Redis

Redis 就是一个使用 C 语言开发的数据库，不过与传统数据库不同的是 **Redis 的数据是存在内存中的**，也就是它是内存数据库，所以读写速度非常快，也支持**持久化**

Redis 被广泛应用于**缓存**方向，也经常用来做**分布式锁**，甚至是**消息队列**

为什么快

- 绝大部分请求是纯粹的内存操作（非常快速）
- 数据处理阶段采用单线程,避免了不必要的上下文切换和竞争条件
- 非阻塞IO - IO多路复用，Redis采用epoll做为I/O多路复用技术的实现

分布式缓存

分布式缓存主要解决的是单机缓存的容量受服务器限制并且无法保存通用的问题。使用的比较多的主要是 **Memcached** 和 **Redis。**不过，现在基本没有看过还有项目使用 **Memcached** 来做缓存，都是直接用 **Redis**

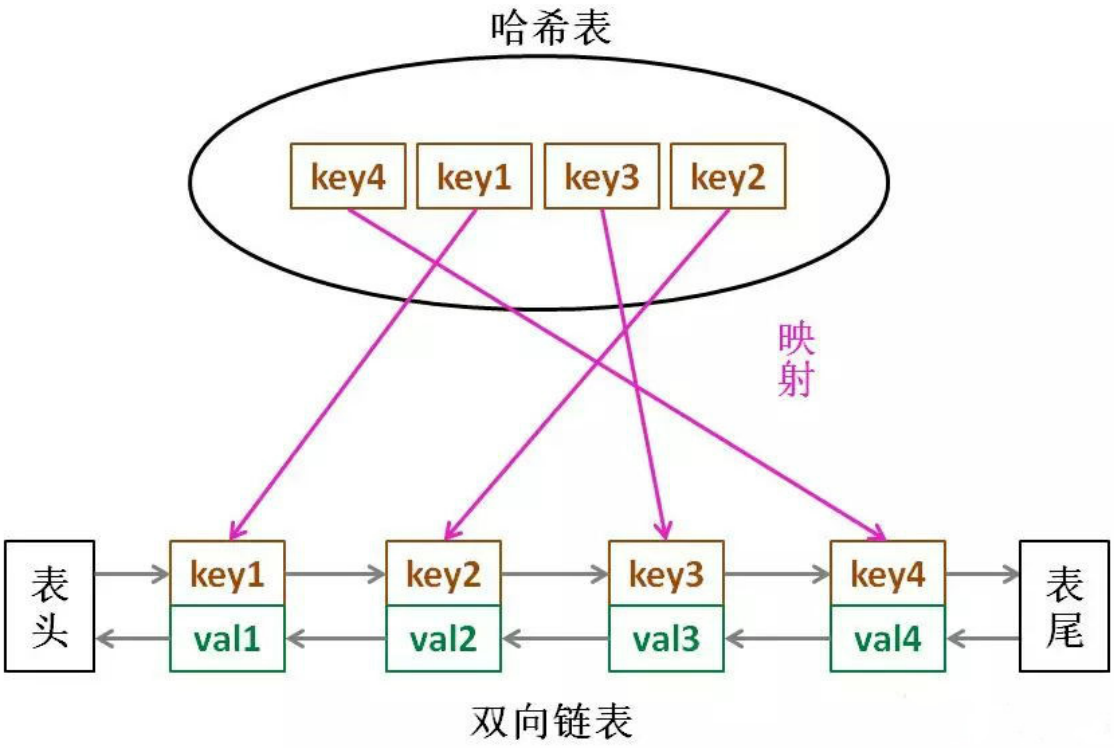
缓存主要是为了提升用户体验以及应对更多的用户

- 高性能；不用从硬盘中读取，直接操作内存，所以速度相当快
- 高并发；一般像 MySQL 这类的数据库的 QPS 大概都在 1w 左右（4 核 8g），但是使用 Redis 缓存之后很容易达到 10w+，甚至最高能达到 30w+（就单机 redis 的情况，redis 集群的话会更高）。

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

LRU算法自实现基本逻辑

LRU：最近最少使用
基本思路：获取刷新，设置刷新，列满移除
put 和 get 方法的时间复杂度为 *O*(1)，我们可以总结出 cache 这个数据结构必要的条件：**查找快，插入快，删除快，有顺序之分**
核心数据结构就是哈希链表，**双向链表和哈希表的结合体。**



借助哈希表赋予了链表快速查找的特性;可以快速查找某个 key 是否存在缓存（链表）中，同时可以快速删除、添加节点

常见数据结构

- string；string 数据结构是简单的 key-value 类型，场景：一般常用在需要计数的场景，比如用户的访问次数、热点文章的点赞转发数量、分布式锁等等
- list；即链表（双向），场景：发布与订阅或者说消息队列、慢查询
- hash；类似于HashMap(数组+链表)，场景：适合存储对象
- set；类似于HashSet，场景：不能重复的数据交集
- sorted set；和 set 相比，sorted set 增加了一个权重参数 score，场景：各种礼物排行榜

Redis线程模型

redis单线程

redis分客户端和服务端，一次完整的redis请求事件有多个阶段（客户端到服务器的网络连接-->redis读写事件发生-->redis服务端的数据处理（单线程）-->数据返回）。平时所说的**redis单线程模型，本质上指的是服务端的数据处理阶段**，不牵扯网络连接和数据返回，这是理解redis单线程的第一步

I/O多路复用

I/O指网络I/O

多路指的是多个TCP连接（Socket或Channel）

复用指的是复用一个或者多个线程

> BIO -> NIO：

> - 网络IO都是通过Socket实现，Server在某一个端口持续监听，客户端通过Socket（IP+Port）与服务器建立连接（ServerSocket.accept），成功建立连接之后，就可以使用Socket中封装的InputStream和OutputStream进行IO交互了。针对每个客户端，Server都会创建一个新线程专门用于处理。
> - 默认情况下，网络IO是阻塞模式，即服务器线程在数据到来之前处于【阻塞】状态，等到数据到达，会自动唤醒服务器线程，着手进行处理。阻塞模式下，一个线程只能处理一个流的IO事件
> - 为了提升服务器线程处理效率，有以下三种思路
>   - **非阻塞[忙轮询]**:采用死循环方式轮询每一个流，如果有IO事件就处理，这样一个线程可以处理多个流，但效率不高，容易导致CPU空转
>   - **Select代理(无差别轮询)**:可以观察多个流的IO事件，如果所有流都没有IO事件，则将线程进入阻塞状态，如果有一个或多个发生了IO事件，则唤醒线程去处理。但是会遍历所有的流，找出流需要处理的流。如果流个数为N，则时间复杂度为O(N)
>   - **Epoll代理**：Select代理有一个缺点，线程在被唤醒后轮询所有的Stream，会存在无效操作。Epoll哪个流发生了I/O事件会通知处理线程，对流的操作都是有意义的，复杂度降低到了O(1)

Redis内部实现采用epoll，采用了epoll+自己实现的简单的事件框架

redis多线程

Redis 4.0 增加的多线程主要是针对一些大键值对的删除操作的命令，使用这些命令就会使用主处理之外的其他线程来“异步处理”

大体上来说，**Redis 6.0 之前主要还是单线程处理**

Redis6.0 引入多线程主要是为了提高网络 IO 读写性能，但是默认是禁用的，只使用主线程

Redis如何判断数据是否过期

Redis通过过期字典来保存数据过期的时间。过期字典存储在redisDb这个结构里

```
typedef struct redisDb {
    ...
    dict *dict;     //数据库键空间,保存着数据库中所有键值对
    dict *expires   // 过期字典,保存着键的过期时间
    ...
} redisDb;
```

过期的数据的删除策略

redis 哨兵模式 vs 集群模式

redis部署的发展演变过程 https://www.cnblogs.com/zhonglongbo/p/13128955.html

- 单机模式，
- 主从复制（读写主节点，读从节点，主到从单向同步复制），高性能
- 哨兵模式，解决主从复制的主节点宕机故障；高可用，但是难支持在线扩容
  - 增加了哨兵集群，哨兵集群不保存数据，客户端通过哨兵集群访问redis数据，解决故障转移；还有主从存活检测，主从运行情况检测，主从切换

    - 在主从基础上，哨兵实现自动化的故障恢复
- 集群模式
    - 集群的键空间被分割为16384个slots（即hash槽），通过hash的方式将数据分到不同的分片上的
    - 分片对应也是主从复制模式

Redis分布式锁，为什么使用setnx，会有什么问题?

最常见的还是redis分布式锁，更推荐使用Zookeeper提供的分布式锁

zk分布式锁

Apache Curator是一个比较完善的Zookeeper客户端框架

- 封装ZooKeeper client与ZooKeeper server之间的连接处理
- 提供了一套Fluent风格的操作API
- 提供ZooKeeper各种应用场景(比如：**分布式锁服务、集群领导选举、共享计数器、缓存机制、分布式队列**等)的抽象封装

https://www.cnblogs.com/erbing/p/9799098.html

https://zhuanlan.zhihu.com/p/111354065

https://www.cnblogs.com/haoxinyue/p/6561896.html（blog很好）

https://zhuanlan.zhihu.com/p/150127393（偏向实践）

Curator提供了InterProcessMutex类来帮助我们实现分布式锁，其内部就是使用的**EPHEMERAL_SEQUENTIAL**类型节点

> 在 ZooKeeper 中，节点类型可以分为持久节点（PERSISTENT ）、临时节点（EPHEMERAL），以及时序节点（SEQUENTIAL ），具体在节点创建过程中，一般是组合使用，可以生成以下 4 种节点类型。不同的组合可以应用到不同的业务场景中
>
> - 持久化节点
> - 临时节点
> - 持久化时序节点
> - 临时时序节点

acquire()方法，会在给定的路径下面创建临时时序节点的时序节点。然后它会和父节点下面的其他节点比较时序。如果客户端创建的临时时序节点的数字后缀最小的话，则获得该锁，函数成功返回。如果没有获得到，即，创建的临时节点数字后缀不是最小的，则启动一个watch监听上一个（排在前面一个的节点）。主线程使用object.wait()进行等待，等待watch触发的线程notifyAll()，一旦上一个节点有事件产生马上再次出发时序最小节点的判断。

release()方法就是释放锁，内部实现就是删除创建的EPHEMERAL_SEQUENTIAL节点

redis实现分布式锁

- setnx
- redisson
- redLock

在 Redis 里，所谓 SETNX，是「**SET** if **N**ot e**X**ists」的缩写，也就是只有不存在的时候才设置，并非单指setnx这个命令，这个命令在redis2.6.12后不建议使用

> SET key value [EX seconds|PX milliseconds] [NX|XX] [KEEPTTL]
>
> - `EX` *seconds* -- Set the specified expire time, in seconds.
> - `PX` *milliseconds* -- Set the specified expire time, in milliseconds
> - `NX` -- Only set the key if it does not already exist.
> - `XX` -- Only set the key if it already exist.
> - `KEEPTTL` -- Retain the time to live associated with the key.

对应的工具方法set或者setIfAbsent

setnx问题：不可重入；进程A拿到后崩了，死锁；超时问题等等

超时，守护线程watch dog 动态续期，这个也是Redisson的解决方式

Redisson是**java的redis客户端之一**，提供了一些api方便操作redis，包括**RedissonLock**这个类，源码中**加锁/释放锁**操作都是用**lua**脚本完成的，封装的非常完善，开箱即用。

redLock是redis官方提出的一种分布式锁的**算法**，Redisson中，就实现了redLock版本的锁，避免了redis异步复制造成的锁丢失

> 对于可靠性，多数文章会告诉你zk锁比Redis锁更可靠，毕竟zk是专业做分布式协调服务的，实际情况是：无论是Redis锁还是zk锁，亦或是基于etcd, consul等实现的分布式锁，都回避不了同一个问题：**持有锁的进程A在处理业务的过程中，同一把锁有可能在A不知情的情况下被进程B拿走**
>
> 分布式系统有几个无法回避的问题，我把它们**称为NPC问题**，N主要指Network delay网络延迟，P是指Process pause进程暂停，C指代Clock drift时钟漂移**
>
> 以进程暂停为例，进程A刚刚拿到分布式锁，然后因为GC等原因进程被冻结暂停，过去1分钟之后，分布式锁超时了，进程B顺利抢到同一把锁，接下来进程A的GC结束了。好了，现在俩进程都觉得自己是锁的唯一持有人，然后开始快乐的编辑数据。
>
> 解决这个问题最简单的办法是**使用fencing token，也就是获取锁的时候，同步创建一个自增的全局唯一id**，同时数据库中会记录最后一次碰见的全局唯一id。这样，当修改数据时，数据库原子性的比较进程提供的全局唯一id是不是最大的，如果不是，说明该进程持有的锁已经过期了。
>
> 如果是使用zk的话，可以使用zxid作为全局唯一id
>
> https://www.zhihu.com/question/452803310/answer/1816290814
>
> https://mp.weixin.qq.com/s/hOPT41HIAGE8iZ5jLlREmg

redis有哪些基本数据类型，用到了哪些数据结构，怎么理解跳表

跳表可以看出类似二叉树的一种结构，什么时候新增一层?

mongodb是真正的内存数据库，与redis一样?

不一样，mongoDB是关系映射型数据库，真正的数据还是存储在文件中

10亿条记录，2亿记录同时过期，会怎样

- 容易造成客户端等待超时的现象；根据过期策略作答，例如：https://blog.csdn.net/dam454450872/article/details/102549282

expire命令，参数Redis中的键，设置过期时间

Redis中有一个保存过期时间的数据结构体，有一个键字典，还有一个过期字典保存了所有键的过期时间

过期键的判断： 根据键，找到过期字典中的过期时间，比对即可

过期键的删除策略：

- 惰性删除，在取出该键的时候对键进行过期检查，即只对当前处理的键做删除操作，不会在其他过期键上花费 CPU 时间（配置 lazyfree_lazy_expire（4.0版本引进），先逻辑再物理）
- 定期删除，默认每秒进行10次过期扫描；为避免循环过度，设置了扫描时间范围（1-25ms）
    - 从过期字典中随机取出20个键
    - 删除这20个键中过期的键
    - 如果过期键比例超过25%，重复步骤1和2