#10.函数Functions

> And that is also the way the human mind works—by the compounding of old ideas into new structures that become new ideas that can themselves be used in compounds, and round and round endlessly, growing ever more remote from the basic earthbound imagery that is each language's soil.
>
> —— Douglas R. Hofstadter, *I Am a Strange Loop*

这也是人类思维的运作方式——将旧的想法复合成为新结构，成为新的想法，而这些想法本身又可以被用于复合，循环往复，无休无止，越来越远离每一种语言赖以生存的基本的土壤。

> This chapter marks the culmination of a lot of hard work. The previous chapters add useful functionality in their own right, but each also supplies a piece of a puzzle. We'll take those pieces—expressions, statements, variables, control flow, and lexical scope—add a couple more, and assemble them all into support for real user-defined functions and function calls.

这一章标志着很多艰苦工作的一个高潮。在前面的章节中，各自添加了一些有用的功能，但是每一章也都提供了一个拼图的碎片。我们整理这些碎片——表达式、语句、变量、控制流和词法作用域，再加上其它功能，并把他们组合起来，以支持真正的用户定义函数和函数调用。

#10.1 Function Calls

#10.1 函数调用

> You're certainly familiar with C-style function call syntax, but the grammar is more subtle than you may realize. Calls are typically to named functions like:

你肯定熟悉C语言风格的函数调用语法，但其语法可能比你意识到的更微妙。调用通常是指向命名的函数，例如：

```
average(1, 2);
```

> But the name of the function being called isn't actually part of the call syntax. The thing being called—the **callee**—can be any expression that evaluates to a function. (Well, it does have to be a pretty *high precedence* expression, but parentheses take care of that.) For example:

但是被调用函数的名称实际上并不是调用语法的一部分。被调用者（**callee**）可以是计算结果为一个函数的任何表达式。(好吧，它必须是一个非常高优先级的表达式，但是圆括号可以解决这个问题。)例如：

```
getCallback()();
```

> There are two call expressions here. The first pair of parentheses has `getCallback` as its callee. But the second call has the entire `getCallback()` expression as its callee. It is the parentheses following an expression that indicate a function call. You can think of a call as sort of like a postfix operator that starts with `(`.

这里有两个函数调用。第一对括号将 `getCallback` 作为其被调用者。但是第二对括号将整个 `getCallback()` 表达式作为其被调用者。表达式后面的小括号表示函数调用，你可以把调用看作是一种以 `(` 开头的后缀运算符。

> This "operator" has higher precedence than any other operator, even the unary ones. So we slot it into the grammar by having the `unary` rule bubble up to a new `call` rule.

这个“运算符”比其它运算符（包括一元运算符）有更高的优先级。所以我们通过让 `unary` 规则跳转到新的 `call` 规则，将其添加到语法中[1]。

```
unary          → ( "!" | "-" ) unary | call ;
call           → primary ( "(" arguments? ")" )* ;
```

> This rule matches a primary expression followed by zero or more function calls. If there are no parentheses, this parses a bare primary expression. Otherwise, each call is recognized by a pair of parentheses with an optional list of arguments inside. The argument list grammar is:

该规则匹配一个基本表达式，后面跟着0个或多个函数调用。如果没有括号，则解析一个简单的基本表达式。否则，每一对圆括号都表示一个函数调用，圆括号内有一个可选的参数列表。参数列表语法是：

```
arguments      → expression ( "," expression )* ;
```

> This rule requires at least one argument expression, followed by zero or more other expressions, each preceded by a comma. To handle zero-argument calls, the `call` rule itself considers the entire `arguments` production to be optional.

这个规则要求至少有一个参数表达式，后面可以跟0个或多个其它表达式，每两个表达式之间用 `,` 分隔。为了处理无参调用， `call` 规则本身认为整个 `arguments` 生成式是可选的。

> I admit, this seems more grammatically awkward than you’d expect for the incredibly common “zero or more comma-separated things” pattern. There are some sophisticated metasyntaxes that handle this better, but in our BNF and in many language specs I’ve seen, it is this cumbersome.

我承认，对于极其常见的 "零或多个逗号分隔的事物 "模式来说，这在语法上似乎比你想象的更难处理。有一些复杂的元语法可以更好地处理这个问题，但在我们的BNF和我见过的许多语言规范中，它就是如此的麻烦。

> Over in our syntax tree generator, we add a new node.

在我们的语法树生成器中，我们添加一个新节点。

tool/GenerateAst.java，在 main()方法中添加代码：

```
      "Binary   : Expr left, Token operator, Expr right",
      // 新增部分开始
      "Call     : Expr callee, Token paren, List<Expr> arguments",
      // 新增部分结束
      "Grouping : Expr expression",
```

> It stores the callee expression and a list of expressions for the arguments. It also stores the token for the closing parenthesis. We'll use that token's location when we report a runtime error caused by a function call.

它存储了被调用者表达式和参数表达式列表，同时也保存了右括号标记。当我们报告由函数调用引起的运行时错误时，会使用该标记的位置。

> Crack open the parser. Where `unary()` used to jump straight to `primary()`, change it to call, well, `call()`.

打开解析器，原来 `unary()` 直接跳转到 `primary()` 方法，将其修改为调用 `call()`。

lox/Parser.java，在 unary()方法中替换一行：

```
      return new Expr.Unary(operator, right);
    }
    // 替换部分开始
    return call();
    // 替换部分结束
  }
```

> Its definition is:

该方法定义为：

lox/Parser.java，在 unary()方法后添加[2]：

```
  private Expr call() {
    Expr expr = primary();

    while (true) {
      if (match(LEFT_PAREN)) {
        expr = finishCall(expr);
      } else {
        break;
      }
    }

    return expr;
  }
```

> The code here doesn't quite line up with the grammar rules. I moved a few things around to make the code cleaner—one of the luxuries we have with a handwritten parser. But it's roughly similar to how we parse infix operators. First, we parse a primary expression, the "left operand" to the call. Then, each time we see a `(`, we call `finishCall()` to parse the call expression using the previously parsed expression as the callee. The returned expression becomes the new `expr` and we loop to see if the result is itself called.

这里的代码与语法规则并非完全一致。为了保持代码简洁，我调整了一些东西——这是我们手写解析器的优点之一。但它与我们解析中缀运算符的方式类似。首先，我们解析一个基本表达式，即调用的左操作数。然后，每次看到 `(`，我们就调用 `finishCall()` 解析调用表达式，并使用之前解析出的表达式作为被调用者。返回的表达式成为新的 `expr`，我们循环检查其结果是否被调用。

> The code to parse the argument list is in this helper:

解析参数列表的代码在下面的工具方法中：

lox/Parser.java，在 unary()方法后添加：

```
    private Expr finishCall(Expr callee) {
      List<Expr> arguments = new ArrayList<>();
      if (!check(RIGHT_PAREN)) {
        do {
          arguments.add(expression());
        } while (match(COMMA));
      }

      Token paren = consume(RIGHT_PAREN,
                            "Expect ')' after arguments.");

      return new Expr.Call(callee, paren, arguments);
    }
```

> This is more or less the `arguments` grammar rule translated to code, except that we also handle the zero-argument case. We check for that case first by seeing if the next token is `)`. If it is, we don't try to parse any arguments.

这或多或少是 `arguments` 语法规则翻译成代码的结果，除了我们这里还处理了无参情况。我们首先判断下一个标记是否 `)` 来检查这种情况。如果是，我们就不会尝试解析任何参数。

> Otherwise, we parse an expression, then look for a comma indicating that there is another argument after that. We keep doing that as long as we find commas after each expression. When we don't find a comma, then the argument list must be done and we consume the expected closing parenthesis. Finally, we wrap the callee and those arguments up into a call AST node.

如果不是，我们就解析一个表达式，然后寻找逗号（表明后面还有一个参数）。只要我们在表达式后面发现逗号，就会继续解析表达式。当我们找不到逗号时，说明参数列表已经结束，我们继续消费预期的右括号。最终，我们将被调用者和这些参数封装成一个函数调用的AST节点。

> #10.1.1 Maximum argument counts

#10.1.1 最大参数数量

> Right now, the loop where we parse arguments has no bound. If you want to call a function and pass a million arguments to it, the parser would have no problem with it. Do we want to limit that?

现在，我们解析参数的循环是没有边界的。如果你想调用一个函数并向其传递一百万个参数，解析器不会有任何问题。我们要对此进行限制吗?

> Other languages have various approaches. The C standard says a conforming implementation has to support *at least* 127 arguments to a function, but doesn't say there's any upper limit. The Java specification says a method can accept *no more than* 255 arguments.

其它语言采用了不同的策略。C语言标准要求在符合标准的实现中，一个函数至少要支持127个参数，但是没有指定任何上限。Java规范规定一个方法可以接受不超过255个参数[3]。

> Our Java interpreter for Lox doesn't really need a limit, but having a maximum number of arguments will simplify our bytecode interpreter in Part III. We want our two interpreters to be compatible with each other, even in weird corner cases like this, so we'll add the same limit to jlox.

Lox的Java解释器实际上并不需要限制，但是设置一个最大的参数数量限制可以简化第三部分中的字节码解释器。即使是在这样奇怪的地方里，我们也希望两个解释器能够相互兼容，所以我们为jlox添加同样的限制。

lox/Parser.java，在 finishCall()方法中添加:

```
do {
  // 新增部分开始
```

```
        if (arguments.size() >= 255) {
          error(peek(), "Can't have more than 255 arguments.");
        }
        // 新增部分结束
        arguments.add(expression());
```

> Note that the code here *reports* an error if it encounters too many arguments, but it doesn't *throw* the error. Throwing is how we kick into panic mode which is what we want if the parser is in a confused state and doesn't know where it is in the grammar anymore. But here, the parser is still in a perfectly valid state—it just found too many arguments. So it reports the error and keeps on keepin' on.

请注意，如果发现参数过多，这里的代码会*报告*一个错误，但是不会*抛出*该错误。抛出错误是进入恐慌模式的方法，如果解析器处于混乱状态，不知道自己在语法中处于什么位置，那这就是我们想要的。但是在这里，解析器仍然处于完全有效的状态，只是发现了太多的参数。所以它会报告这个错误，并继续执行解析。

> #### 10.1.2 Interpreting function calls

10.1.2 解释函数调用

> We don't have any functions we can call, so it seems weird to start implementing calls first, but we'll worry about that when we get there. First, our interpreter needs a new import.

我们还没有任何可以调用的函数，所以先实现函数调用似乎有点奇怪，但是这个问题我们后面再考虑。首先，我们的解释器需要引入一个新依赖。

lox/Interpreter.java

```
import java.util.ArrayList;
import java.util.List;
```

> As always, interpretation starts with a new visit method for our new call expression node.

跟之前一样，解释工作从新的调用表达式节点对应的新的visit方法开始[4]。

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

```
@Override
public Object visitCallExpr(Expr.Call expr) {
  Object callee = evaluate(expr.callee);

  List<Object> arguments = new ArrayList<>();
  for (Expr argument : expr.arguments) {
    arguments.add(evaluate(argument));
  }

  LoxCallable function = (LoxCallable)callee;
  return function.call(this, arguments);
}
```

> First, we evaluate the expression for the callee. Typically, this expression is just an identifier that looks up the function by its name, but it could be anything. Then we evaluate each of the argument expressions in order and store the resulting values in a list.

首先，对被调用者的表达式求值。通常情况下，这个表达式只是一个标识符，可以通过它的名字来查找函数。但它可以是任何东西。然后，我们依次对每个参数表达式求值，并将结果值存储在一个列表中。

> Once we've got the callee and the arguments ready, all that remains is to perform the call. We do that by casting the callee to a LoxCallable and then invoking a `call()` method on it. The Java representation of any Lox object that can be called like a function will implement this interface. That includes user-defined functions, naturally, but also class objects since classes are "called" to construct new instances. We'll also use it for one more purpose shortly.

一旦我们准备好被调用者和参数，剩下的就是执行函数调用。我们将被调用者转换为LoxCallable，然后对其调用 `call()` 方法来实现。任何可以像函数一样被调用的Lox对象的Java表示都要实现这个接口。这自然包括用户定义的函数，但也包括类对象，因为类会被 "调用 "来创建新的实例。稍后我们还将把它用于另一个目的。

> There isn't too much to this new interface.

这个新接口中没有太多内容。

lox/LoxCallable.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;

interface LoxCallable {
  Object call(Interpreter interpreter, List<Object> arguments);
}
```

> We pass in the interpreter in case the class implementing `call()` needs it. We also give it the list of evaluated argument values. The implementer's job is then to return the value that the call expression produces.

我们会传入解释器，以防实现 `call()` 方法的类会需要它。我们也会提供已求值的参数值列表。接口实现者的任务就是返回调用表达式产生的值。

> ### #10.1.3 Call type errors

#10.1.3 调用类型错误

> Before we get to implementing LoxCallable, we need to make the visit method a little more robust. It currently ignores a couple of failure modes

> that we can't pretend won't occur. First, what happens if the callee isn't actually something you can call? What if you try to do this:

在实现 `LoxCallable` 之前，必须先强化一下我们的visit方法。这个方法忽略了两个可能出现的错误场景。第一个，如果被调用者无法被调用，会发生什么？比如：

```
"totally not a function"();
```

> Strings aren't callable in Lox. The runtime representation of a Lox string is a Java string, so when we cast that to LoxCallable, the JVM will throw a ClassCastException. We don't want our interpreter to vomit out some nasty Java stack trace and die. Instead, we need to check the type ourselves first.

在Lox中，字符串不是可调用的数据类型。Lox字符串在运行时中的本质其实是java字符串，所以当我们把它当作 `LoxCallable` 处理的时候，JVM就会抛出 `ClassCastException` 。我们并不想让我们的解释器吐出一坨java堆栈信息然后挂掉。所以，我们自己必须先做一次类型检查。

lox/Interpreter.java，在visitCallExpr接口中新增：

```
// 新增部分开始
if (!(callee instanceof LoxCallable)) {
  throw new RuntimeError(expr.paren,
      "Can only call functions and classes.");
}
// 新增部分结束
LoxCallable function = (LoxCallable)callee;
```

We still throw an exception, but now we're throwing our own exception type, one that the interpreter knows to catch and report gracefully.

我们的实现同样也是抛出错误，但它们能够被解释器捕获并优雅地展示出来。

> ### #10.1.4 Checking arity

#10.1.4 检查元数

> The other problem relates to the function's **arity**. Arity is the fancy term for the number of arguments a function or operation expects. Unary operators have arity one, binary operators two, etc. With functions, the arity is determined by the number of parameters it declares.

另一个问题与函数的**元数**有关。元数是一个花哨的术语，指一个函数或操作所期望的参数数量。一元运算符的元数是1，二元运算符是2，等等。对于函数来说，元数由函数声明的参数数量决定。

```
fun add(a, b, c) {
  print a + b + c;
}
```

> This function defines three parameters, `a`, `b`, and `c`, so its arity is three and it expects three arguments. So what if you try to call it like this:

这个函数定义了三个形参，`a`、`b`和`c`，所以它的元数是3，而且它期望有3个参数。那么如果你用下面的方式调用该函数会怎样：

```
add(1, 2, 3, 4); // Too many.
add(1, 2);       // Too few.
```

> Different languages take different approaches to this problem. Of course, most statically typed languages check this at compile time and refuse to compile the code if the argument count doesn't match the function's arity. JavaScript discards any extra arguments you pass. If you don't pass enough, it fills in the missing parameters with the magic sort-of-like-null-but-not-really value `undefined`. Python is stricter. It raises a runtime error if the argument list is too short or too long.

不同的语言对这个问题采用了不同的方法。当然，大多数静态类型的语言在编译时都会检查这个问题，如果实参与函数元数不匹配，则拒绝编译代码。JavaScript会丢弃你传递的所有多余参数。如果你没有传入的参数数量不足，

它就会用神奇的与 `null` 类似但并不相同的值 `undefined` 来填补缺少的参数。Python更严格。如果参数列表太短或太长，它会引发一个运行时错误。

> I think the latter is a better approach. Passing the wrong number of arguments is almost always a bug, and it's a mistake I do make in practice. Given that, the sooner the implementation draws my attention to it, the better. So for Lox, we'll take Python's approach. Before invoking the callable, we check to see if the argument list's length matches the callable's arity.

我认为后者是一种更好的方法。传递错误的参数数量几乎总是一个错误，这也是我在实践中确实犯的一个错误。有鉴于此，语言实现能越早引起用户的注意就越好。所以对于Lox，我们将采取Python的方法。在执行可调用方法之前，我们检查参数列表的长度是否与可调用方法的元数相符。

lox/Interpreter.java，在 visitCallExpr()方法中添加代码：

```
    LoxCallable function = (LoxCallable)callee;
    // 新增部分开始
    if (arguments.size() != function.arity()) {
      throw new RuntimeError(expr.paren, "Expected " +
          function.arity() + " arguments but got " +
          arguments.size() + ".");
    }
    // 新增部分结束
    return function.call(this, arguments);
```

> That requires a new method on the LoxCallable interface to ask it its arity.

这就需要在 `LoxCallable` 接口中增加一个新方法来查询函数的元数。

lox/LoxCallable.java，在LoxCallable接口中新增：

```
interface LoxCallable {
  // 新增部分开始
  int arity();
  // 新增部分结束
  Object call(Interpreter interpreter, List<Object> arguments);
```

> We *could* push the arity checking into the concrete implementation of `call()`. But, since we'll have multiple classes implementing LoxCallable, that would end up with redundant validation spread across a few classes. Hoisting it up into the visit method lets us do it in one place.

我们可以在`call()`方法的具体实现中做元数检查。但是，由于我们会有多个实现LoxCallable的类，这将导致冗余的验证分散在多个类中。把它提升到访问方法中，这样我们可以在一个地方完成该功能。

> ## #10.2 Native Functions

#10.2 原生函数（本地函数）

> We can theoretically call functions, but we have no functions to call yet. Before we get to user-defined functions, now is a good time to introduce a vital but often overlooked facet of language implementations—**native functions**. These are functions that the interpreter exposes to user code but that are implemented in the host language (in our case Java), not the language being implemented (Lox).

理论上我们可以调用函数了，但是我们还没有可供调用的函数。在我们实现用户自定义函数之前，现在正好可以介绍语言实现中一个重要但经常被忽视的方面——**原生函数（本地函数）**。这些函数是解释器向用户代码公开的，但它们是用宿主语言(在我们的例子中是Java)实现的，而不是正在实现的语言(Lox)。

> Sometimes these are called **primitives**, **external functions**, or **foreign functions**. Since these functions can be called while the user's program is running, they form part of the implementation's runtime. A lot of programming language books gloss over these because they aren't conceptually interesting. They're mostly grunt work.

有时这些函数也被称为**原语**、**外部函数**或**外来函数**[5]。由于这些函数可以在用户程序运行的时候被调用，因此它们构成了语言运行时的一部分。许多编程语言书籍都掩盖了这些内容，因为它们在概念上并不有趣。它们主要是一些比较繁重的工作。

> But when it comes to making your language actually good at doing useful stuff, the native functions your implementation provides are key. They provide access to the fundamental services that all programs are defined in terms of. If you don't provide native functions to access the file system, a user's going to have a hell of a time writing a program that reads and displays a file.

但是说到让你的语言真正擅长做有用的事情，语言提供的本地函数是关键[6]。本地函数提供了对基础服务的访问，所有的程序都是根据这些服务来定义的。如果你不提供访问文件系统的本地函数，那么用户在写一个读取和显示文件的程序时就会有很大的困难。

> Many languages also allow users to provide their own native functions. The mechanism for doing so is called a **foreign function interface** (**FFI**), **native extension**, **native interface**, or something along those lines. These are nice because they free the language implementer from providing access to every single capability the underlying platform supports. We won't define an FFI for jlox, but we will add one native function to give you an idea of what it looks like.

许多语言还允许用户提供自己的本地函数。这样的机制称为**外来函数接口(FFI)**、**本机扩展**、**本机接口**或类似的东西。这些机制很好，因为它们使语言实现者无需提供对底层平台所支持的每一项功能的访问。我们不会为 jlox 定义一个 FFI，但我们会添加一个本地函数，让你知道它是什么样子。

> ### #10.2.1 Telling time

#10.2.1 报时

> When we get to Part III and start working on a much more efficient implementation of Lox, we're going to care deeply about performance. Performance work requires measurement, and that in turn means **benchmarks**. These are programs that measure the time it takes to exercise some corner of the interpreter.

当我们进入第三部分，开始着手开发更有效的Lox实现时，我们就会非常关心性能。性能需要测量，这也就意味着需要**基准测试**。这些代码就是用于测量解释器执行某些代码时所花费的时间。

> We could measure the time it takes to start up the interpreter, run the benchmark, and exit, but that adds a lot of overhead—JVM startup time, OS shenanigans, etc. That stuff does matter, of course, but if you're just trying to validate an optimization to some piece of the interpreter, you don't want that overhead obscuring your results.

我们可以测量启动解释器、运行基准测试代码并退出所消耗的时间，但是这其中包括很多时间开销——JVM启动时间，操作系统欺诈等等。当然，这些东西确实很重要，但如果您只是试图验证对解释器某个部分的优化，你肯定不希望这些多余的时间开销掩盖你的结果。

> A nicer solution is to have the benchmark script itself measure the time elapsed between two points in the code. To do that, a Lox program needs to be able to tell time. There's no way to do that now—you can't implement a useful clock "from scratch" without access to the underlying clock on the computer.

一个更好的解决方案是让基准脚本本身度量代码中两个点之间的时间间隔。要做到这一点，Lox程序需要能够报时。现在没有办法做到这一点——如果不访问计算机上的底层时钟，就无法从头实现一个可用的时钟。

> So we'll add `clock()`, a native function that returns the number of seconds that have passed since some fixed point in time. The difference between

> two successive invocations tells you how much time elapsed between the two calls. This function is defined in the global scope, so let's ensure the interpreter has access to that.

所以我们要添加 `clock()`，这是一个本地函数，用于返回自某个固定时间点以来所经过的秒数。两次连续调用之间的差值可用告诉你两次调用之间经过了多少时间。这个函数被定义在全局作用域内，以确保解释器能够访问这个函数。

lox/Interpreter.java，在 Interpreter类中，替换一行：

```
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
  // 替换部分开始
  final Environment globals = new Environment();
  private Environment environment = globals;
  // 替换部分结束
  void interpret(List<Stmt> statements) {
```

> The `environment` field in the interpreter changes as we enter and exit local scopes. It tracks the *current* environment. This new `globals` field holds a fixed reference to the outermost global environment.

解释器中的 `environment` 字段会随着进入和退出局部作用域而改变，它会跟随当前环境。新加的 `globals` 字段则固定指向最外层的全局作用域。

> When we instantiate an Interpreter, we stuff the native function in that global scope.

当我们实例化一个解释器时，我们将全局作用域中添加本地函数。

lox/Interpreter.java，在 Interpreter类中新增：

```
  private Environment environment = globals;
  // 新增部分开始
  Interpreter() {
    globals.define("clock", new LoxCallable() {
      @Override
```

```
      public int arity() { return 0; }

      @Override
      public Object call(Interpreter interpreter,
                         List<Object> arguments) {
        return (double)System.currentTimeMillis() / 1000.0;
      }

      @Override
      public String toString() { return "<native fn>"; }
    });
  }
  // 新增部分结束
  void interpret(List<Stmt> statements) {
```

> This defines a variable named “clock”. Its value is a Java anonymous class that implements LoxCallable. The `clock()` function takes no arguments, so its arity is zero. The implementation of `call()` calls the corresponding Java function and converts the result to a double value in seconds.

这里有一个名为 `clock` 的变量，它的值是一个实现LoxCallable接口的Java匿名类。这里的 `clock()` 函数不接受参数，所以其元数为0。 `call()` 方法的实现是直接调用Java函数并将结果转换为以秒为单位的double值。

> If we wanted to add other native functions—reading input from the user, working with files, etc.—we could add them each as their own anonymous class that implements LoxCallable. But for the book, this one is really all we need.

如果我们想要添加其它本地函数——读取用户输入，处理文件等等——我们可以依次为它们提供实现LoxCallable接口的匿名类。但是在本书中，这个函数足以满足需要。

> Let’s get ourselves out of the function-defining business and let our users take over . . .

让我们从函数定义的事务中解脱出来，由用户来接管吧。

> ## #10.3 Function Declarations

#10.3 函数声明

> We finally get to add a new production to the `declaration` rule we introduced back when we added variables. Function declarations, like variables, bind a new name. That means they are allowed only in places where a declaration is permitted.

我们终于可以在添加变量时就引入的 `declaration` 规则中添加产生式了。就像变量一样，函数声明也会绑定一个新的名称。这意味中它们只能出现在允许声明的地方。

```
declaration    → funDecl
               | varDecl
               | statement ;
```

> The updated `declaration` rule references this new rule:

更新后的 `declaration` 引用了下面的新规则：

```
funDecl        → "fun" function ;
function       → IDENTIFIER "(" parameters? ")" block ;
```

> The main `funDecl` rule uses a separate helper rule `function`. A function *declaration statement* is the `fun` keyword followed by the actual function-y stuff. When we get to classes, we'll reuse that `function` rule for declaring methods. Those look similar to function declarations, but aren't preceded by `fun`.

主要的 `funDecl` 规则使用了一个单独的辅助规则 `function` 。函数*声明语句*是 `fun` 关键字后跟实际的函数体内容。等到我们实现类的时候，将会复用 `function` 规则来声明方法。这些方法与函数声明类似，但是前面没有 `fun` 。

> The function itself is a name followed by the parenthesized parameter list and the body. The body is always a braced block, using the same grammar rule that block statements use. The parameter list uses this rule:

函数本身是一个名称，后跟带括号的参数列表和函数体。函数体是一个带花括号的块，可以使用与块语句相同的语法。参数列表则使用以下规则：

```
parameters     → IDENTIFIER ( "," IDENTIFIER )* ;
```

> It's like the earlier `arguments` rule, except that each parameter is an identifier, not an expression. That's a lot of new syntax for the parser to chew through, but the resulting AST node isn't too bad.

这就类似于前面的 `arguments` 规则，区别在于参数是一个标识符，而不是一个表达式。这对于解析器来说是很多要处理的新语法，但是生成的AST节点没这么复杂。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Expression : Expr expression",
      // 新增部分开始
      "Function   : Token name, List<Token> params," +
                  " List<Stmt> body",
      // 新增部分结束
      "If         : Expr condition, Stmt thenBranch," +
```

> A function node has a name, a list of parameters (their names), and then the body. We store the body as the list of statements contained inside the curly braces.

函数节点有一个名称、一个参数列表(参数的名称)，然后是函数主体。我们将函数主体存储为包含在花括号中的语句列表。

> Over in the parser, we weave in the new declaration.

在解析器中，我们把新的声明添加进去。

lox/Parser.java，在 declaration()方法中添加：

```
    try {
      // 新增部分开始
      if (match(FUN)) return function("function");
      // 新增部分结束
      if (match(VAR)) return varDeclaration();
```

> Like other statements, a function is recognized by the leading keyword. When we encounter `fun`, we call `function`. That corresponds to the `function` grammar rule since we already matched and consumed the `fun` keyword. We'll build the method up a piece at a time, starting with this:

像其它语句一样，函数是通过前面的关键字来识别的。当我们遇到 `fun` 时，我们就调用 `function`。这步操作对应于 `function` 语法规则，因为我们已经匹配并消费了 `fun` 关键字。我们会一步步构建这个方法，首先从下面的代码开始：

lox/Parser.java，在 expressionStatement()方法后添加：

```
  private Stmt.Function function(String kind) {
    Token name = consume(IDENTIFIER, "Expect " + kind + " name.");
  }
```

> Right now, it only consumes the identifier token for the function's name. You might be wondering about that funny little `kind` parameter. Just like we re-use the grammar rule, we'll reuse the `function()` method later to parse methods inside classes. When we do that, we'll pass in "method" for `kind` so that the error messages are specific to the kind of declaration being parsed.

现在，它只是消费了标识符标记作为函数名称。你可能会对这里的 `kind` 参数感到疑惑。就像我们复用语法规则一样，稍后我们也会复用 `function()` 方法来解

析类中的方法。到时候，我们会在 `kind` 参数中传入 "method"，这样错误信息就会针对被解析的声明类型来展示。

> Next, we parse the parameter list and the pair of parentheses wrapped around it.

接下来，我们要解析参数列表和包裹着它们的一对小括号。

lox/Parser.java，在 function()方法中添加：

```
    Token name = consume(IDENTIFIER, "Expect " + kind + " name.");
    // 新增部分开始
    consume(LEFT_PAREN, "Expect '(' after " + kind + " name.");
    List<Token> parameters = new ArrayList<>();
    if (!check(RIGHT_PAREN)) {
      do {
        if (parameters.size() >= 255) {
          error(peek(), "Can't have more than 255 parameters.");
        }

        parameters.add(
            consume(IDENTIFIER, "Expect parameter name."));
      } while (match(COMMA));
    }
    consume(RIGHT_PAREN, "Expect ')' after parameters.");
    // 新增部分结束
  }
```

> This is like the code for handling arguments in a call, except not split out into a helper method. The outer `if` statement handles the zero parameter case, and the inner `while` loop parses parameters as long as we find commas to separate them. The result is the list of tokens for each parameter's name.

这就像在函数调用中处理参数的代码一样，只是没有拆分到一个辅助方法中。外部的 `if` 语句用于处理零参数的情况，内部的 `while` 会循环解析参数，只要能找到分隔参数的逗号。其结果是包含每个参数名称的标记列表。

> Just like we do with arguments to function calls, we validate at parse time that you don't exceed the maximum number of parameters a function is allowed to have.

就像我们处理函数调用的参数一样，我们在解析时验证是否超过了一个函数所允许的最大参数数。

> Finally, we parse the body and wrap it all up in a function node.

最后，我们解析函数主体，并将其封装为一个函数节点。

lox/Parser.java，在 function()方法中添加：

```
    consume(RIGHT_PAREN, "Expect ')' after parameters.");
    // 新增部分开始
    consume(LEFT_BRACE, "Expect '{' before " + kind + " body.");
    List<Stmt> body = block();
    return new Stmt.Function(name, parameters, body);
    // 新增部分结束
  }
```

> Note that we consume the `{` at the beginning of the body here before calling `block()`. That's because `block()` assumes the brace token has already been matched. Consuming it here lets us report a more precise error message if the `{` isn't found since we know it's in the context of a function declaration.

请注意，在调用`block()`方法之前，我们已经消费了函数体开头的`{`。这是因为`block()`方法假定大括号标记已经匹配了。在这里消费该标记可以让我们在找不到`{`的情况下报告一个更精确的错误信息，因为我们知道当前是在一个函数声明的上下文中。

#10.4 Function Objects

#10.4 函数对象

> We've got some syntax parsed so usually we're ready to interpret, but first we need to think about how to represent a Lox function in Java. We need to keep track of the parameters so that we can bind them to argument values when the function is called. And, of course, we need to keep the code for the body of the function so that we can execute it.

我们已经解析了一些语法，通常我们要开始准备解释了，但是我们首先需要思考一下，在Java中如何表示一个Lox函数。我们需要跟踪形参，以便在函数被调用时可以将形参与实参值进行绑定。当然，我们也要保留函数体的代码，以便我们可以执行它。

> That's basically what the Stmt.Function class is. Could we just use that? Almost, but not quite. We also need a class that implements LoxCallable so that we can call it. We don't want the runtime phase of the interpreter to bleed into the front end's syntax classes so we don't want Stmt.Function itself to implement that. Instead, we wrap it in a new class.

这基本上就是Stmt.Function的内容。我们可以用这个吗？差不多，但还不够。我们还需要一个实现LoxCallable的类，以便我们可以调用它。我们不希望解释器的运行时阶段渗入到前端语法类中，所以我们不希望使用Stmt.Function本身来实现它。相反，我们将它包装在一个新类中。

lox/LoxFunction.java，创建新文件：

```
package com.craftinginterpreters.lox;

import java.util.List;

class LoxFunction implements LoxCallable {
  private final Stmt.Function declaration;
  LoxFunction(Stmt.Function declaration) {
    this.declaration = declaration;
```

```
    }
  }
```

> We implement the `call()` of LoxCallable like so:

使用如下方式实现LoxCallable的 `call()` 方法：

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

```
  @Override
  public Object call(Interpreter interpreter,
                     List<Object> arguments) {
    Environment environment = new Environment(interpreter.globals);
    for (int i = 0; i < declaration.params.size(); i++) {
      environment.define(declaration.params.get(i).lexeme,
          arguments.get(i));
    }

    interpreter.executeBlock(declaration.body, environment);
    return null;
  }
```

> This handful of lines of code is one of the most fundamental, powerful pieces of our interpreter. As we saw in the chapter on statements and state, managing name environments is a core part of a language implementation. Functions are deeply tied to that.

这几行代码是我们的解释器中最基本、最强大的部分之一。正如我们在上一章中所看到的，管理名称环境是语言实现中的核心部分。函数与此紧密相关。

> Parameters are core to functions, especially the fact that a function *encapsulates* its parameters—no other code outside of the function can see them. This means each function gets its own environment where it stores those variables.

参数是函数的核心，尤其是考虑到函数*封装*了其参数——函数之外的代码看不到这些参数。这意味着每个函数都会维护自己的环境，其中存储着那些变量。

> Further, this environment must be created dynamically. Each function *call* gets its own environment. Otherwise, recursion would break. If there are multiple calls to the same function in play at the same time, each needs its *own* environment, even though they are all calls to the same function.

此外，这个环境必须是动态创建的。每次函数*调用*都会获得自己的环境，否则，递归就会中断。如果在同一时刻对相同的函数有多次调用，那么每个调用都需要自身的环境，即便它们都是对相同函数的调用。

> For example, here's a convoluted way to count to three:

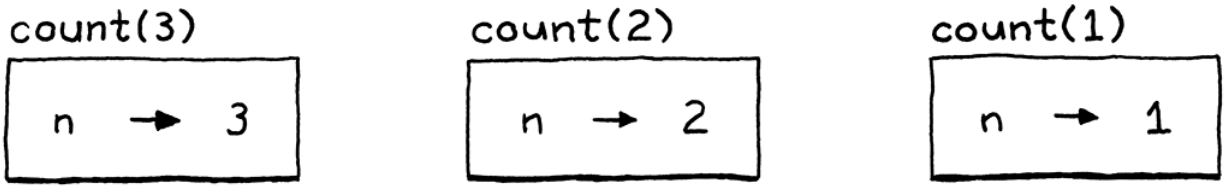
举例来说，下面是一个计数到3的复杂方法：

```
fun count(n) {
  if (n > 1) count(n - 1);
  print n;
}

count(3);
```

> Imagine we pause the interpreter right at the point where it's about to print 1 in the innermost nested call. The outer calls to print 2 and 3 haven't printed their values yet, so there must be environments somewhere in memory that still store the fact that `n` is bound to 3 in one context, 2 in another, and 1 in the innermost, like:

假设一下，如果我们在最内层的嵌套调用中即将打印1的时候暂停了解释器。打印2和3的外部调用还没有打印出它们的值，所以在内存的某个地方一定有环境仍然存储着这样的数据：n在一个上下文中被绑定到3，在另一个上下文中被绑定到2，而在最内层调用中绑定为1，比如：



> That's why we create a new environment at each *call*, not at the function *declaration*. The `call()` method we saw earlier does that. At the beginning of the call, it creates a new environment. Then it walks the parameter and argument lists in lockstep. For each pair, it creates a new variable with the parameter's name and binds it to the argument's value.

这就是为什么我们在每次*调用*时创建一个新的环境，而不是在函数声明时创建。我们前面看到的 `call()` 方法就是这样做的。在调用开始的时候，它创建了一个新环境。然后它以同步的方式遍历形参和实参列表。对于每一对参数，它用形参的名字创建一个新的变量，并将其与实参的值绑定。

> So, for a program like this:

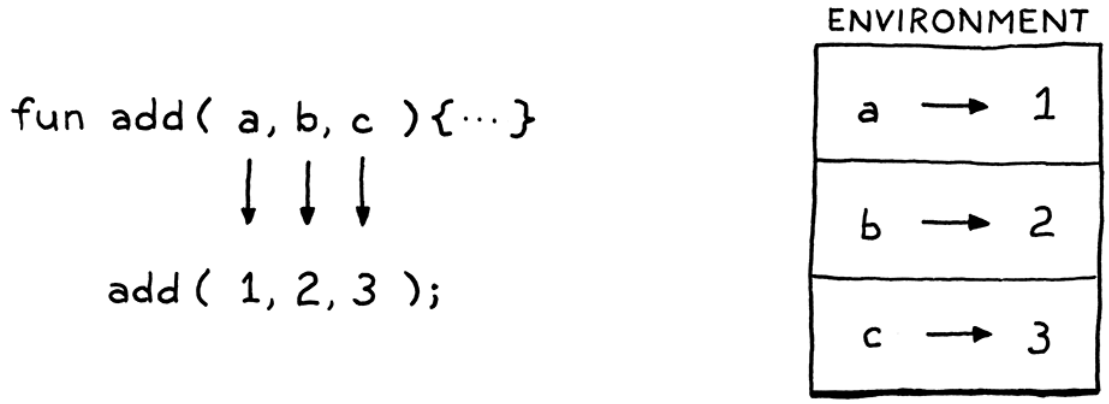
所以，对于类似下面这样的代码：

```
fun add(a, b, c) {
  print a + b + c;
}

add(1, 2, 3);
```

> At the point of the call to `add()`, the interpreter creates something like this:

在调用 `add()` 时，解释器会创建类似下面这样的内容：



> Then `call()` tells the interpreter to execute the body of the function in this new function-local environment. Up until now, the current environment was the environment where the function was being called. Now, we teleport from there inside the new parameter space we've created for the function.

然后 `call()` 会告诉解释器在这个新的函数局部环境中执行函数体。在此之前，当前环境是函数被调用的位置所处的环境。现在，我们转入了为函数创建的新的参数空间中。

> This is all that's required to pass data into the function. By using different environments when we execute the body, calls to the same function with the same code can produce different results.

这就是将数据传入函数所需的全部内容。通过在执行函数主体时使用不同的环境，用同样的代码调用相同的函数可以产生不同的结果。

> Once the body of the function has finished executing, `executeBlock()` discards that function-local environment and restores the previous one that was active back at the callsite. Finally, `call()` returns `null`, which returns `nil` to the caller. (We'll add return values later.)

一旦函数的主体执行完毕， `executeBlock()` 就会丢弃该函数的本地环境，并恢复调用该函数前的活跃环境。最后， `call()` 方法会返回 `null` ， 它向调用者返回 `nil` 。 （我们会在稍后添加返回值）

> Mechanically, the code is pretty simple. Walk a couple of lists. Bind some new variables. Call a method. But this is where the crystalline *code* of the function declaration becomes a living, breathing *invocation*. This is one of my favorite snippets in this entire book. Feel free to take a moment to meditate on it if you're so inclined.

从机制上讲，这段代码是非常简单的。遍历几个列表，绑定一些新变量，调用一个方法。但这就是将代码块变成有生命力的调用执行的地方。这是我在整本

书中最喜欢的片段之一。如果你愿意的话，可以花点时间好好思考一下。

> Done? OK. Note when we bind the parameters, we assume the parameter and argument lists have the same length. This is safe because `visitCallExpr()` checks the arity before calling `call()`. It relies on the function reporting its arity to do that.

完成了吗？好的。注意当我们绑定参数时，我们假设参数和参数列表具有相同的长度。这是安全的，因为 `visitCallExpr()` 在调用 `call()` 之前会检查元数。它依靠报告其元数的函数来做到这一点。

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

```
@Override
public int arity() {
  return declaration.params.size();
}
```

> That's most of our object representation. While we're in here, we may as well implement `toString()`.

这基本就是我们的函数对象表示了。既然已经到了这一步，我们也可以实现 toString()。

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

```
@Override
public String toString() {
  return "<fn " + declaration.name.lexeme + ">";
}
```

> This gives nicer output if a user decides to print a function value.

如果用户要打印函数的值，该方法能提供一个更漂亮的输出值。

```
fun add(a, b) {
  print a + b;
```

```
}

print add; // "<fn add>".
```

> ### #10.4.1 Interpreting function declarations

#10.4.1 解释函数声明

> We'll come back and refine LoxFunction soon, but that's enough to get started. Now we can visit a function declaration.

我们很快就会回头来完善LoxFunction，但是现在已足够开始进行解释了。现在，我们可以访问函数声明节点了。

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

```
@Override
public Void visitFunctionStmt(Stmt.Function stmt) {
  LoxFunction function = new LoxFunction(stmt);
  environment.define(stmt.name.lexeme, function);
  return null;
}
```

> This is similar to how we interpret other literal expressions. We take a function *syntax node*—a compile-time representation of the function—and convert it to its runtime representation. Here, that's a LoxFunction that wraps the syntax node.

这类似于我们介绍其它文本表达式的方式。我们会接收一个函数*语法*节点——函数的编译时表示形式——然后将其转换为运行时表示形式。在这里就是一个封装了语法节点的LoxFunction实例。

> Function declarations are different from other literal nodes in that the declaration *also* binds the resulting object to a new variable. So, after creating

> the LoxFunction, we create a new binding in the current environment and store a reference to it there.

函数声明与其它文本节点的不同之处在于，声明还会将结果对象绑定到一个新的变量。因此，在创建LoxFunction之后，我们在当前环境中创建一个新的绑定，并在其中保存对该函数的引用。

> With that, we can define and call our own functions all within Lox. Give it a try:

这样，我们就可以在Lox中定义和调用我们自己的函数。试一下：

```
fun sayHi(first, last) {
  print "Hi, " + first + " " + last + "!";
}

sayHi("Dear", "Reader");
```

> I don’t know about you, but that looks like an honest-to-God programming language to me.

我不知道你怎么想的，但对我来说，这看起来像是一种虔诚的编程语言。

> ## #10.5 Return Statements

#10.5 Return语句

> We can get data into functions by passing parameters, but we’ve got no way to get results back *out*. If Lox were an expression-oriented language like Ruby or Scheme, the body would be an expression whose value is implicitly the function’s result. But in Lox, the body of a function is a list of statements which don’t produce values, so we need dedicated syntax for

> emitting a result. In other words, `return` statements. I'm sure you can guess the grammar already.

我们可以通过传递参数将数据输入函数中，但是我们没有办法将结果*传出来*。如果Lox是像Ruby或Scheme那样的面向表达式的语言，那么函数体就是一个表达式，其值就隐式地作为函数的结果。但是在Lox中，函数体是一个不产生值的语句列表，所有我们需要专门的语句来发出结果。换句话说，就是 `return` 语句。我相信你已经能猜出语法了。

```
statement      → exprStmt
               | forStmt
               | ifStmt
               | printStmt
               | returnStmt
               | whileStmt
               | block ;

returnStmt     → "return" expression? ";" ;
```

> We've got one more—the final, in fact—production under the venerable `statement` rule. A `return` statement is the `return` keyword followed by an optional expression and terminated with a semicolon.

我们又得到一个 `statement` 规则下的新产生式（实际上也是最后一个）。一个 `return` 语句就是一个 `return` 关键字，后跟一个可选的表达式，并以一个分号结尾。

> The return value is optional to support exiting early from a function that doesn't return a useful value. In statically typed languages, "void" functions don't return a value and non-void ones do. Since Lox is dynamically typed, there are no true void functions. The compiler has no way of preventing you from taking the result value of a call to a function that doesn't contain a `return` statement.

返回值是可选的，用以支持从一个不返回有效值的函数中提前退出。在静态类型语言中，void函数不返回值，而非void函数返回值。由于Lox是动态类型的，

所以没有真正的void函数。在调用一个不包含 `return` 语句的函数时，编译器没有办法阻止你获取其结果值。

```
fun procedure() {
  print "don't return anything";
}

var result = procedure();
print result; // ?
```

> This means every Lox function must return *something*, even if it contains no `return` statements at all. We use `nil` for this, which is why LoxFunction's implementation of `call()` returns `null` at the end. In that same vein, if you omit the value in a `return` statement, we simply treat it as equivalent to:

这意味着每个Lox函数都要返回一些内容，即使其中根本不包含 `return` 语句。我们使用 `nil`，这就是为什么LoxFunction的 `call()` 实现在最后返回 `null`。同样，如果你省略了 `return` 语句中的值，我们将其视为等价于：

```
return nil;
```

> Over in our AST generator, we add a new node.

在AST生成器中，添加一个新节点。

tool/GenerateAst.java，在 main()方法中添加：

```
      "Print      : Expr expression",
      // 新增部分开始
      "Return     : Token keyword, Expr value",
      // 新增部分结束
      "Var        : Token name, Expr initializer",
```

> It keeps the `return` keyword token so we can use its location for error reporting, and the value being returned, if any. We parse it like other statements, first by recognizing the initial keyword.

其中保留了 `return` 关键字标记（这样我们可以使用该标记的位置来报告错误），以及返回的值（如果有的话）。我们像解析其它语句一样来解析它，首先识别起始的关键字。

lox/Parser.java，在 statement()方法中添加：

```
    if (match(PRINT)) return printStatement();
    // 新增部分开始
    if (match(RETURN)) return returnStatement();
    // 新增部分结束
    if (match(WHILE)) return whileStatement();
```

> That branches out to:

分支会跳转到：

lox/Parser.java，在 printStatement()方法后添加：

```
  private Stmt returnStatement() {
    Token keyword = previous();
    Expr value = null;
    if (!check(SEMICOLON)) {
      value = expression();
    }

    consume(SEMICOLON, "Expect ';' after return value.");
    return new Stmt.Return(keyword, value);
  }
```

> After snagging the previously consumed `return` keyword, we look for a value expression. Since many different tokens can potentially start an expression, it's hard to tell if a return value is *present*. Instead, we check if it's *absent*. Since a semicolon can't begin an expression, if the next token is that, we know there must not be a value.

在捕获先前消耗的 `return` 关键字之后，我们会寻找一个值表达式。因为很多不同的标记都可以引出一个表达式，所以很难判断是否存在返回值。相反，我们

检查它是否不存在。因为分号不能作为表达式的开始，如果下一个标记是分号，我们就知道一定没有返回值。

> #10.5.1 Returning from calls

#10.5.1 从函数调用中返回

> Interpreting a `return` statement is tricky. You can return from anywhere within the body of a function, even deeply nested inside other statements. When the return is executed, the interpreter needs to jump all the way out of whatever context it's currently in and cause the function call to complete, like some kind of jacked up control flow construct.

解释 `return` 语句是很棘手的。你可以从函数体中的任何位置返回，甚至是深深嵌套在其它语句中的位置。当返回语句被执行时，解释器需要完全跳出当前所在的上下文，完成函数调用，就像某种顶层的控制流结构。

> For example, say we're running this program and we're about to execute the `return` statement:

举例来说，假设我们正在运行下面的代码，并且我们即将执行 `return` 语句：

```
fun count(n) {
  while (n < 100) {
    if (n == 3) return n; // <--
    print n;
    n = n + 1;
  }
}

count(1);
```

> The Java call stack currently looks roughly like this:

Java调用栈目前看起来大致如下所示：

```
Interpreter.visitReturnStmt()
Interpreter.visitIfStmt()
Interpreter.executeBlock()
Interpreter.visitBlockStmt()
Interpreter.visitWhileStmt()
Interpreter.executeBlock()
LoxFunction.call()
Interpreter.visitCallExpr()
```

> We need to get from the top of the stack all the way back to `call()`. I don't know about you, but to me that sounds like exceptions. When we execute a `return` statement, we'll use an exception to unwind the interpreter past the visit methods of all of the containing statements back to the code that began executing the body.

我们需要从栈顶一直回退到`call()`。我不知道你怎么想，但是对我来说，这听起来很像是异常。当我们执行`return`语句时，我们会使用一个异常来解开解释器，经过所有函数内含语句的visit方法，一直回退到开始执行函数体的代码。

> The visit method for our new AST node looks like this:

新的AST节点的visit方法如下所示：

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

```
@Override
public Void visitReturnStmt(Stmt.Return stmt) {
  Object value = null;
  if (stmt.value != null) value = evaluate(stmt.value);

  throw new Return(value);
}
```

> If we have a return value, we evaluate it, otherwise, we use `nil`. Then we take that value and wrap it in a custom exception class and throw it.

如果我们有返回值，就对其求值，否则就使用 `nil` 。然后我们取这个值并将其封装在一个自定义的异常类中，并抛出该异常。

lox/Return.java，创建新文件：

```
package com.craftinginterpreters.lox;

class Return extends RuntimeException {
  final Object value;

  Return(Object value) {
    super(null, null, false, false);
    this.value = value;
  }
}
```

> This class wraps the return value with the accoutrements Java requires for a runtime exception class. The weird super constructor call with those `null` and `false` arguments disables some JVM machinery that we don't need. Since we're using our exception class for control flow and not actual error handling, we don't need overhead like stack traces.

这个类使用Java运行时异常类来封装返回值。其中那个奇怪的带有 `null` 和 `false` 的父类构造器方法，禁用了一些我们不需要的JVM机制。因为我们只是使用该异常类来控制流，而不是真正的错误处理，所以我们不需要像堆栈跟踪这样的开销。

> We want this to unwind all the way to where the function call began, the `call()` method in LoxFunction.

我们希望可以一直跳出到函数调用开始的地方，也就是LoxFunction中的 `call()` 方法。

lox/LoxFunction.java，在 call()方法中替换一行：

```
          arguments.get(i));
    }
```

```
    // 替换部分开始
    try {
      interpreter.executeBlock(declaration.body, environment);
    } catch (Return returnValue) {
      return returnValue.value;
    }
    // 替换部分结束
    return null;
```

> We wrap the call to `executeBlock()` in a try-catch block. When it catches a return exception, it pulls out the value and makes that the return value from `call()`. If it never catches one of these exceptions, it means the function reached the end of its body without hitting a `return` statement. In that case, it implicitly returns `nil`.

我们将对 `executeBlock()` 的调用封装在一个try-catch块中。当捕获一个返回异常时，它会取出其中的值并将其作为 `call()` 方法的返回值。如果没有捕获任何异常，意味着函数到达了函数体的末尾，而且没有遇到 `return` 语句。在这种情况下，隐式地返回 `nil`。

> Let's try it out. We finally have enough power to support this classic example—a recursive function to calculate Fibonacci numbers:

我们来试一下。我们终于有能力支持这个经典的例子——递归函数计算Fibonacci数[7]：

```
fun fib(n) {
  if (n <= 1) return n;
  return fib(n - 2) + fib(n - 1);
}

for (var i = 0; i < 20; i = i + 1) {
  print fib(i);
}
```

> This tiny program exercises almost every language feature we have spent the past several chapters implementing—expressions, arithmetic, branch-

> ing, looping, variables, functions, function calls, parameter binding, and returns.

这个小程序练习了我们在过去几章中实现的几乎所有语言特性，包括表达式、算术运算、分支、循环、变量、函数、函数调用、参数绑定和返回。

> ## #10.6 Local Functions and Closures

#10.6 局部函数和闭包

> Our functions are pretty full featured, but there is one hole to patch. In fact, it's a big enough gap that we'll spend most of the next chapter sealing it up, but we can get started here.

我们的函数功能已经相当全面了，但是还有一个漏洞需要修补。实际上，这是一个很大的问题，我们将会在下一章中花费大部分时间来修补它，但是我们可以从这里开始。

> LoxFunction's implementation of `call()` creates a new environment where it binds the function's parameters. When I showed you that code, I glossed over one important point: What is the *parent* of that environment?

LoxFunction中的 `call()` 实现创建了一个新的环境，并在其中绑定了函数的参数。当我向你展示这段代码时，我忽略了一个重要的问题：这个环境的父类是什么?

> Right now, it is always `globals`, the top-level global environment. That way, if an identifier isn't defined inside the function body itself, the interpreter can look outside the function in the global scope to find it. In the Fibonacci example, that's how the interpreter is able to look up the recursive call to `fib` inside the function's own body— `fib` is a global variable.

目前，它始终是 `globals`，即顶级的全局环境。这样，如果一个标识符不是在函数体内部定义的，解释器可以在函数外部的全局作用域中查找它。在Fibonacci的例子中，这就是解释器如何能够在函数体中实现对 `fib` 的递归调用——`fib` 是一个全局变量。

> But recall that in Lox, function declarations are allowed *anywhere* a name can be bound. That includes the top level of a Lox script, but also the inside of blocks or other functions. Lox supports **local functions** that are defined inside another function, or nested inside a block.

但请记住，在Lox中，允许在可以绑定名字的*任何地方*进行函数声明。其中包括Lox脚本的顶层，但也包括块或其他函数的内部。Lox支持在另一个函数内定义或在一个块内嵌套的**局部函数**。

> Consider this classic example:

考虑下面这个经典的例子：

```
fun makeCounter() {
  var i = 0;
  fun count() {
    i = i + 1;
    print i;
  }

  return count;
}

var counter = makeCounter();
counter(); // "1".
counter(); // "2".
```

> Here, `count()` uses `i`, which is declared outside of itself in the containing function `makeCounter()`. `makeCounter()` returns a reference to the `count()` function and then its own body finishes executing completely.

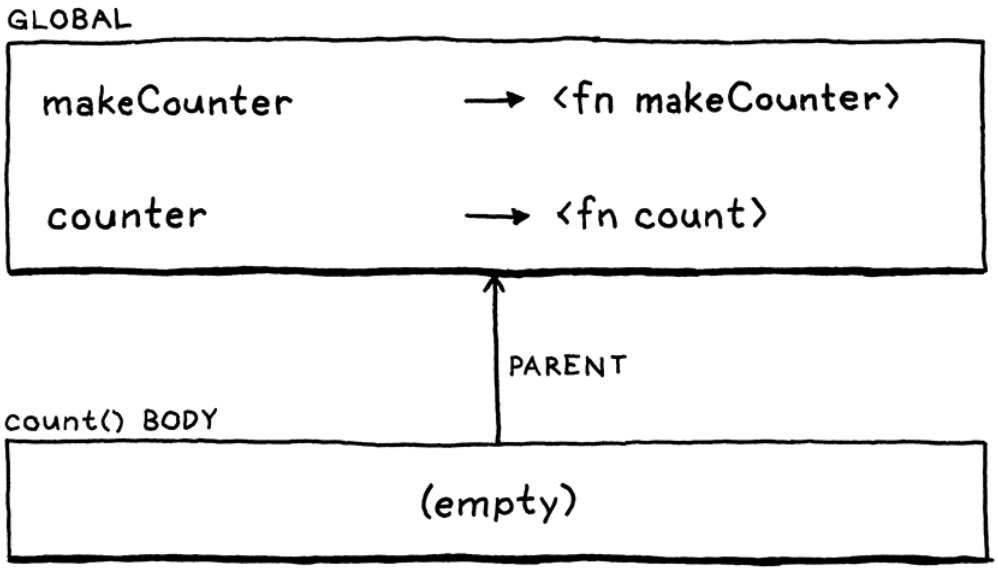
这个例子中，`count()` 使用了 `i`，它是在该函数外部的 `makeCounter()` 声明的。`makeCounter()` 返回对 `count()` 函数的引用，然后它的函数体就执行完成了。

> Meanwhile, the top-level code invokes the returned `count()` function. That executes the body of `count()`, which assigns to and reads `i`, even though the function where `i` was defined has already exited.

同时，顶层代码调用了返回的 `count()` 函数。这就执行了 `count()` 函数的主体，它会对 `i` 赋值并读取 `i`，尽管定义 `i` 的函数已经退出。

> If you've never encountered a language with nested functions before, this might seem crazy, but users do expect it to work. Alas, if you run it now, you get an undefined variable error in the call to `counter()` when the body of `count()` tries to look up `i`. That's because the environment chain in effect looks like this:

如果你以前从未遇到过带有嵌套函数的语言，那么这可能看起来很疯狂，但用户确实希望它能工作。唉，如果你现在运行它，当 `count()` 的函数体试图查找 `i` 时，会在对 `counter()` 的调用中得到一个未定义的变量错误，这是因为当前的环境链看起来像是这样的：



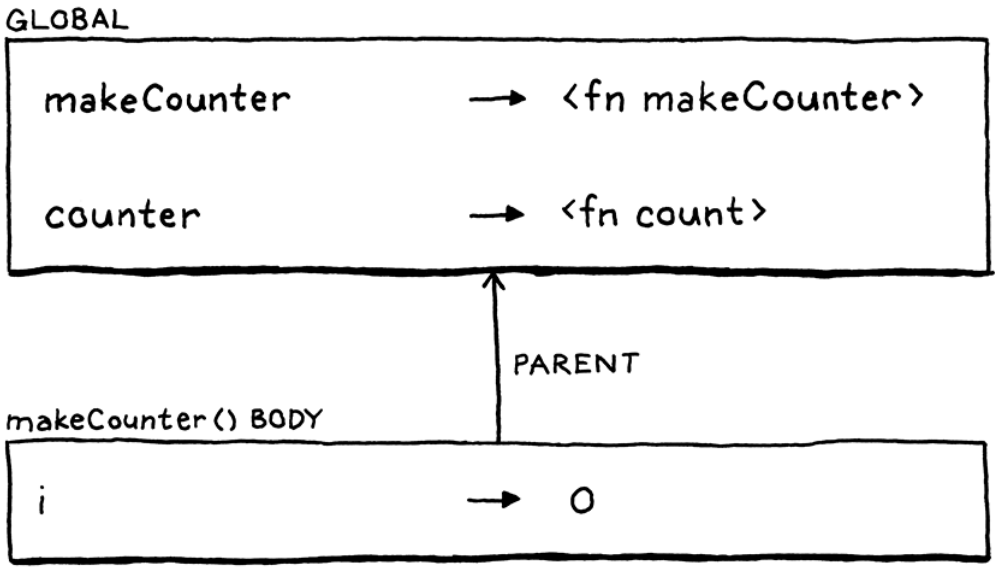
> When we call `count()` (through the reference to it stored in `counter`), we create a new empty environment for the function body. The parent of that is

> the global environment. We lost the environment for `makeCounter()` where `i` is bound.

当我们调用 `count()` 时（通过 `counter` 中保存的引用），我们会为函数体创建一个新的空环境，它的父环境就是全局环境。我们丢失了 `i` 所在的 `makeCounter()` 环境。

> Let's go back in time a bit. Here's what the environment chain looked like right when we declared `count()` inside the body of `makeCounter()`:

我们把时间往回拨一点。我们在 `makeCounter()` 的函数体中声明 `count()` 时，环境链的样子是下面这样：



> So at the point where the function is declared, we can see `i`. But when we return from `makeCounter()` and exit its body, the interpreter discards that environment. Since the interpreter doesn't keep the environment surrounding `count()` around, it's up to the function object itself to hang on to it.

所以，在函数声明的地方，我们可以看到 `i`。但是当我们从 `makeCounter()` 返回并退出其主体时，解释器会丢弃这个环境。因为解释器不会保留 `count()` 外围的环境，所以要靠函数对象本身来保存它。

> This data structure is called a **closure** because it “closes over” and holds on to the surrounding variables where the function is declared. Closures have been around since the early Lisp days, and language hackers have come up with all manner of ways to implement them. For jlox, we’ll do the simplest thing that works. In LoxFunction, we add a field to store an environment.

这种数据结构被称为**闭包**，因为它 "封闭 "并保留着函数声明的外围变量。闭包早在Lisp时代就已经存在了，语言黑客们想出了各种方法来实现闭包。在jlox中，我们将采用最简单的方式。在LoxFunction中，我们添加一个字段来存储环境。

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

```
private final Stmt.Function declaration;
// 新增部分开始
private final Environment closure;
// 新增部分结束
LoxFunction(Stmt.Function declaration) {
```

> We initialize that in the constructor.

我们在构造函数中对其初始化。

lox/LoxFunction.java，在 LoxFunction()构造方法中替换一行：

```
//替换部分开始
LoxFunction(Stmt.Function declaration, Environment closure) {
  this.closure = closure;
  // 替换部分结束
  this.declaration = declaration;
```

> When we create a LoxFunction, we capture the current environment.

当我们创建LoxFunction时，我们会捕获当前环境。

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

```
  public Void visitFunctionStmt(Stmt.Function stmt) {
    // 替换部分开始
    LoxFunction function = new LoxFunction(stmt, environment);
    // 替换部分结束
    environment.define(stmt.name.lexeme, function);
```

> This is the environment that is active when the function is *declared* not when it's *called*, which is what we want. It represents the lexical scope surrounding the function declaration. Finally, when we call the function, we use that environment as the call's parent instead of going straight to `globals`.

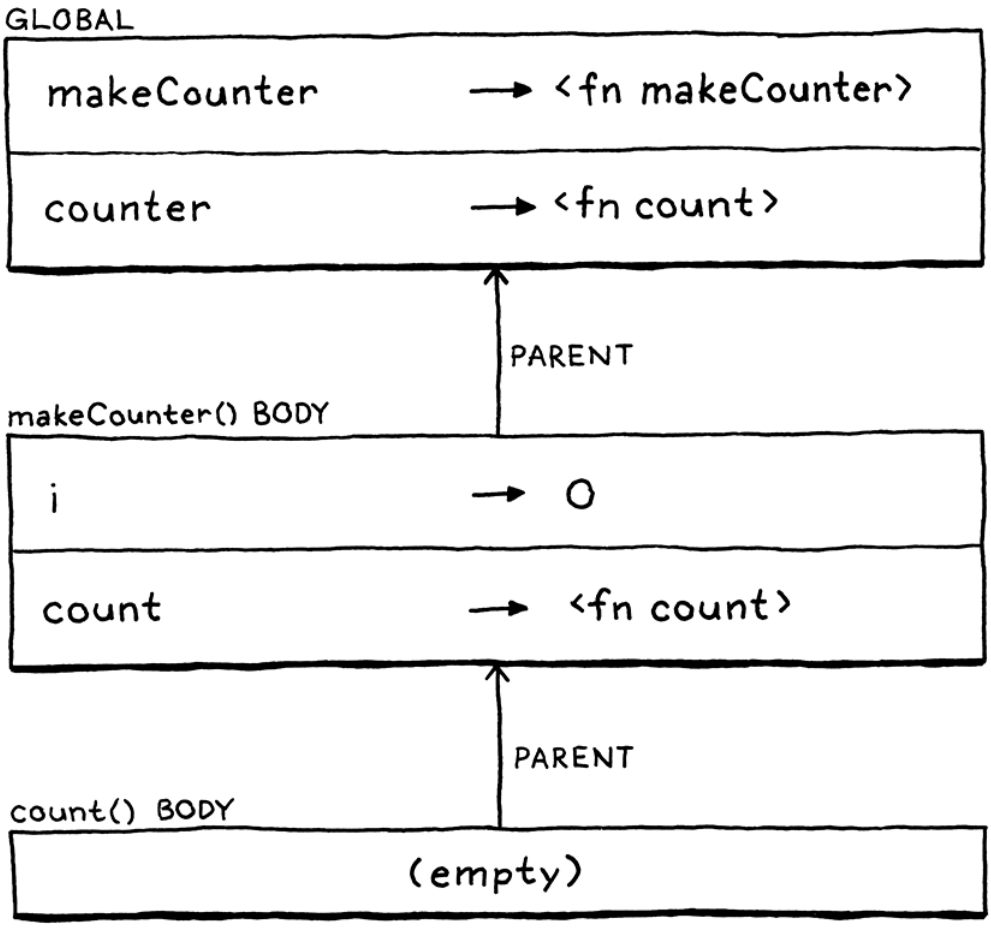
这是函数声明时生效的环境，而不是函数被调用时的环境，这正是我们想要的。它代表了函数声明时的词法作用域。最后，当我们调用函数时，我们使用该环境作为调用的父环境，而不是直接使用 `globals`。

lox/LoxFunction.java，在 call()方法中替换一行：

```
                       List<Object> arguments) {
    // 替换部分开始
    Environment environment = new Environment(closure);
    // 替换部分结束
    for (int i = 0; i < declaration.params.size(); i++) {
```

> This creates an environment chain that goes from the function's body out through the environments where the function is declared, all the way out to the global scope. The runtime environment chain matches the textual nesting of the source code like we want. The end result when we call that function looks like this:

这样就创建了一个环境链，从函数体开始，经过函数被声明的环境，然后到全局作用域。运行时环境链与源代码的文本嵌套相匹配，跟我们想要的一致。当我们调用该函数时，最终的结果是这样的：



> Now, as you can see, the interpreter can still find `i` when it needs to because it's in the middle of the environment chain. Try running that `makeCounter()` example now. It works!

如你所见，现在解释器可以在需要的时候找到`i`，因为它在环境链中。现在尝试运行makeCounter()的例子，起作用了！

> Functions let us abstract over, reuse, and compose code. Lox is much more powerful than the rudimentary arithmetic calculator it used to be. Alas, in our rush to cram closures in, we have let a tiny bit of dynamic scoping leak into the interpreter. In the next chapter, we will explore deeper into lexical scope and close that hole.

函数让我们对代码进行抽象、重用和编排。Lox比之前的初级算术计算器要强大得多。唉，在我们匆匆忙忙支持闭包时，已经让一小部分动态作用域泄露到解释器中了。在下一章中，我们将深入探索词法作用域，堵住这个漏洞。

#CHALLENGES

#习题

1、Our interpreter carefully checks that the number of arguments passed to a function matches the number of parameters it expects. Since this check is done at runtime on every call, it has a performance cost. Smalltalk implementations don't have that problem. Why not?

1、解释器会仔细检查传给函数的实参数量是否与期望的形参数量匹配。由于该检查是在运行时，针对每一次调用执行的，所以会有性能成本。Smalltalk的实现则没有这个问题。为什么呢?

2、Lox's function declaration syntax performs two independent operations. It creates a function and also binds it to a name. This improves usability for the common case where you do want to associate a name with the function. But in functional-styled code, you often want to create a function to immediately pass it to some other function or return it. In that case, it doesn't need a name.

Languages that encourage a functional style usually support **anonymous functions** or **lambdas**—an expression syntax that creates a function without binding it to a name. Add anonymous function syntax to Lox so that this works:

```
fun thrice(fn) {
  for (var i = 1; i <= 3; i = i + 1) {
    fn(i);
  }
}
```

```
thrice(fun (a) {
  print a;
});
// "1".
// "2".
// "3".
```

> How do you handle the tricky case of an anonymous function expression occurring in an expression statement:
>
> ```
> fun () {};
> ```

2、Lox的函数声明语法执行了两个独立的操作。它创建了一个函数，并将其与一个名称绑定。这提高了常见情况下的可用性，即你确实想把一个名字和函数联系起来。但在函数式的代码中，你经常想创建一个函数，以便立即将它传递给其他函数或返回它。在这种情况下，它不需要一个名字。

鼓励函数式风格的语言通常支持**匿名函数**或**lambdas**——一个创建函数而不用将其与名称绑定的表达式语法。在Lox中加入匿名函数的语法，已支持下面的代码：

```
fun thrice(fn) {
  for (var i = 1; i <= 3; i = i + 1) {
    fn(i);
  }
}

thrice(fun (a) {
  print a;
});
// "1".
// "2".
// "3".
```

如何处理在表达式语句中出现匿名函数表达式的棘手情况：

```
fun () {};
```

> 3、Is this program valid?

> ```
> fun scope(a) {
>   var a = "local";
> }
> ```
>
> In other words, are a function's parameters in the *same* scope as its local variables, or in an outer scope? What does Lox do? What about other languages you are familiar with? What do you think a language *should* do?

3、下面的代码可用吗?

```
fun scope(a) {
  var a = "local";
}
```

换句话说，一个函数的参数是跟它的局部变量在同一个作用域内，还是在一个外部作用域内？Lox 是怎么做的？你所熟悉的其他语言呢？你认为一种语言应该怎么做?

---

[^1]

该规则中使用 `*` 符号匹配类似 `fn(1)(2)(3)` 的系列函数调用。这样的代码不是常见的C语言风格，但是在ML衍生的语言族中很常见。在ML中，定义接受多个参数的函数的常规方式是将其定义为一系列嵌套函数。每个函数接受一个参数并返回一个新函数。该函数使用下一个参数，返回另一个函数，以此类推。最终，一旦所有参数都被使用，最后一个函数就完成了操作。这种风格被称为柯里化，是以Haskell Curry（他的名字出现在另一个广为人知的函数式语言中）的名字命名的，它被直接整合到语言的语法中，所以它不像这里看起来那么奇怪。

[^2]

这段代码可以简化为 `while (match(LEFT_PAREN))` 形式，而不是使用这种愚蠢的 `while (true)` 和 `break` 形式。但是不用担心，稍后使用解析器处理对象属性的时候，这种写法就有意义了。

[^3]

如果该方法是一个实例方法，则限制为254个参数。因为 `this` （方法的接收者）就像一个被隐式传递给方法的参数一样，所以也会占用一个参数位置。

[^4]

这是另一个微妙的语义选择。由于参数表达式可能有副作用，因此它们的执行顺序可能是用户可见的。即便如此，有些语言如Scheme和C并没有指定顺序。这样编译器可以自由地重新排序以提高效率，但这意味着如果参数没有按照用户期望的顺序计算，用户可能会感到不愉快。

[^5]

奇怪的是，这些函数的两个名称native和foreign是反义词。也许这取决于选择这个词的人的角度。如果您认为自己生活在运行时实现中(在我们的例子中是Java)，那么用它编写的函数就是本机的。但是，如果您站在语言用户的角度，那么运行时就是用其他“外来”语言实现的。或者本机指的是底层硬件的机器代码语言。在Java中，本机方法是用C或c++实现并编译为本机机器码的方法。

[^6]

几乎每种语言都提供的一个经典的本地函数是将文本打印到标准输出。在Lox中，我将 `print` 作为了内置语句，以便可以在前面的章节中看到代码结果。一旦我们有了函数，我们就可以删除之前的 `print` 语法并用一个本机函数替换它，从而简化语言。但这意味着书中前面的例子不能在后面章节的解释器上运行，反之亦然。所以，在这本书中，我不去修改它。但是，如果您正在为自己的语言构建一个解释器，您可能需要考虑一下。

[^7]

你可能会注意到这是很慢的。显然，递归并不是计算斐波那契数的最有效方法，但作为一个微基准测试，它很好地测试了我们的解释器实现函数调用的速度。