Redis优化经验

使用经验：

千万记住keys和smembers生产环境不要用， 有个网友的测试：

存取100万数据，keys100万数据，smembers 集合存放100万数据 一起执行，显示qps1757，cpu 使用11.0%，cpu马上明显飙高了。

List数据类型：

没有 List 个数限制，单个元素最大值为 512 MB，推荐 list的元素个数小于 8192, value 最大长度不超过 1 MB。

Set数据类型：

没有 set 个数限制，单个元素最大值为 512 MB，推荐 set 的元素个数小于 8192，value 最大长度不超过 1 MB。

Sorted Set：

没有 sorted set 个数限制，单个元素最大值为 512 MB，推荐 sorted set 的元素个数小于 8192， value 最大长度不超过 1 MB。

Hash 数据类型：

没有 field 个数限制，单个元素最大值为 512 MB，推荐元素个数小于 8192， value 最大长度不超过 1 MB。

DB 数限制：

每个实例支持 256 个 DB。

监控警报：

版未提供容量告警，需要用户到云监控中进行配置。配置方法请参见文档。建议设置好以下监控的报警：实例故障、实例主备切换、已使用连接百分比、操作失败数、已用容量百分比、写入带宽使用率、读取带宽使用率。

数据过期删除策略：

主动过期，系统后台会周期性的检测，发现已过期的key时，会将其删除。- 被动过期，当用户访问某个key时，如果该key已经过期，则将其删除。

空闲连接回收机制：

服务端不主动回收 Redis 空闲连接，由用户管理。

数据持久化策略：采用 AOF_FSYNC_EVERYSEC 方式，每秒 fysnc。

内存管理优化

Redis Hash是value内部为一个HashMap，如果该Map的成员数比较少，则会采用类似一维线性的紧凑格式来存储该Map, 即省去了大量指针的内存开销，这个参数控制对应在redis.conf配置文件中下面2项：

hash-max-zipmap-entries 64 hash-max-zipmap-value 512

当value这个Map内部不超过多少个成员时会采用线性紧凑格式存储，默认是64,即value内部有64个以下的成员就是使用线性紧凑存储，超过该值自动转成真正的HashMap。

hash-max-zipmap-value 含义是当 value这个Map内部的每个成员值长度不超过多少字节就会采用线性紧凑存储来节省空间。

以上2个条件任意一个条件超过设置值都会转换成真正的HashMap，也就不会再节省内存了，那么这个值是不是设置的越大越好呢，答案当然是否定的，HashMap的优势就是查找和操作的时间复杂度都是O(1)的，而放弃Hash采用一维存储则是O(n)的时间复杂度，如果

成员数量很少，则影响不大，否则会严重影响性能，所以要权衡好这个值的设置，总体上还是最根本的时间成本和空间成本上的权衡。

list-max-ziplist-value 64 list-max-ziplist-entries 512

list数据类型节点值大小小于多少字节会采用紧凑存储格式、list数据类型多少节点以下会采用去指针的紧凑存储格式。

内存预分配：

Redis内部实现没有对内存分配方面做过多的优化（对比Memcache），在一定程度上会存在内存碎片，不过大多数情况下这个不会成为Redis的性能瓶颈，不过如果在Redis内部存储的大部分数据是数值型的话，Redis内部采用了一个shared integer的 方式来省去分配内存的开销，即在系统启动时先分配一个从1~n 那么多个数值对象放在一个池子中，如果存储的数据恰好是这个数值范围内的数据，则直接从池子里取出该对象，并且通过引用计数的方式来共享，这样在系统存储 了大量数值下，也能一定程度上节省内存并且提高性能，这个参数值n的设置需要修改源代码中的一行宏定义

REDIS_SHARED_INTEGERS，该值 默认是10000，可以根据自己的需要进行修改，修改后重新编译就可以了。

持久化机制：

定时快照方式(snapshot)：

该持久化方式实际是在Redis内部一个定时器事件，每隔固定时间去检查当前数据发生的改变次数与时间是否满足配置的持久化触发的条件，如果满足则通 过操作系统fork调用来创建出一个子进程，这个子进程默认会与父进程共享相同的地址空间，这时就可以通过子进程来遍历整个内存来进行存储操作，而主进程 则仍然可以提供服务，当有写入时由操作系统按照内存页(page)为单位来进行copy-on-write保证父子进程之间不会互相影响。

该持久化的主要缺点是定时快照只是代表一段时间内的内存映像，所以系统重启会丢失上次快照与重启之间所有的数据。

基于语句追加方式(aof)：

aof方式实际类似mysql的基于语句的binlog方式，即每条会使Redis内存数据发生改变的命令都会追加到一个log文件中，也就是说这个log文件就是Redis的持久化数据。

aof的方式的主要缺点是追加log文件可能导致体积过大，当系统重启恢复数据时如果是aof的方式则加载数据会非常慢，几十G的数据可能需要几小时才能加载完，当然这个耗时并不是因为磁盘文件读取速度慢，而是由于读取的所有命令都要在内存中执行一遍。另外由于每条命令都要写log,所以使用aof的方式，Redis的读写性能也会有所下降。

可以考虑将数据保存到不同的Redis实例中，每个实例的内存大小在2G左右，避免将鸡蛋放到一个篮子里，既可以减少缓存失效给系统带来的影响，又可以加快数据恢复的速度，不过同时也给系统设计带来了一定的复杂性。

Redis持久化崩溃问题：

有Redis线上运维经验的人会发现Redis在物理内存使用比较多，但还没有超过实际物理内存总容量时就会发生不稳定甚至崩溃的 问题，有人认为是基于快照方式持久化的fork系统调用造成内存占用加倍而导致的，这种观点是不准确的，因为fork 调用的copy-on-write机制是基于操作系统页这个单位的，也就是只有有写入的脏页会被复制，但是一般你的系统不会在短时间内所有的页都发生了写 入而导致复制，那么是什么原因导致Redis崩溃的呢？

答案是Redis的持久化使用了Buffer IO造 成的，所谓Buffer IO是指Redis对持久化文件的写入和读取操作都会使用物理内存的Page Cache,而大多数数据库系统会使用Direct IO来绕过这层Page Cache并自行维护一个数据的Cache，而当Redis的持久化文件过大(尤其是快照文件)，并对其进行读写时，磁盘文件中的数据都会被加载到物理内 存中作为操作系统对该文件的一层Cache,而这层Cache的数据与Redis内存中管理的数据实际是重复存储的，虽然内核在物理内存紧张时会做 Page Cache的剔除工作，但内核很可能认为某块Page Cache更重要，而让你的进程开始Swap ,这时你的系统就会开始出现不稳定或者崩溃了。我们的经验是当你的Redis物理内存使用超过内存总容量的3/5时就会开始比较危险了。

总结：

1. 根据业务需要选择合适的数据类型，并为不同的应用场景设置相应的紧凑存储参数。
2. 当业务场景不需要数据持久化时，关闭所有的持久化方式可以获得最佳的性能以及最大的内存使用量。
3. 如果需要使用持久化，根据是否可以容忍重启丢失部分数据在快照方式与语句追加方式之间选择其一，不要使用虚拟内存以及diskstore方式。
4. 不要让你的Redis所在机器物理内存使用超过实际内存总量的3/5。

redis.conf中的maxmemory选项，该选项是告诉Redis当使用了多少物理内存后就开始拒绝后续的写入请求，该参数能很好的保护好你的Redis不会因为使用了过多的物理内存而导致swap,最终严重影响性能甚至崩溃。
redis.conf文件中 vm-enabled 为 no

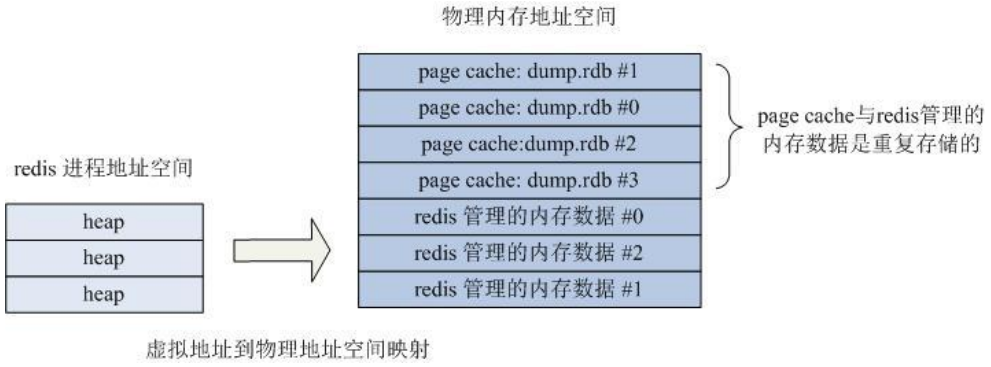
常用内存优化手段与参数
通过我们上面的一些实现上的分析可以看出redis实际上的内存管理成本非常高，即占用了过多的内存，作者对这点也非常清楚，所以提供了一系列的参数和手段来控制和节省内存，我们分别来讨论下。
首先最重要的一点是不要开启Redis的VM选项，即虚拟内存功能，这个本来是作为Redis存储超出物理内存数据的一种数据在内存与磁盘换入换出的一个持久化策略，但是其内存管理成本也非常的高，并且我们后续会分析此种持久化策略并不成熟，所以要关闭VM功能，请检查你的redis.conf文件中 vm-enabled 为 no。
其次最好设置下redis.conf中的maxmemory选项，该选项是告诉Redis当使用了多少物理内存后就开始拒绝后续的写入请求，该参数能很好的保护好你的Redis不会因为使用了过多的物理内存而导致swap,最终严重影响性能甚至崩溃。
另外Redis为不同数据类型分别提供了一组参数来控制内存使用，我们在前面详细分析过Redis Hash是value内部为一个HashMap，如果该Map的成员数比较少，则会采用类似一维线性的紧凑格式来存储该Map, 即省去了大量指针的内存开销，这个参数控制对应在redis.conf配置文件中下面2项：
hash-max-zipmap-entries 64
hash-max-zipmap-value 512
hash-max-zipmap-entries
含义是当value这个Map内部不超过多少个成员时会采用线性紧凑格式存储，默认是64,即value内部有64个以下的成员就是使用线性紧凑存储，超过该值自动转成真正的HashMap。
hash-max-zipmap-value 含义是当 value这个Map内部的每个成员值长度不超过多少字节就会采用线性紧凑存储来节省空间。
以上2个条件任意一个条件超过设置值都会转换成真正的HashMap，也就不会再节省内存了，那么这个值是不是设置的越大越好呢，答案当然是否定的，HashMap的优势就是查找

和操作的时间复杂度都是O(1)的，而放弃Hash采用一维存储则是O(n)的时间复杂度，如果
成员数量很少，则影响不大，否则会严重影响性能，所以要权衡好这个值的设置，总体上还是最根本的时间成本和空间成本上的权衡。
同样类似的参数还有：
list-max-ziplist-entries 512
说明：list数据类型多少节点以下会采用去指针的紧凑存储格式。
list-max-ziplist-value 64
说明：list数据类型节点值大小小于多少字节会采用紧凑存储格式。
set-max-intset-entries 512
说明：set数据类型内部数据如果全部是数值型，且包含多少节点以下会采用紧凑格式存储。
最后想说的是Redis内部实现没有对内存分配方面做过多的优化，在一定程度上会存在内存碎片，不过大多数情况下这个不会成为Redis的性能瓶颈，不过如果在Redis内部存储的大部分数据是数值型的话，Redis内部采用了一个shared integer的方式来省去分配内存的开销，即在系统启动时先分配一个从1~n 那么多个数值对象放在一个池子中，如果存储的数据恰好是这个数值范围内的数据，则直接从池子里取出该对象，并且通过引用计数的方式来共享，这样在系统存储了大量数值下，也能一定程度上节省内存并且提高性能，这个参数值n的设置需要修改源代码中的一行宏定义REDIS_SHARED_INTEGERS，该值默认是10000，可以根据自己的需要进行修改，修改后重新编译就可以了。
Redis的持久化机制
Redis由于支持非常丰富的内存数据结构类型，如何把这些复杂的内存组织方式持久化到磁盘上是一个难题，所以Redis的持久化方式与传统数据库的方式有比较多的差别，Redis一共支持四种持久化方式，分别是：

- 定时快照方式(snapshot)
- 基于语句追加文件的方式(aof)
- 虚拟内存(vm)
- Diskstore方式

在设计思路上，前两种是基于全部数据都在内存中，即小数据量下提供磁盘落地功能，而后两种方式则是作者在尝试存储数据超过物理内存时，即大数据量的数据存储，截止到本文，后两种持久化方式仍然是在实验阶段，并且vm方式基本已经被作者放弃，所以实际能在生产环境用的只有前两种，换句话说Redis目前还只能作为小数据量存储（全部数据能够加载在内存中），海量数据存储方面并不是Redis所擅长的领域。下面分别介绍下这几种持久化方式：
定时快照方式(snapshot)：
该持久化方式实际是在Redis内部一个定时器事件，每隔固定时间去检查当前数据发生的改变次数与时间是否满足配置的持久化触发的条件，如果满足则通过操作系统fork调用来创建出一个子进程，这个子进程默认会与父进程共享相同的地址空间，这时就可以通过子进

程来遍历整个内存来进行存储操作，而主进程则仍然可以提供服务，当有写入时由操作系统按照内存页(page)为单位来进行copy-on-write保证父子进程之间不会互相影响。
该持久化的主要缺点是定时快照只是代表一段时间内的内存映像，所以系统重启会丢失上次快照与重启之间所有的数据。
基于语句追加方式(aof)：
aof方式实际类似mysql的基于语句的binlog方式，即每条会使Redis内存数据发生改变的命令都会追加到一个log文件中，也就是说这个log文件就是Redis的持久化数据。
aof的方式的主要缺点是追加log文件可能导致体积过大，当系统重启恢复数据时如果是aof的方式则加载数据会非常慢，几十G的数据可能需要几小时才能加载完，当然这个耗时并不是因为磁盘文件读取速度慢，而是由于读取的所有命令都要在内存中执行一遍。另外由于每条命令都要写log,所以使用aof的方式，Redis的读写性能也会有所下降。
虚拟内存方式：
虚拟内存方式是Redis来进行用户空间的数据换入换出的一个策略，此种方式在实现的效果上比较差，主要问题是代码复杂，重启慢，复制慢等等，目前已经被作者放弃。
diskstore方式：
diskstore方式是作者放弃了虚拟内存方式后选择的一种新的实现方式，也就是传统的B-tree的方式，目前仍在实验阶段，后续是否可用我们可以拭目以待。
Redis持久化磁盘IO方式及其带来的问题
有Redis线上运维经验的人会发现Redis在物理内存使用比较多，但还没有超过实际物理内存总容量时就会发生不稳定甚至崩溃的问题，有人认为是基于快照方式持久化的fork系统调用造成内存占用加倍而导致的，这种观点是不准确的，因为fork 调用的copy-on-write机制是基于操作系统页这个单位的，也就是只有有写入的脏页会被复制，但是一般你的系统不会在短时间内所有的页都发生了写入而导致复制，那么是什么原因导致Redis崩溃的呢？
答案是Redis的持久化使用了Buffer IO造成的，所谓Buffer IO是指Redis对持久化文件的写入和读取操作都会使用物理内存的Page Cache,而大多数数据库系统会使用Direct IO来绕过这层Page Cache并自行维护一个数据的Cache，而当Redis的持久化文件过大(尤其是快照文件)，并对其进行读写时，磁盘文件中的数据都会被加载到物理内存中作为操作系统对该文件的一层Cache,而这层Cache的数据与Redis内存中管理的数据实际是重复存储的，虽然内核在物理内存紧张时会做Page Cache的剔除工作，但内核很可能认为某块Page Cache更重要，而让你的进程开始Swap ,这时你的系统就会开始出现不稳定或者崩溃了。我们的经验是当你的Redis物理内存使用超过内存总容量的3/5时就会开始比较危险了。
下图是Redis在读取或者写入快照文件dump.rdb后的内存数据图：



总结：

1. 根据业务需要选择合适的数据类型，并为不同的应用场景设置相应的紧凑存储参数。
2. 当业务场景不需要数据持久化时，关闭所有的持久化方式可以获得最佳的性能以及最大的内存使用量。
3. 如果需要使用持久化，根据是否可以容忍重启丢失部分数据在快照方式与语句追加方式之间选择其一，不要使用虚拟内存以及diskstore方式。
4. 不要让你的Redis所在机器物理内存使用超过实际内存总量的3/5。

==========

在使用Redis时，你需要注意以下几点：

1. 掌控储存在Redis中的所有键

数据库的主要功能是储存数据，但是对于开发者来说，因为应用程序需求或者数据使用方法的改变，忽略存储在数据库中的某些数据是非常正常的，在Redis中同样如此。你可能忽视期满某些键，也可能因为应用程序的某个模块弃用而忘掉这些数据。

无论哪种情况，Redis都存储了一些不再使用的数据，平白无故的占用了一些空间。Redis的弱结构数据模式让集中储存的内容很难被弄清，除非你为键使用一套非常成熟的命名法则。使用合适的命名方法会简化你的数据库管理，当你通过你的应用程序或者服务做键的命名空间时（通常情况下是使用冒号来划分键名），你就可以在数据迁移、转换或者删除时轻松的识别。

Redis另一个常见用例是作为热数据项作的第二数据存储，大部分的数据被保存在其他的数据库中，比如PostgreSQL或MongoDB。在这些用例中，当数据从主存储移除时，开发者经常会忘记删除Redis中对应的数据。这种存在跨数据存储的情况下，通常需要做级联删除，这种情况下，可以通过在Redis配置保存特定数据项的所有识别符来实现，从而保证数据在主数据库被删除后，系统会调用一个清理程序来删除所有相关副本和信息。

2. 控制所有键名的长度

在上文我们说过要使用合适的命名规则，并且添加前缀来识别数据走向，因此这一条看起来似乎与之违背。但是，请别忘记，Redis是个内存数据库，键越短你需要的空间就越少。理所当然，当数据库中拥有数百万或者数十亿键时，键名的长度将影响重大。

举个例子：在一个32位的Redis服务器上，如果储存一百万个键，每个值的长度是32-character，那么在使用6-character长度键名时，将会消耗大约96MB的空间，但是如果使用12-character长度的键名时，空间消耗则会提升至111MB左右。随着键的增多，15%的额外开销将产生重大的影响。

3. 使用合适的数据结构

不管是内存使用或者是性能，有的时候数据结构将产生很大的影响，下面是一些可以参考的最佳实践：

取代将数据存储为数千（或者数百万）独立的字符串，可以考虑使用哈希数据结构将相关数据进行分组。哈希表是非常有效率的，并且可以减少你的内存使用；同时，哈希还更有益于细节抽象和代码可读。

合适时候，使用list代替set。如果你不需要使用set特性，List在使用更少内存的情况下可以提供比set更快的速度。

Sorted sets是最昂贵的数据结构，不管是内存消耗还是基本操作的复杂性。如果你只是需要一个查询记录的途径，并不在意排序这样的属性，那么轻建议使用哈希表。

Redis中一个经常被忽视的功能就是bitmaps或者bitsets（V2.2之后）。Bitsets允许你在Redis值上执行多个bit-level操作，比如一些轻量级的分析。

4. 使用SCAN时别使用键

从Redis v2.8开始，SCAN命令已经可用，它允许使用游标从keyspace中检索键。对比KEYS命令，虽然SCAN无法一次性返回所有匹配结果，但是却规避了阻塞系统这个高风险，从而也让一些操作可以放在主节点上执行。

需要注意的是，SCAN 命令是一个基于游标的迭代器。SCAN 命令每次被调用之后， 都会向用户返回一个新的游标，用户在下次迭代时需要使用这个新游标作为 SCAN 命令的游标参数， 以此来延续之前的迭代过程。同时，使用SCAN，用户还可以使用keyname模式和count选项对命令进行调整。

SCAN相关命令还包括SSCAN 命令、HSCAN 命令和 ZSCAN 命令，分别用于集合、哈希键及有续集等。

5. 使用服务器端Lua脚本

在Redis使用过程中，Lua脚本的支持无疑给开发者提供一个非常友好的开发环境，从而大幅度解放用户的创造力。如果使用得当，Lua脚本可以给性能和资源消耗带来非常大的改善。取代将数据传送给CPU，脚本允许你在最接近数据的地方执行逻辑，从而减少网络延时和数据的冗余传输。

在Redis中，Lua一个非常经典的用例就是数据过滤或者将数据聚合到应用程序。通过将处理工作流封装到一个脚本中，你只需要调用它就可以在更短的时间内使用很少的资源来获取一个更小的答案。

专家提示：Lua确实非常棒，但是同样也存在一些问题，比如很难进行错误报告和处理。一个明智的方法就是使用Redis的Pub/Sub功能，并且让脚本通过专用信道来推送日志消息。然后建立一个订阅者进程，并进行相应的处理。