Blog Concurrent und Linked Maps

Warum Maps

Eine der am meisten verwendeten Datenstruktur in der Informatik ist die Hash-Tabelle. Es gibt viele Hash-Tabellen-Implementierungen mit unterschiedlichen Eigenschaften, aber im Allgemeinen bieten sie schnelle Such-, Hinzufügungs- und Löschvorgänge. Go bietet einen integrierten Map-Typ, der eine Hash-Tabelle implementiert. Mehr Infos zur Go Map finden Sie unter https://blog.golang.org/mapsMit Maps können beliebige Daten basierend auf einem Key dynamisch abgelegt werden. Scriptsprachen wie Javascript basieren zentral auf dem Map Konzept und speichern die Daten in der Regel über key/value-Pairs.Der Zugriff auf die Keys erfolgt entweder über einen Tree oder die Hash-Tabelle basierend auf einem Hashcode. Der Hash-Zugriff ist in der Regel der schnellere und damit effizientere Mechanismus. Solcher basiert auf einem indexierten Array:Der Hashcode wird auf die Array Grösse modulo reduziert. Falls eine Kollision mit einem anderen Key besteht, werden beide in einer Liste geführt. Ein Hashcode ist somit nie eindeutig, sollte aber einer Gaußschen Normalverteilung entsprechen. Sobald zu viele Kollisionen bestehen, wird das Hash Array verdoppelt und reorganisiert. Dadurch werden die Kollisionen wieder reduziert. Bei der Suche nach einem Key wird zuerst der Hashcode bestimmt und gesucht, bei Kollisionen wird der Key aus der Liste inhaltlich (equals) abgecheckt und bestimmt. Als Key Types sind in golang nur die folgende zulässig:booleannumericstringpointechannelinterface typesstructs – if all it’s field type is comparablearray – if the type of value of array element is comparableNicht erlaubt sind Slice, Map und Function Types.Dieser Blog zeigt eine Kapselung der Standard Go Map und diverse Varianten wie die Concurrent- und Linked-Map.

Standard Map

In einem ersten Schritt kapseln wir die Go Map über das eigene Struct StdMap:// Package collections definitions
package collections

import "fmt"

// StdMap struct
type StdMap struct {
 Map map[string]interface{}
}

// NewMap func
func NewMap() *StdMap {
 return NewMapByParam(nil)
}

// NewMapByParam func
func NewMapByParam(paramMap map[string]interface{}) *StdMap {
 m := new(StdMap)
 if paramMap == nil {
  m.Map = make(map[string]interface{})
 } else {
  m.Map = paramMap
 }
 return m
}

// IsNull func
func (m *StdMap) IsNull() bool {
 return m.Map == nil
}

// Init func
func (m *StdMap) Init() {
 m.Map = make(map[string]interface{})
}

// HasKey func
func (m *StdMap) HasKey(key string) bool {
 _, ok := m.Map[key]
 return ok
}

// Value func
func (m *StdMap) Value(key string) interface{} {
 return m.Map[key]
}

// ValueOrDefault func
func (m *StdMap) ValueOrDefault(key string, defaultValue interface{}) interface{} {
 val, ok := m.Map[key]
 if ok {
  return val
 }
 return defaultValue
}

// String func
func (m *StdMap) String() string {
 return fmt.Sprintf("%+v", m.Map)
}Die Unit Tests zu StdMap:// Package collections
package collections

import (
 "fmt"
 "testing"
)

func Test_StdMapNewMap(t *testing.T) {
 m := NewMap()
 m.Map["one"] = 1
 m.Map["two"] = 2
 m.Map["three"] = 3
 m.Map["four"] = 4

 fmt.Printf("m = %s
", m.String())
}

func Test_StdMapNewMapByParam(t *testing.T) {
 params := make(map[string]interface{})
 params["one"] = "one"
 m := NewMapByParam(params)
 fmt.Printf("m = %s
", m.String())
}

func TestStdMap_HasKey(t *testing.T) {
 m := NewMap()
 m.Map["one"] = 1
 m.Map["two"] = 2
 m.Map["three"] = 3
 m.Map["four"] = 4

 if !m.HasKey("one") {
  t.Error("m.HasKey("one") is false");
 }

 if m.HasKey("One") {
  t.Error("m.HasKey("One") is true");
 }
}

func Test_StdMapValueOrDefault(t *testing.T) {
 m := NewMap()
 m.Map["one"] = 1
 if m.ValueOrDefault("one", 0) != 1 {
  t.Error("m.ValueOrDefault("one") != 1");
 }
 if m.ValueOrDefault("One", 0) != 0 {
  t.Error("m.ValueOrDefault("One") != 0");
 }
 if m.ValueOrDefault("one", 0) == 0 {
  t.Error("m.ValueOrDefault("one") == 0");
 }
}Die StdMap kann nun angewendet und beliebig erweitert werden. Wir belassen hier das Minimum an Funktionalität.

Concurrent Map

Golang Maps sind nicht Concurrent Safe. Dies betrifft auch unsere StdMap. Mit der StdSyncMap bauen wir eine Concurrent Safe Map:// Package collections definitions
package collections

import (
 "sync"
)

// StdSyncMap struct
type StdSyncMap struct {
 *StdMap // anonymous field StdMap
 lock sync.Mutex
}

// NewSyncMap func
func NewSyncMap() *StdSyncMap {
 m := new(StdSyncMap)
 m.StdMap = NewMap()
 return m
}

// NewSyncMapByParam func
func NewSyncMapByParam(paramMap map[string]interface{}) *StdSyncMap {
 m := new(StdSyncMap)
 m.StdMap = NewMapByParam(paramMap)
 return m
}

// IsNull func
func (m *StdSyncMap) IsNull() bool {
 m.lock.Lock()
 defer m.lock.Unlock()
 return m.StdMap.IsNull()
}

// Init func
func (m *StdSyncMap) Init() {
 m.lock.Lock()
 defer m.lock.Unlock()
 m.StdMap.Init()
}

// HasKey func
func (m *StdSyncMap) HasKey(key string) bool {
 m.lock.Lock()
 defer m.lock.Unlock()
 return m.StdMap.HasKey(key)
}

// Value func
func (m *StdSyncMap) Value(key string) interface{} {
 m.lock.Lock()
 defer m.lock.Unlock()
 return m.StdMap.Value(key)
}

// String func
func (m *StdSyncMap) String() string {
 m.lock.Lock()
 defer m.lock.Unlock()
 return m.StdMap.String()
}StdSyncMap basiert auf der StdMap, überschreibt aber jede Methode und schützt solche vor gleichzeitigem Merhfachzugriff via sync.Mutex.

Linked Map

Golang Maps sind unordentlich und speichern die Keys zufällig nach dem Hashcode und dem indexierten Array. Falls wir eine Ordnung möchten, müssen wir solche zusätzlich bauen, und hierzu erstellen wir die StdLinkedMap:// Package collections definitions
package collections

import (
 "bytes"
 "fmt"
 "strconv"

 "std.ch/errors"
)

// StdLinkedMap struct
type StdLinkedMap struct {
 *StdMap // anonymous field StdMap
 linkedKeys []string
}

// NewLinkedMap func
func NewLinkedMap() *StdLinkedMap {
 m := new(StdLinkedMap)
 m.StdMap = NewMap()
 m.linkedKeys = make([]string, 0)
 return m
}

// NewLinkedMapByParam func
func NewLinkedMapByParam(paramMap map[string]interface{}) *StdLinkedMap {
 m := new(StdLinkedMap)
 m.StdMap = NewMapByParam(paramMap)
 m.linkedKeys = make([]string, 0)
 return m
}

// Add func
func (m *StdLinkedMap) SetValue(key string, value interface{}) {
 if !m.HasKey(key) {
  m.StdMap.Map[key] = value
  m.linkedKeys = append(m.linkedKeys, key)
 }
}

// LinkedKeys func
func (m *StdLinkedMap) LinkedKeys() ([]string, int) {
 length := len(m.linkedKeys)
 return m.linkedKeys, length
}

// LinkedKey func
func (m *StdLinkedMap) LinkedKey(pos int) (string, error) {
 length := len(m.linkedKeys)
 if pos < 0 || pos >= length {
  return "", errors.NewErrorOutOfRange("pos " + strconv.Itoa(pos) + " is out of range")
 }
 return m.linkedKeys[pos], nil
}

// String func
func (m *StdLinkedMap) String() string {
 var buffer bytes.Buffer
 buffer.WriteRune('[')
 for i, key := range m.linkedKeys {
  if i > 0 {
  buffer.WriteRune(',')
  }
  buffer.WriteString(key)
  buffer.WriteRune(':')
  buffer.WriteString(fmt.Sprintf("%+v", m.Map[key]))
  i++
 }
 buffer.WriteRune(']')
 return buffer.String()
}Die Methode String() zeigt die Keys geordnet an, diese Ordnung basiert auf dem linkedKeys Slice.

Concurrent Linked Map

Die Concurrent Safe Variante der Linked Map mit der struct StdSyncLinkedMap zeigt das folgende Listing:// Package collections definitions
package collections

import (
 "bytes"
 "fmt"
 "strconv"
 "sync"

 "std.ch/errors"
)

// StdSyncLinkedMap struct
type StdSyncLinkedMap struct {
 *StdSyncMap // anonymous field StdLinkedMap
 linkedKeys []string
 lock sync.Mutex
}

// NewSyncLinkedMap func
func NewSyncLinkedMap() *StdSyncLinkedMap {
 m := new(StdSyncLinkedMap)
 m.StdSyncMap = NewSyncMap()
 m.linkedKeys = make([]string, 0)
 return

m
}

// NewSyncLinkedMapByParam func
func NewSyncLinkedMapByParam(paramMap map[string]interface{}) *StdSyncLinkedMap {
 m := new(StdSyncLinkedMap)
 m.StdSyncMap = NewSyncMapByParam(paramMap)
 m.linkedKeys = make([]string, 0)
 return m
}

// Add func
func (m *StdSyncLinkedMap) SetValue(key string, value interface{}) {
 m.lock.Lock()
 defer m.lock.Unlock()
 if !m.HasKey(key) {
  m.StdSyncMap.Map[key] = value
 m.linkedKeys = append(m.linkedKeys, key)
 }
}

// LinkedKeys func
func (m *StdSyncLinkedMap) LinkedKeys() ([]string, int) {
 m.lock.Lock()
 defer m.lock.Unlock()
 length := len(m.linkedKeys)
 return m.linkedKeys, length
}

// LinkedKey func
func (m *StdSyncLinkedMap) LinkedKey(pos int) (string, error) {
 m.lock.Lock()
 defer m.lock.Unlock()
 length := len(m.linkedKeys)
 if pos < 0 || pos >= length {
  return "", errors.NewErrorOutOfRange("pos " + strconv.Itoa(pos) + " is out of range")
 }
 return m.linkedKeys[pos], nil
}

// ToString func
func (m *StdSyncLinkedMap) ToString() string {
 m.lock.Lock()
 defer m.lock.Unlock()
 var buffer bytes.Buffer
 buffer.WriteRune('[')
 for i, key := range m.linkedKeys {
  if i > 0 {
   buffer.WriteRune(',')
  }
 buffer.WriteString(key)
  buffer.WriteRune(':')
 buffer.WriteString(fmt.Sprintf("%+v", m.Map[key]))
  i++
 }
 buffer.WriteRune(']')
 return buffer.String()
}Damit sind wir nun bereit für den Einsatz der Map in den wichtigsten Anforderungsgebieten.

Files / Dateien

Sie finden die originalen Dateien zu diesem Blog unter dem Link https://github.com/stackch/demo/tree/master/gomapblog.

Feedback

War dieser Blog für Sie wertvoll. Wir danken für jede Anregung und Feedback

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Das Copyright für sämtliche Inhalte dieser Website liegt bei Simtech AG, Schweiz.
Beachten Sie auch unsere Hinweise zum Urheberrecht, Datenschutz und Haftungsauschluss.
Jeder Hinweis auf Fehler nehmen wir gerne entgegen.

Copyright

2024 Simtech AG, All rights reserved, Powered by stack.ch written in Golang by Daniel Schmutz

https://www.simtech-ag.ch/kollisionen