哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

第三章 软件编码与测试

3-1 代码评审分析与优化

王忠杰
rainy@hit.edu.cn

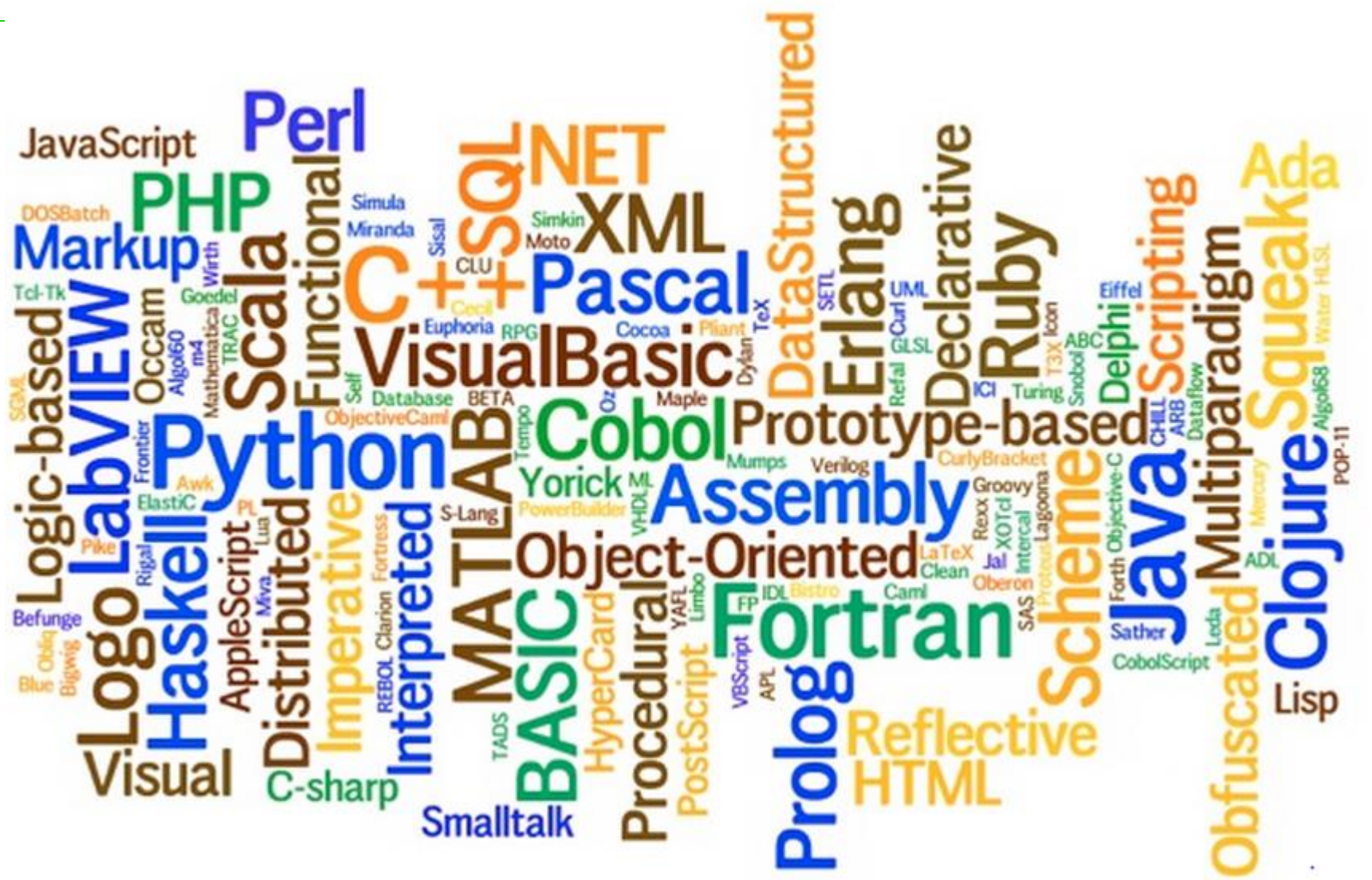
2017年9月27日

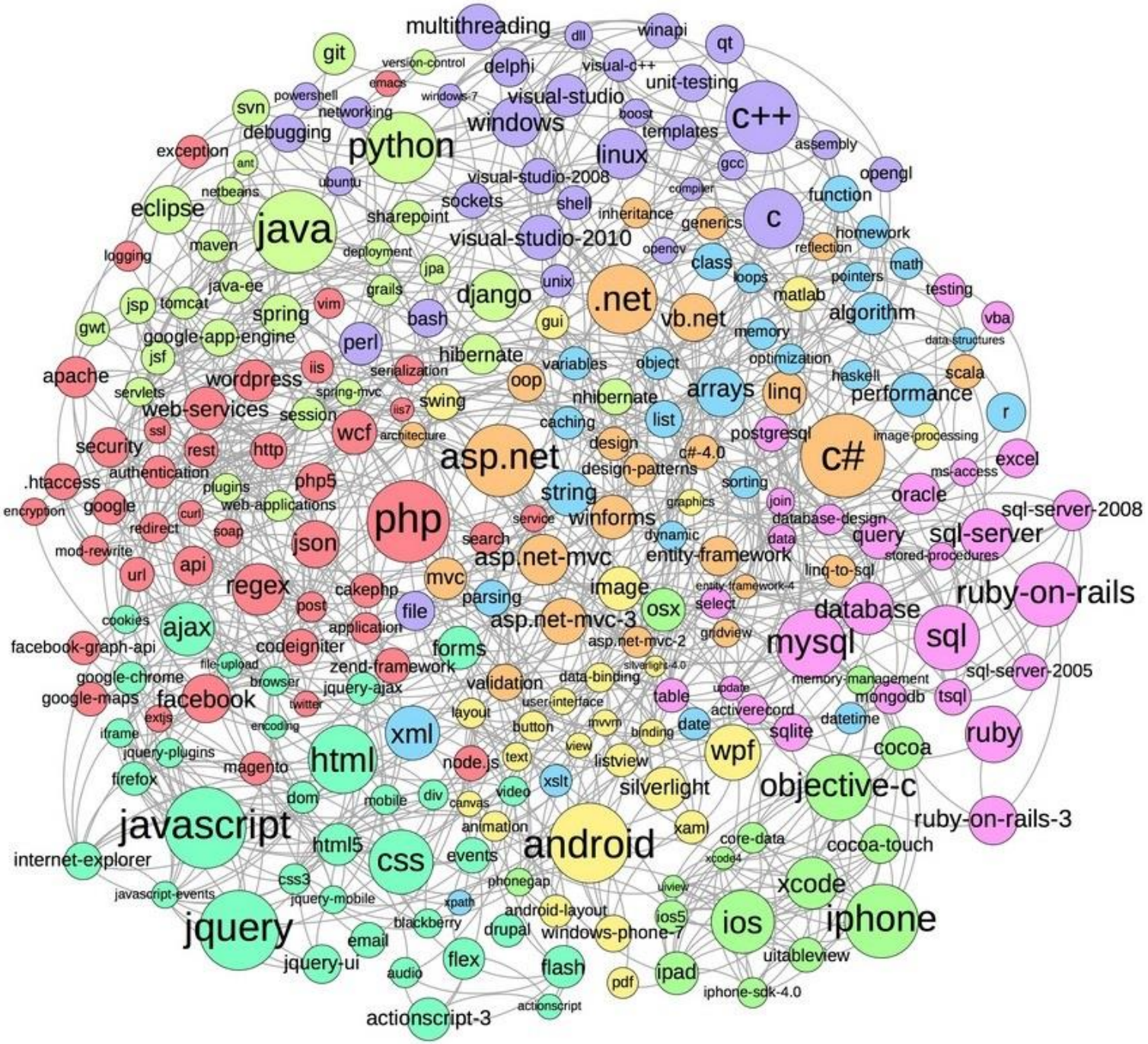
主要内容

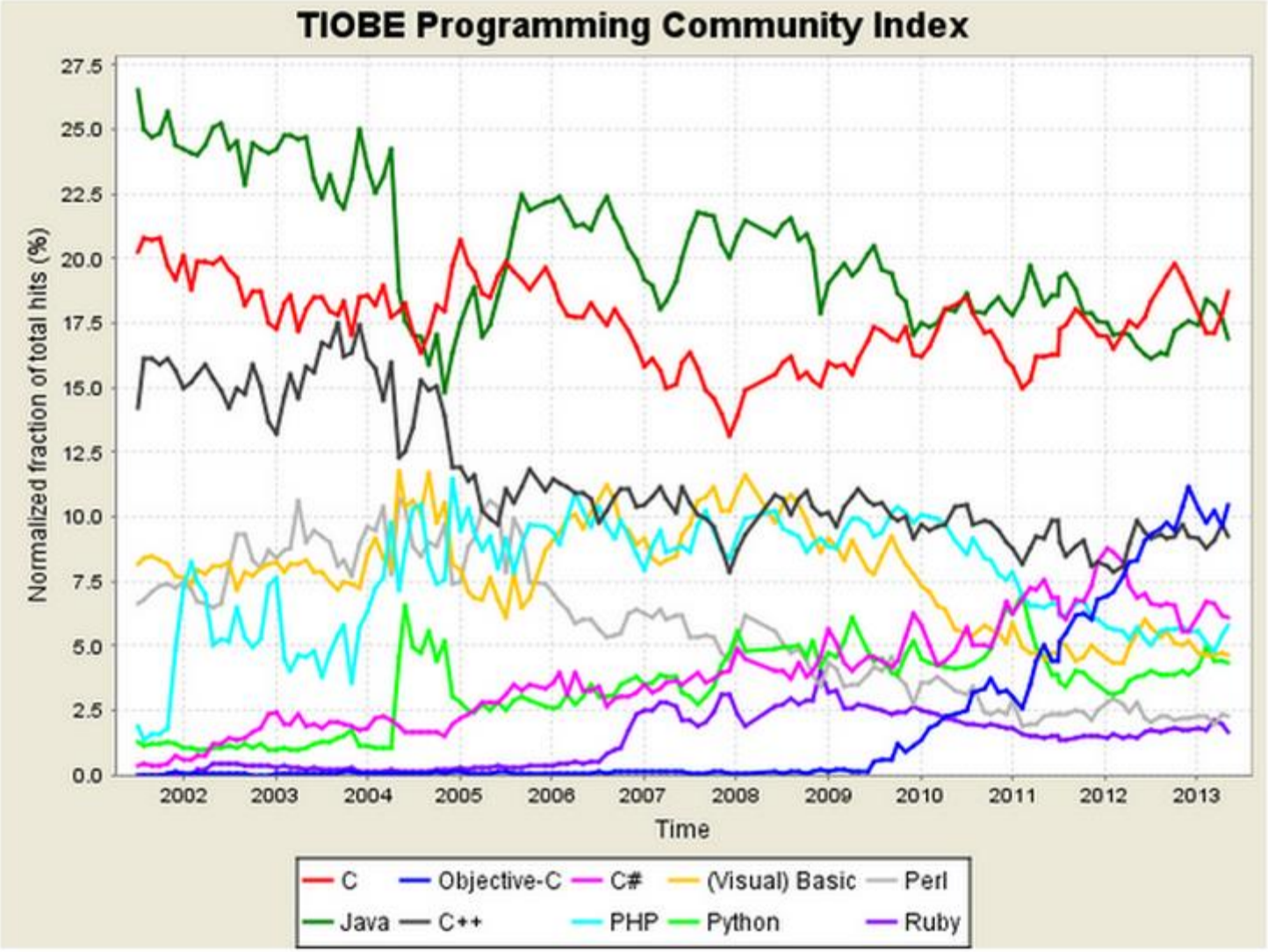
- 1 关于编程语言
- 2 代码的错误与缺陷
- 3 静态程序分析：代码评审 (code review)
- 4 动态程序分析：program profiling

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

1 关于编程语言


JavaScript
Perl
PHP
Markup
Scala
Functional
C++
SQL
NET
XML
Pascal
VisualBasic
DataStructured
Erlang
Declarative
Ruby
Ada
Scripting
Squeak
Multiparadigm
Clojure
Lisp
Obfuscated
Logic-based
LabVIEW
Occam
Python
Cobol
Prototype-based
Assembly
Yorick
MATLAB
BASIC
Object-Oriented
Fortran
Scheme
Java
Logo
Haskell
Distributed
Imperative
Interpreted
AppleScript
Visual
C-sharp
Smalltalk
HyperCard
Procedural
PostScript
Prolog
Reflective
HTML
Delphi
Eiffel
Lua
Tcl-Tk
DOSBatch
Simula
Miranda
Sisal
Simkin
Moto
CLU
Goedel
Wirth
Euphoria
RPG
Cocoa
Pliant
TeX
Cecil
Self
Database
BETA
Oz
Maple
ObjectiveCaml
Tempo
Mumps
Verilog
CurlyBracket
Groovy
Rexx
XOTcl
Intercal
Lagoona
LaTeX
Clean
Jal
Oberon
Caml
IDL
Bistro
FP
YAFL
Limbo
VBScript
APL
SAS
Sather
CobolScript
Forth
Objective-C
CHILL
ARB
Dataflow
Mercury
ADL
Leda
Algol68
POP-11
Water HLSL
SETL
UML
Curl
GLSL
Refal
ICI
Turing
T3X
Icon
Snobol
ABC
Awk
ElastiC
Pike
Rigal
Frontier
PL
Miva
Befunge
Oblique
Bigwig
Blue
REBOL
Clarion
Fortress
S-Lang
PowerBuilder
VHDL
ML
TADS
Algol60
m4
Mathematica
TRAC
Occam
SQML


multithreading
dll
winapi
qt
git
version-control
delphi
visual-c++
unit-testing
emacs
windows-7
visual-studio
c++
svn
powershell
networking
windows
boost
templates
debugging
python
linux
assembly
exception
ant
gcc
opengl
ubuntu
visual-studio-2008
compiler
netbeans
sockets
shell
function
eclipse
java
sharepoint
inheritance
generics
c
homework
maven
visual-studio-2010
opencv
reflection
logging
deployment
jpa
class
loops
pointers
math
java-ee
unix
grails
django
.net
matlab
testing
jsp
tomcat
vim
spring
bash
gui
vb.net
algorithm
vba
gwt
google-app-engine
perl
memory
data-structures
jsf
hibernate
variables
object
optimization
apache
iis
wordpress
serialization
oop
haskell
scala
servlets
spring-mvc
arrays
linq
performance
r
web-services
session
swing
nhibernate
iis7
caching
list
ssl
wcf
architecture
postgresql
image-processing
security
rest
http
asp.net
design
c#-4.0
c#
excel
authentication
design-patterns
ms-access
.htaccess
plugins
php5
sorting
string
oracle
encryption
google
curl
web-applications
graphics
join
sql-server-2008
redirect
soap
service
winforms
database-design
sql-server
php
search
dynamic
data
query
mod-rewrite
json
asp.net-mvc
entity-framework
stored-procedures
url
api
mvc
linq-to-sql
regex
cakephp
image
entity-framework-4
ruby-on-rails
parsing
osx
select
database
cookies
post
file
asp.net-mvc-3
ajax
application
gridview
sql
facebook-graph-api
codeigniter
asp.net-mvc-2
mysql
forms
file-upload
silverlight-4.0
google-chrome
zend-framework
data-binding
memory-management
sql-server-2005
validation
browser
jquery-ajax
update
mongodb
tsql
google-maps
facebook
table
user-interface
twitter
activerecord
extjs
layout
mvvm
date
datetime
ruby
encoding
button
binding
sqlite
iframe
xml
wpf
cocoa
jquery-plugins
view
node.js
text
listview
magento
html
firefox
xslt
silverlight
objective-c
dom
mobile
div
video
canvas
ruby-on-rails-3
javascript
animation
xaml
core-data
android
cocoa-touch
html5
internet-explorer
css
events
xcode4
phonegap
xcode
javascript-events
css3
jquery-mobile
uiview
xpath
android-layout
ios5
ios
iphone
jquery
blackberry
drupal
windows-phone-7
email
uitableview
jquery+ui
flex
flash
audio
pdf
ipad
iphone-sdk-4.0
actionscript
actionscript-3


TIOBE Programming Community Index
Normalized fraction of total hits (%)
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
Time
C
Objective-C
C#
(Visual) Basic
Perl
Java
C++
PHP
Python
Ruby

程序员的编程语言

- **编程语言层出不穷，新的语言是否可取代旧的语言？**
  - C/C++的生命力依旧旺盛；
  - JAVA似乎盛极而衰？
  - Object-C随着Apple的兴盛而越来越受欢迎；
  - PHP/Python/Ruby等更适合Web开发？
- **作为一个程序员，你选择语言和开发技术的标准是什么？**
- **在出现新的流行语言之后，你会做出切换吗？还是决定一辈子只做某种语言的忠实粉丝？**

选择编程语言的标准

- 应用领域：商业？科学计算？...
- 设计模型的特征：数据处理为主？交互性强？...
- 算法复杂性
- 数据结构的复杂性
- 运行效率：时空复杂性
- 后期维护与升级要求：可测试？可复用？易维护？
- 用户要求
- 系统兼容
- 可移植性
- 程序员的能力与水平：学习周期？开发周期？开发工具？
- 语言的未来发展前景

如何选择最合适的编程语言？

- http://www.csdn.net/article/2012-05-31/2806188
- http://developer.51cto.com/art/200902/112005.htm

- 编程语言本身的效率与生产率
- 掌握该编程语言的人数
- 该语言的受欢迎度
- 团队里掌握该语言人员的比例
- 学习难度
- 工具支持和第三方库支持
- 与待开发项目的的吻合度

程序员的修炼之道

持续深入某个技术领域，坚持学习和总结

不断积累自己的代码

github
SOCIAL CODING

stackoverflow

获取其他程序员的经验和教训

记录你的成绩和经历

LinkedIn

课堂讨论

- 有人认为：软件编码是将软件设计模型机械地转换成源程序代码，这是一种低水平的、缺乏创造性的工作。软件程序员是所谓的“软件蓝领”(software blue-collar)，是一碗“青春饭”；
- 又有人认为：“编程既是一门科学，也是一门艺术”，这是一项很有艺术性、独创性、技巧性的实践工作。编程技巧像艺术技巧一样，深不可测、奥妙无穷，程序员像艺术家一样，有发挥创造性的无限空间，且不受年龄和精力的限制。
- 你是否认同上述说法？说出你的理由。

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2 代码的错误与缺陷

F. Brooks对编程的隐喻： 焦油坑(The Tar Pit)

- 史前时期最骇人的景象，莫过于一群巨兽在焦油坑里做垂死前的挣扎。
- 不妨闭上眼睛想像一下，你看到了一群恐龙、长毛象、剑齿虎正在奋力挣脱焦油的束缚，但越挣扎，焦油就缠得越紧，就算他再强壮、再利害，最后都难逃灭顶的命运。
- 过去十年间，大型系统的软件开发工作就像是掉进了焦油坑里。

F. Brooks对编程的隐喻：焦油坑(The Tar Pit)

- **从单一程序到软件系统过程中，所造成复杂度的快速上升，期间并需要包含不同的活动与技能，使得软件开发必须面对多样性的挑战。**
- **软件系统产品(Programming Systems Product)开发的工作量是供个人使用的、独立开发的构件程序的9倍；**
  - 将程序产品化引起了3倍工作量、将构件整合为完整系统所需的设计、集成、测试又强加了3倍工作量。
  - 过去10余年的大型系统开发就犹如陷入了焦油坑，只有极少数的项目满足了目标、进度和预算的要求。
  - 各种开发问题相互纠缠和累积，团队的行动变得越来越慢，最后沉到了坑底。
- **Brooks最早认识到设计程序、开发软件的差别，他以程序与系统、产品的差异，解释两者之间的不同，并以3×3的复杂度加以说明。**

F. Brooks对编程的隐喻：焦油坑(The Tar Pit)

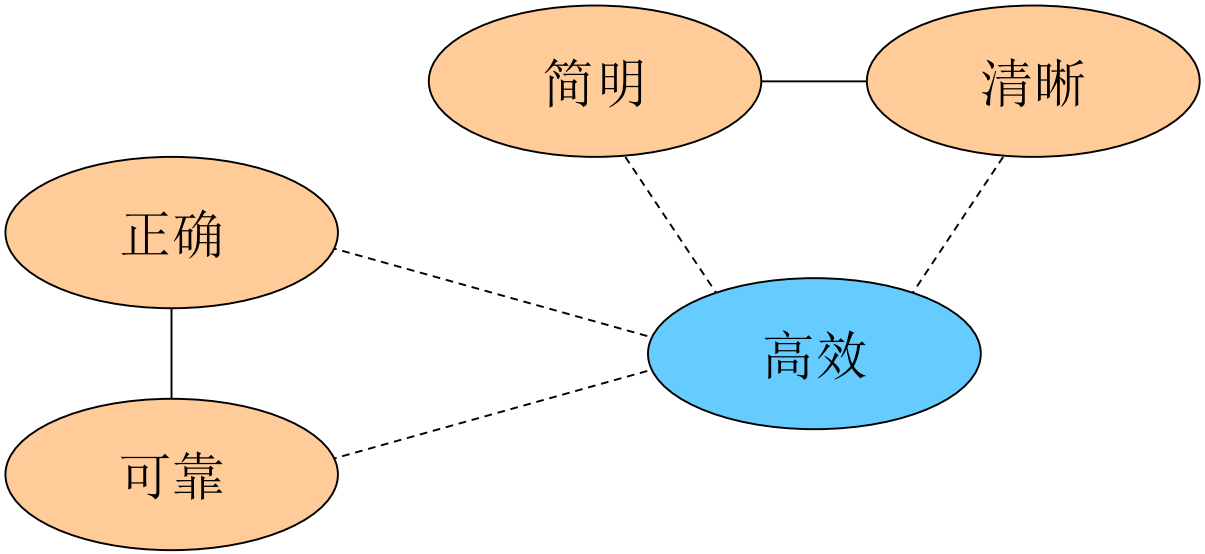
- **编程的乐趣：满足我们内心深处创造渴望和愉悦所有人的共同情感。**
  - 创建新事物；
  - 开发对其他人有用的东西；
  - 将相互啮合的零部件组装成整体的过程；
  - 面对不重复的任务，不断学习的乐趣；
  - 纯粹的思维活动，完全不同于对实际物体的驾驭；
- **编程的苦恼：将做事方式调整到“追求完美”。**
  - 有其他人来设定目标，而且要依靠自己无法控制的事物；
  - 编程的创造性伴随着枯燥艰苦的劳动；
  - 项目越接近结束，项目所面临的问题和困难就越多；
  - 产品在完成前总是面临着陈旧过时的威胁。

编程：
一个许多人痛苦挣扎的焦油坑、一种乐趣和苦恼共存的创造性活动。

你从中得到的快乐大于苦恼么？

代码规范

- **形式规范：使代码“看起来很美”**
  - 面向代码的可读性、可维护性、可移植性
- **内容规范：使代码“运行起来很美”**
  - 面向代码的正确性、可靠性、运行效率

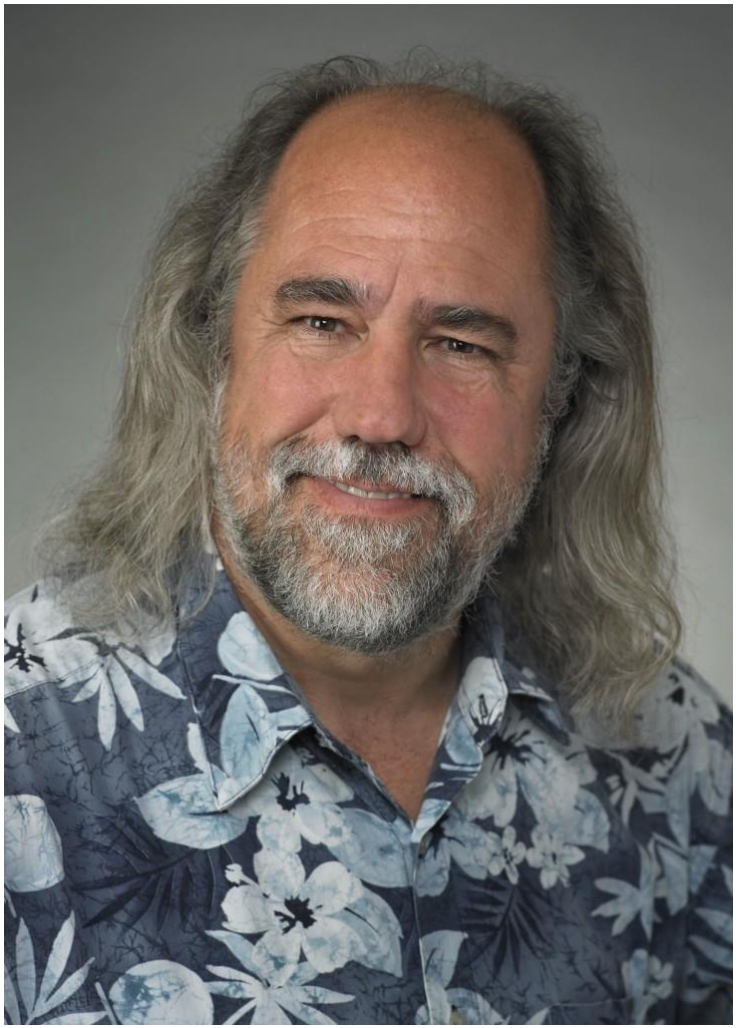


What Stroustrup (inventor of C++) said

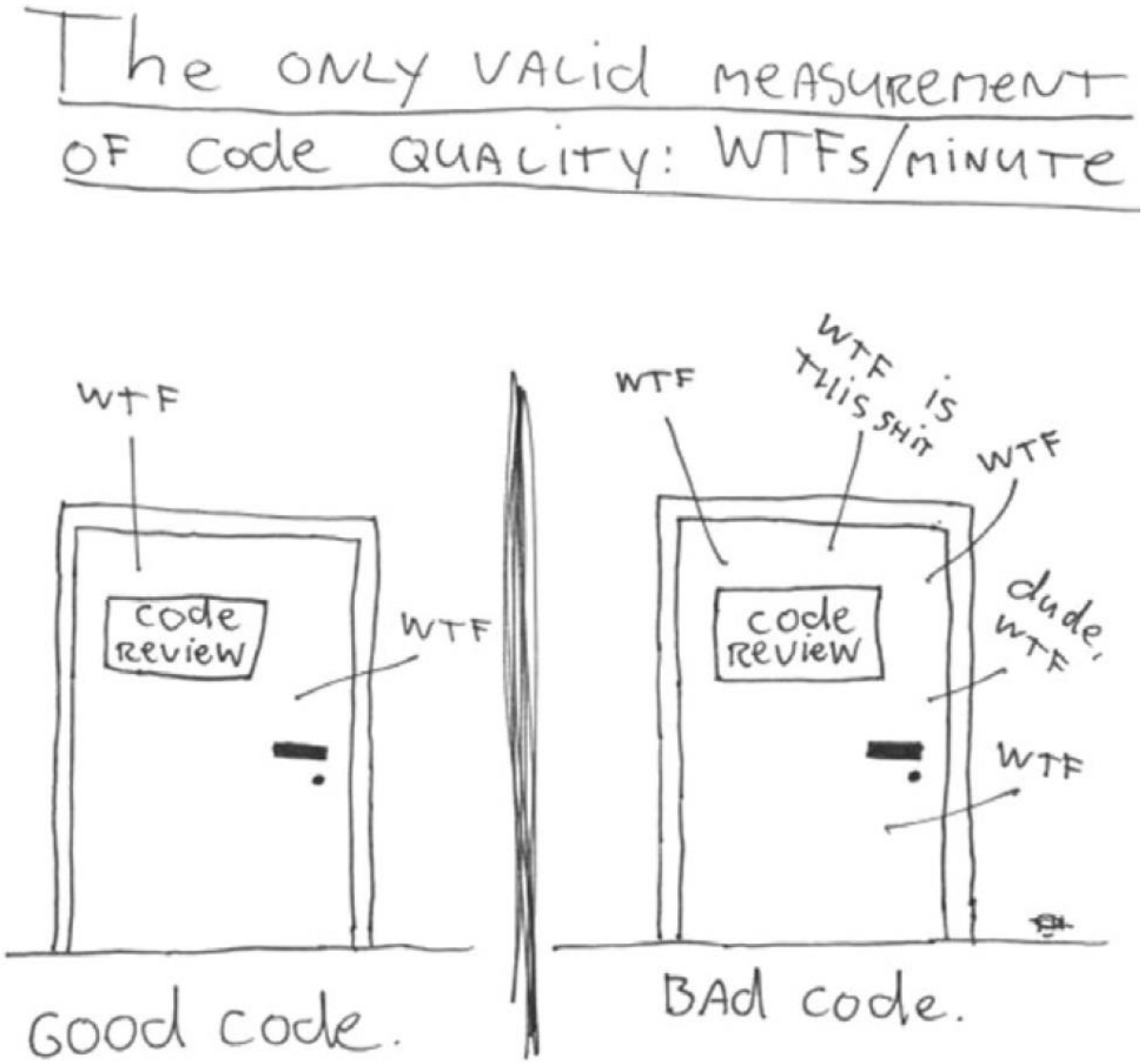
- **"I like my code to be elegant and efficient.**
  - The `logic` should be `straightforward` to make it hard for bugs to hide
  - The dependencies minimal to `ease` `maintenance`
  - `Error` `handling` complete according to an articulated strategy
  - `Performance` close to `optimal` so as not to tempt people to make the code messy with unprincipled optimizations
- **Clean code does one thing well."**

What Grady Booch said about code

- **Clean code is `simple` and `direct`.**
- **Clean code reads like `well-written` prose.**
- **Clean code never obscures the designer's intent but rather is full of crisp `abstractions` and `straightforward` lines of control.**



Code quality measurement: WFTs/min ☺



形式规范：一个例子

代码清单10-1 badly formatted code – big C

```
            #include "stdafx.h"
          #include                  "stdio.h"
        void test();
                  int _tmain
                  (int argc,
                     _TCHAR*                argv[])
                        { test(); return
                                       0; }

                                  char C[25][40];void d(int x,int y)
              {C[x][y]=C[x][y+1]=32;}int f(int x){return (int)x*x*.08;}
              void test(){int i,j;                                        char s[5]="TEST";
       for(i=0;i<25;i++)
      for(j=0;j<40;j++)
    C[i][j]=s[(i+j)%4];
  for(i=1;i<=7;i++)
  {d(18-i,12);
  C[20-f(i)][i+19]=
   C[20-f(i)][20-i]=32;
    }d(10,13);d(9,13);
     d(8,14);d(7,15);
          d(6,16);d(5,18);d(5,20);                                  d(5,22);d(5,26);
               d(6,23);d(6,25);d(7,25);for(i=0;i<25;i++,printf("\n"))
                                  for(j=0;j<40;printf("%c",C[i][j++]));}
```

From 《现代软件工程讲义》

Code Readability Example

Example #a

```
z = ((3*x^2) + (4*x) – 5) – ((2*y^2) – (7*y) + 11) / ((3*x^2) + (4*x) – 5)
```

vs.

Example #b

```
a = ((3*x^2) + (4*x) – 5)
b = ((2*y^2) – (7*y) + 11)
z = (a – b) / a
```

"Although both examples are comprehensible, example b is comprehensible with greater ease (i.e., more readable) then example a."

形式规范

- **标准：简明，易读，无二义性。**
- **例如：**
  - 缩进、对齐
  - 空行、行内空格
  - 行宽与分行
  - 括号
  - 变量和函数命名
  - 下划线/大小写
  - 注释
  - etc

From 《现代软件工程讲义》

Length of names

- The longer the names of classes, variables, methods, etc., the more descriptive they probably are**, to accurately reflect the purpose of the entity.**
  - Names should be descriptive. The longer the name, the more descriptive it is likely to be.
  - Names should also be succinct. The longer the name, the less efficient it is likely to be.
  - `a` is not a good variable name, `Age` is better, but `EmployeeAge` seems much more descriptive.
  - Generally, names consisting of 1 or 2 letters are not good. What is enough depends on your language and the application you're making.
- **Metrics: Average length of *all* names**

➔ Good naming conventions

Name Uniqueness Ratio (UNIQ)

- **Name Uniqueness Ratio (UNIQ)**
  - When two entities have the same name, it's possible that they get mixed. UNIQ measures the uniqueness of all names.
  - It's acceptable to use the same name at many locations. However, the name should refer to the same logical thing.
- For example, variable `userName` should always contain the same type of username in the same data type (`string`). If `userName` can mean one thing in one procedure and another thing somewhere else, the likelihood of confusion increases.

Complexity and LoC

- **Complexity**
  - Complex code isn't likely to be understandable.
- **Lines of code (LoC)**
  - The longer a method gets, the harder it probably is to understand.

Comment density (MCOMM%)

- **The more comments in your code, the easier it is to read - and understand.**
  - Whitespace lines are also important for legibility.
- **Note: Not all comments contain a description of what's happening.** Some comments are simply separators, such '----------. So, it's more sensible to pay attention to meaningful comments and not just any comments.
  - A profusion of comments provides an easy-to-follow natural-language narrative.
- **Measurement:** how many meaningful comments there are per each logical line of code.

```
MCOMM% = MCOMM / LOC
```

形式规范：空行

- **在源代码中适度的使用空行可以使程序结构更加清晰**

- **两个空行分隔一般出现在源文件的各个节之间；**
- **单个空行分隔一般出现在：**
  - 方法定义之间；
  - 局部变量定义和第一个语句之间；
  - 注释之前；
  - 语句的不同逻辑分段之前；

形式规范：空格

- **一行代码只写一条语句，避免使用复杂的语句行；**

- **行内空格一般出现在：**
  - 关键字和括号之间；
  - 参数列表中的逗号之间；
  - 二元运算符之间；
  - 语句中的表达式之间；
  - 赋值符号之间；

```
while(true) {
     ...
}

a+=c+d;

while(d++=s++) {n++;}

printf("size is "+foo+"\n");

for(expr1;expr2;expr3)

myMethod((int)(cp+5),((int)(i+3)+1));
```

形式规范：分行

- 一个代码行最大长度最好控制在70~80个字符之内，超过这个范围应分成多行；
- 分行的位置应在逗号之后、操作符之前；
- 长表达式应在低优先级的操作符处拆分成新行，操作符放在新行之首；
- 拆分出的新行要进行适当的缩进(一般为8个空格)，使排版整齐，语句可读。

形式规范：分行

```
var = someMethod1(longExp1,someMethod2(longExp2,
       longExp3));

longName1 = longName2 * (longName3 + longName4
       -longName5) + 4 * longname6;

if ((condition1 && condition2) || (condition3 &&
   condition4) || !(condition5 && condition6)) {
   doSomethingAboutIt();
}
```

形式规范：分行

```
for (int i = 1; i <= n-1; i++)
{ int t = i; for (int j = i+1; j
<= n; j++) { if ( a[j] < a[t]) {
t = j; if (t <> i) { int work =
a[t]; a[t] = a[i]; a[i] =
work; }}}}
```

```
for (int i = 1; i <= n-1; i++) {
   int t = i;
   for (int j = i+1; j <= n; j++)
   {
        if ( a[j] < a[t]) {
           t = j;
           if (t <> i) {
                int work = a[t];
                a[t] = a[i];
                a[i] = work;
           }
        }
   }
}
```

形式规范：命名规则

- **标识符：类名、变量名、函数/方法名、参数名**
  - 命名应当直观，可以“望文生义”，最好采用英文单词或其组合，应能反映它所代表的实体，意义明确；
  - 命名的长度应符合“最小长度下的最大信息量”原则，过长的英文单词应采通用而合理的缩写；
  - 一般禁止使用单字符变量名(局部循环变量除外)；
  - 不要仅依靠大小写来区分相似的标识符；
  - 不要出现局部变量和全局变量重名的现象；
  - 变量名：使用“名词”或“形容词+名词”的形式；
  - 函数/方法名：使用“动词”或“动词+名词”的形式。

形式规范：命名规则

- 类的命名采用以大学字母开头的单词组合而成；
- 常量名采用全大写的字母单词组合而成，单词之间用下划线分隔；
- 变量名和参数名采用第一个单词首字母小写而后面的单词首字母大写的单词组合；
- 可以使用前缀s_表示静态变量，使用g_表示全局变量，使用m_表示类的成员变量，使用p_表示指针，使用b_表示布尔类型的变量；
- 函数/方法名采用第一个单词首字母小写而后面的单词首字母大写的单词组合。

形式规范：命名规则

```
class CourseOffering {

    static final int MIN_WIDTH = 4;
    static final int GET_THE_CPU = 1;

    int g_numStudents;
    int m_width, m_height;

    void setValue ( int width, int height ) {
         m_width = width;
         m_height = height;
     }
     void calcAvgScore() {
          int i;
          for ( i=0; I < g_numStudents; i++) {…}
     }
}
```

形式规范：声明

- 尽量在变量声明时进行初始化，只有当初值依赖于某些计算得到的值时可以不对其进行初始化；

```
private boolean isDirty = false;


private boolean isDirty;
…
isDirty = getState();
```

形式规范：声明

- 变量声明仅放在代码块的开头，除了循环语句之外，一般不要在首次使用时才进行声明；

```
void myMethod () {
    int int1 = 0; //在函数块前面声明变量
    if (condition) {
        int int2 = 0; //在if语句块前面声明变量
        …
    }
}
//for循环语句中进行循环变量的声明和初始化
for(int i=0; i<maxLoops; i++) {..}
```

形式规范：声明

- 每行只声明一个变量，尤其避免同一行内声明不同类型的变量；

```
int level, size;
int foo, foo_array[];
```

- 变量名不要重复；

```
int count;
...
myMethod() {
    if (condition) {
        int count;
        ...
    }
}
```

形式规范：语句结构

- 避免使用空的**else**语句和嵌套的条件判断语句

```
if (condition) {
    …
}
else {}
```

```
if (condition1) {
    if (condition2) { }
    else if (condition 3)
{}
}
else {
    if (condition 4) {}
}
```

- 避免采用过于复杂的条件测试

```
if ((a>b && ((c<d) || d!=0))
    || (a<=b && ((c>=d) || d==0))
    || e+f<=100 )
{…}
```

- 尽量减少使用“否定”条件的条件语句

```
if (! ( ( c < '0' ) || (c > '9' )))
{…}
```

形式规范：语句结构

- 避免书写复杂的表达式

```
*x +=(*xp=(2*k < n-m) ? C[k+1]:d[k--1]));
```

```
if (2*k < n-m)
      *xp = C[k+1];
else
      *xp = d[k--1];

*x += *xp
```

形式规范：注释

- **三种格式：**
  - 文档风格：一般直接写在类、方法、函数的前面；
  - C语言风格：采用“/*…*/”格式，通常是多行注释，可以用来描述算法等多行的文字；
  - 单行注释：采用“//…”格式，简单说明该行语句的作用。

文档风格的注释

- **通常置于每个程序模块的开头部分，给出程序的整体说明，对于理解程序本身具有引导作用：**
  - 程序标题；
  - 有关本模块功能和目的的说明；
  - 主要算法；
  - 接口说明： 包括调用形式，参数描述，子程序清单；
  - 有关数据描述：重要的变量及其用途，约束或限制条件，以及其它有关信息；
  - 给函数和全局数据加注释
  - 模块位置：在哪一个源文件中，或隶属于哪一个软件包；
  - 开发简历：模块设计者，复审者，复审日期，修改日期及有关说明等。

形式规范：注释

- 程序中不能没有注释，但注释也不能过多；
- 不必要注释含义已经十分清楚的代码；
- 修改代码时应该同时修改注释，以保证注释和代码的一致性
- 注释应当准确易懂，防止出现二义性
- 注释的位置应该与被描述的代码相邻，应该写在程序代码的上方并且和代码左对齐
- 变量定义和分支语句(条件分支、循环语句等)必须写注释，因为这些语句往往是程序实现某一特定功能的关键。

内容规范

- **代码设计规范不光是程序书写的格式问题，而且牵涉到程序设计、模块之间的关系、设计模式等方方面面。**
- **几个例子：**
  - 函数：现代程序设计语言中的绝大部分功能，都在函数(Function, Method)中实现，关于函数最重要的原则是：只做一件事，但是要做好(单一责任原则)。
  - 错误处理：80%的程序代码，都是对各种已经发生和可能发生的错误的处理，如果你认为某事可能会发生，这时就要用错误处理。
  - 参数处理：所有的参数都要验证其正确性，使用“断言”(Assert)；

From 《现代软件工程讲义》

类的编码规范

- 类
  - 使用类来封装面向对象的概念和多态（Polymorphism）。
  - 避免传递类型实体的值，应该用指针传递。换句话说，对于简单的数据类型，没有必要用类来实现。
  - 对于有显式的构造和析构函数，不要建立全局的实体，因为你不知道它们在何时创建和消除。
  - 只有在必要的时候，才使用“类”。
- Class vs. Struct
  - 如果只是数据的封装，用Struct即可。
- **公共/保护/私有成员Public、Private和Protected**
  - 按照这样的次序来说明类中的成员：public、protected、private

From 《现代软件工程讲义》

类的编码规范

- **数据成员**
  - 数据类型的成员用m_name说明。
  - 不要使用公共的数据成员，要用inline访问函数，这样可同时兼顾封装和效率。
- **虚函数Virtual Functions**
  - 使用虚函数来实现多态（Polymorphism）。
  - 只有在非常必要的时候，才使用虚函数。
  - 如果一个类型要实现多态，在基类（Base Class）中的析构函数应该是虚函数。
- **构造函数Constructors**
  - 不要在构造函数中做复杂的操作，简单初始化所有数据成员即可。
  - 构造函数不应该返回错误（事实上也无法返回）。把可能出错的操作放到HrInit()或FInit()中。

From 《现代软件工程讲义》

类的编码规范

- **析构函数**
  - 把所有的清理工作都放在析构函数中。如果有些资源在析构函数之前就释放了，记住要重置这些成员为0或NULL。
  - 析构函数也不应该出错。
- **New和Delete**
  - 如果可能，实现自己的New/Delete，这样可以方便地加上自己的跟踪和管理机制。自己的New/Delete可以包装系统提供的New/Delete。
  - 检查New的返回值。New不一定都成功。
  - 释放指针时不用检查NULL。
- **运算符（Operators）**
  - 在理想状态下，我们定义的类不需要自定义操作符。只有当操作符的确需要时。
  - 运算符不要做标准语义之外的任何动作。例如，“==”的判断不能改变被比较实体的状态。
  - 运算符的实现必须非常有效率，如果有复杂的操作，应定义一个单独的函数。
  - 当你拿不定主意的时候，用成员函数，不要用运算符。

From 《现代软件工程讲义》

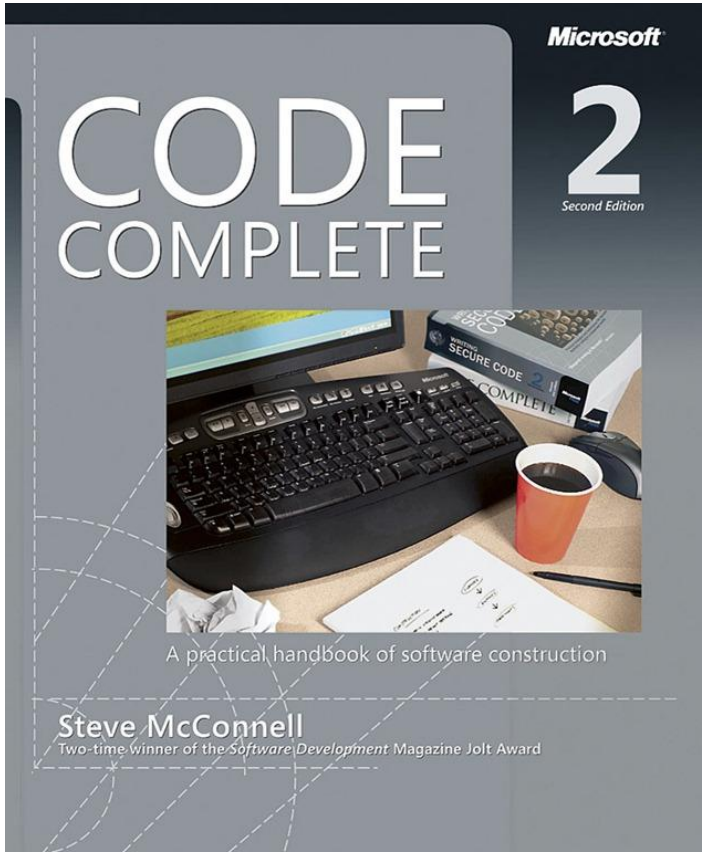
类的编码规范

- **异常（Exceptions）**
  - 异常是在“异乎寻常”的情况下出现的，它的设置和处理都要花费“异乎寻常”的开销，所以不要用异常作为逻辑控制来处理程序的主要流程。
  - 了解异常及处理异常的花销，在C++语言中，这是不可忽视的开销。
  - 当使用异常时，要注意在什么地方清理数据。
  - 异常不能跨过DLL或进程的边界来传递信息，所以异常不是万能的。
- **类型继承（Class Inheritance）**
  - 当有必要的时候，才使用类型继承。
  - 用Const标注只读的参数（参数指向的数据是只读的，而不是参数本身）。
  - 用Const标注不改变数据的函数。

From 《现代软件工程讲义》

程序员必读：《代码大全》(Code Complete)

- **第5-19章：**
  - 高质量子程序的特点
  - 模块化设计
  - 高级结构设计
  - 生成数据
  - 数据名称
  - 变量
  - 基本数据类型
  - 复杂数据类型
  - 顺序结构语句
  - 条件语句
  - 循环语句
  - 少见的控制结构
  - 场景的控制问题
  - 布局和风格
  - 文档



不是需要死记硬背的东西，在编程中多观察体验，逐渐总结经验，符合编码规范。

课堂讨论

- 你认为编程规范真的会起作用吗？难道不会降低我的创造力和工作效率吗？
- 不过，为何那么多大公司针对各种编程语言制订了严格的编码规范？
- 甚至像Python这样的语言，连“缩进”都成了它语法的一部分？
  - Python开发者有意让违反了缩进规则的程序不能通过编译，以此来强制程序员养成良好的编程习惯。
- 你怎么看？

4 代码复审：静态程序分析

代码复审(Review)

- **代码复审：代码是否在“代码规范”的框架内正确地解决了问题**

名 称	形 式	目 的
自我复审	自己 vs. 自己	用同伴复审的标准来要求自己。不一定最有效，因为开发者对自己总是过于自信。如果能持之以恒，则对个人有很大好处
同伴复审	复审者 vs. 开发者	简便易行
团队复审	团队 vs. 开发者	有比较严格的规定和流程，用于关键的代码，以及复审后不再更新的代码。 覆盖率高——有很多双眼睛盯着程序。但是有可能效率不高（全体人员都要到会）

From 《现代软件工程讲义》

代码复审的目的

- **找出代码的错误：**
  - 编码错误，比如一些能碰巧骗过编译器的错误。
  - 不符合项目组的代码规范的地方。
- **发现逻辑错误，程序可以编译通过，但是代码的逻辑是错的。**
- **发现算法错误，比如使用的算法不够优化。**
- **发现潜在的错误和回归性错误——当前的修改导致以前修复的缺陷又重新出现。**
- **发现可能改进的地方。**
- **教育(互相教育)开发人员，传授经验，让更多的成员熟悉项目各部分的代码，同时熟悉和应用领域相关的实际知识。**
- **代码复审中的提问与回应能帮助团队成员互相了解，就像练武之人互相观摩点评一样。在一个新成员加入一个团队的时候，代码复审能非常有效地帮助新成员了解团队的开发策略、编程风格及工作流程。**

From 《现代软件工程讲义》

代码复审的步骤

- **在复审前——**
  - 代码必须成功地编译。
  - 程序员必须测试过代码。
- **“你这样修改了之后，有没有别的功能会受影响?”**
- **“项目中还有别的地方需要类似的修改么?”**
- **“有没有留下足够的说明，让将来维护代码时不会出现问题？”**
- **“对于这样的修改，有没有别的成员需要告知?”**
- **“导致问题的根本原因是什么？我们以后如何能自动避免这样的情况再次出现？”**

From 《现代软件工程讲义》

代码复审的核查表：以子程序为例

- **总体问题：**
  - 如果把程序中的某些部分独立成子程序会更好?
  - 是否用了明晰清楚的动宾词组对过程进行命名？
  - 是否是用返回值的描述来命名函数？
  - 子程序的名称是否描述了它的所有功能？
  - 子程序是不是只做一个事情，且是较强的功能内聚性？
  - 子程序间是是否松散耦合？
  - 子程序的长度是否它的功能和逻辑决定?

代码复审的核查表：以子程序为例

- **参数传递问题**
  - 形式参数与实际参数匹配吗？
  - 子程序中参数的排列合理吗？与相似子程序中的顺序一致吗？
  - 接口是否进行了说明？
  - 子程序中参数个数是不是7 个或者更少？
  - 是否只传递了结构化变量中子程序用得到的部分？
  - 是否用到了每一个输入参数和输出参数？
  - 是否在所有情况下它都会返回一个值？

代码复审的核查表：以子程序为例

- **防错性编程**
  - 是否使用了断言？
  - 子程序对于非法输入数据进行防护了吗？
  - 子程序是否能很好地进行程序终止？
  - 子程序是否能很好地处理全局变量修改情况？
  - 是否不用很麻烦地启用或去掉调试帮助？
  - 是否信息隐蔽、松散耦合，以及使用输出数据检查？
  - 子程序是否检查返回值？

走查(Walkthrough)

- **走查(Walkthrough)**
  - 由设计人员或编程人员组成一个走查小组，通过阅读一段文档或代码，并进行提问和讨论，从而发现可能存在的缺陷、遗漏和矛盾的地方。
  - 类型：设计走查、代码走查。
- **走查过程**
  - 与评审过程类似，即先把材料先发给走查小组每个成员，让他们认真研究程序，然后再开会；
  - 与评审的区别：评审通常是简单地读程序或对照错误检查表进行检查；走查则是按照所提交的测试用例，人工模仿计算机运行一遍，并记录跟踪情况。

走查(Walkthrough)

- **走查是开发者的一次友好的会议，需要仔细规划，并有明确的目的、日程、持续时间和参与人员，许多小组以星期为单位走查。**
  - 走查前几天：召集人将收集的一些要在会上审查的材料(模型、文档、程序代码等)分发给参与者，参与者研究这些材料并在会议之前提交意见。
  - 会议期间：召集人提出大家的意见并对每一项进行讨论。
- **会议时间比较短，一般2-3 小时。**
- **会议的目的是查明问题，而不是干扰开发者。**
- **会议的思想是确认问题的存在，甚至不必去谋求问题的解决。**
  - 会后：将问题分发给相应人员进行解决。

使用静态代码分析发现潜在bug

- **静态代码分析(static code analysis)：在代码构建过程中帮助开发人员快速、有效的定位代码缺陷；**
  - 无需运行被测代码，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性，找出代码隐藏的错误和缺陷，如参数不匹配，有歧义的嵌套语句，错误的递归，非法计算，可能出现的空指针引用等等。
- **“30~70%的代码逻辑设计和编码缺陷是可以通过静态代码分析来发现和修复的”。**
- **主要技术：缺陷模式匹配、类型推断、模型检查、数据流分析等。**
- **常用的Java静态代码分析工具：**
  - Checkstyle
  - FindBugs
  - PMD

5 动态程序分析（Profiling）

程序性能分析

- 性能分析(performance analysis，也称为profiling)：以收集程序运行时信息为手段研究程序行为的分析方法，是一种动态程序分析(dynamic program analysis)的方法。
- 这种方法与静态代码分析相对，主要测量程序的空间或时间复杂度(complexity)、特定指令的使用情形、函数调用的频率及运行时间(frequency and duration)等。
- 性能分析的目的在于决定程序的哪个部分应该被优化(optimization)，从而提高程序的速度或者内存使用效率。
- 性能分析可以由程序的源代码或是可执行文件进行，使用性能分析工具(profiler)。

程序性能分析的两种方法

- **抽样(Sampling)**
  - 当程序运行时，分析工具时不时看一看这个程序运行在哪一个函数内，并记录下来，程序结束后，工具就会得出一个关于程序运行时间分布的大致的印象。这种方法的优点是不需要改动程序，运行较快，可以很快地找到瓶颈。但是不能得出精确的数据，代码中的调用关系(CallTree)也不能准确表示。
- **代码注入(Instrumentation)**
  - 将检测的代码加入到每一个函数中，这样程序的一举一动都被记录在案，程序的各个效能数据都可以被精准地测量。这一方法的缺点是程序的运行时间会大大加长，还会产生很大的数据文件，数据分析的时间也相应增加。同时，注入的代码也影响了程序真实的运行情况。
- **一般的做法：先用抽样的方法找到效能瓶颈所在，然后对特定的模块用代码注入的方法进行详细分析**

http://www.cnblogs.com/xinz/archive/2011/11/20/2255809.html

关于性能分析

- 性能测试→分析→改进→性能测试
- 先debug，再性能分析
- 先保证正确性，再提高效能

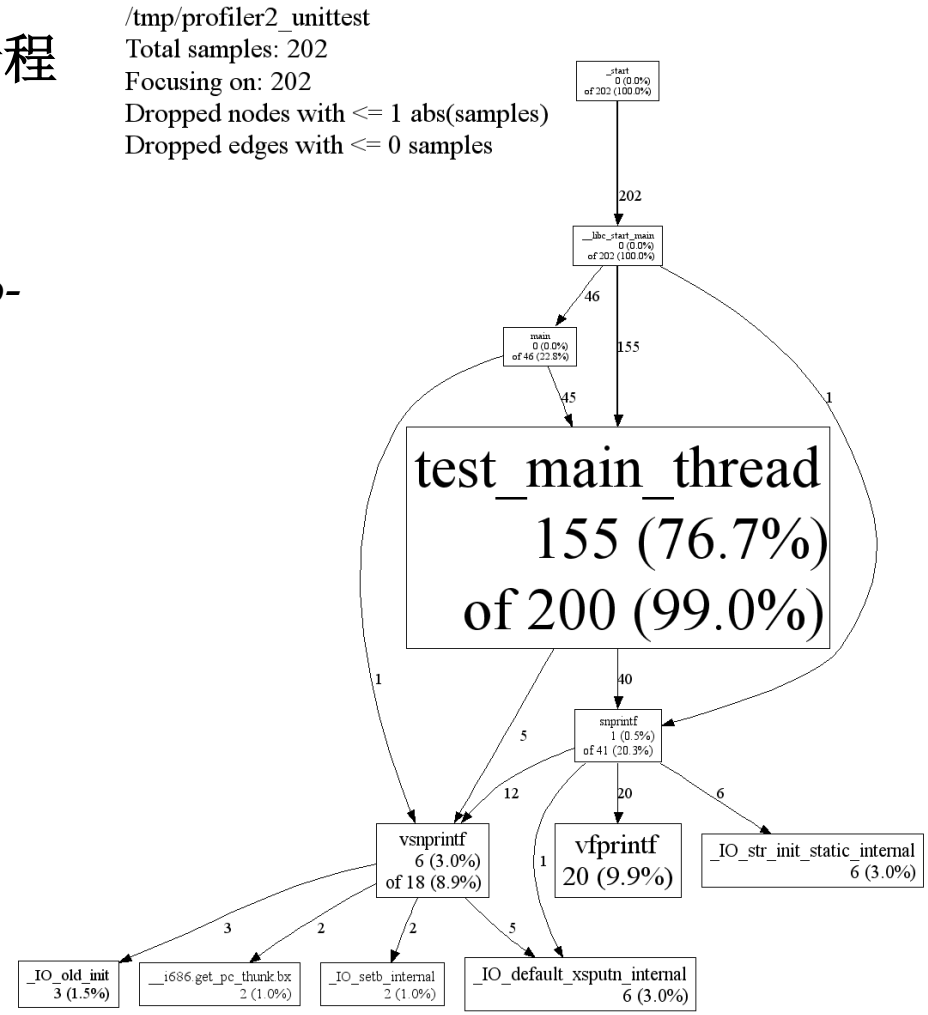
- 是否会用性能分析工具来提高程序质量是一个优秀程序员的标志之一

性能分析的工具

- 不同的编程语言有不同的性能分析工具；
- http://en.wikipedia.org/wiki/List_of_performance_analysis_tools进行了一个很好的汇集；
- 常用：
  - Google提供的gperftools，面向C/C++等
  - Visual Studio Team System Profiler，Microsoft
  - Test and Performance Tools Platform (TPTP)，Eclipse/Java
  - cProfile, profile和pstats，Python

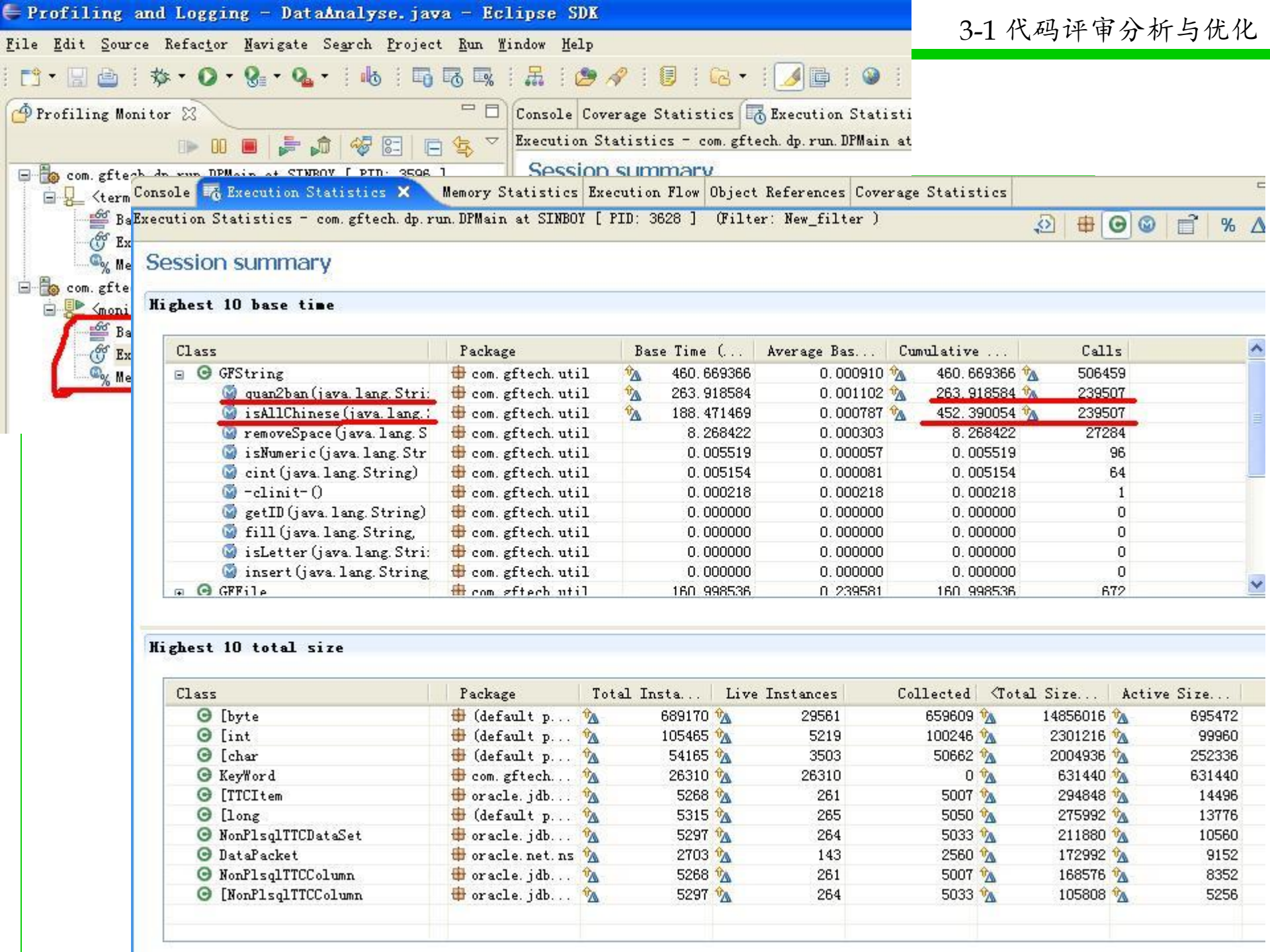
以gperftools (C/C++)为例

- http://code.google.com/p/gperftools
- **Google提供的一套工具，用于分析程序性能，找到程序的性能瓶颈。**
  - malloc free内存分配器tcmalloc；
  - 堆内存检测和内存泄漏分析工具heap-profiler和heap-checker；
  - CPU性能监测工具CPU-profiler。



以TPTP (JAVA)为例

- http://www.eclipse.org/tptp
- **Eclipse Test & Performance Tools Platform，作为Eclipse的插件。**
- **重点关注其中的Tracing & Profiling Tools。**
  - 内存分析(Basic Memory Analysis)；
  - 执行时间分析(Execution Time Analysis)；
  - 代码覆盖(Method Code Coverage)；


Profiling and Logging - DataAnalyse.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Profiling Monitor
Console Coverage Statistics Execution Statisti
Execution Statistics - com.gftech.dp.run.DPMain at
Session summary
Console Execution Statistics Memory Statistics Execution Flow Object References Coverage Statistics
Execution Statistics - com.gftech.dp.run.DPMain at SINBOY [PID: 3628] (Filter: New_filter)
Session summary
Highest 10 base time
Class Package Base Time (... Average Bas... Cumulative ... Calls
GFString com.gftech.util 460.669366 0.000910 460.669366 506459
quan2ban(java.lang.Stri: com.gftech.util 263.918584 0.001102 263.918584 239507
isAllChinese(java.lang.: com.gftech.util 188.471469 0.000787 452.390054 239507
removeSpace(java.lang.S com.gftech.util 8.268422 0.000303 8.268422 27284
isNumeric(java.lang.Str com.gftech.util 0.005519 0.000057 0.005519 96
cint(java.lang.String) com.gftech.util 0.005154 0.000081 0.005154 64
-clinit-() com.gftech.util 0.000218 0.000218 0.000218 1
getID(java.lang.String) com.gftech.util 0.000000 0.000000 0.000000 0
fill(java.lang.String, com.gftech.util 0.000000 0.000000 0.000000 0
isLetter(java.lang.Stri: com.gftech.util 0.000000 0.000000 0.000000 0
insert(java.lang.String com.gftech.util 0.000000 0.000000 0.000000 0
GFFile com.gftech.util 160.998536 0.239581 160.998536 672
Highest 10 total size
Class Package Total Insta... Live Instances Collected Total Size... Active Size...
[byte (default p... 689170 29561 659609 14856016 695472
[int (default p... 105465 5219 100246 2301216 99960
[char (default p... 54165 3503 50662 2004936 252336
KeyWord com.gftech... 26310 26310 0 631440 631440
[TTCItem oracle.jdb... 5268 261 5007 294848 14496
[long (default p... 5315 265 5050 275992 13776
NonPlsqlTTCDataSet oracle.jdb... 5297 264 5033 211880 10560
DataPacket oracle.net.ns 2703 143 2560 172992 9152
NonPlsqlTTCColumn oracle.jdb... 5268 261 5007 168576 8352
[NonPlsqlTTCColumn oracle.jdb... 5297 264 5033 105808 5256

结束

2017年9月27日