够多的请求，如果有足够多的请求，是否错误率已经达到阈值，如果达到则启动断路器熔断机制，这时再有请求过来就会直接到fallback路径。在断路器打开之后，会有一个sleep window（图中为5s），每经过一个sleep window，当有请求过来的时候，断路器会放掉一个请求给remote 服务，让它去试探下游服务是否已经恢复，如果成功，断路器会恢复到正常状态，让后续请求重新请求到remote 服务，否则，保持熔断状态

* 1. Hystrix线程传递和并发策略

Hystrix会把请求放入线程池中执行，可能产生线程的变化，从而导致线程1的上下文数据在线程2中不能正常拿到

* 1. 信号量隔离策略

在请求的时候获取一个信号量，继续请求，请求在一个线程内执行完毕

1. Zuul
   1. 默认使用eureka中集成的负载均衡功能，如果想要使用ribbon，禁止ribbon使用eureka
   2. 四种filter

Pre：鉴权，限流

Route：路由动作的执行者

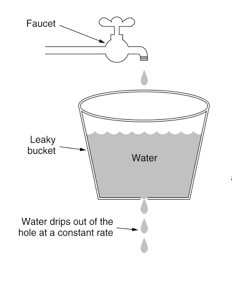
Post：服务返回结果或者异常信息发生执行，对返回信息做一些处理

Error：做全局异常处理

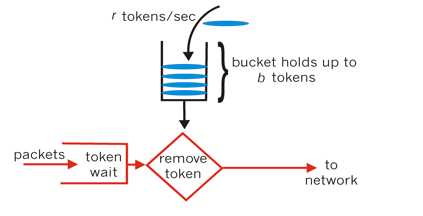
在filter之间，通过RequestContext进行通信

* 1. 限流算法

漏桶：漏桶算法能强行限制数据的传输速率



令牌：对于很多应用场景来说，除了要求能够限制数据的平均传输速率外，还要求允许某种程度的突发传输



可以集成spring-cloud-zuul-ratelimit进行限流

* 1. 多层负载

使用ngnix前加一个软负载，使用OpenResty(整合nginx,Lua,)动态网关

# mybatis

1. 加载流程
2. 插件开发

# Spring

1. IOC实现原理

<https://blog.csdn.net/nuomizhende45/article/details/81158383>

1. AOP实现原理
2. 事务机制

传播机制：

* 1. PROPAGATION\_REQUIRED

支持当前事务，如果当前没有事务，则新建事务

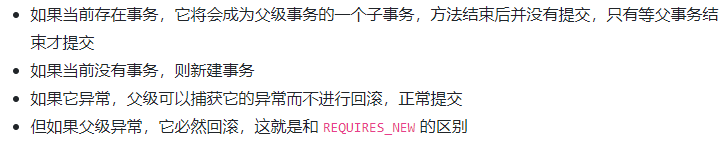
如果当前存在事务，则加入当前事务，合并成一个事务\

* 1. REQUIRES\_NEW

新建事务，如果当前存在事务，则把当前事务挂起

这个方法会独立提交事务，不受调用者的事务影响，父级异常，它也是正常提交

* 1. NESTED



# 网络通信

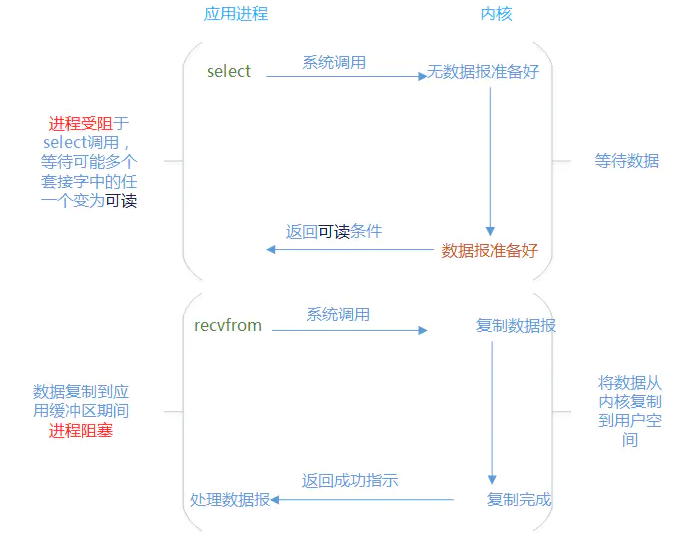
1. 网络I/O模型

阻塞I/O

非阻塞I/O

多路复用I/O

* 1. 不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。
  2. 串起来理解就是很多个网络I/O复用一个或少量的线程来处理这些连接
  3. 阻塞在select和recvfrom



* 1. 从应用进程的角度去理解始终是阻塞的，等待数据和将数据复制到用户进程这两个阶段都是阻塞的。这一点我们从应用程序是可以清楚的得知，比如我们调用一个以I/O复用为基础的NIO应用服务。调用端是一直阻塞等待返回结果的。
  2. 从内核的角度等待Selector上面的网络事件就绪，是阻塞的，如果没有任何一个网络事件就绪则一直等待直到有一个或者多个网络事件就绪。但是从内核的角度考虑，有一点是不阻塞的，就是复制数据，因为内核不用等待，当有就绪条件满足的时候，它直接复制，其余时间在处理别的就绪的条件。这也是大家一直说的非阻塞I/O。实际上是就是指的这个地方的非阻塞。

同步I/O

异步I/O

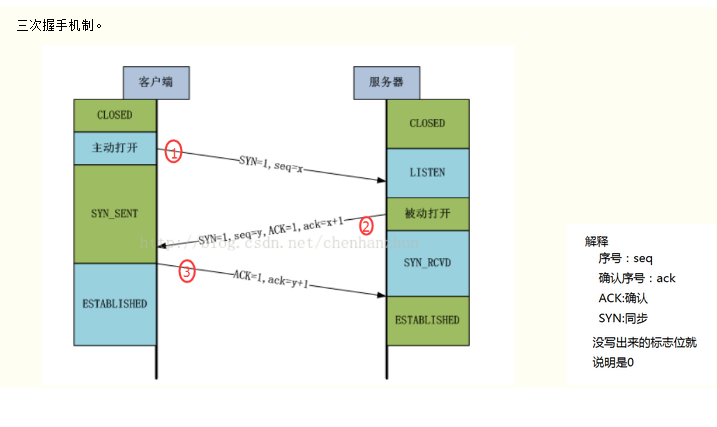
同步IO和异步IO的区别就在于：数据拷贝的时候进程是否阻塞

阻塞IO和非阻塞IO的区别就在于：应用程序的调用是否立即返回

1. http
2. TCP/IP

三次握手

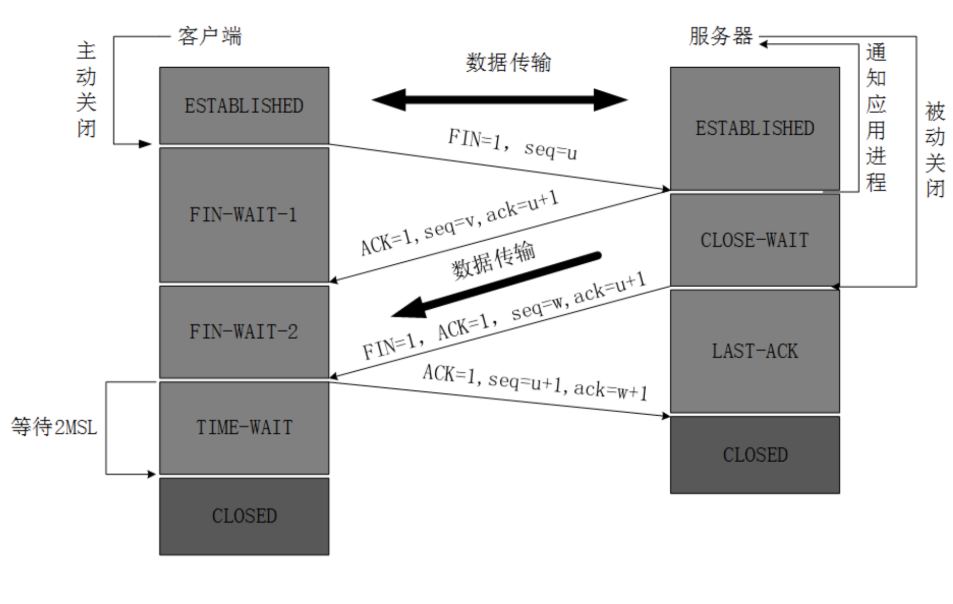
1. 客户端给服务端发SYN=1,随机序列号seq=x
2. 服务端给客户端发SYN=1,ACK=1,ack=x+1,自己随机一个序列号seq=y
3. 客户端给服务端发ACK=1,ack=y+1



如果只有两次握手，在第一次握手的时候client因为网络延迟，没有在预定的时间内到达服务端。Client端已经超时，但是服务端还是会给client端发送确认信息，并认为连接已经建立，但实际链接并未建立

四次挥手

1. 客户端发出FIN=1,seq=u,客户端进入FIN-WAIT-1状态
2. 服务端发出ACK=1,seq=v,ack=u+1,服务端进入CLOSE-WAIT状态
3. 客户端收到确认信息，进入FIN-WAIT-2状态。服务端将最后的数据发送完毕后，进发送FIN=1，ack=u+1，seq=w。服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认
4. 客户端发送ACK=1，ack=w+1，seq=u+1。客户端进入TIME-WAIT（时间等待）状态。此时链接还没有释放，必须经过2∗∗MSL（最长报文段寿命）的时间后才close



为什么要等待2∗∗MSL（最长报文段寿命）的时间后才close客户端

1. 保证TCP协议的全双工连接能够可靠关闭

如果Client直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致Server没有收到Client最后回复的ACK。那么Server就会在超时之后继续发送FIN，此时由于Client已经CLOSED了，就找不到与重发的FIN对应的连接，最后Server就会收到RST而不是ACK，Server就会以为是连接错误把问题报告给高层。这样的情况

虽然不会造成数据丢失，但是却导致TCP协议不符合可靠连接的要求。所以，Client不是直接进入CLOSED，而是要保持TIME\_WAIT，当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。

1. 保证这次连接的重复数据段从网络中消失

如果Client直接CLOSED，然后又再向Server发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题，但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中，这些延迟数据在建立新连接之后才到达Server，由于新连接和老连接的端口号是一样的，又因为TCP协议判断不同连接的依据是socket pair，于是，TCP协议就认为那个延迟的数据是属于新连接的，这样就和真正的新连接的数据包发生混淆了。所以TCP连接还要在TIME\_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失

# 分布式事务

1. 二阶段提交
2. 投票阶段

协调者询问参与者是否可以提交操作，并等待参与者响应。

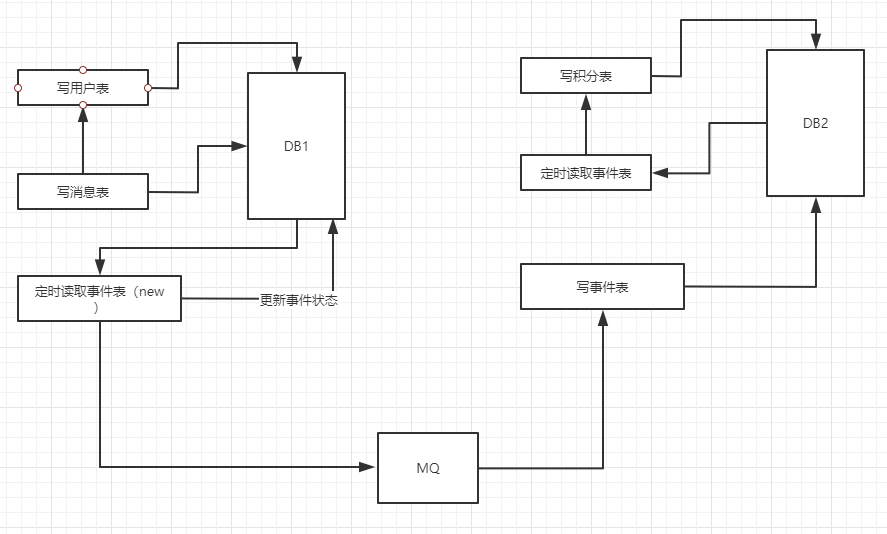
参与者将事务请求将undo，redo信息写入日志

1. 提交阶段

事务在每个参与者上都执行成功，事务协调者发送通知让参与者提交事务；否则，协调者发送通知让参与者回滚事务。

存在问题：

1. 同步阻塞
2. 单点故障
3. 数据不一致
4. TCC补偿事务
5. Try阶段对业务做检测和资源预留，比如转账的钱可以冻结起来
6. Confirm阶段，try阶段提交成功则提交。
7. Cancel阶段，业务执行错误，回滚事务
8. 本地消息表

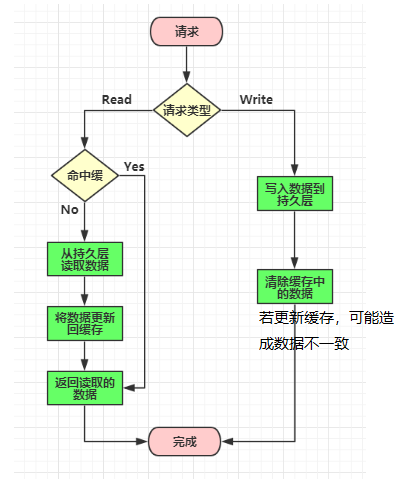


1. MQ事务消息

实现难度大，主流MQ不支持，RocketMQ事务消息部分代码也未开源。

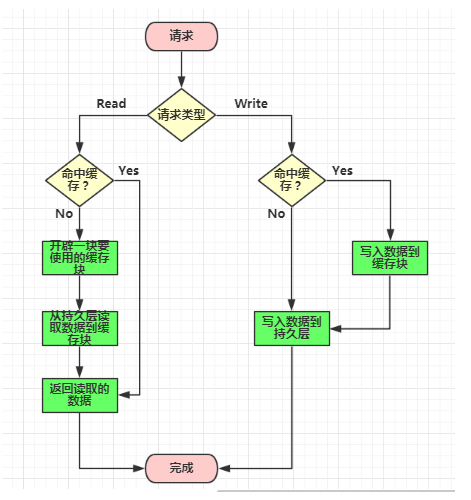
# 分布式缓存

1. 缓存的更新策略
2. Cache Aside模式



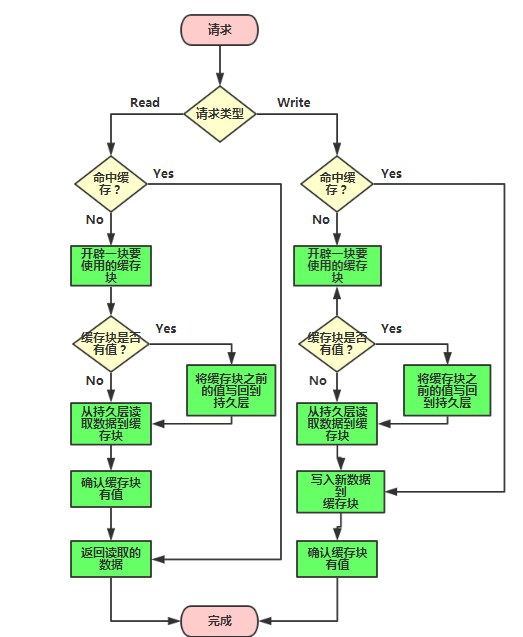
1. Read/Write Through模式

将缓存代替DB看作唯一的存储



1. Write Behind Caching(Write Back)模式

这种模式下所有的操作都走缓存，缓存里的数据再通过异步的方式同步到数据库里面。所以系统的写性能能够大大提升了



1. 缓存失效策略

一般设置缓存的失效时间。分为主动失效和被动失效

1. 缓存淘汰策略

FIFI先释放头部的对象

LRU根据访问的时间先后来淘汰，使用优先队列实现

LFU根据访问的频率来淘汰，使用有限队列实现

1. 缓存穿透

每次查询都穿过缓存，查询DB，可将不存在的数据放入缓存

1. 缓存击穿

在缓存失效的瞬间，大量请求到达DB

解决方案：在更新缓存时使用分布式锁锁住服务

1. 缓存雪崩

大量缓存在同一时间失效

解决方案：缓存加上随机事件

# 分布式基石

1. CAP
2. 一致性

在分布式系统中的所有数据备份，在同一时刻是否同样的值。等同于所有节点访问同一份最新的数据副本

1. 可用性

在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。对数据更新具备高可用性

1. 分区容忍性

分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

# Mongodb