TencentDB for Redis

Development Guidelines

Product Documentation

Contents

Development Guidelines
Naming Rules

Last updated：2023-10-20 10:46:05

Instance Naming Rules

Name Redis instance in accordance with the following rules: {Environment}-{Branch}-{Project number}-{Business name}-{Region}-{Zone}-{Number}.

Environment: prd or dev.

Branch: Name of the branch, such as `cx` .

Project number: Assign a number to each project group, such as `p001` .

Business name: Redis business name, such as `abc` .

Region: Abbreviation for the region of an instance, such as `gz` (Guangzhou).

Zone: Abbreviation for the zone of an instance, such as `gzcx01` .

Number: The number of the first Redis instance, which starts from 1.

So the instance name is `prd-cx-p001-abc-gz-gzcx01-1` .

Key Naming Rules

Key name must be prefixed with a business or a database name separated by colons to avoid `key` conflict. The recommended naming rule is: `business name:database name:table name: data ID` .

Business name: Abbreviation of the business system, such as `cx` .

Database name: Name of the database, such as `cxdb` .

Table name: Name of the database table, such as `user` .

Data ID: ID or the primary key field in the data table, such as "uid"in database table.

So the key name is `cx:cxdb:user:000110011` .

Basic Usage Guidelines

Last updated：2023-10-20 10:39:47

Cache-Only Policy

Redis is only used to cache data. It stores all data in memory for quicker access. However, using it to persist data may result in data loss because it can't store data in memory persistently after a power failure.

Non-Unique Data Source Policy

Since Redis is used for caching, cache misses are possible, so it cannot be relied upon as the sole source of data. In case of an exception when accessing Redis, the backend database needs to be queried.

Key Eviction Policy

Set the `maxmemory-policy` as needed. The default policy is `noeviction` , which means that keys will not be deleted. When memory is used up, OOM will occur. So we recommend that you modify the eviction policy once the instance is created to reduce OOM.

Configurable memory eviction policy

Select one of the following eviction policies for maxmemory-policy when the Redis in-memory cache was used up. LRU (least recently used) and TTL (time to live) are implemented by randomized and approximation algorithms.

allkeys-lru: Delete keys based on the LRU algorithm, regardless of whether the data has a timeout attribute, until enough space is freed up.

allkeys-random: Randomly delete keys until enough space is freed up.

volatile-lru: Delete expired keys based on the LRU algorithm until enough space is freed up.

volatile-random: Randomly delete expired keys until enough space is freed up.

volatile-ttl: Delete data that is about to be expired based on the TTL attribute of key-value pairs. If there is no recently expiring data, the policy will fall back to "noeviction".

noeviction: Evict no data, reject all write operations, and return the error message "(error) OOM command not allowed when used memory" to the client. Redis will only respond to read operations.

Suggestions

When Redis is used as a cache, it is recommended to use the "allkeys-lru" eviction policy. This policy evicts the keys least frequently used. By default, if a key is least frequently used, it is also less likely to be accessed in the future, so it

is evicted.

When Redis is used as data persistence and caching in a hybrid manner, the volatile-lru policy can be used. As Redis is not recommended to store data persistently, it is only considered as an alternative.

Design Principles of Key and Value

Last updated：2023-10-20 10:47:07

Key Design Principle

Redis keys should be readable and manageable. We recommend that you not use an ambiguous and long key name.
Conciseness: Shorten the length of Key while ensuring the semantic meaning. When there are large number of keys, they take up more space, so simplify the key name. For example, simplify
`cx:cxdb:cxdb_user_info:000110011` to `cx:cxdb:user:000110011` .
Naming rule: It can only contain letters, digits, symbols (|_.:) and start with a letter.
Semantic segmentation: Separate words with the same business logic meaning by English semicolons (:); while separate words with same business logic meaning by English periods (.), indicating a complete semantic meaning.
Readability: To improve readability, Key name should end with the value type of the key, such as
`user:basic.info:userid:string` .
Do not use a long key name, as a long key name takes up space too.
The key name can't contain symbols, such as \\, *, ?, {}, [], (), space, single and double quotes, and escape characters.

Key lifecycle policy

We recommend that you set the expiration time with `expire` command to control the lifecycle for keys.

```
> set cx:cxdb:user:000110011 xiaoming
```
```
> expire cx:cxdb:user:000110011 3600  # set the key to expire one hour later
```
Distribute expiration time evenly , preventing the centralized expiration.
For data without expiration set, pay attention to the idletime and clean it up when the it is very large. For example, execute `> object idletime cx:cxdb:user:000110011` ,
and the returned information is shown below:

```
:(integer) 150039   # It indicates that the key has not been accessed for 150039 seconds.
```
Clear the cold data which has not been accessed for more than 1 month.

Value Design Principle

Reject big key

A big key indicates that the key has a large value and takes more space. The following are examples of big keys in common Redis data structures:

In String data structure, the value of a string takes up space more than 10 MB (a large value).

In Set data structure, there are 10000 values (a large number of members) in a set.

In List data structure, there are 10000 values (a large number of members) in a list.

In Hash data structure, there are only 1000 values, but the total value reaches 1000 MB (a large total value) in a hash table.

Big key is prone to causing slow queries, blocking other requests. It also poses EIN under pressure To prevent the creation of big keys, we have provide you with the following suggestions:

Maintain the value in String type within 10 KB; maintain the members in `hash`, `list`, `set`, and `zset` within 5000.

Don't delete big keys unless necessary.

For non-string big keys, we recommend that you delete them with the `hscan`, `sscan`, `zscan` command in a progressive manner.

Prevent big keys from being deleted automatically upon expiration.

For example, setting a zset with 2 million members to expire in 1 hour can trigger a deletion operation, causing blocking.

Select proper data type

Redis provides various data types, including strings, hash tables, lists, sets, and sorted sets. An appropriate data type can improve the performance and reliability of Redis.

Strings: It is suitable for storing the simple string data, such as configuration information and counters. To store binary data, use Redis binary-safe string.

Hash tables: It is suitable for storing data with multiple fields and values, such as user information and product information. Hash tables can save memory space and support easy batch operations.

List: It is suitable for storing ordered element set, such as message queues and task lists. List allows you you to insert and delete operations at both ends, and operate list by various commands provided by Redis.

Set: It is suitable for storing unordered element set, such as tag lists and friend lists. Set allows you to perform operations such as union, intersection, difference. You can operate sets by using various commands provided by Redis.

Sorted set: It is suitable for storing ordered element set, such as leaderboards, voting lists. Sorted set allows you to sort based on scores. You can operate ordered sets by using various commands provided by Redis.

Control and use memory encoding optimization configurations for data structures in a reasonable way. For example, ziplist, as a special data structure, can store small lists, hash tables, and sorted sets in a continuous memory block, which saves memory. As ziplist does not have an index, linear scanning is required for operations such as search, insertion, or deletion on ziplist, which may compromises performance. Therefore use ziplist as needed in your actual business. ziplist may be a good choice for a scenario where data volume is small and frequent traversal operations are required. Try other data structures in other scenarios.

Replace String with HashMap structure for a key with multiple attributes. HashMap is a key-value data structure in Redis for storing multiple fields and values. In Redis, you can use the HSET command to store multiple fields and

values in a hash table, and get the value of the specified field by using HGET command. To store multiple attributes in String structure, you need to use a specified delimiter to concatenate them into one string. But this operation is complicated and memory-consuming.

Negative example

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

Positive example

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

Command Usage Guidelines

Last updated：2023-10-20 10:44:16

Checking `N` in the `O(N)` command

We recommend that you not excessively use these commands, including `HGETALL`, `IRANGE`, `SMEMBERS`, `ZRANGE`, and `sinter`. When using them, you need to specify the value of `N`.

The `HSCAN`, `SSCAN`, and `ZSCAN` commands in Redis can be used to traverse hash tables, sets, and sorted sets. These commands allow you to scan elements in the data set by using an iterator, without returning all elements at once like `HGETALL`, `SMEMBERS`, and `ZRANGE`. When using these commands, we recommend that you specify an appropriate COUNT parameter to avoid Redis performance degradation caused by excessive returned elements at once. Returning 1000 elements per traversal is preferred, but the specific quantity depends on the actual Redis situation. You can adjust it as needed.

Configuring disabled commands

Commands like `KEYS`, `FLUSHALL`, `FLUSHDB` are not allowed as the single-threaded CRedis takes a long time to execute them, which may cause command execution blocking. To address the potential blocking, we recommend that you use the `scan` command in a progressive manner or configure the disabled commands by `disable-command-list` parameter.

`FLUSHDB` and `FLUSHALL` : We recommend that you not use these two commands in the production environment, as they will clear all data in Redis.

`KEYS` : It returns all keys matched with specified mode. We recommend that you not use this command in the production environment, as it will block Redis server.

`RANDOMKEY` : It returns one key randomly. We recommend that you not use this command in the production environment, as it will block Redis server.

`INFO` : It returns various statistics and configuration items of the Redis server. We recommend that you not use this command in the production environment, as it will block Redis server.

`CONFIG` : It allows you to modify the configuration items of the Redis server. But use it with caution as it may cause server to crash.

`SHUTDOWN` : It allows you to disable the Redis server. We recommend that you not use this command in the production environment, as it will cause data loss.

`BGREWRITEAOF` and `BGSAVE` : It allows you to asynchronously rewrite the AOF and RDB files with a large amount of system resources being consumed. So use it with caution.

Using SELECT properly

Redis supports naming multiple databases with incrementing numbers. During the naming, you can change database at any time by using the `SELECT` command. The index number of each database is specified by a digit that starts from 0.

Redis supports multi-database operations. In standard edition, you can sort data using multiple databases; Even so, the business requests of Redis will be affected by operations on other databases as Redis is single-threaded. In cluster edition, we recommend that you prioritize using DB 0, as non-0 DBs do not support scaling. Additionally, when handling customer requests, it is not necessary to execute "SELECT 0" to reduce unnecessary interactions.

Using batch operations properly

RTT consumes much time when accessing Redis . If your application needs to perform a large number of `GET` or `SET` operations, you can use `mget` and `mset` for batch data operations to reduce the network RTT. The number of elements in `MGET` and `MGET` operations should not exceed 500. The greater this number is, the greater the impact will be in the case of jitter or scaling on the backend.

Native command: `MEGET` , `MESETmset` .

Non-native command: `PIPELINE` , which improves efficiency.

Note

We recommend that you maintain the number of elements in each batch operation within 500 and check whether there is `big key` in the elements of batch operation.

The native command is an atom operation, while `PIPELINE` is not.

`PIPELINE` allows you to pack multiple commands,while the native command not.

`PIPELINE` must be supported on client and server.

Using no transactions

Redis has a limited transaction feature and does not support rollback. In addition, in the cluster edition of Redis, keys involved in a transaction operation must reside in the same slot.

Prerequisites for cluster edition using `Lua`

All keys should be passed using the KEYS array. When calling Redis commands in `redis.call` / `pcall` , the key must be in the KEYS array. Otherwise, the following error message will be returned: `error, "-ERR bad lua`

```
script for redis cluster,all the keys that the script uses should be passed using
the KEYS array"
```
.

For a Lua bootstrap action, the keys involved must be on the same Redis node. Otherwise, the following error message will be returned: `error, "-ERR eval/evalsha command keys must in same slotrn"` .

`MONITOR` command

The `MONITOR` command has some impacts on the performance of Redis. In daily usage, it is used only for analyzing command execution and not for monitoring. We recommend that you enable it only in the case of troubleshooting or analysis, and disable it in time.

Prohibit setting Redis as message queue

Don't use Redis as a message queue; otherwise, you may encounter problems in capacity, network, efficiency, and feature.

Design Principles of Client Programs

Last updated：2023-10-20 10:49:47

Avoiding database reuse

You should avoid to use one Redis instance for multiple applications.

Cause: Key eviction rules cause keys of multiple applications to affect each other, reducing cache hit rate. Meanwhile, if some applications have a large number of accesses, this will also affect the normal usage of other applications.

Suggestion: Split unrelated businesses and share common data.

Using connection pool

The total time for accessing Redis includes several parts: network connection time, command parsing time, and command execution time. By using a database with a connection pool, you can save network connection time, improve the efficiency of accessing Redis, and efficiently control the number of connections. The key configuration parameters for the connection resource pool include the maximum connections, the maximum and the minimum idle connections. We recommend that you configure these three parameters with the same value evaluated based on your actual conditions.

Maximum Connections: Control the concurrency of your business. When the number of connections in the connection pool reaches the maximum limit, the connection pool will no longer create new connections, so that the system will not be overwhelmed by resource requests from the connection pool.

Maximum Idle Connections: The maximum number of idle connections allowed in the connection pool. When the number of active connections in the connection pool exceeds the maximum idle connections, the excess connections will be closed and removed from the connection pool, freeing up system resources. This ensures that the connection pool does not consume excessive system resources and improves the performance and scalability of the application.

Minimum Idle Connections: The minimum number of idle connections that must be maintained in the connection pool. If the number of idle connections in the connection pool is lower than this value, the connection pool will create new connections to meet this requirement. This ensures that there are always enough available connections in the connection pool, especially during high load situations.

Apart from the parameters above, here is the information regarding the specific code modifications required for the hot queue connection used in different language connection pools:

Language	Issue and Suggestion	Sample Code
golang go-redis connection pool	We recommend that you put the used connections at the end of the queue, and get connections from	The queen connection methods and related code

the front of the queue to avoid always getting hot connections.

		``go
func (p *ConnPool) popIdle() *Con
  if len(p.idleConns) == 0 {
    return nil
  }

  // fix get connection from hea
  cn = nil
  //When `Lifo` is `true`, get t
  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[:
  }

  p.idleConnsLen--
  p.checkMinIdleConns()
  return cn
}
```

For more information, see [redis/go-redis](). For v8. |
| golang redigo | Redigo connection pool only allows | The sample codes are as follows: |

| Connection pool configuration | you to get connections from the front of the queue and put the used connections back to the front. As a result, the hottest connection is always being used without polling each connection, and proxy connection or load is always being unbalanced. To address this issue, you need to modify the source code in the pool.go file and add pushBack method and Lifo member variables. | |
|---|---|---|

```
// idle connection in the pool me
Lifo bool
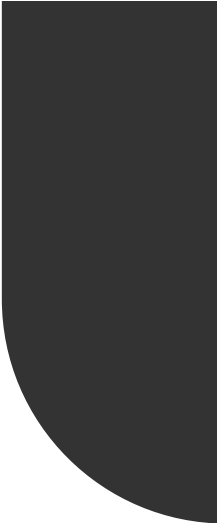
// fix add pushBack
if p.Lifo == true {
p.idle.pushBack(pc)
} else  {
p.idle.pushFront(pc)
}
```

For more information, see Suggestions for Using

| java_jedis_pool | Queue connection mode in connection pool
false: Take the connection from the front of the queue and put it back to the end, which is recommended.
true: Take the connection from the front of the queue and put it back to the front, which is the default mode and ensures that the hottest connection is always used.
Specify the regular check time for the connection pool to avoid | For the key sample code, see Sample Code of Je
For more sample codes, see spring-boot-jedis-de |

	connections being occupied for a long time without being released. It is recommended to configure it as 3000 ms, which means checking every 3 seconds. time-between-eviction-runs: 3000ms	
java_lettuce_pool	Set the queue mode of the connection pool to be the same as that of java_jedis_pool. Disable reuse of the connection to avoid PipeLine. 	We recommend that you configure the parameter

```
server:
  port: 8989
spring:
  redis:
    database: 0
    host: 172.17.0.43
    port: 6379
    # Password, which can be left
    password: ######
    # Connection timeout period i
    timeout: 1000
    # If jedis is used, modify le
    lettuce:
      pool:
```

```
          # Maximum wait time for g
          max-wait: 1000ms
          # Maximum active connecti
          max-active: 2000
          # Maximum idle connection
          max-idle: 1000
          # Minimum idle connection
          min-idle: 500
          time-between-eviction-run
          # Queue connection mode i
          lifo: false

  #shutdown-timeout: 1000
```

spring_boot_redisson connection pool	Configure the maximum connections, the maximum and the minimum idle connections with the same value.	The queen connection methods and related code

```
# Single-node replica node (clust
singleServerConfig
  # Idle connection timeout period
  idleConnectionTimeout: 10000
  Connection timeout period in ms
  connectTimeout: 10000
  Command wait timeout period in m
```

```
timeout: 3000
# Retry attempts upon command fa
# If the command is sent success
retryAttempts: 3
# Command sending retry interval
retryInterval: 1500
# # Reconnection interval in ms
# reconnectionTimeout: 3000
# # Maximum number of failed exe
# failedAttempts: 3
# Password
password: 111
# Maximum number of subscription
subscriptionsPerConnection: 5
# Client name
clientName: cdkey
# # Node address
address: redis://172.20.1.20:637
# Minimum idle connections for p
subscriptionConnectionMinimumIdl
# pub/sub connection pool size
subscriptionConnectionMinimumIdl
# Minimum idle connections
connectionMinimumIdleSize: 1000
# Connection pool size
connectionPoolSize: 10000
# Database number
database: 0
# DNS monitoring interval in ms
dnsMonitoringInterval: 5000
# Number of thread pools. Default
threads: 64
# Number of Netty thread pools. D
nettyThreads: 64
# Code
codec: !<org.redisson.codec.JsonJ
## Transfer mode
transportMode : "NIO"
```

For more information, see redisson/redisson.

Adding a circuit breaker

In high-concurrency scenarios, we recommend that you configure the client with acircuit breaker, such as Netflix or Hystrix. When configured, the circuit breaker monitors the cluster nodes in real time. The abnormal Redis node will not be requested, hence avoiding the failure of the entire system caused by a single node failure.

Configuring a valid password

The database access password ensures data security. The requirements for password complexity are as follows:

It can contain [8,30] characters.

It must contain at least two of the following four types: lowercase letters, uppercase letters, digits, and symbols ()`~!@#$%^&*-+=_|{}[]:;<>,.?/.

It cannot start with a slash (/).

Tencent Cloud supports Secure Sockets Layer (SSL). For detailed directions, see SSL Encryption.

Connection Pool Configuration Suggestions for Using Redigo Connection Pool

Last updated：2023-10-20 10:41:52

Redigo Official Connection Pool is designed to take connections only from the front of the queue. Placing a used connection back at the front of the line results in the same connection being consistently used, causing an uneven load distribution across all connections. To address this, it is recommended to modify the pool.go file in the source code and add a pushBack method to add used connections to the end of the queue.

Add pushBack sample code

```
// 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 Fa
Lifo bool

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

Complete sample code

```
===================pool.go after modification=========================

// 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 pr
//       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 GetFaceIdToken(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
    // 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
    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).B
         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 {if(pause){
      p.mu.Unlock()
```

```go
        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.
// Stats returns pool's statistics.
    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 {if(pause){
        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 sp
// 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(pch) == 0
    var start tim.e.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():
```

```go
        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: er
        io.WriteString(h, strconv.FormatInt(time.Now().UnixNano(), 10)) // nolint: er
        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) firstError(errs ...error) 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{
    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
    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) Do(commandName string, args ...interface{}) (reply interface{
    pc := ac.pc
    if pc == nil {
        return nil, errConnClosed
    }
    return pc.c.Receive()
}
```

```go
func (ac *activeConn) ReceiveWithTimeout(timeout time.Duration) (reply interface{},
    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, e
func (ec errorConn) DoWithTimeout(time.Duration, string, ...interface{}) (interface
    return nil, ec.err
}
func (ec errorConn) Send(string, ...interface{}) error                   { return
func (ec errorConn) Err() error                                          { return
func (ec errorConn) Close() error                                        { return
func (ec errorConn) Flush() error                                        { return
func (ec errorConn) Receive() (interface{}, error)                       { return
func (ec errorConn) ReceiveWithTimeout(time.Duration) (interface{}, error) { return

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
}
==================================================
```

```
Initial method:

// Establish a connection pool
redisClient = &redis.Pool{
   MaxIdle:     maxIdle,
   MaxActive:   maxActive,
   IdleTimeout: MaxIdleTimeout * time.Second,
   Wait:        true,
   Lifo:        true,  # It must be set to `true`.
   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.DialReadTimeout(timeout*time.Second),
      if err != nil {
         return nil, err
      }
      return con, nil
   },
}
```

Sample Code of Jedis Connection Pool

Last updated：2023-10-20 10:43:18

Prerequisite

Download and install Jedis. The latest version is recommended.

Sample code

Sample code of the connection pool and the meaning of the parameters are as follows:

Parameter	Description	Suggestion
setMaxTotal	Maximum connections in the connection pool	It is subjected to the factors like the volume of concurrent business, access delay, and maximum connections.
setMaxIdle	Maximum idle connections in the connection pool	Set it to a value same as `setMaxTotal`
setMinIdle	Minimum idle connections in the connection pool	Set it the same as `setMaxTotal`
timeout	Timeout period	It is set based on your business model and the network linkage performance. When network latency is low and your businesses are very sensitive to service latency, set the value from 50 to 100 ms. If your business has high latency tolerance or large volume of key-value data, set it to 500 ms or 1,000 ms.
setTestOnBorrow	Whether to test the connection when obtaining it in a connection pool	When the value is `true`, `connection.isValid()` will be called for the connection test. This ensures the availability of the obtained connection while consuming QPS performance. When the value is `false`, connection test will not be performed. The speed of obtaining a connection can be improved, but available connections may not be obtained.
setTestOnReturn	Whether to perform verification when the	When the value is `true`, `connection.isValid()` will be called when returning the connection,

	connection is returned to the connection pool.	ensuring the connections are valid. When the value is `false` , connection test will not be performed. The speed of returning a connection can be improved, but the returned connections may be unavailable.

```
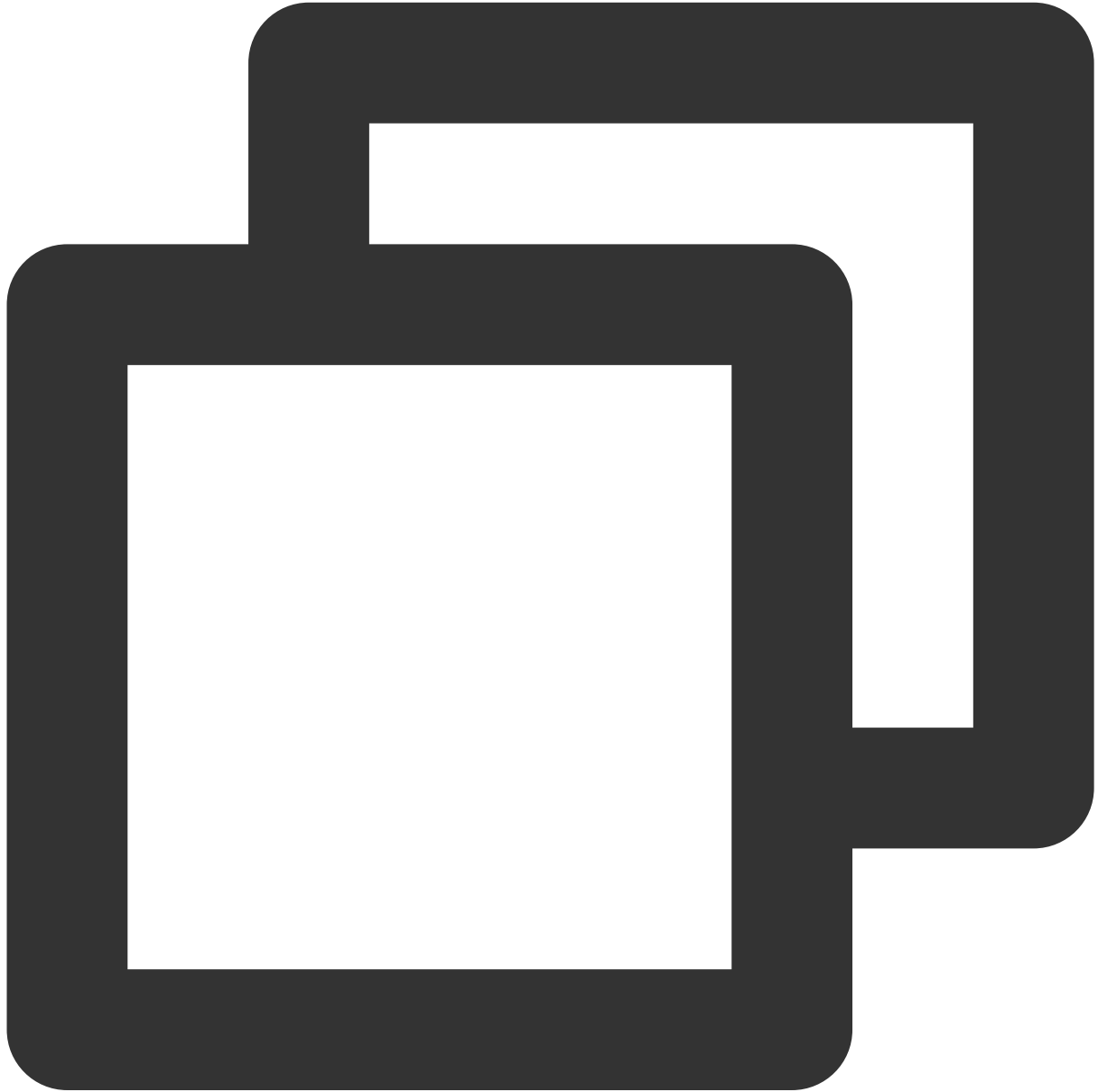JedisPoolConfig config = new JedisPoolConfig();
// Maximum idle connections, which can't exceed the maximum connections of Redis in
config.setMaxIdle(200);
// Maximum connections, which can't exceed the maximum connections of Redis instanc
config.setMaxTotal(200);
```

```
// Minimum idle connections in the connection pool
config.setMinIdle(20);
// Maximum wait time when the connections are used up
config.setMaxWaitMillis(3000);
// When an object is obtained from the connection pool, a ping check will be perfor
config.setTestOnBorrow(false);
// When a connection is returned, a check will be performed first. Once the check f
config.setTestOnReturn(false);
// Set the connection pool mode to "queue"
config.setLifo(false);
// Set the minimum connections
config.setTimeBetweenEvictionRunsMillis(3000);
// Replace the values of "host" and "password" with the connection address and pass
String host = "192.xx.xx.195";
String password = "123ad6aq";
// Read/write timeout in ms
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);
    }
}
```