



下载APP



27 | 增加更丰富的类型第2步：如何支持字符串？

2021-10-15 宫文学

《手把手带你写一门编程语言》

课程介绍 >

**讲述：宫文学**

时长 21:37 大小 19.80M



你好，我是宫文学。

今天我们继续来丰富我们语言的类型体系，让它能够支持字符串。字符串是我们在程序里最常用的数据类型之一。每一门高级语言，都需要对字符串类型的数据提供充分的支持。

但是，跟我们前面讨论过的整型和浮点型数据不同，在 CPU 层面并没有直接支持字符串运算的指令。所以，相比我们前面讲过的这两类数据类型，要让语言支持字符串，我们需要做更多的工作才可以。



那么，在这一节课里，我们就看看要支持字符串类型的话，我们语言需要做哪些工作。在这个过程中，我们会接触到对象内存布局、内置函数（Intrinsics），以及字符串、字面量的表示等知识点。

首先，我们来分析一下，在这种情况下，我们的编译器和运行时需要完成哪些任务，然后我们再次完成它们就可以了。

任务分析

你可以看到，在一些强调易用性的脚本语言里，字符串常常作为内置的数据类型，并拥有更高优先级的支持。比如，在 JavaScript 里，你可以用 + 号连接字符串，并且，其他数据类型和字符串连接时，也会自动转换成字符串。这比在 Java、C 等语言使用字符串更方便。

为了支持字符串类型，实现最基础的字符串操作功能，我们就需要解决下面这几个技术问题：

第一，如何在语言内部表示一个字符串？

在像 JavaScript、Java、Go 和 C# 这样的高级语言中，所有的数据类型可以分为两大类。一类是 CPU 在底层就支持的，就像整数和浮点数，我们一般叫做基础类型（Primitive Type）或者叫做值类型（Value Type）。

这些类型可以直接表示为指令的操作数，在赋值、传参的时候，也是直接传递值。比如，当我们声明一个 number 类型的变量时，我们在语言内部用 CPU 支持的双精度浮点数来存储变量的值就可以了。当给变量赋值的时候，我们也是把这个 double 值用 mov 指令拷贝过去就行。

但对于基础类型之外的复杂数据类型来说，它们并不能受到 CPU 指令级别的直接支持。所以，我们就需要设计，当我们声明一个字符串，以及给字符串赋值的时候，它对应的确切操作是什么。

那么计算机语言的设计者，通常会怎么做呢？我们要**把这些复杂数据类型在内部实现成一个内存对象**，而变量赋值、传参这样的操作，实际上传递的是对象的引用，对象引用能够转换为对象的内存地址。

所以，**从今天这节课开始，我们也将正式支持对象机制**。其实，string 也好，数组也好，还是后面的自定义类型也好，它们在内存里都是一个对象。当进行赋值操作的时候，传递

的都是对象的引用。那这个时候，我们就需要设计对象的内存结构，以及确定什么是对象的引用。

第二，在运行时里提供一些内置函数，用于支持字符串的基本功能。

为了支持字符串类型的数据，我们要能够支持字符串对象的创建、字符串拼接、其他类型的数据转为字符串，还有字符串的比较，等等功能。这些功能是以内置函数（intrinsics）的形式来实现的。编译成汇编代码的时候，我们要调用这些内置函数来完成相应的功能。

第三，我们还要处理一些编译器后端的工作。

在编译器的后端方面，我们要能生成对字符串进行访问和处理的汇编代码。这里面的重点就是，我们要知道如何在汇编代码里表示字符串字面量，以及如何获取字符串字面量的地址。

好了，任务安排妥当了，我们开始行动吧。首先我们来看第一个任务，如何在语言内部表示一个字符串类型的数据。

如何表示一个字符串

这个问题其实又包含三个子问题：字符编码的问题、string 对象的内存布局，以及如何来表示一个对象的引用。

首先，我们看看字符的编码问题。

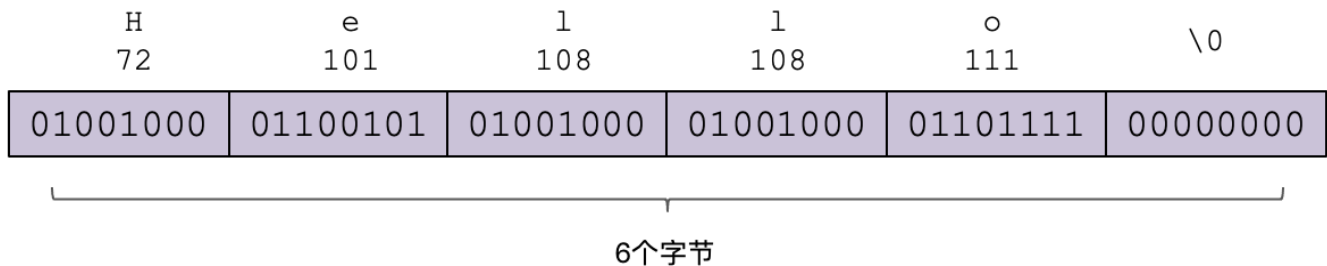
我们知道，CPU 只知道 0101 这些值，并不知道 abcd 这些概念。实际上，是我们人类给每个字符编了码，让 CPU 来理解的。比如规定 65 代表大写字母 a，97 代表小写字母 a，而 48 代表字符 0，这就是广为使用的 ASCII 编码标准。但要支持像中文这么多的字符，ASCII 标准还不够用，就需要 Unicode 这样的编码标准。

不过，在我们当前的实现中，我们还是先做一些简化吧，先不支持 Unicode，只支持 ASCII 码就好了。这样，在内存里，我们只需要用一个字节来表示字符就行了，这跟 C 语言是一样的。至于 Unicode，我们后面再支持。毕竟我们的语言 PlayScript，是一个开源项目，会继续扩展功能。你也可以走在我前面，自己先去思考并实现一下怎么支持 Unicode 编码。

第二，我们看看 string 的内存布局。

如何在内存里表示一个字符串呢？

我们站在巨人的肩膀上，看看 C 语言是怎么做的。在 C 语言中，字符串在内存里就相当于一个 char 的数组，这个数组以 0 结尾。所以，“Hello”在内存里大概是这样保存的，加起来一共是 6 个字节：



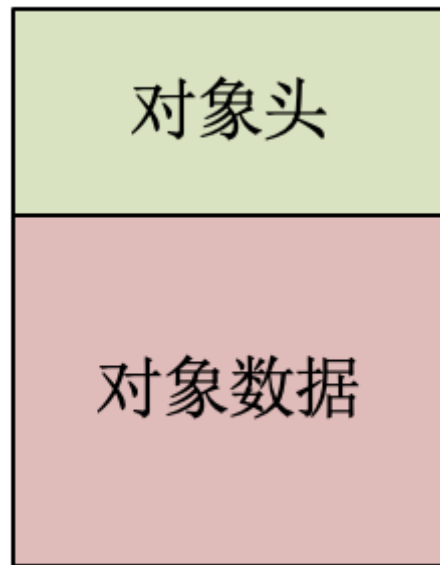
我们也可以借鉴 C 语言的做法，用一个数组来表示字符串。不过，C 语言需要程序员自己去处理字符串使用的内存：要么通过声明一个数组，在栈里申请内存；要么在堆里申请一块内存，使用完毕以后再手工释放掉。

而 JavaScript 是不需要程序员来手工管理内存的，而是采用了自动内存管理机制。自动内存管理机制管理的是一个内存对象。当对象不再被使用以后，就可以被回收。

那么我们的设计，也必须实现自动的内存管理，因为 TypeScript 并没有底层的内存管理能力。

说到内存对象，我们还有一个设计目标，就是在语言内部，对各种类型的对象都有统一的管理机制，包括统一的内存管理机制、统一的运行时类型查询机制等等。这样，才能铺垫好 TypeScript 对象化的基础，并在后面实现更丰富的语言特性。所以，我们就需要对如何在内存里表示一个对象进行一下设计。

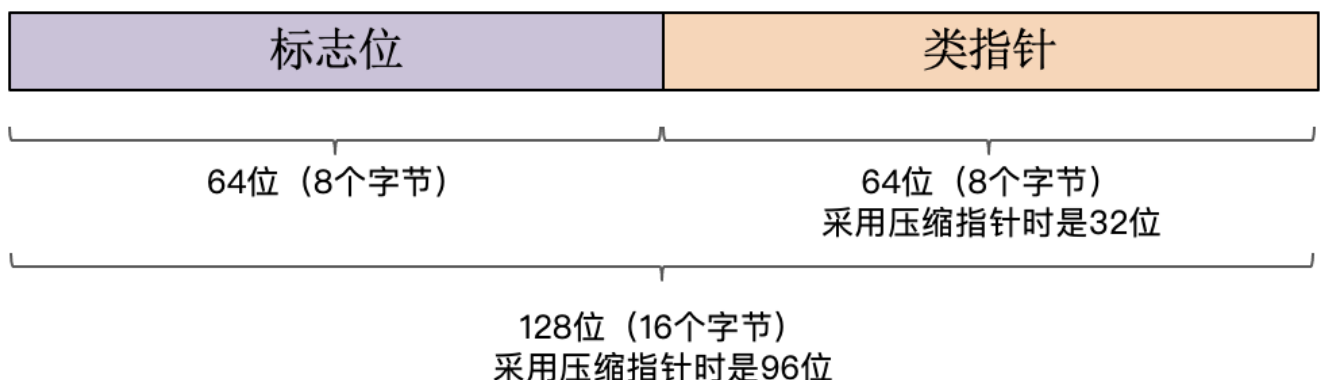
这方面，我们又可以参考一下其他语言是怎么做的。比如，在 Java 等语言里，对象都有一些统一的内存布局设计。其典型特征，就是每个对象都有一个固定的对象头，对象头之后才是对象的实际数据。



对象头里面保存了一些信息，用来对这个对象进行管理。进行哪些管理呢？首先是自动内存管理。对象头里有一些标志位，是用于垃圾收集程序的。比如，通过算法来标记某个对象是否是垃圾。我们在后面会具体实现一个垃圾收集算法，那个时候就会用到这些标志位。

标志位还有一个用途就是并发管理。你可以用一些特殊的指令，锁住一个对象，使得该对象在同一时间只可以被一个线程访问。在锁住对象的时候，也要在对象头做标识。此外，对象头里还有引用了类的定义，这样我们就可以在运行时知道这个对象属于哪个类，甚至通过反射等元编程机制去动态地调用对象的方法。

我们可以参考一下 Java 对象头的设计。它包含类指针和标志位两个部分。类指针指向类定义的地址。标志位就是内部分割成多个部分，用来存放与锁、垃圾收集等标记，还会存放对象的哈希值。



当然，其他语言的对象，也都有类似的内存布局设计。我在 [《编译原理实战课》](#) 中，对 Java、Python 和 Julia 等语言的对象内存布局都做了讨论，如果你感兴趣可以去看看。

参考这些设计，我们也可以做出自己的设计。在 PlayScript 中，我们首先设计一个 Object 对象，里面有一个标志位的字段和一个指向对象的类定义的指针。我们后面再探讨它们的用途。

[复制代码](#)

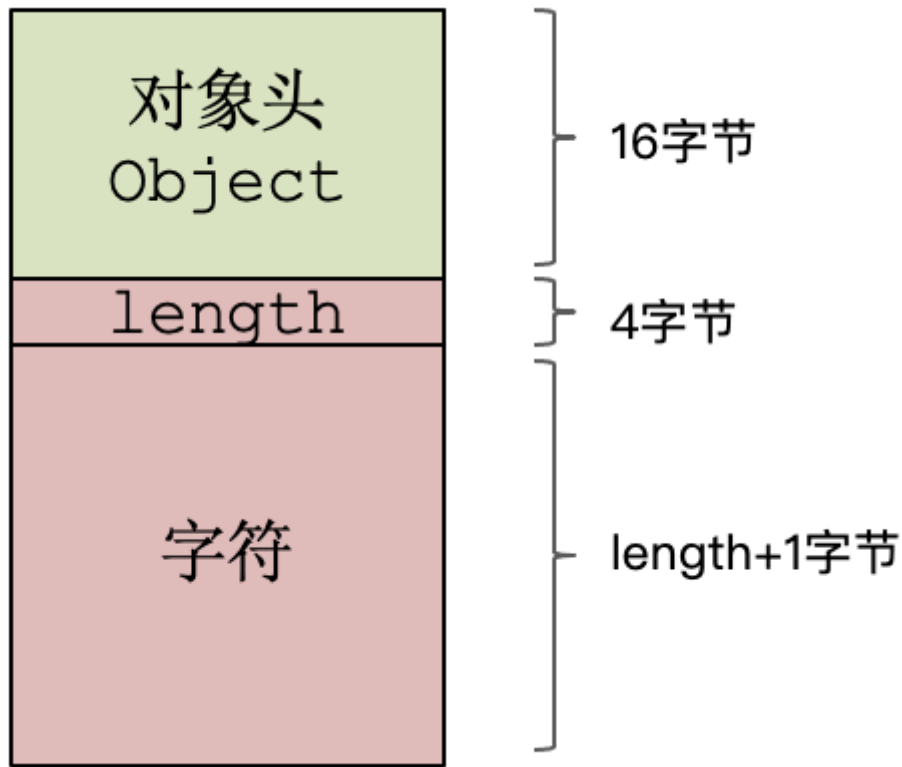
```
1 //所有对象的对象头。目前的设计占用16个字节。
2 typedef struct _Object{
3     //指向类的指针
4     struct _Object * ptrKlass;
5
6     //与并发、垃圾收集有关的标志位。
7     unsigned long flags;
8 }Object;
```

所有对象都要继承自 Object 对象，字符串对象也不例外。我们把字符串对象叫做 PlayString，其数据结构中包含了字符串的长度。真实的字符串数据是接在 PlayString 之后的。而且，我们基于 PlayString 的地址，就能计算出字符串的存储位置，所以并不需要一个单独的指针，这样也就节省了内存空间。

[复制代码](#)

```
1 typedef struct _PlayString{
2     Object object;
3     //字符串的长度
4     size_t length;
5
6     //后面跟以0结尾的字符串，以便复用C语言的一些功能。实际占用内存是length+1。
7     //我们不需要保存这个指针，只需要在PlayString对象地址的基础上增加一个偏移量就行。
8     //char* data;
9 }PlayString;
```

采用这个结构后，实际上 PlayString 的内存布局如下。对象头占 16 个字节，字符串长度占 4 个字节，其余的才是字符串数据，占用空间的大小是字符串的长度再加 1 个字节：



不过，在这里，我们还有一个技术细节需要做一下决策。

C 语言中是以 0 结尾的数组来表示一个字符串的。所以，在 C 语言中，我们每次为字符串申请内存的时候，都要多申请一个字节，用于存放字符串的结尾标志，也就是 0。那么，在我们自己实现的语言里，是否需要也多申请一个字节的空間，也在字符串后面放一个 0 呢？

本质上是不需要的。因为我们已经用了一个字段来表示字符串的长度。不过，像打印字符串这种功能，我们想直接使用 C 语言的标准函数来实现，所以还是决定采用 C 语言存储字符串的格式，尽管这可能会浪费一点存储空间。当然你也可以不使用这个设计，但在实现字符串输出等功能的时候，就需要去做额外的工作了。

好了，我们已经设计出了字符串对象的内存布局。那么再看看最后一个问题，如何表示一个对象引用？

我们前面也说了，当我们给 string 或者其他以对象格式保存的变量赋值的时候，传递的实际是个对象引用。那这个对象引用到底是什么呢？

其实我们知道，不管对象引用怎么设计，都必须能够通过对象引用获得对象的地址，以便操作对象内部的数据。不过在具体实现的时候，理论上有很多种可能性。

比如，你可以给所有在系统中创建的对象编号，在内存里通过某个数据结构来保存对象编号和内存地址的关系，再通过编号来查找出对象的内存地址。

不过，大多数语言，就是直接使用内存地址来作为对象引用的，因为这样可以用最快的速度访问对象的内容，免去了计算对象地址的额外工作量。


我们的实现，也采用同样的设计，即直接拿内存地址作为对象的引用。在 64 位模式下，内存地址是 64 位的，那这个对象引用就可以用一个 64 位的长整型表示。

不过，直接以内存地址作为对象引用，有一个潜在的、需要考虑的技术问题，就是**垃圾收集算法**。有的垃圾收集算法会在内存里移动对象，从一个内存区域拷贝到另一个区域。这个时候，对象的内存地址就会改变，也就是对象引用会改变。这个时候，就需要我们把所有指向这个对象的引用都修改一遍才行。

好了，关于如何在语言内部表示字符串对象，我们就讨论到这里。接下来，我们看看运行时和内置函数方面的工作。

运行时和内置函数

为了支持对 string 对象进行处理，我们需要实现几个内置函数，分别用于创建字符串对象、字符串连接，以及从 number 类型转成字符串类型：

 复制代码

```
1 //创建指定长度的字符串
2 PlayString* string_create_by_length(size_t length);
3 //连接字符串
4 PlayString* string_concat(PlayString* str1, PlayString* str2);
5 //数字型转字符串
6 PlayString* double_to_string(double num);
```

在这些创建 PlayString 对象的函数代码里，你可以更加清楚地看到内存是如何布局的。总的来说，我们是一次性地申请了 PlayString 对象需要的所有内存，这样可以减少内存申请的次数，提高内存访问的性能。我们在实现 C 语言版本的虚拟机的时候，已经体会到这个技术点的重要性。而真正的字符串数据是接在 PlayString 对象之后的。

你可能也注意到了，我不止一次地提到了内置函数这个概念。那什么是内置函数呢？跟 `println` 这样的函数有什么区别呢？`println` 不也是我们在语言里预先提供的函数吗？

其实，我们刚才提到的这几个函数，包括创建字符串对象的函数、字符串连接的函数，跟 `println` 这样的函数是不一样的。哪里不一样呢？`println` 是可以由程序员直接使用的。而像刚才那几个函数，一般只是由编译器和运行时所使用的，程序员一般不能直接使用，所以它们就被称为“内置函数”。当我们在程序里使用 `+` 号来连接两个字符串的时候，编译器会自动去调用字符串连接的内置函数来实现这个功能，这个过程对程序员是透明的。

我们这里提到的内置函数，在一些编译技术有关的文章里通常叫做 `Intrinsics`（在 `GCC` 里会被叫做 `Built-in`）。可能直接叫它的英文名称，更不容易产生混淆。有些 `Intrinsics` 是直接汇编代码编写的，这样性能更高，或者能使用一些特别的指令。

这些内置函数是一个语言的运行时的一部分。如果把程序静态编译，那么它们会以动态库的形式存在，或者直接编译进可执行文件里。如果采用虚拟机的方式运行，那么它们也是虚拟机的组成部分。

好了，实现完了运行时和内置函数，我们就可以生成汇编代码，让语言来支持字符串特性了！

修改编译器后端

在前一节课，我们实现了对浮点数的支持以后，你可能已经在修改编译器后端方面取得了一些经验，这些工作都是有共性的。

在今天这节课，我们主要关注两个技术点：**一个是在汇编代码中保存字符串字面量，一个是在程序里访问这些字符串的地址。**

首先看看字符串字面量的保存。

我们在程序里经常会用到字符串字面量，比如需要打印输出的提示信息，等等。那在汇编代码里是如何表达和使用它们的呢？

作为对比，我们回忆一下，我们在上节课曾经处理过浮点数字面量。它们不能直接作为立即数使用，而是保存在一个单独的文本段里，然后通过标签来访问它们的内存地址。

对字符串字面量的处理也很类似。你可以用 C 语言写一个很简单的字符串处理程序，看看生成的汇编代码是什么样子的。

我这里写了一个 C 语言的示例程序，它只简单地实现了一个字符串打印的功能：

[复制代码](#)

```
1 #include <stdio.h>
2 int main(){
3     printf("Hello PlayScript!\n");
4 }
```

这个例子编译后生成的汇编代码如下：

```
_main:                                     ## @main
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    L_.str(%rip), %rdi             ← 获取字符串的地址。
    movb    $0, %al
    callq   _printf
    xorl    %ecx, %ecx
    movl    %eax, -4(%rbp)                ## 4-byte Spill
    movl    %ecx, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc

                                     ## -- End function
.section    __TEXT,__cstring,cstring_literals ← 以C语言的格式保存的字符串，
L_.str:     ← 标签                                     也就是说，它们也以0结尾。
                                     ## @.str
    .asciz  "Hello PlayScript!\n"
```

在这段代码里，你会看到一个单独的文本段，用来保存 C 语言格式的字符串。

[复制代码](#)

```
1 .section .text, "cstring,cstring_literals"
```

所以，采用 C 语言格式，就意味着每个字符串在内存中都是以 0 结尾的，这跟我们的设计是一样的。

接下来，你会看到两行。第一行是一个标签，用来引用这个字符串常量。第二行用了一个.asciz 伪指令，后面带着字符串具体的值。

```
1 L_.str:                                ## @.str
2     .asciz "Hello PlayScript!"
```

[复制代码](#)

到这里，我们就知道了。**字符串字面量和上一节的浮点数字面量一样，都是保存在文本段的。在编译之后，它们就会进入到可执行文件中。**

知道了这个原理后，其实你就可以搞点黑客小技巧了。比如，你可以用工具修改二进制文件里的字符串常量，篡改程序里显示的信息。

00003f50	55 48 89 e5 48 83 ec 10 48 8d 3d 37 00 00 00 b0	UH..H...H.=7....
00003f60	00 e8 0e 00 00 00 31 c9 89 45 fc 89 c8 48 83 c41..E...H..
00003f70	10 5d c3 90 ff 25 86 40 00 00 00 00 4c 8d 1d 85	.]...%.@....L...
00003f80	40 00 00 41 53 ff 25 75 00 00 00 90 68 00 00 00	@..AS.%u....h...
00003f90	00 e9 e6 ff ff ff 48 65 6c 6c 6f 20 50 6c 61 79Hello Play
00003fa0	53 63 72 69 70 74 21 0a 00 00 00 00 01 00 00 00	Script!.....
00003fb0	1c 00 00 00 00 00 00 00 1c 00 00 00 00 00 00 00
00003fc0	1c 00 00 00 02 00 00 00 50 3f 00 00 34 00 00 00P?...4...
00003fd0	34 00 00 00 74 3f 00 00 00 00 00 00 34 00 00 00	4...t?.....4...
00003fe0	03 00 00 00 0c 00 01 00 10 00 01 00 00 00 00 00
00003ff0	00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00
00004000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

同理，你还可以修改可执行程序中数据区里的一些数据。很多玩游戏的同学会使用这个手段，来修改自己的游戏参数，让自己的账号具备“超能力”。

从这里你也可以看到，**设计一门计算机语言的时候，其实你还要考虑程序的安全因素。**从这个角度看，现代很多语言都采用托管的方式运行，而不是直接编译成在操作系统上运行的可执行程序，这在安全方面是有一些优势的。不过安全性就是另一个话题了，我们不多展开。从这里你能再次体会到，了解一些底层实现机制，能帮助你理解很多技术问题的理解更深刻。

回到原来的话题。现在，我们已经知道如何来存储字符串字面量了。那么如何在汇编代码里访问字符串呢？

要想访问字符串，我们必须获取字符串的地址。在上面的汇编代码里，你会看到一个 `leaq` 指令，也就是 64 位的 `lea` 执行。

`lea` 是 Load Effective Address 的意思。这个指令能够进行内存地址的计算，并把计算结果保存到寄存器里。我们之前就曾经讨论过如何用 `lea` 指令来做加法运算。但它真正的用途其实是计算内存地址的。在这里，我们可以用 `lea` 指令取得字符串字面量的地址，然后把这个地址作为参数传递给打印函数。

对于我们的语言来说，只要我们获取了字符串的地址，接下来就可以做很多其他的事情了，包括：创建我们内置的 `PlayString` 对象，以及将两个 `PlayString` 做拼接、生成新的字符串对象，等等。

懂得原理之后，我们再去修改生成汇编的程序。你可以重点看一下对 `StringLiteral` 对象和 `+` 号运算符的处理。

在处理 `StringLiteral` 时，我们首先把字符串保存到常量区，然后生成一个 `PlayString` 对象，并把对象的地址作为一个操作数返回。

[复制代码](#)

```
1 visitStringLiteral(stringLiteral:StringLiteral):any{
2     //加到常数表里
3     if (this.asmModule != null){
4         //把字符串字面量保存到模块中。基于这些字面量可以生成汇编代码中的一个文本段。
5         let strIndex = this.asmModule.stringConsts.indexOf(stringLiteral.value
6         if( strIndex == -1){
7             this.asmModule.stringConsts.push(stringLiteral.value as string);
8             strIndex = this.asmModule.stringConsts.length - 1;
9         }
10
11         //新申请一个临时变量
12         let tempVar = this.allocateTempVar(getRegisterKind(stringLiteral.theTy
13
14         //用leaq指令把字符串字面量加载到一个变量（虚拟寄存器）
15         let inst = new Inst_2(OpCodes.leaq, new Operand(OperandKind.stringConst,
16         this.getCurrentBB().insts.push(inst);
17
18         //调用一个内置函数来创建PlayString
19         let args:Operand[] = [];
```

```
20     args.push(tempVar);
21
22     //调用内置函数，返回值是PlayString对象的地址
23     return this.callIntrinsics("string_create_by_str", args);
24 }
25 }
```

在处理 + 号时，如果类型是 String，那么就调用内置函数实现字符串的连接。

[复制代码](#)

```
1  switch(bi.op){
2      case Op.Plus: //'+'
3          if (bi.theType === SysTypes.String){ //字符串加
4              let args:Oprand[] = [];
5              args.push(left);
6              args.push(right);
7              this.callIntrinsics("string_concat", args);
8          }
9      ...
10 }
```

现在，我们的语言就能进行基本的字符串处理了。你可以写几个测试程序，并编译成可执行文件试试看，比如下面这个示例程序：

[复制代码](#)

```
1  let s1 = "Hello";
2  let s2:string;
3  s2 = " PlayScript!";
4  let s3 = s1 + s2;
5  println(s3);
```

课程小结

好了，今天的内容就这些了。为了实现对字符串的支持，你要掌握下面这些知识点：

首先，你要知道如何表示字符串。这里面涉及三个技术点。第一个技术点，是如何给字符编码，简单的就是采用 ASCII 编码，复杂一点，就要支持 unicode 这样的编码。第二个技术点，是如何把字符串作为对象在内存中管理。为此，我们设计了一个标准的内存结构，每个对象都有标准的对象头。这个对象头会被用于内存管理、并发管理、元编程等功能。

我们后面几节课的数组对象、自定义类等，也都要遵循这个内存结构。第三个技术点，是如何表达对象的引用。在 PlayScript 里，我们就直接用对象的内存地址就行了。

第二，为了支持字符串对象的创建、连接等操作，我们需要实现几个内置函数。内置函数叫做 Intrinsics，是在语言内部被使用的函数，一般不由程序员直接使用。

第三，我们学习了如何在汇编代码中保存和访问字符串常量。字符串常量是保存在程序的文本段里，可以用标签访问。我们可以用 leaq 指令和 rip 寄存器来计算字符串常量的地址。

最后，其实针对字符串处理，我们还有两个遗留问题：

一个问题是，我们在程序里只为字符串对象申请了内存，从来没有释放过，这显然会造成内存泄漏。这个问题我们放在内存管理的部分再去解决。

第二个问题是我们目前只支持 ASCII 编码，未来我们会扩展到支持 Unicode 编码。

思考题

对象的内存布局设计，是计算机语言的一个重要设计决策。在这节课，我们讨论了 Java 的对象布局设计，作为我们实现的参考。在这里，我想问一下，你熟悉的语言，它的对象内存布局有什么特点？有哪些你喜欢或者不喜欢的地方？欢迎在留言区分享你的观点。

欢迎你把这节课分享给更多感兴趣的朋友，我是宫文学，我们下节课见。

资源链接

1. 这节课的 [🔗 示例代码目录](#)；
2. 对象头有关的代码，在 [🔗 rt/object.h](#)；
3. PlayString 有关的内置函数，在 [🔗 rt/string.h](#)、[🔗 rt/string.c](#)；
4. 对编译器后端的修改，仍然要查看 [🔗 asm_x86-64_d.ts](#)。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 26 | 增强更丰富的类型第1步：如何支持浮点数？

4 周年庆限定

299元随心畅学卡

五门课程任你选，总价值高达千元

畅学卡
¥299

4周年

超值拿下

精选留言 (1)

写留言



奋斗的蜗牛

2021-10-15

太棒了，这些知识点的讲解真是无价之宝
展开

