Tcl 在 Vivado 中的应用

Xilinx©的新一代设计套件 Vivado 相比上一代产品 ISE，在运行速度、算法优化和功能整合等很多方面都有了显著地改进。但是对初学者来说，新的约束语言 XDC 以及脚本语言 Tcl 的引入则成为了快速掌握 Vivado 使用技巧的最大障碍，以至于两年多后的今天，仍有很多用户缺乏升级到 Vivado 的信心。

本文介绍了 Tcl 在 Vivado 中的基础应用，希望起到抛砖引玉的作用，指引使用者在短时间内快速掌握相关技巧，更好地发挥 Vivado 在 FPGA 设计中的优势。

Tcl 的背景介绍和基础语法

Tcl（读作 tickle）诞生于 80 年代的加州大学伯克利分校，作为一种简单高效可移植性好的脚本语言，目前已经广泛应用在几乎所有的 EDA 工具中。Tcl 的最大特点就是其语法格式极其简单甚至可以说僵化，采用纯粹的 [命令 选项 参数] 形式，是名副其实的“工具命令语言”（即 Tcl 的全称 Tool Command Language）。

实际上 Tcl 的功能可以很强大，用其编写的程序也可以很复杂，但要在 Vivado 或大部分其它 EDA 工具中使用，则只需掌握其中最基本的几个部分。

注： 在以下示例中，% 表示 Tcl 的命令提示符，执行回车后，Tcl 会在下一行输出命令执行结果。// 后是作者所加注释，并不是例子的一部分。

设置变量

```
% set myVar “Hello World!”   //设置一个名为 myVar 的变量，其值为 Hello World!
```

打印

```
% puts  $myVar
Hello  World!

% puts  “Hello World!”
Hello  World!
```

```
% puts  myVar
myVar

% puts  {$myVar}
$myVar
```

```
% puts  “\$myVar”
$myVar
```

打印主要通过 puts 语句来执行，配合特殊符号，直接决定最终输出内容。

文件 I/O

写文件

```
%set wfp [open “my_file.txt” w]
file1073b243

%puts $wfp “Hello World!”

%puts $wfp $myVar

%close $wfp
```

读文件

```
%set rfp [open “my_file.txt” r]
file10

%set file_data [read $rfp]
Hello World!
Hello World!

%close $rfp
```

可以看到 Tcl 对文件的操作也是通过设置变量，改变属性以及打印命令来进行的。上述写文件的例子中通过 puts 命令在 my_file.txt 文件中写入两行文字，分别为“Hello World!”和 myVar 变量的值，然后在读文件操作中逐行读取同一文件的内容。

控制流和循环命令

Tcl 语言中用于控制流程和循环的命令与 C 语言及其它高级语言中相似，包括 if、while、for 和 foreach 等等。

具体使用可以参考如下示例，

```
% if {$myVar != 1} {puts “Sweet!”}  //判断 myVar 变量的值，若不等于 1 就打印 Sweet!
Sweet!

% if {$myVar == 1} { puts “$myVar is = 1” } else { puts “$myVar is != 1”} //多条件判断
Hello World is != 1

% foreach x $myVar {puts $x}  //循环读取 myVar 变量的值并打印
Hello
World!

% set x 1  //设置变量 x
1

% while {$x < 5} { puts “x is $x”; set x [expr {$x + 1}] }  //判断变量的值, 打印，变量再赋值。
x is 1
```

子程序/过程

Tcl 中的子程序也叫做过程（Procedures），Tcl 正是通过创建新的过程来增强其内建命令的能力，提供更强的扩展性。具体到 Vivado 的使用中，用户经常可以通过对一个个子程序/过程的创建来扩展或个性化 Vivado 的使用流程。

```
% proc myProc {var} {   //创建一个新的子程序（过程）myProc，用来打印输入变量的值。
  puts $var
}

% myProc “Yay!”        //调用子程序（过程）
Yay!
```

一些特殊符号

符号	具体含义
$	变量置换
[]	命令置换。即执行括号内的命令，且允许嵌套。
{}	1) 文字字符串，即不进行变量置换和命令置换，把其内的特殊符号当作符号本身。 2) 命令组合，即多个命令的组合。
“”	字符串，对其内的分隔符不作处理，但是允许进行变量置换和命令置换。
\	1) 反斜杠置换，允许其后紧跟着的特殊符号在字符串中以其本身符号的形式出现。 2) 出现在行尾，表示当前命令在下一行继续，从而允许一条命令出现在多行。
;	表示命令的结束。用来把多个命令隔开，从而允许多个命令出现在同一行。
#	注释符。必须出现在 Tcl 解析器期望命令的第一个字符出现的地方，其后直到所在行尾的所有字符都被看作注释。但出现在命令符后的#会被当作字符本身。

```
% set flop_name “flop1”      //字符串
flop1

% set myVar “flop_name”     //没有置换变量
flop_name

% set myVar “$flop_name”    //变量置换
flop1
```

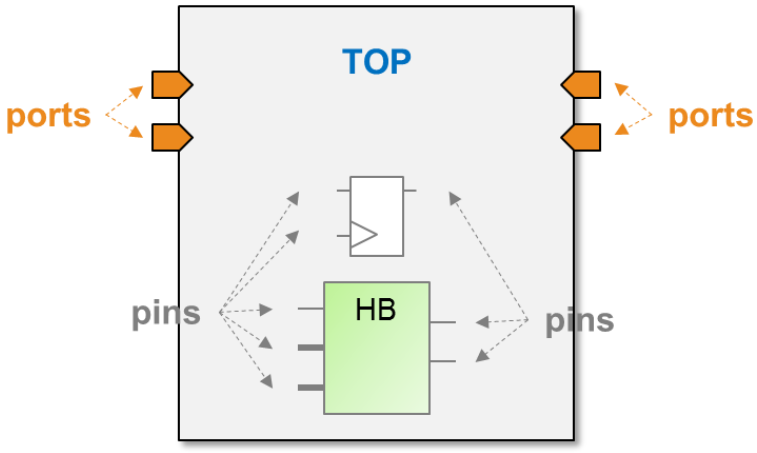
```
% set myVar {$flop_name}     //$代表其本身
$flop_name

% set myVar “\$flop_name”    //反斜杠置换
$flop_name

%set myVar \    //反斜杠允许命令出现在多行
 “$flop_name”
flop1
```

Tcl 语言的基本语法相对简单，但要熟练掌握仍需日常不断练习。Xilinx 网站上有很多相关资料，这里推荐两个跟 Tcl 相关的文档 **UG835** 和 **UG894**，希望对大家学习 Vivado 和 Tcl 有所帮助。

在 Vivado 中使用 Tcl 定位目标

在 Vivado 中使用 Tcl 最基本的场景就是对网表上的目标进行遍历、查找和定位，这也是对网表上的目标进行约束的基础。要掌握这些则首先需要理解 Vivado 对目标的分类。

目标的定义和定位



如上图所示，设计顶层的 I/O 称作 **ports**，其余底层模块或是门级网表上的元件端口都称作 **pins**。而包括顶层在内的各级模块，blackbox 以及门级元件，都称作 **cells**。连线称作 **nets**，加上 XDC 中定义的 **clocks**，在 Vivado 中一共将网表文件中的目标定义为五类。要选取这五类目标，则需用相应的 get_*命令，例如 get_pins 等等。

get_ports

ports 仅指顶层端口，所以 get_ports 的使用相对简单，可以配合通配符“* ”以及 Tcl 语言中处理 list 的命令一起使用。如下所示，

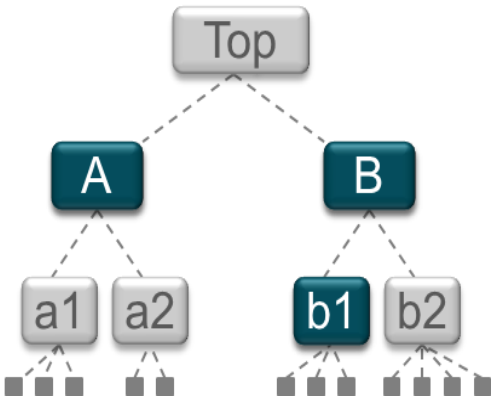
```
get_ports  A          // 仅列出叫做 A 的顶层端口
get_ports  *          //列出所有顶层端口
get_ports  *data*     //列出所有名字中含有 “data” 的顶层端口
```

```
llength [get_ports *]                  // 列出设计中所有顶层端口的数量
lindex  [get_ports *data*] 0           // 列出名字中含有 “data” 的顶层端口中的第一个
foreach x [get_ports *] {puts $x}      // 逐一打印出所有顶层端口
```

get_cells/get_nets

不同于 ports 仅指顶层端口，要定位 cells 和 nets 则相对复杂，首先需要面对层次的问题。这里有个大背景需要明确：Vivado 中 **Tcl/XDC 对网表中目标的搜索是层次化的**，也就是一次仅搜索一个指定的层次 **current_instance**，缺省值为顶层。

以右图所示设计来举例，若要搜索 **A**（**不含 a1,a2**）层次内的所有 cells 和名字中含有 nt 的 nets， 有两种方法：




```
current_instance A   //修改搜索层次到 A
 get_cells *
 get_nets *nt*
current_instance     //修改搜索层次到 Top
```

```
get_cells  A/*  //搜索层次 A 内的所有 cells
```

若要将搜索层次改为 **A+B+b1**，则可以写一个循环，逐一用 **current_instance** 将搜索层次指向 A，B 和 b1，再将搜索到的 cells 或 nets 合成一个 list 输出即可。

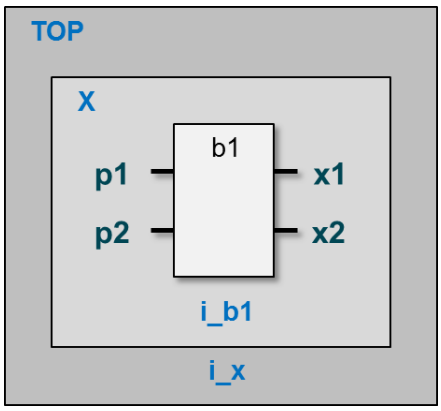
若要将搜索层次改为当前层次以及其下所有子层次，可以使用 **-hierarchical** （在 Tcl 中可以简写为**-hier** ）

```
get_cells  -hierarchical  *        //搜索 Top 层以及其下所有子层次内的所有 cells
get_nets   -hier  *nt*             //搜索 Top 层以及其下所有子层次内的名字中含有 nt 的 nets

current_instance A                 //修改搜索层次到 A
 get_cells -hier  *                //搜索 A 以及其下所有子层次内的所有 cells
 get_nets  -hierarchical *nt*      //搜索 A 以及其下所有子层次内的名字中含有 nt 的 nets
current_instance                   //修改搜索层次到缺省值 Top
```

在使用**-hierarchical**时有一点需要特别留意，即后面所跟的搜索条件仅指目标对象的名字，不能含有代表层次的“/”。下面列出的写法便是一种常见的使用误区，并不能以此搜索到 A 及其下子层次内所有的 cells。

```
get_cells -hier A/*      //搜索 Top 层以及其下所有子层次内名字以“A/”开头的 cells
```

get_pins



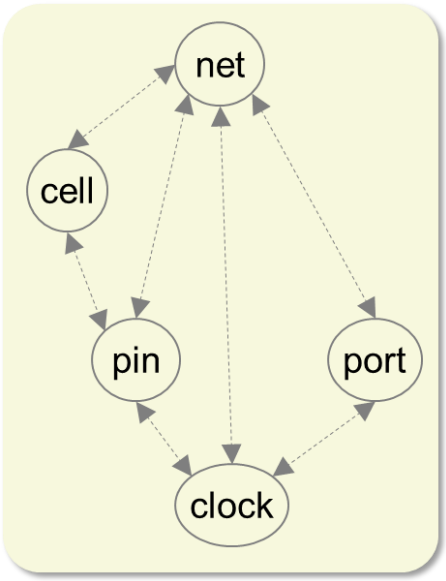
pins 在 Vivado 数据库中有个独特的存在形式，即 **<instance>/<pin>**。这里的“/”不表示层次，而是其名字的一部分，表示这个 pin 所属的实体。也就是说，在使用 get_pins 配合**-hier** 来查找 pins 时，“/”可以作为名字的一部分，出现在搜索条件内（注意与上述 get_cells 和 get_nets 的使用区别）。

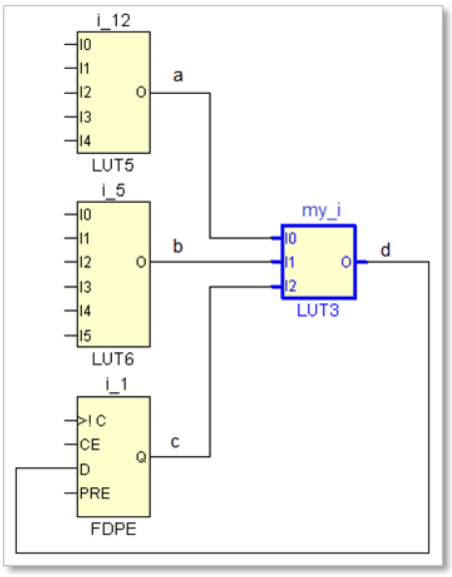
```
get_pins       i_b1/p1    //返回空的 list
get_pins  i_x/i_b1/p1     //找到 i_x/i_b1/p1
get_pins    */i_b1/p1     //找到 i_x/i_b1/p1
get_pins           *p1    //返回空的 list
get_pins -hier     *p1    //找到 i_x/i_b1/p1
get_pins -hier    */p1    //找到 i_x/i_b1/p1
```

```
current_instance   i_x
 get_pins  i_b1/p1   //找到 i_x/i_b1/p1
 get_pins      */p1  //找到 i_x/i_b1/p1
 get_pins      */*   //找到 i_x/i_b1/p1
                          i_x/i_b1/p2
                          i_x/i_b1/x1
                          i_x/i_b1/x2
 get_pins      *     //同上，找到四个 pins
```

目标之间的关系

Tcl 在搜索网表中的目标时，除了上述根据名字条件直接搜索的方式，还可以利用目标间的关系，使用**-of_objects**（在 Tcl 中可以简写为**-of**）来间接搜索特定目标。Vivado 中定义的五类目标间的关系如下页左图所示。

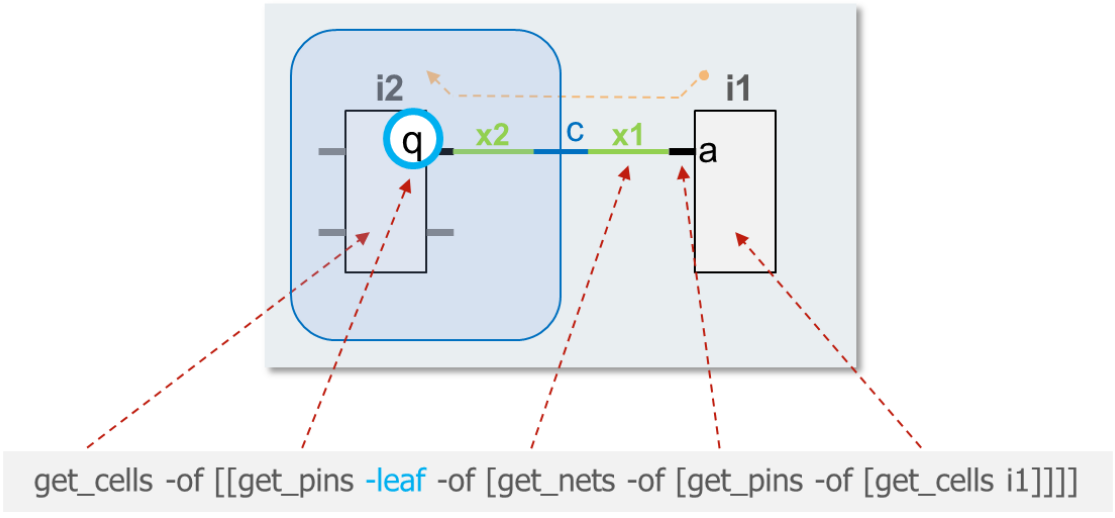




以上示右图的设计来举例，

```
get_pins -of [get_cells my_i]                    //返回 my_i/I0   my_i/I1  my_i/I2  my_i/O

get_nets -of [get_cells my_i]                    //返回 a  b  c  d

get_cells -of [get_nets -of [get_cells my_i]]    //返回 i_12   i_5  i_1 my_i
```

下图是一个更复杂的示例，涉及跨层次搜索。可以看到在 get_pins 时，要加上**-leaf** 才能准确定位到门级元件（或 blackbox）的端口 q。另外，在实际操作中，使用 get_nets 和 get_pins 时，需要视情况而加上其它条件（-filter）才能准确找到下述例子中的 cells （i2）。



高级查找功能

在使用 get_*命令查找网表中的目标时，除了名字这一直接条件，往往还需要辅以其它更复杂的条件判断，这就需要用到高级查找功能：**-filter** 结合 Tcl 支持的各种关系和逻辑运算符（==, !=, =~, !~, <=, >=, >, <, &&, ||）甚至是正则表达式来操作。

```
get_cells  -filter { REF_NAME == FDCE  } *                    //找出所有的 FDCE
get_cells  -filter { REF_NAME == FDCE && INIT == 1'b0 } *     //找出所有初值为 0 的 FDCE
get_cells  -filter { IS_PRIMITIVE == 1 }   *                  //找出所有 primitive
get_ports  -filter { DIRECTION == in && LOC =~ H*  } *        //找出锁定在 H 区的输入管脚
get_ports  -filter { DIRECTION == in && SLEW == FAST  } *data* //找出名字中含有 data 且 slew
                                                              定义为 fast 的输入管脚
```

在创建子程序时也常常用到**-filter**，例如下述 get_p 的子程序/过程就可以用来返回指定管脚的方向属性，告诉用户这是一个输入管脚还是一个输出管脚。

```
proc get_p { direction } {                    //定义一个过程，读取 ports 的 direction 属性
   return [get_ports -filter "DIRECTION == $direction " ]
}
get_p  in                                     //调用上述过程，返回管脚“in”的 direction 属性
```

Tcl 在 Vivado 中的延伸应用

Tcl 在 Vivado 中的应用还远不止上述所列，其它常用的功能包括使用预先写好的 Tcl 脚本来跑设计实现流程，创建高级约束（XDC 不支持循环等高级 Tcl 语法）以及实现复杂的个性化设计流程等等。Tcl 所带来的强大的可扩展性决定了其在版本控制、设计自动化流程等方面具有图形化界面不能比拟的优势。

Vivado 在不断发展更新的过程中，还有很多新的功能，包括 ECO、PR、HD Flow 等等都是从 Tcl 脚本方式开始支持，然后再逐步放入图形化界面中实现。这也解释了为何高端 FPGA 用户和熟练的 Vivado 用户都更偏爱 Tcl 脚本。

篇幅所限，不能一一展开。关于以上 Tcl 在 Vivado 中的延伸应用，敬请关注 Xilinx 官方网站和中文论坛上的更多技术文章。

Ally Zhou 2014-9-12 于 Xilinx 上海 Office