|  |  |
| --- | --- |
| **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)

2023/6/8

V0.70

Created by Shaozhong.Xu

**目录**

[1. 前言 6](#_Toc137155321)

[1.1. Jan语言定义 6](#_Toc137155322)

[1.1.1. Jan语言是什么？ 6](#_Toc137155323)

[1.1.2. Jan语言的区别 7](#_Toc137155324)

[1.1.3. 为什么还要Jan语言？ 7](#_Toc137155325)

[2. 简单例子 8](#_Toc137155326)

[3. 来自Jan的宣言 9](#_Toc137155327)

[3.1. Jan语言的基石 9](#_Toc137155328)

[3.2. Jan语言的准则 9](#_Toc137155329)

[3.3. 通用命名规则 10](#_Toc137155330)

[4. 先谈点哲学 11](#_Toc137155331)

[4.1. 计算机语言和哲学之间的关系是什么？ 11](#_Toc137155332)

[5. 数据类型和变量 13](#_Toc137155333)

[5.1. 数据类型(bcdf-ui) 13](#_Toc137155334)

[5.2. 运算符 14](#_Toc137155335)

[5.3. 变量 14](#_Toc137155336)

[6. 控制语句 17](#_Toc137155337)

[6.1. 顺序语句 17](#_Toc137155338)

[6.2. 循环语句 17](#_Toc137155339)

[6.3. 判断语句 17](#_Toc137155340)

[6.4. 分支语句 18](#_Toc137155341)

[7. 一维数组 19](#_Toc137155342)

[8. 结构体 20](#_Toc137155343)

[9. 函数 21](#_Toc137155344)

[9.1. 函数声明 21](#_Toc137155345)

[9.2. 可变参数 22](#_Toc137155346)

[10. 注解编程 23](#_Toc137155347)

[10.1. @static 23](#_Toc137155348)

[10.2. @const 23](#_Toc137155349)

[10.3. @volatile 24](#_Toc137155350)

[10.4. @pack 24](#_Toc137155351)

[11. 类定义 24](#_Toc137155352)

[11.1. 面向对象 24](#_Toc137155353)

[11.2. 类(class) 25](#_Toc137155354)

[11.3. super和this 26](#_Toc137155355)

[11.4. 接口(interface) 26](#_Toc137155356)

[11.5. Jan文件 27](#_Toc137155357)

[12. 链式编程 29](#_Toc137155358)

[13. 泛型编程 30](#_Toc137155359)

[14. 箭头函数 31](#_Toc137155360)

[15. Jan Object 32](#_Toc137155361)

[15.1. Object类 33](#_Toc137155362)

[16. DataBean模型 33](#_Toc137155363)

[17. String类（Draft） 33](#_Toc137155364)

[18. 集合类（Draft） 34](#_Toc137155365)

[18.1. Array 34](#_Toc137155366)

[18.2. Matrix 35](#_Toc137155367)

[18.3. SparseMatrix 36](#_Toc137155368)

[18.4. Map 36](#_Toc137155369)

[18.5. List 37](#_Toc137155370)

[18.6. Set 38](#_Toc137155371)

[18.7. Tree 38](#_Toc137155372)

[18.8. BinaryTree 38](#_Toc137155373)

[19. 文件I/O（Draft） 39](#_Toc137155374)

[20. 多线程并发编程（Draft） 39](#_Toc137155375)

[21. 消息、事件、计时器（Draft） 39](#_Toc137155376)

[21.1. 消息通信Message类 39](#_Toc137155377)

[21.2. 事件标志EventFlag类 40](#_Toc137155378)

[21.3. 计时器Timer类 40](#_Toc137155379)

[22. 同步和异步处理（Draft） 42](#_Toc137155380)

[22.1. Mutex对象 42](#_Toc137155381)

[22.2. Semphore对象 42](#_Toc137155382)

[22.3. synchronized同步 42](#_Toc137155383)

[23. Socket通信（Draft） 43](#_Toc137155384)

[24. 注解导入（Draft） 43](#_Toc137155385)

[24.1. @getset 44](#_Toc137155386)

[24.2. @pack 45](#_Toc137155387)

[24.3. @sync 45](#_Toc137155388)

[24.4. 条件编译注解 45](#_Toc137155389)

[25. 高级语言扩展（Draft） 46](#_Toc137155390)

[25.1. 泛型和模板 46](#_Toc137155391)

[25.1.1. 泛型编程 46](#_Toc137155392)

[25.1.2. 模板变量 46](#_Toc137155393)

[25.1.3. 箭头函数 47](#_Toc137155394)

[25.1.4. promise函数 47](#_Toc137155395)

[25.2. 字节序问题 48](#_Toc137155396)

[25.3. 地址函数 48](#_Toc137155397)

[25.4. 大小端函数 48](#_Toc137155398)

[25.5. Property类 50](#_Toc137155399)

[25.5.1. Property配置类 50](#_Toc137155400)

[25.5.2. properties文件 50](#_Toc137155401)

[25.6. Json处理类 51](#_Toc137155402)

[25.7. 异常处理Exception类 51](#_Toc137155403)

[25.8. 正则表达式Regex类 52](#_Toc137155404)

[25.9. 魔法函数 52](#_Toc137155405)

[25.10. 反射机制 52](#_Toc137155406)

[25.11. 其他支持库 53](#_Toc137155407)

[26. 发展路线图 54](#_Toc137155408)

[26.1. 第一部曲: 实现 54](#_Toc137155409)

[26.2. 第二部曲: 独立 54](#_Toc137155410)

[26.3. 第三部曲: 扩展 54](#_Toc137155411)

[27. 编译处理器 55](#_Toc137155412)

[28. 开源协作 55](#_Toc137155413)

[29. 后记 56](#_Toc137155414)

[29.1. C/C++版本问题 56](#_Toc137155415)

[29.2. Jan的哲学问题 57](#_Toc137155416)

https://gitee.com/xusz-jan/JanPL.git

# 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。

**大道至简 2015/6/24 xusz**

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。

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。追踪溯源，承前启后，大道至简，返璞归真。

## 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。

主要特性：

1. 类型定义后置(用<类型>指定变量或函数返回的数据类型)。例如： a<int32>
2. 采用类型缩写前缀来定义变量。例如：i4A, 表示定义了[int32 i4A]变量。
3. 采用Python缩进书写
4. 支持@注解编程
5. 支持高级语言特性：链式编程、箭头函数、模板泛型等
6. 创新Smart Object对象模型
7. 支持Agent编程特性
8. 支持反应式编程
9. 简洁的语言支撑库，按需扩展和引用

### Jan语言的区别

**Q**：Jan是C/C++语言吗？

**A**：否。Jan吸收了C/C++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。Jan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。

**Q**：Jan是Java语言吗？

**A**：否。Jan吸收了Java以及Spring/Spring Boot的很多优秀特点，但是它本身是一种新语言。有自己的语法特点、支撑库和应用领域。

### 为什么还要Jan语言？

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，可以用3个词来概括：

1. VACU时代： 易变性/Volatility, 模糊性/Ambiguity, 复杂性/Complexity, 不确定性/Uncertainty
2. CASE时代： 互联/Connected，自动驾驶/Autonomous，共享/Shared & Services，电动/Electric
3. ABCD时代： 人工智能/AI，区块链Block chain，云计算/Cloud Computing，大数据/Big Data

它在不断变化，不断变革，不断创新，不断尝试。

简洁、高效、快速，是时代发展的永远追求，也是软件开发的最大期望。Jan就是往目标前进的一个尝试。

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相同，代码块需要缩进时，默认采用4个半角空格。不建议用Tab键缩进。

**【例1】 Helloworld.jan**

func main():

println(“Hello world, I'm Jan!”)

**【例2】Add.jan**

func main():

i4A = input(“Please input number A :”) //用i4前缀自动定义int32变量

i4B = input(“Please input number B :”) //用i4前缀自动定义int32变量

c = Add(i4A, i4B) //根据Add函数返回类型，自动推导c为int32

println(“The result of ${i4A} + ${i4B} is ${c}”) //模板表达式功能,实现参数替换以及简单计算

func add(a<int32>, b<int32>)<int32>: //强制指定参数a和参数b类型为int32

retrun a+b

**【例3】斐波那契数列递归算法 Fibo.jan**

func fibo( i4N)<int32>: //用i4前缀指定参数类型为int32

if i4N==1 or i4N==2: //or为逻辑或，也可以用 ||

return 1

else:

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

func main():

i4\_n = input(“Input N: ”)

ret = fibo(i4\_n)

println(“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, @export等）

能让编译程序去做的事，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

## 通用命名规则

|  |  |  |
| --- | --- | --- |
| **分类** | **命名规则** | **举例** |
| 类，结构体，枚举，联合 | 大驼峰命名 | PageData |
| 接口 | I + 大驼峰命名 | IPageOperating |
| 一般变量和成员变量 | 全部小写，以下划线隔开 | page\_size |
| 常量和静态变量 | 全部大写，以下划线隔开 | READ\_BUFFER\_SIZE |
| (类，结构体，枚举，联合)的实例变量 | 小驼峰命名 | pageData |
|  |  |  |
|  |  |  |
| 首字符下划线 | 表示“私有，不公开，局部，范围限定”的意思。 | \_page\_size |
|  |  |  |
|  |  |  |
|  |  |  |

# 先谈点哲学

## 计算机语言和哲学之间的关系是什么？

计算机语言的本质是什么？

逻辑符号系统是哲学中的一个重要概念，它是一种用于描述和推理思维的符号系统，是哲学的基础之一。在哲学中，逻辑符号系统被广泛应用于逻辑学、数学和语言哲学等领域。

计算机语言是一种用于描述计算机程序的形式语言，它的**本质是一种逻辑符号系统**，通过符号和规则来描述计算机程序的行为和操作。计算机语言中的符号和规则都是经过精心设计和定义的，具有严格的语法和语义，可以被计算机直接解释和执行。它和自然语言不同，更加精确和规范。

计算机语言和哲学之间的关系是紧密相连的。计算机语言的出现和发展也为哲学提供了新的视角和工具，例如计算机科学家图灵提出的图灵机模型和图灵测试等概念，对哲学和人工智能的发展产生了深远的影响。

自解释语言

定义、推导和执行是逻辑符号系统中的重要概念，它们可以用不同的图形符号来表示：

定义：在逻辑符号系统中，定义通常用符号 "≡" 或者 ":=" 来表示。例如，"A ≡ B" 表示 A 和 B 是等价的，或者 "A := B" 表示将 B 定义为 A。

推导：在逻辑符号系统中，推导通常用箭头符号 "→" 或者 "⊢" 来表示。例如，"A → B" 表示如果 A 成立，则可以推导出 B 成立，或者 "A ⊢ B" 表示从 A 可以推导出 B。

执行：在计算机科学中，执行通常用箭头符号 "→" 或者 "⇒" 来表示。例如，"A → B" 表示将 A 转换为 B，或者 "A ⇒ B" 表示将 A 执行后得到 B。

需要注意的是，不同的逻辑符号系统和计算机语言可能会使用不同的符号来表示定义、推导和执行，上述符号仅为常见的表示方法之一。

在逻辑符号系统中，推导和执行是不同的概念，它们的区别在于目的和过程。

推导是指从已知的前提或者假设出发，通过逻辑推理得出结论的过程。推导的目的是证明某个命题或者结论的正确性，或者找到一个合理的解决方案。在推导中，符号系统中的各个符号和规则都是按照逻辑规律和语法规则来使用的，推导的过程是严格的、精确的和可证明的。

执行是指按照计算机程序的指令和算法，对输入数据进行处理和计算的过程。执行的目的是得到计算机程序的输出结果，或者进行某种操作。在执行中，计算机会按照程序的指令和算法来执行，符号和规则的使用是为了实现计算机程序的功能和操作，执行的过程是具有实际意义的。

在逻辑符号系统中，推导通常用箭头符号 "→" 或者 "⊢" 来表示，而执行通常用箭头符号 "→" 或者 "⇒" 来表示。一般来说，当符号系统中的符号和规则被用于推导过程时，我们称之为推导；而当符号系统被用于计算机程序的执行时，我们称之为执行。

**第一部分 基础篇**

# 数据类型和变量

## 数据类型(bcdf-ui)

（1）简单数据类型

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 数据类型 | 前缀 | 等价通用C语言 | 字节数 | 使用举例 | 补充说明 | |
| **bool** | b | unsigned char | 1 | b\_exist,  exist<bool> | 只能为TRUE(1)或FALSE(0) | |
| **char** | c | signed char | 1 | c\_a, a<char> | 跟C语言保持一致 | |
| **double** | d | double | 8 | d\_weight  weight<double> | double最大范围约为1.8e308 | |
| **float** | f | float | 4 | f\_area  area<float> | 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 | u1\_a, a<uint8>  u2\_b, b<uint16>  u4\_c, c<uint32>  u8\_d, d<uint64> | uint8 与 byte 类型相同  u2\_b等价于 uint16 u2\_b, 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 | i1\_a  i2\_b  i4\_c  i8\_d,i8\_miles | int8 与 char 类型相同  i2\_b等价于 int16 i2\_b，其余类推。 | |
| **?** | 无 | 无 | 不定长 | a<?> | 任意数据类型 | |
| （2）定义复合类型 | | | | | | |
| 数据类型 | 作用 | | 字节数 | 使用举例 | 补充说明 |
| enum | 定义枚举类型 | | 定长 | enum Abc<int32>:  …  a<Abc> | 同Java增强用法(enum class)，类型可定义为基本数据类型 或 string。 |
| union | 定义联合类型 | | 定长 | union Abc:  …  a<Abc> | 同C用法。 |
| func | 定义函数 | | 4 | func add():  … | func即为function的意思和作用 |
| class | 定义类 | | 不定长 | class MyObject:  … | Jan语言的class类 |
| interface | 定义接口 | | 不定长 | interface IReader:  … | 同Java |
| struct | 定义结构体 | | 不定长 | struct Rect:  … | 同标准C。  不支持成员函数。  成员为定长简单数据类型或struct或union |
|  |  | |  |  |  |

## 运算符

支持C/C++运算符(+, -, \*, /, % ，++和--)。

支持bit位操作符： &比特与 |比特或 ~比特非(取反) ^比特异或(xor)

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

其余运算操作（譬如整除取余等），用函数库实现，保持运算符的简洁性。

## 变量

变量是用来存储用户数据的。

编译器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。

如果担心自动推导会出错，那么在变量定义时就需要明确指定数据类型。

1. 变量命名规则（下划线连接法）

所有的变量，都以小写字母或下划线开头。变量的命名使用下划线连接法。可以在变量头部、尾部或者中间，存在0个或1个以上下划线。

例： i4\_width, i4\_read\_buf\_size、read\_buf\_size、\_\_read\_buf\_size都是下划线连接法 变量名

**变量前缀的一个或者多个下划线**，表示该变量为 私有变量/本地变量（只在模块、函数或类里面使用）。

1. 变量默认值规则

所有变量都无默认值，需要初期化赋值后才能被正确使用。

1. 变量声明和初始化

**标准格式1：**变量名<类型> = 初期值 （举例：width<int32>=100）

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

如果<类型>和初期值的类型不一致(如：width<int16>=100.5)，会给出编译警告或者错误。

**标准格式2：**带前缀变量名 = 初期值 (举例：i4\_width=100)

以前缀i2为例：**i2\_width 是标准变量命名法**。等价于i2\_width<int16>

i4\_width 等价为 i4\_width<int32>

**标准格式3：**变量名 = 初期值 (举例：a=100)

初期値后缀定义规则：

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 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 优先定义类型为 int32

**变量默认类型：**

凡是不能自动推导出类型的变量，都被指定为 默认类型。

默认类型由编译选项 JAN\_DEFAULT\_VAR\_TYPE来指定，可以定义其为：int32，或 <?> 或 其他类型。

如未重新指定编译选项 JAN\_DEFAULT\_VAR\_TYPE，则默认类型为：int32

举例：(假设默认类型为int32)

a, f 等价于 a<int32>, f<int32>

**单独类型前缀：**

i2, \_i2, i2\_ 单独的类型前缀，不能独立表示类型，都被定义为默认类型。

推荐用法：i2<int16>, \_i2<int16>, i2\_<int16> 需要明确指定变量类型，这样就不会混淆。

1. 变量声明和初始化

变量类型优先级定义

|  |  |  |
| --- | --- | --- |
| 优先级（从高到低） | 定义格式 | 举例 |
| **1，<类型>** | 变量名<类型> = 初期值 | a<int8> = 0 类型为int8 |
| **2，[类型前缀]** | [类型前缀]变量名 = 初期值 | f\_sum = 0 类型为 float |
| **3，初期値后缀** | 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（默认）  a = 2.5D 类型为 double |
| **4，上述以外** | 变量名 | 默认类型。  由JAN\_DEFAULT\_VAR\_TYPE指定。 |

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

u2\_a<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，u2\_a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>

{a,b,c,d}<float>=1.0 等价于a<float>=1.0, b<float>=1.0, c<float>=1.0, d<float>=1.0

1. 批量变量赋值和值交换

{a, b} = {b, a}

{a, b, c} = {1, 2, 3}

{a, b, c} = {c, a, b}

{a, \_, c} = {1, 2, 3}

其中，单个下划线，表示由编译器分配的临时**匿名变量**，只用于占位，不能被使用。

1. 数据类型变换

变换方法： (指定类型)xx

举例

i2\_b = (int16)i4\_a

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. <?>未知类型变量使用

<?>变量可以被任意赋值。

例如 a<?>= 1200; a= 120.0

其他变量可以用<?>变量赋值，但是需要加上类型转换。

例如

u2\_a = (uint16)a 可以将a<?>赋值给u2A

# 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<100; i +=2)

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

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

**for item<Person> in items** 语义类似于 for(i=0;i<len(items);i++){ item= items[i]}

**支持while循环**

while i<9:

i = i + 2

在循环体中，要退出和继续循环体时，可采用break和continue。

废除do和loop关键字

## 判断语句

if i<100:

xxx

else if 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。

空语句可以使用 pass。

## 分支语句

switch cccc:

case range(0,8):

case 11: //case后面必须为明确的数字或者字符，不能为浮点数。

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。

# 一维数组

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。数组下标从0开始。

**行向量(一维数组)的定义：**

变量名[元素个数]<类型>

**矩阵(二维数组)的定义:**

变量名[元素个数, 元素个数]<类型>

**多维数组的定义**

变量名[元素个数, 元素个数, ...]<类型>

例

names[100]<int32> 等价于 names<int32[100]>

names2[100, 200]<int32> 等价于 names2<int32[100 , 200]>

**数组赋初值：**

names[]<int32> = [1, 2, 3]

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

names[100]<int32>=0 数组元素全部赋初值0

**多维数组**

names[2, 3]<char> = [ {‘a’ ‘b’ ‘c’} , {‘d’ ‘e’ ‘f’} ]

a b c

d e f

元素分别为

names[0, 0]=’a’ names[0,1]=’b’ names[0,2] =’c’

names[1,0]=’d’ names[1,1]=’e’ names[1,2] =’f’

a[2 ; 3 ; 4] 的数字内容如下所示：

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[,,] = [ { {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2 \* 3 = 6，也就是说有 6个元素

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, <?> 类型的成员。

**结构体定义**

struct Abcd:

c<int8>

size<uint16>

i2Height

结构体赋值

数据类型是结构体对象时，可对成员变量进行个别赋值。

struct Rect:

width<int32>

height<int32>

pos\_x<int32>

pos\_y<int32>

rect<Rect>={0} //表示需要对结构体进行初期化

语义等价为：

Rect rect;

memset(&rect, 0, sizeof(Rect));

rect<Rect>:

height=10

pos\_x = 100

语义为：

Rect rect;

rect.height=10;

rect.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：func** 函数名(参数)<返回类型>:

当函数有返回值是，请指定<返回类型>

当未指定<返回类型>时，表示函数为无返回值。举例： printIn()

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，使小驼峰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编程技巧：**

传入结构体（对多个参数进行封装）

传入Map，List等（对多个参数封装）

对函数进行功能上的拆分，一个函数拆分为几个函数

1. 函数返回值

如果函数不要返回值，则不推荐写return。如果写了的话，无任何作用，反而不简洁。

举例：

func example()：

print(“I’m Jan!\n”)

空函数：

func example():

pass //表示空语句，什么也不做。pass空语句的作用在于避免编译器抛出空函数警告。

函数返回值可以为单个数值或者数组，例：

func getData()<int32[]>:

return [10, 2, 100]

合法使用方法：

a<int32[]> = getData() , 那么a[0]为10，a[1]为2，a[2]为100

等价用法 a[]<int32> = getData()

等价用法 a = getData() //编译器使用自动推导，可以推导出 a<int32[]>, 不推荐使用

## 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 compute(…i4Params)<float>: // 根据前缀变量命名规则，推断类型为 <int32[]>

sum = 0.0

for param in i4Params:

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 compute(...params<int32>)<int32> //说明params是<int32[]>类型

func compute(...params)<int32> //说明params是<?[]>类型

**第二部分 高级篇**

# 注解编程

## @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

@static:

aaa<int16> = 0

bbb<int16> = 100

对于其他注解方式（如@const, @volatile等），上述写法同样适用。

## @const

常量命名使用大写字母并用下划线分割单词。

举例：MAX\_ARRAY\_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

@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 \*/

TRUE<int8> = 1

FALSE<int8> = 0

UNKNOWN<int8>= -1

## @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

## @pack

**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，这就是字节对齐要求。

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a<int32>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。

@pack用于指定struct结构的字节对齐模式，默认4字节对齐，可以用@pack(4)指定。

举例：

@pack(4)

struct Abcd:

c<int8>

\_<int8> // 匿名变量, 编译器自动补足或手动补足

size<uint16> // size是2字节，必须双字节地址位置上对齐

i2Height

\_<int8> // 匿名变量，编译器自动补足或手动补足

\_<int8> // 匿名变量，编译器自动补足或手动补足

# 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。

## 类(class)

类定义： **class类名 extends 类名A, 类名B** （如果没有父类的话，extends子句可以省略）

举例： class Aaa extends B, C：

width<int32>

height<int32>

calcArea()<int32>:

return width \* height

**成员变量定义：**

以下划线开头的成员变量为私有成员，只在当前类里面使用，不能被子类使用(类似于private)。

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。

成员变量还可以使用@data注解

@data({get, set}, chain=true)

width<int32>

成员变量的命名推荐：

不建议 以前缀缩略方式定义成员变量。

举例：

（推荐） width<int32>, height<int32>,area<float>

（不推荐） i4Width, i2Height,fArea

原因：不利于后续自动生成get函数或者set函数。

譬如对i4Width自动生成getI4Width, setI4Width函数时，显然其函数名不易理解和使用。

**成员函数：**

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。

以下划线开头的成员函数为私有函数。

除静态和私有的成员函数外，其他成员函数均默认为虚函数，且能够被子类重载。

无函数执行体的函数为抽象虚函数，必须在子类进行实装重载。用注解@abstract标明。

@abstract func calcArea()

@override用于父类函数重载。凡是重载父类函数或者虚函数，都要用注解@override标明。

**成员变量和函数的可见性、可继承性(利用符号+-#)**

+ 相当于C++中的public

* 相当于C++中的private

# 相当于C++中的protected

举例：

+width<int32> // +符号与变量名之间不加空格

func #calcArea():

xxxxx

## super和this

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，采用最近原则。

this表示当前类本身，与C++和Java使用相同。

举例

class A:

func calcTax()<int32>:

...

func calcArea()<int32>:

class B extends A:

@override

func calcTax()<int32>:

class C extends B:

@override

func calcTax()<int32>:

i4A = super<A>.calcTax() 直接找到class A的calcTax方法

i4B = super<B>.calcTax() 等价于 i4B = super.calcTax()，向上找到最近的class B方法

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

return int32(i4A + i4B + i4Area\*0.2)

## 接口(interface)

接口类 使用 interface 来定义。

接口定义：

interface IReader extends 接口A，接口B:

静态常量(常量名不能以下划线开头)

非静态虚函数(函数名不能以下划线开头)

为了方便易懂，接口命名遵循下述规则：

首字母必须为 大写I

命名构成：首字母I + 接口名

例如： IReader，IPlayMusic

接口的用法

接口是特殊的类

接口不能有实例

接口中不能定义变量

接口中的函数都是抽象虚函数，不需要用@abstract标明，在类中需要用@override重载实装

接口也可以被继承

接口的使用

class MyReader implements IReader:

xx

## Jan文件

(1)、文件命名

Jan语言文件统一命名为 xxx.jan。文件名本质上为 模块名。

文件命名规则：

1）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）建议全部小写。

3）文件名后缀为 [.jan]

例如： book.jan

(2)、包定义：

。，小包是指一个Jan文件。

**大包定义**

每一个Jan源文件的在开头都要声明package语句。单个package是指一个目录，被称为大包。

例：package com.ajler.try 必须存在相应的目录路径com/ajler/try。

**导入小包**

小包是指位于大包下面的一个Jan文件。

例：

import com.ajler.try.book 表示从book.jan文件导入所有的export对象

import {类，接口，变量} from 小包 表示从jan文件中导入所指定的export对象

import { a as aa} from 包 as用于给导入对象起别名，避免命名冲突或简化

(3)、Jan文件构成：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | 可否省略 | 目的 | 语法/说明 | 举例 |
| 1 | Must | **package包定义** | package 包名 | package aa.bb.cc |
| 2 | Option | **import引入类** | 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 | **常量、Enum定义、struct定义** | 全局或者本地 | 全局则用export |
| 4 | Option | **函数或者变量** | 全局或者本地 | 全局则用export |
| 5 | Option | **定义类/接口**  (0个，1个多个) | [export]class 类名 extends 父类或接口  interface 接口名 | 全局则用export  class Aaa extend B, C implements IReader  一个或者多个类的定义 |

(4)、关于export

export 用于导出全局变量或者函数或者类，如果不导出的话，只能在当前文件内使用。

例如[aaa.jan]文件：

export window\_max\_width<int32> = 100 //声明和导出能被其他模块（其他文件）使用

export func calcTaxiFee()<float>: //声明和导出全局函数

export class MyClass: //声明和导出全局类

也可以在文件末尾统一定义和声明export对象：

export default:

window\_max\_width,

calcTaxiFee,

MyClass

# 链式编程

链式编程是将多个操作通过点号"."链接在一起成为一个整体，从而更加的简洁方便。链式编程的原理就是每个操作完成后都会返回一个this对象，也就是返回对象本身。

map<HashMap> = new HashMap()

map.put(“key1”, “object1”)

.put(“key2”,object2)

为了操作简便，Jan语言库尽可能提供链式编程特性。

链式编程注解： @data(set, get)

get函数提供链式支持。

set函数默认支持链式编程特性

举例：

class MyData:

@data(set)

width<i4>

使用@data注解，编译时能自动生成链式set函数。

@data(set)支持链式操作的set函数：

func setWidth(width<i4>)<MyData>:

this.width = width

return this;

@data(set, nochain)创建不支持链式操作的set函数：

func setWidth(width<i4>):

this.width = width

# 泛型编程

泛型模板可以用在函数、类、接口里面。

支持泛型的函数定义：

func <T>add(a<T>, b<T>)<T>:

return a + b

支持泛型的类定义：

class <T1, T2>MySample:

…

# 箭头函数

将匿名函数复制给变量的简写方式的函数称为 lambda 表达式，箭头函数就是lambda表达式，它提供了更简洁的function定义方式。

箭头函数允许用更短的语法定义函数，所定义的函数是匿名函数。

箭头函数可用于替代传统函数function() {}。

举例1：

func sum(a, b):

return a + b

改写为箭头函数： func sum = (a, b) => a + b

多行书写： func sum = (a, b) =>:

a + b

举例2：

箭头函数：addOne = num => num + 1 只有1个参数时，可以省略掉括号()。

举例3：

箭头函数：adults = users.filter(user => user.age > 18) 箭头函数用于过滤函数中

**第三部分 语言模型**

# Jan Object

Object类是Jan语言的基类，所有其他类都是Object类的子类。子类可以使用 Object类的所有方法和属性。与Java中的Object类相比较，Jan语言特意追加了很多属性和方法。

clone() 创建并返回一个对象的拷贝

equals() 比较两个对象是否相等

initialize() 初始化该对象

finalize() 释放该对象

getType() 获取对象的运行时对象的类

hashCode() 获取对象的 hash 值

toString() 返回对象的字符串表示形式

同步函数：wait, notify, notifyAll

消息函数：snd, rcv 异步发送和接收消息(JSON数据)

parent

methods

datas

meta

辅助函数

clone

equals

initialize

finalize

getType

hashCode

toString

Jan Object

（this）

同步函数

wait,notify,notifyAll

计算属性

computed

属性变化函数

watch

消息函数

snd, rcv

## Object类

# DataBean模型

# String类（Draft）

为了避免String类的复杂性，将所有与正则表达式相关类型的处理，从String类中去除，由Regex类进行处理。

charAt(index<int32>) 返回指定索引处的 char 值。

compareTo(o<object>) 把这个字符串和另一个对象比较。

compareTo(anotherString<String>, ignoreCase<boolean> ) 按字典顺序比较两个字符串。

compareToIgnoreCase(str<String>) 按字典顺序比较两个字符串，不考虑大小写。

concat(str<String>) 将指定字符串连接到此字符串的结尾。

endsWith(suffix<String>) 测试此字符串是否以指定的后缀结束。

equals(anObject<Object>, ignoreCase<boolean>) 将此字符串与指定的对象比较

getBytes(charsetName<String>) 使用指定字符集将此String 编码为 byte 序列，并存储到新 byte 数组中。

hashCode() 返回此字符串的哈希码。

indexOf(str<String>, fromIndex<int32>) 返回指定子字符串在此字符串中第一次出现处的索引

isEmpty() 判断字符串是否为空。

lastIndexOf(str<int32>, fromIndex<int32>) 返回指定子字符串在此字符串中最右边出现处的索引。

length() 返回字符个数

startsWith(prefix<String>, offset<int32>) 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。

substring(beginIndex<int32>, endIndex<int32>) 返回一个新字符串，它是此字符串的一个子字符串。

toLowerCase() 将此 String 中的所有字符都转换为小写(不考虑Locale规则)。

toString() 返回此对象本身。

toUpperCase() 将此 String 中的所有字符都转换为大写(不考虑Locale规则)。

trim() 返回字符串的副本，忽略前导空白和尾部空白。

trimLeft() 返回字符串的副本，忽略前导空白

trimRight() 返回字符串的副本，忽略尾部空白

# 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, int32> 等价于 Array<int32> a; 里面的元素类型为int32

a<Array<int32>> 与 a<Array, int32>等价，为了避免多重<<>>，建议使用 a<Array, int32>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> = a.remove(b) // c的结果为：{ 12,14,16,18,20 }

c<Array, int32> = a.remove(b) // c的结果为：{ -10, -10, -10, -10, -10 }

c<Array, int32> = a.multiply(b) // c的结果为：{ 11, 24, 39, 56, 75 }，对应位置点乘

c<Array, int32> = a.divide(b） // c的结果为：{ 0, 0, 0, 0, 0 }，同C语言一致，只取整数。

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

暂不提供。

## BinaryTree

暂不提供。

# 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

initialize() 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

clearTimeout

# 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)<int32>:

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<int32>) 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

connect() 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。

### 泛型编程

泛型编程即以一种独立于任何特定类型的方式编写代码。

泛型函数：（自动识别大写T开头的类型为泛型类型）

func mymax(a<T>, b<T>)<T>:

return a < b ? b : a

函数调用： c<int16> = <int16>mymax( a, b)

简单而言，函数前面用template定义模板类型

对于多个泛型类型而言，可以使用同样的方法进行定义。

例如

func myfunc(a<T1>, b<T2>)<T3>:

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

泛型类：

class MyObject<T, T1, T2, T3>:

memeber<T>:

...

func mymax(a<T1>, b<T2>)<T3>:

....

### 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}")

${}括号中，支持简单的运算操作。

### 箭头函数

为什么叫Arrow Function？因为它的定义用的就是一个箭头。

例如：

1. 无参数的箭头函数：

() => 3.14

等价于 func():

return 3.14

1. 单个参数的箭头函数

(x) => x \* x 或者 x => x\*x

等价于 func(x):

return x\*x

1. 多个参数的箭头函数

(x, y) => x+y

等价于 func(x, y):

return x+y

箭头函数的 多语句定义方法如下：

(x,y) => { x = x +2; x \* y }

或者

(x,y) =>

x = x +2

x \* y

### promise函数

## 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。

## 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：madr(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 = madr( u4Width)

获取指定地址内容

cBytes[] = mget( u4Addr, 4)

## 大小端函数

大端(binary.BigEndian) 和 小端(binary.LittleEndian)

读取变量

变量 = binary.Read(cBytes[], binary.BigEndian, type<JanType>)

写入变量

binary.Write(cBytes[], Bytes.BigEndian, 变量<类型>)

内存转换辅助函数

使用举例

i2Data = binary.Read(cBytes, binary.BigEndian, JanType.INT16)

字节序是指多字节数据在计算机内存中存储或网络传输时字节的存储顺序。通常有小端和大端两种方式。

小端：低位字节存放在内存的低地址端，高位字节存放在内存的高地址端。

大端：低位字节存放在内存的高地址端，高位字节存放在内存的低地址端。

Jan语言的字节序是大端，与Java语言相同。

X86体系用小端字节序，即 低内存地址空间存储数据的低位。

## 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")

由于finally处理机制复杂，特别是try/catch/finally/return混合起来，就更难于理解了。

举个例子:

finally语句块中能否包含try/catch?

try语句返回 true时，如何执行到 finally语句块？

try语句块和finally语句块都有 return 语句时该如何理解？

在Jan语言中，特意不支持Java语言中的 类似finally机制。

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>:

.....

## 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>

matches(str<String>, regex<String>)

告知此字符串是否匹配给定的正则表达式。

replaceAll(str<String>, regex<String>, replacement<String>)

替换此字符串所有匹配给定的正则表达式的子字符串。

replaceFirst(str<String>, regex<String>, replacement<String>)

替换此字符串匹配给定的正则表达式的第一个子字符串

split(str<String>, regex<String>, limit<int32>)

根据给定正则表达式的匹配拆分此字符串。

## 魔法函数

（......还在考虑和设计中......）

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，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。

## 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。

Jan语言扩展

Jan独立编译器

Jan翻译器

Jan语言设计

扩展阶段

独立阶段

实现阶段

萌芽阶段

## 第一部曲: 实现

充分利用GoLang语言库和编译器，实现JanPL到GoLang的编译和实现。

## 第二部曲: 独立

初步选定 使用

## 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

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

类函数和成员变量的可见性

+ public

- private

# protected

@static

@private

@protected

联系方式： [1344272573@qq.com](mailto:1344272573@qq.com)

<The End>