



网络通信

- 虚幻引擎交互设计师班 -



火星时代教育



01

概 念 分 析

02

多 线 程 应 用

03

T C P / I P 通 信

04

U D P 建 立 通 信

CONTENT

概念分析

- 虚幻引擎交互设计师班 -



火星时代教育

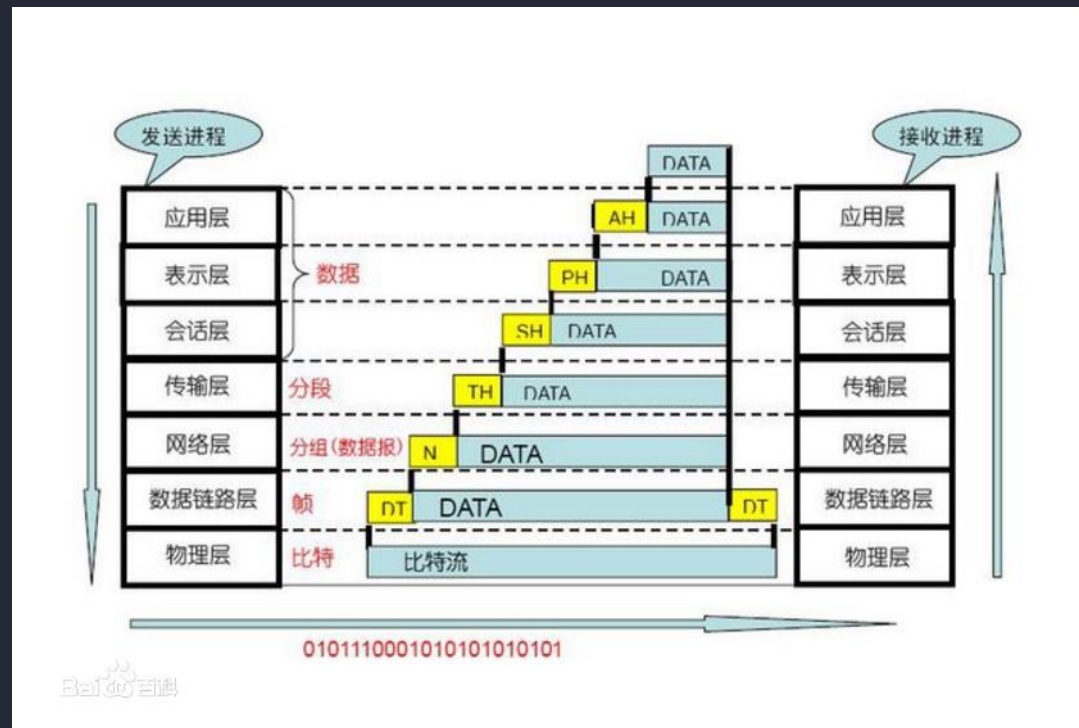
网络通信

网络是用**物理链路**将各个孤立的工作站或主机相连在一起，组成数据**链路**，从而达到**资源共享和通信**的目的。通信是人与人之间通过某种媒体进行的信息交流与传递。网络通信是通过网络将各个孤立的设备进行连接，通过信息交换实现人与人，人与计算机，计算机与计算机之间的通信。

网络通信中最重要的就是网络通信协议。当今网络协议有很多，局域网中最常用的有三个网络协议：MICROSOFT的NETBEUI、NOVELL的IPX/SPX和TCP/IP协议。应根据需要来选择合适的网络协议。

OSI模型

七层模型，亦称OSI（Open System Interconnection）。参考模型是国际标准化组织（ISO）制定的一个用于计算机或通信系统间互联的标准体系，一般称为OSI参考模型或七层模型。它是一个七层的、抽象的模型体，不仅包括一系列抽象的术语或概念，也包括具体的协议。



七层模型

应用层

网络服务与最终用户的一个接口。

协议有：HTTP FTP TFTP SMTP SNMP DNS TELNET HTTPS POP3 DHCP

表示层

数据的表示、安全、压缩。（在五层模型里面已经合并到了应用层）

格式有，JPEG、ASCII、EBCDIC、加密格式等 [2]

会话层

建立、管理、终止会话。（在五层模型里面已经合并到了应用层）

对应主机进程，指本地主机与远程主机正在进行的会话

传输层

定义传输数据的协议端口号，以及流控和差错校验。

协议有：TCP UDP，数据包一旦离开网卡即进入网络传输层

网络层

进行逻辑地址寻址，实现不同网络之间的路径选择。

协议有：ICMP IGMP IP (IPV4 IPV6)

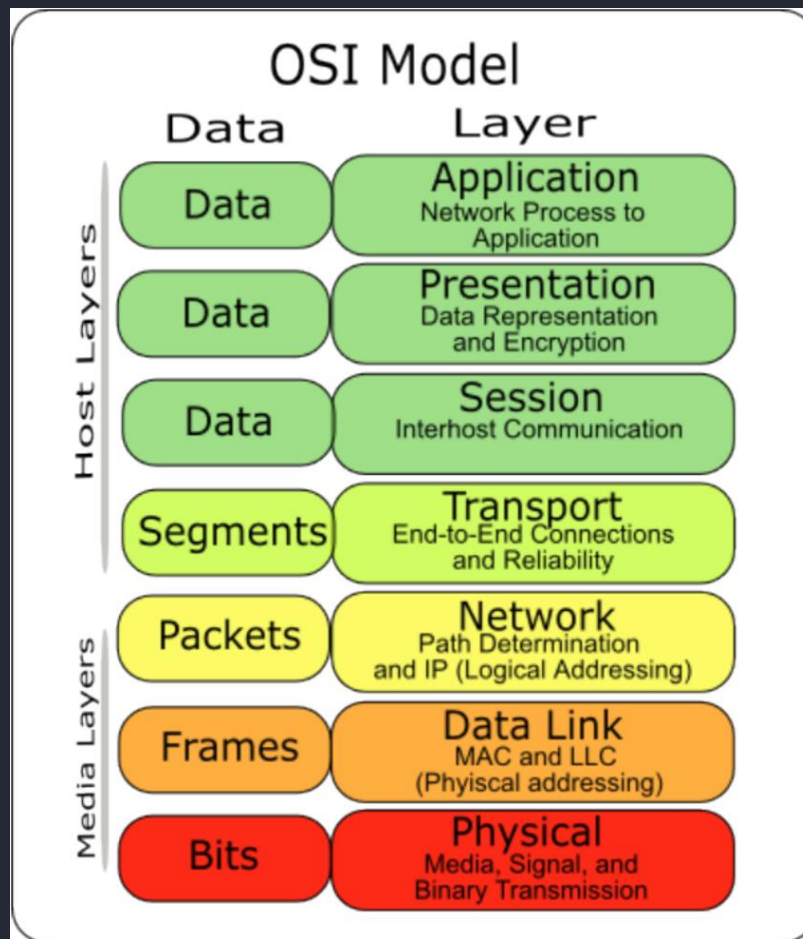
数据链路层

建立逻辑连接、进行硬件地址寻址、差错校验 [3] 等功能。（由底层网络定义协议）

将比特组合成字节进而组合成帧，用MAC地址访问介质，错误发现但不能纠正。

物理层

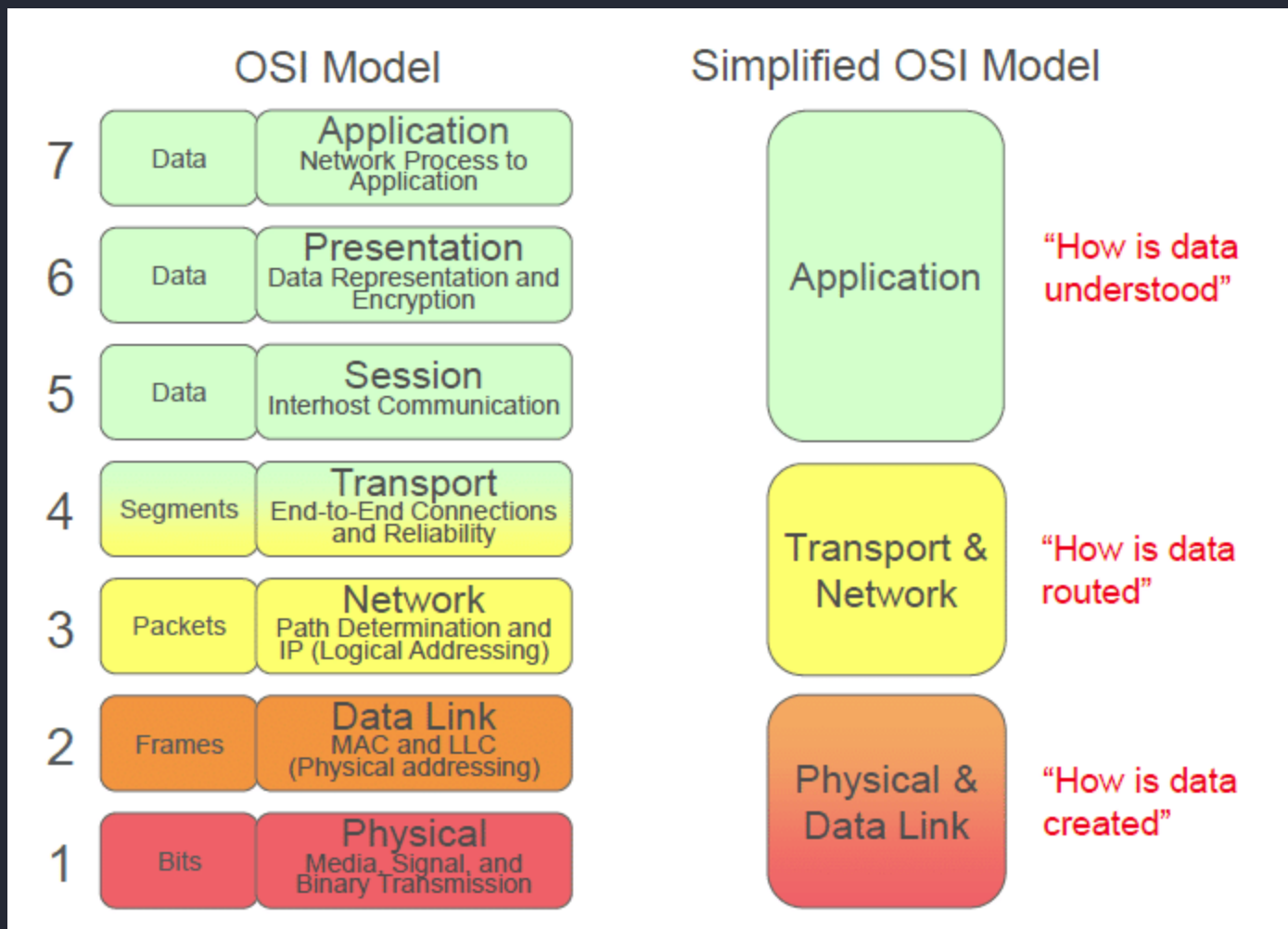
建立、维护、断开物理连接。（由底层网络定义协议）



简单结构

对于OSI七层模型，我们可以简单的理解为

- 1-2如何创建数据
- 3-4如何传输与分发数据
- **5-7如何理解应用数据**



1-1.协议

网络协议

网络协议为计算机网络中**进行数据交换**而建立的**规则**、标准或约定的集合。通信双方对数据传送控制的一种约定。约定中包括对数据格式，同步方式，传送速度，传送步骤，检纠错方式以及控制字符定义等问题做出统一规定，通信双方必须共同遵守，它也叫做链路控制规程。

网络协议三要素

网络协议是由三个要素组成：

- (1) 语义。语义是解释控制信息每个部分的意义。它规定了需要发出何种控制信息，以及完成的动作与做出什么样的响应。
- (2) 语法。语法是用户数据与控制信息的结构与格式，以及数据出现的顺序。
- (3) 时序。时序是对事件发生顺序的详细说明。（也可称为“同步”）。

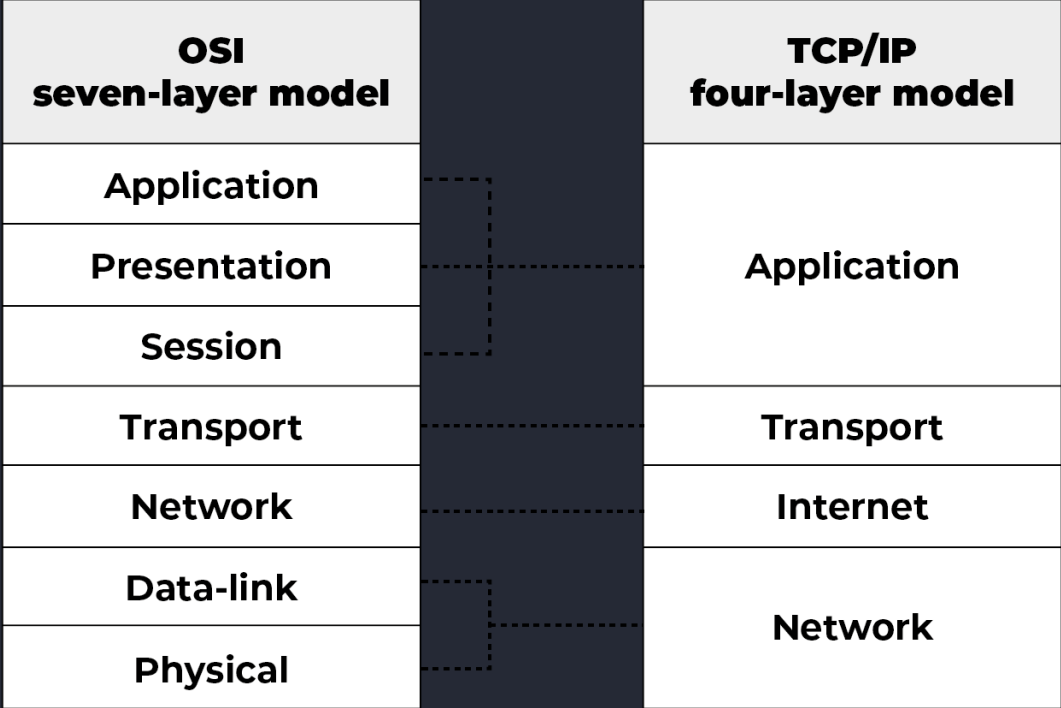
人们形象地把这三个要素描述为：语义表示要做什么，语法表示要怎么做，时序表示做的顺序。

1-2.TPC/IP

TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol, 传输控制协议/网际协议) 是指能够在多个不同网络间实现信息传输的协议簇。TCP/IP协议不仅仅指的是TCP 和IP两个协议, 而是指一个由FTP、SMTP、TCP、UDP、IP等协议构成的**协议簇, 只是因为TCP/IP协议中TCP协议和IP协议最具代表性, 所以被称为TCP/IP协议。**

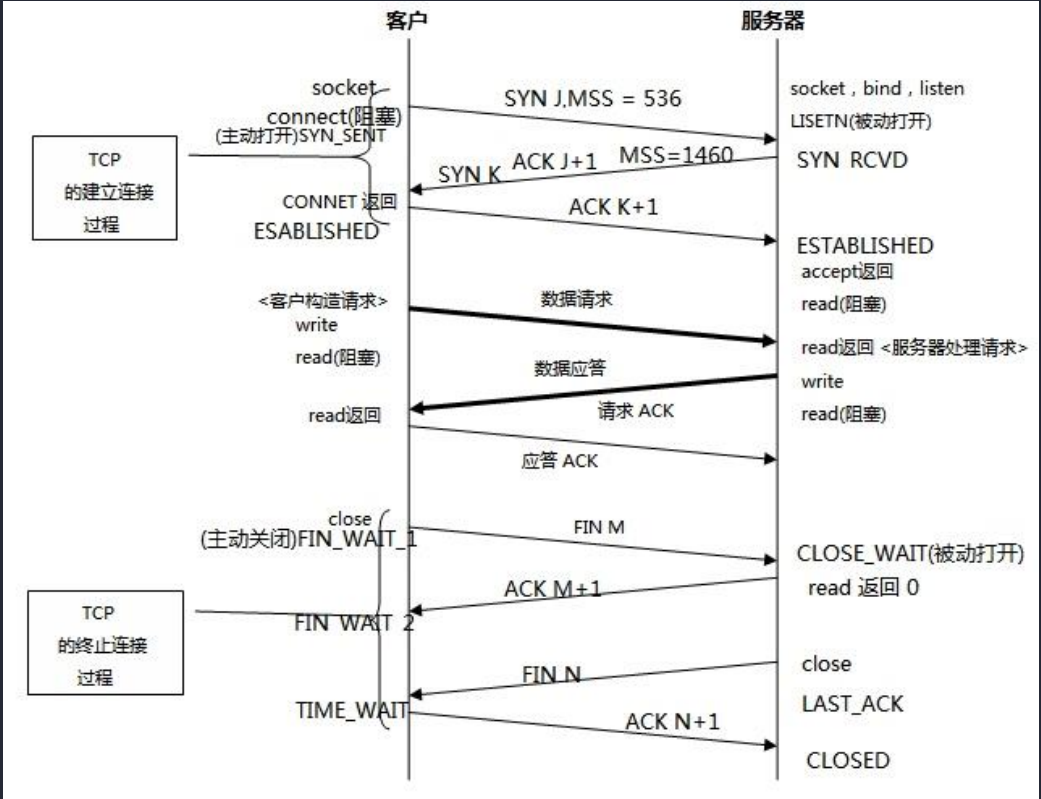
一些典型的 TCP/IP 应用有 FTP、Telnet、SMTP、SNTP、REXEC、TFTP、LPD、SNMP、NFS、INETD 等。RFC 使一些基本相同的 TCP/IP 应用程序实现了标准化, 从而使得不同厂家开发的应用程序可以互相通信



TCP

传输控制协议 (TCP, Transmission Control Protocol) 是一种**面向连接的、可靠的、基于字节流的传输层通信协议**，由IETF的RFC 793 定义。

TCP旨在适应支持多网络应用的分层协议层次结构。 连接到不同但互连的计算机通信网络的主计算机中的成对进程之间依靠TCP提供可靠的通信服务。TCP假设它可以从较低级别的协议获得简单的，可能不可靠的数据报服务。 原则上，TCP应该能够在从硬线连接到分组交换或电路交换网络的各种通信系统之上操作。

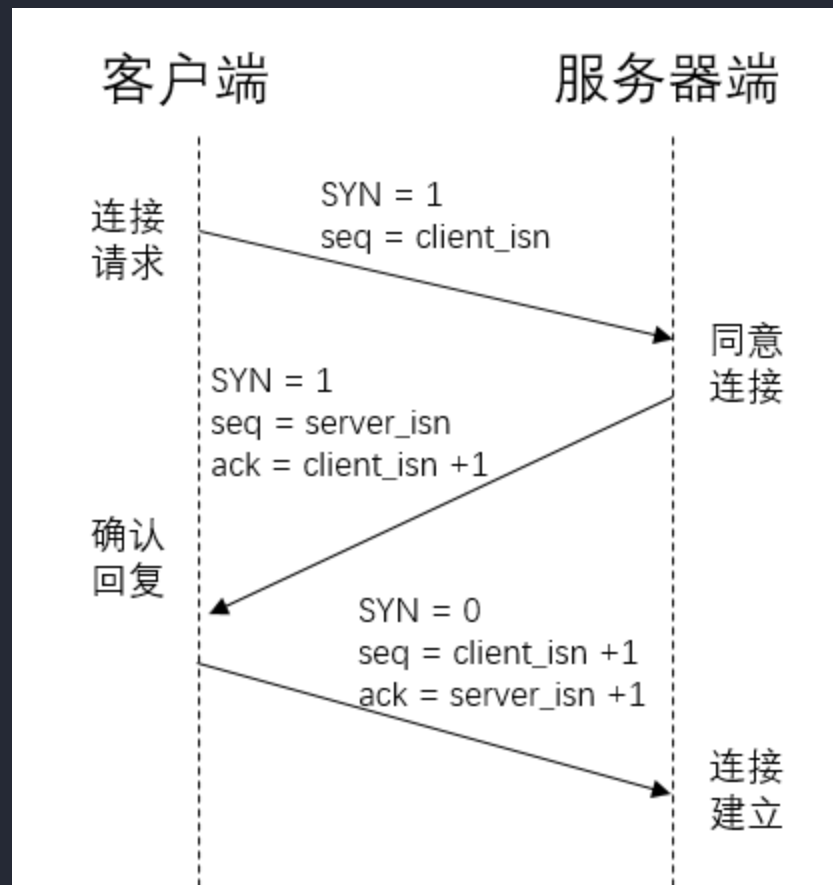


TCP连接

连接需要**三次握手**

1. 客户端发送SYN (TCP第一个数据包非常小, 用于连接) (SEQ=x) 报文给服务器端, 进入SYN_SEND状态。
2. 服务器端收到SYN报文, 回应一个SYN (SEQ=y) ACK (ACK=x+1) 报文, 进入SYN_RECV状态。
3. 客户端收到服务器端的SYN报文, 回应一个ACK (ACK=y+1) 报文, 进入Established状态。

三次握手完成, TCP客户端和服务端成功地建立连接, 可以开始传输数据了。

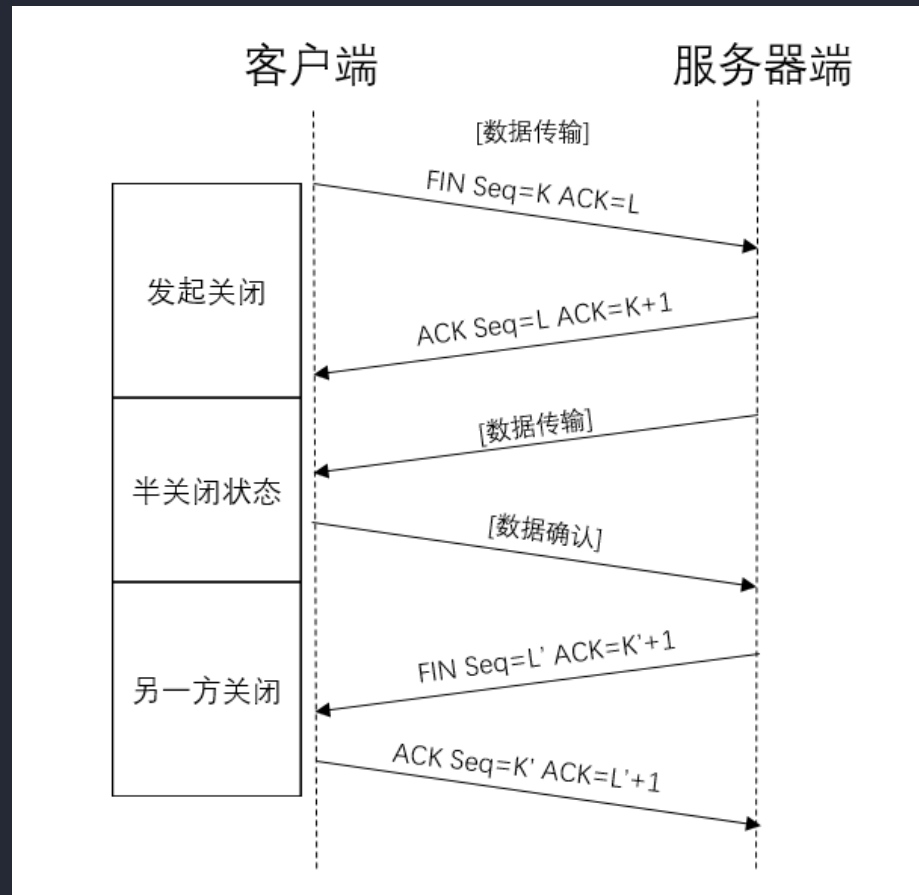


TCP关闭

关闭需要**四次握手**

1. 某个应用进程首先调用close，称该端执行“主动关闭”（active close）。该端的TCP于是发送一个FIN分节，表示数据发送完毕。
2. 接收到这个FIN的对端执行“被动关闭”（passive close），这个FIN由TCP确认。（注意：FIN的接收也作为一个文件结束符（end-of-file）传递给接收端应用进程，放在已排队等候该应用进程接收的任何其他数据之后，因为，FIN的接收意味着接收端应用进程在相应连接上再无额外数据可接收。）
3. 一段时间后，接收到这个文件结束符的应用进程将调用close关闭它的套接字。这导致它的TCP也发送一个FIN。
4. 接收这个最终FIN的原发送端TCP（即执行主动关闭的那一端）确认这个FIN。

既然每个方向都需要一个FIN和一个ACK，因此通常需要4个分节。



IP

IP是Internet Protocol (**网际互连协议**) 的缩写，是TCP/IP体系中的网络层协议。设计IP的目的是提高网络的可扩展性：

1. 解决互联网问题，实现大规模、异构网络的互联互通
2. 分割顶层网络应用和底层网络技术之间的耦合关系，以利于两者的独立发展。

根据端到端的设计原则，IP只为主机提供一种无连接、不可靠的、尽力而为的数据包传输服务。

IP主要包含三方面内容：**IP编址方案、分组封装格式及分组转发规则。**

IP所提供的服务大致可归纳为两类：**IP信息包的传送、IP信息包的分割与重组。**

协议版本

目前我们使用的IP协议主要是IPv4，但IPv4面临诸多问题例如：

- 1. 虽说借助子网化、无类寻址和NAT技术可以提高IP地址使用效率，因特网中IP地址的耗尽仍然是一个没有彻底解决的问题
- 2. IPv4没有提供对实时音频和视频传输这种要求传输最小时延的策略和预留资源支持
- 3. IPv4不能对某些有数据加密和鉴别要求的应用提供支持

为了解决这些问题，IPv6被设计出来了。

IPv4数据包：

版本 (4)	包头长度 (4)	服务类型 (8)	总长度 (16)	
标识 (16)			标志 (3)	片偏移 (13)
生存期 (8)	协议号 (8)		首部校验和 (16)	
源ipv4地址 (32)				
目的ipv4地址 (32)				
选项 (可变长度)			填充 (可变长度)	

IPv6数据包：

版本 (4)	流量类型 (8)	流标签 (20)	
有效荷载长度 (16)		下一个包头 (8)	跳限制 (8)
源ipv6地址 (128)			
目的ipv6地址 (128)			
下一个包头	扩展包头信息		

IPv4 VS IPv6

1. 地址编址方式不同，IPv4用十进制32位长度，中间用点分割，IPv6使用十六进制128位长度，中间用冒号分隔。
2. 路由表大小不同，IPv6路由表更小，通过使用聚类设计，增加了路由转发数据包的速度。
3. IPv6的组播支持流，可以更好的服务于网络传输多媒体
4. IPv6支持自动配置，改进和拓展了DHCP协议，使得网络管理更加方便快捷。
5. IPv6安全性更高。IPv6支持对网络层数据传输加密，并支持对IP报文校验。

IPv4		vs.		IPv6	
Deployed 1981				Deployed 1998	
32-bit IP address				128-bit IP address	
4.3 billion addresses				7.9x10 ²⁸ addresses	
Addresses must be reused and masked				Every device can have a unique address	
Numeric dot-decimal notation				Alphanumeric hexadecimal notation	
192.168.5.18				50b2:6400:0000:0000:6c3a:b17d:0000:10a9 (Simplified - 50b2:6400::6c3a:b17d:0:10a9)	
DHCP or manual configuration				Supports autoconfiguration	

IPv4 Header					IPv6 Header			
Version	IHL	Type of Service	Total Length		Version	Traffic Class	Flow Label	
Identification			Flags	Fragment Offset	Payload Length		Next Header	Hop Limit
Time to Live	Protocol	Header Checksum		Source Address				
Source Address					Source Address			
Destination Address								
Options			Padding		Destination Address			

Legend

Field's Name Kept from IPv4 to IPv6

Fields Not Kept in IPv6

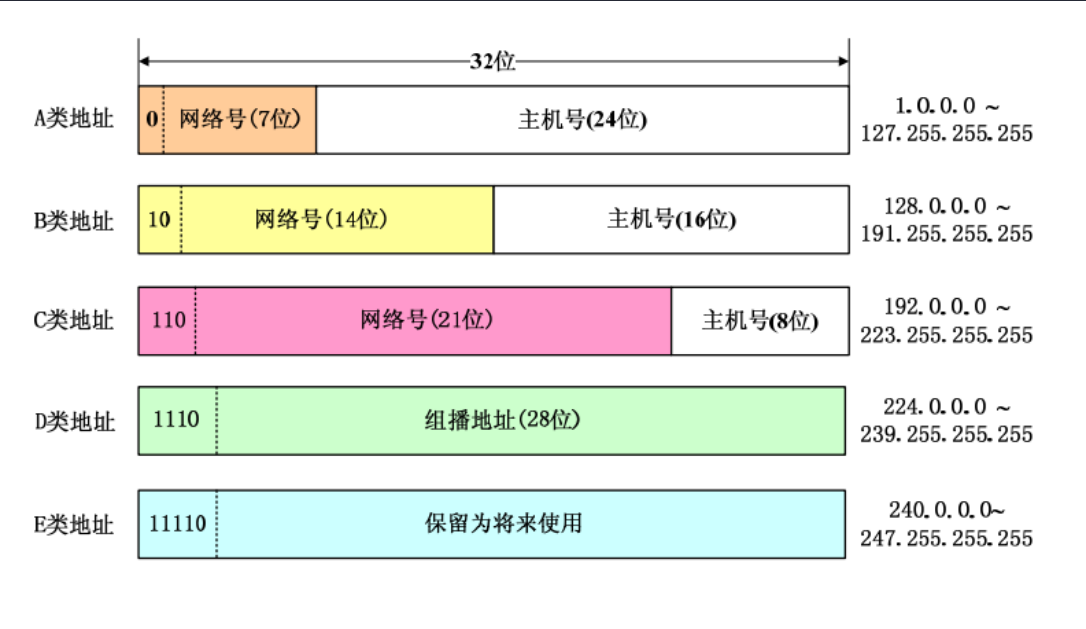
Name and Position Changed in IPv6

New Field in IPv6

IPv4地址

IP地址是用来**识别网络上的设备**，因此，IP地址是由网络地址与主机地址两部分所组成。IP规定网络上的所有设备都必须有**独一无二的IP地址**。每个**IP信息包都必须包含有目的设备的IP地址**，信息包才可以正确地送到目的地。同一设备不可以拥有多个IP地址，所有使用IP的网络设备至少有一个唯一的IP地址。

IP地址被分为五种等级，分别为对应A-E。



特殊IPv4地址

在使用过程中，一些特殊的地址被保留下来，用作特定用途。因此我们在使用IP地址时应注意限制，以免造成不必要的麻烦。一般特殊的IP地址大致分为四种：广播地址，组播地址，0地址，回送地址。

- 广播地址：所有主机号部分为1的地址为广播地址。广播地址分为：直接广播地址和有限广播地址。
- 组播地址：D类IP地址就是组播地址，即在224.0.0.0~239.255.255.255范围内的每个IP地址，实际上代表一组特定的主机。组播地址主要用于电视会议、视频点播等应用。
- 0地址：主机号为0的IP地址从来不分配给任何一个单个的主机号为0，例如，202.112.7.0就是一个典型的C类网络地址，表示该网络本身。
- 回送地址：原本属于A类地址范围内的IP地址127.0.0.0~127.255.255.255却并没有包含在A类地址之内。

多线程

- 虚幻引擎交互设计师班 -



火星时代教育

福特流水线

1913年,福特应用创新理念和反向思维逻辑提出在汽车组装中, 汽车底盘在传送带上以一定速度从一端向另一端前行. 前行中, 逐步装上发动机, 操控系统, 车厢, 方向盘, 仪表, 车灯, 车窗玻璃、车轮, 一辆完整的车组装成了. 第一条流水线使每辆T型汽车的组装时间由原来的12小时28分钟缩短至90分钟, 生产效率提高了8倍!

加速原理

流水线是把一个重复的过程分为若干个子过程, 每个子过程可以和其他子过程并行运作。

进程与线程

进程：是计算机中的程序**关于某数据集合上的一次运行活动**，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。

线程：是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

简单来说，**进程是最小的资源分配单位，线程是最小的运算调度单位（CPU调度的最小单位）**。进程中可以启动多个线程。

多线程

一般在软件设计中，可能要处理多个任务，如果我们采用单一流水线方式进行，那么软件的执行效率会非常低。所以多线程就可以帮助我们解决在同一进程中完成更多的任务。

在多核或多CPU，或支持Hyper-threading的CPU上使用多线程程序设计的好处是显而易见，即提高了程序的执行吞吐率。在单CPU单核的计算机上，使用多线程技术，也可以把进程中负责I/O处理、人机交互而常被阻塞的部分与密集计算的部分分开来执行，编写专门的workhorse线程执行密集计算，从而提高了程序的执行效率。

总的来说，在任务进程中，开启多线程，尤其在多核CPU上可以大幅提升CPU的利用率。对于耗时操作，采用多线程完成，可以提高应用程序的响应速度。多个线程同时活动，彼此重叠进行，可以加快程序的执行效率。

游戏引擎更吃单核？

由于游戏引擎设计的特殊性，在传统的游戏引擎中，一般业务逻辑线程只会被创建为一条（**主要因为游戏中的业务逻辑必须保证运行顺序**），而大部分时间我们都在使用的游戏业务逻辑线程，所以在传统的游戏设计中，游戏引擎更加考验CPU的单核能力。例如最早的7700K CPU，即使过去多年，它依旧是游戏爱好者最喜欢的CPU，主要是因为他的单核睿频极高，算力更强。对于游戏引擎来说，更有优势。但是随着技术的发展，近些年游戏引擎已经开始改变这一设计。游戏引擎也更加倾向多核多线程的工作，大大提升硬件的利用率。

2-1. 虚幻引擎线程

FRunnable

FRunnable是虚幻引擎提供的线程接口类，该类中定义了线程的通知接口，例如初始化，运行，停止，退出等。如希望获取线程逻辑入口，则需要继承此类。

线程接口是响应线程执行逻辑的关键。

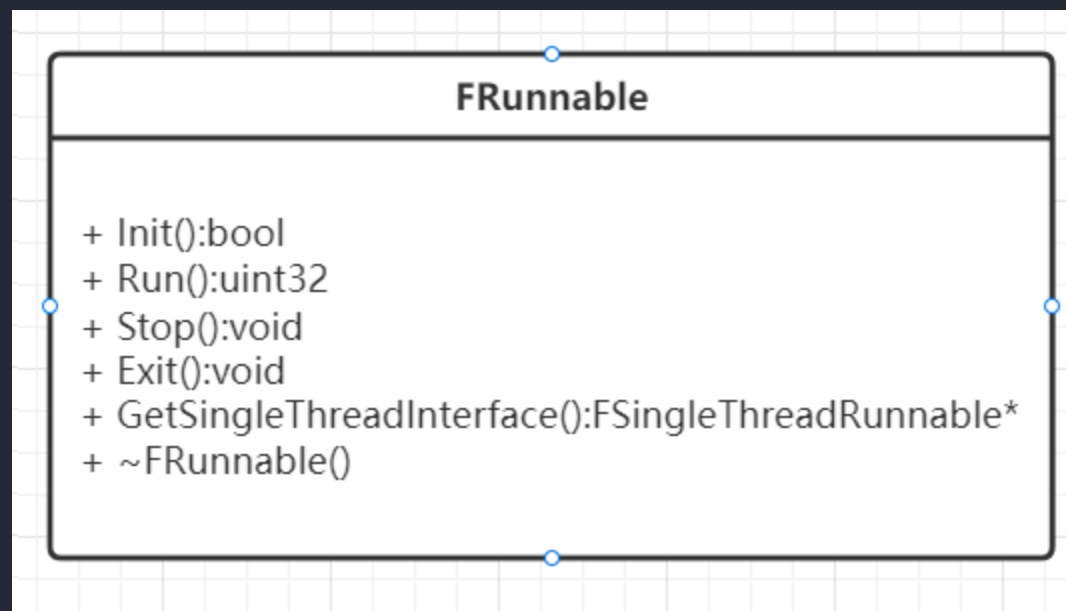
注意：禁止将U类继承此接口

Init函数，当线程被创建成功后被调用

Run函数，当线程执行时被回调

Stop函数，当线程对象调用Kill函数时被调用

Exit函数，当线程执行结束时，被调用



实例代码

实现线程接口类，用于接收线程事件动作，例如初始化，运行，停止等。对于继承线程接口类来说，实现Run函数是必须的。因为对应线程执行的逻辑需要写在Run函数中。

```
#pragma once
#include "CoreMinimal.h"

/**
 *
 */
class UEANT_API FThreadTest : public FRunnable
{
protected:
    //用于接收线程执行逻辑
    virtual uint32 Run() override;
    //用于接收线程停止逻辑
    virtual void Stop() override;
};
```

启动线程

当实现线程接口类后，我们需要使用接口**实例对象**创建**线程对象**，通过调用FRunnableThread中的Create函数完成，参照下图。

```
//创建继承线程接口类的类实例
MyThreadTest = new FThreadTest;
//创建线程对象
MyThread = FRunnableThread::Create(MyThreadTest, ThreadName:TEXT("MyThreadTest"));|
```

此函数将返回线程实例对象（ FRunnableThread 类型指针），如需要后期操作此线程，请存储线程指针。

声明类型

```
protected:
    FThreadTest* MyThreadTest;|
    FRunnableThread* MyThread;
```

关闭线程（执行关闭）

当线程启动时，线程会调用Run函数，来开始执行逻辑。如果Run函数执行结束，则线程完成执行，即被关闭。注意，**当线程关闭后才可以被释放，切勿提前释放。**

注意：**释放线程指针时，顺序应该按照先释放线程指针，再释放Runnable对象指针。切勿颠倒顺序。**

关闭线程 (Kill)

通过调用函数Kill来主动通知线程回调对象，关闭执行逻辑，从而释放线程。（这个过程需要设计）

```
/**
 * Tells the thread to exit. If the caller needs to know when the thread has exited, it should use the bShouldWait value.
 * It's highly recommended not to kill the thread without waiting for it.
 * Having a thread forcibly destroyed can cause leaks and deadlocks.
 *
 * The kill method is calling Stop() on the runnable to kill the thread gracefully.
 *
 * @param bShouldWait  If true, the call will wait infinitely for the thread to exit.
 * @return Always true
 */
virtual bool kill( bool bShouldWait = true ) = 0;
```

bShouldWait标记表明，如果你希望等待线程结束，可以输入true，那么线程结束后，调用kill函数的线程才能继续执行。如果输入false，则不需要等待线程结束，kill调用后直接继续执行下面的逻辑代码。

当Kill函数被调用时，Runnable接口中的Stop函数被回调来接收关闭通知，通过Stop函数修改阻塞逻辑，完成释放动作。

例如：Run函数中添加while循环，条件变量可以在Stop中修改，来结束while的阻塞逻辑。

线程休眠

在执行阻塞逻辑时（例如监听端口消息接收），我们尽量不要长时间让线程占用CPU资源，应在单次逻辑执行完毕后，执行休眠逻辑，休眠可通过调用一下函数完成。单位秒。

```
FPlatformProcess::Sleep(1);
```


2-1. 线程锁

资源共享

在同一进程中的各个线程，都可以共享该进程所拥有的资源，这首先表现在：所有线程都具有相同的地址空间（进程的地址空间），这意味着，线程可以访问该地址空间的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。由于同一个进程内的线程共享内存和文件，所以线程之间互相通信不必调用内核。

线程同步

在同一个进程中的数据资源是共享的，我们经常会遇到一个问题，即多个线程会同时访问相同的数据资源。对于数据资源来说，如果都是读操作，则无任何问题，但是**当读和写同时操作时或是写和写同时操作**，可能会出现一些意外情况。例如一块数据，本身值是0，线程A在读取过程中语句1中是0，但是当执行到语句2时，由于线程B修改了数据，则导致线程A中的原有执行逻辑出错，这是多线程中常见的设计问题。

为了解决这一问题，我们对线程增加了**同步设计**。同步设计是一种机制，它可以有效的保证多个线程之间访问竞争资源的安全性。

最简单的线程同步机制就是增加线程锁。

线程锁

即，在操作过程，线程A对某段数据资源增加锁定标记，在锁定期间，只有线程A能够对数据进行操作，其他线程想要操作数据资源时需要等待。当线程A将资源状态改为非锁定状态，其他的线程才可以操作资源。操作方式依旧是先加锁标记。我们称这样锁为互斥锁。互斥锁保证了每次只有一个线程对数据资源进行写入操作，从而达到多线程下数据资源访问的安全性。

死锁：是指两个或两个以上的**进程**在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

FScopeLock和FCriticalSection

FScopeLock是虚幻引擎中用于实现范围（域）锁定的类，通过使用FScopeLock可以解决多线程之间线程同步问题。

FCriticalSection是用来配合范围锁完成锁定动作，可以被认为是特定锁定标记类。使用语法如下。

```
//一般我们将critical section创建为成员变量
FCriticalSection m_LockSection;
//挡在操作字中需要加锁时，我们创建scope lock，借助锁标记完成锁定
{//域A
    FScopeLock ScopeLock(&m_LockSection);//注意，此锁定在ScopeLock生命结束时会被释放
}
//如果在其他操作域中，需要做线程等待时，我们可以通过如下代码完成
{//域B
    //注意，如果域A访问的线程没有结束访问，则执行域B逻辑则进入等待，只有当域A结束时，m_LockSection被标记释放，域B才可以执行
    FScopeLock ScopeLock(&m_LockSection);//注意，此锁定在ScopeLock生命结束时会被释放
}
```

网络通信

- 虚幻引擎交互设计师班 -



火星时代教育

HTTP通信

HTTP通信

超文本传输协议（Hyper Text Transfer Protocol, HTTP）是一个**简单的请求-响应协议**，它通常运行在**TCP协议**之上。

它指定了客户端可能发送给服务器什么样的消息以及得到什么样的响应。**请求和响应消息的头以ASCII形式给出**；而消息

内容则具有一个类似MIME的格式。这个简单模型是早期Web成功的有功之臣，因为它使开发和部署非常地直截了当。

HTTP方法（动作）

Http在1.1协议版本中定义了8中方法，来应对客户端和服务端不同的信息交换方式，分别为：GET、HEAD、POST、PUT、DELETE、TRACE、OPTIONS、CONNECT。常用的请求方法是GET和POST。

GET: 向指定的资源发出“显示”请求。使用GET方法应该只用在读取资料，而不应当被用于产生“副作用”的操作中，例如在网络应用程序中。其中一个原因是GET可能会被网络爬虫等随意访问。参见安全方法。浏览器直接发出的GET只能由一个url触发。GET上要在url之外带一些参数就只能依靠url上附带querystring。

POST: 向指定资源提交数据，请求服务器进行处理（例如提交表单或者上传文件）。数据被包含在请求本文中。这个请求可能会创建新的资源或修改现有资源，或二者皆有。每次提交，表单的数据被浏览器用编码到HTTP请求的body里。

HTTPS

超文本传输安全协议（英语：HyperText Transfer Protocol Secure，缩写：HTTPS；常称为HTTP over TLS、HTTP over SSL或HTTP Secure）是一种通过计算机网络进行安全通信的传输协议。HTTPS经由HTTP进行通信，但利用SSL/TLS来加密数据包。HTTPS开发的主要目的，是提供对网站服务器的身份认证，保护交换资料的隐私与完整性。这个协议由网景公司（Netscape）在1994年首次提出，随后扩展到互联网上。

HTTPS工作原理

- ① 客户端将它所支持的算法列表和一个用作产生密钥的随机数发送给服务器
- ② 服务器从算法列表中选择一种加密算法，并将它和一份包含服务器公用密钥的证书发送给客户端；该证书还包含了用于认证目的的服务器标识，服务器同时还提供了一个用作产生密钥的随机数
- ③ 客户端对服务器的证书进行验证（有关验证证书，可以参考数字签名），并抽取服务器的公用密钥；然后，再产生一个称作 `pre_master_secret` 的随机密码串，并使用服务器的公用密钥对其进行加密（参考非对称加 / 解密），并将加密后的信息发送给服务器
- ④ 客户端与服务端根据 `pre_master_secret` 以及客户端与服务器的随机数值独立计算出加密和 MAC 密钥（参考 DH 密钥交换算法）
- ⑤ 客户端将所有握手消息的 MAC 值发送给服务器
- ⑥ 服务器将所有握手消息的 MAC 值发送给客户端

HTTPS优缺点

优点

- 使用 HTTPS 协议可认证用户和服务器，确保数据发送到正确的客户机和服务器
- HTTPS 协议是由 SSL+HTTP构建的可进行加密传输、身份认证的网络协议，要比 HTTP安全，可防止数据在传输过程中被窃取、改变，确保数据的完整性。
- HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

缺点

- 相同网络环境下，HTTPS 协议会使页面的加载时间延长近 50%，增加 10%到 20%的耗电。此外，HTTPS 协议还会影响缓存，增加数据开销和功耗。
- HTTPS 协议的安全是有范围的，在黑客攻击、拒绝服务攻击和服务器劫持等方面几乎起不到什么作用。
- 最关键的是，SSL 证书的信用链体系并不安全。特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。
- 成本增加。部署 HTTPS 后，因为 HTTPS 协议的工作要增加额外的计算资源消耗，例如 SSL 协议加密算法和 SSL 交互次数将占用一定的计算资源和服务器成本。在大规模用户访问应用的场景下，服务器需要频繁地做加密和解密操作，几乎每一个字节都需要做加解密，这就产生了服务器成本。随着云计算技术的发展，数据中心部署的服务器使用成本在规模增加后逐步下降，相对于用户访问的安全提升，其投入成本已经下降到可接受程度

虚拟端口（网络端口）

如果我们形容IP，IP就像是指向某个小区的地址，而端口号就像是小区中的某个单元某个门牌号。

端口是设备与外界通讯交流的出口。端口可分为**虚拟端口**和**物理端口**，其中**虚拟端口**指计算机内部或交换机路由器内的端口，不可见。例如计算机中的80端口、21端口、23端口等。

物理端口又称为接口，是可见端口，计算机背板的RJ45网口，交换机路由器集线器等RJ45端口。电话使用RJ11插口也属于物理端口的范畴。

虚幻HTTP通信

步骤

- 引入模块 “HTTP”
- 创建请求对象HttpRequest
- 设置请求方式 (GET or POST)
- 设置请求路径 (URL)
- 设置Header (消息头) 用于标记请求的解释方式。
- 如果是POST, 可以设置携带数据
- 绑定请求通知回调
- 执行请求

由于虚幻引擎封装HTTP请求本身是异步的, 我们不需要考虑请求阻塞问题

代码参考

```
//创建HTTP请求 请将请求对象创建为成员变量保证其生命周期
TSharedPtr<IHttpRequest, ESPMode::ThreadSafe> HttpRequest = FHttpModule::Get().CreateRequest();
//设置请求方法
HttpRequest->SetVerb(TEXT("GET"));
HttpRequest->SetVerb(TEXT("POST"));
//设置访问地址
HttpRequest->SetURL(TEXT("http://www.baidu.com"));
//设置请求头
HttpRequest->SetHeader(HeaderName:TEXT("User-Agent"), HeaderValue:TEXT("X-UnrealEngine-Agent"));
HttpRequest->SetHeader(HeaderName:TEXT("Content-Type"), HeaderValue:TEXT("application/json"));
//如果是POST可以携带数据
// HttpRequest->SetContentAsString()
// HttpRequest->SetContentFromStream()
//绑定回调函数
HttpRequest->OnProcessRequestComplete().BindLambda(InFuncor: [])(FHttpRequestPtr /*Request*/, FHttpResponsePtr /*Response*/, bool) ->void {});
//执行请求
HttpRequest->ProcessRequest();
```


SOCKET通信

Socket

套接字，是对网络中不同主机上的应用进程之间进行**双向通信的端点的抽象**。一个套接字就是网络上进程通信的一端，提供了应用层进程利用网络协议交换数据的机制。从所处的地位来讲，套接字上联应用进程，下联网络协议栈，是应用程序通过网络协议进行通信的接口，是应用程序与网络协议栈进行交互的接口。

在网络通信中，至少需要保证有一对套接字，其中一个运行于客户端，另一个运行于服务器端。

连接分为三个步骤：

1. 服务器监听
2. 客户端请求
3. 连接确认

套接字类型

1.流套接字(SOCK_STREAM)

流套接字用于提供面向连接、可靠的数据传输服务。该服务将保证数据能够实现无差错、无重复送，并按顺序接收。流套接字之所以能够实现可靠的数据服务，原因在于其使用了传输控制协议，即TCP(The Transmission Control Protocol)协议。

2.数据报套接字(SOCK_DGRAM)

数据报套接字提供一种无连接的服务。该服务并不能保证数据传输的可靠性,数据有可能在传输过程中丢失或出现数据重复，且无法保证顺序地接收到数据。数据报套接字使用UDP(User DatagramProtocol)协议进行数据的传输。由于数据报套接字不能保证数据传输的可靠性，对于有可能出现的数据丢失情况，需要在程序中做相应的处理。

3.原始套接字(SOCK_RAW)

原始套接字与标准套接字(标准套接字指的是前面介绍的流套接字和数据报套接字)的区别在于：原始套接字可以读写内核没有处理的IP数据包，而流套接字只能读取TCP协议的数据，数据报套接字只能读取UDP协议的数据。因此，如果要访问其他协议发送的数据必须使用原始套接。

套接字连接服务类型

套接字的连接类型可以分为**面向连接服务**和**无连接服务**：

面向连接服务的主要特点如下：

- (1)数据传输过程必须经过建立连接、维护连接和释放连接3个阶段；
- (2)在传输过程中，各分组不需要携带目的主机的地址；
- (3)可靠性好，但由于协议复杂，通信效率不高。

面向无连接服务的主要特点如下：

- (1)不需要连接的各个阶段；
- (2)每个分组都携带完整的目的主机地址，在系统中独立传送；
- (3)由于没有顺序控制，所以接收方的分组可能出现乱序、重复和丢失现象；
- (4)通信效率高，但可靠性不能确保

面向连接服务

由于面向连接服务需要保持连接，并且需要监听等待对方端消息回传，故**需要启动线程用于阻塞等待**。

TCP协议就是一种面向连接服务，电话系统也是面向连接服务。面向连接时需要通信双方在通信时建立一条通信线路，操作动作分为建立连接，使用连接，释放连接。

数据在传输过程中，不需要携带目标端地址，主要因为数据传输时，双端已经建立连接通道。

优缺点

优点：通信实时性高，信息报文可靠，信息带有回复确认。

缺点：占用通信信道，协议复杂，发送前创建连接开销大

面向无连接服务

面向无连接服务是指不要求发送方和接收方先建立连接。双方可以不构建通信线路，而是在发送数据时携带目标端地址，将数据报文打包发送到线路上。

UDP协议就是一种面向无连接服务，邮政系统也是面向无连接的服务。

优缺点：

优点：信息发送效率高，发送无需等待只需要投递即可

缺点：不保证信息可达，可能丢失，信息无法回复确认

虚幻引擎TCP/IP通信

操作步骤

1. 引入Socket模块
2. 创建Socket对象
3. 构建连接IP
4. 创建连接
5. 启动监听线程，等待消息回复
6. 使用完成后关闭连接

创建连接

以下是请求连接的代码

```
//创建套接字 设置套接字类型为流套接字 套接字名称
FSocket* NativeSocket = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->CreateSocket(SocketType: NAME_Stream, SocketDescription: TEXT("MySocket"));
//创建连接地址
TSharedRef<FIInternetAddr> Address = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->CreateInternetAddr();
//设置连接IP
bool bParse = false;
//尝试解析文本IP 如果成功bParse为真
Address->SetIp(InAddr: TEXT("127.0.0.1"), [&bParse]);
//设置远端端口号
Address->SetPort(8090);
//请求发起连接
NativeSocket->Connect(*Address);
```

发送消息

当建立连接后，就可以通过套接字完成消息发送，发送参照以下代码段截图。

```
bool FNsSocketAnt::Send(const uint8* Data, int32 Count, int32& BytesSend)
{
    if (!NativeSocket)
    {
        NS_LOG(Warning, TEXT("未建立有效链接！请先确保链接正常再使用！"));
        return false;
    }
    return NativeSocket->Send(Data, Count, BytesSent: [&] BytesSend);
}
```

接收消息

接收消息代码，注意接收消息时必须保证启动单独线程，消息接收会导致逻辑阻塞。

```
uint32 FNsSocketZone::Run()
{
    uint8* Buffer;
    int32 BufferSize = 1024;
    Buffer = new uint8[BufferSize];
    int32 RecSize = 0;

    while (bRunning)
    {
        SocketAnt->NativeSocket->Recv(Buffer, BufferSize, BytesRead: [&] RecSize);
        if (RecSize > 0)
        {
            FNsSocketBuffer* BufferData = new FNsSocketBuffer;
            BufferData->Copy(Buffer, RecSize);
            OnReceive.ExecuteIfBound(BufferData);
        }
        else
        {
            NS_LOG(Warning, TEXT("本地或远端主机断开连接! "));
            OnReceive.ExecuteIfBound(nullptr);
            StopRec();
        }
        FPlatformProcess::Sleep(Seconds:0.01f);
    }
    delete[] Buffer;
    return 0;
}
```

销毁套接字

在断开连接后，需要保证套接字被安全的释放，释放代码参照如下。

```
//关闭套接字，注意关闭后接收消息会收到长度为0的数据，需要处理线程关闭
NativeSocket->Close();
//清理套接字
ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->DestroySocket(NativeSocket);
//讲套接字指针指向空
NativeSocket = nullptr;
```

虚幻引擎UDP/IP通信

操作步骤

1. 引入Socket模块 (UDP本身也是套接字操作, 只是无连接服务套接字)
2. 创建Socket对象
3. 进行UDP基础设置
4. 构建接收地址
5. 发送数据

发送数据

UDP消息发送不需要创建连接

```
//创建UDP套接字
FSocket* UdpSocket = FUdpSocketBuilder(InDescription: TEXT("MyUdp")).AsReusable().WithBroadcast();
//设置缓冲 注意缓冲大小决定发送数据的大小, 禁止无限设置缓冲大小
int32 NewSize = 0; //用来记录设置完成的缓存大小
if (!UdpSocket->SetSendBufferSize(Size: 1024, [&] NewSize))
{
    //处理错误逻辑, 不要继续执行
}
//创建地址
TSharedRef<FInternetAddr> Addr = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->CreateInternetAddr();
bool bParse = false;
//设置IP地址
Addr->SetIp(InAddr: TEXT("127.0.0.1"), [&] bParse);
//检查设置IP地址是否正确
if (!bParse)
{
    //处理错误逻辑不要继续执行
}
//设置端口号
Addr->SetPort(8090);
uint8* Data = nullptr; //假定数据
int32 SendCount = 50; //发送数量
int32 BytesSend = 0; //引用传入, 内部告知外部多少数据被发送
UdpSocket->SendTo(Data, SendCount, BytesSent: [&] BytesSend, *Addr);
```

接收数据

UDP客户端数据接收比较特殊，由于客户端不一定有公网IP，**故无法像服务器一样监听某个端口用来接收数据**。与Http和面向连接的Socket最大区别就是UDP接收消息是通过内网穿透完成。内网穿透只为某个远端开放，其他均不可以。所以如果在客户端UDP希望接收消息，我们需要通过Recv和RecvFrom函数均可完成。区别在于：

RecvFrom函数需要指定接收哪个终端地址返回的消息。例如，你用套接字将数据发送到127.0.0.1的8060端口，那么From函数在接收时需要将接收地址设置未127.0.0.1，并且端口指定8060。**此函数需使用发送消息时的套接字**。

Recv函数无需指定接收目标，**使用发送消息的套接字用于接收**，使用方法如面向连接套接字一样。

注意：当启动轮询后，**服务器即可主动给客户端发送消息**，前提是：**客户端必须先给服务器发送过消息**。

是否阻塞？

在虚幻UDP接收中，可以设置是否阻塞接收消息，如果不阻塞接收消息，则在调用Recv函数时没有返回消息，则会直接继续执行，并将接收Size设置为0。如果设置阻塞接收，则会一定等待接收消息，直至有数据被接收到为止。开启阻塞接收只需要创建套接字时，调用AsBlocking函数即可。

```
NativeSocket = FUdpSocketBuilder(InDescription: TEXT("MyNativeUdp")).AsReusable().WithBroadcast().AsBlocking();
```

接收示例代码如下：

```
NativeSocket->Recv(Buffer, BufferSize, BytesRead: [&]ByteRecSize);  
//如果是阻塞接收消息，则只有当收到消息才会执行以下代码，如果是非阻塞，则调用后直接向下执行  
//只是ByteRecSize的值为0  
//判定是否接收到消息  
if (ByteRecSize > 0)  
{  
    UTF8ToTCHAR UTT(reinterpret_cast<const ANSICHAR*>(Buffer), ByteRecSize);  
    UE_LOG(LogTemp, Log, TEXT("接收到消息: %s"), UTT.Get());  
}  
else//如果是阻塞接收，执行此处说明套接字关闭了，如果是非阻塞接收，则表明未收到消息  
{  
    UE_LOG(LogTemp, Log, TEXT("进入休眠"));  
    //说明开放的接收端口没有收到消息  
    FPlatformProcess::Sleep(Seconds:0.5f);  
}
```

接收数据 (Recv) 非阻塞

```
uint8* Buffer = new uint8[1024];
int32 BufferSize = 1024;
int32 ByteSize = 0;

while (bRun)
{
    //此套接字为发送消息时使用的套接字
    NativeSocket->Recv(Buffer, BufferSize, BytesRead: [&] ByteSize);
    if (ByteSize > 0)
    {
        UTF8ToTCHAR UTT(reinterpret_cast<const ANSICHAR*>(Buffer), ByteSize);
        FString msg(Src: UTT.Get(), ExtraSlack: UTT.Length());
        UE_LOG(LogTemp, Log, TEXT("收到服务器UDP消息: %s"), *msg);
    }
    FPlatformProcess::Sleep(Seconds: 0.5f);
}
delete[] Buffer;
```


接收数据 (RecvFrom)

```
uint8* Buffer = new uint8[1024];
int32 BufferSize = 1024;
int32 ByteSize = 0;
//创建接收IP地址, 与发送目标地址相同
TSharedRef<FInternetAddr> Addr = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->CreateInternetAddr();
bool bVaild = true;
Addr->SetIp(InAddr:TEXT("127.0.0.1"), [&bVaild]);
if (!bVaild)
{
    return 0;
}
//设置端口号
Addr->SetPort(8050);
while (bRun)
{
    //此套接字为发送消息套接字, 也可创建新套接字
    NativeSocket->RecvFrom(Buffer, BufferSize, BytesRead: [&] ByteSize, [&]*Addr);
    if (ByteSize > 0)//当返回数据为0时不终止线程
    {
        FUTF8ToTCHAR UTT(reinterpret_cast<const ANSICHAR*>(Buffer), ByteSize);
        FString msg(Src:UTT.Get(), ExtraSlack:UTT.Length());
        UE_LOG(LogTemp, Log, TEXT("收到服务器UDP消息: %s"), *msg);
    }
    FPlatformProcess::Sleep(Seconds:0.5f);
}
delete[] Buffer;
```

销毁套接字

销毁方法与TCP应用套接字销毁一致，具体可以参照之前的代码完成。



Mars游戏星球 



游戏学院公众号

(教程 + 科普 + 就业 + 优秀作品 + 行业资讯)



游戏学院视频号

(教程 + 优秀作品 + 行业资讯 + 学院风采)



花瓣网账号

(优秀作品 + 知识点长图)

Thanks

-火星时代游戏设计学院-