

x.lang

1、语言结构

- 1.1、包声明
- 1.2、引入文件
- 1.3、函数
- 1.4、接口
- 1.5、类
- 1.6、行分隔符
- 1.7、注释
- 1.8、标识符
- 1.9、访问修饰符

2、数据类型

2.1、基本数据类型

- 2.1.1、byte
- 2.1.2、short
- 2.1.3、int
- 2.1.4、long
- 2.1.5、float
- 2.1.6、double
- 2.1.7、ubyte
- 2.1.8、ushort
- 2.1.9、uint
- 2.1.10、ulong
- 2.1.11、bool
- 2.1.12、char
- 2.1.13、string

2.2、复合类型

3、变量类型

- 3.1、格式
- 3.2、类型

4、常量

5、运算符

- 5.1、算术运算符
- 5.2、关系运算符
- 5.3、逻辑运算符
- 5.4、位运算符
- 5.5、赋值运算符
- 5.6、运算符优先级

6、循环语句

- 6.1、格式
- 6.2、循环控制语句

7、条件语句

- 7.1、if .. else ..
- 7.2、三目运算符 (?:)

8、分支语句

9、数组

- 9.1、声明变量和初始化
- 9.2、数组访问处理
- 9.3、遍历数组

- 9.4、数组作为参数
- 9.5、数组作为返回值
- 9.6、多维数组
- 10、错误处理
 - 10.1、标准接口
 - 10.2、声明错误与抛出
 - 10.3、错误捕获
- 11、类
 - 11.1、普通用法
 - 11.1.1、类的定义
 - 11.2、构造方法
 - 11.2.1、格式
 - 11.3、成员
 - 11.3.1、变量（字段）
 - 11.3.2、方法
 - 11.4、继承
 - 11.4.1、单继承
 - 11.4.2、多继承
 - 11.4.3、重写
- 12、重载
- 13、接口
 - 13.1、格式
 - 13.2、实现
- 14、抽象类
 - 14.1、格式
 - 14.2、实现
- 15、枚举
 - 15.1、格式
 - 15.2、使用
 - 15.2.1、声明
 - 15.2.2、外部调用
 - 15.2.3、枚举值迭代
- 16、多态
- 17、注解
- 18、泛型
- 19、并发
- 20、内置函数

x.lang

1、语言结构

按文件划分模块声明

1.1、包声明

所在文件夹名称

```
1  pkg main;
```

1.2、引入文件

引入文件模块

```
1  import "demo/test/filename";
```

1.3、函数

面向过程时称为“函数”

可用 `pub`、`pri` 修饰其可见性

```
1  // 主函数
2  // 名称: `main`, 无参数, 无返回值
3  fn main() {
4  }
5
6  // 一个普通函数
7  // 名称: `doX`, 可见性: `pub`, 无参数, 无返回值
8  pub fn doX() {
9  }
10
11 // 文件内可见
12 // 有参数
13 pri fn private(int x, string a) {
14 }
15
16 // 默认包 (目录) 内可见
17 // 有返回值
18 fn protected() -> string {
19     return "x.lang";
20 }
```

1.4、接口

```
1  /**
2   * 声明一个接口
3   */
4  interface XInterface {
5
6      /**
7       * 接口方法
8       */
9       test(int x) -> string;
10 }
```

1.5、类

```
1  /**
2   * 普通类
3   */
4  pub class XClass {
5      // 成员变量
6      string x;
7
8      /**
9       * 方法
10      * 无需使用 fn 关键字
11      */
12      pub test() {
13      }
14  }
15
16  /**
17   * 继承
18   */
19  pub class AClass: XClass {}
20
21  /**
22   * 继承 + 接口实现
23   */
24  pub class BClass: XClass ~ XInterface {
25
26      /**
27       * 必须实现接口的方法
28       */
```

```
29     pub test(int x) -> string {  
30         return "x.lang";  
31     }  
32 }
```

1.6、行分隔符

必须使用 `;` 作为行分隔符

1.7、注释

- 行注释

```
1  // 这是行注释, 单行有效
```

- 块注释

```
1  /*  
2      这是块注释  
3  */
```

- 文档注释

```
1  /**  
2   * 这是文档注释, 注释文件头、函数、接口、类、字段、方法等, 用于生成 `doc` 文档手册  
3   */
```

1.8、标识符

一个或是多个字母 `[a-zA-Z]` 数字 `[0-9]`、下划线 `_` 组成的序列, 但是第一个字符必须是字母或下划线, 不能是数字。

1.9、访问修饰符

- `pub`
公共开放访问
- `pri`
面向过程：当前文件访问
面向对象：当前类访问
- 默认
面向过程：当前包（目录）访问
面向对象：当前类及其子类访问

修饰符	当前文件/当前类	当前包/子孙类	其他（开放）
<code>pub</code>	✓	✓	✓
默认	✓	✓	✗
<code>pri</code>	✓	✗	✗

2、数据类型

2.1、基本数据类型

2.1.1、`byte`

8 位有符号二进制补码（1 字节）表示的整数

- 最小值 `-128` (-2^7)
- 最大值 `127` ($2^7 - 1$)
- 默认值 `0`
- 例子: `byte a = 100, byte b = -50`

2.1.2、`short`

16 位有符号二进制补码（2 字节）表示的整数

- 最小值 `-32768` (-2^{15})
- 最大值 `32767` ($2^{15} - 1$)
- 默认值 `0`
- 例子: `short a = 100, short b = -200`

2.1.3、int

32 位有符号二进制补码 (4 字节) 表示的整数

- 最小值 $-2,147,483,648$ (-2^{31})
- 最大值 $2,147,483,647$ ($2^{31} - 1$)
- 默认值 `0`
- 例子: `int a = 1000, int b = -23444`

2.1.4、long

64 位有符号二进制补码 (8 字节) 表示的整数

- 最小值 $-9,223,372,036,854,775,808$ (-2^{63})
- 最大值 $9,223,372,036,854,775,807$ ($2^{63} - 1$)
- 默认值 `0`
- 例子: `long a = 10000000000, long b = -23223424444`

2.1.5、float

符合 IEEE-754 的 32 位 (4 字节) 单精度有符号浮点型数

不能用来表示精确的数值, 如货币

- 最小值 $-2,147,483,648$ (-2^{31})
- 最大值 $2,147,483,647$ ($2^{31} - 1$)
- 默认值 `0.0`
- 例子: `float a = 3.14`

2.1.6、double

符合 IEEE-754 的 64 位 (8 字节) 单精度有符号浮点型数

不能用来表示精确的数值, 如货币

- 最小值 $-9,223,372,036,854,775,808$ (-2^{63})
- 最大值 $9,223,372,036,854,775,807$ ($2^{63} - 1$)
- 默认值 `0.0`
- 例子: `double a = 3.14`

2.1.7、`ubyte`

8 位无符号二进制补码 (1 字节) 表示的整数

- 最小值 0
- 最大值 255 ($2^8 - 1$)
- 默认值 0
- 例子: `ubyte a = 100, ubyte b = -50`

2.1.8、`ushort`

16 位无符号二进制补码 (2 字节) 表示的整数

- 最小值 0
- 最大值 65535 ($2^{16} - 1$)
- 默认值 0
- 例子: `ushort a = 100, ushort b = -200`

2.1.9、`uint`

32 位无符号二进制补码 (4 字节) 表示的整数

- 最小值 0
- 最大值 4,294,967,295 ($2^{32} - 1$)
- 默认值 0
- 例子: `uint a = 1000, uint b = -23444`

2.1.10、`ulong`

64 位无符号二进制补码 (8 字节) 表示的整数

- 最小值 0
- 最大值 18,446,744,073,709,551,615 ($2^{64} - 1$)
- 默认值 0
- 例子: `ulong a = 100000000000, ulong b = -23223424444`

2.1.11、`bool`

用一个二进制位 (1 位) 表示的整数

- 只有两个取值: `true` 和 `false`, 即: 1 -> `true`, 0 -> `false`
- 默认值 `false`
- 例子: `bool a = true`

2.1.12、char

一个单一的 32 位 (4 字节) utf-8 编码的 Unicode 字符

在 utf-8 编码中, 从 1 到 4 个字节不等, 一个中文字符占三个字节, 一个英文字符占一个字节

0xxxxxxx 表示文字符号 0~127, 兼容 ASCII 字符集

从 128 到 0x10ffff 表示其他字符

- 最小值 0
- 最大值 4,294,967,295 ($2^{32} - 1$)
- 默认值 0
- 例子: `char a = 'x', char b = '中'`

2.1.13、string

字符串就是一串固定长度的字符连接起来的字符序列

由 char 数组连接构成

使用 utf-8 编码的 Unicode 文本

- 示例: `string x = "x.lang"`

2.2、复合类型

即面向对象模块

详细请参照 面向对象 模块

通过定义一个类来作为一个类型

3、变量类型

采用小驼峰 (camelCase) 命名规范

3.1、格式

所有变量在使用前必须先声明

声明格式如下

```

1 // 指定类型声明
2 type identifier [ = value][, identifier [ = value] ...];
3
4 // 自动推导类型
5 identifier [, identifier [...]] := value [, value [...]];

```

格式说明

- `type`
类型
- `identifier`
变量名，可以使用逗号 `,` 隔开来声明多个同类型变量
- `value`
变量值

3.2、类型

- 局部变量
定义在方法、构造方法、函数或者语句块中的变量，作用域只限于当前代码块
不能被访问修饰符修饰

```

1 type varName;

```

- `type`
变量类型
- `varName`
变量名称
- 成员变量
定义在类中、方法之外的变量，作用域为整个类，可以被类中的任何方法、语句块访问
取决于被访问修饰符的修饰

```

1 [pub | pri] type varName;

```

- `type`
变量类型
- `varName`
变量名称
- 全局变量

定义在类外、函数外、文件内的变量，作用域为整个工程，可以被工程中任意文件、类、函数和代码块访问

取决于被访问修饰符的修饰

```
1  [pub | pri] type varName;
```

- type

变量类型

- varName

变量名称

- 参数变量

方法、函数定义时声明的形参变量，作用域为方法、函数内部

不能被访问修饰符修饰

```
1  [pub | pri] fn functionName(type paramName1, paramName2, type paramName3,  
    ...) -> [returnType [, returnType ...]] {  
2      // 函数体  
3  }  
4  
5  [pub | pri] methodName(type paramName1, paramName2, type paramName3, ...) ->  
    [returnType [, returnType ...]] {  
6      // 方法体  
7  }
```

- functionName

函数名

- methodName

方法名

- type

变量类型

- paramName...

参数变量名称

- returnType

返回值类型

4、常量

一个简单值的标识符，在程序运行时不会被修改的量

使用 `const` 关键字声明定义

可以被访问修饰符修饰

```
1 [pub | pri] const type IDENTIFIER = VALUE;
```

格式说明

- `type`
类型
- `IDENTIFIER`
常量名，采用大写下划线（`UPPER_SNAKE_CASE`）命名规范
- `VALUE`
常量值

5、运算符

5.1、算术运算符

设 `A` 值为 `10`，`B` 值为 `20`

运算符	描述	实例
<code>+</code>	相加	<code>A + B</code> 结果为 <code>30</code>
<code>-</code>	相减	<code>A - B</code> 结果为 <code>-10</code>
<code>*</code>	相乘	<code>A * B</code> 结果为 <code>200</code>
<code>/</code>	相除	<code>B / A</code> 结果为 <code>2</code>
<code>%</code>	求余	<code>A % B</code> 结果为 <code>10</code>
<code>++</code>	自增	<code>A++</code> 结果为 <code>11</code>
<code>--</code>	自减	<code>B--</code> 结果为 <code>19</code>

5.2、关系运算符

设 A 值为 10，B 值为 20

运算符	描述	实例
<code>==</code>	相等	<code>A == B</code> 结果为 <code>false</code>
<code>!=</code>	不等	<code>A != B</code> 结果为 <code>true</code>
<code>></code>	大于	<code>A > B</code> 结果为 <code>false</code>
<code><</code>	小于	<code>A < B</code> 结果为 <code>true</code>
<code>>=</code>	大于等于	<code>A >= B</code> 结果为 <code>false</code>
<code><=</code>	小于等于	<code>A <= B</code> 结果为 <code>true</code>

5.3、逻辑运算符

设 A 值为 `true`，B 值为 `false`

运算符	描述	实例
<code>&&</code>	逻辑 AND 运算符，短路与	<code>A && B</code> 结果为 <code>false</code>
<code> </code>	逻辑 OR 运算符，短路或	<code>A B</code> 结果为 <code>true</code>
<code>!</code>	逻辑 NOT 运算符	<code>!(A && B)</code> 结果为 <code>true</code>

5.4、位运算符

假定 A = 60; B = 13; 其二进制数转换为：

```
1  A = 0011 1100
2
3  B = 0000 1101
4
5  -----
6
7  A&B = 0000 1100
8
9  A|B = 0011 1101
10
11 A^B = 0011 0001
12
```

运算符	描述	实例
&	按位与	A & B 结果为 12，二进制为 0000 1100
	按位或	A B 结果为 61，二进制为 0011 1101
^	异或	A ^ B 结果为 49，二进制为 0011 0001
<<	左移	A << 2 结果为 240，二进制为 1111 0000
>>	右移	A >> 2 结果为 15，二进制为 0000 1111

5.5、赋值运算符

运算符	描述	实例
=	简单赋值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后赋值	C += A 等于 C = C + A
-=	相减后赋值	C -= A 等于 C = C - A
*=	相乘后赋值	C *= A 等于 C = C * A
/=	相除后赋值	C /= A 等于 C = C / A
%=	求余后赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= A 等于 C = C << A
>>=	右移后赋值	C >>= A 等于 C = C >> A
&=	按位与后赋值	C &= A 等于 C = C & A
=	按位或后赋值	C = A 等于 C = C A
^=	异或后赋值	C ^= A 等于 C = C ^ A

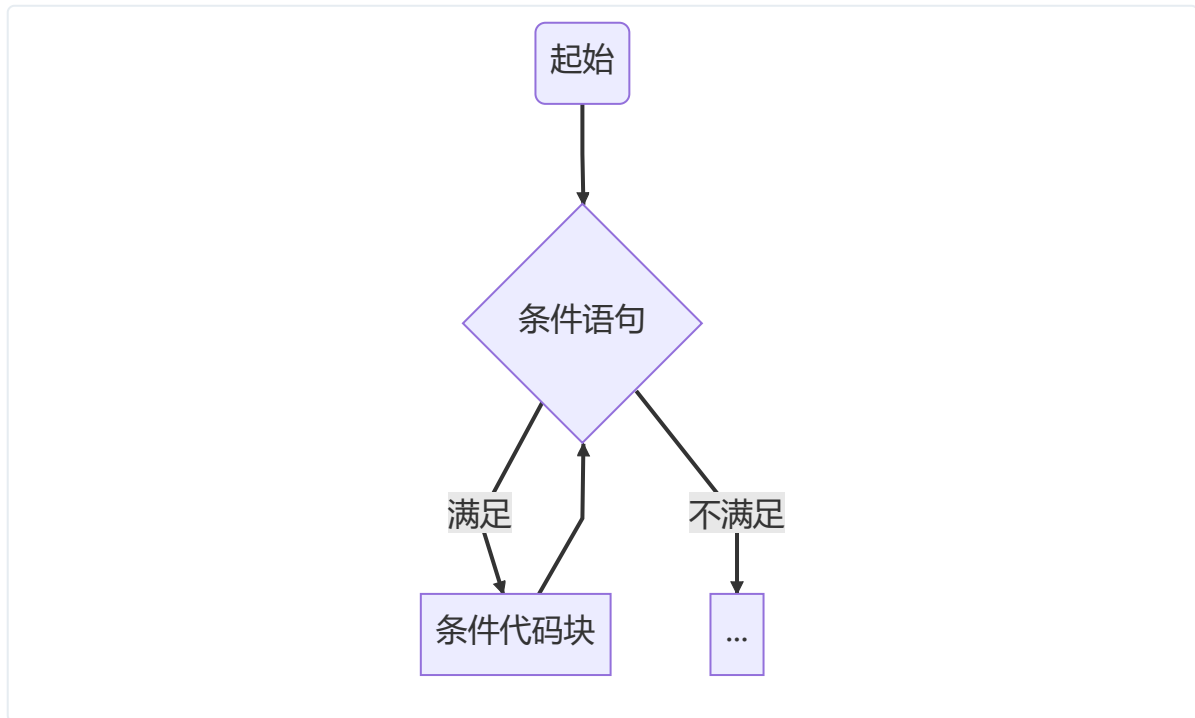
5.6、运算符优先级

下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低

优先级	运算符
5	*、/、%、<<、>>、&、&^
4	+、-、 、^
3	==、!=、<、<=、>、>=
2	&&
1	

6、循环语句

以下为循环语句流程图



6.1、格式

仅提供 `for` 循环语句

```
1 // 普通循环
2 for [[type varName = value]; [value?]; [value++]] {
3     // 条件代码块
4 }
5
6 // 迭代器-1
7 for index, item in iterator {
8 }
9
10 // 迭代器-2
11 for index, item in 0..200 {
12 }
13
14
15
16 // 示例
17 // 无限循环
18 for {
19 }
20 for ; ; {
21 }
```



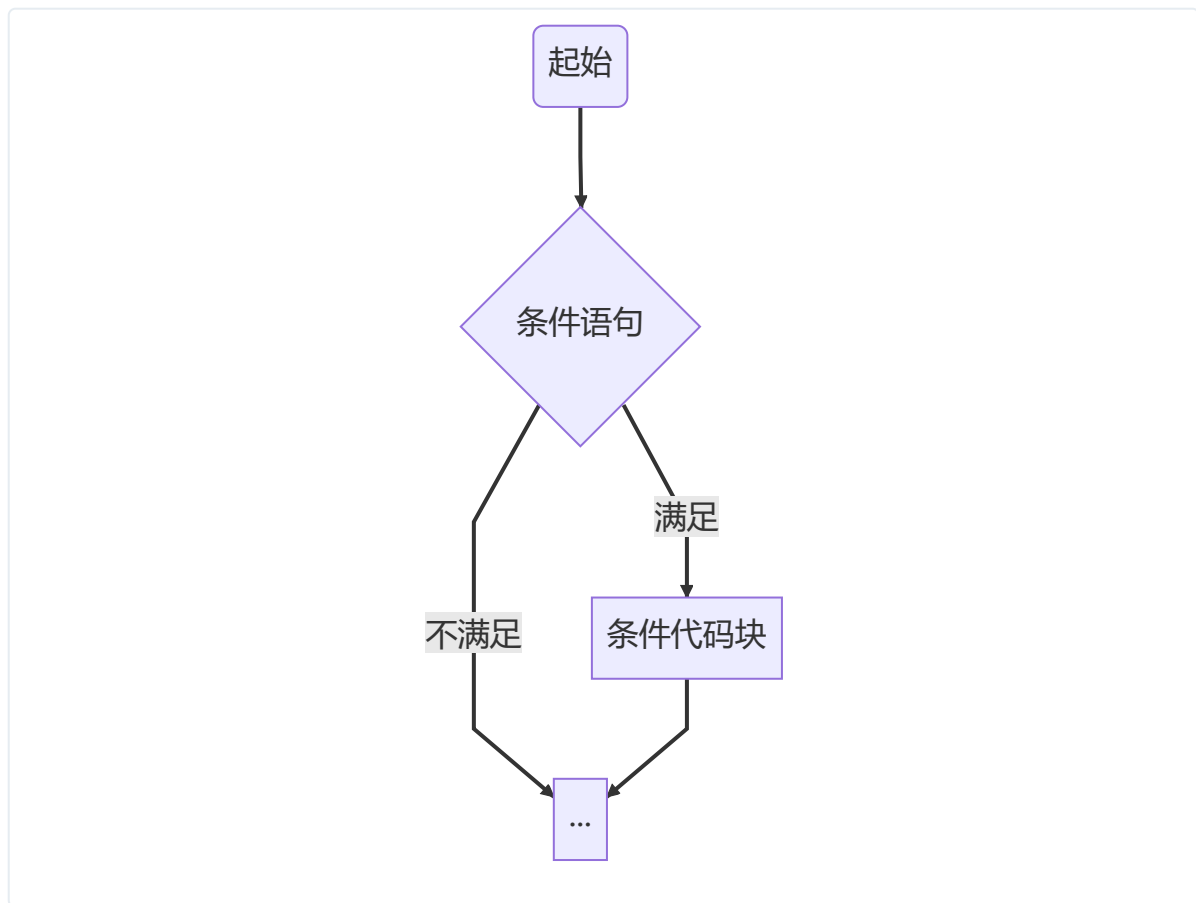
```
22
23 // 定义一个局部变量 x 并初始化
24 for int x = 1; ; {
25 }
26
27 // x < 10 当做循环条件
28 for ; x < 10; {
29 }
30
31 // 每次循环后 x 自增
32 for ; ; x++ {
33 }
34
35 // 三者结合
36 for int x = 1; x < 10; x++ {
37 }
```

6.2、循环控制语句

语句	描述
break	中断当前循环
continue	跳过本轮循环，进入下一轮循环
goto	将控制转移到被标记的语句

7、条件语句

以下为条件语句的流程图



7.1、if .. else ..

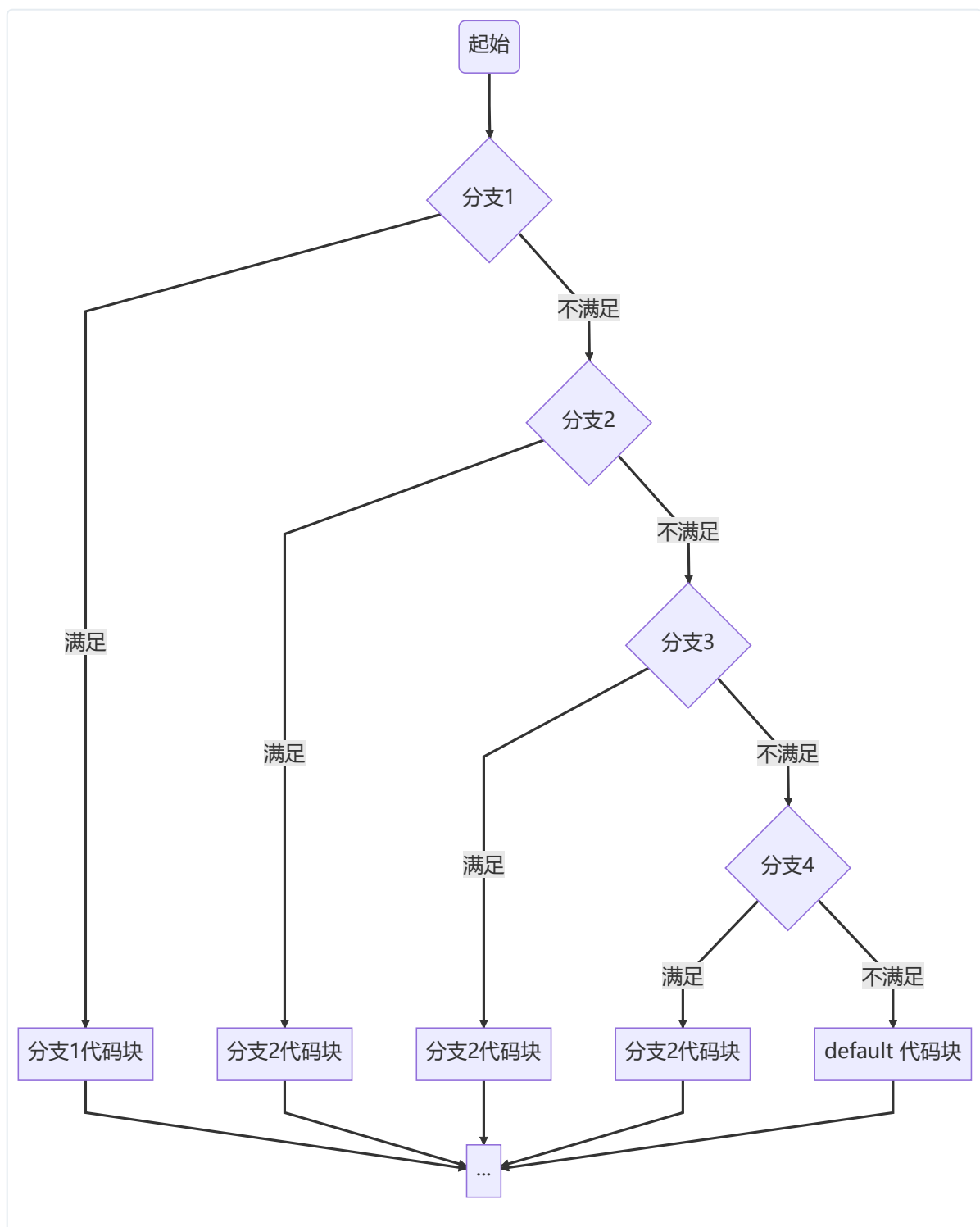
```
1  if 布尔表达式 {
2      // 满足条件时执行的代码块
3  } [else if ... { ... } else { ... }]
4
5  // 示例
6  if a < 10 {
7      println("a < 10");
8  } else if a < 20 {
9      println("a < 20");
10 } else {
11     println("a >= 20");
12 }
```

7.2、三目运算符（?:）

```
1  [type varName = ] 布尔表达式 ? [正向结果 | 三目运算符] : [反向结果 | 三目运算符];
2
3  // 示例
4  // 正常
5  int x = a > 10 ? 1 : 2;
6
7  // 无需接收返回值
8  a > 10 ? execA() : execB();
9
10 // 条套
11 int x = a > 10 ? 1 : a > 20 ? 2 : 3;
```

8、分支语句

以下为分支语句的流程图



when .. case ..

```
1  when 变量 {  
2      值1 -> {语句}  
3      值2 -> {语句}  
4      值3, 值4 -> {语句}  
5      _ -> {default 语句}  
6  }  
7  
8  // 示例
```

```
9  when x {
10      1 -> println("x = 1")
11      2 -> {println("x = 2");}
12      3,4 -> println("x = 3 or x = 4")
13      _ -> println("这里是 default 模块")
14  }
```

9、数组

TODO: 能否支持动态扩容?

9.1、声明变量和初始化

```
1  // 声明一个数组
2  type[size] arrName;
3
4  // 声明并初始化一个数组
5  // 不指定元素时使用当前类型的默认值填充数据
6  type[size] arrName = type[size];
7  // 使用默认初始容量
8  type[] arrName = type[];
9
10 // 指定元素时直接初始化
11 // 指定初始容量
12 type[size] arrName = type[size] {1, 2, 3, 4, ...};
13 // 使用默认初始容量
14 type[] arrName = type[] {1, 2, 3, 4, ...};
15
16
17 // 自动推导类型
18 // 指定初始容量
19 arrName := type[size];
20 // 使用默认初始容量
21 arrName := type[];
22
23 // 指定初始容量
24 arrName := type[size] {1, 2, 3, 4};
25 // 使用默认初始容量
26 arrName := type[] {1, 2, 3, 4};
27
28 // 根据赋值的元素数量来确定初始容量
29 arrName := {1, 2, 3, 4};
```

9.2、数组访问处理

以下操作示例中：

```
1 // 创建一个示例数组
2 int[] arr = int[];
```

- 数组长度

`length` 属性

```
1 // 获取数组的长度
2 int length = arr.length;
```

- 添加元素

- `push(...)` 方法

将元素添加到数组的末尾，并返回新的长度

```
1 int newLength = arr.push(2);
```

- `unshift(...)` 方法

将元素添加到数组的开头，并返回新的长度

```
1 int newLength = arr.unshift(3);
```

- 删除元素

- `pop(...)` 方法

删除数组的最后一个元素，并返回该元素

```
1 int item = arr.pop();
```

- `shift(...)` 方法

删除数组的第一个元素，并返回该元素

```
1 int item = arr.shift();
```

- 读取和更新元素

直接用数组下标访问

如果下标超过数组长度则抛出异常

```
1 // 获取第一个元素
2 int item = arr[0];
3
4 // 更新第二个元素为 100
5 arr[1] = 100;
```

- 反转数组

提供 `reverse()` 方法，实现数组元素的反转

```
1 int[] reversedArr = arr.reverse();
```

- 连接数组

提供 `concat(...)` 方法，实现当前数组与一个或多个数组的拼接

要求待拼接的数组类型与当前数组类型一致，否则语法不通过

```
1 int[] a = int[] {2, 3, 4};
2 int[] b = int[] {5, 4, 8};
3
4 // 拼接一个数组
5 int[] newArr = arr.concat(a);
6
7 // 拼接多个数组
8 int newArr = arr.concat(a, b);
9
10 // 拼接一个自动推导类型的数组
11 int newArr = arr.concat({9, 6, 3, 0});
```

- 连接元素

提供 `join(string with)` 方法，实现将数组元素按指定参数拼接成字符串的功能

```
1 string result = arr.join(", ");
2
3 // 输出结果: 1, 2, 3, 4
4 println(result);
```

- 查找索引

- `indexOf(...)` 方法

查找元素第一次出现的位置索引

```
1 int index = arr.indexOf(2);
```

- `lastIndexOf(...)` 方法

查找元素最后一次出现的位置索引

```
1 int index = arr.lastIndexOf(2);
```

9.3、遍历数组

- 普通遍历

```
1 for int i = 0; i < arr.length; i++ {  
2     println("item[{i}] = {}", arr[i]);  
3 }
```

- 迭代遍历

```
1 for index, item in arr {  
2     println("arr[{index}] = {item}");  
3 }
```

9.4、数组作为参数

数组作为参数时为引用传递，如果被调用的函数内对参数数组进行了写操作，调用者处的数组将会受到同样的影响

```
1 // 调用函数  
2 doX(arr);  
3 // 调用后此处的 arr[1] 元素将是修改后的 100  
4  
5 fn doX(int[] arr) {  
6     // 函数内对参数数组的元素进行了修改  
7     arr[1] = 100;  
8 }
```


9.5、数组作为返回值

返回数组的作用域将自动延伸到与调用方作用域对齐

```
1 // createArr() 返回数组的作用域将与 newArr 对齐
2 int[] newArr = createArr();
3
4 fn createArr() -> int[] {
5     // 返回一个数组
6     return {1, 2, 3};
7 }
```

9.6、多维数组

当声明数组元素的类型也是数组时，将构成多维数组

```
1 // 格式
2 type[size1][size2]... arrName;
3
4 // 示例（二维数组）
5 int[10][20] arr = int[10][20];
```

10、错误处理

10.1、标准接口

需要实现标准接口方能作为错误异常类

- `Error` 接口

抛出异常，进程不受影响

- 该接口的定义源码

```
1 interface Error {
2 }
```

- 实现自定义异常

```
1 pub class MyError ~ Error {
2 }
```

10.2、声明错误与抛出

如果一个函数或方法内需要抛出错误，需要现在函数或方法上声明异常类

提供 `throws` 关键字用来声明错误，`throw` 关键字用来抛出错误

```
1 // 函数
2 pub fn doX() -> int, string throws MyPanic {
3     // 抛出
4     throw MyPanic();
5 }
6
7 pub class A {
8
9     // 方法
10    pub deal() -> int, string throws MyError, AnError {
11        throw MyError();
12    }
13 }
```

10.3、错误捕获

在可能出现异常的调用处使用 `try {...} catch(Error err) {} finally {}` 来捕获错误异常

```
1 try {
2     int x, str = doX();
3     println("x = {x}, str = {str}");
4 } catch(MyError err) {
5     // 错误处理
6 } finally {
7     // 代码片段
8 }
```

11、类

11.1、普通用法

11.1.1、类的定义

- 格式

```
1  /**
2   * 类的 doc 注释
3   */
4  [pub | pri] [sealed] class ClassName {
5  }
```

- 访问修饰符

可使用默认、`pub`、`pri` 三种访问修饰符

- `sealed` 关键字

被 `sealed` 关键字修饰的类不可被其他类继承

- `doc` 文档注释

用于生成 `doc` 文档说明

- `ClassName` 类名

采用大驼峰 (`PascalCase`) 命名规范

- 示例

- 无访问修饰符

```
1  /**
2   * 该类只能包内被访问
3   */
4  class ClassX {
5  }
```

- 有访问修饰符

```

1  /**
2   * 该类不可被访问
3   */
4  pri class ClassX {
5  }
6
7  /**
8   * 该类可被任意访问
9   */
10 pub class ClassX {
11 }

```

- 不可被继承

```

1  sealed class ClassX {
2  }

```

11.2、构造方法

11.2.1、格式

```

1  [pub | pri] ClassName([type paramName [, type paramName1 ...]]) {
2      // 给字段初始化赋值
3  }

```

- 访问修饰符

- pub

可被外界任意访问

- pri

仅可在当前文件内访问

以文件 `/demo/demo.x` 文件为例

```

1  class Demo {
2
3      pri Demo() {}
4  }
5
6  // 仅在当前文件可访问
7  pub fn createDemo() -> Demo {
8      return Demo();
9  }

```

- 默认

仅可被当前包和子孙类访问

- 方法名

与类名一致

- 参数

参数列表与普通函数相同用法

- 返回值

不声明返回值类型

返回值为当前类的一个实例

11.3、成员

成员变量和方法

11.3.1、变量（字段）

命名规范跟“变量”一致

- 格式

```

1  /**
2   * 字段的文档注释
3   */
4  [pub | pri] type fieldName;

```

- 访问修饰符

可使用默认、`pub`、`pri` 三种访问修饰符

- 字段的文档注释

用于生成 `doc` 文档说明

- `type`

字段类型

- `fieldName` 字段名

采用小驼峰 (`camelCase`) 命名规范

- 使用

`pub` 修饰的字段可通过实例直接访问

```
1 class ClassX {
2     pub string name;
3 }
4
5 fn main() {
6     // 使用默认的空参构造函数
7     ClassX x = ClassX();
8     // 通过实例直接访问公开字段
9     println("x.name = {}", x.name);
10
11    // 修改
12    x.name = "X";
13 }
```

`pri` 修饰的字段不可被外界直接访问，仅可在当前类直接访问，外界需借助相应的 `getX()` 和 `setX(...)` 方法来进行访问操作

```
1 class ClassX {
2     pri string name;
3
4     // 需要提供对应的 get 和 set 方法
5
6     pub getName() -> string {
7         return name;
8     }
9
10    pub setName(string name) {
11        this.name = name;
12    }
13 }
14
15 fn main() {
16    // 使用默认的空参构造函数
17    ClassX x = ClassX();
18    // 通过 getX() 方法访问
19    println("x.name = {}", x.getName());
20
21    // 修改
22    x.setName("X");
23 }
```

默认无访问修饰符的字段仅可被其子孙类访问

```
1 class Root {
2     string name;
3 }
4
5 class Parent : Root {
6     string age;
7
8     pub doX() {
9         println("name = {name}, age = {age}");
10    }
11 }
12
13 class X : Parent {
14     string count;
15
16     pub doX() {
17         println("name = {name}, age = {age}, count = {count}");
18    }
19 }
```

11.3.2、方法

类提供用于操作处理字段的一类行为函数

- 格式

```
1 [pub | pri] methodName([type paramName [, type paramName1 ...]]) [->
    returnType [, returnType ...]] [throws ErrorClass [, Error2 ...]] {
2     // 函数体
3 }
```

- 访问修饰符
访问修饰符与字段用法一致
- 方法名
采用小驼峰 (`camelCase`) 命名规范
- 参数列表
无参、一个或多个参数
- 返回值列表
用符号 `->` 指定返回值列表及其类型
- 异常错误
用 `throws` 关键字声明可能会抛出的异常类型

- 示例

```
1  class X {
2
3      /**
4       * pub 修饰
5       * 一个 string 类型的参数 name
6       * 一个 string 类型的返回值
7       * 一个异常错误 MyError 声明
8       */
9      pub doX(string name) -> string throws MyError {
10         if name == "" {
11             throw MyError("姓名不能为空");
12         }
13         return format("name = {name}");
14     }
15 }
16
17 fn main() {
18     X x = X();
19
20     try {
21         string result = x.doX("姓名");
22         println(result);
23     } catch(MyError err) {
24         panic(err);
25     }
26 }
```

11.4、继承

支持单继承与多继承，多继承时必须为父类们指定别名

`sealed` 修饰的类不可被继承

使用 `:` 实现类的继承关系

```
1  [pub | pri] class ClassName : ParentClassName {
2  }
```


11.4.1、单继承

```
1  pub class Parent {
2      pri string name;
3
4      pub Parent(string name) {
5          this.name = name;
6      }
7
8      pub getName() -> string {
9          return name;
10     }
11 }
12
13 pub class X : Parent {
14     pub string age;
15
16     pub X(string name, int age) {
17         super(name);
18         this.age = age;
19     }
20 }
21
22 // 外部访问
23 fn main() {
24     X x = X("姓名", 22);
25     println("x.name = {}, x.age = {}", x.getName(), x.age);
26 }
```

11.4.2、多继承

多继承需要分别指定父类的别名，以解决构造方法的初始化和方法重写冲突的问题

如果所继承的多个父类中有签名相同的方法，则子类必须重新实现该方法

使用 `as` 关键字为每个关键字指定别名

多个父类直接用 `,` 隔开

```
1  pub class Father {
2      pri string name;
3
4      pub Father(string name) {
5          this.name = name;
6      }
7
8      pub getName() -> string {
```

```
9         return name;
10     }
11
12     pub doX() -> string {
13         // 这是 Father 类的 doX() 方法实现
14         return "Father";
15     }
16 }
17
18 pub class Mother {
19     pri int age;
20
21     pub Mother(int age) {
22         this.age = age;
23     }
24
25     pub getAge() -> int {
26         return age;
27     }
28
29     pub doX() -> string {
30         // 这是 Mother 类的 doX() 方法实现
31         return "Mother";
32     }
33 }
34
35 pub class X : Father as fa, Mother as mo {
36     pub int count;
37
38     pub X(string name, int age, count) {
39         // 此处无法使用 super 来初始化父类
40         // 必须使用别名初始化
41         fa(name);
42         mo(age);
43
44         // 父类初始化完后才能初始化当前类的字段
45         this.count = count;
46     }
47
48     /**
49      * 由于两个父类中都有该方法，所以此处必须重写，否则调用方将无法得知该方法的执行逻辑
50      */
51     pub doX() -> string {
52         // 如果以 Father 类的实现为准
53         return fa.doX();
54
55         // 如果以 Mother 类的实现为准
56         return mo.doX();
57
58         // 也可以实现自己的逻辑
```

```

59         string resFa = fa.doX();
60         string resMo = ma.doX();
61         return format("resFa = {resFa}, resMo = {resMo}");
62     }
63 }
64
65 // 外部调用
66 fn main() {
67     X x = X("姓名", 22, 10);
68     println("x.name = {}, x.age = {}, x.count = {}, x.doX() = {}", x.getName(),
        x.getAge(), x.count, x.doX());
69 }

```

11.4.3、重写

对于父类中的某些方法，子类需要有自己的实现逻辑。此时需要通过重写操作来实现该方法的自定义。

要求访问修饰符、方法名、返回值列表、错误异常声明列表 保持一致方能实现重新功能。

```

1  pub class Parent {
2      pri string name;
3
4      pub Parent(string name) {
5          this.name = name;
6      }
7
8      pub getName() -> string {
9          return name;
10     }
11
12     pub doX() -> string {
13         // 这是 Parent 类的 doX() 方法实现
14         return "Parent";
15     }
16 }
17
18 pub class X : Parent {
19     pub string age;
20
21     pub X(string name, int age) {
22         super(name);
23         this.age = age;
24     }
25
26     /**
27      * 重写实现自己的逻辑
28      */
29     pub doX() -> string {
30         string res = super.doX();

```

```

31         return format("res = {res}");
32     }
33 }
34
35 // 外部访问
36 fn main() {
37     X x = X("姓名", 22);
38     println("x.name = {}, x.age = {}, x.doX() = {}", x.getName(), x.age,
39         x.doX());
39 }

```

12、重载

同一个类内，具有相同的方法名，却有不同的参数列表的现象叫做重载，跟返回值类型和异常声明列表无关。

即类提供相同行为的多中方法，至于要调用哪个方法需要由调用方决定。

通过调用方所传的实参列表自动匹配对应的方法被调用。

```

1  pub class X {
2
3      pub doX() -> string throws MyError {}
4
5      pub doX(string name) -> string throws MyError {}
6
7      // 与返回值无关，这种是错的
8      // 因为已经存在相同方法名和参数列表的方法
9      pub doX() {}
10
11     // 与异常类型无关，这种也是错的
12     // 因为已经存在相同方法名和参数列表的方法
13     pub doX(string name) -> int throws MyError {}
14 }

```

13、接口

具有共同性质和行为却行为方式多样化的一类事物的抽象统称

用关键字 `interface` 来定义

13.1、格式

```
1  [pub | pri] interface InterfaceName {  
2  
3      [methodName([type paramName [, ...]]) [ -> returnType [, ...]] [throws Error1  
        [, ...]]];  
4  }
```

- 访问修饰符

访问修饰符与“类”用法一致

- `interface`

用 `interface` 声明接口

- `InterfaceName`

接口名称，命名规范与“类”一致

- 方法声明

`pub` 访问修饰符，不可变更，且无需手动指定

方法签名与“类”一致，但此处不能有方法体，方法签名结束需要以 `;` 结束

13.2、实现

接口不可直接创建实例

需要定义类来实现接口，才能通过创建类的实例来创建具有接口行为的实例

通过符号 `~` 来定义接口实现功能

必须实现接口声明的所有方法

```
1  /**  
2   * 接口声明  
3   */  
4  pub interface XInterface {  
5  
6      /**  
7       * 声明接口方法  
8       */  
9      doX(string param) -> string, int throws MyError;  
10  
11     /**  
12      * 可重载  
13      */  
14     doX(string param, int param1) -> string, int throws MyError;
```

```

15 }
16
17 /**
18  * 通过符号 ~ 来实现接口
19  */
20 pub class X ~ XInterface {
21
22     /**
23      * 实现接口所定义的方法
24      */
25     pub doX(string param) -> string, int throws MyError {
26         return param, 10;
27     }
28
29     /**
30      * 可重载
31      */
32     pub doX(string param, int param1) -> string, int throws MyError {
33         return param, param1;
34     }
35 }

```

14、抽象类

一种特殊的类，可以拥有自己的字段和方法，但也有一些无法实现的方法，这些方法需要由继承该类的子类来具体实现

用关键字 `abs` 来定义

14.1、格式

```

1 [pub | pri] abs class AbsClassName {
2
3     [methodName([type paramName [, ...]]) [ -> returnType [, ...]] [throws Error1
4     [, ...]]];
5 }

```

- 访问修饰符

访问修饰符与“类”用法一致

- `abs`

用 `abs` 关键字声明类为抽象类

- `AbsClassName`

接口名称，命名规范与“类”一致

- 方法声明

`pub` 访问修饰符，不可变更，且无需手动指定

方法签名与“类”一致，但此处不能有方法体，方法签名结束需要以 `;` 结束

14.2、实现

抽象类也不可直接创建实例

需要定义类来继承该类并实现响应的抽象方法，才能通过创建类的实例来创建该抽象类的实例

通过符号 `:` 来继承类

必须实现父类声明的抽象方法

```
1  /**
2   * 声明类为 抽象类
3   */
4  pub abs class AbsClass {
5
6      /**
7       * 声明为抽象方法
8       */
9      doX(string param) -> string, int throws MyError;
10
11     /**
12      * 可重载
13      */
14     pub doX(string param, int param1) -> string, int throws MyError {
15         // 已实现的方法
16         return param, param1;
17     }
18 }
19
20 /**
21 * 通过符号 : 来继承类
22 */
23 pub class X : AbsClass {
24
25     /**
26      * 实现父类所声明的抽象方法
27      */
28     pub doX(string param) -> string, int throws MyError {
29         return param, 10;
30     }
31 }
```

15、枚举

一种特殊的类，所建实例均为“常量”

用关键字 `enum` 来定义。

15.1、格式

```
1  [pub | pri] enum EnumName [ ~ InterfaceName [, InterfaceName1 ...]] {  
2      [MEMBER [, MEMBER1 ...]];  
3  
4      [type fieldName;  
5      ...  
6  
7      [EnumName([type param ...]) {  
8          // 构造方法  
9      }]  
10  
11     [[pub | pri] methodName([type param ...]) [ -> returnType ... throws Error  
12     ... ] {  
13         // 方法体  
14     }]
```

- 访问修饰符

访问修饰符与“类”用法一致

- `enum`

用 `enum` 关键字来声明枚举类

- `EnumName`

类名，命名规范与“类”一致

- 实现接口

可实现一些接口，用法与“类”一致

- 枚举值

一些实例

访问修饰符为 `pub`，不可手动指定

需要通过构造方法创建

命名规范与“常量”一致

多个实例之间用 `,` 分割，结尾用 `;` 指定

- 字段

访问修饰符为 `pri`，不可手动指定

其他用法与“类”一致

- 构造方法
访问修饰符为 `pri`，不可手动指定
其他用法与“类”一致
- 方法
用法与“类”一致

15.2、使用

15.2.1、声明

```
1  /**
2   * 普通声明，无字段和方法等
3   */
4  pub enum Color {
5      RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;
6  }
7
8  /**
9   * 复杂声明，有字段和方法
10   * 还实现了 ToString 接口
11   */
12  pub enum ColorWithName ~ ToString {
13      RED("红色"),
14      ORANGE("橙色"),
15      YELLOW("黄色"),
16      GREEN("绿色"),
17      BLUE("蓝色"),
18      INDIGO("靛色"),
19      VIOLET("紫色"),
20      ;
21
22      string colorName;
23
24      ColorWithName(string colorName) {
25          this.colorName = colorName;
26      }
27
28      pub getColorName() -> string {
29          reutrn colorName;
30      }
31
32      /**
33       * 实现 ToString 接口的 toString() 方法
34       */
35      pub toString() -> string {
36          return format("color = {colorName}");
```

```
37     }  
38 }
```

15.2.2、外部调用

```
1  fn main() {  
2      Color red = Color.RED;  
3      // 输出为: red = RED  
4      println("red = {red}");  
5  
6      // 结合 when 使用  
7      when color {  
8          RED -> println("红色"),  
9          GREEN, VIOLET -> println("绿色或紫色"),  
10         _ -> println("其他颜色")  
11     }  
12  
13     ColorWithName cwn = ColorWithName.BLUE;  
14     // 输出为: cwn = BLUE  
15     println("cwn = {red}");  
16     // 输出为: 颜色名称为: 蓝色  
17     println("颜色名称为: {}", cwn.toString());  
18 }
```

15.2.3、枚举值迭代

枚举类内置 `values()` 方法返回所有的枚举值成员为数组

```
1  fn main() {  
2      ColorWithName[] values = ColorWithName.values();  
3      // 可迭代遍历  
4      for index, item in values {  
5          println("index = {index}, item = {item}, item.name = {}",  
6              item.toString());  
7      }  
8  }
```

16、多态

17、注解

18、泛型

19、并发

20、内置函数