-

https://pkg.go.dev/github.com/gogf/gf/v2/os/gcache

Set

- key-value
- :

```
Set(ctx context.Context, key interface{}, value interface{},
duration time.Duration) error
```

- slicek1

```
func ExampleCache_Set() {
        c := gcache.New()
        c.Set(ctx, "k1", g.Slice{1, 2, 3, 4, 5, 6, 7, 8, 9}, 0)
        fmt.Println(c.Get(ctx, "k1"))

        // Output:
        // [1,2,3,4,5,6,7,8,9] <nil>
}
```

SetAdapter

- : SetAdapter
-

```
SetAdapter(adapter Adapter)
```

-

```
func ExampleCache_SetAdapters() {
        c := gcache.New()
        adapter := gcache.New()
        c.SetAdapter(adapter)
        c.Set(ctx, "k1", g.Slice{1, 2, 3, 4, 5, 6, 7, 8, 9}, 0)
        fmt.Println(c.Get(ctx, "k1"))

        // Output:
        // [1,2,3,4,5,6,7,8,9] <nil>
 }
```

SetIfNotExist

- : keyvaluetruefalse
- :

```
SetIfNotExist(ctx context.Context, key interface{}, value interface
{}, duration time.Duration) (ok bool, err error)
```

- SetIfNotExist

```
func ExampleCache_SetIfNotExist() {
        c := gcache.New()
        // Write when the key name does not exist, and set the
expiration time to 1000 milliseconds
        k1, err := c.SetIfNotExist(ctx, "k1", "v1", 1000*time.
Millisecond)
        fmt.Println(k1, err)

        // Returns false when the key name already exists
        k2, err := c.SetIfNotExist(ctx, "k1", "v2", 1000*time.
Millisecond)
        fmt.Println(k2, err)

        // Print the current list of key values
        keys1, _ := c.Keys(ctx)
        fmt.Println(keys1)

        // It does not expire if `duration` == 0. It deletes the
`key` if `duration` < 0 or given `value` is nil.
        c.SetIfNotExist(ctx, "k1", 0, -10000)

        // Wait 1 second for K1: V1 to expire automatically
        time.Sleep(1200 * time.Millisecond)

        // Print the current key value pair again and find that K1:
V1 has expired
        keys2, _ := c.Keys(ctx)
        fmt.Println(keys2)

        // Output:
        // true <nil>
        // false <nil>
        // [k1]
        // [<nil>]
 }
```

SetMap

- : `map[interface{}]interface{}`
- :

```
SetMap(ctx context.Context, data map[interface{}]interface{},
duration time.Duration) error
```

-

```
func ExampleCache_SetMap() {
         c := gcache.New()
        // map[interface{}]interface{}
        data := g.MapAnyAny{
                "k1": "v1",
                "k2": "v2",
                "k3": "v3",
        }
        c.SetMap(ctx, data, 1000*time.Millisecond)

        // Gets the specified key value
        v1, _ := c.Get(ctx, "k1")
        v2, _ := c.Get(ctx, "k2")
        v3, _ := c.Get(ctx, "k3")

        fmt.Println(v1, v2, v3)

        // Output:
        // v1 v2 v3
 }
```

Size

- : Size
- :

```
Size(ctx context.Context) (size int, err error)
```

-

```
func ExampleCache_Size() {
        c := gcache.New()

        // Add 10 elements without expiration
        for i := 0; i < 10; i++ {
                c.Set(ctx, i, i, 0)
        }

        // Size returns the number of items in the cache.
        n, _ := c.Size(ctx)
        fmt.Println(n)

        // Output:
        // 10
}
```

Update

- : Updatekeykeyexistfalse
- :

```
Update(ctx context.Context, key interface{}, value interface{})
(oldValue *gvar.Var, exist bool, err error)
```

- SetMapUpdatekeyvalue

```
func ExampleCache_Update() {
        c := gcache.New()
        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2", "k3":
"v3"}, 0)

        k1, _ := c.Get(ctx, "k1")
        fmt.Println(k1)
        k2, _ := c.Get(ctx, "k2")
        fmt.Println(k2)
        k3, _ := c.Get(ctx, "k3")
        fmt.Println(k3)

        re, exist, _ := c.Update(ctx, "k1", "v11")
        fmt.Println(re, exist)

        re1, exist1, _ := c.Update(ctx, "k4", "v44")
        fmt.Println(re1, exist1)

        kup1, _ := c.Get(ctx, "k1")
        fmt.Println(kup1)
        kup2, _ := c.Get(ctx, "k2")
        fmt.Println(kup2)
        kup3, _ := c.Get(ctx, "k3")
        fmt.Println(kup3)

        // Output:
        // v1
        // v2
        // v3
        // v1 true
        //  false
        // v11
        // v2
        // v3
 }
```

UpdateExpire

- : `UpdateExpirekeykey-1`
- :

```
UpdateExpire(ctx context.Context, key interface{}, duration time.
Duration) (oldDuration time.Duration, err error)
```

- `UpdateExpirekey`

```
func ExampleCache_UpdateExpire() {
        c := gcache.New()
        c.Set(ctx, "k1", "v1", 1000*time.Millisecond)
        expire, _ := c.GetExpire(ctx, "k1")
        fmt.Println(expire)

        c.UpdateExpire(ctx, "k1", 500*time.Millisecond)

        expire1, _ := c.GetExpire(ctx, "k1")
        fmt.Println(expire1)

        // Output:
        // 1s
        // 500ms
}
```

Values

- : Values
- :

```
Values(ctx context.Context) (values []interface{}, err error)
```

-

```
func ExampleCache_Values() {
        c := gcache.New()

        c.Set(ctx, "k1", g.Map{"k1": "v1", "k2": "v2"}, 0)

        // Values returns all values in the cache as slice.
        data, _ := c.Values(ctx)
        fmt.Println(data)

        // May Output:
        // [map[k1:v1 k2:v2]]
}
```

Close

- : GC
- :

```
Close(ctx context.Context) error
```

- Close

```
func ExampleCache_Close() {
         c := gcache.New()

        c.Set(ctx, "k1", "v", 0)
        data, _ := c.Get(ctx, "k1")
        fmt.Println(data)

        // Close closes the cache if necessary.
        c.Close(ctx)

        data1, _ := c.Get(ctx, "k1")

        fmt.Println(data1)

         // Output:
        // v
        // v
}
```

Contains

- : keyContainstruefalse
- :

```
Contains(ctx context.Context, key interface{}) (bool, error)
```

-

```
func ExampleCache_Contains() {
         c := gcache.New()

        // Set Cache
        c.Set(ctx, "k", "v", 0)

        data, _ := c.Contains(ctx, "k")
        fmt.Println(data)

        // return false
        data1, _ := c.Contains(ctx, "k1")
        fmt.Println(data1)

         // Output:
        // true
        // false
}
```

Data

- : map('key':'value')
- :

```
Data(ctx context.Context) (data map[interface{}]interface{}, err
error)
```

- map[interface{}]interface{}

```
func ExampleCache_Data() {
          c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1"}, 0)
        //c.Set(ctx, "k5", "v5", 0)

        data, _ := c.Data(ctx)
        fmt.Println(data)

         // Output:
        // map[k1:v1]
}
```

Get

- : Getkeynil
- :

```
Get(ctx context.Context, key interface{}) (*gvar.Var, error)
```

-

```
func ExampleCache_Get() {
          c := gcache.New()

        // Set Cache Object
        c.Set(ctx, "k1", "v1", 0)

        data, _ := c.Get(ctx, "k1")
        fmt.Println(data)
         // Output:
        // v1
}
```

GetExpire

- : GetExpirekeykey0key-1
- :

```
GetExpire(ctx context.Context, key interface{}) (time.Duration,
error)
```

-

```
func ExampleCache_GetExpire() {
    c := gcache.New()

        // Set cache without expiration
        c.Set(ctx, "k", "v", 10000*time.Millisecond)

        expire, _ := c.GetExpire(ctx, "k")
        fmt.Println(expire)

         // Output:
        // 10s
}
```

GetOrSet

- : keykey-valuekey
- :

```
GetOrSet(ctx context.Context, key interface{}, value interface{},
duration time.Duration) (result *gvar.Var, err error)
```

- GetOrSetkeyduration

```
func ExampleCache_GetOrSet() {
    c := gcache.New()

        data, _ := c.GetOrSet(ctx, "k", "v", 10000*time.Millisecond)
        fmt.Println(data)

        data1, _ := c.Get(ctx, "k")
        fmt.Println(data1)

        // Output:
        // v
        // v
}
```

GetOrSetFunc

- : keykeyfunckeykey
- :

```
GetOrSetFunc(ctx context.Context, key interface{}, f func(ctx
context.Context) (interface{}, error), duration time.Duration)
(result *gvar.Var, err error)
```

- k1funck2nil

```
func ExampleCache_GetOrSetFunc() {
    c := gcache.New()

        c.GetOrSetFunc(ctx, "k1", func(ctx context.Context) (value
interface{}, err error) {
                return "v1", nil
        }, 10000*time.Millisecond)
        v, _ := c.Get(ctx, "k1")
        fmt.Println(v)

        c.GetOrSetFunc(ctx, "k2", func(ctx context.Context) (value
interface{}, err error) {
                return nil, nil
        }, 10000*time.Millisecond)
        v1, _ := c.Get(ctx, "k2")
        fmt.Println(v1)

        // Output:
        // v1
}
```

GetOrSetFuncLock

- : GetOrSetFunc
- :

```
GetOrSetFuncLock(ctx context.Context, key interface{}, f func(ctx
context.Context) (interface{}, error), duration time.Duration)
(result *gvar.Var, err error)
```

- 1func2

```
func ExampleCache_GetOrSetFuncLock() {
    c := gcache.New()

        c.GetOrSetFuncLock(ctx, "k1", func(ctx context.Context)
(value interface{}, err error) {
                return "v1", nil
        }, 0)
        v, _ := c.Get(ctx, "k1")
        fmt.Println(v)

        c.GetOrSetFuncLock(ctx, "k1", func(ctx context.Context)
(value interface{}, err error) {
                return "update v1", nil
        }, 0)
        v, _ = c.Get(ctx, "k1")
        fmt.Println(v)

        c.Remove(ctx, g.Slice{"k1"}...)

        // Output:
        // v1
        // v1
}
```

Keys

- : key(Slice)
- :

```
Keys(ctx context.Context) (keys []interface{}, err error)
```

-

```
func ExampleCache_Keys() {
    c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1"}, 0)

        // Print the current list of key values
        keys1, _ := c.Keys(ctx)
        fmt.Println(keys1)

        // Output:
        // [k1]
}
```

KeyStrings

- : KeyStrings
- :

```
func (c *Cache) KeyStrings(ctx context.Context) ([]string, error) {
        keys, err := c.Keys(ctx)
        if err != nil {
                return nil, err
        }
        return gconv.Strings(keys), nil
}
```

-

```
func ExampleCache_KeyStrings() {
        c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2"}, 0)

        // KeyStrings returns all keys in the cache as string slice.
        keys,_ := c.KeyStrings(ctx)
        fmt.Println(keys)

        // May Output:
        // [k1 k2]
}
```

Remove

- :
- :

```
Remove(ctx context.Context, keys ...interface{}) (lastValue *gvar.
Var, err error)
```

-

```
func ExampleCache_Remove() {
    c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2"}, 0)

        c.Remove(ctx, "k1")

        data, _ := c.Data(ctx)
        fmt.Println(data)

        // Output:
        // map[k2:v2]
}
```

Removes

- :
- :

```
func (c *Cache) Removes(ctx context.Context, keys []interface{})
error {
        _, err := c.Remove(ctx, keys...)
        return err
}
```

-

```
func ExampleCache_Removes() {
    c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2", "k3":
"v3", "k4": "v4"}, 0)

        c.Removes(ctx, g.Slice{"k1", "k2", "k3"})

        data, _ := c.Data(ctx)
        fmt.Println(data)

        // Output:
        // map[k4:v4]
}
```

Clear

- :
- :

```
func (c *Cache) Clear(ctx context.Context) error
```

-

```
func ExampleCache_Clear() {
        c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2", "k3":
"v3", "k4": "v4"}, 0)

        c.Clear(ctx)

        data, _ := c.Data(ctx)
        fmt.Println(data)

        // Output:
        // map[]
}
```

MustGet

- :MustGetkeynil errpanic(err)
- :

```
func (c *Cache) MustGet(ctx context.Context, key interface{}) *gvar.
Var {
        v, err := c.Get(ctx, key)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustGet() {
         c := gcache.New()

         c.Set(ctx, "k1", "v1", 0)
         k2 := c.MustGet(ctx, "k2")

         k1 := c.MustGet(ctx, "k1")
        fmt.Println(k1)

        fmt.Println(k2)

         // Output:
        // v1
        //
}
```

MustGetOrSet

- : keykey-valuekeyerrpanic(err)
- :

```
func (c *Cache) MustGetOrSet(ctx context.Context, key interface{},
value interface{}, duration time.Duration) *gvar.Var {
        v, err := c.GetOrSet(ctx, key, value, duration)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustGetOrSet() {

        // Create a cache object,
        // Of course, you can also easily use the gcache package
method directly
        c := gcache.New()

        // MustGetOrSet acts like GetOrSet, but it panics if any
error occurs.
        k1 := c.MustGetOrSet(ctx, "k1", "v1", 0)
        fmt.Println(k1)

        k2 := c.MustGetOrSet(ctx, "k1", "v2", 0)
        fmt.Println(k2)

        // Output:
        // v1
        // v1

}
```

MustGetOrSetFunc

- : keykeyfunckeykeyerrpanic(err)
- :

```
func (c *Cache) MustGetOrSetFunc(ctx context.Context, key interface
{}, f func(ctx context.Context) (interface{}, error), duration time.
Duration) *gvar.Var {
        v, err := c.GetOrSetFunc(ctx, key, f, duration)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustGetOrSetFunc() {
        c := gcache.New()

        c.MustGetOrSetFunc(ctx, 1, func(ctx context.Context)
(interface{}, error) {
                return 111, nil
        }, 10000*time.Millisecond)
        v := c.MustGet(ctx, 1)
        fmt.Println(v)

        c.MustGetOrSetFunc(ctx, 2, func(ctx context.Context)
(interface{}, error) {
                return nil, nil
        }, 10000*time.Millisecond)
        v1 := c.MustGet(ctx, 2)
        fmt.Println(v1)

        // Output:
        // 111
        //
}
```

MustGetOrSetFuncLock

- : `MustGetOrSetFuncerrpanic(err)`
- :

```
func (c *Cache) MustGetOrSetFuncLock(ctx context.Context, key
interface{}, f func(ctx context.Context) (interface{}, error),
duration time.Duration) *gvar.Var {
        v, err := c.GetOrSetFuncLock(ctx, key, f, duration)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustGetOrSetFuncLock() {
        c := gcache.New()

        c.MustGetOrSetFuncLock(ctx, "k1", func(ctx context.Context)
(interface{}, error) {
                return "v1", nil
        }, 0)
        v := c.MustGet(ctx, "k1")
        fmt.Println(v)

        // Modification failed
        c.MustGetOrSetFuncLock(ctx, "k1", func(ctx context.Context)
(interface{}, error) {
                return "update v1", nil
        }, 0)
        v = c.MustGet(ctx, "k1")
        fmt.Println(v)

        // Output:
        // v1
        // v1
}
```

MustContains

- : keyContainstruefalseerrpanic(err)
- :

```
func (c *Cache) MustContains(ctx context.Context, key interface{})
bool {
        v, err := c.Contains(ctx, key)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustContains() {
        c := gcache.New()

        // Set Cache
        c.Set(ctx, "k", "v", 0)

        // Contains returns true if `key` exists in the cache, or
else returns false.
        // return true
        data := c.MustContains(ctx, "k")
        fmt.Println(data)

        // return false
        data1 := c.MustContains(ctx, "k1")
        fmt.Println(data1)

        // Output:
        // true
        // false

}
```

MustGetExpire

- : MustGetExpirekeykey0key-1errpanic(err)
- :

```
func (c *Cache) MustGetExpire(ctx context.Context, key interface{})
time.Duration {
        v, err := c.GetExpire(ctx, key)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustGetExpire() {
        c := gcache.New()

        // Set cache without expiration
        c.Set(ctx, "k", "v", 10000*time.Millisecond)

        // MustGetExpire acts like GetExpire, but it panics if any
error occurs.
        expire := c.MustGetExpire(ctx, "k")
        fmt.Println(expire)

        // May Output:
        // 10s
}
```

MustSize

- : MustSizeerrpanic(err)
- :

```
func (c *Cache) MustSize(ctx context.Context) int {
        v, err := c.Size(ctx)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustSize() {
        c := gcache.New()

        // Add 10 elements without expiration
        for i := 0; i < 10; i++ {
                c.Set(ctx, i, i, 0)
        }

        // Size returns the number of items in the cache.
        n := c.MustSize(ctx)
        fmt.Println(n)

        // Output:
        // 10
}
```

MustData

- : `map('key':'value')errpanic(err)`
- :

```
func (c *Cache) MustData(ctx context.Context) map[interface{}]
interface{} {
        v, err := c.Data(ctx)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustData() {
        c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1"}, 0)

        data := c.MustData(ctx)
        fmt.Println(data)

        // Set Cache
        c.Set(ctx, "k5", "v5", 0)
        data1, _ := c.Get(ctx, "k1")
        fmt.Println(data1)

        // Output:
        // map[k1:v1]
        // v1
}
```

MustKeys

- : `MustKeys(slice)errpanic(err)`
- :

```
func (c *Cache) MustKeys(ctx context.Context) []interface{} {
        v, err := c.Keys(ctx)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustKeys() {
        c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2"}, 0)

        // MustKeys acts like Keys, but it panics if any error
occurs.
        keys1 := c.MustKeys(ctx)
        fmt.Println(keys1)

        // May Output:
        // [k1 k2]

}
```

MustKeyStrings

- : MustKeyStrings(slice)errpanic(err)
- :

```
func (c *Cache) MustKeyStrings(ctx context.Context) []string {
        v, err := c.KeyStrings(ctx)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustKeyStrings() {
        c := gcache.New()

        c.SetMap(ctx, g.MapAnyAny{"k1": "v1", "k2": "v2"}, 0)

        // MustKeyStrings returns all keys in the cache as string
slice.
        // MustKeyStrings acts like KeyStrings, but it panics if any
error occurs.
        keys := c.MustKeyStrings(ctx)
        fmt.Println(keys)

        // May Output:
        // [k1 k2]
}
```

MustValues

- : MustValues(slice)errpanic(err)
- :

```
func (c *Cache) MustValues(ctx context.Context) []interface{} {
        v, err := c.Values(ctx)
        if err != nil {
                panic(err)
        }
        return v
}
```

-

```
func ExampleCache_MustValues() {
        c := gcache.New()

        // Write value
        c.Set(ctx, "k1", "v1", 0)

        // Values returns all values in the cache as slice.
        data := c.MustValues(ctx)
        fmt.Println(data)

        // Output:
        // [v1]
}
```