# Golang标准库——hash

## hash

hash包提供hash函数的接口。

### type Hash

// Hash is the common interface implemented by all hash functions.  
//  
// Hash implementations in the standard library (e.g. hash/crc32 and  
// crypto/sha256) implement the encoding.BinaryMarshaler and  
// encoding.BinaryUnmarshaler interfaces. Marshaling a hash implementation  
// allows its internal state to be saved and used for additional processing  
// later, without having to re-write the data previously written to the hash.  
// The hash state may contain portions of the input in its original form,  
// which users are expected to handle for any possible security implications.  
//  
// Compatibility: Any future changes to hash or crypto packages will endeavor  
// to maintain compatibility with state encoded using previous versions.  
// That is, any released versions of the packages should be able to  
// decode data written with any previously released version,  
// subject to issues such as security fixes.  
// See the Go compatibility document for background: https://golang.org/doc/go1compat  
type Hash interface {  
 // 通过嵌入的匿名io.Writer接口的Write方法向hash中添加更多数据，永远不返回错误  
 io.Writer  
  
 // 返回添加b到当前的hash值后的新切片，不会改变底层的hash状态  
 Sum(b []byte) []byte  
  
 // 重设hash为无数据输入的状态  
 Reset()  
  
 // 返回Sum会返回的切片的长度  
 Size() int  
  
 // 返回hash底层的块大小；Write方法可以接受任何大小的数据，  
 // 但提供的数据是块大小的倍数时效率更高  
 BlockSize() int  
}

Hash是一个被所有hash函数实现的公共接口。

### type Hash32

// Hash32 is the common interface implemented by all 32-bit hash functions.  
type Hash32 interface {  
 Hash  
 Sum32() uint32  
}

Hash32是一个被所有32位hash函数实现的公共接口。

### type Hash64

// Hash64 is the common interface implemented by all 64-bit hash functions.  
type Hash64 interface {  
 Hash  
 Sum64() uint64  
}

Hash64是一个被所有64位hash函数实现的公共接口。

* adler32
* crc32
* crc64
* fnv

## adler32

adler32包实现了Adler-32校验和算法，参见[RFC 1950](https://links.jianshu.com/go?to=http://tools.ietf.org/html/rfc1950" \t "_blank)：

Adler-32由两个每字节累积的和组成：

s1是所有字节的累积，s2是所有s1的累积。两个累积值都取65521的余数。s1初始为1，s2初始为0。

Afler-32校验和保存为s2\*65536 + s1。（最高有效字节在前/大端在前）

### Constants

// The size of an Adler-32 checksum in bytes.  
const *Size* = 4

Adler-32校验和的字节数。

### func Checksum

// Checksum returns the Adler-32 checksum of data.  
func Checksum(data []byte) uint32 { return uint32(update(1, data)) }

返回数据data的Adler-32校验和。

### func New

// New returns a new hash.Hash32 computing the Adler-32 checksum. Its  
// Sum method will lay the value out in big-endian byte order. The  
// returned Hash32 also implements encoding.BinaryMarshaler and  
// encoding.BinaryUnmarshaler to marshal and unmarshal the internal  
// state of the hash.  
func New() hash.Hash32 {  
 d := new(digest)  
 d.Reset()  
 return d  
}

返回一个计算Adler-32校验和的hash.Hash32接口。

## crc32

crc32包实现了32位循环冗余校验（CRC-32）的校验和算法，参见：[http://en.wikipedia.org/wiki/Cyclic\_redundancy\_check](https://links.jianshu.com/go?to=http://en.wikipedia.org/wiki/Cyclic_redundancy_check" \t "_blank)

### Constants

// Predefined polynomials.  
const (  
 // 最常用的CRC-32多项式；用于以太网、v.42、fddi、gzip、zip、png、mpeg-2……  
 *IEEE* = 0xedb88320  
  
 // 卡斯塔尼奥利多项式，用在iSCSI；有比IEEE更好的错误探测特性  
 // https://dx.doi.org/10.1109/26.231911  
 *Castagnoli* = 0x82f63b78  
  
 // 库普曼多项式；错误探测特性也比IEEE好  
 // https://dx.doi.org/10.1109/DSN.2002.1028931  
 *Koopman* = 0xeb31d82e  
)

预定义的多项式。

// The size of a CRC-32 checksum in bytes.  
const *Size* = 4

CRC-32校验和的字节长度。

### Variables

// IEEETable is the table for the IEEE polynomial.  
var IEEETable = simpleMakeTable(*IEEE*)

IEEETable是IEEE多项式对应的Table。

### type Table

// Table is a 256-word table representing the polynomial for efficient processing.  
type Table [256]uint32

长度256的uint32切片，代表一个用于高效运作的多项式。

#### func MakeTable

// MakeTable returns a Table constructed from the specified polynomial.  
// The contents of this Table must not be modified.  
func MakeTable(poly uint32) \*Table {  
 switch poly {  
 case *IEEE*:  
 ieeeOnce.Do(ieeeInit)  
 return IEEETable  
 case *Castagnoli*:  
 castagnoliOnce.Do(castagnoliInit)  
 return castagnoliTable  
 }  
 return simpleMakeTable(poly)  
}

返回一个代表poly指定的多项式的Table。

### func Checksum

// Checksum returns the CRC-32 checksum of data  
// using the polynomial represented by the Table.  
func Checksum(data []byte, tab \*Table) uint32 { return Update(0, tab, data) }

返回数据data使用tab代表的多项式计算出的CRC-32校验和。

### func ChecksumIEEE

// ChecksumIEEE returns the CRC-32 checksum of data  
// using the IEEE polynomial.  
func ChecksumIEEE(data []byte) uint32 {  
 ieeeOnce.Do(ieeeInit)  
 return updateIEEE(0, data)  
}

返回数据data使用IEEE多项式计算出的CRC-32校验和。

### func Update

// Update returns the result of adding the bytes in p to the crc.  
func Update(crc uint32, tab \*Table, p []byte) uint32 {  
 switch {  
 case atomic.LoadUint32(&haveCastagnoli) != 0 && tab == castagnoliTable:  
 return updateCastagnoli(crc, p)  
 case tab == IEEETable:  
 // Unfortunately, because IEEETable is exported, IEEE may be used without a  
 // call to MakeTable. We have to make sure it gets initialized in that case.  
 ieeeOnce.Do(ieeeInit)  
 return updateIEEE(crc, p)  
 default:  
 return simpleUpdate(crc, tab, p)  
 }  
}

返回将切片p的数据采用tab表示的多项式添加到crc之后计算出的新校验和。

### func New

// New creates a new hash.Hash32 computing the CRC-32 checksum using the  
// polynomial represented by the Table. Its Sum method will lay the  
// value out in big-endian byte order. The returned Hash32 also  
// implements encoding.BinaryMarshaler and encoding.BinaryUnmarshaler to  
// marshal and unmarshal the internal state of the hash.  
func New(tab \*Table) hash.Hash32 {  
 if tab == IEEETable {  
 ieeeOnce.Do(ieeeInit)  
 }  
 return &digest{0, tab}  
}

创建一个使用tab代表的多项式计算CRC-32校验和的hash.Hash32接口。

### func NewIEEE

// NewIEEE creates a new hash.Hash32 computing the CRC-32 checksum using  
// the IEEE polynomial. Its Sum method will lay the value out in  
// big-endian byte order. The returned Hash32 also implements  
// encoding.BinaryMarshaler and encoding.BinaryUnmarshaler to marshal  
// and unmarshal the internal state of the hash.  
func NewIEEE() hash.Hash32 { return New(IEEETable) }

创建一个使用IEEE多项式计算CRC-32校验和的hash.Hash32接口。

## crc64

crc64包实现64位循环冗余校验或CRC-64校验和。 参见：

[http://en.wikipedia.org/wiki/Cyclic\_redundancy\_check](https://links.jianshu.com/go?to=http://en.wikipedia.org/wiki/Cyclic_redundancy_check" \t "_blank)

### Constants

// The size of a CRC-64 checksum in bytes.  
const *Size* = 8  
  
// Predefined polynomials.  
const (  
 // The ISO polynomial, defined in ISO 3309 and used in HDLC.  
 *ISO* = 0xD800000000000000  
  
 // The ECMA polynomial, defined in ECMA 182.  
 *ECMA* = 0xC96C5795D7870F42  
)

CRC-64校验和的字节数。

### type Table

// Table is a 256-word table representing the polynomial for efficient processing.  
type Table [256]uint64

长度256的uint64切片，代表一个用于高效运作的多项式。

#### func MakeTable

// MakeTable returns a Table constructed from the specified polynomial.  
// The contents of this Table must not be modified.  
func MakeTable(poly uint64) \*Table {  
 buildSlicing8TablesOnce()  
 switch poly {  
 case *ISO*:  
 return &slicing8TableISO[0]  
 case *ECMA*:  
 return &slicing8TableECMA[0]  
 default:  
 return makeTable(poly)  
 }  
}

返回一个代表poly指定的多项式的\*Table。

### func Checksum

// Checksum returns the CRC-64 checksum of data  
// using the polynomial represented by the Table.  
func Checksum(data []byte, tab \*Table) uint64 { return update(0, tab, data) }

返回数据data使用tab代表的多项式计算出的CRC-64校验和。

### func Update

// Update returns the result of adding the bytes in p to the crc.  
func Update(crc uint64, tab \*Table, p []byte) uint64 {  
 return update(crc, tab, p)  
}

返回将切片p的数据采用tab表示的多项式添加到crc之后计算出的新校验和。

### func New

// New creates a new hash.Hash64 computing the CRC-64 checksum using the  
// polynomial represented by the Table. Its Sum method will lay the  
// value out in big-endian byte order. The returned Hash64 also  
// implements encoding.BinaryMarshaler and encoding.BinaryUnmarshaler to  
// marshal and unmarshal the internal state of the hash.  
func New(tab \*Table) hash.Hash64 { return &digest{0, tab} }

创建一个使用tab代表的多项式计算CRC-64校验和的hash.Hash64接口。

## fnv

fnv包实现了FNV-1和FNV-1a（非加密hash函数），算法参见：

[http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo\_hash\_function](https://links.jianshu.com/go?to=http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function" \t "_blank)

### func New32

// New32 returns a new 32-bit FNV-1 hash.Hash.  
// Its Sum method will lay the value out in big-endian byte order.  
func New32() hash.Hash32 {  
 var s sum32 = *offset32* return &s  
}

返回一个新的32位FNV-1的hash.Hash32接口

### func New32a

// New32a returns a new 32-bit FNV-1a hash.Hash.  
// Its Sum method will lay the value out in big-endian byte order.  
func New32a() hash.Hash32 {  
 var s sum32a = *offset32* return &s  
}

返回一个新的32位FNV-1a的hash.Hash32接口

### func New64

// New64 returns a new 64-bit FNV-1 hash.Hash.  
// Its Sum method will lay the value out in big-endian byte order.  
func New64() hash.Hash64 {  
 var s sum64 = *offset64* return &s  
}

返回一个新的64位FNV-1的hash.Hash64接口

### func New64a

// New64a returns a new 64-bit FNV-1a hash.Hash.  
// Its Sum method will lay the value out in big-endian byte order.  
func New64a() hash.Hash64 {  
 var s sum64a = *offset64* return &s  
}

返回一个新的64位FNV-1a的hash.Hash64接口