

整体学习路径整理及当前工作状态报告

作者 : 王雨泽

日期: 2018/11/28

整体学习路径整理及当前工作状态报告

个人描述

叙事型概要自述

部分历史项目概览

 数据传送单元配置工具

 智能柜控制端

 API测试工具

 迷你ERP及会员管理系统

Alioth语言

 关于模块与引用等整体概念和语法设计

 语法设计

 样例代码

 关于元素原型及数据类型的思考和设计

 盒子说与相关思考

 对象与数据类型

 从集合论谈异常处理机制

 元素与元素原型

 基础数据类型和关于字符的思考

 复合数据类型和内置数据类型

 方法对象

 元组对象

 字面表达法与字符串模板

 整数的字面表达法

 浮点数字面表达法

 布尔类型的字面表达法

 字符串类型的字面表达法

 字符类型的字面表达法

 定义与实现以及关于符号表的思考

 定义与分析

 关于建立语法树的思考

 类定义

 别名定义

 复合数据类型定义

 关于多继承与RTTI的思考

 基于二进制抽象语法树的模块开放策略

 枚举定义

 方法定义

 基于静态类型推导的准方法策略

 基于RTTI参数包的准方法策略

 基于RTTI的虚方法策略

 运算符定义

 构造运算符

 析构运算符

 定位运算符

列构造与列赋值(一)
移动运算符以及关于闭包的思考
成员运算符的重载
其他运算符的正反重载与同构

个人描述

我是一个很任性,很没规矩的人,不喜欢上课,不喜欢写作业,却喜欢学习.

上初中时,觉得为什么学习一定要被定义成上课听讲,写作业呢?

上高中时,我觉得考试的作用是查缺补漏,为什么要特意备考去考个好成绩呢?

上大学时,我觉得正常的,不一定就是正确的,我为什么要成为人们口中的人们呢?

在多数老师眼里,我是一个不懂事,不爱学习,不学无术的坏孩子,在同学眼里我是一个把咒语写得衣服上,本子上,身上哪里都有的中二怪胎.

这些可能没错,有很多地方我可能需要改正,但是正因为我这种任性的性格,我只做我喜欢做的事情,让我把对计算机科学的研究持续到了现在,并继续保有非常浓厚的热情.

虽然长时间的学习生涯让我发现,我其实并没有计算机领域的天赋,但是我却有着一颗天大地大没有兴趣重大的玩心,让我在计算机的学习上投入了更多的时间.同时也感谢生命的每个阶段都会有一位贵人出现在我身边,给我指路.

叙事型概要自述

此部分内容无关当前工作,可略

我的计算机学习生涯是从我12岁时开始的;小时候家里对电脑的管教就比较松散,我从三四岁就开始玩电脑游戏,那是我就觉得,所有的游戏都很好玩,但是都不能让我非常满意,所以幼小的我萌生了自己制作一个游戏的想法.

在那之后,我尝试了很多途径,尝试用PPT的触发器做了一个互动性比较强的演示文稿,也尝试过用3D MAX做3D模型,直到12岁时,随着我对搜索引擎的使用,我发现了游戏和软件之间的关系.与此同时,我在从父亲手里继承来的HTC手机中找到了一个名为*Pocket C*的编译器,在这个古老的WindowsPhone平台上可以编译简单的控制台程序和图形程序.

这时的我,对程序一无所知

从此我便迷恋上了这个手机,我非常热爱把思想中的逻辑转化为机器的行为的快感.**Pocket C**编译器带有大量的例程,我看不懂代码,也看不懂英文,基本没有办法阅读.但是我发现,修改一些数字,参数对例程运行效果的影响非常明显,于是我开始通过不断的修改源代码,查看是否报错,运行有什么结果;这些方式理解了**Pocket C**的语法.也是在这个时候,我发现我不能一下子记住所有变化的规律,我开始养成了记笔记的习惯.

在这个手机上编写了一段时间的程序后,我开始思索,可不可以再电脑上写程序呢?庆幸的是,百度并没有因为这个问题很幼稚而拒绝给我答案,我开始逐渐了解到编程语言的概念,这些概念使我对以前的"玩弄"过程又有了更清晰的认识.在尝试安装了**DEVCPP**编译器后,我开始尝试在计算机上编程.

我偷偷的攒下了一星期的午饭钱,在便利店进行了人生中的第一次网购,买了一本K&R的**The C Programming Language 2 edition**,由于兴趣,我只花了两个星期就把这本书看完了,于是我又开始探索更多的未知.在这过程中,我学习了**OpenGL**,**Socket**这些比较基础的库的使用,初步建立了朦胧的**UI**的概念和网络的概念.

这时的我,还满足于学会了一种语法结构

我写了一个小程序,播放一段录好的音频跟使用者打招呼,又把它设置成开机自动启动,给我父亲演示了我的计算机如何欢迎我的使用,从那以后,我父亲便非常支持我学习计算机.

我在网上看C语言的视频,在各种论坛,问答,群聊中活跃;我认识了一位善良的前辈,他推荐我学习单片机,所以送了我一套单片机开发板,于是我又学习了**51**单片机开发,也研究了计算机和单片机通信的方法,学会了串口开发.

依照我的性格,没有什么特殊原因的,我就开始思考程序的原理,计算机的原理,文件后缀名的意义.于是我学习了**PE**结构,**COFF**结构,**ELF**结构,动态链接库的原理,静态链接库的原理.至于计算机底层,我终于明白**CPU**虽然是计算机的核心,但它却也是功能最少的一个,我学习了**8086**汇编,试图理解总线,编址等概念,但是都比较模糊.

在这之后,一块单片机开发板已经理所应当的不能满足我的好奇心了,我开始向父亲请求资助,在网上购买了很多各种芯片,模块,齿轮,电机,舵机,做了一大堆乱七八糟的东西.这时我发现,最让我满足的事情,就是我制定的协议让本来没法关联的两个设备相互协作了.

这时已经初中三年级了,在这过程中,我还在去爷爷家拜访的过程中学习了**HTML**,**CSS**和**JS**这所谓的前端三宝.

当前辈得知我单片机已经玩的差不多了,他又送给我一个树莓派让我学习**ARM**嵌入式开发.但这个假期,我开始了对操作系统的探索,所以没能开始研究**ARM**.

在初中毕业的假期,我开始研究计算机的启动,了解了**POST**开机自检,了解到了**MBR**主引导记录,在不断的尝试中,我终于写出了自己的主引导器,对当时的我来说足以欢呼雀跃.

在这之后,我想走的更远,我开始搜索书籍,资料,看如何在没有系统支持的情况下驱动屏幕,硬盘,于是我学会了使用**BIOS**提供的中断读取磁盘数据,学会了使用**IO**指令读写各个外设的寄存器,以改变它们的配置.

终于我不满足与**16**位实模式的限制,开始向往**32**位保护模式,在学习保护模式的同时接触到了**64**位长模式,也终于清楚了**x86**架构的总线系统,明白了**PCIe**总线系统对内存空间,**IO**空间和配置空间的管理,以**windows**为类比,思考了很多操作系统对底层的处理.在这期间,我也进行了人生中第一次混合编译,将**C**程序和汇编程序都编译成为中间文件格式,再链接成目标代码,了解到了中间层思想,也进一步了解了编程语言的原理.同时为了顺利编译**ELF**格式的内核,我不得不开始接触我认为非常繁杂的**linux**系统,初步接触了**CentOS**(虽然后来发现它不怎么好用).

假期结束,我的操作系统却没能如愿完工,因为我没办法找到相关资料让显卡进入高分辨率之后,对显存映射开启分页切换(现在明白了其实就是通过**IO**口修改显卡的配置).

在高中阶段的初期,我开始研究**STM32**单片机的开发,作为**ARM**学习的入门.也开始研究一些杂七杂八的极客技术.比如研究了如何通过打开远线程将可执行代码注入到正在运行的程序代码空间中.如何通过打开端口的**SO_REUSEADDR**属性来监听其他应用的通信.

在高中,我也终于开始尝试制作游戏,我学习了**GA**遗传算法和**ANN**人工神经网络,使用遗传算法做了一个寻找迷宫出口的小程序,不过行为并不理想.我也开始学习基于**OpenGL**的**GUI**开发,学习了**Windows**的消息机制和钩子机制.

随着图形设计的深入,我发现正常的**OpenGL API**没办法满足更复杂,更炫酷的渲染需求,我开始接触**GLSL**,开始了解到**GPU**和**GPGPU**的存在,又学习了**OpenCL**的并行开发.

我试图设计自己的游戏物理引擎,这时我终于发现**C++**的语法给我带来的限制,我开始萌生了设计新语法的想法.于此同时,我也尝试着制作过**XML**的解析库,**JSON**的解析库,这些经验也为后来编写语法分析器带来了基础.

制作游戏的事情后来也以失败告终,这导致我自卑了很长时间,后来发现一个人做出来的游戏很优秀的情况其实几乎没有,所以我开始意识到团队的重要性.

高中期间,我也尝试设计电路,自己焊接贴片电子元件,甚至购买了腐蚀剂,覆铜板,想要自己制作电路板,结果失败了...因为没有热转印机,没办法将设计的电路图转印在电路板上(后来上大学终于自己买了一台热转印机).

高中时,我已经接触了很多编程语言.我发现,对于同一种解决方案,不同的编程语言仅仅是改变了表达形式,不会影响思路.

这时的我,觉语言不过是表达思想的工具

出于我对我所学过的所有编程语言,有着如同游戏一样的思考,我开始萌生了设计一种自己的编程语言的想法.但是由于这时对编译原理还一无所知,所以并没有实质性的进展.

高考结束,我总算如愿的报入了我喜欢的专业.刚上大学,我就成了全班唯一整天写代码的奇葩.我参加了电子精英社团,在实验室的支持下,我提高了动手实验的频率,进一步对各种驱动电路,数字逻辑加深了了解.

在实验室得到了第一个任务,帮助老师开发智能农业管理系统的上位机,使用我之前接触的串口通信从容应对,后来这个系统成了一位学长的毕业设计,我又使用**Go**语言和**Sciter**前端库给学长重构了一款更炫酷的前端界面.

在学长的引荐下,我参加了我的第一份工作,在这时,我又详细的接触了服务器与服务的详细概念,学会了搭建服务器,配置服务器

这时的我,因为只用了两个晚上就学会了**PHP**而感到自豪

为公司写了一个高速缓冲层后,服务器端没什么任务了,我又被调派开发下位机驱动程序.

这时,我接触到了**SIM800**通信模块,以此为入口,我开始接触物联网领域.

过了不久,我又被安排了开发**DTU**数据传送单元的配置应用程序.

在这期间,我冒出了想创业的想法,在班主任的引导下,决定先感受以下工作的氛围,再决定是否要创业,于是开始了我的第二份工作.

这次,我的主要任务是在**Tomcat**服务器上编写**Java**服务应用,这时我才开始仔细了解**Java**语言,开始对Java的注解机制产生了兴趣.

开学后,我的时间安排不足以继续支持我在外工作,我便辞退了这份工作,也开始整理从中学习到的经验,思考.

我开始研究编程语言的设计,但是突然有一天与朋友交谈的过程中提到了操作系统,朋友对我做过的项目非常感兴趣,于是我便把以前的成果翻找了出来,给他展示.

在给他讲解其中的原理时,我惊讶的发现,时隔多年,我对这其中的原理的理解居然偷偷的成长了,变得更深入,更准确了,于是我也燃起了热情,重新开始编写操作系统.由于这时我已经开始使用**Ubuntu**操作系统了,所以了解到了**Grub**引导器已经实现了初始化机器,跳转到32位这些工作,我开始思考使用高级语言编写一个操作系统内核.

一开始的工作是顺利的,我使用**C++**开发的微型内核,已经可以从键盘读取输入内容,输出到屏幕上.但是,当我继续更进一步时,**C++**的异常处理机制却阻挡了我,**C++**会为了准确拦截一个被抛出的异常,准备栈帧相关信息,而这个动作,通过一个函数实现,而这个函数,则存在于安腾二进制接口标准中,由一个动态链接库实现.

由于当时的一个疏忽,我没有考虑到**C++**的 `-fno-exceptions` 参数,而错过了这个解决办法,所以又将注意力转回了我自己设计的语言上,期待最终能使用我自己开发的编程语言开发我自己的内核.

而这时,我的一位朋友正巧在修编译原理这门课,于是我对语法范式,语言设计上的很多思考,在他那里得到了验证,语言的设计开始有了更大的进展.

随着我对语法的研究,发现编程语言的语法设计,通常都体现着设计者对程序原理的思考,一款语言的语法,定义了这款语言的行事风格,而不同的行事风格则适用于不同的应用场景,不同的语言特性,可以应对不同的环境限制.

这时的我,认为语法是编程语言的灵魂,而编写程序的过程,就是两个灵魂在沟通的过程.

直到现在,我对编程语言的理解,语法的设计,符号的运用,都产生了一些自己的见解,我很庆幸我正在开发一款编程语言,所以我可以把我的想法注入到编程语言的语法设计中.

部分历史项目概览

由于创建这份概览时,已经脱离原工作环境,所以有些应用能展示的界面非常局限.

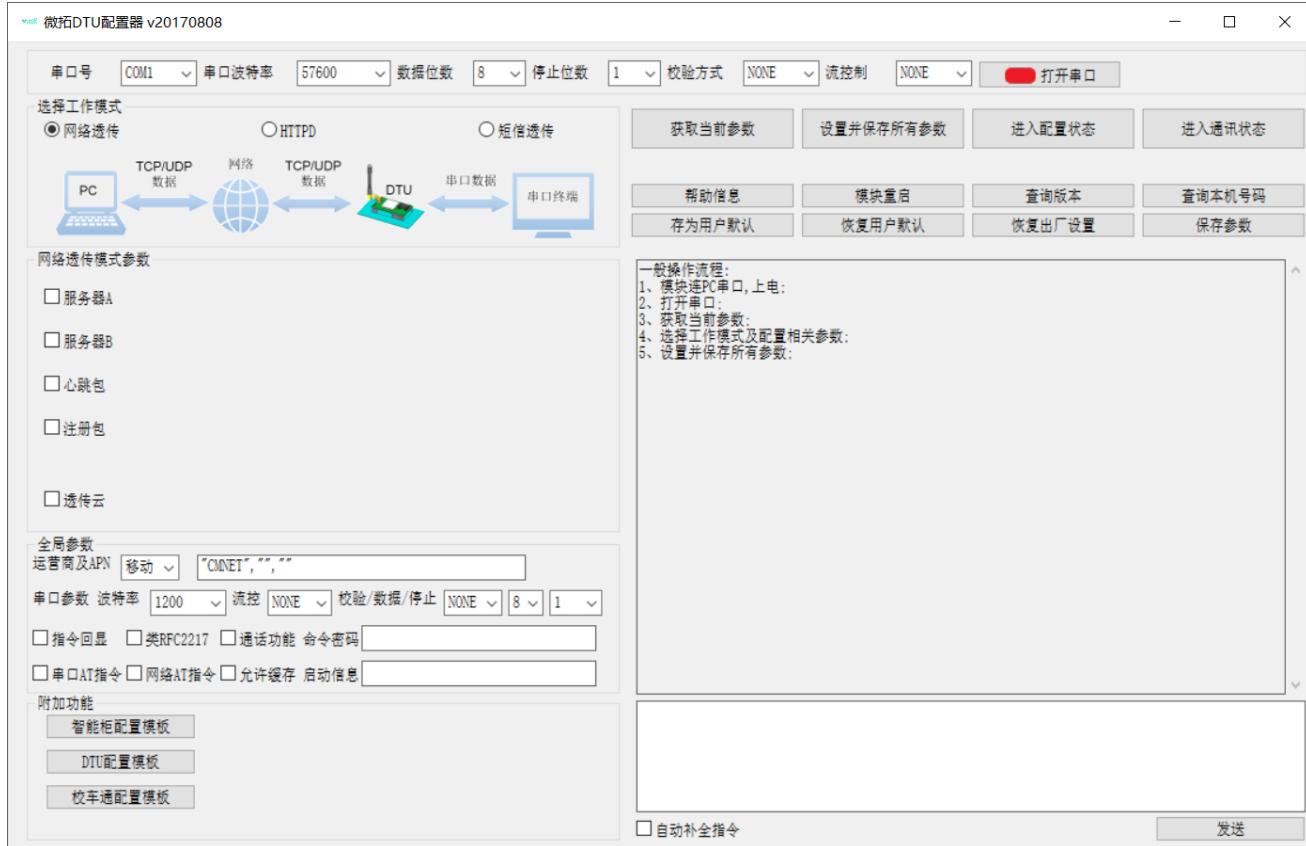
另外还有一部分不带有GUI的程序或组件不在概览范围内.

数据传送单元配置工具

使用RS232协议从计算机串口配置数据传送单元.

此项目需求来自于我当时任职的公司,包括界面设计,架构设计,完全由我一人完成.

最终投入使用.



智能柜控制端

智能柜控制端同样使用串口通信控制智能柜设备.

当串口接通,画面将根据柜子上传的配置信息展示柜子,点击柜门即可打开对应的柜子.

此项目需求来自我当时任职的公司,完全由我一人设计开发,最终投入使用.



API测试工具

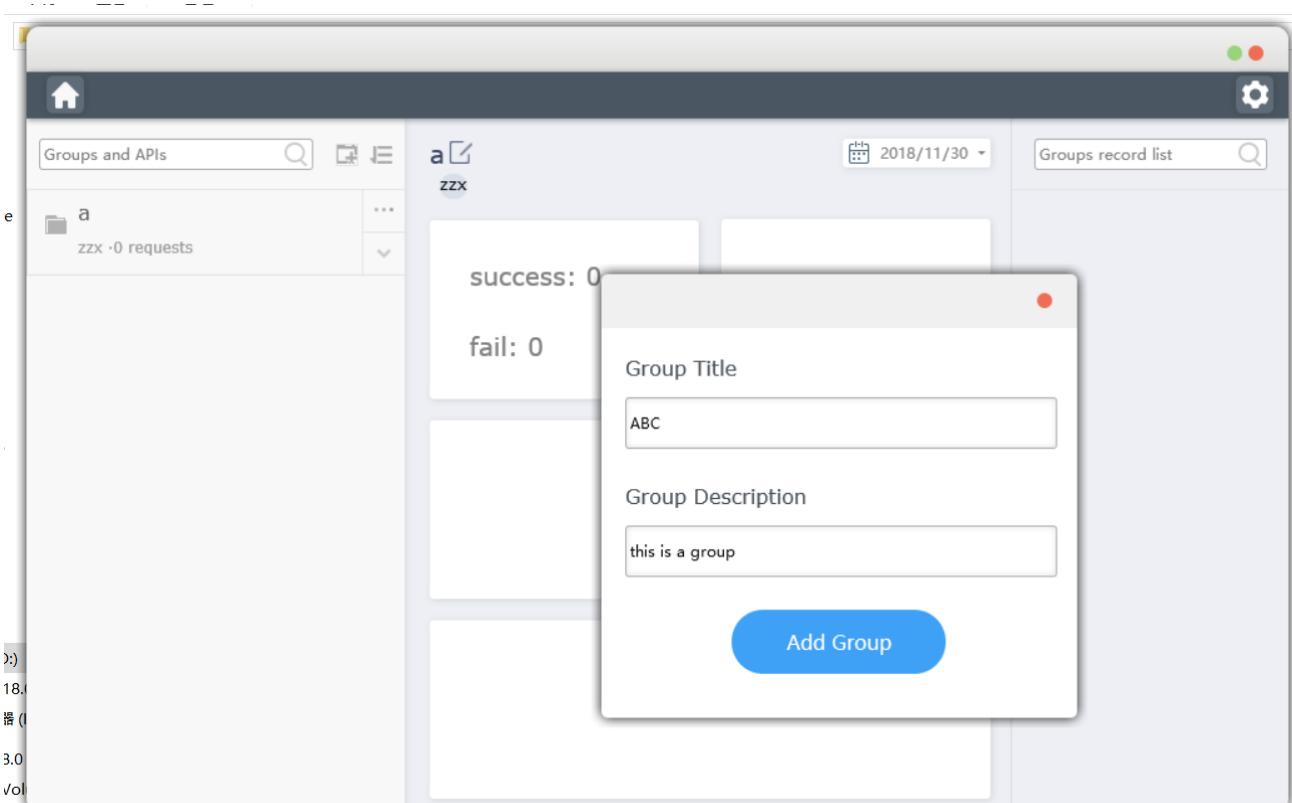
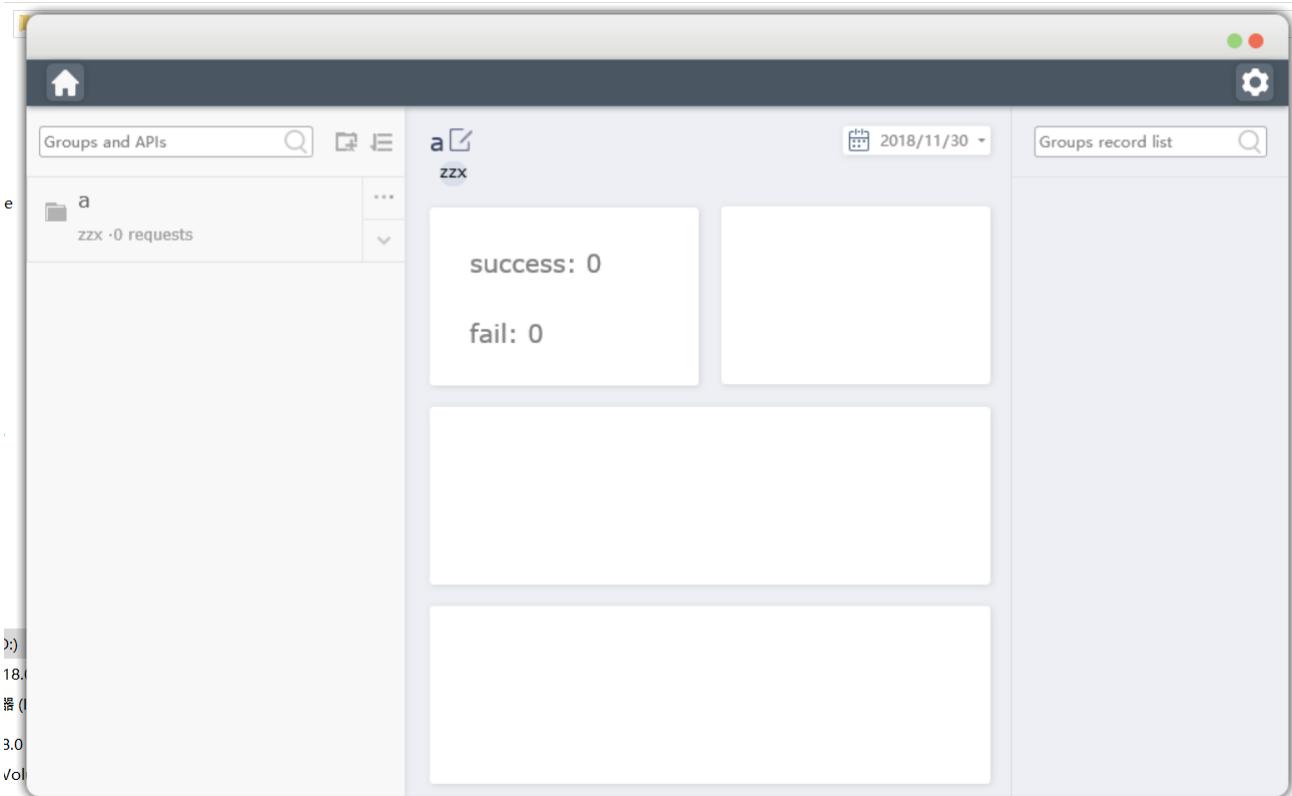
工作在windows上的类似于PostMan的API测试工具.

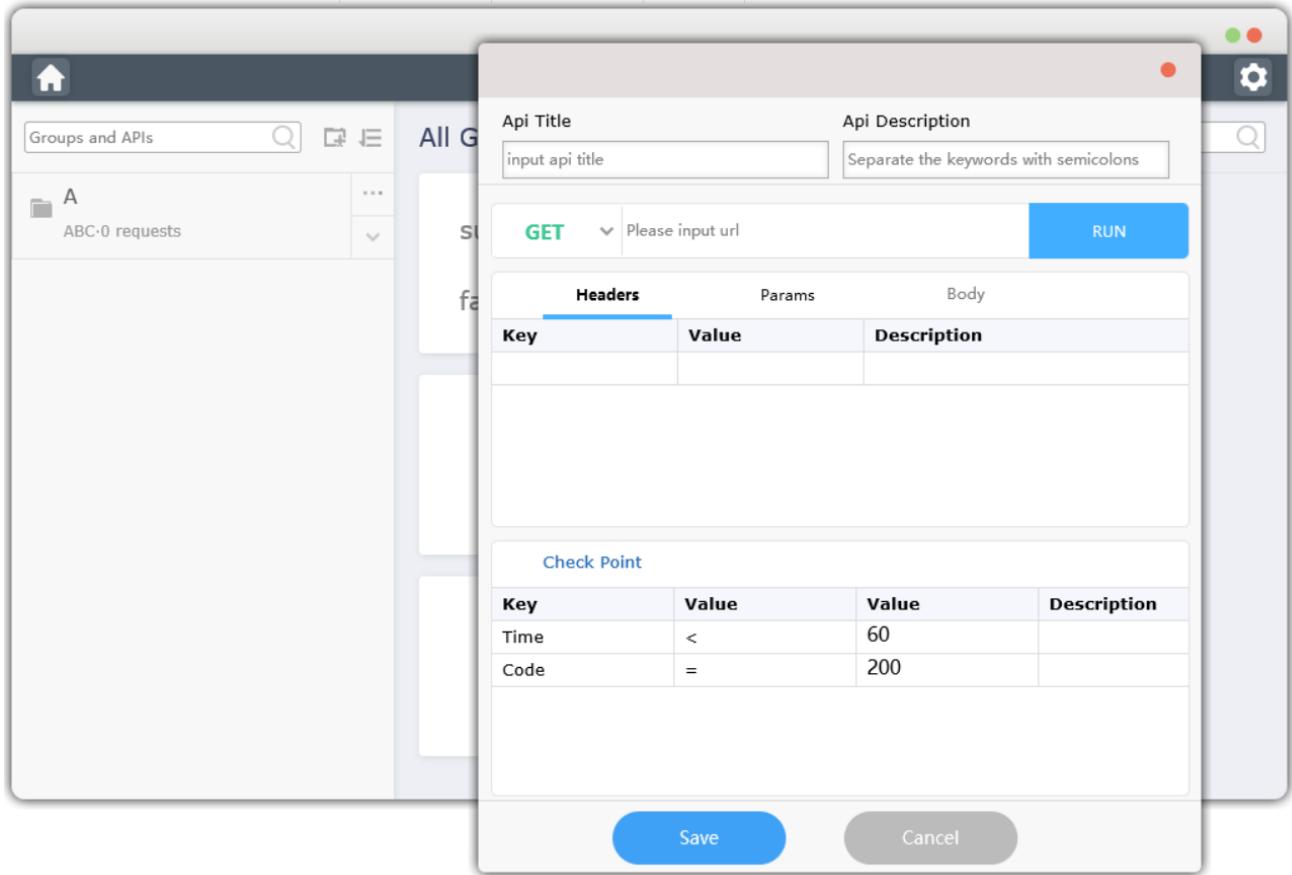
可分组管理网络请求,设定定时请求任务,记录请求的状态日志.

此项目的需求来自于我当时任职的公司,界面的设计由美工完成.

项目使用**C++**作为功能核心,使用**sciter**作为前端框架,由我一人开发.

但是我离职时此项目还没有完工,所以不知道现在有没有人继续把它开发完.





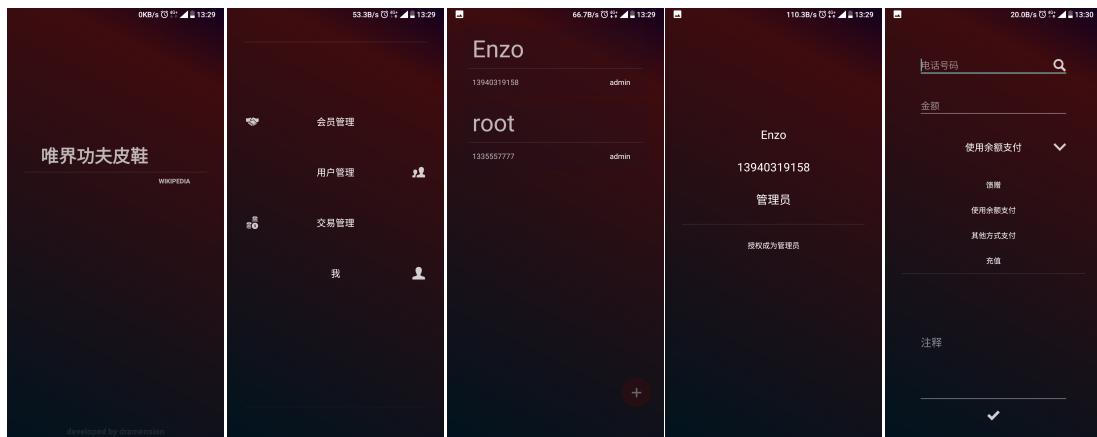
迷你ERP及会员管理系统

此系统包括服务器,客户端和会员端三个部分,其中服务器和客户端由我本人亲自开发,会员端由我的一位朋友友情开发.

此项目需求来自我父亲,整体架构和通信协议由我设计,整个项目由我和一个朋友共同完成.现已投入使用.

其中,服务器使用 PHP 开发,客户端使用 Kotlin 开发,而会员端则完全使用 H5 技术开发.

下列图片均来自客户端截图.



Alioth语言

下述所有思考,论证均来自我个人仅有的知识水平,和理解.

Alioth语言的名字取自北斗七星中最亮的一颗星,玉衡星¹.

Alioth的核心思想是准确表达语义,广泛概括思路.

Alioth的最初语法设计一直都没有相关文档,如果要寻找以前的语法设计,只能到源代码中寻找实现,所以我也很难追溯出一个设计思路的完整成长过程,下述的语法特性多数都只包含最后的设计状态.

关于模块与引用等整体概念和语法设计

Alioth认为任何应用都由模块构成,应用可以从自己所拥有的模块构建成可执行程序,也可以搭配一些向外开放的模块构建一个动态链接库或静态链接库.

Alioth的最小编译单位就是模块,在构建过程中产生的中间代码文件的总数应该等于一个应用所依赖的模块的总数.

Alioth认为,凡是拥有 `.Alioth` 后缀名或在文件头部包含了正确的模块定义的文件,都是源文档.

在同一个应用中,一个模块可以使用多个源文档描述,这可以增强源代码的可读性,也可以隔离同一个模块中,不直接相关联的一些符号.

语法设计

一开始,Alioth像多数高级语言一样,拥有模块定义和模块引用两种语句.

后来,我发现,模块引用语句的作用并不孤立,它的语义是 对模块N存在依赖关系 ,这样的信息单独拿出来是没什么价值的.

如果把一个模块中的所有模块引用语句全添加在一起,它们表达的信息会更完整一些, 对模块N, P, Q存在依赖关系 ,但是者依然不足以编译器做什么.

只有当把模块的定义和模块的引用全放在一起,它们所表达的语义才是完整的: 模块M对模块N, P, Q存在依赖关系 .

所以,Alioth定义:模块定义语句,用于定义源文档所属模块,及此模块的依赖信息.

注: 本文所有的语法都使用EBNF² 描述.

```
label = ?Common C style variable name syntax?;

literal string = '''', ?C style string syntax with special escape sequences?, ''';

literal character = '''', ? C style character syntax with special escape sequences?,
''';

alias = label | "this" ;

module name = label ;

app name = "." | label | literal string | literal character ;

dependency description = module name , [ "@" app name ] , [ "as" alias ] ;

module definition = "module" , module name , [ ":" { dependency description } ], [
";" ] ;
```

样例代码

- 定义一个名为 `Hello` 的模块

由于Alioth语法的特性,所有模块内的定义内容都由关键字引导,所以模块定义不需要特别的结束符,也不会引起歧义.

```
module Hello
```

- 定义一个名为 `Hello` 的模块,依赖于 `foo` 和 `bar` 模块

由于依赖描述的语法特性,多个依赖之间不需要类似于逗号的分隔符

```
module Hello : foo bar
```

- 定义一个模块,依赖与来自不同应用的同名模块

当应用名称的复杂度,超过了一个合法的标签名的描述能力范围时,应用名称可以使用单引号或双引号包括起来.

```
module Hello :
    io @ Alioth
    io @ comm as cio
    sock @ network
    sock @ "mysql connection" as dbsock
    sock @ "ibm:platform tools@3.4.21" as psock
```

- 定义一个模块,存在特殊的依赖

当确切需要从当前正在开发的应用空间中寻找依赖时,可以指定依赖的来源应用名为当前应用名,或者使用 `.` 表示当前应用.

当希望像使用当前模块的内容一样自由使用依赖的模块的定义内容时,可以指定模块的别名为 `this`,要求编译器在两颗语法树之间做符号路由.

```
module Hello :
    basic_string @ Alioth as this
    io @ .
```

关于元素原型及数据类型的思考和设计

盒子说与相关思考

变量是一个盒子,里面装的是数据,不同形状的盒子可以装不同类型的数据.

"盒子理论"在很长一段时间里都作为编程新手对变量的理解形式.

但是,在"引用变量"出现后,它就不再自圆其说了.

若变量是盒子,那引用变量里面装的是什么呢?

对变量 `a`,取引用 `r`.

`a` 和 `r` 都绑定了同一份数据,我们说谁容纳了数据都不满足"盒子用来装数据"的说法.

计算机理论本来就是抽象概念,为了降低理解难度而再次抽象,总会弄巧成拙造成误解.

实际上,变量是用于操作内存中对象的媒介.

在高级语言中,变量也用于限定对象的生命周期.

而引用则与变量类似,是用于操作对象但不限定对象生命周期的媒介.

或者可以说,引用对对象的生命周期的限定有另外的策略,比如Java语言通过**GC**系统将引用与对应的对象的生命周期绑定起来.

由此我们又能联想到指针和右值引用实际上也属于用于操作对象的媒介,这很好理解,因为在**C++**语言里,它们也叫变量.

当然了,变量,引用,指针,右值这些东西本身也都是对象,但是我们发现,它们携带着不同的语义.

- 变量

变量的产生一定会伴随一个对象的构造,变量的消亡总会伴随对应的对象的析构.

- 引用

引用的创建不需要伴随一个对象的构造

引用消亡之前,对应的对象就可能已经析构了

引用消亡之后,对应的对象可能还没析构

- 指针

指针很随意,在生命周期内可以指向任一个对象,

指针也可以随便选择一块地址,指鹿为马.

指针暴露了内存地址线性排布的特性,通过指针可以索引一系列连续存储的对象

通过将指针与特殊的数据结构,如链表,邻接表等结合在一起,也可以将内存中离散存储的对象关联起来.

- 右值引用

右值引用带有特殊的将亡值语义,表示当右值引用的生命周期结束,原对象不再具备意义,只能等待被重利用或析构.

右值引用的存在,主要就是为了表达这个语义,有了这个区分,就不需要专门为这个语义设计一个好看的函数名了.

基于上述思考,我们发现这四者的基本功能是一致的,但又携带了各自风格的语义,不能混淆.

因此,Alioth决定将这四个概念区分开,统称元素,并做出了一些被称为语法盐的语法限制,用于提高Alioth的语义准确度.

对象与数据类型

对象是属性的集合,方法是能在对象上进行的操作.

元素是拥有操作对象的媒介,从集合论的角度可以非常好的讨论关于对象的概念.

首先,定义各个基础数据类型的取值域

$$BIT = \{0, 1\}$$

$$INT8 = BIT^8$$

$$INT16 = BIT^{16}$$

...

基本上,基础数据类型的取值域就是01比特位在特定位长的所有取值情况的集合.

其中的元素,自然就是一个基础数据类型的对象了.

那么如何定义运算呢?

$$R = \{ \langle\langle a, b \rangle, c \rangle \mid a \in INT8 \wedge b \in INT8 \wedge c \in INT16 \wedge c = a + b \}$$

则 R 就是 $int8$ 数据类型的对象的加法运算. 虽然上述 $c=a+b$ 的写法不是很合理, 因为我们既没有定义二进制序列与自然数的对应关系, 也没有定义 $1+1$ 该等于几, 但是篇幅原因, 这里暂且利用常识来描述一些情况.

对于二元关系 R , 其元素的 $first$ 构成的集合称为定义域, 其元素的 $second$ 构成的集合称为值域.

从上述样例我们看出, 方法或者说运算的定义域可以是一个多元序偶, 这很好理解, 因为实际上只接受一个参数的函数也的确不多. 我们不难发现, 由于方法定义域的元素可以是多元序偶, 就导致一个方法的定义域可能会超越任何一个基础数据类型取值域的描述能力, 而且通常是这样的.

这个问题不难解决, 我们取方法的定义域为 C 作为新的数据类型的取值域就行了, 这样就出现了复合数据类型的概念.

从上述思考, 我们总结如下两点

- 数据类型描述对象的取值范围
- 数据类型限制了对象可以进行的操作

另外编程与数学的不同:

- 上述定义中, 一个对象就是一个多元序偶, 其中的元素是不能变化的.
但实际上的编程中, 一个对象的状态一旦改变, 其属性一定会变化.
但是属性的变化可能导致这个对象从某个运算的定义域退出, 这一点数学和编程达成一致.
- 另外, 实际上并不是所有的方法都具有函数特征, 它们的结果可能受到环境的影响.
也即, 方法 M 的定义应该是这样的: $M = \{ \langle\langle args, env \rangle, ret \rangle \mid rules \}$
- Alioth 没办法直接表达一种数据类型的取值范围.
Alioth 要求通过构造运算为对象的成员取值, 利用构造运算的值域
充当对象的取值集合.

从集合论谈异常处理机制

这里也包含了一些对异常机制的思考, 很多高级编程语言都支持异常, 但是并没有

说明什么时候该抛出异常, 什么时候该返回错误状态.

我认为异常机制应该用于弥补编程语言语法规则的不足之处

也就是大多数情况下, 编程语言没办法把方法的定义域和数据类型的取值域一一对应.

当处理不当时, 方法可能会接受到数据类型正确, 但是取值超出定义域内的参数, 这种情况

才应当作为异常处理, 因为不在定义域内的参数, 没有对应的值域元素, 方法不应当给出任何

返回状态.

元素与元素原型

凡是被元素绑定的对象, 都存在于内存中, 因为它们都需要拥有地址

元素是特殊的对象, 因为元素必然拥有一个名称, 在源代码上下文中, 被唯一的符号序列标识.

元素由元素名称和元素原型构成, 元素原型由元素类型和数据类型构成.

元素类型包括变量,指针,引用,右值四种,对应的关键字如下

- `var` --- 变量

变量元素创建之初,一定伴随着对对象的构造.

当变量被闭包而导致移动时,变量元素会与对象解除绑定,变成空元素.

当执行流离开变量所在的作用域时,若变量不是空元素,则变量消亡的同时,会析构所绑定的对象.

变量元素不能用作方法返回类型,因为不合理.

- `ptr` --- 指针

指针元素绑定的就是一个指针对象,可以通过这个对象找到被指针指向的对象.

实际上所有的元素,大多数情况下都是指针对象,少数情况下是类型**ID**和指针对象的组合.

指针元素连同其所绑定的指针对象,对所指向的对象的生命周期没有任何管理能力.

但是指针元素对其所绑定的指针对象的生命周期拥有管理能力,然而并没什么用,

指针对象不过就是个数字,除了构造,析构的时候归零会更安全一点,没什么工作是必须做的.

- `ref` --- 引用

引用元素不能用于构造对象,但是它可以在**GC**系统的支持下延迟对象的析构.

通常情况下,引用元素是一个指针对象,指向所绑定的对象.

存在特殊情况,当引用的指针指向了**GC**空间时,说明对象已经被**GC**系统托管了

此时,引用元素要从**GC**系统取得对象的地址,且当引用元素消亡时,**GC**系统会减少

对象的引用计数,当对象引用计数被清空时,**GC**会负责析构对象.

当被**GC**系统托管的对象在所有的引用元素都消亡之前被析构了,**GC**空间中对象对

应的指针会被置空,这是若引用元素再次被使用,便可以得知对象已经析构,根据程序员

的要求选择抛出异常或放弃操作.(涉及到`?`和`!`两个运算符的使用)

- `val` --- 右值

右值引用不能用来构造对象,但它消亡时,一定会试图析构对象,除非它此时是一个空元素.

右值引用不能用作类成员,不能在动态声明,只能用作参数类型或返回类型.

当调用方法`f()`,传入变量`v`时,若方法`f()`接受的参数类型是右值,则传参所使用的元素会置空,

也就是会变成空元素.在此后的执行流中,对此元素的使用会引起编译器报错,因为编译器认为

这个元素已经离开了这个作用域,虽然实际上它还在,但是由于它被置空了,也就失去了语义.

Alioth是不允许空元素被重利用的.

不同元素类型带有不同的语义,影响着元素的行为特征.

而数据类型则用来限制元素能操作的对象的范围,Alioth是一个强类型语言,就是说不仅对象拥有数据类型,元素也有确定的数据类型,当然由于一些语法糖的存在,元素的数据类型也并不总是确定的,但它总是跟事实相符的,且不会动态变化.

元素概念在不同的语境由不同的语义,语法也有些出入,所以就不在这里集中讲解了,我们会在类定义和参数定义以及实现体中讲解元素不同的语义和语法.

基础数据类型和关于字符的思考

同多数编程语言一样,Alioth提供了一系列基础数据类型供编程人员使用.

如下表展示了Alioth所提供的基础数据类型的相关信息

类型关键字	取值范围	类型描述
bool	true false	布尔类型
int8	$[-2^7, 2^7 - 1]$	8位带符号位整数
int16	$[-2^{15}, 2^{15} - 1]$	16位带符号位整数
int32	$[-2^{31}, 2^{31} - 1]$	32位带符号位整数
int64	$[-2^{63}, 2^{63} - 1]$	64位带符号位整数
uint8	$[0, 2^8 - 1]$	8位无符号整数
uint16	$[0, 2^{16} - 1]$	16位无符号整数
uint32	$[0, 2^{32} - 1]$	32位无符号整数
uint64	$[0, 2^{64} - 1]$	64位无符号整数
float32	$[-3.40 \times 10^{38}, 3.40 \times 10^{38}]$	32位IEEE754浮点数
float64	$[-1.79 \times 10^{308}, 1.79 \times 10^{308}]$	64位IEEE754浮点数

您可能很奇怪,Alioth没有提供类似于 `char` 或 `string` 这样跟字符相关的基础数据类型.

是这样的,字符的编码在不同机器和场景上,非常不统一,其所需要占据的空间大小也不同.Alioth认为,字符并不是为了满足运算需求而产生的数据类型,而是为了展示和存储人类可读的信息而设计的数据类型,所以它不应该是基础数据类型.

Alioth认为应该有一个类,实现了字符的功能,其内部有可能使用 `int8` 来存储一个 `ASCII` 码,也有可能用 `uint16` 存储一个 `UTF-8` 字符,这取决于类的设计者甚至是类的使用者,结合Alioth提供的其他语法特性,这样的思考会增强字符这个概念在Alioth中的灵活性.

从上述思考来看,理论上Alioth中就不应该出现字符串之类的东西了,实际上并不是这样的.Alioth提供了一系列字面对象表达法,用于在程序书写的过程中表达一个取值确定的对象,其中大部分是面向基础数据类型的,而另外一部分则会在编译期间,被自动转换为对应数据类型的对象的构造表达式,这一特性作为Alioth的语法糖存在,在字面表达法小节中会详细讲解.

复合数据类型和内置数据类型

基于基础数据类型,用户可以通过编写类定义来创造新的数据类型,规定其运算规则.

下述语法展示了如何请求一个已经存在的数据类型,而类定义的语法中又包含了Alioth的其他一些语法特性,所以在稍后的章节中会专题讨论.

```
data type =
  { [ "const" ], "*" } , [ "const" ] , ( basic data type | composite type usage ) ;

basic data type =
  "bool"
  | "int8" | "int16" | "int32" | "int64"
```

```

| "uint8" | "uint16" | "uint32" | "uint64"
| "float32" | "float64" ;

composite type usage =
    type name , [ template parameter list ] , { "::" , composite type usage } ;

template parameter list =
"<" element prototype , { element prototype } ">" ;

element prototype =
[ element type ] , data type ;

element type =
"var"
| "ptr"
| "ref"
| "val" ;

```

有些数据类型,为了满足语法特性,应该存在,但是却很难使用源代码描述,这些类型,作为内置数据类型,在使用时,由编译器产生.

方法对象

当把创建一个lambda或将一个方法存入元素中时,编译器会自动创建一个方法对象

方法对象除了可以存储和执行方法以外,还有一个重要的特性保存执行宿主.

设有对象 `obj`,对象由方法 `met`,将对象的 `met` 方法作为返回值返回.

此时对象 `obj` 会被闭包,由**GC**系统托管,因为被返回的方法对象中持有一个指向 `obj` 对象

的引用元素.自然这个方法对象被执行时,就和直接在 `obj` 对象执行 `met` 方法是一模一样的.

lambda表达式语法如下:(语法只限定书写形式,有些错误在语义检查阶段才检查,所以语法阶段不考虑)

```

lambda expression =
    $" , "(" , [ parameter list ] , ")" , [ return prototype ] , executable block ;

parameter list =
    parameter , {"," , parameter } ;

parameter =
[ element type ] , parameter name , [ data type ] ;

return prototype =
    "nil" | element prototype ;

```

其中 `executable block` 是实现的一种,在后面会加以讨论

元组对象

元组对象可以被理解为临时将多个对象绑定在一起回传递.其最主要的用途是在列赋值表达式中被展开.

在方法中,返回一个元组对象,在调用方法时,使用列表达式承接方法的返回值,就可以实现一个方法同时返回

多个对象的效果.

元组拥有独特的字面表达法,就像离散数学中一样,使用一对尖括号包含其中的元素.

样例代码:

```
var tup = <"eric", "clarke", 21, true>;
var idnums = findIdsByInfo( tup );
```

字面表达法与字符串模板

在很多编程语言中,诸如 `123`, `true`, `'x'` 这类的内容,被称为字面常量,这在Alioth中并不完全正确,因为Alioth支持一些字面写法,其对应的并不是一个直接的值,而是一个或几个对象构造语句.

所以,Alioth称这些类型的内容为字面表达法.

整数的字面表达法

Alioth支持二进制,八进制,十进制,十六进制整数的书写形式,如果数字过长,中间还可以穿插单引号用作数字分割.

词法规则如下:

```
binary digit = "0" | "1" ;

octal digit = binary digit | "2" | "3" | "4" | "5" | "6" | "7" ;

decimal digit = octal digit | "8" | "9" ;

hex digit =
    decimal digit
    | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
    | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' ;

literal binary number =
    "0b" , binary digit , { { "'" } , binary digit } ;

literal decimal number =
    decimal digit , { { "'" } , decimal digit } ;

literal octal number =
    "0o" , octal digit , { { "'" } , octal digit } ;

literal hex number =
    "0x" , hex digit , { { "'" } , hex digit } ;
```

浮点数字面表达法

Alioth支持浮点数的指数书写形式.

词法规则如下:

```
float point number =
    literal decimal number , "." literal decimal number , [ "e" , [ "+" | "-" ] ,
literal decimal number ] ;
```

布尔类型的字面表达法

布尔类型的取值范围只有真假,它的字面表达法也限于此两种书写形式,词法如下:

```
literal boolean = "true" | "false" ;
```

字符串类型的字面表达法

Alioth会将源代码中的字面字符串转换成对名为**string**的类的构造表达式的调用.

如果在当前作用域没有找到**string**类,则产生编译错误.

Alioth的字符串支持所有C风格的转义字符,以及一个额外的'\'\$,因为在字符串中,用户可以使用\$()插入一段表达式,实现模板字符串特性.

```
literal string = "" , { character | escape sequence | template expression } , "" ;
character = ?all legal C style characters in literal string except '$'? ;
escape sequence =
    "\a" | "\b" | "\e" | "\n" | "\r" | "\t" | ( "\x" , hex digit , hex digit )
    | "\\" | "\"" | '\'' | "\$" | "\0" ;
template expression =
    $"( , expression , ")" ;
```

这其中,expression的最终结果必须能通过以下两种方式中的一种转换成字符串:

- **string** 类的静态方法 from
- 对象重载了类型转换运算符,可以转换到 **ptr * const int8**

其他纯字符串部分,会使用 **utf8** 编码存入静态数据区,将对应的 **ptr * const int8** 指针传入字符串对象的构造运算符.

样例代码:

```
var age = 21;
stdio << "Hi I'm eric, I'm $"(age) " years old \n";
```

字符类型的字面表达法

字符类型的字面表达法使用单引号包括其内容,其他方面的词法和语法与字符串完全相同,字符字面表达法会被自动转换成名为**char**的类的构造运算符的调用.

一个字符字面表达法,从语法上可以容纳多余一个字符,只是通常的语义不允许而已.

定义与实现以及关于符号表的思考

定义与分析

定义之间是没有先后顺序的

Alioth中的定义确切的,只有类,方法,运算符和枚举四种,并且强调定义无序的概念.

因为Alioth是编译型语言,程序的执行总发生在编译器完全扫描过源代码之后,所以要求所有内容必须先定义再使用就完全没有必要了.

定义的无序性可以简化Alioth源代码的排布,基本上就是随意排布,而不像C++如果存在相对复杂的依赖关系,则各个定义的书写顺序也会变得很复杂.

关于建立语法树的思考

Alioth为一个模块建立语法树的过程大致是

1. 从源代码输入流归约出一个新的语法结构
2. 检查符号冲突等问题,确认无误后将语法结构填入语法树

这时,什么样的代码应该产生冲突呢?

这很复杂,然而经过分析,发现这又很简单.

因为对于类定义和枚举定义来说,只要同一作用域内出现名称冲突,就可以报错了.

但是对于运算符和方法,我们必须要考虑重载的情况.

我们知道,不同方法重载最终的符号肯定是不一样的,但是我们有个很重要的问题.

设有两个方法定义如下:

```
method M( v T ) nil;  
method M( v E ) nil;
```

若 `T` 和 `E` 是同一个类型在不同源代码文件中的不同别名,那么这两个 `M` 就应当是冲突的.

但是第一次建立语法树,是每份文件建立一棵语法树,第二次才是当所有语法树都建立好之后,开始合并.

也就是说,某个 `M` 所在的源文件里可能不包含足以产生冲突的类型信息,甚至可能不包含 `T` 和 `E` 两个类型的定义.

除此之外,方法域名的正确与否也是没办法验证的.

```
method C::A::M() nil ;
```

上述代码表示定义 `M` 函数,属于 `C::A` 这个类,但是在扫描这个方法定义的时候, `C` 类的定义很可能还没被扫描过.

综上所述,虽然同为定义,只有类和枚举的定义可以在扫描之后立刻检查冲突,并添加到语法树中.

而方法和运算符定义的添加过程,应该推迟到所有源代码内的语法树都建立完毕,并且所有的类定义和枚举定义都已经被整合到同一颗语法树上,并检查完冲突之后.

也就是说,方法和运算符的添加,必须在被书写过的定义全都被扫描过的前提下进行.

类定义

类定义的格式有两种,一种是给其他已经存在的数据类型取别名,另一种用于定义新的类型.

别名定义

```
class alias =
    "class" , [ "public" | "private" ] , class name , "=" , data type;

class name = label ;
```

别名定义的主要作用是将复杂的名称化简,当需要改动时,改动别名的目标类型就行了,不会引起大规模代码改动.

样例代码:

```
class private string = dramension::basic_string<int8>;
```

复合数据类型定义

类定义的"复合数据类型"定义格式用于创造一个全新的数据类型.

类定义包含成员的有序声明,描述了此类对象的内存布局.

类定义包含方法和运算符的定义,描述了此类对象的行为特征.

语法如下:

```
class definition =
    class signature , class body ;

class signature =
    "class" , [ "public" | "private" ] , [ "$" ] , class name , [ template definition ]
] , [ ":" composite type usage , { composite type usage } ] ;

class body =
    "{" { layout item definition | method definition | operator definition } "}" ;

layout item definition =
    element type , label , data type , ";" ;
```

其中涉及的方法定义和运算符定义在后续章节会加以讨论;

在类名 `class name` 之前,可选的前缀 `$` 表示此类为抽象类,只有抽象类的所有方法都被实现了的时候,才能被实例化.

抽象类的设计与 `C++` 很相似,其中也可以存在成员布局,这些布局也会被子类继承,只是抽象类的所有方法都相当于 `C++` 中的虚函数,会自动执行最后一层的实现.

类定义的布局中,不能使用 `val`,这不会引起语法错误,但是会引起语义错误.

样例代码:

```
class public map<T> : dramension::hash_map<string,T> {
    var path string;
    ptr next map<T>;
}
```

关于多继承与RTTI的思考

Alioth类的继承关系会被编入目标代码文件,类型描述符表的形式存在.

类型描述符表的个数与实际上存在的类的个数相等,每个描述符表都包含一棵完整的继承树.

这个设计听起来有些不合理,这实在是太浪费空间了,为什么不使用指针复用已经存在的基类的类型描述符呢?

这个问题很简单,假设有 `A` 和 `B` 两个类都继承了一个名为 `X` 的抽象类,那么 `A` 和 `B` 对抽象类的方法的实现是肯定不一样的,至少入口指针绝对不会指向同一个地方,而类型描述符的一个重要功能就是指示抽象方法的入口指针.

所以,类型描述符相同与否,并不取决于这个类继承了哪些类,而是取决于这个类被谁继承了.

类型描述符的结构如下表

偏移量	大小	字段名	语义
0x00	8 B	class id	当前类型的类型id,用于立刻判断
0x08	8 B	layout offset	作为指针或引用距离实际对象首部的偏移量
0x10	4 B	current offset	当前描述符距离描述符表首部的偏移量
0x14	2 B	descriptor size	本描述符大小
0x16	2 B	(<code>m</code>):abstract method count	当前类型中的抽象方法的个数
0x18	m^*8 B	abstract method descriptor table	抽象方法描述符表

类型描述符表结构如下表

偏移量	内容	语义
0x00	类型描述符	对象实际类型的描述符
...	描述符表(s)	对象继承的基类的类型描述符表,这些表之间的顺序随意
...	0	表示此类型描述符表结束

Alioth的类型描述符设计与C++的虚函数表设计相比开销会大一些,因为Alioth需要处理一些对类型一无所知的情况.

比如当一个 `准方法` 被闭包时,传参就不能静态推导类型了,而是改成了传入RTTI参数包 `{类型ID, 对象指针}`.

这种情况下,编译器完全没办法静态的知道这个对象指针到底指向了实际对象的哪个位置,甚至不知道对象的实际类型是什么.

类型描述符表的作用就是将这些信息关联起来,类型描述符表,编译器不仅可以得知当前传参所使用的元素的数据类型如何,也可以知道对象实际上的数据类型如何,同时也可以迅速得知要指向实际对象的首部应该移动多少指针.

您一开始可能会觉得 `layout offset` 这个字段的设置有些多余,实际上它避免了编译器为了恢复实际对象指针,在代码中插入一整段 `switch` 判断语句的尴尬场景.

另外一个值得注意的点,Alioth不会在对象中间穿插类型描述符指针,而是在需要的时候,把它放在元素里.³

这样的好处就是可以避免对象中的类型描述符指针被破坏掉,导致程序崩溃.

在这里思考一种语法假设:

下述代码实际上判断的是v这个元素的数据类型,而v可能只是实际对象的一个基类元素.

```
assume v as ..... { ..... }
```

可以设计如下语法,用于判断实际对象的数据类型,并引导一个v被修正为派生类元素的语境

```
assume .v as ..... { ..... }
```

另外,Alioth进行类型转换的语句语法如下:

```
prototype conversion =
    expression , "as" , element prototype;
```

其语义是转换数据类型

其优选方案,是自然转换,也就是对于基础数据类型,尽力保证数据不出错的情况下转换数据类型.

对派生类和基类之间的转换,偏移元素内部的指针,(同时如果元素携带了类型描述符指针,也要切换)

如果要转换的两个类型之间不存在继承关系,则寻找对应的类型转换运算符 `as` 运算符的重载:

```
prototype conversion operator =
    "operator" , "as" , data type , "{" , { expression } , "}" ;
```

基于二进制抽象语法树的模块开放策略

在C++中,模板类和模板函数无疑是强有力的工具,针对某个问题,使用模板函数,很轻松的就能构建出一个瑞士军刀一样的工具方法来.

但是,C++编译器必须指导模板函数的源代码才能在使用时产生对应的机器码.这对于一些闭源项目或处于代码整洁度的考虑来说都不是个好消息.

Alioth其实和C++是一样的,模板类的模板参数在确定下来之前,完全没办法检查它占据多大空间,它是否满足模板类对它的所使用.

但是,实际上,我们不需要源代码,我们需要的只是抽象语法树而已.所以我做出了折中方案,Alioth在编译静态链接库时,针对模板类会采取特殊的方案.

编译器会把它们的抽象语法树信息使用 `json` 格式表达出来,然后转换成 `abon`⁴ 格式,存入目标代码文件的特殊的数据段中.

当用户希望使用某个闭源库中的模板类时,编译器会自动从目标代码文件中提取抽象语法树信息,用于产生机器码.当最终产生可执行程序时,再将抽象语法树信息数据段剔除掉.

枚举定义

Alioth中的枚举可以设定为可回滚特性,但是是否存在使用价值,还有待考证.

下述代码展示了一个简单的枚举定义:

```
enum Gender {  
    Male  
    Female  
}
```

方法定义

Alioth中也存在主方法,可以用作可执行程序的入口.

但是Alioth不要求主方法有特殊的名字,而是要求主方法使用 `entry` 关键字标记.

Alioth中的方法定义语法如下:

```
method definition =  
    method signature , ( ";" | method body ) ;  
  
method signature =  
    "method" , { "public" | "private" | "entry" } , method name , parameter list , ;  
  
method name =  
    label , { "::" , label } ;  
  
method body =  
    "{" , { statements } , "}" ;
```

其中 `statements` 表示语句,基本上包含所有可能出现的实现,后面会加以讨论.

当 `return prototype` 被省略,或 `return prototype` 中的数据类型被省略时,方法为虚,表示可以返回任意类型的数据.

当 `parameter` 的数据类型被省略时,方法为准,表示可以接受任何数据类型的参数.

准方法的实现策略又分为静态和动态两种.

基于静态类型推导的准方法策略

当编译器可以通过语法树找到一个准方法的实现内容时,此准方法采用静态类型推导策略.

这种情况下,准方法原本只存在于抽象语法树中,不能用于产生机器码.

当扫描其他某个方法时,出现了对准方法的传参调用,编译器首先寻找与传参类型匹配的方法重载.

若寻找无果,但是发现了一个准方法,则编译器使用已知的参数数据类型,剔除准方法中的完全不可能执行的 `assume` 语句分支,将新产生的抽象语法树作为一个实际存在的方法重载,添加进入抽象语法树.

若存在类型 `T` 对一组相连的 `assume` 语句分支都不成立,则编译器报错.

基于**RTTI**参数包的准方法策略

当用户将准方法存入方法对象,或者任何lambda在设计之初就是准,则不能推测方法对象的运行时内容值,也就无法寻找其相关联的语法树信息.

解决方案是,当准方法被存入方法对象,将产生一个使用**RTTI**参数包传参的方法.其中所有原本元素数据类型不确定的参数,都改成由对象指针和对象类型**ID**构成的**RTTI**参数包.

此时方法中的 `assume` 语句的全部分支都会被保留下,被翻译成开销等同于 `if-else` 的动态类型判断语句.

此时,任何一组 `assume` 语句最后必须拥有 `otherwise` 分支,用于处理未考虑的传参.

基于RTTI的虚方法策略

Alioth中,虚方法的产生,几乎不改变开销,尤其是在所有返回对象都是复合数据类型的情况下.

Alioth中虚方法的返回策略是,在所有参数之前,添加一个指针参数,指向一块由调用者开辟的空间,其容量应该大于虚方法可能返回的最大对象所需的空间大小,虚方法使用此空间存放返回的对象.

这也是为什么西方法的开销几乎不变的原因,因为大多数复合数据类型的对象,原本就是被这种策略返回的.

同时虚方法从 `RAX` 寄存器返回一个类型ID用于RTTI,而调用虚方法的元素,会将此值与之前的返回对象指针组合在一起,形成一个RTTI参数包,可供 `assume` 语句判断类型.

运算符定义

运算符可以说算得上是Alioth的灵魂所在,Alioth的很多功能都使用运算符重载来实现,而且Alioth的运算符定义非常灵活,使用也非常灵活.

Alioth中的运算符种类繁多,作用也各不相同.

构造运算符

Alioth从0.4语法设计开始,做出了大胆的突破,将构造方法剔除,改使用构造运算符.

其主要原因就是,运算符的语法可以设计的非常灵活,非常好看.

Alioth中的构造运算符在使用时,有如下特点:

- 构造表达式使用 `{}` 或 `[]` 包含,语义直观
- 构造表达式使用时完全不需要考虑参数的顺序

构造运算符要考虑如下几种情况:

- 常规构造,传入不同的参数,参数名带有不同的语义
- 列构造,传入一系列参数,名称没有意义
- 拷贝构造,从一个实例执行拷贝语义
- 移动构造,从一个实例执行移动语义

上述四种情况对应了四种构造运算符签名的格式.

- 常规构造运算符

同一个类的所有常规构造运算符的参数名称在同一个名称空间内,即不能重复.

同一个类中,可以不传递任何参数的构造运算符只能有一个.

常规构造运算符中的任意参数都可以通过指定默认参数值而省.这得益于构造表达式中参数的顺序不与构造运算符中的定义一一对应的特性.

语法范式:

```
constructor signature =
    "operator" , "{}" , "(" , parameter list , ")" ;
```

样例代码:

```
operator {} ( pname string, age int32 );
```

构造运算符也可以接受参数包,在构造表达式中,参数包使用方括号包括.

```
operator {} ( pcaption string, ... pnamelist );
```

常规的构造运算符通过书写构造表达式来隐式调用,虽然构造表达式看起来和构造运算符非常相似.

构造表达式的语法如下:

```
construct expression =
  "{" ,
  [ composite type usage , "|" ] ,
  [ parameter name , ":" , expression ,
  { "," parameter name , ":" , expression } ,
  "}" ;
```

其中 `composite type usage, "|"` 是类型声明,当编译器可以通过上下文推断构造表达式的类型时,此部分可省.

样例代码:

使用构造表达式引起对象构造:

```
var member = { App::Member| name:"ezr", rank:0 };
```

- 列构造运算符

列构造表达式接受可变参数包⁵,使用方括号引导的构造表达式调用.

语法范式:

```
list constructor signature =
  "operator" , "[...]" , "(" , label , ")" ;
```

样例代码:

```
operator [...] ( args );
```

列构造运算符通过书写列构造表达式来调用

样例代码:

```
var names list<string> = [ "ezr", "lzl", "tracy" ];
```

- 拷贝构造运算符

拷贝构造运算符是当一个对象从一个变量,或指针,或引用构造时隐式调用的运算符.

语法如下⁶:

```
copy constructor signature =
    "operator" , "{=}" , "(" , label , ")" ;
```

样例代码:

```
operator {=} ( another );
```

- 移动构造运算符

移动构造运算符是当一个对象从右值元素构造时隐式调用的运算符.

语法如下:

```
move constructor signature =
    "operator" , "{<}" , "(" , label , ")" ;
```

样例代码:

```
operator {<} ( another );
```

上述的构造运算符都只描述了构造运算符的签名语法,并未提及构造运算符的实现.

这是因为不同的构造运算符,其实现是相同的,所以在这里统一讲述.

首先,对象构造的流程应该如下:

分配空间 -> 传递构造运算符的参数 -> 初始化成员对象 -> 执行构造指令

其中, 执行构造指令 所指的,就是执行构造运算符内的可执行语句.

而在这之前的 初始化成员对象 则是指,在可执行语句操作成员对象之前,为了保证其可用,必须对其进行构造动作.

我们必须有办法正确表达构造运算符希望自己的成员对象如何被构造,否则一来有可能增加开销,二来有些成员对象的构造运算符必须传入参数,只有程序员知道如何传递正确的参数.

如此,我们把构造运算符的实现分成了两个部分, 初始化列表 和 构造指令 .

其中, 初始化列表 描述成员对象该如何构造,未提及的成员对象,尝试调用默认构造运算符.

构造指令 则是在对象的所有成员对象都构造完毕后,要执行的后续指令.

构造运算符语法设计如下:

```
constructor body =
    "{" , [ initial list ] , [ "|" expressions ] , "}" ;

initial list =
    label , ":" , expression , { "," , label , ":" , expression } ;
```

样例代码:

```
operator {} ( name string, age int32, gender Gender ) {
    mname : name,
    mage : age,
    mgender : gender |
    stdio << "new member named $(mname) constructed \n";
}
```

析构运算符

实际上析构运算符并没有什么值得一提的语法特性,不过为了迎合Alioth主打运算符的风格,为其设计了一个好看的签名

样例代码:

```
operator {~} () ;
```

定位运算符

元组`tuple`是一个特殊的内置类型,在思考其相关特性时,我遇到了如下问题.

设有元组`t` 其类型为`tuple<string,int32,bool>`,如果用户希望从中取出类型为`string` 的第二个元素,显然是错误的.

但是如何才能让这样的错误在编译期间就被报告出来呢?

这取决于我们对 取数据 这个动作的理解是怎样的,当我们将其理解为 取数据(索引)->类型 这样的语义时,无论怎么努力,想报出编译错误都很牵强.

于是,我尝试了这样的理解方式 取索引为n的数据() ->类型 这样一来,如果不存在 取索引为1的数据() ->string 这个重载,编译必然报错.

简单来讲,我们将本来应该作为参数,运行时才传入方法的信息,与方法绑定在了一起,使之成为方法的身份标识的一部分.实际上,C++中同时允许使用变量和类型作为模板参数的用意就是这样的,这可以增强语义检查的强度,提前报告更多的错误.

于是,定位运算符的语法设计如下:

```
locate operator =
    "operator" , "#" , index , "()" , ";" ;

index =
    decimal digit , { decimal digit } ;
```

样例代码:

对`tuple` 类型来说,当`tuple` 类型被使用时,对应的定位运算符也会被自动重载.

```
var row = <32,"bob","This is a mail from bob">;
ref msg = row#2;
```

列构造与列赋值(一)

Alioth中不存在`,` 引导的逗号表达式,在Alioth中,逗号用于构造一个列,而列可以参与列赋值.

样例代码:

```
var a int32;
var b int32;
var c int32;
a, b, c = 3, 4, 3+4;
```

这样的代码很好理解,但是看起来有些啰嗦.

这些代码还可以写成这样

```
var a,b,c = 3, 4, 3+4;
```

这是Alioth中的第二种列构造,不同于之前所提到的.

但是这种格式不推荐用于基础数据类型,因为这样的格式要求所有的数据类型都自动推导,不能显式声明.

无论是列构造还是列赋值,其右侧都可以是一个元组,这就是为什么要在这里提到列的概念,因为列赋值或列构造表达式会自动调用元组对象的定位运算符,将取出的值一一赋给左侧对应的对象.

这个语法特性用来在一个方法中同时返回多个对象非常方便.

```
var err, obj = getObject();
```

移动运算符以及关于闭包的思考

闭包就是能读取其他函数内部变量的函数.

在函数式编程中,我们经常设计各种高阶函数⁷来实现科里化.

很多时候被返回的函数还需要继续使用它被创造时,所在环境中的某些变量,这就需要对环境进行闭包了.

在很多动态语言中,闭包得到了很好的支持.因为在这些动态语言中,对象是不是真的一定要在执行流离开作用域后被析构,不过就是一个条件句的问题.

而静态语言却不一样,存在于栈中的对象,当执行流退出其所在的作用域,就意味着对象所处的栈空间即将被释放,即使不析构对象,也没办法再正常访问它了.

Alioth提供了一种策略,当一个对象被闭包时,通过移动⁸的手段延长其生命周期.

具体来说,就是当一个对象被闭包,它会被移动到GC为它开辟的新空间中,这个空间通常在堆中.如此依赖,这个对象就可以继续被访问了.

但是,可能存在一种情况,即对象的地址信息可能需要总保持正确.Alioth提供了移动运算符作为对象移动之后的回调方法,用来给程序员处理对象地址变化事件的机会.

成员运算符的重载

C#语言拥有**getter**和**setter**机制,用于拦截对成员对象的赋值和提取操作.

这个想法非常好,很大程度上简化了代码,但是它将两者两种运算与成员变量的定义捆绑在一起,却让人觉得混乱.

Alioth支持类似的设计,与之恰恰相反的是,**getter**和**setter**的名称不能与任何已存在的对象名重复,那会产生一个符号冲突.

通过重载成员运算符,就可以为类绑定一个虚拟的成员,每次提取它,都会触发对应操作来搜集信息,每次存储它也会触发对应操作来适应变化.

Alioth使用成员运算符的重载是否接受参数来区分它是getter还是setter,语法设计如下:

```
member operator =
    "operator" , "." , label , "(" , [ parameter ] , ")" , [ return prototype ] ;
```

样例代码:

一个与前端界面的文本编辑框绑定的对象

```
class Editor {
    var handle uint32;
    operator . text () string {
        return GetWindowText(handle);
    }
    operator . text ( v string ) {
        SetWindowText( handle, v );
    }
}
```

其他运算符的正反重载与同构

Alioth支持 `+-* /%^&|` 等很多运算符的重载,这在C++语言中很常见.

假设有 `C` 类重载了 `operator + (int32)` 运算符,则与 `C` 类型对象 `o` 相关的下列语句合法:

```
o+3;
```

但是如果我们希望如下代码也合法呢?

```
3+o
```

在C++中,我们不得不重载一个全局运算符,并让它成为 `C` 类的友元,以获得其内部非公开成员的访问权限.

Alioth认为,即使这不会破坏面向对象封装的开闭原则,它也破坏了类型封装的整体美感.

Alioth设计了一种语法,可以重载运算符,使用时,运算符应当出现在对象的另一侧.

我们使用了 `$` 符号,它的语义就是 考虑事物的另一面,或两面都考虑 .

语法很简单,只要在希望重载的运算符之前,加上 `$` 符号,就表示,这是一个反重载.

样例代码:

```
operator $ + ( int32 );
```

当表达式中,运算符的两侧对象分别定义了这个运算符的正反重载,编译器根据运算符的结合性方向选择表达式两侧的哪个对象成为运算符的宿主.

另外,如果一个运算符,正反重载实现的内容完全一样,那重写一次显然是在浪费时间.

Alioth提供了运算符正反同构来应对这样的情况.

在重载运算符的后面加上`$`符号,表示此运算符正反同构.

样例代码

```
operator + $ ( int32 );
```

-
1. [https://en.wikipedia.org/wiki/Epsilon_Ursae_Majoris ↵](https://en.wikipedia.org/wiki/Epsilon_Ursae_Majoris)
 2. [https://en.wikipedia.org/wiki/Extended_Backus%20%93Naur_form ↵](https://en.wikipedia.org/wiki/Extended_Backus%20%93Naur_form)
 3. 有待进一步确认是否合理 [↵](#)
 4. **abon**是Alioth Binary Object Notation之意,是二进制版的**json**,信息密度更高 [↵](#)
 5. 新想法,若可以为可变参数包限制数据类型就可以简化一些设计 [↵](#)
 6. 此处的语法设计与README中有所出入,是最新的想法 [↵](#)
 7. 在编程领域,返回函数的函数被称为高阶函数 [↵](#)
 8. 当lambda表达式使用了某个元素,且这个lambda被传出当前作用域这两条条件同时满足,对象才会被闭包 [↵](#)