代码不能完全按照您的想法运行，它只能完全按照您的写法运行，这是编程界的一条金科玉律。

让您的写法符合您的想法是非常困难的。在这节课中，我们会传授给您一些非常有用技术，帮您处理代码中的 bug 和程序性能问题。

调试代码

打印调试法与日志

"最有效的 debug 工具就是细致的分析，配合恰当位置的打印语句" — Brian Kernighan, *Unix 新手入门*。

调试代码的第一种方法往往是在您发现问题的地方添加一些打印语句，然后不断重复此过程直到您获取了足够的信息并找到问题的根本原因。

另外一个方法是使用日志，而不是临时添加打印语句。日志较普通的打印语句有如下的一些优势：

- 您可以将日志写入文件、socket 或者甚至是发送到远端服务器而不仅仅是标准输出；
- 日志可以支持严重等级（例如 INFO, DEBUG, WARN, ERROR 等），这使您可以根据需要过滤日志；
- 对于新发现的问题，很可能您的日志中已经包含了可以帮助您定位问题的足够的信息。

这里 是一个包含日志的例程序：

```
$ python logger.py
# Raw output as with just prints
$ python logger.py log
# Log formatted output
$ python logger.py log ERROR
# Print only ERROR levels and above
$ python logger.py color
# Color formatted output
```

有很多技巧可以使日志的可读性变得更好，我最喜欢的一个是技巧是对其进行着色。到目前为止，您应该已经知道，以彩色文本显示终端信息时可读性更好。但是应该如何设置呢？

`ls` 和 `grep` 这样的程序会使用 ANSI escape codes，它是一系列的特殊字符，可以使您的 shell 改变输出结果的颜色。例如，执行 `echo -e "\e[38;2;255;0;0mThis is red\e[0m"` 会打印红色的字符串：`This is red` 。只要您的终端支持 真彩色。如果您的终端不支持真彩色（例如 MacOS 的 Terminal.app），您可以使用支持更加广泛的 16 色，例如："\e[31; 1mThis is red\e[0m "。

下面这个脚本向您展示了如何在终端中打印多种颜色（只要您的终端支持真彩色）

```
#!/usr/bin/env bash
for R in $(seq 0 20 255); do
    for G in $(seq 0 20 255); do
        for B in $(seq 0 20 255); do
            printf "\e[38;2;${R};${G};${B}m\e[0m";
        done
    done
done
```

第三方日志系统

如果您正在构建大型软件系统，您很可能会使用到一些依赖，有些依赖会作为程序单独运行。如 Web 服务器、数据库或消息代理都是此类常见的第三方依赖。

和这些系统交互的时候，阅读它们的日志是非常必要的，因为仅靠客户端侧的错误信息可能并不足以定位问题。

幸运的是，大多数的程序都会将日志保存在您的系统中的某个地方。对于 UNIX 系统来说，程序的日志通常存放在 `/var/log`。例如， NGINX web 服务器就将其日志存放于 `/var/log/nginx`。

目前，系统开始使用 **system log**，您所有的日志都会保存在这里。大多数（但不是全部的）Linux 系统都会使用 `systemd`，这是一个系统守护进程，它会控制您系统中的很多东西，例如哪些服务应该启动并运行。`systemd` 会将日志以某种特殊格式存放于 `/var/log/journal`，您可以使用 `journalctl` 命令显示这些消息。

类似地，在 macOS 系统中是 `/var/log/system.log`，但是有更多的工具会使用系统日志，它的内容可以使用 `log show` 显示。

对于大多数的 UNIX 系统，您也可以使用 `dmesg` 命令来读取内核的日志。

如果您希望将日志加入到系统日志中，您可以使用 `logger` 这个 shell 程序。下面这个例子显示了如何使用 `logger` 并且如何找到能够将其存入系统日志的条目。

不仅如此，大多数的编程语言都支持向系统日志中写日志。

```
logger "Hello Logs"
# On macOS
log show --last 1m | grep Hello
# On Linux
journalctl --since "1m ago" | grep Hello
```

正如我们在数据整理那节课上看到的那样，日志的内容可以非常的多，我们需要对其进行处理和过滤才能得到我们想要的信息。

如果您发现您需要对 `journalctl` 和 `log show` 的结果进行大量的过滤，那么此时可以考虑使用它们自带的选项对其结果先过滤一遍再输出。还有一些像 `lnav` 这样的工具，它为日志文件提供了更好的展现和浏览方式。

调试器

当通过打印已经不能满足您的调试需求时，您应该使用调试器。

调试器是一种可以允许我们和正在执行的程序进行交互的程序，它可以做到：

- 当到达某一行时将程序暂停；
- 一次一条指令地逐步执行程序；
- 程序崩溃后查看变量的值；
- 满足特定条件时暂停程序；
- 其他高级功能。

很多编程语言都有自己的调试器。Python 的调试器是 `pdb`.

下面对 `pdb` 支持的命令进行简单的介绍：

- **l**(ist) - 显示当前行附近的 11 行或继续执行之前的显示；

- **s**(tep) - 执行当前行，并在第一个可能的地方停止；
- **n**(ext) - 继续执行直到当前函数的下一条语句或者 return 语句；
- **b**(reak) - 设置断点（基于传入的参数）；
- **p**(rint) - 在当前上下文对表达式求值并打印结果。还有一个命令是 **pp** ，它使用 `pprint` 打印；
- **r**(eturn) - 继续执行直到当前函数返回；
- **q**(uit) - 退出调试器。

让我们使用 `pdb` 来修复下面的 Python 代码（参考讲座视频）

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

注意，因为 Python 是一种解释型语言，所以我们可以通过 `pdb` shell 执行命令。
`ipdb` 是一种增强型的 `pdb` ，它使用 `IPython` 作为 REPL 并开启了 tab 补全、语法高亮、更好的回溯和更好的内省，同时还保留了 `pdb` 模块相同的接口。

对于更底层的编程语言，您可能需要了解一下 `gdb` (以及它的改进版 `pwndbg`) 和 `lldb` 。

它们都对类 C 语言的调试进行了优化，它允许您探索任意进程及其机器状态：寄存器、堆栈、程序计数器等。

专门工具

即使您需要调试的程序是一个二进制的黑盒程序，仍然有一些工具可以帮助到您。当您的程序需要执行一些只有操作系统内核才能完成的操作时，它需要使用 系统调用。有一些命令可以帮助您追踪您的程序执行的系统调用。在 Linux 中可以使用 `strace` ，在 macOS 和 BSD 中可以使用 `dtrace` 。`dtrace` 用起来可能有些别扭，因为它使用的是它自有的 `D` 语言，但是我们可以使用一个叫做 `dtruss` 的封装使其具有和 `strace` (更多信息参考 这里)类似的接口

下面的例子展现来如何使用 `strace` 或 `dtruss` 来显示 `ls` 执行时，对 `stat` 系统调用进行追踪对结果。若需要深入了解 `strace` ，这篇文章 值得一读。

```
# On Linux
sudo strace -e lstat ls -l > /dev/null
4
# On macOS
sudo dtruss -t lstat64_extended ls -l > /dev/null
```

有些情况下，我们需要查看网络数据包才能定位问题。像 `tcpdump` 和 Wireshark 这样的网络数据包分析工具可以帮助您获取网络数据包的内容并基于不同的条件进行过滤。

对于 web 开发， Chrome/Firefox 的开发者工具非常方便，功能也很强大：

- 源码 -查看任意站点的 HTML/CSS/JS 源码；

- 实时地修改 HTML, CSS, JS 代码 - 修改网站的内容、样式和行为用于测试（从这一点您也能看出来，网页截图是不可靠的）；
- Javascript shell - 在 JS REPL 中执行命令；
- 网络 - 分析请求的时间线；
- 存储 - 查看 Cookies 和本地应用存储。

静态分析

有些问题是您不需要执行代码就能发现的。例如，仔细观察一段代码，您就能发现某个循环变量覆盖了某个已经存在的变量或函数名；或是有个变量在被读取之前并没有被定义。
这种情况下 静态分析 工具就可以帮我们找到问题。静态分析会将程序的源码作为输入然后基于编码规则对其进行分析并对代码的正确性进行推理。

下面这段 Python 代码中存在几个问题。 首先，我们的循环变量 `foo` 覆盖了之前定义的函数 `foo`。最后一行，我们还把 `bar` 错写成了 `baz`，因此当程序完成 `sleep` (一分钟)后，执行到这一行的时候便会崩溃。

```
import time

def foo():
    return 42

for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz)
```

静态分析工具可以发现此类的问题。当我们使用 `pyflakes` 分析代码的时候，我们会得到与这两处 bug 相关的错误信息。`mypy` 则是另外一个工具，它可以对代码进行类型检查。这里，`mypy` 会经过我们 `bar` 起初是一个 `int` ，然后变成了 `float`。这些问题都可以在不运行代码的情况下被发现。

```
$ pyflakes foobar.py
foobar.py:6: redefinition of unused 'foo' from line 3
foobar.py:11: undefined name 'baz'

$ mypy foobar.py
foobar.py:6: error: Incompatible types in assignment (expression has type "int", variable
has type "Callable[[], Any]")
foobar.py:9: error: Incompatible types in assignment (expression has type "float", variable
has type "int")
foobar.py:11: error: Name 'baz' is not defined
Found 3 errors in 1 file (checked 1 source file)
```

在 shell 工具那一节课的时候，我们介绍了 `shellcheck`，这是一个类似的工具，但它是应用于 shell 脚本的。

大多数的编辑器和 IDE 都支持在编辑界面显示这些工具的分析结果、高亮有警告和错误的位置。
这个过程通常称为 **code linting** 。风格检查或安全检查的结果同样也可以进行相应的显示。

在 vim 中，有 `ale` 或 `syntastic` 可以帮助您做同样的事情。
在 Python 中， `pylint` 和 `pep8` 是两种用于进行风格检查的工具，而 `bandit` 工具则用于检查安全相关的问题。

对于其他语言的开发者来说，静态分析工具可以参考这个列表：Awesome Static Analysis (您也许会对 *Writing* 一节感兴趣) 。对于 linters 则可以参考这个列表： Awesome Linters。

对于风格检查和代码格式化，还有以下一些工具可以作为补充：用于 Python 的 `black`、用于 Go 语言的 `gofmt`、用于 Rust 的 `rustfmt` 或是用于 JavaScript, HTML 和 CSS 的 `prettier` 。这些工具可以自动格式化您的代码，这样代码风格就可以与常见的风格保持一致。
尽管您可能并不想对代码进行风格控制，标准的代码风格有助于方便别人阅读您的代码，也可以方便您阅读它的代码。

性能分析

即使您的代码能够像您期望的一样运行，但是如果它消耗了您全部的 CPU 和内存，那么它显然也不是个好程序。算法课上我们通常会介绍大 O 标记法，但却没教给我们如何找到程序中的热点。
鉴于 过早的优化是万恶之源，您需要学习性能分析和监控工具，它们会帮助您找到程序中最耗时、最耗资源的部分，这样您就可以有针对性的进行性能优化。

计时

和调试代码类似，大多数情况下我们只需要打印两处代码之间的时间即可发现问题。下面这个例子中，我们使用了 Python 的 `time` 模块。

```
import time, random
n = random.randint(1, 10) * 100

# 获取当前时间
start = time.time()

# 执行一些操作
print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

# 比较当前时间和起始时间
print(time.time() - start)

# Output
# Sleeping for 500 ms
# 0.5713930130004883
```

不过，执行时间（wall clock time）也可能会误导您，因为您的电脑可能也在同时运行其他进程，也可能在此期间发生了等待。 对于工具来说，需要区分真实时间、用户时间和系统时间。通常来说，用户时间 + 系统时间代表了您的进程所消耗的实际 CPU （更详细的解释可以参照 这篇文章）。

- 真实时间 *Real* - 从程序开始到结束流失掉的真实时间，包括其他进程的执行时间以及阻塞消耗的时间（例如等待 I/O 或网络）；
- 用户时间 *User* - CPU 执行用户代码所花费的时间；
- 系统时间 *Sys* - CPU 执行系统内核代码所花费的时间。

例如，试着执行一个用于发起 HTTP 请求的命令并在其前面添加 `time` 前缀。网络不好的情况下您可能会看到下面的输出结果。请求花费了 2s 多才完成，但是进程仅花费了 15ms 的 CPU 用户时间和 12ms 的 CPU 内核时间。

```
$ time curl https://missing.csail.mit.edu &> /dev/null
real    0m2.561s
user    0m0.015s
sys     0m0.012s
```

性能分析工具（profilers）

CPU

大多数情况下，当人们提及性能分析工具的时候，通常指的是 CPU 性能分析工具。
CPU 性能分析工具有两种： 追踪分析器（*tracing*）及采样分析器（*sampling*）。
追踪分析器 会记录程序的每一次函数调用，而采样分析器则只会周期性的监测（通常为每毫秒）您的程序并记录程序堆栈。它们使用这些记录来生成统计信息，显示程序在哪些事情上花费了最多的时间。如果您希望了解更多相关信息，可以参考 这篇 介绍性的文章。

大多数的编程语言都有一些基于命令行的分析器，我们可以使用它们来分析代码。它们通常可以集成在 IDE 中，但是本节课我们会专注于这些命令行工具本身。

在 Python 中，我们使用 `cProfile` 模块来分析每次函数调用所消耗的时间。 在下面的例子中，我们实现了一个基础的 grep 命令：

```
#!/usr/bin/env python

import sys, re

def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")

if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

我们可以使用下面的命令来对这段代码进行分析。通过它的输出我们可以知道，IO 消耗了大量的时间，编译正则表达式也比较耗费时间。因为正则表达式只需要编译一次，我们可以将其移动到 for 循环外面来改进性能。

```
$ python -m cProfile -s tottime grep.py 1000 '^(import|\s*def)[^,]*$' *.py

[omitted program output]
```

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   8000    0.266    0.000    0.292    0.000 {built-in method io.open}
   8000    0.153    0.000    0.894    0.000 grep.py:5(grep)
  17000    0.101    0.000    0.101    0.000 {built-in method builtins.print}
   8000    0.100    0.000    0.129    0.000 {method 'readlines' of '_io._IOBase' objects}
  93000    0.097    0.000    0.111    0.000 re.py:286(_compile)
  93000    0.069    0.000    0.069    0.000 {method 'search' of '_sre.SRE_Pattern' objects}
  93000    0.030    0.000    0.141    0.000 re.py:231(compile)
  17000    0.019    0.000    0.029    0.000 codecs.py:318(decode)
      1    0.017    0.017    0.911    0.911 grep.py:3(<module>)

[omitted lines]
```

关于 Python 的 `cProfile` 分析器（以及其他一些类似的分析器），需要注意的是它显示的是每次函数调用的时间。看上去可能快到反直觉，尤其是如果您在代码里面使用了第三方的函数库，因为内部函数调用也会被看作函数调用。

更加符合直觉的显示分析信息的方式是包括每行代码的执行时间，这也是 *行分析器* 的工作。例如，下面这段 Python 代码会向本课程的网站发起一个请求，然后解析响应返回的页面中的全部 URL：

```python
#!/usr/bin/env python
import requests
from bs4 import BeautifulSoup

# 这个装饰器会告诉行分析器
# 我们想要分析这个函数
@profile
def get_urls():
    response = requests.get('https://missing.csail.mit.edu')
    s = BeautifulSoup(response.content, 'lxml')
    urls = []
    for url in s.find_all('a'):
        urls.append(url['href'])

if __name__ == '__main__':
    get_urls()
```

如果我们使用 Python 的 `cProfile` 分析器，我们会得到超过 2500 行的输出结果，即使对其进行排序，我们仍然搞不懂时间到底都花在哪了。如果我们使用 `line_profiler`，它会基于行来显示时间：

```
$ kernprof -l -v a.py
Wrote profile results to urls.py.lprof
Timer unit: 1e-06 s

Total time: 0.636188 s
File: a.py
Function: get_urls at line 5

Line #  Hits         Time  Per Hit   % Time  Line Contents
==============================================================
 5                                           @profile
 6                                           def get_urls():
```

```
 7         1     613909.0 613909.0     96.5      response =
requests.get('https://missing.csail.mit.edu')
 8         1      21559.0  21559.0      3.4      s = BeautifulSoup(response.content, 'lxml')
 9         1          2.0      2.0      0.0      urls = []
10        25        685.0     27.4      0.1      for url in s.find_all('a'):
11        24         33.0      1.4      0.0          urls.append(url['href'])
```

内存

像 C 或者 C++ 这样的语言，内存泄漏会导致您的程序在使用完内存后不去释放它。为了应对内存类的 Bug，我们可以使用类似 Valgrind 这样的工具来检查内存泄漏问题。

对于 Python 这类具有垃圾回收机制的语言，内存分析器也是很有用的，因为对于某个对象来说，只要有指针还指向它，那它就不会被回收。

下面这个例子及其输出，展示了 memory-profiler 是如何工作的（注意装饰器和 `line-profiler` 类似）。

```
@profile
def my_func():
    a = [1] * (10 ** 6)
    b = [2] * (2 * 10 ** 7)
    del b
    return a

if __name__ == '__main__':
    my_func()
```

```
$ python -m memory_profiler example.py
Line #    Mem usage    Increment   Line Contents
================================================
     3                             @profile
     4      5.97 MB    0.00 MB   def my_func():
     5     13.61 MB    7.64 MB       a = [1] * (10 ** 6)
     6    166.20 MB  152.59 MB       b = [2] * (2 * 10 ** 7)
     7     13.61 MB -152.59 MB       del b
     8     13.61 MB    0.00 MB       return a
```

事件分析

在我们使用 `strace` 调试代码的时候，您可能会希望忽略一些特殊的代码并希望在分析时将其当作黑盒处理。`perf` 命令将 CPU 的区别进行了抽象，它不会报告时间和内存的消耗，而是报告与您的程序相关的系统事件。

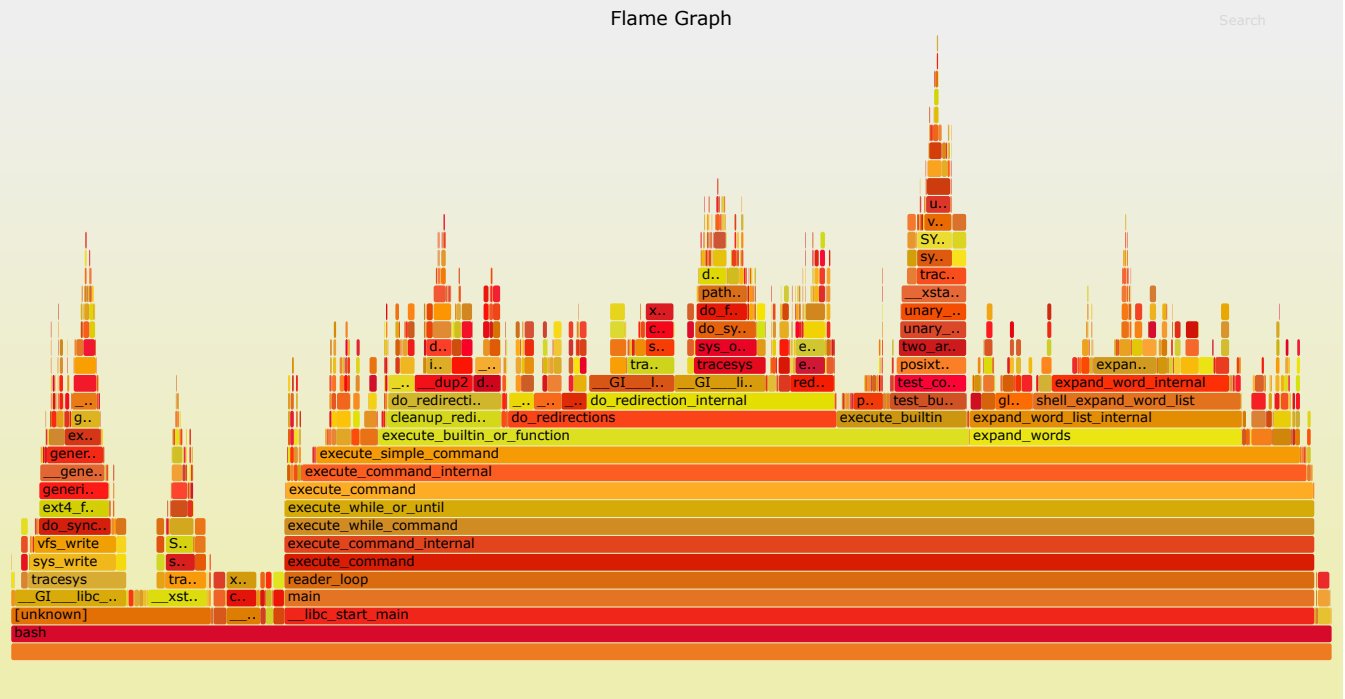
例如，`perf` 可以报告不佳的缓存局部性（poor cache locality）、大量的页错误（page faults）或活锁（livelocks）。下面是关于常见命令的简介：

- `perf list` - 列出可以被 pref 追踪的事件；
- `perf stat COMMAND ARG1 ARG2` - 收集与某个进程或指令相关的事件；
- `perf record COMMAND ARG1 ARG2` - 记录命令执行的采样信息并将统计数据储存在 `perf.data` 中；
- `perf report` - 格式化并打印 `perf.data` 中的数据。

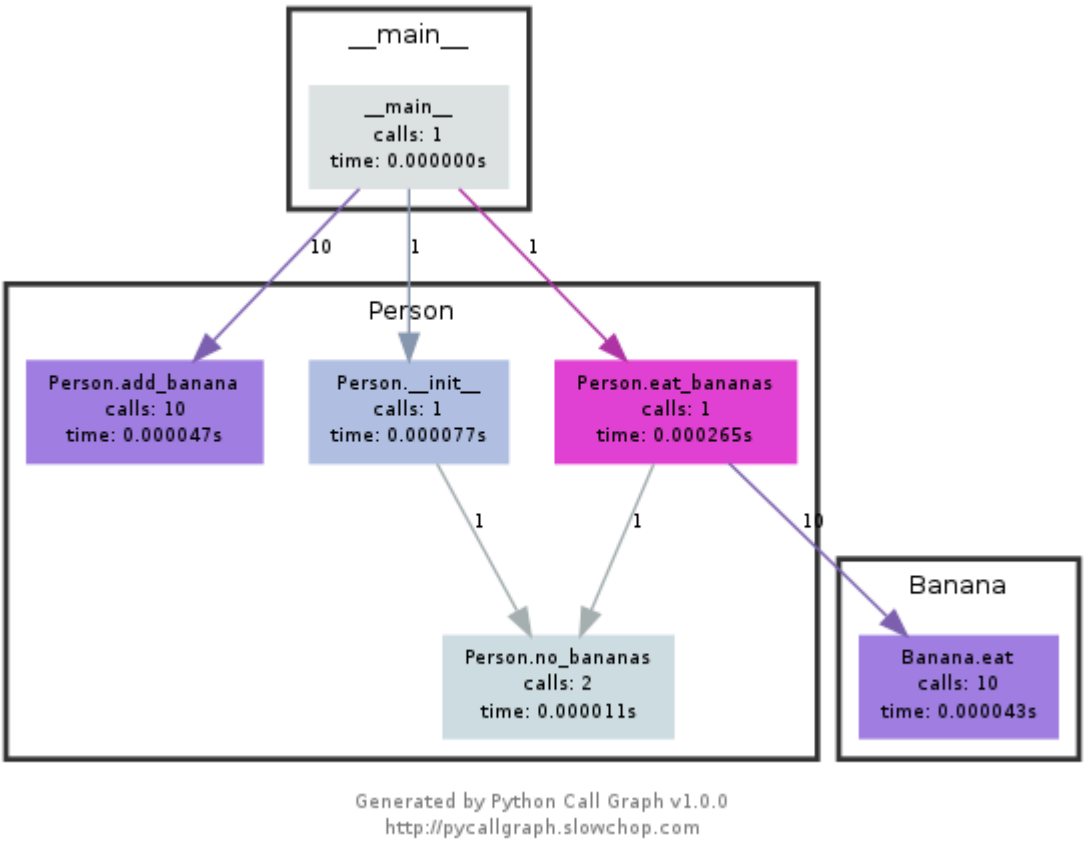
可视化

使用分析器来分析真实的程序时，由于软件的复杂性，其输出结果中将包含大量的信息。人类是一种视觉动物，非常不善于阅读大量的文字。因此很多工具都提供了可视化分析器输出结果的功能。

对于采样分析器来说，常见的显示 CPU 分析数据的形式是 火焰图，火焰图会在 Y 轴显示函数调用关系，并在 X 轴显示其耗时的比例。火焰图同时还是可交互的，您可以深入程序的某一具体部分，并查看其栈追踪（您可以尝试点击下面的图片）。



调用图和控制流图可以显示子程序之间的关系，它将函数作为节点并把函数调用作为边。将它们和分析器的信息（例如调用次数、耗时等）放在一起使用时，调用图会变得非常有用，它可以帮助我们分析程序的流程。
在 Python 中您可以使用 `pycallgraph` 来生成这些图片。



资源监控

有时候，分析程序性能的第一步是搞清楚它所消耗的资源。程序变慢通常是因为它所需要的资源不够了。例如，没有足够的内存或者网络连接变慢的时候。

有很多很多的工具可以被用来显示不同的系统资源，例如 CPU 占用、内存使用、网络、磁盘使用等。

- **通用监控** - 最流行的工具要数 `htop`, 了，它是 `top` 的改进版。`htop` 可以显示当前运行进程的多种统计信息。`htop` 有很多选项和快捷键，常见的有：`<F6>` 进程排序、 `t` 显示树状结构和 `h` 打开或折叠线程。 还可以留意一下 `glances` ，它的实现类似但是用户界面更好。如果需要合并测量全部的进程， `dstat` 是也是一个非常好用的工具，它可以实时地计算不同子系统资源的度量数据，例如 I/O、网络、 CPU 利用率、上下文切换等等；
- **I/O 操作** - `iotop` 可以显示实时 I/O 占用信息而且可以非常方便地检查某个进程是否正在执行大量的磁盘读写操作；
- **磁盘使用** - `df` 可以显示每个分区的信息，而 `du` 则可以显示当前目录下每个文件的磁盘使用情况（**d** isk **u** sage）。`-h` 选项可以使命令以对人类（**h** uman）更加友好的格式显示数据；`ncdu` 是一个交互性更好的 `du` ，它可以让您在不同目录下导航、删除文件和文件夹；
- **内存使用** - `free` 可以显示系统当前空闲的内存。内存也可以使用 `htop` 这样的工具来显示；
- **打开文件** - `lsof` 可以列出被进程打开的文件信息。 当我们需要查看某个文件是被哪个进程打开的时候，这个命令非常有用；
- **网络连接和配置** - `ss` 能帮助我们监控网络包的收发情况以及网络接口的显示信息。`ss` 常见的一个使用场景是找到端口被进程占用的信息。如果要显示路由、网络设备和接口信息，您可以使用 `ip` 命令。注意， `netstat` 和 `ifconfig` 这两个命令已经被前面那些工具所代替了。
- **网络使用** - `nethogs` 和 `iftop` 是非常好的用于对网络占用进行监控的交互式命令行工具。

如果您希望测试一下这些工具，您可以使用 `stress` 命令来为系统人为地增加负载。

专用工具

有时候，您只需要对黑盒程序进行基准测试，并依此对软件选择进行评估。
类似 `hyperfine` 这样的命令行可以帮您快速进行基准测试。例如，我们在 shell 工具和脚本那一节课中我们推荐使用 `fd` 来代替 `find`。我们这里可以用 `hyperfine` 来比较一下它们。

例如，下面的例子中，我们可以看到 `fd` 比 `find` 要快 20 倍。

```
$ hyperfine --warmup 3 'fd -e jpg' 'find . -iname "*.jpg"'
Benchmark #1: fd -e jpg
  Time (mean ± σ):      51.4 ms ±   2.9 ms    [User: 121.0 ms, System: 160.5 ms]
  Range (min … max):    44.2 ms …  60.1 ms    56 runs

Benchmark #2: find . -iname "*.jpg"
  Time (mean ± σ):      1.126 s ±  0.101 s    [User: 141.1 ms, System: 956.1 ms]
  Range (min … max):    0.975 s …  1.287 s    10 runs

Summary
  'fd -e jpg' ran
   21.89 ± 2.33 times faster than 'find . -iname "*.jpg"'
```

和 debug 一样，浏览器也包含了很多不错的性能分析工具，可以用来分析页面加载，让我们可以搞清楚时间都消耗在什么地方（加载、渲染、脚本等等）。 更多关于 Firefox 和 Chrome 的信息可以点击链接。

课后练习

习题解答

调试

1. 使用 Linux 上的 `journalctl` 或 macOS 上的 `log show` 命令来获取最近一天中超级用户的登录信息及其所执行的指令。如果找不到相关信息，您可以执行一些无害的命令，例如 `sudo ls` 然后再次查看。
2. 学习 这份 `pdb` 实践教程并熟悉相关的命令。更深入的信息您可以参考 这份 教程。
3. 安装 `shellcheck` 并尝试对下面的脚本进行检查。这段代码有什么问题吗？请修复相关问题。在您的编辑器中安装一个 linter 插件，这样它就可以自动地显示相关警告信息。

   ```
   #!/bin/sh
   ## Example: a typical script with several problems
   for f in $(ls *.m3u)
   do
     grep -qi hq.*mp3 $f \
       && echo -e 'Playlist $f contains a HQ file in mp3 format'
   done
   ```

4. (进阶题) 请阅读 可逆调试 并尝试创建一个可以工作的例子（使用 `rr` 或 `RevPDB`）。

性能分析

1. 这里 有一些排序算法的实现。请使用 `cProfile` 和 `line_profiler` 来比较插入排序和快速排序的性能。两种算法的瓶颈分别在哪里？然后使用 `memory_profiler` 来检查内存消耗，为什么插入排序更好一些？然后再看看原地排序版本的快排。附加题：使用 `perf` 来查看不同算法的循环次数及缓存命中及丢失情况。

2. 这里有一些用于计算斐波那契数列 Python 代码，它为计算每个数字都定义了一个函数：

```
#!/usr/bin/env python
def fib0(): return 0

def fib1(): return 1

s = """def fib{}(): return fib{}() + fib{}()"""

if __name__ == '__main__':

    for n in range(2, 10):
        exec(s.format(n, n-1, n-2))
    # from functools import lru_cache
    # for n in range(10):
    #     exec("fib{} = lru_cache(1)(fib{})".format(n, n))
    print(eval("fib9()"))
```

   将代码拷贝到文件中使其变为一个可执行的程序。首先安装 `pycallgraph` 和 `graphviz`(如果您能够执行 `dot`，则说明已经安装了 GraphViz.)。并使用 `pycallgraph graphviz -- ./fib.py` 来执行代码并查看 `pycallgraph.png` 这个文件。`fib0` 被调用了多少次？我们可以通过记忆法来对其进行优化。将注释掉的部分放开，然后重新生成图片。这回每个 `fibN` 函数被调用了多少次？

3. 我们经常会遇到的情况是某个我们希望去监听的端口已经被其他进程占用了。让我们通过进程的 PID 查找相应的进程。首先执行 `python -m http.server 4444` 启动一个最简单的 web 服务器来监听 `4444` 端口。在另外一个终端中，执行 `lsof | grep LISTEN` 打印出所有监听端口的进程及相应的端口。找到对应的 PID 然后使用 `kill <PID>` 停止该进程。

4. 限制进程资源也是一个非常有用的技术。执行 `stress -c 3` 并使用 `htop` 对 CPU 消耗进行可视化。现在，执行 `taskset --cpu-list 0,2 stress -c 3` 并可视化。`stress` 占用了 3 个 CPU 吗？为什么没有？阅读 `man taskset` 来寻找答案。附加题：使用 `cgroups` 来实现相同的操作，限制 `stress -m` 的内存使用。

5. (进阶题) `curl ipinfo.io` 命令或执行 HTTP 请求并获取关于您 IP 的信息。打开 Wireshark 并抓取 `curl` 发起的请求和收到的回复报文。(提示：可以使用 `http` 进行过滤，只显示 HTTP 报文)