应用场景：

hbase使用布隆过滤器来查找不存在的的行，以减少磁盘io

垃圾邮件地址过滤。

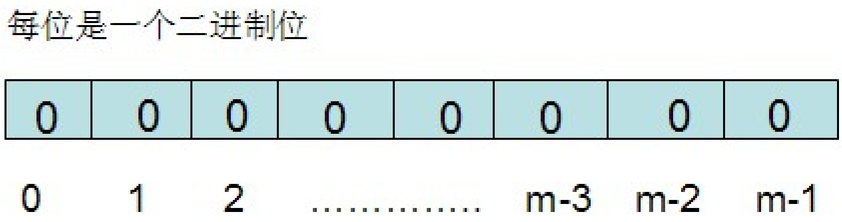
浏览器钓鱼网站警告的实现。

解决缓存击穿

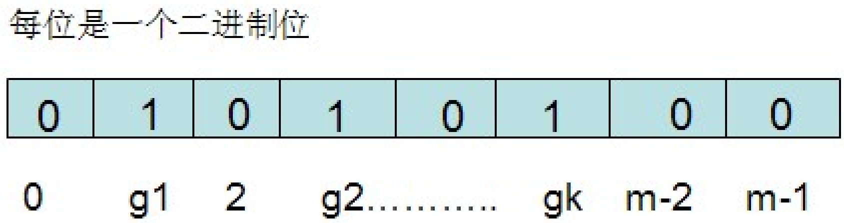
爬虫url地址去重。

实现原理：

布隆过滤器需要的是一个位数组(和位图类似)和K个映射函数(和Hash表类似)，在初始状态时，对于长度为m的位数组array，它的所有位被置0。



对于有n个元素的集合S={S1,S2...Sn},通过k个映射函数{f1,f2,......fk}，将集合S中的每个元素Sj(1<=j<=n)映射为K个值{g1,g2...gk}，然后再将位数组array中相对应的array[g1],array[g2]......array[gk]置为1：



如果要查找某个元素item是否在S中，则通过映射函数{f1,f2,...fk}得到k个值{g1,g2...gk}，然后再判断array[g1],array[g2]...array[gk]是否都为1，若全为1，则item在S中，否则item不在S中。这个就是布隆过滤器的实现原理。

使用Google的guava可以轻松实现布隆过滤器。

假设 Hash 函数以等概率条件选择并设置 Bit Array 中的某一位，M 是该位数组的大小，K是 Hash 函数的个数，错误率是P:

确定过滤器大小(具体见维基百科)：

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

确定哈希函数个数：

$$k = \frac{m}{n} \ln 2.$$

设计hash函数：MurmurHash 算法，该算法同时可以用作一致性hash算法的实现。

缺点：

布隆过滤器用于查出某一元素在某一集合中是否存在，如果布隆过滤器判断不存在，那肯定是不存在，但如果判断存在却不一定存在，存在误判率，即mightcontain

数组长度和hash函数的个数是根据容量和误判率来确定的，误判率默认0.03
误判率不能降低为0 ，越低对存储空间要求越高。

应用场景一：url地址去重

URL的去重方法有很多种，从次到优依次可以分为以下5种：
1、将URL保存到数据库进行去重（假设单个URL的平均长度是100 byte）。
2、将URL放到HashSet中去重（一亿条占用10G内存）。
3、将URL经过MD5之后保存到HashSet（MD5的结果是128 bit也就是16 byte的长度，一亿条占用1.6G的内存，Scrapy采用的就是类似方法）。
4、使用Bitmap或Bloomfilter方法去重（URL经过hash后映射到bit的每一个位上，一亿URL占用约12M，问题是存在冲突）。

```go
package standard

import (
   "crypto/md5"
   "crypto/sha1"
   "fmt"
   "github.com/spaolacci/murmur3"
   "github.com/zhenjl/cityhash"
   "hash"
   "hash/crc64"
   "hash/fnv"
   "io/ioutil"
   "net/url"
   "os"
   "regexp"

   "testing"
)

func TestNew(t *testing.T) {

   var n uint  = 100000
   h := []hash.Hash{fnv.New64(), crc64.New(crc64.MakeTable(crc64.ECMA)),
murmur3.New64(), cityhash.New64(), md5.New(), sha1.New()}
   l := []string{"fnv.New64()", "crc64.New()", "murmur3.New64()",
"cityhash.New64()", "md5.New()", "sha1.New()"}


     for j := range h {
       fmt.Printf("\n\nTesting %d with size %s\n", n, l[j])
       bf := New(n)
       bf.SetHasher(h[j])
       //bf.PrintStats()
```

```go
    }

}


//使用布隆过滤器对URL去重
func TestStandardBloom_Add(t *testing.T) {
  var n uint  = 100000
  bf := New(n)
  urlsList := getUrl("url.txt")

  fmt.Println(len(urlsList))

  for _, item := range urlsList {
    if !bf.Check([]byte(item)) {
      bf.Add([]byte(item))
    }

  }
  fmt.Println(bf.Count())
}


// 获取url
func getUrl(path string) (urlsList []string) {
  file, err := os.Open(path)
  defer file.Close()
  if err != nil {
    return nil
  }
  data, err := ioutil.ReadAll(file)
  if err != nil {
    return nil
```

```
    }
    sreg := regexp.MustCompile(`(https.*)\n?`)
    sall := sreg.FindAllSubmatch(data, -1)
    for _, sitem := range sall {
        fmt.Println(string(sitem[0]))
        surl, err := url.Parse(string(sitem[0]))
        if err != nil {
            panic(err)
        }
        urlsList = append(urlsList, surl.String())
    }

    return urlsList
}
```

测试输出结果：

```
Tests passed: 1 of 1 test – 1 ms
<4 go setup calls>
https://blog.csdn.net/tanyun_sunshine/article/details/80087718

https://blog.csdn.net/tanyun_sunshine/article/details/80087718

https://www.baidu.comhttps://bbs.csdn.net/topics/380233284
3
2
```

布隆过滤器的go语言实现：

接口定义：

```
package bloom

import (
```

```go
	"hash"
	"math"
)

type Bloom interface {
	Add(key []byte) Bloom
	Check(key []byte) bool
	Count() uint
	PrintStats()
	SetHasher(hash.Hash)
	Reset()
	FillRatio() float64
	EstimatedFillRatio() float64
	SetErrorProbability(e float64)
}

//返回hash函数的个数
func K(e float64) uint {
	return uint(math.Ceil(math.Log2(1 / e)))
}
//返回位数组的容量
func M(n uint, p, e float64) uint {
	// m =~ n / ((log(p)*log(1-p))/abs(log e))
	return uint(math.Ceil(float64(n) / ((math.Log(p) * math.Log(1-p)) /
math.Abs(math.Log(e)))))
}

func S(m, k uint) uint {
	return uint(math.Ceil(float64(m) / float64(k)))
}
```

具体实现：

```go
package standard

import (
  "encoding/binary"
  "fmt"
  "github.com/willf/bitset"
  "goland/bloom"
  "hash"
  "hash/fnv"
  "math"
)

type StandardBloom struct {

  //hash函数
    h hash.Hash


  //位数组的大小，即容量
    // m =~ n / ((log(p)*log(1-p))/abs(log e)) （见维基百科）
    m uint


  //hash函数的个数
    // k = log2(1/e)
  // Given that our e is defaulted to 0.001, therefore k ~= 10, which means we
need 10 hash values
  k uint

  // s is the size of the partition, or slice.
  // s = m / k
```

```
  s uint

  // p is the fill ratio of the filter partitions. It's mainly used to calculate m at the start.
  // p is not checked when new items are added. So if the fill ratio goes above p, the likelihood
  // of false positives (error rate) will increase.
  //
  // By default we use the fill ratio of p = 0.5
  //填充率，即位数组中设为1的占比
    p float64

  //错误率，
    // By default we use the error rate of e = 0.1% = 0.001. In some papers this is P (uppercase P)
  e float64

  // n is the number of elements the filter is predicted to hold while maintaining the error rate
  // or filter size (m). n is user supplied. But, in case you are interested, the formula is
  // n =~ m * ( (log(p) * log(1-p)) / abs(log e) )
  n uint

  // b is the set of bit array holding the bloom filters. There will be k b's.
  b *bitset.BitSet

  // c is the number of items we have added to the filter
  c uint

  // bs holds the list of bits to be set/check based on the hash values
  //添加一个元素时，首先计算该元素对应每个hash函数的hash值，
    bs []uint
```

```go
}

//构造函数
func New(n uint) bloom.Bloom {
  var (
    p float64 = 0.5
    e float64 = 0.001
    k uint    = bloom.K(e)
    m uint    = bloom.M(n, p, e)
  )

  return &StandardBloom{
    h:  fnv.New64(),
    n:  n,
    p:  p,
    e:  e,
    k:  k,
    m:  m,
    b:  bitset.New(m),
    bs: make([]uint, k),
  }
}


//设置hash函数
func (this *StandardBloom) SetHasher(h hash.Hash) {
  this.h = h
}

func (this *StandardBloom) Reset() {
  this.k = bloom.K(this.e)
  this.m = bloom.M(this.n, this.p, this.e)
  this.b = bitset.New(this.m)
```

```go
	this.bs = make([]uint, this.k)

	if this.h == nil {
		this.h = fnv.New64()
	} else {
		this.h.Reset()
	}
}

//设置错误率
func (this *StandardBloom) SetErrorProbability(e float64) {
	this.e = e
}

//计算填充率，
func (this *StandardBloom) FillRatio() float64 {
	return float64(this.b.Count()) / float64(this.m)
}

//求取k位的位置索引之后，分别置1
func (this *StandardBloom) Add(item []byte) bloom.Bloom {
	this.bits(item)
	for _, v := range this.bs[:this.k] {
		this.b.Set(v)
	}
	this.c++
	return this
}

func (this *StandardBloom) Count() uint {
	return this.c
}
```

```go
func (this *StandardBloom) Check(item []byte) bool {
  this.bits(item)
  for _, v := range this.bs[:this.k] {
    if !this.b.Test(v) {
      return false
    }
  }

  return true
}

//预估填充率
func (this *StandardBloom) EstimatedFillRatio() float64 {
  return 1 - math.Exp((-float64(this.c)*float64(this.k))/float64(this.m))
}

func (this *StandardBloom) PrintStats() {
  fmt.Printf("m = %d, n = %d, k = %d, s = %d, p = %f, e = %f\n", this.m,
this.n, this.k, this.s, this.p, this.e)
  fmt.Println("Total items:", this.c)
  c := this.b.Count()
  fmt.Printf("Total bits set: %d (%.1f%%)\n", c, float32(c)/float32(this.m)*100)
}

//将item放置到bloom中时，位数组需置1的位置索引， k个。
func (this *StandardBloom) bits(item []byte) {
  this.h.Reset()
  this.h.Write(item)
  s := this.h.Sum(nil)
  a := binary.BigEndian.Uint32(s[4:8])
  b := binary.BigEndian.Uint32(s[0:4])
```

```
    // Reference: Less Hashing, Same Performance: Building a Better Bloom
Filter
    // URL: http://www.eecs.harvard.edu/~kirsch/pubs/bbbf/rsa.pdf
    for i, _ := range this.bs[:this.k] {
        this.bs[i] = (uint(a) + uint(b)*uint(i)) % this.m
    }
}
```