#### MySQL锁

**不同存储引擎支持不同锁机制**

**MyISAM、MEMORY存储引擎采用表级锁**

**InnoDB存储引擎支持行级锁，也支持表级锁，默认行级锁**

##### 行锁 表锁

行锁、表锁、库锁

1. 行锁：访问数据库，绑定整个行
2. 表锁：访问数据库，绑定整个表

区别

* 表锁

开销小，加锁快，不会出现死锁，锁定粒度大，发生锁冲突概率高，并发度最低。

* 行锁

开销大，开锁慢，会出现死锁，锁定粒度小，发生锁冲突的概率低，并发度高。

##### 乐观锁 悲观锁

悲观锁：

每次拿数据都会上锁、其他请求要拿数据只能block直到拿到锁 Synchronized

悲观锁适合写操作多的场景 先加锁可以保证写操作时数据正确。

乐观锁适合读操作多的场景，不加锁让读操作性能大幅提升。

乐观锁：

更新的时候会判断一下（多用于多读的应用类型，提高吞吐量）

CAS算法（Compare and swap）

需要读写的内存值V

进行比较的值 A

写入的新值 B

当且仅当预期A与内存值V相同时，将内存值V修改为B，否则返回V

###### 区别

乐观锁适用于写比较少的情况下，如果经常产生冲突，上层应用会不断retry，反倒降低性能，这种情况下用悲观锁比较合适。

##### 共享锁（悲观锁的一种）

Lock in share mode 共享锁保证并发读非常高效

同时访问资源

InnoDB什么时候使用表锁

1. 事务需要更新大部分数据、表大、使用默认行锁效率低，而且可能造成其他事务长时间锁等待和锁冲突
2. 事务设计多个表，比较复杂，容易引起死锁，造成大量事务回滚，故而一次性锁定事务涉及的表，从而避免死锁、减少数据库因事务回滚带来太多的开销

如果类似情况过多，就要使用MyISAM表

###### 表锁和行锁应用场景

表级锁使用与并发性不高，以查询为主，少量更新的 应用 （小型web应用）

行级锁适用于高并发环境下，对事务完整性要求较高的系统 （在线事务处理系统）

##### 独享锁（悲观锁的一种）

Java ReentrantLock

###### AQS

抽象队列同步器,用来构建锁或者其他同步组件的基础框架，使用一个整型的volatile变量，来维护同步状态，通过内置的FIFO队列完成资源获取线程的排队工作。

|  |  |
| --- | --- |
| 高层类 | Lock、同步器、阻塞队列、Executor、并发容器 |
| 基础类 | AQS、非阻塞数据结构、原子变量类 |
|  | Volatile变量的读/写、CAS |

Concurrent包实现结构如上

一层基于一层实现

##### 分段锁（一种锁设计，不是具体的锁）

ConcurrentHashMap而言，并发实现是通过分段锁的形式实现高效的并发操作。

ConcurrentHashMap中的分段锁称为Segment，类似于HashMap的结构，内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock

（Segment继承了ReentrantLock）

当需要put元素的时候，并不是对整个HashMap进行加锁，而是先通过HashCode来知道他要放在哪一个分段中，然后对这一个分段进行加锁，当多线程put的时候，只要不是放在一个分段中，就真正实现了并行插入。

但是，在统计size的时候，可就是获取HashMap全局信息的时候，需要获取所有分段锁才能统计。

分段锁设计目的是细化锁的粒度，让操作不需要更新整个数组，仅仅针对数组中一项进行加锁操作。

#### JVM调优

* + OutOfMemoryError 内存不足
  + 内存泄露
  + 线程死锁
  + Lock Contention 锁争用
  + Java进程消耗CPU过高

这些问题出现的时候 常通过重启服务器或者调大内存临时解决

1. **JVM内存模型**
2. **JVM垃圾回收算法**
3. **JVM垃圾回收器**
4. **JVM参数详解**
5. **JVM性能调优**

###### JVM内存结构

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Eden | FromSurvivor | To Survivor |  | | | |
| 新生代 | | | 老年代 |  | Jaba虚拟机栈 | 本地方法栈 |
| 堆 | | | | 方法区（永久代） | 栈 | |
| JVM内存结构 | | | | | | |

JVM内存空间分为3大部分：

* + 堆内存
  + 方法区
  + 栈内存

栈内存，可以细分为java虚拟机栈和本地方法栈，堆内存可以划分为新生代和老年代，新生代可以再继续划分为Eden、From Survivor、To Survivor

一部分是线程共享，包括java堆和方法区；另一部分是线程私有，包括虚拟机栈和本地方法栈，以及程序计数器这一小部分内存

##### 堆内存（Heap）

Java堆 是java虚拟机所管理的内存中最大的一块。堆是被所有线程共享的区域，是在虚拟机启动时创建的。堆里面存放的都是对象的实例

此内存区域的唯一目的就是存放对象实例，几乎所有对象实例都在这里分配内存。

堆内存分为两个部分：

新生代、老年代 （比例 1:2） -XX:NewRatio

##### 方法区（Method Area）

永久代，用于存储虚拟机加载的类信息、常量、静态变量，各线程共享的内存区域。

JDK8前 默认大小是64M

JDK8到来，JVM不再有永久代，类元数据信息还在，不再是存储在连续的堆空间上，移动到“Metaspace”本地内存

方法区-永生代相关设置

1. -XX:PermSize = 64MB 最小尺寸，初始分配
2. -XX:MaxPermSize = 256MB 最大允许分配尺寸，按需分配
3. XX：+CMSClassUnloadingEnabled
4. -XX:+CMSPermGenSweepingEnabled 垃圾不回收

默认大小

1. -server 选项下默认MaxPermSize为64m
2. -client 选项下默认MaxPermSize为32m

##### 虚拟机栈（JVM Stack）

Java 虚拟机栈是线程私有，生命周期与线程相同。创建线程的时候就会创建一个java虚拟机栈。

虚拟机执行java程序的时候，每个方法都会创建一个栈帧，栈帧存放在java虚拟机栈中，通过压栈出栈的方式进行方法调用。

栈帧又分为：**局部变量表、操作数栈、动态连接、方法出口等**

局部变量存放在Java虚拟机栈的局部变量表中。

Java8中基本类型的局部变量的值存放在虚拟机栈的局部变量表中，如果是引用型的变量，则只存储对象的引用地址。

##### 本地方法栈（Native Stack）

其与虚拟机栈所发挥的作用非常相似，区别不过是虚拟机栈为虚拟机执行java方法服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

##### 程序计数器（PC Register）

程序计数器记录当前线程执行程序的位置，改变计算器的值来确定执行的下一条指令，比如循环、分支、方法跳转、异常处理，线程恢复都是依赖程序计数器来完成。

Java虚拟机多线程通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是线程私有的。

##### 直接内存

直接内存并不是虚拟机内存的一部分，也不是java虚拟机规范中定义的内存区域。

NIO（jdk1.4）

引入通道与缓冲区的IO方式，调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响堆内存的大小。

JVM参数设置

•-Xms 设置堆的最小空间大小。

• -Xmx 设置堆的最大空间大小。

• -Xmn:设置年轻代大小

• -XX:NewSize 设置新生代最小空间大小。

• -XX:MaxNewSize 设置新生代最大空间大小。

• -XX:PermSize 设置永久代最小空间大小。

• -XX:MaxPermSize 设置永久代最大空间大小。

• -Xss 设置每个线程的堆栈大小

• -XX:+UseParallelGC:选择垃圾收集器为并行收集器。此配置仅对年轻代 有效。即上述配置下,年轻代使用并发收集,而年老代仍旧使用串行收集。

• -XX:ParallelGCThreads=20:配置并行收集器的线程数,即:同时多少个 线程一起进行垃圾回收。此值最好配置与处理器数目相等。

#### 垃圾回收算法

##### 标记清除

标记清除算法将垃圾回收分为两个阶段： 标记阶段、清除阶段

标记阶段：

通过根节点（GC Roots），标记所有从根节点开始的对象，未被标记的对象就是未被引用的垃圾对象。

清除阶段：

清除未被标记的对象。

存活对象较多的情况较为高效 适用于年老代

缺点：

内存碎片，遇到一个比较大的对象时，即大于空闲表的每一块但小于两块的和，会提前触发垃圾回收

扫描整个空间两次 （第一次标记，第二次清除）

#### 复制算法

从根集合节点进行扫描，标记出所有的存活对象，并将这些存活的对象复制到一 块儿新的内存（图中下边的那一块儿内存）上去，之后将原来的那一块儿内存（图 中上边的那一块儿内存）全部回收掉

现在的商业虚拟机都采用这种收集算法来回收新生代

###### 使用场所：

存活对象较少的情况比较高效

扫描了整个空间一次

适用于年轻代：基本98%对象都是“朝生夕死”

###### 缺点：

需要一块空的内存空间

需要复制移动对象

#### 标记整理

老年代常见情况是大部分存活

标记-压缩算法

需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清 理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外 所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

#### 分代收集算法

目前虚拟机使用的回收算法

新生代存活率低，可以使用复制算法，老年代对象存活率高，可以使用标记清除或标记整理算法。

#### 垃圾回收机制

年轻代分为 Eden、survivor Eden:from:to == 8:1:1 。

###### JVM内存结构

* + 新产生的对象优先分配在Eden区（配置PretenureSizeThreshold大于该值直接进入老年代）
  + Eden区满了或者放不下了，这时候存活的对象会复制到from（from还放不下，进入老年代，然后回收Eden内存）
  + 对象会被复制很多次，每复制一次，对象的年龄+1，当对象被复制15次，就会进入年老代（MaxTenuringThreshold配置）
  + 年老代满了或者存放不下将要进入年老代的存活对象的时候，就会发生一次Full GC（耗时严重，需要减少）

Minor GC

对新生代进行回收，不会影响年老代 Minor GC非常频繁，使用速度快、效率高的算法

Full GC

Major GC，对整个堆回收，包括新生代和老年代，速度慢，尽可能减少Full GC的次数

#### 哨兵

哨兵是redis集群架构中非常重要的一个组件，解决主从复制出现故障需要认为干预的问题。

#### Function

1. Redis 集群监控

负责监控Redis master和slave进程是否正常工作

1. Redis 消息通知

如果某个Redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员

1. Redis 故障转移

Master Node挂了，自动转移到Slave Node上

1. Redis 配置中心

如果故障转移发生了，通知client客户端新的master地址

##### 哨兵高可用

原理：当主节点出现故障时，由Redis Sentinel自动完成故障发现和转移，并通知应用方，实现高可用性。

哨兵机制建立多个哨兵节点（进程），共同监控数据节点的运行状况。

同时哨兵节点之间也互相通信，交换对主从节点的监控状况。

每隔1秒哨兵向整个集群：Master主服务器+Slave从服务器+其他Sentinel（哨兵）进程，发送一次ping命令做一次心跳检测。

判断节点是否正常： 主观下线-客观下线

主观下线：一个哨兵节点判定主节点down掉

客观下线：半数哨兵节点 主观判断 主节点下线

原理：当哨兵节点判断出主节点客观下线，会在各个哨兵节点中发起投票机制Raft算法，最终被投为领导者的哨兵节点完成主从自动化切换的过程

Redis复制 –解决单点数据库问题

数据复制到多个副本部署到其他节点上，通过复制，实现Redis的高可用性，实现对数据的冗余备份，保证数据和服务的高度可靠性。

* 从数据库向主数据库发送sync命令
* 主数据库接受同步命令后，保存快照，创建也给RDB文件
* 当主数据库执行完保持快照后，会向从数据库发送RDB文件，而从数据库会接受并载入该文件。
* 主数据库将缓冲区的所有写命令发给从服务器执行
* 以上处理完，主数据库每执行一个写命令，都会被执行的写命令发送给从数据库。

**Redis 主从复制、哨兵和集群三个有什么区别**

主从模式：读写分离，备份，一个Master可以有多个Slaves

哨兵sentinel：监控，自动转移，哨兵发现主服务器挂了，就会从slave中重新选举一个主服务器。

集群：为了解决单机redis容量有限的问题，将数据按一定规则分配到多台机器，内存/QPS不受限于单机，可受益于分布式集群高拓展性

Redis高并发和快速原因

1. Redis是基于内存的，内存读写速度非常快
2. Redis是单线程，省去上下文切换线程的时间
3. Redis采用多路复用技术，可以处理并发连接，非阻塞IO内部实现采用epoll，epoll+自己实现简单的事件框架。Epoll中的读、写、关闭、连接都转化成了事件，然后利用epoll的多路复用特性，不在io上浪费时间。

###### 为什么Redis单线程

官方答案

Redis基于内存操作，同时cpu不是redis的瓶颈，redis的瓶颈最有可能是机器内存的大小或者网络带宽，既然单线程容易实现而且不会成为瓶颈，顺理成章采用。

性能指标

Redis性能，普通笔记本能每秒轻松几十万请求

详细原因：

不需要各种锁性能消耗

单线程多进程集群方案

CPU消耗

###### Redis单线程优劣势

1. 单进程单线程劣势

**代码更清晰，处理逻辑更简单**

**不用考虑锁**

**不存在多进程多线程切换消耗cpu**

1. 单进程单线程弊端

**无法发挥多核cpu性能，可以通过单机多开redis完善**

**IO多路复用技术**

**Redis采用网络IO多路复用技术保证多连接的时候，系统的高吞吐量**

**多路 指多个socket连接**

**复用 指复用一个线程**

**Select poll epoll**

**单个线程高效处理多个连接请求（减少网络io时间消耗），redis在内存中操作数据速度非常快。**

###### Redis高并发快总结

Redis纯内存数据库，一般都是简单的存取操作，线程占用时间很多，时间花费主要集中在io上，所以读取速度快。

Redis使用非阻塞io，io多路复用，使用单线程轮询描述符，将数据库的开关读写都转换成事件，减少线程切换上下文的竞争。

Redis采用单线程的模型，保证每个操作的原子性，减少了线程切换和竞争

数据结构，hash结构读取快， 还有特殊数据结构，压缩表对短数据压缩保存；跳表加快有序数据结构读取的速度。

Redis采用自己实现的事件分离器，效率高，内部采用非阻塞的执行方式，吞吐能力较大。

数据同步一致性解决方案

半同步复制，等主从同步完成后，等主库上的写请求再返回，这就是常说的“半同步复制”。

MySQL半同步复制

异步复制

MySQL默认复制是异步的，主库执行完客户端提交的事务后立即将结果返回给客户端，不关心从库是否接受并处理，这样会有一个问题，主如果crash掉了，此时主已经提交的事务可能并没有传到从库上。

半同步复制

介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接受并写到relay log中才返回给客户端。相对于异步复制，半同步复制提高了数据的安全性，同时他也造成了一定程度的延迟，这个延迟最少是一个TCP/IP往返时间，所以半同步复制最好在低延时的网络中使用。

半同步复制原理

事务在主库写完binlog后需要返回一个已接受，才放回给客户端mysql

确保事务提交后binlog至少传输到一个从库

不保证从库应用完成这个事务的binlog

性能有一定的降低

网络异常或库宕机，卡主库，直到超时或从库恢复

优点：

利用数据库原生功能，简单

缺点：

主屏写请求时延会增长，吞吐量会降低

数据库中间件

流程：

**所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库**

**记录所有路由到写库的key，在主从同步时间窗口内，如果有读请求访问到中间件，此时有可能从库还是旧版本，把key读请求到主库**

**在主从同步时间过完后，对应key读请求继续路由到从库**

**相关中间件有：**

**Canal：阿里旗下 纯java开发，基于数据库增量日志解析，**

**Otter：阿里开源的一个分布式数据库同步系统，，实时同步数据到本机房或跨机房**

**两者区别在于：**

**Otter：目前嵌入式依赖cannal，部署同一个jvm，不产生relay log**

**Otter：目前允许自定义同步逻辑，解决各类需求。**

**优点： 能保证绝对一致**

**缺点：数据库中间件的成本较高**

缓存记录写key法

写流程

如果key发生写操作，保存在cache里，同时设置经验主从同步时间的cache超时时间

然后修改主数据库

1）先到缓存里查看，对应 key 有没有相关数据

2）有相关数据，说明缓存命中，这个 key 刚发生过写操作，此时需要将请求路 由到主库读最新的数据。

3）如果缓存没有命中，说明这个 key 上近期没有发生过写操作，此时将请求路 由到从库，继续读写分离。

优点

数据库中间件，成本低

缺点

保证一致性，引入cache 读写数据库多了缓冲操作

###### 缓存和数据库一致性解决方案

采用延时双删策略

写库前后进行redis.del(key)操作，并且设置合理的超时时间

* 1. 先删除缓存
  2. 写数据库
  3. 休眠500ms
  4. 再次删除缓存

500ms怎么确定，具体休眠多久

评估项目的读数据业务逻辑耗时，确保读请求结束，写请求可以删除读请求造成的缓冲脏数据。

考虑redis和数据库主从同步耗时，写数据的休眠时间：在读数据业务逻辑耗时基础上，加几百ms即可。

设置缓存过期时间

理论上来说，缓存设置过期时间，保证最终一致性的解决方案。所有写操作以数据库为准，只要到达缓存过期时间，后面的读请求自然会从数据库中读取新值然后回填到缓存。

结合双删策略+缓存超时设置。最差的情况就是超时数据不一致，又增加写请求耗时

方案二：异步更新缓存

整体思路

**MySQL binlog 增量订阅消费+消息队列+增量数据更新到redis**

**读redis 热数据基本都在redis**

**写mysql 增删改查操作mysql**

**更新redis数据 mysql数据操作binlog更新redis**

Redis更新

数据操作主要分两大块：

**全量 一次写入redis**

**增量 实时更新 up in del**

读取binlog后分析，利用消息队列，推送更新各台redis缓存数据。

**这样一旦 MySQL 中产生了新的写入、更新、删除等操作，就可以把 binlog 相关 的消息推送至 Redis，Redis 再根据 binlog 中的记录，对 Redis 进行更新。 其实这种机制，很类似 MySQL 的主从备份机制，因为 MySQL 的主备也是通过 binlog 来实现的数据一致性。 这里可以结合使用 canal(阿里的一款开源框架)，通过该框架可以对 MySQL 的 binlog 进行订阅，而 canal 正是模仿了 mysql 的 slave 数据库的备份请求，使 得 Redis 的数据更新达到了相同的效果。 当然，这里的消息推送工具你也可以采用别的第三方：kafka、rabbitMQ 等来实 现推送更新 Redis**