iRules 编程手册

前言

F5 iRules 是 BIG-IP 流量管理系统中一项强大而灵活的功能，可用于管理网络流量。为什么我要重新整理这篇手册?在日常接触的很多工作中，我遇到过各种各样的系统问题，例如: 网络设备、安全设备、负载设备、应用服务器.....它们都在处理协议和数据。虽然他们的功能模块和设计有所区别，但是其工作的本质并没有区别--传递信息，如果对其工作的本质有更多的理解，那么处理和分析问题时会变得轻松一些。而 iRules 可以细粒度的控制传输的协议报文和数据报文，这要比起在控制页面点一点有趣的多，而这也是很多设备不提供或者说对于普通用户不开放的功能。看之前需要把附录文档大致都了解下。

道阻且长、行则将至

第一节、入门部分

第一章、Tcl 编程语言简介

本章主要对Tcl 语言和iRules 进行简要描述。

1.1 什么是Tcl ?

Tcl（发音为“tickle”或缩写）是一种高级、通用、解释型、动态编程语言。它的设计目标非常简单且功能强大。 Tcl 将所有内容都转换为命令的模型，甚至包括变量分配和过程定义等编程结构。 Tcl 支持多种编程范例，包括面向对象、命令式和函数式编程或过程风格。

它通常嵌入到 C 应用程序中，用于快速原型制作、脚本应用程序、GUI 和测试。 Tcl 解释器可用于许多操作系统，允许 Tcl 代码在各种系统上运行。因为 Tcl 是一种非常紧凑的语言，所以它以其完整形式和其他几个占用空间小的版本用于嵌入式系统平台。

Tcl 的功能包括**:**

- 所有操作都是命令，包括语言结构。它们以前缀表示法编写。
- 命令通常接受可变数量的参数（可变参数）。
- 一切都可以动态地重新定义和覆盖。实际上，没有关键字，甚至可以添加或更改控制结构，尽管这是不可取的。
- 所有数据类型都可以作为字符串进行操作，包括源代码。在内部，变量具有整数和双精度等类型，但转换是完全自动的。
- 变量不是声明的，而是赋值的。使用未定义的变量会导致错误。
- 支持完全动态的、基于类的对象系统 TclOO，包括元类、过滤器和混合等高级功能。
- 支持套接字和文件的事件驱动接口，基于时间和用户定义的事件。
- 默认情况下，变量可见性仅限于词法（静态）范围，但 uplevel 和 upvar 允许 proc 与封闭函数的范围进行交互。
- Tcl 本身定义的所有命令都会在不正确的使用时生成错误消息。
- 可扩展性，支持通过 C、C++、Java、Python 调用。
- 使用字节码的解释型语言。
- 完整的 Unicode（3.1 开始，定期更新）支持，1999 年首次发布。
- 支持正则表达式。
- 跨平台：Windows API； Unix、Linux、Macintosh 等。
- 与窗口 (GUI) 界面 Tk 的紧密、跨平台集成。
- 存在多种版本。

1.2 iRules 为什么选择**Tcl** 解释器？

基于上小结对于Tcl 的描述本节从三个方面进行说明:

- Speed 速度
- Embeddability 可嵌入性
- Usability 可用性

Speed

从设备处理能力来说速度是最重要的指标，当系统在细粒度控制函数处理越快；数据包过滤、重写、路由、转发、IP 转换，每个子功能越快，在这些事情上使用的资源就越少。系统就会有更多的性能来构建和完成复杂的逻辑功能。

1. 就这个目的而言，Tcl 比任何其他广泛可用的选项都要快得多(如 Perl 和 Python)，除了 Lua 之外。
2. 性能问题, F5 TMM 流量处理内核 中执行这些操作的函数在其本机状态下的性能远高于它们在每个连接运行的任何解释语言、Tcl 或其他语言中的性能。因此，如果我们的绝大多数命令实际上只是在 Tcl 和 C 之间来回传递控制，那么一个非常成熟的接口就变得至关重要。 与其他类似的语言选项相比，Tcl 拥有更全面的 C 编程

API 之一。考虑到这种情况发生的频率以及它对 iRules 核心功能的重要性，这是一个很大的优势。

3. Tcl 支持编译为字节码的概念，这比直接解释源码执行要快。大多数脚本语言结合了编译和执行功能，以便两者同时有效地发生。使用 Tcl，能够使用不同的模型，允许编译、语法检查等发生在加载时，这意味着在运行时，字节码被处理而不是原始的 iRule，从而跳过大量会涉及的开销。这允许在运行时（意味着当 iRule 必须执行时）占用更小的空间，以换取在加载时（当用户保存 iRule 时）做一些额外的工作。加载一次就可以在每个需要处理的连接上运行。

Embeddability

cl 不仅速度极快，而且可嵌入性极强。在许多快要求效率的低级系统（例如 L2 交换机）中，它作为首选嵌入式解释器有着悠久的历史。这是因为与提供类似功能（或更多功能，如 Perl 和 Java，但稍后会详细介绍）的其他语言相比，Tcl 非常非常小。此外，Tcl 与 C 的集成非常简单。以至于在许多情况下它被认为几乎是免费的，并且任何用 C 编写的东西都可以轻松地通过 Tcl 开放，只需付出极小的努力。请记住，当我谈论用 C 编写的东西时，该列表包括大量程序和系统，包括许多现代内核，例如 Windows 和 Linux。在考虑将 Tcl 与自定义微内核集成时，Tcl 对内核友好并不是一件坏事，就像 TMM 中的情况一样。

除了 Tcl 的高度可嵌入特性之外，您还必须考虑占用空间。整个 Tcl 有几百 KB，包括我们没有在 iRules 中使用的部分。与功能更丰富的表兄弟 Perl 和 Java 以及其它解释器相比，这是微不足道的。例如，Tcl 的整个源代码下载（截至撰写本文时）为 4.3M，而 Perl 为 15M。您越了解 iRules 世界的内部运作，环境的大小就变得非常重要。

大多数人没有考虑或没有意识到的一件事是，每个调用 iRule 的 BIG-IP 连接都会有一个唯一的 Tcl 上下文以及伴随的状态、变量等。这意味着该内存被分配给每个使用 iRule 的连接来存储该 Tcl 结构，允许它与 TMM 唯一接口，并为该特定连接和与之关联的 iRules 执行它需要的操作。请记住，这可能会在繁忙的高端 F5 设备上同时发生数百万次，对我来说，这变得非常令人印象深刻。 Tcl 的几百 KB 和许多其他语言的几兆字节之间的内存占用差异对于单个实例来说已经足够大了。然而，当您谈论几十万甚至一百万个并发实例时，它会变得越来越大，也越来越重要，正如您所想象的那样。

您当然无法在相同的资源占用空间中分配、存储和处理数百万个 Perl 副本。这直接归因于 Tcl 的大小和简单性。 Perl 和其他类似的语言比 Tcl 具有更多的基本功能。当您需要它们时，并且您不担心在如此要求效率环境中遇到资源限制时，这是一件很棒的事情。在我们的世界中，当绝大多数添加的功能都不需要时，每个字节或周期都很重要，开销是奢侈的。

Usability

Tcl 是一种简单易读的语言，可以快速熟悉和掌握，并且易于阅读。

1.3 正式开始 GO！！！

本节描述一些编程基础知识和概念，一直到 F5 术语和概念、iRules 基础知识和用法、组件等。

词汇表

- Commands
- Variables
- Conditionals
- Operators

Commands

命令的概念非常简单。每种编程语言都有许多不同的命令，可以执行这些命令以获得不同的期望结果。主机名查找、生成子进程、访问 shell 以运行命令、在内存中存储信息、引导流量或任何数量的其他事情都是通过一个命令或从相关代码中调用的一系列命令来实现的。从本质上讲，任何代码中的命令都是代码中实际“做某事”的部分。其余部分用于处理信息、创建逻辑支持和流程等。如果没有实际执行操作的命令，那么在运行任何脚本时都不会产生任何效果，尽管逻辑和代码非常漂亮。

要基本了解如何使用几种不同的语言以编程方式调用某些命令，请参见下文：

Command	print	shell command
Language		
Perl	print "Test Text";	system("testcmd");
PHP	print "Test Text";	system('testcmd');
Python	print ("Test Text");	os.system("testcmd");
Tcl (iRules)	puts "Test Text"	exec testcmd

Variables

变量是任何编程语言的组成部分，因为它们用于在内存中存储信息。通过将信息存储在一个变量中，您可以稍后调用它、操纵该信息、将其传递给给定的命令等。变量对于条件测试、字符串操作等许多事情也是必不可少的。

您可以在下面看到几种不同语言的基本示例以及它们处理简单变量的方式：

Command	set variable to static info	set variable to dynamic info	view variable
Language			
Perl	$var = "test";	$var = system("nslookup test.com");	print $var;
PHP	$var = "test";	$var = system("nslookup test.com");	print $var;
Python	var = "test"	var = os.system("nslookup test.com");	print var
Tcl (iRules)	set var "test"	set var [exec nslookup test.com]	puts $var

Conditionals

条件语句是编程中的一种控制结构，允许您在代码中创建逻辑流。

同样，为了了解基本的条件结构是什么样的，我们来看一些相对标准语言的简单图表：

Command Language	if conditional	switch/case conditional
Perl	if($var == 1) { somecode; }	switch ($var) { case "case1" { somecode; } }
PHP	if($var ==1) { somecode; }	switch ($var) { case "case1": somecode; }
Python	if var == 1: somecode	N/A
Tcl (iRules)	if {$var == 1} { somecode }	switch $var { "case1" { somecode } }

Operators

运算符是一个符号，它告诉编译器执行特定的数学或逻辑操作。 Tcl 语言具有丰富的内置运算符，并提供以下类型的运算符：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 三元运算符

Tcl 特性

在构建 iRules 时，有很多选择适合该目的的语言。选择 Tcl 的原因有很多，但其中最主要的是性能、定制和易用性。虽然在将原始的基本功能与其他语言进行比较时，它可能不是最健壮的语言，但 Tcl 很好地满足了 iRules 的目的。

在考虑基于 iRules 的语言时，性能是一个重要因素。 Tcl 轻巧、紧凑且性能高。在每个“脚本”每秒可能执行数十万次的环境中，性能绝对是最重要的。在运行时之前进行编译的能力是性能之战的巨大胜利。与许多解释型语言不同，Tcl 不必在运行时进行解释，而是可以以极其高效的方式进行预编译和运行，与许多类似语言相比，这允许非常高的处理效率。 Tcl 中高效且易于使用的 C API 也是一大优势，因为 iRules 中内置的大多数命令都是自定义构建的。从 iRules 发出的许多调用都使用此 API。

第二章、技术 & 术语

在编写 iRules 时，编程方面的事情只是工作的一半。这是使 iRules 如此独特的原因之一。它是真正的网络感知编程语言，因此您不仅需要了解编程的基础知识，还需要了解 F5 设备的工作原理、以及协议格式和交互机制。

使用 F5 技术时经常会看到的基本术语/技术的词汇表。以下是供 iRules 程序员使用的 F5 技术词汇表：

```
- Virtual Server
- Pool
- Pool Member
- Node
- Profile
- Client Side
- Server Side
- TMM
- CMP
```

Virtual Server (Virtual IP)

Virtual Server 是进入 BIG-IP LTM处理流量的入口。VS 的定义包含了 IP和端口和 VLAN，其中，IP 可以是一个 IP，也可以是用掩码掩出来的一段 IP，端口可以是一个固定的端口，比如 80，也可以是 0 端口，0 端口的意思就是侦听所有的端口。VS 的定义的含义就是对于发送到 BIG-IP LTM 上的流量，只有同时命中 VS 的 IP 和端口的流量才进行处理。

同时，VS 的定义还有 enable 或者 disable 在那个 VLAN 上。默认情况下，BIG-IP LTM 上定义的 VS 是侦听在所有 VLAN 上的。所以，在默认情况下，如果我们配置了一个 VS_external， IP 地址是在 external 网段上，但如果从 internal 网段发送一个请求到 internal VLAN，这个请 求实际上还是会命中 VS_external 的。

所以，最大范围的 VS 定义可以是 0.0.0.0:0 的 VS enable 在所有的 Vlan 上。而最小范围 的 VS 定义是一个特定的地址侦听一个端口，然后还 enable 在一个特定的 vlan 上。当一个 Vlan 上有多个范围不同的 VS 定义的时候，BIG-IP LTM 将遵循最小命中原则。即先命中定义 范围最小的 VS，然后再命中范围最大的 VS。

Pool

Pool 在 LTM 的内部是一个逻辑概念，是指的一组相同服务的资源的组合。Pool 的作用很简单，就是根据自身定义的分发规则，对 VS 接收进来，并且被 Profile 处理之后的 流量进行分发，分发到 Pool 内的 member 去。

Pool Member

一个应用服务，通常情况下，一个 Member 就是后台服务器的一个侦听进程， 是由 IP:port 格式组成。

Node

Node 通常用来表示后台的一个服务 IP 地址，一个 Node 上面可以有一个或多个 Member。Node 不需要进行配置，在配置了 Member 之后自动产生，系统会根据每一个不同 的 Member IP 地址生成一个 Node 地址。

Profile

当流量进入 BIG-IP LTM 之后，怎样去处理和识别进入 VS 的流量，就需要由 Profile 来定义了。Profile 分了几种类型，有协议层的 Profile 比如 TCP profile, UDP profile。有应用层 的 Profile 比如 HTTP profile，FTP Profile。还有 SSL Profile、会话保持相关的 Profile、认证的Profile 和其他一些 Profile 类型。所有的 Profile 都需要关联在 VS 上才能生效。有些 Profile 之 间是互斥的关系，比如 TCP 和 UDP, HTTP 和 FTP，VS 上关联了 TCP profile，就不能再关联 UDP Profile 了。因为一旦关联了某个 Profile，VS 就会按照这个 Profile 的定义方式去处理流 量。所以有些 Profile 也是相互依存的，比如要关联 HTTP Profile，就必须先关联 TCP Profile。 因为 BIG-IP LTM 认为所有的 HTTP 相关处理，都是在 TCP 层面上的，没有基于 UDP 的 HTTP 请求。除了 TCP、UDP 和 SSL 外，所有类型的 Profile 都只能在一个 VS 上关联一次（认证类 的 Profile 除外）。而 TCP、UDP 和 SSL 则分为了 Client Side 和 Server Side。也就是说可以在 Client Side 和 Server Side 使用不同的这几种类型的 Profile。但每侧都只能配置一个 Profile。

Client Side

BIG-IP 是一个完整的代理架构，这意味着代理、客户端和服务器端的每一端都有单独的 IP 堆栈。这意味着随着流量通过 BIG-IP 前进，在某个时候它会从一个堆栈传输到另一个堆栈，反之亦然，从服务器到客户端的响应。

这很重要，因为不同的配置文件、配置对象和 iRules 命令本身仅在不同的上下文中可用或功能不同。重要的是要知道您是否正在尝试或需要影响客户端流量或服务器端流量以完成您想要做的事情。当您深入研究使用 F5 技术，尤其是 iRules 时，您会经常听到这个术语，描述它的最简单方式是“发生在代理架构客户端的任何事情都在客户端语境”。

Server Side

服务器端，正如您可能推断的那样，与上面的完全相反。它是一个术语，用于描述发生在代理架构服务器端的事情。这通常是来自应用程序服务器的响应，但请记住，客户端和服务器端完全取决于代理的哪一侧发起事务。如果您在池中有一台服务器通过 BIG-IP 进行正向代理，那么该服务器现在是代理讨论目的的“客户端”，因此它位于客户端交易，尽管它本身就是一个服务器。

TMM

TMM 是流量管理微内核。它是 F5 专门为处理流量、处理和路由而开发的定制内核。它从头开始设计为高性能、可靠且灵活，可满足我们的需求。 TMM 负责对任何 BIG-IP 进行所有实际流量处理。无论是正在执行的 iRule、检查通过 VIP 的流量的配置文件，还是在流量穿过 F5 设备时接触流量的任何其他内容，它都发生在 TMM 中。 TMM 独立于“主机操作系统”，这可能是目前关于 TMM 唯一需要了解的重要事项。主机操作系统处理诸如 syslog、sshd、httpd 等的事情，同时让 TMM 自由地做它最擅长的事情——处理高得离谱的流量。

CMP

CMP 是“集群多处理”。这是 F5 处理多个核心设备的专有方式。本质上，从非常粗略的意义上讲，设备中的每个核心都被分配了自己的单独 TMM（见上文）来处理该核心的流量处理。然后系统中内置了一个自定义分解器 (DAG)，它决定将流量发送到哪个 TMM 以及哪个核心进行处理。通过这种方式，F5 能够在多 CPU、多核心系统中实现大规模线性可扩展性。这只是该技术的表面，但在讨论 iRules 时，重要的是要知道 CMP 是一件好事，而从 CMP 中破坏或降级通常来说是不好的。希望这能解释为什么会这样，以及 CMP 在基本层面上是什么。

上述内容涉及irules 编写的相关注意事项。

第三章、iRules 基本概念

3.1 什么是 **iRule**?

用最简单的术语来说，iRule 是针对通过 F5 设备的网络流量执行的脚本。不过，这很含糊，所以让我们尝试更多地定义 iRule 中实际发生的事情。这个想法很简单； iRules 使您能够编写简单的、网络感知的代码段，这些代码段将以多种方式影响您的网络流量。无论您是要执行产品内置选项中不提供或者不可用的某种会话保持方式和速率限制功能，还是要通过精细控制给定会话的流量甚至内容来完全自定义处理方式，这就是 iRules 的目标。

iRules 可以路由、重新路由、重定向、检查、修改、延迟、丢弃或拒绝、记录或.....对通过 BIG-IP 的网络流量进行任何其他操作。 iRules 背后的想法是使 BIG-IP 几乎具有无限的灵活性。我们很早就认识到用户需要能够配置他们的系统以没有想到的许多方式与网络流量交互，或者只是极端情况和/或我们正在处理的少数流量用户。因此，与其在每次他们希望能够以与标准 UI 中提供的复选框和下拉列表集合略有不同的方式使用他们的 F5 设备时，强迫他们提交请求来修改我们的核心架构，我们为他们提供了 iRules，从而提供了一种在他们需要时做他们需要做的事情的方法。归根结底，iRules 是一种网络感知的自定义语言，用户可以使用它在网络层的部署中添加业务和应用程序逻辑。您可以在下面看到一个基本的 iRule 示例，这就是 iRule 的样子，我们将在本系列的后续部分中更深入地探索 iRule 的不同部分。如果您还不完全熟悉这些代码，请不要因此而害怕，随着本系列的继续，我们将深入探讨构建 iRules 所需的每个部分。现在的想法是开始让 iRules 看起来和感觉起来更熟悉。

```
# Rename a cookie by inserting a new cookie name with the same
value as the original.  Then remove the old cookie.
when HTTP_REQUEST {
# Check if old cookie exists in request
if { [HTTP::cookie exists "old-cookie-name"] } {
# Insert a new cookie with the new name and old cookie's value
HTTP::cookie insert name "new-cookie-name" value [HTTP::cookie
value "old-cookie-name"]
# Remove the old cookie
HTTP::cookie remove "old-cookie-name"
}
}
```

3.2 iRule 是如何工作的？

就 F5 而言，iRule 首先是配置对象。这意味着它是您的通用 bigip.conf 以及您的池、虚拟服务器、监视器等的一部分。一般来说，它是通过 GUI 或 CLI 输入到系统中的。在 DevCentral 上还有一个 iRules Editor 可供下载，它是一个用于编辑和部署/测试 iRules 的 Windows 工具，非常有用。但是，与大多数配置对象不同，iRule 完全是用户生成和自定义的。 iRule 是一个脚本，毕竟它的核心是脚本。无论 iRule 如何到达那里，无论是 UI、CLI 还是编辑器，一旦 iRule 成为您的配置的一部分，它就会在配置保存后立即编译。

关于 iRule 的一个严重误解是，与大多数解释性脚本语言（如 TCL）一样，每次执行 iRule 时都必须实例化解释器以解析代码并处理它。这根本不是真的，因为每次保存配置时，所有 iRule 都会预编译成所谓的“字节代码”。字节码大部分是经过编译的，并且已经执行了绝大多数解释器任务，因此 TMM 可以直接解释剩余的对象。这使得性能更高，因此增加了可扩展性。

现在 iRule 已保存并预编译，然后必须将其应用到虚拟服务器才能控制流量。出于所有意图和目的，未应用于虚拟的 iRule 被有效禁用。然而，一旦您将 iRule 应用到给定的虚拟服务器，它将在技术上应用于所有通过该虚拟服务器的流量。但请记住，这并不一定意味着所有通过相关虚拟机的流量都会受到影响。 IRules 通常非常有选择性地影响它们影响的流量，无论是修改、重新路由还是其他方式。这是通过 iRule 内的逻辑结构和 iRule 本身的事件定义来完成的。

Events 事件是使 iRules 作为一种语言具有网络感知能力的方式之一。事件是在网络会话流中的给定时间点执行 iRules 代码的一种方式，我们将在本系列的下一部分中对其进行更详细的探讨。如果我只想为应用 iRule 的虚拟服务器的每个新连接执行一段代码，我可以通过在适当的事件中编写一些简单的代码来轻松地做到这一点。事件也很重要，因为它们指示 iRule 在代理链（有时称为 hud 链）中的哪个点执行。由于 BIG-IP 是双向代理，因此 iRules 不仅可以在代理的右侧执行，而且要在网络流的正确时刻执行，这一点很重要。

所以现在你有一个 iRule 添加到你的配置中，它已经在保存配置时自动预编译为字节代码，你将它应用到适当的虚拟服务器，并且 iRule 中的代码调用所需的事件，其中你想让你的代码执行；可以说，现在是奇迹发生的时候。这就是大量 iRules 命令发挥作用的地方。从标头修改到完全有效负载替换，再到创建与外部系统的套接字连接，以及在处理虚拟流量之前发出请求，组合适当的 iRules 命令系列时，可以实现的目标几乎没有限制。这些命令随后由 TMM 处理，这将影响它为给定会话处理的流量所需的任何更改，具体取决于您设计 iRule 的目的。 iRules 的真正力量在很大程度上发挥了作用，这要归功于我们在语言中内置的大量自定义命令，使您能够充分利用 BIG-IP。

3.3 我什么时候使用 **iRule**？

使用 iRule 的理想时机是当您希望在网络层向您的应用程序或应用程序部署添加某种形式的功能时，并且该功能尚未通过 BIG-IP 中的内置配置选项轻松获得.无论是执行某种自定义重定向还是记录有关用户会话的特定信息或大量其他可能性，iRules 都可以为您的部署添加有价值的业务逻辑甚至应用程序功能。 iRules 可以单独应用，即您的 BIG-IP，而不是分布到托管您试图修改或影响的任何应用程序的每台服务器。这可以节省宝贵的管理时间，也可以大大缩短部署时间。部署 iRule 或 iRule 更改通常比修改应用程序以进行快速修复要容易得多。

如，首次引入 iRules 时最常见的用途之一是将所有流量从 HTTP（端口 80）重定向到 HTTPS（端口 443），而不影响连接的主机或请求的 URI。这曾经是（现在仍然是，如下图所示）一个非常简单的 iRule，但当时它不是 BIG-IP 标准配置选项中可用的功能。

```
when HTTP_REQUEST {
HTTP::redirect "https://[HTTP::host][HTTP::uri]"
}
```

3.4 我什么时候不使用 **iRule**？

上面的 HTTP 到 HTTPS 重定向 iRule 示例实际上完美地描述了何时不使用 iRule，因为该功能非常流行，以至于它已作为配置文件选项直接添加到 BIG-IP 配置中。因此，与编写 iRule 来执行相同任务相比，在配置文件中使用该功能更合适，技术上性能更高。一般的经验法则是：任何时候你可以在标准配置选项、配置文件、GUI 或 CLI 中做某事——先在那里做。如果您希望执行无法通过“内置”配置方式完成的任务，那么现在是转向 iRules 以扩展可能性的最佳时机。

这是出于几个原因，其中最重要的是性能。通常，如果编写得当，iRule 具有极高的性能，但是当您可以直接从内置的核心功能运行功能时，与自定义创建的脚本（甚至是 iRule）相比，性能总是有轻微的好处。此外，通过升级维护产品内置的功能比重新测试和管理可以用一些配置选项轻松替换的 iRule 更容易。

第四章、事件 & 优先级

本章将介绍 iRules 中的两个核心概念：事件和优先级。Events & priority

4.1 Events

4.1.1 什么是Events?

iRules 是一种事件驱动的语言。这意味着每次调用整个脚本体时，都不会执行独立代码。相反，代码放置在事件之下，事件就像容器一样，将 iRule 中的代码分类和分离为逻辑部分。这对于让 iRules 真正具有网络感知能力很重要。

在传统脚本环境中编程时，每次调用脚本时都会执行整个脚本是正常的。然而，在基于网络的脚本环境中，理想情况下我们希望在连接流中的特定时刻调用代码段。 例如您想要查找 HTTP 主机名，希望在建立连接、执行握手并发送包含您要查找的信息的适当阶段执行脚本，但是在这之前还有需要一系列的操作，比如IP报文解析，TCP 握手交互等相关操作。但是你并不关心这些过程，只是想处理HTTP 报文中是否有我想处理的 Host 主机名。

正因为如此，iRules 被设计为一种事件驱动的语言。由于 Tcl 的灵活性以及我们根据需要轻松修改它的能力，我们能够扩展它以理解会话流中数据处理的阶段。我们将这些阶段称为事件。 iRules 用户可以简单地调用 HTTP_REQUEST 事件并相信 BIG-IP 将执行处理代码，而不是读取通过网络传输的每个数据包并等待直到匹配到你的代码逻辑中描述 HTTP 请求的信息。

因此，当编写 iRules 时，“事件”是我们添加到 iRules 中的 Tcl 命令，用于描述网络连接会话流中的特定时刻，例如下面的“CLIENT_ACCEPTED”和“HTTP_REQUEST”。

```
#当会话建立完成后，匹配客户端ip 地址是否是192.168.1.1， 如果是就丢弃，不是继续执行。按照代码执行顺序当HTTP_REQUEST 发生时，就是表面含义HTTP 请求报文到达时，根据报文内容输出对应日志。easy
when CLIENT_ACCEPTED {
if { [IP::client_addr] eq “192.168.1.1” } {
drop
}
}
when HTTP_REQUEST {
log local0. “Host: [HTTP::host], Client IP: [IP::client_addr]”
}
```

4.1.2 事件驱动语言有什么好处？

除了能够在所需的特定时间准确执行代码（考虑到通常需要仅在流程中的特定点可用的信息这一点很重要）之外，事件驱动语言对性能也有很大好处。在100 到工序中你只想变更其中一项，而有100 个事件对应这100道工序。

例如，如果您有一个 100 行的 iRule，其中请求事件中有 50 行，响应事件中有 50 行，那么您的 BIG-IP 仅为每个入站请求执行 50 行代码，而不必执行整个规则。这可能看起来不像是大量节省，但请记住，由于通常讨论的规模，在谈论网络端脚本性能时，一切都变得比在传统环境中更重要。在繁忙的环境中，传统脚本可能每秒执行数百次，而在高流量部署中，iRule 可能每秒执行数十万次。

另一种看待这个问题的方式，事件本身是针对网络执行的脚本的容器。 iRule 是一种将这些事件分组到一个逻辑容器中的方法，网络流量的功能或应用程序或业务逻辑段你只想处理其中的某一个阶段。

4.1.3 有哪些可用事件？

事件被分解成协议，每个支持的协议通常有几个事件。事件将始终遵循 BIG-IP 中的配置文件，配置文件表示设备能够解析处理的协议是哪些, 如果你想处理的协议没有对应的配置文件，那么不要着急你可以通过irules 自己编写对应协议的处理数据，例如: 你涉及一个协议叫做， 它基于TCP 协议运行，你给这个协议定义了一本书那么厚的规范，那么你可以根据这个规范使用tcl 代码一步一步解析数据报文。而事件的好处就是监听了协议处理的阶段和格式化了协议报文，并内置许多协议字段的变量。你只需要调用然后执行你的处理逻辑代码。

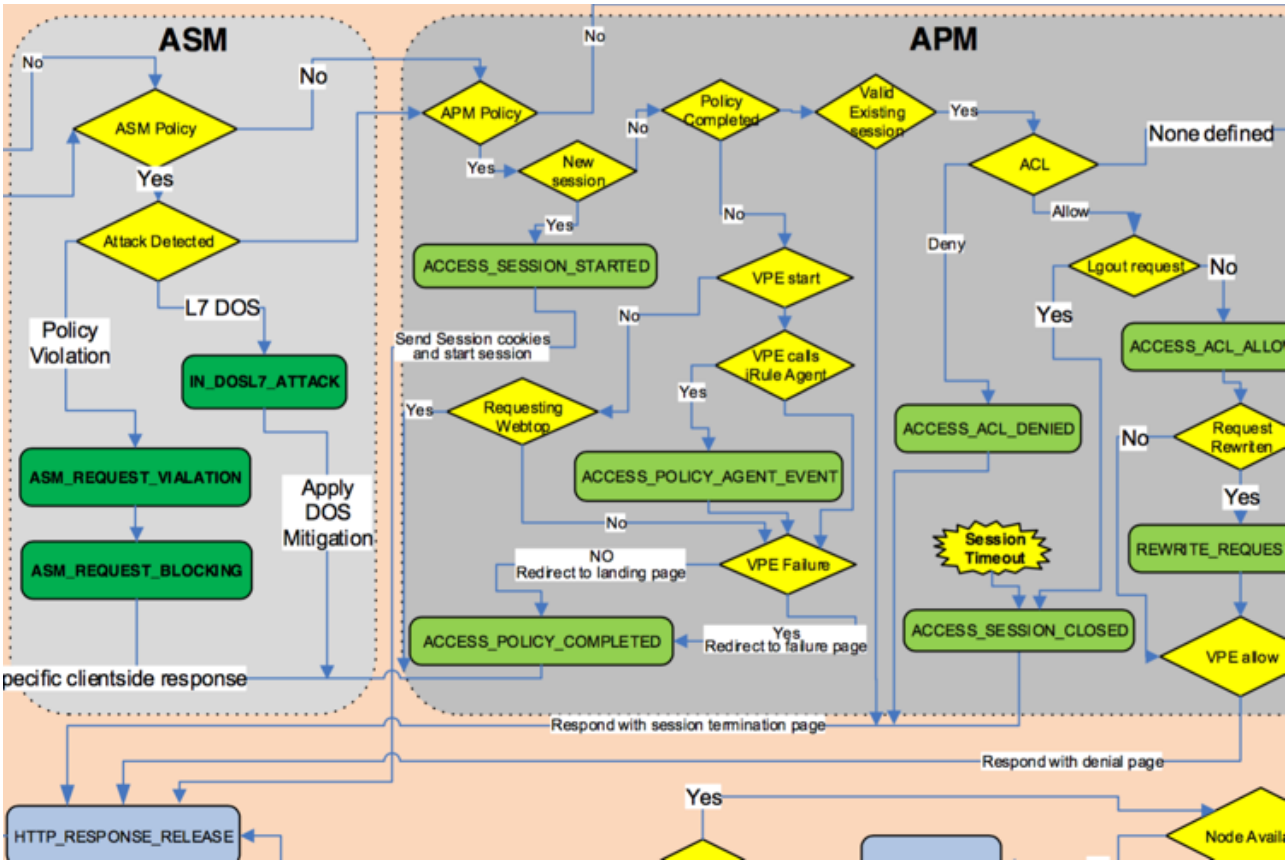
从 HTTP 到 SIP 再到 DNS，厂商提供了绝大多数主要协议事件，并为它们指定了特定的 iRules 事件和可能的关联命令。还值得注意的是，在连接期间可以使用 event disable 命令禁用特定事件。使用它时要小心，因为它肯定会影响应用于同一虚拟服务的任何其他 iRule。

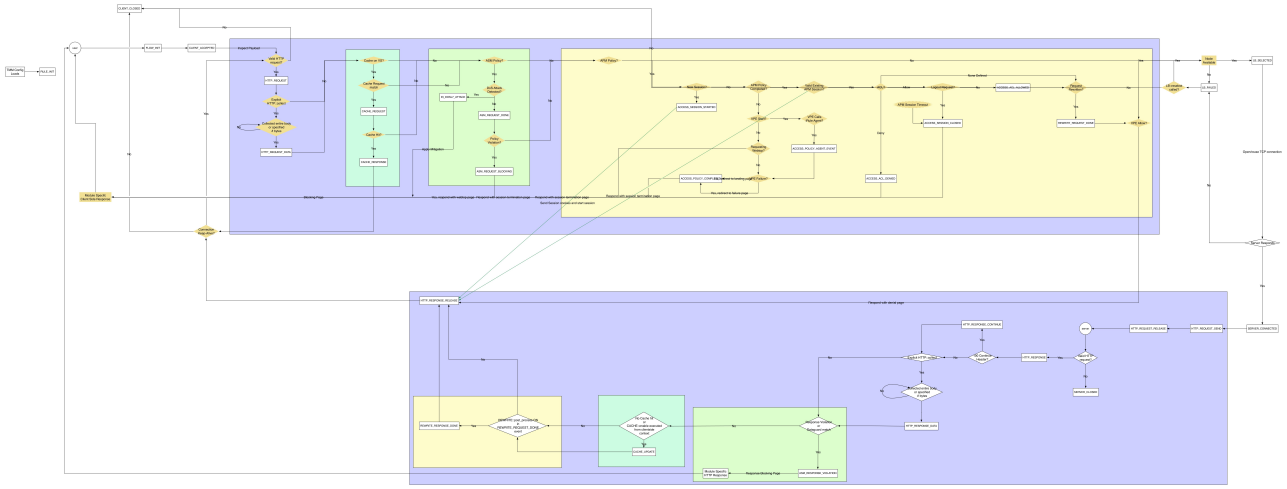
4.1.4事件触发的顺序是什么？

事件触发的方式主要基于通过 BIG-IP 中的代理的网络流。 BIG-IP 中的代理架构由一系列过滤器组成，每个过滤器都设计用于执行特定功能,例如: LHTTP 过滤器使用由 TCP 过滤器从会话中提取的信息。这些过滤器中的每一个都有一组关联的 iRules 事件，这些事件也会在会话生命周期的那个点触发，当连接被过滤或受到影响时回触发特定的过滤器。

随着连接在链上向上传递，OSI 模型自底向下，(我感觉自顶向下那本书不如自底向上那本书😁)越来越多的特定过滤器会触发，连同要匹配的事件，直到最终会话从代理的客户端转发给后端服务器，然后它沿着链向下遍历。响应的回程遵循类似的路径，但具有不同的过滤器，因此事件已就位；这里存在描述 iRule 中事件顺序的部分问题。

没有针对每个连接发生的静态事件集。可能有一些特定的事件总是触发，例如每个连接一次CLIENT_ACCEPTED，但通常没有触发的设置事件结构。触发的事件取决于应用了相关iRule 的虚拟机的配置。由于这种动态特性，很难就给定的 iRule 中将触发哪些事件做出笼统的陈述，更不用说事件顺序了。鉴于虚拟服务器的整个配置，可以确定将触发哪些事件以及触发顺序，但这需要根据具体信息分析。下面是绘制的 iRule 事件的屏幕截图。这显示了事件与给定配置和流量考虑因素相关的方式，这是理解事件顺序的开始。HTTP_REQUEST 事件中不应包含CLIENT_ACCEPTED 事件。





4.1.5 如果在一个**VIP**上关联的多个 **iRule** 中有相同的事件会怎样？

许多人会在单个虚拟服务器上拥有多个 iRule，其中一些会使用类似的事件。在这种情况下会发生什么的问题经常被问到，或者用户会想知道“哪个 CLIENT_ACCEPTED 会先触发？”如果您的虚拟服务器具有多个 iRule，所有 iRule 都使用特定事件，则该事件没有“第一次”或“最后一次”发生，它只会发生一次。

为了进一步了解，首先让我解释一下 iRule 是如何存储和解释的。就所有意图和目的而言，iRules 并不是一种动态解释的语言。也就是说，每次请求进入应用了 iRule 的虚拟服务器时，Tcl 解释器都不会启动。这将是非常低效的。相反，当 iRule 作为配置的一部分保存时，它由 Tcl 引擎解释并预编译为所谓的“字节代码”。这是一个编译状态，它分解了给定 iRule 中的大部分逻辑和命令，以便在执行时它可以非常容易和有效地被 Tcl 引擎触发，在本例中是 TMM 的一部分。

在这个过程中，iRule也被有效地分解和重新组织。在iRule中与每个事件相关的命令被从那个特定的iRule的上下文中移除，而是与那个事件相关。还记得我在上面提到事件与代理架构中的特定过滤器联系在一起的想法吗，当这些过滤器被调用时，事件就会发生？这就是为什么这个概念很重要。你放在HTTP_REQUEST事件下的命令都被从它们所关联的iRules中取出来，而被归入负责执行这些命令的事件下。

因此，没有多个实例 HTTP_REQUEST 或 CLIENT_ACCEPTED 或 SIP_RESPONSE 的概念，也没有给定虚拟服务器发生的任何其他事件的概念。会话流量在链上进行，事件触发，命令在该上下文中执行，无论是客户端还是服务器。无论您有多少 iRule 调用给定事件，该事件在每次连接/请求时只会触发一次，具体取决于事件。而事件的调用取决于事件的优先级。

即使不能从特定的虚拟服务器执行给定事件的多个实例，您仍然可以控制iRules的执行顺序。这是通过我们将讨论的下一个概念，即优先级来实现的。相同事件只会发生一次。

4.2 Priorities

4.2.1 什么是优先级**?**

优先级是与每个 iRule 关联的整数值，它确定 iRule 应用于给定虚拟服务器的执行顺序，数字越小，相对优先级越高。 优先级为 1 的 iRule 会在优先级为 900 的 iRule 之前触发。如果要确保在 iRuleB 执行时在 iRuleA 中设置的变量存在，则需要相应地设置优先级。

4.2.2 如何设置优先级？

无论您是否设置，每个 iRule 都有一个默认优先级。系统将为所有 iRule 隐式设置一个优先级值（默认值为 500），以便它可以保持运行时执行命令的顺序。当保存没有指定优先级的 iRule 时，这是自动完成的。一个细微差别是，在 GUI 中没有明确排序的所有 iRule 都将按从上到下的优先级排序，即使它们都共享隐式值。

但是，如果您想设置特定的优先级，只需通过 priority 命令即可。在 iRule 的开头，您可以声明“priority 100”以指定该 iRule 的优先级，并通过关联指定其中包含的所有命令。如果您打算开始设置特定的优先级，建议您对所有 iRule 都这样做，但是，以确保实现您想要的执行顺序。

```
priority 100
when HTTP_REQUEST {
if {[HTTP::host] eq “mydomain.com” } {
set host [HTTP::host]
}
}
```

4.2.3 为什么优先级很重要？

在许多情况下，事实并非如此。我见过的绝大多数 iRules 部署都没有设置特定的优先级。但是，在您希望确保实现特定执行顺序的情况下，它可能很有用。几个可能的示例可能是，如果您将特定信息从一个 iRule 传递到另一个，或者希望 iRule 对某种修改后的值起作用。不管是什么原因，如果您还没有寻找一种方法来设置优先级，您很可能不需要担心它。

4.2.4 我可以改变数据流处理顺序吗？

回答：no 。人们有时会问他们是否能够做一些事情，比如在请求之前对响应采取行动，在建立连接后立即查看负载平衡决策，或者其他违反数据流逻辑顺序的事请。就其本质而言，事件是根据流量通过代理的方式发生的，没有多少优先级会改变这一点。你绝对可以保证iRuleA的响应事件发生在iRuleB的响应事件之前，但是你不能强制iRuleA的响应事件发生在iRuleB的请求事件之前。你不能还没开始就要结果 😁

您可以使用几个条件和变量实现类似的效果，这是我向需要模仿此行为的任何人推荐的逻辑技巧。请记住，当您在虚拟服务器上有多个 iRule 时，你需要考虑好它们出的事件顺序，否则可能出现意外。请谨慎使用，因为这绝对是有风险的。你不可能更改实际的事件结构或顺序。

```
#在HTTP_RESPONSE 定义变量，在下一次HTTP_REQUEST 中取消变量。不是本次相应对应的事件
when HTTP_REQUEST {
if { [info exists response_fired] } {
unset response_fired
...
}
}
when HTTP_RESPONSE {
set response_fired 1
...
}
```

第五章、变量

变量是任何编程语言中必不可少的东西，本章讨论irules 中变量的使用。

5.1 什么是变量？

在 Tcl 中，没有变量声明的概念。一旦遇到新变量名，Tcl 将定义一个新变量。变量的名称可以包含任何字符和长度。您甚至可以通过将变量括在花括号中来使用空格，但这不是首选。Tcl的变量有两种，分别是简单变量和数组。

set 命令用于为变量赋值。 set 命令的语法是:

```
set variableName value
```

下面显示了一些变量示例：

```
% set variableA 10
10
% set {variable B} test
test
% puts $variableA
10
% puts ${variable B}
test
% set  aa  {1,2,3,4,5}
1,2,3,4,5
```

正如您在上面的程序中看到的，$variableName 用于获取变量的值。

5.2 如何在**iRules** 中使用变量？

任何变量都有两个主要功能：设置和检索。要在 Tcl 中设置变量，您只需使用 set 命令并指定所需的值。这可以是静态或动态值，例如整数或命令的结果。然后将所需的值存储在内存中并与提供的变量名相关联。如下：

```
#Basic variable creation in Tcl
set integer 5
set hostname [HTTP::host]
```

要检索与给定变量关联的值，您只需直接引用该变量即可获得结果值。例如“$integer”和“$hostname”将引用上述示例中的值。而变量与各种控制逻辑配合使用：

```
#Basic variable reference to retrieve and use the value stored in
memory
if {$integer > 0) {
log local0. "Host: $hostname"
}
```

5.3 在 **iRules** 中可以使用哪些变量类型？

考虑到变量只是一个简单的内存结构，并且有许多不同类型的内存结构可供您通过 Tcl 和 iRules 使用。从简单的变量到数组再到表和数据组，内存中的数据管理方式有很多种。然而，出于本文的目的，我们将重点关注实际变量的两种主要类型，并将对其他数据结构的讨论留到以后进行。

iRules 中有两种主要类型的变量，局部变量和全局变量。

5.3.1 Local Variables 局部变量

除非另有说明，否则所有变量都作为 iRule 中的局部变量创建。这意味着什么？好吧，局部变量意味着它被分配了与创建它的 iRule 相同的范围。所有 iRule 本质上都是基于连接的，因此所有局部变量也是基于连接的。这意味着连接决定了给定 iRule 的局部变量和数据的内存空间。例如，如果 connection1 进入并执行 iRule，创建 5 个变量，这些变量将仅存在直到 connection1 关闭并且 BIG-IP 上的连接终止。那时分配给该流的内存将被释放，并且在处理该特定连接的 iRule(s) 时创建的变量将不再可访问。

所有 iRule 和使用上图所示的基本设置命令结构从 iRule 中创建的所有变量都是这种情况。局部变量开销低，易于使用，您永远不必担心它们的内存管理，因为它们会在连接终止时自动清除。重要的是要记住 iRules 作为一个整体，其中的变量是和连接绑定的。当人们期待一个更静态长期存储的变量时，需要小心。

局部变量将占 iRules 中变量使用的绝大部分。它们高效、易于使用，并且根据情况非常有用。这些可以直接设置，如上所示，但通常也是提供输出的命令的结果。还有多种方法可以在 iRules 中引用变量。当使用直接影响变量的命令时，通常最好不要使用“$”并直接引用名称，有时大括号可以让您更清楚地定义变量名称的开头和结尾。一些示例如下所示：

```
#Standard variable reference
log local0. "My host is $host"

#Set variable to the output of a particular command using brackets
[]
set int [expr {5 + 8}]

#Directly manipulating a variable's value means no "$", most times
incr int

#Bracing can allow you to deliniate between variable name and
adjacent characters
log local0. "Today's date is the ${int}th"
```

5.3.2 Global Variables全局变量

实际上，“全局变量”有点用词不当。这是大多数编程语言（包括 Tcl）中使用的通用术语，用于表示存在于本地内存空间之外的变量。即，在 iRules 的情况下，变量存在于单个连接的约束之外。例如，如果我想存储我的日志服务器的 IP 地址，并使它始终对通过 BIG-IP 的每个连接可用，而不必在我的 iRule 每次触发时都重新设置该变量怎么办？那将是一个全局变量。 Tcl 有一个处理这个的机制，而且它相对容易使用。

Tcl 中的全局变量处理需要一个共享内存空间，在irules 中，它会做一些被称为从 CMP 中“降级”的事情。正如您从“F5 技术和术语简介”一文中回忆的那样，CMP 是集群多核处理，是一种允许我们以高效、可扩展的方式将 BIG-IP 中的任务分配给多个内核的技术。由于默认 Tcl 全局变量处理的要求，在 Tcl 中使用全局变量会强制任何通过 Virtual 的连接（违规 iRule 应用到该连接）导致 CMP 降级，一个虚拟服务只使用一个处理核心，而不是所有的处理核心来处理该虚拟服务的流量，not good，会严重限制性能。

因此，我们强烈建议避免使用传统意义上的全局变量。但是那个日志服务器地址呢？仍然确实需要将长期存在的数据存储在内存中并随意使用。正是出于这个目的，我们在 iRules 中包含了一个新的命名空间，称为“静态”命名空间。您可以在不破坏 CMP 的情况下有效地在静态命名空间中设置静态全局变量数据，从而不会降低性能。为此，只需设置您的 static::varname 变量，可能在 RULE_INIT 中，因为该特殊事件仅在加载时运行一次，并且静态变量在范围内是全局的，这意味着它们将保持设置状态，直到重新加载配置。一旦设置了这些变量，您就可以像在任何 iRule 中调用任何其他变量一样调用它们，并且它们将始终可用。它看起来像这样：

```
#Set a static variable value, which will exist until config reload,
living outside of the scope of any one particular iRule
set static::logserver "10.10.1.145"

#Reference that static variable just as you would any other
variable from within any iRule
log $static::logserver "This is a remote log message"
```

为了更全面地了解您可以在 iRules 中访问的不同类型的内存结构，我提供了一个方便的表格来按类型显示内存结构以及有关每个结构的一些信息，以及使用该结构的示例。我们将在本系列的后续部分中更详细地介绍其中一些内容，但我想让您了解可用的不同类型的内存结构。

	Variable Unset	CMP Friendly	Minimum Version	Mirrored	Create, read, delete syntax
Local	Unset command or upon connection close	Yes	v9.0	No	set <key> <value> $<key> unset <key>
Global	Unset command	No	V9.0	No	set ::<key> <value> $::<key> unset ::<key>
Static	Static can't be unset	Yes	v9.7	Yes	set static::<key> <value> $static::<key> unset $static::<key>
Session	Timeout expiration or session delete command	v9.0 – No v10.0 – Yes	v9.0	Yes	session add uie <key> <value> session lookup uie <key> session delete uie <key>
Table	Timeout expiration or table delete command	Yes	v10.1	Yes	table set <key> <value> table lookup <key> table delete <key>

注：表中略有误差。可以取消设置静态变量。

5.4 使用变量时是否会对性能产生影响？

运行任何未缓存在任何脚本中的命令都会产生一些相应的开销，这是没有办法解决的。iRules 中的变量恰好具有极小的成本，一般来说，只要您正确使用它们即可。与变量相关的成本在创建过程中。将所需数据存储在内存中并创建对该数据的引用以供将来使用需要占用资源（尽管数量很少）。访问变量只是调用该引用，因此不会产生额外成本。

然而，一个常见的误解是什么构成了 iRules 中的变量，而不是将执行查询的命令或函数。许多人看到诸如“HTTP::host”或“IP::client_addr”之类的东西，并且由于格式原因，假设它是命令或函数，因此将花费 CPU 周期来查询值并返回它。根本不是这样。这些类型的引用缓存在 TMM 中，因此无论您调用“HTTP::host”一次还是 100 次，您都不会需要更多资源，因为您不是每次都执行查询来确定主机名，而是只是引用一个已经存储在内存中的值。将这些命令和许多其他类似命令视为您可以随意使用的预填充变量数据。也就是当你关联了对应的profile 时这些变量就会生成。

5.5 什么时候应该使用变量？

虽然创建变量的成本很低，但仍有一些开销与之相关。在 iRules 中，由于某些部署的执行率过高，我们倾向于极端的效率意识。在这种情况下，我们建议只在实际需要的地方使用变量，而不是许多编程实践要求经常使用它们作为保持代码整洁的一种手段，即使在没有真正保证的情况下也是如此。

例如，许多新的 iRule 程序员的常见做法是执行类似 set host [HTTP::host] 的操作，无需定义直接使用

然而，正如我刚刚在上面解释的那样，这是完全没有必要的。相反，在您需要引用此信息的任何时候简单地重新使用 HTTP::host 命令会更有效。然而，当您打算以某种方式修改数据时，使用变量确实有意义。例如登录 local0。“我的小写 URI：[string tolower [HTTP::uri]]”

以上将为您提供 HTTP URI 的全小写表示。这是一个非常常见的用例，需要在给定的 iRule 中多次引用 URI 的小写版本并不异常。虽然 HTTP::uri 命令被缓存并且无论您引用它多少次都不会产生额外的开销，但 string tolower 命令不会。因此，在这种情况下，假设您要引用小写 URI 至少 2 次或更多次，以创建一个变量并引用它：

```
set loweruri [string tolower [HTTP::uri]]
log local0. "My lower case URI: $loweruri"
```

实际上，您希望在任何时候都必须使用变量来对具有相关的值重复任何操作。与其多次重复该操作并累积额外的开销，不如执行一次操作、存储结果并引用变量。

第六章、Delimiters 定界符

6.1 空格

tcl 和 iRules 中主要的（并且经常被忽视的）分隔符是空格。

在 tcl 中，基本上一切都是命令。命令有名称、选项和参数。空格（空格或制表符）用于分隔命令名称及其选项和参数，它们只是字符串。换行符或分号用于终止命令。

在此示例中记录 log local0. "iRule, Do you?" 分解命令时，命令的组成部分是：

COMMAND	LOG LOCAL0. "IRULE. DO YOU?"
Command Name	log
Command Parameter	local0.
Command Parameter	"iRule. Do you?"

空格字符也用于分隔 tcl 列表中的元素。

6.2 双引号

您可能已经注意到，上面的第二个参数是一个由多个单词组成的字符串，以空格分隔，但用双引号括起来。这是一个用双引号分组的例子。

一对双引号 " " 之间的所有内容，包括换行符和分号，都被分组并解释为单个连续字符串。调用封闭的字符串时，将丢弃封闭的引号并保留任何空白填充。如果文字双引号字符用反斜杠转义（"），则它可以包含在由双引号分隔的字符串中。变量替换总是在由双引号分隔的字符串中执行。

避免在已经连续的字母数字字符串上使用双引号，因为这会导致解释器进行额外的访问以创建不必要的分组。但是，即使在连续字符串中，如果使用除字母数字之外的任何字符，最好将整个字符串括在引号中以强制解释为连续字符串。例如，最常用的 iRules 命令之一 HTTP::redirect 将一个完全限定的 URL 作为参数，该 URL 应该用双引号引起来：
HTTP::redirect "http://host.domain.com/uri"

在表达式中作为操作数的字符串必须用双引号或大括号括起来，以避免被解释为数学函数。

6.3 方括号

嵌套命令由方括号 [ ] 分隔，这强制将内容作为命令进行内联操作。括号之间的所有内容都被评估为命令，解释器通过丢弃括号并将它们之间的所有内容替换为嵌套命令的结果来重写嵌套它的命令。（如果您编写过任何 shell 脚本，这大致相当于用反引号包围命令，尽管 tcl 允许多重嵌套。）

在此示例中，方括号分隔的命令在最终执行之前被替换并代入日志命令：

```
log local0. "[IP::client_addr]:[TCP::client_addr] requested
[HTTP::uri]"
```

方括号内的字符串不是有效的 tcl 或 iRule 命令，当解释器试图将括号内的字符串解释为命令时，将生成“未定义过程”错误。将方括号用于强制命令替换之外的目的需要转义或反斜杠替换。

6.4 反斜杠替换（转义）

反斜杠替换是通过在它们前面加上反斜杠来转义具有特殊含义的字符（$ {} [] "换行符等）的做法。此示例生成“未定义过程：0-9”错误，因为字符串“0-9”在里面[ ] 被解释为命令：

```
if { [HTTP::uri] matches_regex "[0-9]{2,3}"} {
TCL error: undefined procedure: 0-9     while executing "[0-9]
```

此示例按预期工作，因为方括号已被转义：

```
if { $uri matches_regex "\[0-9\]{2,3}" } {
  body
}
```

此示例也有效，因为大括号内方括号的内容未被评估为命令（并且它也更具可读性）：

```
if { $uri matches_regex {[0-9]{2,3}}}{
  body
}
```

```
$ set  y {/n$x   [expr 10+100]}
/n$x   [expr 10+100]
```

除了使用反斜杠外，TCL提供另外两种方法来使得解释器把分隔符和置换符等特殊字符当作普通字符，而不作特殊处理，这就要使用双引号和花括号({})。（上述 2 种方法在性能方面大致相同。）

反斜杠替换的另一个常见用途是续行：在多行上继续长命令。当一行以反斜杠结尾时，解释器将它和它后面的换行符以及下一行开头的所有空格转换为单个空格字符，然后将连接的行评估为单个命令：

```
log local0. "[IP::client_addr]:[TCP::client_addr] requested
[HTTP::uri] at [clock \
  seconds]. Request headers were [HTTP::header names].  Method was
[HTTP::method]"
```

反斜杠替换（在需要时）不会显着影响性能。

6.5 括号

iRules 中圆括号 ( ) 最常见的用途是执行比较运算符。排除括号并无意中否定第一个操作数而不是比较结果是一个常见的错误，这可能导致神秘的“不能使用非数字字符串作为“！”的操作数错误：如果第一个操作数是非数字的，评估将失败，因为在数学意义上不能否定非数字值。!() “不能使用非数字字符串作为“！”的操作数：如果第一个操作数是非数字的，xz执行将失败，因为在数学意义上不能! 非数字值。

下面是一个演示如何使用括号正确执行否定比较的示例。以下方法按预期记录“不匹配”，因为 $x 是一个字符串且不等于 3：

```
set x xxx
if { !($x == 3) } {
  log "no match"
} else {
  log "match
}
```

这种（错误的）方法会导致上述运行时错误：

```
set x xxx
if { !$x == 3 } {
  log "no match"
} else {
  log "match"
}
TCL error: can't use non-numeric string as operand of "!"
while executing "if { !$x == 3 } {   log "no match" } else {   log
"match" }"
```

iRules 中圆括号的其他用途是用于子匹配表达式的正则表达式分组：

```
regexp -inline (http://)(.*?)(\.)(.*?)(\.)(.*?)
```

并引用数组中的元素：

```
set array(element1) $val
```

6.6 花括号

花括号是 tcl 和 iRules 中使用最广泛但可能最不为人所知的分隔形式。它们可能会造成混淆，因为它们用于许多不同的上下文中，但花括号基本上定义了一个或多个元素的空格分隔列表，或者一个或多个命令的换行符分隔列表。我将尝试举例说明大括号在 iRules 中常用的各种不同方式。

6.6.1 花括号列表

列表通常是用大括号括起来的空格分隔的字符。列表，如大括号，可以嵌套。此示例显示了一个包含 3 个元素的列表，每个元素都包含一个包含 2 个元素的列表：

```
{ { {ab} {cd} } { {ef} {gh} } { {ij} {kl} } }
```

6.6.2 子命令列表

一些命令使用大括号来分隔子命令列表。“when”命令就是一个很好的例子：在每个 iRule 中，大括号用于分隔所有 iRules 事件，封装包含每个触发事件逻辑的命令列表。每个事件声明的第一行必须以左大括号结尾，最后一行必须以右大括号结尾，如下所示：

```
when HTTP_REQUEST {
  body
}
```

还有其他几个命令使用大括号来对命令进行分组，例如 switch：

语法：

```
switch condition body condition body
Examples:

switch {
  case1 { body }
  case2 { body }
}
```

or

```
switch {
  case1 {
    body
  }
  case2 {
   body
  }
}
```

6.6.3 带参数列表的大括号

其他命令使用大括号来定义一个参数，它实际上是一组值 - 一个列表。例如，“**string map**”命令。

```
Syntax:

string map charMap string
Example ("a" will be replaced by "x", "b" by "y", and "c" by "z"):

string map {a x b y c z} aabbcc
#{a x b y c z} 是键值对，a=x,b=y,c=z 的形式
结果:xxyyzz
```

6.6.4 参数列表和子命令列表的大括号

有些命令对参数列表和子命令列表都使用大括号：

```
Syntax:

for start test next body
Example:

for {set  x 0 } { x = 10 } { incr x } {
  body...
  ...
}
```

6.6.5 嵌套大括号

无论在什么上下文中使用它们，都可以嵌套多组花括号。最里面的组将成为下一个组的元素，依此类推。匹配的左右大括号之间的所有字符都包含在该组中，包括换行符、分号和嵌套大括号。调用组时，不包括封闭的（即最外层的）大括号。每个左大括号都必须有一个匹配的右大括号——即使是那些包含在注释中的大括号。当您尝试保存 iRule 时，大括号不匹配会产生语法错误。（其他定界符的不匹配和无效的命令语法也可能会产生大括号不匹配的错误，所以如果你找不到不匹配的大括号，开始注释掉没有大括号的行，直到语法检查通过，然后在你注释的最后一行中修复问题。）

```
#Fully Decode URI
when HTTP_REQUEST {
  # decode original URI.
  set tmpUri [HTTP::uri]
  set uri [URI::decode $tmpUri]

  # repeat decoding until the decoded version equals the previous
value.
  while { $uri ne $tmpUri } {
    set tmpUri $uri
    set uri [URI::decode $tmpUri]
  }

  log local0. "Original URI: [HTTP::uri]"
  log local0. "Fully decoded URI: $uri"
}
```

6.6.6 使用大括号的特殊情况

您还可以使用大括号将字符串括起来，就像双引号一样，但变量不会在其中展开，除非大括号内的命令也引用了它们。由于这种差异，如果您需要处理包含 $、\、[]、() 或其他特殊字符的字符串，则大括号可以在双引号不起作用的地方使用。

大括号也可用于分隔嵌入到其他字符串中的变量名，或者包含除字母、数字和下划线以外的字符的变量名，强制变量名的开始和结束边界，而不是依赖空格来这样做。此示例导致运行时错误，因为第一个变量名称的末尾不清楚：

```
set var1 111
set var2 222
log local0. "$var1xxx$var2"
TCL error: can't read "var1xxx": no such variable     while
executing "log local0. "$var1xxx$var2""
```

但是如果你用大括号分隔变量名，它就可以工作：

```
set var1 111
set var2 222
log local0. "${var1}xxx$var2"

Returns:
111xxx222
```

包含破折号字符的类名必须用大括号分隔，以确保正确评估类名。这些都是有效的类引用：

```
matchclass $::MyClass contains "Deb"
matchclass ${::My-Class} contains "Deb"
```

第七章、控制结构& 操作符

控制结构和运算符，这是 iRules 的两个更基本和重要的部分，它们几乎是您可能编写的任何 iRules 脚本的组成部分。将这些与事件结合起来将为您提供几乎所有 iRule 中的逻辑流的基本框架，当然具有不同程度的复杂性。

7.1 什么是控制结构

控制结构是一种逻辑语句，它允许您进行比较，并根据比较的结果执行一组操作。

就像没有事件一样，您将无法根据网络流发生的情况在适当的时间执行代码；如果没有控制结构，您将无法在某些情况下执行代码的逻辑分离，这显然对于构建功能脚本至关重要。

7.2 有哪些控制结构？

7.2.1 if 条件执行语句

语法:

```
if expr1 ?then? body1 elseif expr2 ?then? body2 elseif ... ?else? ?
bodyN?
```

if 命令将 expr1 计算为表达式（与 expr 计算其参数的方式相同）。表达式的值必须是布尔值（数字值，其中 0 为假，任何为真，或字符串值，例如 true 或 yes 表示 true 和 false 或 no 表示 false）；如果为真，则通过将 body1 传递给 Tcl 解释器来执行 body1。否则 expr2 被评估为表达式，如果为真则执行 body2，依此类推。如果所有表达式的计算结果都不为真，则执行 bodyN。 then 和 else 参数是可选的“干扰词”，使命令更易于阅读。可能有任意数量的 elseif 子句，包括零个。 BodyN 也可以省略，只要 else 也被省略。命令的返回值是已执行的正文脚本的结果，如果所有表达式都不为非零且没有 bodyN，则为空字符串。

irules 示例:

```
if {[info exists $auth]} {
pool auth_pool
}
```

else：

```
if {[info exists $auth]} {
pool auth_pool
} else {
pool other_pool
}
```

elseif:

```
if {[info exists $auth]} {
pool auth_pool
} elseif {[info exists $secondary]} {
pool secondary_pool
} else {
default_pool
}
```

then:

```
if {
    $vbl == 1
    || $vbl == 2
    || $vbl == 3
} then {
    puts "vbl is one, two or three"
}
```

注意:表达式可以是多行的，但在这种情况下，为了清晰起见，使用可选的 then 关键字可能是个好主意：

7.2.2 switch

根据一个给定值进行多个判断执行。

语法:

```
switch ?options? string pattern body ?pattern body ...?

switch ?options? string {pattern body ?pattern body ...?}
```

switch 命令按顺序将其字符串参数与每个模式参数进行匹配。一旦找到匹配字符串的模式，它就会通过递归地将其传递给 Tcl 解释器来评估以下主体参数，并返回该评估的结果。如果最后一个模式参数是默认的，那么它匹配任何东西。如果没有模式参数与字符串匹配并且没有给出默认值，则 switch 命令返回一个空字符串。

示例:

查看检查 URI 并根据内容执行操作的基本 switch 语句如下所示：

```
switch [HTTP::uri] {
“/app1” {
pool http_pool
}
}
```

这实际上是说“检查 HTTP::uri 变量，如果它是 /app1，则将流量发送到 http_pool”。这是一个非常基本的 if 语句。然而，这远不是 switch 的所有功能。如果您想对 URI 进行多个可能的匹配怎么办。 IE。“如果 HTTP::uri 是 /app1，则发送到 http_pool，如果是 /app2，则发送到 http_pool2，如果是 /app3，则发送到 http_pool3”。使用 switch 会很简单，看起来像：

```
switch [HTTP::uri] {
“/app1” {
pool http_pool
}
“/app2” {
pool http_pool2
}
“/app3” {
pool http_pool3
}
}
```

请注意不需要 else 或 elseif 语句，以及简单、干净的逻辑流程。最后让我们看一下逻辑 or 语句，这是一个特别有用的技巧，您可以使用 switch 语句来执行。如果您的逻辑语句看起来像“如果 URI 是 /app1、/app2 或 /app3，则发送到 http_pool”。满足该要求的 switch 语句将非常简单：

```
switch [HTTP::uri] {
“/app1” -
“/app2” -
“/app3” {
pool http_pool
}
}
```

switch 语句后的破折号允许它以逻辑“或”的方式通过，从而可以轻松地将多个匹配案例链接到一个动作，如上所示。

最后，通过 switch 语句，您可以使用 glob 样式的模式匹配。

```
switch -glob [HTTP::uri] {
  "*\\?*ghtml*" {
    #do work here
  }
  {*\?*ghtml*} {
    #do work here
  }
}
```

如果模式类似于 \?ghtml* 并且您使用双引号，请注意字符串替换发生在处理 glob 模式参数之前，因此需要两个反斜杠来转义问号。通过使用大括号，您可以避免字符串替换，并且可以按预期匹配 glob 模式。

7.2.3 while

只要满足一个条件就重复执行脚本.

语法:

```
while test body
```

while 命令将 test 作为表达式求值（与 expr 求值其参数的方式相同）。表达式的值必须是一个合适的布尔值；如果它是一个真值，那么通过将它传递给 Tcl 解释器来执行 body。一旦 body 被执行，然后 test 被再次，并且该过程重复直到最终 test 直到为 false 布尔值。Continue 命令可以在 body 内部执行以终止循环的当前迭代，而 break 命令可以在 body 内部执行以立即终止 while 命令。 while 命令总是返回一个空字符串。

注意：测试应该几乎总是用大括号括起来。否则，将在 while 命令开始执行之前进行变量替换，这意味着表达式中不会考虑循环体所做的变量更改。这很可能导致无限循环。如果 test 包含在大括号中，变量替换将延迟到表达式被计算（在每次循环迭代之前），因此变量的变化将是可见的。例如，在 $x<10 周围使用和不使用大括号尝试以下脚本：

```
set x 0
while {$x<10} {
    puts "x is $x"
    incr x
}
```

tcl 有很多控制结构在这里不一 一列举，请查看附录链接。

7.3 选择哪种控制结构?

选择哪种控制结构 一般没有强制的要求，但是我们一般遵循以下原则:

1. If 是最常用的,如果您要进行 1-3 次比较，if 或 if/else(if) 可能是最合适的，并且最容易使用和理解。
2. 如果您要进行 3-10 次比较，您可能应该使用 switch 语句。一般来说，它们具有更高的性能、更易于阅读并且更易于调试。
3. 超过 10-15 个条目，您应该使用data group 和 class command，后面说

7.4 比较运算符

常用的比较运算符有：

OPERATOR	EXAMPLE
contains	[HTTP::uri] contains “abc”
equal	“$x eq $y” or “$x == 5”
starts_with	[HTTP::uri] starts_with “/app1”
not equal	“$x ne $y” or “$x != 5”
ends_with	[HTTP::path] ends_with “.jpg”

显然还有更多 (>, %lt;, >=, <=, etc) 但以上是最常用的，一般来说。比较运算符不是特定于 iRules 的，所以我不会在这里详细说明它们。可以说最普遍接受的比较运算符将在 Tcl 中工作，在tcl 中的比较运算符都能在irules 中使用。

第八章、日志输出 & 代码注释

在任何编程语言中日志都是一个很重要的工具，便于进行故障排查、生成指定消息并将其记录到 Syslog-ng 实用程序。日志命令可以产生大量输出。在生产环境中小心使用，尤其是在磁盘空间有限的情况下。syslog 工具被限制为每个请求记录 1024 字节。较长的字符串将被截断。

语法:

```
log <message>
log [-noname] <facility>.[<level>] <message>
log [-noname] <remote_ip>[:<remote_port>] <facility>.[<level>]
<message>
```

8.1 什么是日志打印？

无论您是在 BIG-IP 上还是在任何其他平台上，日志记录概念都保持不变。你有一个脚本，你告诉它输出信息，而且，一般来说，你有两个关于将它发送到哪里的选项，假设你没有将它发送到某个地方的套接字或另一个系统。您可以将其发送到可用输出之一，通常是命令行或屏幕，或者您可以将其捕获到文件中。为了使整个“将其捕获到文件中”变得更容易和更可预测，在大多数情况下我们都可以使用标准日志文件。在 BIG-IP iRules 日志条目的情况下，默认情况下将转到 /var/log/ltm。您所要做的就是启动日志命令，您的信息将显示出来以供处理或细读。基本日志条目包含条目的数据和时间、设施、严重性、日志消息等。示例条目如下所示：

```
Oct 4 00:42:51 tmm err tmm[17084]: 01220001:3: - "Host: domain.com"
```

8.2 iRules 中有哪些日志记录类型？

在 iRules 中，您可以通过三种主要方式记录信息: **Local Logging, Remote Logging, High Speed Logging**

8.2.1 Local

本地日志记录是最基本和最常见的日志信息形式。这意味着您在 iRule 中执行日志命令，并将信息发送到 /var/log/ltm，如上所述。这也是所有标准 iRules 信息的位置，包括可能从系统中弹出的任何 iRules 错误消息。这些标准日志由 syslog-ng 处理，这意味着它们以标准 syslog 格式存储，以备不时之需。这也意味着日志记录不是由 TMM 处理，而是由主机操作系统处理。

本地日志记录非常适合在开发过程中进行故障排除和执行临时日志语句。然而，本地日志语句确实会将数据写回主机操作系统和磁盘，这意味着它们不一定适合高流量生产部署。它在生产部署中的性能不是很好，尤其是高流量生产部署。本地日志记录易于使用。您所需要的只是一条日志语句，它指定了日志命令、日志工具以及您要输入到日志文件中的消息。日志工具用于识别日志消息的发送者，通常是守护进程，并且在某些情况下可以指定不同的日志记录行为。出于我们在 iRules 中的目的，我们将始终使用“local0.”的日志工具，除非在极少数情况下，自定义情况。完成的日志命令使用起来非常简单，如下所示：

```
when HTTP_REQUEST {
log local0. “Requested hostname: [HTTP::host] from IP:
[IP::local_addr]”
}
```

8.2.2 Remote

远程登录与本地登录实际上是完全相同的想法，仍然以标准系统日志格式从 TMM 发送，仍然使用日志命令等；有一个主要区别：日志实际上并不存储在本地。相反，使用远程日志记录，您可以指定一个日志服务器 IP，该 IP 将接收直接从 TMM 发送的格式化日志语句。出于性能原因，这是一个更可取的选项，但要注意，如果在每个请求上发送可靠的日志信息，那么高流量的 BIG-IP 很容易使许多日志服务器不堪重负。

要使用 log 命令远程登录，您只需修改 log 命令以包含远程 IP 地址，如下所示：

```
when HTTP_REQUEST {
log local0 10.10.10.1 “Requested hostname: [HTTP::host] from IP:
[IP::local_addr]”
}
```

8.2.3 High Speed Logging

方式以极高的速度从 BIG-IP 发送数据的方法。顾名思义，这些命令最初设计用于日志记录，但实际上它们可用于发送几乎任何您想要的数据，因为通过 HSL 发送的数据没有以任何特定方式格式化。从日志记录的角度来看，HSL 背后的一般思想与通过日志命令进行远程日志记录相同，只是您可以使用 HSL 命令进行更多控制。

因为您专门使用 HSL::send 命令将数据直接发送到打开的 HSL 连接，所以您可以以任何格式发送您想要的任何数据。这也导致人们将 HSL 命令用于除严格记录信息之外的事情。您可以在此处阅读有关特定 HSL 命令的更多信息，但一个简单的示例如下所示：

```
when CLIENT_ACCEPTED {
set hsl [HSL::open -proto UDP -pool syslog_server_pool]
}
when HTTP_REQUEST {
# Log HTTP request as local7.info; see RFC 3164 Section 4.1.1
# "PRI Part" for more info
HSL::send $hsl " [IP::local_addr] [HTTP::uri]\n"
}
```

8.3 日志输出的开销？

是的。从技术上讲，您在 iRule 中执行的任何操作都是有成本的，这完全取决于您执行每个操作的频率，以及该操作的实际开销。虽然日志命令本身的开销很低，但在 TMM 执行其部分工作后其它处理开销会比较大。

iRules，以及几乎所有其他针对 BIG-IP 中线路上的流量运行的东西，都是从 TMM（流量管理微内核）中运行的。这些流程经过高度优化和调整以处理速度操作。告诉 TMM 触发日志命令并发送数据的 iRule 是低开销的，并且可以快速轻松地发生，但是一旦完成，消息就会发送到 syslog-ng。请记住，syslog-ng 由主机操作系统运行，这意味着它不由 TMM 处理。

任何时候你必须让主机操作系统处理大量日志，这可能是一个相当高开销的操作，因为主机操作系统没有像 TMM 处理大量流量的方式那样进行优化。最重要的是，您的日志条目将存储在磁盘上。如果访问主机上的日志，取决于磁盘 I/O，可能多次访问，对于每个请求，记录日志.....是个不好的方式。正因为如此，我们建议将日志记录用于开发和测试，但在将 iRules 投入生产时禁用或至少大大减少。当然，远程日志记录和 HSL 日志记录对 iRule 和 BIG-IP 的整体性能影响要小得多，因为需要开销的大部分，即通过系统日志实际将日志条目存储到磁盘，已完全不需要。如果您需要在高频的生产环境中进行实时日志记录，我强烈建议您尽可能使用远程日志记录或 HSL。

8.4 什么时候应该使用日志？

鉴于上面的讨论，我们什么时候应该使用日志呢，一般不建议在生产环境中使用大量的日志输出。在使用时最好清楚您想实现什么目的。以免对系统造成重大问题。

8.5 什么是注释？

注释是一个极其简单的概念。它是在不影响脚本本身的实际功能的情况下向 iRule（或任何脚本）添加文本的能力。重要的是要记住，其他人最终几乎不可避免地必须以某种方式处理您的代码。注释可能是可读、可用、易于维护的代码与除了原始开发人员之外对任何人都没有意义的混乱字符串之间的区别。

8.6 什么时候使用注释？

总是。只要你认为合适。注释是没有开销的，而且对您自己、工程师（包括在许多情况下试图提供帮助的 F5 工程师）以及任何未来查看您的代码的工程师或开发人员都非常有益。记录您的代码始终是一个好习惯，无论是以外部格式还是直接在线，注释是一种简单易行的方法。注释在开发和测试中也非常有用，可以比较两个命令或代码段，或者暂时删除给定的功能或逻辑。你当然可以很多注释，但这实际上是一个可读性问题。运用你的判断力，不要在你的代码中写一本书，但一般来说，我很少发现我喜欢的代码被过度注释了。当我使用了别人的 iRules 并试图理解它们时，情况就更是如此了。

8.7 iRules 故障排查-如何展开？

这是一个经常被问到的关于 iRules 的问题，最佳答案是.....就在这里。 iRules 没有正式的调试平台，因此日志记录和可靠的注释是您最好的朋友。通过正确注释您的代码并使用日志记录来确定正在发生的事情，从而使事情清晰易懂，同时进行故障排除是非常宝贵的。通常，在对行为问题进行故障排除时确定代码执行的深度是一种很好的做法，方法是将日志消息放置在不同的逻辑结构中，以查看正在匹配和执行哪些案例。从那里开始，检查命令和字符串格式的输出以缩小错误行为发生的范围。

我还强烈建议在 iRules 的开发和故障排除中使用 tclsh，从 11.1 开始，它在 BIG-IP 本身上可用。 Tclsh 是一个 tcl shell，它允许您本机执行 Tcl 命令并立即观察结果。与每次您想要对 iRules 进行小的更改并测试给定命令字符串的功能以确保输出结果时都必须重新生成流量相比，这可能非常有益并且可以节省大量时间。当然，tclsh 无法讲irules 添加到 Tcl 中，但涵盖了基础知识。

第九章、流量或者路由控制

本章简单介绍下流量控制的几种形式。

首先，请记住，BIG-IP 平台是一个完整的双向代理，这意味着我们可以检查任何方向（入口或出口）关联的业务的任何数据并进行控制。这意味着我们可以根据您的需要从技术上影响从客户端到服务器的流量路由，反之亦然。为了简单起见，并且因为它是最常见的用例，让我们将本文的重点放在只处理客户端路由上，例如发生客户端请求时发生的路由，以确定将流量传送到的目标服务器。

- pool
- node
- virtual
- HTTP::redirect
- reject
- drop
- discard

上述命令都可以控制流量的动作，比较常用的命令。

9.1 pool

使系统对指定池或池成员的流量进行负载平衡。如果只指定池成员，系统将负载均衡流量到指定的池成员，而不管池成员监控状态。池成员状态可以通过使用 LB::status 命令来确定。服务器响应失败可能会在 LB_FAILED 事件中被捕获。由于没有可用服务器而无法选择服务器可能会在 LB_FAILED 事件中被捕获或通过使用 active_members 命令在发送流量之前测试池中活动服务器的数量来阻止。）

可以有条件地选择池/成员。如果多个条件匹配，则最后一个匹配项将确定将此流量负载平衡到的池/成员。

语法：

```
pool <pool_name> [member <addr> [<port>]]
pool <pool_name> [member <addr>:<port>]

#v12 change for GTM pool - add CNAME
pool [CNAME] <poolName> [member <memberName>]
```

examples

条件匹配选择pool

```
#if 规则判断，pool 调用指定的 pool
when CLIENT_ACCEPTED {
  if { [IP::addr [IP::client_addr] equals 10.10.10.10] } {
     pool my_pool
 }
}
```

在发送流量之前检查池成员状态：

```
#在执行pool 命令之前，判断节点状态
when HTTP_REQUEST {
   if { [HTTP::uri] ends_with ".gif" } {
      if { [LB::status pool my_Pool member 10.1.2.200 80] eq "down"
} {
         log "Server $ip $port down!"
         pool fallback_Pool
      } else {
         pool my_Pool member 10.1.2.200 80
      }
   }
}
```

要捕获选定的池成员响应失败：

```
#LB_FAILED 一个不常用的功能，但是很有趣，请查看官方解释
when HTTP_REQUEST {
  if { [HTTP::uri] ends_with ".gif" } {
    pool my_Pool member 10.1.2.200 80
  }
}
when LB_FAILED {
  pool my_Pool
  LB::reselect
  log "Selected server [LB::server] did not respond. Re-selecting
node from myPool"
}
```

为了使此功能起作用，如果已经选择了池成员，则必须在池命令之前使用 **LB::reselect** 命令。

9.2 node

直接使用指定的服务器节点（IP 地址和端口号），从而绕过负载平衡策略。请记住，由于您没有将流量路由到 BIG-IP 统计信息中的对象，因此连接信息和状态将无法用于此连接。

如果在 LB::server 命令之前调用节点命令，LB::server 将返回空字符串，因为节点命令会覆盖任何先前的池选择逻辑。

语法：

```
node <addr> [<port>]
node <addr>:<port>
```

examples

```
when HTTP_REQUEST {
  if { [HTTP::uri] ends_with ".gif" } {
     node 10.1.2.200 80
  }
}
when LB_FAILED {
  log local0. "Selected server [LB::server] did not respond. Re-
selecting node from myPool"
  pool myPool
  LB::reselect
}
```

```
when HTTP_REQUEST {
 if { [HTTP::uri] ends_with ".gif" } {
     node "10.1.2.200:80"
  }
}

when LB_FAILED {
  log local0. "Selected server [LB::server] did not respond. Re-
selecting node from myPool"
  pool myPool
  LB::reselect
}
```

9.3 virtual

返回连接流经的关联虚拟服务器的名称。在 9.4.0 及更高版本中，它还可以用于将连接路由到另一个虚拟服务器，而无需离开 BIG-IP。需确保链接是在一个协议栈的，ipv4 不能到 ipv6。

语法:

```
virtual name
virtual [<name>]
```

examples

```
when HTTP_REQUEST {
  log local0. "Current virtual server name: [virtual name]"
}
```

```
when HTTP_REQUEST {
  # Send request to a new virtual server
  virtual my_post_processing_server
}
```

9.4 HTTP::redirect

将当前 HTTP 请求重定向到指定的 URL，或将当前响应替换为到指定 URL 的重定向。此命令立即向客户端发送重定向响应，因此您不能在处理单个 HTTP 请求（或响应）时多次调用此命令，也不能在您发出此命令后使用任何其他修改响应标头或内容的命令。

此命令始终发送 HTTP 302（临时重定向）状态代码。如果您想发送不同的代码，例如 301（永久重定向），您可以使用合适的状态代码和 Location 标头调用 HTTP::respond，例如：

```
HTTP::respond 301 Location https://[HTTP::host][HTTP::uri]
```

在某些情况下，您可能只想在请求到达服务器之前更改请求中的 URL。不需要通知客户端，也不会向用户显示更改，而且避免额外的网络往返将提高应用程序性能。要更改服务器看到的请求 (URI) 而无需重定向客户端，您可以调用 HTTP::path 或 HTTP::uri 命令。如有必要，您可以调用 HTTP::header 来替换 Host 标头：

```
HTTP::header Host newhost.example.com
```

语法：

```
HTTP::redirect <url>
```

examples

将 HTTP 请求或响应重定向到指定的 URL。

```
when HTTP_RESPONSE {
  if { [HTTP::status] == 404} {
    HTTP::redirect "http://www.example.com/newlocation.html"
  }
}
```

9.5 reject

导致连接被拒绝，返回适合协议的重置。当前 iRule 或 VS 上其他 iRule 中的当前事件中的后续代码仍会在发送重置之前执行。

如果 VS 使用的是 FastHTTP，拒绝命令将不起作用，至少在 11.3.0 下是这样。

语法:

```
reject
```

导致连接被拒绝，返回适合协议的重置。在 TCP 的情况下，客户端将收到一个设置了 RST 位的 TCP 段。在 UDP 的情况下，将生成 ICMP 不可达消息。

与流关联的系统连接表条目也被删除。

当前事件中的后续代码仍会在发送重置之前执行。如果有其他 iRule 具有相同的事件，则这些事件也将被执行。有关详细信息，请参见下面的第二个示例。

examples

```
when CLIENT_ACCEPTED {
  if { [TCP::local_port] != 443 }{
    reject
  }
}
```

显示同一事件中的第二个 iRule 中的代码如何在调用 reject 后仍然执行的示例

```
#priority nnn  nnn 数值越小优先级越高。you know
# Rule 1 on VS1
when HTTP_REQUEST priority 100 {
   # This event in this iRule runs first
   reject
   log local0. "Rejecting this request"
}
# Rule 2 on VS1
when HTTP_REQUEST priority 200 {
   # This event in this iRule runs second
   log local0. "This will still execute"
}
```

显示如何在调用 reject 后阻止执行同一事件中的第二个 iRule 中的代码的示例

```
# Rule 1 on VS1
when HTTP_REQUEST priority 100 {
   # This event in this iRule runs first
   reject
   log local0. "rejected"

   # Disable future events in this or any other iRule on the
virtual server
   event disable all

   # Exit this event of this iRule immediately
   return

   # This never gets executed
   log local0. "not hit"
}

# Rule 2 on VS1
when HTTP_REQUEST priority 200 {
   # This event in this iRule runs second but won't be executed
   log local0. "This will still execute"
}
```

9.6 drop

从系统连接表中删除相应的条目。请注意，在 TCP 的情况下，如果后续段到达时未设置 SYN 位（并且未设置 ACK 位），则将发送 RST 作为结果。这是当没有系统连接条目时该类型的段到达时的标准行为。

将系统连接表中的相应条目标记为立即超时。当收割过程下次运行时，连接将被删除。在此之前，数据包将被流接受。这在高数据包速率时很明显。请注意，在 TCP 的情况下，如果后续段到达时未设置 SYN 位（并且未设置 ACK 位），则将发送 RST 作为结果。这是当没有系统连接条目时该类型的段到达时的标准行为。

语法：

```
drop
```

examples

```
when SERVER_CONNECTED {
  if { [IP::addr [IP::client_addr] equals 10.1.1.80] } {
     drop
     log local0. "connection drop from [IP::client_addr]"
  }
 }
```

9.7 discard

从系统连接表中删除相应的条目。请注意，在 TCP 的情况下，如果后续段到达时未设置 SYN 位（并且未设置 ACK 位），则将发送 RST 作为结果。这是当没有系统连接条目时该类型的段到达时的标准行为。

将系统连接表中的相应条目标记为立即超时。当收割过程下次运行时，连接将被删除。在此之前，数据包将被流接受。这在高数据包速率时很明显。请注意，在 TCP 的情况下，如果后续段到达时未设置 SYN 位（并且未设置 ACK 位），则将发送 RST 作为结果。这是当没有系统连接条目时该类型的段到达时的标准行为。

语法：

```
discard
```

examples

```
when SERVER_CONNECTED {
  if { [IP::addr [IP::client_addr] equals 10.1.1.80] } {
     discard
     log local0. "connection discarded from [IP::client_addr]"
  }
 }
```

第二节、中级部分

第一章、字符串处理

字符串操作是 iRules 的重要组成部分，实际上也是我们选择 Tcl 以及许多其他语言的重要组成部分。字符串操作在很多方面、很多地方都很有用。无论是为入站 HTTP 请求重写 URI，还是解析部分 TCP 有效负载并调整bit以读取不同的内容，或者可能只是确定要用于某种目的的字符串的前 n 个字符.....

事实上，在 Tcl 眼中，一切都是字符串，因此有许多强大的工具，您可以使用它们相对轻松地按照您的需要修改字符串，并且效果很好。涵盖所有选项将是一个巨大的工程，但我们将在这里复习基础知识，即 iRules 中最常见的内容，您可以随意研究更晦涩的内容。tcl公开可用的文档适用于大多数相关命令。因此，在本文中，我们将介绍什么是字符串以及您应该关心的原因，以及您最有可能使用的大部分字符串命令。

1.1 什么是字符串？

字符串是一种特殊的数据类型，通常被理解为字符序列，或者作为文字常量，或者以变量形式表示。这意味着基本上任何东西都可以是字符串。名称、IP 地址、URL.....所有这些都是字符串，尤其是在 Tcl 中。一般而言，除非特定类型为整数或某种其他数据类型，否则可以安全地假设它们是字符串。话虽如此，由于 Tcl 不是静态类型语言，因此不允许您明确指定数据类型，因此除了少数特定条件外，它会将所有内容都视为字符串。这对 iRules 来说是一件好事，因为这意味着数据类型不会有太多麻烦，而且您通常可以轻松地以字符串格式操作事物。这意味着更少的编程麻烦，更多地获得您想要的效果。

1.2 常用字符串命令有哪些？

首先，Tcl 中用于处理字符串的最常见和广泛使用的命令非常简单，就是“string”。请注意，这个命令本身并没有什么用处。这个命令有很多很多排列，从改变字符串的大小写到只引用它的一部分，再到重新排序等等。使用这个单一命令和许多子命令，您可以在 iRules 中执行大部分字符串工作。所以问题是哪些“字符串”子命令最常用？

在 Tcl 中，您可以用一个字符串做很多事情，在这里我不将它们全部列出，而且描述起来也没有意义。因此，我将尽力列出一些似乎经常出现在 iRules 中的字符串命令，并讨论每个命令的作用。有关字符串命令和其他 Tcl 基本命令的完整参考，您可以在此处在线找到官方 Tcl 文档 (http://www.tcl.tk/man/tcl8.4/TclCmd/contents.htm)

1.2.1 string tolower

毫无疑问，iRules 中最常见和广泛使用的字符串命令也是最简单的命令之一。 tolower 命令的功能与它听起来的差不多。它将整个字符串的内容转换为小写。意思是，如果您有一个名为 $uri 的变量并且内容是“/Admin/WebAccess”，您可以在执行比较时运行 string tolower 命令以获得不同的结果。

语法:

```
string tolower string ?first? ?last?
```

examples

```
set uri “/Admin/WebAccess”
log local0. “Uri : $uri”
log local0. “Lower Uri: [string tolower $uri]”
```

or

```
% set uri “/Admin/WebAccess”
“/Admin/WebAccess”
% set str  [string tolower $uri]
“/admin/webaccess”
% puts $str
“/admin/webaccess”
%
```

第一个示例将导致第一条日志消息为“Uri: /Admin/WebAccess”，第二条日志消息为“Lower Uri: /admin/webaccess”。这要归功于 string tolower 命令。为什么这在 iRules 中如此有用？因为任何时候你执行基于字符串的比较，重要的是要确保你在相同的情况下比较事物。考虑比较主机名、URI 等，您可能会突然明白为什么这个简单的命令有如此大的价值。这对于数据组之类的事情变得越来越重要，您可以在其中将单个值与广泛的键值进行比较。能够确保它们都在正确的大小写中，然后将传入的比较值强制设置为该大小写非常有用。

请记住，与大多数其他字符串命令一样，这实际上并不修改字符串本身。如果您采用我们上面的示例，其中我们提供了小写 URI 并再次引用 $uri，它仍将保持原始大小写不变。

```
when HTTP_REQUEST {
if {([HTTP::uri] starts_with “/admin”) || ([HTTP::uri] starts_with
“/Admin”)} {
pool auth_pool
}
}
```

转换:

```
when HTTP_REQUEST {
if {[string tolower [HTTP::uri]] starts_with “/admin”} {
pool auth_pool
}
}
```

1.2.2 string length

返回一个十进制字符串，给出字符串中的字符数。请注意，这不一定与用于存储字符串的字节数相同。如果该值是一个字节数组值（例如从读取二进制编码通道返回的值），那么这将返回该值的实际字节长度。

正如您所期望的那样，字符串长度命令返回所讨论字符串的长度。这可以用于许多不同的事情，但到目前为止在 iRules 中观察到的最常见的用例可能是确定给定的命令是否返回了正确的结果。例如：

语法:

```
string length string
```

examples

```
when HTTP_REQUEST {
set cookie_val [HTTP::cookie “x-my-cookie”]
if {[string length $cookie_val > 1} {
log local0. “cookie was passed properly”
pool http_pool
}
}
```

当然，有很多方法可以执行类似的检查，如果您要做的只是确定命令是否返回 null，有些方法甚至更有效，但如果您想检查是否有特定的答案设置为至少 n 个字符，或者对于我见过的其他一些非常方便的用途，string length命令可能很方便。

1.2.3 string range

从字符串中返回一系列连续字符，从索引为第一个的字符开始，以索引为最后一个的字符结束。索引 0 指的是字符串的第一个字符。 first 和 last 可以指定为 index 方法。如果 first 小于零，则将其视为零，如果 last 大于或等于字符串的长度，则将其视为结束。如果 first 大于 last，则返回一个空字符串。

字符串范围命令允许您引用给定字符串的特定部分并仅检索该特定范围的字符。这可能是字符 1-10、第一个字符到第三个字符，或者可能是第 15 个字符到字符串末尾。有许多不同的方法来引用字符串段并使用此命令划分内容，但结果是相同的。它返回您定义的字符串部分的值。

这在 iRules 中一次又一次地被证明是有用的，例如检索 URI 的部分，确保主机名以特定前缀开头，或许多其他看似简单的要求。如果没有 string range 命令，那些任务将是一个令人头疼的问题。请注意，字符串中的第一个字符以 ID 0 而不是 1 开头。

语法：

```
string range string first last
```

examples

例如，如果您正在查看“/myApp?user=bob”的 URI，其中 bob 是一个可变用户名，并且您希望仅返回用户名，您有几个选项，但字符串范围使它相当简单的：

```
when HTTP_REQUEST {
set user [string range [HTTP::uri] 12 end]
log local0. “User: $user”
}
```

or

```
% set uri “/Admin/WebAccess”
“/Admin/WebAccess”
% set str  [string range $uri 0 6]
“/Admin
%
```

下一个示例显示从 HTTP::host 返回的值中删除非标准端口。注意 end-5 的使用，它将使用从零索引中的字符到字符串末尾短五位的字符的范围。

```
when HTTP_REQUEST {
  if { [HTTP::host] ends_with "8010" } {
    set http_host [string range [HTTP::host] 0 end-5]
    HTTP::redirect "https://$http_host[HTTP::uri]"
  }
}
```

or

```
% set uri “www.baidu.com:99987”
“www.baidu.com:99987”
% set str  [string range $uri 0 end-7]
“www.baidu.com
%
```

1.2.4 string map

根据映射中的键值对替换字符串中的子字符串。 mapping 是键值键值的列表.....就像数组get返回的形式一样。字符串中键的每个实例都将替换为其对应的值。如果指定了 -nocase，则不考虑大小写差异进行匹配。键和值都可以是多个字符。替换是按顺序进行的，因此首先检查列表中最先出现的键，依此类推。 string 仅迭代一次，因此较早的密钥替换不会影响以后的密钥匹配。

string range 允许您选择字符串的给定部分并返回它，string map 允许您实际修改内联子字符串。此外，字符串映射不是作用于字符的计数或范围，而是作用于实际的字符串。而对于字符串范围，您可能想要查找 URI 的特定部分，例如前 10 个字符，并查看它们是否与字符串匹配，或者基于它们进行路由或.....；使用字符串映射，您可以实时进行更改，将一个字符串更改为另一个字符串。

语法：

```
string map ?-nocase? mapping string
```

examples

例如，对于字符串范围，您可能有一个逻辑语句，如“使 URI 的前 10 个字符匹配 x”。您可以通过给出开始和结束字符来提供用于获取范围的字符串和所需的字符数。使用字符串映射，你会说“寻找任何看起来像 x 的字符串，并在给定字符串中将其更改为 y”，方法是提供要处理的字符串以及源字符串和目标字符串，意思是“全部更改” http 到 https 的案例”。

```
when HTTP_RESPONSE {
set new_uri [string map {http https} [HTTP::header “Location”]]
HTTP::header replace Location $new_uri
}
```

or

```
% set uri “http://www.baidu.com:99987/test?
http://test=http://123.com”
“http://www.baidu.com:99987/test?http://test=http://123.com”
% set str  [string map {http https} $uri]
“https://www.baidu.com:99987/test?https://test=https://123.com”
%
```

or

```
% string map {abc 1 ab 2 a 3 1 0} 1abcaababcababababc
01321221
```

详细过程:

String Map Multiple Key/Value Example

Mapping	Original String	Resulting String
1st (abc->1)	1abcaababcababababc	11aab1abab1
2nd (ab->2)	11aab1abab1	11a21221
3rd (a->3)	11a21221	11321221
4th (1->0)	11321221	01321221

请注意，对于第四个映射，返回的字符串是 01321221，而不是 00320220。这是为什么？好吧，该字符串仅迭代一次，因此较早的密钥替换不会影响以后的密钥匹配。

1.2.5 string first

string first 命令允许您识别给定子字符串在字符串中的第一次出现。这对于与字符串范围命令结合使用非常有用。例如，如果我想在 URI 中找到第一次出现的“/admin”并收集从该点到结尾的 URI，如果没有 string first 命令，将非常困难。如果我不知道确切的 URI 是什么怎么办？如果我不想收集“/admin”之前的 URI 的可变部分，但必须以某种方式考虑，即使它的长度是可变的，该怎么办？我不能只设置静态范围并单独使用字符串范围命令，所以我必须think 并组合命令。

语法：

```
string first string1 string2 ?startIndex?
```

examples

通过使用 string first 命令，如果我有一个类似于“/users/apps/bob/bobsapp?user=admin”的 URI，其中用户名是可变长度的，我无法确定URI 因为它，但我想检索传入的用户参数，我可以这样做：

```
set user [string range [HTTP::uri] [expr {[string first “user=”
[HTTP::uri]] + 5}] end]
```

or

```
% set uri "/users/apps/bob/bobsapp?user=admin"
/users/apps/bob/bobsapp?user=admin
%
% set user [string range $uri [expr {[string first "user=" $uri] +
5}] end]
admin
%
```

上面所做的是在 URI 中找到第一次出现的“user=”并返回第一个字符的索引。然后加上 5，因为那是字符串“user=”的长度，我们想引用“user=”之后的内容，不包括它，然后取字符串从该点到末尾的范围，并将其作为传入的用户名的值返回。它看起来有点复杂，但如果你将它逐个命令分解，你实际上只是将几个较小的命令串在一起以获得你想要的功能。现在您可以开始了解为什么字符串命令在 iRules 中如此重要和强大。

1.2.6 string last

在 haystackString 中搜索与 needleString 中的字符完全匹配的字符序列。如果找到，则返回 haystackString 中最后一个此类匹配项中第一个字符的索引。如果没有匹配，则返回 -1。如果指定了 lastIndex（以 STRING INDICES 中描述的任何形式），则搜索将仅考虑 haystackString 中位于指定 lastIndex 或之前的字符。例如，

语法：

```
string last needleString haystackString ?lastIndex?
```

examples

类似于 string first，仅返回 haystackString 中最后一个此类匹配中的第一个字符的索引。如果没有匹配，则返回 -1。如果指定了 lastIndex，则搜索将仅考虑 haystackString 中位于指定的 lastIndex 或之前的字符。在此示例中，（结合字符串范围，如果以 URI 中最后一个“/”开头并以 URI 的最后一个字符结尾的字符串包含“.”，则条件返回 true。

```
when HTTP_REQUEST {
  if {[matchclass [HTTP::uri] ends_with $::unmc_extends] or
  [matchclass [HTTP::method] equals $::unmc_methods] or
  [matchclass [HTTP::uri] contains $::unmc_sql] or
  [matchclass [IP::client_addr] equals $::unmc_restrict_ips]}{
    discard
  } elseif {
  ([HTTP::uri] ends_with "/") or
([string range [string last / [HTTP::uri]] end] contains ".") or
  ([HTTP::uri] contains "unmcintranet")}{
    pool unmc-intranet-proxy
  } elseif {[HTTP::uri] contains "google"}{
    HTTP::redirect "http://[HTTP::host]
[HTTP::uri]&restrict=unmcintranet"
  } else {
    HTTP::redirect "http://[HTTP::host][HTTP::uri]/"
  }
}
```

or

```
% string last a 0a23456789abcdef 15
10
```

1.2.7 **string trim**

返回一个等于 string 的值，除了删除 chars 给出的字符串中存在的任何前导或尾随字符。如果未指定字符，则删除空格（字符串为空格的任何字符返回 1 和“”）。

语法:

```
string trim string ?chars?
```

examples

同样，string trim 命令名副其实，它允许您操作给定字符串的开头和结尾以剪掉不需要的字符。也许您有一个字符串需要确保不以空格开头或结尾，或者您正在查看 URI 比较并且需要确保在某些情况下没有尾部斜杠，而在其他情况下则没有。不管怎样，string trim 命令让这一切变得简单。您所要做的就是指定要确保删除的字符，刚才提到的示例中的空格或斜杠，您将被设置为确保标准化比较。例如，如果你想确保某人正在向你域的管理部分发出请求，你可以确定他们将在 URI 的开头有一个斜杠，但他们可能会或可能不会包含尾部斜杠.您可以使用 starts_with 比较运算符来忽略它，但这也会忽略它们可能包含在特定字符串之后的任何其他内容。如果您想要“/admin”或“/admin/”的精确匹配，您可以使用或，或者可以修剪 URI，如下所示：

```
% set uri "/users/apps/bob/bobsapp?user=admin/"
/users/apps/bob/bobsapp?user=admin/
%
% set user [string trim $uri "/"]
users/apps/bob/bobsapp?user=admin
%
```

or

```
when HTTP_REQUEST {
if{[string trim [HTTP::uri] “/”] eq “admin”} {
pool admin_pool
}
}
```

or

```
% set uri "/users/apps/bob/bobsapp?user=admin/"
/users/apps/bob/bobsapp?user=admin/
% set user [string  trimright $uri "/"]
/users/apps/bob/bobsapp?user=admin
%
```

请注意，通过使用 trim 命令，它删除了前面和结尾的斜杠。还有 trimright 和 trimleft 版本，如果需要的话，只修剪字符串的一端。

第二章、列表处理

本章介绍变量的列表对象类型。大多数用于操作列表的 TCL 命令在 iRules 中可用：

- list - 创建列表
- split - 将字符串拆分为适当的 Tcl 列表
- join - 通过将列表元素连接在一起创建一个字符串
- concat - 将列表连接在一起
- llength - 计算列表中元素的数量
- lindex - 从列表中检索元素
- lrange - 返回列表中的一个或多个相邻元素
- linsert - 将元素插入列表
- lreplace - 用新元素替换列表中的元素
- lsort - 对列表的元素进行排序
- lsearch - 查看列表是否包含特定元素

其余tcl 列表处理命令 lrepeat、lreverse 和 dict 在 iRules 中不可用。

2.1 list

此命令返回包含所有参数的列表，如果未指定参数，则返回空字符串。根据需要添加大括号和反斜杠，以便可以在结果上使用 lindex 命令以重新提取原始参数，并且还可以使用 eval 来执行结果列表，其中 arg1 包含命令的名称，另一个args 包含它的参数。 List 产生的结果与 concat 略有不同：concat 在形成列表之前删除了一层分组，而 list 直接从原始参数开始工作。

语法：

```
list ?arg arg ...?
```

examples

```
% list a b "c d e  " "  f {g h}"
a b {c d e  } {  f {g h}}
%
```

2.2 split

返回通过在 splitChars 参数中的每个字符处拆分字符串创建的列表。结果列表的每个元素将由位于 splitChars 中字符实例之间的字符串中的字符组成。如果字符串包含 splitChars 中的相邻字符，或者如果字符串的第一个或最后一个字符在 splitChars 中，将生成空列表元素。如果 splitChars 是空字符串，则字符串的每个字符都会成为结果列表的一个单独元素。 SplitChars 默认为标准空白字符。

语法：

```
split string ?splitChars?
```

examples

SplitChars 默认为标准空白字符。在此示例中，split 用于将从格式化时钟命令返回的小时/分钟/秒分隔为单独的列表元素：

```
% set caTime [clock format [clock seconds] -format {%H:%M:%S}]
12:26:51
% set l [split $caTime :]
12 26 51
%
```

2.3 join

列表参数必须是有效的 Tcl 列表。此命令返回通过将列表的所有元素连接在一起形成的字符串，并用 joinString 分隔每对相邻的元素。 joinString 参数默认为空格字符。

语法：

```
join list ?joinString?
```

examples

制作一个逗号分隔的列表：

```
% set data {1 2 3 4 5}
1 2 3 4 5
% join $data ", "
1, 2, 3, 4, 5
```

在此示例中，用户正在使用 join 和“.”构建 IP 地址。作为 splitChar。

```
foreach num $IPtmp {
  lappend IP [expr ($num + 0x100) % 0x100]
}
set ::attr_value1 [join $IP .]
```

使用 TCL shell，输出如下：

```
% set uri "/users/apps/bob/bobsapp?user=admin/"
/users/apps/bob/bobsapp?user=admin/
% set user [string  trimright $uri "/"]
/users/apps/bob/bobsapp?user=admin
%
% set IPtmp { 0x8c 0xaf 0x55 0x44 }
 0x8c 0xaf 0x55 0x44
% foreach num $IPtmp { lappend IP [expr ($num + 0x100) % 0x100]}
% puts $IP
140 175 85 68
% set IP [join $IP .]
140.175.85.68
%
```

2.4 concat

此命令在修剪每个参数的前导和尾随空白后，将其每个参数与空格连接在一起。如果所有参数都是列表，这与将它们连接成一个列表具有相同的效果。它允许任意数量的参数；如果未提供任何参数，则结果为空字符串。

语法：

```
concat ?arg arg ...?
```

examples

这是一个很好的例子，它使用 LB::server 和虚拟名称的值连接 cookie 的内容

```
HTTP::cookie insert path / name ltmcookie value [concat [virtual
name] [LB::server]]
```

使用 TCL shell，需要变量来代替对 [virtual name] 和 [LB::server] 的调用：

```
% set virtual_name myVip
myVip
% set lb_server "myPool 10.10.10.10 443"
myPool 10.10.10.10 443
% set ltmcookie_value [concat $virtual_name $lb_server]
myVip myPool 10.10.10.10 443
```

2.5 lindex

从列表中检索元素

lindex 命令接受一个参数 list，它将其视为 Tcl 列表。它还接受列表中的零个或多个索引。索引可以在命令行上连续显示，也可以在 Tcl 列表中分组并作为单个参数显示。

语法：

```
lindex list ?index ...?
```

examples

```
% lindex {{a b c} {d e f} {g h i}} 2 1
h
%
```

这个例子来自与上面的 concat 例子相同的 iRule，显示了将列表元素提取到可用变量中：

```
#cookie 的值可以是个列表
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
Secure; HttpOnly
```

```
set vipid [lindex [HTTP::cookie ltmcookie] 0]
set poolid [lindex [HTTP::cookie ltmcookie] 1]
set serverid [lindex [HTTP::cookie ltmcookie] 2]
set portid [lindex [HTTP::cookie ltmcookie] 3]
```

使用我们在上面的 concat 示例中设置的 cookie 变量，我们现在可以提取列表项：

```
% set vipid [lindex $ltmcookie_value 0]
myVip
% set poolid [lindex $ltmcookie_value 1]
myPool
% set serverid [lindex $ltmcookie_value 2]
10.10.10.10
% set portid [lindex $ltmcookie_value 3]
443
```

最好避免变量，但仍然是一个很好的例证。通常你会看到 split 和 lindex 一起使用：

```
set trimID [lindex [split [HTTP::cookie "JSESSIONID"] "!" ] 0]
Stepping through that, you can see first the split occur, then the
indexing:
% set jsessionID_cookie
"zytPJpxVOTnpssqZZRLBgsVMLhGS6M2ZNMZ622yCNvpvOgkpTwzn!956498630!-34
852364"
zytPJpxVOTnpssqZZRLBgsVMLhGS6M2ZNMZ622yCNvpvOgkpTwzn!956498630!-348
52364
% set l [split $jsessionID_cookie !]
zytPJpxVOTnpssqZZRLBgsVMLhGS6M2ZNMZ622yCNvpvOgkpTwzn 956498630
-34852364
% set jsessID [lindex $l 0]
zytPJpxVOTnpssqZZRLBgsVMLhGS6M2ZNMZ622yCNvpvOgkpTwzn
```

鉴于此，您应该查找自定义 iRules 命令 getfield，这是一个 lindex/split 快捷方式。

第三章、catch

catch 命令可用于防止错误中止命令解释。 catch 命令递归调用 Tcl 解释器来执行脚本，并且总是在不引发错误的情况下返回，而不管在执行脚本时可能发生的任何错误。可以简单理解catch 是一个错误捕获命令，不会终止代码的执行。

如果脚本引发错误，catch 将返回一个非零整数值，对应于脚本评估返回的异常返回码。 Tcl 将脚本评估的正常返回代码定义为零 (0) 或 TCL_OK。 Tcl 还定义了四个异常返回代码：1 (TCL_ERROR)、2 (TCL_RETURN)、3 (TCL_BREAK) 和 4 (TCL_CONTINUE)。脚本评估期间的错误由返回代码 TCL_ERROR 指示。其他异常返回代码由 return、break 和 continue 命令以及在记录的其他特殊情况下返回。 Tcl 包也可以定义返回其他整数值作为返回码的新命令，并且使用 return -code 命令的脚本也可以具有 Tcl 定义的五个以外的返回码。

3.1 pool command

此命令用于将连接分配给指定的池名称（以及可选的特定池成员）。 pool 指定要将流量发送到的池。指定要将流量直接发送到的池成员。通常，我们看到 iRules 是这样写的（使用硬编码的池名称）。

```
when HTTP_REQUEST {
  switch -glob [HTTP::uri] {
    "*.gif" -
    "*.jpg" -
    "*.png" {
      pool images_pool
    }
    "*.asp" {
      pool ms_app_pool
    }
    "*.jsp" {
      pool java_app_pool
    }
  }
}
```

创建此 iRule 时，将对池命令进行语法检查和验证，并且由于池名称参数是string，因此保存逻辑将验证给定的池是否存在于配置中。如果它们不存在，将发生保存错误，您必须在保存 iRule 之前配置池。在这种情况下，不会发生运行时异常（除非您从 iRule 下移除池）。

但是，假设您想要构建一个可以随时间扩展的更加动态的解决方案。对于此示例，假设您希望让您的客户端确定将其发送到哪个服务器池。这可能是服务器在 HTTP cookie 中指定池名称，或者只是将其作为 GET 参数附加到 URI 上。我们将在此示例中使用后面的场景。

对于此示例，我们将定义以下可以控制选择 pool 的 GET 参数。

```
pool=name
```

因此，示例 URI 可能是：

```
http://somedomain.com/somefile.ext?
param1=val1&pool=pool_name&pararm2=val2...
```

我们将查询 URI 的 GET 参数以搜索“pool”参数，提取 pool_name 值并将该变量用作 pool 命令中的参数。对于较新版本的 BIG-IP，可以使用 URI::query 命令来提取值，但此实现应该可以一直追溯到 BIG-IP v9.0。

```
when HTTP_REQUEST {
  set namevals [split [HTTP::query] "&"]
  set pool_name ""
  for {set i 0} {$i < [llength $namevals]} {incr i} {
    set params [split [lindex $namevals $i] "="]
    set name [lindex $params 0]
    set val [lindex $params 1]
    switch $name {
      "pool" {
         set pool_name $val
      }
    }
  }
  if { "" ne $pool_name } {
    pool $pool_name
  }
}
```

这个实现有什么问题？如果您的配置在 $pool_name 变量中$pool_name没有在配置文件中配置，则代码会报错。

3.2 catch

接着上述问题，这不是一个理想的解决方案，简单地使用 catch 命令可以避免运行时连接终止，并允许请求继续通过默认的服务器池。

语法：

```
catch script ?varName?
```

在上面的 catch 命令语法中，script 是要执行的代码，resultVarName 是保存错误或结果的变量。如果没有错误，catch 命令返回 0，如果有错误，则返回 1。

catch 命令可用于防止错误中止命令解释。它递归地调用 Tcl 解释器来执行脚本，并且总是在不引发错误的情况下返回，而不管在执行脚本时可能发生的任何错误。

如果脚本引发错误，catch 将返回一个非零整数值，对应于脚本评估返回的异常返回码。Tcl 将脚本评估的正常返回代码定义为零 (0) 或 TCL_OK。 Tcl 还定义了四个异常返回代码：1 (TCL_ERROR)、2 (TCL_RETURN)、3 (TCL_BREAK) 和 4 (TCL_CONTINUE)。脚本评估期间的错误由返回代码 TCL_ERROR 指示。其他异常返回代码由 return、break 和 continue 命令以及在记录的其他特殊情况下返回。 Tcl 包也可以定义返回其他整数值作为返回码的新命令，并且使用 return -code 命令的脚本也可以具有 Tcl 定义的五个以外的返回码。

如果给出了 varName 参数，则它命名的变量将设置为脚本评估的结果。当脚本的返回代码为 1 (TCL_ERROR) 时，存储在 varName 中的值是一条错误消息。当脚本的返回码为 0 (TCL_OK) 时，resultVarName 中存储的值是脚本返回的值。

catch 使用

下面的 iRule 与上面的相同，只是它在动态池分配中使用了异常处理。

```
when HTTP_REQUEST {
  set namevals [split [HTTP::query] "&"]
  set pool_name ""
  for {set i 0} {$i < [llength $namevals]} {incr i} {
    set params [split [lindex $namevals $i] "="]
    set name [lindex $params 0]
    set val [lindex $params 1]
    switch $name {
      "pool" {
         set pool_name $val
      }
    }
```

```
  }
  if { "" ne $pool_name } {
  #使用catch 捕获异常，不会因为错误导致代码中断执行
    if { [catch { pool $pool_name } ] } {
      log local0. "ERROR: Attempting to assign traffic to non-
existant pool $pool_name"
      pool default_pool
    }
  }
}
```

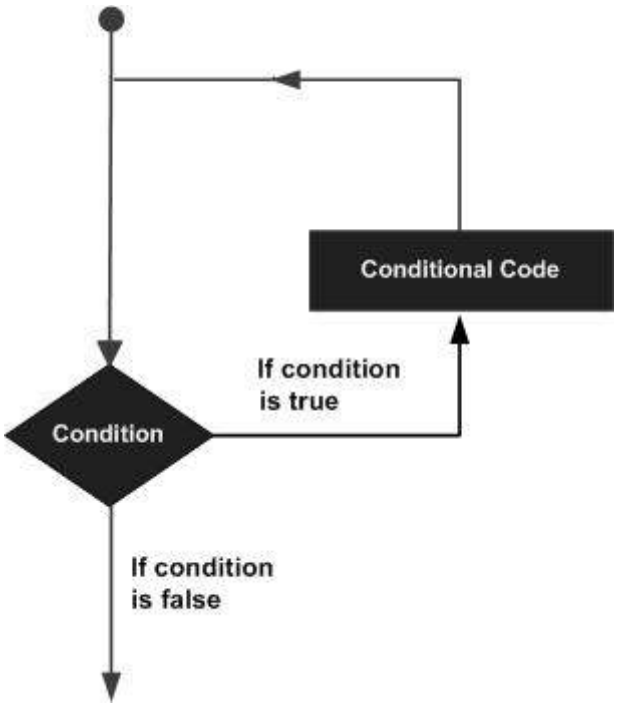
现在，连接指向哪个池的控制权完全掌握在请求 URI 的手中。在极少数情况下，您的应用程序逻辑在网络配置发生之前发生变化，连接将回退到默认服务器池，同时将问题记录到系统日志中以供以后检查。

使用动态变量的能力极大地增强了您可以使用 iRules 命令执行的操作的灵活性。 catch 命令应该围绕任何使用动态变量捕获任何运行时错误的 iRule 命令使用。事实上，如果您愿意，您可以嵌套 catch 命令以获得多层异常处理解决方案，但我将把它留到以后说。

第四章、循环

可能存在一种情况，您需要多次执行一段代码。通常，语句是按顺序执行的：函数中的第一条语句首先执行，然后是第二条，依此类推。

编程语言提供各种控制结构，允许更复杂的执行路径。循环语句允许我们多次执行一条语句或一组语句，以下是大多数编程语言中循环语句的一般形式 -



Tcl语言提供了以下类型的循环来处理循环需求。

SR.NO.	LOOP TYPE & DESCRIPTION
1	while loop:当给定条件为真时，重复一个语句或一组语句。它在执行循环体之前测试条件。
2	for loop: 多次执行一系列语句并缩写管理循环变量的代码。
3	nested loops: 嵌套循环，您可以在任何另一个 while、for 或 do..while 循环中使用一个或多个循环。

4.1 For & While Loops

在第一个示例中，我们使用 for 和 while 循环。这些循环是最常见的循环结构中的两种，并且在很多很多语言中都有使用。虽然这些示例确实有效，但您会注意到需要执行许多步骤才能达到根据相关数据实际执行操作的程度。这些是许多语言中的首选循环，因为它们相对健壮且功能强大，并且开销的差异不太重要。但这对于 iRules 来说有点贵，除非有实际需要。在这种情况下使用 for 还是 while 区别不是很大，但 for 循环用的多一点。

```
when HTTP_REQUEST {
  set domains {bob.com ted.com domain.com}
  set countDomains [llength $domains]

  #for loop
  for {set i 0} { $i < $countDomains } {incr i} {
    set domain [lindex $domains $i]
    # This is slower
  }

  #while loop
  set i 0
  while { $i < $countDomains } {
    set domain [lindex $domains $i]
    incr i
  }
}
```

4.2 循环控制语句

编号	控制声明和描述
1	break statement: 中断语句终止循环或 switch 语句并将执行转移到紧跟在循环或 switch 之后的语句。
2	continue statement:继续声明使循环跳过其主体的其余部分并在重复之前立即重新测试其条件。

考虑到 TMM 的单线程，如无必要不建议使用。

4.3 Foreach Loops

下一个例子在逻辑上是相似的。在这种情况下，它完成了相同的任务，而且开销更少。看看我们如何设置更少的变量，以及为每个循环迭代执行更少的函数？虽然 foreach 有时被认为不如 for 循环强大或灵活，但它的效率明显更高。因此，在可以完成手头任务的情况下，它是迄今为止首选的结构。

```
when HTTP_REQUEST {
  set domains {bob.com ted.com domain.com}
  foreach domain $domains {
  # This is faster, and more elegant
  }
}
```

第五章、Data-Groups

数据组在编写 iRule 时很有用。数据组只是一组相关元素，例如 AOL 客户端的一组 IP 地址。当您指定一个数据组以及类匹配命令或包含运算符时，您无需将多个值列为 iRule 表达式中的参数。

您可以定义三种类型的数据组：地址、整数和字符串。

5.1 什么是 data-group ?

data group 是 iRules 中一种特殊类型的内存结构。它实际上是一个键 -> 值对格式的列表，由 IP 地址/子网、字符串或整数组成。在这个独特的结构中，您可以只拥有一个键列表，或一个匹配键和值的列表。它在几个方面是独特和特殊的，但其中最明显的是它实际上作为配置的一部分永久存储，而不是仅作为 iRule 的一部分或内存中存在。这可以在 big-ip.conf 中内联完成，也可以作为单独的文件（称为外部数据组）完成。出于我们此处讨论的目的，两者的功能实际上相同。因为它存储在盒子上，而不仅仅是在内存中，所以在 iRule 执行之前，数据组会预先填充数据。这使它们成为任何类型的存储列表的理想选择，例如授权 IP 范围或 URI 重定向映射等。这也意味着它们在集群中的所有设备（或成对的两个设备）中被镜像，因为它们实际上是一个配置对象。由于可以通过 CLI 或 GUI 直接访问，因此修改数据组也很简单。您可以添加、修改或删除数据组中的条目，而无需触及引用它的 iRule(s)，这使得它非常适合存储可能需要不时更新的配置位或静态信息。只要在编辑时数据的格式保持正确，就不可能通过修改数据组来破坏 iRule，因为代码本身没有被变更。为此，有两种存储数据组的方法，内部或外部。对于内部数据组，数据集存储在 bigip.conf 文件中。对于外部数据组，它们在自己的文件中维护并从数据组对象中引用。非常大的数据集应该保存在外部数据组中。数据组作为配置对象的唯一（可能）限制因素是 iRules 不能直接影响配置对象。这意味着虽然您可以从 iRule 中读取、排序和引用数据组，但您实际上不能直接修改它们。如果您想更新数据组，则必须通过 CLI/GUI 手动完成，或者通过 TMSH 或 iControl 编写脚本。没有用于修改内容的直接 iRules 访问，使这些从 iRules 的角度来看是只读数据结构。配置对象看起来像这样（v11+，请参阅早期版本的 wiki 中的 class 命令）：

```
ltm data-group internal name_value_dg {
    records {
        name1 {
            data value1
        }
        name2 {
            data "value2 with spaces"
        }
    }
    type string
}
```

5.2 什么是**class ?**

class 与data group完全相同。多年来，我们一直互换使用这些术语，这让一些人感到懊恼和困惑。我们甚至称命令类，而结构称为“数据组”。不要让你感到困惑，它们是同一回事，无论你听到有人提到哪个，它们都在谈论我刚才描述的内存结构。

5.3 **data-group** 有什么好处？

我之前提到过，数据组的唯一缺点之一是它们对于涉及 iRules 的所有意图和目的都是只读的。然而，在考虑数据组的性能时，这在大多数情况下是一个小缺点.数据组无疑是最有效的内存结构，用于执行仅通过几个条目的查找。 if/else 和 switch 在某种程度上还不错，但是超过大约 10 个项目，即使是更高效的 switch 也无法跟上数据组查找的线性扩展。无论您存储 100 个还是 100,000 个条目，查询都大致相同，这要归功于数据组的索引和散列格式。这使它们成为存储大型数据列表以及可以以只读方式表示的频繁执行的查询的最佳选择。底线是：如果您不更新列表中的数据，并且您要处理的项目不止少数，无论是字符串、IP 还是其他，数据组可能是您最好的选择。它们在故障转移、重启等方面也具有弹性，这与没有基于磁盘的配置对象来备份它们的数据结构不同。最后，但在某些情况下并非最不重要，您可以通过 iControl 编写数据组管理脚本，尤其是外部数据组。 TMSH 等。这使其成为批量加载数据或在外部管理数据结构中的条目的理想方式。请记住，您必须重新实例化 iRule 才能识别更改。

5.4 该如何使用？

您将用来访问数据组的命令是class 命令。在 10 之前的版本中还有其他命令（matchclass/findclass），但是从 v10 开始，数据组被彻底修改以更有效和更高性能，并且 class 命令诞生了。它非常强大并且有很多排列。以下是 wiki 中的几个示例：

```
when HTTP_REQUEST {
if { [class match [IP::client_addr] equals localusers_dg ] } {
COMPRESS::disable
}
}
```

```
when HTTP_REQUEST {
set app_pool [class match -value -- [HTTP::uri] starts_with
app_class]
if {$app_pool ne ""} {
pool $app_pool
} else {
pool default_pool
}
}
```

第六章、逻辑验证

有时，iRule 会加载并运行而不会产生任何错误，但不会达到预期的结果。我将通过一个简单的示例概述根据实时流量检查 iRule 逻辑的基本过程：旨在执行双向 HTTP 主机标头修改的 iRule。

原始**iRules :**

```
#HTTP::header replace <name> [<string>] 将http header 中字段头名称为
name 的值替换为string

when HTTP_REQUEST {
 if { [HTTP::host] equals "easyname.domain.com"} {
 HTTP::header replace Location \
[string map -nocase {easyname.domain.com
long.internal.name.domain.com} [HTTP::header value Location]]
 }
}

when HTTP_RESPONSE {
 if { [HTTP::host] equals "long.internal.name.domain.com"} {
 HTTP::header replace Location \
[string map -nocase {long.internal.name.domain.com
easyname.domain.com} [HTTP::header value Location]]
 }
}
```

当流量违反规则运行时，日志中没有看到错误，客户端和服务器之间的流量正常流动，但没有执行预期的替换。

鉴于此类意外行为，验证您的逻辑的第一步是确保您的 iRule 看到并根据其采取行动的信息是您期望它看到并采取行动的信息。为此，您可以围绕条件决策添加一些日志记录。最佳做法是首先在每个决策点之前记录用于做出决策的变量、对象或命令的值，然后在每个触发点之后记录另一条消息以指示预期的代码块确实正在执行。

这是修改后的irules 规则，其中包含一些有关条件的信息记录：

```
when HTTP_REQUEST {
 # First we'll log the 2 header values used in the conditional code
block
 log local0. "Host = [HTTP::host]"
 log local0. "Location = [HTTP::header Location]"

 if { [HTTP::host] equals "easyname.domain.com"} {

 # inside the conditional block, add another log line saying that's
where you are
 log local0. "Host matched, performing replacement operation"
 HTTP::header replace Location \
"[string map -nocase {easyname.domain.com
long.internal.name.domain.com} [HTTP::header value Location]]"

 # you can even log the result of the replacement operation by
running it again with the log command
 log local0. "Replacement text = \
[string map -nocase {easyname.domain.com
long.internal.name.domain.com} [HTTP::header value Location]]"
 }
}

when HTTP_RESPONSE {
 # For the response, we'll again log the header values used in the
conditional code block
 log local0. "Host = [HTTP::host]"
 log local0. "Location = [HTTP::header Location]"

 if { [HTTP::host] equals "long.internal.name.domain.com"} {
 # inside the conditional block, add another log line saying that's
where you are
 log local0. "Host matched, performing replacement operation"
```

```
 HTTP::header replace Location \
"[string map -nocase {long.internal.name.domain.com
easyname.domain.com} [HTTP::header value Location]]"

 # and again, you can log the result of the replacement operation
 log local0. "Replacement text = \
[string map -nocase {long.internal.name.domain.com
easyname.domain.com} [HTTP::header value Location]]"
 }
}
```

HTTP_REQUEST日志:

```
HTTP_REQUEST: Host = easyname.domain.com
HTTP_REQUEST: Location =
HTTP_REQUEST: Host matched, performing replacement operation
HTTP_REQUEST: Replacement text =
```

HTTP_RESPONSE日志:

```
HTTP_RESPONSE: Host =
HTTP_RESPONSE: Location =
```

or

```
HTTP_RESPONSE: Host =
HTTP_RESPONSE: Location = http://long.internal.name.domain.com/uri
```

仔细查看双向发送的实际流量（使用 HTTPwatch、tcpdump 或其它替代跟踪工具），可以发现一些有趣且相关的细节：

1. 来自客户端的请求不包含 Location 标头，仅包含 Host 标头。 LTM 服务器端的请求同时具有 Host 和 Location 标头。
2. Response 不包含 Host 标头，仅包含用于重定向响应的 Location 标头。
3. 内部主机名在 HTTP 负载的超链接中清晰可见（不仅仅是 Host 和 Location 标头）。

最终的 iRule，包括所有正确的引用、操作和优化，并与流配置文件一起实现，如下所示：

```
when HTTP_REQUEST {
```

```
 if { [HTTP::host] equals "easyname.domain.com"} {
 # replace header completely if it matches
 HTTP::header replace Host "long.internal.name.domain.com"
 }
}

when HTTP_RESPONSE {
 if { [HTTP::status] starts_with "3" }{
 # replace the Location header only if the response is a redirect,
 # since no other HTTP server responses contain the hostname in a
header.
 HTTP::header replace Location \
[string map -nocase {long.internal.name.domain.com
easyname.domain.com} [HTTP::header value Location]]
 # depend on stream profile to perform the hostname replacements in
the HTTP payload
 }
}
```

第七章、HSL - 如何使用？

High Speed Logging 自 10.1 版以来一直存在，并且在过去几年中已成为许多项目不可或缺的一部分。在引入 HSL 之前，远程日志记录完全在 syslog 中配置，或者可以通过在日志语句中指定目标在 iRules 中处理。 HSL 对该场景的一项增强是允许为目的地配置服务器池，因此给定服务器池，日志消息肯定会到达某个地方（好吧，对于 TCP，它们肯定会到达！）缺点然而，使用 log 或 HSL::send 命令时，消息只会到达一个目的地。该问题的解决方法是使用尽可能多发送给多个日志服务器。

创建**publisher**

从版本 11.3 开始，向 HSL::open 命令添加了一个新选项，允许您将数据发送到日志 publisher，而不仅仅是发送到池。这允许您将该数据多播到任意数量的服务器。在我的测试设置中，我使用 linux 虚拟机上的别名接口作为目的地，并为每个接口创建了一个池以添加到publisher：

```
ltm pool lp1 {
    members {
        192.168.101.20:514 {
            address 192.168.101.20
        }
```

```
    }
}
ltm pool lp2 {
    members {
        192.168.101.21:514 {
            address 192.168.101.21
        }
    }
}
ltm pool lp3 {
    members {
        192.168.101.22:514 {
            address 192.168.101.22
        }
    }
```

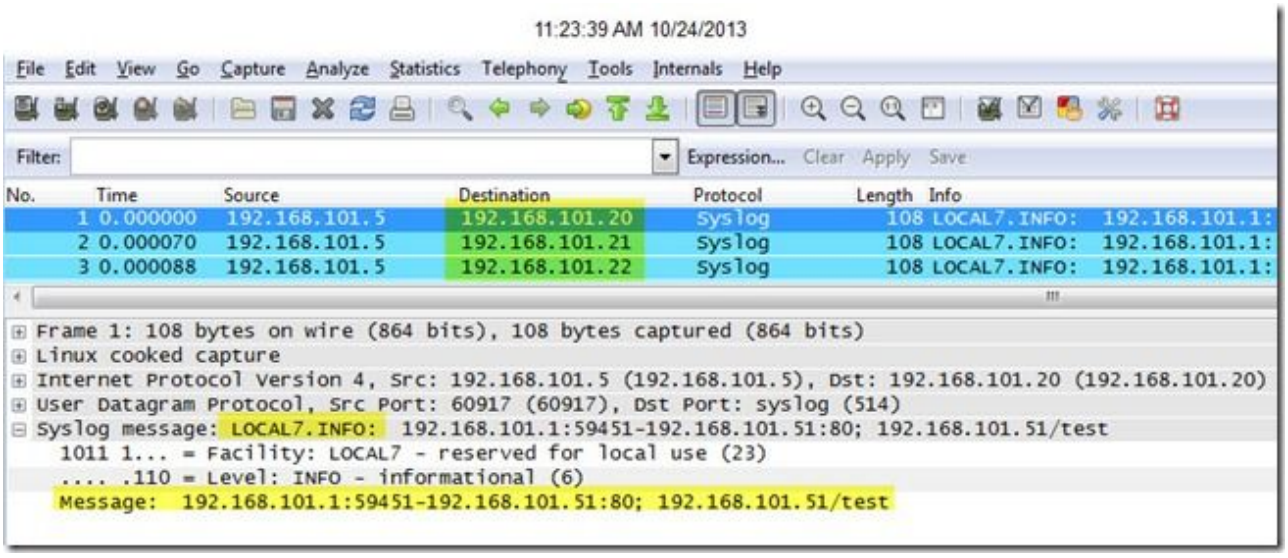
定义池后，我将创建日志目标：

```
sys log-config destination remote-high-speed-log lp1 {
    pool-name lp1
    protocol udp
}
sys log-config destination remote-high-speed-log lp2 {
    pool-name lp2
    protocol udp
}
sys log-config destination remote-high-speed-log lp3 {
    pool-name lp3
    protocol udp
}
```

最后，我创建了用于 iRules 的publisher：

这就是在 HSL::open 中展示 -publisher 选项的 iRules 所需方式：

```
ltm rule testrule {
  when CLIENT_ACCEPTED {
    set lpAll [HSL::open -publisher /Common/lpAll]
  }
  when HTTP_REQUEST {
    HSL::send $lpAll "<190> [IP::client_addr]:[TCP::client_port]-
[IP::local_addr]:[TCP::local_port]; [HTTP::host][HTTP::uri]"
  }
}
```



三个日志服务器都接收到了日志。

第八章、性能评估

“我想知道编写更精细的 iRules 会对 F5 的内存和处理器产生什么影响。有没有办法预测给定规则和流量负载对 F5 资源的影响？”

在本文中，我将向您展示如何收集和解释 iRule 运行时统计数据以确定规则的一个版本是否比另一个版本更有效，或者测试在 LTM 上运行的 iRule 理论上可能的最大连接数。

8.1 收集统计数据

要生成和收集运行时统计信息，您可以将命令“timing on”插入到您的 iRule 中。当您在启用计时的情况下通过 iRule 运行流量时，LTM 将跟踪评估每个 iRule 事件所花费的 CPU 周期数。您可以为整个 iRule 或仅为特定事件启用规则计时。

要为整个 iRule 启用计时，请在规则顶部的第一个“when EVENT_NAME”子句之前插入“timing on”命令。以下是为规则中的所有事件启用计时的示例：

```
rule my_fast_rule {
    timing on
    when HTTP_REQUEST {
       # Do some stuff
    }
 }
```

要仅为特定事件启用 iRule 计时，请在“when EVENT_NAME”声明和左花括号之间插入“计时”命令。下面是一个仅为特定事件启用计时的示例：

```
rule my_slow_rule {
    when HTTP_REQUEST timing on {
       # Do some other stuff
    }
 }
```

（有关通过打开和关闭计时来仅对选定事件计时的更多详细信息，请参阅计时命令文档。）

要获得每个事件的合理平均值，您需要在预期的生产负载下至少运行几千次 iRule 。对于 http 流量，我通常使用来自 BIG-IP 命令行的 apache bench (ab) 执行此操作，但您也可以使用 netcat 和其他 linux 工具来处理非 http 流量。

8.2 查看统计数据

您的 iRule 的统计数据（以 CPU 周期衡量）可以在命令行或控制台通过运行来查看

```
tmsh ltm rule rule_name show all
```

输出包括执行、失败和中止的总数，以及自上次清除统计信息以来每个事件消耗的最小、平均和最大周期。

例如:

```
 RULE rule_name
 +-> HTTP_REQUEST 729 total 0 fail 0 abort
 | Cycles (min, avg, max) = (3693, 3959, 53936)
```

如果您使用 iRules Editor ，则可以在“属性”对话框的“统计信息”选项卡上查看相同的统计信息。（双击左窗格中的 iRule 以打开“属性”对话框。）

第三节、高级部分

本节讨论iRules 高级使用方法。包括会话保持功能的高级使用、字符解析、正则表达式等功能.....

第一章、Tables

本章讨论 session 和tables 两个全局数据共享高级功能，主要解决跨CMP 的数据共享问题和 Weblogic JSessionID Persistence 是有区别的。

1.1 session

如果你有一些需要存储的数据，这些数据需要在多个连接中使用。您需要它既高效又快速，并且不想对数据结构进行大量复杂的管理，session 命令是一个比较好的解决方案。

语法:

```
session add <mode> <key> <data> [<timeout>]
session lookup <mode> <key>
session delete <mode> <key>

   <mode> = simple | source_addr | sticky | dest_addr | ssl | uie |
hash | sip
   <key> = <mode specific value> | { <value> [any virtual | service
| pool] [pool <name>] }
     the latter key specification is used to delete persistence
entries regardless of virtual, service, or pool association.

  <timeout> = The timeout in seconds. Defaults to 180 seconds.  If
the session key is touched (updated or looked up), the timeout
counter starts over again.
```

```
#将用户的数据存储在指定持久化模式的指定键下
session add <mode> <key> <data> [<timeout>]
#返回以前使用会话添加存储的用户数据。如果查找键为空字符串，将触发运行时 TCL 错误并重置连接。因此，最佳做法是在尝试会话查找之前显式检查空键。
session lookup <mode> <key>
#删除以前使用会话添加存储的用户数据
session delete <mode> <key>
#session 支持的类型，tables 不限制
<mode> = simple | source_addr | sticky | dest_addr | ssl | uie | hash | sip
```

examples

ssl握手后将客户端证书保存在会话表中，以便在后续请求期间检索：

```
when CLIENTSSL_CLIENTCERT {
  # 在ssl 握手中定义证书变量，并保存在session 表中
  set ssl_cert [SSL::cert 0]
  session add ssl [SSL::sessionid] $ssl_cert 180
}
when HTTP_REQUEST {
  # 检索会话的证书信息
  set ssl_cert [session lookup ssl [SSL::sessionid]]
}
```

1.2 tables

table 有点类似于data-group ，因为它的核心是包含列表的内存结构。表单独存储在内存中，这意味着它们没有像data-group 那样限制对象类型，但这也使tables 在定义上更加灵活。

table 是跨连接维护的读/写内存结构，使其成为存储需要持久数据的好方法，而不是像普通 iRules 变量那样基于连接。表还允许一些非常强大的概念，即 sub tables和内置的 timeout/lifetime 周期结构。这两件事都增加了表格功能丰富性。

table也非常高效，很像data-group，尽管由于它们更灵活的性质，表的规模略小。table利用 LTM 中的会话表，该table旨在处理进出设备的每个连接，因此具有非常高的性能。table查询共享这种性能，并且可以是一种快速构建简单、持久内存结构或更复杂的接近数据库的方法，如sub-tables，具体取决于具体需求。

1.2.1 table 的优势？

table 是一种高性能、高度可扩展、持久（非连接绑定）、读/写数据结构。仅这一特性就使它在 iRules 中独一无二并且极其强大。根本没有其他东西可以填补这个角色。只有几种方法可以跨连接读取/写入变量数据，而表几乎在所有情况下都是迄今为止最好的选择。

在您的 iRule 中创建相当于内存中的迷你数据库的能力在许多场景中也非常有用。这很容易通过子表来完成。您不仅可以创建平面列表，还可以创建命名列表以根据需要分隔数据。

现在添加一个事实，即有可配置的超时和生命周期选项，这意味着您永远不必担心内存管理或表的内存清理。

1.2.2 如何使用？

语法：

table 命令（类似于 session 命令）是一种访问会话表的方法。 table 命令是 session 命令的超集，具有针对一般用途的改进语法。详细使用请看表命令系列文章。

```
table set      [-notouch] [-subtable <name> | -georedundancy] [-
mustexist|-excl] <key> <value> [<timeout> [<lifetime>]]
table add      [-notouch] [-subtable <name> | -georedundancy] <key>
<value> [<timeout> [<lifetime>]]
table replace  [-notouch] [-subtable <name> | -georedundancy] <key>
<value> [<timeout> [<lifetime>]]
table lookup   [-notouch] [-subtable <name> | -georedundancy] <key>
table incr     [-notouch] [-subtable <name> | -georedundancy] [-
mustexist] <key> [<delta>]
table append   [-notouch] [-subtable <name> | -georedundancy] [-
mustexist] <key>  <string>
table delete   [-subtable <name> | -georedundancy] <key>|-all
table timeout  [-subtable <name> | -georedundancy] [-remaining]
<key>
table timeout  [-subtable <name> | -georedundancy] <key> [<value>]
table lifetime [-subtable <name> | -georedundancy] [-remaining]
<key>
table lifetime [-subtable <name> | -georedundancy] <key> [<value>]
table keys -subtable <name> [-count|-notouch]
```

```
#在会话表或命名子表中设置键/值对，具有指定的超时和生存期。设置操作完成后返回条目的值。如果未指定超时，将使用默认值 180 秒。
table set
#要使用子表还是不使用子表来存储您的会话条目？简短的回答是，如果您需要能够计算键的数量或在一个命令中检索键，则仅使用子表。操作子表中的条目比操作不在子表中的条目具有更高的开销。每个子表本身也占用内存。给定子表中的所有条目都在同一个处理器上。
table set      [-notouch] [-subtable <name>
#与 table set -excl 完全相同
table add
与 table set -mustexist 完全相同
table replace
#在指定表（如果有）中查找与指定键关联的值。如果值未与指定键关联，则返回空值
table lookup
#
```

1.2.2.1 table 基础

首先我去掉table 表那个复杂的功能参数如下:

```
#在会话表中设置一个键/值对，并指定超时时间
table set $key $data 180
#在表中查找与指定键相关的值，返回data
table lookup $key
#删除指定表中带有指定键的键/值对。
table delete $key
```

通常想要跟踪一个客户端对于某个指定URI 的访问计数使用session 方法，部分代码:

```
set newpmbr [session lookup uie $key]
if { $newpmbr == {} } {
    session add uie $key $curpmbr
} else {
    pool [LB::server pool] member $newpmbr
}
```

问题: 由于现在有多个处理器同时工作，第二个连接可能在第一个连接做完查询之后，但在第一个连接添加条目之前做完查询。如果发生这种情况，两个连接都会认为他们是第一个连接，因此计数可能就会被修改一次，导致不同的逻辑判断。

为了解决这样的问题，我们增加了另外两种方法来向会话表添加数据。

```
#table set的作用就像session add一样：如果键在表中不存在，它就添加条目，如果它存在，它就覆盖那里的内容。
table set $key $data
#表添加不会覆盖现有的条目，它只会在键不存在的情况下添加一个条目。
table add $key $data
#table replace则相反：它只会更新一个现有的条目。如果键在会话表中不存在，它不会添加一个新的条目。
table replace $key $data
```

使用table 重写

```
set newpmbr [table add $key $curpmbr]
if { $newpmbr != $curpmbr } {
    pool [LB::server pool] member $newpmbr
}
```

1.2.2.3 进阶一

表结构:

Key	Value	Timeout	Lifetime	touch time	create time
10.1.2.3	UserFoo	180	indefinite	1253830994	1253830993
10.3.6.1	UserBar	Indefinite	3600	1253830995	1253830995
10.2.9.12	UserBaz	Indefinite	3600	1253830996	1253830996

```
table set [-excl|-mustexist] <key> <value> [<timeout> [<lifetime>]]
table add <key> <value> [<timeout> [<lifetime>]]
table replace <key> <value> [<timeout> [<lifetime>]]
```

子表:

子表的概念，它只是一个命名的条目集。它让你把你的条目组织成组，并让你对这些组采取操作。 一个子表本身基本上就是一个表，具有一些特殊能力。不过，子表只包含条目；你不能嵌套子表。下面是它们的样子。

The session table

Key	Value	Timeout	Lifetime	touch time	create time
10.1.2.3	UserFoo	180	indefinite	1253830993	1253830993

Subtable: DNS

Key	Value	Timeout	Lifetime	touch time	create time
User77	5254	3600	3600	1253830772	1253830719

Subtable: POST

Key	Value	Timeout	Lifetime	touch time	create time
admin	allowed	30	100	1253830990	1253830910

它们的使用很简单：每个当前的表子命令都需要一个-子表<名称>参数。

```
table set -subtable $tbl $key $data
table lookup -subtable $othertbl $key
table delete -subtable $thirdtbl $key
```

如果你想为每个用户 、IP、虚拟服务设置子表:

```
table set -subtable "$username" $key $data
table add -subtable "[IP::client_addr]" $key $data
table replace -subtable "[virtual]" $key $data
```

删除所有表:

```
table delete -all -subtable "$username"
```

设置相关时间:

```
table set <key> <value> [<timeout> [<lifetime>]]
```

table incr 分析:

```
table incr     [-notouch] [-subtable <name> | -georedundancy] [-
mustexist] <key> [<delta>]
```

注意事项:

- 不能在RULE_INIT 和其它全局事件中使用 table .
- 会话命令和表命令都作用于相同的数据。例如，如果你用session命令在session表中插入一个条目，那么你以后可以用table命令来查找这个数据（也许在另一个iRule中）。然而，session 命令不能以任何方式访问table 子表中的任何数据。
- 会话表被镜像到备用表上，这一点不能被禁用。
- 子表的开销比较大

examples

限制虚拟服务的访连接数

```
when CLIENT_ACCEPTED {
    #设置子表名变量
    set tbl "connlimit:[IP::client_addr]"
    #设置子表key 变量
    set key "[TCP::client_port]"
    #判断子表tbl key 的条目，也就是客户端数量
    if { [table keys -subtable $tbl -count] > 1000 } {
        #在这个连接上关闭CLIENT_CLOSED，减少关闭的开销，直接reject
        event CLIENT_CLOSED disable
        reject
    } else {
        #设置子表及键
        table set -subtable $tbl $key "ignored" 180
        设置一个延时
        set timer [after 60000 -periodic { table lookup -subtable
$tbl $key }]
    }
}
when CLIENT_CLOSED {
    #取消延时
    after cancel $timer
    #删除表键
    table delete -subtable $tbl $key
}
```

限制单个客户端的访连接数

```
# 限制单个客户端的访连接数20
when CLIENT_ACCEPTED {

   # 检查单个客户端连接数是否超过20
   if { [table keys -subtable connlimit:[IP::client_addr] -count]
>= 20 } {
      reject
   } else {

      #在单个connlimit:[IP::client_addr]子表中设置键值对，key=port
,value=null ,lifetime=180
      table set -subtable connlimit:[IP::client_addr]
[TCP::client_port] "" 180
   }
}

when CLIENT_CLOSED {
   # 当连接关闭时自动删除对应表项
   table delete -subtable connlimit:[IP::client_addr]
[TCP::client_port]
}
```

如果每秒进行超过 100 次 DNS 查询，则将 IP 列入黑名单 10 分钟：

```
when RULE_INIT {
    set static::maxquery 100
    set static::holdtime 600
}
when CLIENT_DATA {
    set srcip [IP::remote_addr]
    if { [table lookup -subtable "blacklist" $srcip] != "" } {
        drop
        return
    }
    set curtime [clock second]
    set key "count:$srcip:$curtime"
    set count [table incr $key]
    table lifetime $key 2
```

```
    if { $count > $static::maxquery } {
        table add -subtable "blacklist" $srcip "blocked" indef
$static::holdtime
        table delete $key
        drop
        return
    }
}
```

第二章、Scan

scan 命令用于根据格式指定符来解析一个字符串并赋值给变量并返回。它接收一个字符串，并根据你的格式参数，将匹配的内容存储在一个或多个变量中。它还会返回所执行的转换数量，所以它也可以在条件中使用。关于该命令中的所有选项，可以在这里找到scan man page，网址是http://tmml.sourceforge.net/doc/tcl/scan.html。

语法：

```
scan string format ?varName varName ...?
```

Options：

- d - 输入的子串必须是一个十进制的整数
- s - 输入的子串由所有的字符组成，直到下一个空格
- n - 不从输入字符串中消耗任何输入。相反，到目前为止，从输入字符串中扫描到的字符总数被存储在变量中。
- [chars] - 输入的子串由一个或多个字符组成，单位为chars。匹配的字符串被存储在变量中。
- [^chars] - 输入的子串由一个或多个不在chars中的字符组成。

examples

```
when HTTP_REQUEST {
  if { [scan [HTTP::host] {%[^:]:%s} host port] == 2 } {
    log local0. "Parsed \$host:\$port: $host:$port
  }
}
```

```
% set httphost "www.test.com:8080"
www.test.com:8080
% scan $httphost {%[^:]:%s} host port
2
% puts "$host $port"
www.test.com 8080
%
```

另一个用例--将一个IP地址分割成多个变量

```
% set ip 10.15.25.30
10.15.25.30
% scan $ip %d.%d.%d.%d ip1 ip2 ip3 ip4
4
% puts "$ip1 $ip2 $ip3 $ip4"
10 15 25 30
%
```

第三章、 Binary Scan

二进制扫描命令，就像这个高级iRules系列中涉及的扫描命令一样，解析字符串。只是，正如其形容词所示，它解析的是二进制字符串。在这篇文章中，我将强调命令的语法和下面的一些格式化字符串选项。请查看手册页，了解完整的格式选项列表。

语法:

```
binary scan string formatString ?varName varName … ?
```

选项

- c - 数据被转化为计数的8位有符号整数，并作为一个列表存储在相应的变量中。如果count是*，那么将扫描字符串中所有剩余的字节。如果count被省略，那么将扫描一个8位整数。
- S - 数据被解释为16位有符号的整数，以big-endian字节顺序表示。这些整数将作为一个列表存储在相应的变量中。如果count是*，那么将扫描字符串中所有剩余的字节。如果count被省略，那么将扫描一个16位的整数。
- H - 数据被转化为一个由十六进制数字组成的字符串，从高到低的顺序表示为一组"0123456789abcdef "的字符序列。数据字节按从头到尾的顺序进行扫描，每个字节中的十六进制数字按从高到低的顺序进行扫描。最后一个字节中的任何额外位都会被忽略。如果count是*，那么将扫描字符串中所有剩余的十六进制数字。如果count被省略，那么将扫描一个十六进制数字。

examples

```
when CLIENT_ACCEPTED {
  set lsec 0
  SSL::disable
  TCP::collect
}
when CLIENT_DATA {
  if { ! [info exists rlen] } {

  binary scan [TCP::payload] cSS rtype sslver rlen
  #log local0. "SSL Record Type $rtype, Version: $sslver, Record
Length: $rlen"

  # SSLv3 / TLSv1.0
  if { $sslver > 767 && $sslver < 770 } {
    if { $rtype != 22 } {
      log local0. "Client-Hello expected, Rejecting.  \
                        Src: [IP::client_addr]:[TCP::remote_port] -
> \
                        Dst: [IP::local_addr]:[TCP::local_port]"
      reject
      return
    }
```

第四章、正则表达式

正则表达式是由Stephen Kleene在20世纪50年代开发的，目的是指定一种正则语言。许多任务可以更简单地使用字符串和扫描而不是正则表达式来完成。正则表达式通常看起来像它所描述的字符串集可能采取的各种形式的总结。

正则表达式或简称为regex，本质上是一个字符串，用于描述或匹配一组字符串，根据某些语法规则。

正则表达式（"RE"）有两种基本类型：扩展RE（"ERE "s）和基本RE（"BRE "s）。对于unix的人来说，ERE与传统的egrep使用的大致相同，而BRE则与传统的ed大致相同。TCL对REs的实现增加了第三种，即高级REs（"AREs"），基本上是带有一些重要扩展的EREs。

4.1 正则表达式示例

那么，一个正则表达式是什么样子的呢？它可以像一串字符一样简单，用来搜索与 "abc "完全匹配的内容

```
{abc}
```

或者是一个内置的转义字符串，用于搜索一个字符串中所有非空格的序列

```
{\S+}
```

或一组字符范围，搜索所有三个小写字母的组合

```
{[a-z][a-z][a-z]}
```

甚至是一个代表信用卡号码的数字序列

```
{(?:3[4|7]\d{13})|(?:4\d{15})|(?:5[1-5]\d{14})|(?:6011\d{12})}
```

关于语法的更多信息，请参阅TCL文档中的re_syntax手册页https://www.tcl.tk/man/tcl8.4/TclCmd/re_syntax.html

4.2 支持正则表达式的命令

在TCL语言中，以下内置的命令支持正则表达式。

- regexp - 与字符串匹配正则表达式
- regsub - 在正则表达式模式匹配的基础上执行替换操作
- lsearch - 查看一个列表是否包含一个特定的元素
- switch - 根据一个给定的值，评估几个脚本中的一个。

iRules也有一个操作符，在 "if"、"matchclass "和 "findclass "等命令中进行正则表达式比较

```
matches_regex - 测试一个字符串是否与一个正则表达式匹配。
```

examples

匹配前四个值是不是数字

```
if {![regexp {^[0-9]+$} [substr $len_msg 0 1]] or ![regexp {^[0-
9]+$} [substr $len_msg 1 1]] or ![regexp {^[0-9]+$} [substr
$len_msg 2 1]] or ![regexp {^[0-9]+$} [substr $len_msg 3 1]] } {
  log local0. "packet_wrong"
  TCP::payload replace 0 [string length [TCP::payload]] ""
  TCP::collect
   return
  }
```

附录:

BIG-IP Commands and Events by Version 按版本划分的BIG-IP命令和事件 - 这些页面列出了每个版本中iRules的变化。

Master List of Commands - iRules命令的文档。

Master List of Operators - iRules操作的文档。

Disabled Tcl Commands - iRules中禁用的核心TCL命令列表。

iRules Common Concepts - 本节旨在涵盖一些更常见的概念，这些概念出现在不同的iRules中

iRules Troubleshooting Tips - iRules故障排除提示

iRules Procedures (procs) - iRules程序（procs） - 程序概述

iRules Event Order Flow Graph -iRules事件流程图--培训课上的图示

https://www.paessler.com/tools/serversimulator 服务器模拟工具

https://wiki.tcl-lang.org/page/Articles tcl 文档

作者: 郑岳　日期 2022年