#7.表达式求值

> You are my creator, but I am your master; Obey!
>
> —— Mary Shelley, *Frankenstein*

你是我的创造者，但我是你的主人，听话！

——Mary Shelley, *科学怪人*

> If you want to properly set the mood for this chapter, try to conjure up a thunderstorm, one of those swirling tempests that likes to yank open shutters at the climax of the story. Maybe toss in a few bolts of lightning. In this chapter, our interpreter will take breath, open its eyes, and execute some code.

如果你想为这一章适当地设定气氛，试着想象一场雷雨，那种在故事高潮时喜欢拉开百叶窗的漩涡式暴风雨。也许再加上几道闪电。在这一章中，我们的解释器将开始呼吸，睁开眼睛，并执行一些代码。

> There are all manner of ways that language implementations make a computer do what the user's source code commands. They can compile it to machine code, translate it to another high-level language, or reduce it to some bytecode format for a virtual machine to run. For our first interpreter, though, we are going to take the simplest, shortest path and execute the syntax tree itself.

对于语言实现来说，有各种方式可以使计算机执行用户的源代码命令。它们可以将其编译为机器代码，将其翻译为另一种高级语言，或者将其还原为某种字节码格式，以便在虚拟机中执行。不过对于我们的第一个解释器，我们要选择最简单、最短的一条路，也就是执行语法树本身。

> Right now, our parser only supports expressions. So, to "execute" code, we will evaluate an expression and produce a value. For each kind of expression syntax we can parse—literal, operator, etc.—we need a corresponding chunk of code that knows how to evaluate that tree and produce a result. That raises two questions:

现在，我们的解释器只支持表达式。因此，为了"执行"代码，我们要计算一个表达式时并生成一个值。对于我们可以解析的每一种表达式语法——字面量，

操作符等——我们都需要一个相应的代码块，该代码块知道如何计算该语法树并产生结果。这也就引出了两个问题：

1. > What kinds of values do we produce?

   我们要生成什么类型的值?

2. > How do we organize those chunks of code?

   我们如何组织这些代码块?

> Taking them on one at a time . . .

让我们来逐个击破。

> ## #7.1 Representing Values

#7.1 值描述

> In Lox, values are created by literals, computed by expressions, and stored in variables. The user sees these as *Lox* objects, but they are implemented in the underlying language our interpreter is written in. That means bridging the lands of Lox's dynamic typing and Java's static types. A variable in Lox can store a value of any (Lox) type, and can even store values of different types at different points in time. What Java type might we use to represent that?

在Lox中，值由字面量创建，由表达式计算，并存储在变量中。用户将其视作Lox对象[1]，但它们是用编写解释器的底层语言实现的。这意味着要在Lox的动态类型和Java的静态类型之间架起桥梁。Lox中的变量可以存储任何（Lox）类

型的值，甚至可以在不同时间存储不同类型的值。我们可以用什么Java类型来表示?

> Given a Java variable with that static type, we must also be able to determine which kind of value it holds at runtime. When the interpreter executes a `+` operator, it needs to tell if it is adding two numbers or concatenating two strings. Is there a Java type that can hold numbers, strings, Booleans, and more? Is there one that can tell us what its runtime type is? There is! Good old java.lang.Object.

给定一个具有该静态类型的Java变量，我们还必须能够在运行时确定它持有哪种类型的值。当解释器执行 `+` 运算符时，它需要知道它是在将两个数字相加还是在拼接两个字符串。有没有一种Java类型可以容纳数字、字符串、布尔值等等？有没有一种类型可以告诉我们它的运行时类型是什么？有的! 就是老牌的 `java.lang.Object` 。

> In places in the interpreter where we need to store a Lox value, we can use Object as the type. Java has boxed versions of its primitive types that all subclass Object, so we can use those for Lox's built-in types:

在解释器中需要存储Lox值的地方，我们可以使用Object作为类型。Java已经将其基本类型的所有子类对象装箱了，因此我们可以将它们用作Lox的内置类型：

Lox type Lox类	**Java representation Java表示**
Any Lox value	Object
`nil`	`null`
Boolean	Boolean
number	Double
string	String

> Given a value of static type Object, we can determine if the runtime value is a number or a string or whatever using Java's built-in `instanceof` operator. In other words, the JVM's own object representation conveniently gives us everything we need to implement Lox's built-in types. We'll have to do a little more work later when we add Lox's notions of functions, classes, and instances, but Object and the boxed primitive classes are sufficient for the types we need right now.

给定一个静态类型为Object的值，我们可以使用Java内置的`instanceof`操作符来确定运行时的值是数字、字符串或其它什么。换句话说，JVM自己的对象表示方便地为我们提供了实现Lox内置类型所需的一切[2]。当稍后添加Lox的函数、类和实例等概念时，我们还必须做更多的工作，但Object和基本类型的包装类足以满足我们现在的需要。

> ## #7.2 Evaluating Expressions

#7.2 表达式求值

> Next, we need blobs of code to implement the evaluation logic for each kind of expression we can parse. We could stuff that code into the syntax tree classes in something like an `interpret()` method. In effect, we could tell each syntax tree node, "Interpret thyself". This is the Gang of Four's Interpreter design pattern. It's a neat pattern, but like I mentioned earlier, it gets messy if we jam all sorts of logic into the tree classes.

接下来，我们需要大量的代码实现我们可解析的每种表达式对应的求值逻辑。我们可以把这些代码放在语法树的类中，比如添加一个`interpret()`方法。然后，我们可以告诉每一个语法树节点"解释你自己"，这就是四人组的解释器模式。这是一个整洁的模式，但正如我前面提到的，如果我们将各种逻辑都塞进语法树类中，就会变得很混乱。

> Instead, we're going to reuse our groovy Visitor pattern. In the previous chapter, we created an AstPrinter class. It took in a syntax tree and recursively traversed it, building up a string which it ultimately returned. That's almost exactly what a real interpreter does, except instead of concatenating strings, it computes values.

相反，我们将重用我们的访问者模式。在前面的章节中，我们创建了一个AstPrinter类。它接受一个语法树，并递归地遍历它，构建一个最终返回的字符串。这几乎就是一个真正的解释器所做的事情，只不过解释器不是连接字符串，而是计算值。

> We start with a new class.

我们先创建一个新类。

lox/Interpreter.java，创建新文件：

```
package com.craftinginterpreters.lox;

class Interpreter implements Expr.Visitor<Object> {
}
```

> The class declares that it's a visitor. The return type of the visit methods will be Object, the root class that we use to refer to a Lox value in our Java code. To satisfy the Visitor interface, we need to define visit methods for each of the four expression tree classes our parser produces. We'll start with the simplest . . .

这个类声明它是一个访问者。访问方法的返回类型将是Object，即我们在Java代码中用来引用Lox值的根类。为了实现Visitor接口，我们需要为解析器生成的四个表达式树类中分别定义访问方法。我们从最简单的开始…

> ### #7.2.1 Evaluating literals

#7.2.1 字面量求值

> The leaves of an expression tree—the atomic bits of syntax that all other expressions are composed of—are literals. Literals are almost values already, but the distinction is important. A literal is a *bit of syntax* that produces a value. A literal always appears somewhere in the user's source code. Lots of values are produced by computation and don't exist anywhere in the code itself. Those aren't literals. A literal comes from the parser's domain. Values are an interpreter concept, part of the runtime's world.

一个表达式树的叶子节点（构成其它表达式的语法原子单位）是字面量[3]。字面符号几乎已经是值了，但两者的区别很重要。字面量是产生一个值的语法单元。字面量总是出现在用户的源代码中的某个地方。而很多值是通过计算产生的，并不存在于代码中的任何地方，这些都不是字面量。字面量来自于解析器领域，而值是一个解释器的概念，是运行时世界的一部分。

> So, much like we converted a literal *token* into a literal *syntax tree node* in the parser, now we convert the literal tree node into a runtime value. That turns out to be trivial.

因此，就像我们在解析器中将字面量*标记*转换为字面量*语法树节点*一样，现在我们将字面量树节点转换为运行时值。这其实很简单。

lox/Interpreter.java，在 Interpreter类中添加：

```
@Override
public Object visitLiteralExpr(Expr.Literal expr) {
  return expr.value;
}
```

> We eagerly produced the runtime value way back during scanning and stuffed it in the token. The parser took that value and stuck it in the literal tree node, so to evaluate a literal, we simply pull it back out.

我们早在扫描过程中就即时生成了运行时的值，并把它放进了语法标记中。解析器获取该值并将其插入字面量语法树节点中，所以要对字面量求值，我们只需把它存的值取出来。

> ### #7.2.2 Evaluating parentheses

#7.2.2 括号求值

> The next simplest node to evaluate is grouping—the node you get as a result of using explicit parentheses in an expression.

下一个要求值的节点是分组——在表达式中显式使用括号时产生的语法树节点。

lox/Interpreter.java，在 Interpreter类中添加：

```java
  @Override
  public Object visitGroupingExpr(Expr.Grouping expr) {
    return evaluate(expr.expression);
  }
```

> A grouping node has a reference to an inner node for the expression contained inside the parentheses. To evaluate the grouping expression itself, we recursively evaluate that subexpression and return it.

一个分组节点中包含一个引用指向对应于括号内的表达式的内部节点[4]。要想计算括号表达式，我们只需要递归地对子表达式求值并返回结果即可。

> We rely on this helper method which simply sends the expression back into the interpreter’s visitor implementation:

我们依赖于下面这个辅助方法，它只是将表达式发送回解释器的访问者实现中：

lox/Interpreter.java，在 Interpreter类中添加：

```java
private Object evaluate(Expr expr) {
  return expr.accept(this);
```

> ## #7.2.3 Evaluating unary expressions

#7.2.3 一元表达式求值

> Like grouping, unary expressions have a single subexpression that we must evaluate first. The difference is that the unary expression itself does a little work afterwards.

像分组表达式一样，一元表达式也有一个必须先求值的子表达式。不同的是，一元表达式自身在完成求值之后还会做一些工作。

lox/Interpreter.java，在 visitLiteralExpr()方法后添加：

```java
@Override
public Object visitUnaryExpr(Expr.Unary expr) {
  Object right = evaluate(expr.right);

  switch (expr.operator.type) {
    case MINUS:
      return -(double)right;
  }

  // Unreachable.
  return null;
}
```

> First, we evaluate the operand expression. Then we apply the unary operator itself to the result of that. There are two different unary expressions, identified by the type of the operator token.

首先，我们计算操作数表达式，然后我们将一元操作符作用于子表达式的结果。我们有两种不同的一元表达式，由操作符标记的类型来区分。

> Shown here is `-`, which negates the result of the subexpression. The subexpression must be a number. Since we don't *statically* know that in Java, we cast it before performing the operation. This type cast happens at runtime when the `-` is evaluated. That's the core of what makes a language dynamically typed right there.

这里展示的是`-`，它会对子表达式的结构取负。子表达式结果必须是数字。因为我们在Java中无法*静态地*知道这一点，所以我们在执行操作之前先对其进行强制转换。这个类型转换是在运行时对`-`求值时发生的。这就是将语言动态类型化的核心所在。

> You can start to see how evaluation recursively traverses the tree. We can't evaluate the unary operator itself until after we evaluate its operand subexpression. That means our interpreter is doing a **post-order traversal**—each node evaluates its children before doing its own work.

你可以看到求值过程是如何递归遍历语法树的。在对一元操作符本身进行计算之前，我们必须先对其操作数子表达式求值。这表明，解释器正在进行**后序遍历**——每个节点在自己求值之前必须先对子节点求值。

> The other unary operator is logical not.

另一个一元操作符是逻辑非。

lox/Interpreter.java，在visitUnaryExpr()方法中添加：

```
    switch (expr.operator.type) {
  // 新增部分开始
        case BANG:
    return !isTruthy(right);
  // 新增部分结束
  case MINUS:
```

> The implementation is simple, but what is this “truthy” thing about? We need to make a little side trip to one of the great questions of Western philosophy: *What is truth?*

实现很简单，但是这里的“真实”指的是什么呢？我们需要简单地讨论一下西方哲学中的一个伟大问题：什么是真理？

> ## #7.2.4 Truthiness and falsiness

#7.2.4 真与假

> OK, maybe we’re not going to really get into the universal question, but at least inside the world of Lox, we need to decide what happens when you use something other than `true` or `false` in a logic operation like `!` or any other place where a Boolean is expected.

好吧，我们不会真正深入这个普世的问题，但是至少在Lox的世界中，我们需要确定当您在逻辑运算（如`!`或其他任何需要布尔值的地方）中使用非`true`或`false`以外的东西时会发生什么？

> We *could* just say it’s an error because we don’t roll with implicit conversions, but most dynamically typed languages aren’t that ascetic. Instead, they take the universe of values of all types and partition them into two sets, one of which they define to be “true”, or “truthful”, or (my favorite) “truthy”, and the rest which are “false” or “falsey”. This partitioning is somewhat arbitrary and gets weird in a few languages.

我们*可以*说这是一个错误，因为我们没有使用隐式转换，但是大多数动态类型语言并不那么严格。相反，他们把所有类型的值分成两组，其中一组他们定义为“真”，其余为“假”。这种划分有些武断，在一些语言中会变得很奇怪[5]。

> Lox follows Ruby’s simple rule: `false` and `nil` are falsey, and everything else is truthy. We implement that like so:

Lox遵循Ruby的简单规则：`false`和`nil`是假的，其他都是真的。我们是这样实现的：

lox/Interpreter.java，在 visitUnaryExpr()方法后添加：

```java
private boolean isTruthy(Object object) {
  if (object == null) return false;
  if (object instanceof Boolean) return (boolean)object;
  return true;
}
```

> ## #7.2.5 Evaluating binary operators

#7.2.5 二元操作符求值

> On to the last expression tree class, binary operators. There’s a handful of them, and we’ll start with the arithmetic ones.

来到最后的表达式树类——二元操作符，其中包含很多运算符，我们先从数学运算开始。

lox/Interpreter.java，在 evaluate()方法后添加[6]：

```java
@Override
public Object visitBinaryExpr(Expr.Binary expr) {
  Object left = evaluate(expr.left);
  Object right = evaluate(expr.right);

  switch (expr.operator.type) {
    case MINUS:
      return (double)left - (double)right;
    case SLASH:
      return (double)left / (double)right;
    case STAR:
      return (double)left * (double)right;
  }

  // Unreachable.
  return null;
}
```

> I think you can figure out what’s going on here. The main difference from the unary negation operator is that we have two operands to evaluate.

我想你能理解这里的实现。与一元取负运算符的主要区别是，我们有两个操作数要计算。

> I left out one arithmetic operator because it’s a little special.

我漏掉了一个算术运算符，因为它有点特殊。

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    switch (expr.operator.type) {
      case MINUS:
        return (double)left - (double)right;
      // 新增部分开始
      case PLUS:
        if (left instanceof Double && right instanceof Double) {
          return (double)left + (double)right;
        }

        if (left instanceof String && right instanceof String) {
          return (String)left + (String)right;
        }

        break;
      // 新增部分结束
      case SLASH:
```

> The `+` operator can also be used to concatenate two strings. To handle that, we don’t just assume the operands are a certain type and *cast* them, we dynamically *check* the type and choose the appropriate operation. This is why we need our object representation to support `instanceof`.

`+`操作符也可以用来拼接两个字符串。为此，我们不能只是假设操作数是某种类型并将其强制转换，而是要动态地检查操作数类型并选择适当的操作。这就是为什么我们需要对象表示能支持`instanceof`。

> Next up are the comparison operators.

接下来是比较操作符。

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    switch (expr.operator.type) {
            // 新增部分开始
      case GREATER:
        return (double)left > (double)right;
      case GREATER_EQUAL:
        return (double)left >= (double)right;
      case LESS:
        return (double)left < (double)right;
      case LESS_EQUAL:
        return (double)left <= (double)right;
      // 新增部分结束
      case MINUS:
```

> They are basically the same as arithmetic. The only difference is that where the arithmetic operators produce a value whose type is the same as the operands (numbers or strings), the comparison operators always produce a Boolean.

它们基本上与算术运算符相同。唯一的区别是，算术运算符产生的值的类型与操作数（数字或字符串）相同，而比较运算符总是产生一个布尔值。

> The last pair of operators are equality.

最后一对是等式运算符。

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case BANG_EQUAL: return !isEqual(left, right);
      case EQUAL_EQUAL: return isEqual(left, right);
```

> Unlike the comparison operators which require numbers, the equality operators support operands of any type, even mixed ones. You can't ask Lox if 3 is *less* than `"three"`, but you can ask if it's *equal* to it.

与需要数字的比较运算符不同，等式运算符支持任何类型的操作数，甚至是混合类型。你不能问Lox 3是否*小于*`"three"`，但你可以问它3是否等于`"three"`。

> Like truthiness, the equality logic is hoisted out into a separate method.

与真假判断一样，相等判断也被提取到了单独的方法中。

lox/Interpreter.java，在 isTruthy()方法后添加：

```
  private boolean isEqual(Object a, Object b) {
    if (a == null && b == null) return true;
    if (a == null) return false;

    return a.equals(b);
  }
```

> This is one of those corners where the details of how we represent Lox objects in terms of Java matter. We need to correctly implement *Lox's* notion of equality, which may be different from Java's.

这是我们使用Java表示Lox对象的细节一角。我们需要正确地实现Lox的相等概念，这可能与Java中不同。

> Fortunately, the two are pretty similar. Lox doesn’t do implicit conversions in equality and Java does not either. We do have to handle `nil`/`null` specially so that we don’t throw a NullPointerException if we try to call `equals()` on `null`. Otherwise, we’re fine. Java’s `.equals()` method on Boolean, Double, and String have the behavior we want for Lox.

幸运的是，这两者很相似。Lox不会在等式中做隐式转换，Java也不会。我们必须对 `nil`/`null` 做特殊处理，这样就不会在对`null`调用`equals()`方法时抛出NullPointerException。其它情况下，都是没问题的。Java中的`.equals()`方法对Boolean、Double和 String的处理都符合Lox的要求[7]。

> And that’s it! That’s all the code we need to correctly interpret a valid Lox expression. But what about an *invalid* one? In particular, what happens when a subexpression evaluates to an object of the wrong type for the operation being performed?

就这样了! 这就是我们要正确解释一个有效的Lox表达式所需要的全部代码。但是*无效的*表达式呢？尤其是，当一个子表达式的计算结果类型与待执行的操作不符时会发生什么?

> ## #7.3 Runtime Errors

#7.3 运行时错误

> I was cavalier about jamming casts in whenever a subexpression produces an Object and the operator requires it to be a number or a string. Those casts can fail. Even though the user’s code is erroneous, if we want to make a usable language, we are responsible for handling that error gracefully.

每当子表达式产生一个对象，而运算符要求它是一个数字或字符串时，我都会轻率地插入强制类型转换。这些类型转换可能会失败。如果我们想做出一个可用的语言，即使用户的代码是错误的，我们也有责任优雅地处理这个错误[8]。

> It's time for us to talk about **runtime errors**. I spilled a lot of ink in the previous chapters talking about error handling, but those were all *syntax* or *static* errors. Those are detected and reported before *any* code is executed. Runtime errors are failures that the language semantics demand we detect and report while the program is running (hence the name).

现在是时候讨论**运行时错误**了。在前面的章节中，我花了很多笔墨讨论错误处理，但这些都是语法或静态错误。这些都是在代码执行之前进行检测和报告的。运行时错误是语言语义要求我们在程序运行时检测和报告的故障（因此得名）。

> Right now, if an operand is the wrong type for the operation being performed, the Java cast will fail and the JVM will throw a ClassCastException. That unwinds the whole stack and exits the application, vomiting a Java stack trace onto the user. That's probably not what we want. The fact that Lox is implemented in Java should be a detail hidden from the user. Instead, we want them to understand that a *Lox* runtime error occurred, and give them an error message relevant to our language and their program.

现在，如果操作数对于正在执行的操作来说是错误的类型，那么Java转换将失败，JVM将抛出一个ClassCastException。这将跳脱出整个调用堆栈并退出应用程序，然后向用户抛出Java堆栈跟踪信息。这可能不是我们想要的。Lox是用Java实现的这一事实应该是一个对用户隐藏的细节。相反，我们希望他们理解此时发生的是Lox运行时错误，并给他们一个与我们的语言和他们的程序相关的错误信息。

> The Java behavior does have one thing going for it, though. It correctly stops executing any code when the error occurs. Let's say the user enters some expression like:

不过，Java的行为确实有一个优点。当错误发生时，它会正确地停止执行代码。比方说，用户输入了一些表达式，比如：

```
2 * (3 / -"muffin")
```

> You can’t negate a muffin, so we need to report a runtime error at that inner `-` expression. That in turn means we can’t evaluate the `/` expression since it has no meaningful right operand. Likewise for the `*`. So when a runtime error occurs deep in some expression, we need to escape all the way out.

你无法对"muffin"取负，所以我们需要在内部的 `-` 表达式中报告一个运行时错误。这又意味着我们无法计算 `/` 表达式，因为它的右操作数无意义，对于 `*` 表达式也是如此。因此，当某个表达式深处出现运行时错误时，我们需要一直跳出到最外层。

> We could print a runtime error and then abort the process and exit the application entirely. That has a certain melodramatic flair. Sort of the programming language interpreter equivalent of a mic drop.

我们可以打印一个运行时错误，然后中止进程并完全退出应用程序。这有一点戏剧性，有点像编程语言解释器中的 "mic drop"。

> Tempting as that is, we should probably do something a little less cataclysmic. While a runtime error needs to stop evaluating the *expression*, it shouldn’t kill the *interpreter*. If a user is running the REPL and has a typo in a line of code, they should still be able to keep the session going and enter more code after that.

尽管这种处理方式很诱人，我们或许应该做一些不那么灾难性的事情。虽然运行时错误需要停止对表达式的计算，但它不应该杀死解释器。如果用户正在运行REPL，并且在一行代码中出现了错误，他们应该仍然能够保持会话，并在之后继续输入更多的代码。

> ## #7.3.1 Detecting runtime errors

#7.3.1 检测运行时错误

> Our tree-walk interpreter evaluates nested expressions using recursive method calls, and we need to unwind out of all of those. Throwing an exception in Java is a fine way to accomplish that. However, instead of using Java's own cast failure, we'll define a Lox-specific one so that we can handle it how we want.

我们的树遍历型解释器通过递归方法调用计算嵌套的表达式，而且我们需要能够跳脱出所有的调用层。在Java中抛出异常是实现这一点的好方法。但是，我们不使用Java自己的转换失败错误，而是定义一个Lox专用的错误，这样我们就可以按照我们想要的方式处理它。

> Before we do the cast, we check the object's type ourselves. So, for unary `-`, we add:

在进行强制转换之前，我们先自己检查对象的类型。因此，对于一元操作符 `-`，我们需要添加代码：

lox/Interpreter.java，在visitUnaryExpr()方法中添加：

```
      case MINUS:
        // 新增部分开始
        checkNumberOperand(expr.operator, right);
        // 新增部分结束
        return -(double)right;
```

> The code to check the operand is:

检查操作数的代码如下：

lox/Interpreter.java，在 visitUnaryExpr()方法后添加：

```java
  private void checkNumberOperand(Token operator, Object operand) {
    if (operand instanceof Double) return;
    throw new RuntimeError(operator, "Operand must be a number.");
  }
```

> When the check fails, it throws one of these:

当检查失败时，代码会抛出一个以下的错误：

lox/RuntimeError.java，新建源代码文件：

```java
package com.craftinginterpreters.lox;

class RuntimeError extends RuntimeException {
  final Token token;

  RuntimeError(Token token, String message) {
    super(message);
    this.token = token;
  }
}
```

> Unlike the Java cast exception, our class tracks the token that identifies where in the user's code the runtime error came from. As with static errors, this helps the user know where to fix their code.

与Java转换异常不同，我们的类会跟踪语法标记，可以指明用户代码中抛出运行时错误的位置[9]。与静态错误一样，这有助于用户知道去哪里修复代码。

> We need similar checking for the binary operators. Since I promised you every single line of code needed to implement the interpreters, I'll run through them all.

我们需要对二元运算符进行类似的检查。既然我答应了要展示实现解释器所需的每一行代码，那么我就把它们逐一介绍一遍。

> Greater than:

大于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
        case GREATER:
                  // 新增部分开始
          checkNumberOperands(expr.operator, left, right);
          // 新增部分结束
          return (double)left > (double)right;
```

> Greater than or equal to:

大于等于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
        case GREATER_EQUAL:
                  // 新增部分开始
          checkNumberOperands(expr.operator, left, right);
          // 新增部分结束
          return (double)left >= (double)right;
```

> Less than:

小于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
        case LESS:
                  // 新增部分开始
          checkNumberOperands(expr.operator, left, right);
          // 新增部分结束
          return (double)left < (double)right;
```

> Less than or equal to:

小于等于：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    case LESS_EQUAL:
              // 新增部分开始
      checkNumberOperands(expr.operator, left, right);
      // 新增部分结束
      return (double)left <= (double)right;
```

> Subtraction:

减法：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    case MINUS:
              // 新增部分开始
      checkNumberOperands(expr.operator, left, right);
      // 新增部分结束
      return (double)left - (double)right;
```

> Division:

除法：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
    case SLASH:
              // 新增部分开始
      checkNumberOperands(expr.operator, left, right);
      // 新增部分结束
      return (double)left / (double)right;
```

> Multiplication:

乘法：

lox/Interpreter.java，在 visitBinaryExpr()方法中添加：

```
      case STAR:
                // 新增部分开始
        checkNumberOperands(expr.operator, left, right);
        // 新增部分结束
        return (double)left * (double)right;
```

> All of those rely on this validator, which is virtually the same as the unary one:

所有这些都依赖于下面这个验证器，它实际上与一元验证器相同[10]：

lox/Interpreter.java，在 checkNumberOperand()方法后添加：

```
  private void checkNumberOperands(Token operator, Object left, Object righ
    if (left instanceof Double && right instanceof Double) return;

    throw new RuntimeError(operator, "Operands must be numbers.");
  }
```

> The last remaining operator, again the odd one out, is addition. Since `+` is overloaded for numbers and strings, it already has code to check the types. All we need to do is fail if neither of the two success cases match.

剩下的最后一个运算符，也是最奇怪的一个，就是加法。由于`+`已经对数字和字符串进行重载，其中已经有检查类型的代码。我们需要做的就是在这两种情况都不匹配时失败。

lox/Interpreter.java，在 visitBinaryExpr()方法中替换一行：

```
          return (String)left + (String)right;
        }
        // 替换部分开始
        throw new RuntimeError(expr.operator,
            "Operands must be two numbers or two strings.");
        // 替换部分结束
      case SLASH:
```

> That gets us detecting runtime errors deep in the bowels of the evaluator. The errors are getting thrown. The next step is to write the code that catches them. For that, we need to wire up the Interpreter class into the main Lox class that drives it.

这样我们就可以在计算器的内部检测运行时错误。错误已经被抛出了。下一步就是编写能捕获这些错误的代码。为此，我们需要将Interpreter类连接到驱动它的Lox主类中。

> ## #7.4 Hooking Up the Interpreter

#7.4 连接解释器

> The visit methods are sort of the guts of the Interpreter class, where the real work happens. We need to wrap a skin around them to interface with the rest of the program. The Interpreter's public API is simply one method.

visit方法是Interpreter类的核心部分，真正的工作是在这里进行的。我们需要给它们包上一层皮，以便与程序的其他部分对接。解释器的公共API只是一种方法。

lox/Interpreter.java，在 Interpreter类中添加：

```
  void interpret(Expr expression) {
    try {
      Object value = evaluate(expression);
      System.out.println(stringify(value));
    } catch (RuntimeError error) {
      Lox.runtimeError(error);
    }
  }
```

> This takes in a syntax tree for an expression and evaluates it. If that succeeds, `evaluate()` returns an object for the result value. `interpret()` converts that to a string and shows it to the user. To convert a Lox value to a string, we rely on:

该方法会接收一个表达式对应的语法树，并对其进行计算。如果成功了，`evaluate()`方法会返回一个对象作为结果值。`interpret()`方法将结果转为字符串并展示给用户。要将Lox值转为字符串，我们要依赖下面的方法：

lox/Interpreter.java，在 isEqual()方法后添加：

```java
  private String stringify(Object object) {
    if (object == null) return "nil";

    if (object instanceof Double) {
      String text = object.toString();
      if (text.endsWith(".0")) {
        text = text.substring(0, text.length() - 2);
      }
      return text;
    }

    return object.toString();
  }
```

> This is another of those pieces of code like `isTruthy()` that crosses the membrane between the user's view of Lox objects and their internal representation in Java.

这是一段像`isTruthy()`一样的代码，它连接了Lox对象的用户视图和它们在Java中的内部表示。

> It's pretty straightforward. Since Lox was designed to be familiar to someone coming from Java, things like Booleans look the same in both languages. The two edge cases are `nil`, which we represent using Java's `null`, and numbers.

这很简单。由于Lox的设计旨在使Java使用者熟悉，因此Boolean之类的东西在两种语言中看起来是一样的。只有两种边界情况是 `nil` (我们用Java的 `null` 表示)和数字。

> Lox uses double-precision numbers even for integer values. In that case, they should print without a decimal point. Since Java has both floating point and integer types, it wants you to know which one you're using. It tells you by adding an explicit `.0` to integer-valued doubles. We don't care about that, so we hack it off the end.

Lox即使对整数值也使用双精度数字[11]。在这种情况下，打印时应该不带小数点。 由于Java同时具有浮点型和整型，它希望您知道正在使用的是哪一种类型。它通过在整数值的双数上添加一个明确的 `.0` 来告知用户。我们不关心这个，所以我们把它去掉。

> ### #7.4.1 Reporting runtime errors

#7.4.1 报告运行时错误

> If a runtime error is thrown while evaluating the expression, `interpret()` catches it. This lets us report the error to the user and then gracefully continue. All of our existing error reporting code lives in the Lox class, so we put this method there too:

如果在计算表达式时出现了运行时错误， `interpret()` 方法会将其捕获。这样我们可以向用户报告这个错误，然后优雅地继续执行。我们现有的所有错误报告代码都在Lox类中，所以我们也把这个方法放在其中：

lox/Lox.java，在 error()方法后添加：

```
  static void runtimeError(RuntimeError error) {
    System.err.println(error.getMessage() +
        "\n[line " + error.token.line + "]");
```

```
      hadRuntimeError = true;
    }
```

> We use the token associated with the RuntimeError to tell the user what line of code was executing when the error occurred. Even better would be to give the user an entire call stack to show how they *got* to be executing that code. But we don't have function calls yet, so I guess we don't have to worry about it.

我们使用与RuntimeError关联的标记来告诉用户错误发生时正在执行哪一行代码。更好的做法是给用户一个完整的调用堆栈，来显示他们是如何执行该代码的。但我们目前还没有函数调用，所以我想我们不必担心这个问题。

> After showing the error, `runtimeError()` sets this field:

展示错误之后，`runtimeError()`会设置以下字段：

lox/Lox.java，在 Lox类中添加：

```
  static boolean hadError = false;
  // 新增部分开始
  static boolean hadRuntimeError = false;
  // 新增部分结束
  public static void main(String[] args) throws IOException {
```

> That field plays a small but important role.

这个字段担任着很小但很重要的角色。

lox/Lox.java，在 runFile()方法中添加：

```
    run(new String(bytes, Charset.defaultCharset()));

    // Indicate an error in the exit code.
    if (hadError) System.exit(65);
    // 新增部分开始
    if (hadRuntimeError) System.exit(70);
```

```
      // 新增部分结束
    }
```

> If the user is running a Lox script from a file and a runtime error occurs, we set an exit code when the process quits to let the calling process know. Not everyone cares about shell etiquette, but we do.

如果用户从文件中运行Lox脚本，并且发生了运行时错误，我们在进程退出时设置一个退出码，以便让调用进程知道。不是每个人都在乎shell的规矩，但我们在乎[12]。

> ### #7.4.2 Running the interpreter

#7.4.2 运行解释器

> Now that we have an interpreter, the Lox class can start using it.

现在我们有了解释器，Lox类可以开始使用它了。

lox/Lox.java，在 Lox类中添加：

```
public class Lox {
  // 新增部分开始
  private static final Interpreter interpreter = new Interpreter();
  // 新增部分结束
  static boolean hadError = false;
```

> We make the field static so that successive calls to `run()` inside a REPL session reuse the same interpreter. That doesn't make a difference now, but it will later when the interpreter stores global variables. Those variables should persist throughout the REPL session.

我们把这个字段设置为静态的，这样在一个REPL会话中连续调用 `run()` 时就会重复使用同一个解释器。目前这一点没有什么区别，但以后当解释器需要存储

全局变量时就会有区别。这些全局变量应该在整个REPL会话中持续存在。

> Finally, we remove the line of temporary code from the last chapter for printing the syntax tree and replace it with this:

最后，我们删除上一章中用于打印语法树的那行临时代码，并将其替换为：

lox/Lox.java，在 run()方法中替换一行：

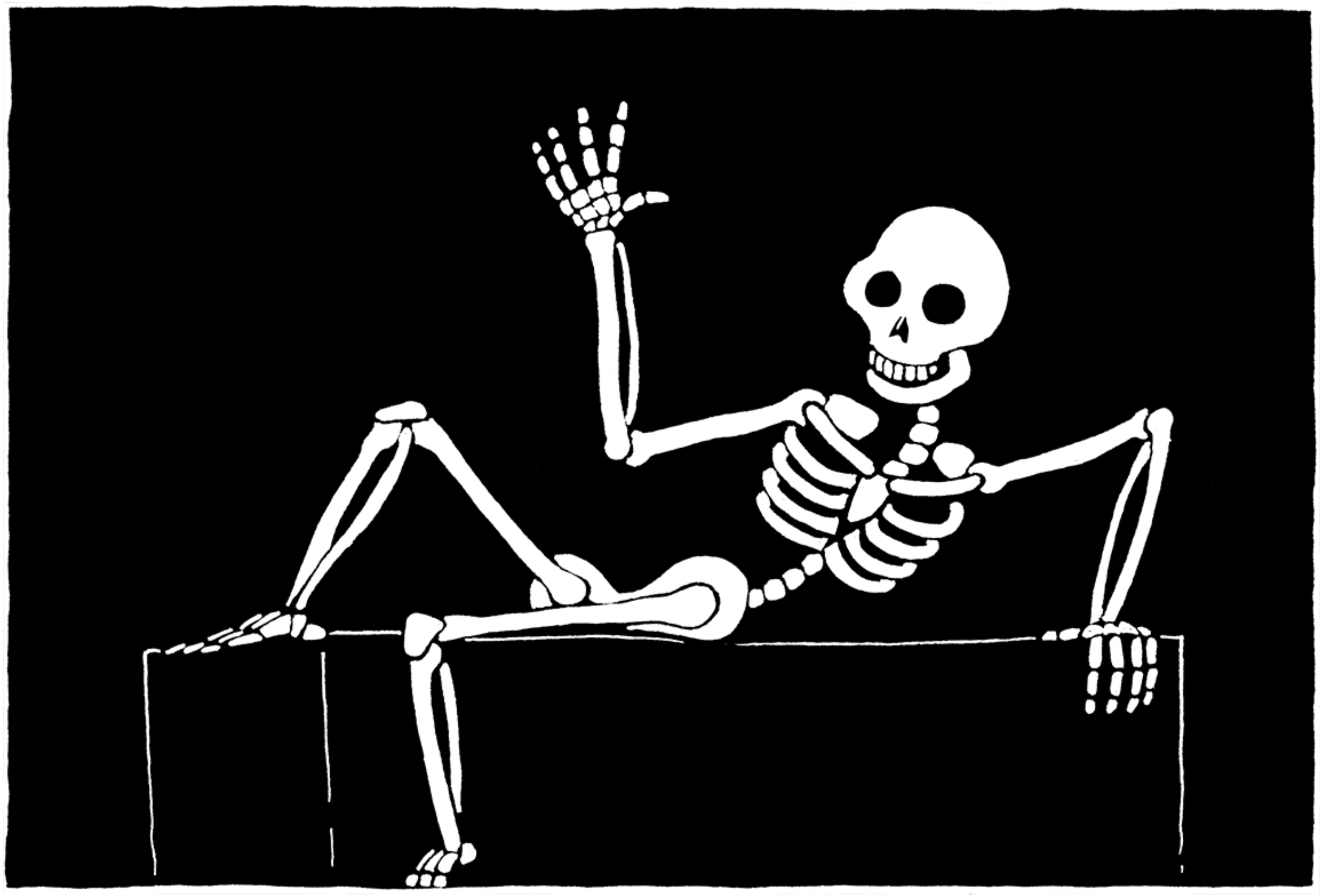
```
    // Stop if there was a syntax error.
    if (hadError) return;
    // 替换部分开始
    interpreter.interpret(expression);
    // 替换部分结束
  }
```

> We have an entire language pipeline now: scanning, parsing, and execution. Congratulations, you now have your very own arithmetic calculator.

我们现在有一个完整的语言管道：扫描、解析和执行。恭喜你，你现在有了你自己的算术计算器。

> As you can see, the interpreter is pretty bare bones. But the Interpreter class and the Visitor pattern we've set up today form the skeleton that later chapters will stuff full of interesting guts—variables, functions, etc. Right now, the interpreter doesn't do very much, but it's alive!

如您所见，这个解释器是非常简陋的。但是我们今天建立的解释器类和访问者模式构成了一个骨架，后面的章节中将填充入有趣的内容（变量，函数等）。现在，解释器的功能并不多，但它是活的!



#CHALLENGES

#习题

1、Allowing comparisons on types other than numbers could be useful. The operators might have a reasonable interpretation for strings. Even comparisons among mixed types, like `3 < "pancake"` could be handy to enable things like ordered collections of heterogeneous types. Or it could simply lead to bugs and confusion.

Would you extend Lox to support comparing other types? If so, which pairs of types do you allow and how do you define their ordering? Justify your

> choices and compare them to other languages.

1、允许对数字之外的类型进行比较可能是个有用的特性。操作符可能对字符串有合理的解释。即使是混合类型之间的比较，如 `3<"pancake"` ，也可以方便地支持异构类型的有序集合。否则可能导致错误和混乱。

你是否会扩展Lox以支持对其他类型的比较？如果是，您允许哪些类型间的比较，以及如何定义它们的顺序？证明你的选择并与其他语言进行比较。

> 2、Many languages define `+` such that if *either* operand is a string, the other is converted to a string and the results are then concatenated. For example, `"scone" + 4` would yield `scone4`. Extend the code in `visitBinaryExpr()` to support that.

2、许多语言对 `+` 的定义是，如果其中一个操作数是字符串，另一个操作数就会被转换成字符串，然后将两个结果拼接起来。例如， `"scone "+4` 的结果应该是 `scone4` 。扩展 `visitBinaryExpr()` 中的代码以支持该特性。

> 3、What happens right now if you divide a number by zero? What do you think should happen? Justify your choice. How do other languages you know handle division by zero, and why do they make the choices they do?
>
> Change the implementation in `visitBinaryExpr()` to detect and report a runtime error for this case.

3、如果你用一个数除以0会发生什么？你认为应该发生什么？证明你的选择。你知道的其他语言是如何处理除零的，为什么他们会做出这样的选择？

更改 `visitBinaryExpr()` 中的实现代码，以检测并报告运行时错误。

> ## #DESIGN NOTE: STATIC AND DYNAMIC TYPING

#设计笔记：静态类型和动态类型

Some languages, like Java, are statically typed which means type errors are detected and reported at compile time before any code is run. Others, like Lox, are dynamically typed and defer checking for type errors until runtime right before an operation is attempted. We tend to consider this a black-and-white choice, but there is actually a continuum between them.

It turns out even most statically typed languages do *some* type checks at runtime. The type system checks most type rules statically, but inserts runtime checks in the generated code for other operations.

For example, in Java, the *static* type system assumes a cast expression will always safely succeed. After you cast some value, you can statically treat it as the destination type and not get any compile errors. But downcasts can fail, obviously. The only reason the static checker can presume that casts always succeed without violating the language's soundness guarantees, is because the cast is checked *at runtime* and throws an exception on failure.

A more subtle example is covariant arrays in Java and C#. The static subtyping rules for arrays allow operations that are not sound. Consider:

```java
Object[] stuff = new Integer[1];
stuff[0] = "not an int!";
```

This code compiles without any errors. The first line upcasts the Integer array and stores it in a variable of type Object array. The second line stores a string in one of its cells. The Object array type statically allows that—strings *are* Objects—but the actual Integer array that `stuff` refers to at runtime should never have a string in it! To avoid that catastrophe, when you store a value in an array, the JVM does a *runtime* check to make sure it's an allowed type. If not, it throws an ArrayStoreException.

Java could have avoided the need to check this at runtime by disallowing the cast on the first line. It could make arrays *invariant* such that an array of Integers is *not* an array of Objects. That's statically sound, but it prohibits common and safe patterns of code that only read from arrays. Covariance is safe if you never *write* to the array. Those patterns were particularly important for usability in Java 1.0 before it supported generics.

> James Gosling and the other Java designers traded off a little static safety and performance—those array store checks take time—in return for some flexibility.
>
> There are few modern statically typed languages that don't make that trade-off *somewhere*. Even Haskell will let you run code with non-exhaustive matches. If you find yourself designing a statically typed language, keep in mind that you can sometimes give users more flexibility without sacrificing *too* many of the benefits of static safety by deferring some type checks until runtime.
>
> On the other hand, a key reason users choose statically typed languages is because of the confidence the language gives them that certain kinds of errors can *never* occur when their program is run. Defer too many type checks until runtime, and you erode that confidence.

有些语言，如Java，是静态类型的，这意味着在任何代码运行之前，会在编译时检测和报告类型错误。其他语言，如Lox，是动态类型的，将类型错误的检查推迟到运行时尝试执行具体操作之前。我们倾向于认为这是一个非黑即白的选择，但实际上它们之间是连续统一的。

事实证明，大多数静态类型的语言也会在运行时进行一些类型检查。类型系统会静态地检查多数类型规则，但在生成的代码中插入了运行时检查以支持其它操作。

例如，在Java中，静态类型系统会假定强制转换表达式总是能安全地成功执行。在转换某个值之后，可以将其静态地视为目标类型，而不会出现任何编译错误。但向下转换显然会失败。静态检查器之所以能够在不违反语言的合理性保证的情况下假定转换总是成功的，唯一原因是，强制转换操作会在运行时进行类型检查，并在失败时抛出异常。

一个更微妙的例子是Java和c#中的协变数组。数组的静态子类型规则允许不健全的操作。考虑以下代码：

```
Object[] stuff = new Integer[1];
stuff[0] = "not an int!";
```

这段代码在编译时没有任何错误。第一行代码将整数数组向上转换并存储到一个对象数组类型的变量中。第二行代码将字符串存储在其中一个单元格里。对象数组类型静态地允许该操作——字符串也是对象——但是 `stuff` 在运行时引用的整数数组中不应该包含字符串！为了避免这种灾难，当你在数组中存储一个值时，JVM会进行运行时检查，以确保该值是允许的类型。如果不是，则抛出ArrayStoreException。

Java可以通过禁止对第一行进行强制转换来避免在运行时检查这一点。它可以使数组保持不变，这样整型数组就不是对象数组。这在静态类型角度是合理的，但它禁止了只从数组中读取数据的常见安全的代码模式。如果你从来不向数组写入内容，那么协变是安全的。在支持泛型之前，这些模式对于Java 1.0的可用性尤为重要。

James Gosling和其他Java设计师牺牲了一点静态安全和性能（这些数组存储检查需要花费时间）来换取一些灵活性。

几乎所有的现代静态类型语言都在某些方面做出了权衡。即使Haskell也允许您运行非穷举性匹配的代码。如果您自己正在设计一种静态类型语言，请记住，有时你可以通过将一些类型检查推迟到运行时来给用户更多的灵活性，而不会牺牲静态安全的太多好处。

另一方面，用户选择静态类型语言的一个关键原因是，这种语言让他们相信：在他们的程序运行时，某些类型的错误永远不会发生。将过多的类型检查推迟到运行时，就会破坏用户的这种信心。

---

[^1]

在这里，我基本可以互换地使用 "值 "和 "对象"。稍后在C解释器中，我们会对它们稍作区分，但这主要是针对实现的两个不同方面（本地数据和堆分配数据）使用不同的术语。从用户的角度来看，这些术语是同义的。

[^2]

我们需要对值做的另一件事是管理它们的内存，Java也能做到这一点。方便的对象表示和非常好的垃圾收集器是我们用Java编写第一个解释器的主要原因。

[^3]

在下一章，当我们实现变量时，我们将添加标识符表达式，它也是叶子节点。

[^4]

有些解析器不为圆括号单独定义树节点。相应地，在解析带圆括号的表达式时，它们只返回内部表达式的节点。在Lox中，我们确实为圆括号创建了一个节点，因为稍后我们需要用它来正确处理赋值表达式的左值。

[^5]

在JavaScript中，字符串是真的，但空字符串不是。数组是真的，但空数组是……也是真的。数字0是假的，但字符串 "0 "是真的。

在 Python 中，空字符串是假的，就像在 JS 中一样，但其他空序列也是假的。

在PHP中，数字0和字符串 "0 "都是假的。大多数其他非空字符串是真实的。明白了吗?

[^6]

你是否注意到我们在这里固定了语言语义的一个细微的点？在二元表达式中，我们按从左到右的顺序计算操作数。如果这些操作数有副作用，那这个选择应该是用户可见的，所以这不是一个简单的实现细节。如果我们希望我们的两个解释器是一致的（提示：我们是一致的），我们就需要确保 clox 也是这样做的。

[^7]

你希望这个表达式的计算结果是什么？`(0 / 0) == (0 / 0)`。根据IEEE 754（它规定了双精度数的行为），用0除以0会得到特殊的**NaN**（不是一个数字）值。奇怪的是，NaN不等于它自己。

在Java中，基本类型double的`==`操作满足该规范，但是封装类Double的`equals()`方法不满足。Lox使用了后者，因此不遵循IEEE。这类微妙的不兼容问题占据了语言开发者生活中令人沮丧的一部分。

[^8]

我们完全可以不检测或报告一个类型错误。当你在C语言中把一个指针转换到与实际被指向的数据不匹配的类型上，C语言就是这样做的。C语言通过

允许这样的操作获得了灵活性和速度，但它也是出了名的危险。一旦你错误地解释了内存中的数据，一切都完了。很少有现代语言接受这样的不安全操作。相反，大多数语言都是**内存安全**的，并通过静态和运行时检查的组合，确保程序永远不会错误地解释存储在内存中的值。

[^9]

我承认 "RuntimeError "这个名字令人困惑，因为Java定义了一个RuntimeException类。关于构建解释器的一件恼人的事情就是，您使用的名称经常与实现语言中已经使用的名称冲突。等我们支持Lox类就好了。

[^10]

另一个微妙的语义选择：在检查两个操作数的类型之前，我们先计算这两个操作数。假设我们有一个函数 `say()` ，它会打印其介绍的参数，然后返回。我们使用这个函数写出表达式： `say("left") - say("right");` 。我们的解释器在报告运行时错误之前会先打印"left"和"right"。相对地，我们也可以指定在计算右操作数之前先检查左操作数。

[^11]

同样，我们要处理这种数字的边界情况，以确保jlox和clox的工作方式相同。像这样处理语言的一个奇怪的边界可能会让你抓狂，但这是工作的一个重要部分。用户会有意或无意地依赖于这些细节，如果实现不一致，他们的程序在不同的解释器上运行时将会中断。

[^12]

如果用户正在运行REPL，则我们不必跟踪运行时错误。在错误被报告之后，我们只需要循环，让用户输入新的代码，然后继续执行。