介绍

NDF语言是用于描述游戏数据的.

它是一种声明性语言,定义互相关联且具有某些属性的对象，以便在游戏运行期间正确的实例化数据.

语法

字符集

所有NDF文件应采用UTF-8或者ANSI编码,且仅限于7位表.
NDF解析器不区分大小写: '`TRUE`'和'`true`'等效.

关键字

`export`, `is`, `template`, `unnamed`, `nil`, `private`, `int`, `string`, `true`, `false`, `div`, `map`

符号

`//`, `/`, `?`, `:`, `=`, `|`, `&`, `<`, `>`, `>=`, `<=`, `!=`, `-`, `+`, `*`, `%`, `,(comma)`, `.(dot)`

快分割符

`{}`, `[]`, `()`, `<>`, `(* *)`, `/* */`, `' '`, `" "`

注释

符号`//`之后的所有内容都将被忽略，直到行结束
由`{}`或`(**)`或`/**/`分隔的块中的所有内容都将被忽略。

内置类型

布尔逻辑

只能是 **true** 或 **false**.

字符串

字符串由单引号或双引号分隔，并且可以包含强调字符.

- `"这是个字符串"`
- `'This ïs ànothér on€'`

整数

整数以十进制或十六进制表示法编写.

- `3`
- `150486`
- `-36584`

- `0xFF00A8`
- `0x1`

浮点数

浮点数通过10为底的自然表示法表示，没有指数或十六进制表示法。这个十进制分隔符是字符`.(点)`.

- `3.1415954`
- `-9.81`
- `654987.1248`

整数部分可以省略(隐式0). 小数部分可以省略(隐式0).

- `.127`
- `53.`

向量

向量是一个由零个或多个元素组成的列表，这些元素包含在`[]`块中，并用`,(逗号)`分隔.

- `[]  // 一个空向量`
- `[1, 2, 3]  //  一个整数向量`
- `["Hello", 'World',]  //  接受逗号分隔符结尾`

对

对是一个包含两个内部对象的对象，它们包含在`()`块中，用`,(逗号)`分隔.

- `(22, 7) // 整数对`
- `("Hello", "World") // 字符对`

表

表是包含零对或多对(一个键及其相关值)的列表。它是通过在[]块之前添加关键字MAP创建的。

```
// 一个整数对应到字符串的表?
MAP[
 (1, 'one'),
 (2, 'two'),
 (3, 'three')
]
```

对象

命名内置值

可以使用关键字is为内置类型的值命名，类似于在编程语言中创建变量。

```
Gravity is 9.81

Currencies is MAP[
('EUR', '€'),
('USD', '$'),
('GBP', '£')
]

Places is ['Antarctica', 'Everest', 'Mars']
```

算术操作

常用的数字运算有：加、减、乘、除、模。

```
Pi is 3 + 0.1415954

PiCube is Pi * Pi * Pi

X is 35 div 8 // 除法是通过关键字“div”使用的

Y is 35 % 8
```

您还可以使用+运算符连接字符串、向量和表。

```
A is [1, 2] + [3, 4]

B is MAP[ (1, 'one')] + MAP[ (2, 'two')]

C is "Hello" + " World!"
```

对象定义

除了内置值外，NDF还允许创建和使用复杂的对象。

对象由其名称和类型定义，可以包含成员值。类型（几乎？）总是以大写字母“T”开头。它们代表游戏的内部数据结构，其定义不可用。

加载数据时，游戏将创建所需类型的对象，并用NDF描述中填写的成员值填充其字段。

```
// 这将创建一个包含两个成员的TExampleType实例
ExampleObject is TExampleType (
    MemberInteger = 12
    MemberString = "something"
)
```

如果类型定义包含未在对象的NDF定义中填充的成员值，则默认为基值。对于数字来说，这通常是0，对于布尔值来说是false，对于容器（字符串、向量、映射）来说是空的。

类型可以有其他类型作为成员，从而产生“嵌套对象”。

```
ExampleObject is TExampleType
(
    innerObject = TOtherType
    (
        ValueString = "I am a member of TExampleType"
    )
)
```

对象可以在没有名称的情况下使用关键字**unnamed**定义，它们被称为未命名对象。未命名的对象只能定义为“顶级对象”，这意味着它们不是另一个对象的成员。

```
unnamed TExampleType
(
    ValueString = "I am an unnamed object"
)
```

命名空间

对象的每个定义，无论是否命名，都会根据其名称创建一个名称空间。我们将使用 **$/Namespace1/Namespace2/Object** 绝对名称的对象表示法。

First example

```
ExampleObject is TExampleType
(
    InnerExample = ExampleObject2 is TExampleType
    (
        InnerExample = ExampleObject3 is TExampleType()
    )
)
```

在该示例中，**ExampleObject3**的绝对名称为 **$/ExampleObject/ExampleObject2/ExampleObject3**。

> **Note**
> Note that objects assigned to members can also be named, as it's the case for ExampleObject2 and ExampleObject3

Second example

```
ExampleObject is TExampleType
(
    InnerExample = TExampleType
    (
        InnerExample = ExampleObject3 is TExampleType()
    )
)
```

这里，ExampleObject3的绝对名称是**$/ExampleObject//ExampleObject3**（注意双斜线）。

Third example

```
ExampleObject is TExampleType
(
    InnerExample = _ is TExampleType
    (
        InnerExample = ExampleObject3 is TExampleType()
    )
)
```

最后，在这种情况下，**ExampleObject3**的绝对名称是**$/ExampleObject/ExampleObject3**。标记`_`（下划线）充当一个不引入名称空间的特殊名称。

引用对象

引用是与对象名称相对应的标签。引用可以为null，在这种情况下等于nil。

某些类型将引用作为成员，因为它们需要能够访问其他对象才能工作。

```
DataHolder is TDataHolder
(
    SomeInt = 456
    SomeString = "A string"
    SomeMap = MAP[ {...} ]
)

DataUser is TDataUser
(
    Condition = false
    DataHolderReference = DataHolder // 这里我们引用DataHolder对象
)
```

在上述情况下，找到的第一个名为**DataHolder**的对象将被视为引用。根据对象的范围，还有其他引用对象的方法。在不深入细节的情况下，在启动过程中加载了许多其他对modder不可见的NDF文件，可修改的文件可能需要引用它们。

这就是其他类型的参考发挥作用的地方：

```
$/Path/To/OtherObject // 绝对引用
~/Path/To/OtherObject // 来自加载命名空间的引用（modder无法知道）
./Path/To/OtherObject // 来自当前命名空间的引用
```

原型

原型是一个常规对象，将用于从中创建另一个对象。任何命名的对象都可以作为原型。

从原型创建对象将把原型的副本复制到新对象中，并允许随意覆盖某些成员。

```
Prototype is TThing
(
    ValueString = 'I am a prototype object'
    ValueInt = 666
)

ObjectFromPrototype is Prototype
(
    ValueString = 'I am just me'
)
```

ObjectFromPrototype有一个从**Prototype**复制的成员**ValueInt=666**，并覆盖其**ValueString**成员。

模板

模板允许以通用方式从参数列表生成对象。它们与C家族语言中的宏非常接近。
它们由关键字**template**、名称、**[ ]**块中包含的参数列表以及模板正文定义。

```
// 什么都不做的基本模板
template MyTemplate // 模板名称
[]     // 模板参数
is TType    // 模板创建的对象的最终类型
()     // 类型成员
```

参数块定义了一个参数列表，用,(逗号)分隔。
参数的语法为 **%name% [: %type%] [= %default%]**. 该名称为必填项，然而：

- 可以指定可选类型，否则将从上下文中推断出来
- 可以指定可选的默认值，否则在使用模板时必须提供该值
  在模板体中，模板参数由其名称括在**< >**中使用。

```
// 更详细的示例
// 假设存在TWeapon和TCharacter类型。

Axe is TWeapon
(
    // 下面是成员们对这种武器的描述
)

Knife is TWeapon ()

template Character
[
    Name : string, // Name必须是字符串
    Level : int = 1, // Level的默认值为1
    Weapon : TWeapon = TWeapon() // Weapon必须是TWeapon
]
is TCharacter
(
    Name = <Name> // <Name>指的是名为“Name”的模板参数
    Level = <Level>
    HP = <Level> * 100
    Weapon = <Weapon>
    Damages = <Weapon>/Damages * (1 + Level div 10)
```

```
)

Hero is Character
(
    Name = "Hero"
    Level = 12
    Weapon = Axe
)

Creep is Character
(
    Name = "Creep"
    Weapon = Knife
    // 未指定Level，使用默认值
)
```

高级模板

范围对象

对象可以在模板体内部声明。

```
template Character
[
    Name : string,
    Level : int = 1,
    Weapon : TWeapon = TWeapon()
]
is TCharacter
(
    Bag is TInventory
    (
        MaxItemCount = <Level> * 3
    )
    Name = <Name>
    Level = <Level>
    HP = <Level> * 100
    Weapon = <Weapon>
    Damages = <Weapon>/Damages * (1 + Level div 10)
    Inventory = Bag
)
```

模板的模板

一个模板可以从另一个模板派生。

```
template CloneHero
[
    Name : string
]
is Character
(
    Name = <Name>
    Level = 12
    Weapon = Axe
)
```

派生模板也可以覆盖作用域对象。

```
template CloneHero
[
    Name : string
]
is Character
(
    Name = <Name>
    Level = 12
    Weapon = Axe

    Bag is TInventory ( MaxItemCount = 0 )
)
```