# 基础

## 创建了一个HashMap拥有10000个元素，会触发扩容吗

答：在jdk1.8之后HashMap在长度大于8的时候，其数据结构会变成红黑树的结构

HashMap的构造方法里有两个成员变量 threshold，loadFactor，其中threshold就是扩容的阈值。**而threshold = initialCapacity \* loadFactor。**

**initialCapacity = (需要存储元素个数 / loadFactor(默认0.75)) + 1。**

**initialCapacity = 10000/0.75+1 =** 13334所以其不会发生扩容

# Jvm

## Jvm内存架构

**堆：**存放实例对象

**方法区：**各线程共享的内存区域，用于存储已被虚拟机加载的类的信息、常量、静态变量、即时编译后的代码缓存等数据

**运行时常量池**：方法区的一部分，用于存放编译器生成的各种字面量与符号引用，在类加载后存放到方法区的运行时常量池中

**元数据区：**元数据区取代了1.7版本及以前的永久代。数据区和永久代本质上都是方法区的实现。方法区存放虚拟机加载的类的信息，静态变量，常量等数据

**本地方法栈：**虚拟机使用到的native方法服务

**虚拟机栈：**线程私有，生命周期与线程一样，每个方法被执行时候，虚拟机都会同步创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

**程序计数器：**所执行的字节码的行号指示器

## 双亲委派机制

某个特定的类加载时，会请求自己的父类加载器加载，依次递归到最顶层的bootstrap，如果父类加载器无法加载该类时，自己再尝试加载

### 作用

**主要：**是为了安全，比如：如果程序员写了一个Java.lang.String类的话就容易产生问题

**次要：**不用重复加载

## 类加载机制

java 虚拟机将编译后的 class 文件加载到内存中，进行校验、转换、解析和初始化，到最终的使用

加载器：BootstrapClassLoader

ExtClassLoader

AppClassLoader

Jvm 在加载一个类的时候，会调用 AppclassLoader 的 LoadClass 方法，在这个方法中会先使用ExtClassLoader 方法来加载类，同样 ExtClassLoader 的 LoadClass 方法中会先使用BootstrapClassLoader 来加载类，如果 BootstrapClassLoader 加载到就直接成功，否则那么ExtClassLoader 自己尝试加载该类，如果仍然没有的话就会有 AppClassLoader 来加载

## jvm 调优参数

- Xms 堆初始值

- Xmx 堆最大可用值

- Xmn 新生代堆最大可用值

- XX : NewRatio ：设置年轻代和老年代的比值

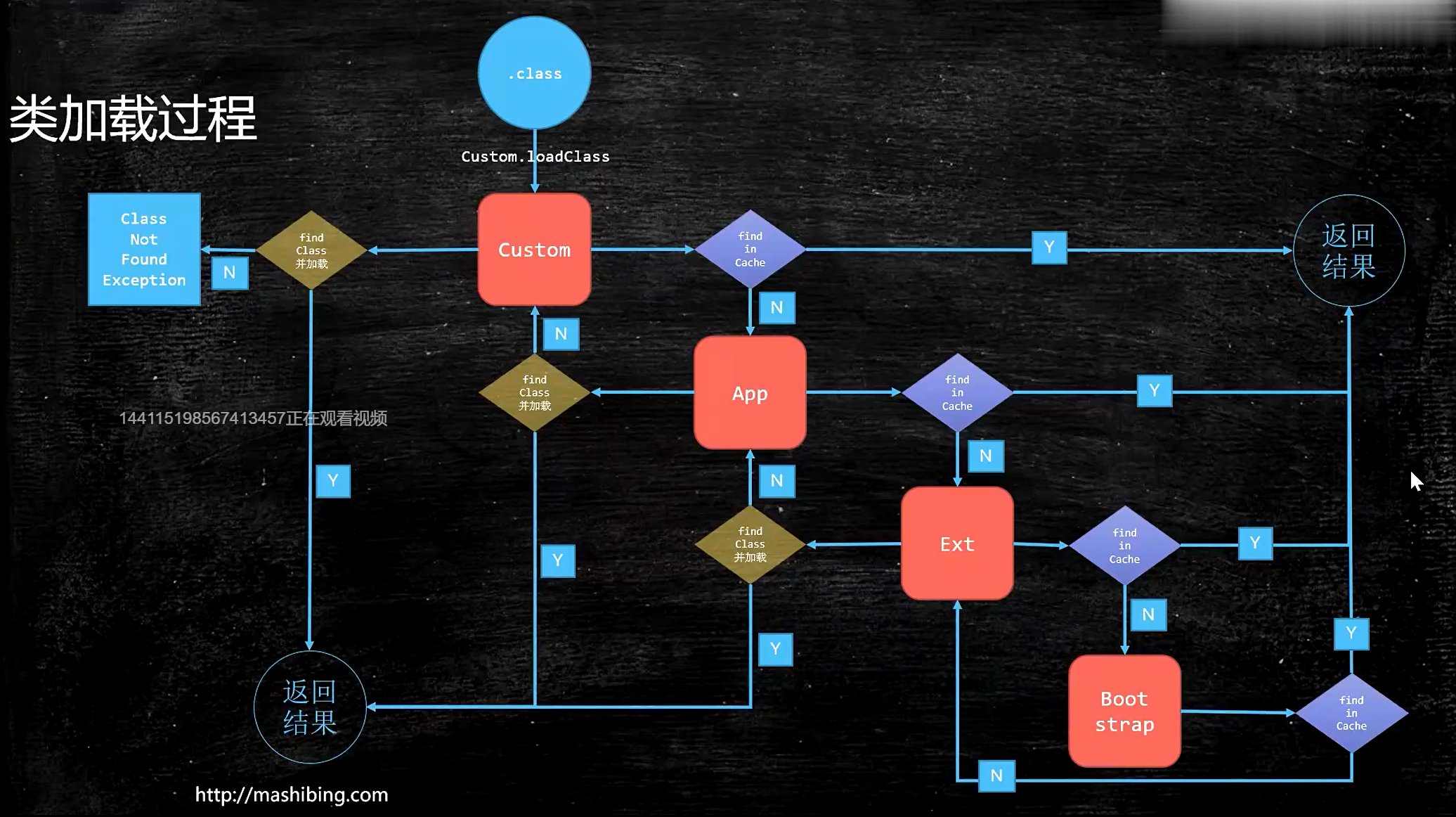
- XX . SurvivorRatio ：设置 Egen 区和两个 Survivor 的比值

- XX : MetaspaceSize ：初始元空间大小，JDK1.8起有效

- XX : MaxMetaspaceSize :元空间的最大值，JDK1.8起有效

查看初始参数

Java -XX:+PrintFlagsInitial



## 垃圾收集器

1、 Serial ( hotspot 虚拟机默认的新生代收集器）、 Serial Old 单线程优点：简单高效，额外内存消耗最小，没有线程交互的开销

2、 ParNew ( Serial 多线程版本）

3、 Paralel Scavenge （基于标记﹣复制算法的新生代收集器）

4、Paraller Old 基于标记﹣整理算法的老年代收集器）: Parallel Scavenge 的老年代版本

5、 CMS （基于标记﹣清除算法）:以获取最短停顿时间为目标的收集器，过程包括初始标记、并发标记仍会 STW :

1）初始标记

2）并发标记

3）重新标记

4）并发清除

6、G1：分区的垃圾收集器，是包括年轻代和年老代的 GC



## 如何判定对象是否是垃圾：

### 1：根可达算法

通过一系列称为“ GC Roots ”的根对象，作为起始节点开始根据引用关系向下搜索，搜索过程走过的路径称为引用链，如果对象到 GC Roots 之间没有任何引用链相连，此对象就是不可能在被使用的

#### 可以作为 GC Root 的对象

1)：方法区中的静态变量2)：字符串常量池中的引用

3): JNI 引用的对象

4)：被同步锁持有的对象

5)：方法堆栈中使用到的参数、局部变量、临时变量等

6)：虚拟机内部的引用，基本数据类型对象的 Class 对象以及系统

### 2：引用计数器

在对象中添加一个引用计数器，每当有一个地方引用它时，计数器加1，当引用失效时，计数器值减1。任何时候计数器值为 O 的对象就是不可被引用的对象

优缺点：效率高，但是难以解决对象之间的循坏依赖的问

判定一个类不再被使用，同时满足以下3个条件：

1）该类的所有实例都已经被回收

2）加载该类的类加载器已经被回收

3）无法在任何地方通过反射访问该类的方法

## 垃圾收集算法

标记﹣清除算法：

缺点：执行效率不稳定、内存碎片化

标记﹣清除算法：

缺点：执行效率不稳定、内存碎片化

复制算法：

缺点：内存浪费太多、当存活率高时，效率太低

优点：无内存碎片问题

标记﹣整理算法：

优点：不浪费空间且无碎片化问题

缺点：存活率高时，效率低

分代收集：

## 执行垃圾回收的场景：

1、YGC 的时机：

edn空间不足

### 2、FGC

1：调用 finalized 方法

2：显视调用 system.gc ()

3: old 区空间不足

JDK 1.8默认使用 Parallel Scavenge + Parallel Old (PS ＋ PO)

新生代使用复制算法回收垃圾，老年代则根据不同的垃圾收集器会有不同的垃圾回收算法。

Eden S0 S1的空间大小比例默认8：1：1

## 四种引用

**强：**new 对象，只要强引用关系存在，永远不会被垃圾收集器回收

**软：**用来描述一些还有用，但非必须的对象。在系统发生内存溢出前，会把这些对象进行第二次回收，如果内存还不够才会抛出内存溢出异常

**弱：**用来描述非必须对象，强度比软引用更弱。被引用的对象只能

生存到下次垃圾回收，无论内存是否足够，都会回收掉被引用的对象

**虚：**最弱的一种引用对象。唯一目的是为了能在这个对象被回收时收到一个系统通知

## 内存占用过高

Jmap -histo pid |head -20

查看占用内存最多的前20个类，然后排查

1：Jvm运行环境慢、卡顿

为什么会慢？

频繁的GC，STW长。响应时间慢

公司决定升级，结果反而比以前的效率更低了

内存越大，FGC时间越长，STW时间越长

2：系统CPU经常100%，如何调优？

1：先找出哪个进程CPU高 (top命令)

2：找出该进程中的哪个线程CPU高 （top -Hp pid 命令）

3：到处该线程的堆栈 （jstack）

4：查找哪个方法（栈帧）消耗时间 (jstack)

5：工作线程占比高 | 垃圾回收线程占比高

3：系统内存飙高，如何查找问题?

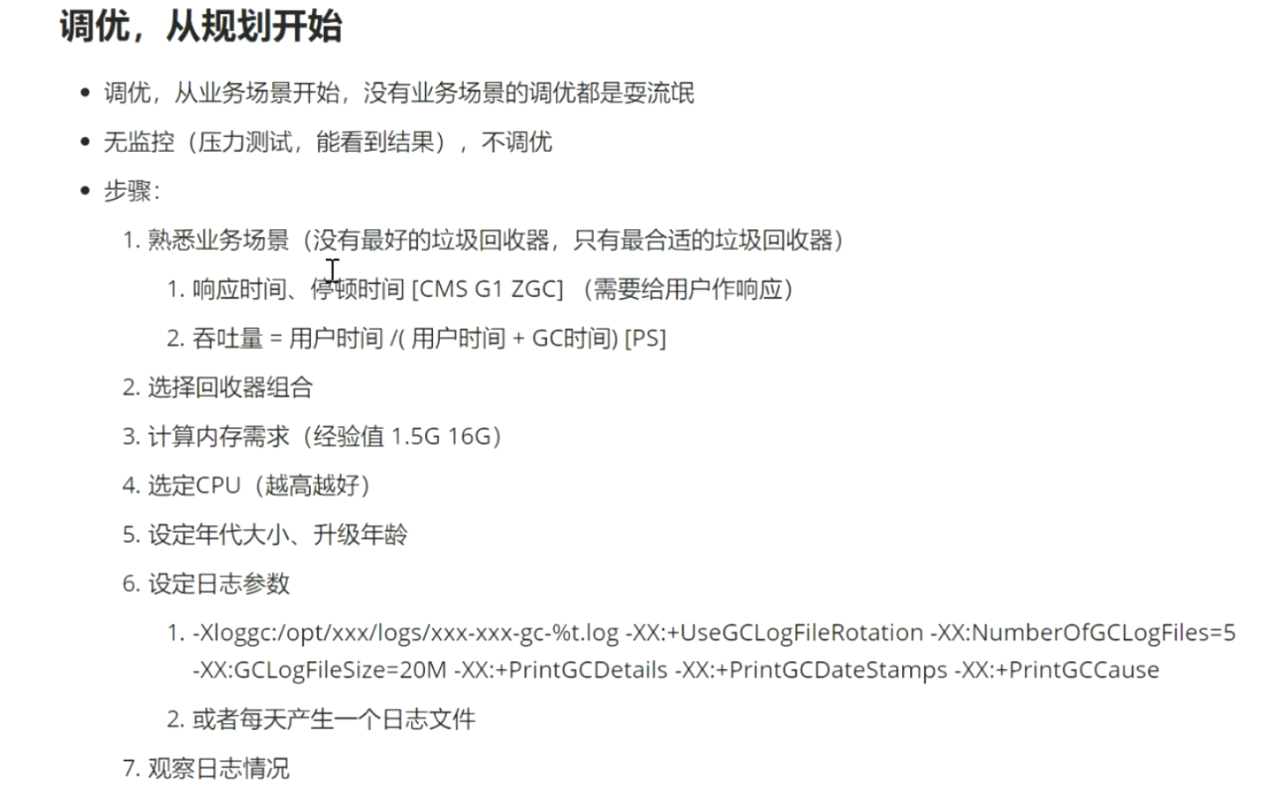
1：导出堆内存 （jmap命令）

2：分析 （jhat jvisualvm mat jprofiler …等工具）

4：如何监控JVM

1:jstat ,jvisualvm jprofiler arthas top

## 预调优



## 如何定位OOM问题

使用jmap -histo pid | head -20 查找有多少实例产生

线上系统内存特别大，jmap执行期间会对进程产生很大影响，甚至卡顿

1：设定参数 HeapDumpOfOutOfMemeryError ,OOM时会自动产生堆转储文件

2：很多服务器备份（高可用），停掉这台服务器对其他服务器不影响

3：在线定位

# 多线程（JUC）

## JMM(Java内存模型 )

描述的是一组规则或规范，通过这组规范定义了程序中的各个变量的访问方式

### 特性

可见性，原子性，有序性

### 关于同步的规定

1. 线程解锁前，必须把共享变量的值刷新回主内存
2. 线程加锁前，必须读取主内存的最新值到自己的工作内存
3. 加锁解锁必须是同一把锁

## Volatile

Java虚拟机提供的轻量级的同步机制

1：保证可见性

2：不保证原子性

3：防止指令重排序

## submit 和 execute 方法的区别

submit方法，会返回一个Future对象，而execute没有。接收参数不一样，execute(）只能接受Runnable类型参数

## 拒绝策略

AbortPolicy 直接抛异常

RejectedExecutionException （默认）

DiscardPolicy 丢弃任务，但是不抛出异常

DiscardOldestPolicy 丢弃老的任务，接受新任务

CallerRunsPolicy交给调用方执行

## synchronized和volititle的区别

volititle 只能作用于变量上，且不可以保证操作的原子性，可以防止指令重排序

synchronized可以作用于变量、方法和类上，可以保证操作的原子性。

## synchronized 和 ReentrantLock 的区别

1: synchronized是关键字属于jvm层面，ReentrantLock是具体的类，是api层面的锁

2：synchronized不需要用户去手动释放锁，当 synchronized代码执行完成之后系统会自动让出线程释放对锁的占用，ReentrantLock则需要用户去手动释放锁，若没有主动释放锁可能会导致死锁现象

3：synchronized不可中断，除非抛出异常或者正常运行完成

ReentrantLock可中断，1：设置超时方法tryLock(long timeout, timeUnit unit) 、2：LockInterruptibly放代码块中，调用interrupt方法()方法

4：加锁是否公平synchronized非公平锁

ReentrantLock两者都可以，默认非公平锁

5：绑定多个条件condition

Synchronized没有，ReentrantLock用来实现分组唤醒，可以精确唤醒，而不是像Synchronized要么唤醒一个线程，要么唤醒全部线程。

## LockSupport

线程的阻塞工具类，所有方法都是静态方法，可以让线程在任意位置阻塞。LockSupport调用的Unsafe中的native代码

Park()、unpark()分别用于阻塞和解除阻塞线程

Park()方法重复使用，信号量只有一个起作用，且unpark可以再park之前使用

## ThreadLocal

线程的局部变量，每个线程单独持有的其他线程不可访问，通常是线程的 private static 字段。可以打印线程 id 以及线程名等信息

## 死锁

死锁是指两个或两个以上的进程在执行过程中，因抢夺资源而造成一种相互等待的现象

定位： jps命令找到死锁对应的线程id

再使用jstack pid 找到导致死锁的方法

## AQS (AbstractQueuedSynchronizer) 抽象队列同步器

用来构建锁或者其他同步器组建的重量级基础框架及整个JUC体系的基石，通过内置的FIFO队列来完成资源获取线程的排队工作，并通过一个int类型变量表示持有锁的状态

通过state+CLH队列

## 自旋锁

是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU

## ABA问题

atomicStampedReference加版本号

## BlockingQueue（阻塞队列）

### ArrayBlockingQueue

由数组结构组成的有界阻塞队列



### LinkedBlockingQueue

由链表结构组成的有界（但大小默认为Integer.MAX\_VALUE）阻塞队列

### SynchronousQueque

不存储元素的阻塞队列，也即单个元素的队列，只有元素被取出后才会添加下一个元素

## 线程池的线程数量设置

Cpu核数 = Runtime.getRuntime().availableProcessors()

CPU密集型：cpu核数+1

IO密集型：1、不是一直执行任务的 cpu核数 \* 2

2、IO大量阻塞，cpu核数/（1-阻塞系数） 阻塞系数在0.8 –0.9之间

# Redis

## 持久化

### AOF

**优点：**机制可以带来更高的数据安全性，即数据持久性

**缺点：**AOF 文件通常要大于 RDB 文件。 AOF 在恢复大数据集时的速度比 RDB 的恢复速度要慢。

### RDB

**优点：** RDB 对 Redis 对外提供的读写服务，影响非常小，可以让 Redis 保持高性能。恢复更快，相比于AOF机制，RDB的恢复速度更快，更适合恢复数据，特别是在数据集非常大的情况

**缺点：**系统一旦在定时持久化之前出现宕机现象，此前没有来得及写入磁盘的数据都将丢失。

采用主从机器的，一般 master 机器采用 AOF 保障数据安全性， Slave机器采用 RDB 较多或 AOF + RDB

## 过期策略

**策略一：**惰性删除就是在客户端使用 key 的时候， Redis 检查这个 key 的过期时间，如果到期了就立即删除。

**策略二：**定时删除，设置 key 的过期时间的同时，为该 key 设置一个定时器，让定时器在 key 的过期时间到来的时候，对其进行删除**策略三：**定期删除

## 双写一致性问题

更新数据库和更新缓存时间，导致数据库和缓存的数据不一致的问题

### 策略

**串行化：**效率低

**先更新数据库再更新缓存:**

A线程刚更新完数据库，B线程更新数据库，B线程再更新缓存， A线程更新缓存。缓存容易有脏数据

**先删除缓存再更新数据库:**

A删除缓存，B请求缓存没有，查询数据库B更新旧缓存,A更新数据库缓存仍然时旧数据

**先更新数据库再删除缓存：**

A更新数据库，B请求数据，请求B将新值写入数据库，B 、删除缓存，A更新缓存会产生脏数据，但概率低。

## 数据“淘汰”策略

- noeviction （默认策略）：对于写请求不再提供服务，直接返回错误（ DEL 请求和部分特殊请求除外）

- allkeys - Iru ：从所有 key 中使用 LRU 算法进行淘汰

- volatile - Iru ：从设置了过期时间的 key 中使用 LRU 算法进行淘汰

- allkeys - random ：从所有 key 中随机淘汰数据

- volatile - random ：从设置了过期时间的 key 中随机淘汰

- volatile - ttl ：在设置了过期时间的 key 中，根据 key 的过期时间进行淘汰，越早过期的越优先被淘汰

当使用 volatile - Iru 、volatile - random 、 volatile - tl 这三种策略时，如果没有 key 可以被淘汰，则和 noeviction 一样返回错误

## 延迟队列如何做的

1: Redis sorted set 利用它的score属性，Redis 通过 score 来为集合中的成员进行从小到大的排序

2:Redis过期回调简单来说我们开启监听 key 是否过期的事件，一旦 key 过期会触发一个 calback 事件。

3：时间轮

4: DelayQueue延时队列：队中元素要implements Delayed接口，接口里只有一个getDelay方法，用于设置延期时间。Order 类中

compareTo 方法负责对队列中的元素进行排序。

## 布隆过滤器

bf.add 添加一个元素

bfexists 判断一个元素是否存在

bf.madd 添加多个元素

bf.mexists 判断多个元素是否存在

将存在的数据向 bitmap 二进制位图中标记，用户请求的时候先通过映射函数查询 bitmap ，存在的就放行，不存在则被拦截，会存在误标记放行的问题，但概率底，成本小。

## 脑裂

当一个集群中的 master 恰好网络故障，导致与sentinal 通信不上了，sentinal 会认为 master 下线，且sentinal选举出一个slave 作为新的master，此时就存在两个 master 了

### 产生的问题

主库的数据还没有同步到从库，结果主库发生了故障，等从库升级为主库后，未同步的数据就丢失了

### 解决方案

可以开发一个外部程序来监控主从库间的复制进度

( master repl \_ offset 和slave \_ repl \_ offset )，通过监控 masterrepl \_ ofset 与slave repl ofset 差值得知复制进度，当复制进度不符台预期设置的 Client 不再从该从库读取数据。

# Mysql

## 分库分表

**分库：**为了解决数据库连接资源不足的问题，和磁盘 O 的性能瓶颈问题

**分表：**为了解决单表数据量太大，sql 语句查询数据时，即使走了索引也非常耗时，此外还解决了消耗 cpu 资源的问题

用到的中间件：sharding

## MVCC(多版本并发控制)

读取数据时通过一种类似快照的方式将数据保存下来，这样读锁和写锁就不冲突了，不同的事务session会看到自己特定版本的数据

MVCC只在读已提交和重复读两个级别下工作，其他隔离级别和mvcc不兼容，因为读未提交总是读取最新的数据行，而不是符合当前事务版本的数据行。而穿行话则会对所有读取的行都加锁

## 主从复制

主从复制中主要有三个线程：master（binlog dump thread）、slave(I/O thread 、SQL thread)

Master一条线程和slave中的两条线程

* 主节点binlog，主从复制的基础时主库记录数据库的所有变更记录到binlog.binlog是数据库服务器启动的那一刻起，保存所有修改数据库结构或内容的一个文件
* 主节点log dump线程，当binlog有变动时，log dump线程读取其内容并发送给从节点
* 从节点I/O线程接收binlog内容，并将其写入到relay log文件中
* 从节点的SQL线程读取relay log文件内容对数据更新进行重放，最终保证主从数据库的一致性

注：主从节点使用binlog文件 + position偏移量来定位主从同步的位置，从节点会保存其已接收到的偏移量，如果从节点发生宕机重启，则会从position的位置发起同步

由于mysql默认的复制方式是异步的，主库把日志发送给从库后不关心是否已经处理，这样会产生一个问题就是假设主库挂了，从库处理失败了，这时候从库升为主库后，日志丢失了。

**全同步复制：**主库写入binlog后强制同步日志到从库，所有的从库都执行完后才返回给客户端，性能会受到严重影响

**半同步复制：**和全同步不同的是，半同步复制是从库写入日志后返回ack确认给主库，主库只需要收到至少一个从库的确认就认为写操作完成

## B＋树

1. 非叶子节点不保存数据，只保存关键字用作索引，所有数据都保存在叶子节点中
2. 叶子节点之间存在指针并且升序排列
3. 每个节点可以存储多个元素，从而使得B+树高度不会太高，减少 IO 提高查询效率
4. 叶子节点之间都存在指针

## 执行计划

重要的字段： id 、type、key 、rows、extra

id 相同：执行顺序由上至下， id 值越大，优先级越高，越先执行

select\_type ：查询的类型，主要是用于区分普通查询、联合查询、子查询等复杂的查询

( SIMPLE 、 PRIMARY 、 SUBQUERY 、 DERIVED 、 UNION )

type ：很重要的指标，结果值从好到坏依次是 system > const > eq \_ ref > ref > fulltext > range > index > ALL

一般来说， sq 查询至少达到 range 级别，最好能达到 ref级别

key ：实际使用的索引，如果为 NULL ，则没有使用索引。

key\_len ：索引中使用的字节数，长度越长，精确度越高但效率越低。所以在不损失精度的情况下，长度越短越好

rows ：大致估算出找到所需的记录所需要读取的行数

## 覆盖索引（ Covoring Incdex )

也叫索引覆盖。就是 select列表中的字段，只用从索引中就能获取，不必根据索引再次读取数据文件，换句话说查询列要被所建的索引覆盖。

## 索引失效的场景

1）Like模糊查询％在前

2）最左匹配使用不等于号

3）对索引进行操作（计算、函数、类型转换等）

4）where 语句中使用or，但是没有把 or 中所有字段加上索引

## MylSAM 与 lnnoDB 的区别

1) InnoDB 支持事务， MyISAM 不支持

2) InnoDB 支持外键， MyISAM 不支持

3）清空整个表时， InnoDB 是一行一行的删除，效率非常慢。 MylSAM 则会重建表。

4) InnoDB 支持行锁（某些情况下还是锁整表，如 update table set a =1 where user like ‘% lee %')

5）从MySQL5.5.5以后， InnoDB 是默认引擎

6) InnoDB 不支持 FULLTEXT 类型的索引

7) InnoDB 中不保存表的行数，

如 select count () from table 时， InnoDB 需要扫描一遍整个表来计算有多少行，但是 MylSAM 只要简单的读出保存好的行数即可。注意的是，当 count (）语句包含 where 条件时 MylSAMM 也需要扫描整个表。

8）对于自增长的字段， InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引。

9) MylSAM 适合查询和插入为主的应用， InnoDB 适合频繁修改以及涉及到安全性较高的应用

## 聚簇索引和非聚簇索引

**聚簇索引：**索引和数据存储在一起，找到索引就能直接找到数据，物理地址连续

**非聚簇索引**：索引和数据不存储在一起，需要通过索引回表得到数据，物理地址不连续

总的来说，聚集索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块

## 锁分类

加锁策略分类：悲观锁和乐观锁

粒度分类：表级锁、行级锁、页面锁、间隙锁

共享策略分为：读锁（共享锁）、写锁（排它锁）、意向共享锁、意向排它锁

## MySQL 事务隔离级别

(1) read uncommitted ：读取尚未提交的数据：不解决脏读

(2) read commited ：读取已经提交的数据：可以解决脏读

(3) repeatable read ：重读读取：可以解决脏读和不可重复读- mysql 默认的

(4) serializable：串行化：可以解决脏读不可重复读和虚读一相当于锁表

redolog ：记录的是数据页的物理修改，它用来恢复提交后的物理数据

## 分布式ID

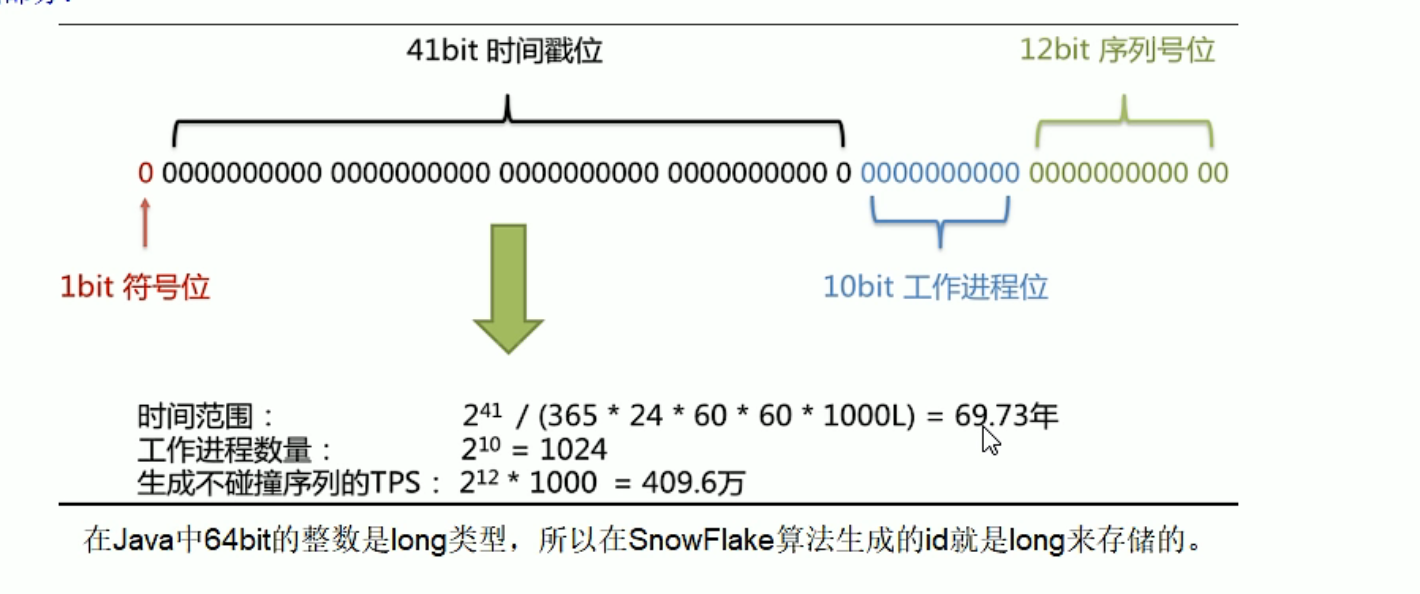
UUID:只考虑唯一性可以使用，但无序会导致索引分裂导致数据插入性能差，mysql官方推荐主键建议越短越好

数据库自增主键：满足了唯一性和自增的要求 数据库的自增ID机制主要原理是通过自增ID和数据库的replace into实现的（先判断是否存在，如果存在先删除再插入，否则直接插入）。但当并发量大的时候性能会很差，或者集群部署的时候一台机器宕机，其他机器会获取ID超过步长，产生ID冲突

Redis集群生成ID,可以满足自增和单调递增的要求，但是维护麻烦，得不偿失

雪花算法：生成一个64bit大小的整数（Long型，长度为19的字符串），分布式系统内不会产生ID碰撞，且效率高。

使用 hutool工具包中的Snowflake类传入workId 和datacenterId



优缺点：

优点： 全局唯一且趋势递增，

不依赖第三方系统稳定性高，生成ID的性能也很高

可以根据自身业务特性分配bit位，更加灵活

缺点

依赖机器始终，如果机器始终回拨，会导致重复ID生成（概率极低）

## 事务的特性

## hash索引和B+树索引的区别

1、在查询速度上，如果是等值查询，那么Hash索引明显有绝对优势，因为只需要经过一次 Hash 算法即可找到相应的键值,前提是键值唯一的。如果键值不是唯一(或存在Hash冲突)，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据，降低了Hash索引的查找效率。

2、Hash 索引是无序的，如果是范围查询检索，这时候 Hash 索引就无法起到作用，即使原先是有序的键值，经过 Hash 算法后，也会变成不连续的了。因此

①、Hash 索引只支持等值比较查询、无法索成范围查询检索，B+tree索引的叶子节点形成有序链表，便于范围查询。

②、Hash 索引无法做 like ‘xxx%’ 这样的部分模糊查询，因为需要对 完整 key 做 Hash 计算。而 B+tree 索引具有最左前缀匹配，可以进行部分模糊查询。

③、Hash索引数据库无法利用索引的数据来避免任何排序运算。B+tree 索引的叶子节点形成有序链表，可用于排序。

3、Hash 索引不支持多列联合索引

4、因为存在哈希碰撞问题，在有大量重复键值情况下，哈希索引的效率极低。B+tree 所有查询都要找到叶子节点，性能稳定

## 什么情况下使用hash索引，什么情况下使用B+树索引

 1、大多数场景下，都会有组合查询，范围查询、排序、分组、模糊查询等查询特征，Hash 索引无法满足要求，建议数据库使用B+树索引。

2、在离散型高，数据基数大，且等值查询时候，Hash索引有优势。

# Spring

## IOC（控制反转）

控制反转：原来的对象是由使用者来控制，有了Spring之后，可以把对象交给Spring来管理

DI（依赖注入）：把对应的属性值注入到具体的对象中。@Autowired,@populateBean完成属性值的注入

容器：存储对象，使用map结构来存储，在Spring中一般存在三级缓存，singletonObject存放完整的bean对象，整个bean的生命周期，从创建到使用再到销毁的过程全部由容器来管理。

## AOP（面向切面）

总：AOP 是IOC的一个扩展功能，主要应用于事务管理和日志打印。通过动态代理来实现

分：Bean的创建过程中有一个步骤可以对bean进行扩展实现，aop本身就是一个扩展功能，所以在BeanPostProcessor的后置处理方法中来进行实现

1：代理对象的创建过程（advice、切面、切点）

2：通过jdk或cglib的方式来生成代理对象

3：在执行方法调用的时候，会调用到生成的字节码文件中，直接会找到DynamicAdvisoreInterceptor类中的intercept方法，从此方法开始执行

4：根据之前定义好的通知来生成拦截器链

5：从拦截器中一次获取每一个通知开始进行执行，在执行过程中为了方便找到下一个通知是哪个会有一个CglibMethodIncation的对象找的时候是从-1的位置开始查找并且执行的。

## JDK 动态代理和 CGLib 动态代理的区别

1：JDK动态代理利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler 来处理，而 cglib 动态代理是利用 asm 开源包，对代理对象类的 class 文件加载进来，通过修改其字节码生成子类来处理。

2：JDK动态代理只能对实现了接口的类生成代理，而不能针对类。CGLIB 是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法（继承）

### Spring 在选择用 JDK 还是 CGLiB 的依据：

(1）当 Bean 实现接口时，Spring 就会用 JDK 的动态代理

(2）当 Bean 没有实现接口时，Spring 使用 CGlib 是实现

(3）可以强制使用 CGlib（在 spring 配置中加入< aop : aspectj - autoproxy proxy - target - class =" true "/>)

### Spring5以后的通知执行顺序

正常: around之前 -> before -> 业务逻辑-> afterReturing -> after -> around之后

异常：around之前 -> before -> 业务逻辑 -> afterThrowing

-> after

## bean 的加载过程

xml --＞加载﹣->解析--＞封装 -＞对象的创建

1：获取 BeanName ，对传入的 name 进行解析，转化为可以从 Map 中获取到 BeanDefinition 的 bean name

2：合并Bean定义进行合并和覆盖，如果父类还有父类，会进行递归合并，以获取完整的 Bean 定义信息

3：实例化，使用构造或者工厂方法创建 Bean 实例

4：属性填充，寻找并且注入依赖，依赖的 Bean 还会递归调用 getBean 方法获取

5：初始化，调用自定义的初始化方法

6：获取最终的 Bean ，如果是 FactoryBean 需要调用 getObject 方法，如果需要类型转换调用 TypeConverter 进行转化

7：整个流程最为复杂的是对循环依赖的解决方案，后续会进行重点分析

## BeanFactor 和 ApplicationContext

BeanFactor 是ApplicationContext 的父接口

## bean 的生命周期：

1：通过反射实例化bean对象

2：通过populateBean方法设置对象属性(存在循环依赖的问题，通过三级缓存处理)

3: 检查Aware相关接口，并设置相关依赖 invokeAwareMethod(完成BeanName,BeanFactory,BeanClassLoader对象的属性设置)

4: 调用BeanPostProcessor的前置处理方法：使用比较多的有：ApplicationContextPostProcessor,设置ApplicationContext,Environment、ResourceLoader等

5：判断是否实现InitializingBean以决定是否调用afterPropertiesSet方法

6：检查是否配置有自定义的init-method

7：BeanPostProcessor后置处理，SpringAOP就是在此实现，AbstractAutoProxyCreator

8：使用getBean()获取bean的对象对象，然后使用

9：判断是否实现了dispoableBean接口，调用destroy() 方法，销毁对象

### bean 的作用域

single （默认）、protoType 、 session 、 request 、global

## 循环依赖的问题的解决方式

**属性赋值阶段构造器的循环依赖**：解决不了（形成一个依赖环，无法解决）

**prototype 的循环依赖**：解决不了( Spring 容器无法完成依赖注入，因为 Spring 容器不进行缓存" prototype "作用域的 bean )

**setter 循环依赖**：比如先实例化 A ，实例化好了之后会将 A 放入缓存中，在 A 的属性赋值时发现依赖 B ，再去实例化 B 并进行 B 的属性赋值，发现 B 依赖于 A 就从缓冲池中取到 A 并对 B 进行属性赋值。 A 进行属性赋值时，将创建好的 B 对象赋值给 A

### @ Resource 和＠ Autowired 的区别

@Autowired 是 spring 的注解，是基于类型匹配，匹配不到再按照类型匹配，可以写在 setter 方法上

#### Autowired如何按名称注入bean属性

使用@Qualifier(“className”)匹配

## @ Controler 和＠ RestControler 的区别

@ RestControler 注解相当于@ ResponseBody +@ Controler 合在一起的作用。

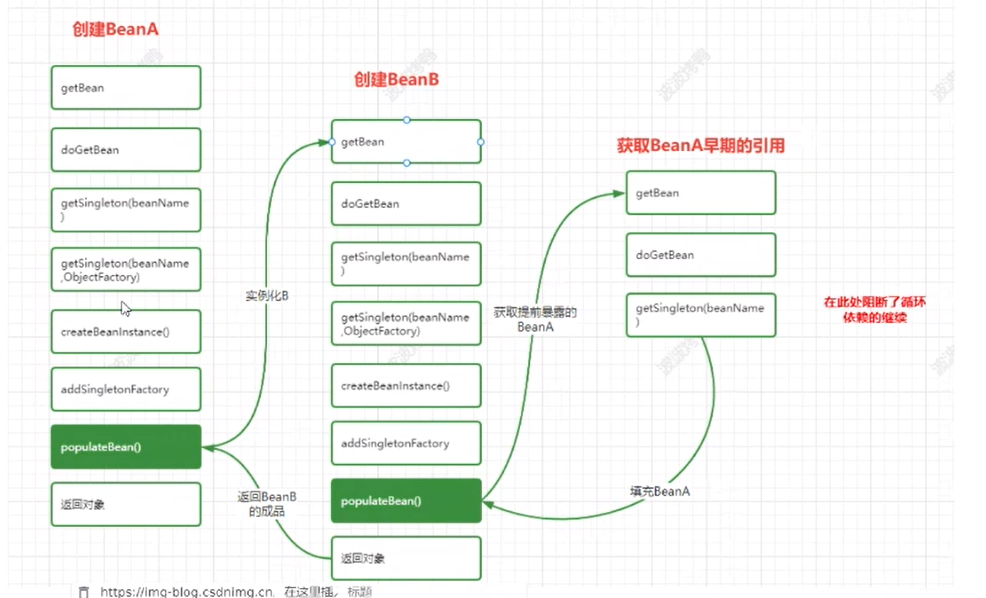
如果返回 json 格式的就用@RestController,因为它会自动将对象实体转换为json格式。如果返回页面的话，就要用Controller注解

## BeanFactory和FactoryBean的区别：

相同点：都是用来创建bean对象的

不同点：使用BeanFactory创建对象的时候，必须遵循严格的生命周期流程。如果要简单的自定义某个对象的创建，同时创建完成的对象交给Spring来管理，那么就需要实现FactoryBean接口了

## 循坏依赖的解决办法

答：

1：构造器循环依赖无法处理

2：setter方法的额循环依赖问题，A实例化之后会被提前暴露在缓存中，当初始化B的 时候，会从缓存中获取被提前暴露的A对象，完成B 对象的获取，然后再对A对象进行赋值

## 三级缓存分别的作用

一级缓存：正式对象

二级缓存：半成品对象

三级缓存：工厂

## Spring事务传播机制

REQUIRED（默认）：支持使用当前事务，如果当前事务不存在，创建一个新事务。

SUPPORTS：支持使用当前事务，如果当前事务不存在，则不使用事务。

MANDATORY：中文翻译为强制，支持使用当前事务，如果当前事务不存在，则抛出Exception。

REQUIRES\_NEW：创建一个新事务，如果当前事务存在，把当前事务挂起。

NOT\_SUPPORTED：无事务执行，如果当前事务存在，把当前事务挂起。

NEVER：无事务执行，如果当前有事务则抛出Exception。

NESTED：嵌套事务，如果当前事务存在，那么在嵌套的事务中执行。如果当前事务不存在，则表现跟REQUIRED一样。

# Springboot：

## 启动流程

1：新建SpringApplication对象

2：调用getRunListeners创建所有spring监听器

3：DefaultApplicationArguments初始化应用应用参数

4：prepareEnvironment根据运行监听器和参数准备spring环境

5：调用createApplicationContext方法创建应用上下文

6：通过prepareContext准备应用上下文

7：refreshContext方法刷新上下文

8：调用started发布应用上下文启动完成事件

9：callRunners方法执行所有runner运行器

10：调用running发布应用上下文就绪事件，返回ApplicationContext对象

## @SpringBootApplication常用注解的原理：

**@SpringBootApplication:这个注解标识了一个SpringBoot工程，它实际上是另外三个注解的组合：**

[**@SpringBootConfiguration**](mailto:a.@SpringBootConfiguration)**:实际就是一个@configuration，表示启动类也是一个配置类**

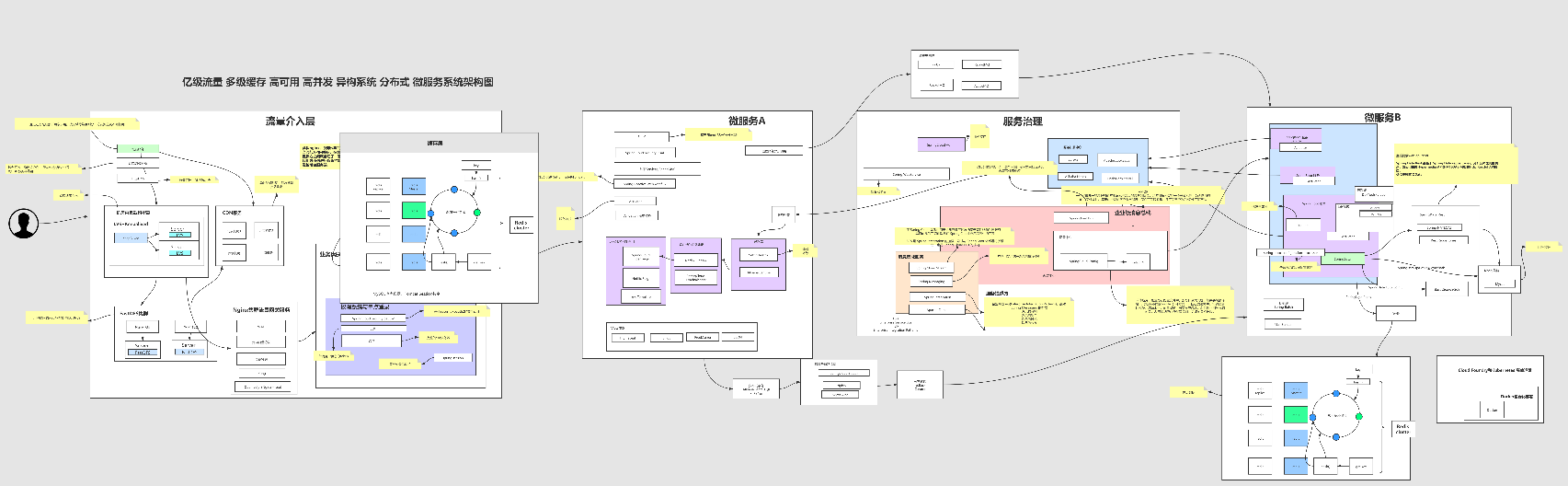
**@EnableAutoConfiguration：向Spring容器中导入了一个Selector,用来加载ClassPath下的SpringFactories所定义的自动配置类，将这些自动加载为配置Bean**

**@ComponentScan:标识扫描路径，因为默认是没有配置实际扫描路径，所以SpringBoot扫描的路径是启动类所在的当前目录**

**2：@Bean注解：用来定义Bean,类似于Xml中的<bean>标签**

**3：@Controller,@Service,@ResponseBody、@Autowired注解等**

# SpringCloud



## GeteWay

特性：

基于Spring Framework 5，Project Reactor 和Spring Boot 2.0进行构建

动态路由：能够匹配任何请求属性

可以对路由指定Predicate(断言)和Filter（过滤器）  
 继承Hystrix的断路器功能

继承Spring Cloud服务发现功能

## Spring cloud Gateway与Zuul的区别

核心概念

1. Route(路由)：构建网关的基本模块，由ID、目标URI一系列的断言和过滤器组成，如果断言为True则匹配该路由
2. Predicate（断言）：开发人员可以匹配HTTP请求中的所有内容（例如请求头或请求参数）如果请求与断言相匹配则进行路由
3. Filter（过滤）：指的是Spring框架中GateWayFilter的实例，使用过滤器可以在请求被路由前或之后对请求进行修改

## Sleuth链路追踪

# Mybatis

动态 sq 标签：

**< if >< set >< where >< foreach >< trim >< include >**

**<choose>**

**<when><otherwise>**

**</ choose >**

## 一级缓存、二级缓存

**一级缓存：sqlsession 上的缓存，默认开启是键值对类型的数据缓存，查询同一个 nameSpace 的同一个 sqlid 且参数一致，会在缓存区中查找对应的健，如果有则不会再次请求数据库查询而是通过缓存获取到，可以减少数据库查询次数，降低数据库的负担。**

**key 的生成方式：namespace + sqlid ＋参数+ ofset --＞生成一个 hash 值作为健，查询的结果作为值**

**查询之间发生了增删改或者sqISession 调用 commint ，会自动清空缓存（可以防止脏读问题）**

**二级缓存：是以 namespace 为标记的缓存，是由一个sqlsessionFactory 创建的**

**sqlsession 之间共享缓存数据。默认不开启，且要求实体类必须实现serializable 接ロ否则会报异常**

**（开启设置：全局：< setting name =" cacheEnaled " value =" true ">; namespace 层：< cache >; sql 层： useCache =" true ")**

**需要 sqlsession 调用 commit 之后，才会将查询的结果放入二级缓存**

## ****一对一、一对多、多对多问题****

**一对一：Java：将一个对象设为另一个对象的属性**

**xml：通过＜association property ="属性名称” javaType ="类名“＞标签封装一对一信息关系**

**一对多：Java ：将多个对象使用 list 集合设为一个对象里的属性**

**xml ：使用<colection property ="属性名称＂ ofType ="类名“＞标签处理一对多的关系**

**多对多：三张表的级联查询，association 和 colection 标签的组合使用**

## #{}和${}的区别

#{}是采用预编译的方式，可以防止sql注入的风险

${}一般用于传入数据库对象，例如传入表名，没有采用预编译，不可以防止sql注入

**注：MyBatis排序时使用order by 动态参数时需要注意，用$而不是#**

## ResultMap和ResultType的区别

都是返回对象信息， resultMap要配置一下，表和类的一一对应关系，所以说就算你的字段名和你的实体类的属性名不一样也没关系，都会给你映射出来，但是，resultType必须字段名一样

resultMap可以用在复杂联合查询上，而resultType不可以。

## 常用的接口和类

SqlSessionFactoryBuilder：读取核心配置文件生成会话工厂的类，初始化[Configuration](https://so.csdn.net/so/search?q=Configuration&spm=1001.2101.3001.7020)对象。

[SqlSessionFactory](https://so.csdn.net/so/search?q=SqlSessionFactory&spm=1001.2101.3001.7020)新建和管理会话的工厂类

Configuration对应mybatis-config配置文件的一个对象,里面包含了mybatis配置信息和映射器配置文件信息

MappedStatement对应着映射器配置文件的每一条statment语句配置

Executor执行器接口,就像发送器

resource 读配置文件

SimpleExecutor执行语句文件

Sqlsession、SqlsessionFactory、SqlsessionFactoryBuilder

# Kafka

## 概念

kafka是一个分布式、支持分区多副本的，基于zookeeper的分布式消息系统。其最大的特性就是可以实时的处理大量数据以满足各种需求场景。

### 名词解释

#### Broker

消息中间件处理节点，一个kafka节点就是一个broker，一个或多个Broker可以组成一个kafka集群

#### Topic

kafka根据topic对消息进行归类，发布到kafka集群的每条消息都需要指定一个Topic

#### Producer

消息生产者，向Broker发送消息的客户端

#### Consumer

消息消费者，从Broker读取消息的客户端

#### ContsumerGroup

每个Consumer属于一个特定的ConsumerGroup，一条消息可以被多个不同Consumer Group消费，但是一个Consumer Group中只能由一个Consumer能够消费该消息

#### Pratition

物理上的概念一个topic可以分为多个partition，每个partition内部消息是有序的。

## Zookeeper在kafka中的作用

Kafka的broker注册，topic的注册，consumer注册，生产者和消费者的负载均衡、确定分区和消费者的关系

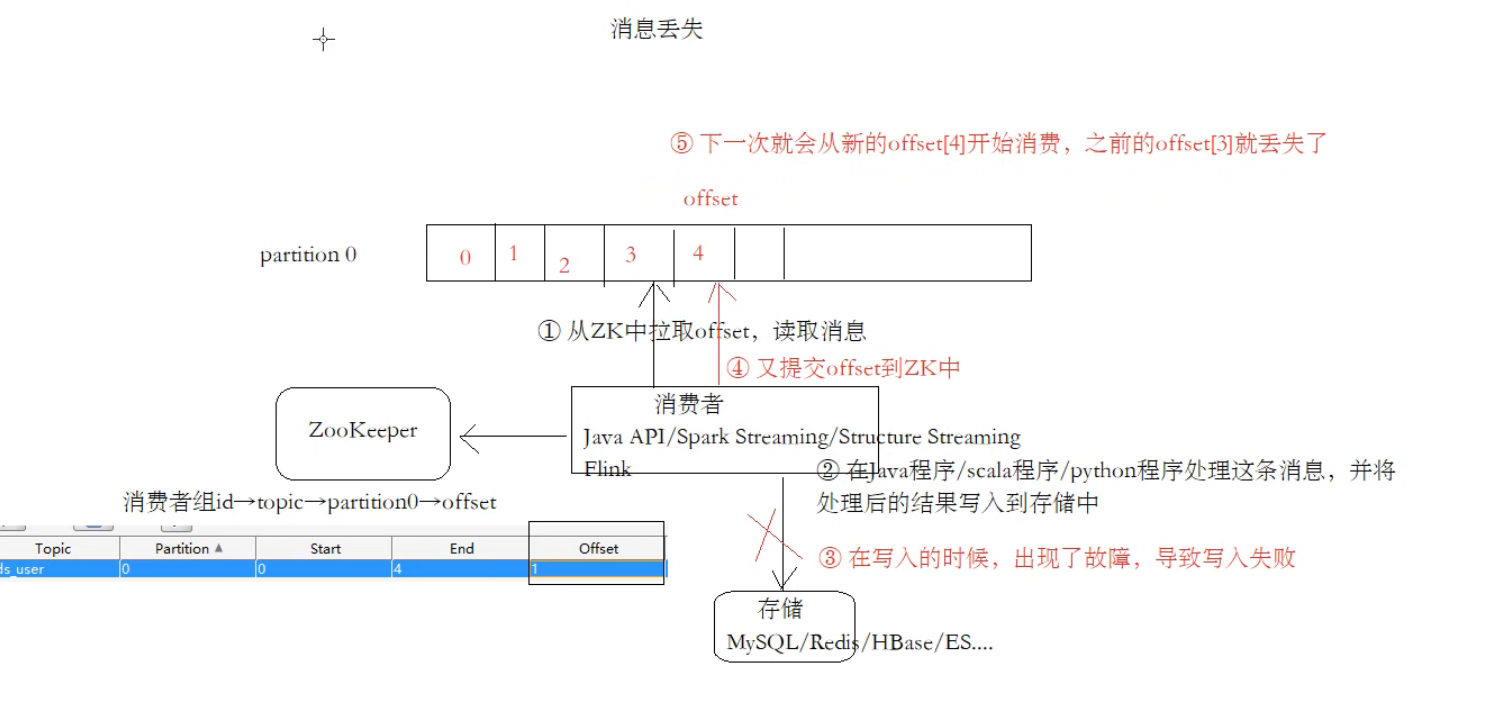
## Kafka集群消费

1、一个partition只能被一个消费组中的一个消费者消费，目的是为了保证消费的顺序性，但是多个partition的消费者消费总的顺序性是得不到保证的

2、Partition的数量决定了消费组中消费者的数量，建议同一个消费组中消费者数量不要超过partition的数量，否则多余的消费者消费不到消息

3、如果消费者挂了，会触发rebalance,会让其他消费者来消费该分区

## 如何保证消息不丢失



**生产者：**可以通过ack机制来确保数据已经写入，使用同步发送

Ack=-1生产者发送过来的数据，Leader和ISR队列里的所有节点收齐数据后应答（可靠性高，但效率低） 可能会产生数据重复消费的问题。里面有默认min.insync.replicas=1配置

最好是ack=-1 + 分区副本大于等于2+ISR应答的最小副本数量大于等于2

Ack=0生产者发送过来的数据，不需要等数据落盘（可靠性差，但效率高）

Ack=1生产者发送过来的数据，Leader收到数据后应答（可靠性适中，但效率始终）

把min.insync.replicas设置成分区备份数>=2

**消费者：**在每次消费完成后只要记录好offeSet值,自动提交改为手动提交

## 消息可靠性

分区多副本，生产这发送数据时配置ack=-1，min.insync.replicas>=2

## 自动提交和手动提交offset

自动提交：消费者poll消息下来后就会自动提交offset,存在消息丢失的情况，消费者还没开始消费，就挂掉了

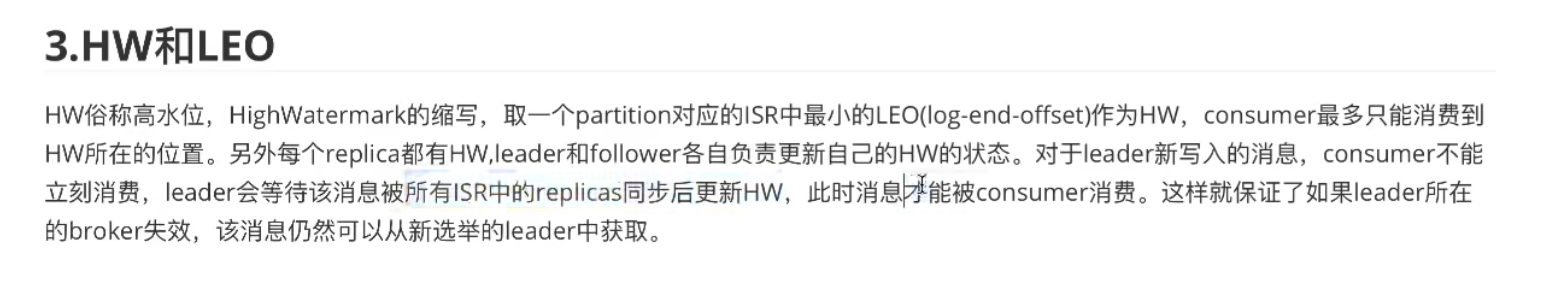
手动提交：在消费完消息后再提交offset,需要把自动提交的配置改成false

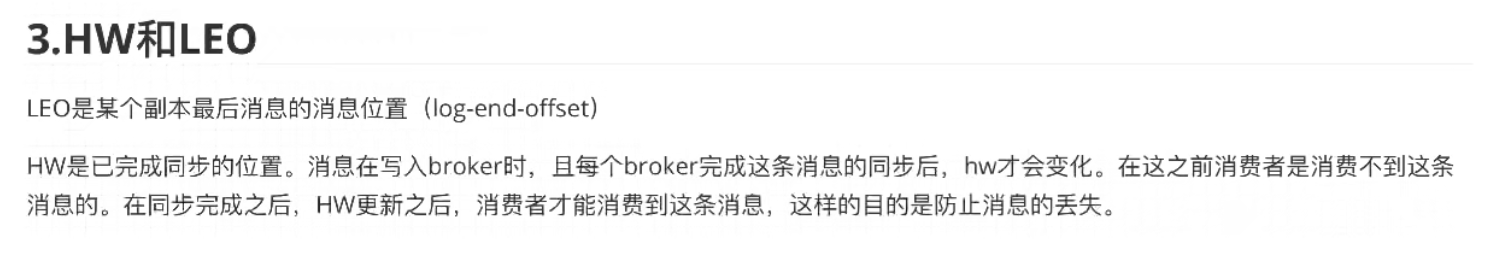
手动同步提交： 在消费完消息后调动同步提交方法，集群返回ack前一直阻塞，返回ack后表示提交成功，执行之后的逻辑

手动异步提交：消费完消息后调用异步提交方法，不需要等待ack,直接执行逻辑，可以设置一个回调方法









## 幂等性

不论生产者向broker发送多少次数据，broker端都会只持久话一条，保证了不重复

## 重复消费问题

发生场景：生产者发送完消息后，因为网络抖动没有收到ack,但实际上broker已经收到，此时生产者进行重试，于是就会收到多条相同的消息，造成消费者的重复消费

解决办法：

生产者关闭重试，会造成消息丢失的情况（不建议）

消费者端解决非幂等性消费问题

1）mysql使用联合主键，防止小童的主键创建出多条记录

2）分布式锁，以业务id为锁保证只有一条记录能够创建成功

## 消息的顺序消费

发送方：ack不能设置为0，使用同步发送

接收方：主题只能设置一个分区，消费组中只能有一个消费者

Kafka的顺序消费使用场景不多，因为会牺牲掉性能，但rocketMq可以有专门的设计

## 消息积压

原因：消费者的消费速度远赶不上生产者的生产速度，导致有大量的消息没有被消费，随着没有被消费的数据越积越多，消费者的寻址性能越来越差，最后导致整个kafka对外提供的服务性能越来越差，从而其他服务访问变慢，造成服务雪崩

解决方案：

1. 在这个消费者中使用多线程，充分利用机器性能
2. 创建多个消费组，多个消费者部署到其他机器上，一起消费，提高消费者的消费速度
3. 创建一个消费者，该消费者在kafka另建一个主题，配上多个分区，多个分区再配上多个消费者，该消费者将poll下来的消息，不进行消费，直接转发到新建的主题上。

## 延时队列的实现

### Kafka实现

1、Kafka中创建对应的topic

1. 生产者将消息发送到topic中，消费者做轮询消费topic中的消息
2. 判断是否超时，如果超时去数据库中修改将该消息为取消状态
3. 如果没有超时，则记录当前offset值，停止消费。继续轮询topic消息

### Redis实现

使用ZSet的score属性，给zset添加数据，时间就是它的score