Openresty中文编程网

version 0.1

Last generated: November 08, 2016

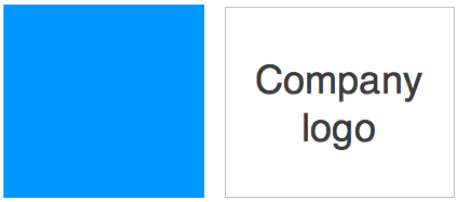

Table of Contents

自动化测试

Summary: 自动化测试在软件开发和维护中起着重要的作用。

== 自动化测试

自动化测试在软件开发和维护中起着重要的作用。 OpenResty 提供了一个数据驱动的测试框架，可以为 NGINX C 模块、Lua 库、甚至 OpenResty 应用，编写声明式测试用例。 测试用例是以一种规范的格式写成的，既有直观的可读性和可维护性，也易于机器的处理。 数据驱动的方法可以让同样的测试用例集，用各种完全不同的方式运行。这样有利于在不同的场景下，或者配合外部的各种工具，暴露出问题。

这个章节介绍的 `Test::Nginx` 测试框架，在几乎所有的 OpenResty 组件中都广泛的用于组织测试套件， 包括 `ngx_http_lua` 模块，大部分的 `lua-resty-*` Lua 库， 以及像 CloudFlare 的 Lua CDN 和 Lua SSL 这样成熟的商业应用。

关键字: 测试（Testing），模拟（Mocking）

介绍

Summary: OpenResty 自身依托于自动化测试来维持高质量，已经有几年了。作为 OpenResty 的核心开发者，我们一直是拥抱测试驱动

== 介绍

OpenResty 自身依托于自动化测试来维持高质量，已经有几年了。作为 OpenResty 的核心开发者，我们一直是拥抱测试驱动（test driven development - TDD）的。我们这几年 TDD 的实践，很棒的一点是为 OpenResty 的所有组件积累了大量的测试用例集。 整体的测试集非常的大，以至于无法在单机上面运行它们。所以我们通常是在 Amazon EC2 集群（已经部署好测试环境）上运行完成所有测试。 所有这些测试集所依赖的核心，通常都由 Test::Nginx 这个 OpenResty 团队开发的测试模块所支撑。

Test::Nginx 提供了一个通用、简单、规范的语言，用来直观的表达和组织测试用例。 它也提供各种强有力的测试模型或“引擎”，在不同配置环境中，运行各种测试用例，希望发现不符合预期的错误。 它还支持扩展这个测试规范语言，通过添加自定义抽象，来完成高级的测试需要，通常用在应用层回归测试。

=== Conceptual Roadmap

=== Overview

Test::Nginx

Summary: 这种简单测试规范格式或称小语言，使得 `Test::Nginx` 的更像是 Perl 世界里 link:https://metacpan.org/pod/distribution/Test-Base/lib/Test/Base.pod[Test::Base] 测试模块提供的各种通用测试语言中的一种方言。事实上，用面向对象的角度看，`Test::Nginx` 是 `Test::Base` 子类。这就意味着 `Test::Base` 提供的所有特性在 `Test::Nginx` 都是可用的，`Test::Nginx` 提供更易用的原语和符号，用来简化 NGINX 和 OpenResty 环境的测试。 `Test::Base` 的核心理念，是基于 `Test::Base` 的测试系统可以被广泛、有效的使用，甚至包括 Haskell 编程和 Linux 内核模块。`Test::Nginx` 只是我们为了 NGINX 和 OpenResty 测试创造的例子。有关 `Test::Base` 框架自身的详细讨论，已经超过了这本书的范畴，但在后续章节中，我们还将介绍 `Test::Nginx` 继承下来有关 `Test::Base` 的重要特性。

== Test::Nginx

link:https://metacpan.org/pod/Test::Nginx[Test::Nginx] 是一个测试框架，可以驱动运行 NGINX 上的任何测试用例代码，自然也可以是 NGINX 的内核代码。使用 Perl 语言编写，因为近数年时间的积累，有丰富的测试设备，以及完整的周边工具链。更进一步，使用者都不需要知道这些测试用例框架是用 Perl 书写的， Test::Nginx 提供了一些简单符号来表达当前测试用例，并把它们用一种规范格式组织起来。

这种简单测试规范格式或称小语言，使得 Test::Nginx 的更像是 Perl 世界里 link:https://metacpan.org/pod/distribution/Test-Base/lib/Test/Base.pod[Test::Base] 测试模块提供的各种通用测试语言中的一种方言。事实上，用面向对象的角度看， Test::Nginx 是 Test::Base 子类。这就意味着 Test::Base 提供的所有特性在 Test::Nginx 都是可用的， Test::Nginx 提供更易用的原语和符号，用来简化 NGINX 和 OpenResty 环境的测试。 Test::Base 的核心理念，是基于 Test::Base 的测试系统可以被广泛、有效的使用，甚至包括 Haskell 编程和 Linux 内核模块。 Test::Nginx 只是我们为了 NGINX 和 OpenResty 测试创造的例子。有关 Test::Base 框架自身的详细讨论，已经超过了这本书的范畴，但在后续章节中，我们还将介绍 Test::Nginx 继承下来有关 Test::Base 的重要特性。

Test::Nginx 通过 link:http://www.cpan.org/[CPAN] （Comprehensive Perl Archive Network）发行，与其他大多数 Perl 的库一致。如果在你的系统中已经安装了 perl （大多数 Linux 版本已经默认包含了 perl ），这时你可以使用下面简单命令完成安装：

[source,bash]

cpan Test::Nginx —-

对于第一次运行 cpan 工具，可能它将提示你配置 cpan 工具，以此来适配你的环境。如果你不确定这些选项，选择默认配置选项（如果有）或者接受所有默认选项。

Test::Nginx 针对不同用户环境，提供了几个不同的测试类。最经常被使用的一个是 Test::Nginx::Socket 。本章的剩余部分焦点将集中在测试类以及它们的子类。从现在开始没有特殊说明，我们将把 Test::Nginx 和 Test::Nginx::Socket 混合，使用 Test::Nginx::Socket 代表测试模块及其子类。

// Alas. GitBook does not support sidebar blocks in its AsciiDoc render. // .Another Test::Nginx

注意: 这里实际上还有另外一个不同的测试框架也叫 Test::Nginx ，是由 Maxim Dounin 创建并由 NGINX 官方团队保留。这个测试模块是通过 link:http://hg.nginx.org/nginx-tests/file/tip[official NGINX test suite] 对外发行，除了这两者都是测试 NGINX 相关代码，它与我们的 Test::Nginx 没有任何联系。NGINX 团队的 Test::Nginx 需要用户在 Perl 中直接写代码来表达测试用例，也就意味着他们在 Test::Nginx 所写的测试用例不是数据驱动并需要适当的 Perl 编程知识。

测试套件设计

Summary: 使用 `Test::Nginx` 驱动我们的测试套件，通常使用一个公共的目录结构以及规范的测试文件名样式，用来组织我们的测试集合。

== 测试套件设计

// 翻译中。。。。 (yuansheng)

使用 Test::Nginx 驱动我们的测试套件，通常使用一个公共的目录结构以及规范的测试文件名样式，用来组织我们的测试集合。 这些会让用户更容易明白，这些测试用例在项目源码中的目录关系，以及测试用例的使用情况。 它不是必须的，然而，按照这种常见约定组织，还是非常推荐的。

Projects using Test::Nginx to drive their test suites usually have a common directory layout and common test file name patterns to organize their tests. This makes the user easy to reason about the location of the test suite in a project source tree and the usage of the tests. It is not really required, however, to use this common convention; it is just highly recommended.

按照约定，这些项目在当前源码的根目录下，有个 t/ 目录用来存放测试文件。 每个测试文件包含一些具有关联度的测试用例，保存在扩展名为 .t 的文件中，可以很容易表明这是 “测试文件”。 下面的目录树结构，是真实项目 link:https://github.com/openresty/headers-more-nginx-module[headers-more-nginx-module] 的测试套件：

By convention, such projects have a t/ directory at the root of their source tree where test files reside in. Each test file contains test cases that are closely related in some way and has the file extension .t to easily identify themselves as “test files”. Below is the directory tree structure of a real-world test suite inside the link:https://github.com/openresty/headers-more-nginx-module[headers-more-nginx-module] project:

…. └── t ├── bug.t ├── builtin.t ├── eval.t ├── input-conn.t ├── input-cookie.t ├── input-ua.t ├── input.t ├── phase.t ├── sanity.t ├── subrequest.t ├── unused.t └── vars.t ….

当你有很多测试文件，你可以对 t/ 使用子目录拆分完成更深的分组归类。例如，link:https://github.com/openresty/lua-nginx-module[lua-nginx-module] 项目，我们在 t/ 目录下就有 023-rewrite/ 和 024-access/ 两个子目录。

When you have many test files, you can also group them further with sub-directories under `t/`. For example, in the link:https://github.com/openresty/lua-nginx-module[lua-nginx-module] project, we have sub-directores like `023-rewrite/` and `024-access/` under its `t/` directory.

本质上，每一个 `.t` 文件是一个 Perl 脚本文件，它的执行由 `perl` 或者 `Perl` 世界比较普遍的工具名叫 link:http://perldoc.perl.org/prove.html[prove] 完成的。 我们通常使用 `prove` 命令行工具来执行这些 `.t` 文件来获取测试结果。 尽管这些 `.t` 文件每一个都是 Perl 脚本，但他们通常都不包含任何 Perl 脚本。 取而代之的，这里声明的所有测试用例，仅仅是放在 `.t` 文件中统一格式的基本 “数据”。

In essence, each `.t` file is a Perl script file runnable by either `perl` or Perl's universal test harness tool named link:http://perldoc.perl.org/prove.html[prove]. We usually use the `prove` command-line utility to run such `.t` files to obtain test results. Although `.t` files are Perl scripts per se, they usually do not have much Perl code at all. Instead, all of the test cases are declared as cleanly formatted “data” in these `.t` files.

注意：我们这里使用的测试套件设计约定，在 Perl 社区已经延用多年。 因为 `Test::Nginx` 使用 Perl 编写的，并且在 Perl 的测试工具链中被重用，我们只需要让 NGINX 和 OpenResty 世界简单的遵循这些约定就好。

NOTE: The test suite layout convention we use here are also used by the Perl community for many years. Because `Test::Nginx` is written in Perl and reuses Perl's testing toolchain, it makes sense for us to simply follow that convention in the NGINX and OpenResty world as well.

Test File Layout

Summary: 测试文件通常带 `.t` 后缀，以此跟源码树中的其他文件相区分。每个测试文件本身是一个 Perl 脚本。

== Test File Layout

// 翻译中。。。。（luozexuan） :special_data_line: DATA

Test files usually have a common file extension, .t , to distinguish themselves from other types of files in the source tree. Each test file is a Perl script per se. Test::Nginx follows a special design that decomposes each test file into two main parts: the first part is a very short prologue that consists of a few lines of Perl code while the second part is a listing of the test cases in a special data format. These two parts are separated by the following special line

测试文件通常带 .t 后缀，以此跟源码树中的其他文件相区分。每个测试文件本身是一个 Perl 脚本。 Test::Nginx 遵循一套特别的设计，把每个测试文件分成两部分：第一部分是非常短的序言（prologue），包括若干行 Perl 代码； 第二部分则是按特定格式组织起来的测试用例列表。 这两部分由下面这一行分隔开来。

…. DATA ….

The perl interpreter or the prove utility stop interpreting the file content as Perl source code until they see this special line. Everything after this line is treated as *data* in plain text that is reachable by the Perl code above this line. The most interesting part of each .t test file is the stuff after this line, i.e., the data part.

perl 解释器或 prove 工具一旦遇到这一行，就会停止以 Perl 代码的形式解释文件内容。 在这之后的文本，都会被当作前面的 Perl 代码所能操作的 *数据* 。 这部分，即数据部分，是每个 .t 测试文件中最有趣的地方，

NOTE: The special {special_data_line} notation is a powerful feature of the Perl programming language that allows embedding arbitrary free-text data in any Perl script files that can be manipulated by the containing Perl scripts themselves. Test::Nginx takes advantage of this feature to allow data-driven test case specifications in a simple format or language that is easily understandable by everyone, even those without any prior experiences in Perl programming.

NOTE: {special_data_line} 标记是 Perl 编程语言的一个强大特性， 它允许你添加任意纯文本数据到 Perl 脚本文件中， 以供该脚本使用。 凭借这一特性， Test::Nginx 实现了一套简单的测试格式，亦可说是语言。 用户无需学习 Perl， 即可上手编写简明易懂的数据驱动的测试用例。

=== The Prologue Part 序言部分

The first part, i.e., the “prologue” above the `{special_data_line}` line is usually just a few lines of Perl code. You do not have to know Perl programming to write them down because they are so simple and seldom or never change. The simplest Perl code prologue is as follows:

第一部分，即位于 `{special_data_line}` 之上的“序言部分”，通常只是寥寥几行 Perl 代码。 这些代码是如此地简单和通用，以至于不需要懂 Perl 语法就能编写。最简单的 Perl 序言如下：

```
[source,perl,linenums]
use Test::Nginx::Socket 'no_plan'; run_tests(); —-
```

The first line is just loading the Perl module (or class), `Test::Nginx::Socket` and passing the option `'no_plan'` to it to disable test plans (we will talk more about test plans in later chapters and we do not bother worrying about it here). `Test::Nginx::Socket` is one of the most popular class in the `Test::Nginx` test framework. The second line just calls the `run_tests` Perl function imported automatically from the `Test::Nginx::Socket` module to run all the test cases defined in the data part of the test file (i.e., the things coming after the `{special_data_line}` line).

这里的第一行加载了 Perl 模块（或类）—— `Test::Nginx::Socket` 并传递 `'no_plan'` 参数来禁用测试计划（test plan，我们后面会提到更多关于它的内容，这里先搁置不管）。 `Test::Nginx::Socket` 是 `Test::Nginx` 测试框架中最常用的类之一。 第二行调用了从 `Test::Nginx::Socket` 模块中自动引入的 `run_tests` 来运行测试文件中定义的所有测试用例（即 `{special_data_line}` 之后的内容）。

There are, however, more complicated prologue parts in many real-world test suites. Such prologues usually define some special environment variables or Perl variables that can be shared and referenced in the test cases defined in the “data part”, or just call some other Perl functions imported by the `Test::Nginx::Socket` module to customize the testing configurations and behaviors for the current test file. We will return to such fancier prologues in later sections. They can be very helpful in some cases.

在实际的测试代码中，序言部分的内容往往会更加复杂。 这些内容包括定义能在各个用例中共享的环境变量或 Perl 变量，或者调用 `Test::Nginx::Socket` 模块中的函数来自定义当前测试文件的配置和行为。 在后面的段落中，我们会回过头介绍这些序言。在某些场景下，它们真的挺有用。

NOTE: Perl allows function calls to omit the parentheses if the context is unambiguous. So we may see Perl function calls without parentheses in real-world test files’ prologue part, like `run_tests;` . We may use such forms in examples presented in later sections because they are more compact.

NOTE: 在上下文不引起歧义的情况下，Perl 允许在调用函数时省略括号。 所以在实际项目中的测试文件的序言部分，我们能够看到省略括号的用法，比如 run_tests; 。在接下来的段落中，我们会使用这种形式，因为它让代码更紧凑。

=== The Data Part 数据部分

The data part is the most important part of any test files powered by Test::Nginx . This is where test cases reside. It uses a simple specification format to express test cases so that the user does not use Perl or any other general-purpose languages to present the tests themselves. This special specification format is an instance of Domain-Specific Languages (DSL) where the “domain” is defined as testing code running upon or inside NGINX. Use of a DSL to present test cases open the door of presenting the test cases as *data* instead of code. This is also why Test::Nginx is a data-driven testing framework.

数据部分是 Test::Nginx 测试文件中最为重要的部分。这是测试用例所在之处。 它设计了一种简单的数据格式，让用户无需使用 Perl 或其他通用语言来撰写测试用例。这种特别的专用格式适用于为运行在NGINX上的代码编写测试这一领域，算是领域特定语言（DSL）的一种。 DSL的使用打开了新世界的大门，使得用户可以把测试用例当作 *数据* 而非代码。 这也是为什么 Test::Nginx 是一个数据驱动的测试框架。

The test case specification in the data part is composed by a series of *test blocks*. Each test block usually corresponds to a single test case, which has a *title*, an optional *description*, and a series of *data sections*. The structure of a test block is described by the following template.

测试用例在数据部分由一组 *测试块* 构成。 每个测试块对应单个策划是用例，包含一个 *标题* 、一个可选的 *描述* 和几个 *数据节*。 下面的模板描述了一个测试块的结构：

[source,test-base]

=== title optional description goes here… — section1 value1 goes here — section2 value2 is here — section3 value3 —-

==== Block Titles 块标题

As we can see, each test block starts with a title line prefixed by three equal sign (===). It is important to *avoid* any leading spaces at the beginning of the line. The title is mandatory and is important to describe the intention of the current test case in the most concise form, and also to identify the test block in the test report when test failures happen. By convention we put a TEST N: prefix in this title, for instance, TEST 3: test the simplest form . Don’t worry about maintaining the test ordinal numbers in these titles yourself, we will introduce a command-line utility called link:https://raw.githubusercontent.com/agentzh/old-openresty/master/bin/reindex[reindex] in a later section that can automatically update the ordinal numbers in the block titles for you.

如你所见，每个测试块的标题以三个等于号（ === ）作为开头。注意行头 *不能* 留空。 标题是必须要有的，你需要在此用精炼的语言描述当前测试用例的意图。 当测试失败时，你还需要靠它定位具体的测试块。简明起见，我们会给标题加上一个 TEST N: 前缀， 比如 TEST3: test the simplest form 。 无需担心测试序号的维护问题，我们稍后会介绍一个名为 link:https://raw.githubusercontent.com/agentzh/old-openresty/master/bin/reindex[reindex] 的命令行工具，它会替你自动更新块标题的序号。

==== Block Descriptions 块描述

Each test block can carry an optional description right after the block title line. This description can span multiple lines if needed. It is a more detailed description of the intention of the test block than the block title and may also give some background information about the current test. Many test cases just omit this part for convenience.

每个测试块可以在块标题下面加上可选的块描述。如果需要的话，这个描述可以分成多行。 比起块标题，它更加详尽地说明了测试块的意图，还可能提供当前测试的一些背景信息。 出于方便，许多测试用例省略了这一部分。

==== Data Sections 数据节

Every test block carries one or more *data sections* right after the block description (if any). Data sections always have a name and a value, which specify any input data fields and the *expected* output data fields.

（如果有的话）块描述后面紧接着一到多个 *数据节*。 数据节总是带有一个名字和对应值，指定输入的数据类型和 *期望的* 输出数据类型。

The name of a data section is the word after the line prefix --- . Spaces are allowed though not syntactically required after --- . We usually use a single space between the prefix and the section name for aesthetic considerations and we hope that you follow this convention as well. The section names usually contain just alphanumeric letters and underscore characters.

一个数据节的名字指 --- 前缀后面的文字。 --- 后面可以带空格，尽管这不是语法上的要求。 出于美观起见，我们通常在前缀和名字间留出一个空格，也希望你能照着做。 数据节的名字通常只由字母、数字和下划线组成。

Section values are specified in two forms. One is all the lines after the section name line, before the next section or the next block. The other form is more concise and specifies the value directly on the same line as the section name, but right after the first colon character (:). The latter form requires that the value contains no line-breaks. Any spaces around the colon are always discarded and never count as a part of the section value; furthermore, the trailing line-break character in the one-line form does not count either.

数据节的值有两种表现形式。一种是名字那一行之后、下一数据节或测试块之前的各行。 另一种形式更为紧凑，你可以直接在名字的同一行，以冒号（：）隔开指定值。 前提是指定的值只有一行。冒号两边的空格会被忽略，不作为值的一部分；另外，行结尾的换行符也不会算入。

If no visible values come after the section name in either form, then the section takes an empty string value, which is still a *defined* value, however. On the other hand, omitting the section name (and value) altogether makes that section *undefined*.

如果名字之后不带任何可见的值，该数据节的值为空字符串，这依然算一个 *有定义* 的值。 但是，如果某个数据节没有名字（也没值），那么它是 *未定义* 的。

Test::Nginx offers various pre-defined data section names that can be used in the test blocks for different purposes. Some data sections are for specifying input data, some are for expected output, and some for controlling whether the current test block should be run at all.

Test::Nginx 预先定义了一些在测试块中有特殊用途的数据节名。 有些数据节用来指定输入数据，有些用来指定期望的输出，还有些用来标记当前测试块是否需要运行。

It is best to explain data sections in a concrete test block example.

下面结合一个具体例子来解释数据节。

[source,test-base]

=== TEST 1: hello, world This is just a simple demonstration of the echo directive provided by ngx_http_echo_module. ngx_http_echo_module 提供的 echo 指令的一个简单演示。 — config location = /t { echo “hello, world!”; } — request GET /t — response_body hello, world! — error_code: 200 —-

Here we have two input data sections, config and request , for specifying a custom NGINX configuration snippet in the default server {} and the HTTP request sent by the test scaffold to the test NGINX server, respectively. In addition, we have one output data section, response_body , for specifying the expected response body output by the test NGINX server. If the actual response body data is different from what we specify under the response_body section, this test case fails. We have another output data section, error_code , which specifies its value on the same line of the section name. We see that a colon character is used to separate the section name and values. Obviously, the error_code section specifies the expected HTTP response status code, which is 200.

这里有两个关于输入的数据节， config 和 request 。 前者自定义 server {} 内的 NGINX 配置，后者指定测试脚手架发给 NGINX 服务器的 HTTP 请求。 另外，还有一个关于输出的数据节—— response_body ，指定测试时期望 NGINX 返回的响应。 如果实际上的响应不同于 response_body 的值，该测试用例就失败。 我们还有另外一

个关于输出的数据节—— error_code ，它的值和名字都在同一行。 可以看到，数据节的名字和值之间以冒号隔开。 显而易见， error_code 数据节所指定的，是期望的 HTTP 响应状态码——在这里是200。

Empty lines around data sections are always discarded by Test::Nginx::Socket . Thus the test block above can be rewritten as below without changing its meaning.

数据节前后的空行会被 TEST::Nginx::Socket 忽略掉。 所以上面的测试块示例可以重写成下面的样子：

[source,test-base]

=== TEST 1: hello, world This is just a simple demonstration of the echo directive provided by ngx_http_echo_module. ngx_http_echo_module 提供的 echo 指令的一个简单演示。

— config location = /t { echo “hello, world!”; }

— request GET /t

— response_body hello, world!

— error_code: 200

Some users prefer this style for aesthetic reasons. We are free to choose whatever form you like.

出于审美上的理由，有些用户更钟爱这种写法。萝卜白菜，各有所爱。

There are also some special data sections that specify neither input nor output. They are just used to *control* how test blocks are run. For example, the ONLY section makes *only* the current test block in the current test file run and all the other test blocks are skipped. This is extremely useful for running an individual test block in any given file, which is a common requirement while debugging a particular test failure. Also, the special SKIP section can skip running the containing test block unconditionally, handy for preparing test cases for future features without introducing any expected test failures. We will visit more such “control sections” in later sections.

有些特殊的数据节跟输入和输出都没有关系。它们只用于 *控制* 测试块运行的方式。举个例子， ONLY 数据节仅让所在的测试块运行，跳过当前测试文件的其他测试块。 在调试某个失败的测试时，使用该数据节仅运行单一的测试块，是个很实用的方法。 另一个特殊的数据节 SKIP 可以无条件地跳过所在的测试块。 在准备针对待开发特性的测试时，你可以先用该数据节避免意料之中的测试错误。 稍后我们会介绍更多类似这样的“控制数据节”。

We shall see, in a later section, that the user can define her own data sections or extending existing ones by writing a little bit of custom Perl code to satisfy her more complicated testing requirements.

在下文我们还会提到，用户可以通过编写 Perl 代码自定义数据节或拓展现有数据节来满足自己复杂的测试要求。

==== Section Filters 节过滤器

Data sections can take one or more *filters*. Filters are handy when you want to adjust or convert the section values in certain ways.

一个数据节可以有一到多个 *过滤器*。过滤器可以用来调整或者转换数据节的值。

Syntactically, filters are specified right after the section name with at least one space character as the separator. Multiple filters are also separated by spaces and are applied in the order they are written.

过滤器放在数据节名字的后面，并至少用一个空格隔开。多 个过滤器之间以空格隔开，并按代码中的顺序依次调用。

`Test::Nginx::Socket` provides many filters for your convenience. Consider the following data section from the aforementioned test block.

`Test::Nginx::Socket` 提供了许多常用的过滤器。看下来自前述测试块的一个数据节：

[source,test-base]

— error_code: 200 —-

If we want to place the section value, 200, in a separate line, like below,

如果你想把200这个值单独放在一行，就像这样，

[source,test-base]

— error_code 200 —-

then the section value would contain a trailing new line, which leads to a test failure. This is because the one-line form always excludes the trailing new-line character while the multi-line form always includes one. To explicitly exclude the trailing new-line in the multi-line form, we can employ the `chomp` filter, as in

那么数据节的值里就会有一个换行符，导致状态码无法匹配。 这是因为单行模式的值会移除结尾的换行符，而多行模式不会。要想在多行模式里去掉结尾的换行，我们可以采用 `chomp` 过滤器，像这样

[source,test-base]

— error_code chomp 200 —-

Now it has exactly the same semantics as the previous one-line form.

现在它的语义跟之前的单行模式一样了。

Some filters have more dramatic effect on the section values. For instance, the `eval` filter evaluates the section value as arbitrary Perl code, and the Perl value resulted from the execution will be used as the final section value. The following section demonstrates using the `eval` filter to produce 4096 a's:

有些过滤器效果更加显著。 比如， `eval` 过滤器会把给定的值当作 Perl 代码执行，返回执行结果作为最终的值。 下面的数据节示范如何用 `eval` 过滤器生成4096个 'a' :

[source,test-base]

— response_body eval "a" x 4096 —-

The original value of the `response_body` section above is a Perl expression where the `x` symbol is a Perl operator is used to construct a string that repeats the string specified as the left-hand-side N times where N is specified by the right-hand-side. The resulting 4096-byte Perl string after evaluating this expression dictated by the `eval` filter will be used as the final section value for comparison with the actual response body data. It is obvious that use of the `eval` filter and a Perl expression here is much more readable and manageable by directly pasting that 4096-byte string in the test block.

`response_body` 数据节原来的值是一个 Perl 表达式，其中 `x` 运算符会把左边的字符串重复 N 次，N是它右边的数字。 `eval` 返回的包含4096个字符的 Perl 字符串会被用作数据节最终的值，跟响应体作比较。 显而易见，无论从可读性还是从可维护性上看，使用 `eval` 过滤器和一个 Perl 表达式要比直接贴上4096个字符更胜一筹。

As with data sections, the user can also define her own filters, as we shall see in a later section.

我们稍后会提到，用户可以定义自己的过滤器，并应用在数据节上。

=== A Complete Example 一个完整的例子

We can conclude this section by a complete test file example given below, with both the prologue part and the data part.

我们以一个完整的，包含序言部分和数据部分的测试文件示例结束本节。

[source,test-base]

use Test::Nginx::Socket ‘no_plan’;

run_tests();

DATA

=== TEST 1: hello, world This is just a simple demonstration of the echo directive provided by ngx_http_echo_module. ngx_http_echo_module 提供的 echo 指令的一个简单演示。 — config location = /t { echo “hello, world!”; } — request GET /t — response_body hello, world! — error_code: 200 —-

We will see how to actually run such test files in the next section.

下一节我们看看如何把它运行起来。

NOTE: The test file layout described in this section is exactly the same as the test files based on other test frameworks derived from `Test::Base`, the superclass of `Test::Nginx::Socket`, except those specialized test sections and specialized Perl functions defined only in `Test::Nginx::Socket`. All the `Test::Base` derivatives share the same basic layout and syntax. They proudly inherit the same veins of blood.

NOTE: 本节中描述的测试文件布局跟其他基于 `TEST::Base` （`Test::Nginx::Socket` 的父类）的测试框架的几乎一样，除了一些特殊的数据节和仅在 `Test::Nginx::Socket` 定义的 Perl 函数。所有衍生自 `Test::Base` 的框架都继承了相同的基本布局和语法。毕竟它们本是同根生。

Running Tests

Summary: 一如大多数基于 Perl 的测试框架，`Test:Nginx` 依靠命令行工具 `prove` 来运行测试文件。

== Running Tests

Like most Perl-based testing frameworks, Test:Nginx relies on Perl's prove command-line utility to run the test files. The prove utility is usually shipped with the standard perl distribution so we should already have it when we have perl installed.

一如大多数基于 Perl 的测试框架， Test:Nginx 依靠命令行工具 prove 来运行测试文件。 prove 工具通常随 perl 一同发布，所以在安装了 perl 之后，我们一般也有了 prove 。

Test::Nginx always invokes a real NGINX server and a real socket client to run the tests. It automatically uses the nginx program found in the system environment PATH . It is your responsibility to specify the right nginx in your PATH environment for the test suite. Usually we just specify the path of the nginx program inside the OpenResty installation tree. For example,

Test::Nginx 总是启动一个真实的 NGINX 服务器和真实的套接字客户端来运行测试。它默认使用系统的环境变量 PATH 来搜索 nginx 程序。你需要保证在这个环境变量里面能够找到它。 通常我们会指定 OpenResty 安装包里面的 nginx 程序路径。举个例子，

[source,bash]

export PATH=/usr/local/openresty/nginx/sbin:$PATH —-

Here we assume that OpenResty is installed to the default prefix, i.e., /usr/local/openresty/ .

这里我们假设 OpenResty 安装在默认路径，即 /usr/local/openresty/ 。

You can always use the which command to verify if the PATH environment is indeed set properly:

你可以使用 which 命令验证 PATH 环境变量是否设置正确：

[source,console]

$ which nginx /usr/local/openresty/nginx/sbin/nginx —-

For convenience, we usually wrap such environment settings in a custom shell script so that we do not risk polluting the system-wide or account-wide environment settings nor take on the burden of manually setting the environments manually for every shell session. For example, I usually have a local bash script named `go` in each project I work on. A typical `go` script might look like below

为了方便，我们通常把类似这样的环境变量设置封装在一个自定义 shell 脚本中，在避免污染全局或本账户的环境变量的同时，也甩掉在每个 shell 会话中手工设置环境变量的负担。 举个例子，我通常在参与的每个项目中放置一个名为 `go` 的本地 bash 脚本。 这个脚本看上去大概是这个样子：

[source,bash]

#!/usr/bin/env bash

export PATH=/usr/local/openresty/nginx/sbin:$PATH

exec prove “$@”

Then we can use this `./go` script to substitute the `prove` utility in any of the subsequent commands involving `prove`.

然后我们可以在要用到 `prove` 工具的场景中，用这个 `./go` 脚本替换掉它。

Because `Test::Nginx` makes heavy use of environment variables for the callers to fine tune the testing behaviors (as we shall see in later sections), such shell wrapper scripts also make it easy to manage all these environment variable settings and hard to get things wrong.

由于调整 `Test::Nginx` 的测试行为重度依赖于对环境变量的使用（接下来我们会看到这一点），这样的包装脚本也让相关环境变量的管理变得简单，让忙中出错变得困难。

NOTE: Please do not confuse the name of this bash script with Google’s Go programming language. It has nothing to do with the Go language in any way.

NOTE: 请不要把这个 bash 脚本的名字跟 Google 的 Go 编程语言弄混。 它跟 Go 语言风牛马不相及。

=== Running A Single File 运行单个测试

If you want to run a single test file, say, t/foo.t , then all you need to do is just typing the following command in your terminal.

假设你想要运行某个测试文件，比如说， t/foo.t ，你所需的只是在终端敲入如下命令。

[source,bash]

prove t/foo.t —-

Here inside t/foo.t we employs the simple test file example presented in the previous section. We repeat the content below for the reader's convenience.

t/foo.t 里的内容正是上一节展示的测试文件示例。我们将它列在下面，以便读者查看。

[source,test-base] .t/foo.t —- use Test::Nginx::Socket 'no_plan';

run_tests();

DATA

=== TEST 1: hello, world This is just a simple demonstration of the echo directive provided by ngx_http_echo_module. ngx_http_echo_module 提供的 echo 指令的一个简单演示。 — config location = /t { echo "hello, world!"; } — request GET /t — response_body hello, world! — error_code: 200 —-

It is worth mentioning that we could run the following command instead if we have a custom wrapper script called ./go for prove (as mentioned earlier in this section):

值得一提的是，假如我们用一个名为 ./go 的脚本包装了 prove （正如本节开头部分提到的），运行命令的方式会是这样：

[source,bash]

./go foo.t —-

When everything goes well, it generates an output like this:

如果一切顺利，会有如下输出：

…. t/foo.t .. ok All tests successful. Files=1, Tests=2, 0 wallclock secs (0.02 usr 0.01 sys + 0.08 cusr 0.03 csys = 0.14 CPU) Result: PASS ….

This is a very concise summary. The first line tells you all tests were passed while the second line gives you a summary of the number of test files (1 in this case), the number of tests (2 in this case), and the wallclock and CPU times used to run all the tests.

这段总结惜字如金。第一行告诉你所有测试都通过了，第二行则给出测试文件数（1）、测试数（2）以及运行全部测试所花费的墙上时间和CPU用时。

It is interesting to see that we have only one test block in the sample test file but in the test summary output by `prove` we see that the number of tests are 2. Why the difference? We can easily find it out by asking `prove` to generate a detailed test report for all the individual tests. This is achieved by passing the `-v` option (meaning “verbose”) to the `prove` command we used earlier:

有趣的是，虽然测试文件示例中只有一个测试块，但是 `prove` 输出的测试总结却显示测试数为2。 为什么会这样？让 `prove` 给所有测试生成一份详细的测试报告来揭示真相。 这需要给之前的 `prove` 命令添加 `-v` 选项（ v 代表 `verbose` ）：

[source,bash,linenums]

prove -v t/foo.t —-

Now the output shows all the individual tests performed in that test file:

现在可以在输出里看到测试文件中每个测试的运行情况：

…. t/foo.t .. ok 1 - TEST 1: hello, world - status code ok ok 2 - TEST 1: hello, world - response_body - response is expected (req 0) 1..2 ok All tests successful. Files=1, Tests=2, 0 wallclock secs (0.01 usr 0.01 sys + 0.07 cusr 0.03 csys = 0.12 CPU) Result: PASS ….

Obviously, the first test is doing the status code check, which is dictated by the `error_code` data section in the test block, and the second test is doing the response body check, required by the `response_body` section. Now the mystery is solved.

显而易见，第一个测试检查状态码，这是 `error_code` 数据节的命令； 而第二个测试检查响应体，则是 `response_body` 要求的。原来如此。

It is worth mentioning that the `--- error_code: 200` section is automatically assumed when no `error_code` section is explicitly provided in the test block. So our test block above can be simplified by removing the `--- error_code: 200` line without affecting the number of tests. This is because that checking 200 response status code is so common that `Test::Nginx` makes it the default. If you expect a different status code, like 500, then just add an explicit `error_code` section.

值得一提的是， 如果测试块中没有指定 `error_code` ，默认会设置 `--- error_code: 200` 。 因此我们可以移除多余的 `--- error_code: 200` ，而不影响测试的总数。 毕竟检查响应状态码是否为200是个常见的需求，所以 `Test::Nginx` 把它设置为默认行为。 如果你期望的是其他状态码，比如500，那么就加上一个 `error_code` 数据节。

From this example, we can see that one test block can contain multiple tests and the number of tests for any given test block can be determined or predicted by looking at the data sections performing output checks. This is important when we provide a “test plan” ourselves to the test file where a “test plan” is the exact number of tests we *expect* the current test file to run. If a different number of tests than the plan were actually run, then the test result would be considered malicious even when all the tests are passed successfully. Thus, a test plan adds a strong constraint on the total number of tests expected to be run. For our `t/foo.t` file here, however, we intentionally avoid providing any test plans by passing the `'no_plan'` argument to the `use` statement that loads the `Test::Nginx::Socket` module. We will revisit the “test plan” feature and explain how to provide one in a later section.

从上面的例子里，我们可以看到一个测试块可以包含多个测试， 而它的测试数则取决于检查输出的数据节。 当我们需要给测试文件设定一个“测试计划”时，要记住，“测试计划”的值等于我们 *期望* 运行的测试数。 如果实际运行的测试数不同于计划的测试数，那么即使全部测试都通过了，测试结果也不会被接受。 所以，测试计划给期望运行的测试总数添加了一个强约束。 不过在 `t/foo.t` 里面，当 `use` 加载 `Test::Nginx::Socket` 的时候，我们传递了 `'no_plan'` 参数，借此避免对测试计划的设定。 我们会在后面继续讨论“测试计划”的概念，并展示如何设定它。

=== Running Multiple Files 运行多个文件

Running multiple test files are straightforward; just specify the file names on the `prove` command line, as in

运行多个文件的方式跟单个文件的差不多，像这样在 `prove` 命令中指定它们的名字即可：

[source,bash]

prove -v t/foo.t t/bar.t t/baz.t —-

If you want to run all the test files directly under the `t/` directory, then using a shell wildcard can be handy:

如果你打算一口气运行 `t/` 文件夹下的所有测试文件，需要借助通配符的力量：

[source,bash]

prove -v t/*.t —-

In case that you have sub-directories under `t/`, you can specify the `-r` option to ask `prove` to recursively traverse the while directory tree rooted at `t/` to find test files:

如果在 `/t` 下有子目录，你可以指定 `-r` 选项，告诉 `prove` 从根目录 `t/` 递归查找测试文件：

[source,bash]

prove -r t/ —-

This command is also the standard way to run the whole test suite of a project.

这个命令也是运行一个项目的所有测试用例的标准方法。

=== Running Individual Test Blocks 运行单个测试块

Test::Nginx makes it easy to run an individual test block in a given file. Just add the special data section ONLY to that test block you want to run individually and prove will skip all the other test blocks while running that test file. For example,

Test::Nginx 提供了运行给定文件的单个测试块的捷径。 仅需在要单独运行的测试块中添加 ONLY 数据节， prove 就会在运行测试文件时跳过其他测试块。 举个例子：

[source,test-base]

=== TEST 1: hello, world This is just a simple demonstration of the echo directive provided by ngx_http_echo_module. ngx_http_echo_module 提供的 echo 指令的一个简单演示。 — config location = /t { echo “hello, world!”; } — request GET /t — response_body hello, world! — ONLY —-

Now prove won’t run any other test blocks (if any) in the same test file.

现在 prove 不再会运行同一测试文件中的其他测试（如果有的话）。

This is very handy while debugging a particular test block. You can focus on one test case at a time without worrying about other unrelated test cases stepping in your way.

这一特性有助于你调试特定的测试块。你可以仅关注某一测试的运行结果，把其他不相关的用例都抛到九宵云外。

When using the link:http://www.vim.org/[Vim] editor, we can quickly insert a --- ONLY line to the test block we are viewing in the vim file buffer, and then type :!prove % in the command mode of vim without leaving the editor window. This works because vim automatically expands the special % placeholder with the path of the current active file being edited. This workflow is great since you never leave your editor window and you never have to type the title (or other IDs) of your test block nor the path of the containing test file. You can quickly jump between test blocks even across different files. Test-driven development usually demands very frequent interactions and iterations, and Test::Nginx is particularly optimized to speed up this process.

在使用 link:http://www.vim.org/[Vim] 编辑器时，我们可以插入一行 --- ONLY 到位于 vim 的当前文件缓冲区的测试块中， 然后在命令模式下输入 :!prove % ，就能在不离开编辑器的同时运行测试。 这是因为 vim 会把 % 占位符展开成当前编辑的文件的路径。 这是个很棒的工作流，因为你既不需要切换编辑器界面，也不需要输入测试块标题（或其他ID）和测试文件的路径。你可以自如地在来自不同文件的测试块间切换。 测试驱动开发通常要求非常频繁地进行交互和迭代，而 Test::Nginx 正是优化了这一过程。

Sometimes you may forget to remove the --- ONLY line from some test files even after debugging, this will incorrectly skip all the other tests in those files. To catch such mistakes, Test::Nginx always reports a warning for files using the ONLY special section, as in

有时候你可能忘记在调试后移除 --- ONLY ，导致文件中的其他测试没有被执行。 为了避免这种疏忽， Test::Nginx 总会对使用了 ONLY 数据节的测试文件发出提醒，像这样：

[source,console]

$ prove t/foo.t t/foo.t .. # I found ONLY: maybe you’re debugging? t/foo.t .. ok All tests successful. Files=1, Tests=2, 0 wallclock secs (0.01 usr 0.00 sys + 0.09 cusr 0.03 csys = 0.13 CPU) Result: PASS —-

This way it is much easier to identify any leftover --- ONLY lines.

现在找出遗留的 --- ONLY 行变得简单多了。

Similar to ONLY , Test::Nginx also provides the LAST data section to make the containing test block become the last test block being run in that test file.

类似于 ONLY ， Test::Nginx 也提供了 LAST 数据节，可以让所在的测试块运行在测试文件的最后。

NOTE: The special data sections ONLY and LAST are actually features inherited from the Test::Base module.

NOTE: ONLY 和 LAST 这样的特殊的数据节实际上都是来自于 Test::Base 模块的特性。

=== Skipping Tests 跳过测试

We can specify the special SKIP data section to skip running the containing test block unconditionally. This is handy when we write a test case that is for a future feature or a test case for a known bug that we haven’t had the time to fix right now. For example,

我们可以使用 SKIP 数据节无条件跳过所在的测试块。 当我们为了一个尚未完成的功能或 bug 修复写一个测试用例，就可以用上它。举个例子：

[source,test-base]

=== TEST 1: test for the future — config location /t { some_fancy_directive; } — request GET /t — response_body blah blah blah — SKIP —-

It is also possible to skip a whole test file in the prologue part. Just replace the `use` statement with the following form.

在序言部分跳过整个测试文件也是可行的。仅需把 `use` 语句替换成下面的形式。

[source,Perl]

use Test::Nginx::Socket skip_all => “some reasons”; —-

Then running the test file gives something like follows.

然后运行这个测试文件会有如下的输出。

…. t/foo.t .. skipped: some reasons ….

NOTE: It is also possible to conditionally skip a whole test file but it requires a little bit of Perl programming. Interested readers can try using a `BEGIN {}` before the `use` statement to calculate the value of the `skip_all` option on the fly.

NOTE: 有选择地跳过整个测试文件也是可能的，不过这需要一点 Perl 编程的技巧。感兴趣的读者可以尝试在 `use` 语句前使用一个 `BEGIN {}` 即时计算出 `skip_all` 选项的值。

=== Test Running Order 测试运行的顺序

==== Test File Running Order 测试文件运行的顺序

Test files are usually run by the alphabetical order of their file names. Some people prefer explicitly controlling the running order of their test files by prefixing the test file names with number sequences like `001-`, `002-`, and etc.

测试文件通常按文件名的字母表顺序依序运行。 有些人会在测试名前面添加数字前缀，比如 `001-` 、 `002-` 等等，来控制测试文件运行顺序。

The test suite of the link:https://github.com/openresty/lua-nginx-module#readme[ngx_http_lua] module follows this practice, for example, which has test file names like below

link:https://github.com/openresty/lua-nginx-module#readme[ngx_http_lua] 的测试套件就是这么做的，它的测试文件命名如下：

…. t/000-sanity.t t/001-set.t t/002-content.t t/003-errors.t … t/139-ssl-cert-by.t ….

Although the prove utility supports running test files in multiple parallel jobs via the -jN option, Test::Nginx does not really support this mode since all the test cases share exactly the same test server directory, t/servroot/, and the same listening ports, as we have already seen, while parallel running requires strictly isolated running environments for each individual thread of execution. One can still manually split the test files into different groups and run each group on a different (virtual) machine or an isolated environment like a Linux container.

尽管 prove 支持通过 -jN 选项并行运行多个测试， Test::Nginx 并不支持这种模式，因为所有测试用例会用到同一个测试服务器目录 t/serroot/ ，监听同一个端口。 而并行运行测试需要给每个线程隔离的运行环境。 当然你还是可以把测试文件分到不同的组里，每个组在一个不同的虚拟／实体机或者隔离环境（如 Linux 容器）下运行。

==== Test Block Running Order 测试块运行的顺序

By default, the Test::Nginx scaffold *shuffles* the test blocks in each file and run them in a *random* order. This behavior encourages writing self-contained and independent test cases and also increases the chance of hitting a bug by actively mutating the relative running order of the test cases. This may, indeed, confuse new comers, coming from a more traditional testing platform.

默认情况下， Test::Nginx 脚手架会 *乱序* 运行每个文件中的测试块。 这一行为鼓励用户编写互不干扰、独立自主的测试用例，同时也通过变换用例的相对顺序提高找出 bug 的概率。 对于熟悉传统测试框架的新用户来说，这确实有点奇怪。

We can always disable this test block shuffling behavior by calling the Perl function, no_shuffle(), imported by the Test::Nginx::Socket module, before the run_tests() call in the test file prologue. For example,

在调用 run_tests() 之前调用 Perl 函数 no_shuffle() ，我们可以关掉乱序运行测试的行为。 这个函数来自于 Test::Nginx::Socket 模块。举个例子，

[source,Perl]

use Test::Nginx::Socket ‘no_plan’;

no_shuffle(); run_tests();

DATA ··· —-

With the no_shuffle() call in place, the test blocks are run in the exact same order as their appearance in the test file.

由于 no_shuffle() 的调用，测试块将严格按照在测试文件中定义的顺序运行。

测试前的准备

Summary: 正如我们上个章节里面看到的， `Test::Nginx` 提供了一个简单规范的格式来表达测试用例。

== 测试前的准备

正如我们上个章节里面看到的， Test::Nginx 提供了一个简单规范的格式来表达测试用例。 每个测试用例都用一个测试块来表示。一个测试块由一个标题、一个可选的描述和几个数据节footnote:[data section 本书统一翻译为『数据节』]， 用于指定输入和期望的输出。在这一节我们将仔细看看对于不同的测试要求，如何准备这样的测试案例。

设计测试用例在很多方面都是一门艺术。有时候可能设计测试案例花费的时间和精力，多于实现要测试的功能， 这个和我们自己的经验有关。 Test::Nginx 努力尝试让编写测试用例尽可能的简单， 但仍然做不到让整个测试用例设计的过程自动化。只有你才确切的知道测试什么，以及如何去测试。 本节讲侧重于 Test::Nginx 提供的基本原语，基于此你可以设计出巧妙和有效的测试用例。

=== 准备 NGINX 的配置

在一个测试块里面，我们可以用不同的数据节来指定我们自定义的代码片段， 放在由 Test::Nginx 生成的最终的 nginx.conf 配置文件的不同位置中。

最常见的是 config 节，它用来在默认测试服务器的 server {} 配置块中插入自定义代码片段。 我们也可以在 nginx.conf 的 http {} 配置块中，用 http_config 节来插入自定义的内容。 main_config 节可以在 NGINX 配置的最上层范围中插入自定义内容。 一起看看下面这个例子。

[source,test-base]

=== TEST 1: — main_config env MY_ENVIRONMENT;

— http_config init_worker_by_lua_block { print(“init”) }

— config location = /t { echo ok; }

— request GET /t — response_body ok —-

这个测试块会生成一个有如下基本结构的 nginx.conf 文件：

[source,nginx]

… env MY_ENVIRONMENT;

http { …

```
init_worker_by_lua_block {
    print("init")
}

server {
    ...

    location = /t {
        echo ok;
    }
} } ----
```

请注意 main_config， http_config，和 config 这几个数据节的值，是如何映射到 NGINX 配置文件的不同位置的。

有疑问的时候，我们总是可以来检查这个测试框架生成的实际 nginx.conf 文件， 它位于当前工作目录（通常是当前项目的根目录）的 t/servroot/conf/nginx.conf 位置中。

Test::Nginx 会为每一个测试块生成一个新的 nginx.conf 文件，这会让每个测试块独立存在成为可能。 测试框架默认会在运行每一个测试块之前，自动启动一个新的 NGINX 服务，然后在这个测试块运行结束后立即关闭服务。 好在 NGINX 是一个轻量的服务器，通常启动和关闭都非常快。所以，测试块运行起来并没有看上去那么慢。

=== 准备发起请求

准备一个请求，最简单的是用 request 数据节, 比如

[source,test-base]

— request GET /t?a=1&b=2 —-

默认使用的是 HTTP/1.1 协议。如果你愿意，可以指明来使用 HTTP/1.0 协议:

[source,test-base]

— request GET /t?a=1&b=2 HTTP/1.0 —-

request 小节的值中，前导空格或者空白行都会被自动丢弃。你甚至可以在最前面加一个 # 字符来增加注释，比如

[source,test-base]

— request

```
# this is a simple test:
GET /t ----
```

你可以通过如下 more_headers 小节来同时增加一些额外的请求头部信息。

[source,test-base]

— request GET /t — more_headers Foo: bar Bar: baz —-

==== Pipelined Requests

Preparing pipelined HTTP requests are also possible. But you need to use the pipelined_requests section instead of request . For instance,

[source,test-base]

=== TEST 1: pipelined requests — config location = /t { echo ok; }

— pipelined_requests eval [“GET /t”, “GET /t”]

— request_body eval [“ok\n”, “ok\n”] —-

It is worth noting that we use the eval filter with the pipelined_requests section to treat the literal value of that section as Perl code. This way we can construct a Perl array of the request strings, which is the expected data format for the pipelined_requests section. Similarly we need a similar trick for the response_body section when checking outputs. With an array of expected response body data, we can expect and check different values for different individual request in the pipeline. Note, however, not every data section supports the same array-typed value semantics as response_body .

=== Checking Responses

We have already visited the response_body and error_code data sections for checking the response body data and response status code, respectively.

The response_body data section always performs an exact whole-string comparison between the section value and the actual response body. It tries to be clever when long string value comparison fails. Consider the following sample output from prove .

…. t/foo.t .. 1/? # Failed test ‘TEST 1: long string test - response_body - response is expected (req 0)’ # at …/test-nginx/lib/Test/Nginx/Socket.pm line 1282. # got: …“IT 2.x is enabled.\x{0a}\x{0a}”… # length: 409 # expected: …“IT 2.x is not enabled.\x{0a}”… # length: 412 # strings begin to differ at char 400 (line 1 column 400) # Looks like you failed 1 test of 2. /tmp/foo.t .. Dubious, test returned 1 (wstat 256, 0x100) Failed 1/2 subtests

Test Summary Report

/tmp/foo.t (Wstat: 256 Tests: 2 Failed: 1) Failed test: 2 Non-zero exit status: 1 Files=1, Tests=2, 0 wallclock secs (0.01 usr 0.00 sys + 0.09 cusr 0.03 csys = 0.13 CPU) Result: FAIL ….

From this test report, we can clearly see that

. it is the test block with the title TEST 1: long string test that is failing, . it is the response_body data section check that fails, . the actual response body data is 409 bytes long while the expected value is 412 bytes, and . the expected value has an additional not word in the string fragment IT 2.x is enabled and the difference starts at the offset 400 in the long string.

Behind the scene, Test::Nginx uses the Perl module link:https://metacpan.org/pod/Test::LongString[Test::LongString] to do the long string comparisons. It is also particularly useful while checking response body data in binary formats.

If your response body data is in a multi-line textual format, then you may also want to use a diff -style output when the data does not match. To achieve this, we can call the no_long_string() Perl function before the run_tests() function call in the prologue part of the test file. Below is such an example.

[source,test-base]

use Test::Nginx::Socket ‘no_plan’;

no_long_string();

run_tests();

DATA

=== TEST 1: — config location = /t { echo “Life is short.”; echo “Moon is bright.”; echo “Sun is shining.”; } — request GET /t — response_body Life is short. Moon is deem. Sun is shining. —-

Note the no_long_string() call in the prologue part. It is important to place it before the run_tests() call otherwise it would be too late for it to take effect, obviously.

Invoking the prove utility (or any shell wrappers for it) to run this test file gives the following details about the test failure:

…. # Failed test ‘TEST 1: - response_body - response is expected (req 0)’ # at …/test-nginx/lib/Test/Nginx/Socket.pm line 1277. # @@ -1,3 +1,3 @@ # Life is short. # -Moon is deem. # +Moon is bright. # Sun is shining. # Looks like you failed 1 test of 2. ….

It is obvious that the second line of the response body output is different.

You can even further disable the diff -style comparison mode by adding a no_diff() Perl function call in the prologue part. Then the failure report will look like this:

…. # Failed test ‘TEST 1: - response_body - response is expected (req 0)’ # at …/test-nginx/lib/Test/Nginx/Socket.pm line 1277. # got: ‘Life is short. # Moon is bright. # Sun is shining. # ‘ # expected: ‘Life is short. # Moon is deem. # Sun is shining. # ‘ # Looks like you failed 1 test of 2. ….

That is, Test::Nginx just gives full listing of the actual response body data and the expected one without any abbreviations or hand-holding.

==== Pattern Matching on Response Bodies

When the request body may change in some ways or you just care about certain key words in a long data string, you can specify a Perl regular expression to do a pattern match against the actual request body data. This is achieved by the response_body_like data section. For example,

[source,test-base]

— response_body_like: age: \d+ —-

Be careful when you are using the multi-line data section value form. A trailing newline character appended to your section value may make your pattern never match. In this case the chomp filter we introduced in an early section can be very helpful here. For example,

[source,test-base]

— response_body_like chomp age: \d+ —-

You can also use the eval filter to construct a Perl regular expression object with a Perl expression, as in

[source,test-base]

— response_body_like eval qr/age: \d+/ —-

This is the most flexible form to specify a pattern.

NOTE: Perl uses the qr quoting structure to explicitly construct regular expression objects. You can use various different quoting forms like qr/.../ , qr!...! , qr#...# , and qr{...} .

==== Checking Response Headers

The response_headers data section can be used to validate response header entries. For example,

[source,test-base]

— response_headers Foo: bar Bar: baz !Blah —-

This section dictates 3 tests actually:

. The response header Foo must appear and must take the value bar ; . The response header Bar must appear and must take the value baz ; and . The response header Blah must not appear or take an empty value.

=== Checking NGINX Error Logs

In addition to responses, the NGINX error log file is also an important output channel for an NGINX server setup.

==== True-False Tests

One immediate testing requirement is to check whether or not a piece of text appears in any error log messages. Such checks can be done via the data sections error_log and no_error_log , respectively. The former ensures that some lines in the error log file contain the string specified as the section value while the latter tests the opposite: ensuring that no line contains the pattern.

For example,

[source,test-base]

— error_log Hello world from my server —-

Then the string `Hello world from my server` (without the trailing new-line) must appear in at least one line of the NGINX error log. You can specify multiple strings in separate lines of the section value to perform different checks, for instance,

```
[source,test-base]

— error_log This is a dog! Is it a cat? —-
```

Then it performs two error log checks, one is to ensure that the string `This is a dog!` appears in some error log lines. The order of these two string patterns do not matter at all.

If one of the string pattern failed to match any lines in the error log file, then we would get a test failure report from `prove` like below.

…. # Failed test ‘TEST 1: simple test - pattern “This is a dog!” matches a line in error.log (req 0)’ ….

If you want to specify a Perl regular expression (regex) as one of the patterns, then you should use the `eval` section filter to construct a Perl-array as the section value, as in

```
[source,test-base]

— error_log eval [ “This is a dog!”, qr/\w+ is a cat\?/, ] —-
```

As we have seen earlier, Perl regexes can be constructed via the `qr/.../` quoting syntax. Perl string patterns in the Perl array specified by double quotes or single quotes are still treated as plain string patterns, as usual. If the array contains only one regex pattern, then you can omit the array itself, as in

```
[source,test-base]

— error_log eval qr/\w+ is a cat\?/ —-
```

`Test::Nginx` puts the error log file of the test NGINX server in the file path `t/servroot/logs/error.log` . As a test writer, we frequently check out this file directly when things go wrong. For example, it is common to make mistakes or typos in the patterns we specify for the `error_log` section. Also, scanning the raw log file can give us insight about the details of the NGINX internal working when the NGINX debugging logs are enabled in the NGINX build.

The `no_error_log` section is very similar to `error_log` but it checks the nonexistence of the string patterns in the NGINX error log file. One of the most frequent uses of the `no_error_log` section is to ensure that there is *no* error level messages in the log file.

[source,test-base]

— no_error_log [error] —-

If, however, there is a line in the nginx error log file that contains the string [error] , then the test fails. Below is such an example.

…. # Failed test ‘TEST 1: simple test - pattern “[error]” should not match any line in error.log but matches line “2016/02/01 11:59:50 [error] 1788#0: *1 lua entry thread aborted: runtime error: content_by_lua(nginx.conf:42):2: bad”’ ….

This is a great way to find the details of the error quickly by just looking at the test report.

Like error_log , this section also supports Perl array values and Perl regex values though the eval filter.

==== Grep Tests

The error_log and no_error_log sections are very handy in quickly checking the appearance of contain patterns in the NGINX error log file. But they have serious limitations in that it is impossible to impose stronger constraints on the relative order of the messages containing the patterns nor on the number of their occurrences.

To address such limitations, Test::Nginx::Socket provides an alternative way to check NGINX error logs in a way similar to the famous UNIX tool, grep . The sections grep_error_log and grep_error_log_out are used for this purpose. The test writer uses the grep_error_log section to specify a pattern, with which the test framework scans through the NGINX error log file and collect all the matched parts of the log file lines along the way, forming a final result. This aggregated log data result is then matched against the expected value specified as the value of the grep_error_log_out section, in a similar way as with the response_body section discussed above.

It is easiest to explain with a simple example.

[source,test-base]

=== TEST 1: simple grep test for error logs — config location = /t { content_by_lua_block { print(“it is matched!”) print(“it is matched!”) print(“it is matched!”) } } — request GET /t — grep_error_log: it is matched! — grep_error_log_out it is matched! it is matched! it is matched! —-

Here we use the Lua function print() provided by the link:https://github.com/openresty/lua-nginx-module#readme[ngx_http_lua] module to generate NGINX error log messages at the notice level. This test case tests the number of the log messages containing the string it is matched! . It is important to note that only the *matched* part of the log file lines are collected in the final result instead of the whole log lines. This simplifies the comparison a lot since NGINX error log messages can contain varying details like timestamps and connection numbers.

A more useful form of this test is to specify a Perl regex pattern in the grep_error_log section. Consider the following example.

[source,test-base]

=== TEST 1: simple grep test for error logs — config location = /t { content_by_lua_block { print("test: before sleeping…") ngx.sleep(0.001) – sleeping for 1ms print("test: after sleeping…") } } — request GET /t — grep_error_log eval: qr/test: .*?.../ — grep_error_log_out test: before sleeping… test: after sleeping… —-

We specify a Perl regex pattern, test: .*?\.\.\. , here to filter out all the error log messages starting with test: and ending with ... . And naturally in this test we also require the relative order of these two messages, that is, before sleeping must appear *before* after sleeping . Otherwise, we shall see failure reports like below:

…. # Failed test 'TEST 1: simple grep test for error logs - grep_error_log_out (req 0)' # at …./lib/Test/Nginx/Socket.pm line 1048. # got: "test: after sleeping…\x{0a}test: before sleeping…\x{0a}" # length: 49 # expected: "test: before sleeping…\x{0a}test: after sleeping…\x{0a}" # length: 49 # strings begin to differ at char 7 (line 1 column 7) ….

As with the response_body section, we can also call the no_long_string() Perl function before run_tests() in the test file prologue, so as to disable the long string output mode and enable the diff mode. Then the test failure would look like this:

…. # Failed test 'TEST 1: simple grep test for error logs - grep_error_log_out (req 0)' # at …/lib/Test/Nginx/Socket.pm line 1044. # @@ -1,2 +1,2 @@ # -test: before sleeping… # test: after sleeping… # +test: before sleeping… ….

Obviously, for this test case, the diff format looks better.

==== Extra Delay Before Log Checks

By default, Test::Nginx::Socket performs the NGINX error log checks not long after it receives the complete HTTP response for the test request. Sometimes, when the log messages are generated by the server after sending out the response, the error log checks may be carried out too early that the messages are not yet written into the log file. In this case, we can specify an extra delay via the wait data section for the test scaffold to wait for the error log messages. Here is an example:

[source,test-base]

=== TEST 1: wait for the timer — config location = /t { content_by_lua_block { local function f(premature) print(“HERE!”) end assert(ngx.timer.at(0.1, f)) } } — request GET /t — error_log HERE! — no_error_log [error] — wait: 0.12 —-

Here we create a timer via the `ngx.timer.at` Lua function, which expires after 0.1 seconds. Due to the asynchronous nature of timers, the request handler does not wait for the timer to expire and immediately finishes processing the current request and sends out a response with an empty body. To check for the log message `HERE!` generated by the timer handler `f`, we have to specify an extra delay for the test scaffold to wait. The 0.12 seconds time is specified in this example but any values larger than 0.1 would suffice. Without the `wait` section, this test case would fail with the following output:

…. # Failed test ‘TEST 1: wait for the timer - pattern “HERE!” matches a line in error.log (req 0)’ ….

Obviously the test scaffold checks the error log too soon, even before the timer handler runs.

=== Section Review

`Test::Nginx::Socket` offers a rich set of data sections for specifying various different input data and expected output data, ranging from NGINX configuration file snippets, test requests, to expected responses and error log messages. We have already demonstrated the power of data driven testing and declarative test case crafting. We want to achieve multiple goals at the same time, that is, not only to make the tests self-contained and highly readable, but also to make the test report easy to interpret and analyze when some of the tests fail. Raw files automatically generated by the test scaffold, like `t/servroot/conf/nginx.conf` and `t/servroot/logs/error.log`, should be checked frequently when manually debugging the test cases. The next section extends the discussion of this section with a focus on testing erroneous cases.

测试错误用例

Summary: 在可靠软件的开发过程中，错误处理占用了大部分时间。程序员们需要在设计测试时关注各种边界条件和错误场景，力图最大化测试覆盖。

== 测试错误用例

在可靠软件的开发过程中，错误处理占用了大部分时间。程序员们需要在设计测试时关注各种边界条件和错误场景，力图最大化测试覆盖。

上一节介绍了 Test::Nginx::Socket 中的数据节（比如用于检查 NGINX 错误日志的 error_log ）。 它们是测试程序正确性的得力助手。 有时我们还需要测试更加极端的场景，比如服务器启动错误、格式错误的响应、错误请求，还有形形色色的超时错误。

=== 期望服务器启动错误

有些场景下，我们期望服务器在启动时出错而非继续运行，比如使用了错误的配置指令，抑或没能满足初始化阶段的一些硬性要求。 如果我们想要测试这种情况，特别是要检查错误日志中是否有某个特定的错误信息，可以使用 must_die 数据节，告知测试脚手架我们 *期望* NGINX 在这次测试中启动失败。

下面的例子测试在 ngx_http_lua 模块的 init_by_lua_block 上下文抛出 Lua 异常的情况。

[source,test-base]

=== TEST 1: dying in init_by_lua_block — http_config init_by_lua_block { error("I am dying!") } — config — must_die — error_log I am dying! —-

init_by_lua_block 中的 Lua 代码会在 NGINX 主进程加载 NGINX 配置文件时运行。 在此抛出 Lua 异常会立刻终止 NGINX 的启动过程。 must_die 告诉测试脚手架，仅当 NGINX 启动失败，测试才算通过。 而 error_log 确保服务器确实是因为抛出了"I am dying!"异常而退出的。

如果我们从上面的测试块移除 --- must_die ，那么该测试甚至不能成功运行：

…. t/a.t .. nginx: [error] init_by_lua error: init_by_lua:2: I am dying! stack traceback: [C]: in function 'error' init_by_lua:2: in main chunk Bailout called. Further testing stopped: TEST 1: dying in init_by_lua_block - Cannot start nginx using command "nginx -p …/t/servroot/ -c …/t/servroot/conf/nginx.conf > /dev/null". ….

默认情况下测试脚手架会把 NGINX 服务器启动失败当作测试中的致命错误。 然而 must_die 的存在，会把它变成一项常规检查。

=== 期望格式错误的响应

HTTP 响应理论上应是格式良好的。不幸的是，梦想很丰满，现实很骨感。 有时候出于某些意外， HTTP 响应会被截断，类似这样导致响应格式错误的情况还有很多。 作为设计错误用例的人，我们总是想要正常地测试像这样的异常情况。

通常来说， Test::Nginx::Socket 默认会检查从测试服务器收到的响应的完整性， 把来自 NGINX 的格式错误的响应当作一种错误。 对于期望是格式错误或者被截断的响应，我们需要通过 ignore_response 告知测试脚手架不去检查响应的格式。

考虑下面这个例子，服务器在发送了响应体的开头部分后立刻关闭了下游的连接。

[source,test-base]

=== TEST 1: aborting response body stream — config location = /t { content_by_lua_block { ngx.print(“hello”) ngx.flush(true) ngx.exit(444) } } — request GET /t — ignore_response — no_error_log [error] —-

content_by_lua_block 语句块中的 ngx.flush(true) 确保 NGINX 中缓冲的所有响应数据确实被刷入系统的发送缓冲区了。 对于运行在本地的客户端而言，这就意味着收到了响应数据。 紧接着， ngx.exit(444) 关闭了当前下游的连接，对于 HTTP 1.1 的分块传输编码（chunked encoding），这将导致响应体被截断。 不要忽略 --- ignore_response 这一行，它告诉测试脚手架不要在意响应的完整性。 如果缺了这一行，我们会在运行 prove 时看到这样的错误：

…. # Failed test ‘TEST 1: aborting response body stream - no last chunk found - 5 # hello # ‘ ….

显然，测试脚手架会抱怨，说缺乏标记分块传输编码数据流结束的“last chunk”。 由于在发送响应体数据的中途断开了连接，服务器没法发送分块传输编码所需的完整响应体。

=== 测试超时错误

超时错误是现实生活中最常见的网络问题之一。 导致超时的原因有很多，比如线路上的丢包，抑或接收端的连接问题，以及耗时操作阻塞了事件循环等等。 大多数应用想要确保超时保护机制能正常工作，避免长时间地等待下去。

在一个自包含的测试框架中，测试和模拟超时异常通常要有特别的技巧。 毕竟，测试时所使用的网络通信仅经过本地的回环网络设备，在延迟和带宽上不会有什么问题。 接下来我们会看到，为了稳定地模拟五花八门的超时错误，测试套件中使用的各种技巧。

==== 连接超时

模拟 TCP 连接中的超时是最简单的。 仅需通过防火墙配置或其他方法，准备一个会丢弃所有接收到的 SYN 包的远程地址，然后跟它建立连接。 我们在 agentzh.org 域名的 12345 端口提供了这样的“黑洞”服务。如果你的测试运行环境允许连外网，那么你就可以用上它。 考虑下下面的测试用例：

[source,test-base]

=== TEST 1: connect timeout — config resolver 8.8.8.8; resolver_timeout 1s;

```
location = /t {
    content_by_lua_block {
        local sock = ngx.socket.tcp()
        sock:settimeout(100) -- ms
        local ok, err = sock:connect("agentzh.org", 12345)
        if not ok then
            ngx.log(ngx.ERR, "failed to connect: ", err)
            return ngx.exit(500)
        end
        ngx.say("ok")
    }
} --- request GET /t --- response_body_like: 500 Internal Server Error --- error_code: 50
0 --- error_log failed to connect: timeout ----
```

这里我们配置了 resolver ，因为需要在请求时通过 Lua 代码解析域名 agentzh.org 。 我们使用 error_log 检查 NGINX 错误日志中是否含有 cosocket 对象的 connect() 方法返回的错误信息。

注意在测试用例中设置的超时限制要相对短一些，这样就不会花过多的时间在等待测试结束。 毕竟测试需要被反复运行。运行测试的次数越多，我们从自动化测试获取的收益就越多。

值得说明的是，测试脚手架的 HTTP 客户端也有一个超时限制，默认 3 秒。 如果你的请求耗时超过 3 秒，你会在测试报告中看到这个错误信息：

…. ERROR: client socket timed out - TEST 1: connect timeout ….

如果注释了示例中的 settimeout ，依赖 cosocket 默认 60 秒的超时限制， 我们就会收到这个信息。

通过设置 timeout 数据节的值，我们可以改变测试脚手架客户端默认的超时限制，像这样：

[source,test-base]

— timeout: 10 —-

现在超时限制是 10 秒而不是之前的 3 秒。

==== 读取超时

模拟读取超时也简单。仅需从一个既不写入数据又不断开连接的对端读取内容。 考虑下面的例子：

[source,test-base]

=== TEST 1: read timeout — main_config stream { server { listen 5678; content_by_lua_block { ngx.sleep(10) – 10 sec } } } — config lua_socket_log_errors off; location = /t { content_by_lua_block { local sock = ngx.socket.tcp() sock:settimeout(100) – ms assert(sock:connect("127.0.0.1", 5678)) ngx.say("connected.") local data, err = sock:receive() – try to read a line if not data then ngx.say("failed to receive: ", err) else ngx.say("received: ", data) end } } — request GET /t — response_body connected. failed to receive: timeout — no_error_log [error] —-

这里我们使用 main_config 定义了一个 TCP 服务器，监听在本地的 5678 端口。 这个服务器建立 TCP 连接后倒头就睡，10 秒后才起来关闭连接。 注意我们在 stream {} 配置块中用的是 link:https://github.com/openresty/stream-lua-nginx-module#readme[ngx_stream_lua] 模块。 location = /t 语句块是这个测试块的核心，它连接了前面定义的服务器并试图从中读一行数据。 显然，100ms 的超时限制会先生效，这下我们可以测试到读取超时的错误处理了。

==== 发送超时

触发发送超时比起连接超时和读取超时要难多了。问题在于写套接字的异步特性。

出于性能考虑，在写的过程中至少有两层缓冲区：

. 在 NGINX 核心的用户态发送缓冲区，和 . 操作系统内核 TCP/IP 栈的套接字发送缓冲区

雪上加霜的是，在写的对端还至少存在一层系统层面上的接收缓冲区。

要想触发一次发送超时，最简单粗暴的方法是塞爆所有发送的缓冲区，且确保对端在应用层面上不做任何读取操作。 所以，仅用少量的测试数据就想在日常环境下重现和模拟发送超时，无异于痴人说梦。

好在，存在一个用户态小技巧可以拦截 libc 包装的套接字 I/O 调用，并基于此实现曾经难于上青天的目标。 我们的 link:https://github.com/openresty/mockeagain[mockeagain] 库实现了这个技巧， 支持在用户指定的输出数据位置触发一次发送超时。

下面的例子恰好在响应体发送了"hello world"之后触发一次发送超时。

[source,test-base]

=== TEST 1: send timeout — config send_timeout 100ms; postpone_output 1;

```
location = /t {
    content_by_lua_block {
        ngx.say("hi bob!")
        local ok, err = ngx.flush(true)
        if not ok then
            ngx.log(ngx.ERR, "flush #1 failed: ", err)
            return
        end

        ngx.say("hello, world!")
        local ok, err = ngx.flush(true)
        if not ok then
            ngx.log(ngx.ERR, "flush #2 failed: ", err)
            return
        end
    }
} --- request GET /t --- ignore_response --- error_log flush #2 failed: timeout --- no_erro
r_log flush #1 failed ----
```

注意用于设置 NGINX 下游写操作的超时限制的 send_timeout 指令。 这里我们使用较小的限制， 100ms ，来确保测试用例尽量快地完成并避免触发测试脚手架客户端默认的 3 秒超时。 postpone_output 1 指令关掉 NGINX 的“postpone output buffer”，让输出数据不会被缓冲起来。 最后，Lua 代码中的 ngx.flush() 确保 *没有* 一个输出过滤器会截留我们的数据。

在运行这个测试用例前，我们必须在 bash 中设置下面的系统环境变量：

[source,bash]

export LD_PRELOAD=”mockeagain.so” export MOCKEAGAIN=”w” export MOCKEAGAIN_WRITE_TIMEOUT_PATTERN=’hello, world’ export TEST_NGINX_EVENT_TYPE=’poll’ —-

让我们一一审视：

. LD_PRELOAD="mockeagain.so" 预先加载 mockeagain 库到当前进程中，当然也包括了测试脚手架启动的 NGINX 服务进程。 如果 mockeagain.so 不在系统库路径中，你可能需要设置 LD_LIBRARY_PATH 来包含它所在的路径。 . MOCKEAGAIN="w" 允许

mockeagain 库拦截并改写非阻塞套接字上的写操作。 . MOCKEAGAIN_WRITE_TIMEOUT_PATTERN='hello, world' 让 mockeagain 在看到给定的字符串 hello, world 出现在输出数据流之后截止数据的发送。 . TEST_NGINX_EVENT_TYPE='poll' 令 NGINX 服务器使用 poll 事件 API 而不是系统默认的 （比如 Linux 上的 epoll ）。因为 mockeagain 暂时只支持 poll 事件。 本质上，这个环境变量只是让测试脚手架生成下面的 nginx.conf 片段。 + [source,nginx] —– events { use poll; } —– + 不过，你需要确保你的 NGINX 或 OpenResty 编译的时候添加了 poll 支持。 总而言之，需要在编译时向 ./configure 指定选项 --with-poll_module 。

现在你应该能让上面的测试通过了！

如果可以的话，我们应该直接在测试文件里设置这些环境变量。 因为一旦缺了它们，这个测试用例就没法通过了。 我们需要在测试文件序言部分一开头（甚至要在 use 语句之前）就添加下面的 Perl 代码片段：

[source,Perl]

BEGIN { $ENV{LD_PRELOAD} = “mockeagain.so”; $ENV{MOCKEAGAIN} = “w”; $ENV{MOCKEAGAIN_WRITE_TIMEOUT_PATTERN} = ‘hello, world’; $ENV{TEST_NGINX_EVENT_TYPE} = ‘poll’; } —-

这里需要使用 BEGIN {} ，因为它会在 Perl 加载任何模块之前运行。 这样当 Test::Nginx::Socket 加载时，设置的环境变量就能生效。

在测试文件中硬编码 mockeagain.so 的路径是个糟糕的主意，毕竟其他测试环境下的 mockeagain 可能位于不同的文件路径。 最好还是让运行测试的人在外面配置包含它的 LD_LIBRARY_PATH 环境变量。

===== 错误排除

如果你在运行上面的测试用例时遇到如下错误，

…. ERROR: ld.so: object ‘mockeagain.so’ from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored. ….

那么你需要检查下 mockeagain.so 所在的路径是否位于 LD_LIBRARY_PATH 环境变量中。 举个例子，我在自己的系统上是这么做的

…. export LD_LIBRARY_PATH=$HOME/git/mockeagain:$LD_LIBRARY_PATH ….

如果你看到的是类似于下面的错误信息，

…. nginx: [emerg] invalid event type “poll” in …/t/servroot/conf/nginx.conf:76 ….

意味着你的 NGINX 或 OpenResty 编译的时候没有添加 poll 模块。 你需要重新编译 NGINX 或 OpenResty，并在编译时传递 --with-poll_module 选项给 ./configure 。

在接下来的 Test Modes 一节，我们还会继续讨论到 mockeagain 。

=== 模拟后端的异常响应

在前面的“读取超时”小节，我们在例子里使用 link:https://github.com/openresty/stream-lua-nginx-module#readme[ngx_stream_lua] 模块模拟了一个仅接受新连接却从不返回数据的后端 TCP 服务器。 毫无疑问，我们还可以在这个模拟服务器中做更多有趣的东西，比如模拟后端服务器返回各种错误响应数据。

举个例子，如果用真实的服务器测试一个 Memcached 客户端，就很难去模拟错误的抑或格式异常的响应。 而用模拟的服务器则易如反掌：

[source,test-base]

=== TEST 1: get() results in an error response — main_config stream { server { listen 1921; content_by_lua_block { ngx.print(“SERVER_ERROR\r\n”) } } } — config location /t { content_by_lua_block { local memcached = require “resty.memcached” local memc = memcached:new()

```
        assert(memc:connect("127.0.0.1", 1921))

        local res, flags, err = memc:get("dog")
        if not res then
            ngx.say("failed to get: ", err)
            return
        end

        ngx.say("get: ", res)
        memc:close()
    }
} --- request GET /t --- response_body failed to get: SERVER_ERROR --- no_error_log [er
ror] ----
```

我们可以随心所欲地仿造 Memcached 服务器的任意响应。太棒了！

NOTE: Test::Nginx::Socket 提供了 tcp_listen 、 tcp_query 、 tcp_reply 等 数据节，在测试脚手架层面上支持模拟 TCP 服务器。如果你不想在你的测试代码中使用 ngx_stream_lua 抑或 NGINX 流子系统，可以用它们代替。事实上，在 ngx_stream_lua 诞生之前， 我们一直依赖于 Test::Nginx::Socket 内置的 TCP 服务器来完成相关测试。同样地， Test::Nginx::Socket 通过 udp_listen 、 udp_query 、 udp_reply 等数据节， 内置了 UDP 服务器支持。你能够在 Test::Nginx::Socket link:https://metacpan.org/pod/Test::Nginx::Socket[官方文档]中读到更详细的说明。

=== 模拟异常客户端

Test::Nginx::Socket 测试框架提供了特定的数据节来辅助模拟异常的 HTTP 客户端。

==== 仿造异常请求

raw_request 数据节可以用来指定测试时发送的请求。它通常跟 eval 节过滤器成双成对， 以便于编码像 \r 这样的特殊字符。看看下面的例子。

[source,test-nginx]

=== TEST 1: missing the Host request header — config location = /t { return 200; } — raw_request eval “GET /t HTTP/1.1\r Connection: close\r \r “ — response_body_like: 400 Bad Request — error_code: 400 —-

这里我们简单地构造出一个没有 Host 头部的畸形请求。 不出所料， NGINX 返回了 400 响应。

与之相对的，我们一直以来使用的 request 数据节会确保发送给测试服务器的请求格式是正确的。

==== 模拟客户端连接中断

客户端连接中断在网络世界里是个令人着迷的现象。有些时候我们希望即使客户端断开了连接， 服务器也能够继续当前流程；另外一些时候，我们仅仅立刻结束整个请求的处理。 无论如何，我们都需要能够在单元测试用例中可靠地模拟客户端连接中断的方法。

之前讲过，测试脚手架客户端的默认超时行为，可以通过 timeout 数据节进行调整。 借助它的功能，我们也能让客户端提前断开连接。所需的只是设置过小的超时时间。 为了避免测试脚手架报客户端超时的错误，还要指定 abort 数据节告知测试脚手架这一点。 让我们用一个简单的测试用例把上面的内容串起来。

[source,test-nginx]

=== TEST 1: abort processing in the Lua callback on client aborts — config location = /t { lua_check_client_abort on;

```
    content_by_lua_block {
        local ok, err = ngx.on_abort(function ()
            ngx.log(ngx.NOTICE, "on abort handler called!")
            ngx.exit(444)
        end)

        if not ok then
            error("cannot set on_abort: " .. err)
        end

        ngx.sleep(0.7)  -- sec
        ngx.log(ngx.NOTICE, "main handler done")
    }
} --- request
GET /t --- timeout: 0.2 --- abort --- ignore_response --- no_error_log [error] main handle
r done --- error_log client prematurely closed connection on abort handler called! ----
```

在这个例子里，借助 timeout 数据节，我们让测试脚手架客户端在 0.2 秒后断开连接。 同样，为了避免测试脚手架报客户端超时错误，我们指定了 abort 数据节。 最后，在 Lua 应用代码里，我们启用了 lua_check_client_abort 指令来检查客户端超时， 并通过 ngx.on_abort API 注册了一个回调函数，以 ngx.exit(444) 终止服务端处理流程。

==== 客户端永不关闭连接

不像现实生活中大多数举止得当的 HTTP 客户端， Test::Nginx::Socket 使用的客户端*永不* 主动关闭连接，除非发生了超时错误（超过了 --- timeout 数据节指定的时间）。 这确保收到“Connection: close”请求头部后，NGINX 服务器总能够正确地关闭连接。

当连接没有被关闭时，服务器会存在“连接泄漏”的问题。举个例子，NGINX 在它的 HTTP 子系统中 使用引用计数（ r->main->count ）来判断一个连接能否被关闭和释放。如果引用计数出了差错， NGINX 可能永远不会结束请求，造成资源泄漏。在这种情况下，对应的测试用例会因客户端超时错误而失败。 举个例子，

[source]

Failed test ‘ERROR: client socket timed out - TEST 1: foo # ‘ —-

就这方面来说， Test::Nginx::Socket 不是个遵纪守法的 HTTP 客户端。 事实上，我们的测试脚手架避免使用一个循规蹈矩的 HTTP 客户端。 大多数测试用例都关注于罕见的错误场景，而一个循规蹈矩的客户端会帮忙掩盖这些问题，而非揭露它们。