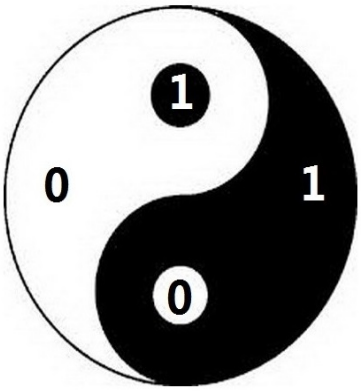


太极真经 ——

太极语言编程指南

曹星明 (太极真人)

太极语言的产生背景

太极语言是我长期自然语言编程偶然获得的一个意外收获。我从1995年开始研究自然语言编程，因此对于自然语言处理和计算机语言学的各种理论关注很多，特别是语法理论和解析技术。同时也对各种计算机语言，编译技术多有研究。尝试和研究过很多高级语言。特别是非常欣赏lisp的优雅和强大的功能。

从2007年开始，我尝试设计和实现解析器。试验了多种不同的解析技术实现方法。从2011年开始尝试实现解释器和编译器。期间完成了dao和daonode这两个项目，开源在Github。dao和daonode是一个逻辑和函数求解器和编译器，综合实现了lisp语言和prolog语言的特性，同时内置了解析器的功能。不过，dao和daonode没有实现具体语法，因此还不能称之为一门语言。

因为我在考虑语言实现的过程中，一直希望语言具有灵活的动态定制和扩展能力。rebol语言一定程度上具有这种能力，但是，还没有达到我期望的那种层次。而且，rebol作为类似lisp的一种方言，遵循了lisp的嵌套列表的语法风格，这也是我一直希望突破的一个方面。我考虑到，要实现这种完全能够动态定制和扩展语法的能力，就无法采用现有的各种解析器产生器，不管是基于LR文法的lex/yacc，bison，jison等工具，还是基于LL文法的Antlr, 以及基于PEG文法的各种解析器产生器。因此我一直在一种简明而又强大的的手写解析器实现方法。这个过程中，如何简明高效地解析左递归文法是个长期困扰我的难题，而左递归文法对于简明地表达运算符表达式是非常必要的，为此我曾经做过非常多的尝试。直到2013年8月，借助coffee-script语言特征的启发，我终于发现了这种满足要求的解析器设计方法，也解决了左递归的问题。由此得到的结果就是Peasy这个github项目。

自然语言编程作为一个困难的研究领域，我在开始的十多年时间里长期处于探索状态，没有很明显的收获，也没发现多少有价值的解决方案。2007年，我开始理解到互联网的力量，转而以互联网为基点思考自然语言处理及自然语言编程的问题，并逐级地有了自己的理解。2012年，我很偶然地获得了解决自然语言编程问题的整体图景。我开始借助各种web技术来实现这个系统的原型。2014年3月，在实现该原型的过程中，我又获得了一个关于web开发框架的思路，我认为，这个思路的实现有赖于某些新的语言特征。为此，我基于internet做了一些技术探索。最后，lispyscript这个github项目直接促发了我设计太极语言的灵感。在接下来的四个月里，我在原有的dao和peasy项目的经验基础上，重新设计和编码实现了太极语言，发布了最初的版本。

太极语言的源流

太极语言的产生可以追溯到lisp之根。太极语言从内核上继承了lisp的精神，甚至比lisp更为lisp。lisp将程序或者说代码看作数据，也就是列表，一种递归数据结构的方法，深刻地影响了我的思维。lisp的宏就是以程序作为数据，并以程序作为结果的构造。

语法上拥抱了主流高级语言的语法。特别是缩进语法。我最早接触的缩进语言是python，非常喜欢这种简洁明快的语法。后来，我更多地使用javascript设计web系统，这期间接触到了coffee-script，大大提高了我的开发效率。coffee-script的语法特征也启发了我设计了Peasy这一解析器系统。从产生Peasy的创意到实现过程我体会到语言和语法和思维的关联。因为正是coffee-script将赋值作为表达式，及其定义函数的语法使得用它手写语法规则特别直观可读，用它手写的解析器可以和专用于描述语法规则的dsl相媲美。从语法上，太极语言和coffee-script有很多相似之处。对此本书在附录中专门给出一节加以对比。

除了以上最重要的两点，太极语言也从很多高级语言吸取了营养。

我曾经探索过rebol语言，感觉它是一种精致的语言。特别是它的方言功能，利用parse函数设计出的GUI方言的表现力令我叹服。受其影响，我设计和实现太极语言时，最开始我不自觉地选择了[]作为代码块的定界符。直到发布前夕，在实现某个功能时，我发现使用[]有某些不便。然后我突然意识到，可以使用更为通用的{}作为代码块定界符。

我也曾经了解过metalua的元语言功能，思维中植入了元编译这个概念。

虽然太极语言借鉴了这么多语言，但是太极语言并不是一个杂乱揉和各种语言特征的混合体。相反，太极语言在吸收的上述每个显著特征都明显地超过了最初的本体，上升到了一个新的层次，同时，也形成了自己的整体风格和设计原则，具有独立的灵魂。

太极语言的特点

太极语言一门新的编程语言。太极语言比现存其它高级语言具有更强的扩展能力和定制能力。

总体上说：太极语言的特性可以总结为：比lisp更加推广的宏功能，比python的语法更重视空白，比C更通用的预处理，比metalua更强大的元编译，比rebol更灵活的方言能力，比coffee-script更优化的目标代码。

太极语言综合了三种创新：lisp的内核结合缩进风格的主流语法，可动态定制扩展的语言，元编译技术。太极语言破除了某些编程语言领域的流行观点和技术限制，是高级语言领域自1958年lisp发明以来的最重大进步。虽然太极语言还刚刚问世，但是我感觉它好象是打开了一扇门，展示出了广阔的发展空间和巨大的发展前景。它诞生于结合web开发自然语言编程原型系统过程中意外降临的偶然机遇，将来也会回馈和推动web开发和自然语言编程系统开发。

当前太极语言以javascript作为编译的目标语言，运行在node.js环境下。因此也可以把太极语言介绍为穿着lisp风格的衬衣，披着coffee-script风格外套的javascript代理人。

太极语言的相关资源

太极语言github项目：github.com/taijiweb/taijilang

发布在npm的包taiji：npmjs.org/package/taiji

google 邮件组：google groups: taijilang

google+帐号：taiji lang

QQ群: 太极语言 194928684

本书的组织

本书的目的

本书的目的是作为太极语言的一个太极语言的使用指南，特别是使用太极语言编程的指南。

本书的体系

本书主要分为两个部分：基础和高级。基础部分主要介绍太极语言作为编程语言的基本知识，特别是绝大多数高级语言所共通的知识。 高级部分介绍太极语言的高级功能。有很多功能是太极语言独有的创新。 附录介绍与太极语言相关的某些附近内容。

各章内容介绍

基础部分

第一章到第八章是本书的基础部分。包含了太极语言的介绍性知识、基本功能以及和大多数高级语言所具有的共通功能。

- 第一章 介绍：本章就是第一章，介绍太极语言的产生背景，技术上的源流，特别是与现有高级语言的关系，太极语言自身的特点，编写本书的目的，以及全书的内容组织。
- 第二章 太极语言概览：介绍太极语言的功能特性，和具有相似特征的高级语言的对比，介绍太极语言的编译过程和中间表示的基本特点，太极图解以太极图的方式阐释高级语言和程序发展过程中观念和技术和技术演进，太极之禅是试图太极语言设计哲学的释宗禅语。
- 第三章 起步：介绍太极语言的基本工具和入门知识。包括npm安装命令，命令行工具的使用，repl，和一些简单的示例。
- 第四章 数据和变量：介绍太极语言的数字（整数和浮点数）、列表和字典等基本数据类型的文字表示，太极语言的变量定义以及与作用域相关的规定。
- 第五章 基本命令和运算：介绍太极语言的基本命令，如注释、打印、单引表达式、求值等，也将介绍太极语言的算符表达式，并给出太极语言预定义的各类运算符及其优先级规则。
- 第六章 控制结构：介绍常见的控制结构在太极语言中的使用方法，包括块结构、条件执行、重复执行、异常等各类语句。
- 第七章 介绍太极语言的函数，包括其定义和调用方法；类的定义和使用，太极语言的模块化功能，包括include!和import！语句的使用。

高级部分

从第八章到第十一章是本书的高级部分，介绍了太极语言的高级内容，体现了太极语言的独特功能。

第八章 编译过程和中间表示： 介绍太极语言的编译过程，涉及到编译过程各个阶段及其操作，包括代码解析，元编译和元变换、表达式转换和代码产生等。同时也介绍了贯穿所有这些阶段的中间表示。这一章是理解和掌握后续各章的基础性内容。

第九章 语法解析：将介绍太极语言关于语法及解析方法的设计原则，实现过程，介绍太极语言的词法、主要语法成分、算符表达式的解析特点。

第十章 元编译和宏：介绍太极语言的元编译特性以及它的最重要特例——宏功能。这包括太极语言发现元编译技术以及它的设计和实现，相关的元算符，并揭示了元编译的若干用途。说明了包含和导入模块与元编译的关系。另外也介绍了元编译的一个最重要特例：宏功能，解释了太极语言中进行宏定义、宏调用和宏扩展的方法。

第十一章 定制语言：介绍了太极语言各种定制和扩展语言的方法。包括定制运算符，利用解析器算符扩展语法规则和控制解析，定制语句等，并解释了如何配合函数和宏来进行定制。

体例

[设计注记]

[术语]

[重要提示]

正名

必也正名乎

名可名，非常名。本书很多地方花费了不少文字介绍概念，谈到可能用到的术语和名词。由于历史的原因和交流的需要，同一个对象，可能会有不同的名词来称呼它。同一个名字，也有可能被用来指代不同的对象。请读者在阅读本书的时候，多注意这些正名的内容。既要注意名字本身，也要联系上下文以求得完整的理解。

特性和功能

太极语言的特性

太极语言具有丰富的和独特的特性。

- 统一的基于json的中间表示

从语法解析结果开始，太极语言编译过程每个阶段和步骤都采用json作为中间表示，其主体是嵌套列表（javascript数组）。这种设计方案使得简化了系统实现，并且令整个系统更易于理解。

- 缩进语法
- 带空格的算符具有更低优先级，多行算符表达式按从上到下逐行执行，更符合人眼直觉和更普遍的习惯

因为历史和技术的原因，越早发明的高级语言越多的考虑计算机的能力，随着计算机能力的提高，后来发明的语言开始更多地尊重人的习惯。因此，越早的语言语法要求越严格，往往忽视代码中的空格。现代语言开始出现利用换行、缩进，空白体现语法的特性，python是一个最流行的代表。然而，由于程序员已经习惯了以前的高级语言，有的人对于这些新的特性并不习惯。有很多人喜欢python的缩进语法，但是也有很多人因此而觉得难以接受。看来，太极语言的这个特征也不可避免地遇到质疑。但是，我相信从长远看，人更深层的直觉和更普遍的习惯会有利于这种特性。

- 一切都是表达式 类似于coffee-script，if语句，switch语句，循环语句，try-catch语句，throw语句等等都可以作为表达式。不同于coffee-script，即使break，continue和return语句也可以出现在任何允许其它语句的位置。
- 类似于coffee-script的各种语言特性
- 产生更优化的目标代码 在实现将一切语句转换为表达式中，太极语言采用了更优化的实现，没有采用封闭函数调用这一实现方法。同时，太极语言编译过程包含了编译优化这一步骤，尽量产生更优化的代码。
- 可扩展可定制的语言

太极语言的功能

基于上述特性，太极语言具备以下功能：

- 用更简洁的语言编写javascript程序
- 管理编译过程，根据环境和配置产生不同的目标代码
- 有助于编写更优化的代码
- 设计dsl
- 统一的基于json的中间表示将为扩展太极语言到新的领域提供便利。

和其它高级语言的比较

和**coffee-script**的比较

太极语言和coffee-script具有非常多的相似之处。最重要的相同点包括：都是运行在javascript语言之上，以node.js作为主要的运行环境，都编译成javascript的语言，都是缩进风格的语法。具体语法细节上，太极语言也大量借鉴了coffee-script，同时，太极语言的当前的实现代码依然基于coffee-script。

当然，太极语言和coffee-script也有着非常重要的不同。这主要体现在语言的扩展性和定制能力方面。太极语言支持元编译和宏，支持动态语法，而coffee-script没有这些特征。因为这些不同，两种语言在设计和实现上也体现出差异：太极语言内核上采用了统一的有利于将程序作为数据来处理的json表示，其主体是嵌套列表；同时，太极语言的解析器完全基于手写方式实现，而coffee-script使用了jison。

因为太极语言和coffee-script有如此多的渊源，因此，在附录中专门有一节比较这两种语言。

和**javascript**的比较

太极语言编译产生javascript目标代码。因此，太极语言代码必须被编译后才能被执行，即使在repl下，每次交互也是执行的编译后的目标代码。

正因上述特性，我们可以在太极语言中充分的利用javascript的语言能力，利用丰富的javascript生态，javascript的程序库，各种框架，工具等等。

太极语言的新的语言特性，例如更直觉的语法，更可读的代码，元编译，宏，类，可定制可扩展的语法，这些能力可以帮助我们提高开发效率，编写更优化，更可移植，可配置，模块化程度更高的代码。

因为太极语言需要编译为javascript才能运行，特别是浏览器不能加载太极语言代码，这会给使用太极语言带来某些不便，特别是在调试方面。太极语言现在还没有源码映射功能，希望尽快实现这一功能，以便能缓解这一问题。虽然有这些不便，从总体上看，太极语言将给我们带来更多的益处。这也是我个人使用coffee-script开发太极语言的强烈体会。

和**lisp**的比较

lisp是一门强大的独特的语言。它对于语言设计极其具有启发性，而且能很大的提升程序员对于程序和软件设计的理解。太极语言的出现，首先必须要感谢lisp的设计思想，包括以嵌套列表作为程序的表示，将代码作为数据的方法，递归函数，尾递归优化，宏和编译时间计算等等。

太极语言虽然采用了主流高级语言的语法，却具有和lisp的同样强大的宏。

和**rebol**的比较

rebol语言也可以看作lisp语系的一门方言。当然，rebol也有很多独特的方面，这包括它采用[] 而不是()作为表达式定界符，作为脚本语言的能力，对各种平台以及internet的支持，它的方言能力等等。但是，rebol没有宏，作为方言能力的基础，它的parse函数的解析能力还有提高的空间。太极语言从rebol学到了很多东西，并有所发展。

和**metalua**的比较

metalua直接带给我关于元编译和元语言的更明晰的概念。metalua也具有语法扩展的能力。

和**C/C++**的比较

太极语言从C/C++学到了预处理这一概念。C/C++预处理所用的符号#if, #include等启发了太极语言的元算符，太极语言从C/C++继承了#这个符号。。

C/C++作为系统编程语言，开发的软件可以实现极高的性能，是用于设计基础性软件、系统软件的工具。对于脚本化的任务，快速变化的需求，强调开发效率的应用，使用C/C++是不合适的。太极语言强调开发效率，编译成为javascript，主要用于web开发相关的环境。

编译过程和中间表示

太极语言编译过程首先可以分成两个阶段，代码解析阶段和编译阶段。而编译阶段又可以进一步划分为元编译，编译变换、表达式转换，编译优化，代码产生等多个步骤。编译过程各阶段各步骤采用了以列表为主体json中间表示。理解这方面的知识对于掌握太极语言的高级功能是必要的。这方面更深入的知识将在第八章专门详加介绍。

太极图解

lisp风格和**c**风格

一种以追寻优美为前提，一种以简单实用为原则。

一种是纯粹的艺术家，一种是实际的工艺家。都有自己的原则，都有自己的审美观。如同理论物理学家与原子能核电站的工程师，各自专工不同，侧重不一。

一个领域的日常运用离不开前者，但是一个领域的整体的发展离不开后者。

一般来讲，lisp风格的研究将成为地底下的根和土，而c风格的研究会成为地面上的枝和叶。

理论和技术的关系，这两个领域的不同探索者之间的关系，可以用牛顿与瓦特。

还有另外两个层面的对比：技术和产品的关系。这方面我们可以用Dennis Riche和乔布斯作为最显著的例子。

一种讲究实用，一种讲究实利。一种面向工具，一种面向客户。一种通过工具的改善本身来赢得内心的快乐，一种能通过唤醒客户的需求来满足自己的欲望。

上述看似差异很大的两种风格，其实它们不是分裂的，对立的，而是互补，可以融合的。随着技术的发展，就会有机会实现两种风格的融合。随着计算机和互联网的高速发展，只是从理论转向技术、工业，转向生产和实用的周期越来越快。

太极语言也可以看作两种风格发展走向融合的一个结果。

太极之禅

有无相生

难易相成

多少善恶

是非因果

人机相应

习惯自然

道名非常

一以贯之

结语

本章介绍了太极语言的总体功能特性，介绍了它和某些其它高级语言的异同，简单描述了太极语言的编译过程和中间表示。太极图解和太极之禅可作为帮助理解高级语言、程序以及太极语言的一个启发式背景线索。

下一章“起步”开始我们来实际地使用太极语言。

安装太极语言

我们先要按照node.js，请选择0.8.0以上的版本。

0.8.0以上的版本已经自动附带安装了node.js的包管理器npm。我们先查看npm是不是已经加到了系统路径。Windows下可以查看计算机/属性/高级系统设置/环境变量/path。[root]#echo $PATH 命令。如果没有，请先将npm的路径添加到系统路径中。

下一步，我们使用下面的命令之一就可以用npm安装太极语言。

- 全局安装，将太极语言安装到npm的全局包文件夹："../npm/node_modules", 并在"../npm"下添加taiji脚本命令

```
npm install taiji -g
```

- 本地安装，将太极语言安装到本地包文件夹node_modules：

```
npm install taiji
```

- 本地安装，将太极语言安装到本地包文件夹node_modules，并向package.json的添加taiji为依赖包（"dependencies"段）：

```
npm install taiji --save
```

- 本地安装，将太极语言安装到本地包文件夹node_modules，并向package.json的添加taiji为开发依赖包（"devDependencies"段）：

```
npm install taiji  --save-dev
```

读入求值打印循环

太极语言命令行工具可以执行读入求值打印循环，用习惯的术语叫repl(read eval print loop)。repl是lisp开创的传统。通过一个交互式的工具，让用户能执行简单的任务，熟悉语法，了解函数的功能、参数配置等。绝大多数动态语言，脚步语言，比如python，ruby，javascript都有这个功能。某些现代的高级语言也提供了repl，例如haskell。

在shell下输入不带参数的太极命令，即可进入repl，如下所示：

```
$> taiji
```

现在系统进入了太极语言的repl。系统提示为

```
taiji>
```

我们可以输入太极语言的语句或表达式：

```
taiji> 1
1
taiji> print "hello world!"
hello world!
```

如果希望退出repl，请按CTRL+C。

下一节我们将介绍一些简单太极语言的语句或表达式。你可以在repl模式下体验这些示例。

一些简单的例子

根据上节介绍，我们进入太极语言的repl，然后会看到如下提示符：

```
taji>
```

在提示符状态下，我们可以尝试输入这些例子：

```
1
```

```
"hello"
```

```
a = 1
```

```
b = 2
```

```
a
```

```
b
```

```
1+2
```

```
a+b
```

```
print "Hello world!"
```

```
if a==1 then print "a is ", a
```

```
for x in [1 2 3] then print x
```

更多的例子

```
let a=1 then let a=2 then print a
```

```
let a=1 then let a=2 then print a
```

```
let a=1 then { let a=2 then print a }; print a
```

```
do print a, print b where a=1, b=2
```

```
i = 0; do print i++ until i==10
```

```
array? #= (obj) -> ` (Object::toString.call(^obj) == '[object Array]')
```

```
array? # 1
```

```
array? # []
```

完整的程序

Hello world

```
taiji language 0.1
print 'Hello world!'
```

太极语言的模块文件由两部分：模块头部和模块体。模块头部是从第一行开始以及随后具有更多缩进的所有各行，一直到第一行没有缩进的行（也就是第一列不是空格或制表符的非空白行）。从该行开始直到文件尾是模块体部分。

模块头部第一行，也就是整个文件第一行，必须以如下文字开始：“taiji language x.y，其中x和y是两个数字，分别代表主版本号和副版本号。解析器会检验这两个版本号，如果不是解析器接受的版本号，将报告一个错误。随后各行可以是任意的格式。可以将作者，许可证，邮件，范例，文档等内容放置在模块头部。

模块体部分包含了有效的太极语言代码，解析器将依据解析模块体的规则产生太极语言的解析结果表示，作为后续编译阶段的输入数据结构。

本章结语

本章介绍了安装太极语言的方法，命令行工具taiji的功能，交互式执行太极语言代码的方法，也给出了一些太极语言的代码示例，并解释了太极语言代码文件的特征和要求。通过这些内容，我们对太极语言有了一个初步的印象。

接下来，从第四章到第七章，我们将全面地一步一步介绍太极语言各方面的基础知识。下一章（变量和数据）首先介绍太极语言处理数据和定义变量的方法。

字符串

太极语言用'some string'，'''some string', "some string", """some string"""四种形式表示字符串。通过使用不同的形式，可以控制字符串是否转义和是否插补。

转义字符串：

字符串转义指将字符串中出现的一个转义字符序列替换为另一个字符。

'...'或"..."中的字符串中如果出现了转义字符序列，将根据转义规则转换成目标字符。'''...'''或"""..."""不会转义其中的字符，即使出现了合乎规则的转义字符序列。 太极语言转义字符序列及其转换规则遵从javascript的习惯。以下是一些常见的转义序列：\n表示新行字符，\r表示回车字符，\t表示制表字符，\表示字符"\"，\'表示字符''', \"表示""。

转义字符串内部可以有续行符号，当行末最后一个字符是"\"并且紧跟新行字符(\r或\n，并且之前不能有空格或制表符)。下一行将和本行连接，续行字符和新行字符都会被丢弃。

插补字符串

字符串插补是指将字符串中满足插补语法的表达式求值后将求值结果转换成字符串插补到结果字符串中。

太极语言中"..."和"""..."""形式的字符串是可以插补的。 当这两种字符串中出现了形如(operatorExpression), [list], {block}，$compacetClauseExpression的字段,太极语言将先求值(operatorExpression), [list], {block}，compacetClauseExpression，然后将求值结果的字符串表示插补到整个结果字符串中。

根据上述规则，可以总结如下：

- '''...''': 既不转义也不插补
- '...': 转义但是不插补
- """...""": 不转义但是可插补
- "...": 既转义又插补

太极语言中上述四种字符串都可以是多行字符串。

列表

太极语言以[]表示列表，或者叫数组。比如[1 2 3], [a b c], [1, 2, 3]

[1 2 3]和[1,2,3]是等价的表示。

[1, 2, 3 4, 5, 6]

表示一个二维数组

[abs 1 sqr 2] 表示[abs(1), sqr(2)]

字典

{. 1:2 .}

{.1:2; 3:4.}

{. 1:2; 3:4; 5:6 .}

{. 1:2; 3: 5:6 .}

{. 1:2; 3: 5:6 7:8 .}

空字典 {}

正则表达式

为便于解析，太极语言对正规表达式有一个比javascript更为简单的规则：

在排除行注释//, 行内注释/*引导符的情况下，赋值号或小括号后面的单个“/”开启一个正规表达式。正规表达式必须在单行内，不能跨行和续行。 其它情况下/都不引导正规表达式，而是除号或别的符号。 行首的/ 将引导代码块注释

变量

高级语言都有变量。用变量可以代表数据，从而使程序更为通用。

变量声明

```
var a
```

变量赋值

```
a = 1

b = 1
```

```
print a, b
```

常量

```
const MON = 'Monday'
const PI = 3.14
```

常量和变量

根据语言的不同，程序可以定义常量和变量。早期的javascript实现在语言层面不区分常量和可变变量，但是新的javascript增加了const关键词，允许定义常量。Javascript有程序库可以产生不可变的数据类型，可以保证其内容一旦产生，即不能再被修改。然而，和变量本身不能被修改，也就是说变量不能够被多次赋值，是两个概念。因为这只能够保证数据本身不可变，无法保证变量不被多次赋值。

coffee-script没有提供const关键字，所以变量都是可变变量。

太极语言赋值的时候，如果遇到的是本地作用域没有定义的变量，会自动添加变量声明，而且通过这种方式定义的变量默认是个常量。

标识符

变量的命名是编程的一个重要问题。

太极语言的标识符规则有javascript基本一致，区别是除了首字符以外，后续字符也允许使用!和？。比如begin!, integer?等也是合法的太极语言标识符。太极语言标识符在需要转换到javascript标识符的时候，会将不合javascript规定的字符进行转换。现在的实现是都转成字符$。

关键字

被转义的关键字用作标识符

在关键字前面加转义字符\，可以取消该单词作为关键字的语法功能，转而用作普通的标识符。使用这个功能的时候要谨慎，不要将javascript的关键字转义成为标识符，这样会导致javascript目标代码中不合适地使用关键字作为变量名、函数名，从而使得目标代码出现语法错误。

变量的作用域

作用域控制变量定义的有效范围。高级语言最常用到词法作用域和动态作用域。现代语言更是以词法作用域作为首选。

在javascript中变量如果没有用var进行声明，而是有赋值产生，那么默认为全局变量。这是非常影响程序的模块性的一个语言特征。因为很可能在不知情或不小心的情况下修改了全局变量，从而改变其它函数、模块的功能。

局部作用域代码块

带有新一层本地作用域的代码块称之为局部作用域代码块，简称为局部代码块，或者局部块。太极语言有几种方法产生局部作用域代码块。使用let，letrec!, letloop!, block!语句, 函数定义语句。作为编译的目标语言，从javascript的角度来看，函数作用域和其它局部块有所不同，因为javascript自身并没有块级作用域，所有的var变量都位于函数级或者文件级别(浏览器下面还有文档级）

let a = 1 then let a = 2 then print a print a

a = 1 block! a = 2 print a print a

coffee-script的处理要好一些。赋值的变量要在模块中各层作用域中都没有定义过，那么总是默认产生一个本地变量声明。但是，如果在同一模块包含该变量的外层作用域预先定义了该变量，那么变量的赋值的是这个外层的变量。如此处理，模块化的程度要高一些，但是，还是有可能出现不注意而错误地修改外层变量的情况。实际上我在开发太极语言解析器中就曾经因此而导致了bug，花费了不少调试时间。

给外层变量赋值

因此，在太极语言中，默认情况下赋值总是针对本地的变量。如果赋值语句左边是一个变量，而该变量在该作用域下是第一次被赋值，那么太极语言将自动产生一个变量声明语句。

如果要对外层变量赋值，必须在变量名之前添加@@符号。

@@outerVariable = 1

本章结语

本章介绍了太极语言代码中的字符串，列表，字典和正规表达式的文字表示，也介绍了太极语言的变量的命名和变量的作用域等知识。

下一章我们介绍太极语言的基本命令和运算。

注释

//././//

基本命令

赋值运算 右结合 普通赋值 扩展赋值

打印 print 或console.log

let 和 where

单引式

求值(eval)

回引与消引 quasiquote

引用式

引用号~，波浪号，为什么不用单引号？尊重主流编程语言将单引号作为单引号字符串的习惯。 波浪号，其它编程语言一般用作位反。而取消求值某种程度与此有类似的涵义。而位反是个不常见的操作，移作它用不会有太多不便。 位反符号哪里去了？ ~是替代的位反符号

回引式

虽然回引式也可以有其它的用途，但是更多地是和宏的使用相关联。因此，本书将在“元编译和宏”一章展开介绍回引式的知识。

javascript运算符

因为太极语言是一门编译成javascript的语言，因此，太极语言提供了javascript的各种运算符，包括一元运算符，二元运算符。其中，一元运算符又包括前缀和后缀运算符。

下面列出的运算符，除非特别说明，则和javascript语言中同样的运算符意义相同。另外，这些运算符彼此之间相对的优先级也和javascript保持一致。

- 一元运算符
    - 前缀: `yield new typeof void delete + - ! ++ --`

    因为太极语言符号解析的特点，`!!`会被解析为单一符号。因此，太极语言将`!!`作为前缀运算符处理。`!!`(!!x相当于!(!(x))),

    因为lisp风格的原子表达式引用号`'`已经被用作字符串定界符，太极语言选择使用~符号作为引用号，不再用它表示javascript的位反运算，位反运算改用`|%`表示。

    - 后缀: `++ --`
- 二元运算符(中缀运算符):`, == != === !== < <= > >= in instanceof << >> >>> + - * / % && ||`

    在javascript中，==和!=运算会对操作数进行类型转换，大多数情况下这不是程序希望的结果。因此，javascript程序大多数使用===和!==。coffee-script只提供了==和!=，分别转换为javascript的===和!==。太极语言遵循了coffee-script关于==和!=的习惯，但是也提供了===和!==，但是各自分别代表javascript代码中不太常用的==和!=。

    另外，为增加程序的可读性，太极语言增加了`and`和`or`表示`&&`和`||`。
- 赋值运算符: `= += -= *= /= %= <<= >>= >>>= &= |= ^= &&= ||=`

扩展运算符

出于自身的需要，太极语言扩充了一些运算符。这些运算符具有某些独特的功能。

- 前缀运算符：`# ## #/ #- #& ~ ^ ^& .. ... @@ @``
- 中缀运算符：`.. ...`
- 赋值运算符: `#= #/=`

算符优先级

系统预置的运算及其优先级

==、=== 和 !=、!===

空格、换行和缩进影响求值顺序

结语

块结构

语句块：一组缩进的代码行，关键字语句中的一个分句中一组句子或子句等等都构成一个语句块。不管是单个语句或者都个语句都可以用{}封包起来。{ print 1 2 3 }

[设计注记] 太极语言是一个缩进语法的语言。现有的其它缩进语法的语言没有显式的语句块定界符。比如python和coffee-script。那么为什么太极语言还要提供{}定界符来封包语句块呢？有心人很容易产生这个疑问。实际上，这个问题有一个自然而直接的答案，因为太极语言最强大的两项特性驱使它必须要有{}这种形式的代码块：元编译能力和动态语法能力以及它们用到的一组算符。

block!局部块语句

在上一节介绍过，利用block!可以定义一个带有新的内层局部作用域的块。

a = 1 block! a = 2 print a print a

条件执行

if语句 有时候又叫双分支语句，分支语句。这是所有高级语言最常见的语句。它的语法如下。 if condition then action [else action] if condition then action else action

if condition then action else acton

if condition then action else action

或者象这样类似python的写法： if condition: action [else action]

if sex=='female' then print "She is a woman." else print print "He is a man."

我们看到，和coffee-script相比，太极语言中的语句格式（特别是then分句的写法）一般要更加自由一点。

switch-case语句

对于多分支，太极语言提供了switch-case语句。在可能的情况下，switch-case语句会被转换为javascript的switch语句。否则，将转换为if-else if-else语句。

switch-case

switch day: case ["Mon", "Tue" , "Wen" , "Thu" , "Fri"] then print "I'm working." case "Sat" then print "I'm playing." else print "I'm having a rest."

重复执行

c-for语句 `for init; test; step then body`

`for i=0; i<10; i++ then print i`

for-in和for-of语句。

`for item [index] in range then body`

`for key [value] in hash then body`

```
guests = {. "张三": 39; "李四": 45; "王五": 22 .}

for guest in guests then
  print guest+"是我们的客人。"
```

```
for course score in sheet:
  print "$course得了$score分"
```

while语句

```
i = 0
while i<5 then
  print "闹钟响$i次了。"
if day=="Sat" or day=="Sun" then print "继续睡。"
else print "我必须起床了。"
```

do-while语句

```
do letter = tryGetALetterFromMailBox() while letter
```

do-until语句

```
do run() until outOfGas()
```

break语句和continue语句

```
tickingForTimeBomb = 30
while 1
  if --tickingForTimeBomb == 0 then break
print 'boom!'
```

```run() while 1 if not atStation() then print "没到站"; continue if reachFinalStop() then print "终点站到了"; break print "到站了" if noOneWantLeft() then continue stop() passengerLeft() run()

label!语句

label@ clause

i = 10 label1@ { while 1 while 1 if i = 0 then break i-- }
```

异常

javascript具有try-catch语句，通过它可以处理异常。太极语言提供了对于的构造

try语句

try fn() catch e then body else print 'other error' finally print 'always do this'

throw语句

抛出异常。

产生javascript的throw语句。

本章结语

本章介绍了太极语言中常见的流程控制方法：包括块，条件执行，重复执行和异常等控制结构。通过这些方法，我们可以实现各种算法，编写小规模的代码。

面对更大规模的应用，需要更多的机制。下一章将介绍更多组织程序结构的方法：函数，类和模块。

函数

函数是太极语言的一等公民，一级成员。这样说的含义是：函数和其它类型的值一样，可以作为值被赋值给变量，赋值给其它对象的成员，可以作为参数传递，本身也可以作为函数的返回值。

函数定义

可以有四种方法定义函数：

```
(parameter[, ...]) -> functionBody
(parameter[, ...]) => functionBody
(parameter[, ...]) |-> functionBody
(parameter[, ...]) |=> functionBody
```

用->和|->定义的函数，函数体中的this和@都相当于javascript中的this。为了方便定义绑定this的函数，和coffee-script类似，用->和|->定义函数，太极语言将在函数定义之前保持this到_this, 函数体中的@和this将被替换为_this。

用->和=>定义的函数，将自动返回函数体最后表达式的值，用|->和|=>定义的函数，不会自动返回函数体最后表达式的值。

用例子来说明：

`square = (n) -> n*n`

会自动返回n*n, 不需要写成n*n

`myPrint = (x) |-> print x`

将编译为`myPrint = function(x){ console.log(x); }`, 没有返回语句，根据javascript的规则，返回值为undefine。

其中parameter可以是几种不同类型的参数：x, x=defaultValue, @x, @x=defaultValue, x..., @x...，现在分别地介绍它们的作用：

参数x后面带有=defaultValue，将给该参数指定默认值。设置默认值的方法是在函数体中添加对应的赋值语句。

参数x前面带有@，将把参数的值也赋值给this.x。可以用这种方法很方便地设置对象的初值。

参数x或者@x后面带有...，可以将对应位置的参数列表赋值给x或者this.x

函数调用

先求值参数，再求值函数整体。

闭包变量的传引用方式

javascript中，闭包变量采用的是传引用方式

传引用方式带来的问题

javascript下的解决 (function(a){...})(a) http://stackoverflow.com/questions/750486/javascript-closure-inside-loops-simple-practical-example

coffeescript的解决方案：do (x=x) -> ... http://rzrsharp.net/2011/06/27/what-does-coffeescripts-do-do.html

一些简单的函数

```
sumN = (n) -> result = 0; i = 0; {while ++i<=n then result += i}; result
```

```
factorial = (n) -> result = 1; i = 1; {while ++i<=n then result *= i}; result
```

```
fibonacci = (n) ->
```

优化递归函数

太极语言可以对某些类型的递归函数进行适当的优化，将递归调用转化为循环。而且，这种优化不止局限于尾递归，对于某些合适的递归函数，即使它们不是尾递归，太极语言也可以进行这种变换。

可以用letloop!语句指示太极语言进行这种变换。不过，需要小心地使用letloop!语句。因为这还只是太极语言的一个试验性特征，没有经过充分的测试，暂时也没有获得一个明确的规范，能够确定哪些递归函数能够进行正确地转换，而哪些会导致错误。

```
letloop!
  f1 = (args...) -> body
   ...
then body
```

用一组例子来说明letloop!的功能：

```
letloop! f = (x, acc) -> if! x==1 acc f(x-1, x+acc) then f(3, 0)
```

上面的函数是尾递归的，将被编译成为下面的javascript代码：

```
var f2 = function (x, acc) {
      var f2;

      while (1)
        if (x === 1)
          return acc;
        else {
          acc = x + acc;
          x = x - 1;
        };
    },
    t = f2(3, 0),
```

```
letloop! f = (x) -> if! x==1 1 x+f(x-1) then f(3)
```

上述函数不是尾递归的，但是，利用letloop!, 将被正确地编译成为下面的javascript代码：

```
    f3 = function (x) {
      var f3 = 1;

      while (1)
        if (x === 1)
          return f3;
        else {
          f3 = x + f3;
          x = x - 1;
        };
    },
    t2 = f3(3),
```

再看看下面求最大公约数的例子

```
letloop! gcd = (a, b) -> if! a>b gcd(a-b, b) {if! b>a gcd(a, b-a) a} then gcd 9 12
```

被转换为如下的javascript代码：

```
gcd = function (a, b) {
      var gcd;

      while (1)
        if (a > b)
          a = a - b;
        else if (b > a)
          b = b - a;
        else return a;
    },
    t4 = gcd(9, 12);
```

太极语言也能处理某些互递归的函数，比如下面的例子

```
letloop!
  odd = (x) -> if! x==0 0 even(x-1)
  even = (x) -> if! x==0 1 odd(x-1)
then odd(3)
```

被编译为如下的javascript代码:

```
    odd = function (x) {
      return loopFn(x, odd);
    },

    even = function (x) {
      return loopFn(x, even);
    },

    loopFn = function (x, fn) {
      var loopFn;

      while (1)
        if (fn === odd)
          if (x === 0)
            return 0;
          else {
            x = x - 1;
            fn = even;
          }
        else if (fn === even)
          if (x === 0)
            return 1;
          else {
            x = x - 1;
            fn = odd;
          };
    },
    t3 = odd(3),
```

类

javascript没有直接提供定义类和基于类的继承方法。但是，利用javascirpt的原型对象和原型继承机制，我们可以很容易地实现类定义和类继承。coffee-script语言根据这个原理提供了语言级的class语句。与之类似，太极语言也提供了class语句。

语句格式

```
class name [extends superClass] [classBody]
```

- 构造函数：

  ```
  :: = (args...) -> body
  ```

- 成员函数

  ```
  ::fn = (args) -> body
  ```

- 成员变量

  ```
  ::fn = value
  ```

引用超类

在构造函数或者成员函数中，可以用super关键字来引用超类。

模块

模块化编程有利于组织大型应用的结构。太极语言提供了帮助模块化编程的构造，主要是三条语句：include!, import!和export！

利用**javascript**的模块化功能

因为太极语言被编译为javascript，太极语言可以使用javascript的一切语句和函数，也可以使用任何的javascript程序库。因此我们也可以使用javascript环境下提供的模块化功能。在node.js下我们可以利用原有的node_modules以及npm包管理工具，在程序中和原来一样的方法使用require，module，exports等原语。在浏览器环境下当然也可以用requirejs，seajs等等工具提供的方法。

太极语言提供的模块语句

include!语句

在目标模块包含到本模块中。

import!语句

将目标模块或者目标模块的某些导出变量导入到本模块中。

export!语句

导出本模块的某些变量。

因为上述模块语句同时影响到编译时间代码和运行时间代码，因此本书将在“元编译和宏”的“包含和导入模块”一节进一步加以解释。

结语

为了构造更大的应用程序，保持程序的清晰可读，容易扩展，编程语言必须提供组织程序结构的机制。函数、类和模块是最常见的代码构造。太极语言提供了定义函数、类和模块的方法。

到本章位置，我们介绍了太极语言的基本知识，我们可以用它来编写程序完成大多数的任务。从下一章开始，我们将介绍太极语言的独特功能。

编译过程

太极语言的编译过程首先可以分成两个阶段，代码解析阶段和编译阶段。而编译阶段可以进一步划分为子阶段：编译时间代码编译阶段和运行时间代码编译阶段。其中编译时间代码也称为元代码。编译时间代码编译子阶段称为元代码编译阶段，或者元编译阶段，或者简称元编译。运行时间代码简称为目标代码，运行时间代码编译阶段也简称为目标编译。

首先来看目标编译的过程：目标编译要经过四个步骤：编译变化，表达式转换，编译优化和代码产生。每个步骤都以一个中间表示作为输入，并以一个中间表示作为输出，直到代码产生步骤，该步骤将产生最终目标代码的文本表示。

元编译阶段先进行元变换，得到元代码用作的编译变换输入中间表示。然后经过和目标编译一样的过程，产生元编译程序的目标代码。元变换过程还会记录遇到的目标代码用作编译变换的中间表示的各个片段，放置在一个数组中。用这个数组作为上述元编译程序的参数，将产生用作目标编译输入的中间表示。

我们看到，太极语言已经将整个编译过程划分成了很多步骤，每个步骤都只完成一件事情。同时，这些步骤都采用了统一的中间表示形式，也就是以列表（按照javascript术语也叫数组）为主体的json表示。

中间表示

中间表示是太极语言的核心一环，理解太极语言的中间表示将对理解太极语言整体概念、语法及解析和具体使用，都会有很大的帮助。因此，在具体讨论编译过程的其它内容之前，让我们先来看一看太极语言解析、编译、求解过程中都将涉及到的中间表示。我们也可以把这一中间表示看作一个语言，因此也可以将中间表示称之为中间语言。将这种形式的程序或代码称为中间代码，有时候也叫做中间程序。 太极语言采用json作为自己的中间表示。json除了原子以外，包括数组（也叫列表）和对象两种结构化数据。作为lisp风格的语言，理论上我们完全可以选择只使用数组这一种结构化容器。但是出于实用性的目的以及实现方面和重构的原因，也使用了对象。

下面介绍源代码和目标代码中的各种构造在中间表示中的形式：

变量或符号：以字符串或json对象。如果是字符串，不应该被封包在""之中。

字符串: 太极语言源代码中的字符串（将被转换成目标代码的字符串）以字符原因串表示。该字符串必须被封闭在""之中，也就是说，这个字符的的第一个和最后一个字符都是"。

数组：中间表示的空数组最终为被转换为目标代码的空数组。否则，该数组表示一个表达式。数组的第一项是该表达式的命令，它可以是个变量、符号或者数组。

数字：源代码的数字在中间表示表示为数字或者json对象。

为了提供错误的源码信息，包括开始和结束行号，列号，文本位置等，字符串，数字，变量或符号有可能被封包在json对象中。数组则有可能附带这些信息。

当前的中间表示在实现上不够优化。第一个问题是附加源码信息的方法不一致，第二个问题是变量、符号、字符串有可能是字符串形式，也可能是json对象形式。这些不一致导致在编译过程中经常需要利用一个函数取得项目的实际值。这不必要地复杂化了编译过程。这个函数调用的频率非常高，降低了编译速度，对编译速度会有所影响。将来可能会采用更为一致，有利于简化编译算法和提高编译速度的实现。

代码解析

代码解析将太极语言程序转换为后续编译阶段的输入。太极语言的代码解析完全依靠手写实现，没有采用任何解析器产生器。本书将在下一章专门太极语言的语法和代码解析方面的内容。

元编译和元变换

元编译阶段先进行元变换，得到元代码用作的编译变换输入中间表示。然后经过和目标编译一样的过程，产生元编译程序的目标代码。元变换过程还会记录遇到的目标代码用作编译变换的中间表示的各个片段，放置在一个数组中。用这个数组作为上述元编译程序的参数，将产生用作目标编译输入的中间表示。

元变换的原理是：对于太极语言程序，如果其中的语句是目标代码，则将其目标代码列表的一项纪录下来。如果是元语句，则进行对应的变换。

编译变换

编译变换是目标代码编译过程的第一个步骤。它的任务是：实现各种预定义的运算符(包括前缀，后缀，二元运算符，赋值运算符等），预定义的表达式及命令(比如if, begin, while, for), 实现函数定义(->, =>, |->, |=>)，产生合适的变量声明（涉及到var, const以及赋值引起的变量定义)，管理变量的作用域。

表达式转换

经过编译变换得到的中间表示保护的构造，如果直接转换为javascript，那么，如果在那些将要转换成javascript表达式的构造中含有语句，将产生不符合javascript要求的代码。因此，编译器通过表达式转换这个步骤重新安排这些构造，以获得一个合乎目标语言要求的形式。

很自然可能会有人产生一个疑问：为什么先进行表达式转换，而不是编译优化？难道表达式转换不应该最靠近代码产生这个步骤吗？这是因为表达式转换过程中，将产生一些临时变量及其赋值语句。将表达式转换步骤放在前面，可以借助随后的编译优化步骤消除掉不必要的变量和赋值。

根据我的研究，在处理表达式转换这个问题上，其它一些语言采用的方法一般是这样的：对某些简单的特殊情况，比如将if语句赋值给某个变量，将赋值移到if语句的两个分支当中。而对其它一些更复杂的情况，则采用如下方法解决：将语句封包成函数调用之中。这种方法的一个问题是会引入函数调用的开销，另一个显而易见的问题是无法处理return语句，也无法处理break或continure语句需要跳转到封包函数之外的情形。coffee-script采用了这种方法。虽然比较而言它已经很广泛地支持将语句转换为表达式，但依然不允许将break，return语句使用在表达式构造中。

太极语言没有采用上述方法。它进行表达式变化的基本思路是：将所有构造分离成两部分：语句部分和表达式部分，然后再加以重新组合。这是一个很通用的设计，可以一致的处理任何语句的表达式转换。当然，转换过程必须确保语义的正确，而副作用是其中需要考虑的重点问题。

编译优化

优化指令 循环外提 函数外提 编译时间计算 无io作用的代码段 尾递归优化，普通递归优化

optimization 1:

if 1 then a else b ---> a

if 0 then a else b --> b

1+2 -> 3, etc.

a and not a

while 0, 1==0, 1==2, false, undefined, null, '', etc

optimization 2:

below need a has no side effects (a is not function call(), etc.)

if a then if a then b else c ---> if a then b

if a then if not a then b else c ---> if a then c

if not a then if a then b else c ---> if not a then c

if not a then if not a then b else c ---> if not a then b

if (if a then b else c) then d else e # no optimzation

if (if a then b else c) then if a then d else e -->

if a then if b then d

else then if c then e

while 0, while false

optimization 3: assign optimization

property optimization is not executed: be careful that getter, setter, watcher

short distance

conservative

http://www.refactoring.com/catalog/replaceIterationWithRecursion.html greatest_commonDivisor = (a, b) -> if (a > b) then return greatest_commonDivisor(a-b, b) else if (b > a) then return greatest_commonDivisor(a, b-a) else return a

greatestCommonDivisor = (a, b) -> while 1 if (a > b) then a -= b else if (b > a) b -= a else return a

代码产生

通过编译优化，得到了优化以后的中间表示。下一步，编译系统将依据这个中间表示执行代码产生步骤，产生最终的目标代码。这个步骤又被细分为两个小的步骤：第一步，产生标记项序列。第二步：产生代码文本。

先介绍第一步，产生标记项序列。这个步骤从优化的中间表示产生一个可以直接文本化的标记项数组。这个步骤的基本算法是：如果中间表示不是数组或者是空数组，那么直接产生合适的目标javascirpt代码的字符串表示。否则，根据数组的第一项执行对应的转换。比如，当第一项是"binary!", 将产生组成二元运算的左操作数，运算符和右操作数三项组成的标记数组。如果是`"if", 则将产生能够通过文本化步骤转换成if语句的标记项数组。如此等等。

标记项数组中可以包含不同的标记项：每一项可以是数组，字符串，或者是带有类型记号的一个json对象。标记的类型记号包括PAREN，BRACKET，BINARY，LIST，BLOCK，STATEMENT，FUNCTION以及其它常数值。这组值可以在下一步用于指示文本化的动作。

再来看产生代码文本这个步骤。这是一个简单直接的步骤，直接依据据标记项数组转换为最终的代码文本。对于标记项中的每一项，如果是字符串则直接连接到目标文本，如果是数组则按次序依次转换所包含的每一项，如果是json对象，则依据其中的类型记号进行指定的转换。

语法及解析方法的设计原则

太极语言设计语法的原则

对空白敏感的语法 排版的关键在于安排页面的空白，通过空白组织文字的结构。 增强可读性的关键是让程序员一瞥之间就能掌握程序的整体，包括整体结构和整体意义。

少即是多，多即是少。 更少的括号，更少的标点，更紧凑的语法，更少的限制，加上适当的空白，意味着能在更短的篇幅提供安排更多的内容，让程序员在更短的时间内输入更多，理解更多。 反过来，需要更多括号，更多标点，规则上增加更多的限制，不依赖人类可读的空白，而是面向计算机的括号匹配来组织结构，将意味着输入更慢，理解更慢，同样的内容需要更长的篇幅。

恨括号，shift键，标点 尽量减少使用括号，特别是远距离配对的括号。 利用行、缩进等方式，取消传统lisp风格语言中对括号的需要。 减少使用依赖shift键的符号。 使用[]而不是()作为s表达式的定界符，允许使用[]表示参数列表，在不影响理解和打破习惯的前提下尽量不使用需要shift键的符号作为常用的语法标记， 减少使用不必要的标点 参数的分隔，数组项目的分隔等不需要使用逗号。

缩进语法 表达式

太极语言在语法上带有两个小妖：一个叫精细鬼，一个叫伶俐虫:)，换句话说，太极语言拥有精细、灵活的语法控制能力。

关于解析器实现过程的注记

解析器是设计一门新语言的关键部分之一，也是非常困难的一项任务。我们可以用解析器产生器来完成这项任务，比如lex/yacc相关的系列产品，包括bison，jison，ply，antlr以及基于peg的解析器产生器等等。虽然这些解析器具有非常强的功能，产生的解析器也具有极其优化的执行速度，有很多的项目和语言都使用了这些工具，经过了长期的实际考验。然而，根据太极语言的设计目标，我无法选用这些产品。因为一直以来我都希望设计一门语法可以自由动态扩展的语言，基于解析器产生器的方案无法实现这一目标。为此，我做了长期的探索和研究。手写解析器也是很多语言选择的一种技术。但是以前的手写解析器其代码都相当复杂，不直观，可读性不好。特别是左递归语法的问题长期以来都没有简单的解决方案。

词法

太极语言没有单独的词法解析阶段。编写词法解析规则，匹配词法标记或单位的方法和语法解析的方法是一致的。

解析之前，在初始化阶段，解析器调用preparse函数，根据给定的文本参数执行一个预解析阶段。这个阶段将分析文本的行列信息，保存各行的起始位置和缩进位置。

太极语言解析器中以下匹配函数完成与词法相关的任务

taijiIdentifier

jsIdentifier

literal

symbol

escapeSymbol

escapeString

identifier

number

string

spaces

space：太极语言的行内注释和空白注释被识别为

bigSpace：跨行的空白，必须至少包含一个新行。

newline：新行，可以是\r, \n, \r\n, \n\r等四种组合。

主要语法成分

太极语言的代码文件（推荐扩展名.tj）由模块头和模块体两部分构成。关于模块头的规则在[起步/一个完整的程序]中已经做过介绍。现在将解释模块体中可能出现的各种语法成分。

逻辑代码行、缩进块、句子和子句

代码文件的模块体由若干逻辑代码行组成。一个逻辑代码行可以是一个文本行，没有附加的缩进块。也可以是一个文本行跟随一个缩进块。第一个文本行被称为引导行。

缩进块是从一个缩进开始，直到遇到一个与开始缩进的行具有相同缩进（具有相同的非空白字符开始列号）或更少的缩进的行。一个缩进块可以包含一个文本行或多个文本行。这些文本行可以是一个逻辑代码行，也可以组成多个逻辑代码行。

一个逻辑行可以是一个或多个句子组成。

句子由多个子句组成，直到遇到分号或者新行。

表达式 模块，块，缩进块列表，连词子句，括号运算式

行与块 模块 一组顶级的块

块 连词子块 以连词开始的一个块

行 分句，连词子句，逗号小句 子句：以分号或换行结束的若干项目的列表 关键词语句：以关键词开始的子句，具有特殊的语法，类似于其它语言中的语句 连词子句：以连词开始，以分号、连词或换行结束的若干项目 逗号小句：以逗号、分号、连词或换行结束的若干项目的列表 连词：以冒号结束的词 if 1 then 2 else 3

```
    if 1
    then 2
    else 3

    使用标准的、习惯的连词，促进交流，增加可读性

  关键字：
    作为符号的字符串，出现在表头将影响程序的求解过程，出现在其它部位，taiji求解器将利用该符号从环境中查
    作为命令的字符
    if
    连词：then else
```

连词及其搭配

其它语法特征 续行符号 行尾紧跟\ 回退缩进提示符号 行首为\

原子成分 字符串 数字 变量名：在算式中只允许javascript风格的标记符，另外允许其中出现#或？，但是不得出现其它符号 运算符：在算式中不允许出现#或？作为运算符。 \identifier: generate javascript word, identifier, etc.

基本项 代码块 封闭在大括号{}中的表达式。 调用式 对象元素访问式 js表达式：在小括号内，由若干操作数通过若干运算符根据一定的优先级组合而成的式子。 冒号式：以冒号引导，以连词、分号或换行结束的若干项目的列表。 连词子句：以连词引导，以分号或换行结束的若干项目的列表

连词

其它语法特征

- 续行 续行符号紧跟换行符号，也就是说行尾的\紧跟换行符号（\n, \r, \r\n, \n\r)

太极语言在语法上带有两个小妖：一个叫精细鬼，一个叫伶俐虫:)，换句话说，太极语言拥有精细、灵活的语法控制能力。

其它语法特征 续行符号 行尾紧跟\ 回退缩进提示符号 行首为\

解析指令

3ab(1+2) sinAcosB a1+b2 A3+D5 AiHj

控制编译的与开发相关的指令 可以指定某一块在某种状态下不予编译。比如 (@quoteExpression =)@test quoteExpression = info { 表示只有在测试状态下才对该段代码予以编译，因为只有在测试状态下才需要探测某些私有成员

解析后的运算符变换

语法规则 列表综合 [expression for item in list if condition]

哈希综合 {expression for item in list if condition }

语法类型 gulp pipe stream: coffee copy clean mocha 否则 >> 应该表示向右移位运算 src >> coffee(...) >> dest

a = chan('a'), b = char('b') na na+ a *na* a! na, nb, na!, na!

w = literal('if') w!, spaces!, x=clauses!, colon!, then!

局部语法

单变量表达式（代数表达式）

数据块 [/ /]

怀念中学时代 当我们刚离开稚气的童年，迈入活力的少年，中学老师教了我们很多的知识，也培养了我们很多习惯。然而，不同的老师总是传授不一样的内容，即使是看起来应该一样的事务。比如，英语老师告诉我们，a是英文的第一个字母，abc是入门的第一课。，而数学老师说a可以表示随便什么东西。abc表示a×b×c，3a+4b表示3×a+4×n。当然，我们的英文作业本上从来没有出现过3a之类的东西。如果出现了，也许老师会疯掉。后来，我们成长为青年，进入了大学，有的人接触了一个叫计算机的东西，教这门课的老师颠覆了我们的很多光年。a×b×c不能写成abc，也不能写成a×b×c。从来没有学生这样尝试过。也许老师看到学生有试图这样尝试的意图，就已经把他赶出了课堂。 那么，那个老师是正确的？哪个时代有更值得怀念？为了工作，我们总是坚持大学养成的习惯，而内心，却总是希望回到更快乐的少年时光。太极语言给我们一个不需要纠结的机会。

本章结语

本章介绍了太极语言在语法以及解析方法（包括解析器及其词法解析）上的特点。理解这些特点对于定制和扩展太极语言会有很大的帮助。

下一章我们将介绍元编译和宏。

元编译概述

太极语言实现了非常丰富而通用的元编译特征。每一个太极语言程序都可以包含两类代码。一类代码被称为编译时间代码，也称为元级代码，元代码。这些代码里面的语句、表达式分别被称为元语句、元表达式。另一类代码被称为运行时间代码，也称为目标代码。也可以理解为程序中的这两类代码处于两个层级：元级和目标级。元级是更高层次的一级，它处理和产生目标级的代码。我们可以通过一些算符或语句指定哪些代码是元代码，控制用元代码生成目标代码的方法等。这些算符称为元算符，语句称为元语句。依据这两类代码，太极语言的编译过程可以分为两个步骤：元编译阶段和目标代码编译阶段。元编译阶段先生成一个元编译程序，目标代码将作为元编译程序处理的数据，或者是它的参数。随后，系统将执行这个元编译程序，产生一个中间表示，这个中间表示经过一系列的编译阶段，产生最终的目标程序。

这就象是M. C. Escher的画作：自己画出自己的手，听起来似乎有些不可思议。然而，太极语言确实做到了这一点。

宏在太极语言中，成为元编译的一个重要的特例。

因为太极语言有如此强大而又独特的元编译功能，因此，我们在描述它的编译过程和相关内容时，引入了很多新的名词，对于原来的概念也引进了一些新的术语。我们还对同一个实体采用了不同的名字。这样做，我们可以从不同的角度来解释这个全新的事物，帮助我们加深理解。也让一个概念在不同的上下文中显得更加清晰，避免混淆。

从下面的示例程序可以更清楚地解释上面的内容。

```
taiji language 0.1

a #= 1

b #/= 2

c = 3

# if a==1 then
  print b
else print c

array? #= (obj) -> ` (Object::toString.call(^obj) == '[object Array]')

array? # []
```

上述例子中，

- `a #= 1` 是元赋值语句，它表示元代码`a = 1`
- `b #/= 2`，是跨元赋值语句，它表示元代码和目标代码同时有`a = 1`
- `c = 3`是普通的目标级代码
- `# if ...`是预处理if语句，根据条件`a==1`(这个条件在元级执行), 目标代码中应该为`print b`。
- `array? #= ...` 在元级定义了函数array?
- `array? # []`是宏调用，将在目标代码产生`(Object::toString.call([]) == '[object Array]'`

太极语言提供了丰富的元算符和元语句。下一节将介绍这些算符和语句。

元算符

元算符介绍

太极语言提供了一组元算符，包括#，##，#/，#&，#/=，#&=，#-等。本章将解释这些算符的功能和用法。

##: 编译时间求值算符。或者称之为元求值算符。##exp 或者 ## clause表示在编译时间求值表达式或子句。

#: 预处理算符。它的功能和C/C++预处理的功能有类似之处，在元编译时对程序做某些处理。如果#后面跟随的不是预处理的几种语句(if, while, let等), 则和等同于##。

#/: 编译时间和运行时间求值算符，简称跨元算符。#/{ x + y }等价于#{ x + y };x+y;

#=: 编译时间赋值算符，简称元赋值。x #= y 等价于 #{ x = y }, 在编译时间将y赋值给x。

#/=：编译时间和运行时间赋值算符，简称跨元赋值。x #/= y 等价于 #{ x = y }; x = y, 在编译时间将y赋值给x。

#-: 退出编译时间求值算符，简称消元算符。在元编译的代码块中使用此算符，可以将此算符引导的算符表达式或子句转到运行时间求值。因此 # #- x + y等价于x +y

#call!: 宏调用算符。列表式 [#call! arg1 arg2 args...]是宏调用。实际代码中，marcroFunc#(arg1, arg2, args...)将被解析为列表式[#call! arg1 arg2 args...]。也可以写成更简单的形式：marcroFunc # arg1 arg2 arg3...。宏调用的功能是要求太极语言在编译时间先进行宏扩展，然后依据宏扩展的结果产生目标代码。宏扩展的方法是在编译时间调用不求值参数，将其直接传给函数进行计算获得结果。

#&: 编译时间求值和运行时间

元算符使用示例

```
##(1+1)
```

在元编译时间求值1+1

```
#(1+1)
```

在元编译时间求值1+1。

```
# (#(1+2) + #(3+4))
```

多重元算符等效于一重元算符，因此上述表达式相当于`#((1+2)+(3+4))

```
#(1+2) + #(3+4)
```

产生目标代码3+7

```
3+.#(1+1)
```

用.将+和#分隔为两个符号。

```
# ~ 1+1
```

```
##a=1
```

##的优先级低于=，因此上式相当于##(a=1)。也可以写成## a=1。

```
#(a=1)
```

等价于前一个示例。

```
a#=1
```

元编译时间赋值。

```
a#=1;#a
```

```
## if 1 then 1 else 2
```

```
if 1 then #1+2 else #3+4
```

```
if 1 then ##1+2 else ##3+4
```

```
compileNoOptimize if 1 then ##1+2 else ##3+4
```

```
array? #= (obj) -> ` (Object::toString.call(^obj) == '[object Array]')
```

```
array? # 1
```

```
array? # []
```

```
`# if 1 then a else b`
```

编译上述代码时将会报告错误：`fail to look up symbol from`environment:a

```
#  #-{ print 1 }
```

上述代码的编译结果是 `console.log(1)`

```
#{ -> #- { print 1 }})()
```

上述代码的编译结果是`[print, 1]`

包含和导入模块

include!语句

称为包含语句。类似于C/C++的#include语句，直接将目标模块包含到本模块中。

编译器在元编译步骤分析处理include!语句。目标模块的元表达式将出现在本模块将成为编译时间代码的一部分，运行时间表达式则将成为本模块的运行时间代码的一部分。

根据上述实现，被包含模块的将成为本模块的当前代码块和当前变量作用域的一部分，被包含模块的顶级变量将直接成为当前变量作用域中可见的变量。这是需要特别注意的一点。

语法：

```
include! "path/to/someModule.tj" [by parseMethod]
```

```
include! './module1.tj'
```

import!语句

称为导入语句。将被导入模块的编译时间代码和运行时间代码同时封包在各自层级的函数中，根据导入语句格式，向本模块引入指定的变量。

```
import! [#][/]name [as [#][/]alias #alias2] [, ...] [from] "path/to/someModule.tj" as [#]
```

以下是一些导入语句的代码示例：

```
import! './hello.tj'
import! './module1.tj'
import! a from './hello.tj'
import! a as a2 from './hello.tj' as html
import! a as A, #b from 'html.tj' as html #html2
```

export!语句

称为导出语句。导出语句用于配合导入语句，指定本模块哪些变量导出时可见。

语句格式：

```
export! [#][/]name [= spaceClauseExpression] [, ...]
```

如果name前没有层级符号，表示输出目标级变量，如果有符号“#”，表示输出元级变量。如果有“#/”，表示同时输出元级和目标级变量。

包含模块和导入模块的文件名习惯

为了识别方便，我们可以通过某种命名习惯区分倾向用于导入的模块和用于包含的模块。在这里我建议在倾向于被包含的文件名后添加符号@以作识别。比如taiji-libraries/@types.tj和taiji-libraries/types.tj。作为一个习惯，而不是语言自身带有的规则，一般来说，不要在用于包含的模块中使用export!语句。当然，我们也可以在被导入文件中包含别的文件。比如上述types.tj就包含了@types.tj

宏定义、宏调用和宏扩展

宏定义

元程序中的函数都有可能被作为宏使用。

[术语]为方便起见，后续文档中用元函数这个名词指称元程序中的函数。

元函数能合法地作为宏使用必须满足以下条件：宏的结果是个列表，或者是字符串，数字，符号，标识符等其它能产生合法的目标代码的形式。换句话说，结果必须能够作为编译变换的良定义形式。否则，将有可能产生不合语法的目标代码。甚至有可能使得后续编译过程出错中止。

宏调用

宏调用可以有以下几种方法

```
#call! macroFunction args...
```

```
macroFunction#(args...)
```

```
macroFunction # args...
```

宏扩展

```
(#macroFunction)(args...)
```

回引表达式

回引号` 键盘最左上角, 这也是lisp系语言的习惯。

消引号^ lisp系语言使用逗号，但是为了遵重包括主流高级语言以至于更一般的习惯，改用^。表示项的含义，将被回引的项重新项上来求值。 主流编程语言中，^一般用作位异或运算符。太极语言改用\^作为替代。

消列引号：lisp系语言使用,@，但是为了遵重包括主流高级语言以至于更一般的逗号使用习惯，改用^@。表示项的含义，将被回引的项重新项上来求值。

回引表达式是lisp发明的一种回引嵌套列表结构的表达式记法。利用它有助于编写宏定义。

太极语言有与lisp类似的回引表达式，但是使用了不同的符号。

- `：与lisp作用相同，引导一个回引表达式
- ^: 相当于lisp下的逗号,
- '^&': 相当于lisp下的,@

示例：

- 例：

```
` ^1
```

等同于1

- 例：

```
`{^1 ^2 ^&{3 4}}
```

会被编译成为如下的javascript代码：

```
[1, 2].concat([3, 4])
```

- 例：

```
`{ ^1 { ^2 ^&{3 4}}}
```

会被编译成为如下的javascript代码：

```
[1, [2].concat([3, 4])]
```

- 例：

```
`[ ^1 [ ^2 ^&[3, 4]]]
```

会被编译成为如下的javascript代码：

```
[\"list!\", [1, [\"list!\", [2].concat([3, 4])]]]
```

上述示例中的concat是可以被优化的，比如，`[1, 2].concat([3, 4])`可以重写为`[1, 2, 3, 4]`。系统目前还没有实现这类优化，有待将来。

比较太极语言和**lisp**的宏

太极语言下的宏功能具有和lisp同样强大的能力。

回引表达式使用不同的符号

太极语言在回引表达式中使用的符号和lisp中不同。

- `：与lisp作用相同，引导一个回引表达式
- ^: 相当于lisp下的逗号，
- '^&': 相当于lisp下的`,@`

lisp中宏调用和函数调用没有记法上的区别，如果不了解宏定义，程序员无法直接从调用位置判断是宏调用或者是函数调用。

太极语言的宏是元编译的一个特例。宏调用在元编译阶段被执行以产生目标代码。

宏定义的区别

lisp的宏定义：`(defmacro macro (parameters...) macroBody)`

宏调用的区别

lisp的宏调用：(macroName args...)

宏扩展的区别

(macroexpand macroName args...)

实现上的区别

lisp的宏一般通过如下方法实现：在编译时间，编译环境中记录了宏的名字和它对应的定义。编译的时候，如果遇到的调用是一个宏，则根据环境中记录的信息进行宏扩展并编译扩展后的表达式。这个过程可能发生递归或者说嵌套，即扩展后的表达式包含另外的宏调用，因此需要进一步的宏扩展。

从下述链接可以得到一些关于lisp及其方言scheme的宏实现知识。 ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v13/schintro_130.html http://stackoverflow.com/questions/3465868/how-to-implement-a-lisp-macro-system

太极语言最初的宏实现类似于上述方法。在发现一般化的元编译技术后，就放弃了上述宏实现方法，改为将宏作为一组元算符利用元编译的思想重新实现。

元编译的用途

优化执行速度

```
array? = (x) -> `(Object::toString.call(^x) == '[object Array]')

array? # []
```

另一个例子

在下面的例子中，text是解析器正在解析的整个文本，cursor是当前全局解析位置，parserLineno和parserRow分别是全局行号和列号。 考虑到处理换行符号，匹配任意文字的解析函数可以这样写 literal = (string) ->
start = cursor; lineno = parserLineno; row = parserRow for c in string if c==text[cur++] if c=='\n' then parserLineno++; parserRow = 0 else if c=='\r' then parserRow = 0; continue else parserRow++ else cursor = start; parserLineno = lineno; parserRow = row; return true

但是在实际应用中，绝大多数情况下我们都不需要考虑换行符号，因此使用如下的函数就可以了。 literal2 = (string) ->
start = cursor; row = parserRow
for c in string if c==text[cursor++] then parserRow++ else cursor = start; parserRow = row; return true

或者更快捷的实现 literal2 = (string) -> if text[cursor...cursor+length]==string then cursor += length; parserRow += length-1; true

下面是解析while语句的函数，作为例子，做了一些简化： whileStatement = > if literl('while') and (test = parser.clause())? and literal('then') and (body=parser.block() or parser.line()) return ['while', test, body]

出于速度的原因，我们应该将上面whileStatement实现代码中的literal替换成literal2。 对于这样一个功能简单的小问题引入两个函数，无疑会增加API的数量，从文档提供和库用户的学习使用的角度来讲都是一种负担。另外仔细分析，上面的通用文字处理函数literal还有优化的空间，由此引出了下面第二种方案。

第二种方案 literal = (string) -> i = 0; lineNumber = 0; length = string.length while c = string[i++] if c=='\n' then lineNumber++; row = 0 else if c=='\r' then _r = true; continue else row++ if lineNumber==0 if _r then -> if text[cursor...cursor+length]==string then cursor += length; parserRow += length-1; true else -> if text[cursor...cursor+length]==string then cursor += length; parserRow += length; true else -> if text[cursor...cursor+length]==string then cursor += length; parserLineno += lineNumber; parserRow += row; true

在此实现下，whileStatement的代码如下： whileStatement = > if literal('while')() and (test = parser.clause())? and literal('then')() and (body=parser.block() or parser.line()) return ['while', test, body] 然而，这样写并不能提升速度，反而会有性能损失，因为每次都必须要计算函数闭包。为了达到性能提升的目的，我们应该把对于的闭包计算提到函数之外： 因此应该这样写 while = *literal('while'); then* = literal('then')

whileStatement = > if while*() and (test = parser.clause())? and then*() and (body=parser.block() or parser.line()) return ['while', test, body]

元编译派上用场： 这种情况正好是太极语言的元编译能力的用武之地。 whileStatement = > if #(literal('while'))() and (test = parser.clause())? and #(literal('then'))() and (body=parser.block() or parser.line()) return ['while', test, body] 太极语言的元编译让我们鱼和熊掌兼得：同时拥有运行效率和编程效率。

元编译

太极语言的元编译功能是一个很曲折而且很有兴味的一个过程。因此，我觉得花点时间描述这个过程是有意义的。

最开始，我只希望太极语言具有和lisp同样功能的宏就够了，并没有将现有各项元编译功能作为太极语言最初设计目标，甚至没有意识到关于所有这些元算符的概念。当时，我对元编译的概念主要源自于对于metalua的某些了解，仅此而已。

在实现太极语言的某个阶段，我发现了可以在编译时间对太极语言的某些部分代码求值，比如1+1这样的代码块。然后，我进一步想到，象if-else-语句这类语句可以用来在编译时间控制目标代码的生成，并自然地联想到C/C++的预处理，比如#if, #define等预处理语句。同时，也基于对metalua的了解，我实现了类似的功能。与太极语言的语法相适应，我为这项功能选择了与metalua不同的符号#和##。这两个符号的选用应该说来源于C/C++预处理。在此之前，我是用#作为太极语言的行注释引导符。因此，我废除了使用“#”作为注释引导符号的决定，用它来作为预处理语句的算符。而“##”则成为直接元编译算符。实现这两个算符以后，我开始经常思索这种元编译和lisp的宏功能之间的关联。

上述元算符最开始的实现方法和最终的实现大不相同。最开始，元编译阶段直接产生目标代码的编译表示，如果遇到了上述元算符，它们将被变换成一个赋值语句，赋值语句的左边是metaList[index], 右边是元代码。该赋值语句整体被编译成javascript代码存放在一个数组中，而编译表示中以[meta! index]代表这个数组的这一项。然后在元编译的后处理阶段，将所有这些代码片段串接成一个完整的程序，执行这个程序的效果是将产生metaList数组，编译变换步骤会利用metaList将[meta! index]转换成对应的结果，产生表达式变换的输入中间表示。

宏功能的最初实现并没有采用上述元编译的方法，而是一个与此无关的并行实现。这种宏功能的实现方法也是我依据lisp宏实现的相关资料并经自己改进而得到的。该方法是将与宏名对应的函数记录在编译过程的变量环境（Environment，env）之中，而不是作为元编译代码一部分。当发现变量是宏函数的时候，按照宏调用的要求执行该函数得到作为编译变换步骤的中间表示的一部分。

上述方法的问题是元编译代码被分割成了一些碎片，失去了作为一个程序应该有的整体结构。另外，我的直觉也意识到，宏可以成为元编译的一个特例。这两个方面成为我后来思考的重点。经过很长时间之后，终于有了线索。我发现可以将上述方法进行一个逆转，即将元编译程序作为一个整体，而运行时间程序的编译表示作为元编译程序的处理数据。我意识到这是一个更完整，更强大的实现。依据这个想法，我对于元编译步骤做了重构，得到了现在的实现。

消元算符的发现和太极语言的模块特性的实现过程很有关系。基于最初的宏实现，太极语言有一个简单而不够完整的模块实现。得到新的元编译实现之后，我发现它也可以帮组我彻底解决原来的模块实现中的问题。我开始着手编码这个实现。实现过程中，我发现必须将需要导入的模块同时在编译时间和运行时间封包在函数中。因此，在已经用元算符封包整个函数之后，必须在其中取消运行时间代码的封包。由此，我自然而然的想到了消元算符。并选择了#-作为它的符号。因为这个算符的发现，我开始了一轮头脑风暴并发现了一些别的元算符，例如：#/, #/=, #call!等等。

本章结语

本章介绍了太极语言的元编译和宏功能。元编译是太极语言的一个独创的功能特性。而类似lisp的宏成为了元编译的一个特例。善用元编译和宏能帮助我们编写出更强大更优化的程序。

下一章介绍太极语言的语言定制能力。

定制运算符

增加或删除运算符

前缀 后缀 中缀

动态改变运算符的优先级

增加、删除或改变定制运算符匹配方法。

解析器算符

太极语言选用百分号%表示多种与解析器以及解析相关的算符。

%: 前缀，表示解析器属性算符，%text表示解析器的文本。

%: 动态解析语句

%%：解析时间执行算符，表示解析时间执行后续表达式，被执行表达式不会被转换。

%/：表示解析时间执行头转换表达式

%!：表示解析时间执行全转换表达式

[设计注记]选择解析器算符，解析器是将无结构的数据，特别是文档分割、解析为某种结构性数据的过程。因此用符号%有这种联想的作用，比较形象。如果我们把解析器具象化为一个寻章琢句的老秀才，那么符号%就好比他戴的那副老花眼镜。

格式控制字符，将控制printf语句中数据的输出格式。

太极语言相当于上述格式控制的逆转意义。通过这种格式控制指令，可以改变程序代码的解析方式，从而使得我们动态地指定安排代码的不同格式。

http://www.cplusplus.com/reference/cstdio/printf/

http://infohost.nmt.edu/tcc/help/pubs/python25/web/str-format.html python

```
>>> print "We have %d pallets of %s today." % (49, "kiwis")
We have 49 pallets of kiwis today.
```

http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(syntax)

% used as comment in TeX, Prolog, MATLAB,[10] Erlang, S-Lang, Visual Prolog

定制关键词语句

预定义的关键字语句

if while let try switch %

删除关键字语句

增加关键字语句

重定义关键字语句

扩展赋值语句

增加定制的赋值语句

重定义赋值语句

本章结语

本章介绍了定制和扩展太极语言的方法，这些方法包括：定制运算符，利用解析器算符扩展解析器，动态解析程序代码，定制关键字语句，定制赋值语句。这些能力与函数和宏定义能力相结合，可以方便灵活地定制太极语言，，根据需求设计领域特定语言。

到本章为止，本书介绍了太极语言的特点，基本知识和高级功能。在“跋”这一节将做总结性回顾。书末的附录列出了一些重点的和参考性的内容。

现在就让我们用太极语言来开发各种类型的应用吧。

太极之禅

有无相生

难易相成

多少善恶

是非因果

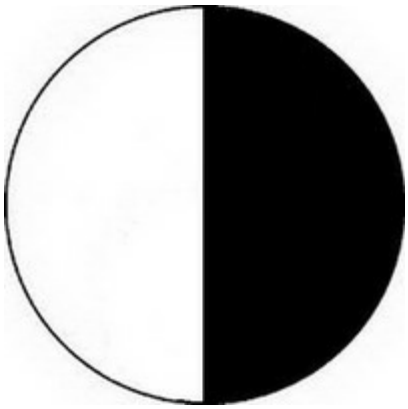
人机相应

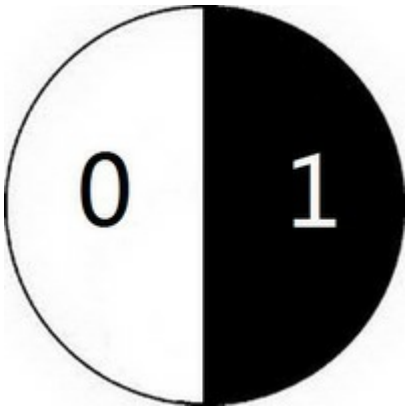
习惯自然

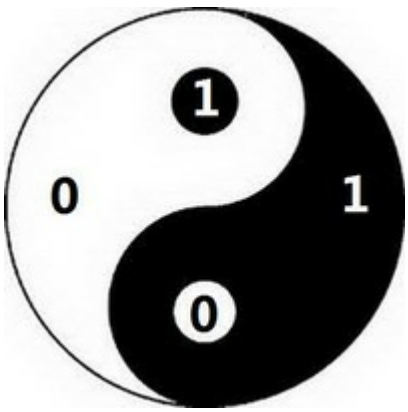
道名非常

一以贯之

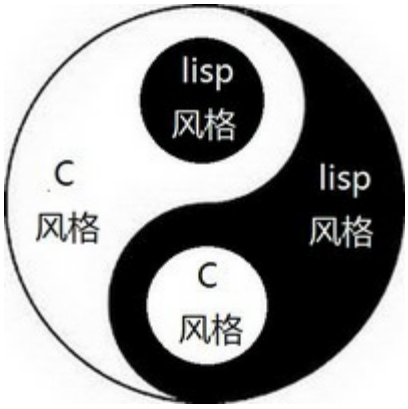
太极图解








C
风格
lisp
风格


lisp
风格
C
风格
lisp
风格
C
风格


代码
数据

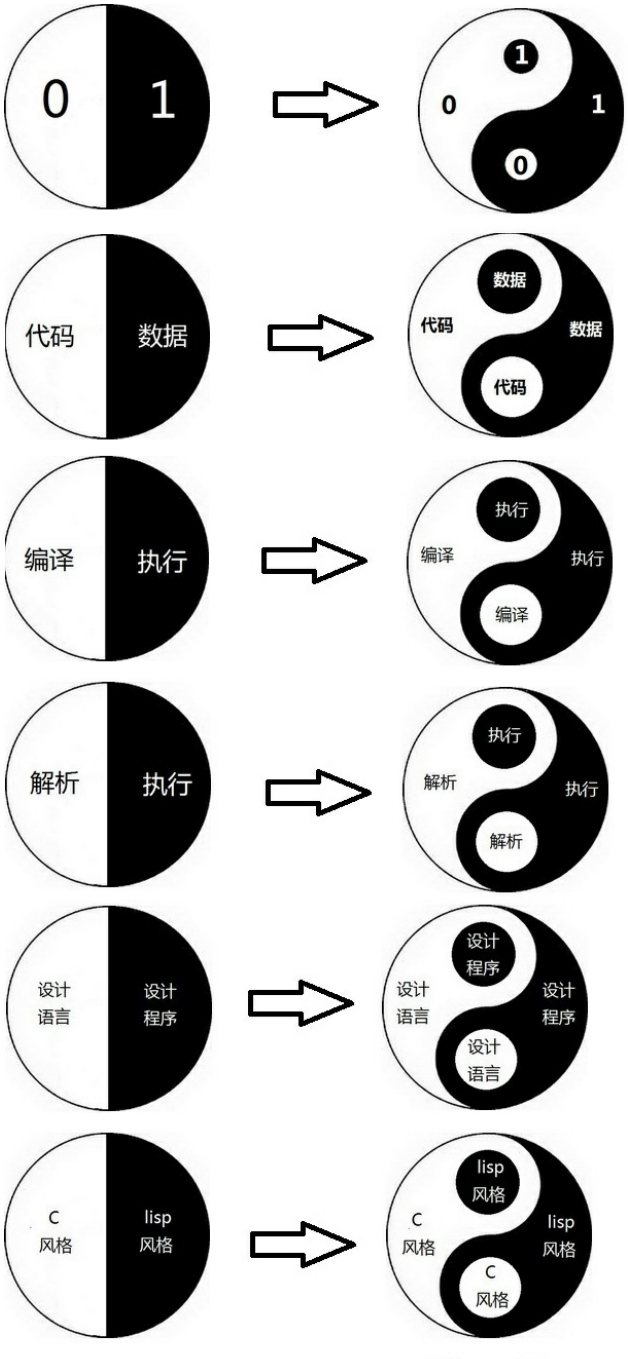

数据
代码
数据
代码


编译
执行


执行
编译
执行
编译


解析
执行


执行
解析
执行
解析


0
1
1
0
0
1
代码
数据
数据
代码
代码
数据
编译
执行
执行
编译
编译
执行
解析
执行
执行
解析
解析
执行
设计
语言
设计
程序
设计
程序
设计
语言
设计
语言
设计
程序
C
风格
lisp
风格
lisp
风格
C
风格
C
风格
lisp
风格

太极图解

常见问题列表 (FAQ)

- 为什么这门语言取名叫太极语言？

从太极图解可以看到部分答案。

- 有那么多语言了，为什么又来一门新语言？

实际上，编程语言不是够多了，而是太少了。日常生活中，人们说话从来不需要遵循僵死的语法规则，而是遵照活泼的历史和现实习惯。每个人都象尤达大师，他/她的表达习惯和句式或多或少总和别人有不一样之处。太极语言将便利人们创造新语言。每一个程序员都能创造新语言。

- 太极语言有什么用？

每种新东西出来，总是不可避免地被问到这个问题。提出这个问题，有的人是在认真地寻求答案，有的人只是不屑地表示怀疑。时间是这个问题的最好答案。真正有用的新东西最终会让那些不屑的怀疑论调显得很可笑。因为真正有用的新东西确实很少见，而且往往要经过很长很长的时间才能体现价值，所以，不经思考的怀疑总是廉价而且风险很小。

对此，我的具体答案是：太极语言可以让我们提高编程效率，以一种新的方式和思维来看待编程，完成更复杂的任务，编写更优化的程序，设定不同的优化方案。从下一个附录，我们还可以看到更多前景。

太极语言速查表(taijilang cheatsheet)

数据类型，字面常量，变量与常量的定义和赋值

- 数字(整数，浮点数，带指数部分浮点数，十六进制，二进制)：`1, 1.2, 1.2e-5, 0x2af, 0b101, 0B110`
- 字符串(转义、插补)：`'转义非插补', '''非转义非插补''', "转义插补(x) [x] {abs x} {.a:1.} $a \$a \()", """插补非转义(x) [x] {abs x} {.a:1.} $a \$a \()"""`
- 列表: `[1 2 3] [Math.abs 1, 2+3] [{Math.abs 1}, true or false]`
- 字典, 空字典(内层字典可以用缩进方式组织并省略`{. .}`): `{. a: 1; a => 2; 1+2: 5; .}, {}`
- 正则表达式：`/regexp string/`
- 变量 `var x = 1; const x = 1; x = 1; @@x = 1`

基本命令

- 模块头(第一行及缩进的随后各行)：`taiji language x.y ... \n some content ...`
- 注释 `//, ///, /* */, /** */, /., /.., /code block comment`
- 求值javascript：`eval javascriptCode`
- 求值太极语言代码块：`taiji expression`
- 打印：`print args...`
- `~ quotedExpression`

运算符和优先级

以下文本中，括号内的数字代表优先级，双引号内的字符串为运算符将被转换成为的中间表示所用的符号，括号内可能还有附带的功能说明或则简单示例。

- 前缀
    - #(200, 预处理), ##(5, 元编译), #/(5, 跨元), #-(5, 消元), #&(5, 元引用),
    - ~(5, "quote!", 引用号), `(5, "quasiquote!", 回引号), ^(160, "unquote!", 消引号), ^&(150, "unquote-splice", 切片消引号)
    - `yield`(20), `new`(140), `typeof`(140), `void`(140), `delete`(140)
    - . .(150, "..x", array1[..5]), . . .(150, "...x", 例如 array1[...5]), @@(210, 指示外层变量)
    - +(180，正号), -(180，符号)
    - !(180), ! !(180, !!x相当于!(!(x))), `not`(180, 相当于!), |%(180, 位反，相当于javascript的~)
    - ++(180, "++x"), --(180, "--x")
- 后缀: ++(180, "x++"), --(180, "x--")
- 中缀(二元运算符)：
    - ,(5)
    - ==(90, "==="), !=(90, "!=="), ===(90, "=="), !==(90, "!="), <(100), <=(100), >, >=(100), `in`(100), `instanceof`(100)
    - <<(110), >>(110), >>>(110), +(120), -(120), *(130), /(130), %(130)
    - . .(105, 例如：a..b, x[a..b]), . . .(105, 例如：a...b, x[a...b])
- 赋值运算符(优先级：20)

控制结构

- block!语句：`block! ...`
- if语句：`if test then thenBody [else elseBody]`
- switch语句: `switch test case valueList then caseBody else elseBody`
- cFor语句：`for init; test; step then body`
- for-in语句：`for item [index] in list then body`
- for-of语句：`for key [value] in dict then body`
- break语句：`break [label]`
- continue语句：`continue [label]`
- label语句: `label@ body`
- try语句: `try body catch e then catchBody else elseBody finally finallyBody`
- throw语句：`throw errorValue`

函数定义

- `(parameter[, ...]) -> functionBody`
- `(parameter[, ...]) => functionBody`

- `(parameter[, ...]) |-> functionBod`
- `(parameter[, ...]) |=> functionBody` 函数定义的参数形式：`x`, `x=defaultValue`, `@x`, `@x=defaultValue`, `x...`, `@x...`

类定义

- 类定义语法：`class name [extends superClass] [classBody]`
- 构造函数：`:: = (args...) -> body`
- 类成员函数: `::fn = (args) -> body`
- 类成员变量: `::x = value`
- 引用超类：`super`

模块

- 包含模块：`include! "path/to/module.tj" parserMethod`
- 导入模块：`import! #/x as x1 #x2, ..., [[from] "path/to/module.tj"] as m #m2`
- 导出变量：`export! #/x, #y, z = 1`
- 使用require语句和module, exports组织模块的导入导出（node.js，seajs等提供的机制）：`require("path/to/javascript/module") exports.x = someValue`

元算符

预处理(`#`), 元编译(`##`), 元赋值（`#=`), 跨元编译(`#/`), 跨元赋值（`#/=`), 跨元引用(`#&`), 跨元引用赋值(`#&=`)

解析算符

- 解析器属性引用：`?attr`
- 定制解析: `? clause then body`
- 解析子句: `?? clause`
- 解析式: `?/ expression`

与coffee-script相同点的比较

相同点

- 编译到javascript
- 缩进语法
- 支持将for，while等语句作为表达式使用
- 支持隐式变量申明。对新变量赋值自动产生局部变量声明。
- @代表this，::代表prototype
- -> 和 => 定义函数
- 支持省略参数(x, y..., z) ->
- 支持默认参数(a, b, x=1, y=2) ->
- 支持自动返回最后语句的值
- 支持字符串插补, 在coffee-script中是"...#{expression} ..."。

不同点

- taijilang超越javascript；coffee-script就是javascript。

  太极语言的丰富的特征使得它不再与javas语言cript具有一一对应的关系，有点类似于C和汇编， 在语言金字塔上位于一个更高的位置。而coffee-script的口号是：coffee-script is just javascript, 它追求的是和javascript的一一对应。

- 太极语言支持lisp风格的宏，coffee-script没有宏功能。
- 太极语言支持元语言，或者说编译时间计算，coffee-script没有元语言功能。
- 太极语言支持动态解析时间计算，动态控制语法。coffee-script没有此功能。
- 太极语言支持根据空格改变算符优先级，coffee-script不支持。

  象 1+2 *3+4* 这样的算式，在太极语言中解释为*(1+2)*(3+4), 而在其它编程语言中都是解释为1+(2*3)+4。太极语言的处理顺应了普通人的阅读直觉。

- 太极语言中任何构造都是表达式，即使是break，continue，return。而这三种构造在coffee-script中不能作为表达式使用。
- 太极语言可以声明常量，coffee-script不能。
- 太极语言中，函数作用域内的赋值总是产生函数内部局部变量。而coffee-script则如果外层词法作用域中已经有同名变量，则将不产生局部变量，赋值将针对外层变量有效。coffee-script比javascript进步一层，但是，在某种情况下还是会出现因为无意中覆盖了外层变量而导致难以觉察的编程错误。太极语言中如果需要对外层变量赋值，必须显式使用@@varName。
- 类似于livescript，太极语言可以通过在->和=>前加“|”抑制自动返回最后一语句的值。coffee-script不支持此特征。
- 太极语言中任何字符串都可以是多行字符串。coffee-script中'...'和"..."是单行字符串。即使写在多行也会去除换行符合并成单行。
- coffee-script只支持用#{...}向字符串插补表达式，太极语言有多种方法，并且比coffee-script的更简捷：$expression, {}, [], ()。
- coffeescript支持literature programming，太极语言不支持。
- coffee-script支持#引导的行注释和###...###块注释。太极语言支持//行注释，/.引导的缩进块注释，/.../注释和/注释代码块

相关资源

项目地址

太极语言github项目：github.com/taijiweb/taijilang

发布在npm的包taiji：npmjs.org/package/taiji

交流

google 邮件组：google groups: taijilang

google+帐号：taiji lang

QQ群: 太极语言 194928684

将来的计划

- 自举

用太极语言实现太极语言。实现了自举的太极语言将更有利于实现下属两项目标：以其它语言作为目标语言，使得太极语言可以将元语言和目标语言配置为不同的语言。

- 各种语言作为目标语言

当前太极语言以javascript作为目标语言。可以考虑以其它语言作为目标语言。相对而言，动态语言更容易实现，比如python，ruby，perl，php等等。静态语言技术上困难要更多一些，但是也不存在无法克服的障碍。
Swift，Go，C++/C，Objective C等等都可以作为目标语言。太极语言的设计特点是有利于这项工作的，因为它的中间表示是通用的json和列表结构，这对于简化实现代码很有帮助。

- 元语言和目标语言配置为不同的语言

现在太极语言是以同一种语言作为元语言（编译时间语言）和目标语言（运行时间语言），具体说是javascript。这种方法当然有很多优点。但是，采用不同的语言作为元语言和目标语言也是可能的，也可能存在某些方面的优势。

- 用某种更快的语言重新实现太极语言，比如go或者C语言

太极语言现在的实现还没有充分考虑自身的优化。特别是解析过程，当前的手写解析器实现还存在很多的优化空间。另一个优化的途径是从动态语言切换到静态的编译型语言，比如用go或者C语言。也许用go是一个不错的选择，因为go语言开发效率高，而且也具有很高的运行时间效率。

- 基于太极语言的模板语言，比如html模板，xml模板，css预处理语言等

- 太极语言web框架

我启动太极语言项目的直接触发点是这样的：在实现基于web的自然语言编程系统的时候，我用到了angularjs，理解到它双向数据绑定的强大。因此，我产生了这样的想法：能否将这种能力扩展到服务器端，建立一个前后端统一的框架。由此，我开始查找相关项目和资料。我很熟悉jade模板语言，但是它无法提供我希望的功能。我探索其它的github项目，偶然遇到了lispyscript这一个项目。在我以前实现dao系统的基础上，lispyscript再次触动了我的灵感。

太极语言产生于web开发的经历中，太极语言的功能也让它很适合成为新一代web开发的工具，应该有可能用它实现我提到的上述web开发框架。

- 用太极语言实现更优化的库。比如underscore或lodash。

目录