关于代码调试
DE
那些事

wklken

理想状态

想清楚，码，运行，

perfect, DONE, 下班

现实是: 坑

- 现实是, 我们会被各种坑
- 被环境坑, 被语言坑, 被依赖坑, 被第三方库坑, 被编辑器坑, 被自己坑(三个月前的自己/昨天的自己/几分钟前的自己), 被数据库坑, 被缓存坑, 被队友坑(这个比较惨), 被需求变更坑(这个也是).....

所以，调试是必不可少的

前奏

在早期尽量避免bug的出现

想清楚了再写代码

想清楚了再写代码

- 如果连需求是什么, 想要做什么都没整明白, 就吭哧吭哧开写, 意图在实践中摸索通向胜利的道路, 是很愚蠢的行为.

想清楚了再写代码

- 整体架构
- 流程分析
- 数据流转
- 细节

想清楚了再写代码

- 脑图
- 逻辑分支分析(伪代码)
- 注释大纲

想清楚了再写代码

```
def dosomething():
    """
    """
    # step0: get params

    # step1: params validate

    # step3: begin process

    # step3.1: xxxx

    # step3.2: xxxx

    # step4: convert result

    # step5: render and return
```

第一时间review

第一时间review

- 写完一段代码第一时间自己review一下
- 变量/数据结构/条件判断/函数调用/参数/返回值/上下文一致性/错误处理/场景完备性……

第一时间review

- 这时候, 是你思路最清晰的时候
- 这时候, 花费的时间是最少的(相比后面调试的时间)
- 这时候, 很多显而易见的问题会被发现

第一时间review

- 所以, 要抑制住刚写完代码立刻运行的那种冲动

语法检查插件

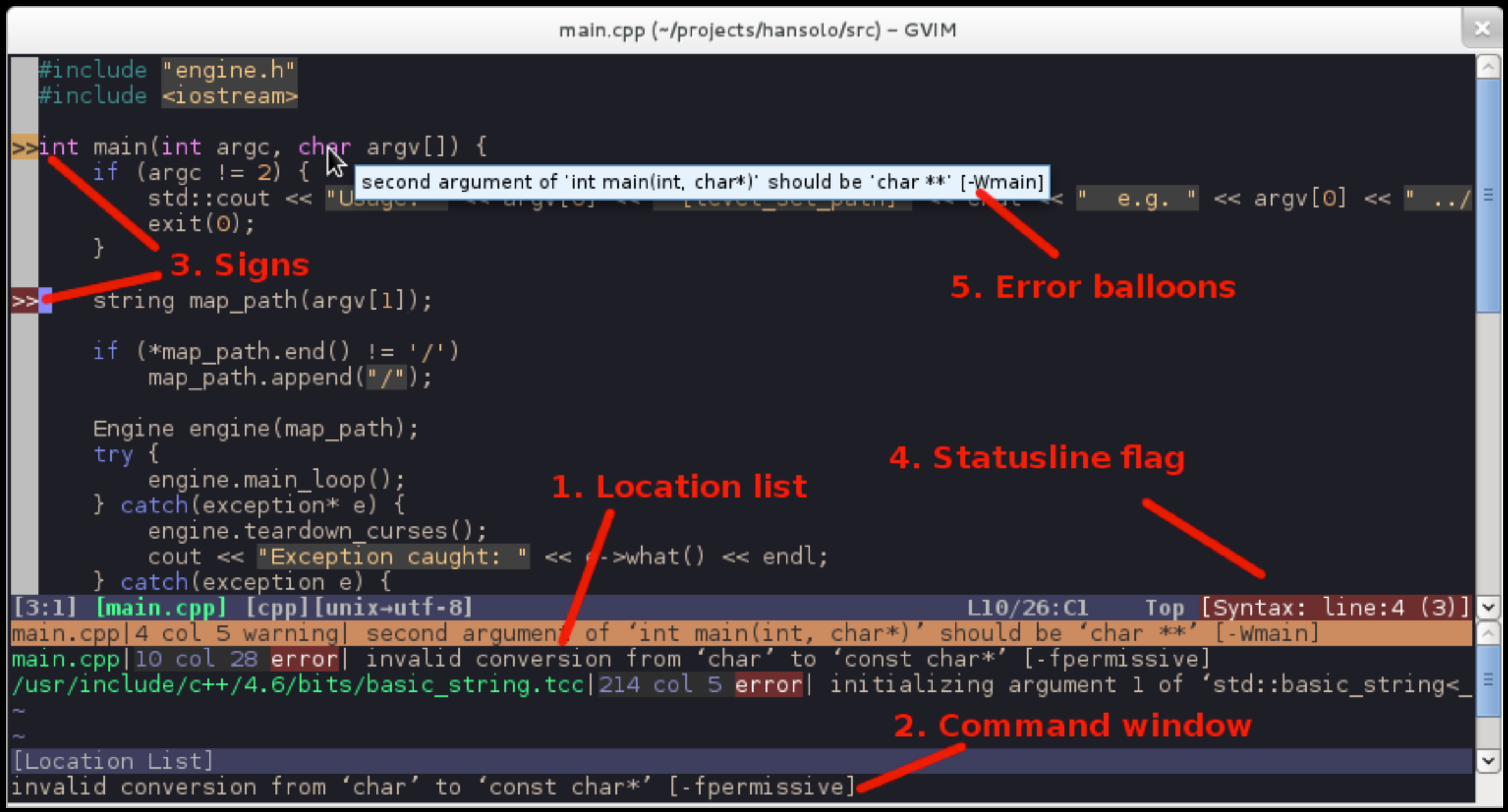
语法检查插件

- 时间, 还是不要花在低级的错误上

语法检查插件

- IndentationError: unexpected indent
- NameError: name 'a' is not defined
- SyntaxError: invalid syntax
- ....

语法检查插件


main.cpp (~/projects/hansolo/src) – GVIM
#include "engine.h"
#include <iostream>
int main(int argc, char argv[]) {
second argument of 'int main(int, char*)' should be 'char **' [-Wmain]
if (argc != 2) {
std::cout << "Usage: " << argv[0] << " [level_set_path]" << endl << " e.g. " << argv[0] << " ../
exit(0);
}
string map_path(argv[1]);
if (*map_path.end() != '/')
map_path.append("/");
Engine engine(map_path);
try {
engine.main_loop();
} catch(exception* e) {
engine.teardown_curses();
cout << "Exception caught: " << e->what() << endl;
} catch(exception e) {
[3:1] [main.cpp] [cpp][unix→utf-8] L10/26:C1 Top [Syntax: line:4 (3)]
main.cpp|4 col 5 warning| second argument of ‘int main(int, char*)’ should be ‘char **’ [-Wmain]
main.cpp|10 col 28 error| invalid conversion from ‘char’ to ‘const char*’ [-fpermissive]
/usr/include/c++/4.6/bits/basic_string.tcc|214 col 5 error| initializing argument 1 of ‘std::basic_string<_
[Location List]
invalid conversion from ‘char’ to ‘const char*’ [-fpermissive]
3. Signs
5. Error balloons
4. Statusline flag
1. Location list
2. Command window

还有一条:

任何情况下，都不要吞掉异常

```
try:
    1 / 0
except:
    pass
```

准备

总要有一些准备的

脚手架代码

脚手架代码

- 可以快速, 加入到现有代码, 打印关键内容(中间值/状态/条件判断)
- 可以快速, 在调试之后找到并删除
- 打印, 足够显眼, 内容足够丰富, 尽量一次定位问题
- 快速: 在编辑器里1-2s内加完

脚手架代码

```
1 # prt<tab>
6 print "TRACK ================= a", a
1
2 # ppint
3 import pprint
4 pprint.pprint(a)
5
6 # rms
7 # TODO: remove this
8 DEBUG = True
9
10 # tr
11 import traceback; traceback.print_exc()
```

vim中定义的代码片段快捷键

其他

其他

- 网络
- 水
- 提前上厕所
- ....

开始

终于可以开工了

保持清醒

保持清醒

- 在不清醒/疲惫/不在状态的时候, 不建议调试大问题
- 无限负能量: 有时候会让你陷入无尽的自我怀疑/迷茫/愤怒/沮丧/窘迫/挫败
  - 自我怀疑 or 怀疑一切 >_<!!!
  - 迷信 >_<#
  - 随机行走调试
  - 在痛苦中挣扎吧..... ORZ

保持清醒

- 不清醒调试, 很容易事倍功半

保持清醒

- 保持清醒的自我, 你得明白你在做什么, 保持清醒
- 要明确: 问题是什么? 现有得到的信息是什么? 我要怎么做?

準備環境

准备环境

- 首先, 你要有环境吧.....
- 确认代码版本一致性
- 确认下环境/数据等, 是否需要一致性

假设代码是正确的

假设代码是正确的

- 先不要动代码, 先不要动代码, 先不要动代码
- 先复现问题, 复现后定位, 定位后再改!

重现问题

重现问题

- “当你发现一个故障时该怎么办？”
- “试着让它再次发生。”

重现问题

- 即: 制造失败
- 问题的症状是什么?
- 复现的步骤

重现问题

- 多试几种情况...输入值的范围/上下限/不同类型/不同格式/错误的格式等

报错信息

报错信息

- 运行代码, 报错了, 有些人会瞬切回编辑器, 开始改代码(作高效状).....其实, 很浪费时间的

报错信息

- 少年, 报错信息你看了么?
- 然后: 看明白了么?

报错信息

- 观察，而不是猜测
- 精确制导才是王道

报错信息

- 错误的类型
- 发生错误的文件/行号/代码
- 当前上下文相关的值

好了，已经可以处理

80%的调试问题

==，别着急，改完了，先review
再提交，防止引入其他问题

剩下20%呢?

一些方法

没有顺序之分，看场景

代码是`抠`过来的么?

- 有时候是从其他地方copy过来的代码
- 但是: 很容易忘记根据当前情况修改必要的值
- 这时候, 可能是变量名上下文意义不同/函数调用参数上下文意义不同, 导致了莫名的bug

当前修改代码老不生效?

- 自己书写改完后的代码, 运行时老是不生效
- 一直在修改, 但是没有什么区别

当前修改代码老不生效?

- 查看的是不是正在调试的页面(坑)
- 改的是不是正确的目录下正确的文件?(大坑)
- 保存了么(编译了么)?(又一个坑)
- 服务重启了么?
- 跟数据库有没有关系/跟缓存有没有关系, 要不要清?
- ……

当前修改代码老不生效?

- print一把

我电脑上是正常的

- 代码是否同一套
- 环境是否一致
- 数据是否一致
- 依赖上游接口返回是否一致

很长的逻辑?

- 函数很长/调用链很长, 代码读很累, 打调试信息打得手疼

很长的逻辑?

- 上二分法吧少年, 无上利器

很多依赖?

- 依赖了A/B/C系统的接口, 依赖了数据库/redis, 依赖了XXX数据, 依赖了YYY配置

很多依赖?

- 排除法!

- 排可能性, 逐一排除, 最终定位

回到正确的代码

- 假设是由于最近的提交导致的，可以尝试回到原先正常的版本，再验证

如果一段代码没动过

- 确实没动过, `真`没动过
- 更有可能是: 依赖(第三方调用结果/返回数据)/数据(表结构/服务返回数据)/环境(数据库/缓存)

bug总是倾向于集中出现的

- bug扎堆
- 之前出过的模块/第三方依赖/环境/数据等
- 根据可能性降序排查

原则: 首先要怀疑自己

- 你不能一旦代码跑不动就怀疑是别人的问题, 然后抛给别人, 这样做同样是很不负责任而且很愚蠢的
- 没有准确定位前, 不要发表任何结论: 一切没有价值的怀疑都是无意义的

原则: 一次只改一个地方

- 你真的看到错误, 应该只修复这个地方

print 还是debug?

- 个人偏好简单粗暴的print, 主要是用的vim+sinppet, 快速高效.
- 当然, 如果用IDE, 也可以用 debug吧, 哪个用哪个
- 十分十分诡异的问题: 上debug, 打断点, 一点点调试吧, 只能这样了.

当一个问题超过半小时

- 歇一歇, 走动走动, 打个水, 呼吸下新鲜空气.
- 这时候有利于脱出情境, 去掉挫败感/愤怒/迷信等
- 很多时候突然灵感一到, 瞬间明了(这种感觉很奇妙)

关于google

- 有些错误信息, 如果觉得比较独特诡异, 可以google下, 你会找到更多的一些信息的.
- 学会用逻辑搜索, 学会全文搜索

关于求助

- 前提, 你自己能把问题想清楚, 并且逻辑清晰地描述出来.(什么业务什么位置的什么逻辑, 报错类型和报错信息, 输入输出, 迄今做了哪些尝试等等) 要学会聪明地问问题, 高效, 尊重自己也尊重别人.
- 有时候, 还没问就秒懂了....小黄鸭调试法
- 求助, 可以获得全新的观点

原则

- 不被同一个问题坑两次
- 同一个问题不求助两次(google两次可以>_<)

END