|  |  |
| --- | --- |
| **JAN语言程序设计**  弯道和树的图片  **The world is hen, and I use Jan**  世界太狠（大变，hen），我用简语言（Jan） | **什么是？**  Jan语言，就是 “简语言”：  全新设计  类Python  强类型静态编译  面向万物互联IoT  大道至简，道法自然。  一种语言，多种平台。  包容既往，拥抱未来。  **我想说？**  I like c  I like c++  I like java  I like python  I like saving time  I like saving money  I like saving the world  I like simple, simple is beauty!  I like stupid, stupid is smart!  I like smart, smart is fast!  If you want to be richer  If you want to run faster  If you want to live longer  So why not to use Jan-Lang! |

**Jan语言程序设计**

(The world is hen, and I use Jan)

2021/6/10

V0.07

Authorized by Shaozhong.Xu

**目录**

[1. 前言 5](#_Toc6966)

[1.1. 计算机语言的发展 5](#_Toc14801)

[1.2. 拥抱新语言 6](#_Toc27947)

[1.3. 大道至简 7](#_Toc22543)

[1.4. Jan语言定义 7](#_Toc12602)

[1.4.1. Jan语言是什么？ 7](#_Toc7712)

[1.4.2. Jan语言的区别 8](#_Toc23659)

[1.4.3. 为什么还要Jan语言？ 8](#_Toc2047)

[2. 简单例子 10](#_Toc12696)

[3. 来自Jan的宣言 11](#_Toc20675)

[3.1. Jan语言的基石 11](#_Toc31887)

[3.2. Jan语言的准则 11](#_Toc1454)

[4. 数据类型和变量 12](#_Toc2938)

[4.1. 数据类型(bcdf-uis) 12](#_Toc648)

[4.2. 运算符 14](#_Toc60)

[4.3. 变量 14](#_Toc10748)

[4.4. @static 16](#_Toc8664)

[4.5. @const 17](#_Toc477)

[4.6. @volatile 17](#_Toc22067)

[4.7. 地址函数 18](#_Toc23153)

[5. 控制语句 18](#_Toc28324)

[5.1. 顺序语句 18](#_Toc8503)

[5.2. 循环语句 18](#_Toc1577)

[5.3. 判断语句 19](#_Toc17094)

[5.4. 分支语句 19](#_Toc26181)

[6. 数组 20](#_Toc23732)

[7. 结构体 21](#_Toc20435)

[8. 函数 23](#_Toc18343)

[8.1. 函数声明 23](#_Toc12103)

[8.2. 可变参数 24](#_Toc767)

[9. 类定义 24](#_Toc16448)

[9.1. 面向对象 24](#_Toc5815)

[9.2. 源文件 25](#_Toc593)

[9.3. 类(class) 26](#_Toc25972)

[9.4. @final 26](#_Toc12651)

[9.5. super 26](#_Toc30677)

[9.6. 接口(interface) 27](#_Toc18007)

[9.7. object类 27](#_Toc21380)

[10. string类（Draft） 28](#_Toc26641)

[11. 集合类（Draft） 28](#_Toc17310)

[11.1. Array 28](#_Toc15632)

[11.2. Matrix 29](#_Toc25947)

[11.3. Map 30](#_Toc17344)

[11.4. List 31](#_Toc9010)

[11.5. Set 31](#_Toc21419)

[12. 文件I/O（Draft） 32](#_Toc22977)

[13. 多线程并发编程（Draft） 32](#_Toc13617)

[14. 消息、事件、计时器（Draft） 33](#_Toc32059)

[14.1. 消息通信Message类 33](#_Toc27582)

[14.2. 事件标志EventFlag类 33](#_Toc2878)

[14.3. 计时器Timer类 34](#_Toc28375)

[15. 同步和异步处理（Draft） 35](#_Toc13356)

[15.1. Mutex对象 35](#_Toc12415)

[15.2. Semphore对象 35](#_Toc16514)

[15.3. synchronized同步 35](#_Toc952)

[16. Socket通信（Draft） 36](#_Toc6599)

[17. 注解导入（Draft） 36](#_Toc19510)

[17.1. @getset 37](#_Toc23555)

[17.2. @pack 38](#_Toc26748)

[17.3. @sync 38](#_Toc15330)

[17.4. 条件编译注解 38](#_Toc4594)

[18. 高级语言扩展（Draft） 39](#_Toc12008)

[18.1. 泛型和模板 39](#_Toc2755)

[18.1.1. 模板函数 39](#_Toc28266)

[18.1.2. 模板变量 39](#_Toc28539)

[18.2. 字节序问题 40](#_Toc29478)

[18.3. Property类 40](#_Toc29372)

[18.3.1. Property配置类 40](#_Toc13992)

[18.3.2. properties文件 41](#_Toc15241)

[18.4. Json处理类 41](#_Toc12521)

[18.5. 异常处理Exception类 42](#_Toc8948)

[18.6. 正则表达式Regex类 42](#_Toc31683)

[18.7. 魔法函数 42](#_Toc13363)

[18.8. 反射机制 43](#_Toc4917)

[18.9. 混合编程 43](#_Toc10859)

[18.10. 其他支持库 43](#_Toc8327)

[19. 发展路线图 44](#_Toc16810)

[19.1. 第一部曲: 创造 44](#_Toc28621)

[19.2. 第二部曲: 独立 44](#_Toc7405)

[19.3. 第三部曲: 扩展 45](#_Toc27236)

[20. 编译处理器 46](#_Toc20775)

[21. 开源协作 46](#_Toc1620)

[22. 后记 48](#_Toc26015)

[22.1. C/C++版本问题 48](#_Toc22970)

[22.2. Jan的哲学问题 49](#_Toc23586)

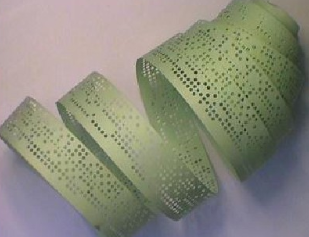
# 前言

历史是一面镜子，鉴古知今，学史明智。重视历史、研究历史、借鉴历史是中华民族5000多年文明史的一个优良传统。当代中国是历史中国的延续和发展[习近平 2019/1/3]。计算机语言的学习、研究和发展，也是同样道理。追踪溯源，承前启后，大道至简，返璞归真。

## 计算机语言的发展

计算机语言总的来说分为机器语言，汇编语言，高级语言三大类。而这三种语言也恰恰是计算机语言发展历史的三个阶段。

**第一阶段（机器语言，1946~1953）：**1946年2月14日，世界上第一台计算机ENAC诞生，使用的是最原始的穿孔卡片。光电阅读及将记录在穿孔卡片的数据读入计算机，计算的结果则利用打孔机在卡片上打出的一些小孔，表示输出的数据。这种卡片上的小孔就组成了只有计算机和专家才能理解的语言，也是完全用0，1代码写的程序，被称为机器语言。虽然后来发展的语言能让人类直接理解但最终送入计算机的还是这种机器语言。



**第二阶段（汇编语言）：**汇编语言用助记符代替了0，1代码，用地址符号或标号代替地址码。这样就用符号代替了机器语言的二进制码。汇编语言也称为符号语言。比起机器语言，汇编大大进步了。尽管还是复杂，用起来容易出错，但在计算机语言发展史上是机器语言向更高级的语言进化的桥梁。

section .data

msg db "hello,world!", 0xA

len equ $.msg;

section .text;

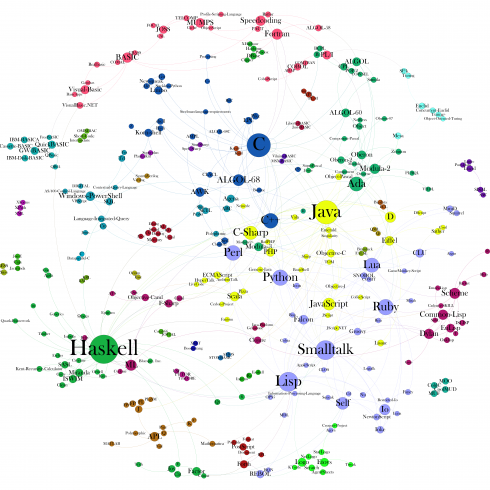
global \_start;

\_start: ;

**第三阶段（高级语言，1954~）：**高级语言是一种接近于人们使用习惯的程序设计语言，发展于20世纪50年代中叶到70年代，现在仍在不断发展，种类也相当多，比如 FORTRAN语言（第1个高级语言），COBOL语言，C语言（1972年，现代程序语言革命的起点），C++，C#， PASCAL，JAVA，PYTHON，GO等等。

**脉络1** 面向对象程序设计（Object-Oriented Programming, 简称OOP，1990s初~现在）是一种起源于60年代的Simula语言，自身理论已经十分完善，并被多种面向对象程序设计语言实现。

**脉络2** 编程语言发展图谱



**脉络3** 编程语言排行榜(TIOBE) （#没有特殊理由，就以正式开始设计Jan语言的时间点为准）



## 拥抱新语言

有几个事实和论断需要明确：

编程语言是在不断变化、演进和创新的；

好的编程语言就是好的软件开发利器，能提高生产效率；

排行榜靠前的语言，肯定有其可取之处，是值得大力关注的；

C/C++以及类C语言（借鉴C语言发展的，如Java，C#等）占有绝对优势的；

随着CASE时代（互联、自动驾驶、共享、电动化）的到来，随着ABCD时代（AI、Block Chain、Cloud、Big Data）的到来，随着AI技术的兴起，随着万物互联（物联网IoT）的发展，涌现出很多新的开发语言和开发技术。

Jan，应运而生。脱胎于C和C++，兼收并蓄Java、Python和Go语言的部分优点，成为一种新的极简开发语言。

## 大道至简

**大道至简 2015/6/24**

十年磨一剑，锋锐未曾开。

左砍浦江月，右劈金玉台。

一剑起风雨，天光云霭霭。

十剑一轮回，月在人徘徊。

天道何繁复，九曲渺云汉。

情理何深玄，世事须近猜。

回顾众生笑，明镜惹尘埃。

大道终须简，至性方开怀。

对于开发语言而言，同样应该是“大道终须简，至性方开怀”。

开发语言应该用简单的语法，用简单的道理，去阐述和构建 日益增加、日益复杂的 软件大厦。

## Jan语言定义

### Jan语言是什么？

**Jan语言，也叫“简语言”，它是一种**

**全新设计的、**

**类Python的、**

**强类型静态编译的、**

**面向万物互联IoT的、**

**简单、快速和高效的、**

**程序设计语言。**

第一位图灵奖获得者Alan J.Perlis说过，"A language that doesn't affect the way you think about programming, is not worth knowing"。Jan语言致力于去改变 程序设计的思维和工作特点，将更多的重点用于 设计，而将更多的细节和实现，留给AI和机器，从而带来思维、方法和效率上的变革。

### Jan语言的区别

**Q**：Jan是C/C++语言吗？

**A**：否。Jan吸收了C/C++/Java/Python/Go的优点。但是Jan有自己的语义、规则，是一门独立的程序设计语言。

**Q**：Jan是Python语言吗？

**A**：否。Jan是一门类似于Python、采用缩进格式书写的语言。它是一门强类型编译语言，不排除将来也可以像Python一样支持动态解释编程。至少在设计之初，它主要是面向IoT编程的，也可以是面向服务器端编程。

### 为什么还要Jan语言？

迄今为止世界上有600多种编程语言，下面是一部分有影响力的编程语言的发展史：

1951 – Regional Assembly Language

1952 – Autocode

1954 – IPL (LISP语言的祖先)

1955 – FLOW-MATIC (COBOL语言的祖先)

1957 – FORTRAN (第一个编译型语言)

1957 – COMTRAN (COBOL语言的祖先)

1958 – LISP

1958 – ALGOL 58

1959 – FACT (COBOL语言的祖先)

1959 – COBOL1959 – RPG1962 – APL

1962 – Simula

1962 – SNOBOL

1963 – CPL (C语言的祖先)

1964 – BASIC1964 – PL/I

1966 – JOSS

1967 – BCPL (C语言的祖先)

1968 – Logo

1969 – B (C语言的祖先)

1970 – Pascal

1970 – Forth

1972 – C1972 – Smalltalk

1972 – Prolog

1973 – ML

1975 – Scheme

1978 – SQL

1980 – C++ (既有类的C语言，更名于1983年7月)

1983 – Ada

1984 – Common Lisp

1984 – MATLAB

1985 – Eiffel

1986 – Objective-C

1986 – Erlang

1987 – Perl1988 – Tcl

1988 – Mathematica

1989 – FL

1990 – Haskell

1991 – Python

1991 – Visual Basic

1993 – Ruby

1993 – Lua

1994 – CLOS (ANSI Common Lisp的一部分)

1995 – Java

1995 – Delphi (Object Pascal)

1995 – Java

1995 – PHP

1996 – WebDNA

1997 – Rebol

1999 – D

2000 – Action

2001 – C#

2001 – Visual Basic .NET

2002 – F#

2003 – Groovy

2003 – Scala

2007 – Clojure

2009 – Go

2011 – Dart

那么，世界上有多少种人类语言呢？ 据统计，有7000多种。

无论是人类语言，还是计算机程序语言，都是百花齐放、争奇斗艳的，都是在不断发展和不断变化的。

任何语言，唯有紧跟时代发展，适应于需求，服务于需求，才能不被淘汰。

当前的时代，是一个VACU时代（易变性/Volatility, 模糊性/Ambiguity, 复杂性/Complexity, 不确定性/Uncertainty），是一个CASE时代（互联/Connected，自动驾驶/Autonomous，共享/Shared & Services，电动/Electric），是一个ABCD时代（人工智能/AI，区块链Block chain，云计算/Cloud Computing，大数据/Big Data），是物联网（IoT）蓬勃发展的时代。它在不断变化，不断变革，不断创新，不断尝试。

简洁、高效、快速，是当今时代的发展要求，也是软件开发的重要追求。Jan就是往这个简洁、快速、高效目标前进的一个重要尝试。

我们注意到：时隔多年以后，C/C++又王者归来，重登编程语言流行榜前列。不是想去鼓吹C/C++重于泰山，唯有C/C++才可以王霸天下，而是想说在当今局势下，毫无疑问C/C++又有了许多重要的用武之地。Jan语言脱胎于C/C++，具有简洁、快速、高效的特点，笃信Jan语言可以找到它的用武之地。

# 简单例子

用几个简单的Jan程序设计实例，来对Jan语言编程有个简单印象。

[后缀] 程序后缀是\*.jan

[注释] 可用 /\* \*/、//、/\*\* \*/ 进行程序注释。

[自动文档生成] /\*\* \*/中的内容会和Java程序一样，可用专门工具jandoc 自动生成文档。

[代码缩进] 代码写法和Python、Ruby相同，代码块需要缩进时，采用4个或者2个半角空格。

不建议用Tab键缩进。

**【例1】 Helloworld.jan**

main():

print(“Hello world, I'm Jan!\n”)

**【例2】Add.jan**

main():

input(“Please input two numbers:”, i4A, i4B)

c = Add(i4A, i4B)

print(“The result of ${a} + ${b} is ${c}”)

add(i4A, i4B)<int32>:

retrun i4A+i4B

**【例3】斐波那契数列递归算法 Fibo.jan**

fibo( i4N)<int32>:

if i4N==1 or i4N==2:

return 1

else:

return fibo(i4N-1) + fibo(i4N-2)

main():

input(n<int32>)

ret = fibo(n）

print(“result is ” , ret)

# 来自Jan的宣言

## Jan语言的基石

约定胜于文档

简洁，简洁，再简洁

包容现实，拥抱变化，迈向未来

靠近Python和Go

全面支持面向对象特性

全面采用AI强化，用机器生成代码

支持泛型、反射和类型指定

支持线程，信号量，并行编程

打通底层和顶层HMI编程通道 （？如何贯通三界，还在构思中...）

I don't know how to change the world, but I change myself first!

I don't know how to reach the perfect, but I try to grow a little!

I don't know what Jan can do for you, but I ask you to join it!

## Jan语言的准则

[少语法]

简洁胜于繁琐

明确优于灵活

变量函数自定义

语句控制逻辑要简单

[少注释]

函数内不要写注释

能用代码说清楚的，就不要写注释

如果代码中非得写注释的话，那就重构代码

[少晦涩]

命名方法用法的一致化

尽可能少的函数参数，推荐0~5个。

尽可能用简单的实现方法，不要用复杂的语法技巧。

不要用多重循环、多重判断等不好的代码风格。

[多智能]

支持注释标签（例如@static, @const, @volatile, @get, @set等）

能让编译程序做的事，就不要去写代码

要做的事情是说清楚what to do, 而将how to do 和 do it well留给机器

**第一部分 基础篇**

# 数据类型和变量

## 数据类型(bcdf-uis)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据类型 | 前缀 | 等价通用C语言 | 字节数 | 使用举例 | 补充说明 |
| （1）定长基本数据类型 | | | | | |
| bool | b | usigned char | 1 | bExist | 只能为TRUE或FALSE |
| char | c | unsigned char | 1 | cA = ''  cBuffer[100] | 单个ASCII字符, 0~255  char也相当于byte类型 |
| double | d | double | 8 | dWeight | double最大范围约为1.8e308 |
| float | f | float | 4 | fArea | float最大范围约为3.4e38  带小数点的数值类型默认为 float |
| uint8  uint16  uint32  uint64 | u1  u2  u4  u8 | unsigned char  unsigned short  unsigned long  unsigned long long | 1  2  4  8 | u1A  u2B  u4C  u8D | uint8 与 char 类型相同  u2B等价于 uint16 u2B, 其余类推。 |
| int8  int16  int32  int64 | i1  i2  i4  i8 | signed char  signed short  signed long  signed long long | 1  2  4  8 | i1A  i2B  i4C  i8D | int8 与 byte 类型相同  i2B等价于 int16 i2B，其余类推。 |
| （2）不定长组合数据类型 | | | | | |
| string | str | 无 | 不定长 | s<string>  strA  ss = "字符串" | utf-8字符串对象 |
| enum | 无 | enum | 不定长 | enum Abc<int32>:  a = 1,  b  enmA<Abc> | 同C用法，类型可定义为定长基本数据类型和string。  为了代码易读性，建议变量用前缀 enm |
| union | 无 | union | 不定长 | union Abc:  i1A  i4B  uniA<Abc> | 同C用法。  为了代码易读性，建议变量用前缀 uni |
| object | 无 | 无 | 不定长 | objA<object> | Jan语言的object对象基类  为了代码易读性，建议变量用前缀 obj |
| struct | 无 | struct | 不定长 | struct Rect:  i4Height  stA<Rect> | 只支持C的struct基本特性。  不支持成员函数.  成员为 定长基本数据类型.  为了代码易读性，建议变量用前缀 st |
| var | 无 | 无 | 不定长 | a  b<var> | 上述任意类型，变量缺省为 var 类型。 |

**常量数据前缀（小写）**

2进制常数使用 0b开头，不能是 0B。举例：0b11111011

8进制常数使用 0o开头，不能是 0O。 举例：0o12346734

16进制常数使用 0x开头，不能是 0X。举例：0x12ab4567

以0开头的整数串，如0123, 在Jan中当作 123处理，不会当作8进制处理，在编译时会给出警告。

以0x, 0b, 0o开头的常数串，数字中间可用 \_ 隔开。 例如：0x11111011 等同于 0x1111\_1011, 0x1\_11111\_11

**常量数据后缀（大写）**

F float 例如 1.2F

D double 例如 3.5D

W int16 例如 100W

L int32 例如 4L 不用小写 l 的原因在于，避免和数字1 或i 相混淆。

LL int64 例如 2LL

UL uint32 例如 100UL

ULL uint64 例如 2000ULL

**无后缀时的默认规则：**

1，整数后面无后缀 时，默认按照 int32处理。

例如 1000 等价于 1000L, 或者 (int32)1000

2，带小数点的 常数（如100, , 299,1，0.3，0.0）等，如无后缀指定的话，默认是 float类型。

**举例，内置类型常量**

enum JanType<int8>:

BOOL = 1

CHAR

DOUBLE

FLOAT

UINT8

UINT16

UINT32

UINT64

INT8

INT16

INT32

INT64

STRING

ENUM

UNION

STRUCT

OBJECT

VAR

## 运算符

支持C/C++运算符。

为减少语句的复杂性(譬如: a = i++和 a = ++i 的区别)，特意不支持运算符 ++ 和 --

增加逻辑运算符 and（等价于 &&），or (等价于 || )，not (等价于 !)。

## 变量

变量是用来存储用户数据的。Jan语言的每个变量，都有自己的类型，在使用它们之前，无须事先声明。系统会自动推导和决定每个变量的类型，并且按照需要进行初始化。如果担心系统推导会出错，那么在变量定义时就需要明确指定数据类型。

所有的变量，都以小写字母或下划线开头。变量的命名使用驼峰法。

举例： i4ReadBufSize、readBufSize、\_i4ReadBufSize、\_readBufSize 都是推荐的变量命名方法。

其中 下划线表示 私有变量。

1. 变量声明和初始化

**标准格式1：**[类型前缀]变量名: 初期值 **等价形式** [类型前缀]变量名 = 初期值

变量名必须为英文字符开头，且首字母必须大写

定义中有[:]或[=]且有初期值：表示先定义变量，再用初期值进行初期化。

定义中有[:]或[=]但无初期值：不符合语法。

定义中无[:]或[=]，表示只定义变量，但无需初期化。

举例

i4Width:10 语义为 int32 i4Width=10 (等价于i4Width=10)

i4Width: 错，冒号后面不能省略初期值。

i4Width 语义为 int32 i4Width

**标准格式2：**变量名<类型>:或= 初期值

语义为： 类型 变量名 = 初期値。

忽视初期値的类型，将变量名 强制定义为所指定的<类型>

如果<类型>和初期值的类型不一致，会给出编译警告或者错误。

**举例**

width<int16>: 100 语义为： int16 width = 100

**标准格式3：**变量名:或= 初期值 **等价形式** 变量名 = 初期值

凡是不满足 [类型前缀][A-Z]开头的变量名，默认都是 int32类型。

为了避免混淆，请不要单独使用 类型前缀字符 作为变量。

**举例**

f 语义为 var f

i2 语义为 var i2

u2 语义为 var u2

u2\_ 语义为 var u2\_

\_u2 语义为 var \_u2

如果给定了 初期値，则按照初期値进行 类型自动设定。

初期値有附加后缀时，以附加后缀的指定类型优先。

当变量名类型定义和初期値赋值类型之间存在不一致时，生成编译警告或者错误。

例如： i2A: 100LL 就不合乎规范， i2A表示int16, 100LL表示 int64， 两者的定义不一致

初期値为 其他变量名时， 自动定义类型为 与赋值变量相同

初期値为 浮点数，例如1.0, 1.0e-3, 2., 2.0F, 0.3 自动定义类型为 float

初期値为 双精度数，例如 1.0D 自动定义类型为 double

初期値为 字符串，例如 "aa1234", "Jan语言" 自动定义类型为 string

初期値为 xxL, 例如20L 自动定义类型为 int32

初期値为 xxLL, 例如 1000LL 自动定义类型为 int64

初期値为 整数或上述以外，例如 0，0x1234，-1L等 自动定义类型为 int32

举例

k:2.0 语义为 float k = 2.0

kk: 1.D 语义为 double kk = 1.

a: strA 语义为 string a = strA

变量类型优先级定义

|  |  |  |
| --- | --- | --- |
| 优先级（从高到低） | 定义格式 | 举例 |
| <类型> | 变量名<类型>: 初期值 | a<int8>:0 类型为int8 |
| [类型前缀] | [类型前缀]变量名: 初期值 | fSum:0 类型为 float |
| 初期値后缀,"",赋值变量 | 变量名: 初期値后缀，"",赋值变量 | a: 10LL 类型为 int64  a: 1.0 类型为 float  a="aa" 类型为 string  a=i2A 类型为 i2A的类型int16 |
| 上述以外 | 变量名 | 默认数值类型为 var  a = 0类型为 int32  b 类型为 int32 |

上述规则彼此矛盾时，由编译器给出编译错误或者警告。

举例

u2A<int16>= 1.0F

从强制定义上看，类型应该为: int16

从变量命名看，应该为uint16，

从赋値上看，应该为 float。

结论：按照优先级，u2A的类型 判定为 int16。但是会给出编译警告或者错误。

1. 批量变量定义

{a,b,c,d}<float> 等价于 a<float>, b<float>, c<float>, d<float>

1. 变量赋值和值交换

a, b = b, a

a, b, c = 1, 2, 3

a, b, c = c, a, b

为减少程序复杂性和难度性，特意不支持 连续赋值：如a = b = c = 100

1. 数据类型变换

变换方法： (基本类型)xx

举例

i2B = (int16)i4A

类型变换时，需要注意 变量类型的取值范围。譬如int32类型的变量0x12345678，强制类型变换为int16时，会舍去高位的0x1234，留下低位的 0x5678。

1. var变量使用

不能判定类型的变量都自动为 var变量。

变量可以被明确定义为 var类型。

例如 a<var>

可以对var变量赋值为 任何其他数据类型，反之则不行。

例如 a = 1200; a= “string” 正确

例如 u2A = a; t<string> = a 错误

## @static

静态变量注解@static用来定义静态变量或静态函数。

@static a<int16>: 0 等价于 static int16 a = 0

@static{

aaa: 0

bbb:100

}<int16>

等价写法：

@static:

aaa<int16>: 0

bbb<int16>:100

如非必要，请不要使用@static{}的使用方式，原因在于 @static: 方式更简洁。

对于其他注解（如@const, @volatile等），同样适用。

## @const

常量命名使用大写字母并用下划线分割。

举例：MAX\_SUITABLE\_SIZE

常量变量用@const来注解。

@const AAA<int8>: 1 等价于 const int8 AAA = 1

@const BBB<int8>: 100 等价于 const int8 BBB = 100

等价写法1：

@const{

AAA:1

BBB:100

}<int8>

等价写法2：

@const:

AAA<int8>:1

BBB<int8>:100

举例，常用内置常量

@const：

OK<int8> = 0 /\* OKAY, SUCCESS \*/

NG<int8> = -1 /\* NOT GOOD, FAILED \*/

NIL<int32> = 0 /\* NULL Memory Address \*/

NULL<int8> = 0 /\* Null, nothing \*/

## @volatile

动态变量使用场景较少，如果却有必要，如同C语言一样，可以直接使用。

碰到动态变量定义，编译器不对 变量进行编译优化。

例如

@volatile a<int16>:100 等价于 volatile int16 a = 100

## 地址函数

Jan语言废弃令人望而生畏的 C/C++指针，以及对指针的各种操作(例如 \*p++， p -> a，取地址符& 等)。

对于地址操作，采用专门的操作函数addr和mget。

获取变量的内存地址函数：addr(变量名)

获取内存地址中所保存的数值 mget(内存地址,字节数)。mget返回byte数组。

获取内存变量地址

u4Addr = addr( u4Width)

获取内存字节内容

cBytes[] = mget( u4Addr, 4)

内存转换辅助函数

mconv(cBytes[], type<JanType>)

mconv(u4Addr /\*内存地址\*/，type<JanType>)

使用举例 i2Data = mconv(cBytes, JanType.INT16)

# 控制语句

Jan语言中存在四种控制语句。

## 顺序语句

按照语句顺序，逐行执行。

例如：

a = 10

b = a \* 100 + 2

print(“%d”, b)

为避免逻辑混乱和增加程序自身的理解难度，Jan语言中禁止使用goto关键字。

可以单行内写多个独立语句，但是彼此间需要用分号”;”隔开。

a = 100; b = “adfa”

## 循环语句

**支持for循环**

for i in range(0, 100, 2): 语义等价于 for (i=0; i<2; i +=2)

for i in range(0,100) 语义等价于 for (i=0; i<2; i+=1)

for i in range(100) 语义等价于 for (i=0; i<2; i+=1)

for each\_item in items 语义类似于 for(i=0;i<len(items);i++){ each\_item= items[i]}

**支持while循环**

while i<9:

i = i + 2

do:

aa = bb

i = i + 2

loop while i<9

在循环体中，要退出和继续循环体时，可采用break和continue。

## 判断语句

if i<100:

xxx

elif i<200 and i>=100:

xxx

else:

xxx

为了处理逻辑的严谨性，不建议 省略掉else语句。

## 分支语句

switch cccc:

case range(0,10):

break

case 11:

break

case 22,24,100:

break

default:

break

为了处理逻辑的严谨性，不建议省掉[default]语句。

# 数组

数组中的所有元素类型都相同。数组必须用[]定义维数和元素个数。

**一维数组的定义：**

变量名[元素个数]<类型>

**二维数组的定义:**

变量名[元素个数, 元素个数]<类型>

三维及以上数组，可以用同样方法定义，为减少程序复杂性，不建议使用多维数组。

例

names[100]<string>

strNames[100]

names2[100][200]<string>

strNames2[100][200]

数组赋初值：

names[]<string>: {“a”,”bbb”,”cc”}

如果忽略括号中的数字，不设置数组大小，Jan语言编译器会根据初始值状况自动确定数组大小。

二维数组初始化

names[2][3]<string>:{ {“a”,”b”,”c”}, {“d”,”e”,”f”} } //按列优先原则赋值

元素分别为

names[0]

[0] [1] [2]

names[1]

[0] [1] [2]

赋值结果等价于：

names[0][0]=”a” names[0][1]=”b” names[0][2] =“c”

names[1][0]=”d” names[1][1]=”e” names[1][2] =”f”

多维数组也是同样处理,譬如a[2][3][4]<int32>, 可以进行降维处理。

元素分别为

a[0]

[0][0] [0][1] [0][2] [0][3]

[1][0] [1][1] [1][2] [1][3]

[2][0] [2][1] [2][2] [2][3]

a[1]

[0][0] [0][1] [0][2] [0][3]

[1][0] [1][1] [1][2] [1][3]

[2][0] [2][1] [2][2] [2][3]

赋值

a[2][3][4] = { { {1,2,3,4},{5,6,7,8},{9,10,11,12} },

{ {21,22,23,24},{25,26,27,28},{29,30,31,32} } }

a[2][3][4] = 0

数组元素的遍历方法：

for i in range(0,2):

for j in range(0,3):

for k in range(0,4):

a[i][j][k] = 0

for aa in a:

print(aa)

数组元素的线性化存储

例如names[2][3]，线性化存储顺序为：[0][0],[0][1],[0][2],[1][0],[1][1],[1][2]

共通函数

len() 返回数组元素个数。

举例： 对于数组names[100] 以及 names[2,3]而言，

len (names) 值为 100

len (names2) 值为6

切片

切片slice只是对现有数组的引用。 slice[开始位置: 结束位置]

特性：

新生成的切片长度： 结束位置 - 开始位置

新生成的切片元素 不包含结束位置所在的 数组元素

新生成的切片是对现有数组或者切片的引用

切片容量是只起始元素到底层元素中的最后一个元素的个数。

例

i4Numbers[]: {0,1,2,3,4,5}

i4Numbers2 = i4Numbers[2:4] // 切片，数值为： 2，3

len() 返回切片数组元素个数。例如len(i4Numbers2)值为2

# 结构体

Jan语言只支持C语言中的struct基本特性，不支持成员函数，成员变量必须为 定长字节类型.

譬如结构体中 不能包含 string , object, var 类型的成员。

**结构体定义**

struct ST\_Abcd:

c<int8>

u2<uint16> //变量名不好！不要用数据类型前缀做变量，其缺省类型为 int32

width **错误** //width没有匹配的数据类型前缀，缺省为不定长数据类型var, 所以错误。

i2Height //等价于i2Height<int16>

**字节对齐**

内存空间都是按照byte划分的，理论上对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定变量的时候经常在特定的内存地址访问，这就需要各类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐要求。也就是让宽度为2的基本数据类型（short等）都位于能被2整除的地址上，让宽度为4的基本数据类型（int等）都位于能被4整除的地址上，以此类推。反之，例如int32 a的地址是0x00fffff3，则其字节分布在0x00fffff3～0x00fffff6空间内，为了读取这个int，cpu必须对 0x00fffff0和0x00fffff4进行两次内存读取，并处理得出的中间结果，两次内存访问将会浪费大量的时间，因为内存访问的速度远小于CPU处理指令的速度。

不论类型是什么,对齐的边界一定是1,2,4,8,16,32,64....中的一个。

默认4字节对齐。可以用@pack(4)指定对齐方式。

等价于下述定义:

@pack(4)

struct ST\_Abcd:

c<int8>

\_<int8> // 匿名变量

u2<uint16> //u2是2字节，必须双字节地址位置上对齐

i2Height

\_<int8> // 匿名变量

\_<int8> // 匿名变量

结构体数值

数据类型是结构体对象时，可对成员变量进行个别赋值。

struct ST\_rect:

width<int32>

height<int32>

pos\_x<int32>

pos\_y<int32>

stRect<ST\_rect>: 0 //直接写0，表示需要对结构体进行初期化

语义等价为：

ST\_rect stRect;

memset(&stRect, 0, sizeof(stRect));

stRect<ST\_rect>: height=10, pos\_x = 100

语义为：

ST\_rect stRect;

stRect.height=10;

stRect.pos\_x=100;

stRect<ST\_rect>: 0, height=10, pos\_x = 100

语义为：

ST\_rect stRect;

memset(&stRect, 0, sizeof(stRect));

stRect.height=10;

stRect.pos\_x=100;

# 函数

函数的作用就是把相对独立的某个功能抽象出来，使之成为一个独立的实体。

## 函数声明

1. 函数声明

**标准格式：**函数名(参数)<返回类型>:

当无返回类型时，表示函数无返回值。

推荐函数首字母为小写（例如add，compute等）。

所有的函数（魔法函数除外），都以 小写字母或者下划线开头，使大驼峰CaseCamel命名规则（如getYear()、getCustomerId() ）。

只要是函数，无论是使用还是声明，后面都需要带()

1. 函数作用域(public和private)

函数默认作用域是 public

如果不想被外部文件使用的话，函数名前加单个下划线。

例如

add(i4A, i4B)<int32>: public函数，等价于C函数 extern int32 add(int32 i4A, int32 i4B)

\_sub(i4A, i4B) <int32>: private函数，类似于C函数 local/static int32 \_sub(int32 i4A, int32 i4B)

1. 函数参数个数

函数参数个数推荐为 0~5个。

参数越多，函数则变得越复杂，代码和逻辑就越难理解。

解决参数过多的通用变成技巧：

传入结构体（结构体中可以包含多个参数信息）

函数进行功能上再拆分

1. 函数返回值

如果函数不需要返回值时，不推荐写return。如果写了的话，相当于是一个空语句，无任何作用。

举例：

example()：

print(“I’m Jan!\n”)

空函数：

example():

pass //表示空语句，什么也不做。

与C和C++不同，函数返回值可以有多个数值，例：

getData()<int32, int16, int16>:

return 10, 2, 100 //不需要用()或者{}括起来。

合法使用方法：

a, \_, b = returnData()

a, \_, \_ = returnData()

a = returnData()

\_, \_, a = returnData()

其中，\_为匿名变量，不用声明就可以直接使用。

## 可变参数

Jan语言函数支持可变参数。

函数声明：函数名称(固定参数，可变参数...)<返回类型>

例如：

func(i4Params...)<float>: // 根据变量命名规则

sum<float> = 0

for param in params:

sum += (float) param

return sum

可变参数的类型由可变参数变量名来决定，也可以显式定义变量类型。

例如：

func(params<int32>...)<int32>

Jan语言的宗旨是至简，但是不等于 无条件简化。

首先得把事情讲清楚，语义上不含糊，然后才是简化。

**第二部分 高级篇**

# 类定义

## 面向对象

面向对象三大特征是：封装、继承和多态。

**封装**是指将某事物的属性和行为包装到对象中，这个对象只对外公布需要公开的属性和行为，而这个公布也是可以有选择性地公布给其它对象。

**继承**是子对象可以继承父对象的属性和行为，亦即父对象拥有的属性和行为，其子对象也就拥有了这些属性和行为。Jan语言中默认对所有父类进行public继承。

譬如class A extends B, C的类定义语句，其等价于 class A extends public B, public C

**多态性**是指允许不同类的对象对同一消息作出响应。比如同样的加法，把两个时间加在一起和把两个整数加在一起肯定完全不同。又比如，同样的选择编辑-粘贴操作，在字处理程序和绘图程序中有不同的效果。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好地解决了应用程序函数同名问题。

Jan语言是面向对象的程序设计语言，支持面向对象的三大特性。

## 源文件

Jan语言文件命名为 Axxx.jan。为了和C/C++文件相区别。

命名规则：

1）文件名全部为英文字母和数字组成，不能包含下划线和中划线，首字符不能为数字。

2）首字母为大写，其余为小写，也可以采用CaseCamel模式，如AaaBbb.jan。

3）文件名后缀为 [.jan]

例如： Book.jan

(2)、包定义：

每一个Jan源文件的第一行都需要声明包的名称，包如同类库或者命名空间的概念，每个Jan文件都属于且仅属于一个包。代码包的名称一般与源码文件所在的目录同名，如果不同名，在构建和安装过程中 会以代码包的名称为准，包名全部小写。

**包的定义** package aa.mytest

**导入包** import 包名 例如： import aa.mytest

import 包名 as 别名 例如： import aa.mytest as tt

import 类，接口，变量 from 包名

(2)、Jan文件格式：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | 可否省略 | 目的 | 语法 | 举例 |
| 1 | Must | package定义 | package 包名 | package aa.bb.cc |
| 2 | Option | 引入需要使用的类 | import 类或包 | import aa.bb.cc.Book  import aa.bb.cc.\*  import aa.bb.cc.Book as bk  import {ee, ff, kk} from aa.bb.cc |
| 3 | Option | 定义包变量 | 变量名<类型>:=初期值 | i4Aaaa |
| 4 | Option | 定义包函数 | 函数名(...)<类型> | Func()<int32> |
| 5 | Option | 定义类名/接口名 | class 类名 extends 父类或接口  interface 接口名 | class Aaa extend B, C, IInterface |
| 6 | 依存于5 | 定义类成员 | 见下节。 | 见下节。 |
| 7 | 依存于5 | 定义类函数 | 见下节。 | 见下节。 |

## 类(class)

类定义的方法： class类名 extends 类名A, 类名B，接口C

包名和类名不能省略，如果没有父类或者接口的话，extends子句可以省略。

与C++不同，Jan语言中，无需指定public或者protected，所继承的父类都是 public方式的。

举例： class Aaa extends B, C：

width<int32>

height<int32>

calcArea()<int32>:

return width \* height

一个Jan文件最多存在0个或1个class。class名与Jan文件名相同。

1. 成员变量

非单下划线开头的成员变量都是保护变量(protected)，能够被子类继承和访问。

以单下划线开头的成员变量都是私有变量(private)，不能被子类继承和访问。

1. 成员函数

非单下划线开头的成员变量都是保护变量(protected)，能够被子类继承和访问。

以单下划线开头的成员变量都是私有变量(private)，不能被子类继承和访问。

1. 虚函数

所有的非静态成员函数，默认都是虚函数，都可以在子类中进行重载，不需要特别声明和指定。

利用@final对成员函数进行注解，表示该成员函数不可以在子类中进行重载。

1. 静态变量和静态函数

利用@static对成员变量进行注解，表示该成员变量是静态成员变量。

利用@static对成员函数进行注解，表示该成员函数是静态成员函数。

## @final

@final修饰类：

该类不可继承

@final修饰方法：

表示该虚函数（成员函数）不能被子类覆盖，也就是不能被重载。

@final修饰变量：

在Java中，final修饰的变量，其属性值第一次初始化后不能被修改.

在Jan语言中，禁止用 final 修饰变量，该修饰final被忽视，不起作用。

## super

用super<父类名>来访问父类的成员函数。

如果省略掉<父类名>时，super从直接的上层父类开始，往顶层查找，直到找到为止。

举例

class A:

calcTax()<int32>:

...

calcArea()<int32>:

class B extends A:

CalcTax()<int32>:

class C extends B:

CalcTax()<int32>:

i4A = super<A>.calcTax()

i4B = super<B>.calcTax() 等价于 i4B = super.CalcTax()

i4Area = super.calcArea() 等价于 i4Area = super<A>.CalcArea() 或i4Area = CalcArea()

return i4A + i4B + i4Area\*0.2

## 接口(interface)

Jan语言设置接口类和interface关键字。

接口类 使用 interface 来定义。

接口定义：

interface IReader extends 接口A，接口B:

函数

接口的用法

接口中不能定义变量

接口可以继承

为了方便易懂，接口命名遵循下述规则：

（1）首字母必须为 大写I

（2）接口的实现类，要去掉首字母I，末尾加 Impl (也就是Implement的缩写)

例如：IReader，IUserLogin等为接口类

例如：ReaderImpl, UserLoginImpl为接口实现类

## object类

# string类（Draft）

# 集合类（Draft）

为了减少标准程序库的复杂度，所有的容器类都不保证线程安全性。

## Array

Array是一维数组容器。数组中的所有元素类型都相同。

a<Array, i4\_t> 等价于 Array<i4\_t> a; 里面的元素类型为i4\_t

a<Array<i4\_t>> 与 a<Array, i4\_t>等价，为了避免多重<<>>，建议使用 a<Array, i4\_t>这样的模式。

Array的成员函数：

append 在末尾添加元素

remove 移除对应元素

add() 末尾添加对应元素

subtract() 移除对应元素

multiply() 对应元素相乘

divide() 对应元素相除

getAt() 获取指定位置的元素

setAt() 设置指定位置的元素

removeAt() 删除指定位置的元素

[] 同getAt或者setAt

fill 填充数值

count 元素个数

copy 拷贝函数

Array类也支持切片操作

Array可以进行 相关数组运算。

例：

a<Array, int32>: {1,2,3,4,5}

b<Array, int32>: {11,12,13,14,15}

c<Array, int32>:

Append(

c = a.remove(b) // c的结果为：{ 12,14,16,18,20 }

c = a.remove(b) // c的结果为：{ -10, -10, -10, -10, -10 }

c = a.multiply(b) // c的结果为：{ 11, 24, 39, 56, 75 }，对应位置点乘

c = a.divide(b） // c的结果为：{ 0, 0, 0, 0, 0 }，同C语言一致，只取整数。

//int(0.49) = 0, int(0.99)=0

计算的前提条件： a 和 b的长度必须一致。

数组也可以和单个变量进行运算。

d<Array, int32> = a + b // d的结果为：{101,102,103,104,105}

其余与此类推。

Array是轻量级别的一维数组、用一维数组存储同类型元素、计算速度快；

## Matrix

Maxtrix可以看成是多维多列的数组。一维多列矩阵就相当于是Array。

矩阵缺省为DenseMatrix。

矩阵叉乘

矩阵的乘法就是矩阵a的第一行乘以矩阵b的第一列，各个元素对应相乘然后求和作为第一元素的值。

矩阵只有当左边矩阵的列数等于右边矩阵的行数时,它们才可以相乘,乘积矩阵的行数等于左边矩阵的行数,乘积矩阵的列数等于右边矩阵的列数 。

a =

1 0 2

-1 3 1

b =

3 1

2 1

1 0

c =

5 1

4 2

矩阵的点乘

就是矩阵各个对应元素相乘, 这个时候要求两个矩阵必须同样大小

a =

1 0

-1 3

b =

3 1

2 1

c = 3 0

-2 3

Matrix的成员函数

new 创建矩阵（由数组，指定类型等创建）

= 矩阵赋值或初期化

+ 矩阵相加（对应元素相加）

- 矩阵相减（对应元素相减）

\* 矩阵叉乘 mxn和nxk的叉乘结果为 mxk

/ 矩阵相除（对应元素相除）

[]或 取矩阵元素

rotate 矩阵旋转

transpose 矩阵转置

invert 矩阵求逆

identify 单位矩阵

multiply 矩阵点乘（对应元素相减）

shape 矩阵取维，例如返回<2,3>, 说明矩阵是2行3列

reshape 矩阵变维

copy 拷贝函数

fill 填充数值

矩阵支持切片操作

## SparseMatrix

## Map

缺省为HashMap，按照Hash算法保存<key, value>对象。

映射Map是一种无序的键值对的集合，Map的键类似于索引，指向数据的值。

Map用于存放 有关联关系的数据。

empty() 清除所有元素

delete() 删除对象

isEmpty() 如果不为空，则返回 TRUE。

add(key, value) 增加元素

remove(key=aa, value=bb) 删除元素

merge 合并

subtract 相减

contains(key=aa, value=bb) 如果包含元素，则返回 TRUE。

[key=aa] 返回指定元素

count() 返回元素个数

　　keys() 返回键的数组

values() 返回值的数组

EntrySet() 返回键和值的Set集合。

例：

countries<Map, string, string>: {"中国":"China", "美国":"America"}

## List

可以看做是保存<key, value>对象的数组。

与Map不同的是，每个元素有 对应的 index。

new() 生成列表

empty() 清除所有元素

isEmpty() 如果不为空，则返回 TRUE。

= 赋值或者初始化

add(index, key, value) 增加元素

remove(index=cc , key=aa, value=bb) 删除元素

merge() 合并

subtract() 相减

revert() 反转

sort() 从小到大排序

move(index, key, steps) 列表项前后移动

contains(index=cc, key=aa, value=bb) 如果包含元素，则返回 TRUE。

[index=cc, key=aa] 返回指定元素(一个或多个)

count() 返回元素个数

　　keys() 返回键的数组

values() 返回值的数组

EntrySet() 返回键和值的Set集合

unique() 去除key重复元素

## Set

Set就是元素集合，例如(2,3,5,100), 其中的元素可以是对象，也可以是数据。

可以看做是 无重复数据的 简单列表

new() 生成集合

empty() 清除所有元素

isEmpty() 如果不为空，则返回 TRUE。

= 赋值或者初始化

add(index, key, value) 增加元素

remove(index=cc , key=aa, value=bb) 删除元素

subtract() 相减

revert() 反转

sort() 从小到大排序

contains(index=cc, key=aa, value=bb) 如果包含元素，则返回 TRUE。

[index=cc, key=aa] 返回指定元素

count() 返回元素个数

　　keys() 返回键的数组

values() 返回值的数组

EntrySet() 返回键和值的Set集合。

intersection() 交集

difference() 差集

merge() 并集

## Tree

## BinaryTree

# 文件I/O（Draft）

文件对象类File来处理 I/O操作。

File成员函数

Open(strFileName, mode)

Close()

ReadBuf()

WriteBuf()

ReadLine()

WriteLine()

SeekTo()

# 多线程并发编程（Draft）

线程类Thread

成员函数

Create 创建

Self 获取线程自身对象

Start 开始

Pause 暂停

Resume 继续执行

Sleep 睡眠

Stop 停止

SetPriority 设置或更改线程优先级

Destroy 销毁

GetStatus 获取线程状态(Running, )

# 消息、事件、计时器（Draft）

## 消息通信Message类

消息通信Message类，实现消息的接收和发送机制。

\_pSendBuf

\_pReceiveBuf

maxSendCnt

maxReceiveCnt

SetSendBuf() 准备发送消息区

SetRecieveBuf() 准备接收消息区

Init() 初始化

成员函数

Create 创建

Clear 清空所有接收和发送消息

CanSend 判断可否发送

Send 发送消息

CanReceive 判断可否接收

Receive 接收消息

## 事件标志EventFlag类

事件标志用于监视 事件变量（u4\_t）的比特位，从而执行所对应的处理函数。一个事件标志类的实例，最多对应32个处理函数。

EventFlag类

成员变量

eventFlag<u4\_t> event\_flag的设定模式

eventList<List> 保存事件比特位和处理函数

成员函数

Create

Clear 清除所有的eventFlag比特位以及所有的处理函数

SetFlag

ClearFlag

CheckFlag

WaitFlag

## 计时器Timer类

执行计时器功能，一个Timer类由一个线程来执行和管理。

如要多个线程来分别管理Timeout函数的话，就生成多个Timer类。

Timer类

成员变量

minSpan // 计时器所能允许的最小时间间隔，譬如50ms

成员函数：

Create

Start

Pause

Resume

Stop

SetTimeout

RemoveTimeout

# 同步和异步处理（Draft）

## Mutex对象

Mutex本质上说就是一把锁，提供对资源的独占访问，所以Mutex主要的作用是用于互斥。Mutex对象的值，只有0和1两个值。这两个值也分别代表了Mutex的两种状态。值为0, 表示锁定状态，当前对象被锁定，用户进程/线程如果试图Lock临界资源，则进入排队等待；值为1，表示空闲状态，当前对象为空闲，用户进程/线程可以Lock临界资源，之后Mutex值减1变为0。

成员函数：

Create 创建

TryLock 判断是否可以加锁

Lock 加锁，进入互斥状态

Unlock 解锁，离开互斥状态

Destroy 销毁

## Semphore对象

也叫信号灯，是在多线程环境下使用的一种设施, 它负责协调各个线程, 以保证它们能够正确、合理的使用公共资源。Jan语言中的信号量通过一个计数器控制对共享资源的访问，信号量的值是一个非负整数，所有通过它的线程都会将该整数减一。如果计数器大于0，则访问被允许，计数器减1；如果为0，则访问被禁止，所有试图通过它的线程都将处于等待状态。

计数器计算的结果是允许访问共享资源的通行证。因此，为了访问共享资源，线程必须从信号量得到通行证，如果该信号量的计数大于0，则此线程获得一个通行证，这将导致信号量的计数递减，否则，此线程将阻塞直到获得一个通行证为止。当此线程不再需要访问共享资源时，它释放该通行证，这导致信号量的计数递增，如果另一个线程等待通行证，则那个线程将在那时获得通行证。

成员函数：

Create 创建

Lock 获取信号量

TryWait 判断是否可以获取信号量

Release 释放信号量

Destroy 销毁

## synchronized同步

对于函数同步，采用注解定义@synchronized

使用实例：

@synchronized

Add(a, b)<i4\_t>:

xxx

return xx

synchronized机制和Java相同。由Jan语言内部编译时保证，同一个函数，只能同时由一个线程执行。自动创建隐式锁，默认锁名为文件名或者类名。

# Socket通信（Draft）

Socket通信分服务器端SocketServer类和客户端SocketClient类。

基础Socket类

Create() 创建一个Socket（基于TCP/IP）

Send() 发送消息

Recieve() 接收消息

Close() 关闭

服务器端 SocketServer类

class SocketServer extends Socket

成员函数

Bind(ip<string>, port<i4\_t>) 绑定Socket

Listen() 限定监听队列长度

Accept() 接收连接并返回一个新的Socket

客户端SocketClient类

class SocketClient extends Socket

成员函数

ClientConnect() 初始化与另一个Socket的连接

在使用时，需要同时生成服务器端SocketServer实例和客户端SocketClient实例。

例如:

socketServer<SocketServer>

socketClient<SocketClient>

# 注解导入（Draft）

Jan语言使用几种@注解导入功能。

如果@注解与潜在规则冲突时，@注解优先，因为@注解是显式的强制定义。

## @getset

在成员变量定义时，可以加上Get/Set注解。

@get 自动生成Get函数

@set 自动生成Set函数

@getset 自动生成Get/Set函数

例如成员变量名为 minSpan<int32>

Get函数自动为：GetMinSpan()<int32>,

Set函数自动为：SetMinSpan(minSpan<int32>)

## @pack

**字节对齐注解用@pack**

用于指示结构体的字节对齐方式。在定义前加上@pack(4)等价于[#paragma pack(4)]

结构体缺省使用 @pack(4)，尽管在效率上可能会有损失，但是不用担心字节对齐方式。

## @sync

函数同步注解用@sync

作用与Java语言的synchronized相同。

为了避免复杂化，只支持函数级别的同步处理。也就是类或者函数内部 不能使用 同步注解。

## 条件编译注解

**@if 条件表达式**

**@elif 条件表达式**

使用方法如同使用 if和elif条件语句。

**@else**

必须与@if配套使用

使用方法如同使用 else条件语句。

使用举例：

@if xxxxx:

Afadfaf

@elif xxxx:

dfafa

@else：

Dfafdafa

配套函数：

defined(xxx) 用于确定Makefile文件中是否定义了 xxx，如果被定义了，则返回true，否则返回false。

xxx 可以直接在 @if 条件表达式中使用。

例如：@if defined(AAA)

@if AAA == 2

**复杂编译条件**

**条件或：**

@if 条件1 or 条件2

@if 条件1 || 条件2

**条件与：**

@if 条件1 and 条件2

@if 条件1 && 条件2

# 高级语言扩展（Draft）

## 泛型和模板

泛型本质是指类型参数化，意思是允许在定义类、接口、方法时使用类型形参，当使用时指定具体类型，所有使用该泛型参数的地方都被统一化，保证类型一致。

### 模板函数

模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码。

C++中的模板函数例：

template <typename T>

T const& Max (T const& a, T const& b) { return a < b ? b:a; }

Jan语言定义：

template Max(a<T>, b<T>)<T>:

return a < b ? b : a

函数调用： c<int16> = <int16>Max( a, b)

简单而言，函数前面用template定义模板类型

对于多个模板类型而言，可以使用同样的方法进行定义。

例如

template Func(a<T1>, b<T2>)<T3>:

函数调用： c = Func(a<int16>, b<float>)<double>

为避免编程过于复杂，Jan语言中不支持 模板类，只支持 模板函数（包括全局函数和类成员函数）。

### 模板变量

字符串是由双引号 " "或者单引号' '配对使用来定义的。成对单引号中可以包含单个双引号，成对双引号中可以包含单个单引号。单引号和双引号的转义字符分别为 \" 和 \'，字符串中可以包含任意个转义字符。

例如：

"a string"，'a string'， "It's a good idea", 'aa " afafa ', " \' \" "等都是合法字符串。

在字符串中可以使用模板变量${}。此处的模板是变量替换的意思。

a = 10

b = 10

c = a+ b

print(" ${a} + ${b} is ${c}")

## 字节序问题

big endian是指低地址存放最高有效字节（MSB），而little endian则是低地址存放最低有效字节（LSB）。

字节序和CPU有关。Motorola的PowerPC系列CPU采用big endian方式存储数据，而Intel的x86系列CPU采用little endian方式存储数据。

比如数字0x12345678在两种不同字节序CPU中的存储顺序如下所示：

Big Endian（MSB）

低地址 高地址

----------------------------------------------------------------------------->

| 12 | 34 | 56 | 78 |

Little Endian（LSB）

低地址 高地址

----------------------------------------------------------------------------->

| 78 | 56 | 34 | 12 |

采用big endian方式存储数据是符合我们人类的思维习惯的。所有网络协议也都是采用big endian的方式来传输数据的，所以有时我们也会把big endian方式称之为网络字节序。JAVA编写的程序则唯一采用big endian方式来存储数据。

Jan语言默认采用big endian模式。

## Property类

尽管Property类和Map类之间无任何继承关系，但是，可以把Property类看作是特殊的映射Map，它的键和值都是String文字串类型。可以用与Map类同样的函数方法对Property类进行操作。

### Property配置类

Create 生成对象

Load 从输入流加载

Store 写入到输出流

//以下与Map操作相同

Clear 清除所有

IsEmpty() 如果不为空，则返回 TRUE。

Add(key, value) 增加元素

Remove(key=aa, value=bb) 删除元素

Merge 合并

Subtract 相减

Contains(key=aa, value=bb) 如果包含元素，则返回 TRUE。

[key=aa] 返回指定元素

Count() 返回元素个数

　　Keys() 返回键的数组

Values() 返回值的数组

EntrySet() 返回键和值的集合

### properties文件

它是一种配置文件，主要用于表达配置信息，文件类型为\*.properties，格式为非xml文本文件，文件的内容是格式是"键=值"的格式，在properties中可以用"#"来作注释。

Property文件示例：

# 这是一个properties文件示例

// 这是注释行

/\* 这也是注释行 \*/

server.ip = 192.168.1.100

server.port = 8000

为了和Java语言保持一致，建议注释行 采用"#"开头。

## Json处理类

JSON（JavaScript Object Notation, JS对象标记）是一种轻量级的数据交换格式。JSON最初是属于JavaScript的一部分，后来由于其良好的可读性和便于快速编写的特性，现在已独立于语言，被众多语言支持和使用。特别是对于网络编程而言，JSON的重要性不言而喻。

**JSON中的键**都是字符串形式，值可以取任意类型。它有3种结构：

值为字符串或者数组类型： {"name":"Jan","age":1}

值为JSON数组： [{"name":"Jan","age":1}, {"name2":"Jan2","age2":2}]

值为对象类型：{"name":"Jan", "description":{"age":0,"sex":"male" }}

{}用于描述一组“不同类型的无序键值对集合”，[]描述一组“相同类型的有序数据集合”

**Json对象类：**

Create 生成，可以由Map、List和Json字符串生成

Add(index=cc, key=aa, value=bb) 增加

Remove(key=aa, value=bb) 删除

GetList 获取Json内部列表

Store 输出Json

[index, key] 取对应元素

## 异常处理Exception类

Jan语言只有一种基本的异常类，按照需要可以自己扩展。

抛出异常： throw Exception(数字编号, "异常内容")

例：

try:

if x>100:

throw Exception(1, "x is too big")

elif x>50:

throw Exception(2, "x is in 51~100")

else

throw Exception(3. "x is too small")

catch e<Exception>:

.....

finally:

print("Finally！")

## 正则表达式Regex类

为了简化String类的设计，以及更好地对正则表达式进行支持和功能扩展，将正则表达式功能独立出来，由专门的正则表达式类Regex来处理。

成员函数：

CheckMatch(strSrc, strReg)<b\_t>

GetMatches(strSrc, strPattern)<string[]>

ReplaceAll(strSrc, strReg, strReplace)<string>

## 魔法函数

魔法函数的形式： \_\_xx\_\_()

\_\_init\_\_() 用于定义变量初始化

\_\_destroy\_\_() 用于执行销毁变量

\_\_class\_\_():<string> 用于返回程序类 // （Optional）将来用于实现Reflection机制

如果存在\_\_init\_\_()函数时，在Jan模块装载或者类实例化时，会自动调用\_\_init\_\_()函数执行初始化。

如果存在\_\_destroy\_\_()函数时，在Jan模块卸载或者类实例销毁时，会自动调用\_\_destroy\_\_()函数。

其他魔法函数还在设计中，今后按照需要再增加。

## 反射机制

尽管很多现代语言（例如Java，Go，C#）都支持反射机制Reflection，但是C++语言本身也不支持反射机制。在Jan初期版本中，出于性能和编程简单性考虑，暂不支持反射机制。

## 混合编程

可以在Jan程序中，插入其他语言程序代码块（一般情况下不推荐使用）。

//c{ //c}

//c++{ //c++}

//asm{ //asm}

其他语言也可以用类似方法进行代码插入(暂时只支持C/C++和汇编)。

## 其他支持库

对于Jan语言而言，还有很多基础的支持库需要构建，在后续版本中进行完善和补充。

譬如 处理事件日期的 TimeDate类，处理字符串的String类，处理网络URL的Url类，处理文件目录的辅助方法，支持输入输出的I/O支持类、自动测试功能、Logger输出功能等。

print(string)

a\*\*b b可以为任意实数

max(ar[]<T>)<T>

min(ar[]<T>)<T>

mean(ar[]<T>)<d\_t>

sort(ar[]<T>)

reverse(ar[]<T>)

**第三部分 实施篇**

# 发展路线图

关于Jan语言的发展，初步考虑分为三个阶段。

## 第一部曲: 创造

需要重新考虑，为此删除原先内容后，先发布。

## 第二部曲: 独立

需要重新考虑，为此删除原先内容后，先发布。

## 第三部曲: 扩展

需要重新考虑，为此删除原先内容后，先发布。

# 编译处理器

需要重新考虑，为此删除原先内容后，先发布。

# 开源协作

Shaozhong.Xu拥有Jan语言著作权。

Jan语言项目是彻底的开源项目, 包含Jan语言在内，完全遵循LGPLv3协议，可以自由修改，自由使用，无须付费。

# 后记

## C/C++版本问题

1. **C语言版本更迭**

| 年份 | C标准 | 通用名 | 别名 | 标准编译选项 | GNU扩展选项 |
| --- | --- | --- | --- | --- | --- |
| 1972 | Birth C | - | - | - | - |
| 1978 | K&R C | - | - | - | - |
| **1989-1990** | **X3.159-1989, ISO/IEC 9899:1990** | **C89** | **C90, ANSI C, ISO C** | **-ansi, -std=c90, -std=iso9899:1990** | **-std=gnu90** |
| 1995 | ISO/IEC 9899/AMD1:1995 | AMD1 | C94, C95 | -std=iso9899:199409 | - |
| 1999 | ISO/IEC 9899:1999 | C99 | - | -std=c99, -std=iso9899:1999 | -std=gnu99 |
| 2011 | ISO/IEC 9899:2011 | C11 | - | -std=c11, -std=iso9899:2011 | -std=gnu11 |
| 2018 | ISO/IEC 9899:2018 | C18 | - | -std=c18, -std=iso9899:2018 | -std=gnu18 |

1. **C++版本更迭**

| **年份** | **C++标准** | **通用名** | **别名** | **标准编译选项** | **GNU扩展选项** |
| --- | --- | --- | --- | --- | --- |
| 1978 | C with Classes | - | - | - | - |
| **1998** | **ISO/IEC 14882:1998** | **C++98** | **-** | **-std=c++98** | **-std=gnu++98** |
| 2003 | ISO/IEC 14882:2003 | C++03 | - | -std=c++03 | -std=gnu++03 |
| 2011 | ISO/IEC 14882:2011 | C++11 | C++0x | std=c++11, std=c++0x | std=gnu++11, std=gnu++0x |
| 2014 | ISO/IEC 14882:2014 | C++14 | C++1y | std=c++14, std=c++1y | std=gnu++14, std=gnu++1y |
| 2017 | ISO/IEC 14882:2017 | C++17 | C++1z | std=c++17, std=c++1z | std=gnu++17, std=gnu++1z |
| 2020 | to be determined | C++20 | C++2a | -std=c++2a | std=gnu++2a |

1. **与Jan语言的关系**

Jan语言可以编译生成C/C++代码，优先采用广泛使用的 C89和C++98标准。

## Jan的哲学问题

1. 大道至简
2. 简单就是美
3. 简洁与明确

譬如Add(a,b)<i4\_t>，那个<i4\_t>是不是可以省略呢？

编译程序能够自动推导出 i4\_t类型，是否有必要写上<i4\_t>？

写上<i4\_t>，更明确一些；省略掉<i4\_t>，更简洁一些。

生活没有对错之分，区别在于取舍。

Jan语言重视简洁，但是在简洁和明确发生冲突时，宁愿多要一点明确。

1. 加法与减法

优先采用减法，使功能和语法达到最简单。

在利用Jan语言开发的时候，根据实际需要，再做扩展(少量加法)。

1. 人与机器

人与机器的定位

AI的运用