

华中科技大学

硕士学位论文

UEFI系统脚本语言与解释器的研究与实现

姓名：陈庆

申请学位级别：硕士

专业：软件工程

指导教师：肖来元

2011-01

## 摘要

在统一可扩展固件接口(英文名 Unified Extensible Firmware Interface 或 UEFI<sup>[1]</sup>)的实现中,基本没有相应的脚本系统的支持,对于一个 UEFI 的开发者来说,编写 UEFI 可执行程序既费时又费力,这自然成了一个很大的不便,长远来看不利于 UEFI 快速的普及和发展。与此同时,UEFI 脚本语言的出现可以大大改善目前的现状。

通过对当前流行的脚本语言的归类 and 对比分析,总结出它们优点和缺点,同时结合 UEFI 开发者面临的日常工作需求,来设计出了 UEFI 脚本语言的语法规则和内在特性。简洁和易用是 UEFI 脚本语言的核心,为了达到这个目的,UEFI 脚本语言将只变量区分为三种,并且涵盖了数组,输入,输出,和基本的条件控制语句和函数调用等功能。即便如此,UEFI 脚本语言是足够可以解决我们 UEFI 开发者当前所遇到的一些困难和麻烦。

一个可以工作的脚本系统的除了它自身语言语法的设计,同时也要包含其解释器的设计和实现。其实解释器和编译器在很多方面都共同点,通过对编译原理的详细研究和探讨,以及对 UEFI 编程环境的深入分析,UEFI 脚本解释器主要包含:词法分析,语法分析,符号表,语法树等部分,它们当中的每一个在脚本解释的阶段都起着非常重要的作用,因此这些部分便是 UEFI 脚本解释器的关键和核心。

为了验证 UEFI 脚本系统的正确性和可工作性,按照 UEFI 所规定的语法,设计出了一些测试用例,从测试的所得出的结果来说,基本上达到了 UEFI 脚本解释器的预期目标:可以有效的减少 UEFI 程序员的工作量;可以降低 UEFI 开发的门槛;可以加速 UEFI 的普及和发展。

**关键词:** 统一可扩展固件接口      脚本语言      解释器

## Abstract

In the Unified Extensible Firmware Interface implementation, there is very limited scripting support system, for developers it is a great inconvenience, it is not conducive to rapid spread UEFI's development. At the same time, the emergence of UEFI scripting language can greatly improve the current situation.

Based on comparative analysis and the classification the current popular scripting language, summing up their advantages and disadvantages, and combining with the UEFI developers daily work demand, to design the UEFI scripting language grammar rules and characteristics. Concise and easy using is core of UEFI scripting language. In order to achieve this goal, UEFI scripting languages has only three kinds of variables, and only cover the array input, output, and basic conditions control, and function calls etc. Even so, UEFI scripting languages are enough for UEFI developers to solve our current encountered difficulties and problems.

A scripting system can work besides its grammatical design, also including the design and implementation of the interpreter. Actually interpreter and compilers are common in many aspects. Through detailed research and exploration, and analysis of UEFI programming environment, UEFI script interpreter, mainly includes: lexical analysis, grammar analysis, syntax analysis, the symbol table, every part plays a very important role in every stage of explanation, so these are the core of UEFI script interpreter.

In order to verify the correctness and workability of the UEFI script system, according to specified grammar of UEFI, designing out some test cases. From the test result, it basically achieved the expected goal of UEFI script interpreter: to reduce the workload of UEFI programmers; reduce threshold of UEFI development: accelerate its development and popularization.

**Key words:** Unified extensible firmware interface (UEFI)      Scripting language  
Interpreter

## 独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期： 年 月 日

## 学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于 ☐ 保密， 在\_\_\_\_\_年解密后适用本授权书。  
☐ 不保密。

（请在以上方框内打“√”）

学位论文作者签名：

日期： 年 月 日

指导教师签名：

日期： 年 月 日

## 1 绪论

### 1.1 课题来源与研究背景

本课题来源我在 Intel 公司实习期间, 对该公司推出的下一代 BIOS<sup>[2]</sup>, UEFI/EFI 系统的开发语言局限性思考之后, 决定为它定制适合该系统的脚本语言。

UEFI 采用 C 语言语法, MakeFile<sup>[3]</sup>和微软 VC 编译器来开发。开发人员可以为该系统开发硬件驱动, 应用程序, 组件等等。即便是一个“Hello, World”程序, 程序员都要写上几十行代码加几个额外的配置文件(编译用), 而且要对系统的提供给编程人员的接口要特别熟悉, 否则将无从下手。这样的话, 极大的降低了开发人员的日常工作的效率, 不利于 UEFI 的迅速普及和发展。

这样, UEFI 脚本语言的出现, 就减少开发者对 UEFI 系统的 API 过多的关注和依赖, 不必写复杂的配置, MakeFile 文件等工作。由于脚本易于学, 易用的特点, 对于开发者来说, 这样以来可以极大的提高他们工作效率, 对于 UEFI 的发展也起到了推动作用。

早在 90 年代中期, Intel 就因为传统的 PC BIOS 的存在的一些先天性的缺点: CPU16 启动模式、依赖 PC AT 的硬件、只能达到 1MB 寻址空间等问题, 无法为大型安腾服务器提供必要的支持时, 进而提出了 Intel Boot Initiative 的项目, 后来更名 EFI。直到 2007 年, Intel 将 EFI 标准的制定和修复的权力交给 UEFI (一个非营利性的合作贸易组织, 其委员会的委员包括 AMD, 美国安迈, 苹果, 戴尔, 惠普, IBM, 超微, 英特尔, 联想, 微软, 凤凰科技十一个公司) 管理, 到了如今, UEFI 就是一个可供开发商任意扩展的, 有统一标准的固件接口规范, 它采用很多当前比较流行的系统开发技术, 例如: 它具有很多模块、生成最终文件的形式是以动态链接的方式, 它具有我们比较熟悉的堆栈方式来支撑这样一个系统。

即便为 UEFI 编写一个类似“hello world”非常简单的 UEFI 应用程序, 都是非常复杂和繁琐的, 例如, 你需要一个 inf 文件为其添加项目需要的各种配置, 需要修改该项目中 MakeFile 选项, 需要了解 UEFI 的整个框架结构, 系统调用。由于这么

多的依赖，就需要开发人员通读 UEFI 系统的种类繁多文档，这样的就极大的降低了开发人员的效率，所以针对 UEFI 系统的脚本语言的出现对他们来说是一种解脱，同样也利于 UEFI 的快速发展。

脚本语言(Scripting language)是一种新兴的计算机编程语言，它也是一种能让开发者比较快速的编写出让电脑听命行事的程序，它的原则是用简洁而又快速的方式去完成某些复杂的事情，由于这项原则，使得脚本语言不论从入门或者到实际运用，它都比 C/C++语言或 Java 之类的高级编程语言要更加容易。UEFI 脚本语言就是基于这样一种思想来设计和实现的。

## 1.2 脚本语言的定义及产生意义

脚本语言是也一种编程语言，它不需要通常的编译步骤，它是解释运行的。目前大多数可执行程序需要生成可执行文件后才能运行。与之相反的是，基于脚本语言的程序是在解释的过程中运行的。在最早的阶段，脚本语言时常被大家称之为批量处理语言或工作控制语言。

脚本语言的发明者在发明脚本语言的时候，目标是把编程按照基于组件的方法，即与其为某个的可执行程序编写成非常多的代码，替代为一种将可执行程序分割成足够小的，它们各自都具备各自功能的，可以很多次的重复使用。这些小的模块小到可以达到一些单独的可执行程序的要求，其余的每个部分都可在这些小的模块的基础上产生。不同的模块有着很多不同的功能，目的是让它们有着不同的用途，同时可为被其它的可执行程序所利用。于此同时，它必须要具备非常好的扩展性，可以让用户增添自己的功能。到了最后，终究需要用一种强的，比较灵活的东西把这些模块“粘”合在一起，使各个模块之间通信变得非常的快速和便捷。从上面的这个过程来看，脚本语言的设计有如拼图游戏一样。

通常来说，脚本语言相对于大家熟知的高级或者低级的程序设计语言有下列的特点和优势：

易学和易用：大多数脚本语言的对初学者来说，它本身需要开发者的技术背景要求不是很高，这样一来，能够很快的找到合适的开发者。

快速开发：脚本语言的原则归纳起来就是让“开发、部署、测试和调试”的周期过程变得很短，很简洁。

动态的代码产生：脚本语言所产生的临时代码能够被及时的执行，这是一个非常好的特性，在很多应用里面（例如 JavaScript 里的动态类型）是必需的。

易于部署：目前主流的脚本语言都可以被开发者随时随地的部署和分发，具有很好的可移植性，而且不需要那么多时间去编译和打包。

## 1.3 脚本语言在国内外研究概况

回顾最近几十年科技的发展和革新，计算机发展尤其比其他产业来得迅速的，很多的计算平台提供了数不胜数的系统编程和脚本语言。回顾在最开始的时候，第一个出现的脚本语言的功能虽然简单而且能完成的工作也比较有限，它是一种被称为作业的控制语言，这种语言在当时被大量的用于那个时代主流的操作系统中，它当时的任务就是把工作按照有限级的顺序排好。在最近的 20 年内，即二十世纪八十年代时 Linux 机器上，C 语言被贝尔实验室的开发者开发出来，它被用于 Unix 系统级别的编程，而 SH, CSH 等壳编程被用于脚本语言的方面的开发。到了二十世纪九十年代的时候，C/C++ 被用于系统编程，而 Visual Basic 等语言用于脚步的开发。在现在这个以网络为主流的时代中，大家所熟知的 Java 语言被用于系统编程或者是 WEB 应用程序的开发，而像 JavaScript, Perl 和 Tcl 等语言被用于脚本的实践。

从第一个脚本语言的出现到现在，脚本语言其实已经在我们的视野里面已经存在了相当长的时间，在最近几年中，有很多的要素的综合起来促使脚本语言的重要性提高了很多层次，并且得到大规模的应用，效果也非常的不错。笔者觉得，其中最重要的一个因素是由于基于网络的应用程序是在向胶着另外一个应用程序而前进的。这种变换可以体现的几个例子就是用户图形界面，Internet 和 COM 框架的出现。

图形用户界面的出现在人们的视野的时候，大概在二十世纪八十年代的早些时候，它来得直观，并且操作简单，所以在二十世纪八十年代后期得到很多公司的重视。在许多系统级的编程项目中，用于图形界面的编程人力物力占了整个项目的一半以上的比重。图形用户界面的编程的工作大多数是基于普通应用的，它的目的不

是创建新的特性，新的功能，它所要做的是把视觉上的一些操作和应用程序内部逻辑结构结合起来。一些流行的操作系统，例如大家熟知的 windows，这些操作系统里面都有很好的图形应用程序来展现屏幕的输出结果并隐藏其内部语言，从而设计者不得不编写代码，因此这样一来一切都变得麻烦起来，例如为图形接口来提供内部使用行为。目前来看，好的快速的开发图形界面环境大多数都是基于脚本语言，例如大家熟知的 Visual Basic, Hyperlard 和 Tcl/tk，我们可以想象，随着图形用户界面的普及和发展，脚本语言也必然会越来越流行。

Internet 的迅速的发展也使脚本语言变得非常流行。Internet 其实只是一种粘合的工具，它本身不用来创建任何新的计算和特性，它只是普通的把大量的目前有的事物连在一起。因特网编程工作的具体的任务之一是把这些组件捆绑在一起工作，这样脚本语言可以非常容易的得到应用。例如：Perl 编写的 CGI 脚本非常的流行，JavaScript 编写的网页非常的炫丽，从而也非常的流行。

脚本语言得到普及的另一个原因是脚本技术本身得到很大程度上的提高。像 Python 和 Perl 等脚本语言与 JCL 最开始发布的时候已经有了长足的进步和发展了。例如，微软公司的 Visual Basic 其实不是真正意义上的脚本语言：刚开始的时候，它就是一个功能比较简单的系统或者称为高级程序设计语言，经过历史的演变，微软公司对它进行了非常多的修改，才成为我们今天看到的这个样子。可以预测到，以后新的脚本语言一定会将比现在的更好用。

我们不能忽视的另外一个因素是，计算机硬件的高速发展也让脚本语言运行起来更快。在过去，用于系统程序设计语言在繁琐的应用程序中得到可接受时间的执行。而在某些极端的情况下，系统程序设计语言似乎也不够高效，因此，有时不得不用汇编语言来编程。然而相比之下，今天的机器比 80 年代的机器快几百倍，同时仍以每 18 个月翻一倍的速度迅速增长。这样一来，机器本身就可以弥补脚本语言效率低下的不足，非常多应用程序可以用来解释后再执行，同时保持着出色的执行结果。例如，大家看到的 Tcl 脚本可以运行非常多的实例并且提供极佳的系统响应速度。我们可以预测到，由于计算机硬件的不断提高，脚本技术将会越来越多，越来越广泛的被应用于各种领域。



## 1.4 论文的主要工作

本文主要从以下六个方面进行论述的：

第一章：简单介绍了本课题的研究背景、研究意义。

第二章：对 UEFI 编程环境及技术做一定的介绍。

第三章：对 UEFI 脚本语言做详细的分析与设计。

第四章：对 UEFI 脚本语言的解释器做详细的分析与设计。

第五章：对 UEFI 脚本系统做一定性能测试。

第六章：总结与展望，对全篇论文做了总结，探讨了以后的努力方向。

## 2 UEFI 编程环境的介绍

UEFI 在 NT32 环境模拟运行的界面效果图如图 2-1 所示。

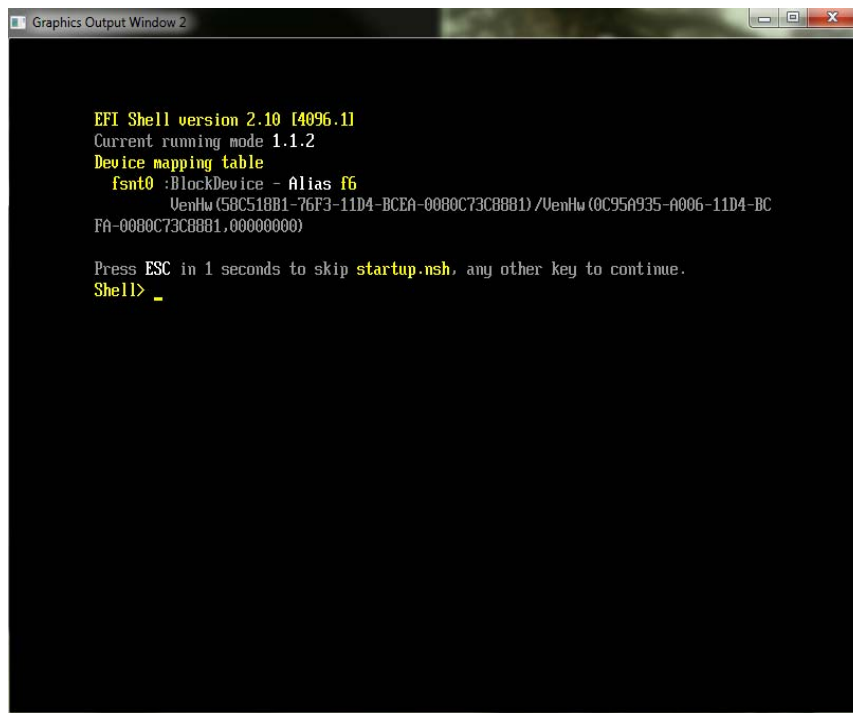


图 2-1 解释器运行环境的效果图

要开发一个 UEFI 脚本语言，必不可少的一个前提条件就是，我们得为其实现一个属于它的脚本解释器，那么脚本解释器的实现就离不开 UEFI 自身的特殊的编程环境，它跟我们平常接触的 windows 或者 Linux 大不一样。这一章大致的讲解几个特别重要的概念，它们对于熟悉和掌握 UEFI 编程是非常关键和必要的。接下来就是分别介绍这些非常重要的 UEFI 知识点，它们大致包括以下一些部分：

- (1) EFI System Table;
- (2) Handle database;
- (3) Protocols;
- (4) UEFI images;
- (5) Events and Task Priority Levels;
- (6) UEFI Driver Model。

## 2.1 EFI System Table

EFI System Table 是 UEFI 结构中最重要数据。从这个数据结构，一个 UEFI 的可执行映像可以获取系统配置信息和一些丰富的系统服务，这些 UEFI 服务包括下列项目：

- (1) EFI Boot Services;
- (2) EFI Runtime Services;
- (3) Protocol services。

EFI Boot Services 和 EFI Runtime Services 分别通过 EFI Boot Service Table 和 EFI Runtime Service Table 来获得的，这两个 Table 都是 EFI System Table 其中的两个数据域。EFI System Table 服务的数目和类型根据每个 UEFI Specification 版本不同而可能不同。EFI Boot Service 和 EFI Runtime Service 在 UEFI2.0 Specification 中定义。

Protocol services 是一组相关的函数指针和数据域，并且以 GUID 命名的。Protocol services 被应用于提供一些 devices 的软件抽象<sup>[4]</sup>，例如 consoles, disks, networks 等等。它们可以被用来为特定平台去扩展一些常见的 services。Protocol 是最基本的使 UEFI 固件保持可扩展性的模块。在 UEFI2.0 中，它定义了超过 30 种不同的 protocols，各种 UEFI 的实现可能会额外的去添加一些其它的 protocols 去扩展它的功能。

它们的关系如图 2-2 所示。

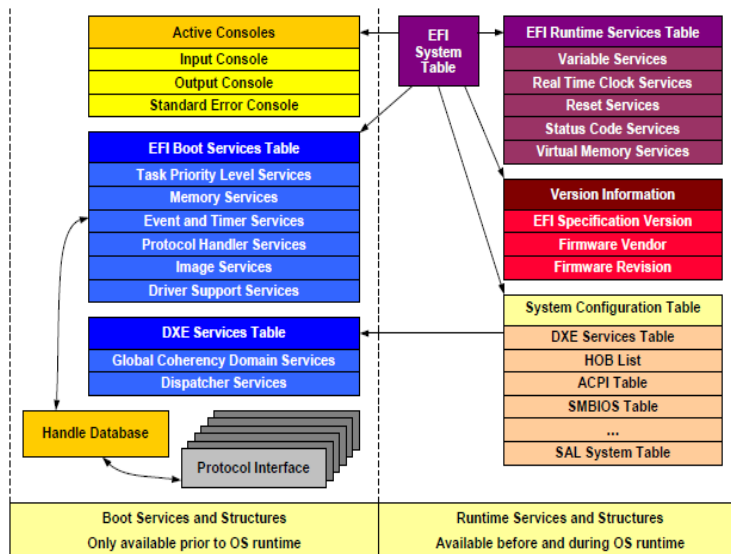


图 2-2 EFI System Table 示例图

它的代码结构如下：

```
typedef struct {
EFI_TABLE_HEADER                Hdr;
CHAR16 *                        FirmwareVendor;
UINT32                          FirmwareRevision;
EFI_HANDLE                      ConsoleInHandle;
EFI_SIMPLE_TEXT_INPUT_PROTOCOL *ConIn;
EFI_HANDLE                      ConsoleOutHandle;
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *ConOut;
EFI_HANDLE                      StandardErrorHandle;
EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;
EFI_RUNTIME_SERVICES *          RuntimeServices;
EFI_BOOT_SERVICES*              BootServices;
UINTN                           NumberOfTableEntries;
EFI_CONFIGURATION_TABLE *       ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

各个域的介绍如表 2-1 所示。

表 2-1 域的介绍

Member	Description
Hdr	它是 EFI System Table 的表头，其中它包括了 EFI_SYSTEM_TABLE_SIGNATURE 和 EFI_SYSTEM_TABLE_REVISION 和一个 EFI_SYSTEM_TABLE 大小的值以及一个 32 位 CRC 校正码
FirmwareVendor	指向一个包含 vendor 的字符串
FirmwareRevision	定义该 firmware 的版本号
ConsoleInHandle	激活状态的输入设备的句柄
ConIn	指向 SIMPLE_INPUT_PROTOCOL 接口的指针
ConsoleOutHandle	激活状态的输出设备的句柄
ConOut	指向 SIMPLE_TEXT_OUTPUT_PROTOCOL 接口的指针
StandardErrorHandle	激活状态的错误输出设备的句柄
StdErr	指向 SIMPLE_TEXT_OUTPUT_PROTOCOL 接口的指针
RuntimeServices	指向 RuntimeServices Table 的指针
BootServices	指向 BootServices Table 的指针
NumberOfTableEntries	内存中可控 Table 的数目
ConfigurationTable	指向可控 Table 的指针

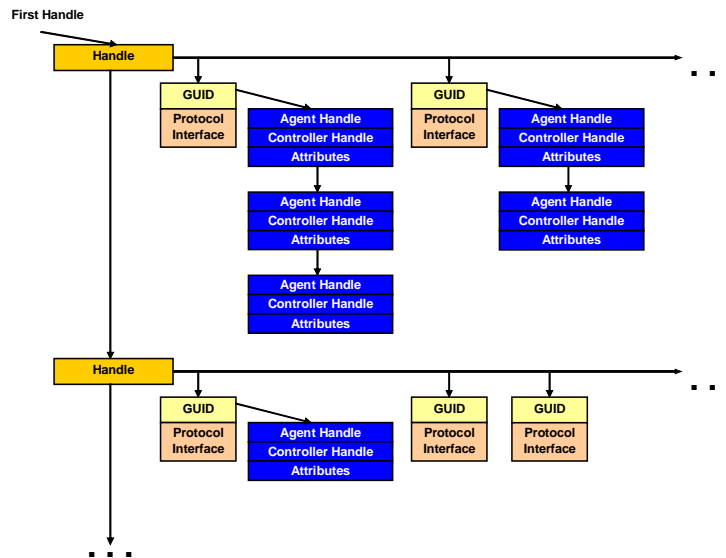
## 2.2 Handle Database

Handle Database 是由一些句柄<sup>[5]</sup>和协议组成的，句柄则是由一个或多个协议的集合，协议则是由一个 GUID 命名的数据结构，一个协议可能是空的，可能包含数据字段可能包含服务字段，或可能包含服务和数据这两种字段。在平台初始化，系统固件的 UEFI 标准驱动程序和应用程序将创建 UEFI 的句柄和将一个或多个协议附加到句柄中去。在 Handle Database 里面的信息是全局的，可以被任何可执行 UEFI image 进行访问。

在 Handle Database 是 UEFI 固件维持对象的中央储存库。在 Handle Database 是由一些 UEFI Handles 组成，每个 UEFI Handle 都有一个 GUID 区分，由系统固件进行维护。句柄的数字提供了一个访问 Handle Database 条目的一个“钥匙”。Handle Database 每个条目是一个或多个协议的集合。该协议的类型，由 GUID 的命名，是被附加到 UEFI Handle 中去决定该句柄类型。UEFI Handle 可能是如下组件：

- (1) 可执行映像，例如 UEFI 的驱动程序<sup>[6]</sup>和 UEFI 的应用程序；
- (2) 设备，如网络控制器和硬盘驱动器分区；
- (3) UEFI 的服务（驱动），如 EFI 压缩器和 EBC 的解释器。

图 2-3 展示了一个 Handle Database 的模型，除了句柄和协议以外，一组对象是与每个协议有关联。这个清单是用来跟踪哪些客户端去“消费”这些协议。此信息对于 UEFI 的驱动程序的操作来说是至关重要的，因为这些信息是让 UEFI 的驱动程序被安全地装载，启动，停止，然后没有任何资源冲突。



## 2.3 Protocols

可扩展性是 UEFI 自身所固有的，在很大程度上，它的这种特性是取决于 Protocols。UEFI 的驱动程序有时候会和 UEFI 的 Protocols 混淆。虽然他们是息息相关的，他们是截然不同的。一个 UEFI 的驱动程序是一个可执行的并且安装好的 UEFI 的映像，它安装了很多种类的 Protocols，以完成其工作。

UEFI 的 Protocols 或是一个组函数指针和数据成员组成的，或者是一些 API，但它们是由规范定义的。至少，该规范将为一个 Protocols 定义一个 GUID。这个数字是该 Protocol 的真实姓名，将被用于检索 Handle Database。该 Protocol 还包括一组典型的程序或者数据结构<sup>[7]</sup>（称为协议接口结构）。以下是从第一个支持 UEFI 2.0 规格的 10.6 章的协议定义的例子。请注意，它定义了两个函数定义和一个数据字段

GUID

```
#define EFI_COMPONENT_NAME_PROTOCOL_GUID \
{ 0x107a772c,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,0xd7,0x3f,0xc1,0x4d }

Protocol Interface Structure

typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
    EFI_COMPONENT_NAME_GET_DRIVER_NAME      GetDriverName;
    EFI_COMPONENT_NAME_GET_CONTROLLER_NAME  GetControllerName;
    CHAR8                                     *SupportedLanguages;
}
```

图 2-4 显示了单独一个 handle 和 Protocol，它们只是 Handle Database 里面的一部分，并且它们是由 UEFI 的驱动程序产生的。该协议是由一个 GUID 和协议接口结构组成。很多时候，UEFI 的驱动程序，将产生一个额外的字段保持私有数据。该协议的界面结构本身只包含指向该协议的功能。该协议的功能其实是包含在 UEFI 的驱动程序。一个 UEFI 的驱动程序可能会产生一个协议或根据驱动程序的复杂性产生许多协议。

并非所有的协议都在 UEFI 2.0 中定义。在 EFI 开发工具包里面包括了一些协议，是不是支持 UEFI 2.0 规范的一部分。这些协议是必要的，以提供一个特定实现的功能，但他们不是在支持 UEFI 2.0 规范定义的，因为它们不是支持引导操作系统或写一 UEFI 的驱动程序的必须。

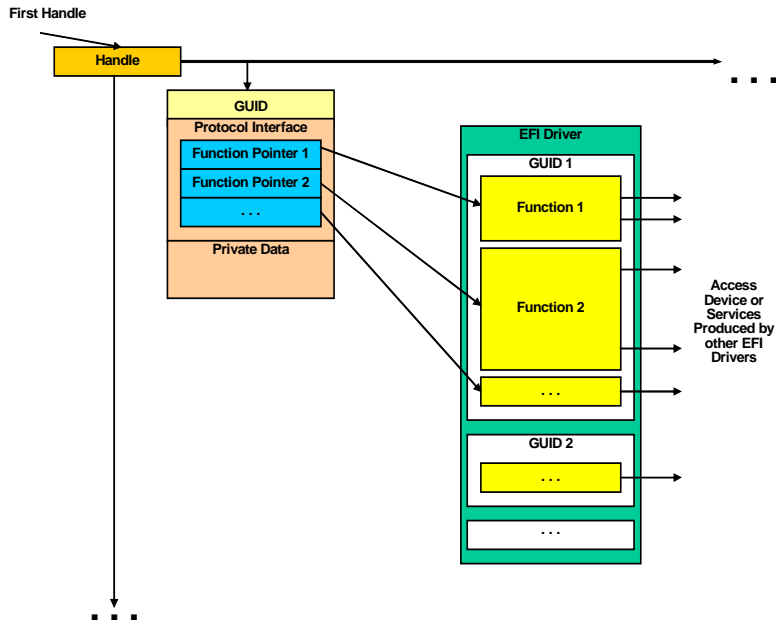


图 2-4 Protocol 组成 示例图

UEFI 的可扩展性允许为平台设计添加自己的特殊协议。这些协议可以用来扩大这种需求，并提供对 UEFI 的一致性访问，提供设备和接口一致性。

任何 UEFI 的驱动程序可以在启动时间来操作这些协议。但是，一旦 `ExitBootServices()` 被调用后，Handle Database 将不再可用。

一个 Handle 可能有多个 Protocols 被附属。但是，它们只能是同类型的协议。换言之，一个单一的 Handle 可能不会产生一个以上的任何一个协议的实例。这可以防止 Handle 有关的实例将被一个给定的请求消耗的恶意性行为。

## 2.4 UEFI images

本节将描述了 UEFI 不同类型的映像。所有的 UEFI 映像包含一个 PE / COFF<sup>[8]</sup> 标头，定义了其可执行代码格式。该代码可以在 IA - 32 的 IA - 64，EBC 的，或 x64 机器上运行。表头将确定处理器类型和映像类型。UEFI 2.0 定义了处理器的类型和以下三个映像类型：

- (1) EFI 应用程序；
- (2) EFI 启动服务的驱动程序；
- (3) EFI 运行时驱动。

UEFI 的映像的加载和装载到内存中去,是伴随启动服务中的 `gBS->LoadImage()` 一起的,包括以下几种支持的存储位置:

- (1) 在 PCI<sup>[9]</sup>扩展卡;
- (2) 系统光盘或系统闪存;
- (3) 如硬盘,软盘媒体设备,光盘,或 DVD;
- (4) 局域网服务器。

一般来说,UEFI 的映像不是被编译到一个特定的地址。相反,UEFI 的映像可以被放置在 UEFI 系统内存中的任何地方。启动服务 `gBS->LoadImage()` 执行以下操作:

- (1) 在内存中为 UEFI 映像分配内存;
- (2) 生成适用于自动迁移内存映象;
- (3) 创建一个新的 `Handle Database`,并安装一个 `EFI_LOADED_IMAGE_PROTOCOL` 的实例。

这个 `EFI_LOADED_IMAGE_PROTOCOL` 的实例包含有关 UEFI 已加载的信息。由于这些信息是在 `Handle Database` 中导出,它是提供给所有 UEFI 的组成部分。

在 UEFI 的映像被 `gBS->LoadImage()` 加载后,它可以 `gBS> StartImage()` 调用。UEFI 的映像头包中包含的入口点是 `gBS> StartImage()` 调用的地址。该调用点总是收到以下两个参数:

- (1) 映像的句柄;
- (2) 一个指向 `EFI System Table` 的指针。

这两个参数分别用来做以下事情:

- (1) 访问在这个平台上的所有的 UEFI 服务;
- (2) 检索 UEFI 的映像信息并加载并在内存中。

## 2.5 Events and Task Priority Levels

事件<sup>[10]</sup>是另一种被 UEFI 服务操作的对象。它们可以被创建和销毁,它们要么在等待的状态或有信号状态。一个 UEFI 的映像可以做下列任何一个事情:



- (1) 创建一个事件；
- (2) 销毁一个事件；
- (3) 检查是否在某个事件是否有信号；
- (4) 等待一个事件发生信号；
- (5) 要求一个事件从等待状态转移到有信号状态。

由于 UEFI 不支持中断，这就有可能对于那些习惯中断的程序员来说是一种挑战。取而代之的是，UEFI 支持轮询的驱动程序。UEFI 的驱动程序最常见的使用是使用一个计时器去定期轮询设备有没有加载。

以下三个因素都与每一个事件相关联：

- (1) 事件优先等级；
- (2) 事件的通知；
- (3) 事件通知的背景。

事件的通知函数在事件状态被检查或当事件被再次等待的时候唤醒。事件通知背景被传递到通知函数中去，并且被执行。TPL 是通知函数被执行的优先等级，一般有 4 个优先等级，它们如表 2-2 所示。

表 2-2 EFI 中定义的事物优先级表

Task Priority Level	Description
TPL_APPLICATION	UEFI 映象执行优先级
TPL_CALLBACK	回调等级
TPL_NOTIFY	I/O 操作的优先级
TPL_HIGH_LEVEL	中断等级

## 2.6 UEFI Driver Model

UEFI 2.0 规范定义了 UEFI 的驱动程序的模型。在 UEFI 驱动程序模型后面的驱动程序于 UEFI 应用程序共享同样的特性。然而，该模型可以让 UEFI 驱动控制更多的驱动程序。表 2-3 列出了 UEFI 的驱动程序模型相关 Protocols，它用于实现这种装载和运行的分离。

表 2-3 装载和启动/停止驱动程序的 Protocols

Protocol	Description
Driver Binding Protocol	提供的函数的启动和停止驱动,以及一个如何能处理某一特定的控制器。特定的 UEFI 驱动模型需要该协议。
Component Name Protocol	以条件来检索某个驱动的功能,形成易读的驱动控制器的管理。在 UEFI 2.0 规格中该协议为选择性的。
Driver Diagnostics Protocol	执行诊断功能,在 UEFI 2.0 规则中这个协议选择性的。
Driver Configuration Protocol	提让用户配置的功能,以至驱动被管理。它也允许一个设备有默认的配置。在 UEFI 2.0 中这个协议是选择性的。

新的 Protocol 是注册在驱动程序中 Handle 里面。在 UEFI 的驱动程序模型中,该驱动程序的入口时候的主要是安装协议并保证成功退出。在系统初始化后,UEFI 能用这些协议来操作这些驱动程序。

在 UEFI 的驱动程序模型定义了两个基本的 UEFI 启动时驱动程序类型

- (1) 设备驱动程序;
- (2) 总线驱动程序。

一个驱动程序,同时具备设备驱动程序和总线驱动的特点,就是一种混合驱动。

## 2.7 本章小结

本章主要是阐述了 UEFI 开发环境的一个框架,它主要包括 6 个重要的组成部分。

它们分别是 EFI System Table, Handle database, Protocols, UEFI images, Events and Task Priority Levels, UEFI Driver Model, 本章分别介绍了它们各自的特性,基本构成和用途。这些都是 UEFI 可执行程序的最基本的数据结构或者编程概念,熟练掌握它们对我们编写 UEFI 脚本解释器起决定性的作用。

### 3 UEFI 脚本语言的分析与设计

在介绍了 UEFI 编程环境后,本章将重点介绍 UEFI 脚本语言的分析 and 详细设计。

#### 3.1 编程语言特性分析

##### 3.1.1 编程语言的特点

任何一种程序设计语言都有三个特点:具有完善的语法和规则、健全的词汇表、解释词汇表的规范。

这些规范基本要涵盖以下几个部分:

- (1) 数据和数据结构;
- (2) 指令及流程控制;
- (3) 引用机制和重用;
- (4) 设计哲学。

很多被广泛使用或者是经得起考验的语言,它们很多共同的特点,例如:拥有负责标准化的组织,经常被修改来满足该语言的正式的规范,并集中讨论扩展或贯彻现有已有的定义。

数据和数据结构

当今的计算机系统内部储存数据都是以两种元的形式来实现的,即开-关模式(on-off)。现实世界中表示信息的各种度量单位,像手机号、名字、度量以及类似的较低端的二元数据,它们都经由编程人员设计好,整理好,把它抽象成为一种在计算机中表示高级含义的概念。

在一个程序当中,有被用来针对处理信息的系统,我们将这种系统叫做编程语言的型态系统<sup>[11]</sup> (type system);对于型态系统方面的一些讨论和发展趋势被称为型态系统的理论(type theory)。编程语言可以被人为的分做类似静态的型态系统(statically typed systems),例如 C++和 Java,和动态型态系统<sup>[11]</sup> (dynamically typed systems),例如 Python, Visual basic, Perl 和 Prolog。前面的一些,它们也可以分为包含了声明

和定义(manifest type)<sup>[11]</sup>的语言，意思就是每一个变量和函数的定义都要非常清楚地被声明，不存在有第二种解释它的方法，同样它们有时被称为 type-inferred 语言。

非常多的编程语言可以在自身的声明的形态的前提上表达出更加有用的数据结构形式例如：数组，列表，堆栈，文件等等。于此同时，面向对象语言<sup>[12]</sup>(Object Oriented Language，又译作“对象导向语言”)，这种新兴于二十世纪后期的一种程序设计语言，它可以让编程人员自己去定义想要的的形式，就是我们通常说的“对象”(objects)，和属于该对象自己的一些函数(functions)或者称方法(methods)。

除了怎样去断定表达式和型态，这两者之间的种种联系，这就衍生了一个非常重要的问题就是编程语言自己如何定义了这些型态，它们让哪些型态充当表达式的值。举个例子就是：像 C 语言这样的比较低级的语言，它们甚至允许编程者自己定义内存区域和编译期的常量；ANSI C 甚至能让表达式返回自己定义过的一种数据结构的值(struct values)。功能性的编程语言常常可以让变量自己去获得使用运行时得出的计算结果，而不是获得该值的可能存在的内存地址。

## 指令及流程控制

一旦数据被确定下来的话，那么计算机要被指导去如何对这些相应的数据进行一些处理，这里比较简单的一些指令可以使用关键字或者是定义好的语法结构来完成，不同的程序设计语言利用一种串行关系来获取这些语句。于此同时，在一个编程语言中也有类似的语句达到控制和判读的功能，例如：分支、循环等等。

## 引用机制和重用

引用的核心思想就是必须有一种非常非直接的重定义储存空间的技巧，最常见的一种方法就是通过对变量进行命名。依据语言的不同，从而引用能包括它指向另外的储存地址的指针值，还有一种与它相同的方法就是对于一组关键词进行重新命名。大多数编程语言使用了宏变量、引用或函数，类等等方法。它们的本质就是，就是让替换后的名字可以使应用程序变得更加方便，并有极好的可重用性。

### 3.1.2 编程语言的分类

编程语言按照语言的等级，我们将它可以分为低级语言和高级语言。低级语言又可以被分为机器语言<sup>[13]</sup>和汇编语言。

## (1) 机器语言

计算机可执行程序是直接能够被计算机识别的二进制代码指令进行读写的，像这样的程序设计语言，我们称之为机器语言。机器语言通常是由计算机一次完成一个最基本的操作指令来达到目的。机器语言是直接跟计算机硬打交道的，因此不同 CPU 型号的其他硬件的计算机，它们所采用的机器语言是不一样的，在这种情况下，要让机器语言得到很多的推广和被人所接受是比较困难的。因此，在大多数情况下，一般只有很少的一部分计算机专家或专业性比较强的工作人员才会使用这种机器语言。举个例子，典型的机器语言程序如下：

1010 1111

0011 0111

0111 0110...等等几百个指令。

机器语言本身，它是一种特定计算机的自然语言，它和计算机的硬件的关系非常大。机器语言只能在能读懂它的机器上运行。用机器语言进行编写程序，就是一个从所有的 CPU 的指令系统中挑到合适的计算机指令，来组成一个个指令系列的一种原始的过程。这种程序即便能被机器容易的理解和执行，但是相对于我们人类本身来说，他们晦涩，不容易记住、非常难被理解、容易出错，通常专业人员才能掌握它的特点，同时，程序员的写这样的程序，生产效率非常的低下，其质量非常不容易得到保证。这种原始的手工方式与当今高速、自动工作的计算机非常的不相称。

## (2) 汇编语言

汇编语言：用指令助记符、符号地址、标号等符号书写程序的语言。

它的出现，是为了克服上面所讲的机器语言，那些不容易读、不容易编、不容易记和容易出错的缺点，程序员用与代码指令含义非常近似的英文单词缩写、字母或者数字等等符号来代替，机器指令代码，例如：用 SUB 表示运算符“-”的对应的机器代码，这样就对应的产生了汇编语言。

所以说，汇编语言可以被解释为用一种比较容易记住的符号表示的针对机器的低级的编程语言，汇编语言亦称符号语言。由于汇编语言是利用了帮助记忆的符号名称来书写程序，比用最底层的二进制的机器语言来编程要容易些，在很大的程度上缩短了编程的人力物力。汇编语言是用助记符代替了二进制代码的机器语言，而且

汇编语言的助记符与二进制指令代码一一对应，完全保持了二进制语言的简洁性。用汇编语言开发程序相对于低级的机器语言来说，它的优点就是对计算机有较好控制，而且也比较容易写出比较高质量高的程序。

由于在汇编语言中引入了助记符，所以当把汇编语言直接编写的程序输入到计算机内，计算机不能以运行二进制机器代码那样来接识别并且执行它，为了能够解释它，我们必须用汇编编译器对汇编的代码进行翻译组织，把它翻译成二进制代码的时候，它才能被计算机运行。以一种可读，具有汇编语法规则书写的文本文件称为汇编源程序，在编译期时要将汇编源程序翻译为计算机目标程序。目标程序一旦被操作系统装载到内存里面去，它就能被计算机顺序的执行。

汇编语言在宏观上如同二进制指令，是一组操作硬件的指令，所以它是面向机器的过程语言，使用起来还是不是很方便和容易，可移植性不好，这是由汇编语言是一种低级的语言形式决定。但是，汇编语言用编写出来的软件和系统，其经过汇编编译器生成的目标文件占用资源少，CPU 指令是没有多余的，这些特点都是高级编程望尘莫及的。

### (3) 高级语言

至此为止，不管是二进制的机器代码还是汇编语言，它们都是针对特定计算机的硬件的直接控制的，所以它们过于依赖机器的硬件，使用汇编的人必须对特定的计算机硬件结构及其工作方式达到非常熟悉的地步，这样一来，对非计算机专业人员是不太可能做到的，因此，这样的程序设计语言，对于程序语言的推广和发展是非常的不利。随着计算机快速的发展，促使人们寻找那些与人类语言非相近的，同时可以被计算机接受的，语法必须是无二义性的、规则要通俗的、比较直观的和容易接受的程序设计语言。这种与人类自然语言非常近似的，且可以被计算机接受的，和执行的计算机程序语言被我们称为高级语言。高级语言是针对开发者的比较容易的一种语言形式。无论是什么类型的计算机，它们只要有其针对的高级语言的编译或解释程序，用这种高级语言得出的可执行程序就可以被他们执行。到了现在，被大家所熟知的高级编程语言有 C++、Java、C#以及 Visual Basic、Visual C++等。这些程序设计语言都是属于系统的软件的一部分。

计算机不能不经过转化而直接去运行高级语言书写的源程序，我们见到的源程序必须在被计算机读懂之前，要通过编译程序去把高级语言转化为能被计算机硬件识别的目标文件，既我们常说的二进制目标文件。像这样的“翻译”通常有两种形式，即编译后运行和边解释边运行。

编译形式其实就是：把一个称为编译器的程序，作为用户一个开发软件安装在计算机操作系统里面，当用户把由高级语言编写的源程序导入到计算机以后，这个编译程序便把源程序编译为能被该计算机识别的二进制目标代码，最后，计算机再去执行该目标程序文件，最后得到结果。

相比之下，解释方式是：当源程序导入到计算机里面的时候，解释程序一边扫描源程序，同时边解释，它会一个个的输入，然后逐句的翻译，计算机同时去执行，于编译形式最大的一个区别就是，解释程序一般不会有目标代码。**Java** 等高级程序设计语言执行编译方式，而像 **Python** 这样的语言是通过解释方式来被系统执行的，像 **C++**、**C** 等编程语言是能编写编译程序的高级编程语言。每一种高级的编程语言，它们都有自身的，事先定义的一些符号、英文词语、语义规范、语句结构的书写格式。高级程序设计语言与人类能看懂的自然交流的语言（这里我们指的是英语）非常的相似，很重要的一个方面就是，它们是与硬件不相关的，或者是隔离的，我们称之为脱离了具体的计算系统，非常容易被广大程序员能掌握。高级编程语言的实用性很不错，互容性也非常好，同时具有很好的可移植性。

现在，世界上每天都有非常多的高级语言的产生。这些高级程序设计语言都有一个目的，那就是让程序员用非常容易能够被理解的方法来编写计算机程序，而且又要满足足够精确而且简单的特点，让计算机能够识别。

高级语言的特点有很多特点，其中很重要的三部分如下：

简洁性：高级语言的每一个语句所能表达的意思都类似于低级语言或者机器语言的非常多语句才表达的，当然，程序员现在只需花费更少的时间，去完成更多的工作；

可移植性：高级语言是与机器的体系结构无关，在一台计算机上运行的可执行的程序只要稍微做点修改或根本不做任何修改，就可以在另一台计算机上来执行。

程序员因此没有必要花费大量时间去了解使用计算机的内部体系结构，而是将精力放在程序算法的设计和实现上，编写出更多的软件。

易读性：这对于需要维护和改进的程序是十分重要，高级程序设计语言可读性再一步一步的增强，我们可以在日新月异变化的编程语言中看到。

### 3.1.3 现在流行的高级编程语言特点分析

到目前为止，可以大致将现在流行高级编程语言特性分为几类：

#### 1) 强类型、弱类型

强类型语言与弱类型语言之间比较明显。只要有隐式的类型转换的语言则是弱类型的，例如 C 语言能将 `char` 隐式转换为 `int`。不存在隐式转换的语言也是存在的，像 Haskell(一种函数式编程的语言)。在 Haskell 里面，我们不能定义一个 `Double` 类型的名字，但是可以以整数作为名字，这是因为整数跟实数的类型是不同的，而且不存在隐式转换。

#### 2) 过程式和面向对象式

过程式设计特点：

- (1) 自上而下(top-down)的设计方式：是一个自顶向下，逐步来求的一种过程；
- (2) 以 `main` 函数作为整个程序的中心，可以将 `main` 函数分割成非常很小的模块，每个模块可以一系列的子函数的调用组成；
- (3) 其主要的一个特征是以函数为单元来组织程序，程序中的数据就是从函数来的。
- (4) `Main` 里面的每一个子函数都可以被提炼成规模更小的子函数。重复这个过程，那么就可完成一个过程式的设计；

过程式设计的优点：它能够容易被开发者学习和掌握，比较和人们直观的的思维相似；

同时，过程式设计也有以下几个缺点：

- (1) 数据是与逻辑分离开，对数据与操作的修改是很困难的；
- (2) 不能适应问题比较多，或者需求经常变化的情况；
- (3) 程序部分之间的依赖性太强了：`Main` 函数要依赖它的子模块，子模块又依赖于更小的子函数；



面向对象设计特点：面向对象编程以对象为中心，是对一系列相关对象的操纵，发送消息给对象，由对象执行相应的操作并返回结果，强调的是对象。

### 3) 函数与闭包<sup>[15]</sup>

凡是包含闭包这种形态的编程语言那么注定它是有函数这个功能的，目前为止但是并不是所有的具有函数功能编程语言都有闭包，同样并不是所有的编程语言都有函数。例如 Windows 的批处理文件就是不对函数支持的脚本语言。

那么闭包定义是什么呢？闭包就是可以使函数能够被上下文执行的一种指针。举个例子：

```
function Add(a)
{
  return function(b)
  {
    return a+b;
  }
}
Inc=Add(1);
Inc10=Add(10);
print(Inc(5));
print(Inc10(5))
```

6 和 15 将作为这个例子的结果，当程序执行到 `Inc=Add(1)` 这句的时候，`Add` 函数返回了一个新的函数，这个函数接受参数 `b` 并返回参数 `a` 和 `b` 相加的结果，函数返回的时候，它将参数 `a` 存储下来，当程序到 `Inc` 和 `Inc10` 的时候，虽然执行的是同一个函数，但是这个函数所看到的 `a` 确是不同的。`a` 的值的不同代表着 `Inc` 和 `Inc10` 执行函数的不同，这样的结果是由于闭包的功能，它是可以指向函数的上下文。当然，一个不支持闭包功能的编程语言中，无法满足的这种写法的。

### 4) 命令式与描述式

一个程序设计语言是命令的方式还是描述的方式是由这门语言是告诉计算机怎样做以及它要做什么的。例如，`SQL`<sup>[16]</sup>和 `Prolog`<sup>[17]</sup>是描述式语言，而 `C++`、`C#`等则是命令式语言。我们在使用 `SQL` 的时候告诉服务器的是我们需要满足什么条件的数

据项，而不是告诉服务器我们需要通过什么计算来获得自己所需要的数据项。描述式的语言的优点在于其可读性好。C# 3.0 为数据查询加入了 LINQ<sup>[18]</sup>让我们可以在 C# 中书写类似 SQL 的代码查询数据。

## 3.2 UEFI 脚本语言的总体设计

### 3.2.1 变量

因为大多数情况下，脚本语言处理的工作并不像编译语言那么复杂，为了满足日常的工作所需并且达到简洁简单，UEFI 脚本语言只需要两种变量就可以了：第一种数值类型，第二种字符类型。UEFI 脚本语言允许你在程序中任何地方使用一个变量。例如：

```
a="b";  
b=12;  
c=a;
```

等等这些都是合法的，上面的 `a` 是一个字符类型的变量，`b` 是一个数值类型的变量，而且变量之间可以互相赋值例如上面的 `c=a`，因为 UEFI 脚本语言是弱类型的语言，我们可以为同一个变量分次赋值不同的类型，例如 `a=10;a="string"`，这样也是合法，并且以最后一次赋值为该变量的最后结果。

变量不需要事先声明，如上面的例子所示，任何地方出现一个合法的标识符时，就意味着 UEFI 脚本内部会增加这个变量，并给予初值。这里的变量也没有静态类型，也不会固定为某一类型，同样它也不会像 C++ 中分常量和非常量之分。我们不用去关心如何提高程序的效率为去考虑每个变量实际所占的内存大小，因为 UEFI 脚本语言不是从这个出发点设计的，所以这样以来就带来了编程的灵活性和简单性。正如现在大多数脚本语言一样，在 UEFI 脚本里，你无法获知一个变量的类型，事实上也没这个必要。

### 3.2.2 作用域

UEFI 脚本语言中只有两个作用域：全局的和局部的。位于自定义函数块外且在 `main` 函数中的变量处于全局作用域；自定义函数内的变量处于局部作用域，与 C 语

言一样，位于函数块内的代码块变量，还是处于局部作用域。

当局部作用域内出现一个全局里的同名变量时，优先取局部作用域里的变量。这同 C 语言一样。并且我们规定变量只能处于 `main` 函数中或者自定义的函数中。当然这样做是力图简洁的目的，因为 UEFI 脚本语言暂时只支持单个文件的解释，我们没有必要把变量放在函数体之外，这也就不存在内部文件和外部文件的区别了。

### 3.2.3 运算符

UEFI 支持的算术运算符如表 3-1 所示。

表 3.1 算术运算符表

运算符	解释	结合方式
!	取否	由右向左
*    /    %	乘，除，取模	由左向右
+       -	加，减	由左向右
==     !=	等于，不等于	由左向右
&&	逻辑与	由左向右
	逻辑或	由左向右

这些运算符的用法大都和 c 语言差不多，大多都是算术运算符针对数值运算的，其中值得注意的是+运算符，假设有以下例子：

```
a=10;
b="string";
c=a+b;
```

这里的 c 就是字符类型了为“string10”，这里默认会有个转换，就是假设数值类型和字符类型相加，得到的结果就是字符类型。

### 3.2.4 控制语句

简单起见，UEFI 控制语句主要有两种 `if` 和 `while`。

(1) (1) `if` 语句

`if` 的语法同 C 语言一样，如：

```
if( a > 0 ) {
.....
```

```
    }  
else{  
.....  
}
```

if(  $a > 0$  )中的  $a > 0$  为条件语句，所有条件语句，都不能为字符串类型，所以以下  
if(“a”) dosomething();

这样就会报一个解释错误，但是我们可以用数值类型作为条件判断语句，例如：

```
b=0;  
if(b+1)  
dosomething();  
else  
{  
.....  
}
```

这样 dosomething()是可以被执行的，而 if(0)这样的是不会被执行的。

## (2) while 语句

```
while(  $a > 0$  )  
{  
a=a-1;  
}
```

While 语句用来控制一个循环的，这也是编程语言必须要满足的一种特性，同样 while 中的条件判断语句的变量不能为字符串类型，当然我们这里不支持 do{}while() 等用法之外，其它的用法和 c 语言一样。

### 3.2.5 函数

由于之前我们简化了变量的类型定义的步骤，无需定义就可以为一个变量进行操作，这样就带来了一个问题，那就是没法区别函数名和变量，为了解决这个问题，我们把函数的定义加上了关键字 function，这样一来就可以区别变量名和函数名了，假设我们不加 function 的话，有下面一个例子：

```
func( a ){ ... }
```

```
func=1;
```

我们就没办法去区别它们了，然后便有了以下定义函数的方法了：

```
function func( a, b )  
{  
}  
}
```

UEFI 中的函数可以没有返回，可以有返回，如果有返回的话，我们没有强制规定它到底返回什么样的类型，就如同变量一致，你可以这次返回数值类型，下次返回字符类型，这都没有关系，重要的是调用者要知道自己怎么调用就行了，下面这个例子充分说明了这个问题：

```
function main()  
{  
  a=Test();  
  b=add(1,2);  
}  
function Test()  
{  
  return "c";  
}  
function add( i,j)  
{  
  return i+j;  
}
```

如你所见，UEFI 函数块不需要声明，也不需要去定义参数类型，它只是充当一个代码重用的作用，非常简单易学。

### 3.2.6 数组

UEFI 脚本语言中的数组是以 `dim` 关键词声明的，例如：

```
dim ary[10];
```

这样就是一个数组了，我们对数组里面的类型不强作要求，这就意味着你可以这样写：

```
ary[0]=10;ary[1]="c";
```

这样都是允许的，值得注意的是操作时尽量避免越界的情况的发生和数组类型的变量是要预先声明的，而且不能动态扩张。

### 3.2.7 输入

输入只要有 `input()` 这个函数去完成的，它的用法：

```
a=input("%d");
```

```
b=input("%s");
```

我们就可以分别得到一个数值类型和字符类型的值了，非常的简单和易用。

### 3.2.8 输出

输出就更简单了，它是由 `print()` 这个函数去完成的，去输出一个变量，你只要这样做：

```
print("d");
```

```
a=10;
```

```
print(a+20);
```

```
print("str"+"ing");
```

这样就可以输出你想输出任何变量了。

## 3.3 几个完整的 UEFI 脚本程序

### 3.3.1 计算阶乘的例子：

这个例子主要是计算  $n!$  的一个程序，当然这里是一个递归的实现，可以看出 UEFI 对递归的支持。

```
function main()
{
n = input( "%d" );
print( "fac(" + n + ") = " + fac( n ) );
}
function fac( n )
{
if( n == 1 )
```

```
{  
    return 1;  
}  
else  
{  
    return fac( n - 1 ) * n;  
}  
}
```

### 3.3.2 一个基本包含完整 UEFI 脚本语法的例子

在一下这个例子中，可以看出它基本的涵盖上 UEFI 的基本的语法规则，包括数组，数字变量，字符变量，判断语句，控制语句，参数传递，输入，输出，函数块以及递归函数等。

```
function main(){  
    dim ary[10];  
    n=10;  
    if(n){  
        while( n ){  
            ary[n-1]=n;  
            test(func(ary[n-1]));  
            n=n-1;  
        }  
    }  
}  
  
function test(n)  
{  
    print("the value is"+n+"\n");  
}  
  
function func(n)  
{  
    return n;  
}
```

}

## 3.4 本章小结

本章主要对 UEFI 脚本语言做一个语法层面的上的全面的设计。

首先分析编程语言的一些特性，然后对当前流行的主流编程语言和脚本语言进行了详细的剖析，得出各自的优缺点，然后针对 UEFI 的特定编程环境去设计了 UEFI 脚本语言的语法规则，最后给出了几个完整的 UEFI 脚本源程序，全面的展示了它自己区别于其它脚本语言的特点。



## 4 UEFI 脚本解释器的设计和实现

本章主要介绍 UEFI 解释器用到的一些原理和技术，以及其具体实现方案。

### 4.1 总体设计

UEFI 脚本解释器工作流程图，接下来我们就介绍每个流程具体的实现和用到的原理和方法。如图 4-1 所示。

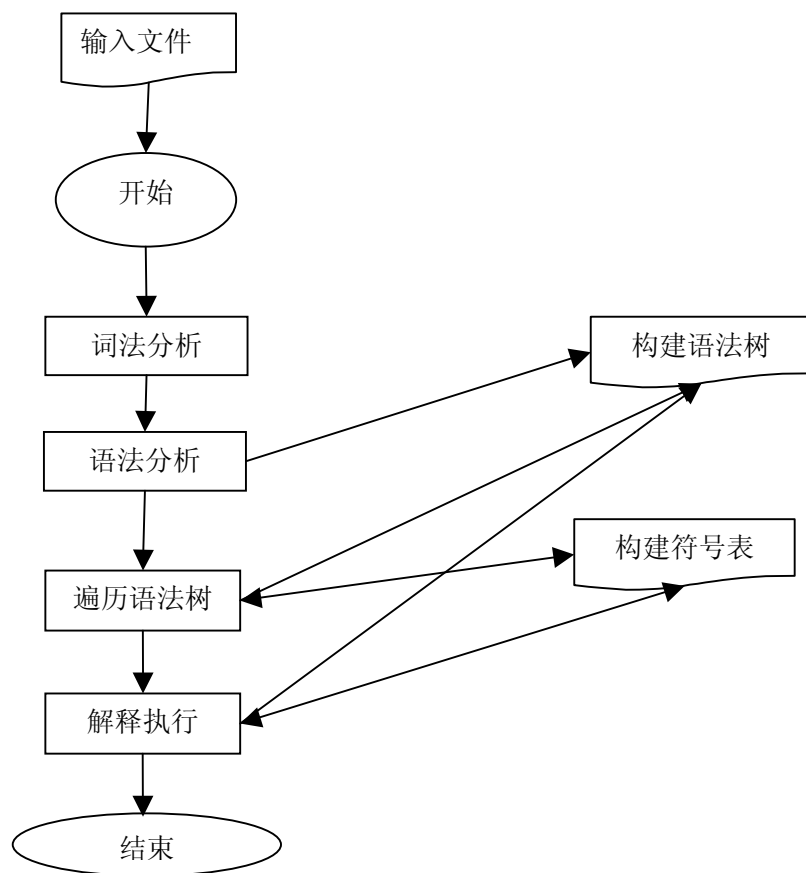


图 4-1 UEFI 脚本解释器工作流程图

### 4.2 语法分析的实现

#### 4.2.1 词法分析的概念

词法分析<sup>[19]</sup>的主要任务是对源程序进行扫描，从中识别出单词，它是编译过程的第一步，也是编译过程中不可缺少的部分。

## 4.2.2 UEFI 词法分析器

词法分析器是负责对源程序进行扫描，从中识别出一个个的单词符号，它从左至右逐个字符地扫描源程序形式的字符流，将这些单个字符组合成一个个单词符号，把作为字符串的源程序转换成由单词符号串组成的中间语言程序供语法分析使用。

词法分析阶段的必要性：

1) 描述单词结构的语法比其它语法结构简单得多，仅用 3 型文法就足够，把单词的识别从整个语法识别中划分出来，可以使我们采用更有效的方法和工具，如状态转移图、有穷自动机等，同时还可利用这些工具建立词法分析程序的自动生成器。

2) 对不设关键字的语言如某些非标准 Fortran<sup>[20]</sup>，其中某些标识符的识别需超前扫描，分析上下文才能准确识别，将这种特殊地方分离出来，有利于保证语法分析方式的一致性。

3) 词法分析与语法分析<sup>[21]</sup>分离，可使整个编译程序的各部分功能更加单一，编译程序结构更加清晰，有利于编译程序的维护。

UEFI 脚本系统把词法分析程序作为独立的一个模块来实现，并将转换后的内部形式的源程序传递给语法分析程序。

单词符号是语言的基本符号，它具有独立的意义且是不可再分的。UEFI 脚本语言包括以下几种类型的单词符号：

- (1) 标识符。用以表示各种名字，如变量名，函数名等等。
- (2) 保留字。如 if, else, while, return 等等。
- (3) 常数。125, 3.8, 0, 1 等等；
- (4) 运算符。如 +, -, \*, /, <, = 等等。
- (5) 分界符。如分号和括号等等。

所有的单词符号如下所示：

```
enum
{
TK_ERROR = -1,
TK_EOF,
TK_ID, TK_NUM, TK_FLOAT, TK_CHAR, TK_STRING,
```

```
TK_IF, TK_ELSE, TK_WHILE, TK_DO, TK_RETURN, TK_BREAK,
TK_FUNCTION, TK_ARRAY,
TK_NE, TK_EQ, TK_LE, TK_GE, TK_OR, TK_AND
};
```

我们将每个单词符号定义为一个数据结构如下所示，`type` 就是保留字里面的或者上用户自定义的变量，否则它便是一个非法的单词符号，解析就会报出错误：

```
struct Token
{
int type;
char *string;
};
```

具体的保留字，存放在一个全局的变量里面，如下：

```
const struct Token reserved_words[] = {
{ TK_IF, "if" },
{ TK_ELSE, "else" },
{ TK_WHILE, "while" },
{ TK_DO, "do" },
{ TK_RETURN, "return" },
{ TK_BREAK, "break" },
{ TK_FUNCTION, "function" },
{ TK_ARRAY, "dim" },
{ 0, NULL }
};
```

扫描时候的状态图如图 4-2 所示。它的过程为：

- ① 从初态 0 开始，若输入符号是一字母，则读进它，并转到状态 1；在状态 1 下，若下一个输入符号是字母或数字则读进它，并重新进入状态 1；
- ② 重复这个过程，直至在状态 1 下发现读入的符号不再是字母或数字时(注意，此时该字符已被读出)，就进入状态 2。
- ③ 状态 2 是终态，至此已识别出一个标识符，识别过程终止。若在状态 0 下输入符号不是字母，则意味着识别不出所给的输入串是一个标识符。

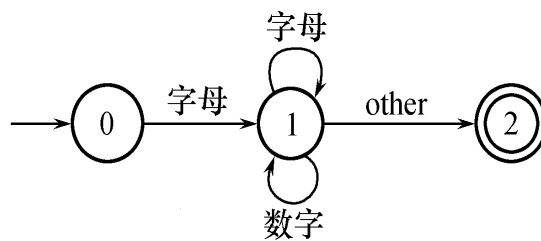


图 4-2 标识符状态转换图

UEFI 词法分析框架图如图 4-3 所示。首先从源文件把脚本文件读到内存中去，然后逐步扫描每个单词，依据图 4-2 所示的状态转移<sup>[22]</sup>图，来一个个的确定标识符是正确的还是错误的。例如假设现在有个表达式为

`a=a+1;`

经过我们的词法分析，它就被拆分成了 `a`、`=`、`a`、`+`、`1` 这 5 个 Token（单词符号）。

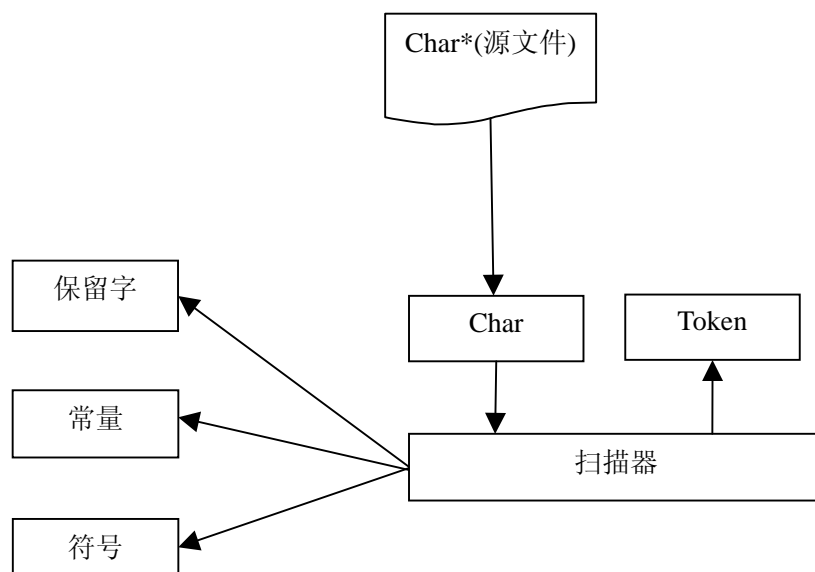


图 4-3 UEFI 词法分析框架图

假设现在有个源文件为如下：

```
function fac(n){
while(n){
n=n-1;
}
i=10;
}
```

经过 UEFI 词法分析器词法分析之后的单词字符，如图 4-4 所示。

```
Function
ID : fac
<
ID : n
>
<
While
<
ID : n
>
<
ID : n
=
ID : n
-
Number : 1
;
}
ID : i
=
Number : 10
;
}
```

图 4-4 词法分析效果图

## 4.3 语法分析的实现

### 4.3.1 语法分析的概念

在计算机科学和程序设计语言学中,语法分析(parsing)是根据某种给定的形式文法<sup>[22]</sup> (formal grammar)对输入的单词(token)序列进行分析并确定其语法结构的一种过程。而语法分析器通常是以编译器或解释器的组件出现的,它的作用是从输入中分析出其结构并将其转换为在后续处理过程中更易于访问的数据结构(一般是树类的数据结构),并检测可能存在的语法错误。语法分析器通常使用一个词法分析器(lexer)从输入的字符流中分离出一个个的‘单词’,并将单词流作为其输入。

程序设计语言构造的文法可使用上下文无关文法或BNF<sup>[23]</sup>表示法来描述,它的特点如下:

- (1) 文法可给出精确易懂的语法规则;
- (2) 可以自动构造出某些类型的文法的语法分析器;
- (3) 文法指出了语言的结构,有助于进一步的语义处理/代码生成;
- (4) 支持语言的演化和迭代<sup>[24]</sup>。

语法分析的作用:

- (1) 从词法分析器获得词法单元的序列,验证该序列可以由语言的文法生成;
- (2) 对于语法错误的程序,报告错误信息;

(3) 对于语法正确的程序，生成语法分析树<sup>[25]</sup>。

语法分析器的流程图如图4-5所示。

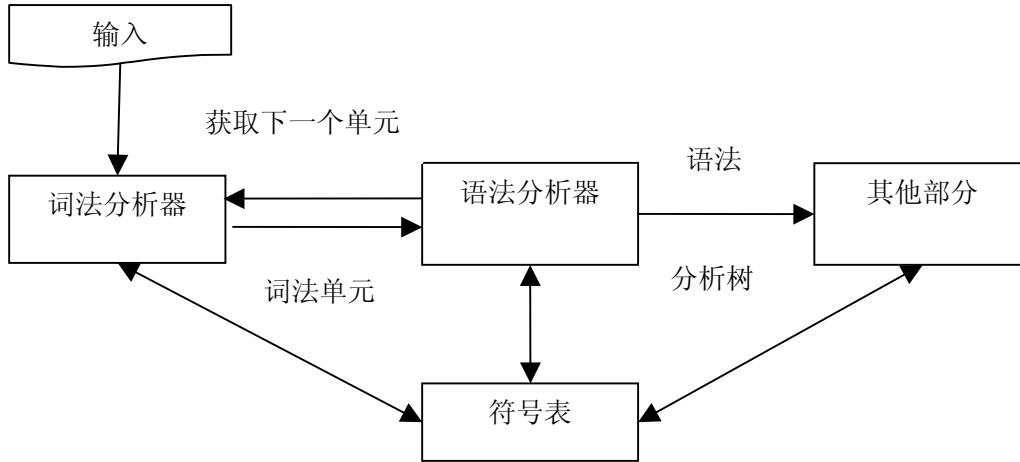


图 4-5 语法分析器流程图

### 4.3.2 UEFI 语法分析器

#### (1) BNF

语法分析中非常重要的一个概念就是 **BNF**，巴科斯范式，完整名称为巴科斯-诺尔范式（也称为巴科斯-瑙尔范式、巴克斯-诺尔范式），即 **BNF**，是一种用于表示上下文无关文法的语言，上下文无关文法描述了一类形式语言。它是由约翰·巴科斯（John Backus）和彼得·诺尔（Peter Naur）首先引入的用来描述计算机语言语法的符号集。

**BNF** 规定是推导规则（产生式）的集合，写为：

<符号> ::= <使用符号的表达式>

这里的 <符号> 是非终结符，而表达式由一个符号序列，或用指示选择的竖杠 '|' 分隔的多个符号序列构成，每个符号序列整体都是左端的符号的一种可能的替代，从未在左端出现的符号叫做终结符。

UEFI 脚本语言采用的 **BNF** 定义如下：

```
iteration_stat  : 'while' '(' exp ')' stat
stat           : labeled_stat
               | exp_stat
```

---

---

```
                | 'if' '(' exp ')' stat 'else' stat
function_stat   :id '(' exp ')' '{'   exp 'stat'
return_stat     : 'return' | exp
```

## (2) 语法树形结构

UEFI 脚本系统的语法树的节点结构采用以下定义的形式:

```
typedef struct Node{
    Type type;
    Bool isStat;
    union {
        int operator;
        union{
            double dval;
            char *sval;
        } val;
    } value;
    Node *child[MAX];
    Node *next;
}Node;
```

首先 `type` 表示该节点是什么类型; `isStat` 表示该节点是申明还是调用; `value` 这个联合指的是要么这个节点是个操作符, 要么它是个字符或者数值变量; `child` 指向它的子节点, `next` 指向于这个节点平行的下个节点的地址。

## (3) 整理出语法树

有了上述的一些工作之后, 接下来, 要做的就是整理出语法树了, 现在举个例子, 现在假设有个 BNF 如下, 它虽然是一个不是完整的 BNF 的例子, 但是足够可以说明的我们的解释器如何根据上面的定义, 来推到正确的语法树:

```
stat -> stat op item | item
op -> + | -
item -> item mul fact | fact
mul -> *
fact -> (stat) | number
```



很容易，我们从它所表达的意思，写成相应的递归<sup>[26]</sup>和非递归的解析函数，其中“|”表达式“或”的关系，其它的每个符号我们可以在它后面找到其定义，于是可以写出以下解析它的函数：

```
stat ()
{
    if( ... ) { left = stat ();}
    right = item();
    return left op right;
}
item ()
{
    if( ... ) left = item()
    right = fact();
    return left mul right;
}
fact()
{
    if( ... ) return stat();
    else return number;
}
```

这个不是很难理解的，可以看 stat 中，函数第一个判断语句便是 BNF 中“|”对应得前一部分的值，假如 stat 存在的话，它就是调用自己的一个递归函数，如果不存在的话，我们直接返回它的后半部分。同理，另外几个函数可以照推。其实 BNF 核心就是一个状态的转换的过程，但是正确的 BNF 文法是有穷的，否则我们将会无穷的在递归或者不能正确的推导，也就不能正确表达编程语言语法规则了。

#### (4) UEFI 语法树结构示例

假设现有一个脚本文件的内容如下，那么 UEFI 脚本解释器如何来解释它的了？其实按照上面的分析，我们基本上可以知道结果了。

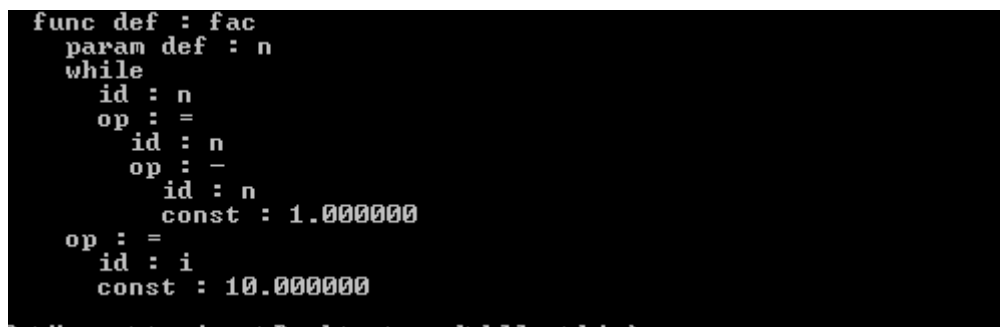
```
function fac(n){
```

```
while(n){  
  n=n-1;  
}  
i=10;  
}
```

UEFI 的脚本解释器的解释的过程是自上而下的一个递归的过程，基本的原理就是上面所阐述的，首先是词法的分拆，然后语法的组装，最后是符号表的查找，经过 UEFI 脚本解释器解释后，它所形成的语法树为：

```
func def : fac  
  param def : n  
  while  
    id : n  
    op : =  
    id : n  
    op : -  
    id : n  
    const : 1.000000  
    op : =  
    id : i  
    const : 10.000000
```

如图 4-6 所示，这样我们可以清楚的看到如我们前面构想的一样，UEFI 的语法树非常简洁和直观。



```
func def : fac  
  param def : n  
  while  
    id : n  
    op : =  
    id : n  
    op : -  
    id : n  
    const : 1.000000  
    op : =  
    id : i  
    const : 10.000000
```

图 4-6 语法树

## 4.4 解释器符号表的实现

在 UEFI 脚本系统中，符号表是用来存放源程序中出现的有关名字的属性信息，这些信息集中反映了名字的语义<sup>[27]</sup>特征属性。符号表在解释过程的地位和作用非常重要，是进行上下文合法性检查和语义处理及代码生成的依据。符号表总体结构的设计和实现是与源语言的复杂性（包括词法结构、语法结构的复杂性）有关，还与对于解释系统在时间效率[28]和空间效率[28]方面的要求有关。

### 4.4.1 符号的类型

UEFI 脚本系统中只有两种符号类型，函数和变量，它们都具有数据类型属性。对函数的数据类型指的是该函数值的数据类型。用以下列数据结构表示 UEFI 符号：

```
typedef struct Symbol
{
    char *name;
    Value val;
    Symbol *next;
} Symbol;
```

Name 是该符号的名称；val 是该符号的值，这里有两种可能：一种就是本事的值，再者就是函数的指针；next 就是链接下一个符号的指针。

### 4.4.2 符号的存储类别

因为 UEFI 中的符号就 3 种，一是字符或者数值变量；二是函数名；三是数组名，为此就有了一下定义：

```
typedef struct Value{
    union{
        char *cval;
        double nval;
        void *addr;
        struct {
            Value *ptr;
        }
    };
};
```

```
size_t size;
};
};
Type type;
};
```

用一个联合<sup>[29]</sup>统一去表示着三种变量的时候，可以节省程序的内存空间,为了区分这符号到底是哪种变量，用 `type` 去区分。这个结构体里面 `cval` 是代表字符串变量指针值，`nval` 是数值类型的值，`addr` 是用户自定义函数的地址，里面另外一个结构体是用来保存数组相关信息。

#### 4.4.3 符号的作用域及可视性

一个符号变量在程序中起作用的范围，称谓它的作用域<sup>[30]</sup>。一般来说，定义该符号的位置及存储类关键字决定了该符号的作用域。UEFI 脚本系统有两个符号表：一个全局的符号表和一个局部的符号表，每当解释器检测到程序调用一个函数时，就会创建一个临时的局部符号表，用于存储局部变量，当函数退出了之后，就把这个符号表销毁掉。

#### 4.4.4 符号表的查找算法

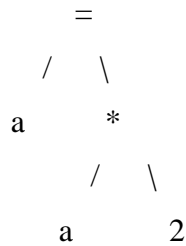
我们用一个线性表来存储 UEFI 中的符号表，对应采用查找的算法<sup>[31]</sup>是 HASH 算法，其中 HASH<sup>[32]</sup>函数如下：

```
static int hash( const char *key )
{
    int value = 0;
    for(int i = 0; key[i] != '\0'; ++ i )
        value = ((value << 4) + key[i] ) % LEN;
    return value;
},
```

解决冲突的方式是用链地址法<sup>[33]</sup>，将所用 `hash` 到同一个位置的符号用链表串联起来，查找的时候只有遍历它一次就行了。

## 4.5 翻译语法树的实现

经过前面这些步骤后，展现在解释器前就是一个语法树和符号表了，接下来要做的就是如何正确去翻译这颗语法树了，现在假设有个语法树如下：



UEFI 脚本程序解释的原理大致是这样的：

```
Value calculate(Node *pRoot){
return pRoot->child[0].value=exp(pRoot->child[1]);
}
Value exp(Node *node){
Switch(node->operator){
case '+':{
value left=fact(node->child[0]);
value right=fact(node->child[1]);
return left+right;
}
.....
}
}
Value fact(node* node){
Switch (node->type){
case ID:{
/*在符号表去查找该符号的值；*/
return Lookin_SysTable(id);
}
case CONT: {
return node->value;
```

```
}  
case OP:{  
    return  node->child[0].value=exp(pRoot->child[1]);  
}  
Case ....{  
}  
}  
}
```

其实这个原理不难理解，当解释到某个节点时，首先去探测它为什么类型，如果为操作符，就肯定一点它的第一个孩子<sup>[34]</sup>必然为返回值，或者为空，当然为空的话，不用去执行它，因为它没有实在的意义。再去检测它的第二个孩子，如果发现它是数值或者字符类型我们直接返回，如果为表达式，我们再递归的解释。函数 `exp` 中探测它到底是个什么表达式，通过 `switch` 来枚举它的类型。对叶子节点<sup>[35]</sup>的解释就是通过 `fact` 函数来实现的，首先看这个节点的类型是什么样的，如果是常量的话，直接返回，如果符号类型，这个时候我们就要到符号表中去查找该符号的值，并且返回它。

解释器解释到一个函数调用节点时，会优先在符号表中查找该函数符号。如果找到，就将其值转换为约定的函数类型，然后整理参数调用之。这个时候代码执行权转接到局部函数<sup>[36]</sup>里面。如果没找到，就在全局符号表<sup>[37]</sup>里查找，找到后就强制转为语法树节点指针，并解释执行该节点下的语句。

## 4.6 本章小结

本章主要讲述了 UEFI 中最核心的部分：UEFI 解释器的设计原理。

首先给出了 UEFI 脚本解释器工作流程，然后将 UEFI 脚本解释器分为词法分析，语法分析，符号表，语法树，翻译语法<sup>[38]</sup>树几个部分，并逐个介绍了它们是如何单独工作，协同工作的。并且给出了相关的例子，以及各个解释阶段所产生的阶段性结果。

## 5 UEFI 脚本系统的测试

### 5.1 系统测试

#### (1) 递归函数测试

一个求迭代<sup>[39]</sup>的递归函数体的内容如图 5-1 所示。

```
U:\>type fac.uefi
function main<>
{
    n=input("<math>\%d</math>");
    print("fac<math>+n</math> + "<math>+n</math>") = "<math>+n</math> + fac<math>(n)</math> <math>+n</math>";
}

function fac<math>(n)</math>
{
    if<math>(n == 1)</math>
    {
        return 1;
    }
    else
    {
        return fac<math>(n-1)</math> * n ;
    }
}
}
```

图 5-1 递归函数体图

运算之后的结果图为图 5-2 所示。

```
U:\UEFI fac.uefi
UEFI interpreter
10
fac<math>(10.000000)</math> = 3628800.000000
U:\>
```

图 5-2 递归函数运算结果图

因为 UEFI 脚本的数值类型都是 double，所以对输出来看都是保留小数点后面 6 位有效值的。我们的输入时 10，它所产生的结果是 3628800.000000，由此可见 UEFI 对递归函数的解释没有问题，而且结果是正确的。

## (2) 对字符串的一些处理

对字符串进行处理的函数体如图 5-3 所示。

```
U:\>type char.uefi
function first( i )
<
    return i+"hello";
>

function second( j )
<
    return j+"world";
>

function main()
<
    reslut = first("i say ") +second "<" ">";
    print(reslut);
>
```

图 5-3 字符处理函数体图

运算之后的结果图为图 5-4 所示。

```
U:\> UEFI char.uefi
UEFI interpreter
i say hello world
U:\>
```

图 5-4 字符处理结果图

程序首先写了两个局部函数，他们分别来拼接各自的字符串，然后再 Main 函数里面，再将它们拼接起来，同样可以看到它得出了预期的结果。

## (3) 全局与局部变量的区分



```
U:\>type char.uefi
function first( i )
{
    return i+"hello";
}

function second( j )
{
    return j+"world";
}

function main()
{
    reslut = first("i say ") +second "<" ">";
    print(reslut);
}
```

图 5-5 局部变量函数体图

运算之后的结果图为图 5-6 所示。

```
U:\> UEFI local.uefi
UEFI interpreter
1.0000000
```

图 5-6 局部变量运算结果图

首先我们在局部函数里面定义了一个名为 `i` 的变量，同时我们在 `main` 函数中也去定义同样名字的变量，当我们去调用 `local` 函数的时候，我们希望它能把 `i` 定位到自己的作用域里面，而不去影响全局变量。从结果上来看，我们可以得出 UEFI 脚本解释器对局部变量的区分也是正确无误的。

## 5.2 本章小结

本章主要对 UEFI 预期的功能进行了测试，并得到了预期的结果。

递归函数<sup>[40]</sup>测试的目的是测试 UEFI 脚本系统对递归这一重要的编程特性的支持性如何，从结果来看，达到预期的目标。

对字符的处理的测试的目标是看 UEFI 脚本系统对字符操作性如何，毕竟字符串的处理是日常开发非常重要的一个方面，从结果来看，基本达到预期的目标。

以及最后对变量定义域的测试结果也达到我们的要求。

## 6 总结与展望

### 6.1 全文总结

本论文是在研究现在流行的编程语言的基础之上，综合了编译原理<sup>[41]</sup>相关知识，把解释器带到了 UEFI 的环境中去，并不拘泥于常规解释器<sup>[42]</sup>的实现，量体裁衣，UEFI 脚本系统把简单和易用的特点放在首位，其设计也是围绕这两个特点进行的。因为本身 UEFI 系统的复杂性把用户入门的门槛提高到了一定的高度。为了降低整个高度和难度，UEFI 脚本系统的任务就是降低用户对该系统的依赖，在用户和系统之间架起一层中间件，将复杂而又繁琐的编程细节减少到最少，展现给用户的是一个透明而又易用的编程环境。

因为 UEFI 脚本系统有以上等特性，所以它能极大的缩减开发者开发周期，节省了相当多的精力，这样必然会推动 UEFI 的迅速发展。同样可以说它可以引起一场 UEFI 编程的革命，让单调的 UEFI 有了新的活力。

本文本着开发一种适合 UEFI 编程环境的脚本语言为出发点，首先阐述了 UEFI 编程环境的重点和要点，因为它区别于传统的操作系统<sup>[43]</sup>，大部分读者可能对 UEFI 比较陌生，所以本文开始就着重挑选了若干重要的 UEFI 编程概念，分别介绍了它们各自的特点和用途。

接着便是 UEFI 脚本语言的详细设计。为了能够设计好的脚本系统，本文就对编程语言的要素和核心进行了剖析，总结出若干重要的结论，这对我们设计编程语言来说是非常重要和必要的。凡事总是要理论和实践结合，前面我们总结了理论知识，接下来就是要结合实践了。本文就从当前一些流行的编程语言和脚本语言的特点进行了归类，发现它们各自的长处和短处，各自适用的范围，总结经验，为了 UEFI 脚本语言设计打下基础。第三章的最后便是 UEFI 脚本语言的详细设计和介绍了，可以看出它的构成非常的简洁，这样做有两个好处，第一个，降低开发难度，第二个使用户学起来更快。在第三章的末尾便是几个完整的 UEFI 脚本的实例了，与 c 语言有几分相似，但是又具有它自己的特点。

在第四章，就是 UEFI 脚本解释器的设计和实现了。总体上它分为：词法分析，语法分析，符号表，语法树几个大块。词法分析的作用是将源文件的各个单词有序的收集起来，为接下来的工作做好准备。语法分析是以 BNF 为基础的，而 BNF 的表达式又是根据本身语法规则来定制的。为了完成 BNF，我们将第三章制定的语法规则翻译成了 BNF 表达式，有了 BNF，必须有相应的数据结构作为支撑，然后本文就介绍了 UEFI 语法树的数据结构，并逐一对它的每个域进行了介绍，而且给出具体例子来展现 UEFI 语法树到底是什么样的。语法分析的时候，其中有个很重要的结构是符号表，它是用来存储用户自定义的变量的地方，当然符号表有两个，一个是局部的，一个是全局的，这样的做可以区分局部变量和全局变量。UEFI 符号表查找算法(Search algorithm)<sup>[44]</sup>是经典的 HASH 算法，很多编译器都采用了类似的算法去查找符号，它的特点就是非常的快，这也正是一个好的解释器必须满足的条件之一。

有了以上的结果后，展现在解释器面前就是一个完整的语法树，UEFI 解释器接下来要做的就是去遍历(traverse)<sup>[45]</sup>这颗树，也就是我们所说的开始解释源程序了。在第四章的末尾，本文对如何遍历这棵树，以及在遍历中碰到的种种问题都做了详细的介绍。最后还给出了一个源程序的经过语法分析得到的完整的语法树，这样就可以更加直观的了解 UEFI 语法树的全貌。

到现在为止，UEFI 系统基本上可以说完成了，最后本文的第五章对 UEFI 系统做了系统的测试，从测试的结果来看，没有发现任何问题，达到了我们预期的目标。相对来说，取得了不错的效果。

从目前取得的成果来看，UEFI 脚本系统是能够胜任日常工作的一些事务。而且鉴于它的语法非常的简洁而且易学，相信肯定会极大地减少 UEFI 开发者的工作量，进而提高他们的效率，从而推动 UEFI 的进一步的发展。

## 6.2 展望

从目前 UEFI 脚本系统的功能来看，虽然它能胜任基本的编程工作，但是相对于当前流行的脚本语言，它还有自己的局限性，主要有一下几个方面：

- (1) 暂不支持面向对象的编程。
- (2) 对第三方的库的支持的不够。
- (3) 不支持文件之间的引用。
- (4) 暂没有支持网络相关的功能。
- (5) 系统调用(system call)<sup>[46]</sup>相关的功能相对来说比较少。

为了 UEFI 脚本语言的长久的发展,实现这些功能是非常必要的,因为这些特性是脚本语言发展的趋势,它们可以使语言本身更加强大,所以在今后的 UEFI 完善的过程中,这 5 个方面就是主要的努力的方向了。

## 致 谢

时光飞逝，在此，向这几年来和我一起走过的，同时给我很多帮助的，关心的良师益友，致以最真诚的感谢！

首先，衷心的感谢我的导师肖来元老师，是您，在我工程实践的时候，一直教会我如何为论文的收集做各种必要的准备，如何把工程实践做好。在着手写论文的时候，肖老师还不断的指导论文写作时需要注意的重点要点，还给了我很多建设性的思路。在论文写作的过程中肖老师还反复的帮我修改，提出建议，尤其是论文最后的阶段，肖老师在百忙之中抽出时间帮我做了非常多的并且重要的修订。肖老师这种平易近人的品格给我留下了非常深刻的印象，从他身上我学到了崇高的敬业的精神和严谨的治学态度。在此论文完成之际，谨向我学业和人生的双重导师肖老师致以深深的谢意和诚挚的敬意！

感谢 Intel 的龚炯，给了我论文的最初的灵感和方向，感谢 Intel 的其它的同事，您们一起帮助我攻克了很多技术难关，使我在非常多的地方，各种能力都得到了较大的发展，是您们教会了我如何将理论和实践有效的结合起来，以至我的论文得以完成。

最后，要特别感谢我的父母。是您们任劳任怨的工作而供我上学、养育了我 25 年，在我失败的时候，在遇到人生很多困难的时候，正是您们给予了我义无反顾的奋斗下去的决心和信心。

最后感谢在百忙之中挤出时间来，参加我论文的评审和答辩工作的各位老师。

## 参考文献

- [1] Vincent Zimmer, Michael Rothman, Robert Hale. Beyond BIOS. Intel Press, 2006: 10-42
- [2] 刘劲鸥. BIOS 详解. 北京: 海洋出版社, 2003: 1-5
- [3] Paul Sheer. Linux. Prentice Hall Ptr, 2001(10): 228-230
- [4] 麦克布林. 软件工艺. 北京: 人民邮电出版社, 200: 80-83
- [5] J. B. Pontalis. Windows. Bison Books, 2003(4): 55-65
- [6] 魏永明, 耿岳, 钟书毅. Linu 设备驱动程序. 北京: 中国电力出版社, 2006: 34-45
- [7] 冯俊. 数据结构. 北京: 清华大学出版社, 2007: 14-18
- [8] John A. Camara. Electrical Engineering Reference Manual for the Power, Electrical and Electronics, and Computer PE Exams. Professional Pubns, 2008(10): 300-306
- [9] 马鸣锦. PCI、PCI-X 和 PCI Express 的原理及体系结构. 北京: 清华大学出版社, 2007: 59-60
- [10] Charles Petzold. Programming Windows. Bison Books, 2004(12): 48-50
- [11] 曾国屏. 系统科学哲学. 北京: 清华大学出版社, 1995: 47-49
- [12] Richard Johnsonbaugh, Martin Kalin. Object-Oriented Programming in C++. Prentice Hall Ptr, 2003(4): 34-41
- [13] Ronald Mak. Writing Compilers and Interpreters. Wiley, 1996(8): 18-20
- [14] 张海藩. 软件工程. 北京: 人民邮电出版社, 2003: 54-56
- [15] Geoffrey James. The Tao of Programming. Professional Pubns, 2006(1): 80-87
- [16] Colby Horner. Beginning SQL. Microsoft Press, 2007(11): 110-111
- [17] Colby Horner. Logic Programming With Prolog. Springer-Verlag New York Inc, 2008(1): 21-23
- [18] Jr., Joseph C. Rattz . Pro LINQ. Apress, 2007(11): 30-34
- [19] Christopher W. Fraser. A Retargetable C Compiler. Professional Pubns, 2003(2): 12-21

- [20] 彭国伦. Fortran 95 程序设计. 北京: 中国电力出版社, 2002: 69-72
- [21] 赵克佳. 高级编译器设计与实现. 北京: 机械工业出版社, 2005: 56-61
- [22] Andrew W. Appel. Modern Compiler Implementation in C. Apress, 2004(12): 21-25
- [23] 张幸儿. 编译程序构造实践. 北京: 科学出版社, 2005: 109-112
- [24] 高仲仪, 金茂忠. 编译原理及编译程序构造. 北京: 北京航空航天大学出版社, 2001: 90-97
- [25] 王海涛, 徐骁栋. 编译器设计. 北京: 清华大学出版社, 2009: 12-14
- [26] 杨萍. 编译器设计基础. 北京: 清华大学出版社, 2009: 58-63
- [27] Rudolf, ingenmann. Languages and Compilers for High Performance Computing. Springer, 2008: 78-81
- [28] Rastislav, Bodik. Compiler Construction. Springer, 2009: 131-132
- [29] 冯速. 编译器工程. 北京: 机械工业出版社 2006: 210-215
- [30] AlfredV. Aho, MonicaS. Lam. Compilers: Principles, Techniques, and Tools. Addison Wesley 2009(1): 43-47
- [31] 陈火旺. 程序设计语言: 编译原理. 第 3 版. 北京: 国防工业出版社 2006: 209-214
- [32] 王生原, 吕映芝, 张素琴. 编译原理课程辅导. 北京: 清华大学出版社 2007: 90-92
- [33] 张幸儿. 计算机编译原理. 第 3 版. 北京: 科学出版社, 2008: 10-30
- [34] 李文生. 编译原理与技术. 北京: 清华大学出版社, 2009: 1-7
- [35] Ronald Mak. Writing Compilers and Interpreters: A Software Engineering Approach. Wiley, 2009(9): 13-16
- [36] Charles N. Fischer, Richard J. LeBlanc. Crafting a Compiler with C. Addison Wesley, 1991(11): 13-16
- [37] Y. N. Srikant, Priti Shankar. The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition. CRC Press, 2007(12): 24-26
- [38] Steven S. Muchnick. Advanced Compiler Design & Implementation. Morgan Kaufmann, 2003(9): 37-42

- [39] Thomas Pittman, James Peters. The Art of Compiler Design: Theory and Practice. Prentice Hall, 1991(11): 107-114
- [40] 费希尔. 编译器构造 C 语言描述. 北京: 机械工业出版社, 2005(7): 208-210
- [41] Alexander Meduna, 杨萍, 王生原. 器设计基础. 北京: 清华大学出版社, 2009: 167-170
- [42] Fraser Christopher W, Hanson David R., 王挺, 黄春. 可变目标 C 编译器: 设计与实现. 北京: 电子工业出版社, 2005: 98-102
- [43] 布鲁姆. 汇编语言程序设计. 北京: 机械工业出版社, 2006: 103-109
- [44] Richard M. Stallman. Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4. 3. 3. CreateSpace, 2009(4): 341-346
- [45] Chuck Hellebuyck. Beginner's Guide To Embedded C Programming: Using The Pic Microcontroller And The Hitech Picc-Lite C Compiler. CreateSpace, 2008(4): 68-70
- [46] Derek Beng Kee Kiong. Compiler Technology: Tools, Translators and Language Implementation. Springer, 2000(4): 90-95