```
2) 5.6
3) quit
Select a number: 1
mysql 5.1
1) 5.1
2) 5.6
3) quit
Select a number: 2
mysql 5.6
1) 5.1
2) 5.6
3) quit
Select a number: 3
```

如果不想用默认的提示符，可以通过重新赋值变量 PS3 来自定义。这下就比较完美了！

第五章　Shell 函数与数组

5.1 函数

格式：

```
func() {
      command
}
```

function 关键字可写，也可不写。
示例 1：

```
#!/bin/bash
func() {
      echo "This is a function."
}
func
# bash test.sh
This is a function.
```

Shell 函数很简单，函数名后跟双括号，再跟双大括号。通过函数名直接调用，不加小括号。
示例 2：函数返回值

```
#!/bin/bash
func() {
      VAR=$((1+1))
      return $VAR
      echo "This is a function."
}
func
echo $?
# bash test.sh
```

```
2
```

return 在函数中定义状态返回值，返回并终止函数，但返回的只能是 0-255 的数字，类似于 exit。
示例 3：函数传参

```
#!/bin/bash
func() {
      echo "Hello $1"
}
func world
# bash test.sh
Hello world
```

通过 Shell 位置参数给函数传参。
函数也支持递归调用，也就是自己调用自己。
例如：

```
#!/bin/bash
test() {
      echo $1
      sleep 1
      test hello
}
test
```

执行会一直在调用本身打印 hello，这就形成了闭环。
像经典的 fork 炸弹就是函数递归调用：
:(){ :|:& };: 或 .(){.|.&};.
这样看起来不好理解，我们更改下格式：

```
:() {
  :|:&
};
:
```

再易读一点：

```
bomb() {
  bomb|bomb&
};
bomb
```

分析下：
:(){ } 定义一个函数，函数名是冒号。
: 调用自身函数
| 管道符
: 再一次递归调用自身函数
:|: 表示每次调用函数":"的时候就会生成两份拷贝。
& 放到后台
; 分号是继续执行下一个命令，可以理解为换行。
: 最后一个冒号是调用函数。
因此不断生成新进程，直到系统资源崩溃。

一般递归函数用的也少，了解下即可！

5.2 数组

数组是相同类型的元素按一定顺序排列的集合。
格式：
array=(元素 1 元素 2 元素 3 ...)
用小括号初始化数组，元素之间用空格分隔。
定义方法 1：初始化数组
array=(a b c)
定义方法 2：新建数组并添加元素
array[下标]=元素
定义方法 3：将命令输出作为数组元素
array=($(command))
数组操作：

```
获取所有元素：
# echo ${array[*]}   # *和@ 都是代表所有元素
a b c
获取元素下标：
# echo ${!a[@]}
0 1 2
获取数组长度：
# echo ${#array[*]}
3
获取第一个元素：
# echo ${array[0]}
a
获取第二个元素：
# echo ${array[1]}
b
获取第三个元素：
# echo ${array[2]}
c
添加元素：
# array[3]=d
# echo ${array[*]}
a b c d
添加多个元素：
# array+=(e f g)
# echo ${array[*]}
a b c d e f g
删除第一个元素：
# unset array[0]     # 删除会保留元素下标
# echo ${array[*]}
b c d e f g
删除数组：
# unset array
```

数组下标从 0 开始。
示例 1：讲 seq 生成的数字序列循环放到数组里面

```
#!/bin/bash
for i in $(seq 1 10); do
      array[a]=$i
      let a++
done
echo ${array[*]}
# bash test.sh
1 2 3 4 5 6 7 8 9 10
```

示例 2：遍历数组元素

```
方法 1：
#!/bin/bash
IP=(192.168.1.1 192.168.1.2 192.168.1.3)
for ((i=0;i<${#IP[*]};i++)); do
      echo ${IP[$i]}
done
# bash test.sh
192.168.1.1
192.168.1.2
192.168.1.3
方法 2：
#!/bin/bash
IP=(192.168.1.1 192.168.1.2 192.168.1.3)
for   IP   in ${IP[*]}; do
        echo $IP
done
```

第六章　Shell 正则表达式

正则表达式在每种语言中都会有，功能就是匹配符合你预期要求的字符串。
Shell 正则表达式分为两种：
基础正则表达式：BRE（basic regular express）
扩展正则表达式：ERE（extend regular express），扩展的表达式有+、?、|和()
下面是一些常用的正则表达式符号，我们先拿 grep 工具举例说明。

符号	描述	示例
.	匹配除换行符(\n)之外的任意单个字符	匹配 123： echo -e "123\n456" |grep '1.3'
^	匹配前面字符串开头	匹配以 abc 开头的行： echo -e "abc\nxyz" |grep ^abc
$	匹配前面字符串结尾	匹配以 xyz 结尾的行：