二

33 实战：Redis 性能测试

为什么需要性能测试?

性能测试的使用场景有很多，例如以下几个：

1. 技术选型，比如测试 Memcached 和 Redis；
2. 对比单机 Redis 和集群 Redis 的吞吐量；
3. 评估不同类型的存储性能，例如集合和有序集合；
4. 对比开启持久化和关闭持久化的吞吐量；
5. 对比调优和未调优的吞吐量；
6. 对比不同 Redis 版本的吞吐量，作为是否升级的一个参考标准。

等等，诸如此类的情况，我们都需要进行性能测试。

性能测试的几种方式

既然性能测试使用场景那么多，那要怎么进行性能测试呢?

目前比较主流的性能测试分为两种：

1. 编写代码模拟并发进行性能测试；
2. 使用 redis-benchmark 进行测试。

因为自己编写代码进行性能测试的方式不够灵活，且很难短时间内模拟大量的并发数，所有作者并不建议使用这种方式。幸运的是 Redis 本身给我们提供了性能测试工具 redis-benchmark（Redis 基准测试），因此我们本文重点来介绍 redis-benchmark 的使用。

基准测试实战

redis-benchmark 位于 Redis 的 src 目录下，我们可以使用 `./redis-benchmark -h` 来查看

基准测试的使用，执行结果如下：

```
Usage: redis-benchmark [-h <host>] [-p <port>] [-c <clients>] [-n <requests>] [-k <

 -h <hostname>      Server hostname (default 127.0.0.1)
 -p <port>          Server port (default 6379)
 -s <socket>        Server socket (overrides host and port)
 -a <password>      Password for Redis Auth
 -c <clients>       Number of parallel connections (default 50)
 -n <requests>      Total number of requests (default 100000)
 -d <size>          Data size of SET/GET value in bytes (default 3)
 --dbnum <db>       SELECT the specified db number (default 0)
 -k <boolean>       1=keep alive 0=reconnect (default 1)
 -r <keyspacelen>   Use random keys for SET/GET/INCR, random values for SADD
  Using this option the benchmark will expand the string __rand_int__
  inside an argument with a 12 digits number in the specified range
  from 0 to keyspacelen-1. The substitution changes every time a command
  is executed. Default tests use this to hit random keys in the
  specified range.
 -P <numreq>        Pipeline <numreq> requests. Default 1 (no pipeline).
 -e                 If server replies with errors, show them on stdout.
                    (no more than 1 error per second is displayed)
 -q                 Quiet. Just show query/sec values
 --csv              Output in CSV format
 -l                 Loop. Run the tests forever
 -t <tests>         Only run the comma separated list of tests. The test
                    names are the same as the ones produced as output.
 -I                 Idle mode. Just open N idle connections and wait.
```

可以看出 redis-benchmark 支持以下选项：

- `-h <hostname>`：服务器的主机名（默认值为 127.0.0.1）。
- `-p <port>`：服务器的端口号（默认值为 6379）。
- `-s <socket>`：服务器的套接字（会覆盖主机名和端口号）。
- `-a <password>`：登录 Redis 时进行身份验证的密码。
- `-c <clients>`：并发的连接数量（默认值为 50）。
- `-n <requests>`：发出的请求总数（默认值为 100000）。
- `-d <size>`：SET/GET 命令所操作的值的数据大小，以字节为单位（默认值为 2）。
- `–dbnum <db>`：选择用于性能测试的数据库的编号（默认值为 0）。
- `-k <boolean>`：1 = 保持连接；0 = 重新连接（默认值为 1）。
- `-r <keyspacelen>`：SET/GET/INCR 命令使用随机键，SADD 命令使用随机值。通过这个选项，基准测试会将参数中的 `__rand_int__` 字符串替换为一个 12 位的整数，这个整数的取值范围从 0 到 keyspacelen-1。每次执行一条命令的时候，用于替换的整数

值都会改变。通过这个参数，默认的测试方案会在指定范围之内尝试命中随机键。

- `-P <numreq>`：使用管道机制处理 `<numreq>` 条 Redis 请求。默认值为 1（不使用管道机制）。
- `-q`：静默测试，只显示 QPS 的值。
- `-csv`：将测试结果输出为 CSV 格式的文件。
- `-l`：循环测试。基准测试会永远运行下去。
- `-t <tests>`：基准测试只会运行列表中用逗号分隔的命令。测试命令的名称和结果输出产生的名称相同。
- `-I`：空闲模式，只会打开 N 个空闲的连接，然后等待。

可以看出 redis-benchmark 带的功能还是比较全的。

基本使用

在安装 Redis 服务端的机器上，我们可以不带任何参数直接执行 `./redis-benchmark` 执行结果如下：

```
[@iZ2ze0nc5n41zomzyqtksmZ:src]$ ./redis-benchmark
====== PING_INLINE ======
  100000 requests completed in 1.26 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.81% <= 1 milliseconds
100.00% <= 2 milliseconds
79302.14 requests per second

====== PING_BULK ======
  100000 requests completed in 1.29 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.83% <= 1 milliseconds
100.00% <= 1 milliseconds
77459.34 requests per second

====== SET ======
  100000 requests completed in 1.26 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.80% <= 1 milliseconds
99.99% <= 2 milliseconds
```

```
100.00% <= 2 milliseconds
79239.30 requests per second

====== GET ======
  100000 requests completed in 1.19 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.72% <= 1 milliseconds
99.95% <= 15 milliseconds
100.00% <= 16 milliseconds
100.00% <= 16 milliseconds
84104.29 requests per second

====== INCR ======
  100000 requests completed in 1.17 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.86% <= 1 milliseconds
100.00% <= 1 milliseconds
85397.09 requests per second

====== LPUSH ======
  100000 requests completed in 1.22 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.79% <= 1 milliseconds
100.00% <= 1 milliseconds
82169.27 requests per second

====== RPUSH ======
  100000 requests completed in 1.22 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.71% <= 1 milliseconds
100.00% <= 1 milliseconds
81900.09 requests per second

====== LPOP ======
  100000 requests completed in 1.29 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.78% <= 1 milliseconds
99.95% <= 13 milliseconds
99.97% <= 14 milliseconds
100.00% <= 14 milliseconds
77399.38 requests per second
```

```
====== RPOP ======
  100000 requests completed in 1.25 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.82% <= 1 milliseconds
100.00% <= 1 milliseconds
80192.46 requests per second

====== SADD ======
  100000 requests completed in 1.25 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.74% <= 1 milliseconds
100.00% <= 1 milliseconds
80192.46 requests per second

====== HSET ======
  100000 requests completed in 1.21 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.86% <= 1 milliseconds
100.00% <= 1 milliseconds
82440.23 requests per second

====== SPOP ======
  100000 requests completed in 1.22 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.92% <= 1 milliseconds
100.00% <= 1 milliseconds
81699.35 requests per second

====== LPUSH (needed to benchmark LRANGE) ======
  100000 requests completed in 1.26 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.69% <= 1 milliseconds
99.95% <= 13 milliseconds
99.99% <= 14 milliseconds
100.00% <= 14 milliseconds
79176.56 requests per second

====== LRANGE_100 (first 100 elements) ======
  100000 requests completed in 1.25 seconds
  50 parallel clients
```

```
  3 bytes payload
  keep alive: 1

99.57% <= 1 milliseconds
99.98% <= 2 milliseconds
100.00% <= 2 milliseconds
80128.20 requests per second

====== LRANGE_300 (first 300 elements) ======
  100000 requests completed in 1.25 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.91% <= 1 milliseconds
100.00% <= 1 milliseconds
80064.05 requests per second

====== LRANGE_500 (first 450 elements) ======
  100000 requests completed in 1.30 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.78% <= 1 milliseconds
100.00% <= 1 milliseconds
76863.95 requests per second

====== LRANGE_600 (first 600 elements) ======
  100000 requests completed in 1.20 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.85% <= 1 milliseconds
100.00% <= 1 milliseconds
83263.95 requests per second

====== MSET (10 keys) ======
  100000 requests completed in 1.27 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

99.65% <= 1 milliseconds
100.00% <= 1 milliseconds
78740.16 requests per second
```

可以看出以上都是对常用的方法 Set、Get、Incr 等进行测试，基本能达到每秒 8W 的处理级别。

精简测试

我们可以使用 `./redis-benchmark -t set,get,incr -n 1000000 -q` 命令，来对 Redis 服务器进行精简测试，测试结果如下：

```
[@iZ2ze0nc5n41zomzyqtksmZ:src]$ ./redis-benchmark -t set,get,incr -n 1000000 -q
SET: 81726.05 requests per second
GET: 81466.40 requests per second
INCR: 82481.03 requests per second
```

可以看出以上测试展示的结果非常的精简，这是因为我们设置了 `-q` 参数，此选项的意思是设置输出结果为精简模式，其中 `-t` 表示指定测试指令， `-n` 设置每个指令测试 100w 次。

管道测试

本课程的前面章节介绍了 Pipeline（管道）的知识，它是用于客户端把命令批量发给服务器端执行的，以此来提高程序的整体执行效率，那接下来我们测试一下 Pipeline 的吞吐量能到达多少，执行命令如下：

```
[@iZ2ze0nc5n41zomzyqtksmZ:src]$ ./redis-benchmark -t set,get,incr -n 1000000 -q -P
SET: 628535.50 requests per second
GET: 654450.25 requests per second
INCR: 647249.19 requests per second
```

我们发现 Pipeline 的测试很快就执行完了，同样是每个指令执行 100w 次，可以看出 Pipeline 的性能几乎是普通命令的 8 倍， `-P 10` 表示每次执行 10 个 Redis 命令。

基准测试的影响元素

为什么每次执行 10 个 Redis 命令，Pipeline 的效率为什么达不到普通命令的 10 倍呢?

这是因为基准测试会受到很大外部因素的影响，例如以下几个：

1. 网络带宽和网络延迟可能是 Redis 操作最大的性能瓶颈，比如有 10w q/s，平均每个请求负责传输 8 KB 的字符，那我们需要的理论带宽是 7.6 Gbits/s，如果服务器配置的是 1 Gbits/s，那么一定会有很多信息在排队等候传输，因此运行效率可想而知，这也是很多 Redis 生产环境之所以效率不高的原因；
2. CPU 可能是 Redis 运行的另一个重要的影响因素，如果 CPU 的计算能力跟不上 Redis 要求的话，也会影响 Redis 的运行效率；
3. 如果 Redis 运行在虚拟设备上，性能也会受影响，因为普通操作在虚拟设备上会有额外的消耗；

4. 普通操作和批量操作（Pipeline）对 Redis 的吞吐量也有很大的影响。

小结

本文介绍了 Redis 自带的性能测试工具 redis-benchmark 也是 Redis 主流的性能测试工具，我们可以轻松模拟指定并发量和指定命令的测试条件，也可以模拟管道测试。测试的结果对于我们做技术选型、版本选择以及数据类型的选择上都有一定的指导意义，但需要注意 Redis 的吞吐量还受到其他因素的影响，例如带宽、CPU 等因素。

上一页 下一页