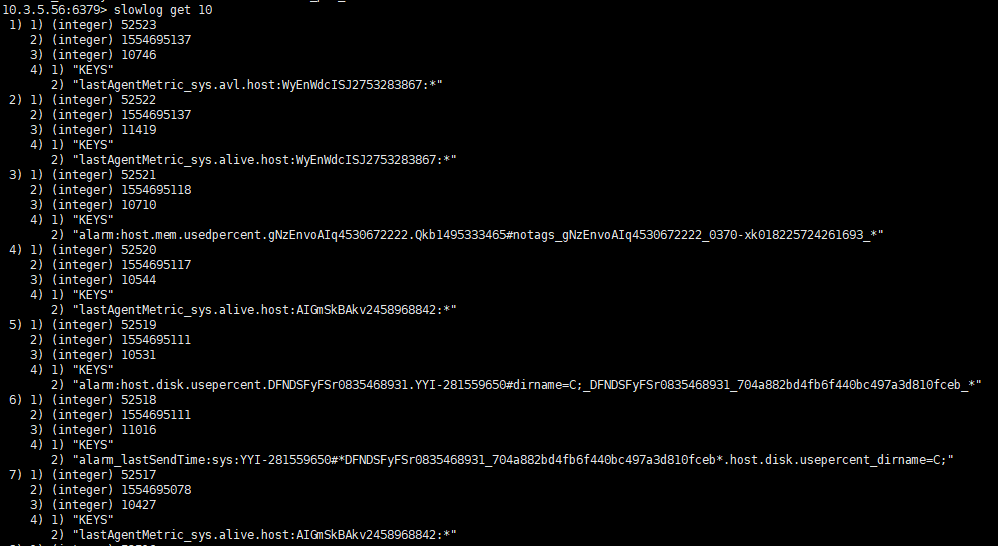
**Redis的KEYS效率问题解决方案**

——Keys与Scan效率测试及新报警解决方案

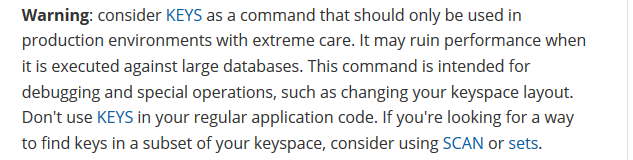
1、优化背景：

报警模块在3月28日上线时，由于修改了一个bug造成redis中key增加（大概增加10000条）。出现redis访问缓慢的问题，影响线上环境正常运行。经过查询慢日志，发现在报警匹配历史记录的过程中，大量使用了keys命令进行key的模糊匹配。大量的keys命令造成了redis阻塞。下图可见，keys的查询耗时基本都为10s左右：



**2、性能分析与测试：**

Redis官网关于keys的描述参见：



通过以上描述，我们可以看到，redis官方并不推荐在生产环境中使用keys。我们对实际数据进行模拟，分别对redis单点和redis集群进行keys及scan测试：

测试中机器配置为：

8核16g服务器用友云虚拟机

单点和集群（3台）。

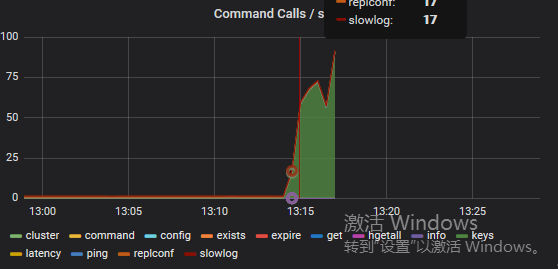
保证平均每个节点的数据量为50000/NODE。

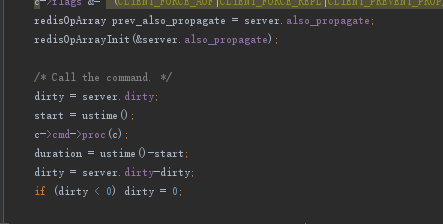
测试结果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试点 | 测试数量 | 线程数 | 调用次数 | 获取时间（ms） |
| 单点keys | 50000/NODE | 1 | 1 | 40 |
| 单点scan | 50000/NODE | 1 | 4 | 97 |
| 单点keys | 50000/NODE | 100 | 1 | 1277 |
| 单点scan | 50000/NODE | 100 | 4 | 2247 |
| 单点keys | 50000/NODE | 700 | 1 | 9485 |
| 单点scan | 50000/NODE | 700 | 4 | 10948 |
| 集群（3node）keys | 50000/NODE | 100 | 1 | 2054 |
| 集群（3node）scan | 50000/NODE | 100 | 4 | 6029 |
| 集群（3node）keys | 50000/NODE | 700 | 1 | 8836 |
| 集群（3node）scan | 50000/NODE | 700 | 4 | 19796 |

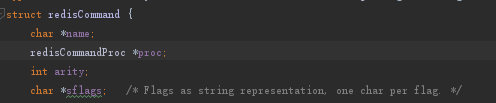
由上图可见，keys与scan在效率上并没有明显的差别，scan甚至比keys还要慢，但为什么官方推荐用scan呢？

下面的图是keys在执行测试过程中监控图曲线，从图中可以很清楚的看到当keys执行了一段时间后，reids出现了阻塞，发生无法响应的情况。暂停keys查询压测后，redis恢复正常。



出现以上情况，并不意外。众所周知，redis的command的处理都是单线程的，keys命令执行的时间复杂度为o（N）。下图为redis源码中的执行命令的call方法。

有代码可见其执行的为redisCommand结构中中的redisCommandProc指针定义的方法。由此可见该方法定义与执行中，并未新启动线程。而是单线程执行。如果执行时间较长容易造成redis的阻塞。



综上所述，Scan的方式并不能解决keys的效率问题，而是通过迭代执行的方式，让每次执行都只会返回少量元素， 而不会出现像 [KEYS](http://doc.redisfans.com/key/keys.html#keys) 命令、 [SMEMBERS](http://doc.redisfans.com/set/smembers.html#smembers) 命令带来阻塞的问题。同样，SCAN的方式时间复杂度也为O(n).所以从性能上其扫描的时间不会比keys的时间短，因为多次执行，反而会比keys的时间长。所以scan替代keys能避免卡顿问题，但效率同样低下。报警需要频繁的时候进行模糊匹配，10s左右的访问，无法满足效率需求。

3、解决方案：

**3.1方案提出：**

这个问题的核心的是减少scan扫描的keys的数量。结合以上场景，提出以下解决方案：

1、减少key的数量。

2、通过key中命名空间或hash路由将key的值限定在一定范围内进行模糊匹配。

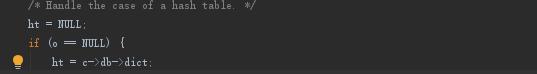
3、官方推荐的方式是sets，即将需要扫描的key尽量放入到sets中进行判断。

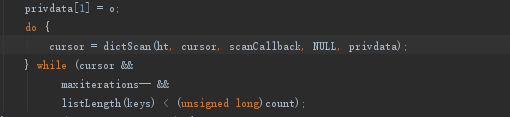
**3.2方案验证：**

方案1：结合实际业务需求，key的数量无法减少。而且，今后的key随着租户的增加会越来越多。

方案2：

通过查看redis源码，当进行scan操作时，返回的是整个数据字典，

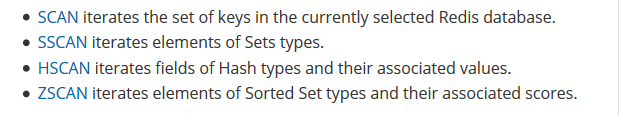


在进行单次扫描时，进行的为迭代调用，当满足cursor为0或扫描次数等于count\*10或获取key>count时跳出。源码上看，并未对key进行空间解析及hash解析。

方案3：官方推荐，可以采用这用方式。在处理单指标模糊匹配时可以满足，但在进行复核指标存储和判断时，多个指标混合在一个set中，为判断和维护匹配都带了难度，增加了代码复杂度。

**3.3最终方案：**

Scan有四种相关命令：



结合上边的方案，问题的核心还是减少扫描的数据，最好有类似于B+tree的结构。源码中既然没有自动解析命名空间的方式。结合业务，我们采用其他的方式变通的解决：将命名空间的定义扩展到数据结构中，通过key选择小范围的数据，再对数据进行模糊匹配，当我们把历史数据指标的缩小到租户维度或策略维度时，匹配的量就会变少。具体实现方式为：将原来的string结构修改为hash结构，在通过hscan方式进行模糊匹配。

如redis的key为：

|  |  |
| --- | --- |
| Key1 | Value |
| alarm:cpu.user.utilization.aEJkiQgsXS7941325404.Zbr1665209877#notags\_086075213aca5002005859914552cddb\_1554803667092 | 3.0 |
| alarm:${metric}.${tid}.${alarmConfid}#${tags}\_${metric.range}\_${sys.curTMillis} |  |

结合我们的数据结果将key拆解为hash方式。

|  |  |  |
| --- | --- | --- |
| Key1 | Key2 | Value |
| alarm: aEJkiQgsXS7941325404.Zbr1665209877 | cpu.user.utilization#notags\_086075213aca5002005859914552cddb\_1554803667092 | 3.0 |
| Alarm：${tid}.${alarmConfid} | ${metric}#${tags}\_${metric.range}\_${sys.curTMillis} |  |

**3.4再次测试：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试点 | 测试数量 | 线程数 | 调用次数 | 获取时间（ms） |
| 单点hscan | 50000/NODE | 100 | 1 | 22 |
| 单点hscan | 50000/NODE | 700 | 1 | 16 |
| 单点hscan | 150000/NODE | 100 | 1 | 26 |
| 单点hscan | 150000/NODE | 700 | 1 | 23 |
| 集群（3node）hscan | 50000/NODE | 100 | 1 | 21 |
| 集群（3node）hscan | 50000/NODE | 700 | 1 | 4 |
| 集群（3node）hscan | 150000/NODE | 100 | 1 | 24 |
| 集群（3node）hscan | 150000/NODE | 700 | 1 | 12 |

以上测试可知，获取时间已经由秒级下降为ms级。无论是线程的增加和key的增加的都不会对hscan的访问产生明显影响，查询数据只是在ms级有所波动甚至出现高并发比低并发快的情况。Redis选择数据的源码如下：



经过该处理后，响应的时间复杂度又由O(N)（N为redis所有的key，不判断是否所选key相关的），降为了O(1),并在key内循判断时间复杂度为O（N）,(N为hash主键key内部的值一般的值为10左右)。