



Scan to Follow

收录于合集  
#微服务

7个 >

在单体应用时，一次服务调用发生在同一台机器上的同一个进程内部，也就是说调用发生在本机内部，因此也被叫作本地方法调用。在进行服务化拆分之后，服务提供者和服务消费者运行在两台不同物理机上的不同进程内，它们之间的调用相比于本地方法调用，可称之为远程方法调用，简称RPC（Remote Procedure Call），那么RPC调用是如何实现的呢？

在介绍RPC调用的原理之前，先来想象一下一次电话通话的过程。首先，呼叫者A通过查询号码簿找到被呼叫者B的电话号码，然后拨打B的电话。B接到来电提示时，如果方便接听的话就会接听；如果不方便接听的话，A就得一直等待。当等待超过一段时间后，电话会因超时被挂断，这个时候A需要再次拨打电话，一直等到B空闲的时候，才能接听。

RPC调用的原理与此类似，我习惯把服务消费者叫作 **客户端**，服务提供者叫作 **服务端**，两者通常位于网络上两个不同的地址，要完成一次RPC调用，就必须先建立网络连接。建立连接后，双方还必须按照某种约定的协议进行网络通信，这个协议就是通信协议。双方能够正常通信后，服务端接收到请求时，需要以某种方式进行处理，处理成功后，把请求结果返回给客户端。为了减少传输的数据大小，还要对数据进行压缩，也就是对数据进行序列化。

上面就是RPC调用的过程，由此可见，想要完成调用，你需要解决四个问题：

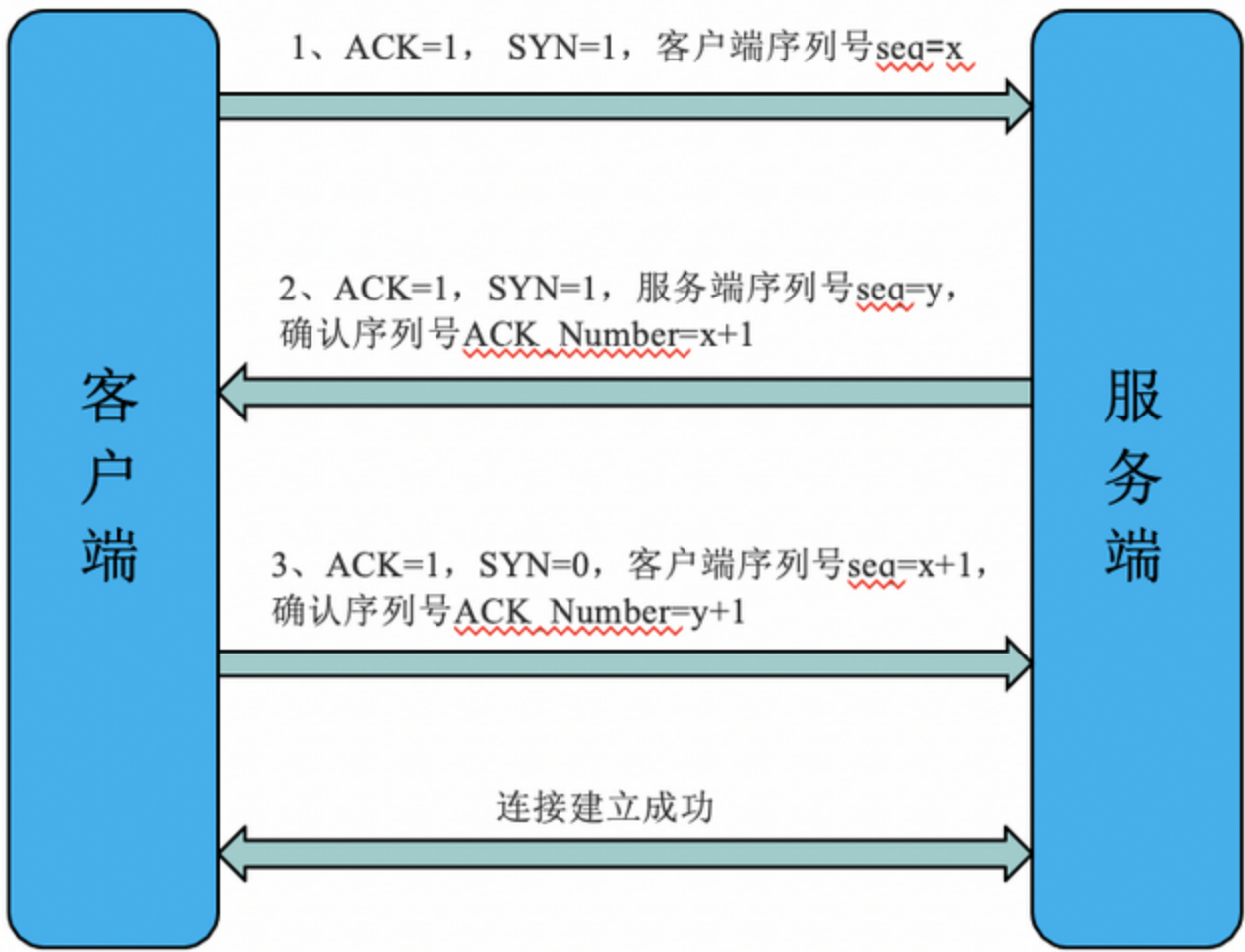
- 客户端和服务端如何建立网络连接？
- 服务端如何处理请求？
- 数据传输采用什么协议？
- 数据该如何序列化和反序列化？

### 客户端和服务端如何建立网络连接？

根据我的实践经验，客户端和服务端之间基于TCP协议建立网络连接最常用的途径有两种。

#### 1. HTTP通信

HTTP通信是基于应用层HTTP协议的，而HTTP协议又是基于传输层TCP协议的。一次HTTP通信过程就是发起一次HTTP调用，而一次HTTP调用就会建立一个TCP连接，经历一次下图所示的“三次握手”的过程来建立连接。

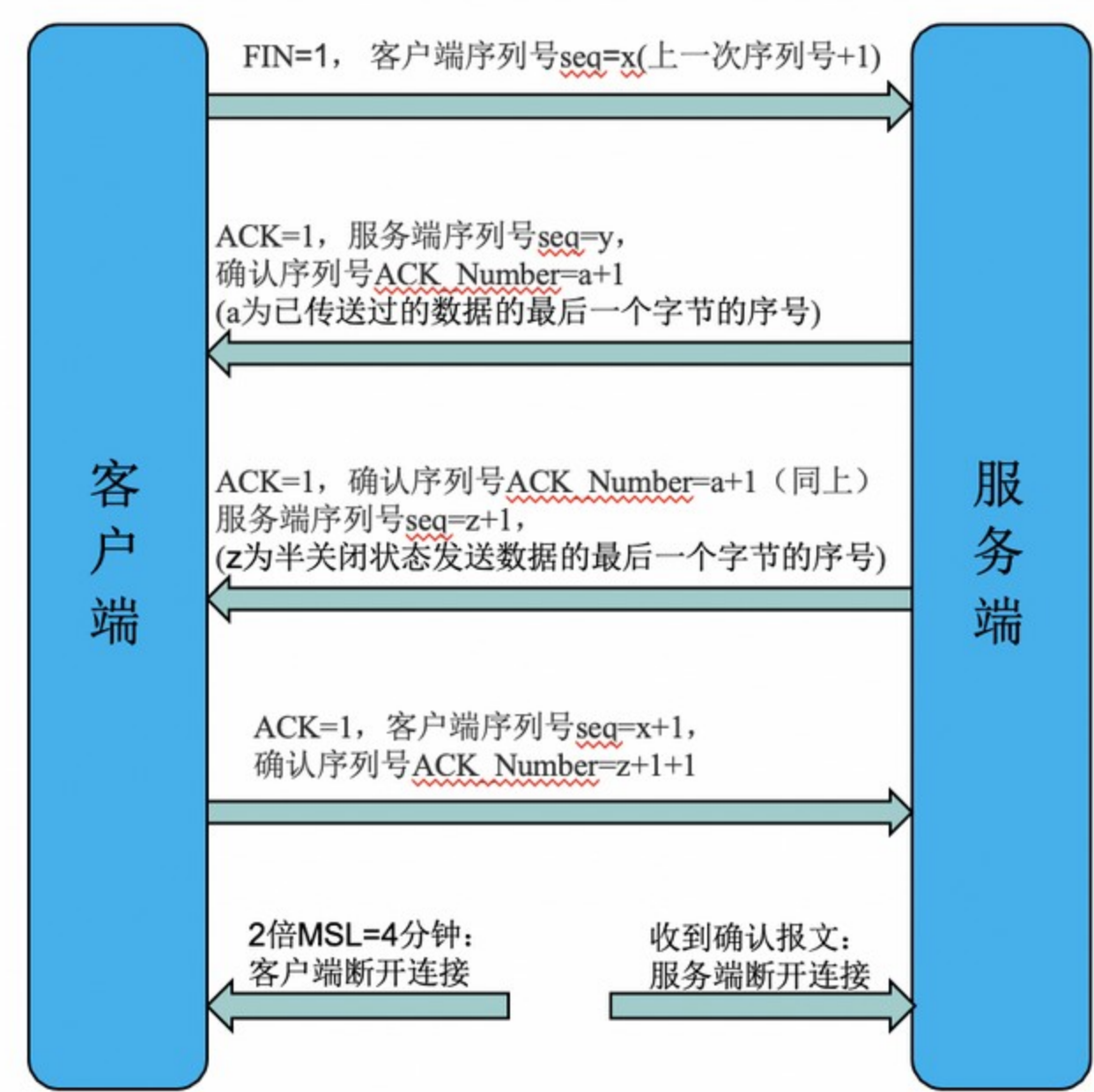


#### 三次握手过程详解

- **第一次握手：**
  - 1、【客户端】向【服务端】发送连接请求报文，标记ACK=1，SYN=1，客户端序列号  $seq=x$ ，客户端进入等待状态。
- **第二次握手：**
  - 1、【服务端】收到请求报文，将收到的报文缓存起来，缓存客户端  $seq=x$
  - 3、【服务端】向【客户端】发送确认报文，生成一个【服务端】  $seq=y$ ，标记ACK=1，SYN=1，【服务端】自己的序列号  $seq=y$ ，确认序列号  $ACK\_Number=x+1$ ，发送给【客户端】
- **第三次握手：**
  - 1、【客户端】收到服务端发送的确认报文，将收到的报文存起来，缓存【服务端】  $seq=y$
  - 2、【客户端】发送确认报文给【服务端】，标记ACK=1，SYN=0，【客户端】自己的序列号  $seq=x+1$ ，确认序列号  $ACK\_Number=y+1$
  - 3、【客户端】、【服务端】都会进入ESTABLISHED (连接已建立状态)



完成请求后，再经历一次“四次挥手”的过程来断开连接。



四次挥手过程详解

- 第一次挥手：
  - 1、【客户端】向【服务端】发送释放连接报文，并停止发送数据，主动关闭 TCP 连接
  - 2、标记FIN=1，【客户端序列号】seq=x，该序号等于前面已经传送过去的的数据的最后一个字节的序号加1
  - 3、这时，客户端 FIN—WAIT-1 (终止等待1)状态，等待服务端确认
- 第二次挥手：
  - 1、【服务端】收到释放连接报文，将收到的报文缓存起来，缓存【客户端】seq=x
  - 2、【服务端】向【客户端】发出确认释放报文，标记ACK=1，【服务端序列号】seq=y，确认序列号ACK\_Number=a+1（a为【服务端】前面已经传送过的数据的最后一个字节的序号）
  - 3、此时【服务端】进入CLOSE—WAIT(关闭等待)状态
  - 4、此时TCP服务器进程应该通知上层的应用进程，因为【客户端】到【服务端】这个方向的连接就释放了，这时TCP处于半关闭状态，即【客户端】已经没有数据要发了，但【服务端】若发送数据，【客户端】仍要接受，也就是说从【服务端】到【客户端】这个方向的连接并没有关闭，这个状态可能会持续一些时间。
- 第三次挥手：
  - 1、【客户端】收到【服务端】确认报文，并缓存起来
  - 2、此时【客户端】进入FIN—WAIT(终止等待2)状态，等待【服务端】发起释放连接报文
  - 3、如果【服务端】没有数据要发送给【客户端】了，【服务端】的应用进程就会通知TCP释放连接
  - 4、此时【服务端】向【客户端】发送释放连接报文，标记FIN=1，确认序列号ACK\_Number=a+1(与第二次挥手的确认号一致)，【服务端序号】seq=z+1(z为半关闭状态发送的数据的最后一个字节的序号)
  - 5、此时【服务端】进入最后确认状态，等待【客户端】确认
- 第四次挥手：
  - 1、【客户端】收到【服务端】释放连接请求，必须发出确认
  - 2、【客户端】向【服务端】发送确认报文，标记ACK=1，确认号序列号ACK\_Numbe=z+1+1，【客户端序号】seq=x+1(x为第一次挥手的seq)
  - 3、此时【客户端】进入等待状态，必须经过时间等待计时器设置的时间2倍MSL(报文最大生存时间)后，【客户端】才进入CLOSED状态，MSL叫做最长报文寿命，RFC建议设为2分钟，实际应用是中是30秒。在这2倍MSL期间【客户端】进入TIME—WAIT状态后，要经过4分钟才能进入到CLOSED状态。
  - 4、【服务端】只要收到了【客户端】的确认后，就进入了CLOSED状态
  - 5、当【客户端】和【服务端】都进入CLOSED状态后，连接就完全释放了

2. Socket通信

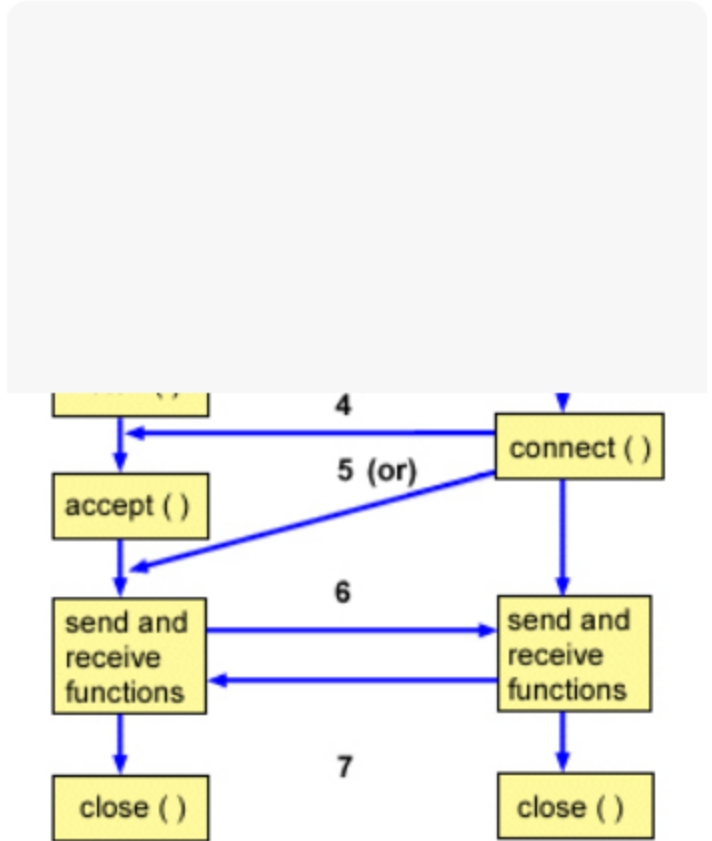
Socket通信是基于TCP/IP协议的封装，建立一次Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket；另一个运行于服务器端，称为ServerSocket。就像下图所描述的，Socket通信的过程分为四个步骤：服务器监听、客户端请求、连接确认、数据传输。

- 服务器监听：ServerSocket通过调用bind()函数绑定某个具体端口，然后调用listen()函数实时监控网络状态，等待客户端的连接请求。
- 客户端请求：ClientSocket调用connect()函数向ServerSocket绑定的地址和端口发起连接请求。



- 服务端连接确认：当ServerSocket监听到或者接收到ClientSocket的连接请求时，调用accept()函数响应ClientSocket的请求，同客户端建立连接。
- 数据传输：当ClientSocket和ServerSocket建立连接后，ClientSocket调用send()函数，ServerSocket调用receive()函数，ServerSocket处理完请求后，调用send()函数，ClientSocket调用receive()函数，就可以得到返回结果。

直接理解可能有点抽象，你可以把这个过程套入前面我举的“打电话”的例子，可以方便你理解Socket通信过程。



当客户端和服务端建立网络连接后，就可以发起请求了。但网络不一定总是可靠的，经常会遇到网络闪断、连接超时、服务端宕机等各种异常，通常的处理手段有两种。

- 链路存活检测：客户端需要定时地发送心跳检测消息（一般是通过ping请求）给服务端，如果服务端连续n次心跳检测或者超过规定的时间都没有回复消息，则认为此时链路已经失效，这个时候客户端就需要重新与服务端建立连接。
- 断连重试：通常有多种情况会导致连接断开，比如客户端主动关闭、服务端宕机或者网络故障等。这个时候客户端就需要与服务端重新建立连接，但一般不能立刻完成重连，而是要等待固定的间隔后再发起重连，避免服务端的连接回收不及时，而客户端瞬间重连的请求太多而把服务端的连接数占满。

## 服务端如何处理请求？

假设这时候客户端和服务端已经建立了网络连接，服务端又该如何处理客户端的请求呢？通常来讲，有三种处理方式。

- 同步阻塞方式（BIO），客户端每发一次请求，服务端就生成一个线程去处理。当客户端同时发起的请求很多时，服务端需要创建很多的线程去处理每一个请求，如果达到了系统最大的线程数瓶颈，新来的请求就没法处理了。
- 同步非阻塞方式（NIO），客户端每发一次请求，服务端并不是每次都创建一个新线程来处理，而是通过I/O多路复用技术进行处理。就是把多个I/O的阻塞复用到同一个select的阻塞上，从而使系统在单线程的情况下可以同时处理多个客户端请求。这种方式的优势是开销小，不用为每个请求创建一个线程，可以节省系统开销。
- 异步非阻塞方式（AIO），客户端只需要发起一个I/O操作后立即返回，等I/O操作真正完成以后，客户端会得到I/O操作完成的通知，此时客户端只需要对数据进行处理就好了，不需要进行实际的I/O读写操作，因为真正的I/O读取或者写入操作已经由内核完成了。这种方式的优势是客户端无需等待，不存在阻塞等待问题。

从前面的描述，可以看出来不同的处理方式适用于不同的业务场景：

- BIO适用于连接数比较小的业务场景，这样的话不至于系统中没有可用线程去处理请求。这种方式写的程序也比较简单直观，易于理解。
- NIO适用于连接数比较多并且请求消耗比较轻的业务场景，比如聊天服务器。这种方式相比BIO，相对来说编程比较复杂。
- AIO适用于连接数比较多而且请求消耗比较重的业务场景，比如涉及I/O操作的图片服务器。这种方式相比另外两种，编程难度最大，程序也不易于理解。

上面两个问题就是“通信框架”要解决的问题，你可以基于现有的Socket通信，在服务消费者和服务提供者之间建立网络连接，然后在服务提供者一侧基于BIO、NIO和AIO三种方式中的任意一种实现服务端请求处理，最后再花费一些精力去解决服务消费者和服务提供者之间的网络可靠性问题。这种方式对于Socket网络编程、多线程编程知识都要求比较高，感兴趣的话可以尝试自己实现一个通信框架。但我建议最为稳妥的方式是使用成熟的开源方案，比如Netty、MINA等，它们都是经过业界大规模应用后，被充分论证是很可靠的方案。

假设客户端和服务端的连接已经建立了，服务端也能正确地处理请求了，接下来完成一次正常地RPC调用还需要解决两个问题，即数据传输采用什么协议以及数据该如何序列化和反序列化。

## 数据传输采用什么协议？

首先来看第一个问题，数据传输采用什么协议？



最常用的有HTTP协议，它是一种开放的协议，各大网站的服务器和浏览器之间的数据传输大都采用了这种协议。还有一些定制的私有协议，比如阿里巴巴开源的Dubbo协议，也可以用于服务端和客户端之间的数据传输。无论是开放的还是私有的协议，都必须定义一个“契约”，以便服务消费者和服务提供者之间能够达成共识。服务消费者按照契约，对传输的数据进行编码，然后通过网络传输过去；服务提供者从网络上接收到数据后，按照契约，对传输的数据进行解码，然后处理请求，再把处理后的结果进行编码，通过网络传输返回给服务消费者；服务消费者再对返回的结果进行解码，最终得到服务提供者处理后的返回值。

通常协议契约包括两个部分：消息头和消息体。其中消息头存放的是协议的公共字段以及用户扩展字段，消息体存放的是传输数据的具体内容。

以HTTP协议为例，主要分为消息头和消息体两部分，其中消息头中存放的是协议的公共字段，比如Server代表是服务端服务器类型、Content-Length代表返回数据的长度、Content-Type代表返回数据的类型；消息体中存放的是具体的返回结果，这里就是一段HTML网页代码。

## 数据该如何序列化和反序列化？

再看第二个问题，数据该如何序列化和反序列化。

一般数据在网络中进行传输前，都要先在发送方一端对数据进行编码，经过网络传输到达另一端后，再对数据进行解码，这个过程就是序列化和反序列化。

为什么要对数据进行序列化和反序列化呢？要知道网络传输的耗时一方面取决于网络带宽的大小，另一方面取决于数据传输量。要想加快网络传输，要么提高带宽，要么减小数据传输量，而对数据进行编码的主要目的就是减小数据传输量。比如一部高清电影原始大小为30GB，如果经过特殊编码格式处理，可以减小到3GB，同样是100MB/s的网速，下载时间可以从300s减小到30s。

常用的序列化方式分为两类：文本类如XML/JSON等，二进制类如PB/Thrift等，而具体采用哪种序列化方式，主要取决于三个方面的因素。

- 支持数据结构类型的丰富度。数据结构种类支持的越多越好，这样的话对于使用者来说在编程时更加友好，有些序列化框架如Hessian 2.0还支持复杂的数据结构比如Map、List等。
- 跨语言支持。序列化方式是否支持跨语言也是一个很重要的因素，否则使用的场景就比较局限，比如Java序列化只支持Java语言，就不能用于跨语言的服务调用了。
- 性能。主要看两点，一个是序列化后的压缩比，一个是序列化的速度。以常用的PB序列化和JSON序列化协议为例来对比分析，PB序列化的压缩比和速度都要比JSON序列化高很多，所以对性能和存储空间要求比较高的系统选用PB序列化更合适；而JSON序列化虽然性能要差一些，但可读性更好，更适合对外部提供服务。

## 总结

今天我们一起了解服务调用需要解决的几个问题：

- 通信框架。它主要解决客户端和服务端如何建立连接、管理连接以及服务端如何处理请求的问题。
- 通信协议。它主要解决客户端和服务端采用哪种数据传输协议的问题。
- 序列化和反序列化。它主要解决客户端和服务端采用哪种数据编解码的问题。

这三个部分就组成了一个完整的RPC调用框架，通信框架提供了基础的通信能力，通信协议描述了通信契约，而序列化和反序列化则用于数据的编/解码。一个通信框架可以适配多种通信协议，也可以采用多种序列化和反序列化的格式，比如服务化框架Dubbo不仅支持Dubbo协议，还支持RMI协议、HTTP协议等，而且还支持多种序列化和反序列化格式，比如JSON、Hession 2.0以及Java序列化等。

收录于合集 #微服务 7

〈 上一篇  
微服务架构之注册中心

下一篇〉  
微服务架构之服务监控与追踪

People who liked this content also liked

1分钟搭建一个VPN服务器

IT仔的笔记本

【架构专题】吃透微服务架构中的业务拆分

架构随笔录

【高可用】系统架构与技术选型

技术二三

