## 一、键值设计

**1. key名设计**

(1)【建议】: 可读性和可管理性

以业务名(或数据库名)为前缀(防止key冲突)，用冒号分隔，

例如：业务名:表名:id / user:pic:1

(2)【建议】：简洁性

保证语义的前提下，控制key的长度，当key较多时，内存占用也不容忽视，

例如：user:{uid}:friends:messages:{mid} ，简化为 u:{uid}:fr:m:{mid}

(3)【强制】：不要包含特殊字符

反例：包含空格、换行、单双引号以及其他转义字符

**2. value设计**

(1)【强制】：拒绝bigkey(防止网卡流量、慢查询)

string类型控制在10KB以内，hash、list、set、zset元素个数不要超过5000。

反例：一个包含200万个元素的list。

**注意：**非字符串的bigkey，不要使用del删除，使用hscan、sscan、zscan方式渐进式删除，同时要注意防止bigkey过期时间自动删除问题(例如一个200万的zset设置1小时过期，会触发del操作，造成阻塞).

(2)【推荐】：选择适合的数据类型。

例如：实体类型(要合理控制和使用数据结构内存编码优化配置,例如ziplist，但也要注意节省内存和性能之间的平衡)

反例：

set user:1:name tom

set user:1:age 19

set user:1:favor football

正例:

hmset user:1 name tom age 19 favor football

更推荐方式：

使用将实例对象序列化后，使用String的KV格式记录，可以将该对象先序列化成一个string或者byte数组后，存放在redis中。

(3) 【推荐】：控制key的生命周期，缓存不是垃圾桶。

建议使用expire设置过期时间(条件允许可以打散过期时间，防止集中过期)，不过期的数据重点关注idletime。

## 二、命令使用

1.【推荐】 O(N)时间复杂度命令要关注N的数量

例如hgetall、lrange、smembers、zrange、sinter等并非不能使用，但是需要明确N的值。有遍历的需求可以使用hscan、sscan、zscan代替。

2.【推荐】：禁用命令

禁止线上使用keys、flushall、flushdb等，通过redis的rename机制禁掉命令，或者使用scan的方式渐进式处理。

3.【推荐】合理使用select

redis的多数据库较弱，使用数字进行区分，很多客户端支持较差，同时多业务用多数据库实际还是单线程处理，会有干扰。

4.【推荐】使用批量操作提高效率

原生命令：例如mget、mset。 非原生命令：可以使用pipeline提高效率。 但要注意控制一次批量操作的元素个数(例如500以内，实际也和元素字节数有关)。

注意两者不同：

a).原生是原子操作，pipeline是非原子操作。

b).pipeline可以打包不同的命令，原生做不到

c).pipeline需要客户端和服务端同时支持。

4.【建议】Redis事务功能较弱，不建议过多使用

Redis的事务功能较弱(不支持回滚)，而且集群版本(自研和官方)要求一次事务操作的key必须在一个slot上(可以使用hashtag功能解决)，所以codis等集群方案不支持事务。codis不支持的命令见 Codis不支持的命令

5.【建议】Redis集群版本在使用Lua上有特殊要求：

a).所有key都应该由 KEYS 数组来传递，redis.call/pcall 里面调用的redis命令，key的位置，必须是KEYS array, 否则直接返回error，"-ERR bad lua script for redis cluster, all the keys that the script uses should be passed using the KEYS array"

b).所有key，必须在1个slot上，否则直接返回error, "-ERR eval/evalsha command keys must in same slot"

6.【建议】必要情况下使用monitor命令时，要注意不要长时间使用。

## 三、业务使用缓存规则

1 将热数据放到缓存中

2 所有缓存信息都应设置过期时间

3 缓存过期时间应当分散以避免集中过期

4 缓存层不应抛出异常

缓存应有降级处理方案，缓存出了问题要能回源到数据库进行处理

5 读的顺序是先缓存，后数据库；写的顺序是先数据库，后缓存

6 可以进行适当的缓存预热

对于上线后可能会有大量读请求的应用，在上线之前可预先将数据写入缓存中

7 数据一致性问题

数据源发生变更时可能导致缓存中数据与数据源中数据不一致，应根据实际业务需求来选择适当的缓存更新策略：

■ 主动更新：在数据源发生变更时同步更新缓存数据或将缓存数据过期。一致性高，维护成本较高。

■ 被动删除：根据缓存设置的过期时间有Redis负责数据的过期删除。一致性较低，维护成本较低。

## 四、客户端使用

1.【推荐】避免多个应用使用同一个Redis实例

不相干的业务拆分，公共数据做服务化。

2.【建议】根据自身业务类型，选好maxmemory-policy(最大内存淘汰策略)，设置好过期时间。

默认策略是volatile-lru，即超过最大内存后，在过期键中使用lru算法进行key的剔除，保证不过期数据不被删除，但是可能会出现OOM问题。

其他策略如下：

allkeys-lru：根据LRU算法删除键，不管数据有没有设置超时属性，直到腾出足够空间为止。

allkeys-random：随机删除所有键，直到腾出足够空间为止。

volatile-random:随机删除过期键，直到腾出足够空间为止。

volatile-ttl：根据键值对象的ttl属性，删除最近将要过期数据。如果没有，回退到noeviction策略。

noeviction：不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error) OOM command not allowed when used memory"，此时Redis只响应读操作。