鲁班编程语言

黄晓川 著

2005.8

目　　录

1. 什么是鲁班 4
2. 脚本鲁班 6
  2.1 第一个鲁班程序 6
  2.2 鲁班脚本基本结构 6
  2.3 常数, 变量和表达式 7
  2.4 鲁班顺序执行语句 8
  2.5 鲁班数据类型 9
    2.5.1 整数和双精度浮点数 9
    2.5.2 逻辑类型 9
    2.5.3 字符类型 10
    2.5.4 字串类型 10
    2.5.5 数组，字典和集合 11
  2.6 鲁班数据类型通用操作 12
    2.6.1 类算术操作 12
    2.6.2 枚举循环语句(foreach) 14
    2.6.3 通用成员函数 14
  2.7 空值 null 和错误类型 error 15
  2.8 高级鲁班脚本技术: 多线程并行及协调 15
  2.9 线程陷阱举例 16
  2.10 鲁班简单的世界: 一切都是对象 17
3. 鲁班部件入门 18
  3.1 鲁班部件构造 18
  3.2 鲁班部件的输入和输出属性 19
  3.3 鲁班部件的多种用法 20
  3.4 鲁班的传值调用规则 23
  3.5 鲁班脚本程序和鲁班部件定义程序 23
4. 鲁班部件组合入门 24
  4.1 简单组合举例 24
  4.2 组合单元详述 25
    4.2.1 不定型单元 25
    4.2.2 定型单元 26
  4.3 组合部件内部执行顺序 27
  4.4 数据流定义规则 29
    4.4.1 能改变自己不能改变外界 29
    4.4.2 数据流不能循环 30
  4.5 过程与组合的选择 30
  4.6 部件界面继承和异步单元 31
5. 鲁班部件界面定义及继承详述 32
  4.1 界面属性的流向和读写权限 32

9. 输入输出对象类型及部件 33
9.1 终端, 文件和网络 SOCKET 33
9.2 文件读写举例 34
9.3 网络读写举例 35
9.4 标准屏幕打印部件: std::println std::print 36
9.5 标准对象恢复部件 std::des 36
12. 鲁班命令行解释器使用说明 37
12.1 运行鲁班程序文件 37
12.2 交互式执行鲁班脚本或部件定义程序 38
12.3 其他命令行参数 39

1. 什么是鲁班

Windows were shakin' all night in my dreams
Everything was exactly the way it seems
- Highlands, Bob Dylan

鲁班是一个面向部件的编程语言. 鲁班也是一个整合语言或者叫脚本语言.鲁班语言中的很多构筑和搭积木,做家具,造房子非常相似. 所以作者把这个新语言命名为鲁班,以凸显鲁班建筑工艺和鲁班语言特色的异曲同工之处. 也算是对自己的文化渊源的一种反映和尊重.

有的朋友可能说, 现在脚本语言已经很多, 我们真的需要另一种脚本语言吗? 鲁班的作者确实对脚本语言非常推崇. 因为容易上手, 非专业人士都可以学会. 然而从实践中作者发现, 现有的脚本语言有一个有一个很大的弱点, 就是没有一个适合的部件模型(Component Model). 就象在沙子上做东西, 容易开始, 可做不大. 有朋友可能又指出, 已有面向对象脚本语言, 可用对象模型来组织代码. 对于这点, 作者个人认为, 在脚本语言里构筑对象模型不是脚本语言强项. 因为对象模型对于专业人士都难于把握, 完全超出了一般脚本语言用户的掌握范围. 用脚本语言写对象类型没有优势, 比不过更正式的语言象 C++和 JAVA.

由此来看, 脚本语言要超越写小脚本程序的圈子, 就需要一个适于脚本语言环境的部件模型. 在编程语言光谱上确实存在一个空白地带. 鲁班的发明就是着眼于填补这个空缺.

鲁班首先是一个脚本语言, 易学易用, 无须编译. 而且鲁班是一个更好的脚本语言, 因为它的语法更简洁, 类似于 C++/JAVA. 一般编程用户几乎不用学新的东西就可以开始用鲁班写程序.

鲁班超越一般脚本语言之处是鲁班部件模型. 鲁班语言把部件定义为属性构成的对象, 类似于 JAVA BEAN 的概念. 部件的定义, 归类, 存放及组合是鲁班语言的核心特色. 还有一点和面向对象语言不一样的是, 如果你只是用鲁班写一点小脚本程序, 你可以不用知道什么是鲁班部件. 鲁班把对新用户的要求降到最低.

还有值得一提的是鲁班的部件组合功能. 部件组合是一种全新的编程方法. 传统的编程都是定义代码的执行顺序, 而部件组合定义的是一个模块内部件之间的数据依赖关系, 就象搭积木. 鲁班引进部件组合是因为它简单而直接, 一看即明. 最好的证明是 EXCEL. 几乎人人都会用 EXCEL. 编写 EXCEL 实际上就是部件组合. 鲁班是第一个正式支持部件组合的软件编程语言.

鲁班语言是一个以人为中心设计的语言, 希望是更多的人能有效的用计算机做事情, 而且能共享他们的工作成果. 鲁班是一个脚本语言, 可作者希望它能走的更远. 所以也叫它整合语言.

顺便说一句, 本书每章开头的引言, 是作者随意摘抄的, 有些是歌词, 有些是有人随意说的话. 和鲁班技术细节没有直接关系. 这些东西很难翻译, 我就不糟蹋别人了. 想看的朋友可以看着玩, 不想看不看就罢了.

2. 脚本鲁班

In the beginner's mind there are many possibilities,
in the expert's mind there are few.
- Shunryu Suzuki, Zen Master

这一章讲述基本的鲁班顺序执行语句构造, 包括变量, 常数, 表达式和语句. 看完这一章就可以用鲁班写一般的脚本程序了. 已经掌握 C++/JAVA 的朋友, 用五分钟就可以看完这章. 因为鲁班的语句和表达式和 C++/JAVA 基本相似, 需要看的只是鲁班扩展和不同的部分. 而那部分也是很简单, 一看即明的.

2.1 第一个鲁班程序

学习一个新的语言的最有效的办法莫过于看看程序例子. 好的语言总是以一个"世界, 你好"程序例子开始. 鲁班也不例外. 下面是鲁班的"世界, 你好"程序.

```
Std::println(obj="Hello, World");
```

把以上代码敲进一个叫"helloworld.lbn"的文件里. 然后在你的操作系统命令行敲以下命令.

```
Mycomputer >  one  helloworld.lbn
Hello, world!
```

这个鲁班程序有一点象 C 的"HELLOWORLD"程序. 里面代码做的事就是调用一个叫做"std::println"的鲁班部件, 调用时把字串"Hello, World"设置到一个叫"obj"的输入属性上. 程序的执行结果是在终端上印出"Hello, World"字串.

2.2 鲁班脚本基本结构

鲁班的脚本程序是由一系列的鲁班语句或表达式构成. 这些语句和表达式之间用分号隔开. 换行和空白字符会被忽略. 这样的结构和 C++/JAVA 是一样的. 以下是一个简单的鲁班脚本程序.

```
X=1;
Y=2;
Z=X+Y;
std::println(obj=Z);
```

把以上程序输入到一个叫"oneplustwo.lbn"的文件里然后在命令行打入以下命令:

```
Mycomputer >  one  oneplustwo.lbn
3
```

从以上程序可以看出鲁班脚本程序的大概样子.

2.3 常数, 变量和表达式

鲁班的表达式和 C++/JAVA 基本一样. 由常数,变量和操作符组成. 以下列举表达式的基本构造. 列举大致按运算的优先级排练.

常数本身是一种表达式. 鲁班的常数有如下几种.

3.1415; 2; "Hello"; 'a'; '\n';true; false;

变量也是表达式. 鲁班变量起名规则和 C 语言一样, 如下都是合法的变量名.

Varx; _varx_; v123;

数据类型对象建造及部件建造.

[1,2,3, x, y, z, a+b]; // 数组建造
{ "one": 1, "two" : 2 }; // 字典建造
{ "one", "two" }; //集合建造
double("3.1415"); //双精度浮点数建造
std::file("mydatafile", 'r'); // 打开文件
mynamespace::mycomponentX(); //部件建造
mynamespace::mycomponetX(prp1 = 100); // 部件类型调用

容器类型成员访问, 对象成员函数调用, 部件属性访问, 部件调用.

Mymap["one"]; // 字典查找
x[1] = y[2]; // 字典或数组成员读取及赋值
mymap.remove("one"); // 对象成员函数调用
linevector = allines.split('\n'); //对象成员函数调用, 分解字串
mycalculator(input1=1.1, input2=2.2); // 部件对象调用

单目操作符运算,

++i; --i; i++; i--; // 操作会改变变量 i 的值
y = -x; // 变量 x 值不变, 运算结果赋予 y

常见算术类操作, 加减乘除及求余数

*1+2; 10.0*3.0; 50%3; 50/3; "Hello, "+"world!";*

数据类型检查操作

objx isa string; // 检查 objx 是否字串类型
typeof(objx) == string; // 检查 objx 是否字串类型

相等关系及顺序关系检查

x == y; x != y; x > y; x < y; x >= y; x <= y;

逻辑表达式

true or false; x and y; not true; not a and not b;

三元条件表达式

x > y ? x: y;

赋值表达式

x = 1.23; x += 1; x[index] = 2; x.property = 3; x=2; x/=3; x=y=z;*

以上表达式中有一些构造包括类型检查和部件调用表达式在以后章节中还会讲述.

2.4 鲁班顺序执行语句

鲁班的顺序执行语句和 C/C++/JAVA 基本一样. 只是加了几个鲁班自己的语句. 以下列举所有的鲁班顺序执行语句.

赋值语句: 赋值语句也是表达式, 在此再列为语句.

```
X = 1; x += 1; x[index] = 2; x.property = 3;  x*=2;  x/=3;  x=y=z=1;
```

条件语句

```
if ( a>b )
        result=a;
```

条件否则语句

```
if ( a>b )
        result = a;
else
        result = b;
```

条件循环语句

```
while ( x < 100 ) ++ x;
```

步进循环语句

```
for( i=0; i<100; ++i)
        std::console().writeline(vec[i]);
for(;;); // 死循环
```

枚举循环语句

```
foreach( key, value in {“one”:1, “two”:2} )
        std::console().writeline(key, ‘ ‘, value);
foreach( element in [ 1,2,3 ] )
        std::console().writeline(element);
```

跳出循环语句

```
for( i=0; ; i++)
        if ( i>100 )  break;
```

继续循环语句, 跳回循环起始点

```
for( i=0; ; i++)
        if ( i<=100 )
                continue;
        else
                break;
```

空白语句

```
;
```

结束语句,终止一个脚本或部件的执行

```
finish;
```

等待语句: 等待所有启动的线程结束

```
wait;
```

特指等待语句: 等待和等待变量相关的线程结束

waitfor x,y,z;

撤消语句: 撤消所有启动的线程并等待它们结束

cancel;

以上线程有关的语句细节在后面有叙述.

组语句. 把一个或多个语句用花括号包起来, 就成为一个组语句.

```
{  x = 1; y = 2; z = x+y; }
if ( z == 3 ) { z = z-3; x =0; y =0; }
```

多个属性设置语句, 一次设置一个部件的多个属性值.确具体的细节再部件定义一章还有讲述

```
Xcomponent.(prp1=1, prp2=2, prp3=3);
```

注解语句. 鲁班的注解语句与 C++/JAVA 一样

```
// 注释
/*  也是注释 */
```

2.5 鲁班数据类型

鲁班语言提供一系列最常用的数据类型给用户. 这些类型包括整数, 双精度浮点数, 字串, 逻辑类型, 字符类型, 数组, 字典, 集合. 以下逐一介绍它们.

2.5.1 整数和双精度浮点数

整数和双精度浮点数是两个不同的数据类型. 但是它们可以自由混合使用. 运算规则和 C 语言一样. 整数的类型关键字是 int, 浮点数类型关键字是 double.

```
10/3;          // 结果为整数 3
10.0/3.0;      //结果为浮点数  3.333333…
10/3.0;        //结果为浮点数  3.333333… 浮点数混合整数, 结果为浮点数
0 == 0.0;      // 结果为 true, 浮点数与整数可混合比较
2 > 0.1;       //结果为 true, 浮点数与整数可混合比较

x=1.23456789;
y=x.format(".4"); // 按指定格式将浮点数转化为字串, 保留小数点后 4 位
                  // 结果为 "1.2346"
z = int(x);  // z = 1, 将浮点数转化为整数
xstring = "1.23456E7";
xdouble = double(xstring);  // x = 1.23456x10^7 将字串转化为浮点数
```

2.5.2 逻辑类型

逻辑类型很简单, 只能是真或者假两种值. 逻辑类型的关键字是 bool, true, false. 逻辑类型可用于逻辑表达式. 逻辑表达式操作符有"或", "与", "非"三种, 对应关键字 or, and, not. 鲁班的条件及循环语句里的条件表达式的运算结果必须是逻辑类型. 否则鲁班会终止当前部件的执行. 这一点和 C 语言不一样. C 会把整数类型隐含转换为逻

辑类型, 而鲁班不做隐含转换. 从整数到逻辑类型的转换要有明确的代码. 以下是例子.

```
Truth = true or false; // 结果为 true
truth2 = bool(1);   // truth2 = true
truth3 = bool(13); // truth3 = true
fiction = bool(0);  // fiction = false
truth4 = truth and truth2 and truth3 or fiction;  // truth4 = true
```

熟悉 C 语言的用户仍然可以使用 C 风格的逻辑操作符, 用*&& II ! 来代替 and, or, not.*

2.5.3 字符类型

字符类型等同于 C 语言里的单字节字符. 以下是一些字符类型操作例子. 字符类型关键字是 char.

```
For( onechar ='a'; onechar <= 'z'; ++onechar)
        std::print(obj=onechar);
```

运行以上鲁班程序会在屏幕上印出 abcdefghijklmnopqrstuvwxyz 二十六个字母.

```
NinetySeven = int ('a');  // NinetySeven = 97,  字符转换为 ASCII 整数
Lettera = char(97);  // Lettera='a', 整数 ASCII 转换为字符
```

2.5.4 字串类型

字串类型关键字是 string. 以下是一个比较完整的例子.

```
S = "Hello"; c = s[1]; // c='e' 左起第二个字符
s = "Hello"; s[0] = 'h'; // s 变成 "hello"
s = "Hello"; foreach( c in s ) std::console().write( c );  // 在屏幕上印出 Hello
s = "Hello"*2;  // result ="HelloHello"
s = -"Hello"; // result = "olleH"
s = "Hello" + "World"; // result="HelloWorld"

// below are member function examples

// SPLIT ! 类似于 PERL 的最常用的 split 函数
s = "Hello World"; words = s.split( ); // 将字串分解成数组,空格为分割符
                                       // words = ["Hello", "World"]

// 去除头尾空白
s = "    Hello";
s.trimfront();  // 去开头空白 s = "Hello"
s = "Hello    \n\t";
s.trimback(); //去结尾空白 s="Hello"

//查找子字串
s="Hello";
index=s.find("Hell"); // index = 0, "Hello" 开头是"Hell"
index2=s.find("Hell",2); // index = null, 查找失败,
```

```
                                    // 从"Hello"的左起第三个字符起没有"Hell"字串
index3=s.find("Heaven"); // index3 = null  查找失败
truth=s.contains("Hell"); // truth = true "Hello"包含"Hell"

// 读取子字串
sub=s.substr(2,3); // sub="ll" 子字串从序号2 到3

// 改变字串内容
s="Hello";
s.replace("Hell", "Heaven"); // s="Heaveno"将字串内的"Hell"换成"Heaven"
s.lower(); // s="heaveno" 变小写
s.upper(); // s = "HEAVENO" 变大写
s.remove(6); // s="HEAVEN", 删除序号6 字符'o'
s.remove("A"); // s="HEVEN"; 删除所有字符'A'
s.insert(1, "A"); // s="HEAVEN"; 在序号1 插入字符'A'
s.clear(); // s="" 清除所有内容
```

2.5.5 数组，字典和集合

数组类型的关键字是 *vector.* 以下是操作代码举例.

```
X = 1; y=2; z=3; v = [x,y,z,4]; // v = [ 1,2,3,4]
v[0] = 10; // v = [10,2,3,4]
total = 0;  foreach( element in v ) total += element; // total =10+2+3+4=19
v.sort(); // v= [2,3,4,10] 排序
v.append(11); //v=[2,3,4,10,11] 添加元素到结尾
v2 = v+  [11,12,13]; // v2= [2,3,4,10,11,12,13] 合并
v2.remove(0); // v2 become [3,4,10,11,12,13] 删除序号0 元素
v2.removeobj(10);  // v2 become [3,4,11,12,13] 删除所有值为10 的元素
v3 = [ v2,v2]; // v3 是二维数组 = [[3,4,11,12,13], [3,4,11,12,13]]

//栈类操作, 压入和弹出( push and pop)
x=[1,2];
x.pushfirst(0); // x =[0,1,2] 压入开头
x.pushlast(3); // x=[0,1,2,3] 压入结尾
y=x.popfirst(); // y=0, x=[1,2,3]; 弹出开头
z=x.poplast(); // y=3, x=[1,2] 弹出结尾
```

字典类型的关键字是 *map.* 字典里的查找关键字项不能重复．以下是代码举例.

```
X=1; y=2; z = 3;
 word2number = { "one" : x, "two" : y,  "three" : z }; //构造字典
word2number["four"] = 4; // 插入或替换字典元素
word2nubmer.remove("one"); // 删除字典元素"one"
```

```
word2number.insert("two",2.0); // 字典内容不变, 因为"two" 已在字典里
union = word2number + { "five":5, "six":6}; // 字典合并
foreach(key,val in word2number )
    std::console().writeline(key,' ',val); // 枚举字典内所有关键字及元素值
emptymap = {:};  // 构造空白字典
```

集合类型的关键字是 *set.* 集合里的元素不能重复．集合是等同于只有关键字的字典．以下是代码举例．

```
  one   = { 1, "hello", 3.14 }; //构造集合
another = { 1,2}; //构造集合
union =   one   + another;  // union = { 1,2,"hello",3.14}  合并集合
minus =   one   – another ; // minus = { "hello", 3.14}, 删除集合共同元素
joint =   one   – minus; // joint = {1}, 结果等于 oneset 和 another 的交集
foreach( val in   one   )
        std::console().writeline(val);  // 枚举集合所有元素
emptyset = {}; //构造空白集合
```

数组，字典和集合相互转换举例:

```
  one   = { 1,2,3}; onevec = vector(   one   ); // 集合到数组
keyvec=['a','b']; valvec = ['A','B'];
 onemap = map(keyvec, valvec); // 两个数组构造一个字典
```

2.6 鲁班数据类型通用操作

此节讲述通用于不同鲁班类型的操作.

2.6.1 类算术操作

简单的说，通用的算术操作符象加减乘除都适用于一般的鲁班数据类型．但是同样的运算不同的数据类型有不同的结果．这一点和面向对象模型里的操作符重载一致．如果操作符不适用被操作对象，结果会是错误类型．以下是举例．

```
// + 加法运算
numtwo = 1 + 1;  // numtwo = 2
stroneone = "1" + "1";  // stroneone = "11"
vec = [1]+[1];  // vec = [ 1, 1]
s = { 1 } + { 1 };  // s = {1},  合并集合
m = { 1:10 }+{ 2:20};  // m = { 1:10, 2:20 }, 合并字典
err = "1" + 1; // err = error,错误, 整数和字串不能相加
err2 = true + false; // err = error 错误, 逻辑类型不能相加

// -  减法运算
zero = 1 – 1; // zero = 0
varset = { 1,2,3} – { 3,4,5 };  // varset = {1,2}删除集合共同元素
```

```
err = [1]-[1];  // err = error, 错误, 数组类型不能相减

// -  求负运算
x = -100; // x = -100
vec = - [ 1,2,3]; // vec =[3,2,1] 数组求负等于颠倒顺序
str  = - “abc”;  // str = “cba” 字串求负等于颠倒顺序
truth = -false;  // truth = true 逻辑类型求负等于颠倒真假
err = - ‘a’;  // err = error, 错误, 字符类型不能求负

// *  乘法运算
x = 2*2; // x = 4
str = “ab”*2; // str = “abab” 字串乘整数等于自我复制
vec = [1,2]*2; // vec = [1,2,1,2] 数组乘整数等于自我复制


//  除法运算 / 只适用于数字
x = 100/3;  // x = 33 integer division
xfloat = 100.0/3.0; // xfloat = 33.3333…. float number

// 求余运算 % 只适用于整数
xmod = 100 % 3 ; // xmod = 1

// ++ 加 1 或者求下一个对象
x = 0;
y = ++x;  // x=1, y=1, x 值增加 1 赋予 y
z = y++;  // z =1; y =2; y 赋予 z 然后 y 值增加 1
x = ‘a’;
++x;  // x= ‘b’  ‘a’ 的下一个是 ‘b’
```

--运算符只是++的逆反. 另外一些运算加赋值操作符有以下例子说明.

```
X += 10; // 等同于 x = x+10;
x -= 10; // 等同于 x = x-10;
x /= 10; // 等同于 x = x/10;
x *= 10; // 等同于 x = x*10;
x %= 10; // 等同于 x = x%10;
```

需要指出的是, 运算加赋值操作符比等同的运算再赋值要快很多. 因为步骤简化了.

可以看出, 鲁班的算术运算操作和 C 基本一样, 只是扩展了把非数值类型也包括进来.

2.6.2 枚举循环语句(foreach)
枚举循环语句适用于容器类型, 包括数组, 字典, 集合和字串类型. 将枚举循环用于非容器类型会导致鲁班终止执行当前部件代码. 如下举例:

```
x = 3.1415;
foreach( y in x )    // 错误! 程序在此终止
        std::println(obj=y);
```

以下是一个正确的枚举循环语句例子.

```
Foreach( x in [10,20,30] ) std::println(obj=x);
```

以上代码执行会在屏幕上印出 10,20,30 每个数字一行.

2.6.3 通用成员函数
所有的鲁班数据类型都有成员函数可以调用.具体的鲁班数据类型定义自己的成员函数. 但是以下成员函数适用于全部或者部分数据类型.

Allfuncs()成员函数. 返回所有成员函数名及调用方式.

```
X = [];
funcmap = x.allfuncs();
foreach( funcname, funcdesc in funcmap)
        std::console().writeline(funcname, ' ', funcdesc);
```

以上程序在屏幕上印出所有数组类型的成员函数的名字和说明.

Hash()成员函数, 当对象被加入字典或集合时此函数被调用来决定存放序号.

```
X = "hello";
h = x.hash(); // default value 0
```

invoke()成员函数, 是鲁班反映机制的一部分. 用于动态调用成员函数

```
x = "hello";
h = x.invoke("hash");
```

以上代码的执行结果与直接调用 hash()成员函数一样.

Serialize()成员函数将对象转换为串行字串.

```
X = { "one":1, "two":2 };
xs = x.serialize();
// 以下代码将串行字串恢复为对象.
Samex = std::des(stream=xs).obj;
if ( samex == x )
        std::println(obj="hey, it works");
```

以上代码会在屏幕上印出"hey, it works". 一个小技巧是当把字串恢复成对象时需要调用一个标准部件"std::des". 把串行字串放到"stream"输入属性端, 然后从"obj"输出端取出恢复的对象.

以下成员函数只适用于字符串和容器类型.

Size()成员函数: 返回容器内元素个数或字符串长度.

X = [10,20,30]; s = x.size(); // s=3 x 内有 3 个元素

contains()成员函数: 检查容器内是否含有指定元素, 或字符串是否有指定子字符串.

X = [10,20,30]; has20 = x.contains(20); // has20=true x 内有元素 20

h="Hello"; hasHell=h.contains("Hell"); // hasHell=true h 含有子串"Hell"

clear()成员函数: 清除容器所有元素或字符串内所有字符.

X = [10,20,30]; x.clear(); // x = [];

h="Hello"; h.clear(); // h="";

最后需要指出的是, 所有成员函数调用, 如果出错, 都会返回错误类型对象.

2.7 空值 null 和错误类型 error

空值 null 是一个特别的常数值, 由关键字 null 代表. 空值 null 可以象一般的常数一样用在表达式里, 就象如下例子.

Thisarray = [null, null]; //定义一个数组有两个空值元素

空值 null 的特别之处有两点. 首先所有变量和部件属性在没有被赋值之前都是空值 null. 第二是空值 null 不属于任何类型.

错误类型 error 是一个特别的类型. 所有鲁班表达式和操作当错误发生时都会返回错误类型值. 在错误类型对象上施加任何操作都会依然返回错误类型. 鲁班代码也可以直接构造错误类型对象. 以下是例子.

```
Errval = 1/0;  // 产生一个被零除错误
if (  errval isa error )
    std::console().writeline(errval);
expliciterr = error("Hey, I am a born error");
std::console().writeline(expliciterr);
```

运行以上代码会在屏幕上印出:

ERROR: Error for +-*/% operation: Divided by zero

ERROR: Hey, I am a born error

2.8 高级鲁班脚本技术: 多线程并行及协调

多线程并行及协调是鲁班语言的一部分. 在鲁班语言里启动线程和等待线程非常简单, 就如下例所示

```
myjobs::dothis(arg=100)  & // 启动一个线程运行 myjobs::dothis
myjobs::dothat(arg =200) & //启动另一个线程运行 myjobs::dothat
{ c.doOne();  d.doTwo(); } & // 把花括号里的语句放在一个线程里启动运行
```

```
wait; // 等待所有的线程结束
```

你也可以选择性地等待线程结束. 鲁班线程可以用线程相关变量来辨别. 鲁班线程相关变量定义为线程代码中最后一个赋值语句的左变量. 以下例子说明怎样用线程相关变量来辨别和等待鲁班线程.

```
X = y.domagic() &  // 启动一个线程, 相关变量 x
 a = b.dogood() & //启动另一个线程, 相关变量 a
waitfor x;  //等待线程 x 结束

{ x = y.domagic(); a = b.dogood(); } & //启动一个线程, 相关变量 a
{ c = d.doevil(); f = g.wastetime(); } & //启动一个线程, 相关变量 f
waitfor f; //等待线程 f 结束
```

对于多个鲁班线程操作的同一个变量, 鲁班保证任何时间只有一个线程在操作同一个变量. 这样可以保证变量内的对象的完整性不至于被多线程同时操作破坏.

另一个问题是如果鲁班脚本结束时怎样处理其他可能正在运行的子线程. 鲁班的处理办法是等待所有的子线程结束才退出. 就象有一个 wait 语句在结尾一样.
另一个线程相关的语句是 cancel 语句, 用来取消所有已启动的线程.

2.9 线程陷阱举例
例子一:

```
for(i=0;i<10000:++i) { x += i; }  &
```

以上貌似简单的程序几乎可以肯定会失败. 因为它会试图启动 10000 个线程. 很少有机器可以在一个进程内启动 10000 个线程. 要把一个循环放入一个线程的正确写法是如下:

```
{ for(i=0;i<10000:++i) { x += i; } }  &
```

例子二:

```
for( i=1; i<=10; ++i )
{
        std::println(obj=i) &
}
```

第一眼看起来, 上面程序应该在屏幕上印出 1,2,3,4,5,6,7,8,9,10 每个数一行. 实际上以上程序只保证印出十个一到十的数. 但究竟是哪十个数顺序怎样则不确定. 具体原因是这样. 这个程序会启动十个线程来打印变量"i"的内容, 由主线程循环改变变量"i"的内容, 每次改变后启动一个线程. 这十加一个线程的执行顺序是不确定的. 所

以不一定会印出 1,2,3,4,5,6,7,8,9,10 顺序. 小测验: 这个程序会印出 1,1,1,1,1,1,1,1,1,1 十个”1”吗?
怎样更好处理类似问题在后面章节还有讲述.

2.10 鲁班简单的世界: 一切都是对象
鲁班与其他面向对象语言象 C++/JAVA 的一大区别是鲁班没有指针(pointer)和位址(reference)类型. 所有变量都只能存储对象本身. 每次赋值都等同与一个新的对象. 就如以下代码所示:

```
a = [1,2,3];
b = a;  // b 得到一个新的对象
b += [4,5]; // b = [1,2,3,4,5]  a = [1,2,3] b,a 现在不同值
```

鲁班的”一切都是对象”设计主要是为了简单性. 增加位址类型会增加对用户的难度. 另外一个很重要的理由是简单对象传递让鲁班部件之间的交互变得更简单和具有可变性, 比如说可以网络远程调用. 细节以后章节会有叙述.

3. 鲁班部件入门

The least flexible component of any system is the user.
- Lowell Jay Arthur

这一章讲述鲁班部件模型的基本构造. 鲁班部件在鲁班语言里的关键字是 *struct* 和 C 语言里的 *struct* 关键字一样. 下面大家可以看见鲁班部件和 C 语言的 *struct* 有某种相似之处. 这一章主要通过例子来讲述怎样构造和使用鲁班部件. 更多的细节和更正式的定义在下一章还有叙述.

3.1 鲁班部件构造

设想我们的"世界, 你好"程序变得很流行, 大家都想用. 怎样让大家可以共用这个程序呢? 鲁班的办法很简单, 就是把这个鲁班脚本程序变成一个鲁班部件. 看如下例子:

```
namespace demo;   //指明所属名字空间

struct helloworld( ) //部件名字定义
as process           // 部件是一个过程
{
        std::println(obj="Hello, world!"); // 原来的脚本代码
}
```

把以上代码敲入一个叫"hello.lbn"的文件里. 然后我们在另一个叫"callhello.lbn"的文件里输入以下鲁班脚本程序:

```
demo::helloworld(=);
```

然后在你的计算机的操作系统命令行打入以下命令启动鲁班执行这个程序.

```
Mycomputer > luban hello.lbn callhello.lbn
Hello, world!
```

在以上例子里, 在"hello.lbn"文件里我们定义了我们的第一个鲁班部件*"demo::helloworld"*. 让我们一行行看这个鲁班部件是怎么定义的. 第一行是指明我们这个部件所属于的名字空间*"demo"*. 所有鲁班部件都必须属于一个名字空间. 所以鲁班部件定义代码的第一行必须指明所属名字空间. 这一点和 Java 相似. 下一行用 *struct* 关键字表明这是一个鲁班部件名字叫做"*helloworld"*. 再下一行的"*as process"*表明这个鲁班部件是一个过程. 鲁班部件可以是一个过程或者组合. 以后章节会讲述什么是组合. 这里大家只要知道到目前为止讲的所有的鲁班程序脚本都是过程就行了. 最后我们把原来的"世界, 你好"脚本程序代码放到一对花括号里, 这个全名叫做*"demo::helloworld"*的鲁班部件就算定义好了.

可以看出, 将鲁班脚本变成鲁班的部件是很简单的. 基本上就是两点, 一是指明一个所属名字空间, 二是给它起个名字. 然后把脚本代码放到一对花扩号里就好了.

在"callhello,lbn"文件里, 我们只是写了一个一行的鲁班脚本程序来调用我们刚定义的鲁班部件*"demo::helloworld"*. 大家可能已经注意到, 调用鲁班部件类型的语法有点特别, 象*"demo::helloworld(=)"*. 其中细节下面会讲述, 这里大家只要知道*"demo::helloworld(=)"*意思是调用鲁班部件*"demo::helloworld"*就行了.

3.2 鲁班部件的输入和输出属性

在上一节的例子里, 我们定义了一个鲁班部件*"demo::helloworld"*. 这个部件只在屏幕上印出"Hello, World!". 调用时不需要任何输入, 也不产生任何输出. 在实际应用中, 一般的鲁班部件会接受输入并产生输出. 下面是一个简单的带输入和输出属性的鲁班部件.

```
namespace demo;

struct greeting
(
        input:
                saywhat, towhom;
        output:
                greetingmsg;
)
as process
{
        msg = input.saywhat + ", " + input.towhom + '!';
        std::println((obj=msg);
        output.greetingmsg = msg;
}
```

以上鲁班代码定义了一个鲁班部件 *"demo::greeting"*. 这个部件有两个输入属性*"saywhat", "towhom"*分别是"说什么"和"对谁说"的意思. 这个部件的输出属性是*"greetingmsg"*. 在这部件内部是一个过程(process). 这个过程将两个输入属性连接成一个标准的问候语, 然后放在*"greetingmsg"*输出属性里. 在这过程中还把产生的问候语打印在屏幕上.

大家可以看见, 上面这个例子和前一个例子不一样的地方是, 上一个例子在部件名字后的扩号里没有内容, 而上面这个例子在扩号里有输入和输出属性的定义. 在部件过程的定义部分, 鲁班总是用 *input.name* 格式表示输入属性, 用 *output.name* 表示输出属性.

现在我们把上面鲁班代码存入到一个叫"iohello.lbn"的文件里. 再写一个小脚本程序来调用这个叫做*"demo::greeting"*的部件, 脚本程序代码如下:

```
demo::greeting(saywhat="Hello", towhom="you");
```

把以上鲁班脚本程序存入一个叫 ”calliohello.lbn”的文件里, 然后在操作系统命令行打入如下命令执行我们编好的鲁班程序.

```
Mycomputer > luban iohello.lbn calliohello.lbn
Hello, you!
```

这个鲁班程序执行时会运行在”calliohello.lbn”里的脚本程序, “calliohello.lbn”会调用鲁班部件*”demo::greeting”*, *“demo::greeting”*被调用时回打印出问候语.
再回头看调用”demo::greeting”的鲁班脚本代码:

demo::greeting(saywhat=”Hello”, towhom=”you”);

从这行代码可以看出调用鲁班部件时, 输入属性的设置是通过把属性值赋予属性名字来实现的. 这一点和 C/C++/JAVA 都不一样. 只有 PYTHON 的函数调用有类似的语法.

3.3 鲁班部件的多种用法

上一节讲述的例子是定义一个鲁班部件, 然后象调用 C 函数一样来调用. 需要指出的是除了能象函数一样调用外, 鲁班部件还有其他多种用法. 首先鲁班部件可以是一个对象, 它的输入和输出属性是构成鲁班部件对象的状态. 这和 C 函数完全不一样. C 函数可以说是没有状态的. 鲁班部件对象可以象一般数据对象一样传递和复制, 其输入输出属性可以被外界读写. 下面例子列举鲁班部件的调用方法.

//第一行
greeter = demo::greeting(saywhat=””, towhom=””); //调用并建造鲁班部件
//第二行
greeter.saywhat = “Hello”; // 写输入属性, 印出 Hello,!
//第三行
greeter.towhom = “world”; //写输入属性, 印出 Hello, world!
//第四行
greeter(saywhat=”Nice to meet you”); // 部件对象动态调用,印出 Nice to meet you, world!
//第五行
greeter(=); //部件对象动态调用,印出 Hello, world!
//第六行
greeter.(saywhat = “How are you”, towhom=”Luban”); //多属性设置, 印出 How are you,Luban!
//第七行
greeter.(=); //空白部件多属性设置,印出 How are you,Luban!
//第八行
msg = greeter.greetingmsg; //读部件对象输出属性
//第九行
std::println(obj=msg);// 印出 How are you, Luban!

把以上代码存入一个叫”allhello.lbn”的文件里. 然后打入如下命令运行这段鲁班脚本:

```
Mycomputer > luban iohello.lbn allhello.lbn
,!
Hello,!
Hello,world!
Nice to meet you,world!
Hello,world!
How are you, Luban!
How are you, Luban!
How are you, Luban!
```

下面我们逐行解释这个鲁班脚本程序:

1. 第一行是鲁班部件对象调用加建造, 它的语法和前面已经讲过的部件类型调用基本一样. 唯一不同的是此处我们把调用结果赋予一个叫"greeter"的变量. 鲁班部件类型调用的结果是产生一个新的鲁班部件对象. 这个新的鲁班部件对象的属性会被设置成调用者所指定的值, 然后鲁班会运行这新的部件再将这个新部件对象返回给调用者. 此处我们把返回的鲁班部件对象赋予变量"greeter". 此处这个新鲁班部件运行时会在屏幕上打印问候信息. 因为我们在"saywhat", "towhom"输入属性设置的都是空字符串, 所以打印出来的问候是", !". 只有标点符号没有词语.

   要点: 鲁班部件类型调用不但会运行被调用的部件类型, 还会产生新的部件对象.

2. 第二和第三行是鲁班属性赋值. 第二行把字符串"Hello"赋给"saywhat"属性. 第三行把字符串"world"赋给"towhom"属性. 每次设置输入属性值的时候, 鲁班部件都会被自动运行一次. 第一次运行时, 属性"saywhat"被赋予字符串"Hello", 但属性"towhom"还是原来的空字符串, 所以打印出"Hello, !". 第二次运行时"towhom"被赋予了字符串"world", 所以打印出"Hello, world!"

   要点: 每次设置鲁班部件对象输入属性值时, 都会自动引发鲁班部件执行.

3. 第四和第五行都是鲁班部件对象动态调用. 可以看出鲁班部件对象动态调用的语法和鲁班类型调用是基本一样的. 不同的只是这里使用的是对象变量名而不是鲁班类型名. 第四行调用时把"saywhat"设置成"Nice to meet you", "towhom"属性则没有改变, 所以鲁班部件运行时打印出"Nice to meet you, world!". 需要指出的是鲁班部件动态调用时也会复制自己, 然后运行, 最后返回新的部件对象. 就和鲁班部件类型调用时一样. 调用时设置的属性值只是对新复制的部件, 对原部件并没有影响. 所以在第五行的另一个鲁班部件对象动态调用时, 我们没有设置任何属性值(在括号里只有等号=表示不另外设置属性值), 打印出的内容是原来的"Hello, world!". 我们在这里没有把鲁班返回的新对象赋予变量, 而是简单放弃了.

要点: 鲁班部件对象动态调用时传递的属性值对原部件对象没有影响, 因为鲁班会复制一个新的部件对象来运行. 传递的属性值只设置在新的对象. 所以原来的鲁班部件对象属性不变.

4. 第六行是一次多个属性设置. 以上说过每次设置部件的输入属性值时都会引发部件运行. 有时侯我们想设置多个输入, 但是我们只需要部件在所有属性设置完以后运行一次. 这种情况就需要做多个属性一次性设置. 在这行代码我们把"greeter"的输入属性"saywhat"设置成"How are you", 把"towhom"设置成"Luban". 虽然有两个属性设置, 但是"greeter"部件只运行一次. 所以打印出一行"How are you, Luban!". 多个属性一次设置的语法和鲁班部件调用的语法几乎一样. 唯一的区别是多个属性一次设置在部件对象名字和括号之间有一个句点号(DOT).

   要点: 多个属性一次设置只会引发鲁班部件执行一次.

5. 第七行是一个空白的多属性设置. 虽然没有任何属性被赋值, 这个鲁班部件仍会被执行一次. 所以又打印一个"How are you, Luban". 请注意虽然"greeter(=)"和"greeter.(=)"都会引发部件执行. 但它们一个是鲁班部件调用*"greeter(=)"*, 调用时会先复制部件, 然后执行. 另一个*"greeter.(=)"*是部件属性设置, 不会复制新对象, 只执行原有的对象. 这一行代码也验证我们上一行的多个属性设置是成功的.

   要点: 如果要只执行鲁班部件而不要复制的新鲁班部件对象, 用属性设置, 不要用鲁班部件对象调用. 空白多属性设置也会引发鲁班部件执行.

6. 第八是读取鲁班部件属性值. 此行读取鲁班部件对象"greeter"的属性"greetingmsg"的值, 赋值给变量"msg". 如果你回到鲁班部件"demo::greeting"的定义, 你可以看见它除了打印问候语外, 还将问候语字符串放在输出属性"greetingmsg"里. 所以我们可以在这里读取这个属性值.

7. 第九行把变量"msg"的值打印出来. 变量的值是"How are you, Luban!". 所以此次打印出地三个"How are you, Luban!".

从以上例子, 我们可以看见你可以对鲁班部件做的操作有:

- **设置部件属性值. 可以用单个或多个属性设置语法. 设置属性值会引发鲁班部件执行.**
- **读取鲁班部件属性值.**
- **调用鲁班部件. 可以是直接鲁班部件类型调用, 或者是鲁班部件对象调用. 调用时可以从新设置属性值. 调用会复制新的鲁班部件对象. 从新设置的属性值对原鲁班部件对象没有影响.**

3.4 鲁班的传值调用规则
在调用鲁班部件时, 属性参数设置传递的一定是数据对象值. 鲁班语言里没有指针和位地址类型. 这一点和 C++/Java 都不一样. 写惯 C 的人可能会写象下面的代码:

```
namespace demo;

struct NotWorkingSwapObj
(
        input:
                x,y;
)
as process
{
        temp = input.x;
        input.x = input.y; // 此处语法错
        input.y = temp; //此处语法错
}
```

以上代码想交换两个变量的内容. 但是鲁班会报告语法错误. 错误是试图给输入属性从新赋值. 鲁班不允许在部件定义内部给输入属性赋值. 给输入属性赋值只能在部件外部进行.

3.5 鲁班脚本程序和鲁班部件定义程序
到现在我们已经看见过两类鲁班程序. 一是鲁班脚本程序. 鲁班脚本程序是简单的鲁班语句和表达式序列. 当碰到鲁班脚本程序, 鲁班解释器会立即运行这脚本程序.
另一种是鲁班部件定义程序, 就象我们上面看见的”demo::greeter”部件定义. 部件定义总是以宣布所在名字空间开始, 因为所有鲁班部件都必须属于一个名字空间. 然后部件定义会指明部件名字, 输入输出属性序列, 和内部实现代码. 鲁班解释器处理鲁班部件定义时, 只是把部件定义存入内部名字空间里, 然后等待调用或建造. 不会直接运行这鲁班部件.
一般的使用方法是用鲁班脚本作为程序起点. 而用鲁班部件定义程序来组成部件库.

4. 鲁班部件组合入门

Poor boy, pickin' up sticks
Build ya a house out of mortar and bricks
- Po' Boy, Bob Dylan

到目前为止, 我们看到的鲁班代码都是顺序执行的过程代码. 在第一章里提过, 鲁班的一个重要特色是部件组合. 部件组合就是用小的鲁班部件连接组合成更大的鲁班部件. 连接组合与顺序调用不同, 连接组合是将要用的部件列举出来并定义它们之间数据依赖关系. 组合的鲁班部件执行的时候, 里面的子部件的执行顺序是由当时的数据更新的流向决定的.
鲁班部件组合与 EXCEL 很相似. 在 EXCEL 里每个格子里要不是个常数, 要不是由一个公式. 公式可以从其他格子里的值计算出当前格子的值. 所以每个公式格子都定义了自己和其他格子的关系. 如果你改变一个格子里的值, 所有直接或间接依赖于这个格子的其他格子都会被从新计算赋值. 计算顺序由依赖关系决定. 而计算的执行则由数据更新引发.
鲁班的组合部件的外部界面与鲁班过程部件没有区别. 所以它们可以混合使用. 你只需要知道部件的界面, 不需要知道它是用组合还是用过程实现的.

4.1 简单组合举例

鲁班部件组合代码格式很简单, 就是一个组合单元列表. 每个组合单元有一个名字和内容定义. 一个组合单元和其他组合单元的数据依赖关系是有它的内容定义决定的. 一个组合单元类似于一个 EXCEL 的格子. 下面是一个简单的组合例子.

```
namespace demo;

struct simplecomp ( output oneoone; )
as composition
{
        A1: 100;
        A2: A1+1;
        output.oneoone: A2;
}
```

以上鲁班组合里有三个组合单元. 单元 A1 里是一个简单常数 100, A2 里是一个表达式"A1+1". 最后一个单元的名字有点特别, 但它的意思很明确, 就是将单元 A2 的值放到输出属性"oneoone"里.
下面一行鲁班代码调用上面定义的鲁班部件"demo::simplecomp".

```
result = demo::simplecomp(=).oneoone; // result = 101
```

以上代码是标准的鲁班部件类型调用, 而且在调用后取出属性"oneoone"的值赋值给变量"result". 变量"result"的值是整数 101.

为了显示组合与过程的区别, 我们可以把”demo::simplecomp”部件的代码改成以下.

```
namespace demo;

struct simplecomp ( output oneoone; )
as composition
{
        output.oneoone: A2;
        A2: A1+1;
        A1: 100;
}
```

新的代码里组合单元的顺序是完全颠倒原来的顺序. 如果我们再调用这个部件, 结果却完全不变. 从这儿我们可以看出, 组合部件里单元的顺序完全不重要. 重要只的是单元之间的数据依赖关系.

4.2 组合单元详述

鲁班组合部件里的单元有两类, 一是定型单元, 一是不定型单元. 这一节里我们详细解说这两类单元的定义和用途.

4.2.1 不定型单元

到现在我们见过的鲁班部件组合里的单元都是不定型单元. 鲁班组合不定型单元里可以是常数, 也可以是表达式, 就象我们上面的例子一样. 不定型单元和 EXCEL 里的格子可以说很相似. 除了有一点, 鲁班的不定型单元里还可以放任意的鲁班脚本程序代码. 这一点是 EXCEL 做不到的. 下面是一个例子.

```
namespace demo;

struct compadhoc
(
        input    in1, in2;
        output  out;
)
as composition
{
        A1: input.in1 + 1;
        A2: input.in2 + 1;
        A3: { std::println(obj="run A3"); A3=A1*A2; }
        A4: { std::println(obj="run A4"); A4=A2*A2; }
        output.out: A3+A4;
}
```

以上组合代码里的单元都是不定型单元. 单元 A1 和 A2 里都是简单表达式, 在运行时鲁班会运算表达式然后把结果放在单元里.

单元 A3 和 A4 也是不定型单元, 里面放的是鲁班脚本程序. A3 和 A4 里的程序相似, 都是先在屏幕上印出自己运行的信息, 然后给自己运算赋值. 需要注意的是脚本程序单元和表达式单元不同之处. 表达式单元的运算结果直接赋值给单元自己. 脚本程序单元没有明确单一的运算结果, 给自己单元赋值必须由明确的赋值语句执行. 在 A3 单元里给自己赋值的语句是 *"A3=A1*A2"*, 这句的意思是把单元 A1 的值乘以单元 A2 的值然后赋值给单元 A3 自己. 在单元 A4 里给自己赋值的语句是*"A4=A2*A2"*, 是将单元 A2 的平方赋值给 A4 自己. 值得注意的是在不定型单元的定义里, 对其他单元的引用等于建立了当前单元和被引用单元之间的数据依赖关系. 鲁班在运行这个部件时就会按这样的依赖顺序和实际的数据更新来决定那个单元会被执行.
上面代码的最后一个单元是将表达式*"A3+A4"*连接到输出属性*"out"*.

为了简单起见, 鲁班部件组合的输出属性单元, 象以上例子的"*output.out*"单元, 只能是表达式, 不能是脚本程序.

4.2.2 定型单元
定型单元里放的是已经定义好的鲁班部件. 用户只需要定义单元里的部件的输入和输出属性的连接. 下面是一个例子.

```
namespace demo;

struct TwoAdder
(
        input op1, op2;
        output result;
)
as process
{
        output.result = input.op1+input.op2;
}

struct FourAdder
(
        input op1,op2,op3,op4;
        output result;
)
as composition
{
        adder1:  demo::TwoAdder(op1=input.op1, op2=input.op2);
        adder2: demo::TwoAdder(op1=input.op3, op2=input.op4);
        adder3: demo::TwoAdder(op1=adder1.result, op2=adder2.result);
        output.result: adder3.result;
}
```

上面的例子是用两输入的加法器部件”demo::TwoAdder”来组合成四输入的加法器”demo::FourAdder”. 我们来看看其中的构造.
程序开始定义了一个简单的加法器部件 ”demo::TwoAdder”. “demo::TwoAdder”的功能是把两个输入属性 ”op1”, “op2” 的值加起来放到输出属性 ”result” 里 . “demo::TwoAdder”是一个过程部件.
关键的部分是部件 ”demo::FourAdder”的构造 . “demo::FourAdder”是一个组合部件 . 里面有三个定型单元”adder1”, “adder2”, “adder3”. 三个单元放的都是上面已经定义好的 ”demo::TwoAdder” 部件 . 其中 ”adder1” 和 ”adder2” 的输入连接到”demo::FourAdder”的四个输入上 . “adder3”的输入连接到”adder1”和”adder2”的输出. 最后将”adder3”的输出连接到”demo::FourAdder”的最终输出属性”result”上.
这个例子用三个两输入加法器组合成一个四输入加法器 . 用鲁班的部件组合构造可以很简明的定义组合所用部件和它们之间的数据依赖关系.
另外值得注意的是定型单元和不定型单元的区别. 当使用不定型单元时, 其他单元引用其中包含的数据对象值本身. 而使用定型单元时, 其他单元引用其中包含的鲁班部件的输出属性.

4.3 组合部件内部执行顺序
鲁班组合部件与过程部件的一大区别是组合部件定义的是内部组合单元之间的数据依赖关系, 而不是执行顺序. 组合单元的真正执行顺序是由运行时的输入数据更新决定的. 这一节就讲述具体的细节.

还是拿上面用过的例子 demo::compadhoc (稍加修改) 来讲.

```
namespace demo;

struct compadhoc
(
        input    in1, in2;
        output  out;
)
as composition
{
        A1:{ std::println(obj=“run A1”); A1= input.in1 + 1; }
        A2: { std::println(obj=“run A2”); A2= input.in2 + 1; }
        A3: { std::println(obj=“run A3”); A3=A1*A1; }
        A4: { std::println(obj=“run A4”); A4=A2*A2; }
        output.out: A3+A4;
}
```

把上面代码存入文件 compadhoc.lbn 然后我们调用这个鲁班部件 demo::compadhoc . 调用的鲁班脚本代码是:

```
x = demo::compadhoc(in1=1, in2=2); // 第一行
```

```
std::println(obj=x.out);   // 第二行
std::println(obj="change input in1"); // 第三行
x.in1=0; // 第四行
std::println(obj=x.out); // 第五行
std::println(obj="change input in2"); // 第六行
y =x(in2=1); // 第七行
std::println(obj=y.out); // 第八行
```

把以上代码存入一个叫 start.lbn 的文件. 然后启动执行:

```
Mycomputer > luban compadhoc.lbn start.lbn
run A1
run A2
run A3
run A4
13
change input in1
run A1
run A3
10
change input in2
run A2
run A4
5
```

以下逐行解释以上程序的运行输出.

- start.lbn 的第一行是调用部件 demo::compadhoc, 调用时把输入 in1 设置成 1, in2 设置成 2, 最后把调用是产生的新部件对象赋值给变量 x.. 调用时鲁班运行部件 demo::compadhoc, 运行时打印出:
  ```
  run A1
  run A2
  run A3
  run A4
  ```
  在 demo::compadhoc 部件内部, 数据流依赖关系是这样定义的:
  input.in1 → A1 → A3 –
  |→ output.out
  input.in2 → A2 → A4 --

从上面的数据流看, A1 一定在 A3 之前, A2 一定在 A4 之前. 上面的程序输出"`run A1 run A2run A3 run A4`"是满足数据流的要求的.

另外有一点需要注意的是, 对于没有数据依赖关系的单元,它们的执行顺序关系是没有定义的. 比如上面例子里的 A1 和 A2, A3 和 A4 之间都没有依赖关系. 所以 A1 和 A2 哪个先执行, A3 和 A4 哪个先执行都是没有定义的. 换句话说, 如果运行结果是"`run A2 run A1 run A4 run A3`", 这也是符合鲁班语言定义的.

这个鲁班部件的功能是计算 `(in1+1)x(in1+1) + (in2+1)x(in2+1).` 所以之后打印的结果是 `13 (in1=1, in2=2)`

- 第四行是改变鲁班部件 x 的输入 in1 的值成 0. 这里请读者注意, 鲁班组合部件和过程部件的最大区别就在这里. 记得以前讲述过, 如果部件的输入改变会引发部件的运行. 鲁班组合部件也是这样. 但是和鲁班过程部件不一样的是, 鲁班过程部件运行时必须运行整个过程, 而鲁班组合部件只运行那些和被改变的输入有数据依赖关系的单元. 换句话说, 鲁班组合部件只执行从被改变的输入起的数据流下游的组合单元.
  在这个例子里, 输入 in1 的数据流下游的组合单元有 A1 和 A3, 所以执行时打印出"run A1 run A3". 执行后输出结果变成 10, 所以在第五行代码打印输出时印出 10.

- 第七行是动态调用鲁班部件对象 x 并把输入 in2 的值设置成 1. 和上面讲述的相似, 动态调用时鲁班组合部件只执行从被改变的输入起的数据流下游的组合单元. 在这里, 输入 in2 的数据流下游的组合单元有 A2 和 A4, 所以执行时打印出"run A2 run A4". 执行后输出结果变成 5, 所以在最后一行代码打印输出时印出 5.

从以上例子可以得出以下结论:

1. 鲁班组合部件的内部单元执行顺序由数据流依赖关系决定. 彼此之间没有数据依赖关系的单元, 则没有明确定义的顺序关系.
2. 改变组合部件的输入属性或者动态调用组合部件对象, 只引发被改变的输入起的数据流下游的组合单元执行.

鲁班组合部件的根据输入属性改变来决定部分执行组合单元的特色 , 可能对于很多应用环境可能很方便.

4.4 数据流定义规则

在编写鲁班组合部件时, 用户可以将表达式, 鲁班脚本程序和已定义的鲁班部件放到组合单元里. 鲁班语言会自己找出单元之间的数据流依赖关系来决定运行的单元和它们之间的顺序. 下面讲述在定义组合单元时必须遵守的规则.

4.4.1 能改变自己不能改变外界

对于鲁班组合部件里的单元来说, 外面的其他单元都是只能读不能写的. 一个单元可以改变的是自己的内容 . 如果一个单元试图去改变其他单元的内容 , 鲁班会报错 . 直接给单元赋值 , 调用单元的成语函数和改变单元属性都是改变单元内容的操作. 不能改变外界只能改变自己的规则是为了保证鲁班组合部件里的数据流是简明单向的. 在一个单元里改变其他单元的内容会使数据流变得复杂和难以理解. 下面是一个例子.

```
namespace demo;

struct WrongComp()
```

```
as composition
{
    A1: 1;
    A2: { A1=0; A2=A1+1; }
}
```

把以上程序存入一个文件 wrongcomp.lbn 里，然后用以下命令启动执行.

```
Mycomputer > luban wrongcomp.lbn –s “demo::WrongComp(=);”

Error when executing Luban code from stdin:
Failed to resolve external symbols for struct evaluation:
Can not set cell content other than self: A1
demo::WrongComp has error and can not be used
Can not resolve external symbol demo::WrongComp
```

上面鲁班程序运行时显示出错误信息说，不能给除了自己以外的其他单元赋值

4.4.2 数据流不能循环

鲁班组合部件不允许在同步单元之间存在循环数据依赖关系. 同步单元和异步单元的区别以后还会讲述. 在这里大家只要知道，到目前为止讲述的所有单元都是同步单元就好了. 下面是一个循环依赖关系的例子.

```
namespace demo;

struct CyclicComp()
as composition
{
    A1: A2+1;
    A2: A1-1;
}
```

上面程序里的单元 A1 和 A2 相互引用, 成为循环依赖关系. 这个鲁班组合部件是不合法的. 把以上程序存入一个文件 cycliccomp.lbn 里，然后用以下命令启动执行.

```
Mycomputer > luban cycliccomp.lbn –s “demo::CyclicComp(=);”
ERROR: Struct evaluation exception: Failed to initialize composition struct for evaluation: Invalid composition, cyclic path involving:  A2 A1
```

鲁班报错说有循环依赖关系, 而且鲁班还列举了循环圈里的单元. 这个例子里单元 A1 和 A2 互相引用, 所以上面信息里说 A2 和 A1 组成循环.

4.5 过程与组合的选择

现在大家知道定义一个鲁班部件可以有两种方式, 一是传统的过程, 二是用组合. 组合方式很多人其实在用 EXCEL Spreadsheet 时就是在用组合方式编程序. 鲁班的组

合部件是又向前进了一大步. 相比 EXCEL Spreadsheet, 组合的鲁班部件有明确的界面可以重用和共享.
一般来说, 如果一个部件内部有复杂的控制结构, 比如循环, 比较, 跳转等, 这样的部件应该用传统的过程代码来编写. 一般来说, 这样的情况在相对较小的部件里比较常见.
如果一个部件内部可以看成一些相对独立的单元, 而在单元之间有明确的数据流依赖关系. 这样的部件就很适合用组合来编写. 一般来说, 在高层的部件里这样的组合比较常见.
还有可能是纯粹是用户个人选择. 比如说, 一个用惯了 EXCEL Spreadsheet 的人会发现组合方式编程很自然.

4.6 部件界面继承和异步单元
鲁班组合里的还有两个很有用的特色会在后面相关章节讲述. 其中和部件界面继有关的构造会在下一章讲述. 和多线程有关的异步单元会在讲述异步部件的第七章讲述.

5. 鲁班部件界面定义及继承详述

Different towns, different people,
Somehow they're all the same.
- Long Time Gone, Bob Dylan

用鲁班编程序有点象搭 LEGO 塑料积木. 先把积木块放到盒子里, 这样当你需要零件的时候知道去盒子里去找. 这一点和鲁班的名字空间很相似, 名字空间就是放鲁班数据和部件类型的盒子.
搭 LEGO 积木的时候, 你还得认识每块积木的连接面的形状, 才能把不同的积木接起来成大的构造. 鲁班部件界面定义是为了同一个目的, 就是将部件和外界的连接方式明确规定出来.
搭 LEGO 积木的时候, 你可能还会发现, 很多的积木的连接面是一样的. 所以把同样连接面的积木归为一类可能会对认识和使用不同的积木会有帮助. 鲁班部件界面继承也是同一个道理. 界面继承可以很好的反映不同部件之间界面的相似性. 这样一来界面类似的部件可以互换, 而且使用部件的人可以不用知道部件具体是什么, 只要知道部件界面就可以了.

4.1 界面属性的流向和读写权限
在前一章的例子里我们已经看见, 鲁班部件的界面由一系列的属性构成. 我们也已经看见属性有输入和输出区别. 现在我们可以更正式的定义, 所有属性都有一个**流向**. 我们已经见过的流向有 ***input 输入***和 ***output 输出.*** 鲁班的属性流向还有还有两类: ***store 存储***类, 和 ***static 全局***类.

此章未完待续…..

9. 输入输出对象类型及部件

Men have become the tools of their tools.
- Henry David Thoreau

在鲁班软件包里包括一些常用的对象类型和部件用于文件和网络的数据输入和输出. 虽然严格说来这些常用工具并不是鲁班语言的组成部分, 就象 *printf* 函数不是 C 语言的一部分一样. 但是所有 C 语言的书都会讲 *printf*. 我们在这里也专用一章讲这些实用的类型和部件.

9.1 终端, 文件和网络 SOCKET

这三个对象类型都用于数据输入输出, 而且它们的界面基本一样. 以下讲述它们的应用细节及举例.

鲁班类型名:

终端:	std::console
文件:	std::file
网络 Socket:	net::socket

以上三个对象类型共同的成员函数列举如下.

write(obj1,obj2…)
　　将对象逐个转化成字符串然后印出或传出
　　返回空值 null, 如果操作失败则返回错误类型

writeline(obj1,obj2…)
　　将对象逐个转化成字符串然后印出或传出, 每个对象后附加一个换行符.
　　返回空值 null, 如果操作失败则返回错误类型

read(int n=1)
　　读入指定数量(n)的字符. 如果 n 没有指定则读一个字符.
　　返回读取的字符串. 如果操作失败则返回错误类型.

readline()
　　逐个读取字符一直到碰到一个换行符.
　　返回读取的字符串. 如果操作失败则返回错误类型.

writeobj(obj1,obj2…)
　　将对象逐个转化成独立于硬件及操作系统的串行流然后写出或传出
　　返回空值 null, 如果操作失败则返回错误类型

readobj()
　　读取串行流内容, 并将串行流恢复成数据对象. 一般情况是串行流是
　　以前的 *writeobj* 操作产生.
　　返回所恢复的数据对象.

一般来说, *write* 和 *writeline* 用来把对象写成字符串显示来给人读 . 而 *writeobj* 和 *readobj* 用来在程序或机器之间传递数据. 不管简单或复杂的数据对象, 鲁班语言保

证用 *writeobj* 写出的对象一定可以用 *readobj* 来读回来. 而且写和读可以在不同的程序甚至机器上.

9.2 文件读写举例
一个一般文字处理的例子:

```
linecount = 0;
wordcount = 0;
txtfile = std::file("mydatafile", 'r'); // 打开一个叫"mydatafile"的文件
alldata = txtfile.readall(); // 读取全部文件内容到 alldata 变量
lines = alldata.split('\n'); // 将 alldata 分行到变量 lines
foreach( line in lines )  // 枚举在 lines 里的每一行
{
        ++linecount;  //增加行数
        words = line.split();   // 将每行分成单词数组 words
        wordcount += words.size(); // 增加单词数
}
//下面打印结果
std::println(obj="lines: "+string(linecount)+" words: "+string(wordcount));
```

以上程序打开一个叫 "mydatafile"的文件, 将文件全部内容读到一个叫 "alldata"的变量里. 然后将内容分行, 将每行再分成单词, 数行数和单词数. 最后将结果打印在屏幕上. 请注意以上代码用了一个新的成员函数 *readall()*这个函数一次将文件的所有内容读出成一个字符串返回.

下面例子代码演示怎样将数据对象以串行流格式写入文件 , 然后怎样再从文件恢复数据对象.

```
fw = std::file("objfile", 'w'); // 打开文件准备写
fw.writeobj([1,2,3]); // 一次将一个数组转化成串行流写入文件
fw.close(); // 关闭文件
fr = std::file("objfile", 'r'); // 打开文件准备读
vec = fr.readobj(); // 将以前写入的数组串行流读出并恢复成对象
std::println(obj=vec); // 打印 [1,2,3]
```

一个文件可以用三种状态打开, 读状态, 写状态和添加状态, 分别由三个字符'r', 'w', 'a' 表示. 读状态只能打开已经存在的文件. 试图用读状态打开一个不存在的文件会返回错误. 写状态会创建文件, 如果文件已经存在, 其内容会被清除. 添加状态也会创建文件, 但如果文件已经存在, 其内容不会被清除, 新的内容被添加到文件尾.

9.3 网络读写举例

以下鲁班例子代码演示怎样使用网络 SOCKET 来实现一个简单的客户-服务器之间的数据传输. 以下例子使用了一个新的数据类型 *net::listener*. 这个数据类型用于网络通讯的服务器一边. *net::listener* 的对象创建时的输入参数是一个 TCP 端口号.它的主要成员函数是 *accept()*. 这个成员函数会在 TCP 端口等待客户从网络上来的连接请求, 收到后返回一个 *net::socket* 对象. 用这个 SOCKET 对象就可以象读写文件那样和客户对话.

服务器脚本:

```
// 此服务器问候客户并告状客户排号是第几
listener = net::listener(6500); // 服务器的 TCP 端口号是 6500
clientcount=0;
while( true ) // 循环
{
        socket = listener.accept(); // 等待接收新的客户连接
        name = socket.readobj(); // 读客户名字
        socket.writeobj("Hello, "+name); // 问候客户
        ++clientcount; // 序号加一
        Socket.writeobj(clientcount); // 将序号送给客户
}
```

客户脚本:

```
socket = net::socket("localhost", 6500); // 连接服务器端口 6500
socket.writeobj("Chinese");  // 告诉服务器自己名字
msg = socket.readobj(); // 接收服务器发的问候
std::console().writeline("The server says: ", msg); // 打印问候
mynumber = socket.readobj(); //接收服务器发的序号
std::console().writeline("My number is ", mynumber); // 打印序号
```

以上两个鲁班程序, 一个是服务器程序, 一个是客户程序. 服务器程序打开一个 TCP 端口 6500, 然后等待客户连接. 收到连接后先读客户送来的名字, 然后发送问候字符串, 最后将客户连接的序号发给客户. 客户程序的读写顺序则相反. 客户先创建一个 SOCKET 连接到服务器的 6500 TCP 端口. 然后将自己名字发给服务器. 再读问候字符串并打印出来. 最后读服务器发来的序号也打印出来.

这个例子程序里服务器和客户在同一个机器上运行 . 在客户程序里写的服务器名字是"localhost". 运行时先启动服务器程序, 然后启动客户程序. 如果服务器和客户在不同的机器上, 只需修改客户程序, 将"localhost"改为服务器机器的名字就好.

9.4 标准屏幕打印部件: std::println std::print
为了客户方便, 鲁班软件包包括标准打印部件 std::println std::print 这两个部件把输入的对象打印到屏幕上. 它们都是用鲁班语言写成. 内部用的是 std::console 对象类型. 以下是它们的源代码.

```
namespace std;

struct println( input obj;)
as process
{   std::console().writeline(input.obj); }

struct print( input obj;)
as process
{   std::console().write(input.obj); }
```

9.5 标准对象恢复部件 std::des
所有的鲁班数据对象或部件都可以转换成串行流字符串. std::des 是鲁班的一个标准部件用来从串行流字符串恢复原来的数据对象. 以下是代码例子.

```
x = {“one”:1, “two”:2};  //创建一个字典对象
xguts = x.serialize();// 转换成串行流字符串
xback = std::des(stream=xguts).obj; // 调用 std::des 恢复对象
truth = xback == x; // truth = true
```

以上代码创建一个字典对象然后转换成 串行流字符串, 再调用 std::des 部件来从串行流恢复原来的对象. 最后比较恢复的对象和原本对象看它们是否一样.

12. 鲁班命令行解释器使用说明

Simplicity in character, in manners, in style;
in all things the supreme excellence is simplicity.
- Henry Wadsworth Longfellow

鲁班代码是用鲁班命令行解释器来启动解释执行. 鲁班命令行解释器可以从文件里读鲁班源程序并解释执行. 鲁班命令行解释器也可以接受从操作系统命令行来的短鲁班脚本, 或者接受用户当时打入的鲁班代码立即执行并显示结果.

12.1 运行鲁班程序文件
运行鲁班脚本文件很简单, 只需要将脚本文件名放在命令行即可. 鲁班解释器会按它们在命令行的顺序逐个运行. 就如如下举例.

在文件 saygood.lbn 里写入如下鲁班代码:

```
std::println(obj="World is good");
```

在文件 saybad.lbn 里写入如下鲁班代码:

```
std::println(obj="World is bad");
```

然后在你的操作系统命令行启动鲁班解释器运行这两个鲁班脚本文件.

```
Mycomputer > luban saygood.lbn saybad.lbn
World is good
World is bad
```

以上例子可以看出怎样运行鲁班脚本文件. 鲁班程序文件除了脚本文件外, 还有鲁班部件定义文件. 鲁班部件定义文件只定义部件, 但没有可立即执行的脚本代码. 部件的运行要由其他的脚本调用. 部件定义文件和脚本文件一样可以放在命令行由鲁班解释器读入并执行. 鲁班解释器会先处理所有的部件定义文件将部件登记到鲁班命名空间, 然后再解释执行脚本文件. 部件定义文件在命令行的顺序可以任意调换而不影响它们的处理. 如下是例子.

在文件 saygood.lbn 里写入如下鲁班代码:

```
namespace demo;

struct SayGood()
as process
{
        std::println(obj="World is good");
}
```

在文件 saybad.lbn 里写入如下鲁班代码:

```
namespace demo;

struct SayBad()
as process
{
        std::println(obj="World is bad");
}
```

在文件 start.lbn 里写入如下鲁班代码:

```
demo::SayGood(=);
demo::SayBad(=);
```

然后在你的操作系统命令行启动鲁班解释器运行这三个鲁班文件.

```
Mycomputer > luban saybad.lbn start.lbn saygood.lbn
World is good
World is bad
```

可以看出以上运行结果和上一个例子一样. 但是程序结构不同. 这个例子里有两个鲁班部件定义文件 saygood.lbn 和 saybad.lbn. 这两个文件定义了两个部件 demo::SayGood 和 demo::SayBad. 可真正启动执行这两个 鲁班部件的是叫 start.lbn 的鲁班脚本文件.

从上面命令行的鲁班文件顺序 saygood.lbn, start.lbn, saybad.lbn 可以看出一是鲁班部件文件的顺序不是执行顺序, 二是部件文件总是先得处理, 所以脚本文件 start.lbn 虽然在部件文件 saygood.lbn 之前 , 它还是能够调用 saygood.lbn 所定义的部件 demo::SayGood.

另外需要指出的是, 在鲁班解释器的命令行, 鲁班部件定义文件的顺序可以任意排列, 不用考虑它们之间的有互相依赖关系. 鲁班解释器会自动处理.

12.2 交互式执行鲁班脚本或部件定义程序

打入以下命令会启动鲁班解释器进入交互式状态.

```
Mycomputer > luban -i
6 imported types
script/s - to start scripting
edit/e  - to edit script using $EDITOR or vi
list/l <namespace name> - to browse name space
quit/q - to quit
Luban> _
```

-i 参数会让解释器进入交互状态 . 在交互状态, 你可以用打入鲁班程序, 查看鲁班名字空间内容或者启动编辑器编辑鲁班程序(BETA1.2 或更高). 以下是举例说明.

```
Mycomputer > luban –i
> list std
   std::console    std::console
   std::deserializer       std::deserializer
   std::file       std::file
   std::print      std::print(input readwrite obj;)
   std::des        std::des(input readwrite string
   stream;output readonly obj;)
   std::printline   std::printline(input  readwrite
   obj;)
   std::println     std::printline(input  readwrite
   obj;)
```

以上例子用了"list"命令来查看鲁班名字空间里的'std"子空间的内容. 鲁班解释器列举了所有"std"含有的数据类型和部件类型.

```
Mycomputer > luban –i
> script
   std::console().writeline("Hello, world");
   ESC\ENTER
   Hello, world
```

以上例子用了"script"命令来打入鲁班脚本程序并执行. 敲 ESC 键然后 ENTER 会让鲁班解释器接受你打入的鲁班脚本程序并执行 . 上例的脚本程序简单在屏幕上打印"Hello, world".

```
Mycomputer > luban –i
> edit
```

如果你用鲁班 BETA1.2 或更高, 以上命令会启动一个外部编辑器来编辑鲁班程序并执行. 你可以用环境变量"EDITOR"来指明你想用的外部编辑器. 比如:

```
Mycomputer > export EDITOR=xemacs
```

如果没有指定的外部编辑器, 鲁班会用"vi". 如果不能启动外部编辑器, 鲁班会报错. 还有需要指出的是, "edit"功能只在 Linux 和有 Cygwin 安装的 WINDOWS 上使用. 如果你只安装了鲁班的 WINDOWS 执行文件包没有安装 CYGWIN, "edit"功能不工作.

12.3 其他命令行参数

-s 参数, 可以用来在命令行直接输入鲁班程序. 如下例:

```
Mycomputer > luban –s “std::console().writeline(100);”
100
```

如果命令行里有其他鲁班程序文件. -s 参数里的程序会在鲁班程序文件处理后执行.

-t 参数, 指定数据类型定义文件名

鲁班软件包有一个鲁班数据类型定义文件”imports”里面含有鲁班的外部数据类型和它们对应的函数库文件名. 如有必要, 你可以用自己的数据类型定义文件, 只需要用-t 参数告状鲁班它的名字. 如下例:

```
Mycomputer > luban –t myimports
```