字节码

意图

将行为编码为虚拟机器上的指令，赋予其数据的灵活性。

动机

制作游戏也许很有趣，但绝不容易。 现代游戏的代码库很是庞杂。 主机厂商和应用市场有严格的质量要求， 小小的崩溃漏洞就能阻止游戏发售。

> 我曾参与制作有六百万行C++代码的游戏。作为对比，控制好奇号火星探测器的软件还没有其一半大小。

与此同时，我们希望榨干平台的每一点性能。 游戏对硬件发展的推动首屈一指，只有坚持不懈地优化才能跟上竞争。

为了保证稳定和性能的需求，我们使用如C++这样的重量级的编程语言，它既有能兼容多数硬件的底层表达能力，又拥有防止漏洞的强类型系统。

我们对自己的专业技能充满自信，但其亦有代价。 专业程序员需要多年的训练，之后又要对抗代码规模的增长。 构建大型游戏的时间长度可以在“喝杯咖啡”和“烤咖啡豆，手磨咖啡豆，弄杯espresso，打奶泡，在拿铁咖啡里拉花。”之间变动。

除开这些挑战，游戏还多了个苛刻的限制：“乐趣”。 玩家需要仔细权衡过的新奇体验。 这需要不断的迭代，但是如果每个调整都需要让工程师修改底层代码，然后等待漫长的编译结束，那就毁掉了创作流程。

法术战斗！

假设我们在完成一个基于法术的格斗游戏。 两个敌对的巫师互相丢法术，直到分出胜负。 我们可以将这些法术都定义在代码中，但这就意味着每次修改法术都会牵扯到工程师。 当设计者想修改几个数字感觉一下效果，就要重新编译整个工程，重启，然后进入战斗。

像现在的许多游戏一样，我们也需要在发售之后更新游戏，修复漏洞或是添加新内容。 如果所有法术都是硬编码的，那么每次修改都意味着要给游戏的可执行文件打补丁。

再扯远一点，假设我们还想支持模组。我们想让玩家创造自己的法术。 如果这些法术都是硬编码的，那就意味着每个模组制造者都得拥有编译游戏的整套工具链， 我们也就不得不开放源代码，如果他们的自创法术上有个漏洞，那么就会把其他人的游戏也搞崩溃。

数据 > 代码

很明显实现引擎的编程语言不是个好选择。 我们需要将法术放在与游戏核心隔绝的沙箱中。 我们想要它们易于修改，易于加载，并与其他可执行部分相隔离。

我不知道你怎么想，但这听上去让我觉得有点像是数据。 如果能在分离的数据文件中定义行为，游戏引擎还能加载并“执行”它们，就可以实现所有目标。

这里需要指出“执行”对于数据的意思。如何让文件中的数据表示为行为呢？这里有几种方式。 与解释器模式[GoF]对比着看会好理解些。

解释器模式

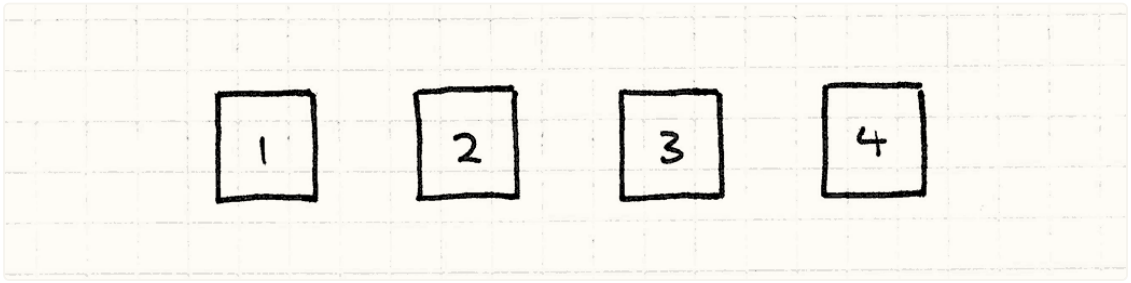
关于这个模式我就能写整整一章，但是有四个家伙的工作早涵盖了这一切， 所以，这里给一些简短的介绍。

它源于一种你想要执行的语言——想想编程语言。

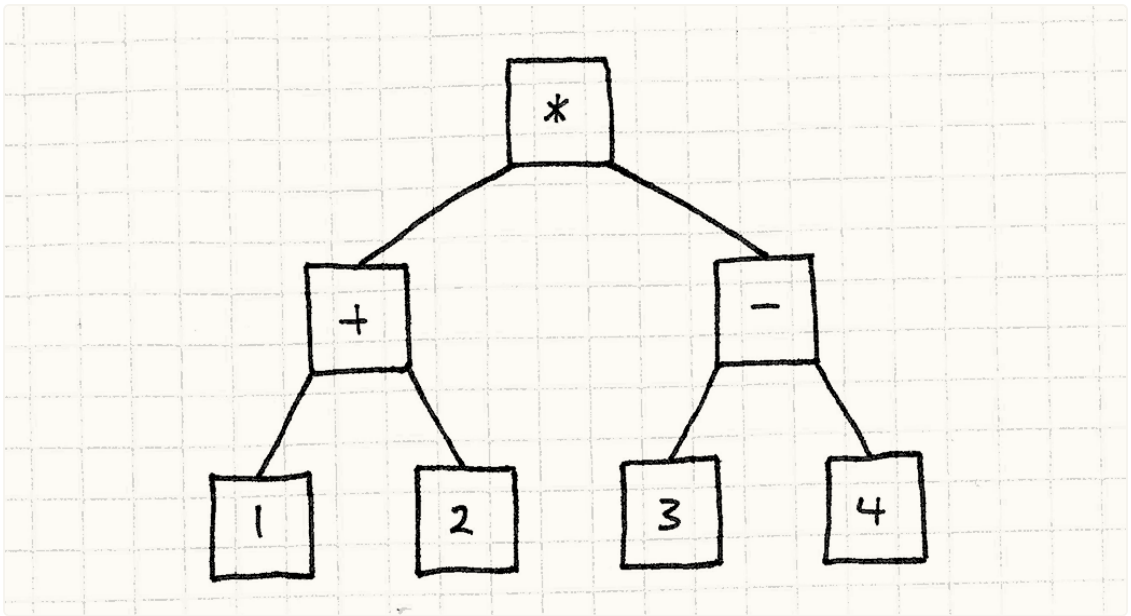
比如，它支持这样的算术表达式

```
(1 + 2) * (3 - 4)
```

然后，把每块表达式，每条语言规则，都装到对象中去。数字字面量都变成对象：



简单地说，它们在原始值上做了个小封装。 运算符也是对象，它们拥有操作数的引用。 如果你考虑了括号和优先级，那么表达式就魔术般变成这样的小树：



> 这里的“魔术”是什么？很简单——**语法分析**。 语法分析器接受一串字符作为输入，将其转为**抽象语法树**，即一个包含了表示文本语法结构的对象集合。
>
> 完成这个你就得到了半个编译器。

解释器模式与创建这棵树无关，它只关于执行这棵树。 它工作的方式非常巧妙。树中的每个对象都是表达式或子表达式。 用真正面向对象的方式描述，我们会让表达式自己对自己

求值。

首先，我们定义所有表达式都实现的基本接口：

```
class Expression
{
public:
  virtual ~Expression() {}
  virtual double evaluate() = 0;
};
```

然后，为我们语言中的每种语法定义一个实现这个接口的类。最简单的是数字：

```
class NumberExpression : public Expression
{
public:
  NumberExpression(double value)
  : value_(value)
  {}

  virtual double evaluate()
  {
    return value_;
  }

private:
  double value_;
};
```

一个数字表达式就等于它的值。加法和乘法有点复杂，因为它们包含子表达式。在一个表达式计算自己的值之前，必须先递归地计算其子表达式的值。像这样：

```
class AdditionExpression : public Expression
{
public:
  AdditionExpression(Expression* left, Expression* right)
  : left_(left),
    right_(right)
  {}

  virtual double evaluate()
  {
    // 计算操作数
    double left = left_->evaluate();
    double right = right_->evaluate();

    // 把它们加起来
    return left + right;
  }

private:
  Expression* left_;
  Expression* right_;
};
```

> 你肯定能想明白乘法的实现是什么样的。

很优雅对吧？只需几个简单的类，现在我们可以表示和计算任意复杂的算术表达式。 只需要创建正确的对象，并正确地连起来。

> Ruby用了这种实现方法差不多15年。在1.9版本，他们转换到了本章所介绍的字节码。看看我给你节省了多少时间！

这是个优美、简单的模式，但它有一些问题。 看看插图，看到了什么？大量的小盒子，以及它们之间大量的箭头。 代码被表示为小物体组成的巨大分形树。这会带来些令人不快的后果：

- 从磁盘上加载它需要实例化并连接成吨的小对象。
- 这些对象和它们之间的指针会占据大量的内存。在32位机上，那个小的算术表达式至少要占据68字节，这还没考虑内存对其呢。

  > 如果你想自己算算，别忘了算上虚函数表指针。

- 顺着那些指针遍历子表达式是对数据缓存的谋杀。同时，虚函数调用是对指令缓存的屠戮。

  > 参见数据局部性一章以了解什么是缓存以及它是如何影响游戏性能的。

将这些拼到一起，怎么念？S-L-O-W。 这就是为什么大多数广泛应用的编程语言不基于解释器模式： 太慢了，也太消耗内存了。

虚拟的机器码

想想我们的游戏。玩家电脑在运行游戏时并不会遍历一堆C++语法结构树。 我们提前将其编译成了机器码，CPU基于机器码运行。机器码有什么好处呢？

- 密集。 它是一块坚实连续的二进制数据块，没有一位被浪费。
- 线性。 指令被打成包，一条接一条地执行。不会在内存里到处乱跳（除非你的控制流代码真真这么干了）。
- 底层。 每条指令都做一件小事，有趣的行为从组合中诞生。
- 速度快。 综合所有这些条件（当然，也包括它直接由硬件实现这一事实），机器码跑得跟风一样快。

这听起来很好，但我们不希望真的用机器代码来写咒语。 让玩家提供游戏运行时的机器码简直是在自找麻烦。我们需要的是机器代码的性能和解释器模式的安全的折中。

如果不是加载机器码并直接执行，而是定义自己的虚拟机器码呢？ 然后，在游戏中写个小模拟器。 这与机器码类似——密集，线性，相对底层——但也由游戏直接掌控，所以可以放心地将其放入沙箱。

> 这就是为什么很多游戏主机和iOS不允许程序在运行时生成并加载机器码。 这是一种拖累，因为最快的编程语言实现就是那么做的。 它们包含了“即时（just-in-time）”编译器，或者JIT，在运行时将语言翻译成优化的机器码。

我们将小模拟器称为虚拟机（或简称“VM”），它运行的二进制机器码叫做字节码。 它有数据的灵活性和易用性，但比解释器模式性能更好。

> 在程序语言编程圈，“虚拟机”和“解释器”是同义词，我在这里交替使用。 当指代GoF的解释器模式，我会加上“模式”来表明区别。

这听起来有点吓人。 这章其余部分的目标是为了展示一下，如果把功能列表缩减下来，它实际上相当通俗易懂。 即使最终没有使用这个模式，你也至少可以对Lua和其他使用了这一模式的语言有个更好的理解。

模式

指令集 定义了可执行的底层操作。 一系列的指令被编码为字节序列。 虚拟机 使用 中间值栈 依次执行这些指令。 通过组合指令，可以定义复杂的高层行为。

何时使用

这是本书中最复杂的模式，无法轻易地加入游戏中。这个模式应当用在你有许多行为需要定义，而游戏实现语言因为如下原因不适用时：

- 过于底层，繁琐易错。
- 编译慢或者其他工具因素导致迭代缓慢。
- 安全性依赖编程者。如果想保证行为不会破坏游戏，你需要将其与代码的其他部分隔开。

当然，该列表描述了一堆特性。谁不希望有更快的迭代循环和更多的安全性？ 然而，世上没有免费的午餐。字节码比本地代码慢，所以不适合引擎的性能攸关的部分。

记住

创建自己的语言或者建立系统中的系统是很有趣的。 我在这里做的是小演示，但在现实项目中，这些东西会像藤蔓一样蔓延。

> 对我来说，游戏开发也正因此而有趣。 不管哪种情况，我都创建了虚拟空间让他人游玩。

每当我看到有人定义小语言或脚本系统，他们都说，“别担心，它很小。”于是，不可避免地，他们增加更多小功能，直到完成了一个完整的语言。 除了，和其它语言不同，它是定制的并拥有棚户区的建筑风格。

> 例如每一种模板语言。

当然，完成完整的语言并没有什么错。只是要确定你做得慎重。 否则，你就要小心地控制你的字节码所能表达的范围。在野马脱缰之前把它拴住。

你需要一个前端

底层的字节码指令性能优越，但是二进制的字节码格式不是用户能写的。 我们将行为移出代码的一个原因是想要以更高层的形式表示它。 如果说写C++太过底层，那么让用户写汇编可不是一个改进方案——就算是你设计的！

> 一个反例的是令人尊敬的游戏RoboWar。 在游戏中，**玩家** 编写类似汇编的语言控制机器人，我们这里也会讨论这种指令集。
>
> 这是我介绍类似汇编的语言的首选。

就像GoF的解释器模式，它假设你有某些方法来生成字节码。 通常情况下，用户在更高层编写行为，再用工具将其翻译为虚拟机能理解的字节码。 这里的工具就是编译器。

我知道，这听起来很吓人。丑话说在前头， 如果没有资源制作编辑器，那么字节码不适合你。 但是，接下来你会看到，也可能没你想的那么糟。

你会想念调试器

编程很难。我们知道想要机器做什么，但并不总能正确地传达——所以我们会写出漏洞。为了查找和修复漏洞，我们已经积累了一堆工具来了解代码做错了什么，以及如何修正。我们有调试器，静态分析器，反编译工具等。 所有这些工具都是为现有的语言设计的：无论是机器码还是某些更高层次的东西。

当你定义自己的字节码虚拟机时，你就得把这些工具抛在脑后了。 当然，可以通过调试器调试虚拟机，但它告诉你虚拟机本身在做什么，而不是正在被翻译的字节码是干什么的。

它当然也不会把字节码映射回编译前的高层次的形式。

如果你定义的行为很简单，可能无需太多工具帮忙调试就能勉强坚持下来。 但随着内容规模增长，还是应该花些时间完成些功能，让用户看到字节码在做什么。 这些功能也许不随游戏发布，但它们至关重要，它们能确保你的游戏能被发布。

> 当然，如果你想要让游戏支持模组，那你**会**发布这些特性，它们就更加重要了。

示例代码

经历了前面几个章节后，你也许会惊讶于它的实现是多么直接。 首先需要为虚拟机设定一套指令集。 在开始考虑字节码之类的东西前，先像思考API一样思考它。

法术的API

如果直接使用C++代码定义法术，代码需要调用何种API呢？ 在游戏引擎中，构成法术的基本操作是什么样的？

大多数法术最终改变一个巫师的状态，因此先从这样的代码开始。

```
void setHealth(int wizard, int amount);
void setWisdom(int wizard, int amount);
void setAgility(int wizard, int amount);
```

第一个参数指定哪个巫师被影响，`0`代表玩家而`1`代表对手。 以这种方式，治愈法术可以治疗玩家的巫师，而伤害法术伤害他的敌人。 这三个小方法能覆盖的法术出人意料地多。

如果法术只是默默地调整数据，游戏逻辑就已经完成了， 但玩这样的游戏会让玩家无聊得要哭。让我们修复这点：

```
void playSound(int soundId);
void spawnParticles(int particleType);
```

这并不影响游戏玩法，但它们增强了游戏的体验。 我们可以增加一些镜头晃动，动画之类的，但这足够我们开始了。

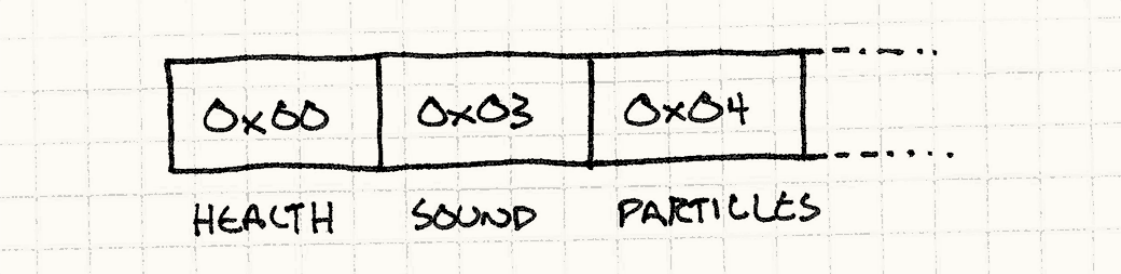
法术指令集

现在让我们把这种程序化的API转化为可被数据控制的东西。 从小处开始，然后慢慢拓展到整体。 现在，要去除方法的所有参数。 假设`set__()`方法总影响玩家的巫师，总直接将状态设为最大值。 同样，FX操作总是播放一个硬编码的声音和粒子效果。

这样，一个法术就只是一系列指令了。 每条指令都代表了想要呈现的操作。我们可以枚举如下：

```
enum Instruction
{
  INST_SET_HEALTH      = 0x00,
  INST_SET_WISDOM      = 0x01,
  INST_SET_AGILITY     = 0x02,
  INST_PLAY_SOUND      = 0x03,
  INST_SPAWN_PARTICLES = 0x04
};
```

为了将法术编码进数据，我们存储了一数组enum值。 只有几个不同的基本操作原语，因此enum值的范围可以存储到一个字节中。 这就意味着法术的代码就是一系列字节——也就是“字节码”。



> 有些字节码虚拟机为每条指令使用多个字节，解码规则也更复杂。 事实上，在x86这样的常见芯片上的机器码更加复杂。
>
> 但单字节对于Java虚拟机和支撑了.NET平台的Common Language Runtime已经足够了，对我们来说也一样。

为了执行一条指令，我们看看它的基本操作原语是什么，然后调用正确的API方法。

```
switch (instruction)
{
  case INST_SET_HEALTH:
    setHealth(0, 100);
    break;

  case INST_SET_WISDOM:
    setWisdom(0, 100);
    break;

  case INST_SET_AGILITY:
    setAgility(0, 100);
    break;

  case INST_PLAY_SOUND:
    playSound(SOUND_BANG);
    break;

  case INST_SPAWN_PARTICLES:
    spawnParticles(PARTICLE_FLAME);
    break;
}
```

用这种方式，解释器建立了沟通代码世界和数据世界的桥梁。我们可以像这样将其放进执行法术的虚拟机：

```
class VM
{
public:
  void interpret(char bytecode[], int size)
  {
    for (int i = 0; i < size; i++)
    {
```

```
      char instruction = bytecode[i];
      switch (instruction)
      {
        // 每条指令的跳转分支……
      }
    }
  }
};
```

输入这些，你就完成了你的首个虚拟机。 不幸的是，它并不灵活。 我们不能设定攻击对手的法术，也不能减少状态上限。我们只能播放声音！

为了获得像一个真正的语言那样的表达能力，我们需要在这里引入参数。

栈式机器

要执行复杂的嵌套表达式，得先从最里面的子表达式开始。 计算完里面的，将结果作为参数向外流向包含它们的表达式， 直到得出最终结果，整个表达式就算完了。

解释器模式将其明确地表现为嵌套对象组成的树，但我们需要指令速度达到列表的速度。我们仍然需要确保子表达式的结果正确地向外传递给包括它的表达式。

但由于数据是扁平的，我们得使用指令的顺序来控制这一点。我们的做法和CPU一样——使用栈。

> 这种架构不出所料地被称为**栈式计算机**。像Forth，PostScript，和Factor 这些语言直接将这点暴露给用户。

```
class VM
{
public:
  VM()
  : stackSize_(0)
  {}

  // 其他代码……

private:
  static const int MAX_STACK = 128;
  int stackSize_;
  int stack_[MAX_STACK];
};
```

虚拟机用内部栈保存值。在例子中，指令交互的值只有一种，那就是数字， 所以可以使用简单的`int`数组。 每当数据需要从一条指令传到另一条，它就得通过栈。

顾名思义，值可以压入栈或者从栈弹出，所以让我们添加一对方法。

```
class VM
{
private:
  void push(int value)
  {
    // 检查栈溢出
    assert(stackSize_ < MAX_STACK);
    stack_[stackSize_++] = value;
  }

  int pop()
  {
    // 保证栈不是空的
    assert(stackSize_ > 0);
    return stack_[--stackSize_];
```

```
  }

  // 其余的代码
};
```

当一条指令需要接受参数，就将参数从栈弹出，如下所示：

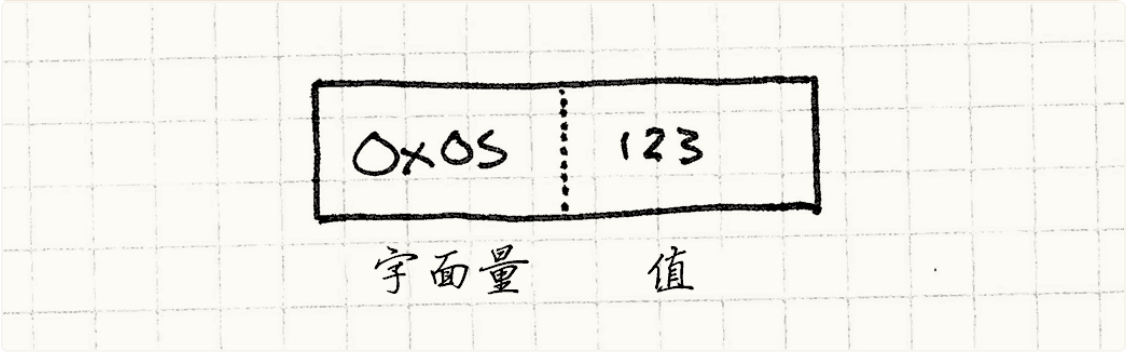
```
switch (instruction)
{
  case INST_SET_HEALTH:
  {
    int amount = pop();
    int wizard = pop();
    setHealth(wizard, amount);
    break;
  }

  case INST_SET_WISDOM:
  case INST_SET_AGILITY:
    // 像上面一样……

  case INST_PLAY_SOUND:
    playSound(pop());
    break;

  case INST_SPAWN_PARTICLES:
    spawnParticles(pop());
    break;
}
```

为了将一些值存入栈中，需要另一条指令：字面量。 它代表了原始的整数值。但是它的值又是从哪里来的呢？ 我们怎么样避免这样追根溯源到无穷无尽呢？

技巧是利用指令是字节序列这一事实——我们可以直接将数值存储在字节数组中。 如下，我们为数值字面量定义了另一条指令类型：

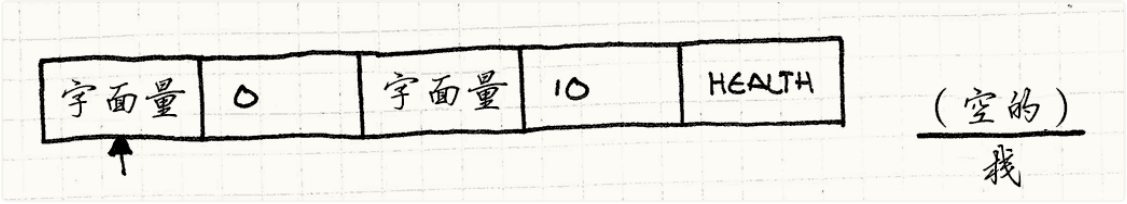
```
case INST_LITERAL:
{
  // 从字节码中读取下一个字节
  int value = bytecode[++i];
  push(value);
  break;
}
```

> 这里，从单个字节中读取值，从而避免了解码多字节整数需要的代码， 但在真实实现中，你会需要支持整个数域的字面量。

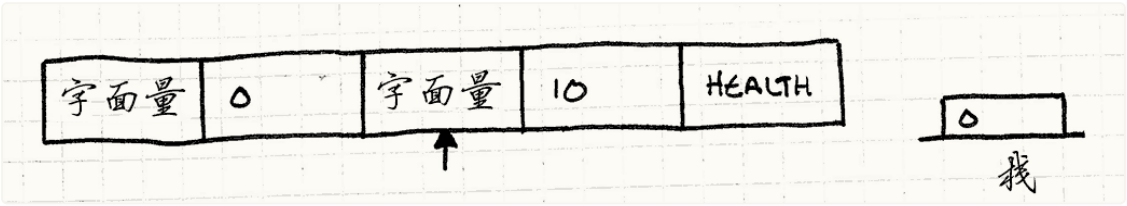


它读取字节码流中的字节作为数值并将其压入栈。

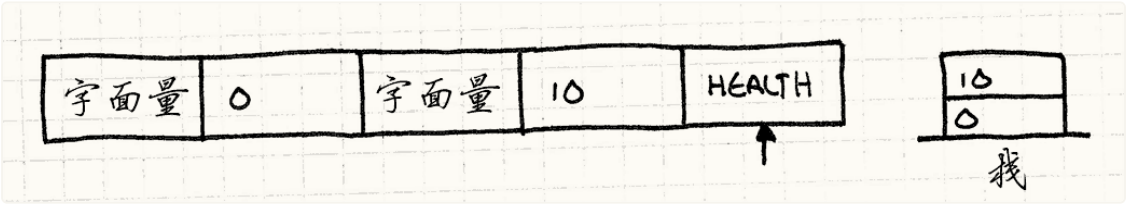
让我们把一些这样的指令串起来看看解释器的执行，感受下栈是如何工作的。 从空栈开始，解释器指向第一个指令：



首先，它执行第一条`INST_LITERAL`，读取字节码流的下一个字节(`0`)并压入栈中。



然后，它执行第二条`INST_LITERAL`，读取`10`然后压入。



最后，执行`INST_SET_HEALTH`。这会弹出`10`存进`amount`，弹出`0`存进`wizard`。然后用这两个参数调用`setHealth()`。

完成！我们获得了将玩家巫师血量设为10点的法术。 现在我们拥有了足够的灵活度，来定义修改任一巫师的状态到任意值的法术。 我们还可以放出不同的声音和粒子效果。

但是……这感觉还是像数据格式。比如，不能将巫师的血量提升为他智力的一半。 设计师希望法术能表达规则，而不仅仅是数值。

行为 = 组合

如果我们视小虚拟机为编程语言，现在它能支持的只有一些内置函数，以及常量参数。 为了让字节码感觉像行为，我们缺少的是组合。

设计师需要能以有趣的方式组合不同的值，来创建表达式。 举个简单的例子，他们想让法术变化一个数值而不是变到一个数值。

这需要考虑到状态的当前值。 我们有指令来修改状态，现在需要添加方法读取状态：

```
case INST_GET_HEALTH:
{
  int wizard = pop();
  push(getHealth(wizard));
  break;
}

case INST_GET_WISDOM:
case INST_GET_AGILITY:
  // 你知道思路了吧……
```

正如你所看到的，这要与栈双向交互。 弹出一个参数来确定获取哪个巫师的状态，然后查找状态的值并压入栈中。

这允许我们创造复制状态的法术。 我们可以创建一个法术，根据巫师的智慧设定敏捷度，或者让巫师的血量等于对方的血量。

有所改善，但仍很受限制。接下来，我们需要算术。 是时候让小虚拟机学习如何计算1 + 1了，我们将添加更多的指令。 现在，你可能已经知道如何去做，猜到了大概的模样。我只

展示加法：

```
case INST_ADD:
{
  int b = pop();
  int a = pop();
  push(a + b);
  break;
}
```

像其他指令一样，它弹出数值，做点工作，然后压入结果。 直到现在，每个新指令似乎都只是有所改善而已，但其实我们已完成大飞跃。 这并不显而易见，但现在我们可以处理各种复杂的，深层嵌套的算术表达式了。

来看个稍微复杂点的例子。 假设我们希望有个法术，能让巫师的血量增加敏捷和智慧的平均值。 用代码表示如下：

```
setHealth(0, getHealth(0) +
    (getAgility(0) + getWisdom(0)) / 2);
```

你可能会认为我们需要指令来处理括号造成的分组，但栈隐式支持了这一点。可以手算如下：

1. 获取巫师当前的血量并记录。
2. 获取巫师敏捷并记录。
3. 对智慧执行同样的操作。
4. 获取最后两个值，加起来并记录。
5. 除以二并记录。
6. 回想巫师的血量，将它和这结果相加并记录。
7. 取出结果，设置巫师的血量为这一结果。

你看到这些“记录”和“回想”了吗？每个“记录”对应一个压入，“回想”对应弹出。 这意味着可以很容易将其转化为字节码。例如，第一行获得巫师的当前血量：

```
LITERAL 0
GET_HEALTH
```

这些字节码将巫师的血量压入堆栈。 如果我们机械地将每行都这样转化，最终得到一大块等价于原来表达式的字节码。 为了让你感觉这些指令是如何组合的，我在下面给你做个示范。

为了展示堆栈如何随着时间推移而变化，我们举个代码执行的例子。 巫师目前有45点血量，7点敏捷，和11点智慧。 每条指令的右边是栈在执行指令之后的模样，再右边是解释指令意图的注释：

```
LITERAL 0    [0]             # 巫师索引
LITERAL 0    [0, 0]          # 巫师索引
GET_HEALTH   [0, 45]         # 获取血量()
LITERAL 0    [0, 45, 0]      # 巫师索引
GET_AGILITY  [0, 45, 7]      # 获取敏捷()
LITERAL 0    [0, 45, 7, 0]   # 巫师索引
GET_WISDOM   [0, 45, 7, 11]  # 获取智慧()
ADD          [0, 45, 18]     # 将敏捷和智慧加起来
LITERAL 2    [0, 45, 18, 2]  # 被除数：2
DIVIDE       [0, 45, 9]      # 计算敏捷和智慧的平均值
ADD          [0, 54]         # 将平均值加到现有血量上。
SET_HEALTH   []              # 将结果设为血量
```

如果你注意每步的栈，你可以看到数据如何魔法一般地在其中流动。 我们最开始压入`0`来查找巫师，然后它一直挂在栈的底部，直到最终的`SET_HEALTH`才用到它。

> 也许“魔法”在这里的门槛太低了。

一台虚拟机

我可以继续下去，添加越来越多的指令，但是时候适可而止了。 如上所述，我们已经有了一个可爱的小虚拟机，可以使用简单，紧凑的数据格式，定义开放的行为。 虽然“字节码”和“虚拟机”的听起来很吓人，但你可以看到它们往往简单到只需栈，循环，和switch语句。

还记得我们最初的让行为呆在沙盒中的目标吗？ 现在，你已经看到虚拟机是如何实现的，很明显，那个目标已经完成。 字节码不能把恶意触角伸到游戏引擎的其他部分，因为我们只定义了几个与其他部分接触的指令。

我们通过控制栈的大小来控制内存使用量，并很小心地确保它不会溢出。 我们甚至可以控制它使用多少时间。 在指令循环里，可以追踪已经执行了多少指令，如果遇到了问题也可以摆脱困境。

> 控制运行时间在例子中没有必要，因为没有任何循环的指令。 可以限制字节码的总体大小来限制运行时间。 这也意味着我们的字节码不是图灵完备的。

现在就剩一个问题了：创建字节码。 到目前为止，我们使用伪代码，再手工编写为字节码。 除非你有很多的空闲时间，否则这种方式并不实用。

语法转换工具

我们最初的目标是创造更高层的方式来控制行为，但是，我们却创造了比C++更底层的东西。 它具有我们想要的运行性能和安全性，但绝对没有对设计师友好的可用性。

为了填补这一空白，我们需要一些工具。 我们需要一个程序，让用户定义法术的高层次行为，然后生成对应的低层栈式机字节码。

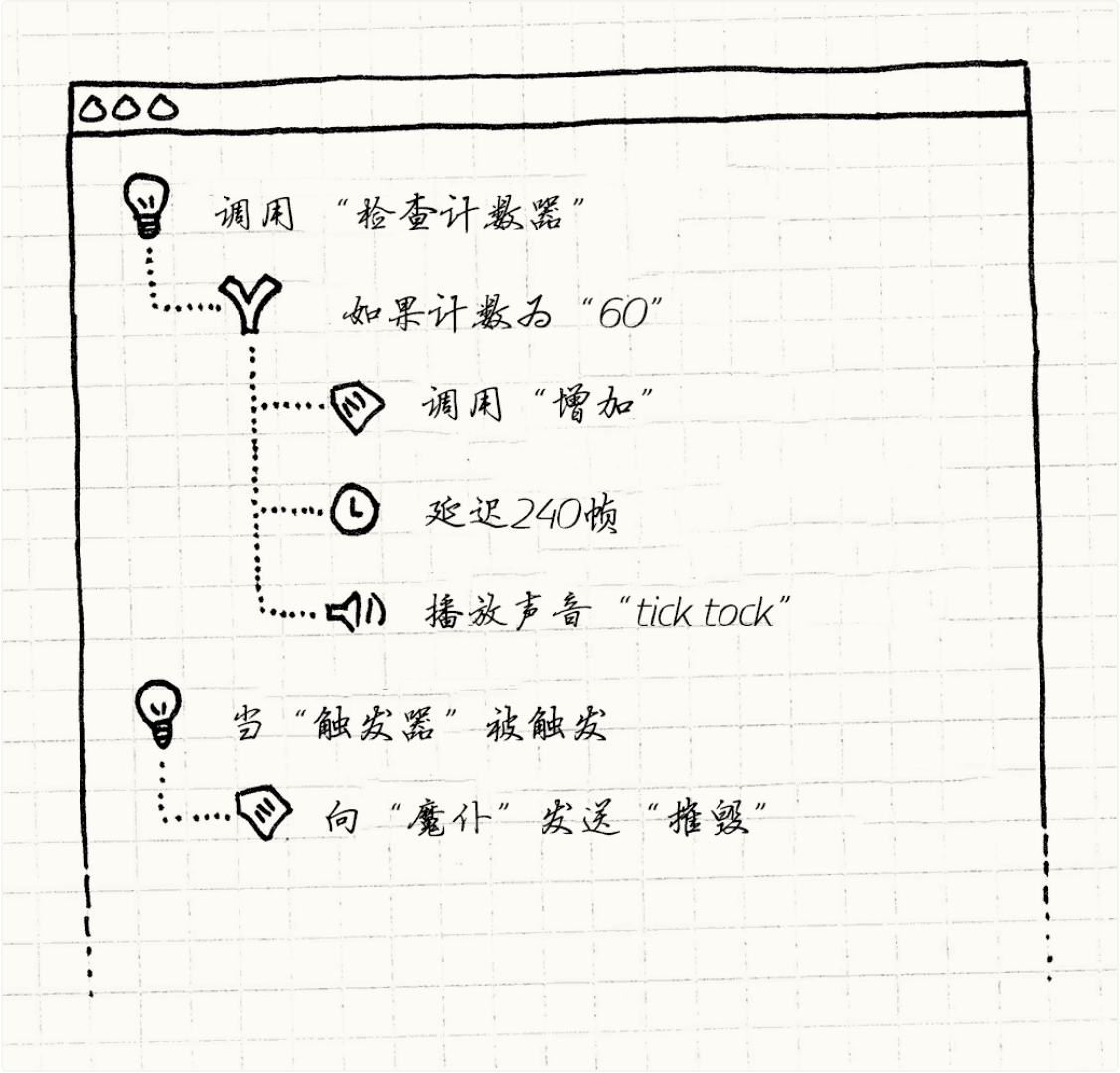
这可能听起来比虚拟机更难。 许多程序员都在大学参加编译器课程，除了被龙书或者”lex“和”yacc”引发了PTSD外，什么也没真正学到。

> 我指的，当然，是经典教材**Compilers: Principles, Techniques, and Tools**。

事实上，编译一个基于文本的语言并不那么糟糕，尽管把这个话题放进这里来要牵扯的东西有点多。但是，你不是非得那么做。 我说，我们需要的是工具——它并不一定是个输入格式是文本文件的编译器。

相反，我建议你考虑构建图形界面让用户定义自己的行为， 尤其是在使用它的人没有很高的技术水平时。 没有花几年时间习惯编译器怒吼的人很难写出没有语法错误的文本。

你可以建立一个应用程序，用户通过单击拖动小盒子，下拉菜单项，或任何有意义的行为创建“脚本”，从而创建行为。



> 我为Henry Hatsworth in the Puzzling Adventure编写的脚本系统就是这么工作的。

这样做的好处是，你的UI可以保证用户无法创建“无效的”程序。与其向他们吐一大堆错误警告，不如主动禁用按钮或提供默认值， 以确保他们创造的东西在任何时间点上都有效。

> 我想要强调错误处理是多么重要。作为程序员，我们趋向于将人为错误视为应当极力避免的的个人耻辱。
>
> 为了制作用户喜欢的系统，你需要接受人性，**包括他们的失败**。是人都会犯错误，但错误同时也是创作的固有基础。 用撤销这样的特性优雅地处理它们，这能让用户更有创意，创作出更好的成果。

这免去了设计语法和编写解析器的工作。 但是我知道，你可能会发现UI设计同样令人不快。 好吧，如果这样，我就没啥办法啦。

毕竟，这种模式是关于使用对用户友好的高层方式表达行为。 你必须精心设计用户体验。要有效地执行行为，又需要将其转换成底层形式。这是必做的，但如果你准备好迎接挑战，这终会有所回报。

设计决策

我想尽可能让本章简短，但我们所做的事情实际上可是创造语言啊。 那可是个宽泛的设计领域，你可以从中获得很多乐趣，所以别沉迷于此反而忘了完成你的游戏。

> 这是本书中最长的章节，看来我失败了。

指令如何访问堆栈？

字节码虚拟机主要有两种：基于栈的和基于寄存器的。栈式虚拟机中，指令总是操作栈顶，如同我们的示例代码所示。例如，`INST_ADD`弹出两个值，将它们相加，将结果压入。

基于寄存器的虚拟机也有栈。唯一不同的是指令可以从栈的深处读取值。不像`INST_ADD`始终弹出其操作数，它在字节码中存储两个索引，指示了从栈的何处读取操作数。

- 基于栈的虚拟机：
  - 指令短小。由于每个指令隐式认定在栈顶寻找参数，不需要为任何数据编码。这意味着每条指令可能会非常短，一般只需一个字节。
  - 易于生成代码。当你需要为生成字节码编写编译器或工具时，你会发现基于栈的字节码更容易生成。由于每个指令隐式地在栈顶工作，你只需要以正确的顺序输出指令就可以在它们之间传递参数。
  - 会生成更多的指令。每条指令只能看到栈顶。这意味着，产生像`a = b + c`这样的代码，你需要单独的指令将`b`和`c`压入栈顶，执行操作，再将结果压入`a`。
- 基于寄存器的虚拟机：
  - 指令较长。由于指令需要参数记录栈偏移量，单个指令需要更多的位。例如，一个Lua指令占用完整的32位——它可能是最著名的基于寄存器的虚拟机了。它采用6位做指令类型，其余的是参数。

    > Lua作者没有指定Lua的字节码格式，它每个版本都会改变。现在描述的是Lua 5.1。要深究Lua的内部构造，读读这个。

  - 指令较少。由于每个指令可以做更多的工作，你不需要那么多的指令。有人说，性能会得以提升，因为不需要将值在栈中移来移去了。

所以，应该选一种？我的建议是坚持使用基于栈的虚拟机。它们更容易实现，也更容易生成代码。Lua转换为基于寄存器的虚拟机从而变得更快，这为寄存器虚拟机博得了声誉，但是这强烈依赖于实际的指令和虚拟机的其他大量细节。

你有什么指令？

指令集定义了在字节码中可以干什么，不能干什么，对虚拟机性能也有很大的影响。这里有个清单，记录了你可能需要的不同种类的指令：

- 外部基本操作原语。这是虚拟机与引擎其他部分交互，影响玩家所见的部分。它们控制了字节码可以表达的真实行为。如果没有这些，你的虚拟机除了消耗CPU循环以外一无所得。
- 内部基本操作原语 这些语句在虚拟机内操作数值——文字，算术，比较操作，以及操纵栈的指令。
- 控制流。我们的例子没有包含这些，但当你需要有条件执行或循环执行，你就会需要控制流。在字节码这样底层的语言中，它们出奇地简单：跳转。

  在我们的指令循环中，需要索引来跟踪执行到了字节码的哪里。跳转指令做的是修改这个索引并改变将要执行的指令。换言之，这就是`goto`。你可以基于它制定各种更高级别的控制流。
- 抽象。如果用户开始在数据中定义很多的东西，最终要重用字节码的部分位，而不是复制和粘贴。你也许会需要可调用过程这样的东西。

最简单的形式中，过程并不比跳转复杂。 唯一不同的是，虚拟机需要管理另一个返回栈。 当执行“call”指令时，将当前指令索引压入栈中，然后跳转到被调用的字节码。 当它到了“return”，虚拟机从堆栈弹出索引，然后跳回索引指示的位置。

数值是如何表示的？

我们的虚拟机示例只与一种数值打交道：整数。 回答这个问题很简单——栈只是一栈的`int`。 更加完整的虚拟机支持不同的数据类型：字符串，对象，列表等。 你必须决定在内部如何存储这些值。

- 单一数据类型：
  - 简单易用 你不必担心标记，转换，或类型检查。
  - 无法使用不同的数据类型。 这是明显的缺点。将不同类型成塞进单一的表示方式——比如将数字存储为字符串——这是自找麻烦。
- 带标记的类型：

  这是动态类型语言中常见的表示法。 所有的值有两部分。 第一部分是类型标识——一个存储了数据的类型的`enum`。其余部分会被解释为这种类型：

```
enum ValueType
{
  TYPE_INT,
  TYPE_DOUBLE,
  TYPE_STRING
};

struct Value
{
  ValueType type;
  union
  {
    int    intValue;
    double doubleValue;
    char*  stringValue;
  };
};
```

  - 数值知道其类型。 这个表示法的好处是可在运行时检查值的类型。 这对动态调用很重要，可以确保没有在类型上面执行其不支持的操作。
  - 消耗更多内存。 每个值都要带一些额外的位来标识类型。在像虚拟机这样的底层，这里几位，那里几位，总量就会快速增加。
- 无标识的**union**：

  像前面一样使用union，但是没有类型标识。 你可以将这些位表示为不同的类型，由你确保没有搞错值的类型。

  这是静态类型语言在内存中表示事物的方式。 由于类型系统在编译时保证没弄错值的类型，不需要在运行时对其进行验证。

  > 这也是**无类型**语言，像汇编和Forth存储值的方式。 这些语言让**用户**保证不会写出误认值的类型的代码。毫无服务态度！

  - 结构紧凑。 找不到比只存储需要的值更加有效率的存储方式。
  - 速度快。 没有类型标识意味着在运行时无需消耗周期检查它们的类型。这是静态类型语言往往比动态类型语言快的原因之一。

- 不安全。这是真正的代价。一块错误的字节码，会让你误解一个值，把数字误解为指针，会破坏游戏安全性从而导致崩溃。

  > 如果你的字节码是由静态类型语言编译而来，你也许认为它是安全的，因为编译不会生成不安全的字节码。 那也许是真的，但记住恶意用户也许会手写恶意代码而不经过你的编译器。
  >
  > 举个例子，这就是为什么Java虚拟机在加载程序时要做**字节码验证**。

- 接口：

  多种类型值的面向对象解决方案是通过多态。接口为不同的类型的测试和转换提供虚方法，如下：

  ```
  class Value
  {
  public:
    virtual ~Value() {}

    virtual ValueType type() = 0;

    virtual int asInt() {
      // 只能在int上调用
      assert(false);
      return 0;
    }

    // 其他转换方法……
  };
  ```

  然后你为每个特定的数据类型设计特定的类，如：

  ```
  class IntValue : public Value
  {
  public:
    IntValue(int value)
    : value_(value)
    {}

    virtual ValueType type() { return TYPE_INT; }
    virtual int asInt() { return value_; }

  private:
    int value_;
  };
  ```

  - 开放。可在虚拟机的核心之外定义新的值类型，只要它们实现了基本接口就行。
  - 面向对象。如果你坚持OOP原则，这是“正确”的做法，为特定类型使用多态分配行为，而不是在标签上做switch之类的。
  - 冗长。必须定义单独的类，包含了每个数据类型的相关行为。 注意在前面的例子中，这样的类定义了所有的类型。在这里，只包含了一个！
  - 低效。为了使用多态，必须使用指针，这意味着即使是短小的值，如布尔和数字，也得裹在堆中分配的对象里。 每使用一个值，你就得做一次虚方法调用。

    在虚拟机核心之类的地方，像这样的性能影响会迅速叠加。 事实上，这引起了许多我们试图在解释器模式中避免的问题。 只是现在的问题不在代码中，而是在值中。

我的建议是：如果可以，只用单一数据类型。 除此以外，使用带标识的union。这是世界上几乎每个语言解释器的选择。

如何生成字节码？

我将最重要的问题留到最后。我们已经完成了消耗和解释字节码的部分，但需你要写制造字节码的工具。典型的解决方案是写个编译器，但它不是唯一的选择。

- 如果你定义了基于文本的语言：
  - 必须定义语法。业余和专业的语言设计师小看这件事情的难度。让解析器高兴很简单，让用户快乐很难。

    语法设计是用户界面设计，当你将用户界面限制到字符构成的字符串，这可没把事情变简单。
  - 必须实现解析器。不管名声如何，这部分其实非常简单。无论使用ANTLR或Bison，还是——像我一样——手写递归下降，都可以完成。
  - 必须处理语法错误。这是最重要和最困难的部分。当用户制造了语法和语义错误——他们总会这么干——引导他们返回到正确的道路是你的任务。解析器只知道接到了意外的符号，给予有用的的反馈并不容易。
  - 可能会对非技术用户关上大门。我们程序员喜欢文本文件。结合强大的命令行工具，我们把它们当作计算机的乐高积木——简单，有百万种方式组合。

    大部分非程序员不这样想。对他们来说，输入文本文件就像为愤怒机器人审核员填写税表，如果忘记了一个分号就会遭到痛斥。
- 如果你定义了一个图形化创作工具：
  - 必须实现用户界面。按钮，点击，拖动，诸如此类。有些人畏惧它，但我喜欢它。如果沿着这条路走下去，设计用户界面和工作核心部分同等重要——而不是硬着头皮完成的乱七八糟工作。

    每点额外工作都会让工具更容易更舒适地使用，并直接导致了游戏中更好的内容。如果你看看很多游戏制作过程的内部解密，经常会发现制作有趣的创造工具是秘诀之一。
  - 有较少的错误情况。由于用户通过交互式一步一步地设计行为，应用程序可以尽快引导他们走出错误。

    而使用基于文本的语言时，直到用户输完整个文件才能看到用户的内容，预防和处理错误更加困难。
  - 更难移植。文本编译器的好处是，文本文件是通用的。编译器简单地读入文件并写出。跨平台移植的工作实在微不足道。

    > 除了换行符。还有编码。

    当你构建用户界面，你必须选择要使用的架构，其中很多是基于某个操作系统。也有跨平台的用户界面工具包，但他们往往要为对所有平台同样适用付出代价——它们在不同的平台上同样差异很大。

参见

- 这一章节的近亲是GoF的解释器模式[GoF]。两种方式都能让你用数据组合行为。

  事实上，最终你两种模式都会使用。你用来构造字节码的工具会有内部的对象树。这也是解释器模式所能做的。

为了编译到字节码，你需要递归回溯整棵树，就像用解释器模式去解释它一样。 唯一的 不同在于，不是立即执行一段行为，而是生成整个字节码再执行。

- Lua是游戏中最广泛应用的脚本语言。 它的内部被实现为一个非常紧凑的，基于寄存器的字节码虚拟机。
- Kismet是个可视化脚本编辑工具，应用于Unreal引擎的编辑器UnrealEd。
- 我的脚本语言Wren，是一个简单的，基于栈的字节码解释器。

← 上一章 ≡ 首页 下一章 →