Redis 开发规范

> 本文介绍了在使用Redis的开发规范，从键值设计、命令使用、客户端使用等方面进行说明，通过本文的介绍可以减少使用Redis过程带来的问题；
>
> - 不合理的使用Redis会导致性能变差、响应慢、服务可用性下降、成本高等风险

一、键值设计

1. key名设计

- 【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，比如业务名:表名:id

```
ai:material:1
```

- 【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}。
```

- 【建议】：使用":"字符进行分层

":"字符除了能区分业务，在集群拆分迁移或删除数据中操作起来会更方便

- 【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大小设计

- 【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，否则可能无法保证性能。

反例：一个包含200万个元素的list。

非字符串的bigkey，不要使用del删除(4以上的版本可以使用unlink异步删除)，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，而且该操作不会不出现在慢查询中

[注]redis 4.0 lazy-free特性已支持key的异步删除

- 【推荐】：选择合适的数据存储服务(Redis不是万金油)

反例：使用Redis列表List做大吞吐消息队列，下游消费不及时会造成Redis内存容量打满

- 【推荐】：选择适合的数据类型

数据类型	存储的值	常见使用场景	常用存储模式
STRING(字符串)	可存储整数、浮点数和字符串，最大512M	计数器,实时指标,缓存串	GET/SET/INCRBY
HASH(散列/哈希表)	无序的键值对(最大2^32-1个字段)	“对象”结构存储	HSET/HGET/HLEN/HKEYS
LIST(列表)	字符串元素列表，存储与元素插入顺序一致(最大2^32-1个元素)	LIST（列表） 字符串元素列表，存储与元素插入顺序一致(最大2^32-1个元素) 队列，堆栈，日志消息队列，timeline **redis不建议做的的队列，以免数据倾斜和阻塞，大的队列建议使用专门的消息队列产品**	LPUSH/LPOP/LRANGE
SET(集合)	存储无序、唯一的字符串元素集合(2^32-1个元素)	元素记录不重复无序的场景（用户已安装的appid,参加的活动）	SADD/SMOVE/SCARD/SMEBMERS
SORTED SET(有序集合)	每个字符串成员附加1浮点数分值(score)，元素排列顺序由其分值大小决定	排行榜(米币用户充值topN)，实时动态	ZADD/ZRANGE/ZCARD

3. 控制key的生命周期，redis不是垃圾桶

由于Redis不能保证数据的强一致性，**强烈建议Key都设置TTL值**，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) 保证不使用的Key能被及时清理或淘汰，使内存复用。

4. 数据打散，避免节点倾斜

在集群模式下，热点Key和大容量Key尽量设计打散；避免集群QPS和容量不均衡，导致部分节点QPS过载和容量过大。缓存热点key一致性要求不高可以缓存在本地，降低Redis请求。

二、命令使用

Redis查询复杂度

1. O(N)命令关注N的数量

redis 那么快，慢查询除了网络延迟，就属于这些批量操作函数。大多数线上问题都是由于这些函数引起；例如hgetall、lrange、smembers、zrange、sinter等并非不能使用，但是需要明确N的值控制在100以内。有遍历的需求可以使用hscan、sscan、zscan代替。

2. 禁用危险命令

禁止线上使用keys、flushall、flushdb、config等，生产环境以及通过redis的rename机制禁掉命令，请不要在生产环境尝试，或者使用scan的方式渐进式处理。

3. Redis事务功能较弱，不建议使用

Redis的事务功能较弱(不支持回滚)，而且集群版本要求一次事务操作的key必须在一个slot上

三、容量管理

1. 容量预估

新申请Redis需要做好容量、QPS、流量（根据qps、平均value大小）评估，可参考 集群申请 进行评估；如果当前redis支持某个活动、新上线需求时，需要再次进行上述评估，以便DBA进行资源调配。

2. 扩缩容

Redis分片调整需要数据做rebalance，扩缩容时间与数据量和集群分片规模成正比，核心业务或强依赖Redis的服务强烈建议 搬迁集群扩容 。

3. 内存限额

Redis**单分片使用内存不要超过10G**，超过10G限额有性能风险，如超过限额请提前清理数据或者找DBA申请分片扩容。

四、客户端使用

强烈要求 lettuce直连访问cluster集群必须要配置拓扑刷新 否则节点迁移维护会导致访问失败

1. 避免多个应用使用一个Redis实例

> 不相干的业务拆分，公共数据做服务化

2. 使用带有连接池的数据库，可以有效控制连接，同时提高效率

```
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();
    // 具体的命令
    jedis.executeCommand()
} catch (Exception e) {
    logger.error("op key {} error: " + e.getMessage(), key, e);
        throw e;
} finally {
    // 注意这里不是关闭连接，在JedisPool模式下，Jedis会被归还给资源池。
    if (jedis != null)
        jedis.close();
}
```

3. 高并发下建议客户端添加熔断功能 (例如 netflix hystrix)

4. 敏感数据设置密码

5. 根据自身业务类型，选好最大内存淘汰策略，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。

五、架构设计

Redis非强一致性的缓存数据库，不能当作DB来使用，DBA不承诺Redis能按照指定的时间点进行数据恢复；研发侧架构设计上必须考虑到Redis中的数据误删等场景的恢复方案。

Redis Development Specifications

> This article introduces the specifications when using Redis, and explains the design requirements for key-value store, commands, client usage, etc. Through this article, we can greatly reduce the problems while using Redis.
>
> - Inappropriate usage of Redis will lead to poor performance, longer response time, reduced availabilty of services and higher costs.

1. Key-Value Store Design

a. Key Naming

- 【Suggestion】: Readability and Manageability

Include business/service name (or database name) as prefix (to prevent duplicate keys) and separate with colons (':') e.g. Business/Service Name: Table Name: id

CSS
```css
ai:material:1
```

- 【Suggestion】: Simplicity

While maintaining context and semantics, keep key length as short as possible. When there are many keys, the memory usage for individual keys can become significant.

Groovy
```groovy
user:{uid}:friends:messages:{mid} can be simplified to u:{uid}:fr:m:{mid}。
```

- 【Suggestion】: Use colons to perform layering

Apart from being used as a separator for naming, the colon (':') can make it more convenient for operations on clusters (splitting, migration, deleting data etc)

- 【Mandatory】: Do not include special characters

Bad Example: Including spaces, newlines, quotation marks and other escape special symbols

b. Size of Value

- 【Mandatory】: Avoid bigkeys (In order to prevent high network traffic and slow queries)

string should be within 10KB，hash、list、set、zset should not exceed more than 5000 elements to guarantee performance.

Bad Example: A list containing more than 2 million elements

For bigkeys that are non-string, do not use del to delete (from ver 4.0 onwards, can use unlink for asynchronous deletion). Instead, use hscan, sscan, zscan method to progressively delete. Take note of the auto delete issue when bigkeys hit expiration time (e.g. A zset containing 2 million elements that has a configuration of expiry after 1 hour will automatically start del operation, causing blocking and this operation will not appear in slow query log)

[Attention] redis 4.0 lazy-free feature already supports asynchronous deletion of key

- 【Suggestion】: Choose a suitable data storage service (Redis is not suitable for all applications)

Bad Example: Using Redis List to increase message queue throughput. The downstream consumption will cause Redis memory capacity to be filled quickly.

- 【Suggestion】： Select the appropriate data type

Data Structures	Values to Store	Common Applications	Common Operations
STRING	• Can store byte string values, integers and floats • Should not exceed 512MB of data	Counter, real time indicator, cache strings	GET/SET/INCRBY
HASH	• Unordered key-value pairs • Maximum 2 ^ 32-1 pairs	Map complex objects inside Redis by using fields for object attributes	HSET/HGET/HLEN/HKEYS
LIST	• Linked list structure: ordered sequence of strings stored in the same order as insertion • Maximum 2 ^ 32-1 elements	String list stored in the same order as insertion, queue, stack, log message queue, timeline **Redis does not recommend making large message queues to avoid data skew and blocking. Large queues are recommended to use a specialized message queueing systems instead of Redis**	LPUSH/LPOP/LRANGE
SET	• Stores an unordered, unique collection of string elements • Maximum 2 ^ 32-1	The element record does not include repeated unordered scenarios (User has already installed appid or participated in activities etc)	SADD/SMOVE/SCARD/SMEBMERS

	elements		
SORTED SET(ZSET)	· Each string has a floating point score (score) attached to it, and the order of the elements is determined by the size of this score	Ranking (MiPay users top N), real-time dynamics	ZADD/ZRANGE/ZCARD

c. Control the lifespan of keys, do not treat Redis as a trash collector

Since Redis does not guarantee strong consistency of data, it is strongly recommended to **set the TTL value of Key** and use `expire` to set the expiration time (ensure conditions allow staggering of the expiration time to prevent centralized expiration) to ensure that unused keys can be cleaned up or eliminated for memory reallocation.

d. Data fragmentation to avoid node skewing

In cluster mode, hotspot keys and large keys are designed to be broken up as much as possible. This is to avoid unbalanced cluster QPS and capacity, resulting in QPS overload and excessive volume on certain nodes. If caching hotkeys do not meet consistency requirements, consider caching locally to reduce Redis requests.

2. Command Use

Redis查询复杂度

a. For O(N) commands, pay attention to the value of N

Redis is very quick, slow queries are usually caused by bulk operations, with the exception of network latency. Most issues faced in live environment are caused by these functions; For example, hgetall, lrange, smembers, zrange, sinter, etc are not unusable, but the value of N needs to be clearly controlled within 100. For traversal, use hscan, sscan, zscan instead.

b. Disabling dangerous commands

Prohibit the use of keys, flushall, flushdb, config etc on live environment by renaming these commands to disable them. Please do not try these commands in the production environment and use the progressive processing of scan method.

c. Redis transactions are weak and not recommended

The transaction function of Redis is weak (does not support rollback), and the cluster version requires that the key of a transaction operation must be within a slot

3. Capacity Management

a. Capacity Planning

Before applying for a new Redis application, users should do a good assessment of capacity, QPS, and traffic (based on QPS, average value size), which can be assessed with reference to 集群申请 ; If the past or current Redis application supports a certain activity, when there is a new online demand, the above assessment needs to be done again so that the DBA can perform resource provisioning.

b. Scaling Up and Down

Redis sharding requires data rebalancing, and the time to scale up and down is proportional to the amount of data and the size of the cluster shard. For core services or services that heavily rely on Redis, it is recommended that users use 搬迁集群扩容

c. Limit to Memory Usage

Do not use more than 10G of memory for a single Redis shard. There is a performance risk if you exceed the 10G limit, so please clean up the data in advance or request a DBA to assist with shard expansion.

4. Client Usage

Strongly request that if lettuce directly queries from cluster, the cluster must be configured to refresh the topology, otherwise node migration or maintenance will lead to failure of queries.

a. Avoid having multiple applications using one Redis instance

> Split unrelated operations and make shared data a service

b. Use a database with connection pooling to effectively control connections while increasing efficiency

PHP

```php
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();
    // specific commands
    jedis.executeCommand()
} catch (Exception e) {
    logger.error("op key {} error: " + e.getMessage(), key, e);
        throw e;
} finally {
    // Note that the connection is not closed here. In JedisPool mode, Jedis
    // will be returned to the resource pool.
    if (jedis != null)
        jedis.close();
}
```

c. Recommended to add a fuse function to clients with high concurrency (e.g. netflix hystrix)

d. Set passwords for sensitive data

e. According to business type, set maxmemory eviction policy and expiry time

> Default policy is set as volatile-lru. Even after exceeding the maximum memory, the key is only evicted using the LRU algorithm if it has an experation value to ensure that non-expired data is not deleted, but there may be OOM problems.
>
> Other policies available:
>
> - allkeys-lru： Delete keys according to the LRU algorithm, regardless of whether the data has set the timeout property or not, until enough space is freed up.
> - allkeys-random： Delete all keys randomly until enough space is freed up.
> - volatile-random: Randomly delete keys with an expiration value until enough space is freed up.
> - volatile-ttl： Remove keys with an expiration value and the shortest remaining time-to-live (TTL) value. If there are none, follow the noeviction policy.
> - noeviction： Does not evict any data, rejects all writes and returns the client error message "(error) OOM command not allowed when used memory", meanwhile Redis only responds to read operations.

5. Architecture Design

Redis does not guarantee consistency and thus cannot be used as a DB. The DBA team does not promise that Redis can recover data according to a specified point in time; The architecture design for Redis must take into account recovery solutions for data misdeletion and other similar scenarios.