Redis-Doc

2022年8月23日 16:20

https://redis.io/docs/

开源(BSD)，内存 数据结构 存储仓库，作为 db，cache，message broker，streaming engine。

redis提供了 数据结构： string，hash，list，set，带范围查询的 sorted set， bitmap，hyperloglog，geospatial indexes，stream。

redis 有内置的 replication，lua脚本，LRU，transaction，不同等级的on-disk persistence，提供了 高可用 (通过 redis sentinel) 和 自动分区 (通过redis cluster)

你可以 执行 原子操作，在上面的 类型上。 如 append 到 string，hash中 值的增加，push 元素到 list，计算set的 交集，并集，差集， 从 sorted set 中获得 rank最高的 元素。

为了获得 高性能，redis 使用了 内存数据集。根据你的 用例，redis 可以通过定期 将 数据集 转储到 磁盘 或 追加 每个命令 到 基于 磁盘的 log 来 持久化你的 数据。
如果你只需要 一个 功能丰富的 网络 内存缓存，你也可以 禁用 持久性。

redis 支持 async replication，具有非常快的 非阻塞 同步 和 自动重连 以及 网络分区 上的 部分 重新同步。

---

https://redis.io/docs/data-types/

redis is a data structure server.
提供了 原生数据类型 帮助你 解决 各种问题， 从 cache 到 queuing 到 event processing.

核心：
String
List
Set
Hash
Sorted set
Stream
Geospatial index
Bitmap
Bitfield
HyperLogLog

扩展

要扩展功能，使用下面的：
编写自定义的 Lua语言的 服务器端function
编写自己的 redis 模块，通过 模块api 或 查看 已有的 模块。
使用 redis stack 提供的 json，querying，time series 等功能。

---

https://redis.io/docs/data-types/strings/

Redis Strings

存储 byte 序列，包括 文本，序列化后的对象，二进制数组。
经常用于 cache， 但也支持 其他功能，可以让你 实现 计数器 和 执行 bit操作。

保存和retrieve
```
> SET user:1 salvatore
OK
> GET user:1
"salvatore"
```

保存序列化后的json，并 让它在 100秒后 过期
```
> SET ticket:27 "\"{'username': 'priya', 'ticket_id': 321}\"" EX 100
```

实现计数器
```
> INCR views:page:2
(integer) 1
> INCRBY views:page:2 10
(integer) 11
```

限制：
默认下，redis 的 一个string 不能超过 512mb。。。。

命令
set 保存一个string值
setnx 保存string值，只有 key不存在的时候。。 对于 实现锁 是有用的。
get 获得 一 个 string值
mget 获得 多个 string值 在一个操作中。

incrby 原子增加（或通过负数减少）保存在 指定key 中的 计数器。
另一个 浮点数 计数器的 ： incrbyfloat

。。counter，保存的是 数值。但是 是 string 下的。。

要在string上执行 bit操作，查看 bitmap 数据类型。

性能
大多数string 操作是 O(1)。 注意 substr, getrange, setrange，这些可能是 O(n)。这些 随

机访问的 string命令 可能造成 性能问题，在处理 大string时。

备选
如果你将 结构化的数据 存储为 序列化字符串，你也需要考虑 hash 或 RedisJSON。

---
https://redis.io/docs/data-types/lists/

redis list

是 string 值的 linked list. 经常用于：
实现stack 和 queue
为 后台工作系统 构建 队列管理。

queue
```
> LPUSH work:queue:ids 101
(integer) 1
> LPUSH work:queue:ids 237
(integer) 2
> RPOP work:queue:ids
"101"
> RPOP work:queue:ids
"237"
```

stack
```
> LPUSH work:queue:ids 101
(integer) 1
> LPUSH work:queue:ids 237
(integer) 2
> LPOP work:queue:ids
"237"
> LPOP work:queue:ids
"101"
```

计算长度
```
> LLEN work:queue:ids
(integer) 0
```

原子：从一个list pop，并push 到另一个list中
```
> LPUSH board:todo:ids 101
(integer) 1
> LPUSH board:todo:ids 273
(integer) 2
```

```
> LMOVE board:todo:ids board:in-progress:ids LEFT LEFT
"273"
> LRANGE board:todo:ids 0 -1
1) "101"
> LRANGE board:in-progress:ids 0 -1
1) "273"
```

创建一个 capped(有封顶的) list, 长度不会超过 100， 你可以 在 lpush 之后使用 ltrim

```
> LPUSH notifications:user:1 "You've got mail!"
(integer) 1
> LTRIM notifications:user:1 0 99
OK
> LPUSH notifications:user:1 "Your package will be delivered at 12:01 today."
(integer) 2
> LTRIM notifications:user:1 0 99
OK
```

限制
list的最大长度是 2^32 - 1 个元素

命令
lpush 增加一个元素 到 list 头。 rpush 增加到 尾巴
lpop 移除并返回 list的 头。 rpop 移除并返回 尾巴
llen 返回 list 长度
lmove 原子移动，从一个list 到另一个list
ltrim 将list 减少到 指定 的元素范围。

阻塞命令
blpop 移除和返回 list 头。如果 list 空，命令阻塞，直到一个元素可用 或 超时。
blmove 原子移动，从一个list 到另一个list。 如果 source list 为空，则阻塞，直到 一个元素可用。

性能
list的 访问 头 或 尾巴 是 O(1)。 通常 操作list 中元素的 命令 是 O(n)，例如 lindex,linsert,lset

备选
考虑 Redis stream 作为一个 list 的 备选，当你需要 保存 和 处理 一系列 不确定的 event。

---

https://redis.io/docs/data-types/sets/

redis set

set是一个 无序集合，包含 唯一的 string。 你可以使用 set 来高效地：
追踪唯一的item
表示关系
执行常见的 集合运算，如交集，并集，差集。

为user123 和 456 保存 最喜欢的书的id
```
> SADD user:123:favorites 347
(integer) 1
> SADD user:123:favorites 561
(integer) 1
> SADD user:123:favorites 742
(integer) 1
> SADD user:456:favorites 561
(integer) 1
```

检测 user 123 是否喜欢 书 742 和 299
```
> SISMEMBER user:123:favorites 742
(integer) 1
> SISMEMBER user:123:favorites 299
(integer) 0
```

user 123 和 456 是否有相同的喜欢的书
```
> SINTER user:123:favorites user:456:favorites
1) "561"
```

user123 有多少本喜欢的书
```
> SCARD user:123:favorites
(integer) 3
```

限制
set的最大大小是 2^32-1

命令
sadd 添加一个新元素到 set
srem 移除指定元素
sismember 测试一个string 是否是 set中成员
sinter 返回 2个或多个 集合 共有的 元素集合
scard 返回 set的 size

性能
大部分set操作，包括 add,remove,check,是 O(1)
对于 十万以上的 数据集，运行 smembers 命令时 需要小心。这个 命令是 O(n) 的，在单个 response中 返回 整个集合。 作为替代方案，考虑 sscan， 它允许你 迭代地 检索 所有成员。

备选
在 大数据集 (或 streaming数据) 上进行 成员检测 会 占用 大量内存。如果你 担心 内存使用情况 并且 不需要 完美的精度， 考虑 bloom过滤器 或 Cuckoo过滤器。
set经常被用来作为 一种索引，如果你需要 索引 和 查询数据，请考虑使用 RediSearch 和 RedisJSON。

---
https://redis.io/docs/data-types/hashes/

redis hash

hash是 field-value对 集合 的 结构类型。你可以使用 hash 来表示 基本对象 和 存储 计数器分组 等。

保存基本用户信息
```
> HSET user:123 username martina firstName Martina lastName Elisa country GB
(integer) 4
> HGET user:123 username
"martina"
> HGETALL user:123
1) "username"
2) "martina"
3) "firstName"
4) "Martina"
5) "lastName"
6) "Elisa"
7) "country"
8) "GB"
```

保存 777设备的 ping，请求，错误 的计数器
```
> HINCRBY device:777:stats pings 1
(integer) 1
> HINCRBY device:777:stats pings 1
(integer) 2
> HINCRBY device:777:stats pings 1
(integer) 3
> HINCRBY device:777:stats errors 1
(integer) 1
> HINCRBY device:777:stats requests 1
(integer) 1
> HGET device:777:stats pings
"3"
> HMGET device:777:stats requests errors
1) "1"
2) "1"
```

命令
hset 设置 hash 的一个 或多个 field 的值
hget 返回 给定的 field 的值
hmget 返回 一个或多个 给定的 field 的值。
hincrby 增加 给定的 field 的值 by 提供的integer

性能
大部分都是O(1)
hkeys, hvals, hgetall 是 O(n) ,n是 filed-value 对的 数量

限制
每个hash 最多保存 2^32 - 1 个 field-value 对。 实际上，你的 hash 只收 部署 redis 的 VM 的 总内存的限制。

---
https://redis.io/docs/data-types/sorted-sets/

redis sorted set

是唯一string 的 集合，并且按照 它们的 score 排序。
当 多个 string 有 相同的 socre， 按照 字典顺序排序。
用例包括：
排行榜
限速器， 你可以使用 sorted set 来构造一个 sliding-window rate limiter 来 防止过多的 api 调用。

更新一个实时的 排行榜
```
> ZADD leaderboard:455 100 user:1
(integer) 1
> ZADD leaderboard:455 75 user:2
(integer) 1
> ZADD leaderboard:455 101 user:3
(integer) 1
> ZADD leaderboard:455 15 user:4
(integer) 1
> ZADD leaderboard:455 275 user:2
(integer) 0
```

获得最高的 3个 score
```
> ZRANGE leaderboard:455 0 4 REV WITHSCORES
1) "user:2"
2) "275"
3) "user:3"
4) "101"
5) "user:1"
6) "100"
```

7) "user:4"
8) "15"

。。这个是 distinct的?

user2的rank
> ZREVRANK leaderboard:455 user:2
(integer) 0

命令
zadd 增加一个新元素和它的score 到 sorted set。如果 元素已存在，则score被更新。
zrange 返回 给定范围内的 sorted set的 成员。
zrank 返回 成员的 rank，假设 是 升序的。
zrevrank 返回 成员的 rank，假设是 降序的。

性能
大部分是 O(log(n))， n是 成员的数量。
在运行 具有大返回值 (>=数万) 的 zrange 要小心，该命令的 时间复杂度是 O(log(n)+m)，其中m是 返回的结果树。

备选
sorted set 有时 用于索引 其他 redis 数据结构。如果你需要 索引 和 查询 你的数据，考虑 RediSearch，RedisJSON。

---
https://redis.io/docs/data-types/streams/

redis stream

数据结构，行为就像 只能append 的log。你可以使用 stream 来 实时 记录和同步 event。
用例：
Event sourcing 事件溯源，如 追踪用户操作。
Sensor monitoring 传感监控，例如 现场设备的读数
Notifications 通知，将每个用户的 通知记录 存储在 单独的 流中。

redis 为每个流entry 生成一个唯一的id。你可以使用这些id 稍后检索 关联的entry 或 读取和处理 流中的 所有 后续条目。

redis stream 支持 多种 修建策略 (以防止 无限增长) 和 多种 消费策略 (查看 xread,xreadgroup,xrange)

将多个 温度 添加到 流中
> XADD temperatures:us-ny:10007 * temp_f 87.2 pressure 29.69 humidity 46

```
"1658354918398-0"
> XADD temperatures:us-ny:10007 * temp_f 83.1 pressure 29.21 humidity 46.5
"1658354934941-0"
> XADD temperatures:us-ny:10007 * temp_f 81.9 pressure 28.37 humidity 43.7
"1658354957524-0"
```

查看id为xx 的 前2条 stream entry

```
> XRANGE temperatures:us-ny:10007 1658354934941-0 + COUNT 2
1) 1) "1658354934941-0"
   2) 1) "temp_f"
      2) "83.1"
      3) "pressure"
      4) "29.21"
      5) "humidity"
      6) "46.5"
2) 1) "1658354957524-0"
   2) 1) "temp_f"
      2) "81.9"
      3) "pressure"
      4) "28.37"
      5) "humidity"
      6) "43.7"
```

从尾巴开始，最多读取100个新的 流条目，如果没有写入条目，则最多 阻塞 300ms

```
> XREAD COUNT 100 BLOCK 300 STREAMS tempertures:us-ny:10007 $
(nil)
```

命令

xadd 增加新entry到stream

xread 读取1个或多个entry， 起始于 给定的 位置 并及时向前移动。

xrange 返回 2个指定的id 之间 entry 的 range

xlen stream的长度

性能

增加entry 到 stream 是 O(1).访问 任何 单个entry 是 O(n)，n是 id 的长度，由于 流id 通常很短并且 有固定长度，因为 这样有效地 减少了 恒定时间查找。

注意流 的实现 是基于 radix tree (基数树) 的

简而言之，redis stream 提供了 高效的插入 和 读取。

---

https://redis.io/docs/data-types/geospatial/

redis geospatial

geospatial 索引 可以让你 存储 坐标 并搜索它们。 此数据结构对于查找 给定半径 或 边框内

的 附近点 很有用。

假设你正在构建一个 移动app，可以让你找到 最近的 所有 电动汽车充电站。

增加充电站
```
> GEOADD locations:ca -122.27652 37.805186 station:1
(integer) 1
> GEOADD locations:ca -122.2674626 37.8062344 station:2
(integer) 1
> GEOADD locations:ca -122.2469854 37.8104049 station:3
(integer) 1
```

找到 1km 半径内的 所有位置，并返回到每个位置的 距离
```
> GEOSEARCH locations:ca FROMLONLAT -122.2612767 37.7936847 BYRADIUS 5 km WITHDIST
1) 1) "station:1"
   2) "1.8523"
2) 1) "station:2"
   2) "1.4979"
3) 1) "station:3"
   2) "2.2441"
```
。。这个应该是 5km吧？

命令

geoadd 增加一个 位置 到 给定的 地理坐标 (注意，经度 在维度 之前)
geosearch 返回 给定半径 或 边界 内的 位置。

---

https://redis.io/docs/data-types/hyperloglogs/

redis hyperloglog

是一种 estimate the cardinality of a set(估计集合基数) 的数据结构。作为一种概率数据结构，它 以 准确性 换取 空间

使用12kb时，提供 0.81% 的标注误差

增加item 到 HyperLogLog
```
> PFADD members 123
(integer) 1
> PFADD members 500
(integer) 1
> PFADD members 12
(integer) 1
```

估计集合中的 成员数量
```
> PFCOUNT members
```

(integer) 3

命令
pfadd 增加一个item 到 HyperLogLog
pfcount 返回 集合中 item数量的 估计
pfmerge 合并2个或多个 HyperLogLog

性能
pfadd，pfcount 是 固定时间和空间。
pfmerge 是 O(n)，n是 sketches(草图) 的数量

限制
可以估计 具有最多 2^64 个成员的 集合。

---
https://redis.io/docs/data-types/bitmaps/

redis bitmap

是string 数据类型的 扩展，可以让你 treat 一个 string 就像 是一个 bit vector。你可以对1个或多个 string 进行 bit操作。
用例：
当集合中的成员 对应于 0-N 时，进行 有效的 集合表示。
权限，一个bit代表一个 权限。类似 文件系统存储权限。

加入你有1000个传感器，标记为0-999，你想快速 确定 某个传感器 是否在 1小时内 对 服务器进行了 ping操作。
你可以用bitmap 来表示 这种情况， bitmap 的key 引用了 当前时间。

在2024年1月1号 0点0分， 传感器123 ping 了服务器
> SETBIT pings:2024-01-01-00:00 123 1
(integer) 0

是否ping 了?
> GETBIT pings:2024-01-01-00:00 123
1

命令
setbit 设置给定的偏移量的 bit 为 0或1
getbit 返回给定offset的 bit值
bitop 对1个或多个 string执行 bit 操作

性能
setbit，getbit 是 O(1)
bitop 是 O(n)， n是 被操作数中 最长的 string 的 长度。

---
https://redis.io/docs/data-types/bitfields/

redis bitfield

让你 set,increment,get 任意位长度的 整数值。如，你可以 对 无符号1-bit integer 到 有符号的63-bit integer 的 任何内容进行操作。

这些值 通过 二进制编码的 redis string 进行保存。 bitfield 支持 原子 读，写，增加，使得 它是一个 管理 counter 和 类似 数字 值 的 好的选择。

假设你正在 跟踪 在线游侠中的 活动。 你希望为 每个玩家 维护2个关键指标： 金币总量 和 杀死的怪物的总数。 你使用32位来保存 counter

你可以 使用 每个玩家 一个 bitfield 来 表示 这些 计数器。

新玩家 以 1000金币 开始 (counter in offset 0)
```
> BITFIELD player:1:stats SET u32 #0 1000
1) (integer) 0
```

杀怪后，获得 50金币，并且 怪物数+1 (offset 1)
```
> BITFIELD player:1:stats INCRBY u32 #0 50 INCRBY u32 #1 1
1) (integer) 1050
2) (integer) 1
```

花999买东西
```
> BITFIELD player:1:stats INCRBY u32 #0 -999
1) (integer) 51
```

读取玩家状态
```
> BITFIELD player:1:stats GET u32 #0 GET u32 #1
1) (integer) 51
2) (integer) 1
```

命令
bitfield 原子set，增加，read 一个或多个值
bitfield_ro 是 bitfield 的 只读的变体。

性能
bitfield 是 O(n)， n是 counter 数量。

---

https://redis.io/docs/manual/eviction/

noeviction: New values aren’t saved when memory limit is reached. When a database uses replication, this applies to the primary database

allkeys-lru: Keeps most recently used keys; removes least recently used (LRU) keys

allkeys-lfu: Keeps frequently used keys; removes least frequently used (LFU) keys

volatile-lru: Removes least recently used keys with the expire field set to true.

volatile-lfu: Removes least frequently used keys with the expire field set to true.

allkeys-random: Randomly removes keys to make space for the new data added.

volatile-random: Randomly removes keys with expire field set to true.

volatile-ttl: Removes keys with expire field set to true and the shortest remaining time-to-live (TTL) value.

The policies volatile-lru, volatile-lfu, volatile-random, and volatile-ttl behave like noeviction if there are no keys to evict matching the prerequisites.

Use the allkeys-lru policy when you expect a power-law distribution in the popularity of your requests. That is, you expect a subset of elements will be accessed far more often than the rest. This is a good pick if you are unsure.

Use the allkeys-random if you have a cyclic access where all the keys are scanned continuously, or when you expect the distribution to be uniform.

Use the volatile-ttl if you want to be able to provide hints to Redis about what are good candidate for expiration by using different TTL values when you create your cache objects.

The volatile-lru and volatile-random policies are mainly useful when you want to use a single instance for both caching and to have a set of persistent keys. However it is usually a better idea to run two Redis instances to solve such a problem.

It is also worth noting that setting an expire value to a key costs memory, so using a policy like allkeys-lru is more memory efficient since there is no need for an expire configuration for the key to be evicted under memory pressure.

What is important about the Redis LRU algorithm is that you are able to tune the precision of the algorithm by changing the number of samples to check for every eviction. This parameter is controlled by the following configuration directive:
maxmemory-samples 5

---
https://redis.io/docs/manual/sentinel/

High availablity with redis sentinel

。。太多了。。

---
https://redis.io/docs/manual/persistence/

redis persistence

RDB, redis database
以 指定时间间隔 执行 数据集的 时间点快照
AOS, append only file
把 server 收到的 所有 写入操作 都 log下来， server启动时 执行log中的命令，恢复数据。命令使用与 redis 协议本身 相同的 格式 ，以 append 的方式 记录日志。当日志变得太大时，redis 能够在后台重写 log(Redis is able to rewrite the log in the background when it gets too big.)。
。。。？ 重写日志？ 那以前的数据呢？ 难道不是 从 创建redis 到 now 的所有 命令？，当然可以筛选掉，毕竟有些del掉了。 但是。。

No persistence,

RDB + AOF,
可以在同一个实例中 结合AOF 和 RDB。 注意，这种情况下，当 redis 重启时，AOF文件将 用于 重建原始数据集，因为它保证是最完整的。
。。？ 还是用 AOF重建，那RDB 有什么用？

RDB优势
是一个时间点的redis中数据，非常紧凑的 单文件。例如，你可能希望 最近24小时内 每小时归档一次RDB文件，并在30天内每天保存一个 RDB文件。
RDB非常适合 灾难恢复，它是一个 可以传输到 远程数据中心 或 Amazon S3 的 压缩文件。
RDB最大限度地提高了 redis 性能，因为 redis 父进程 为了持久化 唯一需要做的就是 派生一个 子进程，让子进程进行 持久化。 父进程永远不会 执行 磁盘 IO 或 类似操作。
RDB 比 AOF，在恢复大数据集时 更快。
在副本上，RDB 支持 重启 和 故障转以后的 部分重新同步。

RDB劣势
如果你需要在 redis 停止工作时 (如断电) 将 数据丢失的 可能性降低到最小，那么 RDB 并不好。 你可以配置生成RDB的 不同保存点 (如，5分钟 和 100次写入后，你可以有多个 保存点)。但是，你通常 每5分钟 或更长时间 创建一次 RDB 快照。所以一旦 redis 没有正确关闭的情况下 停止工作，你应该准备好 丢失 最近几分钟的 数据。

RDB 需要经常 fork 以便 使用 子进程在磁盘上 持久化。如果 数据集 很大，fork 可能很耗时，并且 如果数据集很大 且 cpu性能不好，可能会导致redis 停止工作 几毫秒 - 1秒。AOF也需要fork，但是 频率更低，而且 你可以 调整重写日志的频率， 而不需要 对持久性有任何的 考虑。

AOF优势
更具持久性：你可以有 不同的额fsync 策略： 不fsync，每秒fsync，每次查询时fsync。使用每秒fsync的默认策略，写入性能依然很好。fsync 是后台线程执行的，当没有 fsync进行时，主线程 将执行写入，因此 你只会丢失 1秒的写入(fsync is performed using a background thread and the main thread will try hard to perform writes when no fsync is in progress, so you can only lose one second worth of writes.)。
。。。？fsync， 和后面的
AOF 日志是 只append 的log。所以不会出现 寻道，问题。即使 log的最后 是 写入了 一半的命令 (可能磁盘慢)， redis-check-aof 工具依然能 修复它。
redis可以 后台自动 rewrite AOF 当 AOF变得太大时。 rewrite 是 完全安全的，因为当redis继续附加操作 到 旧文件时，一个完全新的，并且 创建当前数据集 所需 操作最少的文件 会生成，一旦 第二个文件准备好，redis 就会切换，并追加到 新的文件中。
AOF以易于理解和解析的格式 依次 包含 所有操作的 日志。你可以轻松导出 AOF文件。例如，即使你不小心 使用 flushall 命令刷新了所有内容，只要在此 期间没有执行 过 rewrite log，你可以 停止服务器，删除 最新命令，并重新启动 来 恢复 数据。
。。。。rewrite 后的 文件依然很大，然后 + 2个命令后 就达到警戒值，岂不是 又要 rewrite。。

AOF劣势
相同的数据，AOF文件比RDB更大
根据具体的fsync策略，AOF可能比RDB 慢，一般来说， 将 fsync 设置为 每秒，性能依然非常高，在 禁用 fsync的清下， 即使高负载下，它也和 RDB 一样快。 只有RDB 能在 高写入的情况下 提供 最大延迟的 更多保证。
。。fsync是什么？集群间 同步？，不就是 普通的 写入策略，redis 写入到 buffer中， 然后buffer 到 磁盘 就是 fsync 的策略。 no的话是 OS 调度刷盘，性能最好。

版本<7.0 的AOF 劣势
如果在重写期间 有对 数据库的 写入，AOF 可能会使用 大量内存 (这些被缓冲在 内存中的数据 最后被写入 新的 AOF)
。。。最新版应该是 还是写旧的文件。
重写期间的所有 写入命令 都会写入磁盘 2次。
。。。 ？ 不是 缓冲了？。。这个是 现在的 劣势吧。现在会 写旧文件，然后被复制到新文件。
redis 可以在重写结束时 冻结 写入 并将这些写入命令 同步到 新的 AOF文件
。。。？不是劣势吧？？

使用哪个？
你应该 同时使用这2种，如果你想要 和 PostgreSQL 可以提供的 数据安全程度 相当的话。
如果你非常关心数据，但是 可以忍受 几分钟的数据的丢失，那么 单独使用 RDB
很多用户 单独使用 AOF，但 我们不鼓励这样做， 因为 RDB备份 是一个 更好的主意，来 数据库备份，更快的启动，避免AOF引擎的bug事件。

下面介绍 这2种 持久化模型的 更多细节。

。。。

---
https://redis.io/docs/manual/pipelining/

redis pipelining

如何通过 批处理 redis命令 来 优化 往返时间

pipelining 是一种技术 来 优化性能， 通过 一次性发出多个命令 而不等待 每个单独命令的 response。 大多数redis client 都支持 流水线(pipelining)。

本文档描述了 流水线旨在解决的问题 及 流水线在redis中 工作原理。

request/response protocols and round-trip time (RTT)
redis 是tcp server，它使用 client-server 模型 和 所谓的 请求/响应 协议。

这意味着 通常 一个请求 要经过下面的 步骤：

1. 客户端 发送 query 到 server，然后 从socket中读取(通常是 阻塞方式) 服务器的响应。
2. server 处理命令，发送 响应 给 客户端。

网络有慢有快，但是 肯定有消耗。这个网络上消耗的时间 就是 RTT(round trip time)。

如果使用了 loopback 接口，那么RTT 会短很多，通常是 亚毫秒， 但是 如果 执行多次 写入，也会消耗很多时间。

redis pipelining
一个 request/response 服务器 可以实现为： 即使客户端没有读取 旧响应，它也可以处理新请求。 这样就可以向 服务器发送多个命令，而无需等待，然后最后 统一读取 响应。

这就是流水线，是一种 广泛使用了 几十年的 技术。 例如，许多 pop3 协议 实现 已经支持这种功能，从而 大大加快了 从服务器下载 新电子邮件的 过程。

redis 从早期 就支持 流水线，下面是一个 原始的 netcat 的例子：

```
$ (printf "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
+PONG
+PONG
+PONG
```

RTT的消耗 不是 每次调用的， 而是 3个命令 一次 RTT 消耗。

当客户端使用 pipelining 发送命令时，服务器 将 被迫使用 内存 对 回复 进行排队。所以，如果你需要 通过 流水线 发送大量命令时， 最好分批发送，每次包含 合理数量的 命令，例如10k 个命令，读取回复，然后 再发10k个命令。 速度差不多，但是 服务器的 内存只需要对 10k个命令 入队。

这不仅仅是 RTT 的问题
pipelining 不仅仅是一种 减少 往返时间 的方法，它实际上 大大提高了 redis 的吞吐量。这是因为 在不使用 流水线的情况下，从 访问数据结构 和 产生回复 的角度来看，为每个命令提供服务 非常便宜，但是 从 套接字IO 的角度来看，它的成本非常高。这涉及调用 read 和 write 的系统调用，这意味着 从 用户空间 切换到 内核空间。 上下文切换是一个巨大的速度损失。

流水线时，通常使用 单个 read() 系统调用 读取 许多命令，使用 单个write()系统调用 传递多个 回复。
因此，吞吐量 随着 流水线的 延迟 几乎是 线性增长，最终达到 不使用 pipelining 的 10 倍。

Pipelining vs Scripting

redis 2.6 开始有 redis scripting, 可以使用 脚本 来更有效地解决 pipelining 的用例。脚本的一大优势是 它能够 以 最小的 延迟 读 和写数据，使得 读取，计算，写入 等操作非常快。

有时，应用程序 可能还希望 在 pipeline 中 发送 eval 或 evalsha 命令。这完全是可能的，redis 使用 script load 命令 明确支持它 (它可以保证 调用 evalsha 而没有 失败的风险)

---

https://redis.io/docs/manual/pubsub/

Redis Pub/Sub

subscribe, unsubscribe, puslish 实现了 publish/subscribe 规范。
sender 不会 通过 编码 为 它们的消息 指定 特定的 接受者。相反， 发布的消息 被 传入 channel，不需要知道 订阅者。
订阅者 订阅 一个或多个channel， 接收消息，不管 发布者是谁。
发布者和订阅者 这种解耦 可以实现 更大 的可扩展性 和 更动态 的 网络拓扑。

例如，为了订阅 频道 foo 和 bar， 客户端发出 subscribe 来订阅：
SUBSCRIBE foo bar

其他客户端 发送到 这些channel 的消息 将由 redis 推送 到 所有 订阅的 客户端。

订阅channel 的 client 不应该发出命令，尽管 它可以订阅和 取消订阅 其他channel。 订阅和 取消订阅的 回复 以消息的形式发送， 以便客户端 可以读取 连贯的 消息流，其中第一个元素指明 消息的类型。 订阅客户端上下文中 允许的 命令是 SUBSCRIBE, SSUBSCRIBE, SUNSUBSCRIBE, PSUBSCRIBE, UNSUBSCRIBE, PUNSUBSCRIBE, PING, RESET, and QUIT.

。。。

还有 复制， 伸缩， security，事务，等。

Reference， Redis Stack 没有记录。
里面有 优化。debug。协议。等。

==========

https://redis.io/commands/