背景

一、缓存

1. 为什么要用缓存

- **高性能**：对于很多几乎不变的数据(例如商品介绍，参数)，每次使用都要通过读取数据库来访问太慢(通过**磁盘**访问)，使用缓存就快很多(通过**内存**访问)
- **高并发**：MySQL的QPS在1w左右，Redis的QPS在10w以上，通过Redis集群可以更高。将数据库部分数据移到缓存中去，减少对数据库的访问，提高并发量。

> QPS（Query Per Second）：服务器每秒可以执行的查询次数

2. 本地缓存方案

- JDK自带的`HashMap`和`ConcurrentHashMap`，只提供了缓存机制，没有提供过期时间之类的功能
- Spring可以通过注解开启缓存，不易管理，容易**内存溢出**或者**缓存穿透**

3. 分布式缓存方案

- 将缓存部署在其他服务器上，可以多个服务器**共享**缓存
- 本地缓存受**性能**，**容量**限制严重
- 但使用分布式缓存需要引入**额外的服务**，比如Redis或者Memcached

二、Redis

1. 简介

- C语言开发的NoSQL(非关系型)数据库
- 与传统数据库不同，Redis的数据存在内存中

redis为什么那么快

- MySQL是将索引存在内存，硬盘中保存具体的值，redis完全基于**内存**
- **单进程单线程**，避免了不必要的上下文切换，多进程切换，不用考虑加锁解锁，也不会出现死锁
- **数据结构**简单，而且经过专门的设计，操作也简单
- **IO多路复用模型**，而不是阻塞IO

2. 单线程模型

- Redis内部使用**文件事件处理器**（file event handler），这个处理器是单线程的，所以Redis是单线程的

> 文件事件处理器内部包含四部分
>
> - 多个socket
> - IO多路复用程序
> - 文件事件分派器
> - 事件处理器

- Redis通过IO多路复用机制同时监听多个socket
- 多个socket可能对应不同的操作，每个操作对应不同的**文件事件**

- **IO多路复用程序**将socket产生的事件放入**队列**中排队，再由**分派器**将**事件**分派给对应的**事件处理器**处理

3.Redis与Memcached

- 二者都是基于**内存**的数据库
- Redis支持**更复杂的数据结构**和操作，可以用于更多的场景
- Redis支持**持久化**，Memcached只能将数据存在内存中
- Memcached原生不支持分布式**集群**，需要依靠客户端来实现往集群中分片写入数据，Redis原生支持Cluster
- Memcached支持**多核**，Redis只支持**单核。**数据量足够大时Memcached性能优于Redis

Redis的使用

1. 5种数据类型和使用场景

1.1 String

1）单值缓存

SET GET

2）对象缓存

- 使用json格式
- 使用MSET MGET 批量设置和获取值

3）分布式锁

- 使用 SETNX(set if not exists)来获取锁
- 通过expire设置过期时间

> 防止忘记释放锁，或者获取锁的服务突然故障导致无法主动释放锁

- 或者手动使用del释放锁

存在的问题

- 如果加锁和释放锁之间的代码执行时间过长，服务A还未执行完临界区代码，锁就过期了

> 避免用于较长时间的任务，如果出现这种情况

- 在setnx后，expire前，加锁的服务崩溃，锁永远无法释放

> set 可以加参数，让setnx和expire合成一条指令使用

4）计数器

- INCR DECR 原子加减

5）共享用户Session

3.2 Hash

- 每一个key对应的value都是一个哈希表
- 便于归类，但是哈希表中的项不能设置过期时间，只有key可以设置过期时间
- 相比于String存储对象更快，更省空间

1）对象缓存

HSET HGET HMSET HMGET

应用场景：购物车

- key为**用户id** value中的哈希表保存**商品id**和**商品数量**
- HINCRBY 来增加和删除商品
- HGETALL获取购物车中的全部信息

3.3 List

- 双向链表

1）异步队列

- LPUSH BRPOP： LPUSH生产消息，左边插入，BRPOP监听是否有元素入队，没有的话就阻塞，有的话就右侧消费消息

> 如果要实现一次生产，多个人 消费，可以使用**订阅模式** pub/sub
>
> **缺点**：如果消费者下线，生产者生产的消息会丢失
>
> **解决**：使用专门的消息队列，RabbitMQ，RocketMQ等

2）分页查询

- `LRANGE start end` 查询 [start, end]的信息
- 微博关注了很多人，每个关注的人发微博，按时间顺序push到list中
- 查看最新消息，执行`LRANGE 0 5`，如果下拉，则不断执行分页查询

3.4 Set

- 无序集合，每一个key对应的value都是一个无序集合

1）抽奖

- `SADD key userId` 点击抽奖，将抽奖的用户id加入集合
- `SMEMBERS key` 查看所有参与抽奖的用户
- `SRANDMEMBER key [10]` 从中抽取十个人

> SPOP key [count] 是随机删除，适合于先抽三等奖，再抽二等奖一等奖的情景

2）微信微博关注模型

- 为每个用户的关注的人维护一个set
- 通过并集，交集，差集，`SISMEMEBER`可以很方便的得到，共同关注之类的信息

3.5 ZSet

- 相比于set，为每个set中的每个member添加了个权重`score`，用来排序

1）延时队列

- `ZADD zset score key` 用消息内容作为key，时间戳作为score，调用`ZADD`生产消息
- 消费者通过`ZRANGEBYSCORE`来消费信息

4. 常用操作

- `keys *` 根据一定规则获取相应的key

> redis单线程，keys会导致线程阻塞一段时间，直到指令执行完毕才
>
> 线上服务端最好使用scan，不会阻塞，但是有可能获得重复key，在客户端去重即可

2. 并发竞争key

- **描述**：多个系统竞争一个key，最后执行的顺序和我们期望的顺序不同（例如网络抖动，下单，支付，退款三个顺序错了）
- **解决**：分布式锁（redis和zookeeper都可以，但是出于稳定性考虑使用zookeeper）
- **过程**：写入缓存的数据都是从数据库查出来的，查出的数据最后也会写回数据库。在数据库加入时间戳，写之前，判断下当前这个value

3. 双写一致性

- 只要用到缓存，就会有双写
- 一般来说允许缓存和数据库稍有不同
- 如果严格保持一致，只能将**读写请求串行化**，串到一个**内存队列**中

> 系统吞吐量大幅降低，并发高了还容易阻塞

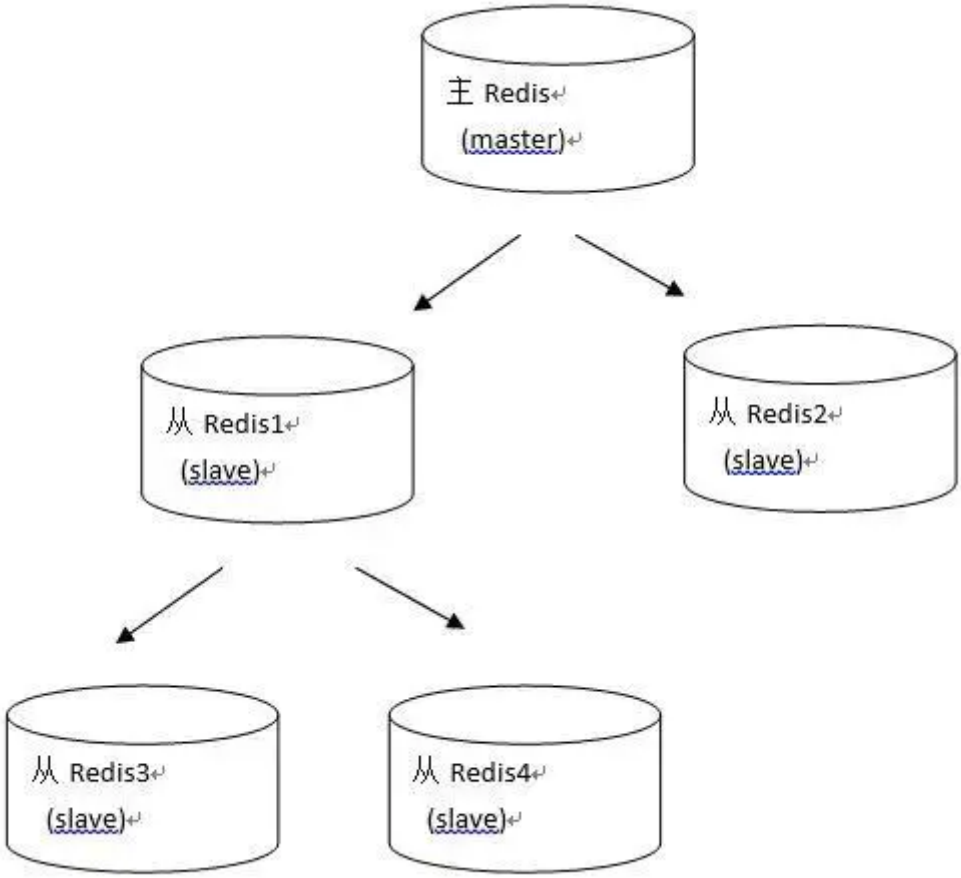
Redis保证可靠性的各种机制

一、使用前

集群高可用如何保证

- **主从复制 + 哨兵机制** 注重**高可用**，master宕机时，自动将从节点升级为master继续提供服务
- **Redis Cluster** 注重扩展性，单个redis内存不足时，使用cluster分片存储

1. 主从复制

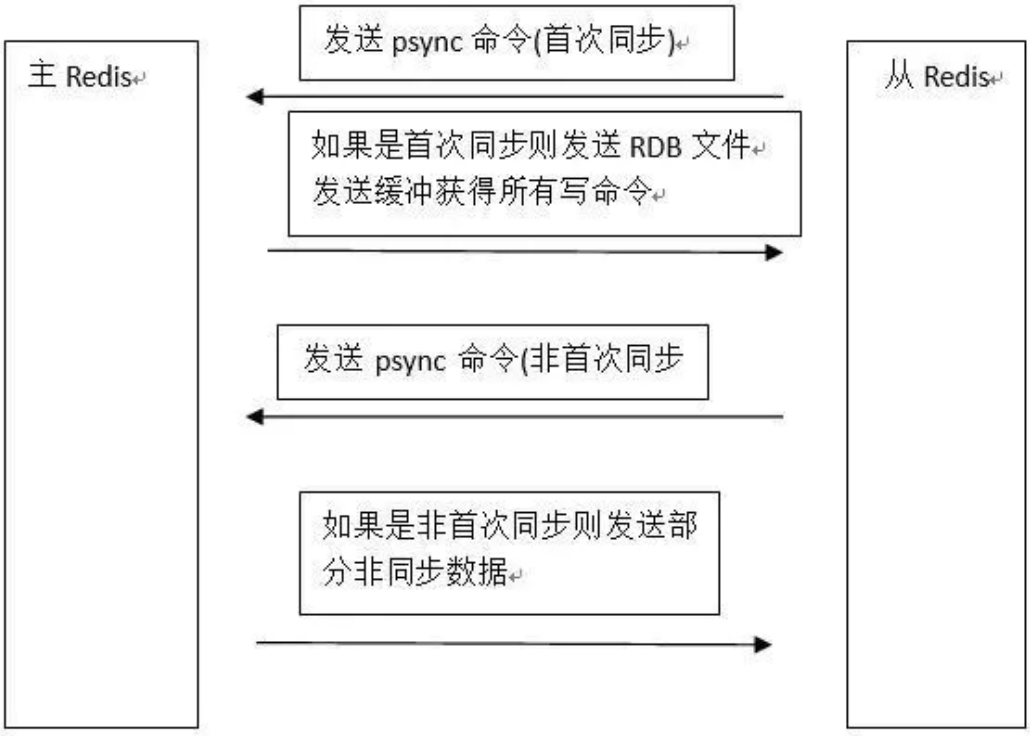


> 搞主从节点的目的：
>
> - 数据备份：在多个节点之间备份，更安全
> - 故障恢复：主节点出现故障，从节点继续提供服务
> - **负载均衡**：**读写分离**，主节点提供写服务，从节点提供读服务，分发大量请求（单个redis节点既要写，又要读，容易到瓶颈）

- 主从复制是为了让**从节点**备份**主节点**的数据
- 主从复制不会阻塞master，同步数据时，master可以继续处理客户端请求

1.1 过程

- 最初的版本是**全部复制**，每次从节点重新启动，都需要从主节点复制全部数据



- 现在的版本是**部分复制**，只有初次启动才需要全部复制，后续从节点重新启动只需要复制一部分
- 从redis首次启动，向主redis发送部分同步请求psync，psync中有**offset**和一个**runid**（机器表示，用来记录从redis上次同步的主redis的id）。
- 如果runid验证失败（说明是第一次同步）则发送RDB文件，如果runid验证成功，根据**offset**同步部分数据（同步AOF日志即可）

1.2 存在的问题

- 没法对master动态选举（如果master挂了，无法从从节点中选出一个master）

2. 哨兵机制（Sentinel)

2.1 作用

- **集群监控**：监控Redis集群中的主节点的运行状态
- **消息通知**：如果有主节点有故障，则通知管理员
- **故障转移**：如果有master节点失效，哨兵会将其中一个**从节点**升级为**主节点**，然后让其他**从节点**都从**新的主节点**复制数据。并且通知客户端新的主节点的地址

2.2 工作原理

- 正常工作时，每个哨兵每秒向集群中的 主，从，哨兵，ping一次，以确定它们在线
- 如果一个节点超过预期时间还未回复，该哨兵将该节点标记为**主观下线**
- 如果Master节点被判断为主观下线，监控这个节点的所有哨兵都会ping它来确定他是否下线
- 有足够多的哨兵（配置文件设置）判断其为**主观下线**，则mastrt节点被标记为**客观下线。**然后就是从从节点中选举一个新的主节点
- 如果没有足够多的哨兵判断，则移除其主观下线状态，如果客观下线的节点回复正常，则会移除其客观下线状态。

3. Redis Cluster

- 多个Master节点（每个master节点还有从节点）
- 将数据划分为多个部分，存储在多台机器里

二、使用中

1. 过期策略

定期删除 + 惰性删除

- **定期删除**：默认每100ms抽取**一些**设置了过期时间的key，检查是否过期，过期则删除

> 为什么不是所有？因为数据可能很多

- **惰性删除**：如果你用一个设置了过期时间的key，会检查一下它是否过期，过期则删除

> 所以内存中有大量过期的key删除不掉，可能导致内存耗尽

2. 内存淘汰机制

MySQL中有2000w数据，redis中有20w数据，如何保证全部是热点数据

- **volatile-lru**：在设置了过期时间的key中，选择最近最久未使用的淘汰
- **allkeys-lru**：在所有key中，移除最近最久未使用的（最常用）
- **volatile-random**：随机淘汰
- **allkeys-random**：随机淘汰

3. 雪崩

3.1 什么是雪崩

- 比如秒杀商品，服务器会将热点数据放在缓存中，如果缓存中午12点刷新，有效时间12个小时，则晚上0点过期
- 0点时，秒杀商品相关的缓存都失效了。这个时候用户发送很多请求，本来缓存是可以抗住的，但是由于缓存失效，这些请求都落在了数据库上。数据库立刻崩溃，数据库尝试重启，重启后又有新的请求来，数据库再次崩溃。

3.2 解决方案

- 批量往缓存中存数据时，每个key的过期时间都加一个随机值即可，避免数据在同一时间大规模失效。
- 如果redis是集群部署，可以将热点数据部署在不同的redis库中。
- 缓存永不失效，如果数据有更新操作就直接更新缓存即可

4. 击穿

4.1什么是缓存击穿

- 类似于雪崩，不过是针对单个key
- 有一个热点key，不断地有高并发的访问，在这个key失效的一瞬间，请求击穿缓存，直接请求数据库，导致数据库崩溃

4.2 解决方案

- 热点数据永不过期

5. 穿透（穿过缓存透数据库）

5.1 什么是缓存穿透

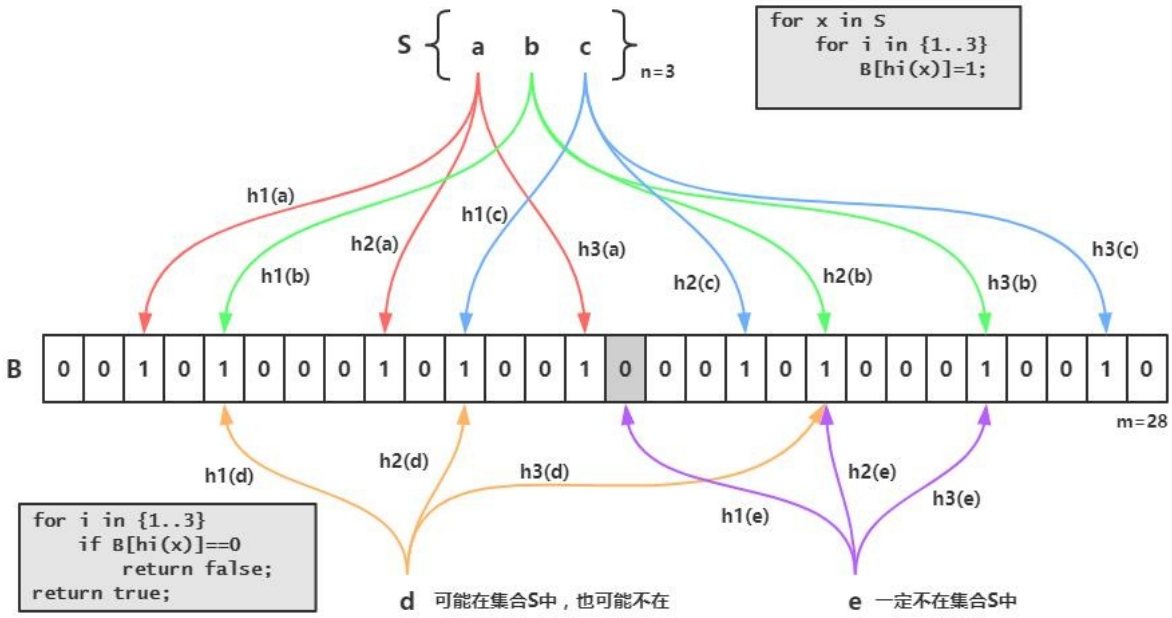
- 用户不断对缓存和数据库中都没有的数据发送请求，导致数据库压力过大，最后击垮数据库

> 比如数据库id由1自增上去，请求访问id为-1或者id极大不存在的数据。这样就可以绕开redis直接向数据库发送请求

5.2 解决方案

- **校验**：接口（controller）层增加校验，比如用户是否登陆，还有参数校验 （id <= 0 直接拦截）
- **缓存无效key**：缓存和数据库都没有的数据，可以将其key的value置为null放在缓存中，设置好过期时间，避免攻击者反复用同一个id暴力攻击
- **限流**：可以对单个ip或者单个用户每秒访问次数做出限制
- **布隆过滤器**

布隆过滤器



- **流程**：请求来首先经过**布隆过滤器**，判断请求的数据是否存在于布隆过滤器中，如果不存在，直接返回false，如果存在继续进行redis，数据库的访问
- **解释**：布隆过滤器可以高效判断一个给定key是否存在于海量数据中。
- **原理**：由k个哈希函数组成，将一个key映射到k位上，将这k位都置1。

> - 例如，对于key `a`来说，通过3个哈希函数，将其映射到第2，第4，第17位上
> - 然后有一个位数组，将数组中的第2,4,17位 置为1
> - 查询时，同样的将输入key映射成3个位，如果这3个位都为1，则说明该key**可能**存在，如果不是都为1，则说明该key**一定不存在**

弊端：

- 误判（如果布隆过滤器中都是黑名单，则可以建立一个易误判的白名单）
- 删除困难（counting 布隆过滤器）

三、使用后

1. 持久化机制

redis完全基于内存，一旦关机，内存中数据全部丢失，所以需要持久化

1.1 RDB 快照

- 默认的持久化方式，复制一份redis副本，然后备份到指定路径
- **原理**：**COW**(Copy On Write)机制，**fork**函数

> fork：redis通过创建子进程来实现RDB快照，父进程继续处理用户请求
>
> COW：子进程创建后，父子进程共享数据段。父进程要修改的数据是数据段的一个副本，子进程在备份的数据是不会被修改的。

- **优点**：对redis性能影响小，因为fork函数。数据恢复时比AOF快。
- **缺点**：间隔时间长（默认5分钟），一旦关机最多丢失近5分钟的数据。如果文件很大，fork子进程很耗时和耗CPU。而且备份的数据比较滞后

1.2 AOF 追加文件

- 默认关闭需要手动打开
- Redis中发生了数据的修改，就将该修改命令写入AOF文件，AOF文件的保存位置和RDB副本一样，可以由用户指定
- 可以手动设定写入AOF的sync属性：每次修改（严重影响性能），或者每秒一次（几乎不影响性能），由系统决定何时写入。
- **优点**：数据更完整，如果机器突然断电，而sync属性是每秒写AOF一次，则只丢失1秒内的数据。适合**误删除紧急恢复。**
- **缺点**：对于相同数据集，AOF文件体积大于RDB，数据恢复也慢。

1.3 使用策略

- RDB是镜像全量持久化，AOF是增量持久化。
- RDB适合做冷备份，AOF适合做热备份
- 所以RDB一般和AOF配合使用。Redis重启时，先使用RDB构建内存，然后用AOF重放**近期**操作指令来使内存恢复到之前的状态。**混合持久化**

> AOF只是RDB之后的这一段时间的操作日志，大小也会小很多