

发现腾讯精彩

首页发现悦读乐问轻知应用我的K吧

搜索

智慧零售

智慧零售研发K吧

腾讯云智慧零售

团队微博团队讨论团队文章团队文档团队活动团队投票团队日历团队会议

go语言http包使用总结

leozhiqin 2018年09月20日 17:23 浏览(132) 收藏(9) 评论(2) 分享

我们在进行服务开发的时候，很多地方会涉及到调用其它服务获取数据，查询信息等情况，而使用http进行调用是其中一种可选的方式。考虑到这种调用是非常频繁的，而且对于我们的业务，其实有很多共性的内容在其中，所以我们在使用go语言进行服务开发时，希望对http的调用进行一些简单的封装。同时封装后还可以加入服务发现逻辑等功能，为上层屏蔽掉底层的一些细节，达到简单易用的目的。所以我们使用go语言提供的http包开发了一个简单的组件，但是在组件使用的过程中，发现http的连接建立后，并没有及时释放，导致服务端建立了大量链接耗尽端口资源，其它请求直接被拒绝服务。

在排查和优化这个问题的过程中，感觉有很多值得总结的地方，在这里记录一下。从我们服务遇到的问题来看，出现大量链接没有释放。那么一定是请求端每个请求都建立了长链接，并且一直没有释放。通过阅读代码发现，这个问题出现的原因是我们需要设置transport的proxy参数，所以代码实现时，每个请求都新建了一个transport实例，而根据go语言transport的实现，http的调用在建立链接时，默认是建立长链接，并且如果不设置IdleConnTimeout参数，那么链接的保持时间是没有限制的（参考：<https://golang.org/src/net/http/transport.go#216>

），也就是直到程序退出才会释放这个链接。

对于这个问题我们需要进行问题复现，并且在后面代码调整的过程中验证我们的改动，所以先实现了一个简单的server和client如下：

```
//server
func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        conn := pool.Get()
        defer conn.Close()
        fmt.Fprintf(w, "Hello!")
    })
    http.ListenAndServe(":8848", nil)
}
```

```
//client
func doReq() {
    contentType := ""
    url := "http://10.12.142.55:8848/test"
```

关于作者



leozhiqin(秦智) SNG云行业产品一部研发一组员工

作者文章

- go语言http包使用总结
- 腾讯视频 Light convs分布式转码系统
- perl网络通信阻塞问题分析

猜你喜欢

- 互动视频后台Go框架入门索引
- Go与Mongodb应用实践
- Redis源码学习之链表
- 为女儿发一枚数字货币
- redis网络框架简介

更多>>

```

retry := uint8(3)
timeout := 500 * time.Millisecond
var req interface{}
rsp, reqErr := util.HttpRpc(url, req, contentType, true, timeout, retry)
fmt.Println("rsp %s, reqErr %s", string(rsp), reqErr)
}
func main() {
    http.DefaultTransport.(*http.Transport).MaxIdleConnsPerHost = 100
    for {
        go doReq()
        go doReq()
        time.Sleep(300 * time.Millisecond)
    }
}

```

在实现了简单的server和client后，我们在不同的机器上启动这两个程序，在server端我们查看链接情况如下：

```

[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50661  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50700  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50690  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50704  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50714  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50678  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50653  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50698  ESTABLISHED
tcp6      0      0 10.12.142.55:8848    10.12.142.122:50686  ESTABLISHED
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848 |wc -l
99
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848 |wc -l
117
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848 |wc -l
135
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848 |wc -l
155
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848 |wc -l
186

```

可以看到成功的链接数量持续增加，复现了我们遇到的问题，下面开始寻找解决方案。首先，我们发现在创建transport实例的过程中，可以传入DisableKeepAlives 这个参数，通过官方文档和源码了解到，如果将这个参数设置为true，那么http在发送请求时，会额外设置Connection这个头的值为close，服务器收到这个请求后会在响应请求后关闭链接（参考：<https://golang.org/src/net/http/transport.go#2065>）。增加这个参数后，我们使用上面的client和server进行测试，发现所有的链接确实请求结束就被关闭了，但是这又带来了另外一个问题，服务端在关闭链接后，链接立即进入了TIME_WAIT状态，TIME_WAIT状态是服务器用来防止端口复用时串包，需要等2MSL（最长报文生存时间）这个端口才能继续使用。如果调用方持续高并发的执行请求，那么也会导致服务器TIME_WAIT量持续增加，最终可能导致服务器无法继续提供服务。server端的情况如下：

```

[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54440  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54419  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54412  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54435  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54443  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54413  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54422  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54423  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54425  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54427  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54421  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54414  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54441  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54433  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54418  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54420  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54437  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54426  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54417  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54429  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54428  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54444  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54436  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54432  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54410  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54439  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54424  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54411  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54442  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54416  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54430  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54431  TIME_WAIT
tcp6      0      0 10.12.142.55:8848 10.12.142.122:54434  TIME_WAIT
[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848|wc -l
102
[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848|wc -l
114
[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848|wc -l
129
[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848|wc -l
141
[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848|wc -l
153
[leozhiqin@TENCENT64 /data/home/leozhiqin]$ netstat |grep 8848|wc -l
169
[leozhiqin@TENCENT64 /data/home/leozhiqin]$

```

所以这种处理方式是没有什么很好的解决问题，接下来想到了让请求端来主动关闭链接，在前面提到了IdleConnTimeout这个参数，如果设置了这个超时时间参数，那么请求端在链接空闲一定时间后，就会主动关闭这个链接，那么服务端就不会出现大量的TIME_WAIT状态了。设置这个参数后，我们在server端的观察证实了这个想法。然而，这种方式也并没有很好的解决问题，调用端高并发时还是会出现大量TIME_WAIT状态。

通过上面两种方式的验证，我们发现大量短链接并发的请求会造成端口回收压力，不管是server端还是client端，总有一方会有这个问题。而且我们知道tcp链接建立时需要进行三次握手，断开时需要进行四次挥手，这个过程其实是既耗时又消耗资源的操作，所以还是需要通过长链接来进行请求。

在go语言中，其实是有对应的链接复用机制的，如果在创建client时，不指定transport，那么会使用默认的transport建立链接，建立的链接是可以复用的，我们在client端使用6个协程并发请求，并每隔300毫秒重复请求一次，在服务端每隔1秒输出8848端口的链接情况，从下图可以看到底层链接被复用了。

```

[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49094  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49095  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49096  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49097  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49098  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49099  ESTABLISHED
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49094  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49095  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49096  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49097  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49098  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49099  ESTABLISHED
[root@TENCENT64 /usr/local/webroot/umall/cgi-bin]# netstat |grep 8848
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49094  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49095  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49096  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49097  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49098  ESTABLISHED
tcp6      0      0 10.12.142.55:8848 10.12.142.122:49099  ESTABLISHED

```

按照这种思路，我们修改了封装的http组件，使得请求相同地址时仅在第一次时创建http的client实例，后续调用时都复用这个实例进行调用。

仅通过服务端链接来看链接是否建立是不太方便的，其实可以在创建链接时加入以下测试代码来更好的观察。

```
func PrintLocalDial(network, addr string) (net.Conn, error) {
    dial := net.Dialer{
        Timeout: 30 * time.Second,
        KeepAlive: 30 * time.Second,
    }
    conn, err := dial.Dial(network, addr)
    if err != nil {
        return conn, err
    }
    fmt.Println("connect done, use", conn.LocalAddr().String())
    return conn, err
}

client = &http.Client{
    Transport: &http.Transport{
        Proxy: http.ProxyURL(proxyUrl),
        Dial: PrintLocalDial,
        IdleConnTimeout: 90 * time.Second,
    },
}
```


修改完成后，再次用上面简单的服务端和请求端进行了测试，在测试时却发现单个协程进行处理时，是没有问题的，正常的复用长链接发送请求，但是当协程并发量达到6个时，每一轮请求都会创建4个新的链接，如果是8个协程并发，会新创建6个链接。

```
[leozhiqin@dev122 ~/go/src/git.code.oa.com/leozhiqin/test_dir]$ ./test_client
connect done, use 10.12.142.122:50318
connect done, use 10.12.142.122:50315
connect done, use 10.12.142.122:50316
connect done, use 10.12.142.122:50319
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
connect done, use 10.12.142.122:50320
connect done, use 10.12.142.122:50321
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
connect done, use 10.12.142.122:50323
connect done, use 10.12.142.122:50322
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
connect done, use 10.12.142.122:50324
connect done, use 10.12.142.122:50325
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
```

同时也会出现大量TIME_WAIT状态，怀疑与单个host最大链接数量有关，查看代码发现，如果未设置MaxIdleConnsPerHost参数，那么会使用DefaultMaxIdleConnsPerHost，而这个值被设置为2（参考：<https://golang.org/src/net/http/transport.go#198>）。考虑到我们存在协程并发请求的情况，我们先将transport中的MaxIdleConnsPerHost设置为100，再次编译测试后，发现没有出现之前新创建链接的情况，所有请求都复用了链接。

```
[leozhiqin@dev122 ~/go/src/git.code.oa.com/leozhiqin/test_dir]$ ./test_client
connect done, use 10.12.142.122:50343
connect done, use 10.12.142.122:50342
connect done, use 10.12.142.122:50341
connect done, use 10.12.142.122:50347
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
rsp %s, reqErr %s Hello! <nil>
AC
```

通过上面的排查验证流程，修复了我们封装的http组件中创建大量链接未回收的问题。在这一过程中，也体会到测试验证的重要性，对于底层实现不太熟悉时，就需要通过测试来辅助我们对提出的解决方案进行分析验证，如果有问题再持续修改优化。



腾讯知识奖201809期评选进行中

点击申报推荐本文参加评选


本文申报

如果觉得我的文章对您有用，请随意赞赏



仅供内部学习与交流，未经授权切勿外传

标签 : [go\(2\)](#) [http\(1\)](#)



本文专属二维码，扫一扫还能分享朋友圈

想要微信公众号推广本文章？[点击获取链接](#)

相关阅读

- Go Http请求EOF错误
- go语言实战向导
- Golang脚本化在HTTP服务的应用探索
- Golang自定义HTTP Serve分组路由实现
- Go程序性能分析



我顶 (8)



收藏 (9)

大家评论



yingjiewang

2018-09-21 09:32:17



 顶  回复




evincai

2018-09-21 16:51:56

全局初始化一个http client 然后复用连接是否更好？

 顶  回复



 切换到更多功能

发表评论