# TCP长连接和短连接

### 什么是长连接、短连接

短连接：连接—>传输数据—>关闭；

长连接：连接—>传输数据—>传输数据🡪… 🡪关闭

长连接的连接数不能过多。

### 数据的发送接收方式

异步，对于每个连接，发送和接收数据分别有不同的线程负责

同步，对于每个连接，由同一个线程发送和接收。同步需要考虑超时问题。

单工，数据传输只支持数据在一个方向上传输

半双工，数据传输允许数据在两个方向上传输，但是，在某一时刻，只允许数据在一个方向上传输，它实际上是一种切换方向的单工通信；

全双工，数据通信允许数据同时在两个方向上传输，要求发送方和接收方都有独立的接收和发送能力。

单个Socket接收和发送可以同时进行，但接收不能同时，发送不能同时。

### 封包、粘包或半包、拆包

定义一个包的大小上限(8M)。（防止一个包过大，造成占用大量内存）。

注：UDP包可能丢失，乱序，但不存在粘包问题。TCP不存在丢包，乱序，但存在粘包问题。

粘包：一个包可能实际上分为了多块，或者多个包合成了一个包。

粘包的三种解决方案：

1. 固定长度包
2. 固定长度包头+可变长度包体（包体的长度在包头中定义）（如tlv报文格式 T和L字段长度固定， T字段表示报文类型（4byte）， L字段表示报文长度(16byte)、V字段表示报文的内容）注：T\V建议使用Encoding.UTF8编码，UTF8编码单元为字符，没有端的问题，L字段使用大端编码。 因为报文类型是一定的，如果是未识别的类型，则丢包。
3. 包头包尾标志（包头标志可以没有） （可尝试使用固定长度+包头包尾）

大包问题：传输的数据量较大，需要切割成多块发送。

### 心跳包

心跳包一般是客户端定时通知服务器端我还在。服务器端一般也要定时轮询所有的连接，如果客户端在规定时间内没有响应，则认为服务器断线。

心跳包用于长连接保活和断线重连。心跳包周期一般可设置为16s。

### 连接的关闭

客户端发起关闭，服务器端receive返回0，然后服务器端就可以关闭连接了。

永远不要主动从服务器端关闭。服务器只有在认为客户端已经关闭或者连接中断时，才关闭连接，清理资源占用。

## Keepalive

Tcp中Keepalive是用来检测死链接的，TCP会在空闲了一段时间后发送数据给对方。KeepAliveTime 值控制 TCP/IP 尝试验证空闲连接是否完好的频率。如果这段时间内没有活动，则会发送保持活动信号。如果网络工作正常，而且接收方是活动的，它就会响应。如果没有收到 keep-alive 应答，keep-alive请求 将在每 KeepAliveInterval 秒重发一次。

### 重连

客户端如果连接或者发送到服务器数据出错或者其它认为客户端已经断开的情况，客户端释放资源并重新创建链接。

### 服务器设计

服务器在不能为每个连接分配一个线程。而是构建多个自定义连接池的方式。每个连接池包含一个接收数据线程，一个数据处理线程和一个数据发送线程。每个连接池同时包含两个队列：一个发送队列和一个接受队列。线程池的数目通常设定为factor\*cpu核数+2。Factor是一个>=1.0的常数因子，通常factor=2.0。接收数据线程扫描连接池中所有连接，如果某个连接有数据，则从该链接中获取数据并解析数据，并放到接受队列中。数据处理线程扫描接收队列，并处理接收到的数据，如果要向客户端发送数据，则把要发送的数据放到发送队列。发送线程扫描发送队列将数据发送到客户端。目前，三个线程处理任务的cpu比重是不一样的。比如数据发送线程，仅仅负责发送数据。**在未来，可以使多个连接池共用数据发送线程。**

如果是短连接，可以考虑使用线程池。

（多路IO方案，如果是长连接，并且可以接受单工通信，可以通过BeginReceive实现通信，一个线程接受连接，放到队列中，另一个线程从队列中获取连接，并BeginReceive调用，）。

private void AcceptCallback(IAsyncResult result)

{

ConnectionInfo connection = new ConnectionInfo();

try

{

// Finish Accept

Socket s = (Socket)result.AsyncState;

connection.Socket = s.EndAccept(result);

connection.Buffer = new byte[255];

lock (\_connections) \_connections.Add(connection);

// Start Receive and a new Accept

connection.Socket.BeginReceive(connection.Buffer, 0,

connection.Buffer.Length, SocketFlags.None,

new AsyncCallback(ReceiveCallback), connection);

\_serverSocket.BeginAccept(new AsyncCallback(AcceptCallback),

result.AsyncState);

}

catch (SocketException exc)

{

CloseConnection(connection);

Console.WriteLine("Socket exception: " + exc.SocketErrorCode);

}

catch (Exception exc)

{

CloseConnection(connection);

Console.WriteLine("Exception: " + exc);

}

}

private void ReceiveCallback(IAsyncResult result)

{

ConnectionInfo connection = (ConnectionInfo)result.AsyncState;

try

{

int bytesRead = connection.Socket.EndReceive(result);

if (0 != bytesRead)

{

lock (\_connections)

{

foreach (ConnectionInfo conn in \_connections)

{

if (connection != conn)

{

conn.Socket.Send(connection.Buffer, bytesRead,

SocketFlags.None);

}

}

}

connection.Socket.BeginReceive(connection.Buffer, 0,

connection.Buffer.Length, SocketFlags.None,

new AsyncCallback(ReceiveCallback), connection);

}

else CloseConnection(connection);

}

catch (SocketException exc)

{

CloseConnection(connection);

Console.WriteLine("Socket exception: " + exc.SocketErrorCode);

}

catch (Exception exc)

{

CloseConnection(connection);

Console.WriteLine("Exception: " + exc);

}

}

新连接到来时，新链接选择连接数最小的连接池。从而平衡各个线程的处理负载。这是基于每个连接数据交换的量是接近的合理假设下的。

通过以上架构，可以看出服务器采用tcp双工通信。

每个请求应该明确自己想要的东西，每个响应应该明确自己的回答。请求之间不应该有上下文关联。请求和响应之间也不应该有上下文关联。

在性能要求高的情况下，可采用protobuf或者json进行序列化和反序列化。一般情况下，采用XmlSerializer或者BinaryFormatter来进行序列化或者反序列化。

负载均衡可以采用主从服务器实现。客户端先向服务器请求可用的从服务器。从服务器的选择按照服务器当前连接最少。主服务器不应该或者应该较少的处理业务请求，主服务器的主要任务是协调各个从服务器和将连接分配到从服务器。

### 三次握手、四次握手

### 大端、小端、网络字节序、主机字节序

网络字节序是TCP/IP协议规定好的数据表示格式，与具体的硬件、操作系统无关。网络字节序使用Big endian模式。

主机字节序有具体的cpu决定。

网络通讯中，发送方一般将数据转换为网络字节序发送，接收方在转换为本地字节序解析。

### 负载均衡

Socket负载均衡和Http负载均衡

LVS\ngnix\apache\客户端自己实现选择

如果事先知道会有多少个连接，在服务器进行选择，做主从服务器。主服务器只有一个，分服务器有多个，分服务器向主服务器进行注册，客户端向主服务器请求可用的服务器，然后断开和主服务器的连接，再和所请求到的服务器进行通信。

长连接当连接数较多时，不能一个连接一个线程。

### 分布式缓存

Memched\redis

事务

### Memcached

Memcached是免费、开源、高性能、分布式内存对象缓存系统，它通过缓存对象来减少对数据库的访问。

### 杂项

无状态设计，即每个响应不必预先知道请求的信息。可以独立处理响应。