云数据库 Redis

开发准则

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

开发准则
- 命名规则
- 基本使用准则
- Key 与 Value 设计原则
- 命令使用准则
- 客户端程序设计准则
- 连接池配置
  - Redigo 连接池使用建议
  - Jedis 连接池代码示例

开发准则
命名规则

最近更新时间：2023-05-11 16:58:59

实例命令规则

Redis 实例按照如下规则进行命名：{环境}-{分公司}-{项目组编号}-{业务名称}-{地域}-{专区}-{序号}

- 环境：prd 或 dev。
- 分公司：所属分公司的名称，例如：cx（财险）。
- 项目组编号：将每个项目组进行编号，例如：p001。
- 业务名称：使用 Redis 的业务名称，例如：abc。
- 地域：实例所属的地域名称缩写，例如：gz（广州）。
- 专区：实例所属专区的名称缩写，例如：gzcx01。
- 序号：Redis 实例的序号，假定为第一个实例。

根据上述情况，可以将实例命名为：prd-cx-p001-abc-gz-gzcx01-1。

Key 命名规则

Key 命名以业务名或数据库名为前缀(防止 key 冲突)，用冒号分隔。建议命令规则为：业务名:数据库名称:数据库表名称:数据的 ID

- 业务名：业务系统的缩写，例如：cx（财险）。
- 数据库名称：数据库的名称，例如：cxdb。
- 数据库表名称：数据库中数据所在表的名称，例如：user。
- 数据的 ID：数据表中的 ID 号，或者使用表的主键字段，例如用户表中可以使用 uid。

根据上述情况，可以将 Key 命名为：cx:cxdb:user:000110011。

基本使用准则

最近更新时间：2023-07-14 11:28:51

缓存定位准则

Redis 仅作为缓存使用。因为 Redis 本身的特性，其所有数据都存储在内存中，所以访问速度快。但如果作为持久化数据库存储数据，由于内存在断电后无法持久化保存数据的原因，有可能会导致数据丢失。

不唯一数据源准则

因为 Redis 是作为缓存使用，所以有一定的几率会命中数据失败，所以不能作为唯一的数据来源使用；在调用 Redis 发生异常后，需要查询后台数据库。

Key 淘汰准则

根据自身业务类型，设置合适的最大内存淘汰策略 maxmemory-policy。默认策略是 noeviction，即不删除键。在内存占满后会出现 OOM 问题，所以建议创建好实例后修改淘汰策略，减少 OOM 问题的出现。

可配置的内存淘汰策略

设置 Redis 内存缓存满后，数据的淘汰策略 maxmemory-policy，可在如下策略中进行选择。如何配置参数，请参见 管理实例参数 。

> **说明：**
> - LRU（Least Recently Used）表示最近最少使用，LRU 算法会记录每个键最近被访问的时间，并在需要淘汰键时优先淘汰最近最少使用的键。
> - TTL（Time To Live）表示设置过期时间。
> - LFU（ Least Frequently Used）即最不经常使用，LFU 算法会记录每个键被访问的次数，并在需要淘汰键时优先淘汰访问次数最少的键。

- allkeys-lru：根据 LRU 算法删除键，不管数据是否设置超时属性，优先淘汰最近最少使用的键，直到腾出足够空间为止。
- allkeys-random：会随机淘汰一些键。
- volatile-lru：根据 LRU 算法删除过期键，优先淘汰设置了过期时间（TTL）的键中最近最少使用的键。
- volatile-random：随机淘汰已过期 Key。
- volatile-ttl：根据键值对象的 TTL 属性，会优先淘汰设置了过期时间的键中 TTL 值较小的键。如果没有，回退到 noeviction 策略。
- volatile-lfu：优先淘汰设置了过期时间（TTL）的键中最不经常使用（LFU）的键。
- allkeys-lfu：优先淘汰最不经常使用（LFU）的键，与 volatile-lfu 不同，allkeys-lfu 策略会淘汰所有键，而不仅是设置了过期时间（TTL）的键。

- noeviction：不会删除任何数据，拒绝所有写入操作，并返回客户端错误信息 `"(error) OOM command not allowed whenused memory"`，此时 Redis 只响应读操作。

内存淘汰策略建议

- 当 Redis 作为缓存使用的时候，推荐使用 allkeys-lru 淘汰策略。该策略会将使用频率最低的 Key 淘汰。默认情况下，使用频率最低则后期命中的概率也最低，所以将其淘汰。
- 当 Redis 作为半缓存半持久化使用时，可以使用 volatile-lru。但因为 Redis 本身不建议保存持久化数据，所以只作为备选方案。

Key 与 Value 设计原则

最近更新时间：2023-05-11 16:56:18

Key 设计原则

Redis Key 命名需具有可读性及可管理性，不建议使用含义不清的 Key 以及特别长的 Key 名。

- 简洁性：保证语义的前提下，可以适当缩短 key 的长度，当 key 较多时，key 占用的内存空间也不容忽视，例如：cx:cxdb:cxdb_user_info:000110011可简化为 cx:cxdb:user:000110011。
- 命名规则：以英文字母开头，命名中只能出现大小写字母、数字、竖线、下划线、英文点号(.)和英文半角冒号(:)。
- 按照语义分割：不同业务逻辑含义使用英文半角冒号(:)分割，同一业务逻辑含义段的单词之间使用英文半角点号 (.)分割，用来表示一个完整的语义。
- 可读性：key 名称以 key 所代表的 value 类型结尾，以提高可读性。例如：user:basic.info:userid:string。
- 不使用过大的 key 名称：key 名称过大也会占用一定的内存空间。
- 禁止包含特殊字符：例如：\、*、?、{}、[]、()、空格、单双引号和转义字符等，如果 key 中存在特殊字符，可能会导致 key 无法检索或检索失败。

Key 生命周期准则

建议使用 expire 设置过期时间，控制 Key 的生命周期。例如：

```
> set cx:cxdb:user:000110011 xiaoming
> expire cx:cxdb:user:000110011 3600  # 设置 Key 一小时后过期
```

- 如果条件允许可以打散过期时间，防止集中过期。
- 对于没有设置过期的数据，重点关注 idletime。在 idletime 非常大时进行清理。例如，执行

  ```
  > object idletime cx:cxdb:user:000110011
  ```

  回显信息如下：

  ```
  :(integer) 150039   # 这里表示key有150039秒未被访问过
  ```
- 当一个 Key 有1个月以上未被访问过，则可以认定为冷数据，并进行清理。

Value 设计原则

拒绝 Big Key

大 Key 具体表现为 Redis 中的 Key 对应的 Value 很大，占用 Redis 空间比较大，本质上是大 Value 问题。对于 Redis 中不同的数据结构类型，常见示例如下所示：

- 对于 String 类型的 Value 值，值超过10MB（数据值太大）。
- 对于 Set 类型的 Value 值，含有的成员数量为10000个（成员数量多）。
- 对于 List 类型的 Value 值，含有的成员数量为10000个（成员数量多）。

- 对于 Hash 格式的 Value 值，含有的成员数量1000个，但所有成员变量的总 Value 值大小为1000MB（成员总的体积过大）。

Big key 很容易造成慢查询，阻塞其他的请求。同时，也会对网卡造成负担。为防止产生大 Key，设计 Value 时，建议参考如下建议：

- 建议 String 类型控制在10KB以内；hash、list、set、zset 元素个数不要超过5000。
- 若非必须，不要使用 del 删除大 Key。
- 对于非字符串的大 Key，建议使用 hscan、sscan、zscan 渐进式删除。
- 防止大 Key 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。

合理选择数据类型

- Redis 提供了多种不同的数据库类型，包括字符串、哈希表、列表、集合和有序集合等。选择合适的数据库类型可以提高 Redis 的性能和可靠性。
  - 字符串类型：适用于存储简单的字符串数据，例如配置信息、计数器等。如果需要存储二进制数据，可以使用 Redis 的二进制安全字符串类型。
  - 哈希表类型：适用于存储多个字段和值的数据，例如用户信息、商品信息等。哈希表可以节省内存空间，并且可以方便地进行批量操作。
  - 列表类型：适用于存储有序的元素集合，例如消息队列、任务列表等。列表可以在两端进行插入和删除操作，并且可以使用 Redis 提供的多种操作命令来操作列表。
  - 集合类型：适用于存储无序的元素集合，例如标签列表、好友列表等。集合可以进行并集、交集、差集等操作，并且可以使用 Redis 提供的多种操作命令来操作集合。
  - 有序集合类型：适用于存储有序的元素集合，例如排行榜、投票列表等。有序集合可以按照分值进行排序，并且可以使用 Redis 提供的多种操作命令来操作有序集合。
- 合理控制和使用数据结构内存编码优化配置，例如 ziplist 是一种特殊的数据结构，它可以将小型列表、哈希表和有序集合存储在一个连续的内存块中，从而节省了内存空间。但由于 ziplist 没有索引，因此在对 ziplist 进行查找、插入或删除操作时，需要进行线性扫描，这可能会导致性能下降。在实际应用中，应该根据具体情况来决定是否使用 ziplist。如果数据量较小且需要频繁进行遍历操作，那么使用 ziplist 可能是一个不错的选择。但是，如果数据量较大且需要频繁进行插入、删除或查找操作，那么使用 ziplist 可能会影响性能，应该考虑使用其他数据结构来代替。
- 此外，如果一个 Key 有多个属性，可以考虑使用 HashMap 类型来代替 String 类型。HashMap 是 Redis 中的一种键值对存储数据结构，可以用于存储多个字段和值。在 Redis 中，可以使用 HSET 命令将多个字段和值存储在一个哈希表中，然后使用 HGET 命令获取指定字段的值。如果使用 String 类型来存储多个属性，则需要使用特定的分隔符将不同的属性值拼接成一个字符串，这样会使得操作复杂，并且可能会浪费内存空间。

反面示例

```
set user:1:name tom
set user:1:age 19
```

```
set user:1:favor football
```

正面示例

```
hmset user:1 name tom age 19 favor football
```

命令使用准则

最近更新时间：2023-05-11 15:16:12

关注 O(N) 命令中的 N

- hgetall、lrange、smembers、zrange、sinter 等命令建议不要过多地使用，当使用时，需要明确 N 的值。
- Redis 中的 hscan、sscan 和 zscan 命令可以用于遍历哈希表、集合和有序集合。这些命令可以通过迭代器逐步扫描数据集中的元素，而不会像 hgetall、smembers 和 zrange 那样一次性返回所有元素。在实际使用中，建议在使用这些命令时指定合适的 COUNT 参数，以避免一次性返回过多的元素导致 Redis 的性能下降。通常情况下，每次遍历返回1000个元素左右是比较合适的选择。但是具体的数量限制还取决于 Redis 的实际环境和硬件配置，需要根据实际情况进行调整。

禁用命令

禁止线上使用 keys、flushall、flushdb 等，因为 CRedis 是单线程工作，这些命令执行时间过长，易导致命令执行阻塞。建议通过 scan 的方式渐进式处理，或通过参数 disable-command-list 配置禁用命令。

- FLUSHDB 和 FLUSHALL：这两个命令可以清空 Redis 中的所有数据，因此在生产环境中应该避免使用。
- KEYS：此命令可以返回与指定模式匹配的所有键，但由于它会阻塞 Redis 服务器，因此在生产环境中不建议使用。
- RANDOMKEY：此命令可以随机返回一个键，但由于它会阻塞 Redis 服务器，因此在生产环境中不建议使用。
- INFO：此命令可以返回 Redis 服务器的各种统计信息和配置选项，但由于它会阻塞 Redis 服务器，因此在生产环境中不建议使用。
- CONFIG：此命令可以用于修改 Redis 服务器的配置选项，但由于它可能会导致服务器崩溃，因此在生产环境中应该谨慎使用。
- SHUTDOWN：此命令可以关闭 Redis 服务器，但由于它会导致数据丢失，因此在生产环境中应该避免使用。
- BGREWRITEAOF 和 BGSAVE：这两个命令可以用于异步地重写 AOF 文件和 RDB 快照文件，但由于它们可能会消耗大量的系统资源，因此在生产环境中应该谨慎使用。

合理使用 Select

Redis 多数据库采用递增数字的命名方式，在使用过程中可随时使用 SELECT 更换数据库。数据库索引号 Index 用数字值指定，以 0 作为起始索引值。
Redis 支持多数据库操作方式，在标准版场景客户可以根据多 DB 进行数据区分。但是 Redis 本身是单线程处理数据，即使使用多 DB，业务请求也会受到其他DB 操作影响。在集群版场景，建议客户优先使用0号 DB，非0 DB 不支持扩容。并且在客户请求时，可以不执行 select 0，减少非必要交互。

适当使用批量操作

应用侧访问 Redis，其中较多一部分耗时是网络 RTT。如果应用需要做大量的 get 或者 set，可以适当使用 mget、mset 进行批量数据操作以降低网络 RTT 开销。使用 mget、mset 一般元素个数不超过500，mget 和 mset 的操作 Key 越大时，后端如果出现抖动或者扩容时，对业务影响也会更大。

- 原生命令：例如 mget、mset。
- 非原生命令：可以使用 pipeline 提高效率。

> **说明：**
> 注意控制一次批量操作的元素个数，建议在500以内，同时注意批量操作的元素中是否有 Big key。
> - 原生是原子操作，pipeline 是非原子操作。
> - pipeline 可以打包不同的命令，原生不支持。
> - pipeline 需要客户端和服务端同时支持。

不建议使用事务

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 必须在同一个 Slot 上。

集群版使用 Lua 的特殊要求

- 所有 Key 都应该由 KEYS 数组来传递。redis.call/pcall 里面调用的 Redis 命令，Key 的位置必须是 KEYS array, 否则直接返回如下错误信息：
  `error，"-ERR bad lua script for redis cluster,all the keys that the script uses should be passed using the KEYS array"`
- 单个 Lua 脚本操作的 Key 必须在同一个节点上，否则直接返回如下错误信息：
  `error, "-ERR eval/evalsha command keys must in same slotrn"`

关于 monitor 命令

Monitor 本身对 Redis 的性能有一定的影响。日常使用时，只用于分析命令的执行，不用于监控。若不进行相关问题排查和分析时，不建议开启。必要情况下，使用 Monitor 命令时，需要注意及时停止，不要长时间开启。

禁止将 Redis 作为消息队列

严禁将 Redis 当作消息队列使用，否则可能会有容量、网络、效率、功能方面的多种问题。

客户端程序设计准则

最近更新时间：2023-05-11 16:52:58

避免 DB 重用

避免多个应用使用同一个 Redis 实例。

- 原因：Key 淘汰规则的存在，多个应用的 Key 会相互影响，导致缓存命中率的下降。同时，若多个应用中有部分存在大量访问，也会影响其他应用的正常使用。
- 建议：将不相干的业务进行拆分，公共数据做服务化。

使用连接池

Redis 整个访问的时间包含几个部分：网络连接时间、命令解析时间、命令实行时间。使用带有连接池的数据库，可节约网络连接时间，加快访问 Redis 的效率，并且可高效控制连接数量。连接资源池关键配置参数包括：连接池最大连接数、最大空间连接数、最小空闲连接数，这三个参数建议配置为相同的数值。具体大小，请业务侧根据实际情况进行评估。

- **连接池最大连接数**：即控制业务并发量。当连接池中的连接数达到最大连接数时，连接池将不再创建新的连接。这有助于确保连接池不会占用过多的系统资源。
- **连接池最大空闲连接数**：是指连接池中允许保持空闲的最大连接数。当连接池中的连接数超过最大空闲连接数时，多余的连接将被关闭并从连接池中移除，从而释放系统资源。这有助于确保连接池不会占用过多的系统资源，同时也有助于提高应用程序的性能和可伸缩性。
- **连接池最小空闲连接数**：指连接池中必须保持的最小空闲连接数。如果连接池中的空闲连接数低于此值，连接池将创建新的连接以满足此要求。这有助于确保在高负载情况下，连接池中始终有足够的可用连接，

除了以上所描述的参数配置之外，不同语言连接池的队列连接方式采用了热连接的方式，需要对应修改代码。具体信息，请参见下表：

多语言类型	问题及建议	代码示例
golang go-redis 连接池	连接池库，建议连接用完放入队尾，而获取连接从队头拿取，避免总是获取热连接。	• 队列连接方式相关代码，如下所示： `func (p *ConnPool) popIdle() *Conn {` `  if len(p.idleConns) == 0 {` `  return nil` `  }` `  // fix get connection from head,default is back` `  cn = nil` `  // Lifo 为true从队头拿，为false从队尾拿`

		`` if p.opt.Lifo == true { cn = p.idleConns[0] p.idleConns = p.idleConns[1:] } else { idx := len(p.idleConns) - 1 cn = p.idleConns[idx] p.idleConns = p.idleConns[:idx] } p.idleConnsLen-- p.checkMinIdleConns() return cn } ``` • 更多信息，请参见 go-redis 连接池官网示例 。v8.11.5版本及之后的版本已经修复队列的连接方式，请确认版本及代码逻辑，避免使用热连接方式。
golang redigo 连接池配置	redigo 官网提供的连接池只支持从队头拿连接，从队头放回已用连接。导致最热连接一直被使用，不轮询每一个连接，无法实现负载均衡，导致 Proxy 连接或是负载总是不均衡的现象。需修改源代码中的pool.go 文件，增加 pushBack 方法和 Lifo 成员变量。	• 代码示例，如下所示： ``` // idle connection in the pool method, True: pushBack, False: pushFront, default False Lifo bool // fix add pushBack if p.Lifo == true { p.idle.pushBack(pc) } else { p.idle.pushFront(pc) } ``` • 更多信息，请参见 Redigo 连接池使用建议 。
java_jedis_pool	• 设置连接池连接队列方式。 ○ **false**：后进后出即从队头拿连接，从队尾放连接。推荐使用该方式。 ○ **true**：后进先出即从队头拿连接，从头放连	• 关键代码示例，请参见 Jedis 连接池代码示例 。 • 完整代码示例，请参见 spring-boot-jedis-demo 。

	接，永远为最热连接，默认为该方式。 • 指定连接池定期检查时间，避免连接可能会被长时间占用而不释放。建议配置为3000ms，表示每3秒检测一次。time-between-eviction-runs: 3000ms	
java_lettuce_pool	• 设置连接池队列方式与java_jedis_pool一致。 • 请关闭连接复用，避免出现 PipeLine。 	配置参数，建议如下所示。完整代码示例，请参见 java_jedis_lettuce_pool 。核心代码，请关注 RedisConfig.java 与 RedisProperties.java。 server: port: 8989 spring: redis: database: 0 host: 172.17.0.43 port: 6379 # 密码 没有则可以不填 password: ###### # 连接超时时间，单位毫秒 timeout: 1000 # 如果使用的jedis 则将lettuce改成jedis即可 lettuce: pool: # 获取连接池中的连接，最大等待时间ms max-wait: 1000ms # 最大活跃链接数 默认8 max-active: 2000 # 最大空闲连接数 默认8 max-idle: 1000 # 最小空闲连接数 默认0 min-idle: 500 time-between-eviction-runs: 3000ms # 设置连接池连接队列方式，false：后进后出即从队头拿连接，从队尾放连接；true:

		后进先出即从队头拿连接，从头放连接，永远拿的是最热连接不推存 lifo: false #shutdown-timeout: 1000
spring_boot_redisson 连接池	请将连接池最大连接数、最大空间连接数、最小空闲连接数，三个参数配置为相同的值。	队列连接方式相关代码，如下所示： # 单主节点配置(clusterServersConfig:集群模式) singleServerConfig # 连接空闲超时，单位：毫秒 idleConnectionTimeout: 10000 # 连接超时，单位：毫秒 connectTimeout: 10000 # 命令等待超时，单位：毫秒 timeout: 3000 # 命令失败重试次数,如果尝试达到 retryAttempts（命令失败重试次数）仍然不能将命令发送至某个指定的节点时，将抛出错误。 # 如果尝试在此限制之内发送成功，则开始启用 timeout（命令等待超时）计时。 retryAttempts: 3 # 命令重试发送时间间隔，单位：毫秒 retryInterval: 1500 # # 重新连接时间间隔，单位：毫秒 # reconnectionTimeout: 3000 # # 执行失败最大次数 # failedAttempts: 3 # 密码 password: 111 # 单个连接最大订阅数量 subscriptionsPerConnection: 5 # 客户端名称 clientName: cdkey # # 节点地址 address: redis://172.20.1.20:6379 # 发布和订阅连接的最小空闲连接数 subscriptionConnectionMinimumIdleSize: 1 # 发布和订阅连接池大小 subscriptionConnectionPoolSize: 50

```
 # 最小空闲连接数
 connectionMinimumIdleSize: 1000
 # 连接池大小
 connectionPoolSize: 10000
 # 数据库编号
 database: 0
 # DNS监测时间间隔，单位：毫秒
 dnsMonitoringInterval: 5000
# 线程池数量,默认值: 当前处理核数量 * 2
threads: 64
# Netty线程池数量,默认值: 当前处理核数量
* 2
nettyThreads: 64
# 编码
codec: !
<org.redisson.codec.JsonJacksonCodec>
{}
# 传输模式
transportMode : "NIO"
```

更多信息，请参见spring_boot_redisson 连接池官网链接 。

增加熔断功能

高并发场景，建议客户端添加熔断逻辑功能，例如：netflix、hystrix。Redis 客户端的熔断器实时检测集群节点，当某一个 Redis 节点出现异常，便不再请求有异常的 Redis 节点，从而避免单个节点的故障导致整体系统的雪崩。

配置合理密码

- 数据库访问密码可以保障数据的安全性。密码复杂度要求：
  - 字符个数为[8,30]。
  - 至少包含小写字母、大写字母、数字和字符 ()`~!@#$%^&*−+=_|{}[]:;<>,.?/ 中的2种。
  - 不能以"/"开头。
- 腾讯云支持 SSL（Secure Sockets Layer）加密认证访问，具体操作，请参见 SSL 加密 。

连接池配置

Redigo 连接池使用建议

最近更新时间：2023-05-11 15:16:12

redigo 官网连接池 仅支持从队头拿连接。从队头放回已用连接，导致最热连接一直被使用，不轮询每一个连接，总会引起 Redis 的 Proxy 出现连接或是负载不均衡的问题。建议修改源代码中的 pool.go 文件，增加 pushBack 方法，将已使用的连接加入在队尾。

增加 pushBack 代码示例

```
// idle connect push list back
func (l *idleList) pushBack(pc *poolConn) {

if l.count == 0 {
l.front = pc
l.back = pc
pc.prev = nil
pc.next = nil
} else {
pc.prev = l.back
l.back.next = pc
l.back = pc
pc.next = nil
}

l.count++
}

// idle connection in the pool method, True: pushBack, False: pushFront, default False
Lifo bool

// fix add pushBack
if p.Lifo == true {
p.idle.pushBack(pc)
} else {
p.idle.pushFront(pc)
}
```

完整代码示例

```
====================整个代码修改后
pool.go==========================

// Copyright 2012 Gary Burd
//
// Licensed under the Apache License, Version 2.0 (the "License"): you may
// not use this file except in compliance with the License. You may obtain
// a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
// WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
// License for the specific language governing permissions and limitations
// under the License.

package redis

import (
  "bytes"
  "context"
  "crypto/rand"
  "crypto/sha1"
  "errors"
  "io"
  "strconv"
  "sync"
  "time"
)

var (
  _ ConnWithTimeout = (*activeConn)(nil)
  _ ConnWithTimeout = (*errorConn)(nil)
)

var nowFunc = time.Now // for testing

// ErrPoolExhausted is returned from a pool connection method (Do, Send,
// Receive, Flush, Err) when the maximum number of database connections in the
// pool has been reached.
var ErrPoolExhausted = errors.New("redigo: connection pool exhausted")

var (
  errConnClosed = errors.New("redigo: connection closed")
```

```
)

// Pool maintains a pool of connections. The application calls the Get method
// to get a connection from the pool and the connection's Close method to
// return the connection's resources to the pool.
//
// The following example shows how to use a pool in a web application. The
// application creates a pool at application startup and makes it available to
// request handlers using a package level variable. The pool configuration used
// here is an example, not a recommendation.
//
//  func newPool(addr string) *redis.Pool {
//    return &redis.Pool{
//      MaxIdle: 3,
//      IdleTimeout: 240 * time.Second,
//      // Dial or DialContext must be set. When both are set, DialContext takes
precedence over Dial.
//      Dial: func () (redis.Conn, error) { return redis.Dial("tcp", addr) },
//    }
//  }
//
//  var (
//    pool *redis.Pool
//    redisServer = flag.String("redisServer", ":6379", "")
//  )
//
//  func main() {
//    flag.Parse()
//    pool = newPool(*redisServer)
//    ...
//  }
//
// A request handler gets a connection from the pool and closes the connection
// when the handler is done:
//
//  func serveHome(w http.ResponseWriter, r *http.Request) {
//      conn := pool.Get()
//      defer conn.Close()
//      ...
//  }
//
// Use the Dial function to authenticate connections with the AUTH command or
// select a database with the SELECT command:
//
//  pool := &redis.Pool{
```

```
//    // Other pool configuration not shown in this example.
//    Dial: func () (redis.Conn, error) {
//      c, err := redis.Dial("tcp", server)
//      if err != nil {
//        return nil, err
//      }
//      if _, err := c.Do("AUTH", password); err != nil {
//        c.Close()
//        return nil, err
//      }
//      if _, err := c.Do("SELECT", db); err != nil {
//        c.Close()
//        return nil, err
//      }
//      return c, nil
//    },
//  }
//
// Use the TestOnBorrow function to check the health of an idle connection
// before the connection is returned to the application. This example PINGs
// connections that have been idle more than a minute:
//
//  pool := &redis.Pool{
//    // Other pool configuration not shown in this example.
//    TestOnBorrow: func(c redis.Conn, t time.Time) error {
//      if time.Since(t) < time.Minute {
//        return nil
//      }
//      _, err := c.Do("PING")
//      return err
//    },
//  }
//
type Pool struct {
  // Dial is an application supplied function for creating and configuring a
  // connection.
  //
  // The connection returned from Dial must not be in a special state
  // (subscribed to pubsub channel, transaction started, ...).
  Dial func() (Conn, error)

  // DialContext is an application supplied function for creating and configuring a
  // connection with the given context.
  //
  // The connection returned from Dial must not be in a special state
```

```
    // (subscribed to pubsub channel, transaction started, ...).
    DialContext func(ctx context.Context) (Conn, error)

    // TestOnBorrow is an optional application supplied function for checking
    // the health of an idle connection before the connection is used again by
    // the application. Argument t is the time that the connection was returned
    // to the pool. If the function returns an error, then the connection is
    // closed.
    TestOnBorrow func(c Conn, t time.Time) error

    // Maximum number of idle connections in the pool.
    MaxIdle int

    // idle connection in the pool method, True: pushBack, False: pushFront, default False
    Lifo bool

    // Maximum number of connections allocated by the pool at a given time.
    // When zero, there is no limit on the number of connections in the pool.
    MaxActive int

    // Close connections after remaining idle for this duration. If the value
    // is zero, then idle connections are not closed. Applications should set
    // the timeout to a value less than the server's timeout.
    IdleTimeout time.Duration

    // If Wait is true and the pool is at the MaxActive limit, then Get() waits
    // for a connection to be returned to the pool before returning.
    Wait bool

    // Close connections older than this duration. If the value is zero, then
    // the pool does not close connections based on age.
    MaxConnLifetime time.Duration

    mu           sync.Mutex    // mu protects the following fields
    closed       bool          // set to true when the pool is closed.
    active       int           // the number of open connections in the pool
    initOnce     sync.Once     // the init ch once func
    ch           chan struct{} // limits open connections when p.Wait is true
    idle         idleList      // idle connections
    waitCount    int64         // total number of connections waited for.
    waitDuration time.Duration // total time waited for new connections.
}

// NewPool creates a new pool.
//
```

```
// Deprecated: Initialize the Pool directly as shown in the example.
func NewPool(newFn func() (Conn, error), maxIdle int) *Pool {
  return &Pool{Dial: newFn, MaxIdle: maxIdle}
}

// Get gets a connection. The application must close the returned connection.
// This method always returns a valid connection so that applications can defer
// error handling to the first use of the connection. If there is an error
// getting an underlying connection, then the connection Err, Do, Send, Flush
// and Receive methods return that error.
func (p *Pool) Get() Conn {
  // GetContext returns errorConn in the first argument when an error occurs.
  c, _ := p.GetContext(context.Background())
  return c
}

// GetContext gets a connection using the provided context.
//
// The provided Context must be non-nil. If the context expires before the
// connection is complete, an error is returned. Any expiration on the context
// will not affect the returned connection.
//
// If the function completes without error, then the application must close the
// returned connection.
func (p *Pool) GetContext(ctx context.Context) (Conn, error) {
  // Wait until there is a vacant connection in the pool.
  waited, err := p.waitVacantConn(ctx)
  if err != nil {
    return errorConn{err}, err
  }

  p.mu.Lock()

  if waited > 0 {
    p.waitCount++
    p.waitDuration += waited
  }

  // Prune stale connections at the back of the idle list.
  if p.IdleTimeout > 0 {
    n := p.idle.count
    for i := 0; i < n && p.idle.back != nil &&
p.idle.back.t.Add(p.IdleTimeout).Before(nowFunc()); i++ {
      pc := p.idle.back
      p.idle.popBack()
```

```
            p.mu.Unlock()
            pc.c.Close()
            p.mu.Lock()
            p.active--
        }
    }

    // Get idle connection from the front of idle list.
    for p.idle.front != nil {
        pc := p.idle.front
        p.idle.popFront()
        p.mu.Unlock()
        if (p.TestOnBorrow == nil || p.TestOnBorrow(pc.c, pc.t) == nil) &&
            (p.MaxConnLifetime == 0 || nowFunc().Sub(pc.created) < p.MaxConnLifetime) {
            return &activeConn{p: p, pc: pc}, nil
        }
        pc.c.Close()
        p.mu.Lock()
        p.active--
    }

    // Check for pool closed before dialing a new connection.
    if p.closed {
        p.mu.Unlock()
        err := errors.New("redigo: get on closed pool")
        return errorConn{err}, err
    }

    // Handle limit for p.Wait == false.
    if !p.Wait && p.MaxActive > 0 && p.active >= p.MaxActive {
        p.mu.Unlock()
        return errorConn{ErrPoolExhausted}, ErrPoolExhausted
    }

    p.active++
    p.mu.Unlock()
    c, err := p.dial(ctx)
    if err != nil {
        p.mu.Lock()
        p.active--
        if p.ch != nil && !p.closed {
            p.ch <- struct{}{}
        }
        p.mu.Unlock()
        return errorConn{err}, err
```

```
    }
    return &activeConn{p: p, pc: &poolConn{c: c, created: nowFunc()}}, nil
}

// PoolStats contains pool statistics.
type PoolStats struct {
    // ActiveCount is the number of connections in the pool. The count includes
    // idle connections and connections in use.
    ActiveCount int
    // IdleCount is the number of idle connections in the pool.
    IdleCount int

    // WaitCount is the total number of connections waited for.
    // This value is currently not guaranteed to be 100% accurate.
    WaitCount int64

    // WaitDuration is the total time blocked waiting for a new connection.
    // This value is currently not guaranteed to be 100% accurate.
    WaitDuration time.Duration
}

// Stats returns pool's statistics.
func (p *Pool) Stats() PoolStats {
    p.mu.Lock()
    stats := PoolStats{
        ActiveCount:  p.active,
        IdleCount:    p.idle.count,
        WaitCount:    p.waitCount,
        WaitDuration: p.waitDuration,
    }
    p.mu.Unlock()

    return stats
}

// ActiveCount returns the number of connections in the pool. The count
// includes idle connections and connections in use.
func (p *Pool) ActiveCount() int {
    p.mu.Lock()
    active := p.active
    p.mu.Unlock()
    return active
}

// IdleCount returns the number of idle connections in the pool.
```

```
func (p *Pool) IdleCount() int {
    p.mu.Lock()
    idle := p.idle.count
    p.mu.Unlock()
    return idle
}

// Close releases the resources used by the pool.
func (p *Pool) Close() error {
    p.mu.Lock()
    if p.closed {
        p.mu.Unlock()
        return nil
    }
    p.closed = true
    p.active -= p.idle.count
    pc := p.idle.front
    p.idle.count = 0
    p.idle.front, p.idle.back = nil, nil
    if p.ch != nil {
        close(p.ch)
    }
    p.mu.Unlock()
    for ; pc != nil; pc = pc.next {
        pc.c.Close()
    }
    return nil
}

func (p *Pool) lazyInit() {
    p.initOnce.Do(func() {
        p.ch = make(chan struct{}, p.MaxActive)
        if p.closed {
            close(p.ch)
        } else {
            for i := 0; i < p.MaxActive; i++ {
                p.ch <- struct{}{}
            }
        }
    })
}

// waitVacantConn waits for a vacant connection in pool if waiting
// is enabled and pool size is limited, otherwise returns instantly.
// If ctx expires before that, an error is returned.
```

```
//
// If there were no vacant connection in the pool right away it returns the time spent
waiting
// for that connection to appear in the pool.
func (p *Pool) waitVacantConn(ctx context.Context) (waited time.Duration, err error) {
  if !p.Wait || p.MaxActive <= 0 {
    // No wait or no connection limit.
    return 0, nil
  }

  p.lazyInit()

  // wait indicates if we believe it will block so its not 100% accurate
  // however for stats it should be good enough.
  wait := len(p.ch) == 0
  var start time.Time
  if wait {
    start = time.Now()
  }

  select {
  case <-p.ch:
    // Additionally check that context hasn't expired while we were waiting,
    // because `select` picks a random `case` if several of them are "ready".
    select {
    case <-ctx.Done():
      p.ch <- struct{}{}
      return 0, ctx.Err()
    default:
    }
  case <-ctx.Done():
    return 0, ctx.Err()
  }

  if wait {
    return time.Since(start), nil
  }
  return 0, nil
}

func (p *Pool) dial(ctx context.Context) (Conn, error) {
  if p.DialContext != nil {
    return p.DialContext(ctx)
  }
  if p.Dial != nil {
```

```
    return p.Dial()
  }
  return nil, errors.New("redigo: must pass Dial or DialContext to pool")
}

func (p *Pool) put(pc *poolConn, forceClose bool) error {
  p.mu.Lock()
  if !p.closed && !forceClose {
    pc.t = nowFunc()

    // fix add pushBack
    if p.Lifo == true {
      p.idle.pushBack(pc)
    } else {
      p.idle.pushFront(pc)
    }

    if p.idle.count > p.MaxIdle {
      pc = p.idle.back
      p.idle.popBack()
    } else {
      pc = nil
    }
  }

  if pc != nil {
    p.mu.Unlock()
    pc.c.Close()
    p.mu.Lock()
    p.active--
  }

  if p.ch != nil && !p.closed {
    p.ch <- struct{}{}
  }
  p.mu.Unlock()
  return nil
}

type activeConn struct {
  p     *Pool
  pc    *poolConn
  state int
}
```

```
var (
  sentinel     []byte
  sentinelOnce sync.Once
)

func initSentinel() {
  p := make([]byte, 64)
  if _, err := rand.Read(p); err == nil {
    sentinel = p
  } else {
    h := sha1.New()
    io.WriteString(h, "Oops, rand failed. Use time instead.")       // nolint: errcheck
    io.WriteString(h, strconv.FormatInt(time.Now().UnixNano(), 10)) // nolint: errcheck
    sentinel = h.Sum(nil)
  }
}

func (ac *activeConn) firstError(errs ...error) error {
  for _, err := range errs[:len(errs)-1] {
    if err != nil {
      return err
    }
  }
  return errs[len(errs)-1]
}

func (ac *activeConn) Close() (err error) {
  pc := ac.pc
  if pc == nil {
    return nil
  }
  ac.pc = nil

  if ac.state&connectionMultiState != 0 {
    err = pc.c.Send("DISCARD")
    ac.state &^= (connectionMultiState | connectionWatchState)
  } else if ac.state&connectionWatchState != 0 {
    err = pc.c.Send("UNWATCH")
    ac.state &^= connectionWatchState
  }
  if ac.state&connectionSubscribeState != 0 {
    err = ac.firstError(err,
      pc.c.Send("UNSUBSCRIBE"),
      pc.c.Send("PUNSUBSCRIBE"),
    )
```

```
      // To detect the end of the message stream, ask the server to echo
      // a sentinel value and read until we see that value.
      sentinelOnce.Do(initSentinel)
      err = ac.firstError(err,
        pc.c.Send("ECHO", sentinel),
        pc.c.Flush(),
      )
      for {
        p, err2 := pc.c.Receive()
        if err2 != nil {
          err = ac.firstError(err, err2)
          break
        }
        if p, ok := p.([]byte); ok && bytes.Equal(p, sentinel) {
          ac.state &^= connectionSubscribeState
          break
        }
      }
    }
    _, err2 := pc.c.Do("")
    return ac.firstError(
      err,
      err2,
      ac.p.put(pc, ac.state != 0 || pc.c.Err() != nil),
    )
}

func (ac *activeConn) Err() error {
    pc := ac.pc
    if pc == nil {
      return errConnClosed
    }
    return pc.c.Err()
}

func (ac *activeConn) Do(commandName string, args ...interface{}) (reply
interface{}, err error) {
    pc := ac.pc
    if pc == nil {
      return nil, errConnClosed
    }
    ci := lookupCommandInfo(commandName)
    ac.state = (ac.state | ci.Set) &^ ci.Clear
    return pc.c.Do(commandName, args...)
}
```

```
func (ac *activeConn) DoWithTimeout(timeout time.Duration, commandName string,
args ...interface{}) (reply interface{}, err error) {
  pc := ac.pc
  if pc == nil {
    return nil, errConnClosed
  }
  cwt, ok := pc.c.(ConnWithTimeout)
  if !ok {
    return nil, errTimeoutNotSupported
  }
  ci := lookupCommandInfo(commandName)
  ac.state = (ac.state | ci.Set) &^ ci.Clear
  return cwt.DoWithTimeout(timeout, commandName, args...)
}

func (ac *activeConn) Send(commandName string, args ...interface{}) error {
  pc := ac.pc
  if pc == nil {
    return errConnClosed
  }
  ci := lookupCommandInfo(commandName)
  ac.state = (ac.state | ci.Set) &^ ci.Clear
  return pc.c.Send(commandName, args...)
}

func (ac *activeConn) Flush() error {
  pc := ac.pc
  if pc == nil {
    return errConnClosed
  }
  return pc.c.Flush()
}

func (ac *activeConn) Receive() (reply interface{}, err error) {
  pc := ac.pc
  if pc == nil {
    return nil, errConnClosed
  }
  return pc.c.Receive()
}

func (ac *activeConn) ReceiveWithTimeout(timeout time.Duration) (reply interface{},
err error) {
  pc := ac.pc
```

```
  if pc == nil {
    return nil, errConnClosed
  }
  cwt, ok := pc.c.(ConnWithTimeout)
  if !ok {
    return nil, errTimeoutNotSupported
  }
  return cwt.ReceiveWithTimeout(timeout)
}

type errorConn struct{ err error }

func (ec errorConn) Do(string, ...interface{}) (interface{}, error) { return nil, ec.err }
func (ec errorConn) DoWithTimeout(time.Duration, string, ...interface{}) (interface{},
error) {
  return nil, ec.err
}
func (ec errorConn) Send(string, ...interface{}) error                { return ec.err }
func (ec errorConn) Err() error                                       { return ec.err }
func (ec errorConn) Close() error                                     { return nil }
func (ec errorConn) Flush() error                                     { return ec.err }
func (ec errorConn) Receive() (interface{}, error)                    { return nil, ec.err }
func (ec errorConn) ReceiveWithTimeout(time.Duration) (interface{}, error) { return
nil, ec.err }

type idleList struct {
  count      int
  front, back *poolConn
}

type poolConn struct {
  c          Conn
  t          time.Time
  created    time.Time
  next, prev *poolConn
}

func (l *idleList) pushFront(pc *poolConn) {
  pc.next = l.front
  pc.prev = nil
  if l.count == 0 {
    l.back = pc
  } else {
    l.front.prev = pc
  }
```

```
    l.front = pc
    l.count++
}


// idle connect push list back
func (l *idleList) pushBack(pc *poolConn) {

    if l.count == 0 {
        l.front = pc
        l.back = pc
        pc.prev = nil
        pc.next = nil
    } else {
        pc.prev = l.back
        l.back.next = pc
        l.back = pc
        pc.next = nil
    }

    l.count++
}


func (l *idleList) popFront() {
    pc := l.front
    l.count--
    if l.count == 0 {
        l.front, l.back = nil, nil
    } else {
        pc.next.prev = nil
        l.front = pc.next
    }
    pc.next, pc.prev = nil, nil
}

func (l *idleList) popBack() {
    pc := l.back
    l.count--
    if l.count == 0 {
        l.front, l.back = nil, nil
    } else {
        pc.prev.next = nil
        l.back = pc.prev
    }
```

```
    pc.next, pc.prev = nil, nil
}
==========================================================
```

初始方法：

```
// 建立连接池
redisClient = &redis.Pool{
    MaxIdle:     maxIdle,
    MaxActive:   maxActive,
    IdleTimeout: MaxIdleTimeout * time.Second,
    Wait:        true,
    Lifo:        true,  # 设置为true，这是重点
    Dial: func() (redis.Conn, error) {
        con, err := redis.Dial("tcp", conf["Host"].(string),
            redis.DialPassword(conf["Password"].(string)),
            redis.DialDatabase(int(conf["Db"].(int64))),
            redis.DialConnectTimeout(timeout*time.Second),
            redis.DialReadTimeout(timeout*time.Second),
            redis.DialWriteTimeout(timeout*time.Second))
        if err != nil {
            return nil, err
        }
        return con, nil
    },
}
```

Jedis 连接池代码示例

最近更新时间：2023-10-10 10:22:21

准备工作

下载客户端 Jedis，推荐使用最新版本。

代码示例

连接池代码示例及其中各个参数的含义，如下所示：

> **说明：**
> 解决连接不均的问题，请设置最大连接数，最小空闲连接数，最大空闭连接数相同，避免连接用完回到连接池时断连和新建连接。业务根据自己的需要设置相应的值。

参数	含义	建议
setMaxTotal	连接池中的最大连接数。	设置该参数，需要考虑业务并发量、访问延迟、最大连接数等因素。
setMaxIdle	连接池最大空闲连接数。	建议与 setMaxTotal 相同。
setMinIdle	资源池允许的最小空闲连接数。	建议与 setMaxTotal 相同。
timeout	超时时间。	该参数需要根据业务模型及网络链路性能设置。 • 一般网络延迟较低，服务耗时非常敏感的业务，可以设置50-100ms。 • 如果业务容忍度高，或者业务访问 kv 数据较大，可以设置500ms、1000ms。
setTestOnBorrow	设置在从连接池中获取连接时是否进行连接测试。	• 如果设置为 `true`，则在获取连接时会调用 `connection.isValid()` 方法进行连接测试。以确保获取到的连接是可用的，但同时会消耗 QPS 性能。 • 如果设置为 `false`，则不会进行连接测试。可以提高连接获取的速度，但是可能会获取到不可用的连接。
setTestOnReturn	设置将连接归还连接池时，是否进行校验。	• 如果设置为 `true`，则在归还连接时会调用 `connection.isValid()` 方法进行连接测试，以确保归还的连接是可用的。 • 如果设置为 `false`，则不会进行连接测试。可以提高归还连

		接的速度，但是可能会归还不可用的连接。通常情况下建议设置为 `true` 。

```
JedisPoolConfig config = new JedisPoolConfig();
// 最大空闲连接数，需自行评估，不超过Redis实例的最大连接数
config.setMaxIdle(200);
// 最大连接数，需自行评估，不超过Redis实例的最大连接数
config.setMaxTotal(200);
//资源池允许的最小空闲连接数
config.setMinIdle(20);
//当资源池连接用尽后，调用者的最大等待时间（单位为毫秒）
config.setMaxWaitMillis(3000);
//从连接池中获取对象时，会先进行ping检查，检查不通过，会从连接池中移走并销毁。
config.setTestOnBorrow(false);
//归还连接时，会进行检查，检查不通过，则销毁。
config.setTestOnReturn(false);
// 解决请求不均，设置连接池方式为队列，设置 lifo 为false。
// false：后进后出即从队头拿连接，从队尾放连接；true: 后进先出即从队头拿连接，从头放连接，永远拿的是最热连接不推存。
config.setLifo(false);
//设置最小连接检查
config.setTimeBetweenEvictionRunsMillis(3000);
// 分别将host和password的值替换为实例的连接地址、密码
String host = "192.xx.xx.195";
String password = "123ad6aq";
//读写超时（单位为毫秒）
int timeout = 2000;
int port = 6379;
JedisPool pool = new JedisPool(config,host,port,timeout,password);
Jedis jedis = null;
boolean broken = false;
try
{
   jedis = pool.getResource();
   /// ... do stuff here ... for example
   jedis.set("redis", "tencent");
   String foobar = jedis.get("redis");
   jedis.zadd("tec", 0, "a");
   jedis.zadd("tec", 0, "b");
   Set < String > sose = jedis.zrange("tec", 0, -1);
}
catch(Exception e)
{
   broken = true;
```

```
}
finally
{
    if(broken)
    {
        pool.returnBrokenResource(jedis);
    }
    else if(jedis != null)
    {
        pool.returnResource(jedis);
    }
}
```