H2

map的结构体定义

```
type Map struct {
  //此锁是为了保护Map的dirty数据
  mu Mutex
  //用来存读的数据，只对类型，不会造成读写冲突
  read atomic.Value

  dirty map[interface{}] *entry
  //当读数据时候，该字段不在read中，尝试从dirty中读取，不管是否在dirty中读取到数据，
missed+1
  //当累计到len(dirty) 时，会将dirty拷贝到read，并将dirty清空，以此提升读性能
  missed int
}
```

在sync.Map中用到了冗余数据结构read、dirty。其中read的类型是atomic.Value，它会通过atomic.Value的Load方法将其断言为readOnly对象，因此read的实际类型为readOnly

```
read, _ := m.read.Load().(readOnly)
```

readOnly的数据结构如下

```
type readOnly struct {
  m map[interface{}]*entry
  //当sync.Map.dirty中包含了某些不存在的key时，amend的值为true
  amended bool

}
```

- amend属性的作用是志明dirty中是否有readOnly.m中未包含的数据，因此当对sync.Map的读操作在read中找不到数据时候，将进一步在dirty中查找
- readOnly.m和Map.dirty中map存储的值类型是*entry,它包含一个指针p，指向存储的value值。

```
type entry struct {
  p unsafe.Pointer
}
```

entry.p的值有三种类型

- nil:entry已经被删除，且m.dirty为nil
- expunged:entry被删除，m.dirty不为nil，但entry不存在m.dirty中
- 其他：entry有效，记录在m.read中，若dirty不为空，也会记录在dirty中

虽然read和dirty存在冗余数据，但是这些数据entry是通过指针指向的，因此，尽管map的value可能会很大，但是空间存储还是足够。

H2 方法

sync.Map有四个方法实现，分别是Load、Store、Delete和Range

Load

加载方法，通过提供的key，查找对应的值value

H3

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool) {
  //从m的read中通过Load方法得到readOnly
  read, _ := m.read.Load().(readOnly)
  //从read的m中查找key
  e, ok := read.m[key]
  //如果在read中没有找到，且表明有新数据在dirty中
  //那么在dirty中查找
  if !ok &&read.amended {
    m.mu.Lock()
    //双重检查避免在本次加锁的时候，有其他的goroutine正好将map中的dirty数据复制到了read中
    //Map.read的并发安全性保障就在他的修改是通过原子操作的，因此需要再一次的load
    read, _ = m.read.Load().(readOnly)
    e, ok := read.m[k]
    if !ok && read.amended {
      e, ok := m.dirty[key]
      //不管是否从dirty中找到数据，都要讲missed加一
      m.missLocked()
    }
    m.mu.Unlock()
  }

  if !ok {
    return nil, false
  }
  //通过map的load方法，将entry.p加载为对应的指针，再返回指针指向的值
  return e.load()
}
```

Map的missLocked函数是保证sync.Map性能的重要函数，他的目的是将存在有所的dirty中的数据，转移导只读现成安全的read中去

```
func (m *Map) missLocked() {
  m.misses++
  if m.misses < len(m.dirty) {
    return
  }
  //将dirty复制到read中
  m.read.Store(readOnly{m: m.dirty})
  m.dirty = nil //将dirty清空
  m.misses = 0  //计数清零
}
```

Store

该方法用于更新或者新增键值对key-value

H3

```
func (m *Map) Store(key,value interface{}) {
  //如果read中存在该兼职，且兼职美哦与被删除，则尝试直接存储
  read, _ := m.read.Load().(readOnly)
  if e, ok := read.m[key]; ok && e.tryStore(&value) {
    return
  }
  //如果不满足上述条件
  m.mu.Lock()
  read, _ = m.read.Load().(readOnly)
  if e, ok := read.m[key]; ok {
    //判断entry是否被标记删除
    if e.unexpungeLocked() {
      //如果entry被标记删除，则将entry加入导dirty中
      m.dirty[key] = e
    }
    //更新entry指向的value地址
    e.storeLocked(&value)
  }else if e,ok := m.dirty[key];ok { //dirty中有该键值，则更新
    e.storeLocked(&value)
  }else{ //dirty和read中均无该键值
    if !read.amended {
      m.dirtyLocked() //从m.read中复制未删除的数据导dirty中
      m.read.Store(readOnly{m:read.m, amended:true})
    }
    //将entry添加到dirty中
    m.dirty[key] = newEntry(value)
  }
}
```

Store每次的操作都是从read开始的，先不满足条件时，才加锁操作dirty。但是由于存在从read中复制数据的情况，当m.read中数据量很大的情况，会对性能造成影响。

Delete

删除某个键值

H3

```
func (m *Map) Delete(key interface{}) {
  read,_ := m.read.Load().(readOnly)
  e, ok := read.m[key]
  if !ok && read.amended {
    m.mu.Lock()
    read, _ = m.read.Load().(readOnly)
    e, ok := read.m[key]
    if !ok && read.amended {
      delete(m.dirty,key)
    }
    m.mu.Unlock()

  }
  if ok {
    e.delete()
  }

}

func (e *entry) delete()(hadValue bool) {
  for {
    p := atomic.LoadPointer(&e.p)
    //如果p指针为空，或者被标记清除
    if p == nil || p ==  expunged {
      return false
    }

    //通过原子操作将e.p标记为nil
    if atomic.CompareAndSwapPointer(&e.p, p, nil{
      return true
    }
  }

}
```

Delete中的逻辑和Store逻辑类似，都是从read开始，如果这个key不在read中，且dirty中有新的数据，则加锁从dirty中删除，并且需要做双重检查。

Range

想要遍历sync.Map，不能通过for range的形式，因此他本身提供了Range方法，通过回调方式遍历

H3

```
func (m *Map) Range(f func(key, value interface{}) bool) {
  read, _ := m.read.Load().(readOnly)
  if read.amended {
    m.mu.Lock()
    read, _ = m.read.Load().(readOnly)
    if read.amended  {
      //将dirty复制到read中
      read = readOnly{m:m.dirty}
      m.read.Store(read)
      m.dirty = nil
      m.misses = 0
    }
    m.mu.Unlock()
  }

  //遍历整理过后的read
  for k,e := range read.m {
    v, ok := e.load()
    if !ok {
      continue
    }
    if !f(k,v) {
      break
    }
  }
}
```

sync.Map总结

- 空间换时间：通过两个冗余的数据结构（read、dirty），减少锁对性能的影响
- 读操作使用read，避免读写冲突
- 动态调整，通过misses值，避免dirty数据过多
- 双检查机制：避免在非原子操作是产生错误
- 延迟删除机制：删除一个键值只是先打标记，只有等提升dirty（复制到read中，并清空自身）时才清理删除的数据。
- 优先从read中读、改和删除，因为对read操作不用加锁，大大提升性能。

H3

使用示例

H3

```
func main() {
  var sm sync.Map
  sm.Store(1,"a")
  sm.Store(2,"b")
  sm.Store("c",3)
  if v, ok := sm.Load("c"); ok {
    fmt.Println(v)
  }

  //删除
  sm.Delete(1)
  //遍历
  //参数func中的参数是遍历获得的key和value，返回一个bool值
  sm.Range(func(key,value interface{}) bool {
    fmt.Println(key, value)
    return true
  })
}
```