深入理解并发安全的 sync.Map

1、 Map 和 sync.Map 的比较

Golang 中内置 map 关键字，但是是非线程安全的，故加入 sync.Map，提供并发安全的 map。

普通map的并发问题

map的并发读写代码

```
func main() {
  m := make(map[int]int)

  go func() {
    for {
      _ = m[1] // 读
    }
  }()

  go func() {
    for {
      m[2] = 2 // 写
    }
  }()

  select {} // 维持主goroutine
}
```

以上是一段并发读写map的代码，其中一个goroutine一直读，另外一个goroutine一直写。即使读写的map键不相同，且不存在"扩容"等操作，代码还是会报错。

```
fatal error: concurrent map read and map write
```

锁+map

那普通map有没有能力实现并发安全呢？答案是肯定的。通过给map额外绑定一个锁（sync.Mutex或sync.RWMutex），封装成一个新的struct，实现并发安全。

定义带有锁的对象M

```
type M struct {
  sync.RWMutex
  Map map[int]int
}
```

执行并发读写

```
func main() {
  m := M{Map: make(map[int]int)}

  go func() {
    for {
      m.RLock()
      v := m.Map[2] // 读
      fmt.Println(v)
      m.RUnlock()
    }
  }()

  go func() {
    for i := 1; i > 0; i++ {
      m.Lock()
      m.Map[2] = i // 写
      m.Unlock()
    }
  }()

  select {}
}
```

在读goroutine读数据时，通过读锁锁定，在写goroutine写数据时，写锁锁定，程序就能并发安全的运行，运行结果示意如下。

```
...
1123
1124
1125
```

但是当 map 的数据量非常大的时候，会引发大量 goroutine 争夺同一把锁，这种现象将直接导致性能的急剧下降。有点类似于 java 中的 hashMap 和 concurrentHashMap。

sync.Map的源码结构（基于1.14.1）

```
type Map struct {
  // 此锁是为了保护Map中的dirty数据
  mu Mutex
  // 用来存读的数据，只读类型，不会造成读写冲突
  read atomic.Value // readOnly
  // dirty包含最新的写入数据（entry），在写的同时，将read中未删除的数据拷贝到dirty中
  // 因为是go中内置的普通map类型，且涉及写操作，所以需要通过mu加锁
  dirty map[interface{}]*entry
  // 当读数据时,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，misses+1
  // 当累计到len（dirty）时，会将dirty拷贝到read中，并将dirty清空，以此提升读性能。
  misses int
}
```

在sync.Map中用到了两个冗余数据结构read、dirty。其中read的类型为atomic.Value，它会通过atomic.Value的Load方法将其断言为readOnly对象。

```
read, _ := m.read.Load().(readOnly) // m为sync.Map
```

因此，read的真实类型即是readOnly，其数据类型如下。

```
type readOnly struct {
  // read 中的go内置map类型，但是它不需要锁。
  m       map[interface{}]*entry
  // 当sync.Map.diry中的包含了某些不在m中的key时，amended的值为true.
  amended bool
}
```

（左右滑动查看完整代码图片）

amended属性的作用是指明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，将进一步到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，指向用户存储的value值。

```
type entry struct {
  p unsafe.Pointer // *interface{}
}
```

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的。

以上是sync.Map的数据结构，下面着重看看它的四个方法实现：Load、Store、Delete和Range。

2、load

Load
加载方法，通过提供的键key，查找对应的值value。

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool) {
  // 首先从m.read中通过Load方法得到readOnly
  read, _ := m.read.Load().(readOnly)
  // 从read中的map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diry中（read.amended为true）
  // 那么，就需要在dirty中查找，这时需要加锁。
  if !ok && read.amended {
    m.mu.Lock()
    // 双重检查:避免在本次加锁的时候，有其他goroutine正好将Map中的dirty数据复制到了read中。
    // 能发生上述可能的原因是以下两行代码语句，并不是原子操作。
    // if !ok && read.amended {
    //   m.mu.Lock()
    // }
    // 而Map.read其并发安全性的保障就在于它的修改是通过原子操作进行的。
    // 因此需要再检查一次read.
    read, _ = m.read.Load().(readOnly)
    e, ok = read.m[key]
    // 如果m.read中key还是不存在，且dirty中有新数据，则检查dirty中的数据。
    if !ok && read.amended {
      e, ok = m.dirty[key]
      // 不管是否从dirty中得到了数据，都会将misses的计数+1
      m.missLocked()
    }
    m.mu.Unlock()
  }
  if !ok {
    return nil, false
  }

  // 通过Map的load方法，将entry.p加载为对应指针，再返回指针指向的值
  return e.load()
}
```

对 dirty 加锁，对 read 不加锁，减少锁的争用。原理就是只将数据锁一段时间，也就是只锁 dirty，当 dirty 满了，那么就拷贝 read，read 是不加锁的。而满没满是看 misses 值是否小于 les(m.dirty)。而且因为不管有没有从 dirty 中拿到数据，misses 都会自加 1，所以保证了数据不会长时间加锁。对数据加锁的方式就是，锁掉读数据的代码，这样就不会

存在多个 goroutine 同时读数据了。而且会对 read，进行两次读，防止在对读 dirty 数据的代码加锁的时候，dirty 中的数据已经拷贝到 read 了。

Map.missLocked函数是保证sync.Map性能的重要函数，它的目的是将存在有锁的dirty中的数据，转移到只读线程安全的read中去。

```
func (m *Map) missLocked() {
  m.misses++ // 计数+1
  if m.misses < len(m.dirty) {
    return
  }
  m.read.Store(readOnly{m: m.dirty}) // 将dirty数据复制到read中去
  m.dirty = nil // dirty清空
  m.misses = 0 // misses重置为0
}
```

3、Store

Store
该方法更新或新增键值对key-value。

```
func (m *Map) Store(key, value interface{}) {
  // 如果m.read中存在该键，且该键没有被标记删除（expunged）
  // 则尝试直接存储（见entry的tryStore方法）
  // 注意： 如果m.dirty中也有该键（key对应的entry），由于都是通过指针指向，所有m.dirty中也会保持最新entry值。
  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，即m.read不存在或者已经被标记删除
  m.mu.Lock()
  read, _ = m.read.Load().(readOnly)
  if e, ok := read.m[key]; ok { // 如果read中有该键
    if e.unexpungeLocked() { // 判断entry是否被标记删除
      // 如果entry被标记删除，则将entry添加进m.dirty中
      m.dirty[key] = e
    }
    // 更新entry指向value地址
    e.storeLocked(&value)
  } else if e, ok := m.dirty[key]; ok { //dirty中有该键：更新
    e.storeLocked(&value)
  } else { // dirty和read中均无该键：新增
    if !read.amended { // 表明dirty中没有新数据，在dirty中增加第一个新键
      m.dirtyLocked() // 从m.read中复制未删除的数据到dirty中
      m.read.Store(readOnly{m: read.m, amended: true})
    }
    m.dirty[key] = newEntry(value) // 将entry增加到dirty中
  }
  m.mu.Unlock()
}
```

（左右滑动查看完整代码图片）

Store的每次操作都是先从read开始，当不满足条件时，才加锁操作dirty。但是由于存在从read中复制数据的情况（例如dirty刚复制完数据给m.read，又来了一个新键），当m.read中数据量很大时，可能对性能造成影响。

Read 中是否有 key，

有且未标记删除，直接修改指针的 val 值。

有但标记删除，插入 dirty，修改 read 的 key 的指针指向即可。

Read 中没有 key

dirty 中有该键，更新 val

dirty 中也没有，将 read 中赋值未删除的数据到 dirty（保持 dirty 中数据是最新的），然后再在 dirty 中用 newEntry 新增 entry。

Delete

删除某键值。

```
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()
  }
}

// 如果read中有该键，则从read中删除，其删除方式是通过原子操作
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
    }
  }
}
```

（左右滑动查看完整代码图片）

Delete中的逻辑和Store逻辑相似，都是从read开始，如果这个key（也即是entry）不在read中，且dirty中有新数据，则加锁从dirty中删除。注意，和Load与Store方法一样，也是需要双检查。

4、Range

Range

想要遍历sync.Map，不能通过for range的形式，因此，它自身提供了Range方法，通过回调的方式遍历。

```go
func (m *Map) Range(f func(key, value interface{}) bool) {
  read, _ := m.read.Load().(readOnly)
  // 判断dirty中是否有新的数据
  if read.amended {
    m.mu.Lock()
    // 双检查
    read, _ = m.read.Load().(readOnly)
    if read.amended {
      // 将dirty中的数据复制到read中
      read = readOnly{m: m.dirty}
      m.read.Store(read)
      m.dirty = nil
      m.misses = 0
    }
    m.mu.Unlock()
  }

  // 遍历已经整合过dirty的read
  for k, e := range read.m {
    v, ok := e.load()
    if !ok {
      continue
    }
    if !f(k, v) {
      break
    }
  }
}
```

5、Sync.Map 的使用例子

Range

想要遍历sync.Map，不能通过for range的形式，因此，它自身提供了Range方法，通过回调的方式遍历。

```
func (m *Map) Range(f func(key, value interface{}) bool) {
  read, _ := m.read.Load().(readOnly)
  // 判断dirty中是否有新的数据
  if read.amended {
    m.mu.Lock()
    // 双检查
    read, _ = m.read.Load().(readOnly)
    if read.amended {
      // 将dirty中的数据复制到read中
      read = readOnly{m: m.dirty}
      m.read.Store(read)
      m.dirty = nil
      m.misses = 0
    }
    m.mu.Unlock()
  }

  // 遍历已经整合过dirty的read
  for k, e := range read.m {
    v, ok := e.load()
    if !ok {
      continue
    }
    if !f(k, v) {
      break
    }
  }
}
```

6、适用场景

Sync.Map 并不是为了代替锁+map 的组合。

两种情况应该选择 sync.Map

1、key 值一次写入，多次读取，因为总是会 double check read

2、多个 goroutine 的读取、写入和覆盖在不相交的 key 集。

原子操作和互斥锁的区别

原子操作即是进行过程中不能被终端的操作，针对某个值的原子操作在被进行的过程中，cpu 绝不会再去进行其他的针对该值的操作。为了实现这样的严谨行，原子操作仅会由一个独立的 cpu 指令代表和完成。原子操作是无锁的，常常通过 cpu 指令直接实现，事实上，其他同步及时的实现常常依赖与原子操作。

1、Go 对原子操作的支持：

Go 语言的 sync/atomic 包提供了对原子操作的支持，用于同步访问整数和指针。

五种：增减、比较并交换、载入、存储、交换。

原子操作支持的类型包括 int32，int64、uint32、uint64、uintptr、unsafe.Pointer

下面演示两种原子操作的实例：

AddInt32

下面的示例演示如何使用 `AddInt32` 函数对int32值执行添加原子操作。在这个例子中， `main goroutine` 创建了1000个的并发 `goroutine` 。每个新创建的 `goroutine` 将整数n加1。

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

func main() {
    var n int32
      var wg sync.WaitGroup
      for i := 0; i < 1000; i++ {
            wg.Add(1)
            go func() {
                  atomic.AddInt32(&n, 1)
                  wg.Done()
            }()
      }
      wg.Wait()

    fmt.Println(atomic.LoadInt32(&n)) // output:1000
}
```

上面的例子里你们可以自己试验一下，如果我们不使用 `atomic.AddInt32(&n, 1)` 而是简单的对变量 `n` 进行自增的话得到结果并不是我们预期的1000，这就是我们在文章《Go并发编程里的数据竞争以及解决之道》里提到过的数据竞争问题，原子操作可确保这些 `goroutine` 之间不存在数据竞争。

CompareAndSwarp

原子操作中的**比较并交换**简称 CAS （Compare And Swap），在 sync/atomic 包中，这类原子操作由名称以 CompareAndSwap 为前缀的若干个函数提供

```
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func CompareAndSwapPointer(addr *unsafe.Pointer,old, new unsafe.Pointer) (swapped bool)
......
```

调用函数后， CompareAndSwap 函数会先判断参数 addr 指向的操作值与参数 old 的值是否相等，仅当此判断得到的结果是 true 之后，才会用参数 new 代表的新值替换掉原先的旧值，否则操作就会被忽略。

我们使用的 mutex 互斥锁类似悲观锁，总是假设会有并发的操作要修改被操作的值，所以使用锁将相关操作放入临界区中加以保护。而使用 CAS 操作的做法趋于乐观锁，总是假设被操作值未曾被改变（即与旧值相等），并一旦确认这个假设的真实性就立即进行值替换。在被操作值被频繁变更的情况下， CAS 操作并不那么容易成功所以需要不断进行尝试，直到成功为止。

```
package main

import (
    "fmt"
    "sync/atomic"
)

var value int32 = 1

func main()  {
    fmt.Println("======old value=======")
    fmt.Println(value)
    fmt.Println("======New value=======")
    fmt.Println(value)

}

//不断地尝试原子地更新value的值,直到操作成功为止
func addValue(delta int32){
    for {
        v := value
        if atomic.CompareAndSwapInt32(&value, v, (v + delta)){
            break
        }
    }
}
```

上面的比较并交换案例中 v:= value 为变量 v 赋值，但要注意，在进行读取 value 的操作的过程中，其他对此值的读写操作是可以被同时进行的，那么这个读操作很可能会读取到一个只被修改了一半的数据。所以 我们要使用 sync/atomic 代码包中为我们提供的以 Load 为前缀的函数，来避免这样的糟糕事情发生。

竞争条件是由于异步的访问共享资源，并试图同时读写该资源而导致的，使用互斥锁和通道的思路都是在线程获得到访问权后阻塞其他线程对共享内存的访问，而使用原子操作解决数据竞争问题则是利用了其不可被打断的特性。

2、原子操作和互斥锁区别：

首先二者均不可被其他线程打断，

Atomic 操作的优势是更轻量，比如 CAS 可以在不行成临界区和创建互斥量的情况下完成并发安全的值替换操作。这可以大大的减少同步对程序性能的损耗。

当然也有劣势，趋于乐观，总是假设被操作值未被改变，并一段确认这个假设的真实性就立即进行值替换，那么在被操作值被频繁变更的情况下，CAS 操作并不容易成功。相反，互斥锁的做法趋于悲观，总假设会有并发的操作要修改被操作的值，并用锁

将相关操作放入临界区进行保护。

总结：

1、互斥锁是一种数据结构，用来让一个线程执行程序的关键不分，完成互斥的多个操作。

2、原子操作是针对某个值的单个互斥操作。

3、互斥锁可以理解为悲观锁，共享资源每次只给一个线程使用，其他线程阻塞，用完后再把资源转让给其他线程。

Go 并发变成里的数据竞争以及解决之道

Go 语言以容易进行并发变成而闻名。

1、数据竞争

要解释什么是数据竞争我们先来看一段程序：

```
package main

import "fmt"

func main() {
    fmt.Println(getNumber())
}

func getNumber() int {
    var i int
    go func() {
        i = 5
    }()

    return i
}
```

这段代码有两个携程：

1、主携程：返回 i

2、自定义携程：设置 i

那么可能返回的 i 值是 0 或 5，那么这就产生了数据竞争。

2、解决数据竞争

1、使用 WaitGroup

解决数据竞争的最直接方法是（如果需求允许的情况下）阻止读取访问，直到写入操作完成：

```
func getNumber() int {
    var i int
    // 初始化一个WaitGroup
    var wg sync.WaitGroup
    // Add(1) 通知程序有一个需要等待完成的任务
    wg.Add(1)
    go func() {
        i = 5
        // 调用wg.Done 表示正在等待的程序已经执行完成了
        wg.Done()
    }()
    // wg.Wait会阻塞当前程序直到等待的程序都执行完成为止
    wg.Wait()
    return i
}
```

2、使用通道阻塞

通道中如果没有值，但如果你要从通道里取值，那么会阻塞程序，直到有值取出，我们可以在我们期待的优先执行的 goroutine 中给通道存值，后运行的 goroutine 中取值来实现避免数据竞争。

使用通道阻塞

这个方法原则上与上一种方法类似，只是我们使用了通道而不是 `WaitGroup` ：

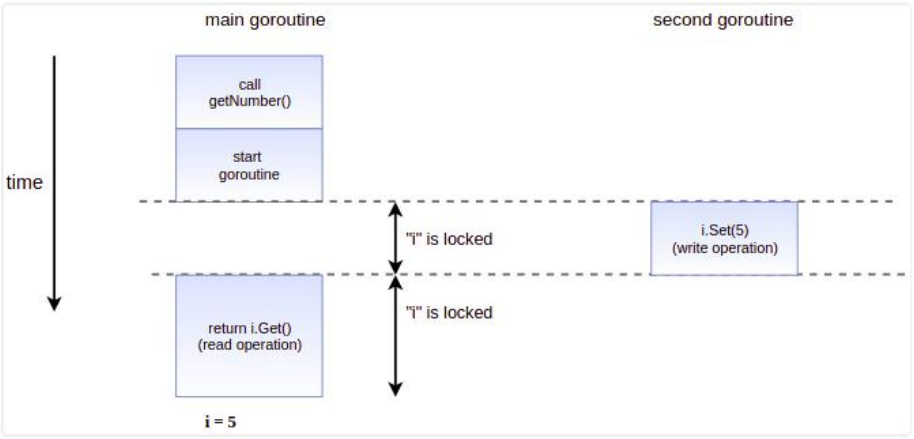
```
func getNumber() int {
    var i int
  // 创建一个通道，在等待的任务完成时会向通道发送一个空结构体
    done := make(chan struct{})
    go func() {
        i = 5
        // 执行完成后向通道发送一个空结构体
        done <- struct{}{}
    }()
  // 从通道接收值将会阻塞程序，直到有值发送给done通道为止
    <-done
    return i
}
```

3、使用 Mutex

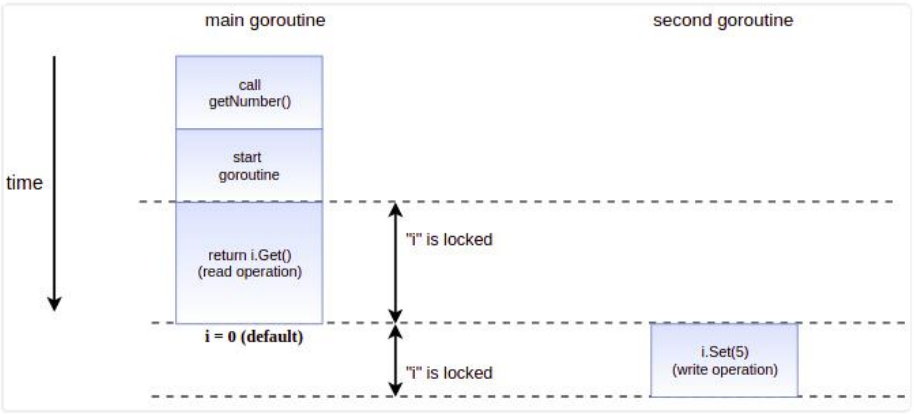
上诉两种方法，保证的是先写入后读取，但假如不需要保证顺序，只需要保证二者是互斥的，不能同时发生即可，那么应该考虑使用 Mutex 互斥锁。

```go
// 首先，创建一个结构体包含我们想用互斥锁保护的值和一个mutex实例
type SafeNumber struct {
    val int
    m   sync.Mutex
}

func (i *SafeNumber) Get() int {、
    i.m.Lock()
    defer i.m.Unlock()
    return i.val
}

func (i *SafeNumber) Set(val int) {
    i.m.Lock()
    defer i.m.Unlock()
    i.val = val
}

func getNumber() int {
    // 创建一个sageNumber实例
    i := &SafeNumber{}
  // 使用Set和Get代替常规赋值和读取操作。
  // 我们现在可以确保只有在写入完成时才能读取，反之亦然
    go func() {
        i.Set(5)
    }()
    return i.Get()
}
```

下面两个图片对应于程序先获取到写锁和先获取到读锁两种可能的情况下程序的执行流程：



先获取到写锁时程序的执行流程



4、mutex Vs channel

大多数新手都视图用通道解决所有并发问题，这是 go 语言的一个很酷的特性，这是不对的，二者都各有好处。

通常，当 goroutine 需要相互通信时使用通道，当确保只有一个 goroutin 能访问代码的关键部分时使用互斥锁。比如这个问题，互斥锁更为适合，你在通道里面传入的空结构体并没有任何通信的作用。

GO 标准库 net/rpc

1、基于 HTTP RPC 同步调用：

服务端：

```go
package main

import (
  "errors"
  "log"
  "net"
  "net/http"
  "net/rpc"
)

type Args struct {
  A, B int
}

type Quotient struct {
  Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
  *reply = args.A * args.B
  return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
  if args.B == 0 {
    return errors.New("divide by 0")
  }

  quo.Quo = args.A / args.B
  quo.Rem = args.A % args.B
  return nil
}

func main() {
  arith := new(Arith)
  rpc.Register(arith)
  rpc.HandleHTTP()
  if err := http.ListenAndServe(":1234", nil); err != nil {
    log.Fatal("serve error:", err)
  }
}
```

客户端：

```go
package main

import (
  "fmt"
  "log"
  "net/rpc"
)

type Args struct {
  A, B int
}

type Quotient struct {
  Quo, Rem int
}

func main() {
  client, err := rpc.DialHTTP("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)

  args = &Args{15, 6}
  var quo Quotient
  err = client.Call("Arith.Divide", args, &quo)
  if err != nil {
    log.Fatal("Divide error:", err)
  }
  fmt.Printf("Divide: %d/%d=%d...%d\n", args.A, args.B, quo.Quo, quo.Rem)
}
```

2、基于 HTTP RPC 异步调用：

客户端：

```go
func main() {
  client, err := rpc.DialHTTP("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args1 := &Args{7, 8}
  var reply int
  multiplyReply := client.Go("Arith.Multiply", args1, &reply, nil)

  args2 := &Args{15, 6}
  var quo Quotient
  divideReply := client.Go("Arith.Divide", args2, &quo, nil)

  ticker := time.NewTicker(time.Millisecond)
  defer ticker.Stop()

  var multiplyReplied, divideReplied bool
  for !multiplyReplied || !divideReplied {
    select {
    case replyCall := <-multiplyReply.Done:
      if err := replyCall.Error; err != nil {
        fmt.Println("Multiply error:", err)
      } else {
        fmt.Printf("Multiply: %d*%d=%d\n", args1.A, args1.B, reply)
      }
      multiplyReplied = true
    case replyCall := <-divideReply.Done:
      if err := replyCall.Error; err != nil {
        fmt.Println("Divide error:", err)
      } else {
        fmt.Printf("Divide: %d/%d=%d...%d\n", args2.A, args2.B, quo.Quo, quo.Rem)
      }
      divideReplied = true
    case <-ticker.C:
      fmt.Println("tick")
    }
  }
}
```

异步调用使用 `client.Go()` 方法，参数与同步调用基本一样。它返回一个 `rpc.Call` 对象：

```
// src/net/rpc/client.go
type Call struct {
  ServiceMethod string
  Args          interface{}
  Reply         interface{}
  Error         error
  Done          chan *Call
}
```

我们可以通过该对象获取此次调用的信息，如方法名、参数、返回值和错误。我们通过监听通道 `Done` 是否有值判断调用是否完成。上面代码中使用一个 `select` 语句轮询两次调用的状态。注意一点，**如果多个通道都有值，`select` 执行哪个 `case` 是随机的**。所以可能先输出 `divide` 的信息：

```
$ go run client.go
Divide: 15/6=2...3
Multiply: 7*8=56
```

3、TCP

TCP

上面我们都是使用 HTTP 协议来实现 rpc 服务的，`rpc` 库也支持直接使用 TCP 协议。首先，服务端先调用 `net.Listen("tcp", ":1234")` 创建一个监听某个 TCP 端口的监听器（Accepter），然后使用 `rpc.Accept(l)` 在此监听器上接受连接并处理：

```
func main() {
  l, err := net.Listen("tcp", ":1234")
  if err != nil {
    log.Fatal("listen error:", err)
  }

  arith := new(Arith)
  rpc.Register(arith)
  rpc.Accept(l)
}
```

然后，客户端调用 `rpc.Dial()` 以 TCP 协议连接到服务端：

```
func main() {
  client, err := rpc.Dial("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("Arith.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```

4、定制方法名

服务端可以继续使用一开始的。

定制方法名

默认情况下，`rpc.Register()` 将方法接收者（`receiver`）的类型名作为方法名前缀。我们也可以自己设置。这时需要调用 `RegisterName(name string, rcvr interface{}) error` 方法：

```
func main() {
  arith := new(Arith)
  rpc.RegisterName("math", arith)
  rpc.HandleHTTP()
  if err := http.ListenAndServe(":1234", nil); err != nil {
    log.Fatal("serve error:", err)
  }
}
```

上面我们将注册的方法名前缀改为 `math` 了，客户端调用时传入的方法名也需要相应的修改：

```
func main() {
  client, err := rpc.DialHTTP("tcp", ":1234")
  if err != nil {
    log.Fatal("dialing:", err)
  }

  args := &Args{7, 8}
  var reply int
  err = client.Call("math.Multiply", args, &reply)
  if err != nil {
    log.Fatal("Multiply error:", err)
  }
  fmt.Printf("Multiply: %d*%d=%d\n", args.A, args.B, reply)
}
```