

Table of Contents

Introduction	0
bash-cn	1
bf-cn	2
c++-cn	3
c-cn	4
clojure-cn	5
clojure-macro-cn	6
coffeescript-cn	7
common-lisp-cn	8
csharp-cn	9
css-cn	10
dart-cn	11
elisp-cn	12
elixir-cn	13
erlang-cn	14
git-cn	15
go-cn	16
groovy-cn	17
haskell-cn	18
java-cn	19
javascript-cn	20
json-cn	21
julia-cn	22
livescript-cn	23
lua-cn	24
markdown-cn	25
matlab-cn	26
perl-cn	27
php-cn	28
python-cn	29
python3-cn	30
r-cn	31
racket-cn	32

ruby-cn	33
rust-cn	34
scala-cn	35
swift-cn	36
tmux-cn	37
typescript-cn	38
visualbasic-cn	39
xml-cn	40
yaml-cn	41

Introduction

category: tool tool: bash contributors:

```
- ["Max Yankov", "https://github.com/golergka"]
- ["Darren Lin", "https://github.com/CogBear"]
- ["Alexandre Medeiros", "http://alemedeiros.sdf.org"]
- ["Denis Arh", "https://github.com/darh"]
- ["akirahirose", "https://twitter.com/akirahirose"]
- ["Anton Strömkvist", "http://lutic.org/"]
- ["Rahil Momin", "https://github.com/iamrahil"]
- ["Gregrory Kielian", "https://github.com/gskielian"]
- ["Etan Reisner", "https://github.com/deryni"]
```

translators:

```
- ["Jinchang Ye", "https://github.com/Alwayswithme"]
- ["Chunyang Xu", "https://github.com/XuChunyang"]
```

filename: LearnBash-cn.sh

lang: zh-cn

Bash 是一个为 GNU 计划编写的 Unix shell，是 Linux 和 Mac OS X 下的默认 shell。以下大多数例子可以作为脚本的一部分运行，也可直接在 shell 下交互执行。

[更多信息](#)

```
#!/bin/bash
# 脚本的第一行叫 shebang，用来告知系统如何执行该脚本：
# 参见： http://en.wikipedia.org/wiki/Shebang\_\(Unix\)
# 如你所见，注释以 # 开头，shebang 也是注释。

# 显示 “Hello world!”
echo Hello world!

# 每一句指令以换行或分号隔开：
echo 'This is the first line'; echo 'This is the second line'

# 声明一个变量：
Variable="Some string"

# 下面是错误的做法：
Variable = "Some string"
# Bash 会把 Variable 当做一个指令，由于找不到该指令，因此这里会报错。

# 也不可以这样：
Variable= 'Some string'
```

```
# Bash 会认为 'Some string' 是一条指令，由于找不到该指令，这里再次报错。
# （这个例子中 'Variable=' 这部分会被当作仅对 'Some string' 起作用的赋值。）

# 使用变量：
echo $Variable
echo "$Variable"
echo '$Variable'

# 当你赋值（assign）、导出（export），或者以其他方式使用变量时，变量名前不加 $。
# 如果要使用变量的值，则加 $。
# 注意：'（单引号）不会展开变量（即会屏蔽掉变量）。

# 在变量内部进行字符串代换
echo ${Variable/Some/A}
# 会把 Variable 中首次出现的 "some" 替换成 "A"。

# 变量的截取
Length=7
echo ${Variable:0:Length}
# 这样会仅返回变量值的前7个字符

# 变量的默认值
echo ${Foo:-"DefaultValueIfFooIsMissingOrEmpty"}
# 对 null（Foo=）和空串（Foo=""）起作用；零（Foo=0）时返回0
# 注意这仅返回默认值而不是改变变量的值

# 内置变量：
# 下面的内置变量很有用
echo "Last program return value: $?"
echo "Script's PID: $$"
echo "Number of arguments: $# "
echo "Scripts arguments: $@"
echo "Scripts arguments separated in different variables: $1 $2..."

# 读取输入：
echo "What's your name?"
read Name # 这里不需要声明新变量
echo Hello, $Name!

# 通常的 if 结构看起来像这样：
# 'man test' 可查看更多的信息
if [ $Name -ne $USER ]
then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# 根据上一个指令执行结果决定是否执行下一个指令
echo "Always executed" || echo "Only executed if first command fails"
echo "Always executed" && echo "Only executed if first command does NOT fail"
```

```

# 在 if 语句中使用 && 和 || 需要多对方括号
if [ $Name == "Steve" ] && [ $Age -eq 15 ]
then
    echo "This will run if $Name is Steve AND $Age is 15."
fi

if [ $Name == "Daniya" ] || [ $Name == "Zach" ]
then
    echo "This will run if $Name is Daniya OR Zach."
fi

# 表达式的格式如下：
echo $(( 10 + 5 ))

# 与其他编程语言不同的是，bash 运行时依赖上下文。比如，使用 ls 时，列出当前目录。
ls

# 指令可以带有选项：
ls -l # 列出文件和目录的详细信息

# 前一个指令的输出可以当作后一个指令的输入。grep 用来匹配字符串。
# 用下面的指令列出当前目录下所有的 txt 文件：
ls -l | grep "\.txt"

# 重定向输入和输出（标准输入，标准输出，标准错误）。
# 以 ^EOF$ 作为结束标记从标准输入读取数据并覆盖 hello.py :
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF

# 重定向可以到输出，输入和错误输出。
python hello.py < "input.in"
python hello.py > "output.out"
python hello.py 2> "error.err"
python hello.py > "output-and-error.log" 2>&1
python hello.py > /dev/null 2>&1
# > 会覆盖已存在的文件，>> 会以累加的方式输出文件中。
python hello.py >> "output.out" 2>> "error.err"

# 覆盖 output.out ，追加 error.err 并统计行数
info bash 'Basic Shell Features' 'Redirections' > output.out 2>> error.err
wc -l output.out error.err

# 运行指令并打印文件描述符（比如 /dev/fd/123）

```

```

# 具体可查看: man fd
echo <(echo "#helloworld")

# 以 "#helloworld" 覆盖 output.out:
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null

# 清理临时文件并显示详情 (增加 '-i' 选项启用交互模式)
rm -v output.out error.err output-and-error.log

# 一个指令可用 $( ) 嵌套在另一个指令内部:
# 以下的指令会打印当前目录下的目录和文件总数
echo "There are $(ls | wc -l) items here."

# 反引号 `` 起相同作用, 但不允许嵌套
# 优先使用 $( ).
echo "There are `ls | wc -l` items here."

# Bash 的 case 语句与 Java 和 C++ 中的 switch 语句类似:
case "$Variable" in
    # 列出需要匹配的字符串
    0) echo "There is a zero.>";
    1) echo "There is a one.>";
    *) echo "It is not null.>";
esac

# 循环遍历给定的参数序列:
# 变量$Variable 的值会被打印 3 次。
for Variable in {1..3}
do
    echo "$Variable"
done

# 或传统的 “for循环” :
for ((a=1; a <= 3; a++))
do
    echo $a
done

# 也可以用于文件
# 用 cat 输出 file1 和 file2 内容
for Variable in file1 file2
do
    cat "$Variable"
done

# 或作用于其他命令的输出
# 对 ls 输出的文件执行 cat 指令。
for Output in $(ls)

```

```

do
    cat "$Output"
done

# while 循环:
while [ true ]
do
    echo "loop body here..."
    break
done

# 你也可以使用函数
# 定义函数:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    return 0
}

# 更简单的方法
bar ()
{
    echo "Another way to declare functions!"
    return 0
}

# 调用函数
foo "My name is" $Name

# 有很多有用的指令需要学习:
# 打印 file.txt 的最后 10 行
tail -n 10 file.txt
# 打印 file.txt 的前 10 行
head -n 10 file.txt
# 将 file.txt 按行排序
sort file.txt
# 报告或忽略重复的行, 用选项 -d 打印重复的行
uniq -d file.txt
# 打印每行中 ',' 之前内容
cut -d ',' -f 1 file.txt
# 将 file.txt 文件所有 'okay' 替换为 'great', (兼容正则表达式)
sed -i 's/okay/great/g' file.txt
# 将 file.txt 中匹配正则的行打印到标准输出
# 这里打印以 "foo" 开头, "bar" 结尾的行
grep "^foo.*bar$" file.txt
# 使用选项 "-c" 统计行数
grep -c "^foo.*bar$" file.txt
# 如果只是要按字面形式搜索字符串而不是按正则表达式, 使用 fgrep (或 grep -F)
fgrep "^foo.*bar$" file.txt

```


以 bash 内建的 'help' 指令阅读 Bash 自带文档:

```
help  
help help  
help for  
help return  
help source  
help .
```

用 man 指令阅读相关的 Bash 手册

```
apropos bash  
man 1 bash  
man bash
```

用 info 指令查阅命令的 info 文档 (info 中按 ? 显示帮助信息)

```
apropos info | grep '^info.*('  
man info  
info info  
info 5 info
```

阅读 Bash 的 info 文档:

```
info bash  
info bash 'Bash Features'  
info bash 6  
info --apropos bash
```

language: bf lang: zh-cn contributors:

```
- ["Prajit Ramachandran", "http://prajitr.github.io/"]
- ["Mathias Bynens", "http://mathiasbynens.be/"]
```

translators:

```
- ["lyuehh", "https://github.com/lyuehh"]
```

Brainfuck 是一个极小的只有8个指令的图灵完全的编程语言。

除"><+-.,[]"之外的任何字符都会被忽略（不包含双引号）。

Brainfuck 包含一个有30,000个单元为0的数组，和一个数据指针指向当前的单元。

8个指令如下：

- + ： 指针指向的单元的值加1
- ： 指针指向的单元的值减1
- > ： 将指针移动到下一个单元(右边的元素)
- < ： 将指针移动到上一个单元(左边的元素)
- . ： 打印当前单元的内容的ASCII值（比如 65 = 'A'）.
- , ： 读取一个字符到当前的单元
- [： 如果当前单元的值是0，则向后调转到对应的]处
-] ： 如果当前单元的值不是0，则向前跳转到对应的[处

[和] 组成了一个while循环。很明显，它们必须配对。

让我们看一些基本的brainfuck 程序。

```
++++++ [ > ++++++++ < - ] > +++++ .
```

这个程序打印字母'A'。首先，它把 #1 增加到6，使用它来作为循环条件，然后，进入循环，将指针移动到 #2，将 #2 的值增加到10，然后移动回 #1，将单元 #1 的值减1，然后继续。循环共进行了6次。

这时，我们在 #1，它的值为0，#2 的值为60，我们移动到 #2，将 #2 的内容加上5，然后将 #2 的内容打印出来，65在ASCII中表示'A'，所以'A'就会被打印出来。

```
, [ > + < - ] > .
```

这个程序从用户的输入中读取一个字符，然后把它复制到 #1。然后我们开始一个循环，移动到 #2，将 #2 的值加1，再移动回 #1，将 #1

的值减1，直到 #1的值为0，这样 #2 里就保存了 #1 的旧值，循环结束时我们在 #1，这时我们移动到 #2，然后把字符以ASCII打印出来。

而且要记住的一点就是，空格在这里只是为了可读性，你可以将他们写成这样：

```
,[>+<-]>.
```

试着思考一下这段程序是干什么的：

```
,>,< [ > [ >+ >+ << -] >> [- << + >>] <<< -] >>
```

这段程序从输入接收2个参数，然后将他们相乘。

先读取2个输入，然后开始外层循环，以 #1 作为终止条件，然后将指针移动到 #2，然后开始 #2 的内层循环，将 #3 加1。但是这里有一个小问题，在内层循环结束的时候，#2 的值是0了，那么下次执行外层循环的时候，就有问题了。为了解决这个问题，我们可以增加 #4 的值，然后把 #4 的值复制到 #2，最后结果就保存在 #3 中了。

好了这就是brainfuck了。也没那么难，是吧？为了好玩，你可以写你自己的 brainfuck程序，或者用其他语言写一个brainfuck的解释器，解释器非常容易 实现，但是如果你是一个自虐狂的话，你可以尝试用brainfuck写一个brainfuk的 解释器。

language: c++ filename: learncpp-cn.cpp contributors:

```
- ["Steven Basart", "http://github.com/xksteven"]
- ["Matt Kline", "https://github.com/mrkline"]
```

translators:

```
- ["Arnie97", "https://github.com/Arnie97"]
```

lang: zh-cn

C++是一种系统编程语言。用它的发明者，[Bjarne Stroustrup](#)的话来说，C++的设计目标是：

- 成为“更好的C语言”
- 支持数据的抽象与封装
- 支持面向对象编程
- 支持泛型编程

C++提供了对硬件的紧密控制（正如C语言一样），能够编译为机器语言，由处理器直接执行。与此同时，它也提供了泛型、异常和类等高层功能。虽然C++的语法可能比某些出现较晚的语言更复杂，它仍然得到了人们的青睐——功能与速度的平衡使C++成为了目前应用最广泛的系统编程语言之一。

```
//////////
// 与C语言的比较
//////////

// C++_几乎_是C语言的一个超集，它与C语言的基本语法有许多相同之处，
// 例如变量和函数的声明，原生数据类型等等。

// 和C语言一样，在C++中，你的程序会从main()开始执行，
// 该函数的返回值应当为int型，这个返回值会作为程序的退出状态值。
// 不过，大多数的编译器（gcc，clang等）也接受 void main() 的函数原型。
// （参见 http://en.wikipedia.org/wiki/Exit\_status 来获取更多信息）
int main(int argc, char** argv)
{
    // 和C语言一样，命令行参数通过argc和argv传递。
    // argc代表命令行参数的数量，
    // 而argv是一个包含“C语言风格字符串”（char *）的数组，
    // 其中每个字符串代表一个命令行参数的内容，
    // 首个命令行参数是调用该程序时所使用的名称。
    // 如果你不关心命令行参数的值，argc和argv可以被忽略。
    // 此时，你可以用int main()作为函数原型。

    // 退出状态值为0时，表示程序执行成功
    return 0;
```

```

}

// 然而，C++和C语言也有一些区别：

// 在C++中，字符字面量的大小是一个字节。
sizeof('c') == 1

// 在C语言中，字符字面量的大小与int相同。
sizeof('c') == sizeof(10)

// C++的函数原型与函数定义是严格匹配的
void func(); // 这个函数不能接受任何参数

// 而在C语言中
void func(); // 这个函数能接受任意数量的参数

// 在C++中，用nullptr代替C语言中的NULL
int* ip = nullptr;

// C++也可以使用C语言的标准头文件，
// 但是需要加上前缀“c”并去掉末尾的“.h”。
#include <cstdio>

int main()
{
    printf("Hello, world!\n");
    return 0;
}

//////////
// 函数重载
//////////

// C++支持函数重载，你可以定义一组名称相同而参数不同的函数。

void print(char const* myString)
{
    printf("String %s\n", myString);
}

void print(int myInt)
{
    printf("My int is %d", myInt);
}

int main()
{
    print("Hello"); // 解析为 void print(const char*)
    print(15); // 解析为 void print(int)
}

```

```

//////////
// 函数参数的默认值
//////////

// 你可以为函数的参数指定默认值，
// 它们将会在调用者没有提供相应参数时被使用。

void doSomethingWithInts(int a = 1, int b = 4)
{
    // 对两个参数进行一些操作
}

int main()
{
    doSomethingWithInts();        // a = 1, b = 4
    doSomethingWithInts(20);      // a = 20, b = 4
    doSomethingWithInts(20, 5);   // a = 20, b = 5
}

// 默认参数必须放在所有的常规参数之后。

void invalidDeclaration(int a = 1, int b) // 这是错误的!
{
}

//////////
// 命名空间
//////////

// 命名空间为变量、函数和其他声明提供了分离的作用域。
// 命名空间可以嵌套使用。

namespace First {
    namespace Nested {
        void foo()
        {
            printf("This is First::Nested::foo\n");
        }
    } // 结束嵌套的命名空间Nested
} // 结束命名空间First

namespace Second {
    void foo()
    {
        printf("This is Second::foo\n")
    }
}

void foo()

```

```

{
    printf("This is global foo\n");
}

int main()
{
    // 如果没有特别指定，就从“Second”中取得所需的内容。
    using namespace Second;

    foo(); // 显示“This is Second::foo”
    First::Nested::foo(); // 显示“This is First::Nested::foo”
    ::foo(); // 显示“This is global foo”
}

//////////
// 输入/输出
//////////

// C++使用“流”来输入输出。<<是流的插入运算符，>>是流提取运算符。
// cin、cout、和cerr分别代表
// stdin（标准输入）、stdout（标准输出）和stderr（标准错误）。

#include <iostream> // 引入包含输入/输出流的头文件

using namespace std; // 输入输出流在std命名空间（也就是标准库）中。

int main()
{
    int myInt;

    // 在标准输出（终端/显示器）中显示
    cout << "Enter your favorite number:\n";
    // 从标准输入（键盘）获得一个值
    cin >> myInt;

    // cout也提供了格式化功能
    cout << "Your favorite number is " << myInt << "\n";
    // 显示“Your favorite number is <myInt>”

    cerr << "Used for error messages";
}

//////////
// 字符串
//////////

// C++中的字符串是对象，它们有很多成员函数
#include <string>

using namespace std; // 字符串也在std命名空间（标准库）中。

```

```

string myString = "Hello";
string myOtherString = " World";

// + 可以用于连接字符串。
cout << myString + myOtherString; // "Hello World"

cout << myString + " You"; // "Hello You"

// C++中的字符串是可变的，具有“值语义”。
myString.append(" Dog");
cout << myString; // "Hello Dog"

//////////
// 引用
//////////

// 除了支持C语言中的指针类型以外，C++还提供了_引用_。
// 引用是一种特殊的指针类型，一旦被定义就不能重新赋值，并且不能被设置为空值。
// 使用引用时的语法与原变量相同：
// 也就是说，对引用类型进行解引用时，不需要使用*；
// 赋值时也不需要&来取地址。

using namespace std;

string foo = "I am foo";
string bar = "I am bar";

string& fooRef = foo; // 建立了一个对foo的引用。
fooRef += ". Hi!"; // 通过引用来修改foo的值
cout << fooRef; // "I am foo. Hi!"

// 这句话的并不会改变fooRef的指向，其效果与“foo = bar”相同。
// 也就是说，在执行这条语句之后，foo == "I am bar"。
fooRef = bar;

const string& barRef = bar; // 建立指向bar的常量引用。
// 和C语言中一样，（指针和引用）声明为常量时，对应的值不能被修改。
barRef += ". Hi!"; // 这是错误的，不能修改一个常量引用的值。

//////////
// 类与面向对象编程
//////////

// 有关类的第一个示例
#include <iostream>

// 声明一个类。
// 类通常在头文件（.h或.hpp）中声明。
class Dog {

```



```

// 成员变量和成员函数默认情况下是私有（private）的。
std::string name;
int weight;

// 在这个标签之后，所有声明都是公有（public）的，
// 直到重新指定“private:”（私有继承）或“protected:”（保护继承）为止
public:

    // 默认的构造器
    Dog();

    // 这里是成员函数声明的一个例子。
    // 可以注意到，我们在此处使用了std::string，而不是using namespace std
    // 语句using namespace绝不当出现在头文件当中。
    void setName(const std::string& dogsName);

    void setWeight(int dogsWeight);

    // 如果一个函数不对对象的状态进行修改，
    // 应当在声明中加上const。
    // 这样，你就可以对一个以常量方式引用的对象执行该操作。
    // 同时可以注意到，当父类的成员函数需要被子类重写时，
    // 父类中的函数必须被显式声明为_虚函数（virtual）_。
    // 考虑到性能方面的因素，函数默认情况下不会被声明为虚函数。
    virtual void print() const;

    // 函数也可以在class body内部定义。
    // 这样定义的函数会自动成为内联函数。
    void bark() const { std::cout << name << " barks!\n" }

    // 除了构造器以外，C++还提供了析构器。
    // 当一个对象被删除或者脱离其定义域时，它的析构函数会被调用。
    // 这使得RAII这样的强大范式（参见下文）成为可能。
    // 为了衍生出子类来，基类的析构函数必须定义为虚函数。
    virtual ~Dog();

}; // 在类的定义之后，要加一个分号

// 类的成员函数通常在.cpp文件中实现。
void Dog::Dog()
{
    std::cout << "A dog has been constructed\n";
}

// 对象（例如字符串）应当以引用的形式传递，
// 对于不需要修改的对象，最好使用常量引用。
void Dog::setName(const std::string& dogsName)
{
    name = dogsName;
}

```

```

void Dog::setWeight(int dogsWeight)
{
    weight = dogsWeight;
}

// 虚函数的virtual关键字只需要在声明时使用，不需要在定义时重复
void Dog::print() const
{
    std::cout << "Dog is " << name << " and weighs " << weight << "kg\n";
}

void Dog::~Dog()
{
    cout << "Goodbye " << name << "\n";
}

int main() {
    Dog myDog; // 此时显示“A dog has been constructed”
    myDog.setName("Barkley");
    myDog.setWeight(10);
    myDog.printDog(); // 显示“Dog is Barkley and weighs 10 kg”
    return 0;
} // 显示“Goodbye Barkley”

// 继承：

// 这个类继承了Dog类中的公有（public）和保护（protected）对象
class OwnedDog : public Dog {

    void setOwner(const std::string& dogsOwner)

    // 重写OwnedDogs类的print方法。
    // 如果你不熟悉子类多态的话，可以参考这个页面中的概述：
    // http://zh.wikipedia.org/wiki/%E5%AD%A7%B1%BB%E5%9E%8B

    // override关键字是可选的，它确保你所重写的是基类中的方法。
    void print() const override;

private:
    std::string owner;
};

// 与此同时，在对应的.cpp文件里：

void OwnedDog::setOwner(const std::string& dogsOwner)
{
    owner = dogsOwner;
}

void OwnedDog::print() const
{

```

```

    Dog::print(); // 调用基类Dog中的print方法
    // "Dog is <name> and weights <weight>"

    std::cout << "Dog is owned by " << owner << "\n";
    // "Dog is owned by <owner>"
}

//////////
// 初始化与运算符重载
//////////

// 在C++中，通过定义一些特殊名称的函数，
// 你可以重载+、-、*、/等运算符的行为。
// 当运算符被使用时，这些特殊函数会被调用，从而实现运算符重载。

#include <iostream>
using namespace std;

class Point {
public:
    // 可以以这样的方式为成员变量设置默认值。
    double x = 0;
    double y = 0;

    // 定义一个默认的构造器。
    // 除了将Point初始化为(0, 0)以外，这个函数什么都不做。
    Point() { };

    // 下面使用的语法称为初始化列表，
    // 这是初始化类中成员变量的正确方式。
    Point (double a, double b) :
        x(a),
        y(b)
    { /* 除了初始化成员变量外，什么都不做 */ }

    // 重载 + 运算符
    Point operator+(const Point& rhs) const;

    // 重载 += 运算符
    Point& operator+=(const Point& rhs);

    // 增加 - 和 -= 运算符也是有意义的，但这里不再赘述。
};

Point Point::operator+(const Point& rhs) const
{
    // 创建一个新的点，
    // 其横纵坐标分别为这个点与另一点在对应方向上的坐标之和。
    return Point(x + rhs.x, y + rhs.y);
}

```

```

Point& Point::operator+=(const Point& rhs)
{
    x += rhs.x;
    y += rhs.y;
    return *this;
}

int main () {
    Point up (0,1);
    Point right (1,0);
    // 这里使用了Point类型的运算符“+”
    // 调用up (Point类型)的“+”方法，并以right作为函数的参数
    Point result = up + right;
    // 显示“Result is upright (1,1)”
    cout << "Result is upright (" << result.x << ',' << result.y << ")\n";
    return 0;
}

//////////
// 异常处理
//////////

// 标准库中提供了一些基本的异常类型
// （参见http://en.cppreference.com/w/cpp/error/exception）
// 但是，其他任何类型也可以作为一个异常被抛出
#include <exception>

// 在_try_代码块中抛出的异常可以被随后的_catch_捕获。
try {
    // 不要用 _new_关键字在堆上为异常分配空间。
    throw std::exception("A problem occurred");
}
// 如果抛出的异常是一个对象，可以用常量引用来捕获它
catch (const std::exception& ex)
{
    std::cout << ex.what();
    // 捕获尚未被_catch_处理的所有错误
} catch (...)
{
    std::cout << "Unknown exception caught";
    throw; // 重新抛出异常
}

//////////
// RAII
//////////

// RAII指的是“资源获取就是初始化”（Resource Allocation Is Initialization），
// 它被视作C++中最强大的编程范式之一。
// 简单说来，它指的是，用构造函数来获取一个对象的资源，
// 相应的，借助析构函数来释放对象的资源。

```

// 为了理解这一范式的用处，让我们考虑某个函数使用文件句柄时的情况：

```
void doSomethingWithAFile(const char* filename)
{
    // 首先，让我们假设一切都会顺利进行。

    FILE* fh = fopen(filename, "r"); // 以只读模式打开文件

    doSomethingWithTheFile(fh);
    doSomethingElseWithIt(fh);

    fclose(fh); // 关闭文件句柄
}
```

// 不幸的是，随着错误处理机制的引入，事情会变得复杂。

// 假设fopen函数有可能执行失败，

// 而doSomethingWithTheFile和doSomethingElseWithIt会在失败时返回错误代码。

// （虽然异常是C++中处理错误的推荐方式，

// 但是某些程序员，尤其是有C语言背景的，并不认可异常捕获机制的作用）。

// 现在，我们必须检查每个函数调用是否成功执行，并在问题发生的时候关闭文件句柄。

```
bool doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r"); // 以只读模式打开文件
    if (fh == nullptr) // 当执行失败是，返回的指针是nullptr
        return false; // 向调用者汇报错误

    // 假设每个函数会在执行失败时返回false
    if (!doSomethingWithTheFile(fh)) {
        fclose(fh); // 关闭文件句柄，避免造成内存泄漏。
        return false; // 反馈错误
    }
    if (!doSomethingElseWithIt(fh)) {
        fclose(fh); // 关闭文件句柄
        return false; // 反馈错误
    }

    fclose(fh); // 关闭文件句柄
    return true; // 指示函数已成功执行
}
```

// C语言的程序员通常会借助goto语句简化上面的代码：

```
bool doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r");
    if (fh == nullptr)
        return false;

    if (!doSomethingWithTheFile(fh))
        goto failure;

    if (!doSomethingElseWithIt(fh))
```

```

        goto failure;

    fclose(fh); // 关闭文件
    return true; // 执行成功

failure:
    fclose(fh);
    return false; // 反馈错误
}

// 如果用异常捕获机制来指示错误的话，
// 代码会变得清晰一些，但是仍然有优化的余地。
void doSomethingWithAFile(const char* filename)
{
    FILE* fh = fopen(filename, "r"); // 以只读模式打开文件
    if (fh == nullptr)
        throw std::exception("Could not open the file.");

    try {
        doSomethingWithTheFile(fh);
        doSomethingElseWithIt(fh);
    }
    catch (...) {
        fclose(fh); // 保证出错的时候文件被正确关闭
        throw; // 之后，重新抛出这个异常
    }

    fclose(fh); // 关闭文件
    // 所有工作顺利完成
}

```

// 相比之下，使用C++中的文件流类（fstream）时，
 // fstream会利用自己的析构器来关闭文件句柄。
 // 只要离开了某一对象的定义域，它的析构函数就会被自动调用。

```

void doSomethingWithAFile(const std::string& filename)
{
    // ifstream是输入文件流（input file stream）的简称
    std::ifstream fh(filename); // 打开一个文件

    // 对文件进行一些操作
    doSomethingWithTheFile(fh);
    doSomethingElseWithIt(fh);

} // 文件已经被析构器自动关闭

```

// 与上面几种方式相比，这种方式有着_明显_的优势：
 // 1. 无论发生了什么情况，资源（此例当中是文件句柄）都会被正确关闭。
 // 只要你正确使用了析构器，就_不会_因为忘记关闭句柄，造成资源的泄漏。
 // 2. 可以注意到，通过这种方式写出来的代码十分简洁。
 // 析构器会在后台关闭文件句柄，不再需要你来操心这些琐事。
 // 3. 这种方式的代码具有异常安全性。

```
// 无论在函数中的何处抛出异常，都不会阻碍对文件资源的释放。

// 地道的C++代码应当把RAII的使用扩展到各种类型的资源上，包括：
// - 用unique_ptr和shared_ptr管理的内存
// - 各种数据容器，例如标准库中的链表、向量（容量自动扩展的数组）、散列表等；
// 当它们脱离作用域时，析构器会自动释放其中储存的内容。
// - 用lock_guard和unique_lock实现的互斥
```

扩展阅读：

<http://cppreference.com/w/cpp> 提供了最新的语法参考。

可以在 <http://cplusplus.com> 找到一些补充资料。

language: c filename: learnc-cn.c contributors:

```
- ["Adam Bard", "http://adambard.com/"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net/"]  
- ["Jakukyo Friel", "http://weakish.github.io"]
```

lang: zh-cn

C语言在今天仍然是高性能计算的主要选择。

C大概是大多数程序员用到的最接近底层的语言了，C语言原生的速度就很高了，但是别忘了C的手动内存管理，它会让你将性能发挥到极致。

```
// 单行注释以//开始。（仅适用于C99或更新的版本。）  
  
/*  
多行注释是这个样子的。（C89也适用。）  
*/  
  
// 常数： #define 关键词  
#define DAYS_IN_YEAR 365  
  
// 以枚举的方式定义常数  
enum days {SUN = 1, MON, TUE, WED, THU, FRI, SAT};  
// MON自动被定义为2，TUE被定义为3，以此类推。  
  
// 用#include来导入头文件  
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
  
// <尖括号>间的文件名是C标准库的头文件。  
// 标准库以外的头文件，使用双引号代替尖括号。  
#include "my_header.h"  
  
// 函数的签名可以事先在.h文件中定义，  
// 也可以直接在.c文件的头部定义。  
void function_1(char c);  
void function_2(void);  
  
// 如果函数出现在main()之后，那么必须在main()之前  
// 先声明一个函数原型  
int add_two_ints(int x1, int x2); // 函数原型
```



```

// 你的程序的入口是一个返回值为整型的main函数
int main() {

// 用printf打印到标准输出，可以设定格式，
// %d 代表整数，\n 代表换行
printf("%d\n", 0); // => 打印 0
// 所有的语句都要以分号结束

////////////////////////////////////
// 类型
////////////////////////////////////

// 在使用变量之前我们必须先声明它们。
// 变量在声明时需要指明其类型，而类型能够告诉系统这个变量所占用的空间

// int型（整型）变量一般占用4个字节
int x_int = 0;

// short型（短整型）变量一般占用2个字节
short x_short = 0;

// char型（字符型）变量会占用1个字节
char x_char = 0;
char y_char = 'y'; // 字符变量的字面值需要用单引号包住

// long型（长整型）一般需要4个字节到8个字节；而long long型则至少需要8个字节（64位）

long x_long = 0;
long long x_long_long = 0;

// float一般是用32位表示的浮点数字
float x_float = 0.0;

// double一般是用64位表示的浮点数字
double x_double = 0.0;

// 整数类型也可以有无符号的类型表示。这样这些变量就无法表示负数
// 但是无符号整数所能表示的范围就可以比原来的整数大一些

unsigned short ux_short;
unsigned int ux_int;
unsigned long long ux_long_long;

// 单引号内的字符是机器的字符集中的整数。
'0' // => 在ASCII字符集中是48
'A' // => 在ASCII字符集中是65

// char类型一定会占用1个字节，但是其他的类型却会因具体机器的不同而各异
// sizeof(T) 可以返回T类型在运行的机器上占用多少个字节
// 这样你的代码就可以在各处正确运行了

```

```

// sizeof(obj)返回表达式（变量、字面量等）的尺寸
printf("%zu\n", sizeof(int)); // => 4 (大多数的机器字长为4)

// 如果`sizeof`的参数是一个表达式，那么这个参数不会被演算（VLA例外，见下）
// 它产生的值是编译期的常数
int a = 1;
// size_t是一个无符号整型，表示对象的尺寸，至少2个字节
size_t size = sizeof(a++); // a++ 不会被演算
printf("sizeof(a++) = %zu where a = %d\n", size, a);
// 打印 "sizeof(a++) = 4 where a = 1" （在32位架构上）

// 数组必须要被初始化为具体的长度
char my_char_array[20]; // 这个数组占据 1 * 20 = 20 个字节
int my_int_array[20]; // 这个数组占据 4 * 20 = 80 个字节
// （这里我们假设字长为4）

// 可以用下面的方法把数组初始化为0:
char my_array[20] = {0};

// 索引数组和其他语言类似 -- 好吧，其实是其他的语言像C
my_array[0]; // => 0

// 数组是可变的，其实就是内存的映射！
my_array[1] = 2;
printf("%d\n", my_array[1]); // => 2

// 在C99 （C11中是可选特性），变长数组（VLA）也可以声明长度。
// 其长度不用是编译期常量。
printf("Enter the array size: "); // 询问用户数组长度
char buf[0x100];
fgets(buf, sizeof buf, stdin);

// strtoul 将字符串解析为无符号整数
size_t size = strtoul(buf, NULL, 10);
int var_length_array[size]; // 声明VLA
printf("sizeof array = %zu\n", sizeof var_length_array);

// 上述程序可能的输出为:
// > Enter the array size: 10
// > sizeof array = 40

// 字符串就是以 NUL (0x00) 这个字符结尾的字符数组，
// NUL可以用'\0'来表示。
// （在字符串字面量中我们不必输入这个字符，编译器会自动添加的）
char a_string[20] = "This is a string";
printf("%s\n", a_string); // %s 可以对字符串进行格式化
/*
也许你会注意到 a_string 实际上只有16个字节长。
第17个字节是一个空字符(NUL)
而第18, 19 和 20 个字符的值是未定义。

```

```

*/

printf("%d\n", a_string[16]); // => 0
// byte #17值为0 (18, 19, 20同样为0)

// 单引号间的字符是字符常量
// 它的类型是`int`, 而 *不是* `char`
// (由于历史原因)
int cha = 'a'; // 合法
char chb = 'a'; // 同样合法 (隐式类型转换)

// 多维数组
int multi_array[2][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 0}
}
// 获取元素
int array_int = multi_array[0][2]; // => 3

////////////////////////////////////
// 操作符
////////////////////////////////////

// 多个变量声明的简写
int i1 = 1, i2 = 2;
float f1 = 1.0, f2 = 2.0;

int a, b, c;
a = b = c = 0;

// 算数运算直截了当
i1 + i2; // => 3
i2 - i1; // => 1
i2 * i1; // => 2
i1 / i2; // => 0 (0.5, 但会被化整为 0)

f1 / f2; // => 0.5, 也许会有很小的误差
// 浮点数和浮点数运算都是近似值

// 取余运算
11 % 3; // => 2

// 你多半会觉得比较操作符很熟悉, 不过C中没有布尔类型
// 而是用整形替代
// (C99中有_Bool或bool。)
// 0为假, 其他均为真. (比较操作符的返回值总是返回0或1)
3 == 2; // => 0 (false)
3 != 2; // => 1 (true)
3 > 2; // => 1
3 < 2; // => 0
2 <= 2; // => 1

```

```

2 >= 2; // => 1

// C不是Python — 连续比较不合法
int a = 1;
// 错误
int between_0_and_2 = 0 < a < 2;
// 正确
int between_0_and_2 = 0 < a && a < 2;

// 逻辑运算符适用于整数
!3; // => 0 (非)
!0; // => 1
1 && 1; // => 1 (且)
0 && 1; // => 0
0 || 1; // => 1 (或)
0 || 0; // => 0

// 条件表达式 ( ? : )
int a = 5;
int b = 10;
int z;
z = (a > b) ? a : b; // 10 “若a > b返回a, 否则返回b。”

// 增、减
char *s = "iLoveC"
int j = 0;
s[j++]; // "i" 返回s的第j项, 然后增加j的值。
j = 0;
s[++j]; // => "L" 增加j的值, 然后返回s的第j项。
// j-- 和 --j 同理

// 位运算
~0x0F; // => 0xF0 (取反)
0x0F & 0xF0; // => 0x00 (和)
0x0F | 0xF0; // => 0xFF (或)
0x04 ^ 0x0F; // => 0x0B (异或)
0x01 << 1; // => 0x02 (左移1位)
0x02 >> 1; // => 0x01 (右移1位)

// 对有符号整数进行移位操作要小心 — 以下未定义:
// 有符号整数位移至符号位 int a = 1 << 32
// 左移位一个负数 int a = -1 << 2
// 移位超过或等于该类型数值的长度
// int a = 1 << 32; // 假定int32位

////////////////////////////////////
// 控制结构
////////////////////////////////////

if (0) {

```

```

    printf("I am never run\n");
} else if (0) {
    printf("I am also never run\n");
} else {
    printf("I print\n");
}

// While循环
int ii = 0;
while (ii < 10) { // 任何非0的值均为真
    printf("%d, ", ii++); // ii++ 在取值过后自增
} // => 打印 "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

int kk = 0;
do {
    printf("%d, ", kk);
} while (++kk < 10); // ++kk 先自增, 再被取值
// => 打印 "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

// For 循环
int jj;
for (jj=0; jj < 10; jj++) {
    printf("%d, ", jj);
} // => 打印 "0, 1, 2, 3, 4, 5, 6, 7, 8, 9, "

printf("\n");

// *****注意*****:
// 循环和函数必须有主体部分, 如果不需要主体部分:
int i;
    for (i = 0; i <= 5; i++) {
        ; // 使用分号表达主体 (null语句)
    }

// 多重分支: switch()
switch (some_integral_expression) {
case 0: // 标签必须是整数常量表达式
    do_stuff();
    break; // 如果不使用break, 控制结构会继续执行下面的标签
case 1:
    do_something_else();
    break;
default:
    // 假设 `some_integral_expression` 不匹配任何标签
    fputs("error!\n", stderr);
    exit(-1);
    break;

```

```

    }

////////////////////////////////////
// 类型转换
////////////////////////////////////

// 在C中每个变量都有类型，你可以将变量的类型进行转换
// （有一定限制）

int x_hex = 0x01; // 可以用16进制字面量赋值

// 在类型转换时，数字本身的值会被保留下来
printf("%d\n", x_hex); // => 打印 1
printf("%d\n", (short) x_hex); // => 打印 1
printf("%d\n", (char) x_hex); // => 打印 1

// 类型转换时可能会造成溢出，而且不会抛出警告
printf("%d\n", (char) 257); // => 1 (char的最大值为255，假定char为8位长)

// 使用<limits.h>提供的CHAR_MAX、SCHAR_MAX和UCHAR_MAX宏可以确定`char`、`signed_char`和`unisi

// 整数型和浮点型可以互相转换
printf("%f\n", (float)100); // %f 格式化单精度浮点
printf("%lf\n", (double)100); // %lf 格式化双精度浮点
printf("%d\n", (char)100.0);

////////////////////////////////////
// 指针
////////////////////////////////////

// 指针变量是用来储存内存地址的变量
// 指针变量的声明也会告诉它所指向的数据的类型
// 你可以使用得到你的变量的地址，并把它们搞乱，;-)

int x = 0;
printf("%p\n", &x); // 用 & 来获取变量的地址
// (%p 格式化一个类型为 void *的指针)
// => 打印某个内存地址

// 指针类型在声明中以*开头
int* px, not_a_pointer; // px是一个指向int型的指针
px = &x; // 把x的地址保存到px中
printf("%p\n", (void *)px); // => 输出内存中的某个地址
printf("%zu, %zu\n", sizeof(px), sizeof(not_a_pointer));
// => 在64位系统上打印“8, 4”。

// 要得到某个指针指向的内容的值，可以在指针前加一个*来取得（取消引用）
// 注意： 是的，这可能让人困惑，'*'在用来声明一个指针的同时取消引用它。
printf("%d\n", *px); // => 输出 0，即x的值

```

```

// 你也可以改变指针所指向的值
// 此时你需要取消引用上添加括号，因为++比*的优先级更高
(*px)++; // 把px所指向的值增加1
printf("%d\n", *px); // => 输出 1
printf("%d\n", x); // => 输出 1

// 数组是分配一系列连续空间的常用方式
int x_array[20];
int xx;
for (xx=0; xx<20; xx++) {
    x_array[xx] = 20 - xx;
} // 初始化 x_array 为 20, 19, 18, ... 2, 1

// 声明一个整型的指针，并初始化为指向x_array
int* x_ptr = x_array;
// x_ptr现在指向了数组的第一个元素(即整数20)。
// 这是因为数组通常衰减为指向它们的第一个元素的指针。
// 例如，当一个数组被传递给一个函数或者绑定到一个指针时，
// 它衰减为(隐式转化为)一个指针。
// 例外：当数组是`&`操作符的参数：
int arr[10];
int (*ptr_to_arr)[10] = &arr; // &arr的类型不是`int *`！
                                // 它的类型是指向数组的指针（数组由10个int组成）
// 或者当数组是字符串字面量（初始化字符数组）
char arr[] = "foobazquirk";
// 或者当它是`sizeof`或`alignof`操作符的参数时：
int arr[10];
int *ptr = arr; // 等价于 int *ptr = &arr[0];
printf("%zu, %zu\n", sizeof arr, sizeof ptr); // 应该会输出"40, 4"或"40, 8"

// 指针的增减多少是依据它本身的类型而定的
// （这被称为指针算术）
printf("%d\n", *(x_ptr + 1)); // => 打印 19
printf("%d\n", x_array[1]); // => 打印 19

// 你也可以通过标准库函数malloc来实现动态分配
// 这个函数接受一个代表容量的参数，参数类型为`size_t`
// 系统一般会从堆区分配指定容量字节大小的空间
// （在一些系统，例如嵌入式系统中这点不一定成立
// C标准对此未置一词。）
int *my_ptr = malloc(sizeof(*my_ptr) * 20);
for (xx=0; xx<20; xx++) {
    *(my_ptr + xx) = 20 - xx; // my_ptr[xx] = 20-xx
} // 初始化内存为 20, 19, 18, 17... 2, 1 (类型为int)

// 对未分配的内存进行取消引用会产生未定义的结果
printf("%d\n", *(my_ptr + 21)); // => 谁知道会输出什么

// malloc分配的区域需要手动释放
// 否则没人能够再次使用这块内存，直到程序结束为止
free(my_ptr);

```

```

// 字符串通常是字符数组，但是经常用字符指针表示
// (它是指向数组的第一个元素的指针)
// 一个优良的实践是使用`const char *`来引用一个字符串字面量，
// 因为字符串字面量不应当被修改（即"foo"[0] = 'a'犯了大忌）
const char* my_str = "This is my very own string";
printf("%c\n", *my_str); // => 'T'

// 如果字符串是数组，（多半是用字符串字面量初始化的）
// 情况就不一样了，字符串位于可写的内存中
char foo[] = "foo";
foo[0] = 'a'; // 这是合法的，foo现在包含"aoo"

function_1();
} // main函数结束

////////////////////////////////////
// 函数
////////////////////////////////////

// 函数声明语法：
// <返回值类型> <函数名称>(<参数>)

int add_two_ints(int x1, int x2){
    return x1 + x2; // 用return来返回一个值
}

/*
函数是按值传递的。当调用一个函数的时候，传递给函数的参数
是原有值的拷贝（数组除外）。你在函数内对参数所进行的操作
不会改变该参数原有的值。

但是你可以通过指针来传递引用，这样函数就可以更改值

例子：字符串本身翻转
*/

// 类型为void的函数没有返回值
void str_reverse(char *str_in){
    char tmp;
    int ii = 0;
    size_t len = strlen(str_in); // `strlen()` 是C标准库函数
    for(ii = 0; ii < len / 2; ii++){
        tmp = str_in[ii];
        str_in[ii] = str_in[len - ii - 1]; // 从倒数第ii个开始
        str_in[len - ii - 1] = tmp;
    }
}

/*
char c[] = "This is a test.";

```



```

str_reverse(c);
printf("%s\n", c); // => ".tset a si sihT"
*/

// 如果引用函数之外的变量，必须使用extern关键字
int i = 0;
void testFunc() {
    extern int i; // 使用外部变量 i
}

// 使用static确保external变量为源文件私有
static int i = 0; // 其他使用 testFunc()的文件无法访问变量i
void testFunc() {
    extern int i;
}
/**你同样可以声明函数为static**

////////////////////////////////////
// 用户自定义类型和结构
////////////////////////////////////

// Typedefs可以创建类型别名
typedef int my_type;
my_type my_type_var = 0;

// struct是数据的集合，成员依序分配，按照
// 编写的顺序
struct rectangle {
    int width;
    int height;
};

// 一般而言，以下断言不成立：
// sizeof(struct rectangle) == sizeof(int) + sizeof(int)
//这是因为structure成员之间可能存在潜在的间隙（为了对齐）[1]

void function_1(){

    struct rectangle my_rec;

    // 通过 . 来访问结构中的数据
    my_rec.width = 10;
    my_rec.height = 20;

    // 你也可以声明指向结构体的指针
    struct rectangle *my_rec_ptr = &my_rec;

    // 通过取消引用来改变结构体的成员...
    (*my_rec_ptr).width = 30;

```

```

// ... 或者用 -> 操作符作为简写提高可读性
my_rec_ptr->height = 10; // Same as (*my_rec_ptr).height = 10;
}

// 你也可以用typedef来给一个结构体起一个别名
typedef struct rectangle rect;

int area(rect r){
    return r.width * r.height;
}

// 如果struct较大,你可以通过指针传递,避免
// 复制整个struct。
int area(const rect *r)
{
    return r->width * r->height;
}

////////////////////////////////////
// 函数指针
////////////////////////////////////
/*
在运行时,函数本身也被存放到某块内存区域当中
函数指针就像其他指针一样(不过是存储一个内存地址) 但却可以被用来直接调用函数,
并且可以四处传递回调函数
但是,定义的语法初看令人有些迷惑

例子: 通过指针调用str_reverse
*/
void str_reverse_through_pointer(char *str_in) {
    // 定义一个函数指针 f.
    void (*f)(char *); // 签名一定要与目标函数相同
    f = &str_reverse; // 将函数的地址在运行时赋给指针
    (*f)(str_in); // 通过指针调用函数
    // f(str_in); // 等价于这种调用方式
}

/*
只要函数签名是正确的,任何时候都能将任何函数赋给某个函数指针
为了可读性和简洁性,函数指针经常和typedef搭配使用:
*/

typedef void (*my_fnp_type)(char *);

// 实际声明函数指针会这么用:
// ...
// my_fnp_type f;

// 特殊字符
'\a' // bell
'\n' // 换行

```

```
'\t' // tab
'\v' // vertical tab
'\f' // formfeed
'\r' // 回车
'\b' // 退格
'\0' // null, 通常置于字符串的最后。
      //  hello\n\0. 按照惯例, \0用于标记字符串的末尾。
'\' ' // 反斜杠
'\?' // 问号
'\'' // 单引号
'\'' // 双引号
'\xhh' // 十六进制数字. 例子: '\xb' = vertical tab
'\ooo' // 八进制数字. 例子: '\013' = vertical tab
```

// 打印格式:

```
"%d"    // 整数
"%3d"   // 3位以上整数 (右对齐文本)
"%s"    // 字符串
"%f"    // float
"%ld"   // long
"%3.2f" // 左3位以上、右2位以上十进制浮
"%7.4s" // (字符串同样适用)
"%c"    // 字母
"%p"    // 指针
"%x"    // 十六进制
"%o"    // 八进制
"%"     // 打印 %
```

```
////////////////////////////////////
```

// 演算优先级

```
////////////////////////////////////
```

```
//-----//
```

//	操作符	组合	//
----	-----	----	----

```
//-----//
```

// () [] -> .	从左到右	//
-----------------	------	----

// ! ~ ++ -- + = *(type)sizeof	从右到左	//
--------------------------------	------	----

// * / %	从左到右	//
----------	------	----

// + -	从左到右	//
--------	------	----

// << >>	从左到右	//
----------	------	----

// < <= > >=	从左到右	//
--------------	------	----

// == !=	从左到右	//
----------	------	----

// &	从左到右	//
------	------	----

// ^	从左到右	//
------	------	----

//	从左到右	//
----	------	----

// &&	从左到右	//
-------	------	----

//	从左到右	//
----	------	----

// ?:	从右到左	//
-------	------	----

// = += -= *= /= %= &= ^= = <<= >>=	从右到左	//
--------------------------------------	------	----

// ,	从左到右	//
------	------	----

```
//-----//
```

更多阅读

最好找一本 [K&R, aka "The C Programming Language"](#), “C程序设计语言”。它是关于C最重要的一本书，由C的创作者撰写。不过需要留意的是它比较古老了，因此有些不准确的地方。

另一个比较好的资源是 [Learn C the hard way](#)

如果你有问题，请阅读[compl.lang.c Frequently Asked Questions](#)。

使用合适的空格、缩进，保持一致的代码风格非常重要。可读性强的代码比聪明的代码、高速的代码更重要。可以参考下[Linux内核编码风格](#)。除了这些，多多Google吧

[1] <http://stackoverflow.com/questions/119123/why-isnt-sizeof-for-a-struct-equal-to-the-sum-of-sizeof-of-each-member>

language: clojure filename: learnclojure-cn.clj contributors:

```
- ["Adam Bard", "http://adambard.com/"]
```

translators:

```
- ["Bill Zhang", "http://jingege.github.io/"]
```

lang: zh-cn

Clojure是运行在JVM上的Lisp家族中的一员。她比Common Lisp更强调纯[函数式编程](#)，且自发布时便包含了一组工具来处理状态。

这种组合让她能十分简单且自动地处理并发问题。

(你需要使用Clojure 1.2或更新的发行版)

； 注释以分号开始。

； Clojure代码由一个个form组成， 即写在小括号里的由空格分开的一组语句。

； Clojure解释器会把第一个元素当做一个函数或者宏来调用，其余的被认为是参数。

； Clojure代码的第一条语句一般是用ns来指定当前的命名空间。

(ns learnclojure)

； 更基本的例子：

； str会使用所有参数来创建一个字符串

(str "Hello" " " "World") ; => "Hello World"

； 数学计算比较直观

(+ 1 1) ; => 2

(- 2 1) ; => 1

(* 1 2) ; => 2

(/ 2 1) ; => 2

； 等号是 =

(= 1 1) ; => true

(= 2 1) ; => false

； 逻辑非

(not true) ; => false

； 嵌套的form工作起来应该和你预想的一样

(+ 1 (- 3 2)) ; = 1 + (3 - 2) => 2

```

; 类型
;;;;;;;;;;;;;;;;;;

; Clojure使用Java的Object来描述布尔值、字符串和数字
; 用函数 `class` 来查看具体的类型
(class 1) ; 整形默认是java.lang.Long类型
(class 1.); 浮点默认是java.lang.Double类型的
(class ""); String是java.lang.String类型的, 要用双引号引起来
(class false) ; 布尔值是java.lang.Boolean类型的
(class nil); "null"被称作nil

; 如果你想创建一组数据字面量, 用单引号(')来阻止form被解析和求值
'+ 1 2) ; => (+ 1 2)
; (单引号是quote的简写形式, 故上式等价于(quote (+ 1 2)))

; 可以对一个引用列表求值
(eval '+ 1 2)) ; => 3

; 集合 (Collection) 和序列
;;;;;;;;;;;;;;;;;;;;;;;;;;

; List的底层实现是链表, Vector的底层实现是数组
; 二者也都是java类
(class [1 2 3]); => clojure.lang.PersistentVector
(class '(1 2 3)); => clojure.lang.PersistentList

; list本可以写成(1 2 3), 但必须用引用来避免被解释器当做函数来求值。
; (list 1 2 3)等价于'(1 2 3)

; 集合其实就是一组数据
; List和Vector都是集合:
(coll? '(1 2 3)) ; => true
(coll? [1 2 3]) ; => true

; 序列 (seqs) 是数据列表的抽象描述
; 只有列表才可称作序列。
(seq? '(1 2 3)) ; => true
(seq? [1 2 3]) ; => false

; 序列被访问时只需要提供一个值, 所以序列可以被懒加载——也就意味着可以定义一个无限序列:
(range 4) ; => (0 1 2 3)
(range) ; => (0 1 2 3 4 ...) (无限序列)
(take 4 (range)) ; (0 1 2 3)

; cons用以向列表或向量的起始位置添加元素
(cons 4 [1 2 3]) ; => (4 1 2 3)
(cons 4 '(1 2 3)) ; => (4 1 2 3)

; conj将以最高效的方式向集合中添加元素。
; 对于列表, 数据会在起始位置插入, 而对于向量, 则在末尾位置插入。
(conj [1 2 3] 4) ; => [1 2 3 4]

```

```

(conj '(1 2 3) 4) ; => (4 1 2 3)

; 用concat来合并列表或向量
(concat [1 2] '(3 4)) ; => (1 2 3 4)

; 用filter来过滤集合中的元素，用map来根据指定的函数来映射得到一个新的集合
(map inc [1 2 3]) ; => (2 3 4)
(filter even? [1 2 3]) ; => (2)

; reduce使用函数来规约集合
(reduce + [1 2 3 4])
; = (+ (+ (+ 1 2) 3) 4)
; => 10

; reduce还能指定一个初始参数
(reduce conj [] '(3 2 1))
; = (conj (conj (conj [] 3) 2) 1)
; => [3 2 1]

; 函数
;;;;;;;;;;;;;;;;;;;;;;;;;;

; 用fn来创建函数。函数的返回值是最后一个表达式的值
(fn [] "Hello World") ; => fn

; (你需要再嵌套一组小括号来调用它)
((fn [] "Hello World")) ; => "Hello World"

; 你可以用def来创建一个变量 (var)
(def x 1)
x ; => 1

; 将函数定义为一个变量 (var)
(def hello-world (fn [] "Hello World"))
(hello-world) ; => "Hello World"

; 你可用defn来简化函数的定义
(defn hello-world [] "Hello World")

; 中括号内的内容是函数的参数。
(defn hello [name]
  (str "Hello " name))
(hello "Steve") ; => "Hello Steve"

; 你还可以用这种简写的方式来创建函数：
(def hello2 #(str "Hello " %1))
(hello2 "Fanny") ; => "Hello Fanny"

; 函数也可以有多个参数列表。
(defn hello3
  ([] "Hello World")

```

```
    ([name] (str "Hello " name)))  
(hello3 "Jake") ; => "Hello Jake"  
(hello3) ; => "Hello World"
```

； 可以定义变参函数，即把&后面的参数全部放入一个序列

```
(defn count-args [& args]  
  (str "You passed " (count args) " args: " args))  
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"
```

； 可以混用定参和变参（用&来界定）

```
(defn hello-count [name & args]  
  (str "Hello " name ", you passed " (count args) " extra args"))  
(hello-count "Finn" 1 2 3)  
; => "Hello Finn, you passed 3 extra args"
```

； 哈希表

```
;;;;;;;;;;
```

； 基于hash的map和基于数组的map（即arraymap）实现了相同的接口，hashmap查询起来比较快，

； 但不保证元素的顺序。

```
(class {:a 1 :b 2 :c 3}) ; => clojure.lang.PersistentArrayMap  
(class (hash-map :a 1 :b 2 :c 3)) ; => clojure.lang.PersistentHashMap
```

； arraymap在足够大的时候，大多数操作会将其自动转换成hashmap，

； 所以不用担心(对大的arraymap的查询性能)。

； map支持很多类型的key，但推荐使用keyword类型

； keyword类型和字符串类似，但做了一些优化。

```
(class :a) ; => clojure.lang.Keyword
```

```
(def stringmap {"a" 1, "b" 2, "c" 3})  
stringmap ; => {"a" 1, "b" 2, "c" 3}
```

```
(def keymap {:a 1, :b 2, :c 3})  
keymap ; => {:a 1, :c 3, :b 2}
```

； 顺便说一下，map里的逗号是可有可无的，作用只是提高map的可读性。

； 从map中查找元素就像把map名作为函数调用一样。

```
(stringmap "a") ; => 1  
(keymap :a) ; => 1
```

； 可以把keyword写在前面来从map中查找元素。

```
(:b keymap) ; => 2
```

； 但不要试图用字符串类型的key来这么做。

```
("a" stringmap)  
; => Exception: java.lang.String cannot be cast to clojure.lang.IFn
```

； 查找不存在的key会返回nil。


```

(stringmap "d") ; => nil

; 用assoc函数来向hashmap里添加元素
(def newkeymap (assoc keymap :d 4))
newkeymap ; => {:a 1, :b 2, :c 3, :d 4}

; 但是要记住的是clojure的数据类型是不可变的!
keymap ; => {:a 1, :b 2, :c 3}

; 用dissoc来移除元素
(dissoc keymap :a :b) ; => {:c 3}

; 集合 (Set)
;;;;;

(class #{1 2 3}) ; => clojure.lang.PersistentHashSet
(set [1 2 3 1 2 3 3 2 1 3 2 1]) ; => #{1 2 3}

; 用conj新增元素
(conj #{1 2 3} 4) ; => #{1 2 3 4}

; 用disj移除元素
(disj #{1 2 3} 1) ; => #{2 3}

; 把集合当做函数调用来检查元素是否存在:
(#{1 2 3} 1) ; => 1
(#{1 2 3} 4) ; => nil

; 在clojure.sets模块下有很多相关函数。

; 常用的form
;;;;;;;;;;;;;;;;;;;;;;;;;

; clojure里的逻辑控制结构都是用宏 (macro) 实现的, 这在语法上看起来没什么不同。
(if false "a" "b") ; => "b"
(if false "a") ; => nil

; 用let来创建临时的绑定变量。
(let [a 1 b 2]
  (> a b)) ; => false

; 用do将多个语句组合在一起依次执行
(do
  (print "Hello")
  "World") ; => "World" (prints "Hello")

; 函数定义里有一个隐式的do
(defn print-and-say-hello [name]
  (print "Saying hello to " name)
  (str "Hello " name))
(print-and-say-hello "Jeff") ;=> "Hello Jeff" (prints "Saying hello to Jeff")

```

```

; let也是如此
(let [name "Urkel"]
  (print "Saying hello to " name)
  (str "Hello " name)) ; => "Hello Urkel" (prints "Saying hello to Urkel")

; 模块
;;;;;;;;;;;;;;;;;;

; 用use来导入模块里的所有函数
(use 'clojure.set)

; 然后就可以使用set相关的函数了
(intersection #{1 2 3} #{2 3 4}) ; => #{2 3}
(difference #{1 2 3} #{2 3 4}) ; => #{1}

; 你也可以从一个模块里导入一部分函数。
(use '[clojure.set :only [intersection]])

; 用require来导入一个模块
(require 'clojure.string)

; 用/来调用模块里的函数
; 下面是从模块`clojure.string`里调用`blank?`函数。
(clojure.string/blank? "") ; => true

; 在`import`里你可以给模块名指定一个较短的别名。
(require '[clojure.string :as str])
(str/replace "This is a test." #"[a-o]" str/upper-case) ; => "THIs Is A tEst."
; (#"用来表示一个正则表达式)

; 你可以在一个namespace定义里用:require的方式来require（或use，但最好不要用）模块。
; 这样的话你无需引用模块列表。
(ns test
  (:require
    [clojure.string :as str]
    [clojure.set :as set]))

; Java
;;;;;;;;;;;;;;;;;;

; Java有大量的优秀的库，你肯定想学会如何用clojure来使用这些Java库。

; 用import来导入java类
(import java.util.Date)

; 也可以在ns定义里导入
(ns test
  (:import java.util.Date
            java.util.Calendar))

```

```

; 用类名末尾加`. `的方式来new一个Java对象
(Date.) ; <a date object>

; 用`. `操作符来调用方法, 或者用`.method`的简化方式。
(. (Date.) getTime) ; <a timestamp>
(.getTime (Date.)) ; 和上例一样。

; 用`/`调用静态方法
(System/currentTimeMillis) ; <a timestamp> (system is always present)

; 用`doto`来更方便的使用(可变)类。
(import java.util.Calendar)
(doto (Calendar/getInstance)
  (.set 2000 1 1 0 0 0)
  .getTime) ; => A Date. set to 2000-01-01 00:00:00

; STM
;;;;;;;;;;;;;;

; 软件内存事务(Software Transactional Memory)被clojure用来处理持久化的状态。
; clojure里内置了一些结构来使用STM。
; atom是最简单的。给它传一个初始值
(def my-atom (atom {}))

; 用`swap!`更新atom。
; `swap!`会以atom的当前值为第一个参数来调用一个指定的函数,
; `swap!`其余的参数作为该函数的第二个参数。
(swap! my-atom assoc :a 1) ; Sets my-atom to the result of (assoc {} :a 1)
(swap! my-atom assoc :b 2) ; Sets my-atom to the result of (assoc {:a 1} :b 2)

; 用`@`读取atom的值
my-atom ;=> Atom<#...> (返回Atom对象)
@my-atom ;=> {:a 1 :b 2}

; 下例是一个使用atom实现的简单计数器
(def counter (atom 0))
(defn inc-counter []
  (swap! counter inc))

(inc-counter)
(inc-counter)
(inc-counter)
(inc-counter)
(inc-counter)

@counter ;=> 5

; 其他STM相关的结构是ref和agent.
; Refs: http://clojure.org/refs
; Agents: http://clojure.org/agents

```

进阶读物

本文肯定不足以讲述关于clojure的一切，但是希望足以让你迈出第一步。

Clojure.org官网有很多文章: <http://clojure.org/>

Clojuredocs.org有大多数核心函数的文档，还带了示例哦:

<http://clojuredocs.org/quickref/Clojure%20Core>

4Clojure是个很赞的用来练习clojure/FP技能的地方: <http://www.4clojure.com/>

Clojure-doc.org (你没看错)有很多入门级的文章: <http://clojure-doc.org/>

language: "clojure macros" filename: learnclojuremacros-zh.clj contributors:

```
- ["Adam Bard", "http://adambard.com/"]
```

translators:

```
- ["Jakukyo Friel", "http://weakish.github.io"]
```

lang: zh-cn

和所有Lisp一样，Clojure内在的[同构性](#)使得你可以穷尽语言的特性，编写生成代码的子过程——“宏”。宏是一种按需调制语言的强大方式。

小心！可以用函数完成的事用宏去实现可不是什么好事。你应该仅在需要控制参数是否或者何时eval的时候使用宏。

你应该熟悉Clojure.确保你了解[Y分钟学Clojure](#)中的所有内容。

```
;; 使用defmacro定义宏。宏应该输出一个可以作为clojure代码演算的列表。
;;
;; 以下宏的效果和直接写(reverse "Hello World")一致。

(defmacro my-first-macro []
  (list reverse "Hello World"))

;; 使用macroexpand或macroexpand-1查看宏的结果。
;;
;; 注意，调用需要引用。
(macroexpand '(my-first-macro))
;; -> (#<core$reverse clojure.core$reverse@xxxxxxxx> "Hello World")

;; 你可以直接eval macroexpand的结果
(eval (macroexpand '(my-first-macro)))
;; -> (\d \l \o \r \W \space \o \l \l \e \H)

;; 不过一般使用以下形式，更简短，更像函数：
(my-first-macro) ; -> (\d \l \o \r \W \space \o \l \l \e \H)

;; 创建宏的时候可以使用更简短的引用形式来创建列表
(defmacro my-first-quoted-macro []
  '(reverse "Hello World"))

(macroexpand '(my-first-quoted-macro))
;; -> (reverse "Hello World")
;; 注意reverse不再是一个函数对象，而是一个符号。
```

```
;; 宏可以传入参数。
(defmacro inc2 [arg]
  (list + 2 arg))

(inc2 2) ; -> 4

;; 不过，如果你尝试配合使用引用列表，会导致错误，
;; 因为参数也会被引用。
;; 为了避免这个问题，clojure提供了引用宏的另一种方式：`
;; 在`之内，你可以使用~获得外圈作用域的变量。
(defmacro inc2-quoted [arg]
  `( + 2 ~arg))

(inc2-quoted 2)

;; 你可以使用通常的析构参数。用~@展开列表中的变量。
(defmacro unless [arg & body]
  `(if (not ~arg)
      (do ~@body))) ; 别忘了 do!

(macroexpand '(unless true (reverse "Hello World")))

;; ->
;; (if (clojure.core/not true) (do (reverse "Hello World")))

;; 当第一个参数为假时，(unless)会演算、返回主体。
;; 否则返回nil。

(unless true "Hello") ; -> nil
(unless false "Hello") ; -> "Hello"

;; 需要小心，宏会搞乱你的变量
(defmacro define-x []
  '(do
    (def x 2)
    (list x)))

(def x 4)
(define-x) ; -> (2)
(list x) ; -> (2)

;; 使用gensym来获得独有的标识符
(gensym 'x) ; -> x1281 (or some such thing)

(defmacro define-x-safely []
  (let [sym (gensym 'x)]
    `(do
      (def ~sym 2)
      (list ~sym))))

(def x 4)
```

```

(define-x-safely) ; -> (2)
(list x) ; -> (4)

;; 你可以在 ` 中使用 # 为每个符号自动生成gensym
(defmacro define-x-hygenically []
  `(do
    (def x# 2)
    (list x#)))

(def x 4)
(define-x-hygenically) ; -> (2)
(list x) ; -> (4)

;; 通常会配合宏使用帮助函数。
;; 让我们创建一些帮助函数来支持（无聊的）算术语法：

(declare inline-2-helper)
(defn clean-arg [arg]
  (if (seq? arg)
    (inline-2-helper arg)
    arg))

(defn apply-arg
  "Given args [x (+ y)], return (+ x y)"
  [val [op arg]]
  (list op val (clean-arg arg)))

(defn inline-2-helper
  [[arg1 & ops-and-args]]
  (let [ops (partition 2 ops-and-args)]
    (reduce apply-arg (clean-arg arg1) ops)))

;; 在创建宏前，我们可以先测试
(inline-2-helper '(a + (b - 2) - (c * 5))) ; -> (- (+ a (- b 2)) (* c 5))

; 然而，如果我们希望它在编译期执行，就需要创建宏
(defmacro inline-2 [form]
  (inline-2-helper form))

(macroexpand '(inline-2 (1 + (3 / 2) - (1 / 2) + 1)))
; -> (+ (- (+ 1 (/ 3 2)) (/ 1 2)) 1)

(inline-2 (1 + (3 / 2) - (1 / 2) + 1))
; -> 3（事实上，结果是3N，因为数字被转化为带/的有理分数）

```

扩展阅读

Clojure for the Brave and True系列的编写宏 <http://www.braveclojure.com/writing-macros/>

官方文档 <http://clojure.org/macros>

何时使用宏? http://dunsmor.com/lisp/onlisp/onlisp_12.html

language: coffeescript contributors:

- ["Tenor Biel", "<http://github.com/L8D>"]
 - ["Xavier Yao", "<http://github.com/xavieryao>"] translators:
 - ["Xavier Yao", "<http://github.com/xavieryao>"] filename: coffeescript-cn.coffee lang: zh-cn
-

CoffeeScript是逐句编译为JavaScript的一种小型语言，且没有运行时的解释器。作为JavaScript的替代品之一，CoffeeScript旨在编译人类可读、美观优雅且速度不输原生的代码，且编译后的代码可以在任何JavaScript运行时正确运行。

参阅 [CoffeeScript官方网站](#) 以获取CoffeeScript的完整教程。

```
# CoffeeScript是一种很潮的编程语言，
# 它紧随众多现代编程语言的趋势。
# 因此正如Ruby和Python，CoffeeScript使用井号标记注释。

###
大段落注释以此为例，可以被直接编译为 '/' *' 和 '* /' 包裹的JavaScript代码。

在继续之前你需要了解JavaScript的基本概念。

示例中 => 后为编译后的JavaScript代码
###

# 赋值：
number    = 42  #=> var number = 42;
opposite  = true #=> var opposite = true;

# 条件：
number = -42 if opposite #=> if(opposite) { number = -42; }

# 函数：
square = (x) -> x * x #=> var square = function(x) { return x * x; }

fill = (container, liquid = "coffee") ->
  "Filling the #{container} with #{liquid}..."
#=>var fill;
#
#fill = function(container, liquid) {
#  if (liquid == null) {
#    liquid = "coffee";
#  }
#  return "Filling the " + container + " with " + liquid + "...";
#};

# 区间：
```

```

list = [1..5] #=> var list = [1, 2, 3, 4, 5];

# 对象:
math =
  root: Math.sqrt
  square: square
  cube: (x) -> x * square x
#=> var math = {
#   "root": Math.sqrt,
#   "square": square,
#   "cube": function(x) { return x * square(x); }
#}

# Splat:
race = (winner, runners...) ->
  print winner, runners
#=> race = function() {
#   var runners, winner;
#   winner = arguments[0], runners = 2 <= arguments.length ? __slice.call(arguments, 1) :
#   return print(winner, runners);
#};

# 存在判断:
alert "I knew it!" if elvis?
#=> if(typeof elvis !== "undefined" && elvis !== null) { alert("I knew it!"); }

# 数组推导:
cubes = (math.cube num for num in list)
#=> cubes = (function() {
#   var _i, _len, _results;
#   _results = [];
#   for (_i = 0, _len = list.length; _i < _len; _i++) {
#     num = list[_i];
#     _results.push(math.cube(num));
#   }
#   return _results;
# })();

foods = ['broccoli', 'spinach', 'chocolate']
eat food for food in foods when food isnt 'chocolate'
#=> foods = ['broccoli', 'spinach', 'chocolate'];
#
# for (_k = 0, _len2 = foods.length; _k < _len2; _k++) {
#   food = foods[_k];
#   if (food !== 'chocolate') {
#     eat(food);
#   }
# }
#}

```

language: "Common Lisp" filename: commonlisp-cn.lisp contributors:

- ["Paul Nathan", "<https://github.com/pnathan>"] translators:
 - ["Mac David", "<http://macdavid313.com>"]
 - ["mut0u", "<http://github.com/mut0u>"] lang: zh-cn
-

ANSI Common Lisp 是一个广泛通用于各个工业领域的、支持多种范式的编程语言。这门语言也经常被引用作“可编程的编程语言”（可以写代码的代码）。

免费的经典的入门书籍《[实用 Common Lisp 编程](#)》

许多人都抱怨上面这本书的翻译。《[ANSI Common Lisp](#)》也许对中文读者更友好一些。

另外还有一本热门的近期出版的 [Land of Lisp](#).

```
;;;;;;;;;;;;;
;;; 0. 语法
;;;;;;;;;;;;;

;;; 一般形式

;; Lisp有两个基本的语法单元：原子（atom），以及S-表达式。
;; 一般的，一组S-表达式被称为“组合式”。

10 ; 一个原子；它对自身进行求值

:THING ;同样是一个原子；它被求值为一个符号 :thing

t ;还是一个原子，代表逻辑真值。

(+ 1 2 3 4) ; 一个S-表达式。

'(4 :foo t) ;同样是一个S-表达式。

;;; 注释

;; 一个分号开头的注释表示仅用于此行（单行）；两个分号开头的则表示一个所谓标准注释；
;; 三个分号开头的意味着段落注释；
;; 而四个分号开头的注释用于文件头注释（译者注：即对该文件的说明）。

#| 块注释
   可以涵盖多行，而且...
   #|
       他们可以被嵌套！
   |#
|#
```

;;; 运行环境

```
;; 有很多不同的Common Lisp的实现；并且大部分的实现是一致（可移植）的。
;; 对于入门学习来说，CLISP是个不错的选择。
```

```
;; 可以通过QuickLisp.org的Quicklisp系统管理你的库。
```

```
;; 通常，使用文本编辑器和“REPL”来开发Common Lisp;
;; （译者注：“REPL”指读取-求值-打印循环）。
;; “REPL”允许对程序进行交互式的运行、调试，就好像在系统“现场”操作。
```

[illegible]

;;; 符号

'foo' ; => F00 注意到这个符号被自动转换成大写了。

;; `intern` 由一个给定的字符串而创建相应的符号

```
(intern "AAAA") ; => AAAA
```

```
(intern "aaa") ; => |aaa|
```

;;; 数字

99999999999999999999999999999999	; 整型数
#b111	; 二进制 => 7
#o111	; 八进制 => 73
#x111	; 十六进制 => 273
3.14159s0	; 单精度
3.14159d0	; 双精度
1/2	; 分数
#C(1 2)	; 复数

```
;; 使用函数时，应当写成这样的形式: (f x y z ...);
;; 其中，f是一个函数（名），x, y, z为参数;
;; 如果你想创建一个“字面”意义上（即不求值）的列表， 只需使用单引号 ' ,
;; 从而避免接下来的表达式被求值。即，只“引用”这个数据（而不求值）。
```

$$^t(+1\ 2) ; \Rightarrow (+1\ 2)$$

;; 你同样也可以手动地调用一个函数（译者注：即使用函数对象来调用函数）：

```
(funcall #' + 1 2 3) ; => 6
```

:: 一些算术运算符

```
(+ 1 1)           ; => 2
(- 8 1)          ; => 7
(* 10 2)         ; => 20
(expt 2 3)       ; => 8
(mod 5 2)        ; => 1
```

```

(/ 35 5)          ; => 7
(/ 1 3)           ; => 1/3
(+ #C(1 2) #C(6 -4)) ; => #C(7 -2)

;;; 布尔运算
t                ; 逻辑真（任何不是nil的值都被视为真值）
nil              ; 逻辑假，或者空列表
(not nil)        ; => t
(and 0 t)        ; => t
(or 0 nil)       ; => 0

;;; 字符
#\A              ; => #\A
#\λ              ; => #\GREEK_SMALL_LETTER_LAMDA（希腊字母Lambda的小写）
#\u03BB         ; => #\GREEK_SMALL_LETTER_LAMDA（Unicode形式的小写希腊字母Lambda）

;;; 字符串被视为一个定长字符数组
"Hello, world!"
"Benjamin \"Bugsy\" Siegel" ;反斜杠用作转义字符

;; 可以拼接字符串
(concatenate 'string "Hello " "world!") ; => "Hello world!"

;; 一个字符串也可被视作一个字符序列
(elt "Apple" 0) ; => #\A

;; `format`被用于格式化字符串
(format nil "~a can be ~a" "strings" "formatted")

;; 利用`format`打印到屏幕上是非常简单的
;; （译者注：注意到第二个参数是t，不同于刚刚的nil）；~% 代表换行符
(format t "Common Lisp is groovy. Dude.~%")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. 变量
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 你可以通过`defparameter`创建一个全局（动态）变量
;; 变量名可以是除了：()[]{}", '`;#| 这些字符之外的其他任何字符

;; 动态变量名应该由*号开头与结尾！
;; （译者注：这个只是一个习惯）

(defparameter *some-var* 5)
*some-var* ; => 5

;; 你也可以使用Unicode字符：
(defparameter *AΛB* nil)

;; 访问一个在之前从未被绑定的变量是一种不规范的行为（即使依然是可能发生的）；

```

```
;; 不要尝试那样做。
```

```
;; 局部绑定: 在(let ...)语句内, 'me'被绑定到"dance with you"上。
```

```
;; `let`总是返回在其作用域内最后一个表达式的值
```

```
(let ((me "dance with you"))
```

```
  me)
```

```
;; => "dance with you"
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 3. 结构体和集合
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 结构体
```

```
(defstruct dog name breed age)
```

```
(defparameter *rover*
```

```
  (make-dog :name "rover"
```

```
            :breed "collie"
```

```
            :age 5))
```

```
*rover* ; => #S(DOG :NAME "rover" :BREED "collie" :AGE 5)
```

```
(dog-p *rover*) ; => t ;; ewww)
```

```
(dog-name *rover*) ; => "rover"
```

```
;; Dog-p, make-dog, 以及 dog-name都是由defstruct创建的!
```

```
;;; 点对单元(Pairs)
```

```
;; `cons`可用于生成一个点对单元, 利用`car`以及`cdr`将分别得到第一个和第二个元素
```

```
(cons 'SUBJECT 'VERB) ; => '(SUBJECT . VERB)
```

```
(car (cons 'SUBJECT 'VERB)) ; => SUBJECT
```

```
(cdr (cons 'SUBJECT 'VERB)) ; => VERB
```

```
;;; 列表
```

```
;; 所有列表都是由点对单元构成的“链表”。它以'nil' (或者'()) 作为列表的最后一个元素。
```

```
(cons 1 (cons 2 (cons 3 nil))) ; => '(1 2 3)
```

```
;; `list`是一个生成列表的便利途径
```

```
(list 1 2 3) ; => '(1 2 3)
```

```
;; 并且, 一个引用也可被用做字面意义上的列表值
```

```
'(1 2 3) ; => '(1 2 3)
```

```
;; 同样的, 依然可以用`cons`来添加一项到列表的起始位置
```

```
(cons 4 '(1 2 3)) ; => '(4 1 2 3)
```

```
;; 而`append`也可用于连接两个列表
```

```
(append '(1 2) '(3 4)) ; => '(1 2 3 4)
```

```
;; 或者使用`concatenate`
```

```
(concatenate 'list '(1 2) '(3 4))
```

;; 列表是一种非常核心的数据类型，所以有非常多的处理列表的函数

;; 例如：

```
(mapcar #'1+ '(1 2 3))           ; => '(2 3 4)
(mapcar #'+ '(1 2 3) '(10 20 30)) ; => '(11 22 33)
(remove-if-not #'evenp '(1 2 3 4)) ; => '(2 4)
(every #'evenp '(1 2 3 4))        ; => nil
(some #'oddp '(1 2 3 4))          ; => T
(butlast '(subject verb object))  ; => (SUBJECT VERB)
```

;;; 向量

;; 向量的字面意义是一个定长数组

;; （译者注：此处所谓“字面意义”，即指#(.....)的形式，下文还会出现）

```
#(1 2 3) ; => #(1 2 3)
```

;; 使用`concatenate`来将两个向量首尾连接在一起

```
(concatenate 'vector #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)
```

;;; 数组

;; 向量和字符串只不过是数组的特例

;; 二维数组

```
(make-array (list 2 2))
```

;; (make-array '(2 2)) 也是可以的

```
; => #2A((0 0) (0 0))
```

```
(make-array (list 2 2 2))
```

```
; => #3A(((0 0) (0 0)) ((0 0) (0 0)))
```

;; 注意：数组的默认初始值是可以指定的

;; 下面是如何指定的示例：

```
(make-array '(2) :initial-element 'unset)
```

```
; => #(UNSET UNSET)
```

;; 若想获取数组[1][1][1]上的元素：

```
(aref (make-array (list 2 2 2)) 1 1 1)
```

```
; => 0
```

;;; 变长向量

;; 若将变长向量打印出来，那么它的字面意义上的值和定长向量的是一样的

```
(defparameter *adjvec* (make-array '(3) :initial-contents '(1 2 3)
                                     :adjustable t :fill-pointer t))
```

```
*adjvec* ; => #(1 2 3)
```

```
;; 添加新的元素:
```

```
(vector-push-extend 4 *adjvec*) ; => 3
```

```
*adjvec* ; => #(1 2 3 4)
```

```
;;; 不怎么严谨地说, 集合也可被视为列表
```

```
(set-difference '(1 2 3 4) '(4 5 6 7)) ; => (3 2 1)
```

```
(intersection '(1 2 3 4) '(4 5 6 7)) ; => 4
```

```
(union '(1 2 3 4) '(4 5 6 7)) ; => (3 2 1 4 5 6 7)
```

```
(adjoin 4 '(1 2 3 4)) ; => (1 2 3 4)
```

```
;; 然而, 你可能想使用一个更好的数据结构, 而并非一个链表
```

```
;;; 在Common Lisp中, “字典”和哈希表的实现是一样的。
```

```
;; 创建一个哈希表
```

```
(defparameter *m* (make-hash-table))
```

```
;; 给定键, 设置对应的值
```

```
(setf (gethash 'a *m*) 1)
```

```
;; (通过键)检索对应的值
```

```
(gethash 'a *m*) ; => 1, t
```

```
;; 注意此处有一细节: Common Lisp往往返回多个值。`gethash`返回的两个值是t, 代表找到了这个元素; 返回t
```

```
;; (译者注: 返回的第一个值表示给定的键所对应的值或者nil; )
```

```
;; (第二个是一个布尔值, 表示在哈希表中是否存在这个给定的键)
```

```
;; 例如, 如果可以找到给定的键所对应的值, 则返回一个t, 否则返回nil
```

```
;; 由给定的键检索一个不存在的值, 则返回nil
```

```
;; (译者注: 这个nil是第一个nil, 第二个nil其实是指该键在哈希表中也不存在)
```

```
(gethash 'd *m*) ;=> nil, nil
```

```
;; 给定一个键, 你可以指定其对应的默认值:
```

```
(gethash 'd *m* :not-found) ; => :NOT-FOUND
```

```
;; 在此, 让我们看一看怎样处理`gethash`的多个返回值。
```

```
(multiple-value-bind
```

```
  (a b)
```

```
  (gethash 'd *m*))
```

```
  (list a b))
```



```

; => (NIL NIL)

(multiple-value-bind
  (a b)
  (gethash 'a *m*))
(list a b))
; => (1 T)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. 函数
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 使用`lambda`来创建一个匿名函数。
;; 一个函数总是返回其形式体内最后一个表达式的值。
;; 将一个函数对象打印出来后的形式是多种多样的...

(lambda () "Hello World") ; => #

;; 使用`funcall`来调用lambda函数
(funcall (lambda () "Hello World")) ; => "Hello World"

;; 或者使用`apply`
(apply (lambda () "Hello World") nil) ; => "Hello World"

;; 显式地定义一个函数（译者注：即非匿名的）
(defun hello-world ()
  "Hello World")
(hello-world) ; => "Hello World"

;; 刚刚上面函数名"hello-world"后的()其实是函数的参数列表
(defun hello (name)
  (format nil "Hello, ~a " name))

(hello "Steve") ; => "Hello, Steve"

;; 函数可以有可选形参并且其默认值都为nil

(defun hello (name &optional from)
  (if from
    (format t "Hello, ~a, from ~a" name from)
    (format t "Hello, ~a" name)))

(hello "Jim" "Alpacas") ; => Hello, Jim, from Alpacas

;; 你也可以指定那些可选形参的默认值
(defun hello (name &optional (from "The world"))
  (format t "Hello, ~a, from ~a" name from))

(hello "Steve")
; => Hello, Steve, from The world

```

```

(hello "Steve" "the alpacas")
; => Hello, Steve, from the alpacas

;; 当然，你也可以设置所谓关键字形参；
;; 关键字形参往往比可选形参更具灵活性。

(defun generalized-greeter (name &key (from "the world") (honorific "Mx"))
  (format t "Hello, ~a ~a, from ~a" honorific name from))

(generalized-greeter "Jim") ; => Hello, Mx Jim, from the world

(generalized-greeter "Jim" :from "the alpacas you met last summer" :honorific "Mr")
; => Hello, Mr Jim, from the alpacas you met last summer

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. 等式
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Common Lisp具有一个十分复杂的用于判断等价的系统，下面只是其中一部分的例子

;; 若要比数值是否等价，使用`=`
(= 3 3.0) ; => t
(= 2 1) ; => nil

;; 若要比对象的类型，则使用`eql`
;; (译者注：抱歉，翻译水平实在有限，下面是我个人的补充说明)
;; (`eq` 返回真，如果对象的内存地址相等)
;; (`eql` 返回真，如果两个对象内存地址相等，或者对象的类型相同，并且值相等)
;; (例如同为整形数或浮点数，并且他们的值相等时，二者`eql`等价)
;; (想要弄清`eql`，其实有必要先了解`eq`)
;; ([可以参考](http://stackoverflow.com/questions/547436/whats-the-difference-between-eq-))
;; (可以去CLHS上分别查看两者的文档)
;; (另外，《实用Common Lisp编程》的4.8节也提到了两者的区别)
(eql 3 3) ; => t
(eql 3 3.0) ; => nil
(eql (list 3) (list 3)) ; => nil

;; 对于列表、字符串、以及位向量，使用`equal`
(equal (list 'a 'b) (list 'a 'b)) ; => t
(equal (list 'a 'b) (list 'b 'a)) ; => nil

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. 控制流
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; 条件判断语句

(if t ; “test”，即判断语句
    "this is true" ; “then”，即判断条件为真时求值的表达式
    "this is false") ; “else”，即判断条件为假时求值的表达式

```

```
; => "this is true"
```

```
;; 在“test”（判断）语句中，所有非nil或者非()的值都被视为真值
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(GROUCHO ZEPP0)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope)
; => 'YEP
```

```
;; `cond` 将一系列测试语句串联起来，并对相应的表达式求值
(cond ((> 2 2) (error "wrong!"))
      ((< 2 2) (error "wrong again!"))
      (t 'ok)) ; => 'OK
```

```
;; 对于给定值的数据类型，`typecase` 会做出相应地判断
(typecase 1
  (string :string)
  (integer :int))
```

```
; => :int
```

```
;;; 迭代
```

```
;; 当然，递归是肯定被支持的：
```

```
(defun walker (n)
  (if (zerop n)
      :walked
      (walker (1- n))))
```

```
(walker) ; => :walked
```

```
;; 而大部分场合下，我们使用`DOLIST`或者`LOOP`来进行迭代
```

```
(dolist (i '(1 2 3 4))
  (format t "~a" i))
```

```
; => 1234
```

```
(loop for i from 0 below 10
      collect i)
```

```
; => (0 1 2 3 4 5 6 7 8 9)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. 可变性
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 使用`setf`可以对一个已经存在的变量进行赋值：
```

;; 事实上，刚刚在哈希表的例子中我们已经示范过了。

```
(let ((variable 10))
  (setf variable 2))
; => 2
```

;; 所谓好的Lisp编码风格就是为了减少使用破坏性函数，防止发生副作用。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. 类与对象
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

;; 我们就不写什么有关动物的类了，下面给出的人力车的类

```
(defclass human-powered-conveyance ()
  ((velocity
    :accessor velocity
    :initarg :velocity)
   (average-efficiency
    :accessor average-efficiency
    :initarg :average-efficiency))
  (:documentation "A human powered conveyance"))
```

;; `defclass`，后面接类名，以及超类列表

;; 再接着是槽的列表（槽有点像Java里的成员变量），最后是一些可选的特性

;; 例如文档说明“:documentation”

;; 如果超类列表为空，则默认该类继承于“standard-object”类（standard-object又是T的子类）

;; 这种默认行为是可以改变的，但你最好有一定的基础并且知道自己到底在干什么；

;; 参阅《The Art of the Metaobject Protocol》来了解更多信息。

```
(defclass bicycle (human-powered-conveyance)
  ((wheel-size
    :accessor wheel-size
    :initarg :wheel-size
    :documentation "Diameter of the wheel.")
   (height
    :accessor height
    :initarg :height)))

(defclass recumbent (bicycle)
  ((chain-type
    :accessor chain-type
    :initarg :chain-type)))

(defclass unicycle (human-powered-conveyance) nil)

(defclass canoe (human-powered-conveyance)
  ((number-of-rowers
    :accessor number-of-rowers
```

```
:initarg :number-of-rowers)))
```

;; 在REPL中对human-powered-conveyance类调用`DESCRIBE`后结果如下:

```
(describe 'human-powered-conveyance)
```

```
; COMMON-LISP-USER::HUMAN-POWERED-CONVEYANCE
; [symbol]
;
; HUMAN-POWERED-CONVEYANCE names the standard-class #:
; Documentation:
;   A human powered conveyance
; Direct superclasses: STANDARD-OBJECT
; Direct subclasses: UNICYCLE, BICYCLE, CANOE
; Not yet finalized.
; Direct slots:
;   VELOCITY
;   Readers: VELOCITY
;   Writers: (SETF VELOCITY)
;   AVERAGE-EFFICIENCY
;   Readers: AVERAGE-EFFICIENCY
;   Writers: (SETF AVERAGE-EFFICIENCY)
```

;; 注意到这些有用的返回信息——Common Lisp一直是一个交互式的系统。

;; 若要定义一个方法;

;; 注意, 我们计算自行车轮子周长时使用了这样一个公式: $C = d * \pi$

```
(defmethod circumference ((object bicycle))
  (* pi (wheel-size object)))
```

;; pi在Common Lisp中已经是一个内置的常量。

;; 假设我们已经知道了效率值 (“efficiency value”) 和船桨数大概呈对数关系;

;; 那么效率值的定义应当在构造器/初始化过程中就被完成。

;; 下面是一个Common Lisp构造实例时初始化实例的例子:

```
(defmethod initialize-instance :after ((object canoe) &rest args)
  (setf (average-efficiency object) (log (1+ (number-of-rowers object)))))
```

;; 接着初构造一个实例并检查平均效率...

```
(average-efficiency (make-instance 'canoe :number-of-rowers 15))
; => 2.7725887
```

```
;;;;;;;;;;;;;
;; 8. 宏
;;;;;;;;;;;;;
```

;; 宏可以让你扩展语法

;; 例如, Common Lisp并没有自带WHILE循环——所以让我们自己来为他添加一个;
;; 如果按照汇编程序的直觉来看, 我们会这样写:

```
(defmacro while (condition &body body)
  "While `condition` is true, `body` is executed.

`condition` is tested prior to each execution of `body`"
  (let ((block-name (gensym)))
    `(tagbody
      (unless ,condition
        (go ,block-name))
      (progn
        ,@body)
      ,block-name)))
```

;; 让我们来看看它的高级版本:

```
(defmacro while (condition &body body)
  "While `condition` is true, `body` is executed.

`condition` is tested prior to each execution of `body`"
  `(loop while ,condition
    do
      (progn
        ,@body)))
```

;; 然而, 在一个比较现代化的编译环境下, 这样的WHILE是没有必要的;
;; LOOP形式的循环和这个WHILE同样的好, 并且更易于阅读。

;; 注意反引号'`', 逗号','以及'@'这三个符号;
;; 反引号'`'是一种所谓“quasiquote”的引用类型的运算符, 有了它, 之后的逗号“,”才有意义。
;; 逗号“,”意味着解除引用(unquote, 即开始求值);
;; “@”符号则表示将当前的参数插入到当前整个列表中。
;; (译者注: 要想真正用好、用对这三个符号, 需要下一番功夫)
;; (甚至光看《实用 Common Lisp 编程》中关于宏的介绍都是不够的)
;; (建议再去读一读Paul Graham的两本著作《ANSI Common Lisp》和《On Lisp》)

;; 函数`gensym`创建一个唯一的符号——这个符号确保不会出现在其他任何地方。
;; 这样做是因为, 宏是在编译期展开的
;; 而在宏中声明的变量名极有可能和常规代码中使用的变量名发生冲突。

;; 可以去《实用 Common Lisp 编程》中阅读更多有关宏的内容。

拓展阅读

继续阅读《实用 Common Lisp 编程》一书

致谢

非常感谢Scheme社区的人们，我基于他们的成果得以迅速的写出这篇有关Common Lisp的快速入门同时也感谢

- [Paul Khuong](#)，他提出了很多有用的点评。

译者寄语

“祝福那些将思想镶嵌在重重括号之内的人们。”

language: c# contributors:

```
- ["Irfan Charania", "https://github.com/irfancharania"]
- ["Max Yankov", "https://github.com/golengka"]
- ["Melvyn Laily", "http://x2a.yt"]
- ["Shaun McCarthy", "http://www.shaumccarthy.com"]
```

translators:

```
- ["Jakukyo Friel", "http://weakish.github.io"]
```

filename: LearnCSharp-cn.cs

lang: zh-cn

C#是一个优雅的、类型安全的面向对象语言。使用**C#**，开发者可以在.NET框架下构建安全、健壮的应用程序。

[更多关于C#的介绍](#)

```
// 单行注释以 // 开始
/*
多行注释是这样的
*/
///

/// XML文档注释
///


// 声明应用用到的命名空间
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Dynamic;
using System.Linq;
using System.Linq.Expressions;
using System.Net;
using System.Threading.Tasks;
using System.IO;


// 定义作用域，将代码组织成包
namespace Learning
{
    // 每个 .cs 文件至少需要包含一个和文件名相同的类
    // 你可以不这么干，但是这样不好。
```



```

public class LearnCSharp
{
    // 基本语法 - 如果你以前用过 Java 或 C++ 的话, 可以直接跳到后文「有趣的特性」
    public static void Syntax()
    {
        // 使用 Console.WriteLine 打印信息
        Console.WriteLine("Hello World");
        Console.WriteLine(
            "Integer: " + 10 +
            " Double: " + 3.14 +
            " Boolean: " + true);

        // 使用 Console.Write 打印, 不带换行符号
        Console.Write("Hello ");
        Console.Write("World");

        //////////////////////////////////////
        // 类型和变量
        //
        // 使用    定义变量
        //////////////////////////////////////

        // Sbyte - 有符号 8-bit 整数
        // (-128 <= 754="" sbyte="" <="127)" fooSbyte="100;" Byte="" -="" 无符号
        // 字符串不可修改:  fooString[1] = 'X' 是行不通的;

        // 根据当前的locale设定比较字符串, 大小写不敏感
        string.Compare(fooString, "x", StringComparison.CurrentCultureIgnoreCase);

        // 基于sprintf的字符串格式化
        string fooFs = string.Format("Check Check, {0} {1}, {0} {1:0.0}", 1, 2);

        // 日期和格式
        DateTime fooDate = DateTime.Now;
        Console.WriteLine(fooDate.ToString("hh:mm, dd MMM yyyy"));

        // 使用 @ 符号可以创建跨行的字符串。使用 "" 来表示 "
        string bazString = @"Here's some stuff
on a new line! "Wow!", the masses cried";

        // 使用const或read-only定义常量
        // 常量在编译期演算
        const int HOURS_I_WORK_PER_WEEK = 9001;

        //////////////////////////////////////
        // 数据结构
        //////////////////////////////////////

        // 数组 - 从0开始计数
        // 声明数组时需要确定数组长度
        // 声明数组的格式如下:

```

```

// [] = new [];
int[] intArray = new int[10];

// 声明并初始化数组的其他方式:
int[] y = { 9000, 1000, 1337 };

// 访问数组的元素
Console.WriteLine("intArray @ 0: " + intArray[0]);
// 数组可以修改
intArray[1] = 1;

// 列表
// 列表比数组更常用, 因为列表更灵活。
// 声明列表的格式如下:
// List = new List();
List<int> intList = new List<int>();
List<string> stringList = new List<string>();
List<int> z = new List<int> { 9000, 1000, 1337 }; // i
// <>用于泛型 - 参考下文

// 列表无默认值
// 访问列表元素时必须首先添加元素
intList.Add(1);
Console.WriteLine("intList @ 0: " + intList[0]);

// 其他数据结构:
// 堆栈/队列
// 字典 (哈希表的实现)
// 哈希集合
// 只读集合
// 元组 (.Net 4+)

////////////////////////////////////
// 操作符
////////////////////////////////////
Console.WriteLine("\n->Operators");

int i1 = 1, i2 = 2; // 多重声明的简写形式

// 算术直截了当
Console.WriteLine(i1 + i2 - i1 * 3 / 7); // => 3

// 取余
Console.WriteLine("11%3 = " + (11 % 3)); // => 2

// 比较操作符
Console.WriteLine("3 == 2? " + (3 == 2)); // => false
Console.WriteLine("3 != 2? " + (3 != 2)); // => true
Console.WriteLine("3 > 2? " + (3 > 2)); // => true
Console.WriteLine("3 < 2? " + (3 < 2)); // => false
Console.WriteLine("2 <= 2? " + (2 <= 2)); // => true

```

```

Console.WriteLine("2 >= 2? " + (2 >= 2)); // => true

// 位操作符
/*
~      取反
<<    左移（有符号）
>>    右移（有符号）
&      与
^      异或
|      或
*/

// 自增、自减
int i = 0;
Console.WriteLine("\n->Inc/Dec-rementation");
Console.WriteLine(i++); //i = 1. 事后自增
Console.WriteLine(++i); //i = 2. 事先自增
Console.WriteLine(i--); //i = 1. 事后自减
Console.WriteLine(--i); //i = 0. 事先自减

////////////////////
// 控制结构
////////////////////
Console.WriteLine("\n->Control Structures");

// 类似C的if语句
int j = 10;
if (j == 10)
{
    Console.WriteLine("I get printed");
}
else if (j > 10)
{
    Console.WriteLine("I don't");
}
else
{
    Console.WriteLine("I also don't");
}

// 三元表达式
// 简单的 if/else 语句可以写成:
// <条件> ? <真> : <假>
int toCompare = 17;
string isTrue = toCompare == 17 ? "True" : "False";

// While 循环
int fooWhile = 0;
while (fooWhile < 100)
{
    //迭代 100 次, fooWhile 0->99
}

```

```

        fooWhile++;
    }

    // Do While 循环
    int fooDoWhile = 0;
    do
    {
        //迭代 100 次, fooDoWhile 0->99
        fooDoWhile++;
    } while (fooDoWhile < 100);

    //for 循环结构 => for(<初始条件>; <条件>; <步>)
    for (int fooFor = 0; fooFor < 10; fooFor++)
    {
        //迭代10次, fooFor 0->9
    }

    // foreach循环
    // foreach 循环结构 => foreach(<迭代器类型> <迭代器> in <可枚举结构>)
    // foreach 循环适用于任何实现了 IEnumerable 或 IEnumerable 的对象。
    // .Net 框架下的集合类型(数组, 列表, 字典...)
    // 都实现了这些接口
    // (下面的代码中, ToCharArray()可以删除, 因为字符串同样实现了IEnumerable)
    foreach (char character in "Hello World".ToCharArray())
    {
        //迭代字符串中的所有字符
    }

    // Switch 语句
    // switch 适用于 byte、short、char和int 数据类型。
    // 同样适用于可枚举的类型
    // 包括字符串类, 以及一些封装了原始值的类:
    // Character、Byte、Short和Integer。
    int month = 3;
    string monthString;
    switch (month)
    {
        case 1:
            monthString = "January";
            break;
        case 2:
            monthString = "February";
            break;
        case 3:
            monthString = "March";
            break;
        // 你可以一次匹配多个case语句
        // 但是你在添加case语句后需要使用break
        // (否则你需要显式地使用goto case x语句)
        case 6:
        case 7:
    }

```

```

        case 8:
            monthString = "Summer time!!";
            break;
        default:
            monthString = "Some other month";
            break;
    }

    //////////////////////////////////////
    // 转换、指定数据类型
    //////////////////////////////////////

    // 转换类型

    // 转换字符串为整数
    // 转换失败会抛出异常
    int.Parse("123");//返回整数类型的"123"

    // TryParse会尝试转换类型，失败时会返回缺省类型
    // 例如 0
    int tryInt;
    if (int.TryParse("123", out tryInt)) // Funciton is boolean
        Console.WriteLine(tryInt);      // 123

    // 转换整数为字符串
    // Convert类提供了一系列便利转换的方法
    Convert.ToString(123);
    // or
    tryInt.ToString();
}

////////////////////////////////////
// 类
////////////////////////////////////
public static void Classes()
{
    // 参看文件尾部的对象声明

    // 使用new初始化对象
    Bicycle trek = new Bicycle();

    // 调用对象的方法
    trek.SpeedUp(3); // 你应该一直使用setter和getter方法
    trek.Cadence = 100;

    // 查看对象的信息.
    Console.WriteLine("trek info: " + trek.Info());

    // 实例化一个新的Penny Farthing
    PennyFarthing funbike = new PennyFarthing(1, 10);
    Console.WriteLine("funbike info: " + funbike.Info());
}

```

```

        Console.Read();
    } // 结束main方法

// 终端程序 终端程序必须有一个main方法作为入口
public static void Main(string[] args)
{
    OtherInterestingFeatures();
}

//
// 有趣的特性
//

// 默认方法签名

public // 可见性
static // 允许直接调用类，无需先创建实例
int, //返回值
MethodSignatures(
    int maxCount, // 第一个变量，类型为整型
    int count = 0, // 如果没有传入值，则缺省值为0
    int another = 3,
    params string[] otherParams // 捕获其他参数
)
{
    return -1;
}

// 方法可以重名，只要签名不一样
public static void MethodSignature(string maxCount)
{
}

//泛型
// TKey和TValue类由用用户调用函数时指定。
// 以下函数模拟了Python的SetDefault
public static TValue SetDefault(
    IDictionary dictionary,
    TKey key,
    TValue defaultItem)
{
    TValue result;
    if (!dictionary.TryGetValue(key, out result))
        return dictionary[key] = defaultItem;
    return result;
}

// 你可以限定传入值的范围
public static void IterateAndPrint(T toPrint) where T: IEnumerable
{

```

```

// 我们可以进行迭代，因为T是可枚举的
foreach (var item in toPrint)
    // item为整数
    Console.WriteLine(item.ToString());
}

public static void OtherInterestingFeatures()
{
    // 可选参数
    MethodSignatures(3, 1, 3, "Some", "Extra", "Strings");
    MethodSignatures(3, another: 3); // 显式指定参数，忽略可选参数

    // 扩展方法
    int i = 3;
    i.Print(); // 参见下面的定义

    // 可为null的类型 对数据库交互、返回值很有用
    // 任何值类型 (i.e. 不为类) 添加后缀 ? 后会变为可为null的值
    // <类型>? <变量名> = <值>
    int? nullable = null; // Nullable 的简写形式
    Console.WriteLine("Nullable variable: " + nullable);
    bool hasValue = nullable.HasValue; // 不为null时返回真
    // ?? 是用于指定默认值的语法糖
    // 以防变量为null的情况
    int notNullable = nullable ?? 0; // 0

    // 变量类型推断 - 你可以让编译器推断变量类型：
    var magic = "编译器确定magic是一个字符串，所以仍然是类型安全的";
    // magic = 9; // 不工作，因为magic是字符串，而不是整数。

    // 泛型
    //
    var phonebook = new Dictionary() {
        {"Sarah", "212 555 5555"} // 在电话簿中加入新条目
    };

    // 调用上面定义为泛型的SETDEFAULT
    Console.WriteLine(SetDefault(phonebook, "Shaun", "No Phone")); // 没有电话
    // 你不用指定TKey、TValue，因为它们会被隐式地推导出来
    Console.WriteLine(SetDefault(phonebook, "Sarah", "No Phone")); // 212 555 55

    // Lambda表达式 - 允许你用一行代码搞定函数
    Func square = (x) => x * x; // 最后一项为返回值
    Console.WriteLine(square(3)); // 9

    // 可抛弃的资源管理 - 让你很容易地处理未管理的资源
    // 大多数访问未管理资源 (文件操作符、设备上下文, etc.)的对象
    // 都实现了IDisposable接口。
    // using语句会为你清理IDisposable对象。
    using (StreamWriter writer = new StreamWriter("Log.txt"))
    {

```

```

        writer.WriteLine("这里没有什么可疑的东西");
        // 在作用域的结尾，资源会被回收
        // （即使有异常抛出，也一样会回收）
    }

    // 并行框架
    // http://blogs.msdn.com/b/csharpfaq/archive/2010/06/01/parallel-programming
    var websites = new string[] {
        "http://www.google.com", "http://www.reddit.com",
        "http://www.shawnmccarthy.com"
    };
    var responses = new Dictionary();

    // 为每个请求新开一个线程
    // 在运行下一步前合并结果
    Parallel.ForEach(websites,
        new ParallelOptions() {MaxDegreeOfParallelism = 3}, // max of 3 threads
        website =>
        {
            // Do something that takes a long time on the file
            using (var r = WebRequest.Create(new Uri(website)).GetResponse())
            {
                responses[website] = r.ContentType;
            }
        });

    // 直到所有的请求完成后才会运行下面的代码
    foreach (var key in responses.Keys)
        Console.WriteLine("{0}:{1}", key, responses[key]);

    // 动态对象（配合其他语言使用很方便）
    dynamic student = new ExpandoObject();
    student.FirstName = "First Name"; // 不需要先定义类！

    // 你甚至可以添加方法（接受一个字符串，输出一个字符串）
    student.Introduce = new Func(
        (introduceTo) => string.Format("Hey {0}, this is {1}", student.FirstName,
        Console.WriteLine(student.Introduce("Beth")));

    // IQUERYABLE - 几乎所有的集合都实现了它，
    // 带给你 Map / Filter / Reduce 风格的方法
    var bikes = new List();
    bikes.Sort(); // Sorts the array
    bikes.Sort((b1, b2) => b1.Wheels.CompareTo(b2.Wheels)); // 根据车轮数排序
    var result = bikes
        .Where(b => b.Wheels > 3) // 筛选 - 可以连锁使用 （返回IQueryable）
        .Where(b => b.IsBroken && b.HasTassles)
        .Select(b => b.ToString()); // Map - 这里我们使用了select，所以结果是IQueryable

    var sum = bikes.Sum(b => b.Wheels); // Reduce - 计算集合中的轮子总数

```



```

// 创建一个包含基于自行车的一些参数生成的隐式对象的列表
var bikeSummaries = bikes.Select(b=>new { Name = b.Name, IsAwesome = !b.IsBroken })
// 很难演示，但是编译器在代码编译完成前就能推导出以上对象的类型
foreach (var bikeSummary in bikeSummaries.Where(b => b.IsAwesome))
    Console.WriteLine(bikeSummary.Name);

// ASPARALLEL
// 邪恶的特性 — 组合了Linq和并行操作
var threeWheelers = bikes.AsParallel().Where(b => b.Wheels == 3).Select(b => b.Name)
// 以上代码会并发地运行。会自动新开线程，分别计算结果。
// 适用于多核、大数据量的场景。

// LINQ - 将IQueryable映射到存储，延缓执行
// 例如 LinqToSql 映射数据库，LinqToXml 映射XML文档
var db = new BikeRepository();

// 执行被延迟了，这对于查询数据库来说很好
var filter = db.Bikes.Where(b => b.HasTassles); // 不运行查询
if (42 > 6) // 你可以不断地增加筛选，包括有条件的筛选，例如用于“高级搜索”功能
    filter = filter.Where(b => b.IsBroken); // 不运行查询

var query = filter
    .OrderBy(b => b.Wheels)
    .ThenBy(b => b.Name)
    .Select(b => b.Name); // 仍然不运行查询

// 现在运行查询，运行查询的时候会打开一个读取器，所以你迭代的是一个副本
foreach (string bike in query)
    Console.WriteLine(result);

}

} // 结束LearnCSharp类

// 你可以在同一个 .cs 文件中包含其他类

public static class Extensions
{
    // 扩展函数
    public static void Print(this object obj)
    {
        Console.WriteLine(obj.ToString());
    }
}

// 声明类的语法：
// class <类名>{
//     //数据字段，构造器，内部函数。
//     // 在Java中函数被称为方法。
// }

```

```

public class Bicycle
{
    // 自行车的字段、变量
    public int Cadence // Public: 任何地方都可以访问
    {
        get // get - 定义获取属性的方法
        {
            return _cadence;
        }
        set // set - 定义设置属性的方法
        {
            _cadence = value; // value是被传递给setter的值
        }
    }
    private int _cadence;

    protected virtual int Gear // 类和子类可以访问
    {
        get; // 创建一个自动属性, 无需成员字段
        set;
    }

    internal int Wheels // Internal: 在同一程序集内可以访问
    {
        get;
        private set; // 可以给get/set方法添加修饰符
    }

    int _speed; // 默认为private: 只可以在这个类内访问, 你也可以使用`private`关键词
    public string Name { get; set; }

    // enum类型包含一组常量
    // 它将名称映射到值 (除非特别说明, 是一个整型)
    // enum元素的类型可以是byte、sbyte、short、ushort、int、uint、long、ulong。
    // enum不能包含相同的值。
    public enum BikeBrand
    {
        AIST,
        BMC,
        Electra = 42, // 你可以显式地赋值
        Gitane // 43
    }
    // 我们在Bicycle类中定义的这个类型, 所以它是一个内嵌类型。
    // 这个类以外的代码应当使用`Bicycle.Brand`来引用。

    public BikeBrand Brand; // 声明一个enum类型之后, 我们可以声明这个类型的字段

    // 静态方法的类型为自身, 不属于特定的对象。
    // 你无需引用对象就可以访问他们。
    // Console.WriteLine("Bicycles created: " + Bicycle.bicyclesCreated);

```

```

static public int BicyclesCreated = 0;

// 只读值在运行时确定
// 它们只能在声明或构造器内被赋值
readonly bool _hasCardsInSpokes = false; // read-only private

// 构造器是创建类的一种方式
// 下面是一个默认的构造器
public Bicycle()
{
    this.Gear = 1; // 你可以使用关键词this访问对象的成员
    Cadence = 50; // 不过你并不总是需要它
    _speed = 5;
    Name = "Bontrager";
    Brand = BikeBrand.AIST;
    BicyclesCreated++;
}

// 另一个构造器的例子（包含参数）
public Bicycle(int startCadence, int startSpeed, int startGear,
               string name, bool hasCardsInSpokes, BikeBrand brand)
    : base() // 首先调用base
{
    Gear = startGear;
    Cadence = startCadence;
    _speed = startSpeed;
    Name = name;
    _hasCardsInSpokes = hasCardsInSpokes;
    Brand = brand;
}

// 构造器可以连锁使用
public Bicycle(int startCadence, int startSpeed, BikeBrand brand) :
    this(startCadence, startSpeed, 0, "big wheels", true, brand)
{
}

// 函数语法
// <返回值> <函数名称>(<参数>)

// 类可以为字段实现 getters 和 setters 方法 for their fields
// 或者可以实现属性（C#推荐使用这个）
// 方法的参数可以有默认值
// 在有默认值的情况下，调用方法的时候可以省略相应的参数
public void SpeedUp(int increment = 1)
{
    _speed += increment;
}

public void SlowDown(int decrement = 1)
{

```

```

        _speed -= decrement;
    }

    // 属性可以访问和设置值
    // 当只需要访问数据的时候，考虑使用属性。
    // 属性可以定义get和set，或者是同时定义两者
    private bool _hasTassles; // private variable
    public bool HasTassles // public accessor
    {
        get { return _hasTassles; }
        set { _hasTassles = value; }
    }

    // 你可以在一行之内定义自动属性
    // 这个语法会自动创建后备字段
    // 你可以给getter或setter设置访问修饰符
    // 以便限制它们的访问
    public bool IsBroken { get; private set; }

    // 属性的实现可以是自动的
    public int FrameSize
    {
        get;
        // 你可以给get或set指定访问修饰符
        // 以下代码意味着只有Bicycle类可以调用Framesize的set
        private set;
    }

    //显示对象属性的方法
    public virtual string Info()
    {
        return "Gear: " + Gear +
            " Cadence: " + Cadence +
            " Speed: " + _speed +
            " Name: " + Name +
            " Cards in Spokes: " + (_hasCardsInSpokes ? "yes" : "no") +
            "\n-----\n"
        ;
    }

    // 方法可以是静态的。通常用于辅助方法。
    public static bool DidWeCreateEnoughBycles()
    {
        // 在静态方法中，你只能引用类的静态成员
        return BicyclesCreated > 9000;
    } // 如果你的类只需要静态成员，考虑将整个类作为静态类。

} // Bicycle类结束

// PennyFarthing是Bicycle的一个子类

```

```

class PennyFarthing : Bicycle
{
    // (Penny Farthings是一种前轮很大的自行车。没有齿轮。)

    // 调用父构造器
    public PennyFarthing(int startCadence, int startSpeed) :
        base(startCadence, startSpeed, 0, "PennyFarthing", true, BikeBrand.Electra)
    {
    }

    protected override int Gear
    {
        get
        {
            return 0;
        }
        set
        {
            throw new ArgumentException("你不可能在PennyFarthing上切换齿轮");
        }
    }

    public override string Info()
    {
        string result = "PennyFarthing bicycle ";
        result += base.ToString(); // 调用父方法
        return result;
    }
}

// 接口只包含成员的签名，而没有实现。
interface IJumpable
{
    void Jump(int meters); // 所有接口成员是隐式地公开的
}

interface IBreakable
{
    bool Broken { get; } // 接口可以包含属性、方法和事件
}

// 类只能继承一个类，但是可以实现任意数量的接口
{
    int damage = 0;

    public void Jump(int meters)
    {
        damage += meters;
    }

    public bool Broken

```

```
        {
            get
            {
                return damage > 100;
            }
        }
    }

    ///

    /// 连接数据库，一个 LinqToSql 的示例。
    /// EntityFramework Code First 很棒 (类似 Ruby 的 ActiveRecord，不过是双向的)
    /// http://msdn.microsoft.com/en-us/data/jj193542.aspx
    ///

    public class BikeRespository : DbSet
    {
        public BikeRespository()
            : base()
        {
        }

        public DbSet Bikes { get; set; }
    }
} // 结束 Namespace
```

没有涉及到的主题

- Flags
- Attributes
- 静态属性
- Exceptions, Abstraction
- ASP.NET (Web Forms/MVC/WebMatrix)
- Winforms
- Windows Presentation Foundation (WPF)

扩展阅读

- [DotNetPerls](#)
- [C# in Depth](#)
- [Programming C#](#)
- [LINQ](#)
- [MSDN Library](#)
- [ASP.NET MVC Tutorials](#)
- [ASP.NET Web Matrix Tutorials](#)
- [ASP.NET Web Forms Tutorials](#)

- [Windows Forms Programming in C#](#)
- [C# Coding Conventions](#)

language: css contributors:

```
- ["Mohammad Valipour", "https://github.com/mvalipour"]
- ["Marco Scannadinari", "https://github.com/marcoms"]
```

translators:

```
- ["Jakukyo Friel", "https://weakish.github.io"]
```

lang: zh-cn

filename: learncss-cn.css

早期的web没有样式，只是单纯的文本。通过CSS，可以实现网页样式和内容的分离。

简单来说，CSS可以指定HTML页面上的元素所使用的样式。

和其他语言一样，CSS有很多版本。最新的版本是CSS 3. CSS 2.0兼容性最好。

你可以使用[dabblet](#)来在线测试CSS的效果。

```
/* 注释 */

/* #####
## 选择器
#####*/

/* 一般而言，CSS的声明语句非常简单。 */
选择器 { 属性: 值; /* 更多属性...*/ }

/* 选择器用于指定页面上的元素。

针对页面上的所有元素。 */
* { color:red; }

/*
假定页面上有这样一个元素

<div class='some-class class2' id='someId' attr='value' />
*/

/* 你可以通过类名来指定它 */
.some-class { }

/* 给出所有类名 */
.some-class.class2 { }
```



```
/* 标签名 */
div { }

/* id */
#someId { }

/* 由于元素包含attr属性，因此也可以通过这个来指定 */
[attr] { font-size:smaller; }

/* 以及有特定值的属性 */
[attr='value'] { font-size:smaller; }

/* 通过属性的值的开头指定 */
[attr^='val'] { font-size:smaller; }

/* 通过属性的值的结尾来指定 */
[attr$='ue'] { font-size:smaller; }

/* 通过属性的值的部分来指定 */
[attr~='lu'] { font-size:smaller; }

/* 你可以把这些全部结合起来，注意不同部分间不应该有空格，否则会改变语义 */
div.some-class[attr$='ue'] { }

/* 你也可以通过父元素来指定。*/

/* 某个元素是另一个元素的直接子元素 */
div.some-parent > .class-name {}

/* 或者通过该元素的祖先元素 */
div.some-parent .class-name {}

/* 注意，去掉空格后语义就不同了。
你能说出哪里不同么？ */
div.some-parent.class-name {}

/* 你可以选择某元素前的相邻元素 */
.i-am-before + .this-element { }

/* 某元素之前的同级元素（相邻或不相邻） */
.i-am-any-before ~ .this-element {}

/* 伪类允许你基于页面的行为指定元素（而不是基于页面结构） */

/* 例如，当鼠标悬停在某个元素上时 */
:hover {}

/* 已访问过的链接*/
:visited {}
```

```

/* 未访问过的链接*/
:link {}

/* 当前焦点的input元素 */
:focus {}

/* #####
## 属性
#####*/

选择器 {

    /* 单位 */
    width: 50%; /* 百分比 */
    font-size: 2em; /* 当前字体大小的两倍 */
    width: 200px; /* 像素 */
    font-size: 20pt; /* 点 */
    width: 5cm; /* 厘米 */
    width: 50mm; /* 毫米 */
    width: 5in; /* 英尺 */

    /* 颜色 */
    background-color: #F6E; /* 短16位 */
    background-color: #F262E2; /* 长16位 */
    background-color: tomato; /* 颜色名称 */
    background-color: rgb(255, 255, 255); /* rgb */
    background-color: rgb(10%, 20%, 50%); /* rgb 百分比 */
    background-color: rgba(255, 0, 0, 0.3); /* rgb 加透明度 */

    /* 图片 */
    background-image: url(/path-to-image/image.jpg);

    /* 字体 */
    font-family: Arial;
    font-family: "Courier New"; /* 使用双引号包裹含空格的字体名称 */
    font-family: "Courier New", Trebuchet, Arial; /* 如果第一个
                                                字体没找到，浏览器会使用第二个字体，一次类推 */
}

```

使用

CSS文件使用 `.css` 后缀。

```

<!-- 你需要在文件的 <head> 引用CSS文件 -->
<link rel='stylesheet' type='text/css' href='filepath/filename.css' />

<!-- 你也可以在标记中内嵌CSS。不过强烈建议不要这么干。 -->
<style>
    选择器 { 属性:值; }
</style>

<!-- 也可以直接使用元素的style属性。
这是你最不该做的事情。 -->
<div style='property:value;'>
</div>

```

优先级

同一个元素可能被多个不同的选择器指定，因此可能会有冲突。

假定CSS是这样的：

```

/*A*/
p.class1[attr='value']

/*B*/
p.class1 {}

/*C*/
p.class2 {}

/*D*/
p {}

/*E*/
p { property: value !important; }

```

然后标记语言为：

```

<p style='/*F*/ property:value;' class='class1 class2' attr='value'>
</p>

```

那么将会按照下面的顺序应用风格：

- **E** 优先级最高，因为它使用了 **!important**，除非很有必要，尽量避免使用这个。
- **F** 其次，因为它是嵌入的风格。
- **A** 其次，因为它比其他指令更具体。
- **C** 其次，虽然它的具体程度和 **B** 一样，但是它在 **B** 之后。
- 接下来是 **B**。
- 最后是 **D**。

兼容性

CSS2 的绝大部分特性兼容各种浏览器和设备。现在 CSS3 的兼容性也越来越好了。但是兼容性问题仍然是需要留意的一个问题。

[QuirksMode CSS](#)是关于这方面最好的资源。

扩展阅读

- [理解CSS的风格优先级: 特定性, 继承和层叠](#)
- [QuirksMode CSS](#)
- [Z-Index - The stacking context](#)

language: dart lang: zh-cn filename: learndart-cn.dart contributors:

```
- ["Joao Pedrosa", "https://github.com/jpedrosa/"]
```

translators:

```
- ["Guokai Han", "https://github.com/hanguokai/"]
```

Dart 是编程语言王国的新人。它借鉴了许多其他主流语言，并且不会偏离它的兄弟语言 JavaScript 太多。就像 JavaScript 一样，Dart 的目标是提供良好的浏览器集成。

Dart 最有争议的特性必然是它的可选类型。

```
import "dart:collection";
import "dart:math" as DM;

// 欢迎进入15分钟的 Dart 学习。 http://www.dartlang.org/
// 这是一个可实际执行的向导。你可以用 Dart 运行它
// 或者在线执行！可以把代码复制/粘贴到这个网站。 http://try.dartlang.org/

// 函数声明和方法声明看起来一样。
// 函数声明可以嵌套。声明使用这种 name() {} 的形式，
// 或者 name() => 单行表达式; 的形式。
// 右箭头的声明形式会隐式地返回表达式的结果。
example1() {
  example1nested1() {
    example1nested2() => print("Example1 nested 1 nested 2");
    example1nested2();
  }
  example1nested1();
}

// 匿名函数没有函数名。
example2() {
  example2nested1(fn) {
    fn();
  }
  example2nested1(() => print("Example2 nested 1"));
}

// 当声明函数类型的参数的时候，声明中可以包含
// 函数参数需要的参数，指定所需的参数名即可。
example3() {
  example3nested1(fn(informSomething)) {
    fn("Example3 nested 1");
  }
}
```

```

    }
    example3planB(fn) { // 或者不声明函数参数的参数
        fn("Example3 plan B");
    }
    example3nested1((s) => print(s));
    example3planB((s) => print(s));
}

```

// 函数有可以访问到外层变量的闭包。

```

var example4Something = "Example4 nested 1";
example4() {
    example4nested1(fn(informSomething)) {
        fn(example4Something);
    }
    example4nested1((s) => print(s));
}

```

// 下面这个包含 `sayIt` 方法的类声明，同样有一个可以访问外层变量的闭包，
// 就像前面的函数一样。

```

var example5method = "Example5 sayIt";
class Example5Class {
    sayIt() {
        print(example5method);
    }
}
example5() {
    // 创建一个 Example5Class 类的匿名实例，
    // 并调用它的 sayIt 方法。
    new Example5Class().sayIt();
}

```

// 类的声明使用这种形式 `class name { [classBody] }`。

// `classBody` 中可以包含实例方法和变量，

// 还可以包含类方法和变量。

```

class Example6Class {
    var example6InstanceVariable = "Example6 instance variable";
    sayIt() {
        print(example6InstanceVariable);
    }
}
example6() {
    new Example6Class().sayIt();
}

```

// 类方法和变量使用 `static` 关键词声明。

```

class Example7Class {
    static var example7ClassVariable = "Example7 class variable";
    static sayItFromClass() {
        print(example7ClassVariable);
    }
    sayItFromInstance() {

```

```

    print(example7ClassVariable);
  }
}
example7() {
  Example7Class.sayItFromClass();
  new Example7Class().sayItFromInstance();
}

// 字面量非常方便，但是对于在函数/方法的外层的字面量有一个限制，
// 类的外层或外面的字面量必需是常量。
// 字符串和数字默认是常量。
// 但是 array 和 map 不是。他们需要用 "const" 声明为常量。
var example8A = const ["Example8 const array"],
    example8M = const {"someKey": "Example8 const map"};
example8() {
  print(example8A[0]);
  print(example8M["someKey"]);
}

// Dart 中的循环使用标准的 for () {} 或 while () {} 的形式，
// 以及更加现代的 for (.. in ..) {} 的形式，或者
// 以 forEach 开头并具有许多特性支持的函数回调的形式。
var example9A = const ["a", "b"];
example9() {
  for (var i = 0; i < example9A.length; i++) {
    print("Example9 for loop '${example9A[i]}'");
  }
  var i = 0;
  while (i < example9A.length) {
    print("Example9 while loop '${example9A[i]}'");
    i++;
  }
  for (var e in example9A) {
    print("Example9 for-in loop '${e}'");
  }
  example9A.forEach((e) => print("Example9 forEach loop '${e}'"));
}

// 遍历字符串中的每个字符或者提取其子串。
var example10S = "ab";
example10() {
  for (var i = 0; i < example10S.length; i++) {
    print("Example10 String character loop '${example10S[i]}'");
  }
  for (var i = 0; i < example10S.length; i++) {
    print("Example10 substring loop '${example10S.substring(i, i + 1)}'");
  }
}

// 支持两种数字格式 int 和 double 。
example11() {

```

```
var i = 1 + 320, d = 3.2 + 0.01;
print("Example11 int ${i}");
print("Example11 double ${d}");
}
```

// DateTime 提供了日期/时间的算法。

```
example12() {
    var now = new DateTime.now();
    print("Example12 now '${now}'");
    now = now.add(new Duration(days: 1));
    print("Example12 tomorrow '${now}'");
}
```

// 支持正则表达式。

```
example13() {
    var s1 = "some string", s2 = "some", re = new RegExp("^s.+?g\\$");
    match(s) {
        if (re.hasMatch(s)) {
            print("Example13 regexp matches '${s}'");
        } else {
            print("Example13 regexp doesn't match '${s}'");
        }
    }
    match(s1);
    match(s2);
}
```

// 布尔表达式必需被解析为 true 或 false。

// 因为不支持隐式转换。

```
example14() {
    var v = true;
    if (v) {
        print("Example14 value is true");
    }
    v = null;
    try {
        if (v) {
            // 不会执行
        } else {
            // 不会执行
        }
    } catch (e) {
        print("Example14 null value causes an exception: '${e}'");
    }
}
```

// try/catch/finally 和 throw 语句用于异常处理。

// throw 语句可以使用任何对象作为参数。

```
example15() {
    try {
        try {
```



```

        throw "Some unexpected error.";
    } catch (e) {
        print("Example15 an exception: '${e}'");
        throw e; // Re-throw
    }
} catch (e) {
    print("Example15 catch exception being re-thrown: '${e}'");
} finally {
    print("Example15 Still run finally");
}
}

```

// 要想有效地动态创建长字符串，
 // 应该使用 `StringBuffer`。 或者 `join` 一个字符串的数组。

```

example16() {
    var sb = new StringBuffer(), a = ["a", "b", "c", "d"], e;
    for (e in a) { sb.write(e); }
    print("Example16 dynamic string created with "
        "StringBuffer '${sb.toString()}'");
    print("Example16 join string array '${a.join()}'");
}

```

// 字符串连接只需让相邻的字符串字面量挨着，
 // 不需要额外的操作符。

```

example17() {
    print("Example17 "
        "concatenate "
        "strings "
        "just like that");
}

```

// 字符串使用单引号或双引号做分隔符，二者并没有实际的差异。
 // 这种灵活性可以很好地避免内容中需要转义分隔符的情况。
 // 例如，字符串内容里的 `HTML` 属性使用了双引号。

```

example18() {
    print('Example18 <a href="etc">'
        "Don't can't I'm Etc"
        '</a>');
}

```

// 用三个单引号或三个双引号表示的字符串
 // 可以跨越多行，并且包含行分隔符。

```

example19() {
    print('''Example19 <a href="etc">
Example19 Don't can't I'm Etc
Example19 </a>''');
}

```

// 字符串可以使用 `$` 字符插入内容。
 // 使用 `$ { [expression] }` 的形式，表达式的值会被插入到字符串中。
 // `$` 跟着一个变量名会插入变量的值。

```
// 如果要在字符串中插入 $ ，可以使用 \$ 的转义形式代替。
example20() {
    var s1 = '\${s}', s2 = '\$s';
    print("Example20 \$ interpolation ${s1} or $s2 works.");
}

// 可选类型允许作为 API 的标注，并且可以辅助 IDE，
// 这样 IDE 可以更好地提供重构、自动完成和错误检测功能。
// 目前为止我们还没有声明任何类型，并且程序运行地很好。
// 事实上，类型在运行时会被忽略。
// 类型甚至可以是错的，并且程序依然可以执行，
// 好像和类型完全无关一样。
// 有一个运行时参数可以让程序进入检查模式，它会在运行时检查类型错误。
// 这在开发时很有用，但是由于增加了额外的检查会使程序变慢，
// 因此应该避免在部署时使用。
class Example21 {
    List<String> _names;
    Example21() {
        _names = ["a", "b"];
    }
    List<String> get names => _names;
    set names(List<String> list) {
        _names = list;
    }
    int get length => _names.length;
    void add(String name) {
        _names.add(name);
    }
}

void example21() {
    Example21 o = new Example21();
    o.add("c");
    print("Example21 names '${o.names}' and length '${o.length}'");
    o.names = ["d", "e"];
    print("Example21 names '${o.names}' and length '${o.length}'");
}

// 类的继承形式是 class name extends AnotherClassName {} 。
class Example22A {
    var _name = "Some Name!";
    get name => _name;
}

class Example22B extends Example22A {}

example22() {
    var o = new Example22B();
    print("Example22 class inheritance '${o.name}'");
}

// 类也可以使用 mixin 的形式：
// class name extends SomeClass with AnotherClassName {}。
// 必需继承某个类才能 mixin 另一个类。
```

```

// 当前 mixin 的模板类不能有构造函数。
// Mixin 主要是用来和辅助的类共享方法的，
// 这样单一继承就不会影响代码复用。
// Mixin 声明在类定义的 "with" 关键词后面。
class Example23A {}
class Example23Utils {
    addTwo(n1, n2) {
        return n1 + n2;
    }
}
class Example23B extends Example23A with Example23Utils {
    addThree(n1, n2, n3) {
        return addTwo(n1, n2) + n3;
    }
}
example23() {
    var o = new Example23B(), r1 = o.addThree(1, 2, 3),
        r2 = o.addTwo(1, 2);
    print("Example23 addThree(1, 2, 3) results in '${r1}'");
    print("Example23 addTwo(1, 2) results in '${r2}'");
}

```

// 类的构造函数名和类名相同，形式为
 // SomeClass() : super() {}, 其中 ": super()" 的部分是可选的，
 // 它用来传递参数给父类的构造函数。

```

class Example24A {
    var _value;
    Example24A({value: "someValue"}) {
        _value = value;
    }
    get value => _value;
}
class Example24B extends Example24A {
    Example24B({value: "someOtherValue"}) : super(value: value);
}
example24() {
    var o1 = new Example24B(),
        o2 = new Example24B(value: "evenMore");
    print("Example24 calling super during constructor '${o1.value}'");
    print("Example24 calling super during constructor '${o2.value}'");
}

```

// 对于简单的类，有一种设置构造函数参数的快捷方式。
 // 只需要使用 this.parameterName 的前缀，
 // 它就会把参数设置为同名的实例变量。

```

class Example25 {
    var value, anotherValue;
    Example25({this.value, this.anotherValue});
}
example25() {
    var o = new Example25(value: "a", anotherValue: "b");
}

```

```

    print("Example25 shortcut for constructor '${o.value}' and "
          "'${o.anotherValue}");
}

```

// 可以在大括号 {} 中声明命名参数。
 // 大括号 {} 中声明的参数的顺序是随意的。
 // 在中括号 [] 中声明的参数也是可选的。

```

example26() {
  var _name, _surname, _email;
  setConfig1({name, surname}) {
    _name = name;
    _surname = surname;
  }
  setConfig2(name, [surname, email]) {
    _name = name;
    _surname = surname;
    _email = email;
  }
  setConfig1(surname: "Doe", name: "John");
  print("Example26 name '${_name}', surname '${_surname}', "
        "email '${_email}");
  setConfig2("Mary", "Jane");
  print("Example26 name '${_name}', surname '${_surname}', "
        "email '${_email}");
}

```

// 使用 final 声明的变量只能被设置一次。
 // 在类里面，final 实例变量可以通过常量的构造函数参数设置。

```

class Example27 {
  final color1, color2;
  // 更灵活一点的方法是在冒号 : 后面设置 final 实例变量。
  Example27({this.color1, color2}) : color2 = color2;
}

example27() {
  final color = "orange", o = new Example27(color1: "lilac", color2: "white");
  print("Example27 color is '${color}");
  print("Example27 color is '${o.color1}' and '${o.color2}");
}

```

// 要导入一个库，使用 import "libraryPath" 的形式，或者如果要导入的是
 // 核心库使用 import "dart:libraryName" 。还有一个称为 "pub" 的包管理工具，
 // 它使用 import "package:packageName" 的约定形式。
 // 看下这个文件顶部的 import "dart:collection"; 语句。
 // 导入语句必需在其它代码声明之前出现。IterableBase 来自于 dart:collection 。

```

class Example28 extends IterableBase {
  var names;
  Example28() {
    names = ["a", "b"];
  }
  get iterator => names.iterator;
}

```

```
example28() {
    var o = new Example28();
    o.forEach((name) => print("Example28 '${name}'"));
}
```

```
// 对于控制流语句，我们有：
// * 必需带 break 的标准 switch 语句
// * if-else 和三元操作符 ..?....
// * 闭包和匿名函数
// * break, continue 和 return 语句
```

```
example29() {
    var v = true ? 30 : 60;
    switch (v) {
        case 30:
            print("Example29 switch statement");
            break;
    }
    if (v < 30) {
    } else if (v > 30) {
    } else {
        print("Example29 if-else statement");
    }
    callItForMe(fn()) {
        return fn();
    }
    rand() {
        v = new DM.Random().nextInt(50);
        return v;
    }
    while (true) {
        print("Example29 callItForMe(rand) '${callItForMe(rand)}'");
        if (v != 30) {
            break;
        } else {
            continue;
        }
        // 不会到这里。
    }
}
```

```
// 解析 int，把 double 转成 int，或者使用 ~/ 操作符在除法计算时仅保留整数位。
// 让我们也来场猜数游戏吧。
```

```
example30() {
    var gn, tooHigh = false,
        n, n2 = (2.0).toInt(), top = int.parse("123") ~/ n2, bottom = 0;
    top = top ~/ 6;
    gn = new DM.Random().nextInt(top + 1); // +1 because nextInt top is exclusive
    print("Example30 Guess a number between 0 and ${top}");
    guessNumber(i) {
        if (n == gn) {
            print("Example30 Guessed right! The number is ${gn}");
        }
    }
}
```

```

    } else {
        tooHigh = n > gn;
        print("Example30 Number ${n} is too "
            "${tooHigh ? 'high' : 'low'}. Try again");
    }
    return n == gn;
}
n = (top - bottom) ~/ 2;
while (!guessNumber(n)) {
    if (tooHigh) {
        top = n - 1;
    } else {
        bottom = n + 1;
    }
    n = bottom + ((top - bottom) ~/ 2);
}
}

// 程序的唯一入口点是 main 函数。
// 在程序开始执行 main 函数之前，不期望执行任何外层代码。
// 这样可以帮助程序更快地加载，甚至仅惰性加载程序启动时需要的部分。
main() {
    print("Learn Dart in 15 minutes!");
    [example1, example2, example3, example4, example5, example6, example7,
        example8, example9, example10, example11, example12, example13, example14,
        example15, example16, example17, example18, example19, example20,
        example21, example22, example23, example24, example25, example26,
        example27, example28, example29, example30
    ].forEach((ef) => ef());
}

```

延伸阅读

Dart 有一个综合性网站。它涵盖了 API 参考、入门向导、文章以及更多，还包括一个有用的在线试用 Dart 页面。 <http://www.dartlang.org/> <http://try.dartlang.org/>

language: elisp contributors:

```
- ["Bastien Guerry", "http://bzg.fr"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net"]
```

filename: learn-emacs-lisp-zh.el

lang: zh-cn

```
;; 15分钟学会Emacs Lisp (v0.2a)
;; (作者: bzg, https://github.com/bzg
;; 译者: lichenbo, http://douban.com/people/lichenbo)
;;
;; 请先阅读Peter Norvig的一篇好文:
;; http://norvig.com/21-days.html
;; (译者注: 中文版请见http://blog.youxu.info/21-days/)
;;
;; 之后安装GNU Emacs 24.3:
;;
;; Debian: apt-get install emacs (视具体发行版而定)
;; MacOSX: http://emacsformacosx.com/emacs-builds/Emacs-24.3-universal-10.6.8.dmg
;; Windows: http://ftp.gnu.org/gnu/windows/emacs/emacs-24.3-bin-i386.zip
;;
;; 更多信息可以在这里找到:
;; http://www.gnu.org/software/emacs/#Obtaining

;; 很重要的警告:
;;
;; 按照这个教程来学习并不会对你的电脑有任何损坏
;; 除非你自己在学习的过程中愤怒地把它砸了
;; 如果出现了这种情况, 我不会承担任何责任
;;
;; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 打开emacs
;;
;; 按'q'消除欢迎界面
;;
;; 现在请注意窗口底部的那一个灰色长条
;;
;; "*scratch*" 是你现在编辑界面的名字。
;; 这个编辑界面叫做一个"buffer"。
```

```

;;
;; 每当你打开Emacs时，都会默认打开这个scratch buffer
;; 此时你并没有在编辑任何文件，而是在编辑一个buffer
;; 之后你可以将这个buffer保存到一个文件中。
;;
;; 之后的"Lisp interaction" 则是表明我们可以用的某组命令
;;
;; Emacs在每个buffer中都有一组内置的命令
;; 而当你激活某种特定的模式时，就可以使用相应的命令
;; 这里我们使用`lisp-interaction-mode'，
;; 这样我们就可以使用内置的Emacs Lisp（以下简称Elisp）命令了。

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 分号是注释开始的标志
;;
;; Elisp 是由符号表达式构成的（即"s-表达式"或"s式"）：
(+ 2 2)

;; 这个s式的意思是 "对2进行加2操作"。

;; s式周围有括号，而且也可以嵌套：
(+ 2 (+ 1 1))

;; 一个s式可以包含原子符号或者其他s式
;; 在上面的例子中，1和2是原子符号
;; (+ 2 (+ 1 1)) 和 (+ 1 1) 是s式。

;; 在 `lisp-interaction-mode' 中你可以计算s式。
;; 把光标移到闭括号后，之后按下ctrl+j（以后简写为'C-j'）

(+ 3 (+ 1 2))
;;          ^ 光标放到这里
;; 按下`C-j' 就会输出 6

;; `C-j' 会在buffer中插入当前运算的结果

;; 而`C-xC-e' 则会在emacs最底部显示结果，也就是被称作"minibuffer"的区域
;; 为了避免把我们的buffer填满无用的结果，我们以后会一直用`C-xC-e'

;; `setq' 可以将一个值赋给一个变量
(setq my-name "Bastien")
;; `C-xC-e' 输出 "Bastien"（在 mini-buffer 中显示）

;; `insert' 会在光标处插入字符串：
(insert "Hello!")
;; `C-xC-e' 输出 "Hello!"

;; 在这里我们只传给了insert一个参数"Hello!"，但是
;; 我们也可以传给它更多的参数，比如2个：

```



```
(insert "Hello" " world!")
;; `C-xC-e' 输出 "Hello world!"

;; 你也可以用变量名来代替字符串
(insert "Hello, I am " my-name)
;; `C-xC-e' 输出 "Hello, I am Bastien"

;; 你可以把s式嵌入函数中
(defun hello () (insert "Hello, I am " my-name))
;; `C-xC-e' 输出 hello

;; 现在执行这个函数
(hello)
;; `C-xC-e' 输出 Hello, I am Bastien

;; 函数中空括号的意思是我们不需要接受任何参数
;; 但是我们不能一直总是用my-name这个变量
;; 所以现在我们使我们的函数接受一个叫做"name"的参数

(defun hello (name) (insert "Hello " name))
;; `C-xC-e' 输出 hello

;; 现在我们调用这个函数，并且将"you"作为参数传递

(hello "you")
;; `C-xC-e' 输出 "Hello you"

;; 成功！

;; 现在我们可以休息一下

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; 下面我们在新的窗口中新建一个名为 "*test*" 的buffer:

(switch-to-buffer-other-window "*test*")
;; `C-xC-e' 这时屏幕上会显示两个窗口，而光标此时位于*test* buffer内

;; 用鼠标单击上面的buffer就会使光标移回。
;; 或者你可以使用 `C-xo' 使得光标跳到另一个窗口中

;; 你可以用 `progn' 命令将s式结合起来：
(progn
  (switch-to-buffer-other-window "*test*")
  (hello "you"))
;; `C-xC-e' 此时屏幕分为两个窗口，并且在*test* buffer中显示"Hello you"

;; 现在为了简洁，我们需要在每个s式后面都使用`C-xC-e'来执行，后面就不再说明了

;; 记得可以用过鼠标或者`C-xo'回到*scratch*这个buffer。
```

;; 清除当前buffer也是常用操作之一:

```
(progn
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello "there"))
```

;; 也可以回到其他的窗口中

```
(progn
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello "you")
  (other-window 1))
```

;; 你可以用 `let' 将一个值和一个局部变量绑定:

```
(let ((local-name "you"))
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (hello local-name)
  (other-window 1))
```

;; 这里我们就不需要使用 `progn' 了, 因为 `let' 也可以将很多s式组合起来。

;; 格式化字符串的方法:

```
(format "Hello %s!\n" "visitor")
```

;; %s 是字符串占位符, 这里被"visitor"替代。

;; \n 是换行符。

;; 现在我们用格式化的方法再重写一下我们的函数:

```
(defun hello (name)
  (insert (format "Hello %s!\n" name)))
```

```
(hello "you")
```

;; 我们再用 `let' 新建另一个函数:

```
(defun greeting (name)
  (let ((your-name "Bastien"))
    (insert (format "Hello %s!\n\nI am %s."
                    name           ; the argument of the function
                    your-name      ; the let-bound variable "Bastien"
                    ))))
```

;; 之后执行:

```
(greeting "you")
```

;; 有些函数可以和用户交互:

```
(read-from-minibuffer "Enter your name: ")
```

;; 这个函数会返回在执行时用户输入的信息

;; 现在我们让 `greeting' 函数显示你的名字:

```
(defun greeting (from-name)
  (let ((your-name (read-from-minibuffer "Enter your name: ")))
    (insert (format "Hello!\n\nI am %s and you are %s."
                    from-name ; the argument of the function
                    your-name ; the let-bound var, entered at prompt
                    ))))
```

```
(greeting "Bastien")
```

;; 我们让结果在另一个窗口中显示:

```
(defun greeting (from-name)
  (let ((your-name (read-from-minibuffer "Enter your name: ")))
    (switch-to-buffer-other-window "*test*")
    (erase-buffer)
    (insert (format "Hello %s!\n\nI am %s." your-name from-name))
    (other-window 1)))
```

;; 测试一下:

```
(greeting "Bastien")
```

;; 第二节结束, 休息一下吧。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;
```

;; 我们将一些名字存到列表中:

```
(setq list-of-names '("Sarah" "Chloe" "Mathilde"))
```

;; 用 `car' 来取得第一个名字:

```
(car list-of-names)
```

;; 用 `cdr' 取得剩下的名字:

```
(cdr list-of-names)
```

;; 用 `push' 把名字添加到列表的开头:

```
(push "Stephanie" list-of-names)
```

;; 注意: `car' 和 `cdr' 并不修改列表本身, 但是 `push' 却会对列表本身进行操作。

;; 这个区别是很重要的: 有些函数没有任何副作用 (比如 `car')

;; 但还有一些却是有的 (比如 `push')。

;; 我们来对 `list-of-names' 列表中的每一个元素都使用 hello 函数:

```
(mapcar 'hello list-of-names)
```

;; 将 `greeting' 改进, 使我们能够对 `list-of-names' 中的所有名字执行:

```
(defun greeting ()
  (switch-to-buffer-other-window "*test*")
  (erase-buffer)
  (mapcar 'hello list-of-names)
  (other-window 1))
```

```
(greeting)
```

```
;; 记得我们之前定义的 `hello' 函数吗？ 这个函数接受一个参数，名字。  
;; `mapcar' 调用 `hello'，并将 `list-of-names' 作为参数先后传给 `hello'
```

```
;; 现在我们对显示的buffer中的内容进行一些更改：
```

```
(defun replace-hello-by-bonjour ()  
  (switch-to-buffer-other-window "*test*")  
  (goto-char (point-min))  
  (while (search-forward "Hello")  
    (replace-match "Bonjour"))  
  (other-window 1))
```

```
;; (goto-char (point-min)) 将光标移到buffer的开始  
;; (search-forward "Hello") 查找字符串"Hello"  
;; (while x y) 当x返回某个值时执行y这个s式  
;; 当x返回`nil' (空)，退出循环
```

```
(replace-hello-by-bonjour)
```

```
;; 你会看到所有在*test* buffer中出现的"Hello"字样都被换成了"Bonjour"
```

```
;; 你也会得到以下错误提示: "Search failed: Hello".  
;;  
;; 如果要避免这个错误，你需要告诉 `search-forward' 这个命令是否在  
;; buffer的某个地方停止查找，并且在什么都没找到时是否应该不给出错误提示
```

```
;; (search-forward "Hello" nil t) 可以达到这个要求：
```

```
;; `nil' 参数的意思是：查找并不限于某个范围内  
;; `t' 参数的意思是：当什么都没找到时，不给出错误提示
```

```
;; 在下面的函数中，我们用到了s式，并且不给出任何错误提示：
```

```
(defun hello-to-bonjour ()  
  (switch-to-buffer-other-window "*test*")  
  (erase-buffer)  
  ;; 为 `list-of-names' 中的每个名字调用hello  
  (mapcar 'hello list-of-names)  
  (goto-char (point-min))  
  ;; 将"Hello" 替换为"Bonjour"  
  (while (search-forward "Hello" nil t)  
    (replace-match "Bonjour"))  
  (other-window 1))
```

```
(hello-to-bonjour)
```

```
;; 给这些名字上个色：
```

```
(defun boldify-names ()  
  (switch-to-buffer-other-window "*test*")
```

```

(goto-char (point-min))
(while (re-search-forward "Bonjour \\(\\.+\\)!" nil t)
  (add-text-properties (match-beginning 1)
    (match-end 1)
    (list 'face 'bold)))

(other-window 1))

```

;; 这个函数使用了 `re-search-forward':
 ;; 和查找一个字符串不同，你用这个命令可以查找一个模式，即正则表达式

;; 正则表达式 "Bonjour \\(\\.+\\)!" 的意思是：
 ;; 字符串 "Bonjour "，之后跟着
 ;; 一组 | \\(... \\) 结构
 ;; 任意字符 | . 的含义
 ;; 有可能重复的 | + 的含义
 ;; 之后跟着 "!" 这个字符串

;; 准备好了？试试看。

```
(boldify-names)
```

;; `add-text-properties' 可以添加文字属性，比如文字样式

;; 好的，我们成功了！

;; 如果你想对一个变量或者函数有更多的了解：
 ;;
 ;; C-h v 变量 回车
 ;; C-h f 函数 回车
 ;;
 ;; 阅读Emacs Lisp官方文档：
 ;;
 ;; C-h i m elisp 回车
 ;;
 ;; 在线阅读Emacs Lisp文档：
 ;; https://www.gnu.org/software/emacs/manual/html_node/eintr/index.html

;; 感谢以下同学的建议和反馈：
 ;; - Wes Hardaker
 ;; - notbob
 ;; - Kevin Montuori
 ;; - Arne Babenhauserheide
 ;; - Alan Schmitt

language: elixir contributors:

```
- ["Joao Marques", "http://github.com/mrshankly"]
```

translators:

```
- ["lidashuang", "http://github.com/lidashuang"]
```

filename: learnelixir-cn.ex

lang: zh-cn

Elixir 是一门构建在Erlang VM 之上的函数式编程语言。Elixir 完全兼容 Erlang, 另外还提供了更标准的语法, 特性。

```
# 这是单行注释，注释以井号开头

# 没有多行注释
# 但你可以堆叠多个注释。

# elixir shell 使用命令 `iex` 进入。
# 编译模块使用 `elixirc` 命令。

# 如果安装正确，这些命令都会在环境变量里

## -----
## -- 基本类型
## -----

# 数字
3      # 整型
0x1F   # 整型
3.0    # 浮点类型

# 原子(Atoms)，以 `:` 开头
:hello # atom

# 元组(Tuple) 在内存中的存储是连续的
{1,2,3} # tuple

# 使用`elem`函数访问元组(tuple)里的元素：
elem({1, 2, 3}, 0) #=> 1

# 列表(list)
```

```

[1,2,3] # list

# 可以用下面的方法访问列表的头尾元素：
[head | tail] = [1,2,3]
head #=> 1
tail #=> [2,3]

# 在elixir,就像在Erlang, `=` 表示模式匹配 (pattern matching)
# 不是赋值。
#
# 这表示会用左边的模式(pattern)匹配右侧
#
# 上面的例子中访问列表的头部和尾部就是这样工作的。

# 当左右两边不匹配时, 会返回error, 在这个
# 例子中, 元组大小不一样。
# {a, b, c} = {1, 2} #=> ** (MatchError) no match of right hand side value: {1,2}

# 还有二进制类型 (binaries)
<<1,2,3>> # binary

# 字符串(Strings) 和 字符列表(char lists)
"hello" # string
'hello' # char list

# 多行字符串
"""
I'm a multi-line
string.
"""
#=> "I'm a multi-line\nstring.\n"

# 所有的字符串(Strings)以UTF-8编码:
"héllò" #=> "héllò"

# 字符串(Strings)本质就是二进制类型(binaries), 字符列表(char lists)本质是列表(lists)
<<?a, ?b, ?c>> #=> "abc"
[?a, ?b, ?c]   #=> 'abc'

# 在 elixir中, `?a` 返回 `a` 的 ASCII 整型值
?a #=> 97

# 合并列表使用 `++`, 对于二进制类型则使用 `<>`
[1,2,3] ++ [4,5]      #=> [1,2,3,4,5]
'hello ' ++ 'world'   #=> 'hello world'

<<1,2,3>> <> <<4,5>>  #=> <<1,2,3,4,5>>
"hello " <> "world"   #=> "hello world"

## -----
## -- 操作符(Operators)

```

```

## -----

# 一些数学运算
1 + 1 ==> 2
10 - 5 ==> 5
5 * 2 ==> 10
10 / 2 ==> 5.0

# 在 elixir 中，操作符 `/` 返回值总是浮点数。

# 做整数除法使用 `div`
div(10, 2) ==> 5

# 为了得到余数使用 `rem`
rem(10, 3) ==> 1

# 还有 boolean 操作符: `or`, `and` and `not`.
# 第一个参数必须是boolean 类型
true and true ==> true
false or true ==> true
# 1 and true ==> ** (ArgumentError) argument error

# Elixir 也提供了 `||`, `&&` 和 `!` 可以接受任意的类型
# 除了`false` 和 `nil` 其它都会被当作true.
1 || true ==> 1
false && 1 ==> false
nil && 20 ==> nil

!true ==> false

# 比较有: `==`, `!=`, `===`, `!==`, `<=`, `>=`, `<` 和 `>`
1 == 1 ==> true
1 != 1 ==> false
1 < 2 ==> true

# `===` 和 `!==` 在比较整型和浮点类型时更为严格:
1 == 1.0 ==> true
1 === 1.0 ==> false

# 我们也可以比较两种不同的类型:
1 < :hello ==> true

# 总的排序顺序定义如下:
# number < atom < reference < functions < port < pid < tuple < list < bit string

# 引用Joe Armstrong : “实际的顺序并不重要,
# 但是, 一个整体排序是否经明确界定是非常重要的。”

## -----
## -- 控制结构(Control Flow)
## -----

```



```

# `if` 表达式
if false do
  "This will never be seen"
else
  "This will"
end

# 还有 `unless`
unless true do
  "This will never be seen"
else
  "This will"
end

# 在Elixir中，很多控制结构都依赖于模式匹配

# `case` 允许我们把一个值与多种模式进行比较：
case {:one, :two} do
  {:four, :five} ->
    "This won't match"
  {:one, x} ->
    "This will match and assign `x` to `:two`"
  _ ->
    "This will match any value"
end

# 模式匹配时，如果不需要某个值，通用的做法是把值 匹配到 `_`
# 例如，我们只需要要列表的头元素：
[head | _] = [1,2,3]
head #=> 1

# 下面的方式效果一样，但可读性更好
[head | _tail] = [:a, :b, :c]
head #=> :a

# `cond` 可以检测多种不同的分支
# 使用 `cond` 代替多个 `if` 表达式嵌套
cond do
  1 + 1 == 3 ->
    "I will never be seen"
  2 * 5 == 12 ->
    "Me neither"
  1 + 2 == 3 ->
    "But I will"
end

# 经常可以看到最后一个条件等于 'true'，这将总是匹配。
cond do
  1 + 1 == 3 ->
    "I will never be seen"

```

```

2 * 5 == 12 ->
  "Me neither"
true ->
  "But I will (this is essentially an else)"
end

# `try/catch` 用于捕获被抛出的值，它支持 `after` 子句，
# 无论是否值被捕获，`after` 子句都会被调用
# `try/catch`
try do
  throw(:hello)
catch
  message -> "Got #{message}."
after
  IO.puts("I'm the after clause.")
end
#=> I'm the after clause
# "Got :hello"

## -----
## -- 模块和函数(Modules and Functions)
## -----

# 匿名函数 (注意点)
square = fn(x) -> x * x end
square.(5) #=> 25

# 也支持接收多个子句和卫士(guards)。
# Guards 可以进行模式匹配
# Guards 使用 `when` 关键字指明：
f = fn
  x, y when x > 0 -> x + y
  x, y -> x * y
end

f.(1, 3) #=> 4
f.(-1, 3) #=> -3

# Elixir 提供了很多内建函数
# 在默认作用域都是可用的
is_number(10) #=> true
is_list("hello") #=> false
elem({1,2,3}, 0) #=> 1

# 你可以在一个模块里定义多个函数，定义函数使用 `def`
defmodule Math do
  def sum(a, b) do
    a + b
  end
end

```

```

def square(x) do
  x * x
end

Math.sum(1, 2) #=> 3
Math.square(3) #=> 9

# 保存到 `math.ex`, 使用 `elixirc` 编译你的 Math 模块
# 在终端里: elixirc math.ex

# 在模块中可以使用 `def` 定义函数, 使用 `defp` 定义私有函数
# 使用 `def` 定义的函数可以被其它模块调用
# 私有函数只能在本模块内调用
defmodule PrivateMath do
  def sum(a, b) do
    do_sum(a, b)
  end

  defp do_sum(a, b) do
    a + b
  end
end

PrivateMath.sum(1, 2)      #=> 3
# PrivateMath.do_sum(1, 2) #=> ** (UndefinedFunctionError)

# 函数定义同样支持 guards 和 多重子句:
defmodule Geometry do
  def area({:rectangle, w, h}) do
    w * h
  end

  def area({:circle, r}) when is_number(r) do
    3.14 * r * r
  end
end

Geometry.area({:rectangle, 2, 3}) #=> 6
Geometry.area({:circle, 3})      #=> 28.2599999999999801048
# Geometry.area({:circle, "not_a_number"})
#=> ** (FunctionClauseError) no function clause matching in Geometry.area/1

#由于不变性, 递归是Elixir的重要组成部分
defmodule Recursion do
  def sum_list([head | tail], acc) do
    sum_list(tail, acc + head)
  end

  def sum_list([], acc) do

```

```

    acc
  end
end

Recursion.sum_list([1,2,3], 0) #=> 6

# Elixir 模块支持属性，模块内建了一些属性，你也可以自定义属性
defmodule MyMod do
  @moduledoc """
    内置的属性，模块文档
    """

  @my_data 100 # 自定义属性
  IO.inspect(@my_data) #=> 100
end

## -----
## -- 记录和异常(Records and Exceptions)
## -----

# 记录就是把特定值关联到某个名字的结构体
defrecord Person, name: nil, age: 0, height: 0

joe_info = Person.new(name: "Joe", age: 30, height: 180)
#=> Person[name: "Joe", age: 30, height: 180]

# 访问name的值
joe_info.name #=> "Joe"

# 更新age的值
joe_info = joe_info.age(31) #=> Person[name: "Joe", age: 31, height: 180]

# 使用 `try` `rescue` 进行异常处理
try do
  raise "some error"
rescue
  RuntimeError -> "rescued a runtime error"
  _error -> "this will rescue any error"
end

# 所有的异常都有一个message
try do
  raise "some error"
rescue
  x in [RuntimeError] ->
    x.message
end

## -----
## -- 并发(Concurrency)
## -----

```

```

# Elixir 依赖于 actor并发模型。在Elixir编写并发程序的三要素：
# 创建进程，发送消息，接收消息

# 启动一个新的进程使用`spawn`函数，接收一个函数作为参数

f = fn -> 2 * 2 end #=> #Function<erl_eval.20.80484245>
spawn(f) #=> #PID<0.40.0>

# `spawn` 函数返回一个pid(进程标识符)，你可以使用pid向进程发送消息。
# 使用 `<-` 操作符发送消息。
# 我们需要在进程内接收消息，要用到 `receive` 机制。

defmodule Geometry do
  def area_loop do
    receive do
      {:rectangle, w, h} ->
        IO.puts("Area = #{w * h}")
        area_loop()
      {:circle, r} ->
        IO.puts("Area = #{3.14 * r * r}")
        area_loop()
    end
  end
end

# 编译这个模块，在shell中创建一个进程，并执行 `area_loop` 函数。
pid = spawn(fn -> Geometry.area_loop() end) #=> #PID<0.40.0>

# 发送一个消息给 `pid`， 会在receive语句进行模式匹配
pid <- {:rectangle, 2, 3}
#=> Area = 6
#   {:rectangle,2,3}

pid <- {:circle, 2}
#=> Area = 12.56000000000000049738
#   {:circle,2}

# shell也是一个进程(process)，你可以使用`self`获取当前 pid
self() #=> #PID<0.27.0>

```

参考文献

- [Getting started guide from elixir webpage](#)
- [Elixir Documentation](#)
- ["Learn You Some Erlang for Great Good!"](#) by Fred Hebert
- ["Programming Erlang: Software for a Concurrent World"](#) by Joe Armstrong

language: erlang lang: zh-cn contributors:

```
- ["Giovanni Cappellotto", "http://www.focustheweb.com/"]
```

translators:

```
- ["Jakukyo Friel", "http://weakish.github.io"]
```

filename: erlang-cn.erl

```
% 百分比符号标明注释的开始。

%% 两个符号通常用于注释函数。

%%% 三个符号通常用于注释模块。

% Erlang 里使用三种标点符号：
% 逗号 (`,` ) 分隔函数调用中的参数、数据构建和模式。
% 句号 (`.`) (后跟空格) 分隔函数和 shell 中的表达式。
% 分号 (`;`) 分隔语句。以下环境中使用语句：
% 函数定义和 `case`、`if`、`try..catch`、`receive` 表达式。

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 1. 变量和模式匹配
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Num = 42. % 变量必须以大写字母开头。

% Erlang 的变量只能赋值一次。如果给变量赋不同的值，会导致错误：
Num = 43. % ** exception error: no match of right hand side value 43

% 大多数语言中 `=` 表示赋值语句，在 Erlang 中，则表示模式匹配。
% `Lhs = Rhs` 实际上意味着：
% 演算右边(Rhs)，将结果与左边的模式匹配。
Num = 7 * 6.

% 浮点数
Pi = 3.14159.

% Atoms 用于表示非数字的常量。
% Atom 以小写字母开始，包含字母、数字、`_` 和 `@`。
Hello = hello.
OtherNode = example@node.

% Atom 中如果包含特殊字符，可以用单引号括起。
```

```

AtomWithSpace = 'some atom with space'.

% Erlang 的元组类似 C 的 struct.
Point = {point, 10, 45}.

% 使用模式匹配操作符`=`获取元组的值。
{point, X, Y} = Point.  % X = 10, Y = 45

% 我们可以使用`_`存放我们不感兴趣的变量。
% `_`被称为匿名变量。和其他变量不同，
% 同一个模式中的多个`_`变量不必绑定到相同的值。
Person = {person, {name, {first, joe}, {last, armstrong}}, {footsize, 42}}.
{_, {_, {_, Who}, _}, _} = Person.  % Who = joe

% 列表使用方括号，元素间使用逗号分隔。
% 列表的元素可以是任意类型。
% 列表的第一个元素称为列表的 head，其余元素称为列表的 tail。
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}].

% 若`T`是一个列表，那么`[H|T]`同样是一个列表，head为`H`，tail为`T`。
% `|`分隔列表的 head 和 tail.
% `[]`是空列表。
% 我们可以使用模式匹配操作来抽取列表中的元素。
% 如果我们有一个非空的列表`L`，那么`[X|Y] = L`则
% 抽取 L 的 head 至 X，tail 至 Y（X、Y需为未定义的变量）。
[FirstThing|OtherThingsToBuy] = ThingsToBuy.
% FirstThing = {apples, 10}
% OtherThingsToBuy = {pears, 6}, {milk, 3}

% Erlang 中的字符串其实是由整数组成的数组。字符串使用双引号。
Name = "Hello".
[72, 101, 108, 108, 111] = "Hello".

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 2. 循序编程
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Module 是 Erlang 代码的基本单位。我们编写的所有函数都储存在 module 中。
% Module 存储在后缀为`.erl`的文件中。
% Module 必须事先编译。编译好的 module 以`.beam`结尾。
-module(geometry).
-export([area/1]). % module 对外暴露的函数列表

% `area`函数包含两个分句，分句间以分号相隔。
% 最后一个分句以句号加换行结尾。
% 每个分句由头、体两部分组成。
% 头部包含函数名称和用括号括起的模式，
% 尾部包含一系列表达式，如果头部的模式和调用时的参数匹配，这些表达式会被演算。
% 模式匹配依照定义时的顺序依次进行。
area({rectangle, Width, Ht}) -> Width * Ht;

```

```

area({circle, R})          -> 3.14159 * R * R.

% 编译文件为 geometry.erl.
c(geometry). % {ok,geometry}

% 调用函数时必须使用 module 名和函数名。
geometry:area({rectangle, 10, 5}). % 50
geometry:area({circle, 1.4}). % 6.15752

% 在 Erlang 中，同一模块中，参数数目不同，名字相同的函数是完全不同的函数。
-module(lib_misc).
-export([sum/1]). % 对外暴露的`sum`函数接受一个参数：由整数组成的列表。
sum(L) -> sum(L, 0).
sum([], N) -> N;
sum([H|T], N) -> sum(T, H+N).

% fun 是匿名函数。它们没有名字，不过可以赋值给变量。
Double = fun(X) -> 2*X end. % `Double` 指向匿名函数 #Fun<erl_eval.6.17052888>
Double(2). % 4

% fun 可以作为函数的参数和返回值。
Mult = fun(Times) -> ( fun(X) -> X * Times end ) end.
Triple = Mult(3).
Triple(5). % 15

% 列表解析是创建列表的表达式（不使用fun、map 或 filter）。
% `[F(X) || X <- L]` 表示 "由 `F(X)` 组成的列表，其中 `X` 取自列表 `L`"。
L = [1,2,3,4,5].
[2*X || X <- L]. % [2,4,6,8,10]
% 列表解析可以使用生成器，也可以使用过滤器，过滤器用于筛选列表的一部分。
EvenNumbers = [N || N <- [1, 2, 3, 4], N rem 2 == 0]. % [2, 4]

% Guard 是用于增强模式匹配的结构。
% Guard 可用于简单的测试和比较。
% Guard 可用于函数定义的头部，以`when`关键字开头，或者其他可以使用表达式的地方。
max(X, Y) when X > Y -> X;
max(X, Y) -> Y.

% guard 可以由一系列 guard 表达式组成，这些表达式以逗号分隔。
% `GuardExpr1, GuardExpr2, ..., GuardExprN` 为真，当且仅当每个 guard 表达式均为真。
is_cat(A) when is_atom(A), A == cat -> true;
is_cat(A) -> false.
is_dog(A) when is_atom(A), A == dog -> true;
is_dog(A) -> false.

% guard 序列 `G1; G2; ...; Gn` 为真，当且仅当其中任意一个 guard 表达式为真。
is_pet(A) when is_dog(A); is_cat(A) -> true;
is_pet(A) -> false.

% Record 可以将元组中的元素绑定到特定的名称。
% Record 定义可以包含在 Erlang 源代码中，也可以放在后缀为`.hrl`的文件中（Erlang 源代码中 include

```



```

-record(todo, {
    status = reminder, % Default value
    who = joe,
    text
}).

% 在定义某个 record 之前，我们需要在 shell 中导入 record 的定义。
% 我们可以使用 shell 函数`rr` (read records 的简称)。
rr("records.hrl"). % [todo]

% 创建和更新 record。
X = #todo{}.
% #todo{status = reminder, who = joe, text = undefined}
X1 = #todo{status = urgent, text = "Fix errata in book"}.
% #todo{status = urgent, who = joe, text = "Fix errata in book"}
X2 = X1#todo{status = done}.
% #todo{status = done,who = joe,text = "Fix errata in book"}

% `case` 表达式。
% `filter` 返回由列表`L`中所有满足`P(x)`为真的元素`X`组成的列表。
filter(P, [H|T]) ->
    case P(H) of
        true -> [H|filter(P, T)];
        false -> filter(P, T)
    end;
filter(P, []) -> [].
filter(fun(X) -> X rem 2 == 0 end, [1, 2, 3, 4]). % [2, 4]

% `if` 表达式。
max(X, Y) ->
    if
        X > Y -> X;
        X < Y -> Y;
        true -> nil;
    end.

% 警告: `if` 表达式里至少有一个 guard 为真, 否则会触发异常。

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% 3. 异常
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 当遇到内部错误或显式调用时, 会触发异常。
% 显式调用包括 `throw(Exception)`, `exit(Exception)` 和
% `erlang:error(Exception)`.
generate_exception(1) -> a;
generate_exception(2) -> throw(a);
generate_exception(3) -> exit(a);
generate_exception(4) -> {'EXIT', a};
generate_exception(5) -> erlang:error(a).

```

% Erlang 有两种捕获异常的方法。其一是将调用包裹在`try...catch`表达式中。

```
catcher(N) ->
  try generate_exception(N) of
    Val -> {N, normal, Val}
  catch
    throw:X -> {N, caught, thrown, X};
    exit:X -> {N, caught, exited, X};
    error:X -> {N, caught, error, X}
  end.
```

% 另一种方式是将调用包裹在`catch`表达式中。

% 此时异常会被转化为一个描述错误的元组。

```
catcher(N) -> catch generate_exception(N).
```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% 4. 并发

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Erlang 依赖于 actor并发模型。在 Erlang 编写并发程序的三要素:

% 创建进程, 发送消息, 接收消息

% 启动一个新的进程使用`spawn`函数, 接收一个函数作为参数

```
F = fun() -> 2 + 2 end. % #Fun<erl_eval.20.67289768>
```

```
spawn(F). % <0.44.0>
```

% `spawn` 函数返回一个pid(进程标识符), 你可以使用pid向进程发送消息。

% 使用 `!` 操作符发送消息。

% 我们需要在进程内接收消息, 要用到 `receive` 机制。

```
-module(caculateGeometry).
-compile(export_all).
caculateAera() ->
  receive
    {rectangle, W, H} ->
      W * H;
    {circle, R} ->
      3.14 * R * R;
    _ ->
      io:format("We can only caculate area of rectangles or circles.")
  end.
```

% 编译这个模块, 在 shell 中创建一个进程, 并执行 `caculateArea` 函数。

```
c(caculateGeometry).
```

```
CaculateAera = spawn(caculateGeometry, caculateAera, []).
```

```
CaculateAera ! {circle, 2}. % 12.56000000000000049738
```

% shell也是一个进程(process), 你可以使用`self`获取当前 pid

```
self(). % <0.41.0>
```



References

- ["Learn You Some Erlang for great good!"](#)
- ["Programming Erlang: Software for a Concurrent World" by Joe Armstrong](#)
- [Erlang/OTP Reference Documentation](#)
- [Erlang - Programming Rules and Conventions](#)

category: tool tool: git contributors:

```
- ["Jake Prather", "http://github.com/JakeHP"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net"]
```

lang: zh-cn

Git是一个分布式版本控制及源代码管理工具

Git可以为你的项目保存若干快照，以此来对整个项目进行版本管理

版本

什么是版本控制

版本控制系统就是根据时间来记录一个或多个文件的更改情况的系统。

集中式版本控制 **VS** 分布式版本控制

- 集中式版本控制的主要功能为同步，跟踪以及备份文件
- 分布式版本控制则更注重共享更改。每一次更改都有唯一的标识
- 分布式系统没有预定的结构。你也可以用git很轻松的实现SVN风格的集中式系统控制

[更多信息](#)

为什么要使用 **Git**

- 可以离线工作
- 和他人协同工作变得简单
- 分支很轻松
- 合并很容易
- Git系统速度快，也很灵活

Git 架构

版本库

一系列文件，目录，历史记录，提交记录和头指针。 可以把它视作每个源代码文件都带有历史记录属性数据结构

一个Git版本库包括一个 `.git` 目录和其工作目录

`.git` 目录(版本库的一部分)

`.git` 目录包含所有的配置、日志、分支信息、头指针等 [详细列表](#)

工作目录 (版本库的一部分)

版本库中的目录和文件，可以看做就是你工作时的目录

索引(`.git` 目录)

索引就是git中的 `staging` 区. 可以算作是把你的工作目录与Git版本库分割开的一层 这使得开发者能够更灵活的决定要将在版本库中添加什么内容

提交

一个 `git` 提交就是一组更改或者对工作目录操作的快照 比如你添加了5个文件，删除了2个文件，那么这些变化就会被写入一个提交比如你添加了5个文件，删除了2个文件，那么这些变化就会被写入一个提交中 而这个提交之后也可以被决定是否推送到另一个版本库中

分支

分支其实就是一个指向你最后一次的提交的指针 当你提交时，这个指针就会自动指向最新的提交

头指针 与 头(`.git` 文件夹的作用)

头指针是一个指向当前分支的指针，一个版本库只有一个当前活动的头指针 而头则可以指向版本库中任意一个提交，每个版本库也可以有多个头

其他形象化解释

- [给计算机科学家的解释](#)
- [给设计师的解释](#)

命令

初始化

创建一个新的git版本库。这个版本库的配置、存储等信息会被保存到`.git`文件夹中

```
$ git init
```

配置

更改设置。可以是版本库的设置，也可以是系统的或全局的

```
# 输出、设置基本的全局变量
$ git config --global user.email
$ git config --global user.name

$ git config --global user.email "MyEmail@Zoho.com"
$ git config --global user.name "My Name"
```

[关于git的更多设置](#)

帮助

git内置了对命令非常详细的解释，可以供我们快速查阅

```
# 查找可用命令
$ git help

# 查找所有可用命令
$ git help -a

# 在文档当中查找特定的命令
# git help <命令>
$ git help add
$ git help commit
$ git help init
```

状态

显示索引文件（也就是当前工作空间）和当前的头指针指向的提交的不同

```
# 显示分支，为跟踪文件，更改和其他不同
$ git status

# 查看其他的git status的用法
$ git help status
```

添加

添加文件到当前工作空间中。如果你不使用 `git add` 将文件添加进去，那么这些文件也不会添加到之后的提交之中

```
# 添加一个文件
$ git add HelloWorld.java

# 添加一个子目录中的文件
$ git add /path/to/file/HelloWorld.c

# 支持正则表达式
$ git add ./*.java
```

分支

管理分支，可以通过下列命令对分支进行增删改查

```
# 查看所有的分支和远程分支
$ git branch -a

# 创建一个新的分支
$ git branch myNewBranch

# 删除一个分支
$ git branch -d myBranch

# 重命名分支
# git branch -m <旧名称> <新名称>
$ git branch -m myBranchName myNewBranchName

# 编辑分支的介绍
$ git branch myBranchName --edit-description
```

检出

将当前工作空间更新到索引所标识的或者某一特定的工作空间

```
# 检出一个版本库，默认将更新到master分支
$ git checkout

# 检出到一个特定的分支
$ git checkout branchName

# 新建一个分支，并且切换过去，相当于"git branch <名字>; git checkout <名字>"
$ git checkout -b newBranch
```

clone

这个命令就是将一个版本库拷贝到另一个目录中，同时也将 分支都拷贝到新的版本库中。这样就可以在新的版本库中提交到远程分支

```
# clone learnxinyminutes-docs
$ git clone https://github.com/adambard/learnxinyminutes-docs.git
```

commit

将当前索引的更改保存为一个新的提交，这个提交包括用户做出的更改与信息

```
# 提交时附带提交信息
$ git commit -m "Added multiplyNumbers() function to HelloWorld.c"
```

diff

显示当前工作空间和提交的不同

```
# 显示工作目录和索引的不同
$ git diff

# 显示索引和最近一次提交的不同
$ git diff --cached

# 显示工作目录和最近一次提交的不同
$ git diff HEAD
```

grep

可以在版本库中快速查找

可选配置：

```
# 感谢Travis Jeffery提供的以下用法：
# 在搜索结果中显示行号
$ git config --global grep.lineNumber true

# 是搜索结果可读性更好
$ git config --global alias.g "grep --break --heading --line-number"
```

```
# 在所有的java中查找variableName
$ git grep 'variableName' -- '*.java'

# 搜索包含 "arrayListName" 和, "add" 或 "remove" 的所有行
$ git grep -e 'arrayListName' --and \( -e add -e remove \)
```

更多的例子可以查看：[Git Grep Ninja](#)

log

显示这个版本库的所有提交

```
# 显示所有提交
$ git log

# 显示某几条提交信息
$ git log -n 10

# 仅显示合并提交
$ git log --merges
```

merge

合并就是将外部的提交合并到自己的分支中

```
# 将其他分支合并到当前分支
$ git merge branchName

# 在合并时创建一个新的合并后的提交
$ git merge --no-ff branchName
```

mv

重命名或移动一个文件

```
# 重命名
$ git mv HelloWorld.c HelloNewWorld.c

# 移动
$ git mv HelloWorld.c ./new/path/HelloWorld.c

# 强制重命名或移动
# 这个文件已经存在，将要覆盖掉
$ git mv -f myFile existingFile
```

pull

从远端版本库合并到当前分支

```
# 从远端origin的master分支更新版本库
# git pull <远端> <分支>
$ git pull origin master
```

push

把远端的版本库更新

```
# 把本地的分支更新到远端origin的master分支上
# git push <远端> <分支>
# git push 相当于 git push origin master
$ git push origin master
```

rebase (谨慎使用)

将一个分支上所有的提交历史都应用到另一个分支上 不要在一个已经公开的远端分支上使用`rebase`.

```
# 将experimentBranch应用到master上面
# git rebase <basebranch> <topicbranch>
$ git rebase master experimentBranch
```

[更多阅读](#)

reset (谨慎使用)

将当前的头指针复位到一个特定的状态。这样可以使你撤销merge、pull、commits、add等 这是个很强大的命令，但是在使用时一定要清楚其所产生的后果

```
# 使 staging 区域恢复到上次提交时的状态，不改变现在的工作目录
$ git reset

# 使 staging 区域恢复到上次提交时的状态，覆盖现在的工作目录
$ git reset --hard

# 将当前分支恢复到某次提交，不改变现在的工作目录
# 在工作目录中所有的改变仍然存在
$ git reset 31f2bb1

# 将当前分支恢复到某次提交，覆盖现在的工作目录
# 并且删除所有未提交的改变和指定提交之后的所有提交
$ git reset --hard 31f2bb1
```

rm

和add相反，从工作空间中去掉某个文件

```
# 移除 HelloWorld.c
$ git rm HelloWorld.c

# 移除子目录中的文件
$ git rm /path/to/the/file/HelloWorld.c
```

更多阅读

- [tryGit](#) - 学习Git的有趣方式
- [git-scm](#) - 视频教程
- [git-scm](#) - 文档
- [Atlassian Git](#) - 教程与工作流程
- [SalesForce Cheat Sheet](#)
- [GitGuys](#)

language: Go lang: zh-cn filename: learn-go-cn.go contributors:

```
- ["Sonia Keys", "https://github.com/soniakeys"]
- ["pantaovay", "https://github.com/pantaovay"]
- ["lidashuang", "https://github.com/lidashuang"]
```

发明Go语言是出于更好地完成工作的需要。Go不是计算机科学的最新发展潮流，但它却提供了解决现实问题的最新最快的方法。

Go拥有命令式语言的静态类型，编译很快，执行也很快，同时加入了对于目前多核CPU的并发计算支持，也有相应的特性来实现大规模编程。

Go语言有非常棒的标准库，还有一个充满热情的社区。

```
// 单行注释
/* 多行
   注释 */

// 导入包的子句在每个源文件的开头。
// Main比较特殊，它用来声明可执行文件，而不是一个库。
package main

// Import语句声明了当前文件引用的包。
import (
    "fmt"          // Go语言标准库中的包
    "net/http"     // 一个web服务器包
    "strconv"      // 字符串转换
)

// 函数声明：Main是程序执行的入口。
// 不管你喜欢还是不喜欢，反正Go就用了花括号来包住函数体。
func main() {
    // 往标准输出打印一行。
    // 用包名fmt限制打印函数。
    fmt.Println("Hello world!")

    // 调用当前包的另一个函数。
    beyondHello()
}

// 函数可以在括号里加参数。
// 如果没有参数的话，也需要一个空括号。
func beyondHello() {
    var x int    // 变量声明，变量必须在使用之前声明。
    x = 3        // 变量赋值。
    // 可以用:=来偷懒，它自动把变量类型、声明和赋值都搞定了。
```

```

    y := 4
    sum, prod := learnMultiple(x, y) // 返回多个变量的函数
    fmt.Println("sum:", sum, "prod:", prod) // 简单输出
    learnTypes() // 少于y分钟，学的更多!
}

// 多变量和多返回值的函数
func learnMultiple(x, y int) (sum, prod int) {
    return x + y, x * y // 返回两个值
}

// 内置变量类型和关键词
func learnTypes() {
    // 短声明给你所想。
    s := "Learn Go!" // String类型

    s2 := `A "raw" string literal
can include line breaks.` // 同样是String类型

    // 非ascii字符。Go使用UTF-8编码。
    g := 'Σ' // rune类型，int32的别名，使用UTF-8编码

    f := 3.14195 // float64类型，IEEE-754 64位浮点数
    c := 3 + 4i // complex128类型，内部使用两个float64表示

    // Var变量可以直接初始化。
    var u uint = 7 // unsigned 无符号变量，但是实现依赖int型变量的长度
    var pi float32 = 22. / 7

    // 字符转换
    n := byte('\n') // byte是uint8的别名

    // 数组类型编译的时候大小固定。
    var a4 [4] int // 有4个int变量的数组，初始为0
    a3 := [...]int{3, 1, 5} // 有3个int变量的数组，同时进行了初始化

    // Slice 可以动态的增删。Array和Slice各有千秋，但是使用slice的地方更多些。
    s3 := []int{4, 5, 9} // 和a3相比，这里没有省略号
    s4 := make([]int, 4) // 分配一个有4个int型变量的slice，全部被初始化为0

    var d2 [][]float64 // 声明而已，什么都没有分配
    bs := []byte("a slice") // 类型转换的语法

    p, q := learnMemory() // 声明p,q为int型变量的指针
    fmt.Println(*p, *q) // * 取值

    // Map是动态可增长关联数组，和其他语言中的hash或者字典相似。
    m := map[string]int{"three": 3, "four": 4}
    m["one"] = 1

    // 在Go语言中未使用的变量在编译的时候会报错，而不是warning。

```

```

// 下划线 _ 可以使你“使用”一个变量，但是丢弃它的值。
_,_,_,_,_,_,_,_ = s2, g, f, u, pi, n, a3, s4, bs
// 输出变量
fmt.Println(s, c, a4, s3, d2, m)

learnFlowControl() // 回到流程控制
}

// Go全面支持垃圾回收。Go有指针，但是不支持指针运算。
// 你会因为空指针而犯错，但是不会因为增加指针而犯错。
func learnMemory() (p, q *int) {
    // 返回int型变量指针p和q
    p = new(int) // 内置函数new分配内存
    // 自动将分配的int赋值0，p不再是空的了。
    s := make([]int, 20) // 给20个int变量分配一块内存
    s[3] = 7 // 赋值
    r := -2 // 声明另一个局部变量
    return &s[3], &r // & 取地址
}

func expensiveComputation() int {
    return 1e6
}

func learnFlowControl() {
    // If需要花括号，括号就免了
    if true {
        fmt.Println("told ya")
    }
    // 用go fmt 命令可以帮你格式化代码，所以不用怕被人吐槽代码风格了，
    // 也不用容忍被人的代码风格。
    if false {
        // pout
    } else {
        // gloat
    }
    // 如果太多嵌套的if语句，推荐使用switch
    x := 1
    switch x {
    case 0:
    case 1:
        // 隐式调用break语句，匹配上一个即停止
    case 2:
        // 不会运行
    }
    // 和if一样，for也不用括号
    for x := 0; x < 3; x++ { // ++ 自增
        fmt.Println("iteration", x)
    }
    // x在这里还是1。为什么？

```

```

// for 是go里唯一的循环关键字，不过它有很多变种
for { // 死循环
    break // 骗你的
    continue // 不会运行的
}
// 和for一样，if中的:=先给y赋值，然后再和x作比较。
if y := expensiveComputation(); y > x {
    x = y
}
// 闭包函数
xBig := func() bool {
    return x > 100 // x是上面声明的变量引用
}
fmt.Println("xBig:", xBig()) // true （上面把y赋给x了）
x /= 1e5 // x变成10
fmt.Println("xBig:", xBig()) // 现在是false

// 当你需要goto的时候，你会爱死它的！
goto love
love:

    learnInterfaces() // 好东西来了！
}

// 定义Stringer为一个接口类型，有一个方法String
type Stringer interface {
    String() string
}

// 定义pair为一个结构体，有x和y两个int型变量。
type pair struct {
    x, y int
}

// 定义pair类型的方法，实现Stringer接口。
func (p pair) String() string { // p被叫做“接收器”
    // Sprintf是fmt包中的另一个公有函数。
    // 用 . 调用p中的元素。
    return fmt.Sprintf("(%d, %d)", p.x, p.y)
}

func learnInterfaces() {
    // 花括号用来定义结构体变量，:=在这里将一个结构体变量赋值给p。
    p := pair{3, 4}
    fmt.Println(p.String()) // 调用pair类型p的String方法
    var i Stringer // 声明i为Stringer接口类型
    i = p // 有效！因为p实现了Stringer接口（类似java中的塑型）
    // 调用i的String方法，输出和上面一样
    fmt.Println(i.String())

    // fmt包中的Println函数向对象要它们的string输出，实现了String方法就可以这样使用了。

```

```

// （类似java中的序列化）
fmt.Println(p) // 输出和上面一样，自动调用String函数。
fmt.Println(i) // 输出和上面一样。

learnErrorHandling()
}

func learnErrorHandling() {
    // ", ok"用来判断有没有正常工作
    m := map[int]string{3: "three", 4: "four"}
    if x, ok := m[1]; !ok { // ok 为false，因为m中没有1
        fmt.Println("no one there")
    } else {
        fmt.Print(x) // 如果x在map中的话，x就是那个值喽。
    }
    // 错误可不只是ok，它还可以给出关于问题的更多细节。
    if _, err := strconv.Atoi("non-int"); err != nil { // _ discards value
        // 输出"strconv.ParseInt: parsing "non-int": invalid syntax"
        fmt.Println(err)
    }
    // 待会再说接口吧。同时，
    learnConcurrency()
}

// c是channel类型，一个并发安全的通信对象。
func inc(i int, c chan int) {
    c <- i + 1 // <-把右边的发送到左边的channel。
}

// 我们将用inc函数来并发地增加一些数字。
func learnConcurrency() {
    // 用make来声明一个slice，make会分配和初始化slice，map和channel。
    c := make(chan int)
    // 用go关键字开始三个并发的goroutine，如果机器支持的话，还可能是并行执行。
    // 三个都被发送到同一个channel。
    go inc(0, c) // go is a statement that starts a new goroutine.
    go inc(10, c)
    go inc(-805, c)
    // 从channel中读取结果并打印。
    // 打印出什么东西是不可预知的。
    fmt.Println(<-c, <-c, <-c) // channel在右边的时候，<-是读操作。

    cs := make(chan string) // 操作string的channel
    cc := make(chan chan string) // 操作channel的channel
    go func() { c <- 84 }() // 开始一个goroutine来发送一个新的数字
    go func() { cs <- "wordy" }() // 发送给cs
    // Select类似于switch，但是每个case包括一个channel操作。
    // 它随机选择一个准备好通讯的case。
    select {
    case i := <-c: // 从channel接收的值可以赋给其他变量
        fmt.Println("it's a", i)
    }
}

```



```

    case <-cs: // 或者直接丢弃
        fmt.Println("it's a string")
    case <-cc: // 空的，还没作好通讯的准备
        fmt.Println("didn't happen.")
}
// 上面c或者cs的值被取到，其中一个goroutine结束，另外一个一直阻塞。

learnWebProgramming() // Go很适合web编程，我知道你也想学！
}

// http包中的一个简单的函数就可以开启web服务器。
func learnWebProgramming() {
    // ListenAndServe第一个参数指定了监听端口，第二个参数是一个接口，特定是http.Handler。
    err := http.ListenAndServe(":8080", pair{})
    fmt.Println(err) // 不要无视错误。
}

// 使pair实现http.Handler接口的ServeHTTP方法。
func (p pair) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // 使用http.ResponseWriter返回数据
    w.Write([]byte("You learned Go in Y minutes!"))
}

```

更进一步

Go的根源在[Go官方网站](#)。在那里你可以学习入门教程，通过浏览器交互式地学习，而且可以读到很多东西。

强烈推荐阅读语言定义部分，很简单而且很简洁！(as language definitions go these days.)

学习Go还要阅读[Go标准库的源代码](#)，全部文档化了，可读性非常好，可以学到go, go style和go idioms。在[文档](#)中点击函数名，源代码就出来了！

language: Groovy filename: learngroovy-cn.groovy contributors:

```
- ["Roberto Pérez Alcolea", "http://github.com/rpalcolea"]
```

translators:

```
- ["Todd Gao", "http://github.com/7c00"]
```

lang: zh-cn

Groovy - Java平台的动态语言。[了解更多](#)。

```
/*
  安装:

  1) 安装 GVM - http://gvmtool.net/
  2) 安装 Groovy: gvm install groovy
  3) 启动 groovy 控制台, 键入: groovyConsole

*/

// 双斜线开始的是单行注释
/*
像这样的是多行注释
*/

// Hello World
println "Hello world!"

/*
  变量:

  可以给变量赋值, 以便稍后使用
*/

def x = 1
println x

x = new java.util.Date()
println x

x = -3.1499392
println x

x = false
```

```
println x

x = "Groovy!"
println x

/*
    集合和映射
*/

//创建一个空的列表
def technologies = []

/** 往列表中增加一个元素 */

// 和Java一样
technologies.add("Grails")

// 左移添加，返回该列表
technologies << "Groovy"

// 增加多个元素
technologies.addAll(["Gradle","Griffon"])

/** 从列表中删除元素 */

// 和Java一样
technologies.remove("Griffon")

// 减号也行
technologies = technologies - 'Grails'

/** 遍历列表 */

// 遍历列表中的元素
technologies.each { println "Technology: $it"}
technologies.eachWithIndex { it, i -> println "$i: $it"}

/** 检查列表内容 */

//判断列表是否包含某元素，返回boolean
contained = technologies.contains( 'Groovy' )

// 或
contained = 'Groovy' in technologies

// 检查多个元素
technologies.containsAll(['Groovy','Grails'])

/** 列表排序 */

// 排序列表（修改原列表）
```

```

technologies.sort()

// 要想不修改原列表，可以这样：
sortedTechnologies = technologies.sort( false )

/**/ 列表操作 /**/

//替换列表元素
Collections.replaceAll(technologies, 'Gradle', 'gradle')

//打乱列表
Collections.shuffle(technologies, new Random())

//清空列表
technologies.clear()

//创建空的映射
def devMap = [:]

//增加值
devMap = ['name':'Roberto', 'framework':'Grails', 'language':'Groovy']
devMap.put('lastName','Perez')

//遍历映射元素
devMap.each { println "$it.key: $it.value" }
devMap.eachWithIndex { it, i -> println "$i: $it"}

//判断映射是否包含某键
assert devMap.containsKey('name')

//判断映射是否包含某值
assert devMap.containsValue('Roberto')

//取得映射所有的键
println devMap.keySet()

//取得映射所有的值
println devMap.values()

/*
    Groovy Beans

    GroovyBeans 是 JavaBeans，但使用了更简单的语法

    Groovy 被编译为字节码时，遵循下列规则。

    * 如果一个名字声明时带有访问修饰符（public, private, 或者 protected），
      则会生成一个字段（field）。

    * 名字声明时没有访问修饰符，则会生成一个带有public getter和setter的
      private字段，即属性（property）。

```

- * 如果一个属性声明为**final**，则会创建一个**final**的**private**字段，但不会生成**setter**。
- * 可以声明一个属性的同时定义自己的**getter**和**setter**。
- * 可以声明具有相同名字的属性 and 字段，该属性会使用该字段。
- * 如果要定义**private**或**protected**属性，必须提供声明为**private**或**protected**的**getter**和**setter**。
- * 如果使用显式或隐式的 **this**（例如 **this.foo**，或者 **foo**）访问类的在编译时定义的属性，**Groovy**会直接访问对应字段，而不是使用**getter**或者**setter**
- * 如果使用显式或隐式的 **foo** 访问一个不存在的属性，**Groovy**会通过元类（**meta class**）访问它，这可能导致运行时错误。

```
*/
```

```
class Foo {
    // 只读属性
    final String name = "Roberto"

    // 只读属性，有public getter和protected setter
    String language
    protected void setLanguage(String language) { this.language = language }

    // 动态类型属性
    def lastName
}
```

```
/*
    逻辑分支和循环
*/
```

```
//Groovy支持常见的if - else语法
def x = 3

if(x==1) {
    println "One"
} else if(x==2) {
    println "Two"
} else {
    println "X greater than Two"
}
```

```
//Groovy也支持三元运算符
def y = 10
def x = (y > 1) ? "worked" : "failed"
assert x == "worked"
```

```
//for循环
```

//使用区间（range）遍历

```
def x = 0
for (i in 0 .. 30) {
    x += i
}
```

//遍历列表

```
x = 0
for( i in [5,3,2,1] ) {
    x += i
}
```

//遍历数组

```
array = (0..20).toArray()
x = 0
for (i in array) {
    x += i
}
```

//遍历映射

```
def map = ['name':'Roberto', 'framework':'Grails', 'language':'Groovy']
x = 0
for ( e in map ) {
    x += e.value
}
```

/*

运算符

在Groovy中以下常用运算符支持重载：

<http://www.groovy-lang.org/operators.html#Operator-Overloading>

实用的groovy运算符

*/

//展开（spread）运算符：对聚合对象的所有元素施加操作

```
def technologies = ['Groovy','Grails','Gradle']
technologies*.toUpperCase() // 相当于 technologies.collect { it?.toUpperCase() }
```

//安全导航（safe navigation）运算符：用来避免NullPointerException

```
def user = User.get(1)
def username = user?.username
```

/*

闭包

Groovy闭包好比代码块或者方法指针，它是一段代码定义，可以以后执行。

更多信息见：<http://www.groovy-lang.org/closures.html>

*/

//例子：

```
def clos = { println "Hello World!" }
```

```

println "Executing the Closure:"
clos()

//传参数给闭包
def sum = { a, b -> println a+b }
sum(2,4)

//闭包可以引用参数列表以外的变量
def x = 5
def multiplyBy = { num -> num * x }
println multiplyBy(10)

// 只有一个参数的闭包可以省略参数的定义
def clos = { print it }
clos( "hi" )

/*
    Groovy可以记忆闭包结果 [1][2][3]
*/
def cl = {a, b ->
    sleep(3000) // 模拟费时操作
    a + b
}

mem = cl.memoize()

def callClosure(a, b) {
    def start = System.currentTimeMillis()
    mem(a, b)
    println "Inputs(a = $a, b = $b) - took ${System.currentTimeMillis() - start} msecs."
}

callClosure(1, 2)
callClosure(1, 2)
callClosure(2, 3)
callClosure(2, 3)
callClosure(3, 4)
callClosure(3, 4)
callClosure(1, 2)
callClosure(2, 3)
callClosure(3, 4)

/*
    Expando

    Expando类是一种动态bean类，可以给它的实例添加属性和添加闭包作为方法

    http://mrhaki.blogspot.mx/2009/10/groovy-goodness-expando-as-dynamic-bean.html
*/
def user = new Expando(name:"Roberto")

```

```

assert 'Roberto' == user.name

user.lastName = 'Pérez'
assert 'Pérez' == user.lastName

user.showInfo = { out ->
    out << "Name: $name"
    out << ", Last name: $lastName"
}

def sw = new StringWriter()
println user.showInfo(sw)

/*
    元编程(MOP)
*/

//使用ExpandoMetaClass增加行为
String.metaClass.testAdd = {
    println "we added this"
}

String x = "test"
x?.testAdd()

//拦截方法调用
class Test implements GroovyInterceptable {
    def sum(Integer x, Integer y) { x + y }

    def invokeMethod(String name, args) {
        System.out.println "Invoke method $name with args: $args"
    }
}

def test = new Test()
test?.sum(2,3)
test?.multiply(2,3)

//Groovy支持propertyMissing, 来处理属性解析尝试
class Foo {
    def propertyMissing(String name) { name }
}
def f = new Foo()

assertEquals "boo", f.boo

/*
    类型检查和静态编译
    Groovy天生是并将永远是一门动态语言, 但也支持类型检查和静态编译

```



```
    更多: http://www.infoq.com/articles/new-groovy-20
*/
//类型检查
import groovy.transform.TypeChecked

void testMethod() {}

@TypeChecked
void test() {
    testMeethod()

    def name = "Roberto"

    println naameee
}

//另一例子
import groovy.transform.TypeChecked

@TypeChecked
Integer test() {
    Integer num = "1"

    Integer[] numbers = [1,2,3,4]

    Date date = numbers[1]

    return "Test"
}

//静态编译例子
import groovy.transform.CompileStatic

@CompileStatic
int sum(int x, int y) {
    x + y
}

assert sum(2,5) == 7
```

进阶资源

[Groovy文档](#)

[Groovy web console](#)

加入[Groovy用户组](#)

图书

- [Groovy Goodness] (<https://leanpub.com/groovy-goodness-notebook>)
- [Groovy in Action] (<http://manning.com/koenig2/>)
- [Programming Groovy 2: Dynamic Productivity for the Java Developer] (<http://shop.oreilly.com/product/9781937785307.do>)

[1] <http://roshandawrani.wordpress.com/2010/10/18/groovy-new-feature-closures-can-now-memorize-their-results/> [2] <http://www.solutionsiq.com/resources/agileiq-blog/bid/72880/Programming-with-Groovy-Trampoline-and-Memoize> [3] <http://mrhaki.blogspot.mx/2011/05/groovy-goodness-cache-closure-results.html>

language: Haskell filename: learn-haskell-zh.hs contributors:

```
- ["Adit Bhargava", "http://adit.io"]
```

translators:

```
- ["Peiyong Lin", ""]  
- ["chad luo", "http://yuki.rocks"]
```

lang: zh-cn

Haskell 是一门实用的函数式编程语言，因其 **Monads** 与类型系统而闻名。而我使用它则是因为它异常优雅。用 **Haskell** 编程令我感到非常快乐。

```
-- 单行注释以两个减号开头  
{- 多行注释像这样  
    被一个闭合的块包围  
-}  
  
-----  
-- 1. 简单的数据类型和操作符  
-----  
  
-- 数字  
3 -- 3  
-- 数学计算  
1 + 1 -- 2  
8 - 1 -- 7  
10 * 2 -- 20  
35 / 5 -- 7.0  
  
-- 默认除法不是整除  
35 / 4 -- 8.75  
  
-- 整除  
35 `div` 4 -- 8  
  
-- 布尔值  
True  
False  
  
-- 布尔操作  
not True -- False  
not False -- True  
1 == 1 -- True
```

```
1 /= 1 -- False
1 < 10 -- True
```

```
-- 在上面的例子中，`not` 是一个接受一个参数的函数。
-- Haskell 不需要括号来调用函数，所有的参数都只是在函数名之后列出来
-- 因此，通常的函数调用模式是：
--   func arg1 arg2 arg3...
-- 你可以查看函数部分了解如何自行编写。
```

```
-- 字符串和字符
"This is a string." -- 字符串
'a' -- 字符
'对于字符串你不能使用单引号。' -- 错误！
```

```
-- 连接字符串
"Hello " ++ "world!" -- "Hello world!"
```

```
-- 一个字符串是一系列字符
['H', 'e', 'l', 'l', 'o'] -- "Hello"
"This is a string" !! 0 -- 'T'
```

```
-----
-- 列表和元组
-----
```

```
-- 一个列表中的每一个元素都必须是相同的类型。
-- 下面两个列表等价
[1, 2, 3, 4, 5]
[1..5]
```

```
-- 区间也可以这样
['A'..'F'] -- "ABCDEF"
```

```
-- 你可以在区间中指定步进
[0,2..10] -- [0, 2, 4, 6, 8, 10]
[5..1] -- 这样不行，因为 Haskell 默认递增
[5,4..1] -- [5, 4, 3, 2, 1]
```

```
-- 列表下标
[0..] !! 5 -- 5
```

```
-- 在 Haskell 你可以使用无限列表
[1..] -- 一个含有所有自然数的列表
```

```
-- 无限列表的原理是，Haskell 有“惰性求值”。
-- 这意味着 Haskell 只在需要时才会计算。
-- 所以当你获取列表的第 1000 项元素时，Haskell 会返回给你：
[1..] !! 999 -- 1000
-- Haskell 计算了列表中第 1 至 1000 项元素，但这个无限列表中剩下的元素还不存在。
-- Haskell 只有在需要时才会计算它们。
```

```

-- 连接两个列表
[1..5] ++ [6..10]

-- 往列表头增加元素
0:[1..5] -- [0, 1, 2, 3, 4, 5]

-- 其它列表操作
head [1..5] -- 1
tail [1..5] -- [2, 3, 4, 5]
init [1..5] -- [1, 2, 3, 4]
last [1..5] -- 5

-- 列表推导 (list comprehension)
[x*2 | x <- [1..5]] -- [2, 4, 6, 8, 10]

-- 附带条件
[x*2 | x <- [1..5], x*2 > 4] -- [6, 8, 10]

-- 元组中的每一个元素可以是不同类型，但是一个元组的长度是固定的
-- 一个元组
("haskell", 1)

-- 获取元组中的元素（例如，一个含有 2 个元素的元组）
fst ("haskell", 1) -- "haskell"
snd ("haskell", 1) -- 1

-----

-- 3. 函数
-----

-- 一个接受两个变量的简单函数
add a b = a + b

-- 注意，如果你使用 ghci (Haskell 解释器)，你需要使用 `let`，也就是
-- let add a b = a + b

-- 调用函数
add 1 2 -- 3

-- 你也可以使用反引号中置函数名：
1 `add` 2 -- 3

-- 你也可以定义不带字母的函数名，这样你可以定义自己的操作符。
-- 这里有一个做整除的操作符
(//) a b = a `div` b
35 // 4 -- 8

-- Guard: 一个在函数中做条件判断的简单方法
fib x
  | x < 2 = x

```

```

| otherwise = fib (x - 1) + fib (x - 2)

-- 模式匹配与 Guard 类似。
-- 这里给出了三个不同的 fib 定义。
-- Haskell 会自动调用第一个符合参数模式的声明
fib 1 = 1
fib 2 = 2
fib x = fib (x - 1) + fib (x - 2)

-- 元组的模式匹配
foo (x, y) = (x + 1, y + 2)

-- 列表的模式匹配
-- 这里 `x` 是列表中第一个元素，`xs` 是列表剩余的部分。
-- 我们可以实现自己的 map 函数：
myMap func [] = []
myMap func (x:xs) = func x:(myMap func xs)

-- 匿名函数带有一个反斜杠，后面跟着所有的参数
myMap (\x -> x + 2) [1..5] -- [3, 4, 5, 6, 7]

-- 在 fold（在一些语言称为 `inject`）中使用匿名函数
-- foldl1 意味着左折叠（fold left），并且使用列表中第一个值作为累加器的初始值。
foldl1 (\acc x -> acc + x) [1..5] -- 15

-----

-- 4. 其它函数
-----

-- 部分调用
-- 如果你调用函数时没有给出所有参数，它就被“部分调用”。
-- 它将返回一个接受余下参数的函数。
add a b = a + b
foo = add 10 -- foo 现在是一个接受一个数并对其加 10 的函数
foo 5 -- 15

-- 另一种等价写法
foo = (+10)
foo 5 -- 15

-- 函数表合
-- (.) 函数把其它函数链接到一起。
-- 例如，这里 foo 是一个接受一个值的函数。
-- 它对接受的值加 10，并对结果乘以 5，之后返回最后的值。
foo = (*5) . (+10)

-- (5 + 10) * 5 = 75
foo 5 -- 75

-- 修正优先级
-- Haskell 有另外一个函数 `$` 可以改变优先级。

```

```

-- `\$` 使得 Haskell 先计算其右边的部分，然后调用左边的部分。
-- 你可以使用 `` 来移除多余的括号。

-- 修改前
(even (fib 7)) -- False

-- 修改后
even . fib $ 7 -- False

-- 等价地
even $ fib 7 -- False

-----

-- 5. 类型声明
-----

-- Haskell 有一个非常强大的类型系统，一切都有一个类型声明。

-- 一些基本的类型：
5 :: Integer
"hello" :: String
True :: Bool

-- 函数也有类型
-- `not` 接受一个布尔型返回一个布尔型
-- not :: Bool -> Bool

-- 这是接受两个参数的函数
-- add :: Integer -> Integer -> Integer

-- 当你定义一个值，声明其类型是一个好做法
double :: Integer -> Integer
double x = x * 2

-----

-- 6. 控制流和 If 语句
-----

-- if 语句：
haskell = if 1 == 1 then "awesome" else "awful" -- haskell = "awesome"

-- if 语句也可以有多行，注意缩进：
haskell = if 1 == 1
           then "awesome"
           else "awful"

-- case 语句
-- 解析命令行参数：
case args of
  "help" -> printHelp
  "start" -> startProgram

```

```

_ -> putStrLn "bad args"

-- Haskell 没有循环，它使用递归
-- map 对一个列表中的每一个元素调用一个函数
map (*2) [1..5] -- [2, 4, 6, 8, 10]

-- 你可以使用 map 来编写 for 函数
for array func = map func array

-- 调用
for [0..5] $ \i -> show i

-- 我们也可以像这样写
for [0..5] show

-- 你可以使用 foldl 或者 foldr 来分解列表
-- foldl <fn> <initial value> <list>
foldl (\x y -> 2*x + y) 4 [1,2,3] -- 43

-- 等价于
(2 * (2 * (2 * 4 + 1) + 2) + 3)

-- foldl 从左开始, foldr 从右
foldr (\x y -> 2*x + y) 4 [1,2,3] -- 16

-- 现在它等价于
(2 * 3 + (2 * 2 + (2 * 1 + 4)))

-----

-- 7. 数据类型
-----

-- 在 Haskell 中声明你自己的数据类型:
data Color = Red | Blue | Green

-- 现在你可以在函数中使用它:
say :: Color -> String
say Red = "You are Red!"
say Blue = "You are Blue!"
say Green = "You are Green!"

-- 你的数据类型也可以有参数:
data Maybe a = Nothing | Just a

-- 这些都是 Maybe 类型:
Just "hello" -- `Maybe String` 类型
Just 1       -- `Maybe Int` 类型
Nothing      -- 对任意 `a` 为 `Maybe a` 类型

-----

-- 8. Haskell IO

```



```

-----

-- 虽然不解释 Monads 就无法完全解释 IO，但大致了解并不难。

-- 当执行一个 Haskell 程序时，函数 `main` 就被调用。
-- 它必须返回一个类型 `IO ()` 的值。例如：
main :: IO ()
main = putStrLn $ "Hello, sky! " ++ (say Blue)
-- putStrLn 的类型是 String -> IO ()

-- 如果你的程序输入 String 返回 String，那样编写 IO 是最简单的。
-- 函数
--   interact :: (String -> String) -> IO ()
-- 输入一些文本，对其调用一个函数，并打印输出。

countLines :: String -> String
countLines = show . length . lines

main' = interact countLines

-- 你可以认为一个 `IO ()` 类型的值是表示计算机做的一系列操作，类似命令式语言。
-- 我们可以使用 `do` 声明来把动作连接到一起。
-- 举个例子
sayHello :: IO ()
sayHello = do
    putStrLn "What is your name?"
    name <- getLine -- 这里接受一行输入并绑定至 "name"
    putStrLn $ "Hello, " ++ name

-- 练习：编写只读取一行输入的 `interact`

-- 然而，`sayHello` 中的代码将不会被执行。唯一被执行的动作是 `main` 的值。
-- 为了运行 `sayHello`，注释上面 `main` 的定义，替换为：
--   main = sayHello

-- 让我们来更进一步理解刚才所使用的函数 `getLine` 是怎样工作的。它的类型是：
--   getLine :: IO String
-- 你可以认为一个 `IO a` 类型的值代表了一个运行时会生成一个 `a` 类型值的程序。
-- （可能伴随其它行为）
-- 我们可以通过 `<-` 保存和重用这个值。
-- 我们也可以实现自己的 `IO String` 类型函数：
action :: IO String
action = do
    putStrLn "This is a line. Duh"
    input1 <- getLine
    input2 <- getLine
    -- `do` 语句的类型是它的最后一行
    -- `return` 不是关键字，只是一个普通函数
    return (input1 ++ "\n" ++ input2) -- return :: String -> IO String

-- 我们可以像调用 `getLine` 一样调用它

```

```

main'' = do
    putStrLn "I will echo two lines!"
    result <- action
    putStrLn result
    putStrLn "This was all, folks!"

-- `IO` 类型是一个 "Monad" 的例子。
-- Haskell 通过使用 Monad 使得其本身为纯函数式语言。
-- 任何与外界交互的函数（即 IO）都在它的类型声明中标记为 `IO`。
-- 这告诉我们什么样的函数是“纯洁的”（不与外界交互，不修改状态），
-- 什么样的函数不是“纯洁的”。
-- 这个功能非常强大，因为纯函数并发非常容易，由此在 Haskell 中做并发非常容易。

-----

-- 9. Haskell REPL
-----

-- 键入 `ghci` 开始 REPL。
-- 现在你可以键入 Haskell 代码。
-- 任何新值都需要通过 `let` 来创建
let foo = 5

-- 你可以通过命令 `:t` 查看任何值的类型
>:t foo
foo :: Integer

-- 你也可以运行任何 `IO ()` 类型的动作
> sayHello
What is your name?
Friend!
Hello, Friend!

```

Haskell 还有许多内容，包括类型类 (typeclasses) 与 Monads。这些都是令 Haskell 编程非常有趣的好东西。我们最后给出 Haskell 的一个例子，一个快速排序的实现：

```

qsort [] = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
    where lesser = filter (< p) xs
          greater = filter (>= p) xs

```

安装 Haskell 很简单。你可以[从这里](#)获得。

你可以从优秀的 [Learn you a Haskell](#) 或者 [Real World Haskell](#) 找到更平缓的入门介绍。

name: java category: language language: java lang: zh-cn filename: LearnJava-zh.java contributors:

```
- ["Jake Prather", "http://github.com/JakeHP"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net"]
```

Java是一个通用的程序语言, 包含并发, 基于类的面向对象等特性 [阅读更多](#)

```
// 单行注释
/*
多行注释
*/
/**
JavaDoc (Java文档) 注释是这样的。可以用来描述类和类的属性。
*/

// 导入 java.util中的 ArrayList 类
import java.util.ArrayList;
// 导入 java.security 包中的所有类
import java.security.*;

// 每个 .java 文件都包含一个public类, 这个类的名字必须和这个文件名一致。
public class LearnJava {

    // 每个程序都需要有一个main函数作为入口
    public static void main (String[] args) {

        // 使用 System.out.println 来输出到标准输出
        System.out.println("Hello World!");
        System.out.println(
            "Integer: " + 10 +
            " Double: " + 3.14 +
            " Boolean: " + true);

        // 如果要在输出后不自动换行, 可以使用System.out.print方法。
        System.out.print("Hello ");
        System.out.print("World");

        //////////////////////////////////////
        // 类型与变量
        //////////////////////////////////////
```

```
// 用 <type> <name> 来声明变量
// 字节类型 - 8位补码表示
// (-128 <= 字节 <= 127)
byte fooByte = 100;

// 短整型 - 16位补码表示
// (-32,768 <= 短整型 <= 32,767)
short fooShort = 10000;

// 整型 - 32位补码表示
// (-2,147,483,648 <= 整型 <= 2,147,483,647)
int fooInt = 1;

// 长整型 - 64位补码表示
// (-9,223,372,036,854,775,808 <= 长整型 <= 9,223,372,036,854,775,807)
long fooLong = 100000L;
// L可以用来表示一个数字是长整型的。
// 其他的数字都默认为整型。

// 注意: Java中没有无符号类型

// 浮点型 - 即 IEEE 754 规定的32位单精度浮点类型
float fooFloat = 234.5f;
// f用来表示一个数字是浮点型的。
// 否则会被默认当做是双精度浮点型。

// 双精度浮点型 - 即 IEEE 754 规定的64位双精度浮点类型
double fooDouble = 123.4;

// 布尔类型 - true 与 false
boolean fooBoolean = true;
boolean barBoolean = false;

// 字符类型 - 16位 Unicode编码字符
char fooChar = 'A';

// 用 final 可以使一个常量不可更改
final int HOURS_I_WORK_PER_WEEK = 9001;

// 字符串
String fooString = "My String Is Here!";

// \n 代表一个新的换行
String barString = "Printing on a new line?\nNo Problem!";
// \t 代表一个新的制表符
String bazString = "Do you want to add a tab?\tNo Problem!";
System.out.println(fooString);
System.out.println(barString);
System.out.println(bazString);

// 数组
```

```

// 数组在声明时大小必须已经确定
// 数组的声明格式:
//<数据类型> [] <变量名> = new <数据类型>[<数组大小>];
int [] intArray = new int[10];
String [] stringArray = new String[1];
boolean [] booleanArray = new boolean[100];

// 声明并初始化数组也可以这样:
int [] y = {9000, 1000, 1337};

// 随机访问数组中的元素
System.out.println("intArray @ 0: " + intArray[0]);

// 数组下标从0开始并且可以被更改
intArray[1] = 1;
System.out.println("intArray @ 1: " + intArray[1]); // => 1

// 其他数据类型
// ArrayLists - 类似于数组, 但是功能更多, 并且大小也可以改变
// LinkedLists
// Maps
// HashMaps

////////////////////////////////////
// 操作符
////////////////////////////////////
System.out.println("\n->Operators");

int i1 = 1, i2 = 2; // 多重声明可以简化

// 算数运算
System.out.println("1+2 = " + (i1 + i2)); // => 3
System.out.println("2-1 = " + (i2 - i1)); // => 1
System.out.println("2*1 = " + (i2 * i1)); // => 2
System.out.println("1/2 = " + (i1 / i2)); // => 0 (0.5 truncated down)

// 取余
System.out.println("11%3 = " + (11 % 3)); // => 2

// 比较操作符
System.out.println("3 == 2? " + (3 == 2)); // => false
System.out.println("3 != 2? " + (3 != 2)); // => true
System.out.println("3 > 2? " + (3 > 2)); // => true
System.out.println("3 < 2? " + (3 < 2)); // => false
System.out.println("2 <= 2? " + (2 <= 2)); // => true
System.out.println("2 >= 2? " + (2 >= 2)); // => true

// 位运算操作符
/*
~      取反, 求反码
<<    带符号左移

```

```

>>      带符号右移
>>>     无符号右移
&        和
^        异或
|        相容或
*/

// 自增
int i = 0;
System.out.println("\n->Inc/Dec-rementation");
// ++ 和 -- 操作符使变量加或减1。放在变量前面或者后面的区别是整个表达
// 式的返回值。操作符在前面时，先加减，后取值。操作符在后面时，先取值
// 后加减。
System.out.println(i++); // 后自增 i = 1, 输出0
System.out.println(++i); // 前自增 i = 2, 输出2
System.out.println(i--); // 后自减 i = 1, 输出2
System.out.println(--i); // 前自减 i = 0, 输出0

////////////////////////////////////
// 控制结构
////////////////////////////////////
System.out.println("\n->Control Structures");

// If语句和C的类似
int j = 10;
if (j == 10){
    System.out.println("I get printed");
} else if (j > 10) {
    System.out.println("I don't");
} else {
    System.out.println("I also don't");
}

// While循环
int fooWhile = 0;
while(fooWhile < 100)
{
    //System.out.println(fooWhile);
    //增加计数器
    //遍历99次, fooWhile 0->99
    fooWhile++;
}
System.out.println("fooWhile Value: " + fooWhile);

// Do While循环
int fooDoWhile = 0;
do
{
    //System.out.println(fooDoWhile);
    //增加计数器
    //遍历99次, fooDoWhile 0->99

```

```

        fooDoWhile++;
    }while(fooDoWhile < 100);
    System.out.println("fooDoWhile Value: " + fooDoWhile);

    // For 循环
    int fooFor;
    //for 循环结构 => for(<起始语句>; <循环进行的条件>; <步长>)
    for(fooFor=0; fooFor<10; fooFor++){
        //System.out.println(fooFor);
        //遍历 10 次, fooFor 0->9
    }
    System.out.println("fooFor Value: " + fooFor);

    // Switch Case 语句
    // switch可以用来处理 byte, short, char, 和 int 数据类型
    // 也可以用来处理枚举类型, 字符串类, 和原始数据类型的包装类:
    // Character, Byte, Short, 和 Integer
    int month = 3;
    String monthString;
    switch (month){
        case 1:
            monthString = "January";
            break;
        case 2:
            monthString = "February";
            break;
        case 3:
            monthString = "March";
            break;
        default:
            monthString = "Some other month";
            break;
    }
    System.out.println("Switch Case Result: " + monthString);

    //////////////////////////////////////
    // 类型转换
    //////////////////////////////////////

    // 数据转换

    // 将字符串转换为整型
    Integer.parseInt("123");//返回整数123

    // 将整型转换为字符串
    Integer.toString(123);//返回字符串"123"

    // 其他的数据也可以进行互相转换:
    // Double
    // Long

```

```

// String

// 类型转换
// 你也可以对java对象进行类型转换，但其中会牵扯到很多概念
// 在这里可以查看更详细的信息：
// http://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

////////////////////////////////////
// 类与函数
////////////////////////////////////

System.out.println("\n->Classes & Functions");

// (Bicycle类定义如下)

// 用new来实例化一个类
Bicycle trek = new Bicycle();

// 调用对象的方法
trek.speedUp(3); // 需用getter和setter方法
trek.setCadence(100);

// toString 可以把对象转换为字符串
System.out.println("trek info: " + trek.toString());

} // main 方法结束
} // LearnJava 类结束

// 你也可以把其他的非public类放入到.java文件中

// 类定义的语法：
// <public/private/protected> class <类名>{
//     //成员变量，构造函数，函数
//     //Java中函数被称作方法
// }

class Bicycle {

    // Bicycle 类的成员变量和方法
    public int cadence; // Public: 任意位置均可访问
    private int speed; // Private: 只在同类中可以访问
    protected int gear; // Protected: 可以在同类与子类中可以访问
    String name; // default: 可以在包内中可以访问

    // 构造函数是初始化一个对象的方式
    // 以下是一个默认构造函数
    public Bicycle() {
        gear = 1;
    }
}

```



```

        cadence = 50;
        speed = 5;
        name = "Bontrager";
    }

    // 一下是一个含有参数的构造函数
    public Bicycle(int startCadence, int startSpeed, int startGear, String name) {
        this.gear = startGear;
        this.cadence = startCadence;
        this.speed = startSpeed;
        this.name = name;
    }

    // 函数语法:
    // <public/private/protected> <返回值类型> <函数名称>(<参数列表>)

    // Java类中经常会用getter和setter来对成员变量进行操作

    // 方法声明的语法:
    // <作用域> <返回值类型> <方法名>(<参数列表>)
    public int getCadence() {
        return cadence;
    }

    // void返回值函数没有返回值
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

    public void slowDown(int decrement) {
        speed -= decrement;
    }

    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    // 返回对象属性的方法
    @Override

```

```

        public String toString() {
            return "gear: " + gear +
                " cadence: " + cadence +
                " speed: " + speed +
                " name: " + name;
        }
    } // Bicycle 类结束

    // PennyFarthing 是 Bicycle 的子类
    class PennyFarthing extends Bicycle {
        // (Penny Farthings 是前轮很大的 Bicycle, 并且没有齿轮)

        public PennyFarthing(int startCadence, int startSpeed){
            // 通过super调用父类的构造函数
            super(startCadence, startSpeed, 0, "PennyFarthing");
        }

        // 你可以用@注释来表示需要重载的方法
        // 了解更多的注释使用方法, 可以访问下面的地址:
        // http://docs.oracle.com/javase/tutorial/java/annotations/
        @Override
        public void setGear(int gear) {
            gear = 0;
        }
    }
}

```

更多阅读

下面的链接只是为了便于大家理解这些主题而给出的, 对于具体的例子请大家自行Google

其他主题:

- [Java 官方教程](#)
- [Java 访问修饰符](#)
- 面向对象程序设计概念:
 - [继承](#)
 - [多态](#)
 - [抽象](#)
- [异常](#)
- [接口](#)
- [泛型](#)
- [Java代码规范](#)

language: javascript category: language name: javascript filename: javascript-zh.js contributors:

```
- ["Adam Brenecki", "http://adam.brenecki.id.au"]
- ["Ariel Krakowski", "http://www.learneroo.com"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net"]
- ["Guodong Qu", "https://github.com/jasonqu"]
```

lang: zh-cn

JavaScript于1995年由网景公司的Brendan Eich发明。最初发明的目的是作为一个简单的网站脚本语言，来作为复杂网站应用java的补充。但由于它与网页结合度很高并且由浏览器内置支持，所以javascript变得比java在前端更为流行了。

不过 JavaScript 可不仅仅只用于浏览器： Node.js，一个基于Google Chrome V8引擎的独立运行时环境，也越来越流行。

很欢迎来自您的反馈，您可以通过下列方式联系到我： [@adambrenecki](#), 或者 adam@brenecki.id.au.

```
// 注释方式和C很像，这是单行注释
/* 这是多行
   注释 */

// 语句可以以分号结束
doStuff();

// ... 但是分号也可以省略，每当遇到一个新行时，分号会自动插入（除了一些特殊情况）。
doStuff()

// 因为这些特殊情况会导致意外的结果，所以我们在这里保留分号。

////////////////////////////////////
// 1. 数字、字符串与操作符

// Javascript 只有一种数字类型(即 64位 IEEE 754 双精度浮点 double)。
// double 有 52 位表示尾数，足以精确存储大到  $9 \times 10^{15}$  的整数。
3; // = 3
1.5; // = 1.5

// 所有基本的算数运算都如你预期。
1 + 1; // = 2
0.1 + 0.2; // = 0.30000000000000004
8 - 1; // = 7
10 * 2; // = 20
```

```
35 / 5; // = 7

// 包括无法整除的除法。
5 / 2; // = 2.5

// 位运算也和其他语言一样；当你对浮点数进行位运算时，
// 浮点数会转换为*至多* 32 位的无符号整数。
1 << 2; // = 4

// 括号可以决定优先级。
(1 + 3) * 2; // = 8

// 有三种非数字的数字类型
Infinity; // 1/0 的结果
-Infinity; // -1/0 的结果
NaN; // 0/0 的结果

// 也有布尔值。
true;
false;

// 可以通过单引号或双引号来构造字符串。
'abc';
"Hello, world";

// 用! 来取非
!true; // = false
!false; // = true

// 相等 ===
1 === 1; // = true
2 === 1; // = false

// 不等 !=
1 !== 1; // = false
2 !== 1; // = true

// 更多的比较操作符
1 < 10; // = true
1 > 10; // = false
2 <= 2; // = true
2 >= 2; // = true

// 字符串用+连接
"Hello " + "world!"; // = "Hello world!"

// 字符串也可以用 < 、> 来比较
"a" < "b"; // = true

// 使用“==”比较时会进行类型转换...
"5" == 5; // = true
```

```

null == undefined; // = true

// ...除非你是用 ===
"5" === 5; // = false
null === undefined; // = false

// ...但会导致奇怪的行为
13 + !0; // 14
"13" + !0; // '13true'

// 你可以用`charAt`来得到字符串中的字符
"This is a string".charAt(0); // = 'T'

// ...或使用`substring`来获取更大的部分。
"Hello world".substring(0, 5); // = "Hello"

// `length` 是一个属性，所以不要使用 ()。
"Hello".length; // = 5

// 还有两个特殊的值：`null`和`undefined`
null; // 用来表示刻意设置的空值
undefined; // 用来表示还没有设置的值(尽管`undefined`自身实际是一个值)

// false, null, undefined, NaN, 0 和 "" 都是假的；其他的都视作逻辑真
// 注意 0 是逻辑假而 "0"是逻辑真，尽管 0 == "0"。

////////////////////////////////////
// 2. 变量、数组和对象

// 变量需要用`var`关键字声明。Javascript是动态类型语言，
// 所以你无需指定类型。 赋值需要用 `=`
var someVar = 5;

// 如果你在声明时没有加var关键字，你也不会得到错误...
someOtherVar = 10;

// ...但是此时这个变量就会在全局作用域被创建，而非你定义的当前作用域

// 没有被赋值的变量都会被设置为undefined
var someThirdVar; // = undefined

// 对变量进行数学运算有一些简写法：
someVar += 5; // 等价于 someVar = someVar + 5; someVar 现在是 10
someVar *= 10; // 现在 someVar 是 100

// 自增和自减也有简写
someVar++; // someVar 是 101
someVar--; // 回到 100

// 数组是任意类型组成的有序列表
var myArray = ["Hello", 45, true];

```

```

// 数组的元素可以用方括号下标来访问。
// 数组的索引从0开始。
myArray[1]; // = 45

// 数组是可变的，并拥有变量 length。
myArray.push("World");
myArray.length; // = 4

// 在指定下标添加/修改
myArray[3] = "Hello";

// javascript中的对象相当于其他语言中的“字典”或“映射”：是键-值对的无序集合。
var myObj = {key1: "Hello", key2: "World"};

// 键是字符串，但如果键本身是合法的js标识符，则引号并非必须的。
// 值可以是任意类型。
var myObj = {myKey: "myValue", "my other key": 4};

// 对象属性的访问可以通过下标
myObj["my other key"]; // = 4

// ... 或者也可以用 . ，如果属性是合法的标识符
myObj.myKey; // = "myValue"

// 对象是可变的；值也可以被更改或增加新的键
myObj.myThirdKey = true;

// 如果你想要获取一个还没有被定义的值，那么会返回undefined
myObj.myFourthKey; // = undefined

////////////////////////////////////////
// 3. 逻辑与控制结构

// 本节介绍的语法与Java的语法几乎完全相同

// `if` 语句和其他语言中一样。
var count = 1;
if (count == 3){
    // count 是 3 时执行
} else if (count == 4){
    // count 是 4 时执行
} else {
    // 其他情况下执行
}

// while循环
while (true) {
    // 无限循环
}

```

```

// Do-while 和 While 循环很像，但前者会至少执行一次
var input;
do {
    input = getInput();
} while (!isValid(input))

// `for` 循环和C、Java中的一样：
// 初始化；继续执行的条件；迭代。
for (var i = 0; i < 5; i++){
    // 遍历5次
}

// && 是逻辑与，|| 是逻辑或
if (house.size == "big" && house.colour == "blue"){
    house.contains = "bear";
}
if (colour == "red" || colour == "blue"){
    // colour是red或者blue时执行
}

// && 和 || 是“短路”语句，它在设定初始化值时特别有用
var name = otherName || "default";

// `switch` 语句使用`===`检查相等性。
// 在每一个case结束时使用 'break'
// 否则其后的case语句也将被执行。
grade = 'B';
switch (grade) {
    case 'A':
        console.log("Great job");
        break;
    case 'B':
        console.log("OK job");
        break;
    case 'C':
        console.log("You can do better");
        break;
    default:
        console.log("Oy vey");
        break;
}

////////////////////////////////////
// 4. 函数、作用域、闭包

// JavaScript 函数由`function`关键字定义
function myFunction(thing){
    return thing.toUpperCase();
}
myFunction("foo"); // = "FOO"

```

```

// 注意被返回的值必须开始于`return`关键字的那一行，
// 否则由于自动的分号补齐，你将返回`undefined`。
// 在使用Allman风格的时候要注意。
function myFunction()
{
    return // <- 分号自动插在这里
    {
        thisIsAn: 'object literal'
    }
}
myFunction(); // = undefined

// javascript中函数是一等对象，所以函数也能够赋给一个变量，
// 并且被作为参数传递 — 比如一个事件处理函数：
function myFunction(){
    // 这段代码将在5秒钟后被调用
}
setTimeout(myFunction, 5000);
// 注意：setTimeout不是js语言的一部分，而是由浏览器和Node.js提供的。

// 函数对象甚至不需要声明名称 — 你可以直接把一个函数定义写到另一个函数的参数中
setTimeout(function(){
    // 这段代码将在5秒钟后被调用
}, 5000);

// JavaScript 有函数作用域；函数有其自己的作用域而其他的代码块则没有。
if (true){
    var i = 5;
}
i; // = 5 - 并非我们在其他语言中所期望得到的undefined

// 这就导致了人们经常使用的“立即执行匿名函数”的模式，
// 这样可以避免一些临时变量扩散到全局作用域去。
(function(){
    var temporary = 5;
    // 我们可以访问修改全局对象（"global object"）来访问全局作用域，
    // 在web浏览器中是`window`这个对象。
    // 在其他环境如Node.js中这个对象的名字可能会不同。
    window.permanent = 10;
})();
temporary; // 抛出引用异常ReferenceError
permanent; // = 10

// javascript最强大的功能之一就是闭包。
// 如果一个函数在另一个函数中定义，那么这个内部函数就拥有外部函数的所有变量的访问权，
// 即使在外部函数结束之后。
function sayHelloInFiveSeconds(name){
    var prompt = "Hello, " + name + "!";
    // 内部函数默认是放在局部作用域的，
    // 就像是用`var`声明的。
    function inner(){

```



```

        alert(prompt);
    }
    setTimeout(inner, 5000);
    // setTimeout是异步的，所以 sayHelloInFiveSeconds 函数会立即退出，
    // 而 setTimeout 会在后面调用inner
    // 然而，由于inner是由sayHelloInFiveSeconds“闭合包含”的，
    // 所以inner在其最终被调用时仍然能够访问`prompt`变量。
}
sayHelloInFiveSeconds("Adam"); // 会在5秒后弹出 "Hello, Adam!"

////////////////////////////////////
// 5. 对象、构造函数与原型

// 对象可以包含方法。
var myObj = {
    myFunc: function(){
        return "Hello world!";
    }
};
myObj.myFunc(); // = "Hello world!"

// 当对象中的函数被调用时，这个函数可以通过`this`关键字访问其依附的这个对象。
myObj = {
    myString: "Hello world!",
    myFunc: function(){
        return this.myString;
    }
};
myObj.myFunc(); // = "Hello world!"

// 但这个函数访问的其实是其运行时环境，而非定义时环境，即取决于函数是如何调用的。
// 所以如果函数被调用时不在这个对象的上下文中，就不会运行成功了。
var myFunc = myObj.myFunc;
myFunc(); // = undefined

// 相应的，一个函数也可以被指定为一个对象的方法，并且可以通过`this`访问
// 这个对象的成员，即使在函数被定义时并没有依附在对象上。
var myOtherFunc = function(){
    return this.myString.toUpperCase();
}
myObj.myOtherFunc = myOtherFunc;
myObj.myOtherFunc(); // = "HELLO WORLD!"

// 当我们通过`call`或者`apply`调用函数的时候，也可以为其指定一个执行上下文。
var anotherFunc = function(s){
    return this.myString + s;
}
anotherFunc.call(myObj, " And Hello Moon!"); // = "Hello World! And Hello Moon!"

// `apply`函数几乎完全一样，只是要求一个array来传递参数列表。

```

```
anotherFunc.apply(myObj, [" And Hello Sun!"]); // = "Hello World! And Hello Sun!"
```

// 当一个函数接受一系列参数，而你想传入一个array时特别有用。

```
Math.min(42, 6, 27); // = 6
```

```
Math.min([42, 6, 27]); // = NaN (uh-oh!)
```

```
Math.min.apply(Math, [42, 6, 27]); // = 6
```

// 但是`call`和`apply`只是临时的。如果我们希望函数附着在对象上，可以使用`bind`。

```
var boundFunc = anotherFunc.bind(myObj);
```

```
boundFunc(" And Hello Saturn!"); // = "Hello World! And Hello Saturn!"
```

// `bind`也可以用来部分应用一个函数（柯里化）。

```
var product = function(a, b){ return a * b; }
```

```
var doubler = product.bind(this, 2);
```

```
doubler(8); // = 16
```

// 当你通过`new`关键字调用一个函数时，就会创建一个对象，

// 而且可以通过`this`关键字访问该函数。

// 设计为这样调用的函数就叫做构造函数。

```
var MyConstructor = function(){
```

```
    this.myNumber = 5;
```

```
}
```

```
myNewObj = new MyConstructor(); // = {myNumber: 5}
```

```
myNewObj.myNumber; // = 5
```

// 每一个js对象都有一个‘原型’。当你要访问一个实际对象中没有定义的一个属性时，

// 解释器就回去找这个对象的原型。

// 一些JS实现会让你通过`__proto__`属性访问一个对象的原型。

// 这虽然对理解原型很有用，但是它并不是标准的一部分；

// 我们后面会介绍使用原型的标准方式。

```
var myObj = {  
    myString: "Hello world!"
```

```
};
```

```
var myPrototype = {
```

```
    meaningOfLife: 42,
```

```
    myFunc: function(){
```

```
        return this.myString.toLowerCase()
```

```
    }
```

```
};
```

```
myObj.__proto__ = myPrototype;
```

```
myObj.meaningOfLife; // = 42
```

// 函数也可以工作。

```
myObj.myFunc() // = "hello world!"
```

// 当然，如果你要访问的成员在原型当中也没有定义的话，解释器就会去找原型的原型，以此类推。

```
myPrototype.__proto__ = {
```

```
    myBoolean: true
```

```
};
```

```
myObj.myBoolean; // = true
```

// 这其中并没有对象的拷贝；每个对象实际上是持有原型对象的引用。

// 这意味着当我们改变对象的原型时，会影响到其他以这个原型为原型的对象。

```
myPrototype.meaningOfLife = 43;
```

```
myObj.meaningOfLife; // = 43
```

// 我们知道 `__proto__` 并非标准规定，实际上也没有标准办法来修改一个已存在对象的原型。

// 然而，我们有两种方式为指定原型创建一个新的对象。

// 第一种方式是 `Object.create`，这个方法是在最近才被添加到Js中的，

// 因此并不是所有的JS实现都有这个方法

```
var myObj = Object.create(myPrototype);
```

```
myObj.meaningOfLife; // = 43
```

// 第二种方式可以在任意版本中使用，不过必须通过构造函数。

// 构造函数有一个属性`prototype`。但是它 *不是* 构造函数本身的原型；相反，

// 是通过构造函数和`new`关键字创建的新对象的原型。

```
MyConstructor.prototype = {  
  myNumber: 5,  
  getMyNumber: function(){  
    return this.myNumber;  
  }  
};
```

```
var myNewObj2 = new MyConstructor();  
myNewObj2.getMyNumber(); // = 5
```

```
myNewObj2.myNumber = 6  
myNewObj2.getMyNumber(); // = 6
```

// 字符串和数字等内置类型也有通过构造函数来创建的包装类型

```
var myNumber = 12;
```

```
var myNumberObj = new Number(12);
```

```
myNumber == myNumberObj; // = true
```

// 但是它们并非严格等价

```
typeof myNumber; // = 'number'
```

```
typeof myNumberObj; // = 'object'
```

```
myNumber === myNumberObj; // = false
```

```
if (0){
```

```
  // 这段代码不会执行，因为0代表假
```

```
}
```

// 不过，包装类型和内置类型共享一个原型，

// 所以你实际可以给内置类型也增加一些功能，例如对string:

```
String.prototype.firstCharacter = function(){  
  return this.charAt(0);  
}
```

```
"abc".firstCharacter(); // = "a"
```

// 这个技巧经常用在“代码填充”中，来为老版本的javascript子集增加新版本js的特性，

// 这样就可以在老的浏览器中使用新功能了。

```
// 比如，我们知道Object.create并没有在所有的版本中都实现，
// 但是我们仍然可以通过“代码填充”来实现兼容：
if (Object.create === undefined){ // 如果存在则不覆盖
    Object.create = function(proto){
        // 用正确的原型来创建一个临时构造函数
        var Constructor = function(){};
        Constructor.prototype = proto;
        // 之后用它来创建一个新的对象
        return new Constructor();
    }
}
```

更多阅读

[Mozilla 开发者 网络](#) 提供了优秀的介绍 **Javascript**如何在浏览器中使用的文档。而且它是wiki，所以你也可以自行编辑来分享你的知识。

MDN的 [A re-introduction to JavaScript](#) 覆盖了这里提到的绝大多数话题的细节。该导引的大多数内容被限定在只是**Javascript**这个语言本身； 如果你想了解**Javascript**是如何在网页中被应用的，那么可以查看 [Document Object Model](#)

[Learn Javascript by Example and with Challenges](#) 是本参考的另一个版本，并包含了挑战习题。

[Javascript Garden](#) 是一个深入 讲解所有**Javascript**反直觉部分的导引。

[JavaScript: The Definitive Guide](#) 是一个经典的指导参考书。

除了这篇文章的直接贡献者之外，这篇文章也参考了这个网站上 [Louie Dinh](#) 的 **Python** 教程，以及 [Mozilla](#)开发者网络上的[JS Tutorial](#)。

language: json contributors:

- ["Anna Harren", "<https://github.com/iirelu>"] translators:
 - ["Zach Zhang", "<https://github.com/checkcheckzz>"] filename: learnjson-cn.json lang: zh-cn
-

因为JSON是一个极其简单的数据交换形式，这个最有可能将会是曾经最简单 的Learn X in Y Minutes。

最纯正形式的JSON没有实际的注解，但是大多数解析器将会 接受C-风格(`//`, `/* */`)的注解。为了这个目的，但是， 一切都将会是100%有效的JSON。幸亏，它是不言自明的。

```
{
  "numbers": 0,
  "strings": "Hellø, wørld. All unicode is allowed, along with \"escaping\".",
  "has bools?": true,
  "nothingness": null,

  "big number": 1.2e+100,

  "objects": {
    "comment": "Most of your structure will come from objects.",

    "array": [0, 1, 2, 3, "Arrays can have anything in them.", 5],

    "another object": {
      "comment": "These things can be nested, very useful."
    }
  },

  "silliness": [
    {
      "sources of potassium": ["bananas"]
    },
    [
      [1, 0, 0, 0],
      [0, 1, 0, 0],
      [0, 0, 1, "neo"],
      [0, 0, 0, 1]
    ]
  ],

  "that was short": "And, you're done. You now know everything JSON has to offer."
}
```

language: Julia filename: learn-julia-zh.jl contributors:

```
- ["Jichao Ouyang", "http://oyanglul.us"]
```

translators:

```
- ["Jichao Ouyang", "http://oyanglul.us"]
```

lang: zh-cn

```
# 单行注释只需要一个井号
#= 多行注释
    只需要以 '#=' 开始 '=#' 结束
    还可以嵌套.
=#

#####
## 1. 原始类型与操作符
#####

# Julia 中一切皆是表达式。

# 这是一些基本数字类型。
3 # => 3 (Int64)
3.2 # => 3.2 (Float64)
2 + 1im # => 2 + 1im (Complex{Int64})
2//3 # => 2//3 (Rational{Int64})

# 支持所有的普通中缀操作符。
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20
35 / 5 # => 7.0
5 / 2 # => 2.5 # 用 Int 除 Int 永远返回 Float
div(5, 2) # => 2 # 使用 div 截断小数点
5 \ 35 # => 7.0
2 ^ 2 # => 4 # 次方，不是二进制 xor
12 % 10 # => 2

# 用括号提高优先级
(1 + 3) * 2 # => 8

# 二进制操作符
~2 # => -3 # 非
3 & 5 # => 1 # 与
```

[illegible]

```
#####
## 2. 变量与集合
#####

# 给变量赋值就是声明变量
some_var = 5 # => 5
some_var # => 5

# 访问未声明变量会抛出异常
try
    some_other_var # => ERROR: some_other_var not defined
catch e
    println(e)
end

# 变量名需要以字母开头。
# 之后任何字母，数字，下划线，叹号都是合法的。
SomeOtherVar123! = 6 # => 6

# 甚至可以用 unicode 字符
🍷 = 8 # => 8
# 用数学符号非常方便
2 * π # => 6.283185307179586

# 注意 Julia 的命名规约：
#
# * 变量名为小写，单词之间以下划线连接('\'_')。
#
# * 类型名以大写字母开头，单词以 CamelCase 方式连接。
#
# * 函数与宏的名字小写，无下划线。
#
# * 会改变输入的函数名末位为 !。
# 这类函数有时被称为 mutating functions 或 in-place functions.

# 数组存储一系列值，index 从 1 开始。
a = Int64[] # => 0-element Int64 Array

# 一维数组可以以逗号分隔值的方式声明。
b = [4, 5, 6] # => 包含 3 个 Int64 类型元素的数组: [4, 5, 6]
b[1] # => 4
b[end] # => 6

# 二维数组以分号分隔维度。
matrix = [1 2; 3 4] # => 2x2 Int64 数组: [1 2; 3 4]

# 使用 push! 和 append! 往数组末尾添加元素
push!(a,1) # => [1]
push!(a,2) # => [1,2]
push!(a,4) # => [1,2,4]
```



```

push!(a,3)      # => [1,2,4,3]
append!(a,b) # => [1,2,4,3,4,5,6]

# 用 pop 弹出末尾元素
pop!(b)         # => 6 and b is now [4,5]

# 可以再放回去
push!(b,6)      # b 又变成了 [4,5,6].

a[1] # => 1 # 永远记住 Julia 的 index 从 1 开始!

# 用 end 可以直接取到最后索引. 可用作任何索引表达式
a[end] # => 6

# 还支持 shift 和 unshift
shift!(a) # => 返回 1, 而 a 现在是 [2,4,3,4,5,6]
unshift!(a,7) # => [7,2,4,3,4,5,6]

# 以叹号结尾的函数名表示它会改变参数的值
arr = [5,4,6] # => 包含三个 Int64 元素的数组: [5,4,6]
sort(arr) # => [4,5,6]; arr 还是 [5,4,6]
sort!(arr) # => [4,5,6]; arr 现在是 [4,5,6]

# 越界会抛出 BoundsError 异常
try
    a[0] # => ERROR: BoundsError() in getindex at array.jl:270
    a[end+1] # => ERROR: BoundsError() in getindex at array.jl:270
catch e
    println(e)
end

# 错误会指出发生的行号, 包括标准库
# 如果你有 Julia 源代码, 你可以找到这些地方

# 可以用 range 初始化数组
a = [1:5] # => 5-element Int64 Array: [1,2,3,4,5]

# 可以切割数组
a[1:3] # => [1, 2, 3]
a[2:end] # => [2, 3, 4, 5]

# 用 splice! 切割原数组
arr = [3,4,5]
splice!(arr,2) # => 4 ; arr 变成了 [3,5]

# 用 append! 连接数组
b = [1,2,3]
append!(a,b) # a 变成了 [1, 2, 3, 4, 5, 1, 2, 3]

# 检查元素是否在数组中
in(1, a) # => true

```

```

# 用 length 获得数组长度
length(a) # => 8

# Tuples 是 immutable 的
tup = (1, 2, 3) # => (1,2,3) # an (Int64,Int64,Int64) tuple.
tup[1] # => 1
try:
    tup[1] = 3 # => ERROR: no method setindex!((Int64,Int64,Int64),Int64,Int64)
catch e
    println(e)
end

# 大多数组的函数同样支持 tuples
length(tup) # => 3
tup[1:2] # => (1,2)
in(2, tup) # => true

# 可以将 tuples 元素分别赋给变量
a, b, c = (1, 2, 3) # => (1,2,3) # a is now 1, b is now 2 and c is now 3

# 不用括号也可以
d, e, f = 4, 5, 6 # => (4,5,6)

# 单元素 tuple 不等于其元素值
(1,) == 1 # => false
(1) == 1 # => true

# 交换值
e, d = d, e # => (5,4) # d is now 5 and e is now 4

# 字典Dictionaries store mappings
empty_dict = Dict{Any,Any}()

# 也可以用字面量创建字典
filled_dict = ["one"=> 1, "two"=> 2, "three"=> 3]
# => Dict{ASCIIString,Int64}

# 用 [] 获得键值
filled_dict["one"] # => 1

# 获得所有键
keys(filled_dict)
# => KeyIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# 注意, 键的顺序不是插入时的顺序

# 获得所有值
values(filled_dict)
# => ValueIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>1,"two"=>2])
# 注意, 值的顺序也一样

```

```

# 用 in 检查键值是否已存在, 用 haskey 检查键是否存在
in(("one", 1), filled_dict) # => true
in(("two", 3), filled_dict) # => false
haskey(filled_dict, "one") # => true
haskey(filled_dict, 1) # => false

# 获取不存在的键的值会抛出异常
try
    filled_dict["four"] # => ERROR: key not found: four in getindex at dict.jl:489
catch e
    println(e)
end

# 使用 get 可以提供默认值来避免异常
# get(dictionary, key, default_value)
get(filled_dict, "one", 4) # => 1
get(filled_dict, "four", 4) # => 4

# 用 Sets 表示无序不可重复的值的集合
empty_set = Set() # => Set{Any}()
# 初始化一个 Set 并定义其值
filled_set = Set{Int64}(1, 2, 3, 4) # => Set{Int64}(1, 2, 3, 4)

# 添加值
push!(filled_set, 5) # => Set{Int64}(5, 4, 2, 3, 1)

# 检查是否存在某值
in(2, filled_set) # => true
in(10, filled_set) # => false

# 交集, 并集, 差集
other_set = Set{Int64}(3, 4, 5, 6) # => Set{Int64}(6, 4, 5, 3)
intersect(filled_set, other_set) # => Set{Int64}(3, 4, 5)
union(filled_set, other_set) # => Set{Int64}(1, 2, 3, 4, 5, 6)
setdiff(Set{Int64}(1, 2, 3, 4), Set{Int64}(2, 3, 5)) # => Set{Int64}(1, 4)

#####
## 3. 控制流
#####

# 声明一个变量
some_var = 5

# 这是一个 if 语句, 缩进不是必要的
if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10 # elseif 是可选的.
    println("some_var is smaller than 10.")
else # else 也是可选的.

```

```

    println("some_var is indeed 10.")
end
# => prints "some var is smaller than 10"

# For 循环遍历
# Iterable 类型包括 Range, Array, Set, Dict, 以及 String.
for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # 可用 $ 将 variables 或 expression 转换为字符串into strings
end
# prints:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

# You can use 'in' instead of '='.
for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end
# prints:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

for a in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("${a[1]} is a ${a[2]}")
end
# prints:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

for (k,v) in ["dog"=>"mammal","cat"=>"mammal","mouse"=>"mammal"]
    println("$k is a $v")
end
# prints:
#   dog is a mammal
#   cat is a mammal
#   mouse is a mammal

# While 循环
x = 0
while x < 4
    println(x)
    x += 1 # x = x + 1
end
# prints:
#   0
#   1
#   2

```

```

# 3

# 用 try/catch 处理异常
try
  error("help")
catch e
  println("caught it $e")
end
# => caught it RuntimeException("help")

#####
## 4. 函数
#####

# 用关键字 'function' 可创建一个新函数
#function name(arglist)
#  body...
#end
function add(x, y)
  println("x is $x and y is $y")

  # 最后一行语句的值为返回
  x + y
end

add(5, 6) # => 在 "x is 5 and y is 6" 后会打印 11

# 还可以定义接收可变长参数的函数
function varargs(args...)
  return args
  # 关键字 return 可在函数内部任何地方返回
end
# => varargs (generic function with 1 method)

varargs(1,2,3) # => (1,2,3)

# 省略号 ... 被称为 splat.
# 刚刚用在了函数定义中
# 还可以用在函数的调用
# Array 或者 Tuple 的内容会变成参数列表
Set([1,2,3]) # => Set{Array{Int64,1}}([1,2,3]) # 获得一个 Array 的 Set
Set([1,2,3]...) # => Set{Int64}(1,2,3) # 相当于 Set(1,2,3)

x = (1,2,3) # => (1,2,3)
Set(x) # => Set{(Int64,Int64,Int64)}((1,2,3)) # 一个 Tuple 的 Set
Set(x...) # => Set{Int64}(2,3,1)

# 可定义可选参数的函数
function defaults(a,b,x=5,y=6)

```

```

        return "$a $b and $x $y"
    end

    defaults('h','g') # => "h g and 5 6"
    defaults('h','g','j') # => "h g and j 6"
    defaults('h','g','j','k') # => "h g and j k"
    try
        defaults('h') # => ERROR: no method defaults(Char,)
        defaults() # => ERROR: no methods defaults()
    catch e
        println(e)
    end

    # 还可以定义键值对的参数
    function keyword_args(;k1=4,name2="hello") # note the ;
        return ["k1"=>k1,"name2"=>name2]
    end

    keyword_args(name2="ness") # => ["name2"=>"ness","k1"=>4]
    keyword_args(k1="mine") # => ["k1"=>"mine","name2"=>"hello"]
    keyword_args() # => ["name2"=>"hello","k1"=>4]

    # 可以组合各种类型的参数在同一个函数的参数列表中
    function all_the_args(normal_arg, optional_positional_arg=2; keyword_arg="foo")
        println("normal arg: $normal_arg")
        println("optional arg: $optional_positional_arg")
        println("keyword arg: $keyword_arg")
    end

    all_the_args(1, 3, keyword_arg=4)
    # prints:
    #   normal arg: 1
    #   optional arg: 3
    #   keyword arg: 4

    # Julia 有一等函数
    function create_adder(x)
        adder = function (y)
            return x + y
        end
        return adder
    end

    # 这是用 "stabby lambda syntax" 创建的匿名函数
    (x -> x > 2)(3) # => true

    # 这个函数和上面的 create_adder 一模一样
    function create_adder(x)
        y -> x + y
    end

```

```

# 你也可以给内部函数起个名字
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end

add_10 = create_adder(10)
add_10(3) # => 13

# 内置的高阶函数有
map(add_10, [1,2,3]) # => [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) # => [6, 7]

# 还可以使用 list comprehensions 替代 map
[add_10(i) for i=[1, 2, 3]] # => [11, 12, 13]
[add_10(i) for i in [1, 2, 3]] # => [11, 12, 13]

#####
## 5. 类型
#####

# Julia 有类型系统
# 所有的值都有类型；但变量本身没有类型
# 你可以用 `typeof` 函数获得值的类型
typeof(5) # => Int64

# 类型是一等值
typeof(Int64) # => DataType
typeof(DataType) # => DataType
# DataType 是代表类型的类型，也代表他自己的类型

# 类型可用作文档化，优化，以及调度
# 并不是静态检查类型

# 用户还可以自定义类型
# 跟其他语言的 records 或 structs 一样
# 用 `type` 关键字定义新的类型

# type Name
#   field::OptionalType
#   ...
# end
type Tiger
    taillength::Float64
    coatcolor # 不附带类型标注的相当于 `::Any`
end

# 构造函数参数是类型的属性

```

```

tiger = Tiger(3.5,"orange") # => Tiger(3.5,"orange")

# 用新类型作为构造函数还会创建一个类型
sherekhan = typeof(tiger)(5.6,"fire") # => Tiger(5.6,"fire")

# struct 类似的类型被称为具体类型
# 他们可被实例化但不能有子类型
# 另一种类型是抽象类型

# abstract Name
abstract Cat # just a name and point in the type hierarchy

# 抽象类型不能被实例化，但是可以有子类型
# 例如，Number 就是抽象类型
subtypes(Number) # => 6-element Array{Any,1}:
      Complex{Float16}
      Complex{Float32}
      Complex{Float64}
      Complex{T<:Real}
      ImaginaryUnit
      Real
subtypes(Cat) # => 0-element Array{Any,1}

# 所有的类型都有父类型；可以用函数 `super` 得到父类型。
typeof(5) # => Int64
super(Int64) # => Signed
super(Signed) # => Real
super(Real) # => Number
super(Number) # => Any
super(super(Signed)) # => Number
super(Any) # => Any
# 所有这些类型，除了 Int64，都是抽象类型。

# <: 是类型集成操作符
type Lion <: Cat # Lion 是 Cat 的子类型
    mane_color
    roar::String
end

# 可以继续为你的类型定义构造函数
# 只需要定义一个同名的函数
# 并调用已有的构造函数设置一个固定参数
Lion(roar::String) = Lion("green",roar)
# 这是一个外部构造函数，因为他再类型定义之外

type Panther <: Cat # Panther 也是 Cat 的子类型
    eye_color
    Panther() = new("green")
    # Panthers 只有这个构造函数，没有默认构造函数
end
# 使用内置构造函数，如 Panther，可以让你控制

```



```

# 如何构造类型的值
# 应该尽可能使用外部构造函数而不是内部构造函数

#####

## 6. 多分派
#####

# 在Julia中，所有的具名函数都是类属函数
# 这意味着他们都是有很大小方法组成的
# 每个 Lion 的构造函数都是类属函数 Lion 的方法

# 我们来看一个非构造函数的例子

# Lion, Panther, Tiger 的 meow 定义为
function meow(animal::Lion)
    animal.roar # 使用点符号访问属性
end

function meow(animal::Panther)
    "grrr"
end

function meow(animal::Tiger)
    "rawwwr"
end

# 试试 meow 函数
meow(tigger) # => "rawwr"
meow(Lion("brown","ROAAR")) # => "ROAAR"
meow(Panther()) # => "grrr"

# 再看看层次结构
issubtype(Tiger,Cat) # => false
issubtype(Lion,Cat) # => true
issubtype(Panther,Cat) # => true

# 定义一个接收 Cats 的函数
function pet_cat(cat::Cat)
    println("The cat says $(meow(cat))")
end

pet_cat(Lion("42")) # => prints "The cat says 42"
try
    pet_cat(tigger) # => ERROR: no method pet_cat(Tiger,)
catch e
    println(e)
end

# 在面向对象语言中，通常都是单分派
# 这意味着分派方法是通过第一个参数的类型决定的
# 在Julia中，所有参数类型都会被考虑到

```

```

# 让我们定义有多个参数的函数，好看看区别
function fight(t::Tiger,c::Cat)
    println("The $(t.coatcolor) tiger wins!")
end
# => fight (generic function with 1 method)

fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The orange tiger wins!

# 让我们修改一下传入具体为 Lion 类型时的行为
fight(t::Tiger,l::Lion) = println("The $(l.mane_color)-maned lion wins!")
# => fight (generic function with 2 methods)

fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The green-maned lion wins!

# 把 Tiger 去掉
fight(l::Lion,c::Cat) = println("The victorious cat says $(meow(c))")
# => fight (generic function with 3 methods)

fight(Lion("balooa!"),Panther()) # => prints The victorious cat says grrr
try
    fight(Panther(),Lion("RAWR")) # => ERROR: no method fight(Panther,Lion)
catch
end

# 在试试让 Cat 在前面
fight(c::Cat,l::Lion) = println("The cat beats the Lion")
# => Warning: New definition
#   fight(Cat,Lion) at none:1
# is ambiguous with
#   fight(Lion,Cat) at none:2.
# Make sure
#   fight(Lion,Lion)
# is defined first.
#fight (generic function with 4 methods)

# 警告说明了无法判断使用哪个 fight 方法
fight(Lion("RAR"),Lion("brown","rarr")) # => prints The victorious cat says rarr
# 结果在老版本 Julia 中可能会不一样

fight(l::Lion,l2::Lion) = println("The lions come to a tie")
fight(Lion("RAR"),Lion("brown","rarr")) # => prints The lions come to a tie

# Under the hood
# 你还可以看看 llvm 以及生成的汇编代码

square_area(l) = l * l      # square_area (generic function with 1 method)

```

```
square_area(5) #25
```

```
# 给 square_area 一个整形时发生什么
```

```
code_native(square_area, (Int32,))
```

```
#      .section      __TEXT,__text,regular,pure_instructions
#      Filename: none
#      Source line: 1          # Prologue
#      push    RBP
#      mov RBP, RSP
#      Source line: 1
#      movsxd  RAX, EDI        # Fetch 1 from memory?
#      imul    RAX, RAX        # Square 1 and store the result in RAX
#      pop RBP                # Restore old base pointer
#      ret                    # Result will still be in RAX
```

```
code_native(square_area, (Float32,))
```

```
#      .section      __TEXT,__text,regular,pure_instructions
#      Filename: none
#      Source line: 1
#      push    RBP
#      mov RBP, RSP
#      Source line: 1
#      vmulss  XMM0, XMM0, XMM0 # Scalar single precision multiply (AVX)
#      pop RBP
#      ret
```

```
code_native(square_area, (Float64,))
```

```
#      .section      __TEXT,__text,regular,pure_instructions
#      Filename: none
#      Source line: 1
#      push    RBP
#      mov RBP, RSP
#      Source line: 1
#      vmulsd  XMM0, XMM0, XMM0 # Scalar double precision multiply (AVX)
#      pop RBP
#      ret
#
```

```
# 注意 只要参数中又浮点类型, Julia 就使用浮点指令
```

```
# 让我们计算一下圆的面积
```

```
circle_area(r) = pi * r * r      # circle_area (generic function with 1 method)
```

```
circle_area(5)                  # 78.53981633974483
```

```
code_native(circle_area, (Int32,))
```

```
#      .section      __TEXT,__text,regular,pure_instructions
#      Filename: none
#      Source line: 1
#      push    RBP
#      mov RBP, RSP
#      Source line: 1
#      vcvtsi2sd XMM0, XMM0, EDI      # Load integer (r) from memory
#      movabs  RAX, 4593140240        # Load pi
```

```
#      vmulsd  XMM1, XMM0, QWORD PTR [RAX]  # pi * r
#      vmulsd  XMM0, XMM0, XMM1             # (pi * r) * r
#      pop RBP
#      ret
#
```

code_native(circle_area, (Float64,))

```
#      .section  __TEXT,__text,regular,pure_instructions
#      Filename: none
#      Source line: 1
#      push    RBP
#      mov RBP, RSP
#      movabs  RAX, 4593140496
#      Source line: 1
#      vmulsd  XMM1, XMM0, QWORD PTR [RAX]
#      vmulsd  XMM0, XMM1, XMM0
#      pop RBP
#      ret
#
```

language: LiveScript filename: learnLivescript.ls contributors:

```
- ["Christina Whyte", "http://github.com/kurisuwhyte/"]
```

translators:

```
- ["ShengDa Lyu", "http://github.com/SDLyu/"]
```

lang: zh-cn

LiveScript 是一种具有函数式特性且编译成 JavaScript 的语言，能对应 JavaScript 的基本语法。还有些额外的特性如：柯里化，组合函数，模式匹配，还有借镜于 Haskell, F# 和 Scala 的许多特点。

LiveScript 诞生于 [Coco](#)，而 Coco 诞生于 [CoffeeScript](#)。LiveScript 目前已释出稳定版本，开发中的新版本将会加入更多特性。

非常期待您的反馈，你可以通过 [@kurisuwhyte](#) 与我连系 :)

```
# 与 CoffeeScript 一样，LiveScript 使用 # 单行注解。
```

```
/*  
  多行注解与 C 相同。使用注解可以避免被当成 JavaScript 输出。  
*/
```

```
# 语法的部份，LiveScript 使用缩进取代 {} 来定义区块，  
# 使用空白取代 () 来执行函数。
```

```
#####  
## 1. 值类型  
#####
```

```
# `void` 取代 `undefined` 表示未定义的值  
void          # 与 `undefined` 等价但更安全（不会被覆写）
```

```
# 空值则表示成 Null。  
null
```

```
# 最基本的值类型数据是逻辑类型：  
true  
false
```

```
# 逻辑类型的一些别名，等价于前者：  
on; off
```

```
yes; no
```

```
# 数字与 JS 一样，使用倍精度浮点数表示。
```

```
10
```

```
0.4      # 开头的 0 是必要的
```

```
# 可以使用底线及单位后缀提高可读性，编译器会自动略过底线及单位后缀。
```

```
12_344km
```

```
# 字符串与 JS 一样，是一种不可变的字元序列：
```

```
"Christina"      # 单引号也可以！
```

```
"""Multi-line  
    strings  
    are  
    okay  
    too."""
```

```
# 在前面加上 \ 符号也可以表示字符串：
```

```
\keyword          # => 'keyword'
```

```
# 数组是值的有序集合。
```

```
fruits =  
    * \apple  
    * \orange  
    * \pear
```

```
# 可以用 [] 简洁地表示数组：
```

```
fruits = [ \apple, \orange, \pear ]
```

```
# 你可以更方便地建立字符串数组，并使用空白区隔元素。
```

```
fruits = <[ apple orange pear ]>
```

```
# 以 0 为起始值的数组下标获取元素：
```

```
fruits[0]          # => "apple"
```

```
# 对象是无序键值对集合（更多细节将在下面章节讨论）。
```

```
person =  
    name: "Christina"  
    likes:  
        * "kittens"  
        * "and other cute stuff"
```

```
# 你也可以用更简洁的方式表示对象：
```

```
person = {name: "Christina", likes: ["kittens", "and other cute stuff"]}
```

```

# 可以通过键值获取值:
person.name      # => "Christina"
person["name"]   # => "Christina"

# 正则表达式的使用跟 JavaScript 一样:
trailing-space = /\s$/      # dashed-words 变成 dashedWords

# 你也可以用多行描述表达式! (注解和空白会被忽略)
funRE = //
    function\s+(.+)          # name
    \s* \((.*)\) \s*         # arguments
    { (.* ) }                # body
    //

#####
## 2. 基本运算
#####

# 数值操作符与 JavaScript 一样:
1 + 2    # => 3
2 - 1    # => 1
2 * 3    # => 6
4 / 2    # => 2
3 % 2    # => 1

# 比较操作符大部份也一样, 除了 `==` 等价于 JS 中的 `===`,
# JS 中的 `==` 在 LiveScript 里等价于 `~=`,
# `===` 能进行对象、数组和严格比较。
2 == 2    # => true
2 == "2"  # => false
2 ~= "2"  # => true
2 === "2" # => false

[1,2,3] == [1,2,3]    # => false
[1,2,3] === [1,2,3]   # => true

+0 == -0    # => true
+0 === -0   # => false

# 其它关系操作符包括 <、<=、> 和 >=

# 逻辑值可以通过 `or`、`and` 和 `not` 结合:
true and false # => false
false or true  # => true
not false      # => true

# 集合也有一些便利的操作符

```

```

[1, 2] ++ [3, 4]           # => [1, 2, 3, 4]
'a' in <[ a b c ]>         # => true
'name' of { name: 'Chris' } # => true

#####

## 3. 函数

#####

# 因为 LiveScript 是函数式特性的语言，你可以期待函数在语言里被高规格的对特。
add = (left, right) -> left + right
add 1, 2           # => 3

# 加上 ! 防止函数执行后的返回值
two = -> 2
two!

# LiveScript 与 JavaScript 一样使用函数作用域，且一样拥有闭包的特性。
# 与 JavaScript 不同的地方在于，`=` 变量赋值时，左边的对象永远不用变量宣告。

# `:=` 操作符允许*重新赋值*父作用域里的变量。

# 你可以解构函数的参数，从不定长度的参数结构里获取感兴趣的值。
tail = ([head, ...rest]) -> rest
tail [1, 2, 3] # => [2, 3]

# 你也可以使用一元或二元操作符转换参数。当然也可以预设传入的参数值。
foo = (a = 1, b = 2) -> a + b
foo!    # => 3

# 你可以以拷贝的方式传入参数来避免副作用，例如：
copy = (^target, source) ->
  for k,v of source => target[k] = v
  target
a = { a: 1 }
copy a, { b: 2 }      # => { a: 1, b: 2 }
a                     # => { a: 1 }

# 使用长箭头取代短箭头来柯里化一个函数：
add = (left, right) --> left + right
add1 = add 1
add1 2                # => 3

# 函数里有一个隐式的 `it` 变量，意味着你不用宣告它。
identity = -> it
identity 1            # => 1

# 操作符在 LiveScript 里不是一个函数，但你可以简单地将它们转换成函数！
# Enter the operator sectioning:

```



```
divide-by-2 = (/ 2)
[2, 4, 8, 16].map(divide-by-2) .reduce (+)
```

LiveScript 里不只有应用函数，如同其它良好的函数式语言，你可以合并函数获得更多发挥：

```
double-minus-one = (- 1) . (* 2)
```

除了普通的数学公式合并 `f . g` 之外，还有 `>>` 和 `<<` 操作符定义函数的合并顺序。

```
double-minus-one = (* 2) >> (- 1)
```

```
double-minus-one = (- 1) << (* 2)
```

说到合并函数的参数，LiveScript 使用 `|>` 和 `<|` 操作符将参数传入：

```
map = (f, xs) --> xs.map f
```

```
[1 2 3] |> map (* 2)          # => [2 4 6]
```

你也可以选择填入值的位置，只需要使用底线 `_` 标记：

```
reduce = (f, xs, initial) --> xs.reduce f, initial
```

```
[1 2 3] |> reduce (+), _, 0    # => 6
```

你也能使 `_` 让任何函数变成偏函数应用：

```
div = (left, right) -> left / right
```

```
div-by-2 = div _, 2
```

```
div-by-2 4          # => 2
```

最后，也很重要的，LiveScript 拥有後呼叫特性， 可以是基於回调的代码

（你可以试试其它函数式特性的解法，比如 Promises）：

```
readFile = (name, f) -> f name
```

```
a <- readFile 'foo'
```

```
b <- readFile 'bar'
```

```
console.log a + b
```

等同於：

```
readFile 'foo', (a) -> readFile 'bar', (b) -> console.log a + b
```

```
#####
```

```
## 4. 模式、判断和流程控制
```

```
#####
```

流程控制可以使用 `if...else` 表达式：

```
x = if n > 0 then \positive else \negative
```

除了 `then` 你也可以使用 `=>`

```
x = if n > 0 => \positive
```

```
    else      \negative
```

过於复杂的流程可以用 `switch` 表达式代替：

```
y = {}
```

```
x = switch
  | (typeof y) is \number => \number
  | (typeof y) is \string => \string
  | 'length' of y          => \array
  | otherwise              => \object      # `otherwise` 和 `_` 是等价的。
```

函数主体、宣告式和赋值式可以表式成 `switch`，这可以省去一些代码：

```
take = (n, [x, ...xs]) -->
  | n == 0 => []
  | _      => [x] ++ take (n - 1), xs
```

```
#####
```

5. 推导式

```
#####
```

在 JavaScript 的标准函式库里有一些辅助函数能帮助处理列表及对象
(LiveScript 则带有一个 prelude.ls，作为标准函式库的补充)，
推导式能让你使用优雅的语法且快速地处理这些事：

```
oneToTwenty = [1 to 20]
evens        = [x for x in oneToTwenty when x % 2 == 0]
```

在推导式里 `when` 和 `unless` 可以当成过滤器使用。

对象推导式在使用上也是同样的方式，差别在于你使用的是对象而不是数组：

```
copy = { [k, v] for k, v of source }
```

```
#####
```

6. OOP

```
#####
```

虽然 LiveScript 是一门函数式语言，但具有一些命令式及面向对象的特性。
像是 class 语法和一些借镜於 CoffeeScript 的类别继承语法糖：

```
class Animal
  (@name, kind) ->
    @kind = kind
  action: (what) -> "*#{@name} (a #{@kind}) #{what}*"

class Cat extends Animal
  (@name) -> super @name, 'cat'
  purr: -> @action 'purrs'
```

```
kitten = new Cat 'Mei'
kitten.purr!      # => "*Mei (a cat) purrs*"

# 除了类别的单一继承模式之外，还提供了像混入 (Mixins) 这种特性。
```

Mixins 在语言里被当成普通对象：

```
Huggable =
  hug: -> @action 'is hugged'
```

```
class SnugglyCat extends Cat implements Huggable

kitten = new SnugglyCat 'Purr'
kitten.hug!      # => "*Mei (a cat) is hugged*"
```

延伸阅读

LiveScript 还有许多强大之处，但这些应该足够启发你写些小型函数式程式了。 [LiveScript](#)有更多关于 LiveScript 的资讯 和线上编译器等着你来试！

你也可以参考 [prelude.js](#)，和一些 [#livescript](#) 的网络聊天室频道。

language: Lua lang: zh-cn contributors:

```
- ["Tyler Neylon", "http://tylerneylon.com/"]
- ["Rob Hoelz", "http://hoelz.ro"]
- ["Jakukyo Friel", "http://weakish.github.io"]
- ["Craig Roddin", "craig.roddin@gmail.com"]
- ["Amr Tamimi", "https://amrtamimi.com"]
```

translators:

```
- ["Jakukyo Friel", "http://weakish.github.io"]
```

filename: lua-cn.lua

```
-- 单行注释以两个连字符开头

--[[
    多行注释
--]]

-----

-- 1. 变量和流程控制
-----

num = 42 -- 所有的数字都是双精度浮点型。
-- 别害怕，64位的双精度浮点型数字中有52位用于
-- 保存精确的整型值；对于52位以内的整型值，
-- 不用担心精度问题。

s = 'walternate' -- 和Python一样，字符串不可变。
t = "也可以用双引号"
u = [[ 多行的字符串
        以两个方括号
        开始和结尾。]]
t = nil -- 撤销t的定义；Lua 支持垃圾回收。

-- 块使用do/end之类的关键字标识：
while num < 50 do
    num = num + 1 -- 不支持 ++ 或 += 运算符。
end

-- If语句：
if num > 40 then
    print('over 40')
elseif s ~= 'walternate' then -- ~= 表示不等于。
```

```

-- 像Python一样，用 == 检查是否相等；字符串同样适用。
io.write('not over 40\n') -- 默认标准输出。
else
-- 默认全局变量。
thisIsGlobal = 5 -- 通常使用驼峰。

-- 如何定义局部变量：
local line = io.read() -- 读取标准输入的下一行。

-- ..操作符用于连接字符串：
print('Winter is coming, ' .. line)
end

-- 未定义的变量返回nil。
-- 这不是错误：
foo = anUnknownVariable -- 现在 foo = nil.

```

```

aBoolValue = false

```

```

-- 只有nil和false为假；0和 ''均为真！
if not aBoolValue then print('false') end

```

```

-- 'or'和 'and'短路
-- 类似于C/js里的 a?b:c 操作符：
ans = aBoolValue and 'yes' or 'no' --> 'no'

```

```

karlSum = 0
for i = 1, 100 do -- 范围包含两端
    karlSum = karlSum + i
end

```

```

-- 使用 "100, 1, -1" 表示递减的范围：
fredSum = 0
for j = 100, 1, -1 do fredSum = fredSum + j end

```

```

-- 通常，范围表达式为begin, end[, step].

```

```

-- 循环的另一种结构：
repeat
    print('the way of the future')
    num = num - 1
until num == 0

```

```

-----
-- 2. 函数。
-----

```

```

function fib(n)
    if n < 2 then return n end
    return fib(n - 2) + fib(n - 1)
end

```

```
-- 支持闭包及匿名函数：
function adder(x)
  -- 调用adder时，会创建返回的函数，
  -- 并且会记住x的值：
  return function (y) return x + y end
end
a1 = adder(9)
a2 = adder(36)
print(a1(16)) --> 25
print(a2(64)) --> 100

-- 返回值、函数调用和赋值都可以
-- 使用长度不匹配的list。
-- 不匹配的接收方会被赋值nil；
-- 不匹配的发送方会被丢弃。

x, y, z = 1, 2, 3, 4
-- x = 1、y = 2、z = 3，而 4 会被丢弃。

function bar(a, b, c)
  print(a, b, c)
  return 4, 8, 15, 16, 23, 42
end

x, y = bar('zaphod') --> 打印 "zaphod nil nil"
-- 现在 x = 4, y = 8，而值15..42被丢弃。

-- 函数是一等公民，可以是局部的，也可以是全局的。
-- 以下表达式等价：
function f(x) return x * x end
f = function (x) return x * x end

-- 这些也是等价的：
local function g(x) return math.sin(x) end
local g; g = function (x) return math.sin(x) end
-- 以上均因'local g'，使得g可以自引用。
local g = function(x) return math.sin(x) end
-- 等价于 local function g(x)...，但函数体中g不可自引用

-- 顺便提下，三角函数以弧度为单位。

-- 用一个字符串参数调用函数，可以省略括号：
print 'hello' --可以工作。

-- 调用函数时，如果只有一个table参数，
-- 同样可以省略括号（table详情见下）：
print {} -- 一样可以工作。

-----
-- 3. Table。
```

```

-----

-- Table = Lua唯一的组合数据结构;
--      它们是关联数组。
-- 类似于PHP的数组或者js的对象，
-- 它们是哈希表或者字典，也可以当列表使用。

-- 按字典/map的方式使用Table:

-- Dict字面量默认使用字符串类型的key:
t = {key1 = 'value1', key2 = false}

-- 字符串key可以使用类似js的点标记:
print(t.key1)  -- 打印 'value1'.
t.newKey = {}  -- 添加新的键值对。
t.key2 = nil   -- 从table删除 key2。

-- 使用任何非nil的值作为key:
u = {'@!#' = 'qbert', [{}] = 1729, [6.28] = 'tau'}
print(u[6.28])  -- 打印 "tau"

-- 数字和字符串的key按值匹配的
-- table按id匹配。
a = u['@!#']  -- 现在 a = 'qbert'.
b = u[{}]     -- 我们或许期待的是 1729， 但是得到的是nil:
-- b = nil , 因为没有找到。
-- 之所以没找到，是因为我们用的key与保存数据时用的不是同
-- 一个对象。
-- 所以字符串和数字是移植性更好的key。

-- 只需要一个table参数的函数调用不需要括号:
function h(x) print(x.key1) end
h{key1 = 'Sonmi~451'}  -- 打印'Sonmi~451'.

for key, val in pairs(u) do  -- 遍历Table
    print(key, val)
end

-- _G 是一个特殊的table，用于保存所有的全局变量
print(_G['_G'] == _G)  -- 打印'true'。

-- 按列表/数组的方式使用:

-- 列表字面量隐式添加整数键:
v = {'value1', 'value2', 1.21, 'gigawatts'}
for i = 1, #v do  -- #v 是列表的大小
    print(v[i])  -- 索引从 1 开始!! 太疯狂了!
end

-- 'list'并非真正的类型，v 其实是一个table，
-- 只不过它用连续的整数作为key，可以像list那样去使用。

```

```
-----  
-- 3.1 元表 (metatable) 和元方法 (metamethod)。  
-----
```

```
-- table的元表提供了一种机制，支持类似操作符重载的行为。  
-- 稍后我们会看到元表如何支持类似js prototype的行为。
```

```
f1 = {a = 1, b = 2} -- 表示一个分数 a/b.  
f2 = {a = 2, b = 3}
```

```
-- 这会失败：  
-- s = f1 + f2
```

```
metafraction = {}  
function metafraction.__add(f1, f2)  
    local sum = {}  
    sum.b = f1.b * f2.b  
    sum.a = f1.a * f2.b + f2.a * f1.b  
    return sum  
end
```

```
setmetatable(f1, metafraction)  
setmetatable(f2, metafraction)
```

```
s = f1 + f2 -- 调用在f1的元表上的__add(f1, f2) 方法
```

```
-- f1, f2 没有关于元表的key，这点和js的prototype不一样。  
-- 因此你必须用getmetatable(f1)获取元表。  
-- 元表是一个普通的table，  
-- 元表的key是普通的Lua中的key，例如__add。
```

```
-- 但是下面一行代码会失败，因为s没有元表：  
-- t = s + s  
-- 下面提供的与类相似的模式可以解决这个问题：
```

```
-- 元表的__index 可以重载用于查找的点操作符：  
defaultFavs = {animal = 'gru', food = 'donuts'}  
myFavs = {food = 'pizza'}  
setmetatable(myFavs, {__index = defaultFavs})  
eatenBy = myFavs.animal -- 可以工作！感谢元表
```

```
-- 如果在table中直接查找key失败，会使用  
-- 元表的__index 递归地重试。
```

```
-- __index的值也可以是function(tbl, key)  
-- 这样可以支持自定义查找。
```

```
-- __index、__add等的值，被称为元方法。  
-- 这里是一个table元方法的清单：
```

```
-- __add(a, b)                for a + b
```



```
-- __sub(a, b)          for a - b
-- __mul(a, b)          for a * b
-- __div(a, b)          for a / b
-- __mod(a, b)          for a % b
-- __pow(a, b)          for a ^ b
-- __unm(a)             for -a
-- __concat(a, b)       for a .. b
-- __len(a)             for #a
-- __eq(a, b)           for a == b
-- __lt(a, b)           for a < b
-- __le(a, b)           for a <= b
-- __index(a, b) <fn or a table> for a.b
-- __newindex(a, b, c)  for a.b = c
-- __call(a, ...)       for a(...)
```

-- 3.2 与类相似的table和继承。

-- Lua没有内建的类；可以通过不同的方法，利用表和元表
-- 来实现类。

-- 下面是一个例子，解释在后面：

```
Dog = {} -- 1.

function Dog:new() -- 2.
    local newObj = {sound = 'woof'} -- 3.
    self.__index = self -- 4.
    return setmetatable(newObj, self) -- 5.
end

function Dog:makeSound() -- 6.
    print('I say ' .. self.sound)
end

mrDog = Dog:new() -- 7.
mrDog:makeSound() -- 'I say woof' -- 8.
```

-- 1. Dog看上去像一个类；其实它是一个table。

-- 2. 函数tablename:fn(...) 等价于
-- 函数tablename.fn(self, ...)
-- 冒号(:)只是添加了self作为第一个参数。
-- 阅读7 & 8条 了解self变量是如何得到其值的。

-- 3. newObj是类Dog的一个实例。

-- 4. self = 被继承的类。通常self = Dog，不过继承可以改变它。
-- 如果把newObj的元表和__index都设置为self，
-- newObj就可以得到self的函数。

-- 5. 备忘：setmetatable返回其第一个参数。

-- 6. 冒号(:)的作用和第2条一样，不过这里
-- self是一个实例，而不是类

```

-- 7. 等价于Dog.new(Dog)，所以在new()中，self = Dog。
-- 8. 等价于mrDog.makeSound(mrDog); self = mrDog。

-----

-- 继承的例子：

LoudDog = Dog:new()                                -- 1.

function LoudDog:makeSound()
    local s = self.sound .. ' '                    -- 2.
    print(s .. s .. s)
end

seymour = LoudDog:new()                             -- 3.
seymour:makeSound() -- 'woof woof woof'            -- 4.

-- 1. LoudDog获得Dog的方法和变量列表。
-- 2. 因为new()的缘故，self拥有了一个'sound' key，参见第3条。
-- 3. 等价于LoudDog.new(LoudDog)，转换一下就是
--     Dog.new(LoudDog)，这是因为LoudDog没有'new' key，
--     但是它的元表中有 __index = Dog。
--     结果：seymour的元表是LoudDog，并且
--     LoudDog.__index = Dog。所以有seymour.key
--     = seymour.key, LoudDog.key, Dog.key
--     从其中第一个有指定key的table获取。
-- 4. 在LoudDog可以找到'makeSound'的key；
--     等价于LoudDog.makeSound(seymour)。

-- 如果有必要，子类也可以有new()，与基类相似：
function LoudDog:new()
    local newObj = {}
    -- 初始化newObj
    self.__index = self
    return setmetatable(newObj, self)
end

-----

-- 4. 模块

-----

--[[ 我把这部分给注释了，这样脚本剩下的部分可以运行

```

```

-- 假设文件mod.lua的内容类似这样:
local M = {}

local function sayMyName()
    print('Hrunkner')
end

function M.sayHello()
    print('Why hello there')
    sayMyName()
end

return M

-- 另一个文件可以使用mod.lua的功能:
local mod = require('mod') -- 运行文件mod.lua.

-- require是包含模块的标准做法。
-- require等价于:      (针对没有被缓存的情况; 参见后面的内容)
local mod = (function ()
    <contents of mod.lua>
end)()
-- mod.lua被包在一个函数体中, 因此mod.lua的局部变量
-- 对外不可见。

-- 下面的代码可以工作, 因为在这里mod = mod.lua 中的 M:
mod.sayHello() -- Says hello to Hrunkner.

-- 这是错误的; sayMyName只在mod.lua中存在:
mod.sayMyName() -- 错误

-- require返回的值会被缓存, 所以一个文件只会被运行一次,
-- 即使它被require了多次。

-- 假设mod2.lua包含代码"print('Hi!')".
local a = require('mod2') -- 打印Hi!
local b = require('mod2') -- 不再打印; a=b.

-- dofile与require类似, 但是不缓存:
dofile('mod2') --> Hi!
dofile('mod2') --> Hi! (再次运行, 与require不同)

-- loadfile加载一个lua文件, 但是并不运行它。
f = loadfile('mod2') -- Calling f() runs mod2.lua.

-- loadstring是loadfile的字符串版本。
g = loadstring('print(343)') -- 返回一个函数。
g() -- 打印343; 在此之前什么也不打印。

--]]

```

参考

为什么？我非常兴奋地学习lua，这样我就可以使用[Löve 2D游戏引擎](#)来编游戏。

怎么做？我从[BlackBulletIV](#)的面向程序员的[Lua指南](#)入门。接着我阅读了官方的[Lua编程](#)一书。

[lua-users.org](#)上的[Lua简明参考](#)应该值得一看。

本文没有涉及标准库的内容：

- [string library](#)
- [table library](#)
- [math library](#)
- [io library](#)
- [os library](#)

顺便说一下，整个文件是可运行的Lua; 保存为 `learn-cn.lua` 用命令 `lua learn-cn.lua` 启动吧！

本文首次撰写于 [tylernelon.com](#) 同时也有 [github gist](#) 版.

使用Lua，欢乐常在！

language: markdown contributors:

```
- ["Dan Turkel", "http://danturkel.com/"]
```

translators:

```
- ["Fangzhou Chen", "https://github.com/FZSS"]
```

filename: learnmarkdown-cn.md

lang: zh-cn

Markdown 由 John Gruber 于 2004 年创立. 它旨在成为一门容易读写的语法结构, 并可以便利地转换成 HTML (以及其他很多) 格式。

欢迎您多多反馈以及分支和请求合并。

```
<!-- Markdown 是 HTML 的父集, 所以任何 HTML 文件都是有效的 Markdown。
这意味着我们可以在 Markdown 里使用任何 HTML 元素, 比如注释元素,
且不会被 Markdown 解析器所影响。不过如果你在 Markdown 文件内创建了 HTML 元素,
你将无法在 HTML 元素的内容中使用 Markdown 语法。-->
```

```
<!-- 在不同的解析器中, Markdown 的实现方法有所不同。
此教程会指出当某功能是否通用及是否只对某一解析器有效。 -->
```

```
<!-- 标头 -->
<!-- 通过在文本前加上不同数量的hash(#), 你可以创建相对应的 <h1>
到 <h6> HTML元素。-->
```

```
# 这是一个 <h1>
## 这是一个 <h2>
### 这是一个 <h3>
#### 这是一个 <h4>
##### 这是一个 <h5>
##### 这是一个 <h6>
```

```
<!-- 对于 <h1> 和 <h2> 元素, Markdown 额外提供了两种添加方式。 -->
这是一个 h1
=====

这是一个 h2
-----
```

```
<!-- 简易文本样式 -->
<!-- 文本的斜体, 粗体, 和删除线在 Markdown 中可以轻易地被实现。-->
```

此文本为斜体。

此文本也是。

此文本为粗体。

__此文本也是__

此文本是斜体加粗体。

或者这样。

__这个也是! __

<!-- 在 GitHub 采用的 Markdown 中 -->

~~此文本为删除线效果。~~

<!-- 单个段落由一句或多句邻近的句子组成，这些句子由一个或多个空格分隔。-->

这是第一段落。这句话在同一个段落里，好玩么？

现在我是第二段落。

这句话也在第二段落！

这句话在第三段落！

<!-- 如果你插入一个 HTML中的
标签，你可以在段末加入两个以上的空格，然后另起一段。-->

此段落结尾有两个空格（选中以显示）。

上文有一个
 ！

<!-- 段落引用可由 > 字符轻松实现。-->

> 这是一个段落引用。你可以

> 手动断开你的句子，然后在每句句子前面添加 “>” 字符。或者让你的句子变得很长，以至于他们自动得断开。

> 只要你的文字以“>” 字符开头，两种方式无异。

> 你也对文本进行

>> 多层引用

> 这多机智啊！

<!-- 序列 -->

<!-- 无序序列可由星号，加号或者减号来建立 -->

* 项目

* 项目

* 另一个项目

或者

+ 项目

+ 项目

+ 另一个项目

或者

- 项目
- 项目
- 最后一个项目

<!-- 有序序列可由数字加点来实现 -->

1. 项目一
2. 项目二
3. 项目三

<!-- 即使你的标签数字有误，Markdown 依旧会呈现出正确的序号，不过这并不是一个好主意-->

1. 项目一
1. 项目二
1. 项目三

<!-- （此段与前例一模一样） -->

<!-- 你也可以使用子序列 -->

1. 项目一
2. 项目二
3. 项目三
 - * 子项目
 - * 子项目
4. 项目四

<!-- 代码段落 -->

<!-- 代码段落（HTML中 <code>标签）可以由缩进四格（spaces）或者一个制表符（tab）实现-->

```
This is code
So is this
```

<!-- 在你的代码中，你仍然使用tab可以进行缩进操作 -->

```
my_array.each do |item|
  puts item
end
```

<!-- 内联代码可由反引号 ` 实现 -->

John 甚至不知道 `go_to()` 方程是干嘛的！

<!-- 在GitHub的 Markdown中，对于代码你可以使用特殊的语法 -->

\\\`ruby <!-- 插入时记得移除反斜线， 仅留\\\`ruby ! -->

```
def foobar
  puts "Hello world!"
end
\\`\\`\\` <!-- 这里也是，移除反斜线，仅留 `` -->

<!-- 以上代码不需要缩进，而且 GitHub 会根据``后表明语言来进行语法高亮 -->

<!-- 水平线 (<hr />) -->
<!-- 水平线可由三个或以上的星号或者减号创建，可带可不带空格。 -->

***
---
- - -
*****

<!-- 链接 -->
<!-- Markdown 最棒的地方就是简易的链接制作。链接文字放在中括号[]内，
在随后的括弧()内加入url。-->

[点我点我!](http://test.com/)

<!-- 你也可以为链接加入一个标题：在括弧内使用引号 -->

[点我点我!](http://test.com/ "连接到Test.com")

<!-- 相对路径也可以有 -->

[去 music](/music/).

<!-- Markdown同样支持引用样式的链接 -->

[点此链接][link1]以获取更多信息！
[看一看这个链接][foobar] 如果你愿意的话。

[link1]: http://test.com/ "Cool!"
[foobar]: http://foobar.biz/ "Alright!"

<!-- 链接的标题可以处于单引号中，括弧中或是被忽略。引用名可以在文档的任意何处，
并且可以随意命名，只要名称不重复。-->

<!-- “隐含式命名”的功能可以让链接文字作为引用名 -->

[This][] is a link.

[this]: http://thisisalink.com/

<!-- 但这并不常用 -->

<!-- 图像 -->
<!-- 图像与链接相似，只需在前添加一个感叹号 -->
```



```
![这是我图像的悬停文本(alt text)](http://imgur.com/myimage.jpg "可选命名")
```

```
<!-- 引用样式也同样起作用 -->
```

```
![这是我的悬停文本.][myimage]
```

```
[myimage]: relative/urls/cool/image.jpg "在此输入标题"
```

```
<!-- 杂项 -->
```

```
<!-- 自动链接 -->
```

```
<http://testwebsite.com/> 与  
[http://testwebsite.com/](http://testwebsite.com/) 等同
```

```
<!-- 电子邮件的自动链接 -->
```

```
<foo@bar.com>
```

```
<!-- 转义字符 -->
```

我希望 *将这段文字置于星号之间* 但是我不希望它被
斜体化，所以我就：*这段置文字于星号之间*。

```
<!-- 表格 -->
```

```
<!-- 表格只被 GitHub 的 Markdown 支持，并且有一点笨重，但如果你真的要用的话： -->
```

第一列	第二列	第三列	
:-----	:-----:	-----:	
左对齐	居个中	右对齐	
某某某	某某某	某某某	

```
<!-- 或者，同样的 -->
```

第一列		第二列		第三列
:--		:-:		--:
这太丑了		药不能		停

```
<!-- 结束! -->
```

更多信息, 请于[此处](#)参见 John Gruber 关于语法的官方帖子, 及于[此处](#) 参见 Adam Pritchard 的摘要笔记。

language: Matlab contributors:

```
- ["mendozao", "http://github.com/mendozao"]  
- ["jamesscottbrown", "http://jamesscottbrown.com"]
```

translators:

```
- ["sunxb10", "https://github.com/sunxb10"]
```

lang: zh-cn

MATLAB 是 MATrix LABoratory（矩阵实验室）的缩写，它是一种功能强大的数值计算语言，在工程和数学领域中应用广泛。

如果您有任何需要反馈或交流的内容，请联系本教程作者 [@the_ozinator](#)、osvaldo.t.mendoza@gmail.com。

```
% 以百分号作为注释符  
  
%{  
多行注释  
可以  
这样  
表示  
%}  
  
% 指令可以随意跨行，但需要在跨行处用 '...' 标明：  
a = 1 + 2 + ...  
+ 4  
  
% 可以在MATLAB中直接向操作系统发出指令  
!ping google.com  
  
who % 显示内存中的所有变量  
whos % 显示内存中的所有变量以及它们的类型  
clear % 清除内存中的所有变量  
clear('A') % 清除指定的变量  
openvar('A') % 在变量编辑器中编辑指定变量  
  
clc % 清除命令窗口中显示的所有指令  
diary % 将命令窗口中的内容写入本地文件  
ctrl-c % 终止当前计算  
  
edit('myfunction.m') % 在编辑器中打开指定函数或脚本  
type('myfunction.m') % 在命令窗口中打印指定函数或脚本的源码
```

```

profile on % 打开 profile 代码分析工具
profile off % 关闭 profile 代码分析工具
profile viewer % 查看 profile 代码分析工具的分析结果

help command % 在命令窗口中显示指定命令的帮助文档
doc command % 在帮助窗口中显示指定命令的帮助文档
lookfor command % 在所有 MATLAB 内置函数的头部注释块的第一行中搜索指定命令
lookfor command -all % 在所有 MATLAB 内置函数的整个头部注释块中搜索指定命令

% 输出格式
format short % 浮点数保留 4 位小数
format long % 浮点数保留 15 位小数
format bank % 金融格式, 浮点数只保留 2 位小数
fprintf('text') % 在命令窗口中显示 "text"
disp('text') % 在命令窗口中显示 "text"

% 变量与表达式
myVariable = 4 % 命令窗口中将新创建的变量
myVariable = 4; % 加上分号可使命令窗口中不显示当前语句执行结果
4 + 6 % ans = 10
8 * myVariable % ans = 32
2 ^ 3 % ans = 8
a = 2; b = 3;
c = exp(a)*sin(pi/2) % c = 7.3891

% 调用函数有两种方式:
% 标准函数语法:
load('myFile.mat', 'y') % 参数放在括号内, 以英文逗号分隔
% 指令语法:
load myFile.mat y % 不加括号, 以空格分隔参数
% 注意在指令语法中参数不需要加引号: 在这种语法下, 所有输入参数都只能是文本文字,
% 不能是变量的具体值, 同样也不能是输出变量
[V,D] = eig(A); % 这条函数调用无法转换成等价的指令语法
[~,D] = eig(A); % 如果结果中只需要 D 而不需要 V 则可以这样写

% 逻辑运算
1 > 5 % 假, ans = 0
10 >= 10 % 真, ans = 1
3 ~= 4 % 不等于 -> ans = 1
3 == 3 % 等于 -> ans = 1
3 > 1 && 4 > 1 % 与 -> ans = 1
3 > 1 || 4 > 1 % 或 -> ans = 1
~1 % 非 -> ans = 0

% 逻辑运算可直接应用于矩阵, 运算结果也是矩阵
A > 5

```

% 对矩阵中每个元素做逻辑运算，若为真，则在运算结果的矩阵中对应位置的元素就是 1

A(A > 5)

% 如此返回的向量，其元素就是 A 矩阵中所有逻辑运算为真的元素

% 字符串

a = 'MyString'

length(a) % ans = 8

a(2) % ans = y

[a,a] % ans = MyStringMyString

b = '字符串' % MATLAB目前已经可以支持包括中文在内的多种文字

length(b) % ans = 3

b(2) % ans = 符

[b,b] % ans = 字符串字符串

% 元组 (cell 数组)

a = {'one', 'two', 'three'}

a(1) % ans = 'one' - 返回一个元组

char(a(1)) % ans = one - 返回一个字符串

% 结构体

A.b = {'one', 'two'};

A.c = [1 2];

A.d.e = false;

% 向量

x = [4 32 53 7 1]

x(2) % ans = 32, MATLAB中向量的下标索引从1开始，不是0

x(2:3) % ans = 32 53

x(2:end) % ans = 32 53 7 1

x = [4; 32; 53; 7; 1] % 列向量

x = [1:10] % x = 1 2 3 4 5 6 7 8 9 10

% 矩阵

A = [1 2 3; 4 5 6; 7 8 9]

% 以分号分隔不同的行，以空格或逗号分隔同一行中的不同元素

% A =

% 1 2 3

% 4 5 6

% 7 8 9

A(2,3) % ans = 6, A(row, column)

A(6) % ans = 8

% (隐式地将 A 的三列首尾相接组成一个列向量，然后取其下标为 6 的元素)

```
A(2,3) = 42 % 将第 2 行第 3 列的元素设为 42
% A =
```

```
%      1      2      3
%      4      5      42
%      7      8      9
```

```
A(2:3,2:3) % 取原矩阵中的一块作为新矩阵
%ans =
```

```
%      5      42
%      8      9
```

```
A(:,1) % 第 1 列的所有元素
%ans =
```

```
%      1
%      4
%      7
```

```
A(1,:) % 第 1 行的所有元素
%ans =
```

```
%      1      2      3
```

```
[A ; A] % 将两个矩阵上下相接构成新矩阵
%ans =
```

```
%      1      2      3
%      4      5      42
%      7      8      9
%      1      2      3
%      4      5      42
%      7      8      9
```

```
% 等价于
vertcat(A, A);
```

```
[A , A] % 将两个矩阵左右相接构成新矩阵
```

```
%ans =
```

```
%      1      2      3      1      2      3
%      4      5      42     4      5      42
%      7      8      9      7      8      9
```

```
% 等价于
horzcat(A, A);
```

```

A(:, [3 1 2]) % 重新排布原矩阵的各列
%ans =

%      3      1      2
%     42      4      5
%      9      7      8

size(A) % 返回矩阵的行数和列数, ans = 3 3

A(1, :) = [] % 删除矩阵的第 1 行
A(:, 1) = [] % 删除矩阵的第 1 列

transpose(A) % 矩阵转置, 等价于 A'
ctranspose(A) % 矩阵的共轭转置 (对矩阵中的每个元素取共轭复数)

% 元素运算 vs. 矩阵运算
% 单独运算符就是对矩阵整体进行矩阵运算
% 在运算符加上英文句点就是对矩阵中的元素进行元素计算
% 示例如下:
A * B % 矩阵乘法, 要求 A 的列数等于 B 的行数
A .* B % 元素乘法, 要求 A 和 B 形状一致 (A 的行数等于 B 的行数, A 的列数等于 B 的列数)
% 元素乘法的结果是与 A 和 B 形状一致的矩阵, 其每个元素等于 A 对应位置的元素乘 B 对应位置的元素

% 以下函数中, 函数名以 m 结尾的执行矩阵运算, 其余执行元素运算:
exp(A) % 对矩阵中每个元素做指数运算
expm(A) % 对矩阵整体做指数运算
sqrt(A) % 对矩阵中每个元素做开方运算
sqrtm(A) % 对矩阵整体做开方运算 (即试图求出一个矩阵, 该矩阵与自身的乘积等于 A 矩阵)

% 绘图
x = 0:0.1:2*pi; % 生成一向量, 其元素从 0 开始, 以 0.1 的间隔一直递增至 2*pi (pi 就是圆周率)
y = sin(x);
plot(x,y)
xlabel('x axis')
ylabel('y axis')
title('Plot of y = sin(x)')
axis([0 2*pi -1 1]) % x 轴范围是从 0 到 2*pi, y 轴范围是从 -1 到 1

plot(x,y1,'-',x,y2,'--',x,y3,':') % 在同一张图中绘制多条曲线
legend('Line 1 label', 'Line 2 label') % 为图片加注图例
% 图例数量应当小于或等于实际绘制的曲线数目, 从 plot 绘制的第一条曲线开始对应

% 在同一张图上绘制多条曲线的另一种方法:
% 使用 hold on, 令系统保留前次绘图结果并在其上直接叠加新的曲线,
% 如果没有 hold on, 则每个 plot 都会首先清除之前的绘图结果再进行绘制。
% 在 hold on 和 hold off 中可以放置任意多的 plot 指令,
% 它们和 hold on 前最后一个 plot 指令的结果都将显示在同一张图中。
plot(x, y1)

```

```

hold on
plot(x, y2)
plot(x, y3)
plot(x, y4)
hold off

loglog(x, y) % 对数-对数绘图
semilogx(x, y) % 半对数 (x 轴对数) 绘图
semilogy(x, y) % 半对数 (y 轴对数) 绘图

fplot (@(x) x^2, [2,5]) % 绘制函数 x^2 在 [2, 5] 区间的曲线

grid on % 在绘制的图中显示网格, 使用 grid off 可取消网格显示
axis square % 将当前坐标系设定为正方形 (保证在图形显示上各轴等长)
axis equal % 将当前坐标系设定为相等 (保证在实际数值上各轴等长)

scatter(x, y); % 散点图
hist(x); % 直方图

z = sin(x);
plot3(x,y,z); % 绘制三维曲线

pcolor(A) % 伪彩色图 (热图)
contour(A) % 等高线图
mesh(A) % 网格曲面图

h = figure % 创建新的图片对象并返回其句柄 h
figure(h) % 将句柄 h 对应的图片作为当前图片
close(h) % 关闭句柄 h 对应的图片
close all % 关闭 MATLAB 中所用打开的图片
close % 关闭当前图片

shg % 显示图形窗口
clf clear % 清除图形窗口中的图像, 并重置图像属性

% 图像属性可以通过图像句柄进行设定
% 在创建图像时可以保存图像句柄以便于设置
% 也可以用 gcf 函数返回当前图像的句柄
h = plot(x, y); % 在创建图像时显式地保存图像句柄
set(h, 'Color', 'r')
% 颜色代码: 'y' 黄色, 'm' 洋红色, 'c' 青色, 'r' 红色, 'g' 绿色, 'b' 蓝色, 'w' 白色, 'k' 黑色
set(h, 'Color', [0.5, 0.5, 0.4])
% 也可以使用 RGB 值指定颜色
set(h, 'LineStyle', '--')
% 线型代码: '--' 实线, '---' 虚线, ':' 点线, '-.' 点划线, 'none' 不划线
get(h, 'LineStyle')
% 获取当前句柄的线型

% 用 gca 函数返回当前图像的坐标轴句柄
set(gca, 'XDir', 'reverse'); % 令 x 轴反向

```

```

% 用 subplot 指令创建平铺排列的多张子图
subplot(2,3,1); % 选择 2 x 3 排列的子图中的第 1 张图
plot(x1); title('First Plot') % 在选中的图中绘图
subplot(2,3,2); % 选择 2 x 3 排列的子图中的第 2 张图
plot(x2); title('Second Plot') % 在选中的图中绘图


% 要调用函数或脚本，必须保证它们在你的当前工作目录中
path % 显示当前工作目录
addpath /path/to/dir % 将指定路径加入到当前工作目录中
rmpath /path/to/dir % 将指定路径从当前工作目录中删除
cd /path/to/move/into % 以制定路径作为当前工作目录


% 变量可保存到 .mat 格式的本地文件
save('myFileName.mat') % 保存当前工作空间中的所有变量
load('myFileName.mat') % 将指定文件中的变量载入到当前工作空间


% .m 脚本文件
% 脚本文件是一个包含多条 MATLAB 指令的外部文件，以 .m 为后缀名
% 使用脚本文件可以避免在命令窗口中重复输入冗长的指令


% .m 函数文件
% 与脚本文件类似，同样以 .m 作为后缀名
% 但函数文件可以接受用户输入的参数并返回运算结果
% 并且函数拥有自己的工作空间（变量域），不必担心变量名称冲突
% 函数文件的名称应当与其所定义的函数的名称一致（比如下面例子中函数文件就应命名为 double_input.m）
% 使用 'help double_input.m' 可返回函数定义中第一行注释信息
function output = double_input(x)
    % double_input(x) 返回 x 的 2 倍
    output = 2*x;
end
double_input(6) % ans = 12


% 同样还可以定义子函数和内嵌函数
% 子函数与主函数放在同一个函数文件中，且只能被这个主函数调用
% 内嵌函数放在另一个函数体内，可以直接访问被嵌套函数的各个变量


% 使用匿名函数可以不必创建 .m 函数文件
% 匿名函数适用于快速定义某函数以便传递给另一指令或函数（如绘图、积分、求根、求极值等）
% 下面示例的匿名函数返回输入参数的平方根，可以使用句柄 sqr 进行调用：
sqr = @(x) x.^2;
sqr(10) % ans = 100
doc function_handle % find out more

```



```

% 接受用户输入
a = input('Enter the value: ')

% 从文件中读取数据
fopen(filename)
% 类似函数还有 xlsread (excel 文件)、importdata (CSV 文件)、imread (图像文件)

% 输出
disp(a) % 在命令窗口中打印变量 a 的值
disp('Hello World') % 在命令窗口中打印字符串
fprintf % 按照指定格式在命令窗口中打印内容

% 条件语句 (if 和 elseif 语句中的括号并非必需, 但推荐加括号避免混淆)
if (a > 15)
    disp('Greater than 15')
elseif (a == 23)
    disp('a is 23')
else
    disp('neither condition met')
end

% 循环语句
% 注意: 对向量或矩阵使用循环语句进行元素遍历的效率很低!!
% 注意: 只要有可能, 就尽量使用向量或矩阵的整体运算取代逐元素循环遍历!!
% MATLAB 在开发时对向量和矩阵运算做了专门优化, 做向量和矩阵整体运算的效率高于循环语句
for k = 1:5
    disp(k)
end

k = 0;
while (k < 5)
    k = k + 1;
end

% 程序运行计时: 'tic' 是计时开始, 'toc' 是计时结束并打印结果
tic
A = rand(1000);
A*A*A*A*A*A*A;
toc

% 链接 MySQL 数据库
dbname = 'database_name';
username = 'root';
password = 'root';
driver = 'com.mysql.jdbc.Driver';
dburl = ['jdbc:mysql://localhost:8889/' dbname];
javaclasspath('mysql-connector-java-5.1.xx-bin.jar'); % 此处 xx 代表具体版本号

```

```

% 这里的 mysql-connector-java-5.1.xx-bin.jar 可从 http://dev.mysql.com/downloads/connecto
conn = database(dbname, username, password, driver, dburl);
sql = ['SELECT * from table_name where id = 22'] % SQL 语句
a = fetch(conn, sql) % a 即包含所需数据


% 常用数学函数
sin(x)
cos(x)
tan(x)
asin(x)
acos(x)
atan(x)
exp(x)
sqrt(x)
log(x)
log10(x)
abs(x)
min(x)
max(x)
ceil(x)
floor(x)
round(x)
rem(x)
rand % 均匀分布的伪随机浮点数
randi % 均匀分布的伪随机整数
randn % 正态分布的伪随机浮点数


% 常用常数
pi
NaN
inf


% 求解矩阵方程（如果方程无解，则返回最小二乘近似解）
% \ 操作符等价于 mldivide 函数，/ 操作符等价于 mrdivide 函数
x=A\b % 求解 Ax=b，比先求逆再左乘 inv(A)*b 更加高效、准确
x=b/A % 求解 xA=b


inv(A) % 逆矩阵
pinv(A) % 伪逆矩阵


% 常用矩阵函数
zeros(m, n) % m x n 阶矩阵，元素全为 0
ones(m, n) % m x n 阶矩阵，元素全为 1
diag(A) % 返回矩阵 A 的对角线元素
diag(x) % 构造一个对角阵，对角线元素就是向量 x 的各元素
eye(m, n) % m x n 阶单位矩阵
linspace(x1, x2, n) % 返回介于 x1 和 x2 之间的 n 个等距节点
inv(A) % 矩阵 A 的逆矩阵
det(A) % 矩阵 A 的行列式

```

```
eig(A) % 矩阵 A 的特征值和特征向量
trace(A) % 矩阵 A 的迹（即对角线元素之和），等价于 sum(diag(A))
isempty(A) % 测试 A 是否为空
all(A) % 测试 A 中所有元素是否都非 0 或都为真（逻辑值）
any(A) % 测试 A 中是否有元素非 0 或为真（逻辑值）
isequal(A, B) % 测试 A 和 B 是否相等
numel(A) % 矩阵 A 的元素个数
triu(x) % 返回 x 的上三角这部分
tril(x) % 返回 x 的下三角这部分
cross(A, B) % 返回 A 和 B 的叉积（矢量积、外积）
dot(A, B) % 返回 A 和 B 的点积（数量积、内积），要求 A 和 B 必须等长
transpose(A) % A 的转置，等价于 A'
fliplr(A) % 将一个矩阵左右翻转
flipud(A) % 将一个矩阵上下翻转

% 矩阵分解
[L, U, P] = lu(A) % LU 分解: PA = LU, L 是下三角阵, U 是上三角阵, P 是置换阵
[P, D] = eig(A) % 特征值分解: AP = PD, D 是由特征值构成的对角阵, P 的各列就是对应的特征向量
[U, S, V] = svd(X) % 奇异值分解: XV = US, U 和 V 是酉矩阵, S 是由奇异值构成的半正定实数对角阵

% 常用向量函数
max % 最大值
min % 最小值
length % 元素个数
sort % 按升序排列
sum % 各元素之和
prod % 各元素之积
mode % 众数
median % 中位数
mean % 平均值
std % 标准差
perms(x) % x 元素的全排列
```

相关资料

- 官方网页: <http://www.mathworks.com/products/matlab/>
- 官方论坛: <http://www.mathworks.com/matlabcentral/answers/>

name: perl category: language language: perl filename: learnperl-cn.pl contributors:

```
- ["Korjavin Ivan", "http://github.com/korjavin"]
```

translators:

```
- ["Yadong Wen", "https://github.com/yadongwen"]
```

lang: zh-cn

Perl 5是一个功能强大、特性齐全的编程语言，有25年的历史。

Perl 5可以在包括便携式设备和大型机的超过100个平台上运行，既适用于快速原型构建，也适用于大型项目开发。

```
# 单行注释以#号开头

#### Perl的变量类型

# 变量以$号开头。
# 合法变量名以英文字母或者下划线起始，
# 后接任意数目的字母、数字或下划线。

### Perl有三种主要的变量类型：标量、数组和哈希。

## 标量
# 标量类型代表单个值：
my $animal = "camel";
my $answer = 42;

# 标量类型值可以是字符串、整型或浮点类型，Perl会根据需要自动进行类型转换。

## 数组
# 数组类型代表一系列值：
my @animals = ("camel", "llama", "owl");
my @numbers = (23, 42, 69);
my @mixed   = ("camel", 42, 1.23);

## 哈希
# 哈希类型代表一个键/值对的集合：

my %fruit_color = ("apple", "red", "banana", "yellow");
```

可以使用空格和“=>”操作符更清晰的定义哈希：

```
my %fruit_color = (  
    apple => "red",  
    banana => "yellow",  
);
```

perldata中有标量、数组和哈希更详细的介绍。（perldoc perldata）。

可以用引用构建更复杂的数据类型，比如嵌套的列表和哈希。

逻辑和循环结构

Perl有大多数常见的逻辑和循环控制结构

```
if ( $var ) {  
    ...  
} elsif ( $var eq 'bar' ) {  
    ...  
} else {  
    ...  
}
```

```
unless ( condition ) {  
    ...  
}
```

上面这个比"if (!condition)"更可读。

有Perl特色的后置逻辑结构

```
print "Yow!" if $zippy;  
print "We have no bananas" unless $bananas;
```

```
# while  
while ( condition ) {  
    ...  
}
```

for和foreach

```
for ( $i = 0; $i <= $max; $i++ ) {  
    ...  
}
```

```
foreach (@array) {  
    print "This element is $_\n";  
}
```

正则表达式

Perl对正则表达式有深入广泛的支持，perlrequick和perlretut等文档有详细介绍。简单来说：

```

# 简单匹配
if (/foo/)      { ... } # 如果 $_ 包含"foo"逻辑为真
if ($a =~ /foo/) { ... } # 如果 $a 包含"foo"逻辑为真

# 简单替换

$a =~ s/foo/bar/;      # 将$a中的foo替换为bar
$a =~ s/foo/bar/g;     # 将$a中所有的foo替换为bar

#### 文件和输入输出

# 可以使用“open()”函数打开文件用于输入输出。

open(my $in, "<", "input.txt") or die "Can't open input.txt: $!";
open(my $out, ">", "output.txt") or die "Can't open output.txt: $!";
open(my $log, ">>", "my.log")      or die "Can't open my.log: $!";

# 可以用"<>"操作符读取一个打开的文件句柄。 在标量语境下会读取一行，
# 在列表语境下会将整个文件读入并将每一行赋给列表的一个元素：

my $line = <$in>;
my @lines = <$in>;

#### 子程序

# 写子程序很简单：

sub logger {
    my $logmessage = shift;
    open my $logfile, ">>", "my.log" or die "Could not open my.log: $!";
    print $logfile $logmessage;
}

# 现在可以像内置函数一样调用子程序：

logger("We have a logger subroutine!");

```

使用Perl模块

Perl模块提供一系列特性来帮助你避免重新发明轮子，CPAN是下载模块的好地方(<http://www.cpan.org/>)。Perl发行版本本身也包含很多流行的模块。

perlfaq有很多常见问题和相应回答，也经常有对优秀CPAN模块的推荐介绍。

深入阅读

- [perl-tutorial](http://perl-tutorial.org/)
- [www.perl.com的learn站点](http://www.perl.org/learn.html)
- [perldoc](http://perldoc.perl.org/)
- 以及 perl 内置的: ``perldoc perlintro``

language: PHP contributors:

```
- ["Malcolm Fell", "http://emarref.net/"]  
- ["Trismegiste", "https://github.com/Trismegiste"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net"]
```

filename: learnphp-zh.php

lang: zh-cn

这份教程所使用的版本是 PHP 5+.

```
<?php // PHP必须被包围于 <?php ? > 之中  
  
// 如果你的文件中只有php代码，那么最好省略结束括号标记  
  
// 这是单行注释的标志  
  
# 井号也可以，但是//更常见  
  
/*  
    这是多行注释  
*/  
  
// 使用 "echo" 或者 "print" 来输出信息到标准输出  
print('Hello '); // 输出 "Hello " 并且没有换行符  
  
// () 对于echo和print是可选的  
echo "World\n"; // 输出 "World" 并且换行  
// (每个语句必须以分号结尾)  
  
// 在 <?php 标签之外的语句都被自动输出到标准输出  
?>Hello World Again!  
<?php  
  
/*****  
 * 类型与变量  
 */  
  
// 变量以$开始  
// 变量可以以字母或者下划线开头，后面可以跟着数字、字母和下划线
```



```

// 布尔值是大小写无关的
$boolean = true; // 或 TRUE 或 True
$boolean = false; // 或 FALSE 或 False

// 整型
$int1 = 12; // => 12
$int2 = -12; // => -12
$int3 = 012; // => 10 (0开头代表八进制数)
$int4 = 0x0F; // => 15 (0x开头代表十六进制数)

// 浮点型 (即双精度浮点型)
$float = 1.234;
$float = 1.2e3;
$float = 7E-10;

// 算数运算
$sum = 1 + 1; // 2
$difference = 2 - 1; // 1
$product = 2 * 2; // 4
$quotient = 2 / 1; // 2

// 算数运算的简写
$number = 0;
$number += 1; // $number 自增1
echo $number++; // 输出1 (运算后自增)
echo ++$number; // 输出3 (自增后运算)
$number /= $float; // 先除后赋值给 $number

// 字符串需要被包含在单引号之中
$sgl_quotes = '$String'; // => '$String'

// 如果需要在字符串中引用变量，就需要使用双引号
$dbl_quotes = "This is a $sgl_quotes."; // => 'This is a $String.'

// 特殊字符只有在双引号中有用
$escaped = "This contains a \t tab character.";
$unescaped = 'This just contains a slash and a t: \t';

// 可以把变量包含在一对大括号中
$money = "I have ${number} in the bank.";

// 自 PHP 5.3 开始, nowdocs 可以被用作多行非计算型字符串
$nowdoc = <<<'END'
Multi line
string
END;

// 而Heredocs则可以用作多行计算型字符串
$heredoc = <<<END
Multi line
$sgl_quotes

```

```

END;

// 字符串需要用 . 来连接
echo 'This string ' . 'is concatenated';

/*****
 * 数组
 */

// PHP 中的数组都是关联型数组，也就是某些语言中的哈希表或字典

// 在所有PHP版本中均适用：
$associative = array('One' => 1, 'Two' => 2, 'Three' => 3);

// PHP 5.4 中引入了新的语法
$associative = ['One' => 1, 'Two' => 2, 'Three' => 3];

echo $associative['One']; // 输出 1

// 声明为列表实际上是给每个值都分配了一个整数键（key）
$array = ['One', 'Two', 'Three'];
echo $array[0]; // => "One"

/*****
 * 输出
 */

echo('Hello World!');
// 输出到标准输出
// 此时标准输出就是浏览器中的网页

print('Hello World!'); // 和echo相同

// echo和print实际上也属于这个语言本身，所以我们省略括号
echo 'Hello World!';
print 'Hello World!';

$paragraph = 'paragraph';

echo 100; // 直接输出标量
echo $paragraph; // 或者输出变量

// 如果你配置了短标签，或者使用5.4.0及以上的版本
// 你就可以使用简写的echo语法
?>
<p><?= $paragraph ?></p>
<?php

$x = 1;

```

```

$y = 2;
$x = $y; // $x 现在和 $y 的值相同
$z = &$y;
// $z 现在持有 $y 的引用。现在更改 $z 的值也会更改 $y 的值，反之亦然
// 但是改变 $y 的值不会改变 $x 的值

echo $x; // => 2
echo $z; // => 2
$y = 0;
echo $x; // => 2
echo $z; // => 0

/*****
 * 逻辑
 */
$a = 0;
$b = '0';
$c = '1';
$d = '1';

// 如果assert的参数为假，就会抛出警告

// 下面的比较都为真，不管它们的类型是否匹配
assert($a == $b); // 相等
assert($c != $a); // 不等
assert($c <> $a); // 另一种不等的表示
assert($a < $c);
assert($c > $b);
assert($a <= $b);
assert($c >= $d);

// 下面的比较只有在类型相同、值相同的情况下才为真
assert($c === $d);
assert($a !== $d);
assert(1 === '1');
assert(1 !== '1');

// 变量可以根据其使用来进行类型转换

$integer = 1;
echo $integer + $integer; // => 2

$string = '1';
echo $string + $string; // => 2 (字符串在此时被转化为整数)

$string = 'one';
echo $string + $string; // => 0
// 输出0，因为'one'这个字符串无法被转换为整数

// 类型转换可以将一个类型视作另一种类型

```

```

$boolean = (boolean) 1; // => true

$zero = 0;
$boolean = (boolean) $zero; // => false

// 还有一些专用的函数来进行类型转换
$integer = 5;
$string = strval($integer);

$var = null; // 空值

/*****
 * 控制结构
 */

if (true) {
    print 'I get printed';
}

if (false) {
    print 'I don\'t';
} else {
    print 'I get printed';
}

if (false) {
    print 'Does not get printed';
} elseif(true) {
    print 'Does';
}

// 三目运算符
print (false ? 'Does not get printed' : 'Does');

$x = 0;
if ($x === '0') {
    print 'Does not print';
} elseif($x == '1') {
    print 'Does not print';
} else {
    print 'Does print';
}

// 下面的语法常用于模板中:
?>

<?php if ($x): ?>

```

This is displayed if the test is truthy.

```
<?php else: ?>
```

This is displayed otherwise.

```
<?php endif; ?>
```

```
<?php
```

// 用switch来实现相同的逻辑

```
switch ($x) {  
    case '0':  
        print 'Switch does type coercion';  
        break; // 在case中必须使用一个break语句,  
               // 否则在执行完这个语句后会直接执行后面的语句  
    case 'two':  
    case 'three':  
        // 如果$variable是 'two' 或 'three', 执行这里的语句  
        break;  
    default:  
        // 其他情况  
}
```

// While, do...while 和 for 循环

```
$i = 0;  
while ($i < 5) {  
    echo $i++;  
}; // 输出 "01234"
```

```
echo "\n";
```

```
$i = 0;  
do {  
    echo $i++;  
} while ($i < 5); // 输出 "01234"
```

```
echo "\n";
```

```
for ($x = 0; $x < 10; $x++) {  
    echo $x;  
} // 输出 "0123456789"
```

```
echo "\n";
```

```
$wheels = ['bicycle' => 2, 'car' => 4];
```

// Foreach 循环可以遍历数组

```
foreach ($wheels as $wheel_count) {  
    echo $wheel_count;  
} // 输出 "24"
```

```
echo "\n";
```

```
// 也可以同时遍历键和值
foreach ($wheels as $vehicle => $wheel_count) {
    echo "A $vehicle has $wheel_count wheels";
}

echo "\n";

$i = 0;
while ($i < 5) {
    if ($i === 3) {
        break; // 退出循环
    }
    echo $i++;
} // 输出 "012"

for ($i = 0; $i < 5; $i++) {
    if ($i === 3) {
        continue; // 跳过此次遍历
    }
    echo $i;
} // 输出 "0124"

/*****
 * 函数
 */

// 通过"function"定义函数:
function my_function () {
    return 'Hello';
}

echo my_function(); // => "Hello"

// 函数名需要以字母或者下划线开头,
// 后面可以跟着任意的字母、下划线、数字。

function add ($x, $y = 1) { // $y 是可选参数, 默认值为 1
    $result = $x + $y;
    return $result;
}

echo add(4); // => 5
echo add(4, 2); // => 6

// $result 在函数外部不可访问
// print $result; // 抛出警告

// 从 PHP 5.3 起我们可以定义匿名函数
$inc = function ($x) {
    return $x + 1;
}
```

```

};

echo $inc(2); // => 3

function foo ($x, $y, $z) {
    echo "$x - $y - $z";
}

// 函数也可以返回一个函数
function bar ($x, $y) {
    // 用 'use' 将外部的参数引入到里面
    return function ($z) use ($x, $y) {
        foo($x, $y, $z);
    };
}

$bar = bar('A', 'B');
$bar('C'); // 输出 "A - B - C"

// 你也可以通过字符串调用函数
$function_name = 'add';
echo $function_name(1, 2); // => 3
// 在通过程序来决定调用哪个函数时很有用
// 或者，使用 call_user_func(callable $callback [, $parameter [, ... ]]);

/*****
 * 导入
 */

<?php
// 被导入的php文件也必须以php开标签开始

include 'my-file.php';
// 现在my-file.php就在当前作用域中可见了
// 如果这个文件无法被导入（比如文件不存在），会抛出警告

include_once 'my-file.php';
// my-file.php中的代码在其他地方被导入了，那么就不会被再次导入
// 这会避免类的多重定义错误

require 'my-file.php';
require_once 'my-file.php';
// 和include功能相同，只不过如果不能被导入时，会抛出错误

// my-include.php的内容：
<?php

return 'Anything you like.';
// 文件结束

// Include和Require函数也有返回值

```

```

$value = include 'my-include.php';

// 被引入的文件是根据文件路径或者include_path配置来查找到的
// 如果文件最终没有被找到，那么就会查找当前文件夹。之后才会报错
/* */

/*****
 * 类
 */

// 类是由class关键字定义的

class MyClass
{
    const MY_CONST      = 'value'; // 常量

    static $staticVar    = 'static';

    // 属性必须声明其作用域
    public $property      = 'public';
    public $instanceProp;
    protected $prot      = 'protected'; // 当前类和子类可访问
    private $priv        = 'private';    // 仅当前类可访问

    // 通过 __construct 来定义构造函数
    public function __construct($instanceProp) {
        // 通过 $this 访问当前对象
        $this->instanceProp = $instanceProp;
    }

    // 方法就是类中定义的函数
    public function myMethod()
    {
        print 'MyClass';
    }

    final function youCannotOverrideMe()
    {
    }

    public static function myStaticMethod()
    {
        print 'I am static';
    }
}

echo MyClass::MY_CONST;      // 输出 'value';
echo MyClass::$staticVar;    // 输出 'static';
MyClass::myStaticMethod(); // 输出 'I am static';

// 通过new来新建实例

```



```

$my_class = new MyClass('An instance property');
// 如果不传递参数，那么括号可以省略

// 用 -> 来访问成员
echo $my_class->property;      // => "public"
echo $my_class->instanceProp;  // => "An instance property"
$my_class->myMethod();          // => "MyClass"

// 使用extends来生成子类
class MyOtherClass extends MyClass
{
    function printProtectedProperty()
    {
        echo $this->prot;
    }

    // 方法覆盖
    function myMethod()
    {
        parent::myMethod();
        print ' > MyOtherClass';
    }
}

$my_other_class = new MyOtherClass('Instance prop');
$my_other_class->printProtectedProperty(); // => 输出 "protected"
$my_other_class->myMethod();                // 输出 "MyClass > MyOtherClass"

final class YouCannotExtendMe
{
}

// 你可以使用“魔法方法”来生成getter和setter方法
class MyMapClass
{
    private $property;

    public function __get($key)
    {
        return $this->$key;
    }

    public function __set($key, $value)
    {
        $this->$key = $value;
    }
}

$x = new MyMapClass();
echo $x->property; // 会使用 __get() 方法

```

```
$x->property = 'Something'; // 会使用 __set() 方法
```

```
// 类可以是被定义成抽象类（使用 abstract 关键字）或者  
// 去实现接口（使用 implements 关键字）。  
// 接口需要通过interface关键字来定义
```

```
interface InterfaceOne
```

```
{  
    public function doSomething();  
}
```

```
interface InterfaceTwo
```

```
{  
    public function doSomethingElse();  
}
```

```
// 接口可以被扩展
```

```
interface InterfaceThree extends InterfaceTwo
```

```
{  
    public function doAnotherContract();  
}
```

```
abstract class MyAbstractClass implements InterfaceOne
```

```
{  
    public $x = 'doSomething';  
}
```

```
class MyConcreteClass extends MyAbstractClass implements InterfaceTwo
```

```
{  
    public function doSomething()  
    {  
        echo $x;  
    }  
  
    public function doSomethingElse()  
    {  
        echo 'doSomethingElse';  
    }  
}
```

```
// 一个类可以实现多个接口
```

```
class SomeOtherClass implements InterfaceOne, InterfaceTwo
```

```
{  
    public function doSomething()  
    {  
        echo 'doSomething';  
    }  
  
    public function doSomethingElse()  
    {
```

```

        echo 'doSomethingElse';
    }
}

/*****
 * 特征
 */

// 特征 从 PHP 5.4.0 开始包括，需要用 "trait" 这个关键字声明

trait MyTrait
{
    public function myTraitMethod()
    {
        print 'I have MyTrait';
    }
}

class MyTraitfulClass
{
    use MyTrait;
}

$cls = new MyTraitfulClass();
$cls->myTraitMethod(); // 输出 "I have MyTrait"

/*****
 * 命名空间
 */

// 这部分是独立于这个文件的
// 因为命名空间必须在一个文件的开始处。

<?php

// 类会被默认的放在全局命名空间中，可以被一个\来显式调用

$cls = new \MyClass();

// 为一个文件设置一个命名空间
namespace My\Namespace;

class MyClass
{
}

// (或者从其他文件中)

```

```
$cls = new My\Namespace\MyClass;

//或者从其他命名空间中
namespace My\Other\Namespace;

use My\Namespace\MyClass;

$cls = new MyClass();

// 你也可以为命名空间起一个别名

namespace My\Other\Namespace;

use My\Namespace as SomeOtherNamespace;

$cls = new SomeOtherNamespace\MyClass();

*/
```

更多阅读

访问 [PHP 官方文档](#)

如果你对最佳实践感兴趣（实时更新） [PHP The Right Way](#).

如果你很熟悉善于包管理的语言 [Composer](#).

如要了解通用标准，请访问[PHP Framework Interoperability Group's PSR standards](#).

language: python contributors:

```
- ["Louie Dinh", "http://ldinh.ca"]
```

translators:

```
- ["Chenbo Li", "http://binarythink.net"]
```

filename: learnpython-zh.py

lang: zh-cn

Python 由 Guido Van Rossum 在90年代初创建。它现在是最流行的语言之一 我喜爱python是因为它有极为清晰的语法，甚至可以说，它就是可以执行的伪代码

很欢迎来自您的反馈，你可以在@louiedinh 和 [louiedinh \[at\] \[google's email service\]](mailto:louiedinh[at]google's email service) 这里找到我

注意: 这篇文章针对的版本是Python 2.7，但大多也可使用于其他Python 2的版本 如果是Python 3，请在网络上寻找其他教程

```
# 单行注释
""" 多行字符串可以用
    三个引号包裹，不过这也可以被当做
    多行注释
"""

#####
## 1. 原始数据类型和操作符
#####

# 数字类型
3 # => 3

# 简单的算数
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20
35 / 5 # => 7

# 整数的除法会自动取整
5 / 2 # => 2

# 要做精确的除法，我们需要引入浮点数
2.0 # 浮点数
11.0 / 4.0 # => 2.75 精确多了
```

```
# 括号具有最高优先级
(1 + 3) * 2 # => 8

# 布尔值也是基本的数据类型
True
False

# 用 not 来取非
not True # => False
not False # => True

# 相等
1 == 1 # => True
2 == 1 # => False

# 不等
1 != 1 # => False
2 != 1 # => True

# 更多的比较操作符
1 < 10 # => True
1 > 10 # => False
2 <= 2 # => True
2 >= 2 # => True

# 比较运算可以连起来写！
1 < 2 < 3 # => True
2 < 3 < 2 # => False

# 字符串通过 " 或 ' 括起来
"This is a string."
'This is also a string.'

# 字符串通过加号拼接
"Hello " + "world!" # => "Hello world!"

# 字符串可以被视为字符的列表
"This is a string"[0] # => 'T'

# % 可以用来格式化字符串
"%s can be %s" % ("strings", "interpolated")

# 也可以用 format 方法来格式化字符串
# 推荐使用这个方法
"{0} can be {1}".format("strings", "formatted")
# 也可以用变量名代替数字
"{name} wants to eat {food}".format(name="Bob", food="lasagna")

# None 是对象
None # => None
```

```

# 不要用相等 `==` 符号来和None进行比较
# 要用 `is`
"etc" is None # => False
None is None # => True

# 'is' 可以用来比较对象的相等性
# 这个操作符在比较原始数据时没多少用，但是比较对象时必不可少

# None, 0, 和空字符串都被算作 False
# 其他的均为 True
0 == False # => True
"" == False # => True

#####
## 2. 变量和集合
#####

# 很方便的输出
print "I'm Python. Nice to meet you!"

# 给变量赋值前不需要事先声明
some_var = 5 # 一般建议使用小写字母和下划线组合来做为变量名
some_var # => 5

# 访问未赋值的变量会抛出异常
# 可以查看控制流程一节来了解如何异常处理
some_other_var # 抛出 NameError

# if 语句可以作为表达式来使用
"yahoo!" if 3 > 2 else 2 # => "yahoo!"

# 列表用来保存序列
li = []
# 可以直接初始化列表
other_li = [4, 5, 6]

# 在列表末尾添加元素
li.append(1) # li 现在是 [1]
li.append(2) # li 现在是 [1, 2]
li.append(4) # li 现在是 [1, 2, 4]
li.append(3) # li 现在是 [1, 2, 4, 3]
# 移除列表末尾元素
li.pop() # => 3 li 现在是 [1, 2, 4]
# 重新加进去
li.append(3) # li is now [1, 2, 4, 3] again.

# 像其他语言访问数组一样访问列表
li[0] # => 1

```

```
# 访问最后一个元素
li[-1] # => 3

# 越界会抛出异常
li[4] # 抛出越界异常

# 切片语法需要用到列表的索引访问
# 可以看做数学之中左闭右开区间
li[1:3] # => [2, 4]
# 省略开头的元素
li[2:] # => [4, 3]
# 省略末尾的元素
li[:3] # => [1, 2, 4]

# 删除特定元素
del li[2] # li 现在是 [1, 2, 3]

# 合并列表
li + other_li # => [1, 2, 3, 4, 5, 6] - 并不会不改变这两个列表

# 通过拼接来合并列表
li.extend(other_li) # li 是 [1, 2, 3, 4, 5, 6]

# 用 in 来返回元素是否在列表中
1 in li # => True

# 返回列表长度
len(li) # => 6

# 元组类似于列表，但它是不可改变的
tup = (1, 2, 3)
tup[0] # => 1
tup[0] = 3 # 类型错误

# 对于大多数的列表操作，也适用于元组
len(tup) # => 3
tup + (4, 5, 6) # => (1, 2, 3, 4, 5, 6)
tup[:2] # => (1, 2)
2 in tup # => True

# 你可以将元组解包赋给多个变量
a, b, c = (1, 2, 3) # a 是 1, b 是 2, c 是 3
# 如果不加括号，将会被自动视为元组
d, e, f = 4, 5, 6
# 现在我们可以看看交换两个数字是多么容易的事
e, d = d, e # d 是 5, e 是 4

# 字典用来储存映射关系
empty_dict = {}
```



```
# 字典初始化
filled_dict = {"one": 1, "two": 2, "three": 3}

# 字典也用中括号访问元素
filled_dict["one"] # => 1

# 把所有的键保存在列表中
filled_dict.keys() # => ["three", "two", "one"]
# 键的顺序并不是唯一的，得到的不一定是这个顺序

# 把所有的值保存在列表中
filled_dict.values() # => [3, 2, 1]
# 和键的顺序相同

# 判断一个键是否存在
"one" in filled_dict # => True
1 in filled_dict # => False

# 查询一个不存在的键会抛出 KeyError
filled_dict["four"] # KeyError

# 用 get 方法来避免 KeyError
filled_dict.get("one") # => 1
filled_dict.get("four") # => None
# get 方法支持在不存在的时候返回一个默认值
filled_dict.get("one", 4) # => 1
filled_dict.get("four", 4) # => 4

# setdefault 是一个更安全的添加字典元素的方法
filled_dict.setdefault("five", 5) # filled_dict["five"] 的值为 5
filled_dict.setdefault("five", 6) # filled_dict["five"] 的值仍然是 5

# 集合储存无顺序的元素
empty_set = set()
# 初始化一个集合
some_set = set([1, 2, 2, 3, 4]) # some_set 现在是 set([1, 2, 3, 4])

# Python 2.7 之后，大括号可以用来表示集合
filled_set = {1, 2, 2, 3, 4} # => {1 2 3 4}

# 向集合添加元素
filled_set.add(5) # filled_set 现在是 {1, 2, 3, 4, 5}

# 用 & 来计算集合的交
other_set = {3, 4, 5, 6}
filled_set & other_set # => {3, 4, 5}

# 用 | 来计算集合的并
filled_set | other_set # => {1, 2, 3, 4, 5, 6}
```

```

# 用 - 来计算集合的差
{1, 2, 3, 4} - {2, 3, 5}  # => {1, 4}

# 用 in 来判断元素是否存在于集合中
2 in filled_set  # => True
10 in filled_set  # => False

#####
## 3. 控制流程
#####

# 新建一个变量
some_var = 5

# 这是个 if 语句，在 python 中缩进是很重要的。
# 下面的代码片段将会输出 "some var is smaller than 10"
if some_var > 10:
    print "some_var is totally bigger than 10."
elif some_var < 10:  # 这个 elif 语句是不必须的
    print "some_var is smaller than 10."
else:  # 这个 else 也不是必须的
    print "some_var is indeed 10."

"""
用for循环遍历列表
输出：
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    # 你可以用 % 来格式化字符串
    print "%s is a mammal" % animal

"""
`range(number)` 返回从0到给定数字的列表
输出：
    0
    1
    2
    3
"""
for i in range(4):
    print i

"""
while 循环
输出：
    0

```

```

1
2
3
"""
x = 0
while x < 4:
    print x
    x += 1 # x = x + 1 的简写

# 用 try/except 块来处理异常

# Python 2.6 及以上适用:
try:
    # 用 raise 来抛出异常
    raise IndexError("This is an index error")
except IndexError as e:
    pass # pass 就是什么都不做, 不过通常这里会做一些恢复工作

#####
## 4. 函数
#####

# 用 def 来新建函数
def add(x, y):
    print "x is %s and y is %s" % (x, y)
    return x + y # 通过 return 来返回值

# 调用带参数的函数
add(5, 6) # => 输出 "x is 5 and y is 6" 返回 11

# 通过关键字赋值来调用函数
add(y=6, x=5) # 顺序是无所谓的

# 我们也可以定义接受多个变量的函数, 这些变量是按照顺序排列的
def varargs(*args):
    return args

varargs(1, 2, 3) # => (1,2,3)

# 我们也可以定义接受多个变量的函数, 这些变量是按照关键字排列的
def keyword_args(**kwargs):
    return kwargs

# 实际效果:
keyword_args(big="foot", loch="ness") # => {"big": "foot", "loch": "ness"}

# 你也可以同时将一个函数定义成两种形式
def all_the_args(*args, **kwargs):
    print args

```

```

    print kwargs
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# 当调用函数的时候, 我们也可以进行相反的操作, 把元组和字典展开为参数
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args) # 等价于 foo(1, 2, 3, 4)
all_the_args(**kwargs) # 等价于 foo(a=3, b=4)
all_the_args(*args, **kwargs) # 等价于 foo(1, 2, 3, 4, a=3, b=4)

# 函数在 python 中是一等公民
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3) # => 13

# 匿名函数
(lambda x: x > 2)(3) # => True

# 内置高阶函数
map(add_10, [1, 2, 3]) # => [11, 12, 13]
filter(lambda x: x > 5, [3, 4, 5, 6, 7]) # => [6, 7]

# 可以用列表方法来对高阶函数进行更巧妙的引用
[add_10(i) for i in [1, 2, 3]] # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5] # => [6, 7]

#####
## 5. 类
#####

# 我们新建的类是从 object 类中继承的
class Human(object):

    # 类属性, 由所有类的对象共享
    species = "H. sapiens"

    # 基本构造函数
    def __init__(self, name):
        # 将参数赋给对象成员属性
        self.name = name

    # 成员方法, 参数要有 self
    def say(self, msg):

```

```

        return "%s: %s" % (self.name, msg)

# 类方法由所有类的对象共享
# 这类方法在调用时，会把类本身传给第一个参数
@classmethod
def get_species(cls):
    return cls.species

# 静态方法是不需要类和对象的引用就可以调用的方法
@staticmethod
def grunt():
    return "*grunt*"

# 实例化一个类
i = Human(name="Ian")
print i.say("hi")      # 输出 "Ian: hi"

j = Human("Joel")
print j.say("hello")   # 输出 "Joel: hello"

# 访问类的方法
i.get_species() # => "H. sapiens"

# 改变共享属性
Human.species = "H. neanderthalensis"
i.get_species() # => "H. neanderthalensis"
j.get_species() # => "H. neanderthalensis"

# 访问静态变量
Human.grunt() # => "*grunt*"

#####
## 6. 模块
#####

# 我们可以导入其他模块
import math
print math.sqrt(16) # => 4

# 我们也可以从一个模块中导入特定的函数
from math import ceil, floor
print ceil(3.7) # => 4.0
print floor(3.7) # => 3.0

# 从模块中导入所有的函数
# 警告：不推荐使用
from math import *

# 简写模块名

```

```
import math as m
math.sqrt(16) == m.sqrt(16)  # => True

# Python的模块其实只是普通的python文件
# 你也可以创建自己的模块，并且导入它们
# 模块的名字就和文件的名字相同

# 也可以通过下面的方法查看模块中有什么属性和方法
import math
dir(math)
```

更多阅读

希望学到更多？试试下面的链接：

- [Learn Python The Hard Way](#)
- [Dive Into Python](#)
- [The Official Docs](#)
- [Hitchhiker's Guide to Python](#)
- [Python Module of the Week](#)

language: python3 contributors:

```
- ["Louie Dinh", "http://pythonpracticeprojects.com"]
- ["Steven Basart", "http://github.com/xksteven"]
- ["Andre Polykanine", "https://github.com/Oire"]
```

translators:

```
- ["Geoff Liu", "http://geoffliu.me"]
```

filename: learnpython3-cn.py

lang: zh-cn

Python是由吉多·范罗苏姆(Guido Van Rossum)在90年代早期设计。它是如今最常用的编程 语言之一。它的语法简洁且优美，几乎就是可执行的伪代码。

欢迎大家斧正。英文版原作Louie Dinh @louiedinh 或着Email louiedinh [at] [谷歌的信箱服务]。中文翻译Geoff Liu。

注意：这篇教程是特别为Python3写的。如果你想学旧版Python2，我们特别有另一篇教程。

```
# 用井字符开头的是单行注释

""" 多行字符串用三个引号
    包裹，也常被用来做多
    行注释
"""

#####
## 1. 原始数据类型和运算符
#####

# 整数
3 # => 3

# 算术没有什么出乎意料的
1 + 1 # => 2
8 - 1 # => 7
10 * 2 # => 20

# 但是除法例外，会自动转换成浮点数
35 / 5 # => 7.0
5 / 3 # => 1.6666666666666667
```

```
# 整数除法的结果都是向下取整
5 // 3      # => 1
5.0 // 3.0 # => 1.0 # 浮点数也可以
-5 // 3     # => -2
-5.0 // 3.0 # => -2.0

# 浮点数的运算结果也是浮点数
3 * 2.0 # => 6.0

# 模除
7 % 3 # => 1

# x的y次方
2**4 # => 16

# 用括号决定优先级
(1 + 3) * 2 # => 8

# 布尔值
True
False

# 用not取非
not True # => False
not False # => True

# 逻辑运算符, 注意and和or都是小写
True and False #=> False
False or True #=> True

# 整数也可以当作布尔值
0 and 2 #=> 0
-5 or 0 #=> -5
0 == False #=> True
2 == True #=> False
1 == True #=> True

# 用==判断相等
1 == 1 # => True
2 == 1 # => False

# 用!=判断不等
1 != 1 # => False
2 != 1 # => True

# 比较大小
1 < 10 # => True
1 > 10 # => False
2 <= 2 # => True
2 >= 2 # => True
```



```

# 大小比较可以连起来！
1 < 2 < 3 # => True
2 < 3 < 2 # => False

# 字符串用单引双引都可以
"这是个字符串"
'这也是个字符串'

# 用加号连接字符串
"Hello " + "world!" # => "Hello world!"

# 字符串可以被当作字符列表
"This is a string"[0] # => 'T'

# 用.format来格式化字符串
"{ } can be { }".format("strings", "interpolated")

# 可以重复参数以节省时间
"{0} be nimble, {0} be quick, {0} jump over the {1}".format("Jack", "candle stick")
#=> "Jack be nimble, Jack be quick, Jack jump over the candle stick"

# 如果不想数参数，可以用关键字
"{name} wants to eat {food}".format(name="Bob", food="lasagna") #=> "Bob wants to eat la

# 如果你的Python3程序也要在Python2.5以下环境运行，也可以用老式的格式化语法
"%s can be %s the %s way" % ("strings", "interpolated", "old")

# None是一个对象
None # => None

# 当与None进行比较时不要用 ==，要用is。is是用来比较两个变量是否指向同一个对象。
"etc" is None # => False
None is None # => True

# None, 0, 空字符串, 空列表, 空字典都算是False
# 所有其他值都是True
bool(0) # => False
bool("") # => False
bool([]) #=> False
bool({}) #=> False

#####
## 2. 变量和集合
#####

# print是内置的打印函数
print("I'm Python. Nice to meet you!")

# 在给变量赋值前不用提前声明
# 传统的变量命名是小写，用下划线分隔单词

```

```
some_var = 5
some_var # => 5

# 访问未赋值的变量会抛出异常
# 参考流程控制一段来学习异常处理
some_unknown_var # 抛出NameError

# 用列表(list)储存序列
li = []
# 创建列表时也可以同时赋给元素
other_li = [4, 5, 6]

# 用append在列表最后追加元素
li.append(1) # li现在是[1]
li.append(2) # li现在是[1, 2]
li.append(4) # li现在是[1, 2, 4]
li.append(3) # li现在是[1, 2, 4, 3]
# 用pop从列表尾部删除
li.pop() # => 3 且li现在是[1, 2, 4]
# 把3再放回去
li.append(3) # li变回[1, 2, 4, 3]

# 列表存取跟数组一样
li[0] # => 1
# 取出最后一个元素
li[-1] # => 3

# 越界存取会造成IndexError
li[4] # 抛出IndexError

# 列表有切割语法
li[1:3] # => [2, 4]
# 取尾
li[2:] # => [4, 3]
# 取头
li[:3] # => [1, 2, 4]
# 隔一个取一个
li[::2] # => [1, 4]
# 倒排列表
li[::-1] # => [3, 4, 2, 1]
# 可以用三个参数的任何组合来构建切割
# li[始:终:步伐]

# 用del删除任何一个元素
del li[2] # li is now [1, 2, 3]

# 列表可以相加
# 注意: li和other_li的值都不变
li + other_li # => [1, 2, 3, 4, 5, 6]

# 用extend拼接列表
```

```

li.extend(other_li)    # li现在是[1, 2, 3, 4, 5, 6]

# 用in测试列表是否包含值
1 in li    # => True

# 用len取列表长度
len(li)    # => 6

# 元组是不可改变的序列
tup = (1, 2, 3)
tup[0]    # => 1
tup[0] = 3    # 抛出TypeError

# 列表允许的操作元组大都可以
len(tup)    # => 3
tup + (4, 5, 6)    # => (1, 2, 3, 4, 5, 6)
tup[:2]    # => (1, 2)
2 in tup    # => True

# 可以把元组合列表解包，赋值给变量
a, b, c = (1, 2, 3)    # 现在a是1, b是2, c是3
# 元组周围的括号是可以省略的
d, e, f = 4, 5, 6
# 交换两个变量的值就这么简单
e, d = d, e    # 现在d是5, e是4

# 用字典表达映射关系
empty_dict = {}
# 初始化的字典
filled_dict = {"one": 1, "two": 2, "three": 3}

# 用[]取值
filled_dict["one"]    # => 1

# 用keys获得所有的键。因为keys返回一个可迭代对象，所以在这里把结果包在list里。我们下面会详细介绍可选
# 注意：字典键的顺序是不定的，你得到的结果可能和以下不同。
list(filled_dict.keys())    # => ["three", "two", "one"]

# 用values获得所有的值。跟keys一样，要用list包起来，顺序也可能不同。
list(filled_dict.values())    # => [3, 2, 1]

# 用in测试一个字典是否包含一个键
"one" in filled_dict    # => True
1 in filled_dict    # => False

# 访问不存在的键会导致KeyError

```

```

filled_dict["four"]    # KeyError

# 用get来避免KeyError
filled_dict.get("one")    # => 1
filled_dict.get("four")    # => None
# 当键不存在的时候get方法可以返回默认值
filled_dict.get("one", 4)    # => 1
filled_dict.get("four", 4)    # => 4

# setdefault方法只有当键不存在的时候插入新值
filled_dict.setdefault("five", 5)    # filled_dict["five"]设为5
filled_dict.setdefault("five", 6)    # filled_dict["five"]还是5

# 字典赋值
filled_dict.update({"four":4})    #=> {"one": 1, "two": 2, "three": 3, "four": 4}
filled_dict["four"] = 4    # 另一种赋值方法

# 用del删除
del filled_dict["one"]    # 从filled_dict中把one删除

# 用set表达集合
empty_set = set()
# 初始化一个集合，语法跟字典相似。
some_set = {1, 1, 2, 2, 3, 4}    # some_set现在是{1, 2, 3, 4}

# 可以把集合赋值于变量
filled_set = some_set

# 为集合添加元素
filled_set.add(5)    # filled_set现在是{1, 2, 3, 4, 5}

# & 取交集
other_set = {3, 4, 5, 6}
filled_set & other_set    # => {3, 4, 5}

# | 取并集
filled_set | other_set    # => {1, 2, 3, 4, 5, 6}

# - 取补集
{1, 2, 3, 4} - {2, 3, 5}    # => {1, 4}

# in 测试集合是否包含元素
2 in filled_set    # => True
10 in filled_set    # => False

```

```

#####
## 3. 流程控制和迭代器
#####

```

```

# 先随便定义一个变量
some_var = 5

# 这是个if语句。注意缩进在Python里是有意义的
# 印出"some_var比10小"
if some_var > 10:
    print("some_var比10大")
elif some_var < 10:    # elif句是可选的
    print("some_var比10小")
else:                  # else也是可选的
    print("some_var就是10")

"""
用for循环语句遍历列表
打印:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    print("{} is a mammal".format(animal))

"""
"range(number)"返回数字列表从0到给的数字
打印:
    0
    1
    2
    3
"""
for i in range(4):
    print(i)

"""
while循环直到条件不满足
打印:
    0
    1
    2
    3
"""
x = 0
while x < 4:
    print(x)
    x += 1  # x = x + 1 的简写

# 用try/except块处理异常状况
try:
    # 用raise抛出异常
    raise IndexError("This is an index error")

```

```

except IndexError as e:
    pass    # pass是无操作，但是应该在这里处理错误
except (TypeError, NameError):
    pass    # 可以同时处理不同类的错误
else:      # else语句是可选的，必须在所有的except之后
    print("All good!")    # 只有当try运行完没有错误的时候这句才会运行

```

Python提供一个叫做可迭代(iterable)的基本抽象。一个可迭代对象是可以被当作序列的对象。比如说上面range返回的对象就是可迭代的。

```

filled_dict = {"one": 1, "two": 2, "three": 3}
our_iterable = filled_dict.keys()
print(our_iterable) # => range(1,10) 是一个实现可迭代接口的对象

```

```

# 可迭代对象可以遍历
for i in our_iterable:
    print(i)    # 打印 one, two, three

```

```

# 但是不可以随机访问
our_iterable[1] # 抛出TypeError

```

```

# 可迭代对象知道怎么生成迭代器
our_iterator = iter(our_iterable)

```

```

# 迭代器是一个可以记住遍历的位置的对象
# 用__next__可以取得下一个元素
our_iterator.__next__() #=> "one"

```

```

# 再一次调取__next__时会记得位置
our_iterator.__next__() #=> "two"
our_iterator.__next__() #=> "three"

```

```

# 当迭代器所有元素都取出后，会抛出StopIteration
our_iterator.__next__() # 抛出StopIteration

```

```

# 可以用list一次取出迭代器所有的元素
list(filled_dict.keys()) #=> Returns ["one", "two", "three"]

```

```

#####

```

```

## 4. 函数

```

```

#####

```

```

# 用def定义新函数
def add(x, y):
    print("x is {} and y is {}".format(x, y))
    return x + y    # 用return语句返回

```

```

# 调用函数

```

```

add(5, 6)    # => 印出"x is 5 and y is 6"并且返回11

# 也可以用关键字参数来调用函数
add(y=6, x=5)    # 关键字参数可以用任何顺序

# 我们可以定义一个可变参数函数
def varargs(*args):
    return args

varargs(1, 2, 3)    # => (1, 2, 3)

# 我们也可以定义一个关键字可变参数函数
def keyword_args(**kwargs):
    return kwargs

# 我们来看看结果是什么:
keyword_args(big="foot", loch="ness")    # => {"big": "foot", "loch": "ness"}

# 这两种可变参数可以混着用
def all_the_args(*args, **kwargs):
    print(args)
    print(kwargs)
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# 调用可变参数函数时可以做跟上面相反的, 用*展开序列, 用**展开字典。
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)    # 相当于 foo(1, 2, 3, 4)
all_the_args(**kwargs)    # 相当于 foo(a=3, b=4)
all_the_args(*args, **kwargs)    # 相当于 foo(1, 2, 3, 4, a=3, b=4)

# 函数作用域
x = 5

def setX(num):
    # 局部作用域的x和全局域的x是不同的
    x = num    # => 43
    print (x)    # => 43

def setGlobalX(num):
    global x
    print (x)    # => 5
    x = num    # 现在全局域的x被赋值

```

```

    print (x) # => 6

setX(43)
setGlobalX(6)

# 函数在Python是一等公民
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)    # => 13

# 也有匿名函数
(lambda x: x > 2)(3)    # => True

# 内置的高阶函数
map(add_10, [1, 2, 3])    # => [11, 12, 13]
filter(lambda x: x > 5, [3, 4, 5, 6, 7])    # => [6, 7]

# 用列表推导式可以简化映射和过滤。列表推导式的返回值是另一个列表。
[add_10(i) for i in [1, 2, 3]]    # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]    # => [6, 7]

#####
## 5. 类
#####

# 定义一个继承object的类
class Human(object):

    # 类属性，被所有此类的实例共用。
    species = "H. sapiens"

    # 构造方法，当实例被初始化时被调用。注意名字前后的双下划线，这是表明这个属
    # 性或方法对Python有特殊意义，但是允许用户自行定义。你自己取名时不应该用这
    # 种格式。
    def __init__(self, name):
        # Assign the argument to the instance's name attribute
        self.name = name

    # 实例方法，第一个参数总是self，就是这个实例对象
    def say(self, msg):
        return "{name}: {message}".format(name=self.name, message=msg)

    # 类方法，被所有此类的实例共用。第一个参数是这个类对象。
    @classmethod
    def get_species(cls):

```



```

        return cls.species

# 静态方法。调用时没有实例或类的绑定。
@staticmethod
def grunt():
    return "*grunt*"

# 构造一个实例
i = Human(name="Ian")
print(i.say("hi"))      # 印出 "Ian: hi"

j = Human("Joel")
print(j.say("hello"))   # 印出 "Joel: hello"

# 调用一个类方法
i.get_species()         # => "H. sapiens"

# 改一个共用的类属性
Human.species = "H. neanderthalensis"
i.get_species()         # => "H. neanderthalensis"
j.get_species()         # => "H. neanderthalensis"

# 调用静态方法
Human.grunt()           # => "*grunt*"

#####
## 6. 模块
#####

# 用import导入模块
import math
print(math.sqrt(16))    # => 4.0

# 也可以从模块中导入个别值
from math import ceil, floor
print(ceil(3.7))        # => 4.0
print(floor(3.7))       # => 3.0

# 可以导入一个模块中所有值
# 警告：不建议这么做
from math import *

# 如此缩写模块名字
import math as m
math.sqrt(16) == m.sqrt(16)    # => True

# Python模块其实就是普通的Python文件。你可以自己写，然后导入，
# 模块的名字就是文件的名字。

```

```

# 你可以这样列出一个模块里所有的值
import math
dir(math)

#####
## 7. 高级用法
#####

# 用生成器(generators)方便地写惰性运算
def double_numbers(iterable):
    for i in iterable:
        yield i + i

# 生成器只有在需要时才计算下一个值。它们每一次循环只生成一个值，而不是把所有的
# 值全部算好。这意味着double_numbers不会生成大于15的数字。
#
# range的返回值也是一个生成器，不然一个1到900000000的列表会花很多时间和内存。
#
# 如果你想用一个Python的关键字当作变量名，可以加一个下划线来区分。
range_ = range(1, 900000000)
# 当找到一个 >=30 的结果就会停
for i in double_numbers(range_):
    print(i)
    if i >= 30:
        break

# 装饰器(decorators)
# 这个例子中，beg装饰say
# beg会先调用say。如果返回的say_please为真，beg会改变返回的字符串。
from functools import wraps

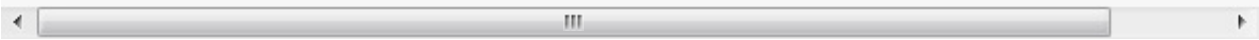
def beg(target_function):
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        msg, say_please = target_function(*args, **kwargs)
        if say_please:
            return "{} {}".format(msg, "Please! I am poor :(")
        return msg

    return wrapper

@beg
def say(say_please=False):
    msg = "Can you buy me a beer?"
    return msg, say_please

```

```
print(say()) # Can you buy me a beer?  
print(say(say_please=True)) # Can you buy me a beer? Please! I am poor :(
```



想继续学吗？

线上免费材料（英文）

- [Learn Python The Hard Way](#)
- [Dive Into Python](#)
- [Ideas for Python Projects](#)
- [The Official Docs](#)
- [Hitchhiker's Guide to Python](#)
- [Python Module of the Week](#)
- [A Crash Course in Python for Scientists](#)

书籍（也是英文）

- [Programming Python](#)
- [Dive Into Python](#)
- [Python Essential Reference](#)

language: R contributors:

```
- ["e99n09", "http://github.com/e99n09"]
- ["isomorphismes", "http://twitter.com/isomorphismes"]
```

translators:

```
- ["小柴", "http://weibo.com/u/2328126220"]
- ["alswl", "https://github.com/alswl"]
```

filename: learnr-zh.r

lang: zh-cn

R 是一门统计语言。它有很多数据分析和挖掘程序包。可以用来统计、分析和制图。你也可以在 LaTeX 文档中运行 `R` 命令。

```
# 评论以 # 开始

# R 语言原生不支持 多行注释
# 但是你可以像这样来多行注释

# 在窗口里按回车键可以执行一条命令

#####
# 不用懂编程就可以开始动手了
#####

data()      # 浏览内建的数据集
data(rivers) # 北美主要河流的长度（数据集）
ls()        # 在工作空间中查看「河流」是否出现
head(rivers) # 撇一眼数据集
# 735 320 325 392 524 450
length(rivers) # 我们测量了多少条河流？
# 141
summary(rivers)
#   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  135.0   310.0   425.0   591.2   680.0  3710.0
stem(rivers) # 茎叶图（一种类似于直方图的展现形式）
#
# The decimal point is 2 digit(s) to the right of the |
#
#   0 | 4
#   2 | 01122333455556666777888889990001111223333344455555666688888999
```

```
# 4 | 111222333445566779001233344567
# 6 | 000112233578012234468
# 8 | 045790018
# 10 | 04507
# 12 | 1471
# 14 | 56
# 16 | 7
# 18 | 9
# 20 |
# 22 | 25
# 24 | 3
# 26 |
# 28 |
# 30 |
# 32 |
# 34 |
# 36 | 1
```

`stem(log(rivers))` # 查看数据集的方式既不是标准形式，也不是取log后的结果！看起来，是钟形曲线形式

```
# The decimal point is 1 digit(s) to the left of the |
#
# 48 | 1
# 50 |
# 52 | 15578
# 54 | 44571222466689
# 56 | 023334677000124455789
# 58 | 00122366666999933445777
# 60 | 122445567800133459
# 62 | 112666799035
# 64 | 00011334581257889
# 66 | 003683579
# 68 | 0019156
# 70 | 079357
# 72 | 89
# 74 | 84
# 76 | 56
# 78 | 4
# 80 |
# 82 | 2
```

`hist(rivers, col="#333333", border="white", breaks=25)` # 试试用这些参数画画（译者注：给
`hist(log(rivers), col="#333333", border="white", breaks=25)` #你还可以做更多式样的绘图

还有其他一些简单的数据集可以被用来加载。R 语言包括了大量这种 `data()`

`data(discoveries)`

`plot(discoveries, col="#333333", lwd=3, xlab="Year", main="Number of important discoveri`

译者注：参数为（数据源，颜色，线条宽度，X 轴名称，标题）

`plot(discoveries, col="#333333", lwd=3, type = "h", xlab="Year", main="Number of importa`

```
sort(discoveries)
# [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2
# [26] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3
# [51] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4
# [76] 4 4 4 4 5 5 5 5 5 5 5 6 6 6 6 6 6 7 7 7 7 8 9 10 12
```

```
#
# The decimal point is at the |
#
# 0 | 0000000000
# 1 | 000000000000
# 2 | 000000000000000000000000000000
# 3 | 000000000000000000000000
# 4 | 00000000000000
# 5 | 00000000
# 6 | 0000000
# 7 | 0000
# 8 | 0
# 9 | 0
# 10 | 0
# 11 |
# 12 | 0
```

12

#	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
#	0.0	2.0	3.0	3.1	4.0	12.0

1 4 6 1 4 6 4

```
# [1] 0.07528471 1.03499859 1.34809556 -0.82356087 0.61638975 -1.88757271
# [7] -0.59975593 0.57629164 1.08455362
```

```
#####  
# 基础编程  
#####
```

```
# 数值
```

```
#“数值”指的是双精度的浮点数
```

```
5      # 5  
class(5)    # "numeric"  
5e4      # 50000          # 用科学技术法方便的处理极大值、极小值或者可变的量级  
6.02e23    # 阿伏伽德罗常数#  
1.6e-35     # 布朗克长度
```

```
# 长整数并用 L 结尾
```

```
5L      # 5
```

```
#输出5L
```

```
class(5L)    # "integer"
```

```
# 可以自己试一试? 用 class() 函数获取更多信息
```

```
# 事实上, 你可以找一些文件查阅 `xyz` 以及xyz的差别
```

```
# `xyz` 用来查看源码实现, ?xyz 用来看帮助
```

```
# 算法
```

```
10 + 66     # 76  
53.2 - 4     # 49.2  
2 * 2.0      # 4  
3L / 4       # 0.75  
3 %% 2       # 1
```

```
# 特殊数值类型
```

```
class(NaN)    # "numeric"
```

```
class(Inf)     # "numeric"
```

```
class(-Inf)    # "numeric"          # 在以下场景中会用到 integrate( dnorm(x), 3, Inf ) -- 消除
```

```
# 但要注意, NaN 并不是唯一的特殊数值类型.....
```

```
class(NA)      # 看上面
```

```
class(NULL)    # NULL
```

```
# 简单列表
```

```
c(6, 8, 7, 5, 3, 0, 9)    # 6 8 7 5 3 0 9  
c('alef', 'bet', 'gimmel', 'dalet', 'he')
```

```
c('Z', 'o', 'r', 'o') == "Zoro"    # FALSE FALSE FALSE FALSE
```

```
# 一些优雅的内置功能
```

```
5:15    # 5  6  7  8  9 10 11 12 13 14 15
```

```
seq(from=0, to=31337, by=1337)
```

```
# [1]      0  1337  2674  4011  5348  6685  8022  9359 10696 12033 13370 14707
```

```
# [13] 16044 17381 18718 20055 21392 22729 24066 25403 26740 28077 29414 30751
```

```
letters
```

```
# [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
```

```
# [20] "t" "u" "v" "w" "x" "y" "z"
```

```
month.abb    # "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

```
# Access the n'th element of a list with list.name[n] or sometimes list.name[[n]]
```

```
# 使用 list.name[n] 来访问第 n 个列表元素，有时候需要使用 list.name[[n]]
```

```
letters[18]   # "r"
```

```
LETTERS[13]   # "M"
```

```
month.name[9]  # "September"
```

```
c(6, 8, 7, 5, 3, 0, 9)[3]    # 7
```

```
# 字符串
```

```
# 字符串和字符在 R 语言中没有区别
```

```
"Horatio"    # "Horatio"
```

```
class("Horatio") # "character"
```

```
substr("Fortuna multis dat nimis, nulli satis.", 9, 15)    # "multis "
```

```
gsub('u', 'ø', "Fortuna multis dat nimis, nulli satis.")    # "Fortøna møltis dat nimis,"
```

```
# 逻辑值
```

```
# 布尔值
```

```
class(TRUE)    # "logical"
```

```
class(FALSE)   # "logical"
```

```
# 和我们预想的一样
```

```
TRUE == TRUE    # TRUE
```

```
TRUE == FALSE   # FALSE
```

```
FALSE != FALSE  # FALSE
```

```
FALSE != TRUE   # TRUE
```

```
# 缺失数据 (NA) 也是逻辑值
```

```
class(NA)       # "logical"
```

```
#定义NA为逻辑型
```



```

# 因子
# 因子是为数据分类排序设计的（像是排序小朋友们的年级或性别）
levels(factor(c("female", "male", "male", "female", "NA", "female"))))    # "female" "mal

factor(c("female", "female", "male", "NA", "female"))
# female female male NA female
# Levels: female male NA

data(infert)    # 自然以及引产导致的不育症
levels(infert$education)    # "0-5yrs" "6-11yrs" "12+ yrs"

# 变量

# 有许多种方式用来赋值
x = 5 # 这样可以
y <- "1" # 更推荐这样
TRUE -> z # 这样可行，但是很怪

#我们还可以使用强制转型
as.numeric(y)    # 1
as.character(x)    # "5"

# 循环

# for 循环语句
for (i in 1:4) {
  print(i)
}

# while 循环
a <- 10
while (a > 4) {
  cat(a, "...", sep = "")
  a <- a - 1
}

# 记住，在 R 语言中 for / while 循环都很慢
# 建议使用 apply()（我们一会介绍）来错做一串数据（比如一系列或者一行数据）

# IF/ELSE

# 再来看这些优雅的标准
if (4 > 3) {
  print("Huzzah! It worked!")
} else {
  print("Noooo! This is blatantly illogical!")
}

# =>

```

```

# [1] "Huzzah! It worked!"

# 函数

# 定义如下
jiggle <- function(x) {
  x + rnorm(x, sd=.1)    #add in a bit of (controlled) noise
  return(x)
}

# 和其他 R 语言函数一样调用
jiggle(5)    # 5±ε. 使用 set.seed(2716057) 后, jiggle(5)==5.005043

#####
# 数据容器: vectors, matrices, data frames, and arrays
#####

# 单维度
# 你可以将目前我们学习到的任何类型矢量化, 只要它们拥有相同的类型
vec <- c(8, 9, 10, 11)
vec    # 8 9 10 11
# 矢量的类型是这一组数据元素的类型
class(vec)    # "numeric"
# If you vectorize items of different classes, weird coercions happen
#如果你强制的将不同类型数值矢量化, 会出现特殊值
c(TRUE, 4)    # 1 4
c("dog", TRUE, 4)    # "dog" "TRUE" "4"

#我们这样来取内部数据, (R 的下标索引顺序 1 开始)
vec[1]    # 8
# 我们可以根据条件查找特定数据
which(vec %% 2 == 0)    # 1 3
# 抓取矢量中第一个和最后一个字符
head(vec, 1)    # 8
tail(vec, 1)    # 11
#如果下标溢出或不存会得到 NA
vec[6]    # NA
# 你可以使用 length() 获取矢量的长度
length(vec)    # 4

# 你可以直接操作矢量或者矢量的子集
vec * 4    # 16 20 24 28
vec[2:3] * 5    # 25 30
# 这里有许多内置的函数, 来表现向量
mean(vec)    # 9.5
var(vec)    # 1.666667
sd(vec)    # 1.290994
max(vec)    # 11
min(vec)    # 8
sum(vec)    # 38

```

```

# 二维（相同元素类型）

#你可以为同样类型的变量建立矩阵
mat <- matrix(nrow = 3, ncol = 2, c(1,2,3,4,5,6))
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
# 和 vector 不一样的是，一个矩阵的类型真的是 「matrix」，而不是内部元素的类型
class(mat) # => "matrix"
# 访问第一行的字符
mat[1,]    # 1 4
# 操作第一行数据
3 * mat[,1]    # 3 6 9
# 访问一个特定数据
mat[3,2]    # 6
# 转置整个矩阵（译者注：变成 2 行 3 列）
t(mat)
# =>
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6

# 使用 cbind() 函数把两个矩阵按列合并，形成新的矩阵
mat2 <- cbind(1:4, c("dog", "cat", "bird", "dog"))
mat2
# =>
#      [,1] [,2]
# [1,] "1"  "dog"
# [2,] "2"  "cat"
# [3,] "3"  "bird"
# [4,] "4"  "dog"
class(mat2)    # matrix
# Again, note what happened!
# 注意
# 因为矩阵内部元素必须包含同样的类型
# 所以现在每一个元素都转化成字符串
c(class(mat2[,1]), class(mat2[,2]))

# 按行合并两个向量，建立新的矩阵
mat3 <- rbind(c(1,2,4,5), c(6,7,0,4))
mat3
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    4    5
# [2,]    6    7    0    4
# 哈哈，数据类型都一样的，没有发生强制转换，生活真美好

# 二维(不同的元素类型)

```

```

# 利用 data frame 可以将不同类型数据放在一起
dat <- data.frame(c(5,2,1,4), c("dog", "cat", "bird", "dog"))
names(dat) <- c("number", "species") # 给数据列命名
class(dat)      # "data.frame"
dat
# =>
#   number species
# 1      5      dog
# 2      2      cat
# 3      1     bird
# 4      4      dog
class(dat$number) # "numeric"
class(dat[,2])    # "factor"
# data.frame() 会将字符向量转换为 factor 向量

# 有很多精妙的方法来获取 data frame 的子数据集
dat$number      # 5 2 1 4
dat[,1]         # 5 2 1 4
dat[, "number"] # 5 2 1 4

# 多维（相同元素类型）

# 使用 array 创建一个 n 维的表格
# You can make a two-dimensional table (sort of like a matrix)
# 你可以建立一个 2 维表格（有点像矩阵）
array(c(c(1,2,4,5),c(8,9,3,6)), dim=c(2,4))
# =>
#      [,1] [,2] [,3] [,4]
# [1,]    1    4    8    3
# [2,]    2    5    9    6
#你也可以利用数组建立一个三维的矩阵
array(c(c(c(2,300,4),c(8,9,0)),c(c(5,60,0),c(66,7,847)))), dim=c(3,2,2))
# =>
# , , 1
#
#      [,1] [,2]
# [1,]    2    8
# [2,]  300    9
# [3,]    4    0
#
# , , 2
#
#      [,1] [,2]
# [1,]    5   66
# [2,]   60    7
# [3,]    0  847

#列表（多维的，不同类型的）

# R语言有列表的形式

```

```

list1 <- list(time = 1:40)
list1$price = c(rnorm(40,.5*list1$time,4)) # 随机
list1

# You can get items in the list like so
# 你可以这样获得列表的元素
list1$time
# You can subset list items like vectors
# 你也可以和矢量一样获取他们的子集
list1$price[4]

#####
# apply()函数家族
#####

# 还记得 mat 么?
mat
# =>
#      [,1] [,2]
# [1,]    1    4
# [2,]    2    5
# [3,]    3    6
# Use apply(X, MARGIN, FUN) to apply function FUN to a matrix X
# 使用(X, MARGIN, FUN)将函数 FUN 应用到矩阵 X 的行 (MAR = 1) 或者 列 (MAR = 2)
# That is, R does FUN to each row (or column) of X, much faster than a
# R 在 X 的每一行/列使用 FUN, 比循环要快很多
apply(mat, MAR = 2, myFunc)
# =>
#      [,1] [,2]
# [1,]    3   15
# [2,]    7   19
# [3,]   11   23
# 还有其他家族函数 ?lapply, ?sapply

# 不要被吓到, 虽然许多人在此都被搞混
# plyr 程序包的作用是用来改进 apply() 函数家族

install.packages("plyr")
require(plyr)
?plyr

#####
# 载入数据
#####

# "pets.csv" 是网上的一个文本
pets <- read.csv("http://learnxinyminutes.com/docs/pets.csv")
pets
head(pets, 2) # 前两行
tail(pets, 1) # 最后一行

```

```

# 以 .csv 格式来保存数据集或者矩阵
write.csv(pets, "pets2.csv") # 保存到新的文件 pets2.csv
# set working directory with setwd(), look it up with getwd()
# 使用 setwd() 改变工作目录, 使用 getwd() 查看当前工作目录

# 尝试使用 ?read.csv 和 ?write.csv 来查看更多信息

#####
# 画图
#####

# 散点图
plot(list1$time, list1$price, main = "fake data") # 译者注: 横轴 list1$time, 纵轴 wlist1$price
# 回归图
linearModel <- lm(price ~ time, data = list1) # 译者注: 线性模型, 数据集为list1, 以价格对时间
linearModel # 拟合结果
# 将拟合结果展示在图上, 颜色设为红色
abline(linearModel, col = "red")
# 也可以获取各种各样漂亮的分析图
plot(linearModel)

# 直方图
hist(rpois(n = 10000, lambda = 5), col = "thistle") # 译者注: 统计频数直方图

# 柱状图
barplot(c(1,4,5,1,2), names.arg = c("red", "blue", "purple", "green", "yellow"))

# 可以尝试使用 ggplot2 程序包来美化图片
install.packages("ggplot2")
require(ggplot2)
?ggplot2

```

获得 R

- 从 <http://www.r-project.org/> 获得安装包和图形化界面
- RStudio 是另一个图形化界面

- ["th3rac25", "<https://github.com/voila>"]
- ["Eli Barzilay", "<https://github.com/elibarzilay>"]
- ["Gustavo Schmidt", "<https://github.com/gustavoschmidt>"] translators:
 - ["lyuehh", "<https://github.com/lyuehh>"]

[illegible]

```
;; 接下来, 是一些数学运算
(+ 1 1) ; => 2
(- 8 1) ; => 7
(* 10 2) ; => 20
(expt 2 3) ; => 8
(quotient 5 2) ; => 2
(remainder 5 2) ; => 1
(/ 35 5) ; => 7
(/ 1 3) ; => 1/3
(exact->inexact 1/3) ; => 0.3333333333333333
(+ 1+2i 2-3i) ; => 3-1i

;;; 布尔类型
#t ; 为真
#f ; 为假, #f 之外的任何值都是真
(not #t) ; => #f
(and 0 #f (error "doesn't get here")) ; => #f
(or #f 0 (error "doesn't get here")) ; => 0

;;; 字符
#\A ; => #\A
#\λ ; => #\λ
#\u03BB ; => #\λ

;;; 字符串是字符组成的定长数组
"Hello, world!"
"Benjamin \"Bugsy\" Siegel" ; \是转义字符
"Foo\tbar\41\x21\u0021\a\r\n" ; 包含C语言的转义字符, 和Unicode
"λx:(μa.α→a).xx" ; 字符串可以包含Unicode字符

;; 字符串可以相加
(string-append "Hello " "world!") ; => "Hello world!"

;; 一个字符串可以看做是一个包含字符的列表
(string-ref "Apple" 0) ; => #\A

;; format 可以用来格式化字符串
(format "~a can be ~a" "strings" "formatted")

;; 打印字符串非常简单
(sprintf "I'm Racket. Nice to meet you!\n")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. 变量
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 你可以使用 define 定义一个变量
;; 变量的名字可以使用任何字符除了: ()[]{}",` ;#\
(define some-var 5)
some-var ; => 5

;; 你也可以使用Unicode字符
```



```

(define  $\subseteq$  subset?)
( $\subseteq$  (set 3 2) (set 1 2 3)) ; => #t

;; 访问未赋值的变量会引发一个异常
; x ; => x: undefined ...

;; 本地绑定: `me' 被绑定到 "Bob", 并且只在 let 中生效
(let ([me "Bob"])
  "Alice"
  me) ; => "Bob"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. 结构和集合
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 结构体
(struct dog (name breed age))
(define my-pet
  (dog "lassie" "collie" 5))
my-pet ; => #
(dog? my-pet) ; => #t
(dog-name my-pet) ; => "lassie"

;;; 对 (不可变的)
;; `cons' 返回对, `car' 和 `cdr' 从对中提取第1个
;; 和第2个元素
(cons 1 2) ; => '(1 . 2)
(car (cons 1 2)) ; => 1
(cdr (cons 1 2)) ; => 2

;;; 列表

;; 列表由链表构成, 由 `cons' 的结果
;; 和一个 `null' (或者 '()) 构成, 后者标记了这个列表的结束
(cons 1 (cons 2 (cons 3 null))) ; => '(1 2 3)
;; `list' 给列表提供了一个非常方便的可变参数的生成器
(list 1 2 3) ; => '(1 2 3)
;; 一个单引号也可以用来表示一个列表字面量
'(1 2 3) ; => '(1 2 3)

;; 仍然可以使用 `cons' 在列表的开始处添加一项
(cons 4 '(1 2 3)) ; => '(4 1 2 3)

;; `append' 函数可以将两个列表合并
(append '(1 2) '(3 4)) ; => '(1 2 3 4)

;; 列表是非常基础的类型, 所以有*很多*操作列表的方法
;; 下面是一些例子:
(map add1 '(1 2 3)) ; => '(2 3 4)
(map + '(1 2 3) '(10 20 30)) ; => '(11 22 33)
(filter even? '(1 2 3 4)) ; => '(2 4)

```

```
(count even? '(1 2 3 4))      ; => 2
(take '(1 2 3 4) 2)           ; => '(1 2)
(drop '(1 2 3 4) 2)           ; => '(3 4)
```

;;; 向量

;; 向量是定长的数组

```
 #(1 2 3) ; => '#(1 2 3)
```

;; 使用 `vector-append' 方法将2个向量合并

```
(vector-append #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)
```

;;; Set(翻译成集合也不太合适，所以不翻译了..)

;; 从一个列表创建一个Set

```
(list->set '(1 2 3 1 2 3 3 2 1 3 2 1)) ; => (set 1 2 3)
```

;; 使用 `set-add' 增加一个成员

;; (函数式特性：这里会返回一个扩展后的Set，而不是修改输入的值)

```
(set-add (set 1 2 3) 4) ; => (set 1 2 3 4)
```

;; 使用 `set-remove' 移除一个成员

```
(set-remove (set 1 2 3) 1) ; => (set 2 3)
```

;; 使用 `set-member?' 测试成员是否存在

```
(set-member? (set 1 2 3) 1) ; => #t
```

```
(set-member? (set 1 2 3) 4) ; => #f
```

;;; 散列表

;; 创建一个不变的散列表 (可变散列表的例子在下面)

```
(define m (hash 'a 1 'b 2 'c 3))
```

;; 根据键取得值

```
(hash-ref m 'a) ; => 1
```

;; 获取一个不存在的键是一个异常

```
; (hash-ref m 'd) => 没有找到元素
```

;; 你可以给不存在的键提供一个默认值

```
(hash-ref m 'd 0) ; => 0
```

;; 使用 `hash-set' 来扩展一个不可变的散列表

;; (返回的是扩展后的散列表而不是修改它)

```
(define m2 (hash-set m 'd 4))
```

```
m2 ; => '#hash((b . 2) (a . 1) (d . 4) (c . 3))
```

;; 记住，使用 `hash` 创建的散列表是不可变的

```
m ; => '#hash((b . 2) (a . 1) (c . 3)) <-- no="" `d'="" ;="" 使用="" `hash-remove'="" ;
```

```
;;;;;;;;;;;;;
```

```
;; 3. 函数
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 使用 `lambda' 创建函数
;; 函数总是返回它最后一个表达式的值
(lambda () "Hello World") ; => #
;; 也可以使用 Unicode 字符 `λ'
(λ () "Hello World")      ; => 同样的函数

;; 使用括号调用一个函数，也可以直接调用一个 lambda 表达式
((lambda () "Hello World")) ; => "Hello World"
((λ () "Hello World"))      ; => "Hello World"

;; 将函数赋值为一个变量
(define hello-world (lambda () "Hello World"))
(hello-world) ; => "Hello World"

;; 你可以使用函数定义的语法糖来简化代码
(define (hello-world2) "Hello World")

;; `()` 是函数的参数列表
(define hello
  (lambda (name)
    (string-append "Hello " name)))
(hello "Steve") ; => "Hello Steve"
;; 同样的，可以使用语法糖来定义：
(define (hello2 name)
  (string-append "Hello " name))

;; 你也可以使用可变参数，`case-lambda'
(define hello3
  (case-lambda
    [(()) "Hello World"]
    [(name) (string-append "Hello " name)]))
(hello3 "Jake") ; => "Hello Jake"
(hello3) ; => "Hello World"
;; ... 或者给参数指定一个可选的默认值
(define (hello4 [name "World"])
  (string-append "Hello " name))

;; 函数可以将多余的参数放到一个列表里
(define (count-args . args)
  (format "You passed ~a args: ~a" (length args) args))
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"
;; ... 也可以使用不带语法糖的 `lambda' 形式：
(define count-args2
  (lambda args
    (format "You passed ~a args: ~a" (length args) args)))

;; 你可以混用两种用法
(define (hello-count name . args)
```

```

    (format "Hello ~a, you passed ~a extra args" name (length args)))
(hello-count "Finn" 1 2 3)
; => "Hello Finn, you passed 3 extra args"
;; ... 不带语法糖的形式:
(define hello-count2
  (lambda (name . args)
    (format "Hello ~a, you passed ~a extra args" name (length args))))

;; 使用关键字
(define (hello-k #:name [name "World"] #:greeting [g "Hello"] . args)
  (format "~a ~a, ~a extra args" g name (length args)))
(hello-k) ; => "Hello World, 0 extra args"
(hello-k 1 2 3) ; => "Hello World, 3 extra args"
(hello-k #:greeting "Hi") ; => "Hi World, 0 extra args"
(hello-k #:name "Finn" #:greeting "Hey") ; => "Hey Finn, 0 extra args"
(hello-k 1 2 3 #:greeting "Hi" #:name "Finn" 4 5 6)
; => "Hi Finn, 6 extra args"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. 判断是否相等
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 判断数字使用 `=`
(= 3 3.0) ; => #t
(= 2 1) ; => #f

;; 判断对象使用 `eq?'
(eq? 3 3) ; => #t
(eq? 3 3.0) ; => #f
(eq? (list 3) (list 3)) ; => #f

;; 判断集合使用 `equal?'
(equal? (list 'a 'b) (list 'a 'b)) ; => #t
(equal? (list 'a 'b) (list 'b 'a)) ; => #f

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. 控制结构
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; 条件判断

(if #t ; 测试表达式
    "this is true" ; 为真的表达式
    "this is false") ; 为假的表达式
; => "this is true"

;; 注意, 除 `#f` 之外的所有值都认为是真
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(Groucho Zeppo)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope)

```

```
; => 'yep
```

```
;; `cond' 会进行一系列的判断来选择一个结果  
(cond [(> 2 2) (error "wrong!")]  
      [(< 2 2) (error "wrong again!")]  
      [else 'ok]) ; => 'ok
```

```
;;; 模式匹配
```

```
(define (fizzbuzz? n)  
  (match (list (remainder n 3) (remainder n 5))  
    [(list 0 0) 'fizzbuzz]  
    [(list 0 _) 'fizz]  
    [(list _ 0) 'buzz]  
    [_ #f]))
```

```
(fizzbuzz? 15) ; => 'fizzbuzz  
(fizzbuzz? 37) ; => #f
```

```
;;; 循环
```

```
;; 循环可以使用递归(尾递归)
```

```
(define (loop i)  
  (when (< i 10)  
    (printf "i=~a\n" i)  
    (loop (add1 i))))  
(loop 5) ; => i=5, i=6, ...
```

```
;; 类似的, 可以使用 `let` 定义
```

```
(let loop ((i 0))  
  (when (< i 10)  
    (printf "i=~a\n" i)  
    (loop (add1 i)))) ; => i=0, i=1, ...
```

```
;; 看上面的例子怎么增加一个新的 `loop' 形式, 但是 Racket 已经有了一个非常
```

```
;; 灵活的 `for' 了:
```

```
(for ([i 10])  
  (printf "i=~a\n" i)) ; => i=0, i=1, ...  
(for ([i (in-range 5 10)])  
  (printf "i=~a\n" i)) ; => i=5, i=6, ...
```

```
;;; 其他形式的迭代
```

```
;; `for' 允许在很多数据结构中迭代:
```

```
;; 列表, 向量, 字符串, Set, 散列表, 等...
```

```
(for ([i (in-list '(l i s t))])  
  (displayln i))
```

```
(for ([i (in-vector #(v e c t o r))])  
  (displayln i))
```

```

(for ([i (in-string "string")])
  (displayln i))

(for ([i (in-set (set 'x 'y 'z))])
  (displayln i))

(for ([k v] (in-hash (hash 'a 1 'b 2 'c 3 ))])
  (printf "key:~a value:~a\n" k v))

;;; 更多复杂的迭代

;; 并行扫描多个序列 (遇到长度小的就停止)
(for ([i 10] [j '(x y z)]) (printf "~a:~a\n" i j))
; => 0:x 1:y 2:z

;; 嵌套循环
(for* ([i 2] [j '(x y z)]) (printf "~a:~a\n" i j))
; => 0:x, 0:y, 0:z, 1:x, 1:y, 1:z

;; 带有条件判断的 `for`
(for ([i 1000]
      #:when (> i 5)
      #:unless (odd? i)
      #:break (> i 10))
  (printf "i=~a\n" i))
; => i=6, i=8, i=10

;;; 更多的例子帮助你加深理解..
;; 和 `for` 循环非常像 -- 收集结果

(for/list ([i '(1 2 3)])
  (add1 i)) ; => '(2 3 4)

(for/list ([i '(1 2 3)] #:when (even? i))
  i) ; => '(2)

(for/list ([i 10] [j '(x y z)])
  (list i j)) ; => '((0 x) (1 y) (2 z))

(for/list ([i 1000] #:when (> i 5) #:unless (odd? i) #:break (> i 10))
  i) ; => '(6 8 10)

(for/hash ([i '(1 2 3)])
  (values i (number->string i)))
; => '#hash((1 . "1") (2 . "2") (3 . "3"))

;; 也有很多其他的内置方法来收集循环中的值:
(for/sum ([i 10]) (* i i)) ; => 285
(for/product ([i (in-range 1 11)]) (* i i)) ; => 13168189440000
(for/and ([i 10] [j (in-range 10 20)]) (< i j)) ; => #t
(for/or ([i 10] [j (in-range 0 20 2)]) (= i j)) ; => #t

```

```
;; 如果需要合并计算结果, 使用 `for/fold`
(for/fold ([sum 0]) ([i '(1 2 3 4)]) (+ sum i)) ; => 10
;; (这个函数可以在大部分情况下替代普通的命令式循环)

;;; 异常

;; 要捕获一个异常, 使用 `with-handlers` 形式
(with-handlers ([exn:fail? (lambda (exn) 999)])
  (+ 1 "2")) ; => 999
(with-handlers ([exn:break? (lambda (exn) "no time")])
  (sleep 3)
  "pew") ; => "pew", 如果你打断了它, 那么结果 => "no time"

;; 使用 `raise` 抛出一个异常后者其他任何值
(with-handlers ([number?      ; 捕获抛出的数字类型的值
                  identity]) ; 将它们作为普通值
  (+ 1 (raise 2))) ; => 2

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. 可变的值
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 使用 `set!` 给一个已经存在的变量赋一个新值
(define n 5)
(set! n (add1 n))
n ; => 6

;; 给那些明确地需要变化的值使用 `boxes` (在其他语言里类似指针
;; 或者引用)
(define n* (box 5))
(set-box! n* (add1 (unbox n*)))
(unbox n*) ; => 6

;; 很多 Racket 诗句类型是不可变的 (对, 列表, 等), 有一些既是可变的
;; 又是不可变的 (字符串, 向量, 散列表
;; 等...)

;; 使用 `vector` 或者 `make-vector` 创建一个可变的向量
(define vec (vector 2 2 3 4))
(define wall (make-vector 100 'bottle-of-beer))
;; 使用 `vector-set!` 更新一项
(vector-set! vec 0 1)
(vector-set! wall 99 'down)
vec ; => #(1 2 3 4)

;; 创建一个空的可变散列表, 然后操作它
(define m3 (make-hash))
(hash-set! m3 'a 1)
(hash-set! m3 'b 2)
(hash-set! m3 'c 3)
(hash-ref m3 'a) ; => 1
```

```
(hash-ref m3 'd 0) ; => 0
(hash-remove! m3 'a)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. 模块
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 模块让你将你的代码组织为多个文件，成为可重用的模块，
;; 在这里，我们使用嵌套在本文的整个大模块
;; 里的子模块(从 "#lang" 这一行开始)
```

```
(module cake racket/base ; 基于 racket/base 定义一个 `cake` 模块
```

```
(provide print-cake) ; 这个模块导出的函数
```

```
(define (print-cake n)
  (show "  ~a  " n #\.)
  (show " .-~a-. " n #\|)
  (show " | ~a | " n #\space)
  (show " ---~a--- " n #\-))
```

```
(define (show fmt n ch) ; 内部函数
  (printf fmt (make-string n ch))
  (newline)))
```

```
;; 使用 `require` 从模块中得到所有 `provide` 的函数
(require 'cake) ; 这里的 `` 表示是本地的子模块
(print-cake 3)
; (show "~a" 1 #\A) ; => 报错, `show' 没有被导出, 不存在
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 8. 类和对象
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; 创建一个 fish% 类(%是给类绑定用的)
```

```
(define fish%
  (class object%
    (init size) ; 初始化的参数
    (super-new) ; 父类的初始化
    ;; 域
    (define current-size size)
    ;; 公共方法
    (define/public (get-size)
      current-size)
    (define/public (grow amt)
      (set! current-size (+ amt current-size)))
    (define/public (eat other-fish)
      (grow (send other-fish get-size)))))
```

```
;; 创建一个 fish% 类的示例
```

```
(define charlie
```



```

(new fish% [size 10]))

;; 使用 `send' 调用一个对象的方法
(send charlie get-size) ; => 10
(send charlie grow 6)
(send charlie get-size) ; => 16

;; `fish%' 是一个普通的值，我们可以用它来混入
(define (add-color c%)
  (class c%
    (init color)
    (super-new)
    (define my-color color)
    (define/public (get-color) my-color)))
(define colored-fish% (add-color fish%))
(define charlie2 (new colored-fish% [size 10] [color 'red]))
(send charlie2 get-color)
;; 或者，不带名字
(send (new (add-color fish%) [size 10] [color 'red]) get-color)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 9. 宏
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 宏让你扩展这门语言的语法

;; 让我们定义一个while循环
(define-syntax-rule (while condition body ...)
  (let loop ()
    (when condition
      body ...
      (loop))))

(let ([i 0])
  (while (< i 10)
    (displayln i)
    (set! i (add1 i))))

;; 宏是安全的，你不能修改现有的变量
(define-syntax-rule (swap! x y) ; !表示会修改
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))

(define tmp 2)
(define other 3)
(swap! tmp other)
(sprintf "tmp = ~a; other = ~a\n" tmp other)
;; 变量 `tmp` 被重命名为 `tmp_1`
;; 避免名字冲突
;; (let ([tmp_1 tmp])

```

```
;; (set! tmp other)
;; (set! other tmp_1))

;; 但它们仍然会导致错误代码，比如：
(define-syntax-rule (bad-while condition body ...)
  (when condition
    body ...
    (bad-while condition body ...)))
;; 这个宏会挂掉，它产生了一个无限循环，如果你试图去使用它
;; 编译器会进入死循环

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 10. 契约
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; 契约限制变量从模块中导入

(module bank-account racket
  (provide (contract-out
    [deposit (-> positive? any)] ; 数量一直是正值
    [balance (-> positive?)]))

  (define amount 0)
  (define (deposit a) (set! amount (+ amount a)))
  (define (balance) amount)
  )

(require 'bank-account)
(deposit 5)

(balance) ; => 5

;; 客户端尝试存储一个负值时会出错
;; (deposit -5) ; => deposit: contract violation
;; expected: positive?
;; given: -5
;; more details....
```

进一步阅读

想知道更多吗？ 尝试 [Getting Started with Racket](#)

language: ruby filename: learnruby-zh.rb lang: zh-cn contributors:

- ["David Underwood", "<http://theflyingdeveloper.com>"]
 - ["Joel Walden", "<http://joelwalden.net>"]
 - ["Luke Holder", "<http://twitter.com/lukeholder>"]
 - ["lidashuang", "<https://github.com/lidashuang>"]
 - ["ftwbzhao", "<https://github.com/ftwbzhao>"] translators:
 - ["Lin Xiangyu", "<https://github.com/oa414>"]
-

```
# 这是单行注释
```

```
=begin
```

```
这是多行注释
```

```
没人用这个
```

```
你也不该用
```

```
=end
```

```
# 首先，也是最重要的，所有东西都是对象
```

```
# 数字是对象
```

```
3.class #=> Fixnum
```

```
3.to_s #=> "3"
```

```
# 一些基本的算术符号
```

```
1 + 1 #=> 2
```

```
8 - 1 #=> 7
```

```
10 * 2 #=> 20
```

```
35 / 5 #=> 7
```

```
# 算术符号只是语法糖而已
```

```
# 实际上是调用对象的方法
```

```
1.+(3) #=> 4
```

```
10.* 5 #=> 50
```

```
# 特殊的值也是对象
```

```
nil # 空
```

```
true # 真
```

```
false # 假
```

```
nil.class #=> NilClass
```

```
true.class #=> TrueClass
```

```
false.class #=> FalseClass
```

```
# 相等运算符
1 == 1 #=> true
2 == 1 #=> false

# 不等运算符
1 != 1 #=> false
2 != 1 #=> true
!true  #=> false
!false #=> true

# 除了false自己, nil是唯一的值为false的对象

!nil    #=> true
!false  #=> true
!0      #=> false

# 更多比较
1 < 10  #=> true
1 > 10  #=> false
2 <= 2  #=> true
2 >= 2  #=> true

# 字符串是对象

'I am a string'.class #=> String
'I am a string too'.class #=> String

placeholder = "use string interpolation"
"I can #{placeholder} when using double quoted strings"
#=> "I can use string interpolation when using double quoted strings"

# 输出值
puts "I'm printing!"

# 变量
x = 25 #=> 25
x #=> 25

# 注意赋值语句返回了赋的值
# 这意味着你可以用多重赋值语句

x = y = 10 #=> 10
x #=> 10
y #=> 10

# 按照惯例, 用 snake_case 作为变量名
snake_case = true

# 使用具有描述性的运算符
path_to_project_root = '/good/name/'
```

```
path = '/bad/name/'

# 符号 (Symbols, 也是对象)
# 符号是不可变的, 内部用整数类型表示的可重用的值。
# 通常用它代替字符串来有效地表示有意义的值。

:pending.class #=> Symbol

status = :pending

status == :pending #=> true

status == 'pending' #=> false

status == :approved #=> false

# 数组

# 这是一个数组
array = [1, 2, 3, 4, 5] #=> [1, 2, 3, 4, 5]

# 数组可以包含不同类型的元素

[1, "hello", false] #=> [1, "hello", false]

# 数组可以被索引
# 从前面开始
array[0] #=> 1
array[12] #=> nil

# 像运算符一样, [var]形式的访问
# 也就是一个语法糖
# 实际上是调用对象的[] 方法
array.[0] #=> 1
array.[12] #=> nil

# 从尾部开始
array[-1] #=> 5

# 同时指定开始的位置和长度
array[2, 3] #=> [3, 4, 5]

# 或者指定一个范围
array[1..3] #=> [2, 3, 4]

# 像这样往数组增加一个元素
array << 6 #=> [1, 2, 3, 4, 5, 6]

# 哈希表是Ruby的键值对的基本数据结构
# 哈希表由大括号定义
```

```

hash = {'color' => 'green', 'number' => 5}

hash.keys #=> ['color', 'number']

# 哈希表可以通过键快速地查询
hash['color'] #=> 'green'
hash['number'] #=> 5

# 查询一个不存在地键将会返回nil
hash['nothing here'] #=> nil

# 用 #each 方法来枚举哈希表:
hash.each do |k, v|
  puts "#{k} is #{v}"
end

# 从Ruby 1.9开始, 用符号作为键的时候有特别的记号表示:

new_hash = { defcon: 3, action: true}

new_hash.keys #=> [:defcon, :action]

# 小贴士: 数组和哈希表都是可枚举的
# 它们可以共享一些有用的方法, 比如each, map, count 等等

# 控制流

if true
  "if statement"
elsif false
  "else if, optional"
else
  "else, also optional"
end

for counter in 1..5
  puts "iteration #{counter}"
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

# 然而
# 没人用for循环
# 用`each`来代替, 就像这样

(1..5).each do |counter|
  puts "iteration #{counter}"
end

```

```
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

counter = 1
while counter <= 5 do
  puts "iteration #{counter}"
  counter += 1
end
#=> iteration 1
#=> iteration 2
#=> iteration 3
#=> iteration 4
#=> iteration 5

grade = 'B'

case grade
when 'A'
  puts "Way to go kiddo"
when 'B'
  puts "Better luck next time"
when 'C'
  puts "You can do better"
when 'D'
  puts "Scraping through"
when 'F'
  puts "You failed!"
else
  puts "Alternative grading system, eh?"
end

# 函数

def double(x)
  x * 2
end

# 函数（以及所有的方法块）隐式地返回了最后语句的值
double(2) #=> 4

# 当不存在歧义的时候括号是可有可无的
double 3 #=> 6

double double 3 #=> 12

def sum(x,y)
  x + y
end
```

```

# 方法的参数通过逗号分隔
sum 3, 4 #=> 7

sum sum(3,4), 5 #=> 12

# yield
# 所有的方法都有一个隐式的块参数
# 可以用yield参数调用

def surround
  puts "{"
  yield
  puts "}"
end

surround { puts 'hello world' }

# {
# hello world
# }

# 用class关键字定义一个类
class Human

  # 一个类变量，它被这个类地所有实例变量共享
  @@species = "H. sapiens"

  # 构造函数
  def initialize(name, age=0)
    # 将参数name的值赋给实例变量@name
    @name = name
    # 如果没有给出age，那么会采用参数列表中地默认地值
    @age = age
  end

  # 基本的 setter 方法
  def name=(name)
    @name = name
  end

  # 基本地 getter 方法
  def name
    @name
  end

  # 一个类方法以self.开头
  # 它可以被类调用，但不能被类的实例调用
  def self.say(msg)
    puts "#{msg}"
  end
end

```



```
end

def species
  @@species
end

end

# 类的例子
jim = Human.new("Jim Halpert")

dwight = Human.new("Dwight K. Schrute")

# 让我们来调用一些方法
jim.species #=> "H. sapiens"
jim.name #=> "Jim Halpert"
jim.name = "Jim Halpert II" #=> "Jim Halpert II"
jim.name #=> "Jim Halpert II"
dwight.species #=> "H. sapiens"
dwight.name #=> "Dwight K. Schrute"

# 调用对象的方法
Human.say("Hi") #=> "Hi"
```

language: rust contributors:

```
- ["P1start", "http://p1start.github.io/"]
```

translators:

```
- ["Guangming Mao", "http://maogm.com"]
```

filename: learnrust-cn.rs

lang: zh-cn

Rust 是由 Mozilla 研究院开发的编程语言。Rust 将底层的性能控制与高级语言的便利性和安全保障结合在了一起。

而 Rust 并不需要一个垃圾回收器或者运行时即可实现这个目的，这使得 Rust 库可以成为一种 C 语言的替代品。

Rust 第一版（0.1 版）发布于 2012 年 1 月，3 年以来一直在紧锣密鼓地迭代。因为更新太频繁，一般建议使用每夜构建版而不是稳定版，直到最近 1.0 版本的发布。

2015 年 3 月 15 日，Rust 1.0 发布，完美向后兼容，最新的每夜构建版提供了缩短编译时间等新特性。Rust 采用了持续迭代模型，每 6 周一个发布版。Rust 1.1 beta 版在 1.0 发布时同时发布。

尽管 Rust 相对来说是一门底层语言，它提供了一些常见于高级语言的函数式编程的特性。这让 Rust 不仅高效，并且易用。

```
// 这是注释，单行注释...
/* ...这是多行注释 */

////////////////
// 1. 基础    //
////////////////

// 函数 (Functions)
// `i32` 是有符号 32 位整数类型(32-bit signed integers)
fn add2(x: i32, y: i32) -> i32 {
    // 隐式返回 (不要分号)
    x + y
}

// 主函数(Main function)
fn main() {
    // 数字 (Numbers) //

    // 不可变绑定
```

```

let x: i32 = 1;

// 整形/浮点型数 后缀
let y: i32 = 13i32;
let f: f64 = 1.3f64;

// 类型推导
// 大部分时间, Rust 编译器会推导变量类型, 所以不必把类型显式写出来。
// 这个教程里面很多地方都显式写了类型, 但是只是为了示范。
// 绝大部分时间可以交给类型推导。
let implicit_x = 1;
let implicit_f = 1.3;

// 算术运算
let sum = x + y + 13;

// 可变变量
let mut mutable = 1;
mutable = 4;
mutable += 2;

// 字符串 (Strings) //

// 字符串字面量
let x: &str = "hello world!";

// 输出
println!("{}", f, x); // 1.3 hello world

// 一个 `String` - 在堆上分配空间的字符串
let s: String = "hello world".to_string();

// 字符串分片(slice) - 另一个字符串的不可变视图
// 基本上就是指向一个字符串的不可变指针, 它不包含字符串里任何内容, 只是一个指向某个东西的指针
// 比如这里就是 `s`
let s_slice: &str = &s;

println!("{}", s, s_slice); // hello world hello world

// 数组 (Vectors/arrays) //

// 长度固定的数组 (array)
let four_ints: [i32; 4] = [1, 2, 3, 4];

// 变长数组 (vector)
let mut vector: Vec<i32> = vec![1, 2, 3, 4];
vector.push(5);

// 分片 - 某个数组(vector/array)的不可变视图
// 和字符串分片基本一样, 只不过是针对数组的
let slice: &[i32] = &vector;

```

```

// 使用 `{:?}` 按调试样式输出
println!("{:?} {:?}", vector, slice); // [1, 2, 3, 4, 5] [1, 2, 3, 4, 5]

// 元组 (Tuples) //

// 元组是固定大小的一组值，可以是不同类型
let x: (i32, &str, f64) = (1, "hello", 3.4);

// 解构 `let`
let (a, b, c) = x;
println!("{}", a, b, c); // 1 hello 3.4

// 索引
println!("{}", x.1); // hello

//////////
// 2. 类型 (Type) //
//////////

// 结构体 (Struct)
struct Point {
    x: i32,
    y: i32,
}

let origin: Point = Point { x: 0, y: 0 };

// 匿名成员结构体，又叫“元组结构体”（‘tuple struct’）
struct Point2(i32, i32);

let origin2 = Point2(0, 0);

// 基础的 C 风格枚举类型 (enum)
enum Direction {
    Left,
    Right,
    Up,
    Down,
}

let up = Direction::Up;

// 有成员的枚举类型
enum OptionalI32 {
    AnI32(i32),
    Nothing,
}

let two: OptionalI32 = OptionalI32::AnI32(2);
let nothing = OptionalI32::Nothing;

```

```

// 泛型 (Generics) //

struct Foo<T> { bar: T }

// 这个在标准库里面有实现, 叫 `Option`
enum Optional<T> {
    SomeVal(T),
    NoVal,
}

// 方法 (Methods) //

impl<T> Foo<T> {
    // 方法需要一个显式的 `self` 参数
    fn get_bar(self) -> T {
        self.bar
    }
}

let a_foo = Foo { bar: 1 };
println!("{}", a_foo.get_bar()); // 1

// 接口 (Traits) (其他语言里叫 interfaces 或 typeclasses) //

trait Frobnicate<T> {
    fn frobnicate(self) -> Option<T>;
}

impl<T> Frobnicate<T> for Foo<T> {
    fn frobnicate(self) -> Option<T> {
        Some(self.bar)
    }
}

let another_foo = Foo { bar: 1 };
println!("{}", another_foo.frobnicate()); // Some(1)

////////////////////////////////////
// 3. 模式匹配 (Pattern matching) //
////////////////////////////////////

let foo = OptionalI32::AnI32(1);
match foo {
    OptionalI32::AnI32(n) => println!("it's an i32: {}", n),
    OptionalI32::Nothing => println!("it's nothing!"),
}

// 高级模式匹配
struct FooBar { x: i32, y: OptionalI32 }
let bar = FooBar { x: 15, y: OptionalI32::AnI32(32) };

```

```

match bar {
  FooBar { x: 0, y: OptionalI32::AnI32(0) } =>
    println!("The numbers are zero!"),
  FooBar { x: n, y: OptionalI32::AnI32(m) } if n == m =>
    println!("The numbers are the same"),
  FooBar { x: n, y: OptionalI32::AnI32(m) } =>
    println!("Different numbers: {} {}", n, m),
  FooBar { x: _, y: OptionalI32::Nothing } =>
    println!("The second number is Nothing!"),
}

```

```

////////////////////////////////////
// 4. 流程控制 (Control flow) //
////////////////////////////////////

```

```

// `for` 循环
let array = [1, 2, 3];
for i in array.iter() {
  println!("{}", i);
}

```

```

// 区间 (Ranges)
for i in 0u32..10 {
  print!("{}", i);
}
println!("");
// 输出 `0 1 2 3 4 5 6 7 8 9 `

```

```

// `if`
if 1 == 1 {
  println!("Maths is working!");
} else {
  println!("Oh no...");
}

```

```

// `if` 可以当表达式
let value = if true {
  "good"
} else {
  "bad"
};

```

```

// `while` 循环
while 1 == 1 {
  println!("The universe is operating normally.");
}

```

```

// 无限循环
loop {
  println!("Hello!");
}

```

```

}

////////////////////////////////////
// 5. 内存安全和指针 (Memory safety & pointers) //
////////////////////////////////////

// 独占指针 (Owned pointer) - 同一时刻只能有一个对象能“拥有”这个指针
// 意味着 `Box` 离开他的作用域后，会被安全地释放
let mut mine: Box<i32> = Box::new(3);
*mine = 5; // 解引用
// `now_its_mine` 获取了 `mine` 的所有权。换句话说，`mine` 移动 (move) 了
let mut now_its_mine = mine;
*now_its_mine += 2;

println!("{}", now_its_mine); // 7
// println!("{}", mine); // 编译报错，因为现在 `now_its_mine` 独占那个指针

// 引用 (Reference) - 引用其他数据的不可变指针
// 当引用指向某个值，我们称为“借用”这个值，因为是被不可变的借用，所以不能被修改，也不能移动
// 借用一直持续到生命周期结束，即离开作用域
let mut var = 4;
var = 3;
let ref_var: &i32 = &var;

println!("{}", var); // 不像 `box`，`var` 还可以继续使用
println!("{}", *ref_var);
// var = 5; // 编译报错，因为 `var` 被借用了
// *ref_var = 6; // 编译报错，因为 `ref_var` 是不可变引用

// 可变引用 (Mutable reference)
// 当一个变量被可变地借用时，也不可再用
let mut var2 = 4;
let ref_var2: &mut i32 = &mut var2;
*ref_var2 += 2;

println!("{}", *ref_var2); // 6
// var2 = 2; // 编译报错，因为 `var2` 被借用了
}

```

更深入的资料

Rust 还有很多很多其他内容 - 这只是 Rust 最基本的功能，帮助你了解 Rust 里面最重要的东西。如果想深入学习 Rust，可以去读 [The Rust Programming Language](#) 或者上 [reddit /r/rust](#) 订阅。同时 [irc.mozilla.org](#) 的 #rust 频道上的小伙伴们也非常欢迎新来的朋友。

你可以在这个在线编译器 [Rust playpen](#) 上尝试 Rust 的一些特性 或者上[官方网站](#)。

language: Scala filename: learnscala-zh.scala contributors:

```
- ["George Petrov", "http://github.com/petrovg"]
- ["Dominic Bou-Samra", "http://dbousamra.github.com"]
- ["Geoff Liu", "http://geoffliu.me"]
```

translators:

```
- ["Peiyong Lin", ""]
- ["Jinchang Ye", "http://github.com/alwayswithme"]
- ["Guodong Qu", "https://github.com/jasonqu"]
```

lang: zh-cn

Scala - 一门可拓展的语言

```
/*
 自行设置：

 1) 下载 Scala - http://www.scala-lang.org/downloads
 2) unzip/untar 到您喜欢的地方，并把 bin 子目录添加到 path 环境变量
 3) 在终端输入 scala，启动 Scala 的 REPL，您会看到提示符：

scala>

这就是所谓的 REPL（读取-求值-输出循环，英语：Read-Eval-Print Loop），
您可以在其中输入合法的表达式，结果会被打印。
在教程中我们会进一步解释 Scala 文件是怎样的，但现在先了解一点基础。
*/

////////////////////////////////////
// 1. 基础
////////////////////////////////////

// 单行注释开始于两个斜杠

/*
 多行注释，如您之前所见，看起来像这样
*/

// 打印并强制换行
println("Hello world!")
println(10)
```



```
// 没有强制换行的打印
print("Hello world")

// 通过 var 或者 val 来声明变量
// val 声明是不可变的，var 声明是可修改的。不可变性是好事。
val x = 10 // x 现在是 10
x = 20 // 错误：对 val 声明的变量重新赋值
var y = 10
y = 20 // y 现在是 20

/*
Scala 是静态语言，但注意上面的声明方式，我们没有指定类型。
这是因为类型推导的语言特性。大多数情况，Scala 编译器可以推测变量的类型，
所以您不需要每次都输入。可以像这样明确声明变量类型：
*/
val z: Int = 10
val a: Double = 1.0

// 注意从 Int 到 Double 的自动转型，结果是 10.0，不是 10
val b: Double = 10

// 布尔值
true
false

// 布尔操作
!true // false
!false // true
true == false // false
10 > 5 // true

// 数学运算像平常一样
1 + 1 // 2
2 - 1 // 1
5 * 3 // 15
6 / 2 // 3
6 / 4 // 1
6.0 / 4 // 1.5

// 在 REPL 计算一个表达式会返回给您结果的类型和值

1 + 7

/* 上行的结果是：

scala> 1 + 7
res29: Int = 8
```

这意味着计算 `1 + 7` 的结果是一个 `Int` 类型的对象，其值为 `8`

注意 "res29" 是一个连续生成的变量名，用以存储您输入的表达式结果，您看到的输出可能不一样。

```
*/

"Scala strings are surrounded by double quotes"
'a' // Scala 的字符
// '不存在单引号字符串' <= 这会导致错误

// String 有常见的 Java 字符串方法
"hello world".length
"hello world".substring(2, 6)
"hello world".replace("C", "3")

// 也有一些额外的 Scala 方法，另请参见: scala.collection.immutable.StringOps
"hello world".take(5)
"hello world".drop(5)

// 字符串改写：留意前缀 "s"
val n = 45
s"We have $n apples" // => "We have 45 apples"

// 在要改写的字符串中使用表达式也是可以的
val a = Array(11, 9, 6)
s"My second daughter is ${a(0) - a(2)} years old." // => "My second daughter is 5 years
s"We have double the amount of ${n / 2.0} in apples." // => "We have double the amount of
s"Power of 2: ${math.pow(2, 2)}" // => "Power of 2: 4"

// 添加 "f" 前缀对要改写的字符串进行格式化
f"Power of 5: ${math.pow(5, 2)}%1.0f" // "Power of 5: 25"
f"Square root of 122: ${math.sqrt(122)}%1.4f" // "Square root of 122: 11.0454"

// 未处理的字符串，忽略特殊字符。
raw"New line feed: \n. Carriage return: \r." // => "New line feed: \n. Carriage return:

// 一些字符需要转义，比如字符串中的双引号
"They stood outside the \"Rose and Crown\"" // => "They stood outside the "Rose and Crow

// 三个双引号可以使字符串跨越多行，并包含引号
val html = """<form id="daform">
    <p>Press belo', Joe</p>
    <input type="submit">
</form>"""

////////////////////////////////////
// 2. 函数
////////////////////////////////////

// 函数可以这样定义：
//
// def functionName(args...): ReturnType = { body... }
```

```
//
// 如果您以前学习过传统的编程语言，注意 return 关键字的省略。
// 在 Scala 中，函数代码块最后一条表达式就是返回值。
def sumOfSquares(x: Int, y: Int): Int = {
    val x2 = x * x
    val y2 = y * y
    x2 + y2
}

// 如果函数体是单行表达式，{ } 可以省略：
def sumOfSquaresShort(x: Int, y: Int): Int = x * x + y * y

// 函数调用的语法是熟知的：
sumOfSquares(3, 4) // => 25

// 在多数情况下（递归函数是需要注意的例外），函数返回值可以省略，
// 变量所用的类型推导一样会应用到函数返回值中：
def sq(x: Int) = x * x // 编译器会推断得知返回值是 Int

// 函数可以有默认参数
def addWithDefault(x: Int, y: Int = 5) = x + y
addWithDefault(1, 2) // => 3
addWithDefault(1) // => 6

// 匿名函数是这样的：
(x: Int) => x * x

// 和 def 不同，如果语义清晰，匿名函数的参数类型也可以省略。
// 类型 "Int => Int" 意味着这个函数接收一个 Int 并返回一个 Int。
val sq: Int => Int = x => x * x

// 匿名函数的调用也是类似的：
sq(10) // => 100

// 如果您的匿名函数中每个参数仅使用一次，
// Scala 提供一个更简洁的方式来定义他们。这样的匿名函数极为常见，
// 在数据结构部分会明显可见。
val addOne: Int => Int = _ + 1
val weirdSum: (Int, Int) => Int = (_ * 2 + _ * 3)

addOne(5) // => 6
weirdSum(2, 4) // => 16

// return 关键字是存在的，但它只从最里面包裹了 return 的 def 函数中返回。
// 警告：在 Scala 中使用 return 容易出错，应该避免使用。
// 在匿名函数中没有效果，例如：
def foo(x: Int): Int = {
    val anonFunc: Int => Int = { z =>
        if (z > 5)

```

```

    return z // 这一行令 z 成为 foo 函数的返回值!
  else
    z + 2 // 这一行是 anonFunc 函数的返回值
  }
anonFunc(x) // 这一行是 foo 函数的返回值
}

/*
 * 译者注：此处是指匿名函数中的 return z 成为最后执行的语句，
 * 在 anonFunc(x) 下面的表达式（假设存在）不再执行。如果 anonFunc
 * 是用 def 定义的函数，return z 仅返回到 anonFunc(x)，
 * 在 anonFunc(x) 下面的表达式（假设存在）会继续执行。
 */

////////////////////////////////////
// 3. 控制语句
////////////////////////////////////

1 to 5
val r = 1 to 5
r.foreach( println )

r foreach println
// 附注： Scala 对点和括号的要求想当宽松，注意其规则是不同的。
// 这有助于写出读起来像英语的 DSL(领域特定语言) 和 API(应用编程接口)。

(5 to 1 by -1) foreach ( println )

// while 循环
var i = 0
while (i < 10) { println("i " + i); i+=1 }

while (i < 10) { println("i " + i); i+=1 } // 没错，再执行一次，发生了什么？为什么？

i // 显示 i 的值。注意 while 是经典的循环方式，它连续执行并改变循环中的变量。
// while 执行很快，比 Java 的循环快，但像上面所看到的那样用组合子和推导式
// 更易于理解和并行化。

// do while 循环
do {
  println("x is still less than 10");
  x += 1
} while (x < 10)

// Scala 中尾递归是一种符合语言习惯的递归方式。
// 递归函数需要清晰的返回类型，编译器不能推断得知。
// 这是一个 Unit。
def showNumbersInRange(a:Int, b:Int):Unit = {
  print(a)
  if (a < b)

```

```

    showNumbersInRange(a + 1, b)
}
showNumbersInRange(1,14)

// 条件语句

val x = 10

if (x == 1) println("yeah")
if (x == 10) println("yeah")
if (x == 11) println("yeah")
if (x == 11) println("yeah") else println("nay")

println(if (x == 10) "yeah" else "nope")
val text = if (x == 10) "yeah" else "nope"

////////////////////////////////////
// 4. 数据结构
////////////////////////////////////

val a = Array(1, 2, 3, 5, 8, 13)
a(0)
a(3)
a(21)    // 抛出异常

val m = Map("fork" -> "tenedor", "spoon" -> "cuchara", "knife" -> "cuchillo")
m("fork")
m("spoon")
m("bottle")    // 抛出异常

val safeM = m.withDefaultValue("no lo se")
safeM("bottle")

val s = Set(1, 3, 7)
s(0)
s(1)

/* 这里查看 map 的文档 -
 * http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Map
 * 并确保你会阅读
 */

// 元组

(1, 2)

(4, 3, 2)

```

```

(1, 2, "three")

(a, 2, "three")

// 为什么有这个?
val divideInts = (x:Int, y:Int) => (x / y, x % y)

divideInts(10,3) // 函数 divideInts 同时返回结果和余数

// 要读取元组的元素, 使用 _._n, n是从1开始的元素索引
val d = divideInts(10,3)

d._1

d._2

////////////////////////////////////
// 5. 面向对象编程
////////////////////////////////////

/*
  旁白: 教程中到现在为止我们所做的一切只是简单的表达式(值, 函数等)。
  这些表达式可以输入到命令行解释器中作为快速测试, 但它们不能独立存在于 Scala
  文件。举个例子, 您不能在 Scala 文件上简单的写上 "val x = 5"。相反 Scala 文件
  允许的顶级结构是:

  - objects
  - classes
  - case classes
  - traits

  现在来解释这些是什么。
*/

// 类和其他语言的类相似, 构造器参数在类名后声明, 初始化在类结构体中完成。
class Dog(br: String) {
  // 构造器代码在此
  var breed: String = br

  // 定义名为 bark 的方法, 返回字符串
  def bark = "Woof, woof!"

  // 值和方法作用域假定为 public。"protected" 和 "private" 关键字也是可用的。
  private def sleep(hours: Int) =
    println(s"I'm sleeping for $hours hours")

  // 抽象方法是没有方法体的方法。如果取消下面那行注释, Dog 类必须被声明为 abstract
  // abstract class Dog(...) { ... }
  // def chaseAfter(what: String): String
}

```

```
val mydog = new Dog("greyhound")
println(mydog.breed) // => "greyhound"
println(mydog.bark) // => "Woof, woof!"
```

// "object" 关键字创造一种类型和该类型的单例。
// Scala 的 class 常常也含有一个“伴生对象”，class 中包含每个实例的行为，所有实例
// 共用的行为则放入 object 中。两者的区别和其他语言中类方法和静态方法类似。
// 请注意 object 和 class 可以同名。

```
object Dog {
  def allKnownBreeds = List("pitbull", "shepherd", "retriever")
  def createDog(breed: String) = new Dog(breed)
}
```

// Case 类是有额外内建功能的类。Scala 初学者常遇到的问题之一便是何时用类
// 和何时用 case 类。界线比较模糊，但通常类倾向于封装，多态和行为。类中的值
// 的作用域一般为 private，只有方法是暴露的。case 类的主要目的是放置不可变
// 数据。它们通常只有几个方法，且方法几乎没有副作用。

```
case class Person(name: String, phoneNumber: String)
```

// 创造新实例，注意 case 类不需要使用 "new" 关键字

```
val george = Person("George", "1234")
val kate = Person("Kate", "4567")
```

// 使用 case 类，您可以轻松得到一些功能，像 getters：
george.phoneNumber // => "1234"

// 每个字段的相等性比较（无需覆盖 .equals）
Person("George", "1234") == Person("Kate", "1236") // => false

// 简单的拷贝方式
// otherGeorge == Person("george", "9876")
val otherGeorge = george.copy(phoneNumber = "9876")

// 还有很多。case 类同时可以用于模式匹配，接下来会看到。

// 敬请期待 Traits !

```
////////////////////////////////////
// 6. 模式匹配
////////////////////////////////////
```

// 模式匹配是一个强大和常用的 Scala 特性。这是用模式匹配一个 case 类的例子。
// 附注：不像其他语言，Scala 的 case 不需要 break，其他语言中 switch 语句的
// fall-through 现象不会发生。

```
def matchPerson(person: Person): String = person match {
```

```

// Then you specify the patterns:
case Person("George", number) => "We found George! His number is " + number
case Person("Kate", number) => "We found Kate! Her number is " + number
case Person(name, number) => "We matched someone : " + name + ", phone : " + number
}

val email = "(.*)@(.*)".r // 定义下一个例子会用到的正则

// 模式匹配看起来和 C 语言家族的 switch 语句相似，但更为强大。
// Scala 中您可以匹配很多东西：
def matchEverything(obj: Any): String = obj match {
  // 匹配值：
  case "Hello world" => "Got the string Hello world"

  // 匹配类型：
  case x: Double => "Got a Double: " + x

  // 匹配时指定条件
  case x: Int if x > 10000 => "Got a pretty big number!"

  // 像之前一样匹配 case 类：
  case Person(name, number) => s"Got contact info for $name!"

  // 匹配正则表达式：
  case email(name, domain) => s"Got email address $name@$domain"

  // 匹配元组：
  case (a: Int, b: Double, c: String) => s"Got a tuple: $a, $b, $c"

  // 匹配数据结构：
  case List(1, b, c) => s"Got a list with three elements and starts with 1: 1, $b, $c"

  // 模式可以嵌套
  case List(List((1, 2, "YAY"))) => "Got a list of list of tuple"
}

// 事实上，你可以对任何有 "unapply" 方法的对象进行模式匹配。
// 这个特性如此强大以致于 Scala 允许定义一个函数作为模式匹配：
val patternFunc: Person => String = {
  case Person("George", number) => s"George's number: $number"
  case Person(name, number) => s"Random person's number: $number"
}

////////////////////////////////////
// 7. 函数式编程
////////////////////////////////////

// Scala 允许方法和函数作为其他方法和函数的参数和返回值。

val add10: Int => Int = _ + 10 // 一个接受一个 Int 类型参数并返回一个 Int 类型值的函数

```



```

List(1, 2, 3) map add10 // List(11, 12, 13) - add10 被应用到每一个元素

// 匿名函数可以被使用来代替有命名的函数:
List(1, 2, 3) map (x => x + 10)

// 如果匿名函数只有一个参数可以用下划线作为变量
List(1, 2, 3) map (_ + 10)

// 如果您所应用的匿名块和匿名函数都接受一个参数, 那么你甚至可以省略下划线
List("Dom", "Bob", "Natalia") foreach println

// 组合子

// 译注: val sq: Int => Int = x => x * x
s.map(sq)

val sSquared = s. map(sq)

sSquared.filter(_ < 10)

sSquared.reduce (_+_ )

// filter 函数接受一个 predicate (函数根据条件 A 返回 Boolean) 并选择
// 所有满足 predicate 的元素
List(1, 2, 3) filter (_ > 2) // List(3)
case class Person(name:String, age:Int)
List(
  Person(name = "Dom", age = 23),
  Person(name = "Bob", age = 30)
).filter(_.age > 25) // List(Person("Bob", 30))

// Scala 的 foreach 方法定义在某些集合中, 接受一个函数并返回 Unit (void 方法)
// 另请参见:
// http://www.scala-lang.org/api/current/index.html#scala.collection.IterableLike@foreach
val aListOfNumbers = List(1, 2, 3, 4, 10, 20, 100)
aListOfNumbers foreach (x => println(x))
aListOfNumbers foreach println

// For 推导式

for { n <- s } yield sq(n)

val nSquared2 = for { n <- s } yield sq(n)

for { n <- nSquared2 if n < 10 } yield n

for { n <- s; nSquared = n * n if nSquared < 10 } yield nSquared

/* 注意, 这些不是 for 循环, for 循环的语义是‘重复’, 然而 for 推导式定义

```

```
两个数据集的关系。 */
```

```
////////////////////////////////////
```

```
// 8. 隐式转换
```

```
////////////////////////////////////
```

```
/* 警告 警告：隐式转换是 Scala 中一套强大的特性，因此容易被滥用。
```

```
 * Scala 初学者在理解它们的工作原理和最佳实践之前，应抵制使用它的诱惑。
```

```
 * 我们加入这一章节仅因为它们在 Scala 的库中太过常见，导致没有用隐式转换的库
```

```
 * 就不可能做有意义的事情。这章节主要让你理解和使用隐式转换，而不是自己声明。
```

```
*/
```

```
// 可以通过 "implicit" 声明任何值（val，函数，对象等）为隐式值，
```

```
// 请注意这些例子中，我们用到第5部分的 Dog 类。
```

```
implicit val myImplicitInt = 100
```

```
implicit def myImplicitFunction(breed: String) = new Dog("Golden " + breed)
```

```
// implicit 关键字本身不改变值的行为，所以上面的值可以照常使用。
```

```
myImplicitInt + 2 // => 102
```

```
myImplicitFunction("Pitbull").breed // => "Golden Pitbull"
```

```
// 区别在于，当另一段代码“需要”隐式值时，这些值现在有资格作为隐式值。
```

```
// 一种情况是隐式函数参数。
```

```
def sendGreetings(toWhom: String)(implicit howMany: Int) =
```

```
  s"Hello $toWhom, $howMany blessings to you and yours!"
```

```
// 如果提供值给 “howMany”，函数正常运行
```

```
sendGreetings("John")(1000) // => "Hello John, 1000 blessings to you and yours!"
```

```
// 如果省略隐式参数，会传一个和参数类型相同的隐式值，
```

```
// 在这个例子中，是 “myImplicitInt”：
```

```
sendGreetings("Jane") // => "Hello Jane, 100 blessings to you and yours!"
```

```
// 隐式的函数参数使我们可以模拟其他函数式语言的 type 类（type classes）。
```

```
// 它经常被用到所以有特定的简写。这两行代码是一样的：
```

```
def foo[T](implicit c: C[T]) = ...
```

```
def foo[T : C] = ...
```

```
// 编译器寻找隐式值另一种情况是你调用方法时
```

```
// obj.method(...)
```

```
// 但 "obj" 没有一个名为 "method" 的方法。这样的话，如果有一个参数类型为 A
```

```
// 返回值类型为 B 的隐式转换，obj 的类型是 A，B 有一个方法叫 "method"，这样
```

```
// 转换就会被应用。所以作用域里有上面的 myImplicitFunction，我们可以这样做：
```

```
"Retriever".breed // => "Golden Retriever"
```

```
"Sheperd".bark // => "Woof, woof!"
```

```
// 这里字符串先被上面的函数转换为 Dog 对象，然后调用相应的方法。
```

```
// 这是相当强大的特性，但再次提醒，请勿轻率使用。
```

```
// 事实上，当你定义上面的隐式函数时，编译器会作出警告，除非你真的了解
```

```
// 你正在做什么否则不要使用。
```

```

////////////////////////////////////
// 9. 杂项
////////////////////////////////////

// 导入类
import scala.collection.immutable.List

// 导入所有子包
import scala.collection.immutable._

// 一条语句导入多个类
import scala.collection.immutable.{List, Map}

// 使用 ‘=>’ 对导入进行重命名
import scala.collection.immutable.{ List => ImmutableList }

// 导入所有类，排除其中一些。下面的语句排除了 Map 和 Set:
import scala.collection.immutable.{Map => _, Set => _, _}

// 在 Scala 文件用 object 和单一的 main 方法定义程序入口:
object Application {
  def main(args: Array[String]): Unit = {
    // stuff goes here.
  }
}

// 文件可以包含多个 class 和 object, 用 scalac 编译源文件


// 输入和输出

// 按行读文件
import scala.io.Source
for(line <- Source.fromFile("myfile.txt").getLines())
  println(line)

// 用 Java 的 PrintWriter 写文件
val writer = new PrintWriter("myfile.txt")
writer.write("Writing line for line" + util.Properties.lineSeparator)
writer.write("Another line here" + util.Properties.lineSeparator)
writer.close()

```

更多的资源

为没耐心的人准备的 [Scala](#)

[Twitter Scala school](#)

[The Scala documentation](#)

在浏览器尝试 [Scala](#)

加入 [Scala](#) 用户组

language: swift filename: learnswift-cn.swift contributors:

- ["Grant Timmerman", "<http://github.com/grant>"] translators:
 - ["Xavier Yao", "<http://github.com/xavieryao>"]
 - ["Joey Huang", "<http://github.com/kamidox>"]
 - ["CY Lim", "<http://github.com/cylim>"] lang: zh-cn
-

Swift 是 Apple 开发的用于 iOS 和 OS X 开发的编程语言。Swift 于2014年 Apple WWDC（全球开发者大会）中被引入，用以与 Objective-C 共存，同时对错误代码更具弹性。Swift 由 Xcode 6 beta 中包含的 LLVM 编译器编译。

Swift 的官方语言教程 [Swift Programming Language](#) 可以从 iBooks 免费下载.

亦可参阅: [Apple's getting started guide](#) ——一个完整的Swift 教程

```
// 导入外部模块
import UIKit

//
// MARK: 基础
//

// XCODE 支持给注释代码作标记，这些标记会列在 XCODE 的跳转栏里，支持的标记为
// MARK: 普通标记
// TODO: TODO 标记
// FIXME: FIXME 标记

// Swift2.0 println() 及 print() 已经整合成 print()。
print("Hello, world") // 这是原本的 println(), 会自动进入下一行
print("Hello, world", terminator: "") // 如果不要自动进入下一行，需设定结束符为空串

// 变量 (var) 的值设置后可以随意改变
// 常量 (let) 的值设置后不能改变
var myVariable = 42
let π = "value" // 可以支持 unicode 变量名
let π = 3.1415926
let myConstant = 3.1415926
let explicitDouble: Double = 70 // 明确指定变量类型为 Double，否则编译器将自动推断变量类型
let weak = "keyword"; let override = "another keyword" // 语句之间可以用分号隔开，语句末尾不需分号
let intValue = 0007 // 7
let largeIntValue = 77_000 // 77000
let label = "some text " + String(myVariable) // 类型转换
let piText = "Pi = \(π), Pi 2 = \(π * 2)" // 格式化字符串

// 条件编译
// 使用 -D 定义编译开关
```

```

    #if false
        print("Not printed")
        let buildValue = 3
    #else
        let buildValue = 7
    #endif
    print("Build value: \(buildValue)") // Build value: 7

    /*
        Optionals 是 Swift 的新特性，它允许你存储两种状态的值给 Optional 变量：有效值或 None 。
        可在值名称后加个问号 (?) 来表示这个值是 Optional。

        Swift 要求所有的 Optional 属性都必须有明确的值，如果为空，则必须明确设定为 nil

        Optional<T> 是个枚举类型
    */
    var someOptionalString: String? = "optional" // 可以是 nil
    // 下面的语句和上面完全等价，上面的写法更推荐，因为它更简洁，问号 (?) 是 Swift 提供的语法糖
    var someOptionalString2: Optional<String> = "optional"

    if someOptionalString != nil {
        // 变量不为空
        if someOptionalString!.hasPrefix("opt") {
            print("has the prefix")
        }

        let empty = someOptionalString?.isEmpty
    }
    someOptionalString = nil

    /*
        使用 (!) 可以解决无法访问optional值的运行错误。若要使用 (!) 来强制解析，一定要确保 Optional
    */

    // 显式解包 optional 变量
    var unwrappedString: String! = "Value is expected."
    // 下面语句和上面完全等价，感叹号 (!) 是个后缀运算符，这也是个语法糖
    var unwrappedString2: ImplicitlyUnwrappedOptional<String> = "Value is expected."

    if let someOptionalStringConstant = someOptionalString {
        // 由于变量 someOptionalString 有值，不为空，所以 if 条件为真
        if !someOptionalStringConstant.hasPrefix("ok") {
            // does not have the prefix
        }
    }
}

// Swift 支持可保存任何数据类型的变量
// AnyObject == id
// 和 Objective-C `id` 不一样，AnyObject 可以保存任何类型的值 (Class, Int, struct, 等)
var anyObjectVar: AnyObject = 7
anyObjectVar = "Changed value to a string, not good practice, but possible."

```

```

/*
    这里是注释

    /*
        支持嵌套的注释
    */
*/

//
// Mark: 数组与字典（关联数组）
//

/*
    Array 和 Dictionary 是结构体，不是类，他们作为函数参数时，是用值传递而不是指针传递。
    可以用 `var` 和 `let` 来定义变量和常量。
*/

// Array
var shoppingList = ["catfish", "water", "lemons"]
shoppingList[1] = "bottle of water"
let emptyArray = [String]() // 使用 let 定义常量，此时 emptyArray 数组不能添加或删除内容
let emptyArray2 = Array<String>() // 与上一语句等价，上一语句更常用
var emptyMutableArray = [String]() // 使用 var 定义变量，可以向 emptyMutableArray 添加数组元
var explicitEmptyMutableStringArray: [String] = [] // 与上一语句等价

// 字典
var occupations = [
    "Malcolm": "Captain",
    "kaylee": "Mechanic"
]
occupations["Jayne"] = "Public Relations" // 修改字典，如果 key 不存在，自动添加一个字典元素
let emptyDictionary = [String: Float]() // 使用 let 定义字典常量，字典常量不能修改里面的值
let emptyDictionary2 = Dictionary<String, Float>() // 与上一语句类型等价，上一语句更常用
var emptyMutableDictionary = [String: Float]() // 使用 var 定义字典变量
var explicitEmptyMutableDictionary: [String: Float] = [:] // 与上一语句类型等价

//
// MARK: 控制流
//

// 数组的 for 循环
let myArray = [1, 1, 2, 3, 5]
for value in myArray {
    if value == 1 {
        print("One!")
    } else {
        print("Not one!")
    }
}

```

```

}

// 字典的 for 循环
var dict = ["one": 1, "two": 2]
for (key, value) in dict {
    print("\(key): \(value)")
}

// 区间的 loop 循环: 其中 `...` 表示闭环区间, 即[-1, 3]; `..<` 表示半开闭区间, 即[-1,3)
for i in -1...shoppingList.count {
    print(i)
}
shoppingList[1...2] = ["steak", "peacons"]
// 可以使用 `..<` 来去掉最后一个元素

// while 循环
var i = 1
while i < 1000 {
    i *= 2
}

// repeat-while 循环
repeat {
    print("hello")
} while 1 == 2

// Switch 语句
// Swift 里的 Switch 语句功能异常强大, 结合枚举类型, 可以实现非常简洁的代码, 可以把 switch 语句想象
// 它支持字符串, 类实例或原生数据类型 (Int, Double, etc)
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let localScopeValue where localScopeValue.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(localScopeValue)?"
default: // 在 Swift 里, switch 语句的 case 必须处理所有可能的情况, 如果 case 无法全部处理, 则必
    let vegetableComment = "Everything tastes good in soup."
}

//
// MARK: 函数
//

// 函数是一个 first-class 类型, 他们可以嵌套, 可以作为函数参数传递

// 函数文档可使用 reStructuredText 格式直接写在函数的头部
/**
    A greet operation

```



```

- A bullet in docs
- Another bullet in the docs

:param: name A name
:param: day A day
:returns: A string containing the name and day value.
*/
func greet(name: String, day: String) -> String {
    return "Hello \$(name), today is \$(day)."
}
greet("Bob", day: "Tuesday")

// 第一个参数表示外部参数名和内部参数名使用同一个名称。
// 第二个参数表示外部参数名使用 `externalParamName`，内部参数名使用 `localParamName`
func greet2(requiredName requiredName: String, externalParamName localParamName: String)
    return "Hello \$(requiredName), the day is \$(localParamName)"
}
greet2(requiredName:"John", externalParamName: "Sunday")    // 调用时，使用命名参数来指定参数

// 函数可以通过元组 (tuple) 返回多个值
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
let pricesTuple = getGasPrices()
let price = pricesTuple.2 // 3.79
// 通过下划线 (_) 来忽略不关心的值
let (_, price1, _) = pricesTuple // price1 == 3.69
print(price1 == pricesTuple.1) // true
print("Gas price: \$(price)")

// 可变参数
func setup(numbers: Int...) {
    // 可变参数是个数组
    let _ = numbers[0]
    let _ = numbers.count
}

// 函数变量以及函数作为返回值返回
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)

// 强制进行指针传递 (引用传递)，使用 `inout` 关键字修饰函数参数
func swapTwoInts(inout a: Int, inout b: Int) {
    let tempA = a

```

```

    a = b
    b = tempA
}
var someIntA = 7
var someIntB = 3
swapTwoInts(&someIntA, b: &someIntB)
print(someIntB) // 7


//
// MARK: 闭包
//
var numbers = [1, 2, 6]

// 函数是闭包的一个特例 ({}))

// 闭包实例
// `->` 分隔了闭包的参数和返回值
// `in` 分隔了闭包头 (包括参数及返回值) 和闭包体
// 下面例子中, `map` 的参数是一个函数类型, 它的功能是把数组里的元素作为参数, 逐个调用 `map` 参数传递
numbers.map({
    (number: Int) -> Int in
    let result = 3 * number
    return result
})

// 当闭包的参数类型和返回值都是已知的情况下, 且只有一个语句作为其返回值时, 我们可以简化闭包的写法
numbers = numbers.map({ number in 3 * number })
// 我们也可以使用 $0, $1 来指代第 1 个, 第 2 个参数, 上面的语句最终可简写为如下形式
// numbers = numbers.map({ $0 * 3 })

print(numbers) // [3, 6, 18]

// 简洁的闭包
numbers = numbers.sort { $0 > $1 }

print(numbers) // [18, 6, 3]


//
// MARK: 结构体
//

// 结构体和类非常类似, 可以有属性和方法

struct NamesTable {
    let names: [String]

    // 自定义下标运算符
    subscript(index: Int) -> String {
        return names[index]
    }
}

```

```

    }
}

// 结构体有一个自动生成的隐含的命名构造函数
let namesTable = NamesTable(names: ["Me", "Them"])
let name = namesTable[1]
print("Name is \(name)") // Name is Them

//
// MARK: 类
//

// 类和结构体的有三个访问控制级别，他们分别是 internal (默认), public, private
// internal: 模块内部可以访问
// public: 其他模块可以访问
// private: 只有定义这个类或结构体的源文件才能访问

public class Shape {
    public func getArea() -> Int {
        return 0;
    }
}

// 类的所有方法和属性都是 public 的
// 如果你只是需要把数据保存在一个结构化的实例里面，应该用结构体

internal class Rect: Shape {
    // 值属性 (Stored properties)
    var sideLength: Int = 1

    // 计算属性 (Computed properties)
    private var perimeter: Int {
        get {
            return 4 * sideLength
        }
        set {
            // `newValue` 是个隐含的变量，它表示将要设置进来的新值
            sideLength = newValue / 4
        }
    }
}

// 延时加载的属性，只有这个属性第一次被引用时才进行初始化，而不是定义时就初始化
// subShape 值为 nil，直到 subShape 第一次被引用时才初始化为一个 Rect 实例
lazy var subShape = Rect(sideLength: 4)

// 监控属性值的变化。
// 当我们需要在属性值改变时做一些事情，可以使用 `willSet` 和 `didSet` 来设置监控函数
// `willSet`: 值改变之前被调用
// `didSet`: 值改变之后被调用
var identifier: String = "defaultID" {
    // `willSet` 的参数是即将设置的新值，参数名可以指定，如果没有指定，就是 `newValue`

```

```

willSet(someIdentifier) {
    print(someIdentifier)
}
// `didSet` 的参数是已经被覆盖掉的旧的值，参数名也可以指定，如果没有指定，就是 `oldValue`
didSet {
    print(oldValue)
}
}

// 命名构造函数 (designated inits)，它必须初始化所有的成员变量，
// 然后调用父类的命名构造函数继续初始化父类的所有变量。
init(sideLength: Int) {
    self.sideLength = sideLength
    // 必须显式地在构造函数最后调用父类的构造函数 super.init
    super.init()
}

func shrink() {
    if sideLength > 0 {
        --sideLength
    }
}

// 函数重载使用 override 关键字
override func getArea() -> Int {
    return sideLength * sideLength
}
}

// 类 `Square` 从 `Rect` 继承
class Square: Rect {
    // 便捷构造函数 (convenience inits) 是调用自己的命名构造函数 (designated inits) 的构造函数
    // Square 自动继承了父类的命名构造函数
    convenience init() {
        self.init(sideLength: 5)
    }
    // 关于构造函数的继承，有以下几个规则：
    // 1. 如果你没有实现任何命名构造函数，那么你就继承了父类的所有命名构造函数
    // 2. 如果你重载了父类的所有命名构造函数，那么你就自动继承了所有的父类快捷构造函数
    // 3. 如果你没有实现任何构造函数，那么你继承了父类的所有构造函数，包括命名构造函数和便捷构造函数
}

var mySquare = Square()
print(mySquare.getArea()) // 25
mySquare.shrink()
print(mySquare.sideLength) // 4

// 类型转换
let aShape = mySquare as Shape

// 使用三个等号来比较是不是同一个实例

```

```

if mySquare === aShape {
    print("Yep, it's mySquare")
}

class Circle: Shape {
    var radius: Int
    override func getArea() -> Int {
        return 3 * radius * radius
    }

    // optional 构造函数, 可能会返回 nil
    init?(radius: Int) {
        self.radius = radius
        super.init()

        if radius <= 0 {
            return nil
        }
    }
}

// 根据 Swift 类型推断, myCircle 是 Optional<Circle> 类型的变量
var myCircle = Circle(radius: 1)
print(myCircle?.getArea())    // Optional(3)
print(myCircle!.getArea())    // 3
var myEmptyCircle = Circle(radius: -1)
print(myEmptyCircle?.getArea()) // "nil"
if let circle = myEmptyCircle {
    // 此语句不会输出, 因为 myEmptyCircle 变量值为 nil
    print("circle is not nil")
}

//
// MARK: 枚举
//

// 枚举可以像类一样, 拥有方法

enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func getIcon() -> String {
        switch self {
            case .Spades: return "♠"
            case .Hearts: return "♥"
            case .Diamonds: return "♦"
            case .Clubs: return "♣"
        }
    }
}

```

```

// 当变量类型明确指定为某个枚举类型时，赋值时可以省略枚举类型
var suitValue: Suit = .Hearts

// 非整型的枚举类型需要在定义时赋值
enum BookName: String {
    case John = "John"
    case Luke = "Luke"
}
print("Name: \$(BookName.John.rawValue)")

// 与特定数据类型关联的枚举
enum Furniture {
    // 和 Int 型数据关联的枚举记录
    case Desk(height: Int)
    // 和 String, Int 关联的枚举记录
    case Chair(brand: String, height: Int)

    func description() -> String {
        switch self {
            case .Desk(let height):
                return "Desk with \$(height) cm"
            case .Chair(let brand, let height):
                return "Chair of \$(brand) with \$(height) cm"
        }
    }
}

var desk: Furniture = .Desk(height: 80)
print(desk.description()) // "Desk with 80 cm"
var chair = Furniture.Chair(brand: "Foo", height: 40)
print(chair.description()) // "Chair of Foo with 40 cm"

//
// MARK: 协议
// 与 Java 的 interface 类似
//

// 协议可以让遵循同一协议的类型实例拥有相同的属性，方法，类方法，操作符或下标运算符等
// 下面代码定义一个协议，这个协议包含一个名为 enabled 的计算属性且包含 buildShape 方法
protocol ShapeGenerator {
    var enabled: Bool { get set }
    func buildShape() -> Shape
}

// 协议声明时可以添加 @objc 前缀，添加 @objc 前缀后，
// 可以使用 is, as, as? 等来检查协议兼容性
// 需要注意，添加 @objc 前缀后，协议就只能被类来实现，
// 结构体和枚举不能实现加了 @objc 的前缀
// 只有添加了 @objc 前缀的协议才能声明 optional 方法
// 一个类实现一个带 optional 方法的协议时，可以实现或不实现这个方法

```

```

// optional 方法可以使用 optional 规则来调用
@objc protocol TransformShape {
    optional func reshape()
    optional func canReshape() -> Bool
}

class MyShape: Rect {
    var delegate: TransformShape?

    func grow() {
        sideLength += 2

        // 在 optional 属性，方法或下标运算符后面加一个问号，可以优雅地忽略 nil 值，返回 nil。
        // 这样就不会引起运行时错误 (runtime error)
        if let reshape = self.delegate?.canReshape?() where reshape {
            // 注意语句中的问号
            self.delegate?.reshape?()
        }
    }
}

//
// MARK: 其它
//

// 扩展：给一个已经存在的数据类型添加功能

// 给 Square 类添加 `CustomStringConvertible` 协议的实现，现在其支持 `CustomStringConvertible`
extension Square: CustomStringConvertible {
    var description: String {
        return "Area: \(self.getArea()) - ID: \(self.identifier)"
    }
}

print("Square: \(mySquare)") // Area: 16 - ID: defaultID

// 也可以给系统内置类型添加功能支持
extension Int {
    var customProperty: String {
        return "This is \(self)"
    }

    func multiplyBy(num: Int) -> Int {
        return num * self
    }
}

print(7.customProperty) // "This is 7"
print(14.multiplyBy(3)) // 42

```

```

// 泛型：和 Java 及 C# 的泛型类似，使用 `where` 关键字来限制类型。
// 如果只有一个类型限制，可以省略 `where` 关键字
func findIndex<T: Equatable>(array: [T], _ valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}

let foundAtIndex = findIndex([1, 2, 3, 4], 3)
print(foundAtIndex == 2) // true

// 自定义运算符：
// 自定义运算符可以以下面的字符打头：
//      / = - + * % < > ! & | ^ . ~
// 甚至是 Unicode 的数学运算符等
prefix operator !!! {}

// 定义一个前缀运算符，使矩形的边长放大三倍
prefix func !!! (inout shape: Square) -> Square {
    shape.sideLength *= 3
    return shape
}

// 当前值
print(mySquare.sideLength) // 4

// 使用自定义的 !!! 运算符来把矩形边长放大三倍
!!!mySquare
print(mySquare.sideLength) // 12

// 运算符也可以是泛型
infix operator <-> {}
func <-><T: Equatable> (inout a: T, inout b: T) {
    let c = a
    a = b
    b = c
}

var foo: Float = 10
var bar: Float = 20

foo <-> bar
print("foo is \$(foo), bar is \$(bar)") // "foo is 20.0, bar is 10.0"

```

category: tool tool: tmux filename: LearnTmux-cn.txt contributors:

```
- ["mdl", "https://github.com/mdl"]
```

translators:

```
- ["Arnie97", "https://github.com/Arnie97"]
```

lang: zh-cn

[tmux](#)是一款终端复用工具。在它的帮助下，你可以在同一个控制台上建立、访问并控制多个终端。你可以断开与一个 **tmux** 终端的连接，此时程序将在后台运行，当你需要时，可以随时重新连接到这个终端。

```

tmux [command]      # 运行一条命令
                    # 如果单独使用 'tmux' 而不指定某个命令，将会建立一个新的会话

new                  # 创建一个新的会话
  -s "Session"       # 创建一个会话，并命名为“Session”
  -n "Window"        # 创建一个窗口，并命名为“Window”
  -c "/dir"          # 在指定的工作目录中启动会话

attach              # 连接到上一次的会话（如果可用）
  -t "#"             # 连接到指定的会话
  -d                 # 断开其他客户端的会话

ls                  # 列出打开的会话
  -a                 # 列出所有打开的会话

lsw                 # 列出窗口
  -a                 # 列出所有窗口
  -s                 # 列出会话中的所有窗口

lsp                 # 列出窗格
  -a                 # 列出所有窗格
  -s                 # 列出会话中的所有窗格
  -t "#"             # 列出指定窗口中的所有窗格

kill-window         # 关闭当前窗口
  -t "#"             # 关闭指定的窗口
  -a                 # 关闭所有窗口
  -a -t "#"          # 关闭除指定窗口以外的所有窗口

kill-session        # 关闭当前会话
  -t "#"             # 关闭指定的会话
  -a                 # 关闭所有会话
  -a -t "#"          # 关闭除指定会话以外的所有会话

```

快捷键

通过“前缀”快捷键，可以控制一个已经连入的 **tmux** 会话。

(C-b) = Ctrl + b # 在使用下列快捷键之前，需要按这个“前缀”快捷键

(M-1) = Meta + 1 或 Alt + 1

?	# 列出所有快捷键
:	# 进入 tmux 的命令提示符
r	# 强制重绘当前客户端
c	# 创建一个新窗口
!	# 将当前窗格从窗口中移出，成为为一个新的窗口
%	# 将当前窗格分为左右两半
"	# 将当前窗格分为上下两半
n	# 切换到下一个窗口
p	# 切换到上一个窗口
{	# 将当前窗格与上一个窗格交换
}	# 将当前窗格与下一个窗格交换
s	# 在交互式界面中，选择并连接至另一个会话
w	# 在交互式界面中，选择并激活一个窗口
0 至 9	# 选择 0 到 9 号窗口
d	# 断开当前客户端
D	# 选择并断开一个客户端
&	# 关闭当前窗口
x	# 关闭当前窗格
Up, Down Left, Right	# 将焦点移动至相邻的窗格
M-1 到 M-5	# 排列窗格： # 1) 水平等分 # 2) 垂直等分 # 3) 将一个窗格作为主要窗格，其他窗格水平等分 # 4) 将一个窗格作为主要窗格，其他窗格垂直等分 # 5) 平铺
C-Up, C-Down C-Left, C-Right	# 改变当前窗格的大小，每按一次增减一个单位
M-Up, M-Down M-Left, M-Right	# 改变当前窗格的大小，每按一次增减五个单位

配置 ~/.tmux.conf

tmux.conf 可以在 **tmux** 启动时自动设置选项，类似于 **.vimrc** 或 **init.el** 的用法。

```
# tmux.conf 示例
# 2014.10

### 通用设置
#####

# 启用 UTF-8 编码
setw -g utf8 on
set-option -g status-utf8 on

# 命令回滚/历史数量限制
set -g history-limit 2048

# 从 1 开始编号，而不是从 0 开始
set -g base-index 1

# 启用鼠标
set-option -g mouse-select-pane on

# 重新加载配置文件
unbind r
bind r source-file ~/.tmux.conf

### 快捷键设置
#####

# 取消默认的前缀键 C-b
unbind C-b

# 设置新的前缀键 `
set-option -g prefix `

# 多次按下前缀键时，切换到上一个窗口
bind C-a last-window
bind ` last-window

# 按下F11/F12，可以选择不同的前缀键
bind F11 set-option -g prefix C-a
bind F12 set-option -g prefix `

# Vim 风格的快捷键绑定
setw -g mode-keys vi
set-option -g status-keys vi

# 使用 Vim 风格的按键在窗格间移动
bind h select-pane -L
bind j select-pane -D
bind k select-pane -U
```

```
bind l select-pane -R

# 循环切换不同的窗口
bind e previous-window
bind f next-window
bind E swap-window -t -1
bind F swap-window -t +1

# 较易于使用的窗格分割快捷键
bind = split-window -h
bind - split-window -v
unbind ''
unbind %

# 在嵌套使用 tmux 的情况下，激活最内层的会话，以便向其发送命令
bind a send-prefix

### 外观主题
#####

# 状态栏颜色
set-option -g status-justify left
set-option -g status-bg black
set-option -g status-fg white
set-option -g status-left-length 40
set-option -g status-right-length 80

# 窗格边框颜色
set-option -g pane-active-border-fg green
set-option -g pane-active-border-bg black
set-option -g pane-border-fg white
set-option -g pane-border-bg black

# 消息框颜色
set-option -g message-fg black
set-option -g message-bg green

# 窗口状态栏颜色
setw -g window-status-bg black
setw -g window-status-current-fg green
setw -g window-status-bell-attr default
setw -g window-status-bell-fg red
setw -g window-status-content-attr default
setw -g window-status-content-fg yellow
setw -g window-status-activity-attr default
setw -g window-status-activity-fg yellow

### 用户界面
#####
```

```
# 通知方式
setw -g monitor-activity on
set -g visual-activity on
set-option -g bell-action any
set-option -g visual-bell off

# 自动设置窗口标题
set-option -g set-titles on
set-option -g set-titles-string '#H:#S.#I.#P #W #T' # 窗口编号,程序名称,是否活动

# 调整状态栏
set -g status-left "#[fg=red] #H#[fg=green]:#[fg=white]#S#[fg=green] |[default]"

# 在状态栏中显示性能计数器
# 需要用到 https://github.com/thewtex/tmux-mem-cpu-load
set -g status-interval 4
set -g status-right "#[fg=green] | #[fg=white]#(tmux-mem-cpu-load)#[fg=green] | #[fg=cya
```

参考资料

[Tmux 主页](#)

[Tmux 手册](#)

[FreeBSDChina Wiki](#)

[Archlinux Wiki](#))

[Tmux 快速教程](#)

[如何在 tmux 状态栏中显示 CPU / 内存占用的百分比](#)

[管理复杂 tmux 会话的工具 - tmuxinator](#)

language: TypeScript category: language contributors:

```
- ["Philippe Vl  rick", "https://github.com/pvlerick"]
```

translators:

```
- ["Shawn Zhang", "https://github.com/shawnzhang009"]
```

filename: learntypescript-cn.ts

lang: zh-cn

TypeScript是一门为开发大型JavaScript应用而设计的语言。TypeScript在JavaScript的基础上增加了类、模块、接口、泛型和静态类型（可选）等常见的概念。它是JavaScript的一个超集：所有JavaScript代码都是有效的TypeScript代码，所以任何JavaScript项目都可以无缝引入TypeScript。TypeScript编译器会把TypeScript代码编译成JavaScript代码。

本文只关注TypeScript额外增加的区别于JavaScript的语法，。

如需测试TypeScript编译器，你可以在[Playground](#)写代码，它会自动编译成JavaScript代码然后直接显示出来。

```
// TypeScript有三种基本类型
var isDone: boolean = false;
var lines: number = 42;
var name: string = "Anders";

// 如果不知道是什么类型，可以使用"any"(任意)类型
var notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // 亦可，定义为布尔型

// 对于集合的声明，有类型化数组和泛型数组
var list: number[] = [1, 2, 3];
// 另外一种，使用泛型数组
var list: Array<number> = [1, 2, 3];

// 枚举：
enum Color {Red, Green, Blue};
var c: Color = Color.Green;

// 最后，"void"用于函数没有任何返回的特殊情况下
function bigHorribleAlert(): void {
    alert("I'm a little annoying box!");
}
```

```

// 函数是"第一等公民"(first class citizens), 支持使用箭头表达式和类型推断

// 以下是相等的, TypeScript编译器会把它们编译成相同的JavaScript代码
var f1 = function(i: number): number { return i * i; }
// 返回推断类型的值
var f2 = function(i: number) { return i * i; }
var f3 = (i: number): number => { return i * i; }
// 返回推断类型的值
var f4 = (i: number) => { return i * i; }
// 返回推断类型的值, 单行程式可以不需要return关键字和大括号
var f5 = (i: number) => i * i;

// 接口是结构化的, 任何具有这些属性的对象都与该接口兼容
interface Person {
  name: string;
  // 可选属性, 使用"?"标识
  age?: number;
  // 函数
  move(): void;
}

// 实现"Person"接口的对象, 当它有了"name"和"move"方法之后可被视为一个"Person"
var p: Person = { name: "Bobby", move: () => {} };
// 带了可选参数的对象
var validPerson: Person = { name: "Bobby", age: 42, move: () => {} };
// 因为"age"不是"number"类型所以这不是一个"Person"
var invalidPerson: Person = { name: "Bobby", age: true };

// 接口同样可以描述一个函数的类型
interface SearchFunc {
  (source: string, subString: string): boolean;
}
// 参数名并不重要, 参数类型才是重要的
var mySearch: SearchFunc;
mySearch = function(src: string, sub: string) {
  return src.search(sub) != -1;
}

// 类 - 成员默认为公共的(public)
class Point {
  // 属性
  x: number;

  // 构造器 - 这里的public/private关键字会为属性生成样板代码和初始化值
  // 这个例子中, y会被同x一样定义, 不需要额外代码
  // 同样支持默认值

  constructor(x: number, public y: number = 0) {
    this.x = x;
  }
}

```



```

// 函数
dist() { return Math.sqrt(this.x * this.x + this.y * this.y); }

// 静态成员
static origin = new Point(0, 0);
}

var p1 = new Point(10, 20);
var p2 = new Point(25); //y为0

// 继承
class Point3D extends Point {
  constructor(x: number, y: number, public z: number = 0) {
    super(x, y); // 必须显式调用父类的构造器
  }

  // 重写
  dist() {
    var d = super.dist();
    return Math.sqrt(d * d + this.z * this.z);
  }
}

// 模块, "."可以作为子模块的分隔符
module Geometry {
  export class Square {
    constructor(public sideLength: number = 0) {
    }
    area() {
      return Math.pow(this.sideLength, 2);
    }
  }
}

var s1 = new Geometry.Square(5);

// 引入模块并定义本地别名
import G = Geometry;

var s2 = new G.Square(10);

// 泛型
// 类
class Tuple<T1, T2> {
  constructor(public item1: T1, public item2: T2) {
  }
}

// 接口
interface Pair<T> {
  item1: T;

```

```
    item2: T;
}

// 以及函数
var pairToTuple = function<T>(p: Pair<T>) {
    return new Tuple(p.item1, p.item2);
};

var tuple = pairToTuple({ item1:"hello", item2:"world"});

// 引用定义文件
// <reference path="jquery.d.ts" />

// 模板字符串(使用反引号的字符串)
// 嵌入变量的模板字符串
var name = 'Tyrone';
var greeting = `Hi ${name}, how are you?`
// 有多行内容的模板字符串
var multiline = `This is an example
of a multiline string`;
```

参考资料

- [TypeScript官网](#)
- [TypeScript语言规范说明书\(pdf\)](#)
- [Anders Hejlsberg - TypeScript介绍](#)
- [GitHub源码](#)
- [Definitely Typed - 类型定义仓库](#)

language: Visual Basic contributors:

```
- ["Brian Martin", "http://brianmartin.biz"]
```

translators:

```
- ["Abner Chou", "http://github.com/NoahDragon"]
```

lang: zh-cn

filename: learnvisualbasic.vb-cn

```
Module Module1

    Sub Main()
        ' 让我们先从简单的终端程序学起。
        ' 单引号用来生成注释（注意是半角单引号，非全角单引号’）
        ' 为了方便运行此示例代码，我写了个目录索引。
        ' 可能你还不了解以下代码的意义，但随着教程的深入，
        ' 你会渐渐理解其用法。
        Console.Title = ("Learn X in Y Minutes")
        Console.WriteLine("NAVIGATION") ' 显示目录
        Console.WriteLine("")
        Console.ForegroundColor = ConsoleColor.Green
        Console.WriteLine("1. Hello World Output") ' Hello world 输出示例
        Console.WriteLine("2. Hello World Input") ' Hello world 输入示例
        Console.WriteLine("3. Calculating Whole Numbers") ' 求整数之和
        Console.WriteLine("4. Calculating Decimal Numbers") ' 求小数之和
        Console.WriteLine("5. Working Calculator") ' 计算器
        Console.WriteLine("6. Using Do While Loops") ' 使用 Do While 循环
        Console.WriteLine("7. Using For While Loops") ' 使用 For While 循环
        Console.WriteLine("8. Conditional Statements") ' 条件语句
        Console.WriteLine("9. Select A Drink") ' 选饮料
        Console.WriteLine("50. About") ' 关于
        Console.WriteLine("Please Choose A Number From The Above List")
        Dim selection As String = Console.ReadLine
        Select Case selection
            Case "1" ' Hello world 输出示例
                Console.Clear() ' 清空屏幕
                HelloWorldOutput() ' 调用程序块
            Case "2" ' Hello world 输入示例
                Console.Clear()
                HelloWorldInput()
            Case "3" ' 求整数之和
                Console.Clear()
```

```

        CalculatingWholeNumbers()
    Case "4" ' 求小数之和
        Console.Clear()
        CalculatingDecimalNumbers()
    Case "5" ' 计算器
        Console.Clear()
        WorkingCalculator()
    Case "6" ' 使用 do while 循环
        Console.Clear()
        UsingDoWhileLoops()
    Case "7" ' 使用 for while 循环
        Console.Clear()
        UsingForLoops()
    Case "8" ' 条件语句
        Console.Clear()
        ConditionalStatement()
    Case "9" ' If/Else 条件语句
        Console.Clear()
        IfElseStatement() ' 选饮料
    Case "50" ' 关于本程序和作者
        Console.Clear()
        Console.Title = ("Learn X in Y Minutes :: About")
        MsgBox("This tutorial is by Brian Martin (@BrianMartinn)")
        Console.Clear()
        Main()
        Console.ReadLine()

End Select
End Sub

' 一、对应程序目录1，下同

' 使用 private subs 声明函数。
Private Sub HelloWorldOutput()
    ' 程序名
    Console.Title = "Hello World Ouput | Learn X in Y Minutes"
    ' 使用 Console.Write("") 或者 Console.WriteLine("") 来输出文本到屏幕上
    ' 对应的 Console.Read() 或 Console.ReadLine() 用来读取键盘输入
    Console.WriteLine("Hello World")
    Console.ReadLine()
    ' Console.WriteLine()后加Console.ReadLine()是为了防止屏幕输出信息一闪而过
    ' 类似平时常见的“单击任意键继续”的意思。
End Sub

' 二
Private Sub HelloWorldInput()
    Console.Title = "Hello World YourName | Learn X in Y Minutes"
    ' 变量
    ' 用来存储用户输入的数据
    ' 变量声明以 Dim 开始，结尾为 As VariableType （变量类型）。

```

```

' 此教程中，我们希望知道你的姓名，并让程序记录并输出。
Dim username As String
' 我们定义username使用字符串类型（String）来记录用户姓名。
Console.WriteLine("Hello, What is your name? ") ' 询问用户输入姓名
username = Console.ReadLine() ' 存储用户名到变量 username
Console.WriteLine("Hello " + username) ' 输出将是 Hello + username
Console.ReadLine() ' 暂停屏幕并显示以上输出
' 以上程序将询问你的姓名，并和你打招呼。
' 其它变量如整型（Integer）我们用整型来处理整数。
End Sub

' 三
Private Sub CalculatingWholeNumbers()
    Console.Title = "Calculating Whole Numbers | Learn X in Y Minutes"
    Console.Write("First number: ") ' 输入一个整数：1, 2, 50, 104, 等等
    Dim a As Integer = Console.ReadLine()
    Console.Write("Second number: ") ' 输入第二个整数
    Dim b As Integer = Console.ReadLine()
    Dim c As Integer = a + b
    Console.WriteLine(c)
    Console.ReadLine()
    ' 以上程序将两个整数相加
End Sub

' 四
Private Sub CalculatingDecimalNumbers()
    Console.Title = "Calculating with Double | Learn X in Y Minutes"
    ' 当然，我们还需要能够处理小数。
    ' 只需要将整型（Integer）改为小数（Double）类型即可。

    ' 输入一个小数： 1.2, 2.4, 50.1, 104.9, 等等
    Console.Write("First number: ")
    Dim a As Double = Console.ReadLine
    Console.Write("Second number: ") ' 输入第二个数
    Dim b As Double = Console.ReadLine
    Dim c As Double = a + b
    Console.WriteLine(c)
    Console.ReadLine()
    ' 以上代码能实现两个小数相加
End Sub

' 五
Private Sub WorkingCalculator()
    Console.Title = "The Working Calculator | Learn X in Y Minutes"
    ' 但是如果你希望有个能够处理加减乘除的计算器呢？
    ' 只需将上面代码复制粘贴即可。
    Console.Write("First number: ") ' 输入第一个数
    Dim a As Double = Console.ReadLine
    Console.Write("Second number: ") ' 输入第二个数
    Dim b As Integer = Console.ReadLine
    Dim c As Integer = a + b

```

```
Dim d As Integer = a * b
Dim e As Integer = a - b
Dim f As Integer = a / b
```

' 通过以下代码我们可以将以上所算的加减乘除结果输出到屏幕上。

```
Console.Write(a.ToString() + " + " + b.ToString())
' 我们希望答案开头能有3个空格，可以使用String.PadLeft(3)方法。
Console.WriteLine(" = " + c.ToString.PadLeft(3))
Console.Write(a.ToString() + " * " + b.ToString())
Console.WriteLine(" = " + d.ToString.PadLeft(3))
Console.Write(a.ToString() + " - " + b.ToString())
Console.WriteLine(" = " + e.ToString.PadLeft(3))
Console.Write(a.ToString() + " / " + b.ToString())
Console.WriteLine(" = " + e.ToString.PadLeft(3))
Console.ReadLine()
```

End Sub

' 六

Private Sub UsingDoWhileLoops()

' 如同以上的代码一样

' 这次我们将询问用户是否继续 (Yes or No?)

' 我们将使用Do While循环，因为我们不知到用户是否需要使用一次以上。

Console.Title = "UsingDoWhileLoops | Learn X in Y Minutes"

Dim answer As String ' 我们使用字符串变量来存储answer (答案)

Do ' 循环开始

```
    Console.Write("First number: ")
    Dim a As Double = Console.ReadLine
    Console.Write("Second number: ")
    Dim b As Integer = Console.ReadLine
    Dim c As Integer = a + b
    Dim d As Integer = a * b
    Dim e As Integer = a - b
    Dim f As Integer = a / b
```

```
    Console.Write(a.ToString() + " + " + b.ToString())
    Console.WriteLine(" = " + c.ToString.PadLeft(3))
    Console.Write(a.ToString() + " * " + b.ToString())
    Console.WriteLine(" = " + d.ToString.PadLeft(3))
    Console.Write(a.ToString() + " - " + b.ToString())
    Console.WriteLine(" = " + e.ToString.PadLeft(3))
    Console.Write(a.ToString() + " / " + b.ToString())
    Console.WriteLine(" = " + e.ToString.PadLeft(3))
    Console.ReadLine()
```

' 询问用户是否继续，注意大小写。

Console.Write("Would you like to continue? (yes / no)")

' 程序读入用户输入

answer = Console.ReadLine() ' added a bracket here

' 当用户输入"yes"时，程序将跳转到Do，并再次执行

Loop While answer = "yes"

End Sub

' 七

Private Sub UsingForLoops()

' 有一些程序只需要运行一次。
' 这个程序我们将实现从10倒数计数。

Console.Title = "Using For Loops | Learn X in Y Minutes"

' 声明变量和Step（步长,即递减的速度,如-1, -2, -3等）。

For i As Integer = 10 To 0 Step -1

 Console.WriteLine(i.ToString) ' 将计数结果输出的屏幕

Next i ' 计算新的i值

Console.WriteLine("Start")

Console.ReadLine()

End Sub

' 八

Private Sub ConditionalStatement()

Console.Title = "Conditional Statements | Learn X in Y Minutes"

Dim userName As String = Console.ReadLine

Console.WriteLine("Hello, What is your name? ") ' 询问用户姓名

userName = Console.ReadLine() ' 存储用户姓名

If userName = "Adam" Then

 Console.WriteLine("Hello Adam")

 Console.WriteLine("Thanks for creating this useful site")

 Console.ReadLine()

Else

 Console.WriteLine("Hello " + userName)

 Console.WriteLine("Have you checked out www.learnxinyminutes.com")

 Console.ReadLine() ' 程序停止, 并输出以上文本

End If

End Sub

' 九

Private Sub IfElseStatement()

Console.Title = "If / Else Statement | Learn X in Y Minutes"

' 有时候我们需要考虑多于两种情况。
' 这时我们就需要使用If/ElseIf条件语句。
' If语句就好似个自动售货机, 当用户输入A1, A2, A3, 等去选择物品时,
' 所有的选择可以合并到一个If语句中

Dim selection As String = Console.ReadLine() ' 读入用户选择

Console.WriteLine("A1. for 7Up") ' A1 七喜

Console.WriteLine("A2. for Fanta") ' A2 芬达

Console.WriteLine("A3. for Dr. Pepper") ' A3 胡椒医生

Console.WriteLine("A4. for Diet Coke") ' A4 无糖可乐

Console.ReadLine()

If selection = "A1" Then

 Console.WriteLine("7up")

 Console.ReadLine()

ElseIf selection = "A2" Then

```
        Console.WriteLine("fanta")
        Console.ReadLine()
    ElseIf selection = "A3" Then
        Console.WriteLine("dr. pepper")
        Console.ReadLine()
    ElseIf selection = "A4" Then
        Console.WriteLine("diet coke")
        Console.ReadLine()
    Else
        Console.WriteLine("Please select a product") ' 请选择你需要的产品
        Console.ReadLine()
    End If

End Sub

End Module
```

参考

我（译注：原作者）在命令行下学习的VB。命令行编程使我能够更好的了解程序编译运行机制，并使学习其它语言变得容易。

如果希望进一步学习VB，这里还有更深层次的 [VB教学（英文）](#)。

所有代码均通过测试。只需复制粘帖到Visual Basic中，并按F5运行即可。

language: xml contributors:

- ["João Farias", "<https://github.com/JoaoGFarias>"] translators:
 - ["Zach Zhang", "<https://github.com/checkcheckzz>"] filename: learnxml-cn.xml lang: zh-cn
-

XML是一种标记语言，被设计用来存储数据和传输数据。

不像HTML, XML不指定怎样显示或格式化数据，只是携带它。

- XML 语法

<!-- XML中的注解像这样 -->

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

<!-- 上面是一个典型的XML文件。

它以一个声明开始，通知一些元数据（自选的）

XML使用一个树的结构。上面的文件中，根节点是'bookstore'，它有三个孩子节点，所有的'books'。那些节点有更多的孩子节点，等等。。。

节点用开放/关闭标签创建， 并且孩子就是在开发和关闭标签之间的节点。-->

<!-- XML 携带两类信息：

- 1 - 属性 -> 那是关于一个元素的元数据。
通常，XML解析器使用这些信息去正确地存储数据。
它通过在开放标签里出现在插入语中表示。
- 2 - 元素 -> 那是纯数据。
那就是解析器将从XML文件提取的东西。
元素出现在开放和关闭标签之间，没插入语。-->

<!-- 下面，一个有两个属性的元素-->

```
<file type="gif" id="4293">computer.gif</file>
```

- 良好格式的文件 x 验证

一个XML文件是良好格式的如果它是语法正确的。 但是， 使用文件定义，比如DTD和XML概要，在文件中插入更多的限制是可能的。

一个遵守一个文件定义的XML文件被叫做有效的，对于那个文件来说。

有了这个工具，你能够在应用逻辑之外检查XML数据。

<!-- 下面，你能够看到一个简化版本的增加了DTD定义的bookstore文件。-->

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Bookstore.dtd">
<bookstore>
  <book category="COOKING">
    <title>Everyday Italian</title>
    <price>30.00</price>
  </book>
</bookstore>
```

<!-- 这个DTD可能是像这样的:-->

```
<!DOCTYPE note
[
  <!ELEMENT bookstore (book+)>
  <!ELEMENT book (title,price)>
  <!ATTLIST book category CDATA "Literature">
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
```

<!-- 这个DTD以一个声明开始。

接下来，根节点被声明，它需要一个或多个孩子节点'book'。

每个'book'应该准确包含一个'title'和'price'和一个被叫做'category'的缺省值为"Literature"的属性。

这个'title'和'price'节点包含一个解析过的字符数据。-->

<!-- 这个DTD可以在XML文件中本身被声明。-->

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE note
[
  <!ELEMENT bookstore (book+)>
  <!ELEMENT book (title,price)>
  <!ATTLIST book category CDATA "Literature">
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>

<bookstore>
  <book category="COOKING">
    <title>Everyday Italian</title>
    <price>30.00</price>
  </book>
</bookstore>
```

language: yaml contributors:

- ["Adam Brenecki", "<https://github.com/adambrenecki>"] translators:
 - ["Zach Zhang", "<https://github.com/checkcheckzz>"] filename: learnyaml-cn.yaml lang: zh-cn
-

YAML是一个数据序列化语言，被设计成人类直接可写可读的。

它是JSON的严格超集，增加了语法显著换行符和缩进，就像Python。但和Python不一样，YAML根本不容许文字制表符。

```
# YAML中的注解看起来像这样。

#####
# 标量类型 #
#####

# 我们的根对象（它们在整个文件里延续）将会是一个地图，
# 它等价于在别的语言里的一个字典，哈希表或对象。
key: value
another_key: Another value goes here.
a_number_value: 100
scientific_notation: 1e+12
boolean: true
null_value: null
key with spaces: value
# 注意到字符串不需要被引用。但是，它们可以被引用。
"Keys can be quoted too.": "Useful if you want to put a ':' in your key."

# 多行字符串既可以写成像一个'文字块'(使用 |)，
# 或像一个'折叠块'(使用 '>')。
literal_block: |
    This entire block of text will be the value of the 'literal_block' key,
    with line breaks being preserved.

    The literal continues until de-dented, and the leading indentation is
    stripped.

        Any lines that are 'more-indented' keep the rest of their indentation -
        these lines will be indented by 4 spaces.
folded_style: >
    This entire block of text will be the value of 'folded_style', but this
    time, all newlines will be replaced with a single space.

    Blank lines, like above, are converted to a newline character.

    'More-indented' lines keep their newlines, too -
    this text will appear over two lines.
```

```
#####
# 集合类型 #
#####

# 嵌套是通过缩进完成的。
a_nested_map:
    key: value
    another_key: Another Value
    another_nested_map:
        hello: hello

# 地图不用有字符串键值。
0.25: a float key

# 键值也可以是多行对象，用?表明键值的开始。
? |
    This is a key
    that has multiple lines
: and this is its value

# YAML也容许键值是集合类型，但是很多语言将会抱怨。

# 序列（等价于表或数组）看起来像这样：
a_sequence:
    - Item 1
    - Item 2
    - 0.5 # 序列可以包含不同类型。
    - Item 4
    - key: value
      another_key: another_value
    -
      - This is a sequence
      - inside another sequence

# 因为YAML是JSON的超集，你也可以写JSON风格的地图和序列：
json_map: {"key": "value"}
json_seq: [3, 2, 1, "takeoff"]

#####
# 其余的YAML特点 #
#####

# YAML还有一个方便的特点叫'锚'，它让你简单地在整个文件里重复内容。
# 两个键值将会有相同的值：
anchored_content: &anchor_name This string will appear as the value of two keys.
other_anchor: *anchor_name

# YAML还有标签，你可以用它显示地声明类型。
explicit_string: !!str 0.5
# 一些解析器实现特定语言的标签，就像这个为了Python的复数类型。
```

```
python_complex_number: !!python/complex 1+2j

#####
# 其余的YAML类型 #
#####

# 字符串和数字不是仅有的YAML可以理解的标量。
# ISO 格式的日期和日期时间文字也是可以解析的。
datetime: 2001-12-15T02:59:43.1Z
datetime_with_spaces: 2001-12-14 21:59:43.10 -5
date: 2002-12-14

# 这个!!binary标签表明一个字符串实际上是一个二进制blob的base64编码表示。
gif_file: !!binary |
  R0lGODlhDAAMAIQAAP//9/X17unp5wZmZgAAAOfn515eXvPz7Y60juDg4J+fn5
  OTk6enp56enmlpaWNjY60jo4SEhP/++f/++f/++f/++f/++f/++f/++f/++f/+
  +f/++f/++f/++f/++f/++SH+Dk1hZGUgd2l0aCBHSU1QACwAAAAADAAMAAFLC
  AgjoEwnuNAF0hpEMTRiggcz4BNJHrv/zCFcLiwMWYNG84BwwEeECcggoBADs=

# YAML还有一个集合类型，它看起来像这样：
set:
  ? item1
  ? item2
  ? item3

# 像Python一样，集合仅是有null数值的地图；上面的集合等价于：
set2:
  item1: null
  item2: null
  item3: null
```