



TypeScript

- 虚幻引擎交互开发工程师 -



火星时代教育

概述

- 虚幻引擎交互开发工程师 -



火星时代教育

TS

- 起源于微软，是一款开源的编程语言，通过在Javascript基础上添加静态类型定义而构建。TypeScript通过TypeScript编译器或Babel转译为JavaScript代码，可运行在任何浏览器，任何操作系统。
 - 它是JavaScript的一个**超集**，而且本质上向这个语言添加了可选的静态类型和基于类的面向对象编程。
 - TypeScript扩展了JavaScript的语法，所以任何现有的JavaScript程序可以运行在TypeScript环境中。
- TypeScript是为大型应用的开发而设计，并且可以编译为JavaScript。

为什么选用TS

- 语法结构简单易懂，尤其从C++过渡学习成本低
- 带有面向对象编程特性，支持继承
- 带有接口特性，对于抽象行为，设计松耦合非常友好
- 带有类型批注与类型检查，使得编码过程中杜绝类型错误问题
- 编辑器友好，借助VS Code插件可以高效进行生产
- 支持泛型，更容易抽象针对类型设计的程序结构
- 对大型项目构建友好，对于设计模式实施成本较低
- 产品的迭代和后期重构困难较低

安装NodeJs

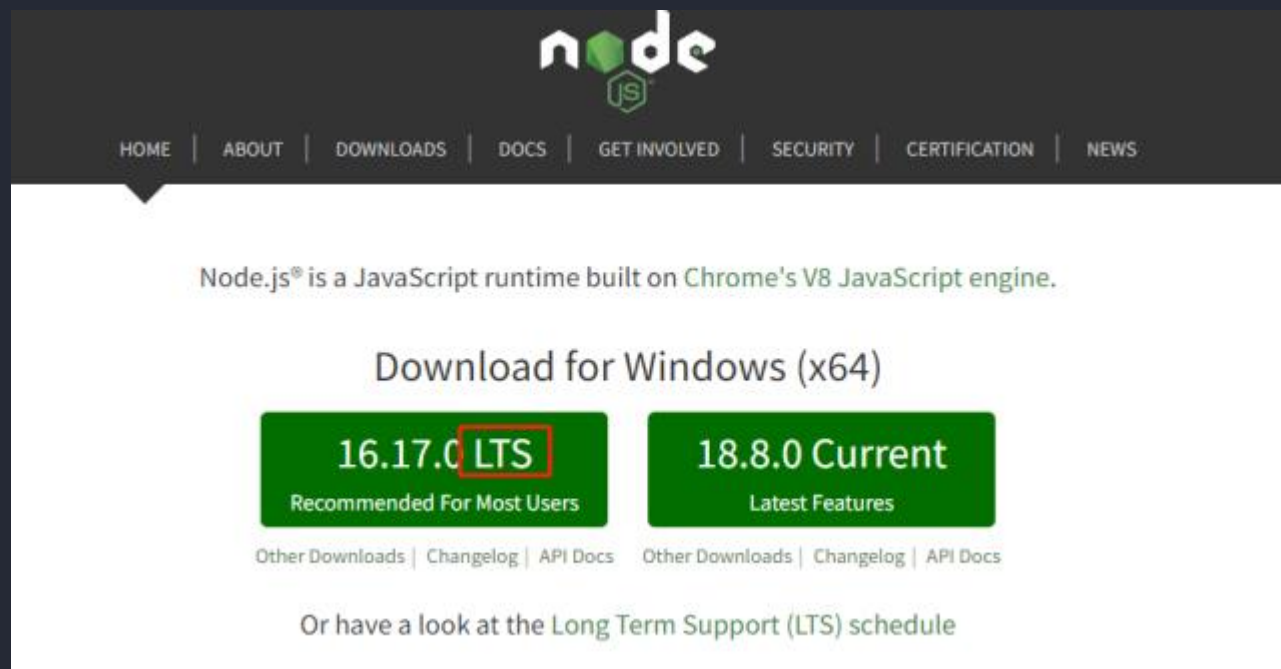
- 虚幻引擎交互开发工程师 -

下载NodeJs

登录Nodejs官方网站: <https://nodejs.org>

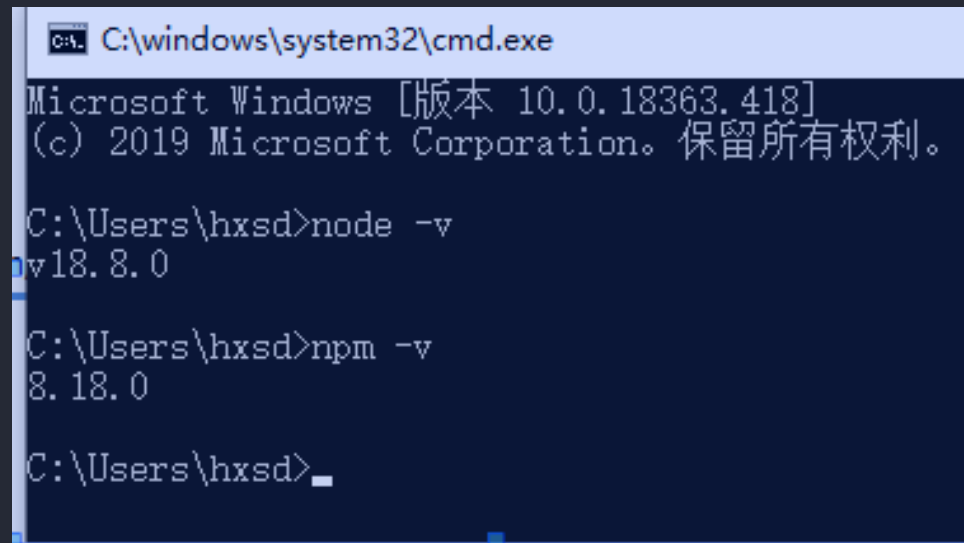
下载软件发行版本, 然后直接安装

安装就是一直执行下一步即可。



检查安装结果

在命令窗口中查看node与npm版本，如果可以正常查看，则表明nodejs安装成功。



```
C:\windows\system32\cmd.exe
Microsoft Windows [版本 10.0.18363.418]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\hxs>node -v
v18.8.0

C:\Users\hxs>npm -v
8.18.0

C:\Users\hxs>
```

安装编辑器

- 虚幻引擎交互开发工程师 -

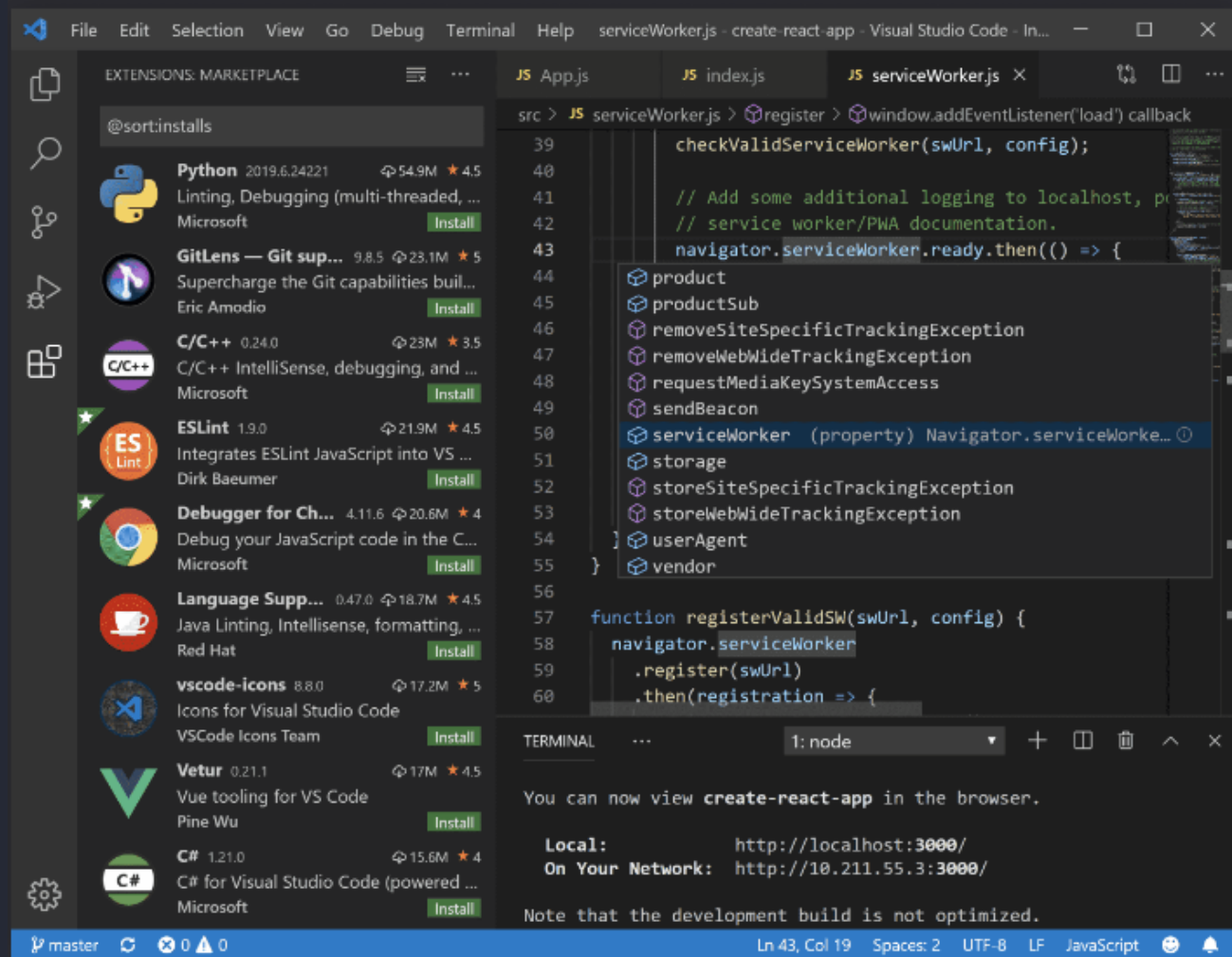


火星时代教育

Visual Studio Code

下载地址: <https://code.visualstudio.com/>

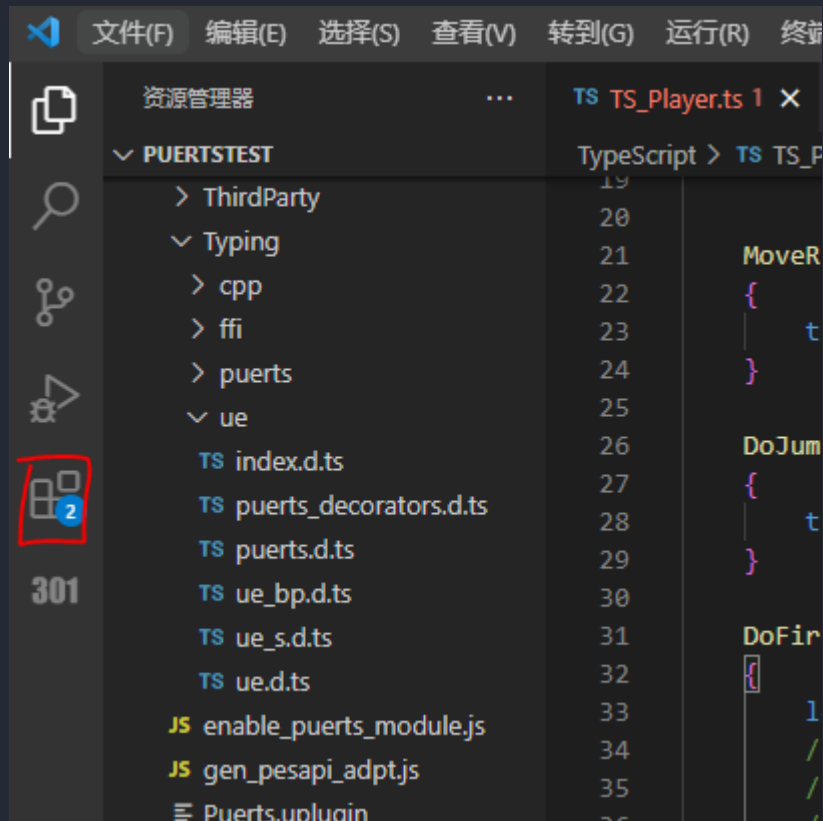
下载最新版本安装即可。



安装汉化环境

在软件中找到插件面板（ctrl+shift+x）在列表中搜索语言包插件
插件名称“Chinese 简体”，然后安装语言包。安装完成后需要
重启软件，被设置。

如设置失败，可手动配置，ctrl+shift+p打开配置选项框，选择
language，配置界面默认语言。



安装调试插件

在软件插件列表搜索 “typescript Debugger” ，
找到右图插件， 然后进行安装。



搭建工程环境

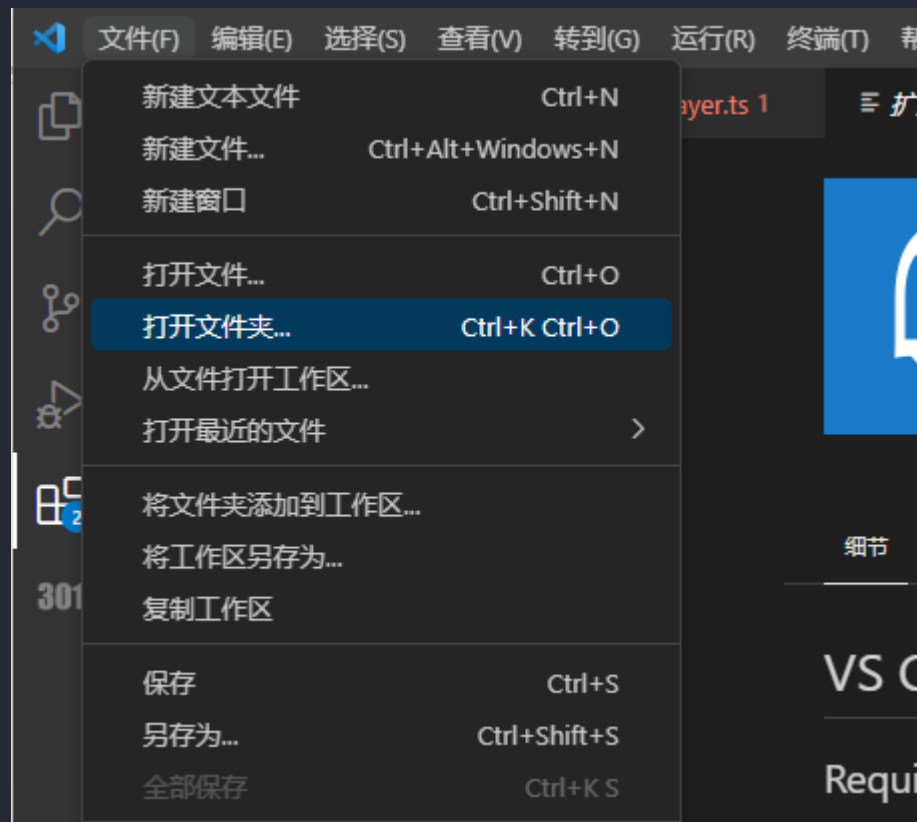
- 虚幻引擎交互开发工程师 -



火星时代教育

创建项目文件夹

打开VS Code，在文件中选择“打开文件夹”，将路径配置到磁盘中的某个**非中文**的空路径中。信任该路径。



安装ts-node

找到vs code的终端-新建终端。然后在终端控制台输入 “npm install ts-node” ，回车等待安装完成。

如果安装缓慢或是卡住，可以更换国内镜像

执行指令如下：

npm config set registry **URL**

其中URL为国内镜像地址，例如淘宝镜像：

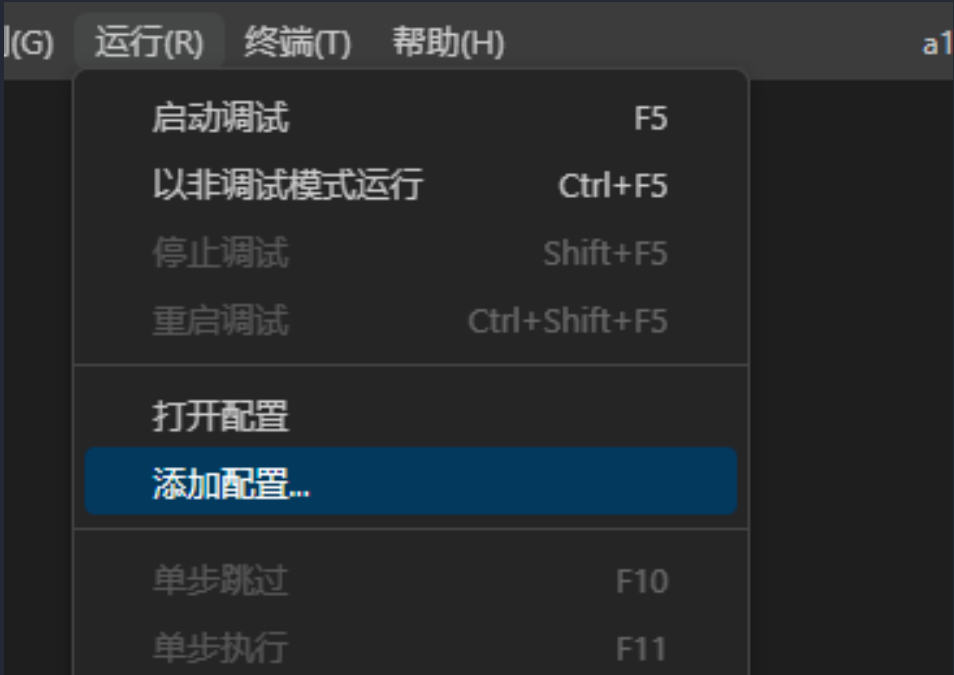
<https://registry.npmmirror.com>

```
ts>npm config set registry https://registry.npmmirror.com
```



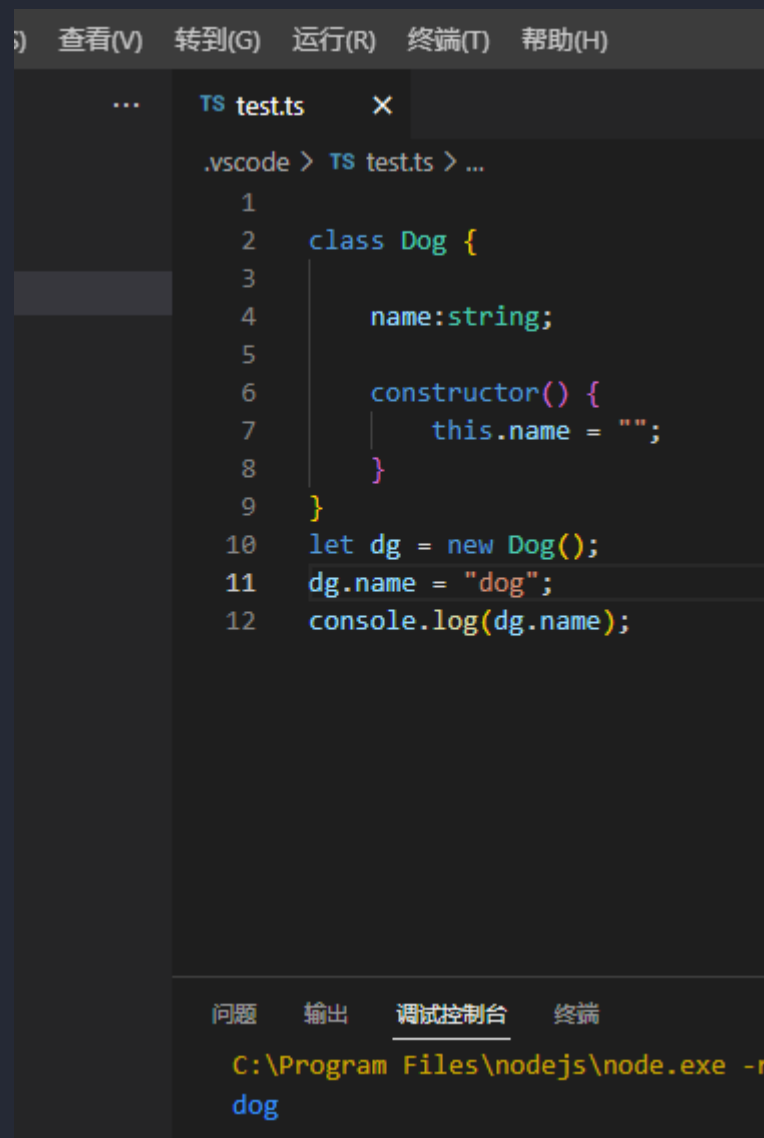
配置调试器

安装完ts-node后，需要进行调试器配置，由于ts-node安装到当前工程目录，故配置简单，点击运行-添加配置-选择TS-Debug即可。



环境测试

在项目添加ts文件，编写测试代码，ctrl+f5启动运行，如果控制台输出内容则表明环境测试成功。



```

查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)

... TS test.ts X
.vscode > TS test.ts > ...
1
2 class Dog {
3
4     name:string;
5
6     constructor() {
7         this.name = "";
8     }
9 }
10 let dg = new Dog();
11 dg.name = "dog";
12 console.log(dg.name);

问题 输出 调试控制台 终端
C:\Program Files\nodejs\node.exe -r
dog
```


基础语法格式

- 虚幻引擎交互开发工程师 -



火星时代教育

基础格式

行注释: `// 内容`

块注释: `/*内容*/`

行切换分割: `;`, 注意js中行分割符号可以省略, 但不建议省略。

```
// 行注释
```

```
/*跨行注释
```

```
这也是被注释的内容*/
```

```
console.log("Hello World");//行分隔符可以省略但不建议省略
```

日志输出

console是js的日志输出类，可以帮助我们输出日志，输出类型分为三种，普通，警告，错误。参考如下代码。

```
console.log("普通日志");  
console.warn("警告日志");  
console.error("错误日志");
```

数据类型

- 虚幻引擎交互开发工程师 -



火星时代教育

基础类型

类型	关键字	说明
数字型	number	双精度浮点，大小64位
字符串	string	可以使用单引号或双引号进行标注
布尔型	boolean	逻辑值，true和false
void	void	标注方法未出现返回内容
null	null	对象型数据为缺失
undefined	undefined	初始变量为未定义值

命名规则

- 可以包含字母或是数字
- 可以使用符号 “_” 和 “\$”，其他符号不能使用
- 不能以数字作为名称首个字符
- 区分大小写

声明语法

对于声明变量，**TS允许隐式推断类型**，也可以显示标记类型，其中let关键字和var关键字都可以用来做类型声明语法开端标记。

```
let n1 = 10; //声明number类型（注意ts中没有整型）
var n2 = 10; //同上，不指出变量类型，由右侧进行指出标注
let n3; //any类型，无明确指出类型
let n4 : number; //显示标注类型为数字型

n1 = 'a'; //非法
n3 = 1900; //合法
n3 = 'a'; //合法
n4 = 'a'; //非法
```

any类型：在声明变量时未明确指出类型或无法推断类型，则类型为any类型。**后续可以将变量指定任意类型。**

let和var

都是用来声明类型时的关键字。

let：声明的变量属于局部操作域（块作用域），声明周期只在局部域内有效。

var：声明的变量作用于全局域，声明周期在全局域有效，并且可访问。

```
{  
  var n = 100; //作用域全局域  
  let m = 100; //局部作用域创建，生命周期只在局部操作域有效  
}  
console.log(n); //正确的  
console.log(m); //错误的
```

建议使用let尽量替代var

枚举

枚举类型是用来描述有穷序列集合

```
enum Color {Red, Blue, Yello};  
  
let c1 : Color = Color.Red;
```

元组

元组类型构建的数据允许在数据中**按指定类型顺序**添加多种类型数据。**元组中访问超过数据个数位置数据，将出现语法错误提示。**

```
let mt : [number, string, boolean];  
mt = [5, "a", true]; //正确  
mt = [5, 50, true]; //错误  
  
console.log(mt[0]); //输出第一个元素  
console.log(mt.length); //获取长度
```

nervr

nervr表示在ts中永远不会存在的一种类型。

```
type a = 'a';
type b = 'b';

type c = never
type c = a & b; //c是一种交叉类型 类型是nervr
```

设计目的是为了解决执行流程中的意外逃脱。在流程管控上有被应用。

```
type a = 'a';
type b = 100;
type d = 'd';

type c = a | b; //c联合类型

function Pc(pp : c) {
  switch (pp) {
    case 'a':
      break;
    case 100:
      break;
    default:
      //语法检查，编译器会检查，如果pp类型中不会出现其他值类型则语法正确，否则错误
      //执行到default说明pp不属于给定的类型，那他应该是nervr类型，但是如果不是nervr类型则语法错误
      const mm : never = pp;
      break;
  }
}
```

nerver使用

借助nerver在语法阶段检查逻辑，提出警告。

如果c类型等于 $a \mid b \mid d$ ，则const mm语法错误。

```
type a = 'a';
type b = 100;
type d = 'd';

type c = a | b; //c联合类型

function Pc(pp : c) {
  switch (pp) {
    case 'a':
      break;
    case 100:
      break;
    default:
      //语法检查，编译器会检查，如果pp类型中不会出现其他值类型则语法正确，否则错误
      //执行到default说明pp不属于给定的类型，那他应该是never类型，但是如果不是never类型则语法错误
      const mm : never = pp;
      break;
  }
}
```

数组

- 虚幻引擎交互开发工程师 -



火星时代教育

数组

数组是ts中常用的数据存储容器，与C++不同，ts中的数组不会约束数组的大小，具有自动扩容能力，访问下标可以是任意整型，包括复数。

```
let ary : number[] = [5, 10, 3, 5, 6];  
let ary1 : Array<number> = [5, 6 ];//也可以声明数组  
  
ary[10] = 100;//正确，但是在前面没有数据的位置会被装填undefined类型  
console.log(ary[6]);//undefined  
console.log(ary[-1]);//undefined
```

遍历数组

```
let ary : number[] = [4, 5, 6];

for (let index = 0; index < ary.length; index++) {

}

for (const key in ary) {
    //由于ary是数组，所以key是数组的下标
    ary[key]; //是值
}

for (const val of ary) {
    //val是元素值
}

ary.forEach(val =>{
    console.log(val); //值
})

ary.every((val, index) =>{
    console.log(val); //元素值
    console.log(index); //位置
    return true; //继续向下遍历
})

ary.some((val, index) =>{
    console.log(val); //元素值
    console.log(index); //位置
})
```

数组的一些操作

```
ary.length;//获取长度
//访问
ary[3];
ary.at(3);//获取数据，等同于ary[3] 如果存在数据返回，不存在返回undefined
//修改
ary[0] = 10;//修改数据
ary.fill(3);//将所有位置使用3进行装填
//添加
ary.push(11);//在数组末尾添加元素
ary.unshift(6, 2, 3);//将元素按照顺序添加到数组头，[6, 2, 3, 原数组...]
//删除
ary.splice(0, 2);//移除0位开始向后的2个元素，并将其他元素前移
ary.pop();//弹出末尾元素，并从数组中移除末尾元素
ary.shift();//删除队首元素
//合并数组
ary.concat(ary1);
//查询元素
ary.indexOf(5);//返回5在数组中第一次出现位置，如没有返回-1
//分割数组
let nary = ary.slice(2, 5);//返回从数组位置2-5的元素，并装填到新数组。5如果超过数组大小则停止装填。
```


二维数组

TS中支持二维数组

```
let ary : number[][] = [[1, 3, 4], [2, 3, 4]];
```

特殊关键字

- 虚幻引擎交互开发工程师 -



火星时代教育

type关键字

给与已知类型进行别名创建。可以用来进行基本类型，联合类型，元组，对象型等别称

```
type n = number;  
let m : n = 100; //n是number类型的别称 等同于语法 let m : number = 100;  
  
type n1 = 100; //尽量不要书写  
let m1 : n1 = 100; //n1是100的类型别称，等同于语法 let m1 : 100 = 100; 类似常量
```

```
type n = number; //数字型  
type ob = {}; //对象型  
type f1 = () => {}; //函数型  
type dt = [number, string]; //元组型  
type un = number | string; //联合型  
type mix = number & string; //交叉型
```

const关键字

标记内容不可被更改，主要起到保护数据作用。注意，声明时需要初始化。

```
const n : number = 100;
```

```
n = 10; //错误 无法修改
```

```
const n1 : number; //错误必须给与初始化值
```

typeof关键字

用来推断给定数据的类型

```
let n : string = 'aa';  
  
console.log(typeof n); //输出string
```

交叉与联合类型

- 虚幻引擎交互开发工程师 -



火星时代教育

交叉类型

将多个类型合并为一个类型，通过使用&符号进行关联，包含给定的所有类型特性。

```
type Box = {  
  weight : number;  
  height : number;  
}  
  
type Circle = {  
  radiuse : number;  
}  
  
type Shape = Box & Circle;  
  
let sh : Shape = {weight : 10, height : 10, radiuse: 3};
```

联合类型

将多个类型联合在一起，用来表示值是其中的一种类型（类似C++中的联合体），通过符号|分割。注意，数组也可以构建为联合类型。

```
let mp : number | string = 'a';
```

```
mp = 100; //正确
```

```
mp = 'c'; //正确
```

```
mp = true; //错误
```

```
//联合类型数组
```

```
let ary : string[] | number[];
```

```
ary = [1, 2, 3];
```

```
|
```


运算符

- 虚幻引擎交互开发工程师 -



火星时代教育

算术运算符

与C++运算符规则一样，包括加，减，乘，除，取余，自增，自减。同时增加一种指数运算 “**” 。

```
let n : number = 50 ** 2; //50的2次方
```

关系运算符

与C++运算符规则一样, 包括 "==" "!=" ">" "<" ">=" "<="

逻辑运算符

与C++运算符规则一样，包括 “||” “&&” “!” “

赋值运算符

与C++运算符规则一样，包括 “=” “+=” “-=” “*=” “/=”

位运算符

符号	说明
&	按位与
	按位或
~	按位取反
^	按位异或
<<	左移
>>	右移
>>>	无符号右移

赋值运算符（位运算）

位运算符的赋值运算符和赋值运算符一致，只是将运算方式修改为位运算方式。

$\ll=$, $\gg=$, $\gg=$, $\&=$, $|=$ 与 $\wedge=$

三目运算符

三目运算符与C++的操作保持一致

```
let n : string = 'aa';  
  
let n1 : number = n == 'a' ? 10 : 499;
```

注意在三目运算中，如果返回结果值类型不同，并且未强制约束声明参数类型，则参数类型为联合类型。

```
let m = Math.random() > 0.5 ? 'a' : 10; // m是联合类型 string | number  
let m1 : number = Math.random() > 0.5 ? 'a' : 10; // 错误，无法将联合类型赋值number类型
```


循环

- 虚幻引擎交互开发工程师 -



火星时代教育

for循环

与C++基本一致，但增加了对于容器的快捷便捷访问。例如for..in

```
for (let index = 0; index < 5; index++) {  
  console.log(index);  
}  
  
let mt : [number, string, boolean] = [10, 'a', true];  
  
for(let m in mt)  
{  
  console.log(m + ' ' + mt[m]); //数据位置，mt[m]是内容  
}  
  
let ary : number[] = [4, 5, 6];  
  
for(let m in ary)  
{  
  console.log(m + ' ' + ary[m]); //数据位置，ary[m]是内容  
}
```

for..of

for..of返回容器中数据内容

```
let ary : number[] = [4, 5, 6];

for(let m of ary)
{
    console.log(m); //返回数据内容
}
```

forEach

forEach借助匿名函数遍历容器中元素

```
let ary : number[] = [4, 5, 6];

ary.forEach(element => {
    console.log(element);
});

ary.forEach((val, index, arry)=>{
    console.log(val);//值
    console.log(index);//下标
    console.log(arry);//数组
})
```

while与do..while

与C++保持一致

循环流程控制关键字

`break`: 终止循环执行, 并跳出最近一层循环。

`continue`: 提前结束本次循环, 并开始下次循环。

流程控制语句（条件）

- 虚幻引擎交互开发工程师 -



火星时代教育

流程控制条件语句

if:

if..else:

if..else if...else:

switch:

函数

- 虚幻引擎交互开发工程师 -



火星时代教育

函数语法

function name (parameter : type) : return_type { body }

function: 语法标记, 表明函数开始

parameter: 参数如果有可以填入, 填入时可以明确类型, 或是省略类型, 省略类型为any

return_type: 返回类型, 可以省略, 也可以明确指出

```
function Say(msg : string) {  
  |   console.log(msg);  
}
```

建议标注参数类型, 与返回类型, ts鼓励明确类型

特殊返回值

函数在编写时可以明确返回值类型，但是有三个返回值类型比较特殊，void，any，never。

```
function f1() : void {  
    return true;//语法错误 如果直接填写 return;是正确的  
}  
  
function f2(num : number) : any {  
    if (num == 0) {  
        return 'a';  
    } else {  
        return true;  
    }  
}  
  
function f3() : never { //表明永远无法执行到函数结尾，执行到行尾此函数返回undefined  
    while (true) {  
  
    }  
}
```

默认参数

同c++语法一样，函数参数可以填入默认值。注意参数设定默认值后，此参数后的参数均需要给出默认值，如未给出，语法不错，但是前面的默认参数调用时无法留空。

```
function f1(num : number = 1) : void {  
    console.log(num);  
}  
//语法无措，但是默参无效  
function f2(num : number = 1, str : string) : void {  
    console.log(num);  
}
```

可选参数

- 可选参数是调用函数时不填写参数值也是正确的。
- 但是需要注意，当有参数是可选参数时，此参数后的参数禁止填写普通参数，但可以是带有默认参数的，或是也是可选参数。
- **当可选参数未填时，参数值为undefined。**

```
function f1(name : string, age ? : number) : void {  
    }  
//错误，无默认值的参数不能在可选参数后面  
function f2(name ? : string, age : number) {  
    }  
//正确  
function f3(name ? : string, age : number = 10) {  
    }  
//正确  
function f4(name ? : string, age ? : number) {  
    }  
  
f1('hi', 10);  
f1('hi');//依旧正确
```

剩余参数

类似C++可变参数，但是与可变参数相比可控性更强，格式更固定。**剩余参数必须是参数队列的最后一个。**
调用时填写参数，如果未填，数组长度为0，如果填写，将把所有参数存储到数组中。

```
function f1(age : number, ...friends : string[]) : void {  
  
}  
  
f1(10);  
f1(10, 'a');  
f1(10, 'a', 'b', 'c');
```

函数重载

与C++不同，由于C++属于静态语言（编写过程中结构类型是确定的），而TS有JS拓展而来，JS属于动态语言（运行时决定类型），所以对于重载JS是没有必要的（参数可以随意填写，并不强约束）。但是TS增加了类型约束，故可以使用重载操作，但是总体来说更复杂一些。

```
//声明
function f1(params:number) : void;
function f1(params:string, str : string) : void;
//定义 定义函数需要保证覆盖满足所有声明函数参数结构
function f1(params:any, st ? : string) : any {

}
```

匿名函数

- 虚幻引擎交互开发工程师 -



火星时代教育

匿名函数

匿名函数是指没有函数名称的函数。匿名函数就是单纯的声明函数不带有名称。

```
let f1 = function () {}
```

```
//匿名自调用， 但是注意自调用函数结构不能带有参数  
(function () { console.log("匿名"); }) ();
```

Lambda表达式

- 虚幻引擎交互开发工程师 -



火星时代教育

Lambda

在TS被称之为箭头函数。是一种简短的函数表达式，语句更加精炼。

关于参数捕获，在匿名中可以捕获当前域内所有的内容，字段可读可写，函数可调用。

```
let f1 = () => {};  
let f2 = a => console.log(a); // 单一参数，单一语句可以省略括号和花括号  
let f3 = (a, b) => console.log(a + b); // 多参数不能省略括号  
let f4 = (a, b) => { // 多语句不能省略花括号  
  console.log(a);  
  console.log(b);  
}
```

Array

- 虚幻引擎交互开发工程师 -



火星时代教育

Array

Array是TS中添加的一种数据容器，也被称为数组。创建语法如下：

```
//创建空数组
let ary : Array<number> = [];
let ary1 : number[] = [];
//创建长度为4的数组
let ary2 : Array<number> = new Array<number>(4);
//多维度
let ary3 : number[][] = [[1, 2, 3], [2, 5, 3]];
//做为参数
function f1(str:string[]) {}
//做为返回值
function f2() : string[] { return new Array<string>(); }
```

操作API

every 检查每个元素是否符合给定的条件。

例如：检查数组中是否所有的数都是偶数？

```
let ary : number[] = [2, 4, 6, 8, 9];

let bHave = ary.every((element, index, arry)=>{
  if(element % 2 == 0)
  {
    return true;
  }
  return false; //只要返回false则停止向下检查执行
});

console.log(bHave);
```

操作API

`some` 检查数组元素是否存在符合条件的元素内容，返回真假结果。Every是所有都要满足条件，`some`只要有一个即可。

例如：检查数组中是否包含偶数？

```
let ary : number[] = [2, 4, 6, 8, 9, 18];  
  
let bHave = ary.some((element) =>{  
    return element % 2 == 0;  
});
```

操作API

filter 检查数组元素是否符合条件，并将符合条件的元素装填到新的数组返回。

例如：将数组中的所有偶数取出来。

```
let ary : number[] = [2, 4, 6, 8, 9, 18];

let newArray = ary.filter((element, index, array)=>{
  |   return element % 2 == 0;
});

newArray.forEach(element => {
  |   console.log(element);
});
```


操作API

join 将数组所有元素放置到一个字符串中。

```
let ary : number[] = [2, 4, 6, 8, 9, 18];  
  
let str = ary.join();  
  
console.log(str); //2,4,6,8,9,18
```

操作API

reduce 将数组中的相邻元素按照给定的规则执行逻辑，并返回结果，将结果与下一个元素进行相同规则执行，最终返回执行结果。

```
let ary : number[] = [2, 4, 6, 8, 9, 18];

//求和
let result = ary.reduce((a, b)=>{
  |   return a + b;
});
//求最大数
let max = ary.reduce((a, b)=>{
  |   return a > b ? a : b;
});
```



Map

- 虚幻引擎交互开发工程师 -



火星时代教育

Map

数据存储时需要提供键与值，并且键值是对应的关联关系。任何的值（对象和基础数据类型）都可以作为键或是值。

在TS中如果明确指出键和值类型，则填入时需要进行检查。

```
//不约束键值的类型和值类型，同一个map中可以装入不同的键类型
let m1 = new Map();
//约束类型 必须按照给键和值类型进行填入
let m2 = new Map([[10, 0]]);
let m3 = new Map<number, string>();
let m4 = new Map<number, string>([[1, 'a']]);
let m5 : Map<number, string> = new Map([[10, 'a']]);
```

读，写，查

```
let m1 : Map<number, string> = new Map();

//写入和修改
m1.set(10, 'a');//如果存在key则修改，不存在则添加
//检查是否存在key
m1.has(10);
//读取key对应的值
m1.get(10);
//获取容器大小
m1.size;
```

删除

```
m1.clear();//清理掉所有  
m1.delete(10);//删除给定的键对应的数据，成功返回true，失败返回false
```

遍历

```
for (const iterator of m1) {  
  iterator[0]; // 键  
  iterator[1]; // 值  
}
```

```
m1.forEach((value, key) => {});
```

类

- 虚幻引擎交互开发工程师 -



火星时代教育

类

TS的主要特性就是在JS基础上增加了面向对象。类是属性和方法的集合，是抽象事物的表现手段。

```
class Animal {  
    //成员属性也称作字段  
    name : string;  
    age : number;  
    health; //未明确标注类型 不建议  
    //所有属性必须在构造函数中初始化 可以进行多构造函数编写 遵循函数重载规则  
    constructor();  
    constructor(num:number);  
    constructor(para ? : number)  
    {  
        this.name = '未知'; //类内调用属性需加this  
        this.age = 0;  
        this.health = 100;  
    }  
  
    SaySelf() : void {  
        console.log(this.name);  
    }  
}
```

规则

- 成员属性（字段）添加时必须构造函数中初始化。
- 成员属性添加时可以不用明确描述类型。但建议指出类型，这符合TS设计。
- 构造函数可以带有参数，并支持重载，重载遵循函数重载方式。
- 成员函数不需要使用function关键字标注开始。
- 类内调用成员内容需要使用this进行指认。
- 类名称遵守变量命名规则

实例化对象与使用

创建对象通过关键字new完成，对象释放无需管理。

```
//通过无参构造创建对象
let a : Animal = new Animal();
//调用字段
a.age = 10;
//调用函数
a.SaySelf();
```

检查实例类型

TS中添加关键字instanceof，用于检查实例对象源自哪种类型。

Instanceof关键字不支持检查接口类型（**接口类型无法被实例化**）。

```
class A{  
  
}  
  
class B extends A{  
  
}  
  
class C extends A{  
  
}  
  
let obj : A = new B();  
//检查obj是否继承自B类型  
if(obj instanceof B){  
  
}
```

继承

- 虚幻引擎交互开发工程师 -



火星时代教育

继承

TS支持继承操作，通过使用关键字extends完成。

```
class Animal {  
    //成员属性也称作字段  
    name : string;  
    age : number;  
    health; //未明确标注类型 不建议  
    //所有属性必须在构造函数中初始化 可以进行多构造函数编写 遵循函数重载规则  
    constructor();  
    constructor(num:number);  
    constructor(para ? : number)  
    {  
        this.name = '未知'; //类内调用属性需加this  
        this.age = 0;  
        this.health = 100;  
    }  
  
    SaySelf() : void{  
        console.log(this.name);  
    }  
}  
  
class Dog extends Animal {  
    constructor() {  
        //super(); //必须要显示调用父类的构造  
        super(0); //二选一  
    }  
}
```

注意

子类继承父类后，如果子类编写构造函数，则必须调用父类构造（任意一个即可）。通过使用关键字super完成。

TS中不支持多继承，但是不同类型数据如果希望具备相同的行为特性可以借助接口完成。

重写

TS支持面向对象中的多态特性，可以重写父类方法，通过里氏转换将子类对象存储到父类上，父类对象调用重写方法展现子类对象行为结果。

由于TS没有重写关键字，展示的操作类似重定义，如果需要调用父类方法需要借助super关键字。

```
class A {  
  Say(): void  
  {  
    console.log('a');  
  }  
}  
  
class B extends A {  
  Say(): void { //重写  
    //super.Say();//显示调用父类同名函数  
    console.log('b');  
  }  
}  
  
let obj : A = new B();//里氏转换  
obj.Say();//输出b
```


访问修饰符

与C++一致，TS提供public, protected, private访问修饰符。

```
class A {  
    private name : string; //只有自己能访问  
    protected age : number; //自己和子类能访问  
    public health : number; //公开访问，可以省略public  
  
    private Say() : void {} //方法与字段标注修饰一样  
}
```

接口

- 虚幻引擎交互开发工程师 -

接口

TS遵循OOP编程特性，通过抽象类对行为进行规范并给出必要的属性，衍生类主要是拓展父类，并具象化父类。

但是在常规面向对象编程设计中，衍生类的多继承同时带来很多理解困扰。

例如：动物下面有猫和狗的具象类，飞行器下面有直升飞机和喷气式飞机的具象类，在C++中如果想要猫和狗能够具备飞行的能力，往往可以通过继承来完成，但是这种继承关系就令人匪夷所思了。毕竟猫和狗不属于飞行器。

接口主要对行为的入口负责，不对行为的业务结果负责。只负责描述抽象行为，不负责具象行为结果。同时应用时具备多态的特性，并能够表现多态的执行结果。这就是接口的优势。

简而言之，通过另外一种规则的设定，规避了多继承中的继承冗余与继承关系混乱问题。

接口中只带有函数，并不具体实现函数，并且不具备属性。

操作

一个类可以继承多个接口，一个类可以继承一个父类并继承一个接口

```
//创建接口类
interface IFly{
    //添加接口行为但不定义接口行为
    DoFly() : void;
}

class Animal{
}

//继承父类后可以再实现（继承）接口
class Dog extends Animal implements IFly{
    //继承接口后必须实现接口函数
    DoFly(): void {
        console.log('狗儿会飞');
    }
}

//只继承接口
class Cat implements IFly{
    DoFly(): void {

    }
}

//继承接口后，Dog类实例也属于接口类型实例，根据里氏转换规则进行赋值
let sdog : IFly = new Dog();
sdog.DoFly();//多态，打印‘狗儿会飞’
```

总结

- 接口内只能编写方法声明，但不能定义
- 接口内禁止添加属性字段
- 实现接口的类必须将接口内的所有方法实现
- 一个类可以实现多个接口，多个接口之间通过逗号分割。但是一个类只能继承一个父类。
- 接口建议首字母使用 “I” 来声明
- 接口遵守多态特性与里氏转换规则
- 接口类型无法被实例化

实现多个接口

```
//创建接口类
interface IFly{
    //添加接口行为但不定义接口行为
    DoFly() : void;
}

interface IRun{
    DoRun() : void;
}

//实现两个接口
class Hero implements IFly, IRun{
    DoRun(): void {
        throw new Error("Method not implemented.");
    }
    DoFly(): void {
        throw new Error("Method not implemented.");
    }
}
```

命名空间

- 虚幻引擎交互开发工程师 -



火星时代教育

概述

TS中支持构建命名空间（老版本中称为之“内部模块”），通过使用关键字“namespace”来创建。

```
//创建命名空间  
namespace NS{  
  
}
```


内容导出

在命名空间中添加的内容如果在空间外部使用需要通过关键字export进行导出。

```
//创建命名空间
namespace NS{
    //如果外部使用则需要导出
    export class Ball{}
}

let b1 : NS.Ball = new NS.Ball();
```

嵌套空间

TS中的空间允许进行嵌套，也就是空间内已经可以添加空间。嵌套空间外部访问需要进行导出。

```
//创建命名空间
namespace NS{
    //如果外部使用则需要导出
    export class Ball{}
    //嵌套空间
    export namespace NS1{
        export class Box{}
    }
}
```

别名操作

空间内的内容进行导出后，可以通过导入关键字import进行别名化操作。

```
//创建命名空间
namespace NS{
    //如果外部使用则需要导出
    export class Ball{}
    //嵌套空间
    export namespace NS1{
        export class Box{}
    }
}
//导入空间后进行别名操作
import NS1 = NS.NS1;

let bx : NS1.Box = new NS1.Box;
```

Thanks

-火星时代游戏设计学院-