maphash

package | standard library

Version: go1.20.1 Latest | Published: Feb 14, 2023 | License: BSD-3-Clause | Imports: 1 | Imported by: 240

Details Valid go.mod file ? Redistributable license ? Tagged version ? Stable version ?

Learn more

Repository cs.opensource.google/go/go

Links Report a Vulnerability

Documentation

Documentation

Overview

Package maphash provides hash functions on byte sequences. These hash functions are intended to be used to implement hash tables or other data structures that need to map arbitrary strings or byte sequences to a uniform distribution on unsigned 64-bit integers. Each different instance of a hash table or data structure should use its own Seed.

The hash functions are not cryptographically secure. (See crypto/sha256 and crypto/sha512 for cryptographic use.)

▶ Example

Index

Examples

Package

Constants

This section is empty.

Variables

This section is empty.

Functions

func Bytes

added in go1.19

```
func Bytes(seed Seed, b []byte) uint64
```

Bytes returns the hash of b with the given seed.

Bytes is equivalent to, but more convenient and efficient than:

```
var h Hash
h.SetSeed(seed)
h.Write(b)
return h.Sum64()
```

func String

added in go1.19

```
func String(seed Seed, s string) uint64
```

String returns the hash of s with the given seed.

String is equivalent to, but more convenient and efficient than:

```
var h Hash
h.SetSeed(seed)
h.WriteString(s)
return h.Sum64()
```

Types

type Hash

```
type Hash struct {
    // contains filtered or unexported fields
}
```

A Hash computes a seeded hash of a byte sequence.

The zero Hash is a valid Hash ready to use. A zero Hash chooses a random seed for itself during the first call to a Reset, Write, Seed, or Sum64 method. For control over the seed, use SetSeed.

The computed hash values depend only on the initial seed and the sequence of bytes provided to the Hash object, not on the way in which the bytes are provided. For example, the three sequences

```
h.Write([]byte{'f','o','o'})
h.WriteByte('f'); h.WriteByte('o'); h.WriteByte('o')
h.WriteString("foo")
```

all have the same effect.

Hashes are intended to be collision-resistant, even for situations where an adversary controls the byte sequences being hashed.

A Hash is not safe for concurrent use by multiple goroutines, but a Seed is. If multiple goroutines must compute the same seeded hash, each can declare its own Hash and call SetSeed with a common Seed.

func (*Hash) BlockSize

```
func (h *Hash) BlockSize() int
```

BlockSize returns h's block size.

func (*Hash) Reset

```
func (h *Hash) Reset()
```

Reset discards all bytes added to h. (The seed remains the same.)

func (*Hash) Seed

```
func (h *Hash) Seed() Seed
```

Seed returns h's seed value.

func (*Hash) SetSeed

```
func (h *Hash) SetSeed(seed Seed)
```

SetSeed sets h to use seed, which must have been returned by MakeSeed or by another Hash's Seed method. Two Hash objects with the same seed behave identically. Two Hash objects with different seeds will very likely behave differently. Any bytes added to h before this call will be discarded.

func (*Hash) Size

```
func (h *Hash) Size() int
```

Size returns h's hash value size, 8 bytes.

func (*Hash) Sum

```
func (h *Hash) Sum(b []byte) []byte
```

Sum appends the hash's current 64-bit value to b. It exists for implementing hash.Hash. For direct calls, it is more efficient to use Sum64.

func (*Hash) Sum64

```
func (h *Hash) Sum64() uint64
```

Sum64 returns h's current 64-bit value, which depends on h's seed and the sequence of bytes added to h since the last call to Reset or SetSeed.

All bits of the Sum64 result are close to uniformly and independently distributed, so it can be safely reduced by using bit masking, shifting, or modular arithmetic.

func (*Hash) Write

```
func (h *Hash) Write(b []byte) (int, error)
```

Write adds b to the sequence of bytes hashed by h. It always writes all of b and never fails; the count and error result are for implementing io.Writer.

func (*Hash) WriteByte

```
func (h *Hash) WriteByte(b byte) error
```

WriteByte adds b to the sequence of bytes hashed by h. It never fails; the error result is for implementing io.ByteWriter.

func (*Hash) WriteString

```
func (h *Hash) WriteString(s string) (int, error)
```

WriteString adds the bytes of s to the sequence of bytes hashed by h. It always writes all of s and never fails; the count and error result are for implementing io.StringWriter.

type Seed

```
type Seed struct {
    // contains filtered or unexported fields
```

```
}
```

A Seed is a random value that selects the specific hash function computed by a Hash. If two Hashes use the same Seeds, they will compute the same hash values for any given input. If two Hashes use different Seeds, they are very likely to compute distinct hash values for any given input.

A Seed must be initialized by calling MakeSeed. The zero seed is uninitialized and not valid for use with Hash's SetSeed method.

Each Seed value is local to a single process and cannot be serialized or otherwise recreated in a different process.

func MakeSeed

```
func MakeSeed() Seed
```

MakeSeed returns a new random seed.

Source Files

View all

maphash.go

Connect

Twitter

GitHub

Slack

r/golang

Meetup

Golang Weekly

Report an Issue

Google