第 5 章

粘包问题及解决方法

在基于 TCP 的端到端通信中，如果一端连续发送两个或者两个以上的数据包，对端在一次接收时，收到的数据包数量可能大于 1 个，也可能是几个完整的数据包加上一个完整包的一部分数据，这些统称为粘包。

5.1 网络通信粘包的表现

为了展示粘包的具体表现，这里以 4.6 节“TCP 回显服务器示例”中的示例为基础，对其有针对性地进行改造。改造时服务器端代码保持不变，只修改客户端，改造后的客户端被命名为 sticky_echo_client，代码如下：

```
//Chapter5/sticky_echo_client/src/demo.cj

from std import socket.*
from std import console.*
from std import sync.*
from std import time.*
from std import random.*

//回显服务器端口
let port: UInt16 = 9990

//回显服务器地址
let echoServerAddress = "127.0.0.1"

//异常退出标志
var quit = false

main() {
    //回显服务器客户端
    let echoClient = TcpSocket(echoServerAddress, port)

    //连接回显服务器
    echoClient.connect()
```

```
    //启动一个线程,用于读取服务器的消息
    spawn {
        try {
            readFromEchoServer(echoClient)
        } catch (exp: SocketException) {
            println("Error reading data from socket:${exp}")
        } catch (exp: Exception) {
            println(exp)
        }
        quit =true
        println("Enter to quit!")
    }

    let rand =Random()

    for (i in 0..100) {
        echoClient.write(i.toString().toArray())
        sleep(Duration.microsecond * rand.nextInt64(100))
    }

    //循环读取用户的输入并发送到回显服务器
    while (true) {
        let readContent =Console.stdIn.readln().getOrThrow()

        //服务器端出现异常,退出程序
        if (quit) {
            return
        }
        echoClient.write(readContent.toArray())
        if (readContent =="quit") {
            return
        }
    }
}

//从 Socket 读取数据并打印输出
func readFromEchoServer(echoSocket: TcpSocket) {
    //存放从 Socket 读取数据的缓冲区
    let buffer =Array<UInt8>(1024, item: 0)

    while (true) {
        //从 Socket 读取数据
        var readCount =echoSocket.read(buffer)

        //把接收的数据转换为字符串
        let content =String.fromUtf8(buffer[0..readCount])

        //输出读取的内容,加上前缀 S:
```

```
            println("S:${content}")

            //如果收到了退出指令就关闭连接
            if (content =="quit") {
                echoSocket.close()
                return
            }
        }
    }
```

可以看到,改造的方式就是在 main 函数里添加了如下代码:

```
    let rand =Random()

    for (i in 0..100) {
        echoClient.write(i.toString().toArray())
        sleep(Duration.microsecond * rand.nextInt64(100))
    }
```

该部分代码会循环 100 次,把 0～99 的数字依次发送给服务器端,只是每次发送时会随机挂起 0～100μs 的时间,在理想情况下,客户端会收到服务器端的 100 次回复,每次回复的都是客户端发送到服务器端的 0～99 的数字,但是,实际情况怎样呢? 为了达到更好的演示效果,客户端运行在本地,服务器端被部署到互联网上的某台服务器上,下面是某次程序运行的结果,客户端的输出如下(Windows 系统):

```
.\main.exe
S:0
S:1234
S:5678910
S:111213
S:1415
S:161718
S:192021
S:222324
S:2526
S:272829
S:303132
S:3334
S:353637
S:383940
S:4142
S:434445
S:464748
S:495051
S:5253
S:545556
S:575859
```

```
S:606162
S:6364
S:65666768
S:6970
S:717273
S:747576
S:777879
S:8081
S:828384
S:858687
S:8889
S:909192
S:939495
S:9697
S:9899
```

对应的服务器端输出如下(Ubuntu 系统)：

```
./main
Echo server started listening on port 9990
New connection accepted, remote address is:223.99.217.219:52023
223.99.217.219:52023:0
223.99.217.219:52023:1234
223.99.217.219:52023:5678910
223.99.217.219:52023:111213
223.99.217.219:52023:1415
223.99.217.219:52023:161718
223.99.217.219:52023:192021
223.99.217.219:52023:222324
223.99.217.219:52023:2526
223.99.217.219:52023:272829
223.99.217.219:52023:303132
223.99.217.219:52023:3334
223.99.217.219:52023:353637
223.99.217.219:52023:383940
223.99.217.219:52023:4142
223.99.217.219:52023:434445
223.99.217.219:52023:464748
223.99.217.219:52023:495051
223.99.217.219:52023:5253
223.99.217.219:52023:545556
223.99.217.219:52023:575859
223.99.217.219:52023:606162
223.99.217.219:52023:6364
223.99.217.219:52023:65666768
223.99.217.219:52023:6970
223.99.217.219:52023:717273
223.99.217.219:52023:747576
223.99.217.219:52023:777879
```

```
223.99.217.219:52023:8081
223.99.217.219:52023:828384
223.99.217.219:52023:858687
223.99.217.219:52023:8889
223.99.217.219:52023:909192
223.99.217.219:52023:939495
223.99.217.219:52023:9697
223.99.217.219:52023:9899
```

虽然客户端将数据发送给服务器端时是按照从0～99的顺序逐次发送的，但是，服务器端在接收的时候，并不是每次都接收一个数字，有的是两个，也有3个和4个的情况，这就是一个典型的通信粘包的表现。

5.2 粘包产生的原因

TCP是一种面向流的数据传输协议，传输的对象是连续的字节流，内容之间并没有明确的分界标志，严格来讲，并不存在粘包的问题，而通常所讲的粘包，更多的是一种逻辑上的概念，也就是人为地把TCP传输的字节流划分成了一个个数据包，发送端确定了数据包之间的边界，但接收端并不能保证按照数据包的边界来接收。例如，要发送两个数据包，第1个数据包10字节，第2个数据包15字节，在发送端是分两次发送的，接收端可能是两次接收，第1次10字节，第2次15字节，也可能是一次性接收25字节，或者第1次接收10字节，第2次接收5字节，第3次接收10字节，每次接收的字节数都不是固定的，详细的接收端数据包分析如图5-1所示。

图5-1 接收端数据包分析

之所以会出现接收端和发送端数据批次不一定匹配的情况，原因比较多，这与网络状态、接收端的运行情况、发送端的网络配置等因素都有关系，具体来讲，大体可以分为以下几类。

1. 发送端启用了 Nagle 算法

对于小包，发送端可能会将其累计起来，到了一定的数据量或者其他条件满足时才发送给接收端，这是导致粘包的一个重要原因，详情见 3.4 节的第 1 部分 NODELAY。

2. TCP 的滑动窗口机制

根据滑动窗口机制，发送端一次发送数据量的多少并不完全是由自己决定的，还要受接收端的缓存大小限制，这也会导致发送端原本计划一次发送的数据包被分为多次发送。滑动窗口的详细解释参考 3.1.4 节。

3. MSS 和 MTU 分片

MSS（Maximum Segment Size）表示 TCP 报文中数据部分的最大长度，MTU（Maximum Transmission Unit）是链路层一次可以发送最大数据的限制，以太网中一般为 1500 字节；如果一次需要发送的数据大于 MSS 或者 MTU，则数据会被拆分成多个包进行传输，这也会导致粘包的产生。

4. 接收端不及时接收

如果接收端不能及时接收缓冲区的数据包，则在其后的某次接收中就会出现接收多个数据包的情况。

5.3　粘包解决方法

通过 5.2 节的分析可知，粘包产生的主要原因是接收端不知道数据包的边界，因为 TCP 协议本身并不包含数据包的边界信息，所以解决粘包问题的主要思路就是给数据包定义边界，常用的解决方案主要有两种，一种是以指定的字符或者字符串作为结束标志，另一种是把数据包分为固定的包首和可变的包体两部分，在包首部分标识包体的大小；其实，这两种方案在本质上都是设计应用层的协议，从应用层入手解析字节流，从而得到正确的数据包。

5.3.1　指定数据包结束标志

指定数据包结束标志是一种比较简单的确定数据包边界的方法，通过一些特殊字符作为分界标志，在接收的数据中如果发现了该分界标志，就认为本包数据结束了。当然，这种方式还要考虑一些特殊情形，例如，如果发送的内容本身包含作为分界标志的特殊字符，就需要对这些字符进行转义处理。

为了演示这种解决方法，下面以 4.6 节中的示例为基础，把回车换行作为数据包分界标志，也就是说客户端每次发送消息时都在消息末尾添加上回车换行符，服务器端解析时，也以回车换行符作为消息结束标志，为了简单起见，这里假设客户端发送的内容不包含回车换行符，并且每次发送的数据大小不超过 1024 字节；带分界标志的回显服务器示例代码如下，

首先是服务器端：

```
//Chapter5/echo_server_with_crlf/src/demo.cj

from std import socket.*
from std import console.*

main() {
    //服务监听端口
    let port: UInt16 = 9990

    let socketAddress = SocketAddress("0.0.0.0", port)

    //回显 TcpSocket 服务器端
    let echoServer = TcpServerSocket(bindAt: socketAddress)

    println("Echo server started listening on port ${port}")

    //启动一个线程,用于监听客户端连接
    spawn {
        //绑定到本地端口
        echoServer.bind()
        while (true) {
            let echoSocket = echoServer.accept()
            println("New connection accepted, remote address
is:${echoSocket.remoteAddress}")

            //启动一个线程,用于处理新的 Socket
            spawn {
                try {
                    dealWithEchoSocket(echoSocket)
                } catch (err: SocketException) {
                    println(err.message)
                }
            }
        }
    }

    //监听控制台输入,如果输入 quit 就退出程序
    while (true) {
        let readContent = Console.stdIn.readln().getOrThrow().trimAscii()

        //如果用户输入 quit 就退出程序
        if (readContent == "quit") {
            return
        }
    }
}
```

```
//从 Socket 读取数据并回写到 Socket
func dealWithEchoSocket(echoSocket: TcpSocket) {
    //存放从 Socket 读取数据的缓冲区
    let buffer =Array<UInt8>(1024, item: 0)

    //已经写入缓冲区的有效字节数
    var readableCount =0

    while (true) {
        //从 Socket 读取数据
        var readCount =echoSocket.read(buffer[readableCount..])

        //如果读取的字节数为 0,则表明对端关闭,直接退出
        if (readCount ==0) {
            return
        }

        //已经读取的字节数
        readableCount =readableCount +readCount

        //查找已经读取的内容里是否有回车换行符
        var pos =getCRLFPos(buffer[0..readableCount])

        //如果查找到了回车换行符,就将内容输出到 Socket,一直循环,直到找不到回车换行
        //符,然后从外层循环再次读取 Socket
        while (let Some(crlfPos) <-pos) {
            //把接收的数据转换为字符串,不包括最后的回车换行
            let content =String.fromUtf8(buffer[0..crlfPos])

            //把 content 写入 echoSocket
            writeToEchoSocketWithCRLF(echoSocket, content)

            //如果接收的内容是 quit 就关闭连接
            if (content =="quit") {
                echoSocket.close()
                return
            }

            //回车换行后未处理的字节数
            let undealByteLen =readableCount -crlfPos -2

            //把未处理的字节复制到缓冲区首部
            buffer.copyTo(buffer, crlfPos +2, 0, undealByteLen)

            //把未处理的字节数作为缓冲区有效字节数
            readableCount =undealByteLen
```

```
            //查找下一个回车换行符
            pos =getCRLFPos(buffer[0..readableCount])
        }
    }
}

//在数组 buf 中查找回车换行符第 1 次出现的位置
public func getCRLFPos(buf: Array<UInt8>) {
    var pos: ?Int64 =None

    //如果 buf 包含的字节数小于 2,则直接返回 None
    if (buf.size <2) {
        return pos
    }

    //从第 1 个可读位置开始
    var indx =0

    //遍历数组查找匹配位置
    while (indx <buf.size -1) {
        //如果从 indx 开始的两个连续字符是回车换行,就认为匹配,将 indx 设置为匹配位置
        //并跳出循环
        if (buf[indx] ==13 && buf[indx +1] ==10) {
            pos =indx
            break
        }

        //如果不匹配就从数组的下一个位置开始判断
        indx++
    }

    return pos
}

//把 content 和回车换行写入 echoSocket
func writeToEchoSocketWithCRLF(echoSocket: TcpSocket, content: String) {
    println("${echoSocket.remoteAddress}:${content}")
    let contentCrlf =content +"\r\n"
    echoSocket.write(contentCrlf.toArray())
}
```

服务器端处理 Socket 数据收发的是函数 dealWithEchoSocket,该函数把从 Socket 读取的数据都存放在缓冲区 buffer 中,然后从 buffer 中循环查找回车换行符,如果查到了,就将内容回写到客户端,否则跳出循环,再次从 Socket 读取数据。

客户端代码如下：

```
//Chapter5/echo_client_with_crlf/src/demo.cj

from std import socket.*
from std import console.*
from std import sync.*
from std import time.*
from std import random.*

//回显服务器端口
let port: UInt16 = 9990

//回显服务器地址
let echoServerAddress = "127.0.0.1"

//异常退出标志
var quit = false

main() {
    //回显服务器客户端
    let echoClient = TcpSocket(echoServerAddress, port)

    //连接回显服务器
    echoClient.connect()

    //启动一个线程,用于读取服务器的消息
    spawn {
        try {
            readFromEchoServer(echoClient)
        } catch (exp: SocketException) {
            println("Error reading data from socket:${exp}")
        } catch (exp: Exception) {
            println(exp)
        }
        quit = true
        println("Enter to quit!")
    }

    let rand = Random()

    for (i in 0..100) {
        writeToEchoSocketWithCRLF(echoClient, i.toString())
        sleep(Duration.microsecond * rand.nextInt64(100))
    }

    //循环读取用户的输入并发送到回显服务器
    while (true) {
        let readContent = Console.stdIn.readln().getOrThrow().trimAscii()
```