17丨巨人的肩膀：HTTP协议与Go标准库原理

2022-11-17 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

讲述：郑建勋

时长 10:43　大小 9.78M

你好，我是郑建勋。

在正式开始这节课之前，我想给你分享一段话。

> 学校教给我们很多寻找答案的方法。在我所研究的每一个有趣的问题中，挑战都是寻找正确的问题。当 Mike Karels 和我开始研究 TCP 拥堵时，我们花了几个月的时间盯着协议和数据包的痕迹，问：“为什么会失败？”。

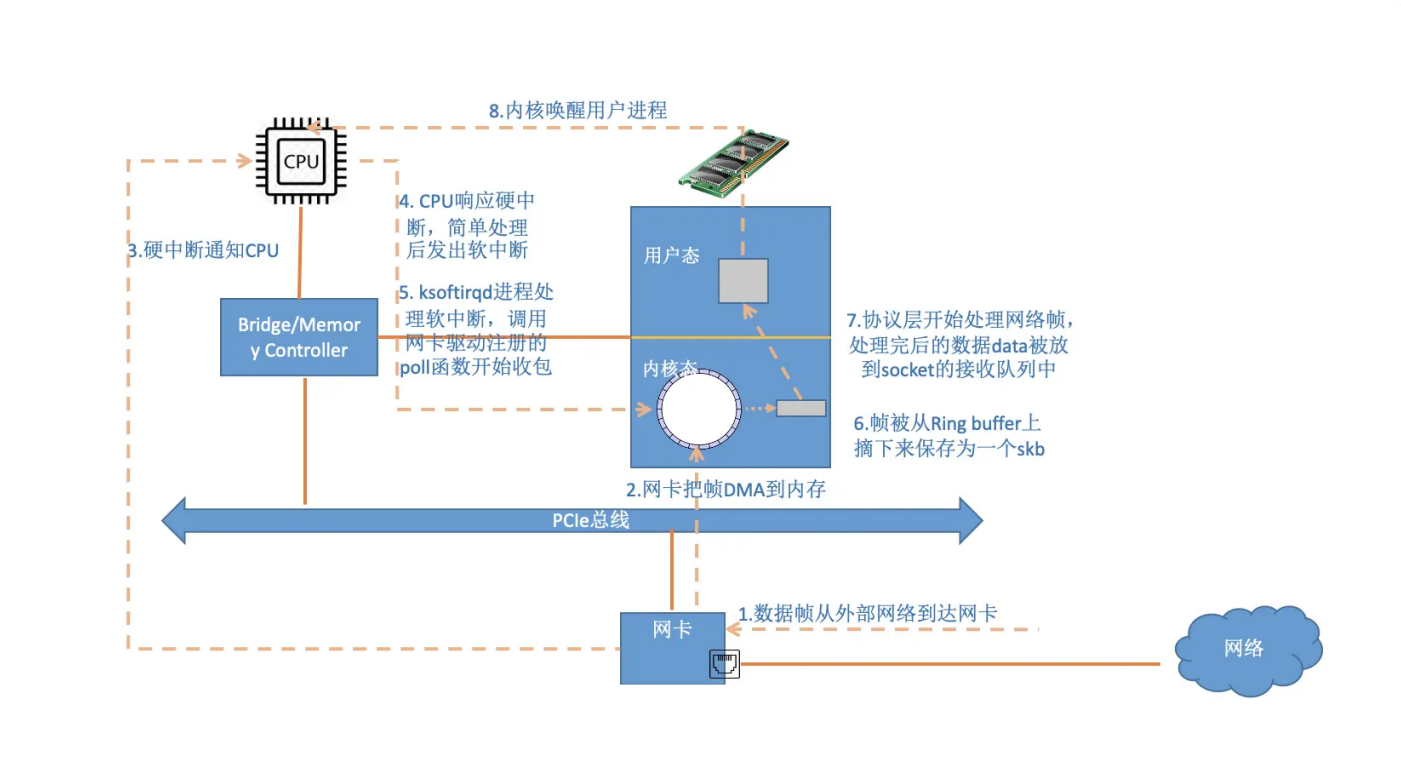
> 有一天，在迈克的办公室里，我们中的一个人说：“我之所以搞不清楚它为什么会失败，是因为我不明白它一开始是怎么工作的。”这被证明是一个正确的问题，它迫使我们弄清楚使 TCP 工作的 “ack clocking”。在那之后，剩下的事情就很容易了。

这段话摘自《Computer Networking A Top-Down Approach 6th》这本书，它启发我们，解决网络方面的问题，核心就是要搞懂它的工作原理。

在上节课，我介绍了一个最简单的 HTTP 请求，了解了一个网络数据包是如何跨过千山万水到达对端的服务器的。不过就像前面引文说的那样，这是不够的。所以这节课，让我们更近一步，来看看当数据包到达对端服务器之后，操作系统和硬件会如何处理数据包，同时我将介绍 HTTP 协议，带你理清 Go HTTP 标准库高效处理网络数据包的底层设计原理。

操作系统处理数据包流程

当数据包到达目的地设备后，数据包需要经过复杂的逻辑处理最终到达应用程序。数据包的处理过程大致可以分为 7 步，我们可以根据下面这张流程图一层一层来理解。



操作系统处理数据包流程 来自《深入理解linux网络》

当网络设备接收到数据，并储存到设备的缓冲区后（该缓冲区可能位于设备的内存，也可能通过 DMA 写入到主机内存），会首先通知操作系统内核对已接收的数据进行处理。

接收到新的数据包后，网卡设备将生成一个硬件中断信号。这个信号通常是由设备发送给中断控制器，再由中断控制器转发给 CPU。CPU 接到信号后，当前执行的任务被打断，转而执行由设备驱动注册的中断处理程序，中断处理程序将处理对应的设备事件。

Linux 将中断处理程序分为两个部分：上半部和下半部，这是深思熟虑的结果。中断处理程序的上半部接受到中断时就立即执行，但是只做比较紧急的工作，这些工作都是在所有中断被禁止的情况下完成的。所以上半部要快，否则其他的中断就得不到及时的处理，导致硬件数据丢

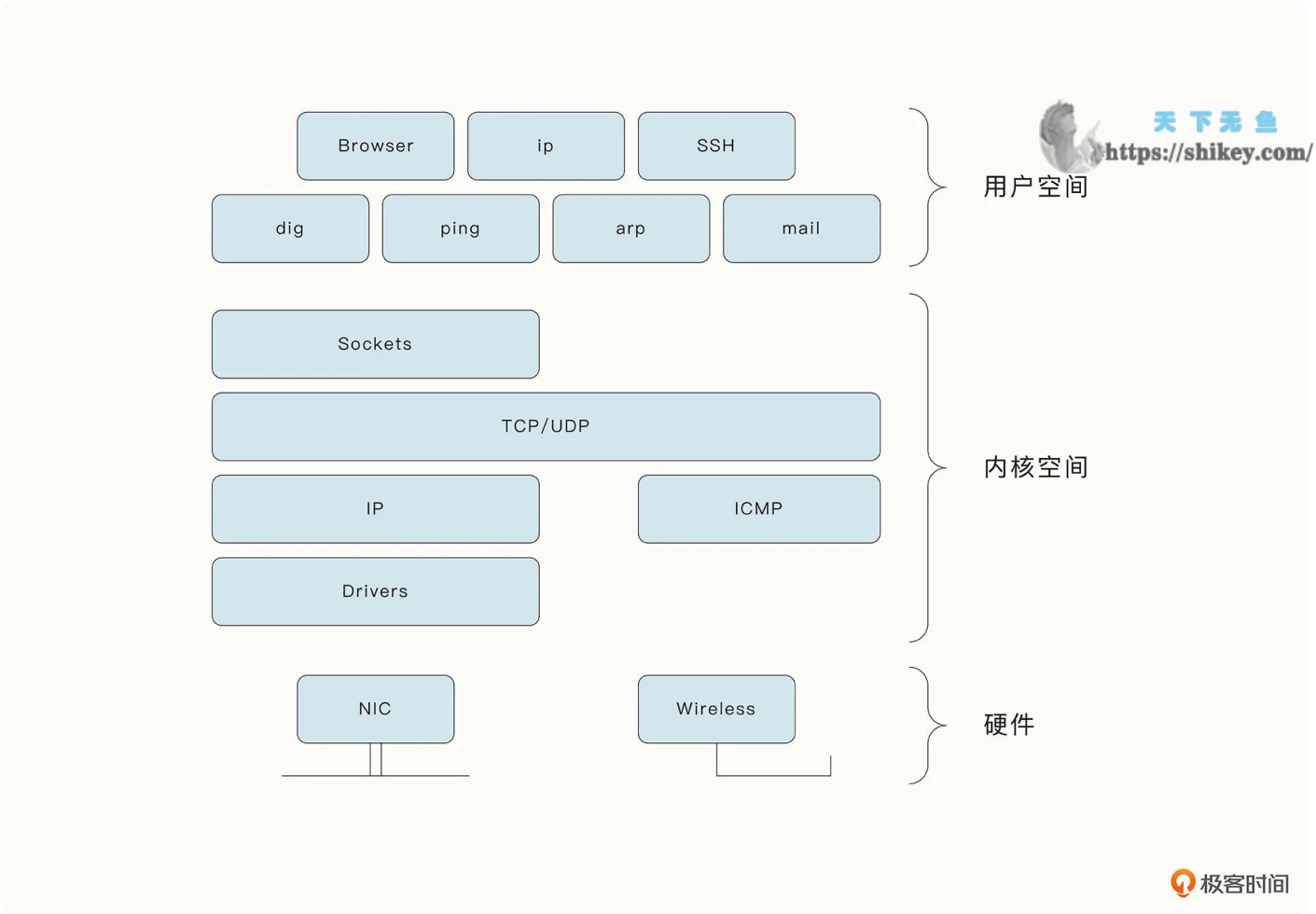
失。而耗时又不紧急的工作被推迟到下半部去做。上半部处理程序会将数据帧加入到内核的输入队列中，通知内核做进一步处理，并快速返回。

硬中断完成后，有可能会直接执行下半部处理程序，也有可能有更重要的任务要执行。当出现第二种情况，在后续会由每一个 CPU 中都维护的 ksoftirqd 内核线程完成下半部处理程序。下半部处理程序中，操作系统会检查并剥离数据包 Header，判断当前数据包应该被哪一个上层协议接收，依次传递到上层处理。整个过程中还穿插了 hook 函数，可以提供防火墙和拦截的功能（这是 iptables 等工具的工作原理）。

要深入了解 Linux 操作系统在这一过程的细节，你还可以参考《Understanding Linux Network Internals》这本书。

还记得之前提到的分层网络模型吗, 现代操作系统在处理网络协议栈时，链路层 Ethernet 协议、网络层 IP 协议、以及传输层 TCP 协议在当前都是在操作系统内核实现的。而应用层是在用户态，由应用程序实现的。应用程序与操作系统之间交流的接口是通过操作系统提供的 Socket 系统调用 API 完成的。

下面这张图列出了硬件、操作系统内核、用户态空间中分别对应的组件和交互。操作系统与硬件之间通过设备驱动进行通信，而应用程序与操作系统之间通过 Socket 系统调用 API 进行通信。



硬件、内核、用户空间对应的组件

Socket 相关的网络编程 API 用于实现连接、断开、监听、IO 多路复用等功能。如果你想进一步了解 Socket 网络编程的细节，可以参考《UNIX Network Programming》。

另外，有许多工具可以方便查看、调试网络相关的状态，例如 netstat 工具可以查看网络连接状态，ss 工具可以查看与 Socket 相关的信息。更多的网络工具还可以查看《Systems Performance, 2nd Edition》。

HTTP 协议解析

之前我们提到，TCP 协议的处理是在操作系统内核实现的。操作系统最终会剥离 TCP Header，识别具体的端口号，并通过 Socket 接口将数据传递到指定的应用程序，例如浏览器。当今使用最广泛的网络协议 HTTP 就位于应用层，对 HTTP 协议的处理也是在应用程序中完成的。

Go 语言提供的 HTTP 库对 HTTP 协议进行了深度封装，这样开发者就不用自己实现协议中各种复杂和特殊的情况了，这是一种生产力的释放。

不过，就像武侠小说中学会了武功招式还需要深厚的内功才能发挥出招式的威力一样，开发的便利并不意味着我们不用了解 HTTP 协议。

例如，服务器返回了一个 499 状态码是什么意思；如何让程序模拟浏览器访问服务器；如果使用 HTTP 代理访问外部网络，HTTP 协议有什么问题；为什么需要 HTTPS、HTTP/2。要回答这些问题，都需要我们对 HTTP 协议有深入的了解。

下面我用 curl 命令访问外部网站，帮助你在这个过程中更好地理解 HTTP 协议。

复制代码

```
» curl www.baidu.com -vvv
*   Trying 110.242.68.3...
* TCP_NODELAY set
* Connected to www.baidu.com (110.242.68.3) port 80 (#0)
> GET / HTTP/1.1
> Host: www.baidu.com
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Accept-Ranges: bytes
< Content-Length: 2381
< Content-Type: text/html
< Date: Mon, 27 Jun 2022 16:04:17 GMT
< Set-Cookie: BDORZ=27315; max-age=86400; domain=.baidu.com; path=/
<
<!DOCTYPE html>
...
* Closing connection 0
```

使用 curl 命令的 -vvv 标识可以打印出详细的协议日志，下面我逐行解读一下。

- 第一行：curl 命令是执行 HTTP 请求的常见工具。其中，www.baidu.com是域名。*
- 第二行通过 DNS 解析出它对应的 IP 地址为 110.242.68.3。
- 第三行：“TCP_NODELAY set ”表明 TCP 正在无延迟地直接发送数据，TCP_NODELAY 是 TCP 众多选项的一个。
- 第四行：“port 80” 代表连接到服务的 80 端口， 80 端口其实是 HTTP 协议的默认端口。
- 有> 标识的 5-8 行数据才是真正发送到服务器的 HTTP 请求。数据 GET / HTTP/1.1 表明当前使用的是 HTTP 的 GET 方法，并且协议版本是 HTTP1.1。

- HTTP 协议可以在请求头中加入多个 key-value 信息。第六行“Host: www.baidu.com”表示当前请求的主机，我们这里是百度的域名。第七行“User-Agent”表示最终用户发出 HTTP 请求的计算机程序，在这里是 curl 工具。

- 另外，第七行 Accept 标头还告诉我们 Web 服务器客户端可以理解的内容类型。这里 */* 表示类型不限，可能是图片、视频、文字等。
- 十到十七行表示服务端返回的信息，它们的规则跟前面类似。“HTTP/1.1 200 OK ”是服务器的响应。服务器使用 HTTP 版本和响应状态码进行响应。状态码 1XX 表示信息，2XX 表示成功，3XX 表示重定向，4XX 表示请求有问题，5XX 表示服务器异常。在这里状态码为 200 ，说明响应成功了。
- 第十二行：“Content-Length: 2381 ”表明服务器返回消息的大小为 2381 字节。下一行的“Content-Type: text/html ”表明当前返回的是 HTML 文本。
- 第十四行：“Date: Mon, 27 Jun 2022 16:04:17 GMT ”是当前 Web 服务器生成消息时的格林威治时间。
- 第十五行：“Set-Cookie”的意思是，服务器让客户端设置 cookie 信息，这样客户端再次请求该网站时，HTTP 请求头中将带上 cookie 信息，这样服务器可以减少鉴权操作，从而加快消息处理和返回的速度。
- 一连串响应头的后面是百度返回的 HTML 文本，这就是我们在浏览器上访问页面的源代码，并最终被浏览器渲染。
- 最后 * Closing connection 0 表明连接最终被关闭。

如果你想更深入地学习 HTTP 协议，推荐你阅读《HTTP: The Definitive Guide》 和《High Performance Browser Networking》这两本书。

HTTP 协议的困境

HTTP1.0 版本诞生于 1995 年，在 20 余年里，HTTP 协议已经成为了最广泛使用的网络协议。但是随着互联网的快速发展，网络环境发生了不少变化，包括：

- 更多的资源分布在不同的主机中，当我们访问一个网页时，这个网页的图片等资源可能来自于外部几十个网站；
- 资源占用的空间也越来越大；
- 另外，相对于网络带宽，网络往返延迟变成最大的瓶颈。

在这个背景下，HTTP 开始面临一系列性能问题，例如头阻塞问题（Head of Line）等。这就有了后续一系列对 HTTP 协议的优化，并产生了 HTTP/2 和 QUIC 协议。

如果你想更深入地学习 HTTP/2 协议，推荐你阅读《HTTP/2 in Action》。如果你想深入了解 QUIC 协议，推荐你阅读 QUIC 的协议文档以及这篇解释 HTTP/3 的文章。

HTTP 请求底层原理

我之前在第 7 讲已经介绍过，借助 epoll 多路复用的机制和 Go 语言的调度器，Go 可以在同步编程的语义下实现异步 I/O 的网络编程。在这个认知基础上，我们继续来看看 HTTP 标准库如何高效实现 HTTP 协议的请求与处理。

我们还是继续使用发送请求这个简单的例子。当 http.Get 函数完成基本的请求封装后，会进入到核心的主入口函数 Transport.roundTrip，参数中会传递 request 请求数据。

Transport.roundTrip 函数会选择一个合适的连接来发送这个 request 请求，并返回 response。整个流程主要分为两步：

- 使用 getConn 函数来获得底层 TCP 连接；
- 调用 roundTrip 函数发送 request 并返回 response，此外还需要处理特殊协议，例如重定向、keep-alive 等。

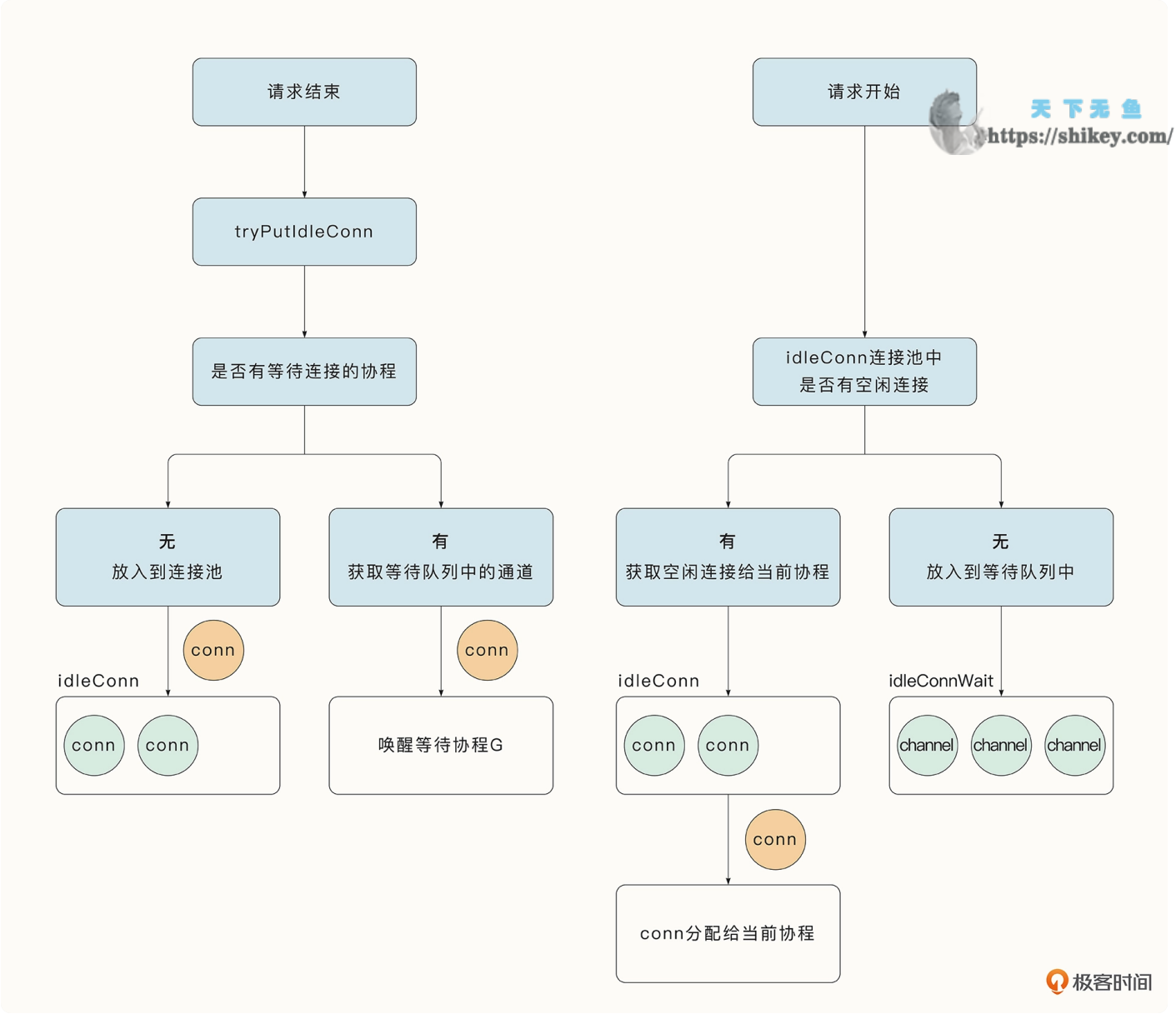
要注意的是，并不是每一次 getConn 函数都需要经过 TCP 的 3 次握手才能建立新的连接。具体的 getConn 函数如下：

复制代码

```
// 获取链接
func (t *Transport) getConn(treq *transportRequest, cm connectMethod) (pc *pers
    ...

  // 第一步，查看idle conn连接池中是否有空闲链接，如果有，则直接获取到并返回。如果没有，当前w
  if delivered := t.queueForIdleConn(w); delivered {
    pc := w.pc
    return pc, nil
  }

  // 如果没有闲置的连接，则尝试与对端进行tcp连接。
    // 注意这里连接是异步的，这意味着当前请求是有可能提前从另一个刚闲置的连接中拿到请求的。这取
  t.queueForDial(w)
```

```
  // Wait for completion or cancellation.
  // 拿到conn后会close(w.ready)
  select {
  case <-w.ready:
    return w.pc, w.err
   // 处理请求的退出与
  case <-req.Cancel:
    return nil, errRequestCanceledConn
     ...
    }
    return nil, err
  }
}

```


可以看到，Go 标准库在这里使用了连接池来优化获取连接的过程。之前已经与服务器完成请求的连接一般不会立即被销毁（HTTP/1.1 默认使用了 keep-alive:true，可以复用连接），而是会调用 tryPutIdleConn 函数放入到连接池中。通过下图左侧的部分可以看到请求结束后连接的归宿（连接也可能会直接被销毁，例如请求头中指定了 keep-alive 属性为 false 或者连接已经超时）。



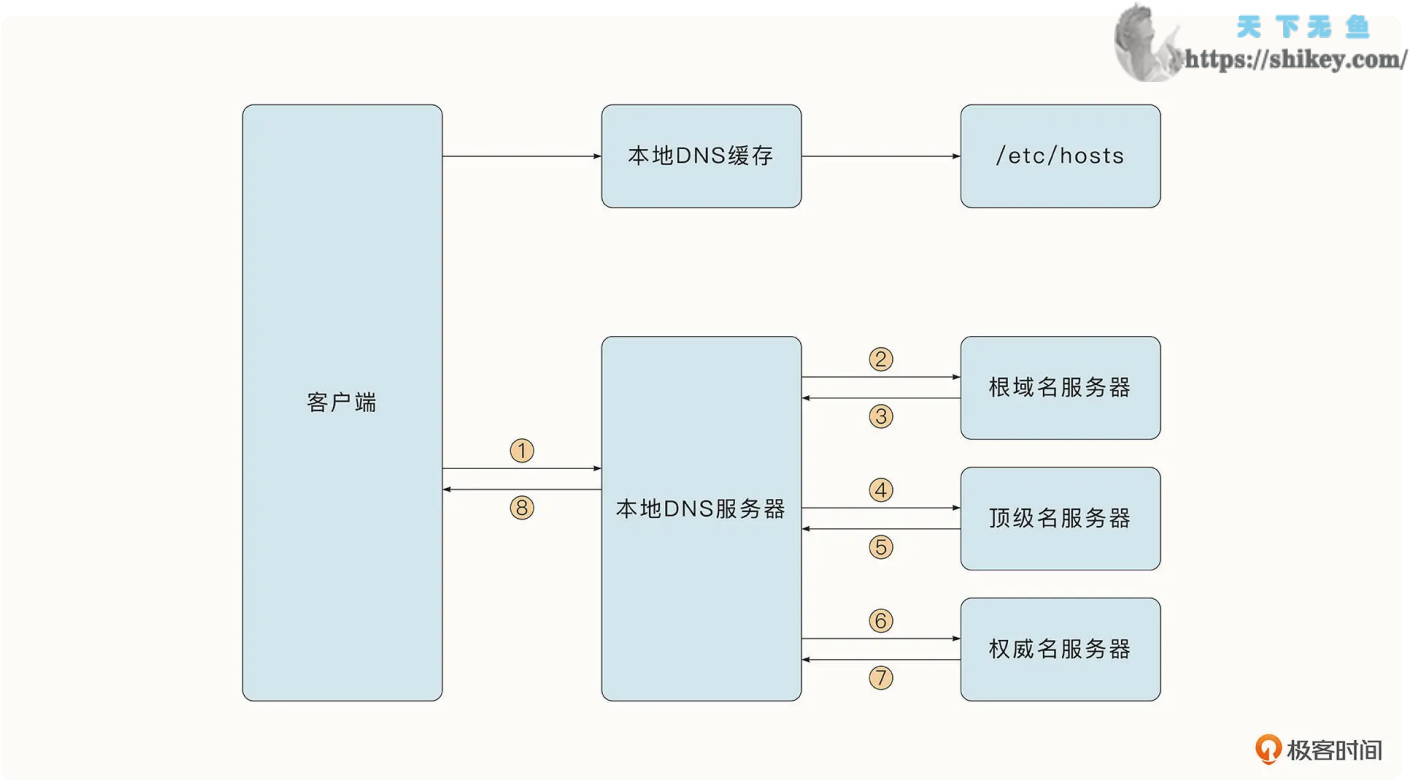
使用连接池的收益是非常明显的，因为复用连接之后就不用再进行 TCP 三次握手了，这大大减少了请求的时间。举个特别的例子，在使用了 HTTPS 协议时，在三次握手基础上还增加了额外的鉴权协调，初始化的建连过程甚至需要花费几十到上百毫秒。

另外连接池的设计也很有讲究，例如连接池中的连接到了一定的时间需要强制关闭。获取连接时的逻辑如下：

- 当连接池中有对应的空闲连接时，直接使用该连接；
- 当连接池中没有对应的空闲连接时，正常情况下会通过异步与服务端建连的方式获取连接，并将当前协程放入到等待队列中。

连接的第一步是通过 Resolver.resolveAddrList 方法访问 DNS 服务器，获取 www.baidu.com 网站对应的 IP 地址。下面这张图展示了借助 DNS 协议查找域名对应的 IP

地址的过程。

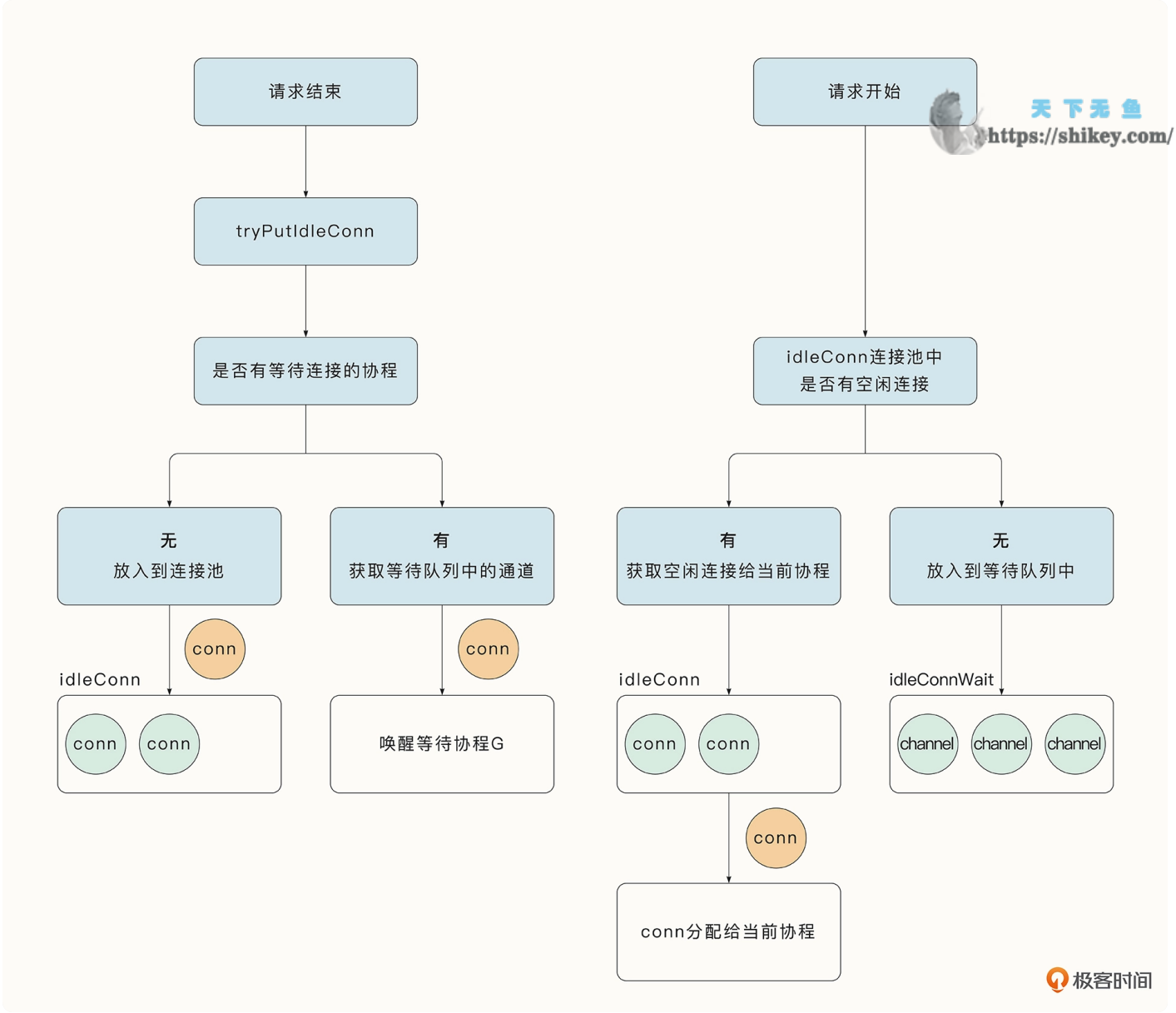


DNS解析流程

客户端首先查看是否有本地缓存，如果没有，则会用递归方式从权威域名服务器中获取 DNS 信息并缓存下来。如果你想深入了解 DNS 协议，可以参考《Introduction to Computer Networks and Cybersecurity》的第二章。

在与远程服务器建连的过程中，当前的协程会进入阻塞等待的状态。正常情况下，当前请求的协程会等待连接完毕。但是因为建立连接的过程还是比较耗时的，所以如果在这个过程中正好有一个其他连接使用完了，协程就会优先使用该连接。

这种巧妙的设计依托了轻量级协程的优势，获取连接的具体流程如下图右侧所示：

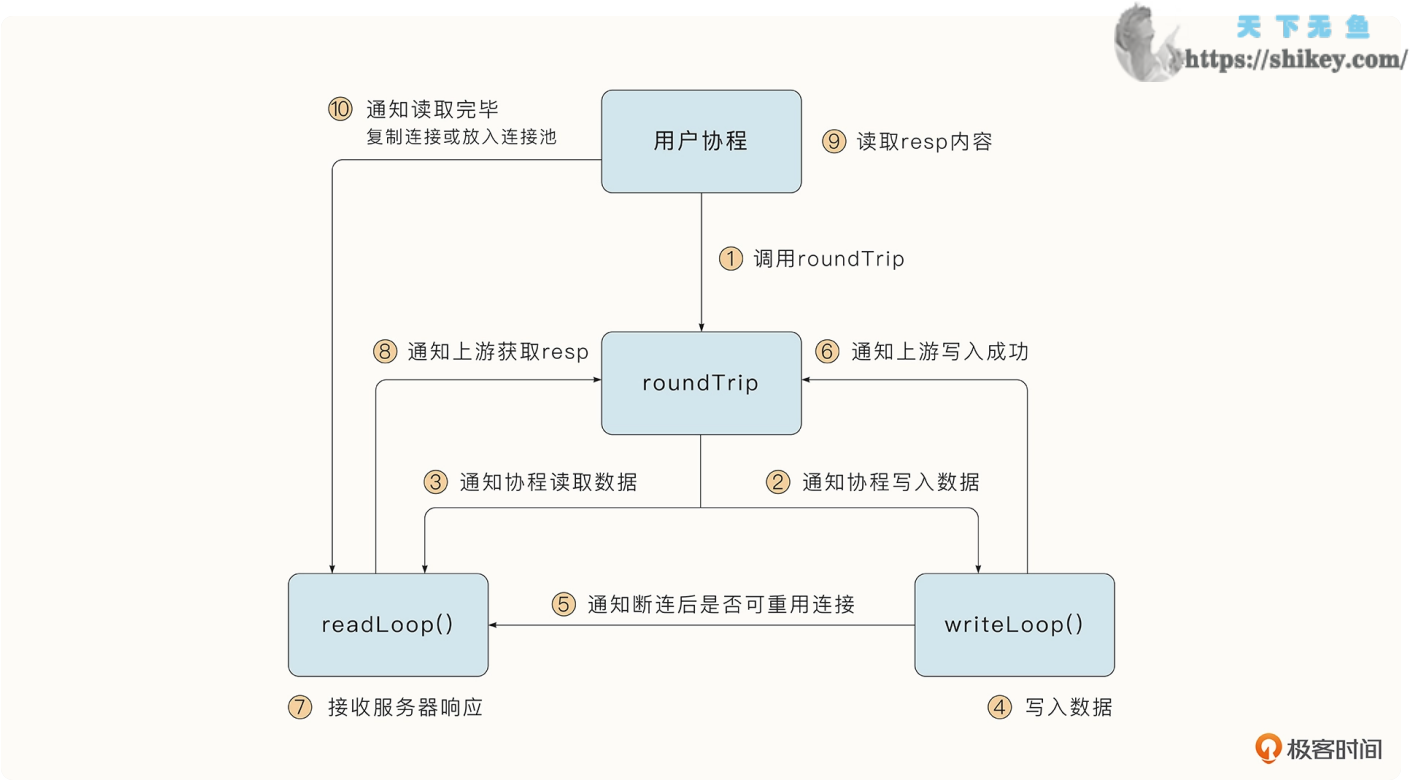


为利用协程并发的优势，Transport.roundTrip 协程获取到连接后，会调用 Transport.dialConn 创建读写 buffer 以及读数据与写数据的两个协程，分别负责处理发送请求和服务器返回的消息：

复制代码

```
func (t *Transport) dialConn(ctx context.Context, cm connectMethod) (pconn *per
  ...
  // buffer
  pconn.br = bufio.NewReaderSize(pconn, t.readBufferSize())
  pconn.bw = bufio.NewWriterSize(persistConnWriter{pconn}, t.writeBufferSize())

  // 创建读写通道，writeLoop用于发送request，readLoop用于接收响应。roundTrip函数中会通过
    // pconn.br给readLoop使用，pconn.bw给writeLoop使用
  go pconn.readLoop()
  go pconn.writeLoop()
}
```

整个处理流程和协程间协调如下图所示：



HTTP请求的完整流程

HTTP 请求调用的核心函数是 roundTrip，它会首先传递请求给 writeLoop 协程，让 writeLoop 协程写入数据。接着，通知 readLoop 协程让它准备好读取数据。等 writeLoop 成功写入数据后，writeLoop 会通知 readLoop 断开后是否可以重用连接。然后 writeLoop 会通知上游写入是否成功。如果写入失败，上游会直接关闭连接。

当 readLoop 接收到服务器发送的响应数据之后，会通知上游并且将 response 数据返回到上游，应用层会获取返回的 response 数据，并进行相应的业务处理 。应用层读取完毕 response 数据后，HTTP 标准库会自动调用 close 函数，该函数会通知 readLoop“数据读取完毕”。这样 readLoop 就可以进一步决策了。readLoop 需要判断是继续循环等待服务器消息，还是将当前连接放入到连接池中，或者是直接销毁。

Go HTTP 标准库使用了连接池等技术帮助我们更好地管理连接、高效读写消息、并托管了与操作系统之间的交互。

总结

数据包到达操作系统之后，借助硬件、驱动、CPU 与操作系统之间的紧密配合，可以被快速处理。操作系统对网络协议做了大量的托管处理，依靠内核对链路层、网络层、传输层进行了处理。而应用层则需要依靠具体的应用程序进行处理。

在内核与应用程序之间，内核暴露了 Socket 编程的 API 和很多 I/O 多路复用的机制，它们被 Go 标准库封装起来，而 HTTP 标准库又在此基础上封装了 HTTP 协议。HTTP 标准库处理复杂多样的协议语言（HTTP、HTTPS、HTTP/2），通过连接池对连接进行复用管理，并且将读写分离为两个协程，高效处理数据并形成清晰的语义。

课后题

学完这节课，我也给你留两道思考题吧。

1. 你认为 Go 标准库的这种实现方式有哪些不足的地方？
2. Go 标准库使用了连接池，你觉得实现一个连接池应该考虑哪些因素？

欢迎你在留言区留下自己思考的结果，也可以把这节课分享给对这个话题感兴趣的同事和朋友，我们下节课再见！

上一篇　16 | 网络爬虫： 一次HTTP请求的魔幻旅途

下一篇　18 | 依赖管理：Go Module 用法与原理

精选留言 (4)

木杉

2022-11-17 来自上海

贴那么多书单 还不如多粘贴一些代码 可以学习呢

kkgo
2022-11-17 来自广东

到现在还没有进入爬虫的设计，太多基础了

4

无尽蔚蓝 神
2022-11-19 来自上海

这一讲的内容应该出现在真正需要使用到该讲的知识的时候，而不是现在就把理论一股脑抛出来。个人觉得，没有实战结合的理论，就是凭空增加认知负荷

1

Geek_crazydaddy
2022-11-17 来自江苏

1.没想到啥不妥的地方，连接池的连接数量？比如大量并行请求可能会有建立大量连接？
2.保留的连接数量以及连接何时释放

1