redis为什么这么快（端口号：6379）

- **内存存储**：Redis是使用内存存在，避免了磁盘IO上的开销。数据存储在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
- **单线程实现**（redis6.0以前）Redis使用单个线程处理请求，避免了多个线程之间线程切换和锁资源争用的开销。单线程是指一个线程处理所有网络请求
- **非阻塞IO**：Redis使用多路复用IO技术，将epoll作为I/O多路复用技术的实现，再加上Redis自身的事件处理模型将epoll中的连接、读写、关闭都转换为事件，不在网络I/O上浪费时间。
- **优化的数据结构**：Redis有诸多可以直接应用的优化数据结构的实现，应用层可以直接使用原生的数据结构提升性能

为什么要用Redis做缓存

- **从高并发的角度说**：直接操作缓存能够承受的请求远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。
- **从高性能上来说**：　用户第一次访问数据库中的某些数据。因为是从硬盘上读取的所以这个过程会比较慢。将该用户访问的数据存在缓存中，下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据。

Redis的常用场景

- **缓存**：缓存现在几乎是所有中大型网站都在用的必杀技，合理的利用缓存不仅能够提升网站访问速度，还能大大降低数据库的压力.Redis提供了键过期功能，也提供了灵活的键淘汰策略，所以，现在Redis用在缓存的场合比较多
- **排行榜**：很多网站都有排行榜应用的，如京东的月度销量榜单、商品按时间的上新排行榜等。Redis提供的有序集合数据类构能实现各种复杂的排行榜应用。
- **计数器**：什么是计数器，如电商网站商品的浏览量、视频网站视频的播放数等。为了保证数据实时效，每次浏览都得给+1，并发量高时如果每次都请求数据库操作无疑是种挑战和压力。Redis提供的incr命令来实现计数器功能，内存操作，性能非常好，非常适用于这些计数场景。
- **分布式锁**：在很多互联网公司中都使用了分布式技术，分布式技术带来的技术挑战是对同一个资源的并发访问，如全局ID、减库存、秒杀等场景，并发量不大的场景可以使用数据库的悲观锁、乐观锁来实现，但在并发量高的场合中，利用数据库锁来控制资源的并发访问是不太理想的，大大影响了数据库的性能。可以利用Redis的setnx功能来编写分布式的锁，如果设置返回1说明获取锁成功，否则获取锁失败，实际应用中要考虑的细节要更多。
- **社交网络**：点赞、踩、关注/被关注、共同好友等是社交网站的基本功能，社交网站的访问量通常来说比较大，而且传统的关系数据库类型不适合存储这种类型的数据，Redis提供的哈希、集合等数据结构能很方便的的实现这些功能。如在微博中的共同好友，通过Redis的set能够很方便得出。
- **最新列表**：Redis列表结构，LPUSH可以在列表头部插入-一个内容ID作为关键字，LTRIM可 用来限制列表的数量，这样列表永远为N个ID，无需查询最新的列表，直接根据ID去到对应的内容页即可。

Redis数据结构介绍

- **String**：字符串类型是 Redis 最基础的数据结构，键都是字符串类型，而且其他几种数据结构都是在字符串类型的基础上构建的。string 数据结构是简单的 key-value 类型。虽然 Redis 是用 C 语言写的，但是 Redis 并没有使用 C 的字符串表示，而是自己构建了一种 **简单动态字符串**（simple dynamic string，**SDS**）。相比于 C 的原生字符串，Redis 的 SDS 不光可以保存文本数据还可以保存二进制数据，并且获取字符串长度复杂度为 O(1)（C 字符串为 O(N)）,除此之外,Redis 的 SDS API 是安全的，不会造成缓冲区溢出。

string的内部编码

int：8 个字节的长整形
embstr：小于等于 39 个字节的字符串
raw：大于 39 个字节的字符串

stirng的常用命令

`set,get,strlen,exists,decr,incr,setex` 等等。

string的应用场景

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等等。

- **hash**: hash 是一个 string 类型的 field 和 value 的映射表，**特别适合用于存储对象**，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。 比如我们可以 hash 数据结构来存储用户信息，商品信息等等。

hash的内部编码

ziplist 压缩列表：当哈希类型元素个数和值小于配置值（默认 512 个和 64 字节）时会使用 ziplist 作为内部实现，使用更紧凑的结构实现多个元素的连续存储，在节省内存方面比 hashtable 更优秀。

hashtable 哈希表：当哈希类型无法满足 ziplist 的条件时会使用 hashtable 作为哈希的内部实现，因为此时 ziplist 的读写效率会下降，而 hashtable 的读写时间复杂度都为 O(1)。

hash的常用命令

`hset,hmset,hexists,hget,hgetall,hkeys,hvals` 等。

hash的应用场景

系统中对象数据的存储。缓存用户信息，每个用户属性使用一对 field-value，但只用一个键保存。

优点：简单直观，如果合理使用可以减少内存空间使用。

缺点：要控制哈希在 ziplist 和 hashtable 两种内部编码的转换，hashtable 会消耗更多内存。

- **list**:

list的内部编码

ziplist 压缩列表：跟哈希的 zipilist 相同，元素个数和大小小于配置值（默认 512 个和 64 字节）时使

用。

linkedlist 链表：当列表类型无法满足 ziplist 的条件时会使用linkedlist。

Redis 3.2 提供了 quicklist 内部编码，它是以一个 ziplist 为节点的 linkedlist，它结合了两者的优势，为列表类提供了一种更为优秀的内部编码实现。

list的常用命令

`rpush,lpop,lpush,rpop,lrange、llen` 等。

list的应用场景

发布与订阅或者说消息队列、慢查询。

lpush + lpop = 栈、lpush + rpop = 队列、lpush + ltrim = 优先集合、lpush + brpop = 消息队列。

- **set**:

set 类似于 Java 中的 `HashSet` 。Redis 中的 set 类型是一种无序集合，集合中的元素没有先后顺序。当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。比如：你可以将一个用户所有的关注人存在一个集合中，将其

所有粉丝存在一个集合。Redis 可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程。

set的内部编码

intset 整数集合：当集合中的元素个数小于配置值（默认 512 个时），使用 intset。
hashtable 哈希表：当集合类型无法满足 intset 条件时使用 hashtable。当某个元素不为整数时，也会使用 hashtable。

set的常用命令

`sadd,spop,smembers,sismember,scard,sinterstore,sunion` 等。

set的应用场景

需要存放的数据不能重复以及需要获取多个数据源交集和并集等场景。

sadd = 标签、spop/srandmember = 生成随机数，比如抽奖、sadd + sinter = 社交需求。

- **sorted set/zset**

和 set 相比，sorted set 增加了一个权重参数 score，使得集合中的元素能够按 score 进行有序排列，还可以通过 score 的范围来获取元素的列表。有点像是 Java 中 HashMap 和 TreeSet 的结合体。

sorted set的内部编码

ziplist 压缩列表：当有序集合元素个数和值小于配置值（默认128 个和 64 字节）时会使用 ziplist 作为内部实现。
skiplist 跳跃表：当 ziplist 不满足条件时使用，因为此时 ziplist 的读写效率会下降。

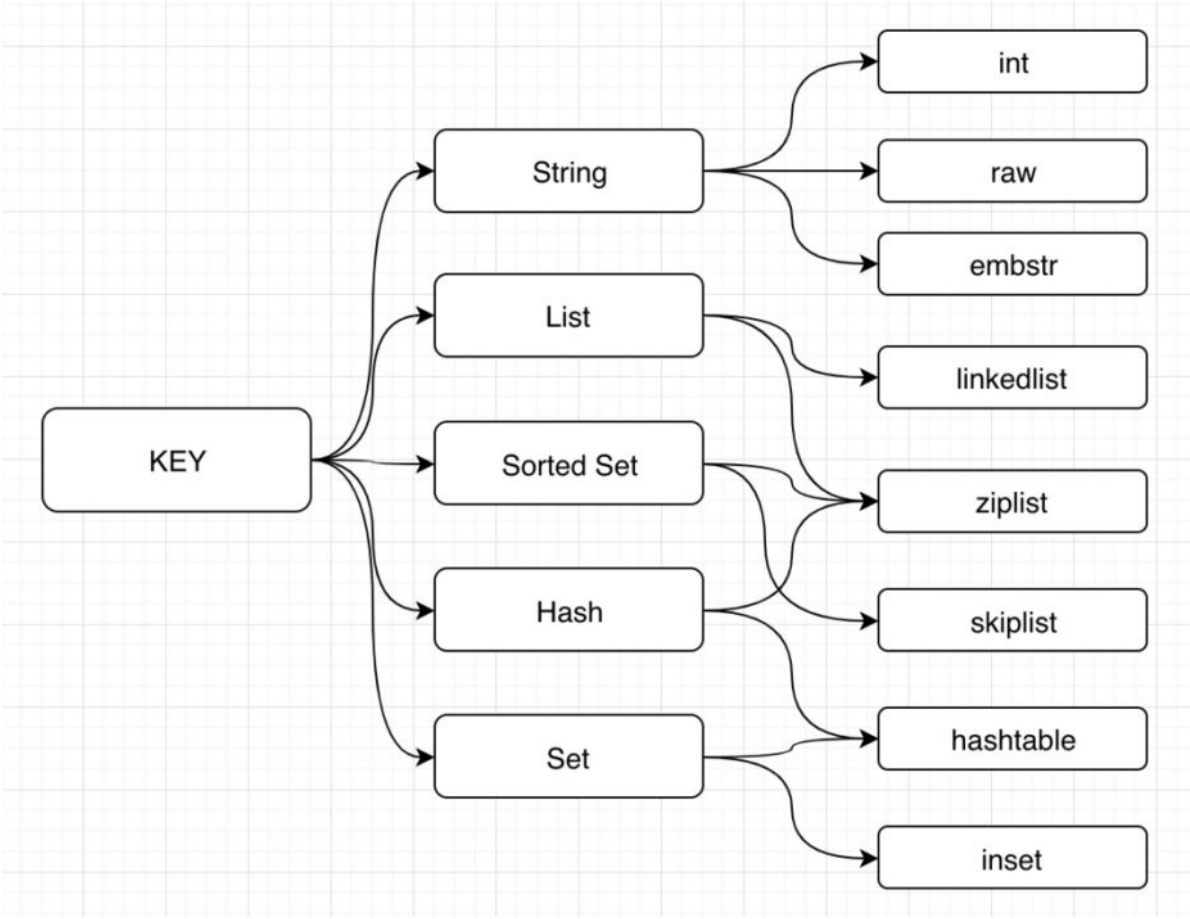
sorted set的常用命令

`zadd,zcard,zscore,zrange,zrevrange,zrem` 等。

sorted set的应用场景

需要对数据根据某个权重进行排序的场景。比如在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息。

有序集合的典型使用场景就是排行榜系统，例如用户上传了一个视频并获得了赞，可以使用 zadd 和zincrby。如果需要将用户从榜单删除，可以使用 zrem。如果要展示获取赞数最多的十个用户，可以使用 zrange。



五种常用的数据结构

- **String**: String是 最常用的一种 数据类型，普通的key- value存储都可以归为此类。其中Value既可以
  是数字也可以是字符串。使用场景:常规key-value缓存应用。常规计数:微博数，粉丝数。
- **Hash**: Hash是- -个键值(key => value)对集合。Redishash 是-一个string 类型的field和value的映射表，hash特别适合用于存储对象，并且可以像数据库中update-一个属性一样只修改某一项属性
  值。
- **Set**: Set是 -一个无序的天然去重的集合，即Key-Set。 此外还提供了交集、并集等-一系列直接操作集
  合的方法，对于求共同好友、共同关注什么的功能实现特别方便。
- **List**: List是 -一个有序可重复的集合，其遵循FIFO的原则，底层是依赖双向链表实现的，因此支持正
  向、反向双重查找。通过List,我们可以很方面的获得类似于最新回复这类的功能实现。
- **SortedSet**: 类似于java中的TreeSet，是Set的可排序版。此外还支持优先级排序，维护了一个score
  的参数来实现。适用于排行榜和带权重的消息队列等场景。

三种特殊的数据类型:

- Bitmap: 位图，Bitmap想象成- 一个以位为单位数组，数组中的每个单元只能存0或者1，数组的下标在Bitmap中叫做偏移量。使用Bitmap实现统计功能，更省空间。如果只需要统计数据的二值状态，例如商品有没有、用户在不在等，就可以使用Bitmap，因为它只用一个bit位就能表示0或1。
- Hyperloglog。 HyperLogLog 是一种用于统计基数的数据集合类型，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。每个HyperLogLog键只需要花费12 KB内存，就可以计算接近2^64个不同元素的基数。场景:统计网页的UV (即Unique Visitor,不重复访客，一个人访问某个网站多次，但是还是只计算为一次)。要注意，HyperLogLog 的统计规则是基于概率完成的，所以它给出的统计结果是有一定误差的，标准误算率是0.81%。

- Geospatial: 主要用于存储地理位置信息，并对存储的信息进行操作，适用场景如朋友的定位、附近的人、打车距离计算等。

持久化

为了能够重用Redis数据，或者防止系统故障，我们需要将Redis中的数据写入到磁盘空间中，即持久化。Redis提供了两种不同的持久化方法可以将数据存储在磁盘中，一种叫快照RDB，另一种叫只追加文件AOF。

- **RDB**：在指定的时间间隔内将内存中的数据集快照写入磁盘(Snapshot)，它恢复时是将快照文件直接读到内存里。

  优势:适合大规模的数据恢复;对数据完整性和一致性要求不高

  劣势:在一定间隔时间做一次备份， 所以如果Redis意外down掉的话，就会丢失最后- -次快照后的所有修改。

- **AOF**:
  以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，Redis启动之初会读取该文件重新构建数据，换言之，Redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作。AOF采用文件追加方式，文件会越来越大，为避免出现此种情况，新增了重写机制，当AOF文件的大小超过所设定的阈值时，Redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集.。
  优势:
  - 每修改同步: appendfsync always同步持久化，每次发生数据变更会被立即记录到磁盘，性能较差但数据完整性比较好
  - 每秒同步:
    appendfsync everysec 异步操作，每秒记录，如果一秒内宕机，有数据丢失
  - 不同步:1 appendfsync no 从不同步

  劣势
  - 相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb
  - aof运行效率要慢于rdb，每秒同步策略效率较好，不同步效率和rdb相同

如何选择合适的持久化方式

- 如果是数据不那么敏感，且可以从其他地方重新生成补回的，那么可以关闭持久化。
- 如果是数据比较重要，不想再从其他地方获取，且可以承受数分钟的数据丢失，比如缓存等，那么可以只使用RDB。
- 如果是用做内存数据库，要使用Redis的持久化，建议是RDB和AOF都开启，或者定期执行bgsave做快照备份，RDB方式更适合做数据的备份，AOF可以保证数据的不丢失。
  补充:Redis4.0对于持久化机制的优化
  Redis4.0相对与3.X版本其中一个比较大的变化是4.0添加了新的混合持久化方式。

Tips：Redis4.0 添加了新的混合持久化方式：前半段是RDB格式的全量数据后半段是AOF格式的增量数据

过期键的删除策略、淘汰策略

Redis过期键的删除策略
Redis的过期删除策略就是:惰性删除和定期删除两种策略配合使用。
惰性删除:惰性删除不会去主动删除数据，而是在访问数据的时候，再检查当前键值是否过期，如果过期则执行删除并返回null给客户端，如果没有过期则返回正常信息给客户端。它的优点是简单，不需要对过期的数据做额外的处理，只有在每次访问的时候才会检查键值是否过期，缺点是删除过期键不及时，造成了一定的空间浪费。.
定期删除:Redis会周期性的随机测试一批设置了过期时间的key并进行处理。测试到的已过期的key将被删除。
附:删除key常见的三种处理方式。

定时删除：在设置某个key的过期时间同时，我们创建一个定时器，让定时器在该过期时间到来时，立即执行对其
进行删除的操作。

- 优点:定时删除对内存是最友好的，能够保存内存的key一旦过期就能立即从内存中删除。
- 缺点:对CPU最不友好，在过期键比较多的时候，删除过期键会占用一-部分CPU时间，对服务器的响应
  时间和吞吐量造成影响。

惰性删除
设置该key过期时间后，我们不去管它，当需要该key时，我们在检查其是否过期，如果过期，我们就删掉它，反之返回该key。

- 优点:对CPU友好，我们只会在使用该键时才会进行过期检查，对于很多用不到的key不用浪费时间进行过期检查。
- 缺点:对内存不友好，如果一个键已经过期，但是一直没有使用，那么该键就会一直存在内存中，如果数据库中有很多这种使用不到的过期键，这些键便永远不会被删除，内存永远不会释放。从而造成内存泄漏。

定期删除
每隔一段时间，我们就对一些key进行检查，删除里面过期的key。

- **优点**:可以通过限制删除操作执行的时长和频率来减少删除操作对CPU的影响。另外定期删除，也能有效释放过期键占用的内存。
- **缺点**:难以确定删除操作执行的时长和频率。如果执行的太频繁，定期删除策略变得和定时删除策略一样，对CPU不友好。如果执行的太少，那又和惰性删除-样了，过期键占用的内存不会及时得到释

放。另外最重要的是，在获取某个键时，如果某个键的过期时间已经到了，但是还没执行定期删除，那么就会返回这个键的值，这是业务不能忍受的错误。

Redis内存淘汰策略

Redis是不断的删除一些过期数据，但是很多没有设置过期时间的数据也会越来越多，那么Redis内存不够用的时候是怎么处理的呢?答案就是淘汰策略。此类的当Redis的内存超过最大允许的内存之后，Redis会触发内存淘汰策略，删除-些不常用的数据，以保证Redis服务器的正常运行。

- volatile-lru：利用LRU算法移除设置过过期时间的key
- allkeys-lru： 在键空间，移除最近最少使用的key（最常用）
- volatile-random:从已设置过期时间的数据集(server.db[j.expires) 中任意选择数据淘汰
- allkeys-random: 从数据集(server.db[i].dict) 中任意选择数据淘汰

Redisv4.0后增加以下两种:

- volatile-lfu:从已设置过期时间的数据集(server.db[i].expires)中挑选最不经常使用的数据淘汰(LFU(LeastFrequentlyUsed)算法，也就是最频繁被访问的数据将来最有可能被访问到)
- allkeys-lfu: 当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的key。

redis缓存异常

- 缓存与数据库的数据不一致
- 缓存雪崩
- 缓存击穿
- 缓存穿透

如何保证缓存与数据库双写时的数据一致性?

延时双删:

- 先淘汰缓存
- 再写数据库
- 休眠1秒，再次淘汰缓存，这么做，可以将1s内所造成的缓存脏数据，再次删除。确保读请求结束，写请求可以删除读请求造成的缓存脏数据。自行评估自己的项目的读数据业务逻辑的耗时，写数据的休眠时间则在读数据业务逻辑的耗时基础上，加上几百ms即可

```
public void write(String key ,Object data){
    Redis.delKey(key);
    db.updateData(data);
    Thread.sleep(1000);
    Redis.delKey(key);
}
```

缓存击穿

缓存击穿跟缓存雪崩有点类似，缓存雪崩是大规模的key失效，而缓存击穿是某个热点的key失效，大并发集中对其进行请求，就会造成大量请求读缓存没读到数据，从而导致高并发访问数据库，引起数据库压力剧增。这种现象就叫做缓存击穿。

解决方案:

- 在缓存失效后，通过互斥锁或者队列来控制读数据写缓存的线程数量，比如某个key只允许一个线程查询数据和写缓存，其他线程等待。这种方式会阻塞其他的线程，此时系统的吞吐量会下降
- 热点数据缓存永远不过期。永不过期实际包含两层意思:
    - 物理不过期，针对热点key不设置过期时间

    - 逻辑过期，把过期时间存在key对应的value里， 如果发现要过期了，通过一个后台的异步线程进行缓存的构建

缓存穿透

缓存穿透是指用户请求的数据在缓存中不存在即没有命中，同时在数据库中也不存在，导致用户每次请求该数据都要去数据库中查询一遍。如果有恶意攻击者不断请求系统中不存在的数据，会导致短时间大量请求落在数据库上，造成数据库压力过大，甚至导致数据库承受不住而宕机崩溃。
解决方法:

- 将无效的key存放进Redis中:
  当出现Redis查不到数据，数据库也查不到数据的情况，我们就把这个key保存到Redis中，设置value="null"，并设置其过期时间极短，后面再出现查询这个key的请求的时候，直接返回null，就不需要再查询数据库了。但这种处理方式是有问题的，假如传进来的这个不存在的Key值每次都是随机的，那存进Redis也没有意义。
- 使用布隆过滤器:
  如果布隆过滤器判定某个key不存在布隆过滤器中，那么就一定不存在， 如果判定某个key存在，那么很大可能是存在(存在一定的误判率)。于是我们可以在缓存之前再加一个布隆过滤器，将数据库中的所有key都存储在布隆过滤器中，在查询Redis前先去布隆过滤器查询key是否存在，如果不存在就直接返回，不让其访问数据库，从而避免了对底层存储系统的查询压力。

缓存雪崩

如果缓在某一个时刻出现大规模的key失效，那么就会导致大量的请求打在了数据库上面，导致数据库压力巨大，如果在高并发的情况下，可能瞬间就会导致数据库宕机。这时候如果运维马上又重启数据库，马上又会有新的流量把数据库打死。这就是缓存雪崩。造成缓存雪崩的关键在于同一时间的大规模的key失效，主要有两种可能:第一种是Redis宕机，第二种可能就是采用了相同的过期时间。

造成缓存雪崩的关键在于同一时间的大规模的key失效，主要有两种可能:第-种是Redis宕机，第二种可能就是采用了相同的过期时间。

解决方案:
1、**事前:**

- 均匀过期:设置不同的过期时间，让缓存失效的时间尽量均匀，避免相同的过期时间导致缓存雪崩，造成大量数据库的访问。如把每个Key的失效时间都加个随机值，setRedis (Key, value,time + Math. random()*10000），保证数据不会在同一时间大面积失效。
- 分级缓存:第一级缓存失效的基础上，访问二级缓存，每一级缓存的失效时间都不同。
- 热点数据缓存永远不过期。永不过期实际包含两层意思:
    - 物理不过期，针对热点key不设置过期时间
    - 逻辑过期，把过期时间存在key对应的value里， 如果发现要过期了，通过一个后台的异步线程进行缓存的构建
- 保证Redis缓存的高可用，防止Redis宕机导致缓存雪崩的问题。可以使用主从+哨兵，Redis集群来避免Redis全盘崩溃的情况。

2、**事中:**

- 互斥锁:在缓存失效后，通过互斥锁或者队列来控制读数据写缓存的线程数量，比如某个key只允许一个线程查询数据和写缓存，其他线程等待。这种方式会阻塞其他的线程，此时系统的吞吐量会下降.
- 使用熔断机制，限流降级。当流量达到一.定的阈值，直接返回”系统拥挤"之类的提示，防止过多的请求打在数据库上将数据库击垮，至少能保证一部分用户是可以正常使用，其他用户多刷新几次也能得到结果。

3、**事后:**
开启Redis持久化机制，尽快恢复缓存数据，一旦重启，就能从磁盘上自动加载数据恢复内存中的数据。

缓存预热

缓存预热是指系统上线后，提前将相关的缓存数据加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题，用户直接查询事先被预热的缓存数据。如果不进行预热，那么Redis初始状态数据为空，系统上线初期，对于高并发的流量，都会访问到数据库中，对数据库造成流量的压力。
缓存预热解决方案:

- 数据量不大的时候，工程启动的时候进行加载缓存动作;
- 数据量大的时候，设置一个定时任务脚本，进行缓存的刷新;
- 数据量太大的时候，优先保证热点数据进行提前加载到缓存。

缓存降级

缓存降级是指缓存失效或缓存服务器挂掉的情况下，不去访问数据库，直接返回默认数据或访问服务的内存数据。降级一般是有损的操作，所以尽量减少降级对于业务的影响程度。在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅;从而梳理出哪些必须誓死保护，哪.些可降级;比如可以参考日志级别设置预案:

- 一般:比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级;
- 警告:有些服务在一段时间内成功率有波动(如在95~100%之间)，可以自动降级或人工降级，并发送告警;
- 错误:比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级;
- 严重错误:比如因为特殊原因数据错误了，此时需要紧急人工降级。

Redis为何使用单线程

核心意思就是，对于一个DB来说，CPU 通常不会是瓶颈，因为大多数请求不会是CPU密集型的，而是I/O密集型。具体到Redis的话，如果不考虑RDB/AOF等持久化方案，Redis是 完全的纯内存操作，执行速度是非常快的，因此这部分操作通常不会是性能瓶颈，Redis真正的性能瓶颈在于网络I/O，也就是客户端和服务端之间的网络传输延迟，因此Redis选择了单线程的I/O多路复用来实现它的核心网络模型。

Redis6.0为何引入多线程

Reids的网络I/O瓶颈已经越来越明显了

随着互联网的飞速发展，互联网业务系统所要处理的线上流量越来越大，Redis的单线程模式会导致系统消耗很多CPU时间在网络I/O上从而降低吞吐量，要提升Redis的性能有两个方向:

- 优化网络I/O模块
    - 零拷贝技术或者DPDK技术
        - 零拷贝技术有其局限性，无法完全适配Redis这一类复杂的网络I/O场景，更多网络I/O对CPU时间的
          消耗和Linux零拷贝技术。而DPDK技术通过旁路网卡I/O绕过内核协议栈的方式又太过于复杂以及需
          要内核甚至是硬件的支持。
    - 利用多核优势
- 提高机器内存读写的速度

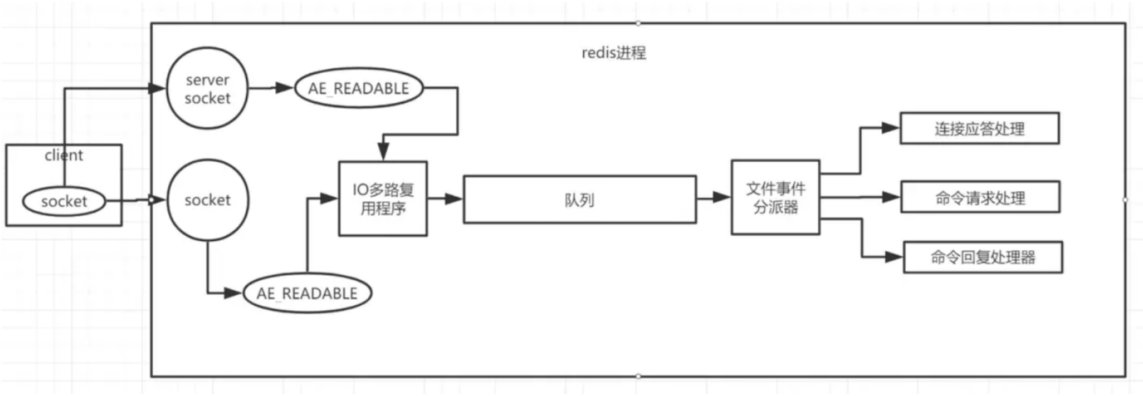
总结两个原因:

- 充分利用服务器CPU资源，目前主线程只能利用一个核

- 多线程任务可以分摊Redis同步IO读写负荷

介绍一下Redis的线程模型

redis6.0：之前Redis是基于reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器(file eventhandler)。由于这个文件事件处理器是单线程的，所以Redis才叫做单线程的模型。采用I0多路复用机制同时监听多个Socket，根据socket.上的事件来选择对应的事件处理器来处理这个事件。



文件事件处理器的结构包含了四个部分:

- 多个Socket。 Socket会产生AE_ *READABLE*和*AE* WRITABLE事件:
  - 当socket变得可读时或者有新的可以应答的socket出现时，socket就会产生一个AE_ READABLE事件
  - 当socket变得可写时，socket就会产生一一个AE_ WRITABLE事件。
- I0多路复用程序
- 文件事件分派器
- 事件处理器。事件处理器包括:连接应答处理器、命令请求处理器、命令回复处理器，每个处理器对应不同的socket事件:
  - 如果是客户端要连接Redis，那么会为socket关联连接应答处理器
  - 如果是客户端要写数据到Redis(读、写请求命令)，那么会为socket关联命令请求处理器
  - 如果是客户端要从Redis读数据，那么会为socket关联命令回复处理器

多个socket会产生不同的事件，不同的事件对应着不同的操作，I0多路复用程序监听着这些Socket,当这些Socket产生了事件，I0多路复用程序会将这些事件放到一个队列中，通过这个队列，以有序、同步、每次一个事件的方式向文件时间分派器中传送。当事件处理器处理完一个事件后，I0多路复用程序才会继续向文件分派器传送下一个事件。

Redis多线程的实现机制

流程简述如下:

- 主线程负责接收建立连接请求，获取Socket放入全局等待读处理队列。
- 主线程处理完读事件之后，通过RR (Round Robin)将这些连接分配给这些I0线程。
- 主线程阻塞等待I0线程读取Socket完毕。
- 主线程通过单线程的方式执行请求命令，请求数据读取并解析完成，但并不执行。
- 主线程阻塞等待I0线程将数据回写Socket完毕。

Redis6.0开启多线程后，是否会有线程并发安全问题

从实现机制可以看出，Redis的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程顺序执行。
所以我们不需要去考虑控制Key、Lua、事务，LPUSH/LPOP等等的并发及线程安全问题。

Redis6.0与Memcached多线程模型的对比

- 相同点:都采用了Master线程-Worker线程的模型。
- 不同点:Memcached执行主逻辑也是在Worker线程里，模型更加简单，实现了真正的线程隔离，符合我们对线程隔离的常规理解。而Redis把处理逻辑交还给Master线程，虽然一定程度 上增加了模型复杂度，但也解决了线程并发安全等问题。

Redis事务支持隔离性

Redis是单进程程序，并且它保证在执行事务时，不会对事务进行中断，事务可以运行直到执行完所有事务队列中的命令为止。因此，Redis的事务是总是带有隔离性的。

Redis为什么不支持事务回滚

- Redis命令只会因为错误的语法而失败，或是命令用在了错误类型的键上面，这些问题不能在入队时发现，这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中.
- 因为不需要对回滚进行支持，所以Redis的内部可以保持简单且快速。

Redis常见使用方式有哪些

- Redis单副本
    - 采用单个Redis节点部署架构，没有备用节点实时同步数据，不提供数据持久化和备份策略，适用于数据可靠性要求不高的纯缓存业务场景
        - 优点:
            - 架构简单，部署方便;
            - 高性价比:缓存使用时无需备用节点(单实例可用性可以用supervisor或crontab保证)，当然为了满足业务的高可用性，也可以牺牲-一个 备用节点，但同时刻只有一个实例对外提供服务;
            - 高性能。
        - 缺点:
            - 不保证数据的可靠性;
            - 在缓存使用，进程重启后，数据丢失，即使有备用的节点解决高可用性，但是仍然不能解决缓存预热问题，因此不适用于数据可靠性要求高的业务|;
            - 高性能受限于单核CPU的处理能力(Redis是单线程机制)，CPU为主要瓶颈，所以适合操作命令简单，排序、计算较少的场景。也可以考虑用Memcached替代。
- Redis多副本（主从）
    - Redis多副本，采用主从(replication)部署结构，相较于单副本而言最大的特点就是主从实例间数据实时同步，并且提供数据持久化和备份策略。主从实例部署在不同的物理服务器上，根据公司的基础环境配置，可以实现同时对外提供服务和读写分离策略。
- **Redis Sentinel**

主从模式下，当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这种方式并不推荐，实际生产中，我们优先考虑哨兵模式。这种模式下，master宕机，哨兵会自动选举master并将其他的slave指向新的master。

Redis Sentinel是社区版本推出的原生高可用解决方案，其部署架构主要包括两部分: Redis Sentinel集群和Redis数据集群。
其中Redis Sentinel集群是由若干Sentinel节点组成的分布式集群，可以实现故障发现、故障自动转移、配置中心和客户端通知。Redis Sentinel的节点数量要满足2n+1 (n>=1) 的奇数个。

- 优点:

  1.无中心架构

2.节点间数据共享，可动态调整数据分布

3.可扩展性：可线性扩展到1000多个节点，节点可动态添加或删除

4.高可用性：部分节点不可用时，集群仍可用。通过增加Slave做standby数据副本，能够实现故障自高可用性:部分节点不可用时，集群仍可用.通过增加从做备用数据副本，能够实现故障自动failover， 节点之间通过gossip协议交换状态信息，用投票机制完成Slave到Master的角色提升;动故障转移，节点之间通过绯闻协议交换状态信息，用投票机制完成从到主人的角色提升；

5.降低运维成本，提高系统的扩展性和可用性

- 缺点：

  1.数据通过异步复制，不保证数据的强一致性

主从复制的原理

Redis主从复制的核心原理：Redis的主从复制是提高Redis的可靠性的有效措施，主从复制的流程如下:

1.集群启动时,主从库间会先建立连接,为全量复制做准备

2.主库将所有数据同步给从库。从库收到数据后，在本地完成数据加载，这个过程依赖于内存快照RDB

3.在主库将数据同步给从库的过程中，主库不会阻塞，仍然可以正常接收请求。否则，redis的服务就被中断了。但是，这些请求中的写操作并没有记录到刚刚生成的RDB文件中。为了保证主从库的数据一敬性,主库会在内存中用专门的replication buffer,记录RDB文件生成收到的所有写操作。

4.最后，也就是第三个阶段，主库会把第二阶段执行过程中新收到的写命令。 再发送给从库。具体的操作是，当主库完成RDB文件发送后，就会把此时replocation buffer中修改操作发送给从库，从库再执行这些操作。这样一来，主从库就实现同步了

5.后续主库和从库都可以处理客户端读操作。写操作只能交给主库处理，主库接收到写操作后,还会将写操作发送给从库，实现增量同步

Redis主从架构数据会丢失嘛

有两种数据丢失的情况:
1.异步复制导致的数据丢失:因为master->slave的复制是异步的，所以可能有部分数据还没复制到slave，master就宕机了 ，此时这些部分数据就丢失了。
2.脑裂导致的数据丢失:某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着，此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master。这个时候，集群里就会有两个master，也就是所谓的脑裂。此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master,还继续写向旧master的数据
可能也丢失了。因此旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新从新的master复制数据。

如何解决主从架构数据丢失的问题

min-slaves-to-write默认情况下是0，min-slaves-max-1ag 默认情况下是10。
上面的配置的意思是要求至少有1个slave，数据复制和同步的延迟不能超过10秒。如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了。减小min-slaves-max-1ag参数的值，这样就可以避免在发生故障时大量的数据丢失，一旦发现延迟超过了该值就不会往master中写入数据。那么对于client，我们可以采取降级措施，将数据暂时写入本地缓存和磁盘中，在一段时间后重新写入master来保证数据不丢失;也可以将数据写入kafka消息队列，隔一段时间去消费kafka中的数据。

Redis哨兵怎么工作

- 每个Sentinel以每秒钟一次的频率向它所知的Master，Slave 以及其他Sentinel实例发送一个PING命令。
- 如果一个实例(instance) 距离最后一-次有效回复PING命令的时间超过down-after-milliseconds选项所指定的值，则这个实例会被当前Sentinel标记为主观下线。
- 如果一个Master被标记为主观下线，则正在监视这个Master的所有Sentinel要以每秒一次的频率确认Master的确进入了主观下线状态。
- 当有足够数量的Sentinel (大于等于配置文件指定的值)在指定的时间范围内确认Master的确进入了主观下线状态，则Master会 被标记为客观下线。
- 当Master被Sentinel标记为客观下线时，Sentinel 向下线的Master的所有Slave发送INFO命令的频率会从10秒一次改为每秒- -次(在一般情况下，每个Sentinel会以每10秒一次的频率向它已知的所有Master，Slave发送 INFO命令)。
- 若没有足够数量的Sentinel同意Master已经下线，Master 的客观下线状态就会变成主观下线。若Master重新向Sentinel的PING命令返回有效回复，Master的主观下线状态就会被移除。
- sentinel节点会与其他sentinel节点进行"沟通"，投票选举一个sentinel节 点进行故障处理，在从节点中选取一个主节点，其他从节点挂载到新的主节点上自动复制新主节点的数据。

转移故障时会从剩下的salve选举一个新的master，被选举为master的标准是什么?

如果一个master被认为客观下线了，而且大部分哨兵都允许了主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个salve来，会考虑salve的一些信息

- **跟master断开连接的时长**：如果一个slave跟master断开连接已经超过了down-after-milliseconds的10倍，外加master宕 机的时长，那么slave就被认为不适合选举为master.
- **slave优先级**:按照slave优先级进行排序，slave priority越低，优先级就越高
- **复制offset:**如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后， 优先级就越高
- **run id:**如果上面两个条件都相同，那么选择一个runid比较小的那个slave

什么是分布式锁?为什么用分布式锁?

锁在程序中的作用就是同步工具，保证共享资源在同一时刻只能被--个线程访问，Java中的锁我们都很熟悉了，像synchronized 、Lock都 是我们经常使用的，但是Java的锁只能保证单机的时候有效，分布式集群环境就无能为力了，这个时候我们就需要用到分布式锁。分布式锁，顾名思义，就是分布式项目开发中用到的锁，可以用来控制分布式系统之间同步访问共享资源。
思路是:在整个系统提供一个全局、唯一的获取锁的"东西"，然后每个系统在需要加锁时，都去问这个"东西"拿到一把锁，这样不同的系统拿到的就可以认为是同一把锁。至于这个"东西"，可以是Redis、Zookeeper，也可以是数据库。
一般来说，分布式锁需要满足的特性有这么几点:

- 互斥性:在任何时刻，对于同一条数据，只有一台应用可以获取到分布式锁;
- 高可用性:在分布式场景下，一小部分服务器宕机不影响正常使用，这种情况就需要将提供分布式锁的服务以集群的方式部署;
- 防止锁超时:如果客户端没有主动释放锁，服务器会在一段时间之后自动释放锁，防止客户端宕机或者网络不可达时产生死锁;
- 独占性:加锁解锁必须由同一台服务器进行，也就是锁的持有者才可以释放锁，不能出现你加的锁，别人给你解锁了。

分布式锁的三个核心要素

加锁

当一个线程执行setnx返回1，说明key原本不存在，该线程成功得到了锁;

当一个线程执行setnX返回0，说明键已经存在，该线程抢锁失败；

```
setx key chen
```

解锁

有加锁就得有解锁。当得到的锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式就是执行del指令。

释放锁之后，其他线程就可以继续执行setnx命令来获得锁。

```
del key
```

锁超时

锁超时知道的是:如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住，别的线程别想进来。所以，setnx的key必须设置一个超时时间，以保证即使没有被显示释放，这把锁也要在一段时间后自动释放。setnx不支持超时参数，所以需要额外指令

```
expire key 30
```

上述分布式锁存在的问题

SETNX和EXPIRE非原子性
假设一个场景中，某一个线程刚执行setnx，成功得到了锁。此时setnx刚执行成功，还未来得及执行expire命令，节点就挂掉了。此时这把锁就没有设置过期时间，别的线程就再也无法获得该锁。
解决措施:
由于setnx指令本身是不支持传入超时时间的，而在Redis2.6.12版本.上为set指令增加了可选参数,用法如下:

```
SET key value [EX seconds] [PX milliseconds] [NX |Xx]
```

- EX second:设置键的过期时间为second秒;
- PX millisecond:设置键的过期时间为millisecond毫秒;
- NX: 只在键不存在时，才对键进行设置操作;
- XX:只在键已经存在时，才对键进行设置操作;
- SET操作完成时，返回OK，否则返回nil。

锁误解除
如果线程A成功获取到了锁，并且设置了过期时间30秒，但线程A执行时间超过了30秒，锁过期自动释放，此时线程B获取到了锁;随后A执行完成，线程A使用DEL命令来释放锁，但此时线程B加的锁还没有执行完成，线程A实际释放的线程B加的锁。

- **解决办法:**
  在del释放锁之前加一个判断，验证当前的锁是不是自己加的锁。

具体在加锁的时候把当前线程的id当做value，可生成一个UUID标识当前线程，在删除之前验证key对应的value是不是自己线程的id。还可以使用lua脚本做验证标识和解锁操作。

超时解锁导致并发

如果线程A成功获取锁并设置过期时间30秒，但线程A执行时间超过了30秒，锁过期自动释放，此时线程B获取到了锁，线程A和线程B并发执行。A、B两个线程发生并发显然是不被允许的，一般有两种方式解决该问题:

- 将过期时间设置足够长，确保代码逻辑在锁释放之前能够执行完成。
- 为获取锁的线程增加守护线程，为将要过期但未释放的锁增加有效时间。

不可重复入

当线程在持有锁的情况下再次请求加锁，如果-一个锁支持一个线程多次加锁，那么这个锁就是可重入的。如果一个不可重入锁被再次加锁，由于该锁已经被持有，再次加锁会失败。Redis可通过对锁进行重入计数，加锁时加1，解锁时减1，当计数归0时释放锁。

RedLock

Redlock是一种算法，Redlock 也就是Redis Distributed Lock，可用实现多节点Redis的分布式锁。是一种算法，Redlock也就是Redis分布式锁，可用实现多节点Redis的分布式锁。RedLock官方推荐，Redisson完成 了对Redlock算法封装。此种方式具有以下特性:

- 互斥访问:即永远只有一个client能拿到锁
- 避免死锁:最终client都可能拿到锁，不会出现死锁的情况，即使锁定资源的服务崩溃或者分区，仍然能释放锁。
- 容错性:只要大部分Redis节点存活(一半以上)，就可以正常提供服务

如果现在有个读超高并发的系统，用redis来抗住大部分读请求，你怎么设计

如果是读高并发的话，先看读并发的数量级是多少，因为Redis单机的读QPS在万级，每秒几万没问题，使用一主多从+哨兵集群的缓存架构来承载每秒10W+的读并发，主从复制，读写分离。使用哨兵集群主要是提高缓存架构的可用性，解决单点故障问题。主库负责写，多个从库负责读，支持水平扩容，根据读请求的QPS来决定加多少个Redis从实例。如果读并发继续增加的话，只需要增加Redis从实例就行了。如果需要缓存1T+的数据，选择Redis cluster模式，每个主节点存一部分数据，假设一个master存32G，那只需要n*32G>=1T, n个这样的master节点就可以支持1T+的海量数据的存储了。