



下载APP



加餐 | 阶段答疑：这些代码里的小知识点你都知道吗？

2021-10-06 叶剑峰

《手把手带你写一个Web框架》

课程介绍 >



讲述：叶剑峰

时长 12:59 大小 11.90M



你好，我是轩脉刃。

上节课国庆特别放送，我们围绕业务架构和基础架构，聊了聊这两种方向在工作上以及在后续发展上的区别，也讲了做系统架构设计的一些术。今天就回归课程，特别整理了关于课程的五个共性问题来解答一下。

Q1、GitHub 分支代码跑不起来怎么办？

GitHub 中的每个分支代码都是可以跑起来的，我本人亲测过了。出现这个问题，可能为有的同学只使用 `go run main.go`。



go run main.go 只会运行编译运行的指定文件，而一旦当前目录下有其他文件，就不会运行到了，所以比如在 geekbang/02 或者 geekbang/03 分支中，根目录下有其他文件，就不能运行了。**你需要使用 go build 先编译，然后使用 ./coredemo 来运行。**

另外因为最近 Go 版本更新了，有同学问到这个问题：go mod 能指定 1.xx.x 版本么？比如想要把 go.mod 中指定 go 版本的 go 1.17 修改为 go 1.17.1，希望我的项目最低要求 1.17.1。但是 Goland 老是把版本号修改回 go 1.17，是不是我哪里设置有问题？

这是一个小知识点，不过估计不是每个人都知道。其实这里不是设置有问题，而是 go.mod 要求就是如此。

指定 go 版本的地方叫 go directive。它的格式是：

[复制代码](#)

```
1 GoDirective = "go" GoVersion newline .
2 GoVersion = string | ident . /* valid release version; see above */
3 The version must be a valid Go release version: a positive integer followed by
```

其中所谓的 valid release version 为必须是像 1.17 这样，前面是一个点，前面是正整数（其实现在也只能是 1），后面是非负整数。

go 的版本形如 1.2.3-pre。

一般最多由两个点组成，其中 1 叫做 major version，主版本，非常大的改动的时候才会升级这个版本。而 2 叫做 minor version，表示有一些接口级别的增加，但是会保证向后兼容，才会升级这个版本。而 3 叫做 patch version，顾名思义，一些不影响接口，但是打了一些补丁和修复的版本。

而最后的 pre 叫做 pre-release suffix。可以理解是和 beta 版本一样的概念，在 release 版本出现之前，预先投放在市场的试用版本。

所以 **go mod 中的格式只能允许 major version 和 minor version**。它认为，使用者关注这两个版本号就行，这样能保证使用者在使用 golang 标准库的时候，源码接口并没有增加和修改，不管你使用什么 patch version，你的业务代码都能跑起来。

Q2、思维导图怎么画？

从第一节课讲 Go 标准库的源码开始，我们就频繁用到思维导图的方法。这个方法非常好用，特别是复杂的逻辑跳转，画完图之后也就对逻辑基本了解了。当时建议课后你也自己做一下，留言区有同学画了自己的思维导图非常棒，逻辑是正确的。

但是在画图的过程中，我们会出现新的问题，尤其是基础不那么扎实的同学，在不能一眼看出来每行代码都是干什么的，比如可能过多关注分支细节，不知道才能更好地剥离出主逻辑代码。

那怎么才能快速分辨什么是主线、什么是细节呢？

我提供一个思路，在使用思维导图的时候，对于比较复杂的逻辑，我们要**在头脑中模拟一下，要实现这个逻辑，哪些是关键步骤，然后用寻找这些关键步骤的方法来去源码中阅读。**

比如 FileServer，是用来实现静态文件服务器的，首先我们先在头脑中有个模拟，我要先对接上 ServerHTTP 方法，然后要把判断请求路径，要是请求的是文件，那么我就把文件内容拷贝到请求输出中不就行了么。

那么是不是这样的呢，我们带着这种模拟查看代码就能找到代码的关键点有两个。

一是 fileHandler，我们能和 ListenAndServe 连接起来，它提供了 ServeHTTP 的方法，这个是请求处理的入口函数：

 复制代码

```
1 // 返回了fileHandler方法
2 func FileServer(root FileSystem) Handler {
3     return &fileHandler{root}
4 }
5
6 func (f *fileHandler) ServeHTTP(w ResponseWriter, r *Request) {
7     // 请求的入口
8     serveFile(w, r, f.root, path.Clean(upath), true)
9 }
```

二是 `FileServer` 最本质的函数，封装了 `io.CopyN`，基本逻辑是：如果是读取文件夹，就遍历文件夹内所有文件，把文件名直接输出返回值；如果是读取文件，就设置文件的阅读指针，使用 `io.CopyN` 读取文件内容输出返回值。这里如果需要多次读取文件，创建 `Goroutine`，并为每个 `Goroutine` 创建阅读指针。

[复制代码](#)

```
1 func serveFile(w ResponseWriter, r *Request, fs FileSystem, name string, redir
2     ...
3     serveContent(w, r, d.Name(), d.ModTime(), sizeFunc, f)
4 }
5
6 func serveContent(w ResponseWriter, r *Request, name string, modtime time.Time
7     ...
8     if size >= 0 {
9         ranges, err := parseRange(rangeReq, size)
10        ...
11        switch {
12            case len(ranges) == 1:
13                ...
14                ra := ranges[0]
15                if _, err := content.Seek(ra.start, io.SeekStart); err != nil {
16                    Error(w, err.Error(), StatusRequestedRangeNotSatisfiable)
17                    return
18                }
19                sendSize = ra.length
20                code = StatusPartialContent
21                w.Header().Set("Content-Range", ra.contentRange(size))
22            case len(ranges) > 1:
23                ...
24                go func() {
25                    for _, ra := range ranges {
26                        part, err := mw.CreatePart(ra.mimeHeader(ctype, size))
27                        ...
28                        if _, err := content.Seek(ra.start, io.SeekStart); err != nil {
29                            pw.CloseWithError(err)
30                            return
31                        }
32                        if _, err := io.CopyN(part, content, ra.length); err != nil {
33                            pw.CloseWithError(err)
34                            return
35                        }
36                    }
37                    mw.Close()
38                    pw.Close()
39                }()
40            }
41            ...
42        }
43        w.WriteHeader(code)
```

```
44     if r.Method != "HEAD" {  
45         io.CopyN(w, sendContent, sendSize)  
46     }  
47 }
```

按照这种先模拟步骤再去对比源码寻找的方式，很好用，即使你的模拟步骤出了问题，也会引导你思考，为什么出了问题？哪里出了问题，促使你更仔细地查看源码。不妨试试。

Q3、http.Server 源码为什么是两层循环？

第一节课用思维导图分析了一下 http.Server 的源码，有同学问了这么一个问题：

go c.serve(connCtx) 里面为什么还有一个循环？c 指的是一个 connection，我理解不是每个连接处理一次就好了吗，为啥还有一个 for 循环呢？

这里其实有一个扩展知识，HTTP 的 keep-alive 机制。

HTTP 层有个 keep-alive，它主要是用于客户端告诉服务端，这个连接我还会继续使用，在使用完之后不要关闭。这个设置会影响 Web 服务的哪几个方面呢？

性能

这个设置首先会在性能上对客户端和服务端性能上有一定的提升。很好理解的是，少了 TCP 的三次握手和四次挥手，第二次传递数据就可以通过前一个连接，直接进行数据交互了。当然会提升服务性能了。

服务器 TIME_WAIT 的时间

由于 HTTP 服务的发起方一般都是浏览器，即客户端，但是先执行完逻辑，传输完数据的一定是服务端。那么一旦没有 keep-alive 机制，服务端在传送完数据之后，会率先发起连接断开的操作。

由于 TCP 的四次挥手机制，先发起连接断开的一方，会在连接断开之后进入到 TIME_WAIT 的状态，达到 2MSL 之久。

设想，如果没有开启 HTTP 的 keep-alive，那么这个 TIME_WAIT 就会留在服务端，由于服务端资源是非常有限的，**我们当然倾向于服务端不会同一时间 hold 住过多的连接，这种 TIME_WAIT 的状态应该尽量在客户端保持。**那么这个 HTTP 的 keep-alive 机制就起到非常重要的作用了。

所以基于这两个原因，现在的浏览器发起 Web 请求的时候，都会带上 connection:keep-alive 的头了。

而我们的 Go 服务器，使用 net/http 在启动服务的时候，则会按照当前主流浏览器的设置，默认开启 keep-alive 机制。服务端的意思就是，只要浏览器端发送的请求头里，要求我开启 keep-alive，我就可以支持。


所以在源码这段服务一个连接的 conn.server 中，会看到有一个 for 循环，在这个循环中循环读取请求。

[复制代码](#)

```
1 // 服务一个连接
2 func (c *conn) serve(ctx context.Context) {
3     c.remoteAddr = c.rwc.RemoteAddr().String()
4     ctx = context.WithValue(ctx, LocalAddrContextKey, c.rwc.LocalAddr())
5
6     ...
7     // 循环读取，每次读取一次请求
8     for {
9         w, err := c.readRequest(ctx)
10        ...
11        // 判断开启keep-alive
12        if !w.conn.server.doKeepAlives() {
13            return
14        }
15        ...
16    }
17 }
```

那要关闭 keep-alive 怎么办呢？你也可以在 for 循环中的 w.conn.server.doKeepAlives 看到，

它判断如果服务端的 disableKeepAlives 不是 0，则设置了关闭 keep-alive，就不进行 for 循环了。

 复制代码

```
1 // 判断是否开启keep-alive
2 func (s *Server) doKeepAlives() bool {
3     // 如果没开keep-alive，或者在shutdown过程中，就返回false
4     return atomic.LoadInt32(&s.disableKeepAlives) == 0 && !s.shuttingDown()
5 }
```

Q5、为什么 context 作为函数的第一个参数？

context 作为第一个参数在实际工作中是非常有用的一个实践。不管是设计一个函数，还是设计一个结构体的方法或者服务，我们一旦养成了将第一个参数作为 context 的习惯，那么这个 context 在相互调用的时候，就会传递下去。这里会带来两大好处：

1. 链路通用内容传递。

在 context 中，是可以通过 WithValue 方法，将某些字段封装在 context 里面，并且传递的。最常见的字段是 traceId、spanId。而在日志中带上这些 ID，再将日志收集起来，我们就能进行分析了。这也是现在比较流行的全链路分析的原理。

2. 链路统一设置超时。

我们在定义一个服务的时候，将第一个参数固定设置为 context，就可以通过这个 context 进行超时设置，而这个超时设置，是由上游调用方来设置的，这样就形成了一个统一的超时设置机制。比如 A 设置了 5s 超时，自己使用了 1s，传递到下游 B 服务的时候，设置 B 的 context 超时时长为 4s。这样全链路超时传递下去，就能保持统一设置了。

Q5、服务雪崩 case 有哪些？

在第二节课中我们完成了添加 Context 为请求设置超时时间，提到超时很有可能造成雪崩，有同学问到相关问题，引发了我对服务雪崩场景的思考，这里我也简单总结一下。

雪崩的顾名思义，一个服务中断导致其他服务也中断，进而导致大片服务都中断。这里我们最常见的雪崩原因有下列几个：

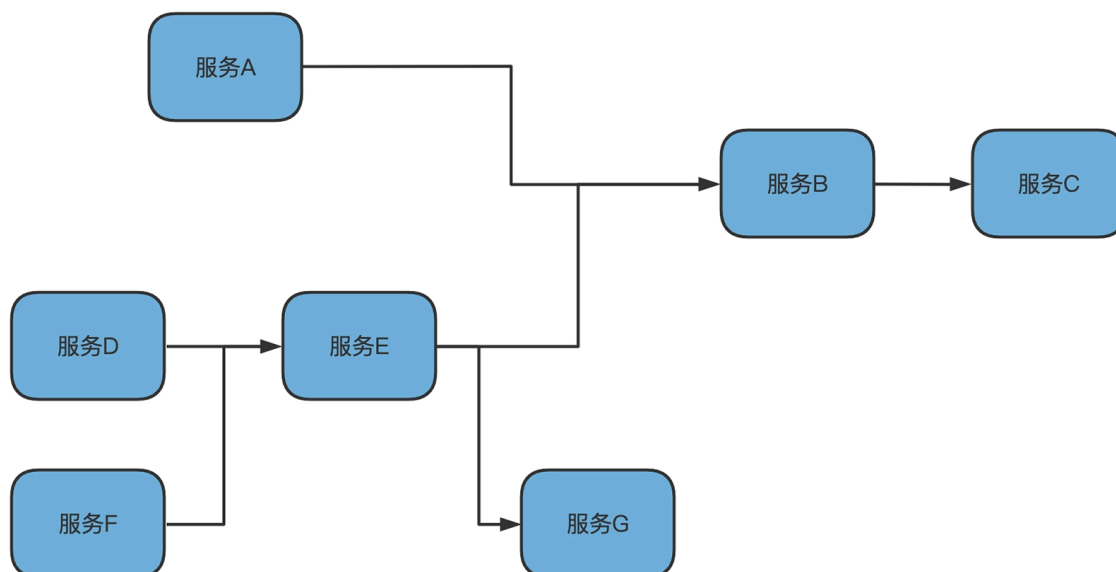
超时设置不合理

服务雪崩最常见的就是**下游服务没设置超时**，导致上游服务不可用，也是我们设置 Context 的原因。



极客时间

比如像上图的，A->B->C，C 的超时不合理，导致 B 请求不中止，而进而堆积，B 服务逐渐不可用，同理导致 A 服务也不可用。而在微服务盛行的链式结构中这种影响面会更大。



极客时间

按照前面的分析，除了 G 之外，其他的节点都会收到波及。

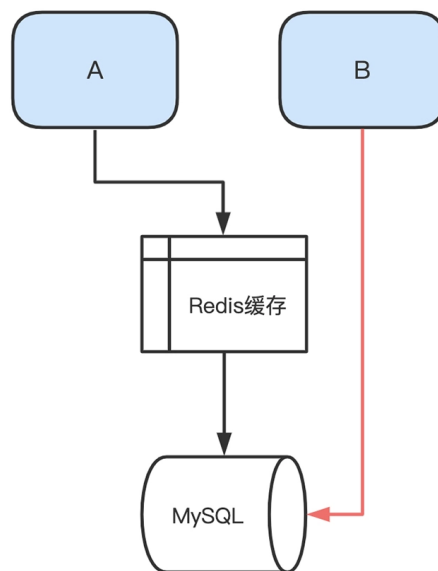
重试加大流量

我们在下游调用的时候，经常会使用重试机制来防止网络抖动问题，但是**重试机制一旦使用不合理**，也有可能導致下游服务的不可用。

理论上，越下层的服务可承受的 QPS 应该越高。在微服务链路中，有某个下游服务的 QPS，比如上图中 C 的 QPS 没有预估正确，当正常请求量上来，C 先扛不住，而扛不住返回的错误码又会让上游服务不断增加重试机制，进一步加剧了下游服务的不可用，进而整个系统雪崩。

缓存雪崩

缓存雪崩顾名思义就是，**原本应该打在缓存中的请求全部绕开缓存，打到了 DB**，从而导致 DB 不可用，而 DB 作为一个下游服务节点，不可用会导致上游都出现雪崩效应（这里的 DB 也有可能是各种数据或者业务服务）。



 极客时间

为什么会出现缓存雪崩呢，我列了一下工作中经常遇到的缓存导致雪崩的原因，有如下三种：

1. 被攻击

在平时写代码中我们日常使用这样的逻辑：“根据请求中的某个值建立 key 去缓存中获取，获取不到就去数据库中获取”。但是这种逻辑其实很容易被攻击者利用，攻击者**只需要建立大量非合理的 key**，就可以打穿缓存进入数据库进行请求。请求量只要足够大，就可以导致绕过缓存，让数据库不可用。

2. 缓存瞬时失效

“通过第一个请求建立缓存，建立之后一段时间后失效”。这也是一个经常出现瞬时缓存雪崩的原因。因为有可能在第一次批量建立了缓存后，进行业务逻辑，而**后续并没有更新缓存时长**，那就可能导致批量在统一时间内缓存失效。缓存失效后大批量的请求会涌入后端数据库，导致数据库不可用。

3. 缓存热 key

还有一种情况是缓存中的**某个 key 突然有大批量的请求涌入**，而缓存的分布式一般是按照 key 进行节点分布的。这样会导致某个缓存服务节点流量过于集中，不可用。而缓存节点不可用又会导致大批量的请求穿透缓存进入数据库，导致数据库不可用。

关于留言问题的回答，今天就暂时到这里了，之后也会收集做统一答疑。所以欢迎你继续留言给我。下节课见~

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐 | 国庆特别放送：什么是业务架构，什么是基础架构？

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。