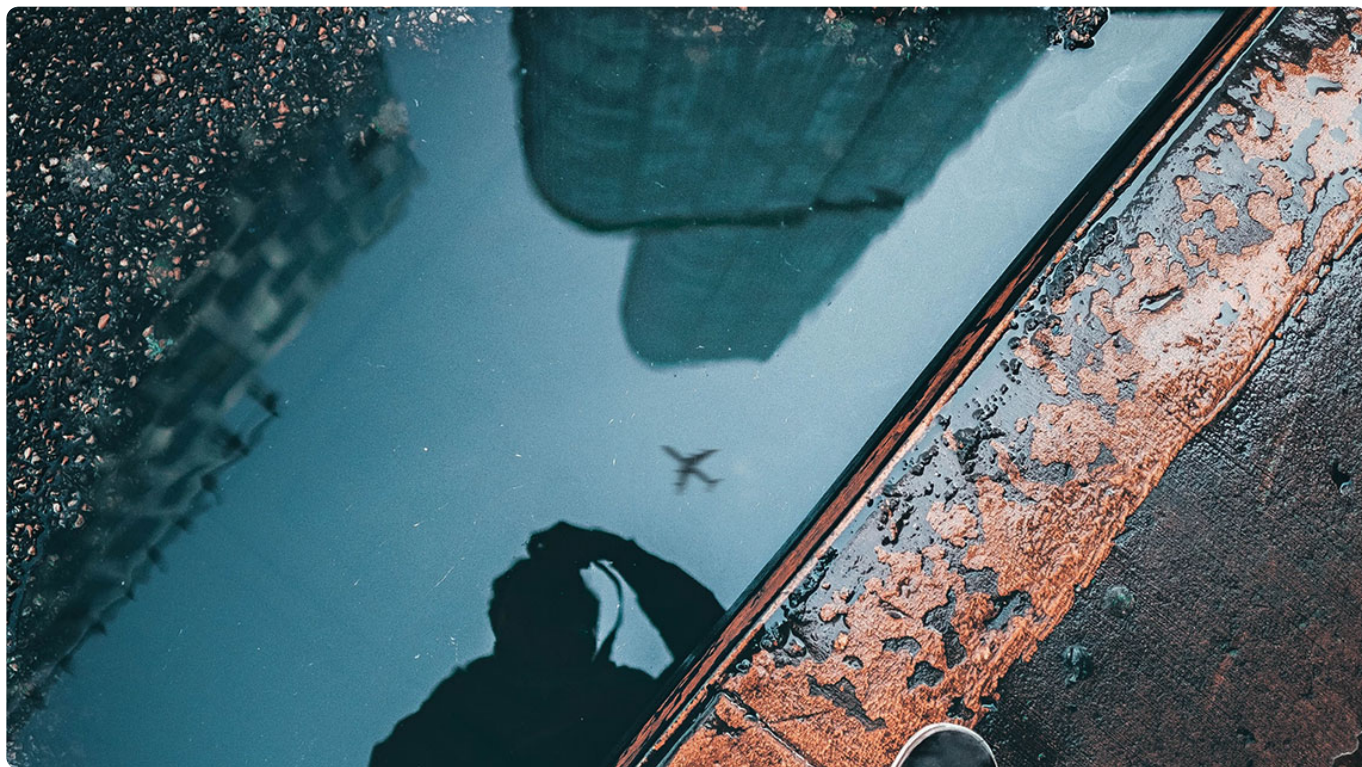


15 | 可编程的互联网世界

2019-06-05 许式伟

许式伟的架构课

[进入课程 >](#)



讲述：姚迪迈

时长 18:24 大小 16.86M



你好，我是七牛云许式伟。

前面我们讨论架构思维的时候说过，架构的第一步是做需求分析。需求分析之后呢？是概要设计。概要设计做什么？是做子系统的划分。它包括这样一些内容：

子系统职责范围的定义；

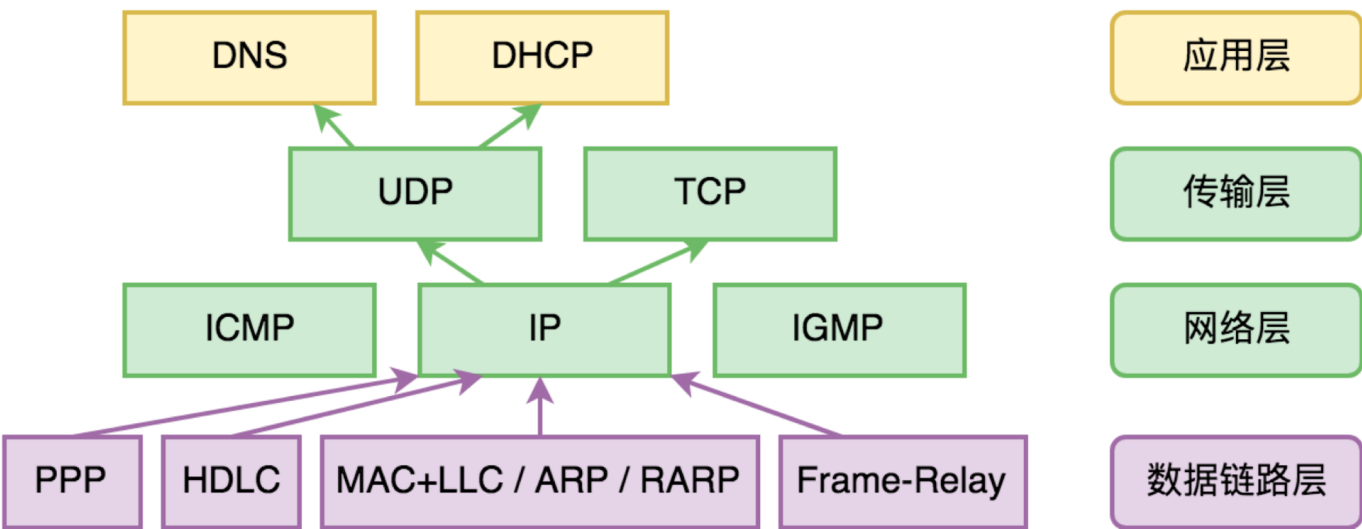
子系统的规格（接口），子系统与子系统之间的边界；

需求分解与组合的过程，系统如何满足需求、需求适用性（变化点）的应对策略。

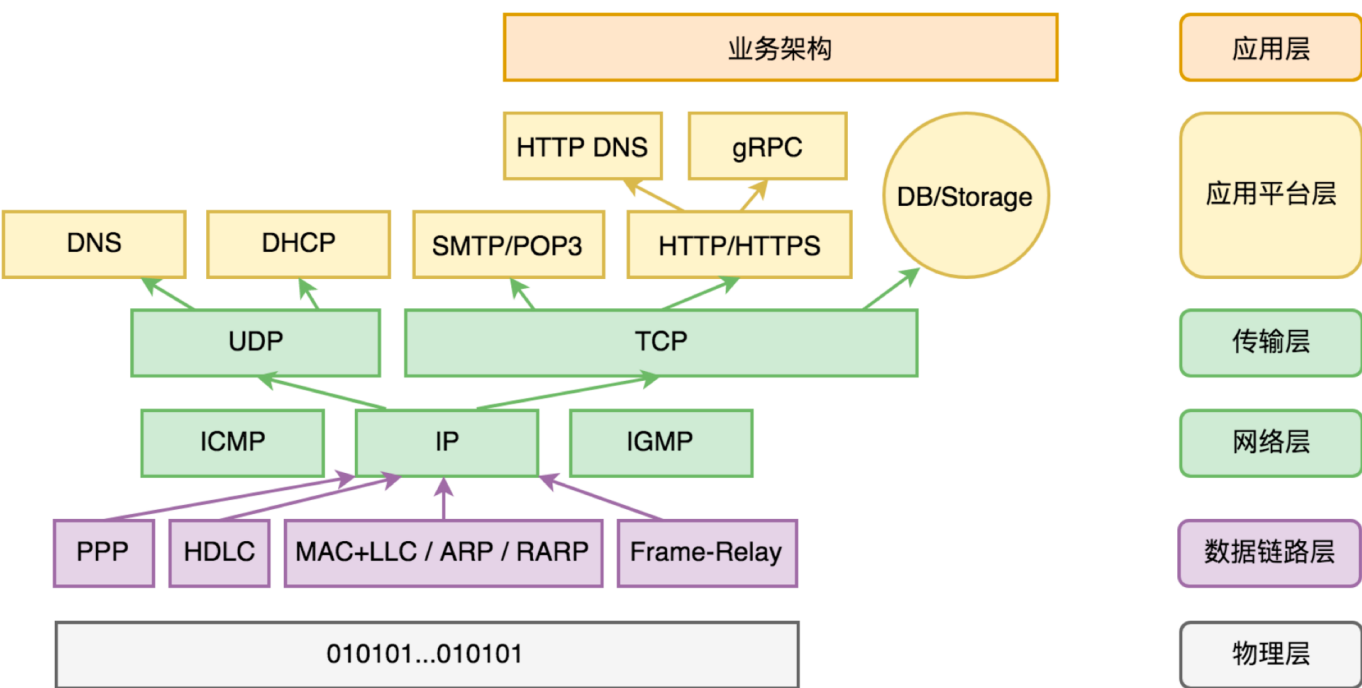
对于我们理解这个精彩的互联网世界来说，理解它的子系统的划分思路是非常非常重要的。

网络应用程序的全视图

在上一节 “14 | IP 网络：连接世界的桥梁” 中我们介绍了 IP 网络的工作原理。我们还画了一幅与数据传输这件事本身有关的网络协议图，如下：



那么，从一个典型的网络应用程序角度来说，它的完整视图又是什么样子的呢？



上图是我给出的答案。当然，它并不代表所有的网络应用程序，但这不影响我们借它的结构来解释网络世界是怎么划分子系统的，每个子系统都负责了些什么。

第一层是物理层。你可以理解为网络设备的原生能力，它定义了硬件层次来看的基础网络协议。

第二层是数据链路层。它负责解决的是局部网络世界的数据传输能力。网络数据传输技术会层出不穷，今天主流有固网、WiFi、3G/4G，明天有 5G/6G，未来也必然还会出现更快速的网络新技术。

这些网络技术虽然都有自己独特的链路层协议，但都可以很自然融入整个互联网世界。原因在于什么？在于 IP 网络。

所以第三层是 IP 网络层，它负责的是互联网世界的一体化，彼此包容与协作。如果拿单机的应用程序的全视图来类比的话，IP 网络类似于单机体系中的操作系统。

在单机体系，操作系统是一台计算机真正可编程的开始。同样地，互联网世界的体系中，IP 网络是互联网“操作系统”的核心，是互联网世界可编程的开始。

第四层是 TCP/UDP 传输层。它也是互联网“操作系统”的重要组成部分，和 IP 网络一起构成互联网“操作系统”的内核。IP 网络解决的是网如何通的问题，而传输层解决的是如何让互联网通讯可信赖的问题，从而大幅降低互联网应用程序开发的负担。

互联网并不是世界上的第一张网。但是只有拥有了 TCP/IP 这一层“操作系统”，这才真正实现了网络价值的最大化：连接一切。

有了操作系统，应用软件才得以蓬勃发展。上图我们列出的应用层协议，仅仅只是沧海一粟。但是，要说当前最主流的应用层协议，无疑当属 HTTP 协议（超文本传输协议，HyperText Transfer Protocol）和 SMTP/POP3 协议了。

HTTP 协议是因为万维网（World Wide Web，简称 WWW）这个应用场景而诞生，冲着传输静态网页而去的。但是由于设计上的开放性，几经演进到今天，已经俨然成为一个通用传输协议了。

通用到什么程度？DNS 地址簿这样的基础协议，也搞出来一个新的 HTTP DNS。当然今天 HTTP DNS 还只是传统 DNS 协议的补充，使用还并不广泛。但由此可知人们对 HTTP 协议的喜爱。

除了呈现网页之外，HTTP 协议也经常用来作为业务开放协议 RESTful API 的承载。另外，一些通用 RPC 框架也基于 HTTP 协议，比如 Google 的 gRPC 框架。

SMTP/POP3 协议是电子邮件 (Email) 应用所采用的, 它们没有像 HTTP 协议那么被广泛借用, 只是局限于电子邮件应用领域。但 SMTP/POP3 协议使用仍然极为广泛, 原因是因为电子邮件是最通用的连接协议, 它连接了人和人, 连接了企业和企业。

我们都很佩服微信的成功, 因为它连接了几乎所有的中国人。但是相比电子邮件, 微信仍然只是小巫见大巫, 因为电子邮件连接了世界上的每一个人和企业。

这是怎么做到的? 因为开放的力量。如果说有谁能够打败微信, 那么我个人一个基本的思考是: 用微信的方式打败微信恐怕很难, 但微信是封闭协议, 开放也许是一个打败微信的机会?

还有其他很多应用层协议上图没有列出来, 比如 FTP、NFS、Telnet 等等。它们大都应用范围相对小, 甚至有一些渐渐有被 HTTP 协议替代的趋势。

对于一个网络应用程序来说, 它往往还依赖存储和数据库 (DB/Storage)。目前存储和数据库这块使用 HTTP 的还不多, 除了对象存储 (Object Storage), 大部分还是直接基于 TCP 协议为主。

对象存储作为一种最新颖的存储类型, 现在主流都是基于 HTTP 协议来提供 RESTful API, 比如七牛云的对象存储服务。

所以你可以看到, 网络应用程序所基于的基础平台, 比单机软件要庞大得多。前面我们介绍的单机软件所依赖的 CPU + 编程语言 + 操作系统就不说了, 它一样要依赖。

上图所示的网络世界所构建的庞大基础平台, 从物理层 -> 数据链路层 -> 网络层 -> 传输层 -> 应用平台层, 也都是我们业务架构的依赖点。选择自定义网络协议, 基于 gRPC, 还是基于 HTTP 提供 RESTful API? 这是架构师需要做出的决策之一。

应用层协议与网关

上一节 “[14 | IP 网络：连接世界的桥梁](#)” 中我们谈到两台主机是如何通讯时, 我们介绍了让局域网主机能够上网的 NAT 技术。NAT 网关本质上是一个透明代理 (中间人), 工作在网络协议的第四层, 即传输层, 基于 TCP/UDP 协议。


如果我们限定传输的数据包一定是某种应用层协议时, 就会出现所谓的应用层网关, 工作在网络协议的第七层, 所以有时候我们也叫七层网关。

我们熟知的 Nginx、Apache 都可以用作应用层网关。应用层协议通常我们采用的是 HTTP/HTTPS 协议。

为什么 HTTP 协议这么受欢迎，甚至获得了传输层协议才有的待遇，出现专用的网关？

这得益于 HTTP 协议的良好设计。

我们一起来看看 HTTP 协议长什么样。先看获取资源的 GET 请求（Request）：


 复制代码

```
1 GET /abc/example?id=123 HTTP/1.1
2 Host: api.qiniu.com
3 User-Agent: curl/7.54.0
4 Accept: */*
5
```

HTTP 协议的请求（Request）分协议头和正文两部分，中间以空行分隔。GET 请求一般正文为空。


协议头的第一行是请求的命令行，具体分为三部分，以空格分隔。第一部分为命令，常见有 GET、HEAD、PUT、POST、DELETE 等。第二部分是请求的资源路径。第三部分为协议版本。

协议头从第二行开始，每行均为请求的上下文环境或参数，我们不妨统一叫字段（Field）。格式为：

 复制代码

```
1 字段名：字段值
```

HTTP 服务器收到一个请求后，往往会返回这样一个回复（Response）：

 复制代码

```
1 HTTP/1.1 200 OK
2 Content-Type: text/html; charset=utf-8
```



```
3 Content-Length: 68
4 ETag: W/"fb751fe2cb812eb5d466ed9e3c3cd519"
5
6 <html><head><title>Hello</title></head><body>qiniu.com</body></html>
```

HTTP 请求 (Request) 和回复 (Response) 格式上只有第一行不同。回复的第一行也分为三部分，以空格分割。


第一部分为协议版本。

第二部分是状态码 (Status Code)，用来表征请求的结果，200 表示成功，4xx 通常表示请求 (Request) 本身不合法，5xx 则通常表示 HTTP 服务器有异常。

第三部分是状态文本 (Status Text)，方便接收方看到回复后可以立刻判断问题，而不用去查状态码对应的文档。

当协议正文非空的时候，往往还需要用 Content-Type 字段来指示协议正文的格式。例如这里我们用 text/html 表征返回的协议正文是一个 html 文档。Content-Length 字段则用来指示协议正文的长度。

我们再来看一下修改资源的 POST 请求：

 复制代码

```
1 POST /abc/example HTTP/1.1
2 Host: api.qiniu.com
3 User-Agent: curl/7.54.0
4 Authorization: Qiniu dXNlcj14dXNoaXdlaSZwYXNzd2Q9MTIzCg
5 Content-Type: application/x-www-form-urlencoded; charset=utf-8
6 Content-Length: 18
7
8 id=123&title=Hello
```

和 GET 不一样，修改资源往往需要授权，所以往往会有 Authorization 字段。另外这里我们用 Content-Type 字段表示我们协议正文用了表单 (form) 格式。

最后我们看下删除资源的 DELETE 请求：

```
1 DELETE /abc/example HTTP/1.1
2 Host: api.qiniu.com
3 User-Agent: curl/7.54.0
4 Authorization: Qiniu dXNlcj14dXNoaXdlaSZwYXNzd2Q9MTIzCg
5 Content-Type: application/json
6 Content-Length: 11
7
8 {"id": 123}
```

删除和修改完全类似。除了我这里刻意换了一种 Content-Type，协议正文用 json 格式了。实际业务中当然不是这样，通常会选择一致的表达方法。

大致了解了 HTTP 协议的样子，我们一起来分析一下它到底好在哪里？

毫无疑问，最关键的是它的协议头设计。具体表现在如下这些方面。

极其开放的协议头设计。虽然 HTTP 定义了很多标准的协议头字段（Field），但是用户还是可以加自己的字段，惯例上以 X- 开头。例如，七牛引入了 X-Reqid 作为请求的内部调用过程的跟踪线索。关于 X-Reqid 本专栏后续我们还会继续谈到。

规范了业务的表达范式。虽然业务有千千万万种可能，但是实质上不外乎有什么资源，以及对资源的 CURD（创建 - 修改 - 读取 - 删除）。相对应地，在 HTTP 协议中以“资源路径”表达资源，以 PUT-POST-GET-DELETE 表达 CURD 操作（也有一些服务以 POST 而不是用 PUT 请求来创建资源）。

规范了应用层的路由方式。我们知道，在传输层网络的路由基于 IP 地址，但是对于应用而言，IP 地址是一个无意义的字段，在 HTTP 协议头中，有一个字段是强制的，那就是 Host 字段，它用来表征请求的目标主机。通常，在正式生产环境下它是个域名，比如 api.qiniu.com。以域名来表征目标主机，无疑更加能够体现业务特性。故而，对应用层而言，“域名 + 资源路径”是更好的路由依据，方便进行业务的切分。


正因为 HTTP 协议的这些好处，逐渐地它成为了网络应用层协议的模板。无论业务具体是什么样子的，都可以基于 HTTP 协议表达自己的业务逻辑。

TCP/IP 层编程接口

理解清楚了我们网络应用程序的结构，也理解了我们最主流的应用层协议 HTTP 协议，那么我们就可以考虑去实现一个互联网软件了。

从编程接口来说，网络的可编程性是从网络层 IP 协议开始。这是最底层的网络“操作系统”的能力体现。

从基于 IP 协议的网络视角来看，数据是并不是源源不断的流（stream），而是一个个大小有明确限制的 IP 数据包。IP 协议是无连接的，它可以在不连接对方的情况下向其发送数据。规格示意如下：

 复制代码

```
1 package net
2
3 type IPAddr struct {
4     IP    IP
5     Zone string // IPv6 scoped addressing zone
6 }
7
8 func DialIP(network string, laddr, raddr *IPAddr) (*IPConn, error)
9 func ListenIP(network string, laddr *IPAddr) (*IPConn, error)
10
11 func (c *IPConn) Read(b []byte) (int, error)
12 func (c *IPConn) ReadFrom(b []byte) (int, Addr, error)
13 func (c *IPConn) ReadFromIP(b []byte) (int, *IPAddr, error)
14 func (c *IPConn) Write(b []byte) (int, error)
15 func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error)
16 func (c *IPConn) WriteToIP(b []byte, addr *IPAddr) (int, error)
17 func (c *IPConn) Close() error
```

IP 协议本身只定义了数据的目标 IP，那么这个 IP 地址对应的计算机收到数据后，究竟应该交给哪个软件应用程序来处理收到的数据呢？

为了解决这个问题，在 IP 协议的基础上定义了两套传输层的协议：UDP 和 TCP 协议。它们都引入了端口（port）的概念。

端口很好地解决了软件间的冲突问题。一个 IP 地址 + 端口，我们通常记为 ip:port，代表了软件层面上来说唯一定位的通讯地址。每个软件只处理自己所使用的 ip:port 的数据。

当然，既然 IP 和端口被传输层一起作为唯一地址，端口上一定程度上缓解了 IPv4 地址空间紧张的问题。

虽然从设计者的角度来说，最初端口的设计意图，更多是作为应用层协议的区分。例如 port = 80 表示 HTTP 协议，port = 25 表示 SMTP 协议。

应用协议的多样化很容易理解，这是应用的多样化决定的。尽管从架构的角度，我们并不太建议轻易去选择创造新的协议，我们会优先选择 HTTP 这样成熟的应用层协议。但是随着时间的沉淀，还是会不断诞生新的优秀的应用层协议。

但是，为什么需要有多套传输层的协议（TCP 和 UDP）呢？


还是因为应用需求是多样的。底层的 IP 协议不保证数据是否到达目标，也不保证数据到达的次序。出于编程便捷性的考虑，TCP 协议就产生了。

TCP 协议包含了 IP 数据包的序号、重传次数等信息，它可以解决丢包重传，纠正乱序，确保了数据传输的可靠性。

但是 TCP 协议对传输协议的可靠性保证，对某些应用场景来说并不是一个好特性。最典型的就是音视频的传输。在网络比较差的情况下，我们往往希望丢掉一些帧，但是由于 TCP 重传机制的存在，可能会反而加剧了网络拥塞的情况。

这种情况下，UDP 协议就比较理想，它在 IP 协议基础上的额外开销非常小，基本上可以认为除了引入端口（port）外并没有额外做什么，非常适合音视频的传输需求。

编程接口来说，TCP 的编程接口看起来是这样的：

 复制代码


```
1 package net
2
3 type TCPAddr struct {
4     IP    IP
5     Port  int
6     Zone  string // IPv6 scoped addressing zone
7 }
8
9 func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)
10 func ListenTCP(network string, laddr *TCPAddr) (*TCPLListener, error)
11
```

```

12 func (c *TCPConn) Read(b []byte) (int, error)
13 func (c *TCPConn) Write(b []byte) (int, error)
14 func (c *TCPConn) Close() error
15
16 func (l *TCPListener) Accept() (Conn, error)
17 func (l *TCPListener) AcceptTCP() (*TCPConn, error)
18 func (l *TCPListener) Close() error

```

UDP 的编程接口看起来是这样的：

 复制代码

```

1 package net
2
3 type UDPAddr struct {
4     IP    IP
5     Port int
6     Zone string // IPv6 scoped addressing zone
7 }
8
9 func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)
10 func ListenUDP(network string, laddr *UDPAddr) (*UDPConn, error)
11
12 func (c *UDPConn) Read(b []byte) (int, error)
13 func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
14 func (c *UDPConn) ReadFromUDP(b []byte) (int, *UDPAddr, error)
15 func (c *UDPConn) Write(b []byte) (int, error)
16 func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
17 func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
18 func (c *UDPConn) Close() error

```

对比看，IP 和 UDP 的区别非常小，都是无连接的协议，唯一差别就是 UDPAddr 在 IPAddr 基础上增加了一个端口。也正因为如此，我们很少有应用程序会直接基于 IP 协议来编程。

客户端来说，无论 TCP 还是 UDP，使用方式都很像，其示意代码如下：

 复制代码

```


1 c, err := net.Dial("tcp", addrServer)
2 c.Write(...)
3 c.Read(...)
4 c.Close()

```

net.Dial 背后会根据 network 字段选择调用 DialTCP 还是 DialUDP。然后我们就像操作一个文件一样来操作就行，理解上非常简单，只是 UDP 的读写在应用层面需要考虑可能会丢包。


但是服务端不太一样。服务端并不知道谁会给自己发信息，它只能监听自己的“邮箱”，不时看看是不是有人来信了。

对于 TCP 协议，服务端示意代码如下：

 复制代码

```
1 l, err := net.Listen("tcp", addrServer)
2 for {
3     c, err := l.Accept()
4     if err != nil {
5         错误处理
6         continue
7     }
8     go handleConnection(c)
9 }
```

对于 UDP 协议，服务端示意代码如下：

 复制代码

```
1 c, err := net.ListenUDP("udp", addrServer)
2 for {
3     n, srcAddr, err := c.ReadFromUDP(...)
4     if err != nil {
5         错误处理
6         continue
7     }
8     // 根据 srcAddr.IP+port 确定是谁发过来的包，怎么处理
9 }
```


由于 TCP 基于连接 (connection)，所以每 Accept 一个连接后，我们可以有一个独立的执行体 (goroutine) 去处理它。但是 UDP 是无连接的，需要我们手工根据请求的来源

IP+port 来判断如何分派。

HTTP 层编程接口

尽管基于 TCP/IP 层编程是一个选择，但是在当前如果没有特殊的理由，架构师做业务架构的时候，往往还是优先选择基于 HTTP 协议。

我们简单来看一下 HTTP 层的编程接口：


 复制代码

```
1 package http
2
3 func Get(url string) (*Response, error)
4 func Post(url, contentType string, body io.Reader) (*Response, error)
5 func PostForm(url string, data url.Values) (*Response, error)
6
7 func NewRequest(method, url string, body io.Reader) (*Request, error)
8
9 var DefaultClient = new(Client)
10 func (c *Client) Do(req *Request) (*Response, error)
11
12 func NewServeMux() *ServeMux
13 func (mux *ServeMux) Handle(pattern string, handler Handler)
14 func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))
15 func ListenAndServe(addr string, handler Handler) error
16 func ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
```

对于 HTTP 客户端，使用上要比 TCP/UDP 简单得多，常见情况下直接调用 Get、Post 这些函数调用就满足业务需求。

在需要在 HTTP 协议头写一些额外字段的，会略微麻烦一点，需要先 NewRequest 生成一个请求，并添加一些字段（Field），然后再调用 Client.Do 去发起请求。整体上比调用 Read/Write 这样的基础 IO 函数要简便得多。

对于 HTTP 服务端，使用上的示意代码如下：

 复制代码

```
1 mux := http.NewServeMux()
2 mux.HandleFunc("/abc/example", handleAbcExample)
3 mux.HandleFunc("/abc/hello/", handleAbcHello)
```

简单解释一下，一个 HTTP 服务器最基础的就是需要有根据 “资源路径” 的路由能力，这依赖 ServeMux 对象来完成。

简单对比可以看出，基于 HTTP 协议的编程接口，和基于 TCP/IP 协议裸写业务，其复杂程度完全不可同日而语。前者一个程序的架子已经呈现，基本上只需要填写业务逻辑就好。这也是采纳通用的应用层协议的威力所在。

结语

这一节我们希望给大家呈现的是应用程序的全貌。当然，我们现在看到的仍然是非常高维的样子，后面在 “服务端开发” 一章，我们将进一步展开所有的细节。

在应用层协议介绍上，我们很难有全面的介绍，因而我们把侧重点放在 HTTP 协议的概要介绍上。同样，后面我们在 “服务端开发” 一章会进一步介绍 HTTP 协议。

最后，我们整理了基于 TCP/UDP 协议编程和基于 HTTP 协议编程的主体逻辑。虽然介绍非常简要，但通过对比我们仍然可以感受到业务架构基于成熟的应用层协议的优势所在。

如果你对今天的内容有什么思考与解读，欢迎给我留言，我们一起讨论。网络编程本章就到此结束，后面我们有专门的章节来进一步展开。下一节，我们将探讨操作系统的最后一个子系统：安全管理。


如果你觉得有所收获，也欢迎把文章分享给你的朋友。感谢你的收听，我们下期再见。

许式伟的架构课

从源头出发, 带你重新理解架构设计

许式伟
七牛云 CEO



新版升级: 点击「 请朋友读」, 20位好友免费读, 邀请订阅更有**现金**奖励。

© 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 14 | IP 网络: 连接世界的桥梁

下一篇 16 | 安全管理: 数字世界的守护

精选留言 (16)

 写留言



lckfa李钊

2019-06-05

 9

最近对网络协议很感兴趣, 也订阅了极客的其他网络专栏, 所以相比而言, 对上一节的内容不是很

满意, 觉得没有讲到架构的点上, 今天发现许老师把前一节内容又更新了下, 还是很有心的。看了今天的内容, 觉得老师还是理性的, 上一节是本节的引入点, 现代系统如果没有网络, 谈架构可能都是多余的, 所以网络基础至关重要。还记得大学时基于传统的tcp写...

展开 ▾



fjpcode

2019-06-05

 4

老师的洞察力总能深入到事务的本质, 对网络协议来龙去脉的分析, 让人感觉非常顺畅。

应用平台层网络协议的选择，对一个网络应用项目来说是至关重要的，这方面也踩过不少坑。感谢老师的梳理，对网络协议的认知又深入了一步。



小智e

2019-06-05

👍 3

打卡，再次被老师的高维思想折服

展开 ∨



coderfocus

2019-06-05

👍 1

许老师看问题的角度 高屋建瓴 满满的收获

展开 ∨



Aaron Che...

2019-06-05

👍 1

打卡15 HTTP编程逻辑

展开 ∨



王克

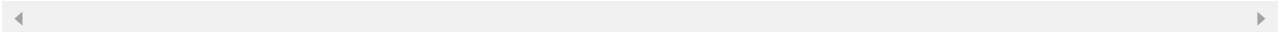
2019-06-06

👍

近几年来HTTPS协议发展的很好，老师能简单的介绍下优缺点吗？

展开 ∨

作者回复: 下一讲谈https



涵

2019-06-06

👍

老师好，请问使用socket编程是不是就是在TCP/UDP层面的？您觉得未来Restful api会占据垄断地位，成为PaaS上的SaaS服务间的标准调用协议吗？谢谢。

作者回复: 1、是的

2、看起来有这个迹象



一笔一画

2019-06-05



老师，请教一下，我可否这样认为，越往上层的协议其效率越低，到HTTP这样高阶的协议，它的性能会不会是一个瓶颈？

作者回复: 不能这么看，是做的事情越来越多。HTTP协议首先是TCP协议的，只不过干了更多的业务，如果不用HTTP，并不代表你可以节约这块的成本，只不过换成你自己写的而已



jaxsong

2019-06-05



之前面试遇到个问题，怎么优化UDP使数据更安全，对这个一直很疑惑
展开 ∨



虎哥

2019-06-05



相对应地，在 HTTP 协议中以“资源路径”表达资源，以 PUT-POST-GET-DELETE 表达 CURD 操作（也有一些服务以 POST 而不是用 PUT 请求来创建资源）。这段写错了吧，post 是 C put 是U吧

展开 ∨

作者回复: 不同人用的惯例不一样，从实际我看到的api案例看，put用的比较少



Elong

2019-06-05



Nginx 可以用作应用层网关，那么应用层网关怎么理解？可以理解成我们通常在 Nginx 中对 HTTP 请求根据不同的api做相关的反向代理、负载均衡等相关配置，也就是所谓的api网关？

展开 ∨

作者回复: 嗯



hua168

2019-06-05



就是讲了常用协议TCP/IP协议族、邮件协议。
概述讲了http、tcp，没讲udp
HTTP可以看《http权威指南》
tcp/ip可以看“tcp/ip详解”系列，3卷，分别为：
《TCP/IP详解 卷1：协议》...

展开 ∨



马哲富

2019-06-05

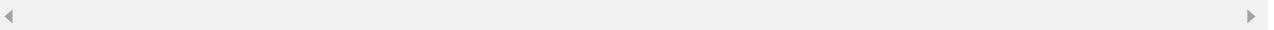


许老师好!

"由于TCP基于连接，所以每Accept一个连接后，我们可以有一个独立的执行体去处理它；但是UDP是无连接的，需要我们手工根据来源IP+Port来判断如何分派"，是否可以理解为基于TCP的协议，服务端每接收一个数据包就处理一个，而基于UDP的协议，服务端接收到数据包之后是先分派后处理？而这个分派的条件是IP+Port，1.为啥要分派？2.为...

展开 ∨

作者回复: 服务器是同时服务很多客户的，不同客户的请求无关，一个客户是通过请求包的来源ip+port来区分。



1900

2019-06-05



"如果我们限定传输的数据包一定是某种应用层协议时，就会出现所谓的应用层网关"，这句话很费解，为什么“传输的数据包”会是“某种应用层协议”？数据包和协议是两个东西吧？

展开 ∨

作者回复: 协议只是数据格式而已



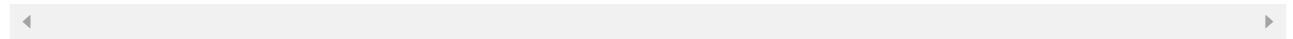
1900

2019-06-05



对象存储作为目前新颖的一种存储类型，它相对于传统网络存储类型的优势在哪里呢？另外它也有什么不足或者局限么？

作者回复: 后面服务端开发会讨论这个问题。最大的区别其实不是协议的选择，而是编程模型的变化。传统网络存储延续了本地文件系统的习惯，基本上都是filesystem的树状元数据组织方式，对象存储是key-value这种平面结构。



觉

2019-06-05



感恩大佬分享 随喜大佬

展开 ∨