Cocos2d-x 游戏开发教程

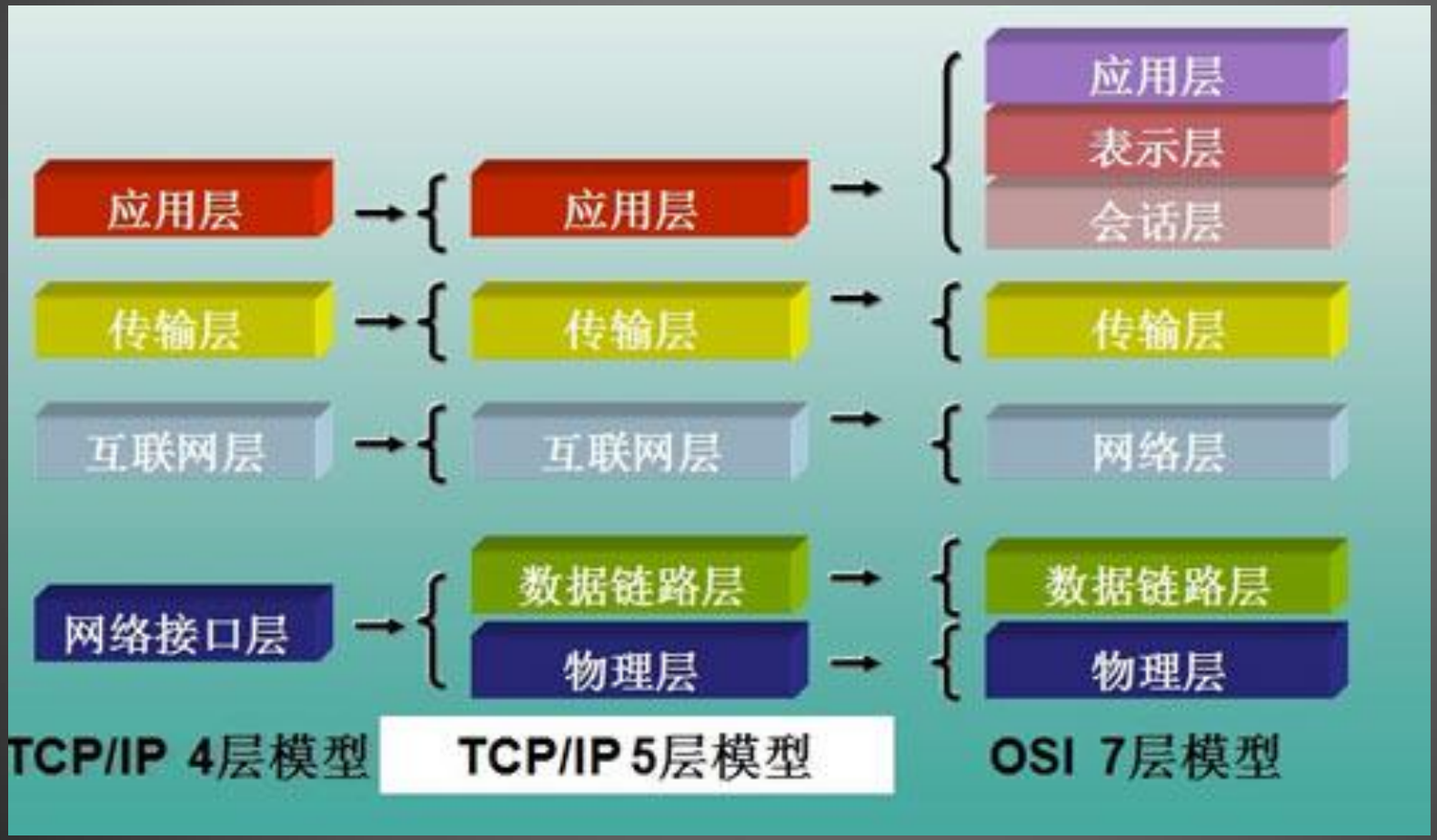
网络与常用算法

目 录 / contents

网络基本概念

网络模型：

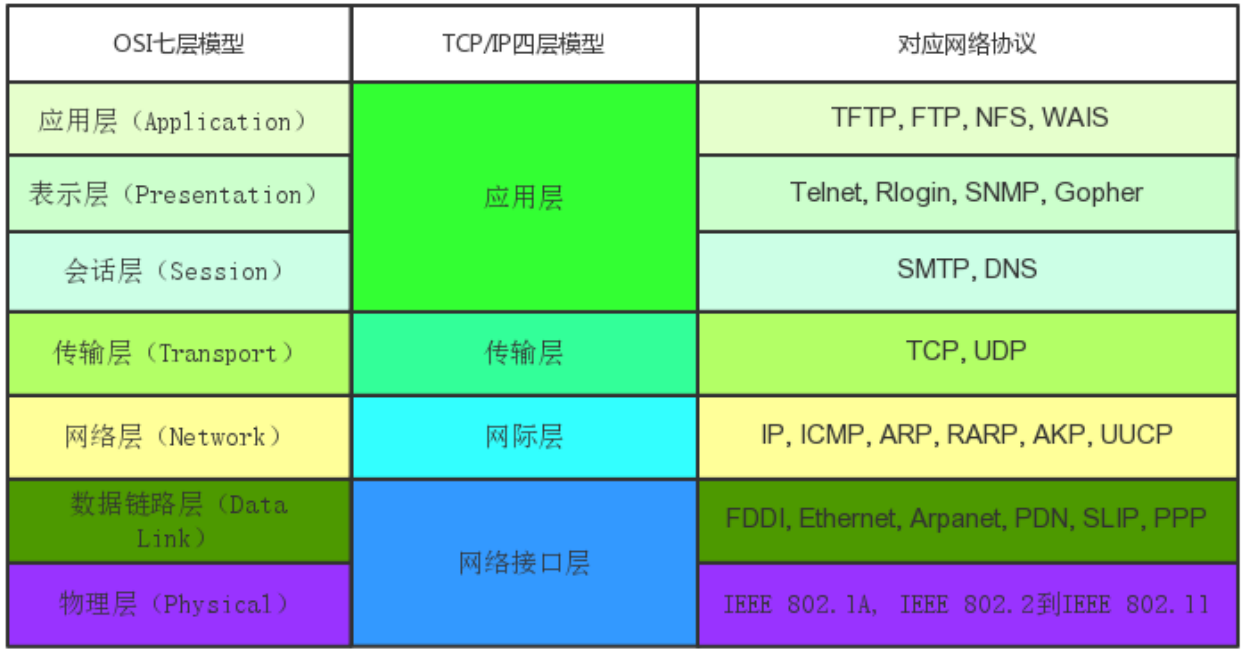
OSI层模型、TCP/IP的层模型如下所示：



网络基本概念

网络模型：

TCP/IP各层对应的协议如下所示：



OSI七层模型	TCP/IP四层模型	对应网络协议
应用层（Application）	应用层	TFTP, FTP, NFS, WAIS
表示层（Presentation）		Telnet, Rlogin, SNMP, Gopher
会话层（Session）		SMTP, DNS
传输层（Transport）	传输层	TCP, UDP
网络层（Network）	网际层	IP, ICMP, ARP, RARP, AKP, UUCP
数据链路层（Data Link）	网络接口层	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP
物理层（Physical）		IEEE 802.1A, IEEE 802.2到IEEE 802.11

IP协议：对应于网络层，是网络层的协议，

TCP协议：对应于传输层，是传输层的协议，主要解决数据如何在网络中传输。

HTTP协议：对应于应用层，是应用层的协议，主要解决如何包装数据。

Socket：本身不是协议，而是对TCP/IP协议的封装和应用的调用接口API，通过它我们才能使用TCP/IP协议。

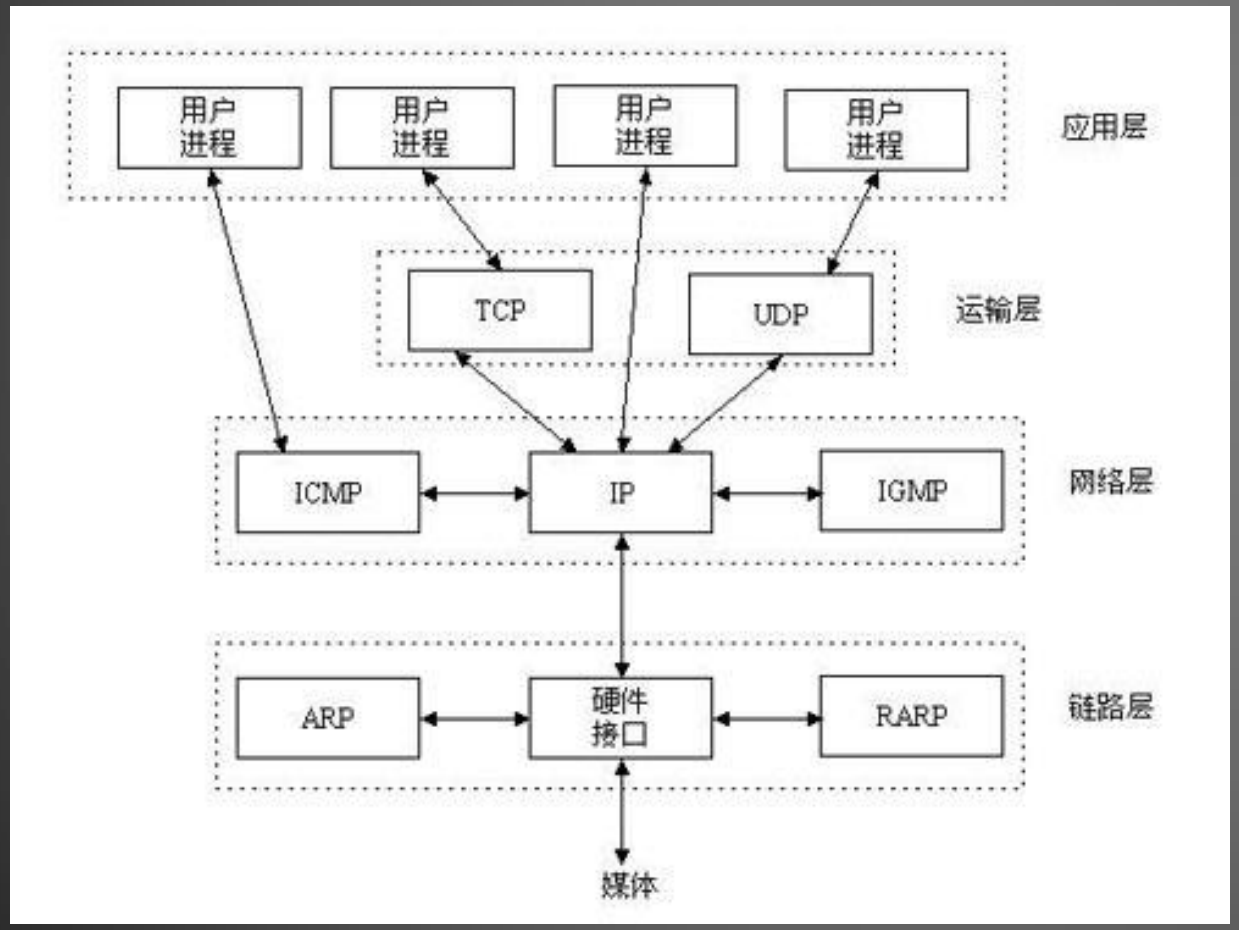
网络基本概念

TCP/IP协议：

- TCP/IP指传输控制协议/网际协议 (Transmission Control Protocol / Internet Protocol)。
- TCP/IP是供已连接因特网的计算机进行通信的通信协议。
- TCP/IP定义了电子设备（比如计算机）如何连入因特网，以及数据如何在它们之间传输的标准。

网络基本概念

协议的关系：



网络基本概念

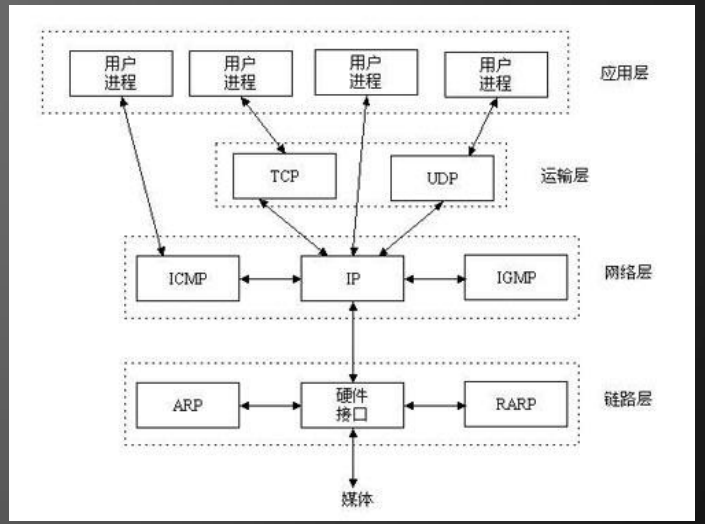
HTTP协议和TCP/IP协议的关系:

“我们在传输数据时，可以只使用（传输层）TCP/IP协议，但是那样的话，如果没有应用层，便无法识别数据内容。

如果想要使传输的数据有意义，则必须使用到应用层协议。

应用层协议有很多，比如HTTP、FTP、TELNET等，也可以自己定义应用层协议。

WEB使用HTTP协议作应用层协议，以封装HTTP文本信息，然后使用TCP/IP做传输层协议将它发到网络上。”

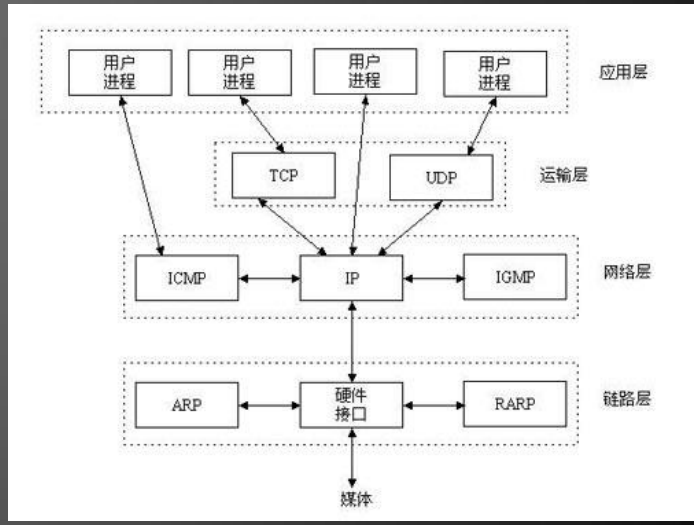


网络基本概念

Socket与TCP/IP协议的关系:

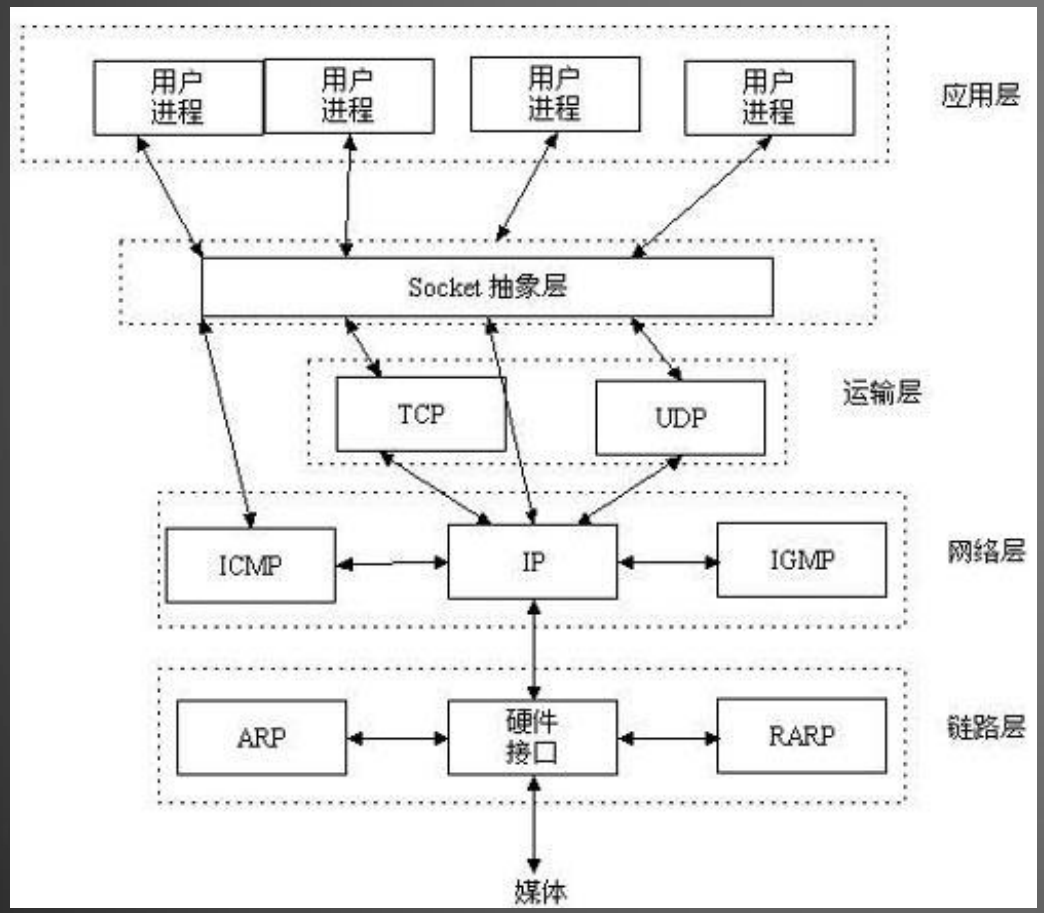
网络有一段关于socket和TCP/IP协议关系的说法比较容易理解：

“TCP/IP只是一个协议栈，就像操作系统的运行机制一样，必须要具体实现，同时还要提供对外的操作接口。这个就像操作系统会提供标准的编程接口，比如win32编程接口一样，TCP/IP也要提供可供程序员做网络开发所用的接口，这就是Socket编程接口。”



网络基本概念

Socket在哪里？



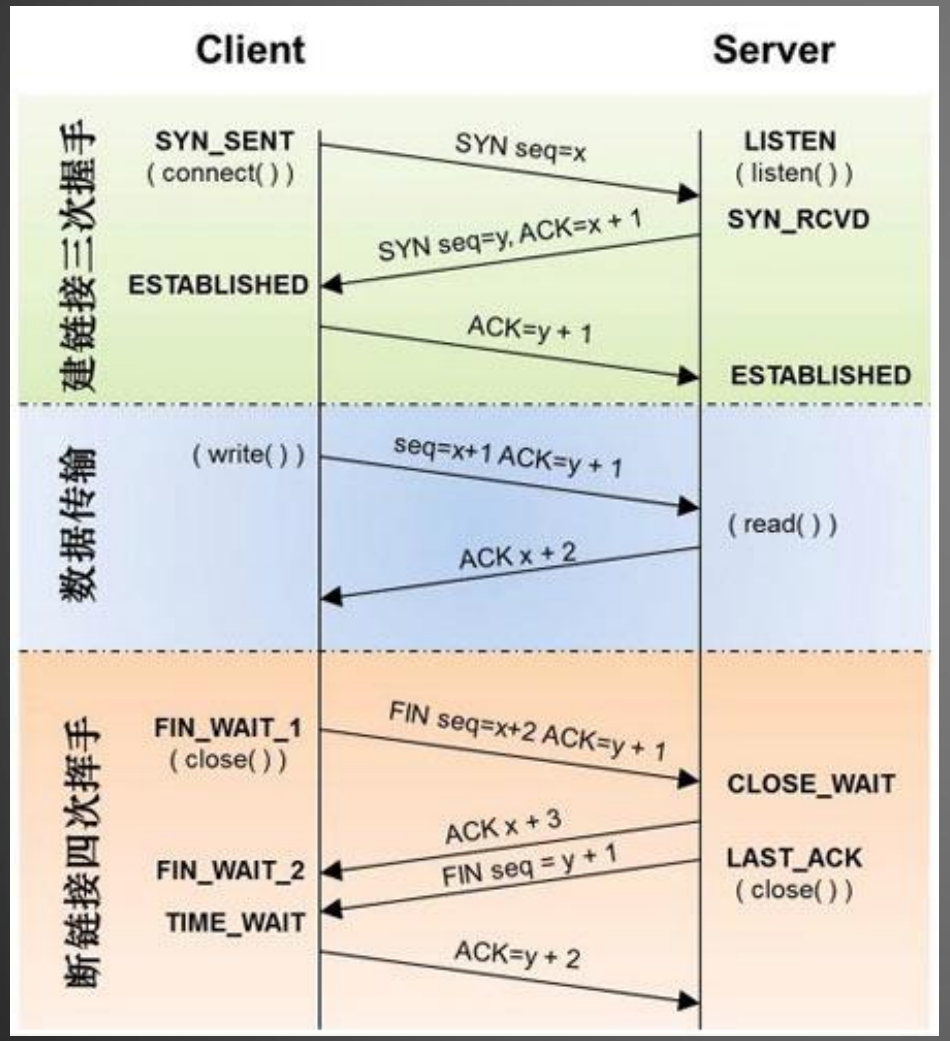
CSDN上有个比较形象的描述：HTTP是轿车，提供了封装或者显示数据的具体形式；Socket是发动机，提供了网络通信的能力。

实际上，传输层的TCP是基于网络层的IP协议的，而应用层的HTTP协议又是基于传输层的TCP协议的，而Socket本身不算是协议，就像上面所说，它只是提供了一个针对TCP或者UDP编程的接口

COCOS2DX

网络基本概念

TCP连接：



建立起一个TCP连接需要经过“三次握手”：

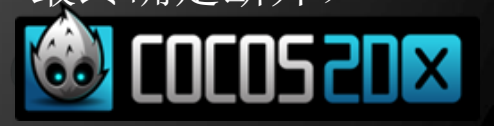
（1）第一次握手：客户端发送syn包（syn=j）到服务器，并进入SYN_SEND状态，等待服务器确认。

（2）第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN_RECV状态。

（3）第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK（ack=k+1），此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

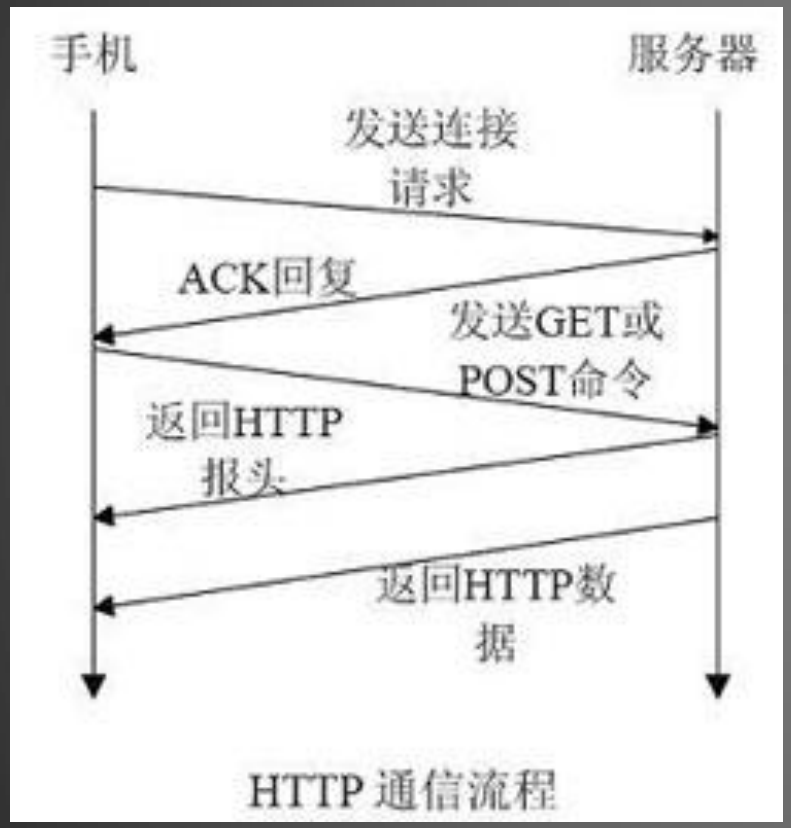
注意：握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。

（4）断开连接时：服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次挥手”（过程就不细写了，就是服务器和客户端交互，最终确定断开）



网络基本概念

HTTP连接：



HTTP 通信流程

HTTP连接使用的是“请求—响应”的方式。最显著的特点是客户端发送的每次请求都需要服务器回送响应，在请求结束后，会主动释放连接。从建立连接到关闭连接的过程称为“一次连接”。

（1）在HTTP 1.0中，客户端的每次请求都要求建立一次单独的连接，在处理完本次请求后，就自动释放连接。

（2）在HTTP 1.1中则可以在一次连接中处理多个请求，并且多个请求可以重叠进行，不需要等待一个请求结束后再发送下一个请求。

由于HTTP在每次请求结束后都会主动释放连接，因此HTTP连接是一种“短连接”。

网络基本概念

Socket原理：

- 套接字（socket）概念

套接字（socket）是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

套接字：{ IP地址 : 端口号 }

应用层通过传输层进行数据通信时，TCP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要通过同一个TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了套接字(Socket)接口。应用层可以和传输层通过Socket接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

网络基本概念

Socket原理：

- 建立socket连接

建立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 。
套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

（a）服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

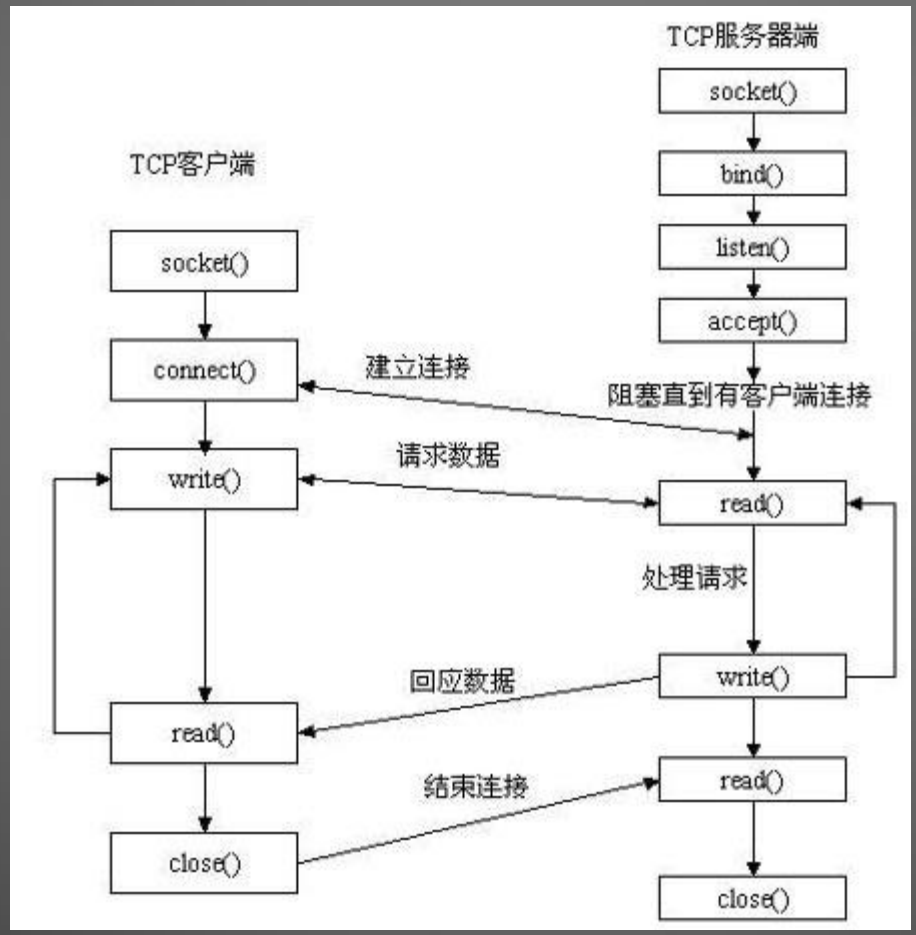
（b）客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

（c）连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

网络基本概念

Socket原理：

- 建立socket连接



网络基本概念

Socket连接与TCP连接：

创建Socket连接时，可以指定使用的传输层协议，Socket可以支持不同的传输层协议（TCP或UDP），当使用TCP协议进行连接时，该Socket连接就是一个TCP连接

Socket连接与HTTP连接：

由于通常情况下Socket连接就是TCP连接，因此Socket连接一旦建立，通信双方即可开始相互发送数据内容，直到双方连接断开。但在实际网络应用中，客户端到服务器之间的通信往往需要穿越多个中间节点，例如路由器、网关、防火墙等，大部分防火墙默认会关闭长时间处于非活跃状态的连接而导致 Socket 连接断连，因此需要通过轮询告诉网络，该连接处于活跃状态。

而HTTP连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是Socket连接，服务器就可以直接将数据传送给客户端；若双方建立的是HTTP连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端。因此，客户端定时向服务器端发送连接请求，不仅可以保持在线，同时也是在“询问”服务器是否有新的数据，如果有就将数据传给客户端。

[如何使用socket：Cocos2d-x网络篇03：Socket连接]

网络基本概念

Cookie机制

Cookies是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器。网络服务器用HTTP头向客户端发送cookies，在客户终端，浏览器解析这些cookies并将它们保存为一个本地文件，它会自动将同一服务器的任何请求缚上这些cookies 。

具体来说cookie机制采用的是在客户端保持状态的方案。它是在用户端的会话状态的存贮机制，他需要用户打开客户端的cookie支持。cookie的作用就是为了解决HTTP协议无状态的缺陷所作的努力。

正统的cookie分发是通过扩展HTTP协议来实现的，服务器通过在HTTP的响应头中加上一行特殊的指示以提示浏览器按照指示生成相应的cookie。然而纯粹的客户端脚本如JavaScript也可以生成cookie。而cookie的使用是由浏览器按照一定的原则在后台自动发送给服务器的。浏览器检查所有存储的cookie，如果某个cookie所声明的作用范围大于等于将要请求的资源所在的位置，则把该cookie附在请求资源的HTTP请求头上发送给服务器。

网络基本概念

Cookie机制

cookie的内容主要包括：名字，值，过期时间，路径和域。路径与域一起构成cookie的作用范围。若不设置过期时间，则表示这个cookie的生命期为浏览器会话期间，关闭浏览器窗口，cookie就消失。这种生命期为浏览器会话期的cookie被称为会话cookie。会话cookie一般不存储在硬盘上而是保存在内存里，当然这种行为并不是规范规定的。若设置了过期时间，浏览器就会把cookie保存到硬盘上，关闭后再次打开浏览器，这些cookie仍然有效直到超过设定的过期时间。存储在硬盘上的cookie可以在不同的浏览器进程间共享，比如两个IE窗口。而对于保存在内存里的cookie，不同的浏览器有不同的处理方式。

如何使用 HttpClient

HttpClient是HTTP客户端的接口。

HttpClient封装了各种对象，处理cookies，身份认证，连接管理等。

如何使用 HttpClient

HttpClient的使用一般包含下面7个步骤：

1. 引入头文件和命名空间
2. 创建 HttpRequest 的实例；
3. 设置某种连接方法的类型（GET、POST等），这里通过setUrl传入待连接的地址；
4. 设置响应回调函数，读取response；
5. 添加请求到HttpClient任务队列；
6. 释放连接。无论执行方法是否成功，都必须释放连接；
7. 对得到后的内容进行处理。

COCOS2DX

如何使用 HttpClient

1. 引入头文件和命名空间

```
#include "network/HttpClient.h“
using namespace cocos2d::network;
```

2. 使用HttpRequest无参数的构造函数构建实例

```
HttpRequest* request = new HttpRequest();
```

如何使用 HttpClient

3. 设置连接方法的类型和待连接的地址

```
request->setRequestType(HttpRequest::Type::GET);
request->setUrl("http://www.httpbin.org/get");
```

4. 设置回调

```
Request->setResponseCallback(
        CC_CALLBACK_2(HelloWorld::onHttpComplete,this)
);
```

如何使用 HttpClient

5. 添加请求到HttpClient任务队列

```
cocos2d::network::HttpClient::getInstance()->send(request);
```

6. 释放连接

```
request->release();
```

如何使用 HttpClient

7. 处理网络回调函数

```
void HelloWorld::onHttpRequestCompleted(HttpClient *sender, HttpResponse * response) {
    if (!response) {
        return;
    }
    if (!response->isSucceed())
    {
        log("response failed");
        log("error buffer: %s", response->getErrorBuffer());
        return;
    }
    std::vector<char> *buffer = response->getResponseData();
    printf("Http Test, dump data: ");
    for (unsigned int i = 0; i < buffer->size(); i++)
    {
        printf("%c", (*buffer)[i]);
    }
    printf("\n");
}
```

如何使用 HttpClient

GET请求示例：

```
HttpRequest* request = new HttpRequest();
request->setUrl("http://www.baidu.com");
request->setRequestType(HttpRequest::Type::GET);
request->setResponseCallback(CC_CALLBACK_2(HelloWorld::onHttpRequestCompleted, this));
request->setTag("GET test");
cocos2d::network::HttpClient::getInstance()->send(request);
request->release();
```

如何使用 HttpClient

POST请求示例：

```
HttpRequest* request = new HttpRequest();
request->setUrl("http://httpbin.org/post");
request->setRequestType(HttpRequest::Type::POST);
request->setResponseCallback(CC_CALLBACK_2(HelloWorld::onHttpRequestCompleted, this));

// write the post data
const char* postData = "visitor=cocos2d&TestSuite=Extensions Test/NetworkTest";
request->setRequestData(postData, strlen(postData));
request->setTag("POST test");
cocos2d::network::HttpClient::getInstance()->send(request);
request->release();
```

HttpClient session

session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。

当程序需要为某个客户端的请求创建一个session的时候，服务器首先检查这个客户端的请求里是否已包含了一个session标识 - 称为session id，如果已包含一个session id则说明以前已经为此客户端创建过session，服务器就按照session id把这个session检索出来使用（如果检索不到，可能会新建一个），如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个session id将被在本次响应中返回给客户端保存。

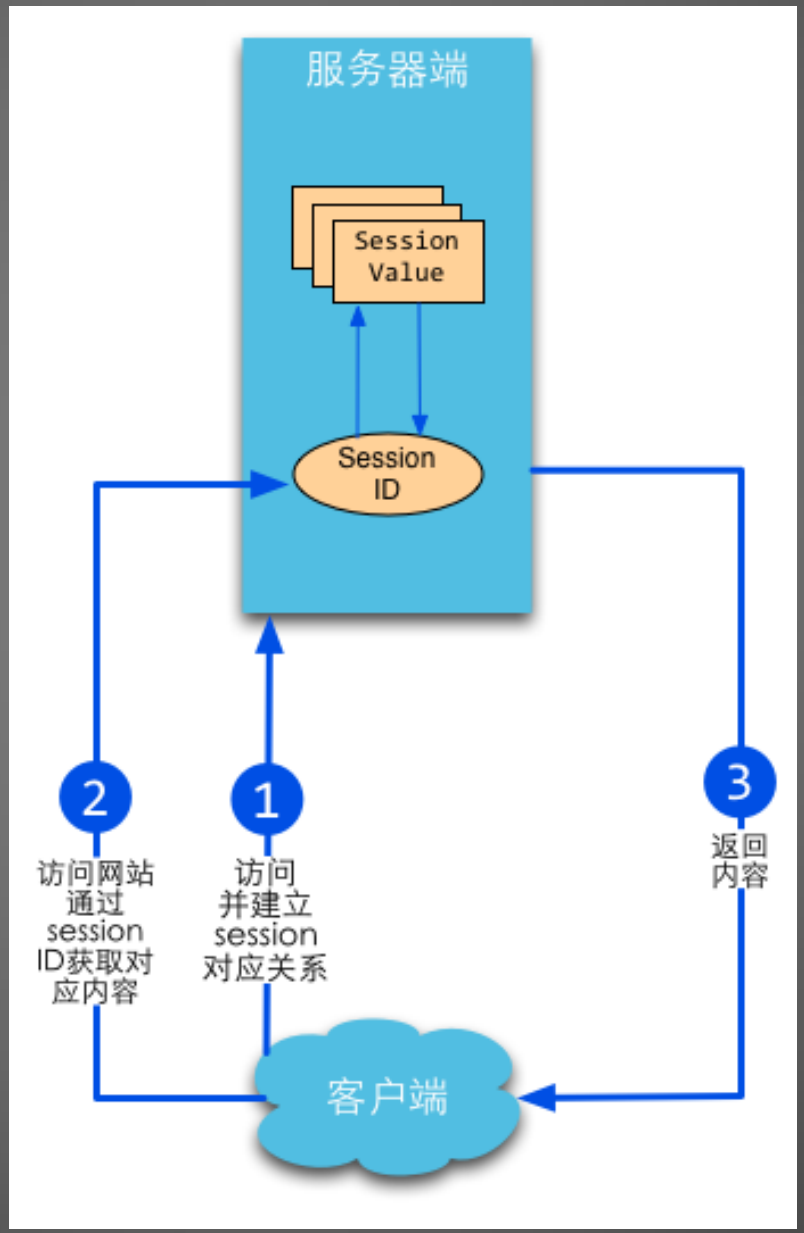
HttpClient session

保存这个session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发回给服务器。一般这个cookie的名字都是类似于SEEESIONID，而。比如weblogic对于web应用程序生成的cookie，JSESSIONID=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764，它的名字就是JSESSIONID。

session，简而言之就是在服务器上保存用户操作的历史信息。服务器使用session id来标识session，session id由服务器负责产生，保证随机性与唯一性，相当于一个随机密钥，避免在握手或传输中暴露用户真实密码。但该方式下，仍然需要将发送请求的客户端与session进行对应，所以可以借助cookie机制来获取客户端的标识（即session id），也可以通过GET方式将id提交给服务器。

HttpClient session

session的原理图：



HttpClient session

POST请求示例：

```
HttpRequest* request = new HttpRequest();
request->setUrl("http://httpbin.org/post");
request->setRequestType(HttpRequest::Type::POST);
request->setResponseCallback(CC_CALLBACK_2(HelloWorld::onHttpRequestCompleted, this));
// write the post data
const char* postData = "visitor=cocos2d&TestSuite=Extensions Test/NetworkTest";
request->setRequestData(postData, strlen(postData));
request->setTag("POST test");

// write request header
vector<string> headers;
headers.push_back("Cookie: GAMESESSIONID=MTQwNjI1Mjg3MnxEdi1CQkFFQ180SUFBUkFCRU");
request->setHeaders(headers);

cocos2d::network::HttpClient::getInstance()->send(request);
request->release();
```

Cocos2d-x 3.0 Json用法

Cocos2d-x 3.0 加入了rapidjson库用于json解析。位于external/json下。

rapidjson 项目地址：http://code.google.com/p/rapidjson/

wiki：http://code.google.com/p/rapidjson/wiki/UserGuide

Cocos2d-x 3.0 Json用法

使用rapidjson解析json串：

1. 引入头文件

```
#include "json/rapidjson.h"
#include "json/document.h"
```

2. Json解释

```
std::string str = "{\"hello\" : \"word\"}";
CCLOG("%s\n", str.c_str());
rapidjson::Document d;
d.Parse<0>(str.c_str());
if (d.HasParseError())  //打印解析错误
{
    CCLOG("GetParseError %s\n", d.GetParseError());
}

if (d.IsObject() && d.HasMember("hello")) {

    CCLOG("%s\n", d["hello"].GetString());//打印获取hello的值
}
```

打印结果

```
cocos2d: {"hello" : "word"}

cocos2d: word
```

COCOS2D-X

Cocos2d-x 3.0 Json用法

使用rapidjson生成json串：

1. 引入头文件

```
#include "json/document.h"
#include "json/writer.h"
#include "json/stringbuffer.h"
using namespace  rapidjson;
```

2.生成json串

```
rapidjson::Document document;
document.SetObject();
rapidjson::Document::AllocatorType& allocator = document.GetAllocator();
rapidjson::Value array(rapidjson::kArrayType);
rapidjson::Value object(rapidjson::kObjectType);
object.AddMember("int", 1, allocator);
object.AddMember("double", 1.0, allocator);
object.AddMember("bool", true, allocator);
object.AddMember("hello", "你好", allocator);
array.PushBack(object, allocator);

document.AddMember("json", "json string", allocator);
document.AddMember("array", array, allocator);

StringBuffer buffer;
rapidjson::Writer<StringBuffer> writer(buffer);
document.Accept(writer);

CCLOG("%s", buffer.GetString());
```

打印结果

```
cocos2d: {"json":"json string","array":[{"int":1,"double":1,"bool":true,"hello":"你好"}]}
```

常用算法

在游戏关卡中常常会放置一些怪物（即NPC），这些怪物通常在一个区域内走来走去，这个区域被称为“巡逻区域”;一旦玩家的角色进入怪物的“视野”，怪物就会发现玩家角色，并主动向其所在的位置移动，这个区域称为“警戒区域”；当玩家角色和怪物更加靠近时，会进入到怪物的“攻击区域”，这时怪物会对玩家角色进行伤害。在某些RPG（Role-Playing Game）中，NPC 在不利的情况下还会选择主动逃跑。如何模拟这些行为逻辑，目前游戏业已经有一些比较成熟的方法。

常用算法

随机寻路算法：

随机寻路算法适合模拟游戏中那些没有什么头脑的生物，它们总是在场景中漫无目的地走来走去。可以用以下的代码进行模拟：

```
npc_x_velocity = -5 + rand() % 10;
npc_y_velocity = -5 + rand() % 10;
int npc_move_count = 0;
while (++npc_move_count < num) {
    npc_x += npc_x_velocity;
    npc_y += npc_y_velocity;
} //end while
```

在上例中，NPC会选取一个随机方向和速率运动一会儿，然后再选取另一个。当然，还可以加上更多的随机性，如，改变运动方向的时间不是固定的num个周期，或者更倾向于朝某个方向等。实际编程中还必须考虑到碰撞检测，当NPC 遇到障碍物后，会随机选取一个前进的方向，继续行走。

COCOS2DX

常用算法

跟踪算法-普通版：

当游戏中的主角进入到NPC 的“警戒区域”后，控制NPC对象移向被跟踪的对象。
跟踪算法可以模拟这一行为：

```
void Bat_AI(void)
{
    if (ghost.x > bat.x)
        bat.x++;
    else if (ghost.x < bat.x)
        bat.x--;
    if (ghost.y > bat.y)
        bat.y++;
    else if (ghost.y < bat.y)
        bat.y--;
    //其他代码
} // end Bat_AI
```

代码运行时，NPC能够非常精确、迅速地追踪到目标。

常用算法

跟踪算法-矢量版：

起源：普通版的跟踪算法NPC会精确的跟踪目标。这会使得NPC看上去显得有点假。

算法设计如下：

假设AI 控制的对象称作跟踪者（tracker）并有以下属性：

```
Position: (tracker.x, tracker.y)
Velocity: (tracker.xv, tracker.yv)
```

被跟踪对象称作跟踪目标（target），有如下属性：

```
Position: (target.x, target.y)
Velocity: (target.xv, target.yv)
```

常用算法

跟踪算法-矢量版：

调整跟踪者的速度向量的常用逻辑循环：

1. 计算从跟踪者到跟踪目标的向量：
   TV = (target.x -tracker.x, target.y-tracker.y) = (tvx, tvy)， 规格化TV——也就是说(tvx, tvy)/ Vector_Length(tvx， tvy)使得最大长度为1.0， 记其为TV*。
   记住Vector_Length()只是计算从原点(0， 0)开始的矢量长度。

2. 调整跟踪者当前的速度向量，加上一个按rate比例缩放过的TV*：

```
tracker.x += rate*tvx;
tracker.y += rate*tvy;
```

   注意： 当rate >1.0时， 跟踪向量会合得更快，跟踪算法对目标跟踪得更紧密，并更快地修正目标的运动。

COCOS2DX

常用算法

跟踪算法-矢量版：

调整跟踪者的速度向量的常用逻辑循环：

3. 跟踪者的速度向量修改过之后，有可能向量的速度会溢出最大值，就是说， 跟踪者一旦锁定了目标的方向，就会继续沿着该方向加速。所以，需要设置一个上界，让跟踪者的速度从某处慢下来。可做如下改进：

```
tspeed = Vector_Length(tracker.xv, tracker.yv);
if(tspeed>max_SPEED){
    tracker.xv*=0.7;
    tracker.yv*=0.7;
}
```

也可以选择其它的边界值0.5 或0.9 等均可。如果追求完美，甚至可以计算出确切的溢出，并从向量中缩去相应的数量。追踪过程中同样也会遇到障碍物，因此，碰撞检测是必不可少的。程序员可以根据不同的游戏类型设计碰撞后的行为逻辑。

COCOS2DX

常用算法

闪避算法：

这个技术是让游戏的NPC能避开玩家角色的追击，跟前面的跟踪代码很相似，跟踪算法的对立面就是闪避算法，只要把上例中的等式翻转，闪避算法就成了，下面是转换后的代码：

```
if(ghost.x > bat.x)
    bat.x--;
else if(ghost.x < bat.x)
    bat.x++;
if(ghost.y > bat.y)
    bat.y--;
else if(ghost.y < bat.y)
    bat.y++;
```

常用算法

以上介绍的3 个算法可以模拟NPC 的一些简单的寻路、跟踪和闪避行为，在小游戏中会经常用到。但是，在较大型的游戏中使用这样简单的算法就会大大影响游戏效果了。因此，大型游戏的人工智能算法都较复杂。

内容节选自《游戏编程中的寻路算法研究》

THE END

THANKS FOR WATCHING