Java面试题-缓存+Redis

一.使用缓存的合理性问题

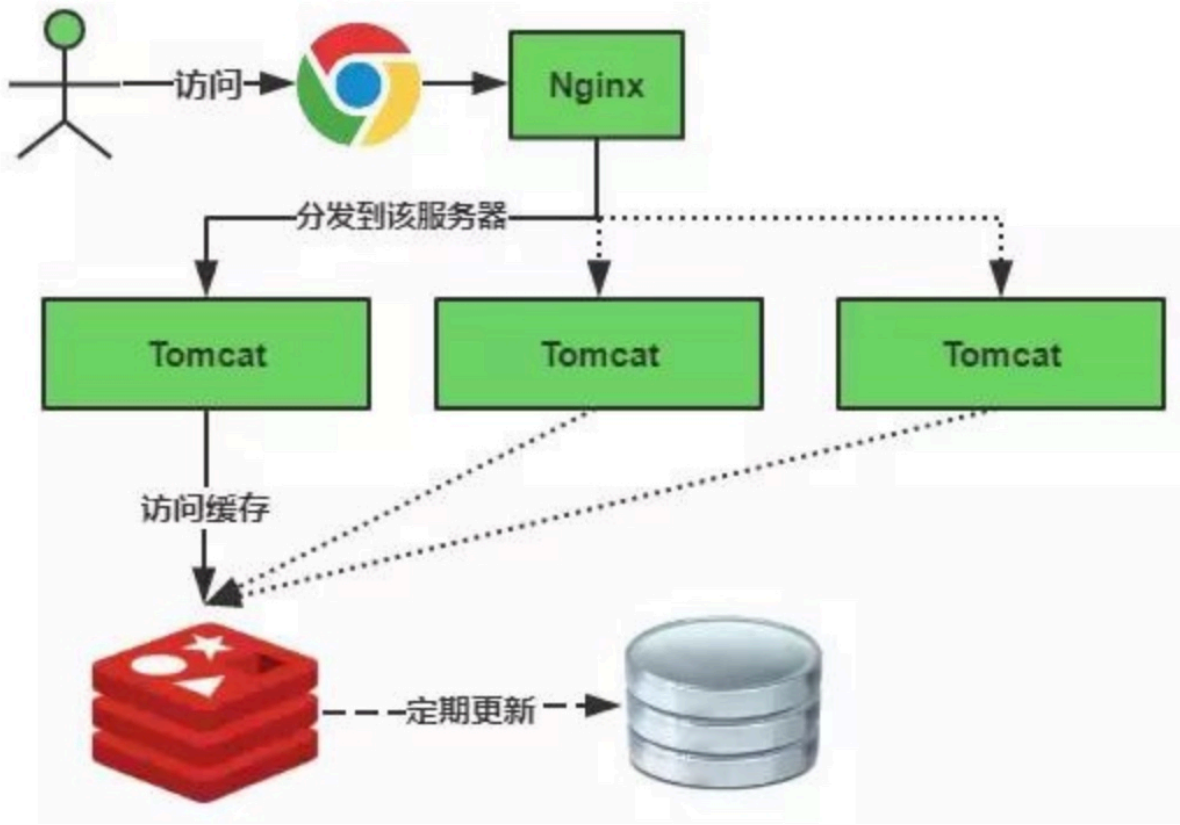
- 热点数据，缓存才有价值
- 频繁修改的数据，看情况考虑使用缓存
- 数据不一致性
- 缓存更新机制
- 缓存可用性
- 缓存服务降级
- 缓存预热
- 缓存穿透

二.缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

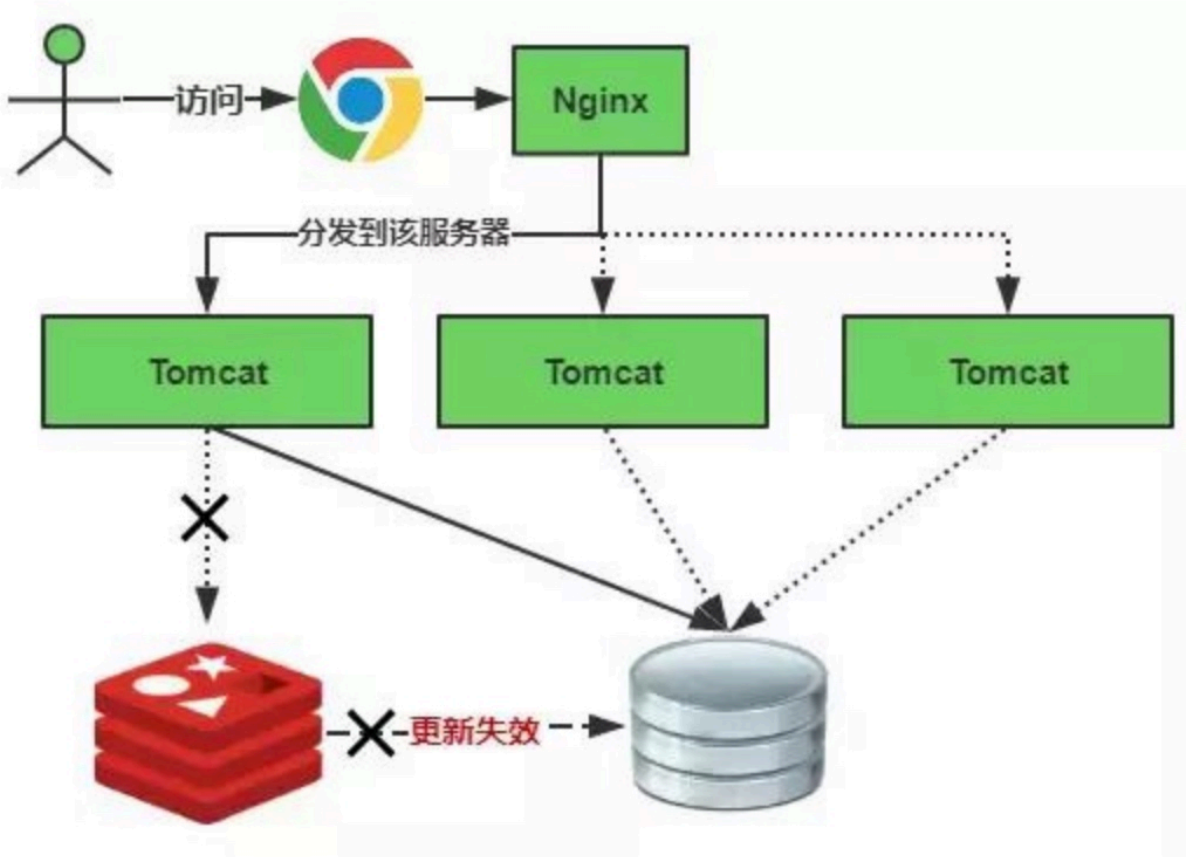
1.缓存雪崩

缓存雪崩我们可以简单的理解为：由于原有缓存失效，新缓存还未到这个期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

缓存正常从Redis中获取，示意图如下：



缓存失效瞬间示意图如下：



缓存雪崩的解决方案：

- （1）碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队，伪代码如下：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    String lockKey = cacheKey;

    String cacheValue = CacheHelper.get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        synchronized(lockKey) {
            cacheValue = CacheHelper.get(cacheKey);
            if (cacheValue != null) {
                return cacheValue;
            } else {
                //这里一般是sql查询数据
                cacheValue = GetProductListFromDB();
                CacheHelper.Add(cacheKey, cacheValue, cacheTime);
            }
        }
        return cacheValue;
    }
}
```

加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

注意：

加锁排队的解决方式是处理分布式环境的并发问题，有可能还要解决分布式锁的问题；线程还会被阻塞，用户体验很差！因此，在真正的高并发场景下很少使用！

- （2）给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存，实例伪代码如下：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    //缓存标记
    String cacheSign = cacheKey + "_sign";

    String sign = CacheHelper.Get(cacheSign);
    //获取缓存值
    String cacheValue = CacheHelper.Get(cacheKey);
    if (sign != null) {
        return cacheValue; //未过期，直接返回
    } else {
        CacheHelper.Add(cacheSign, "1", cacheTime);
        ThreadPool.QueueUserWorkItem((arg) -> {
            //这里一般是 sql查询数据
            cacheValue = GetProductListFromDB();
            //日期设缓存时间的2倍，用于脏读
            CacheHelper.Add(cacheKey, cacheValue, cacheTime * 2);
        });
        return cacheValue;
    }
}
```

解释说明：

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际key的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长1倍，例：标记缓存时间30分钟，数据缓存设置为60分钟。 这样，当缓存标记key过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

关于缓存崩溃的解决方法，这里提出了三种方案：使用锁或队列、设置过期标志更新缓存、为key设置不同的缓存失效时间，还有一各被称为“二级缓存”的解决方法，有兴趣的读者可以自行研究。

2.缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回

空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

缓存穿透解决方案：

（1）采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

（2）如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓存中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";

    String cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    }

    cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        //数据库查询不到，为空
        cacheValue = GetProductListFromDB();
        if (cacheValue == null) {
            //如果发现为空，设置个默认值，也缓存起来
            cacheValue = string.Empty;
        }
        CacheHelper.Add(cacheKey, cacheValue, cacheTime);
        return cacheValue;
    }
}
```

把空结果也给缓存起来，这样下次同样的请求就可以直接返回空了，即可以避免当查询的值为空时引起的缓存穿透。同时也可以单独设置个缓存区域存储空值，对要查询的key进行预先校验，然后再放行给后面的正常缓存处理逻辑。

3.缓存预热

缓存预热就是系统上线后，提前将相关的缓存数据直接加载到缓存系统。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！

缓存预热解决方案：

（1）直接写个缓存刷新页面，上线时手工操作下；

（2）数据量不大，可以在项目启动的时候自动进行加载；

（3）定时刷新缓存；

4.缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

5.缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

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：

（1）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；

（2）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；

（3）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；

（4）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。

四.缓存崩溃

- 碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。
- 加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间 key 是锁着的，这时过来 1000 个请求 999 个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法。

五.缓存降级

1.页面降级

在大促或者某些特殊情况下，某些页面占用了一些稀缺服务资源，在紧急情况下可以对其整个降级，以达到丢卒保帅；

2.页面片段降级

比如商品详情页中的商家部分因为数据错误了，此时需要对其进行降级；

3.页面异步请求降级

比如商品详情页上有推荐信息/配送至等异步加载的请求，如果这些信息响应慢或者后端服务有问题，可以进行降级；

4.服务功能降级

比如渲染商品详情页时需要调用一些不太重要的服务：相关分类、热销榜等，而这些服务在异常情况下直接不获取，即降级即可；

5.读降级

比如多级缓存模式，如果后端服务有问题，可以降级为只读缓存，这种方式适用于对读一致性要求不高的场景；

6.写降级

比如秒杀抢购，我们可以只进行Cache的更新，然后异步同步扣减库存到DB，保证最终一致性即可，此时可以将DB降级为Cache。

7.爬虫降级

在大促活动时，可以将爬虫流量导向静态页或者返回空数据，从而保护后端稀缺资源。

8.自动开关降级

自动降级是根据系统负载、资源使用情况、SLA等指标进行降级。

9.超时降级

当访问的数据库/http服务/远程调用响应慢或者长时间响应慢，且该服务不是核心服务的话可以在超时后自动降级；比如商品详情页上有推荐内容/评价，但是推荐内容/评价暂时不展示对用户购物流程不会产生很大的影响；对于这种服务是可以超时降级的。如果是调用别人的远程服务，和对方定义一个服务响应最大时间，如果超时了则自动降级。

六.选择合适的数据存储方案

1.关系型数据库 MySQL

MySQL 是一个最流行的关系型数据库，在互联网产品中应用比较广泛。一般情况下，MySQL 数据库是选择的第一方案，基本上有 80% ~ 90% 的场景都是基于 MySQL 数据库的。因为，需要关系型数据库进行管理，此外，业务存在许多事务性的操作，需要保证事务的强一致性。同时，可能还存在一些复杂的 SQL 的查询。值得注意的是，前期尽量减少表的联合查询，便于后期数据量增大的情况下，做数据库的分库分表。

2.内存数据库 Redis

随着数据量的增长，MySQL 已经满足不了大型互联网类应用的需求。因此，Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓存服务降级、缓存穿透、缓存预热等缓存使用问题。

3.文档数据库 MongoDB

MongoDB 是对传统关系型数据库的补充，它非常适合高伸缩性的场景，它是可扩展性的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

此外，日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

MongoDB 还适合存储大尺寸的数据，GridFS 存储方案就是基于 MongoDB 的分布式文件存储系统。

4.列族数据库 HBase

HBase 适合海量数据的存储与高性能实时查询，它是运行于 HDFS 文件系统之上，并且作为 MapReduce 分布式处理的目标数据库，以支撑离线分析型应用。在数据仓库、数据集市、商业智能等领域发挥了越来越多的作用，在数以千计的企业中支撑着大量的大数据分析场景的应用。

5.全文搜索引擎 ElasticSearch

在一般情况下，关系型数据库的模糊查询，都是通过 like 的方式进行查询。其中，like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。ElasticSearch 作为一个建立在全文搜索引擎 Apache Lucene 基础上的实时的分布式搜索和分析引擎，适用于处理实时搜索应用场景。此外，使用 ElasticSearch 全文搜索引擎，还可以支持多词条查询、匹配度与权重、自动联想、拼写纠错等高级功能。因此，可以使用 ElasticSearch 作为关系型数据库全文搜索的功能补充，将要进行全文搜索的数据缓存一份到 ElasticSearch 上，达到处理复杂的业务与提高查询速度的目的。

ElasticSearch 不仅仅适用于搜索场景，还非常适合日志处理与分析的场景。**著名的 ELK 日志处理方案，由 ElasticSearch、LogStash 和 KibAna 三个组件组成**，包括了日志收集、聚合、多维度查询、可视化显示等。

七.聊聊 Redis 使用场景

- 缓存
- 会话缓存
- 时效性
- 访问频率
- 计数器
- 社交列表
- 记录用户判定信息
- 交集、并集和差集
- 热门列表与排行榜
- 最新动态
- 消息队列

八.Redis 有哪些类型

在 Redis 中有五种数据类型

- String：字符串
- Hash：字典
- List：列表
- Set：集合
- Sorted Set：有序集合

九.Redis 持久化机制

Redis 有两种持久化机制：

1.RDB

RDB 持久化方式会在一个特定的间隔保存那个时间点的一个数据快照

2.AOF

AOF 持久化方式则会记录每一个服务器收到的写操作。在服务启动时，这些记录的操作会逐条执行从而重建出原来的数据。写操作命令记录的格式跟 Redis 协议一致，以追加的方式进行保存

Redis 的持久化是可以禁用的，就是说你可以让数据的生命周期只存在于服务器的运行时间里。两种方式的持久化是可以同时存在的，但是当 Redis 重启时，AOF 文件会被优先用于重建数据。

十.Redis 内部结构

Redis 内部使用一个 redisObject 对象来表示所有的 key 和 value。

- type： 代表一个 value 对象具体是何种数据类型。
- encoding： 是不同数据类型在 redis 内部的存储方式，比如：type=string 代表 value 存储的是一个普通字符串，那么对应的 encoding 可以是 raw 或者是 int，如果是 int 则代表实际 redis 内部是按数值型类存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如："123" "456"这样的字符串。
- vm 字段：只有打开了 Redis 的虚拟内存功能，此字段才会真正的分配内存，该功能默认是关闭状态的。 Redis 使用 redisObject 来表示所有的 key/value 数据是比较浪费内存的，当然这些内存管理成本的付出主要也是为了给 Redis 不同数据类型提供一个统一的管理接口，实际作者也提供了多种方法帮助我们尽量节省内存使用。

十一.Redis 为什么是单线程的

因为 CPU 不是 Redis 的瓶颈。Redis 的瓶颈最有可能是机器内存或者网络带宽。（以上主要来自官方 FAQ）既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章地采用单线程的方案了。

十二.Redis 内存淘汰机制

Redis 内存淘汰指的是用户存储的一些键被可以被 Redis 主动地从实例中删除，从而产生读 miss 的情况，那么 Redis 为什么要有这种功能？这就是我们需要探究的设计初衷。Redis 最常见的两种应用场景为缓存和持久存储，首先要明确的一个问题是内存淘汰策略更适合于那种场景？是持久存储还是缓存？

假设我们有一个 Redis 服务器，服务器物理内存大小为 1G 的，我们需要存在 Redis 中的数据量很小，这看起来似乎足够用很长时间了，随着业务量的增长，

我们放在 Redis 里面的数据越来越多了，数据量大小似乎超过了 1G，但是应用还可以正常运行，这是因为操作系统的可见内存并不受物理内存限制，而是虚拟内存，物理内存不够用没关系，操作系统会从硬盘上划出一片空间用于构建虚拟内存，比如32位的操作系统的可见内存大小为 `2^32`，而用户空间的可见内存要小于 `2^32` 很多，大概是 3G 左右。好了，我们庆幸操作系统为我们做了这些，但是我们需要知道这背后的代价是不菲的，不合理的使用内存有可能发生频繁的 swap，频繁 swap 的代价是惨痛的。所以回过头来看，作为有追求的程序员，我们还是要小心翼翼地使用好每块内存，把用户代码能解决的问题尽量不要抛给操作系统解决。

内存的淘汰机制的初衷是为了更好地使用内存，用一定的缓存 miss 来换取内存的使用效率。

1.如何用

作为 Redis 用户，我们如何使用 Redis 提供的这个特性呢?

```
# maxmemory <bytes>
```

我们可以通过配置 `redis.conf` 中的 `maxmemory` 这个值来开启内存淘汰功能，至于这个值有什么意义，我们可以通过了解内存淘汰的过程来理解它的意义：

- 客户端发起了需要申请更多内存的命令（如set）
- Redis 检查内存使用情况，如果已使用的内存大于 `maxmemory` 则开始根据用户配置的不同淘汰策略来淘汰内存（key），从而换取一定的内存
- 如果上面都没问题，则这个命令执行成功

`maxmemory` 为 0 的时候表示我们对 Redis 的内存使用没有限制

2.内存淘汰策略

内存淘汰只是 Redis 提供的一个功能，为了更好地实现这个功能，必须为不同的应用场景提供不同的策略，内存淘汰策略讲的是为实现内存淘汰我们具体怎么做，要解决的问题包括淘汰键空间如何选择？在键空间中淘汰键如何选择?

Redis 提供了下面几种淘汰策略供用户选择，其中默认的策略为 `noeviction` 策略：

- noeviction：当内存使用达到阈值的时候，所有引起申请内存的命令会报错
- allkeys-lru：在主键空间中，优先移除最近未使用的key

- volatile-lru：在设置了过期时间的键空间中，优先移除最近未使用的 key
- allkeys-random：在主键空间中，随机移除某个 key
- volatile-random：在设置了过期时间的键空间中，随机移除某个 key
- volatile-ttl：在设置了过期时间的键空间中，具有更早过期时间的 key 优先移除

这里补充一下主键空间和设置了过期时间的键空间，举个例子，假设我们有一批键存储在Redis中，则有那么一个哈希表用于存储这批键及其值，如果这批键中有一部分设置了过期时间，那么这批键还会被存储到另外一个哈希表中，这个哈希表中的值对应的是键被设置的过期时间。设置了过期时间的键空间为主键空间的子集。

3.如何选择淘汰策略

我们了解了 Redis 大概提供了这么几种淘汰策略，那么如何选择呢？淘汰策略的选择可以通过下面的配置指定：

```
# maxmemory-policy noeviction
```

但是这个值填什么呢？为解决这个问题，我们需要了解我们的应用请求对于 Redis 中存储的数据集的访问方式以及我们的诉求是什么。同时 Redis 也支持 Runtime 修改淘汰策略，这使得我们不需要重启 Redis 实例而实时的调整内存淘汰策略。

下面看看几种策略的适用场景：

- allkeys-lru：如果我们的应用对缓存的访问符合幂律分布（也就是存在相对热点数据），或者我们不太清楚我们应用的缓存访问分布状况，我们可以选择 allkeys-lru 策略
- allkeys-random：如果我们的应用对于缓存 key 的访问概率相等，则可以使用这个策略
- volatile-ttl：这种策略使得我们可以向 Redis 提示哪些 key 更适合被 eviction

另外，`volatile-lru` 策略和 `volatile-random` 策略适合我们将一个Redis实例既应用于缓存和又应用于持久化存储的时候，然而我们也可以通过使用两个 Redis 实例来达到相同的效果，值得一提的是将key设置过期时间实际上会消耗更多的内存，因此我们建议使用 `allkeys-lru` 策略从而更有效率的使用内存。

4.非精准的 LRU

上面提到的 LRU（Least Recently Used）策略，实际上 Redis 实现的 LRU 并不是可靠的 LRU，也就是名义上我们使用 LRU 算法淘汰键，但是实际上被淘汰的键并不一定是真正的最久没用的，这里涉及到一个权衡的问题，如果需要在全部键空间内搜索最优解，则必然会增加系统的开销，Redis 是单线程的，也就是同一个实例在每一个时刻只能服务于一个客户端，所以耗时的操作一定要谨慎。为了在一定成本内实现相对的 LRU，早期的 Redis 版本是基于采样的 LRU，也就是放弃全部键空间内搜索解改为采样空间搜索最优解。自从 Redis3.0 版本之后，Redis 作者对于基于采样的 LRU 进行了一些优化，目的是在一定的成本内让结果更靠近真实的 LRU。

十三.Redis 集群方案与实现

- 客户端分片
- 基于代理的分片
- 路由查询
- 客户端分片
- 由客户端决定 key 写入或者读取的节点
- 包括 Jedis 在内的一些客户端，实现了客户端分片机制

1.路由查询

将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行。

2.开源方案

Sentinel