20 | 生产加速：C 项目需要考虑的编码规范有哪些？

2022-01-28 于航

讲述：于航

时长 13:51 大小 12.69M

你好，我是于航。

在本模块前面的几讲中，我主要介绍了可以为项目编码提速的 C 标准库，以及优化 C 代码的相关技巧。而在接下来的三讲中，我将为你介绍大型 C 项目在工程化协作时需要关注的编码规范、自动化测试和结构化编译。当项目由小变大，参与人数由少变多时，这些便是我们不得不考虑的重要内容。

和一个人参与项目、写代码时的“单打独斗”相比，多人协作从理论上来看可以大幅提高生产效率。但现实情况却可能是，效率在提升的同时，代码质量下降、沟通成本变高等一系列问题也随之而来。甚至在某些情况下，团队人数的增加反而会导致项目推进效率的降低。

那为什么会出现这样的问题呢？这是因为，当参与到项目开发中的人员数量逐渐增多时，工程师们对于代码编写规范，以及项目开发生命周期（SDLC，Software Development Life Cycle）等关键事项没有形成统一的标准。而因为代码审查不通过导致的频繁返工甚至妥协，

以及协作流程上的不明确与延期，使得项目的迭代周期变长，进而生产效率下降。因此，如何为团队制定统一的编码规范，并明确 SDLC 的整体流程以及其中各节点的重点注意事项，就成为了决定团队协作效率的一个重要因素。

那么，这一讲我们就来聊一聊，对于使用 C 语言编写的、需要多人协作的项目，我们应该从哪些角度来制定团队的编码规范。这里我会以 GNU C 编码规范为例来进行介绍，在此之前，我们先来看看什么是 GNU。

GNU 与 GUN？傻傻分不清楚

你应该对 GNU 这个由三个英文字母组成的标识并不陌生，虽然有时我也会把它与“文明用语” GUN 搞混。不过你可能还不清楚，**它实际上是 GNU’s Not Unix 的递归首字母缩写词，而且读音类似于中文“革奴”的发音**。

具体来讲，GNU 最初是指由 Richard Matthew Stallman 于 1983 年创立的一项计划，该计划旨在创建一套完全自由的操作系统和相关系统软件的集合。该计划执行至今，除了仍然处在开发阶段的 GNU 操作系统内核 Hurd 以外，其他相关系统软件都已经得到了社区的广泛应用。其中不乏我们熟知的 GCC 编译器、GDB 调试器，以及 Emacs 文本编译器等。目前（截止到 2022 年 1 月），一共有 384 个 GNU 软件包被托管到 GNU 的官方开发站点上。其中，GCC 和 GDB 的第十二个大版本也处在紧张的开发过程中。

当年发起 GNU 计划的一个初衷，是让业界重视软件界合作互助的团结精神，而在经过了将近 40 年的发展后，这一精神可以说在 GNU 旗下的项目上被体现得淋漓尽致。以 GCC 为例，就该项目而言，被正式记载到官方贡献者列表中的开发者就超过了 500 人。而如果将它的 Git 仓库下载下来，并通过相关命令统计所有提交过代码的贡献者人数（根据邮箱区分），这个数字将高达 3500。由此可见，GCC 作为 GNU 旗下使用人数最多的开源项目之一，它能够一直以来被社区广泛采纳，除了源于自身极高的项目质量外，也离不开贡献者之间默契的协作，其中就包括在编码规范上的统一。

为了保证 GNU 旗下的开源项目都能够采用统一的代码格式，Richard Stallman 和一众来自 GNU 开源项目的志愿者，从 1992 年起便开始着手制定一套统一的编码规范，GNU Coding Standards（后简称“GNU 编码规范”）。制定这套编码规范，是为了保持 GNU 操作系统以及相关软件有干净、一致的编码风格，并保持足够高的可移植性与可靠性。这套编码规范更适用于 C 语言编写的程序，但其中的许多规则与原则也同样适用于其他编程语言。

接下来，我们就以 GNU 编码规范为例，来看一看围绕 C 语言项目制定编码规范时，具体都需要考虑哪些方面。在充分了解这些内容后，你就可以根据自己团队与项目的情况，来选择性制定更加合适的编码规范了。

GNU C 编码规范

GNU 编码规范将 C 语言项目需要考虑的编码细则分为了多个不同类别，下面我们来依次看看这些内容中最常用的 9 个方面。至于更具体的信息，你可以参考这个链接。

格式

这一类别中的规则主要约束了 C 代码的具体编写格式。比如，GNU 规范建议将每一行代码的字符数量保持在 79 个以内，以便在大多数情况下获得最好的代码可读性；C 函数定义时使用的开始花括号 “{” 建议作为每行的第一个字符，这样有助于兼容大多数代码分析工具。而类似地，为了更好地识别函数定义，函数名称同样需要放置于每行的起始位置，同时也利于某些特定工具对函数的识别。

除此之外，在函数定义或声明中，若函数参数过长，则需要将超过行长度限制的参数放到函数名所在的下一行，并与上一行中第一个参数的开头保持对齐。而对于结构或枚举类型，可以选择在符合行长度限制的情况下，将它们在一行内完整定义。或采用类似函数的定义方式，将包裹定义的开始与结束花括号放置在单独一行，两行之间为具体的定义内容。

你可以看下这个例子，它便符合我在上面提到的几点 GNU 编码规范对代码格式的要求：

复制代码

```
int
lots_of_args (int an_integer, long a_long, short a_short,
              double a_double, float a_float)
{
  // ...
}
struct foo
{
  int a, b;
}
struct bar { int x, y; }
```

这里我们提到的仅是一些较为重要的代码格式化规则，但除此之外，还有很多其他 GNU 代码格式要求的规范细节。关于这些内容，你可以参考 GNU 旗下的 `indent` 工具。

这个工具提供了多种不同选项，可以对 C 源代码应用多种不同的代码格式化风格，你可以点击这个链接来了解这些信息。这些风格主要针对缩进、括号，以及换行等字符或语法元素在特定代码场景中的不同使用方式。但无论哪种方式，**你都需要确保在同一个 C 项目中，仅使用唯一的一种风格（规则）来格式化你的代码**。

注释

注释的用途显而易见。一个优秀程序的最基本特征，就是可以让任何一个工程师都能在最短的时间内，了解这个程序的基本实现逻辑。而注释就是达成这一目的的“秘密武器”。

在文件方面，GNU 规范建议程序 main 函数所在源文件应以描述程序基本用途的注释作为开头，而其他源文件则以文件名和描述该源文件基本功能的注释作为开头。

对于函数，则需要为其添加用于描述函数基本功能、参数类型、参数用途、参数可能取值，以及返回值含义（如果有）等内容的注释信息。对于每一个全局和静态变量，也应该为其添加相应的注释，来描述它们的基本用途与可能取值。除基本 C 语法代码外，诸如预处理指令 #else 与 #endif 也需跟随有相应的注释，以标注相应分支的位置和含义。

一般情况下，为保证国际化和通用性，应以英语形式撰写注释。若代码与注释在同一行，则注释应起始于该行代码最后一个字符后间隔两个空格的位置。注释应使用完整的句子，并在一般情况下保持首字母大写。最后需要注意的是，**若在注释中谈论到了变量的值，则应将该变量对应标识符完全大写**。比如在注释文本“the inode number NODE_NUM” 中提到的 NODE_NUM，便代表 C 代码中变量 node_num 对应的具体值。

语法约定

这部分规范对 C 语言编码时需要遵循的一些基本习惯作出了规定。这里举几个例子：

- 显式地为所有使用到的值标注类型，尤其是直接使用在表达式中的数字字面量值；
- 选择性地为编译器添加 “-Wall” 参数，并根据警告信息来优化代码；
- 选择性地看待静态代码分析工具给出的建议，在不影响程序正常功能的情况下，不要为了满足它们的要求而牺牲代码可读性；

- 外部函数和后续才会使用函数的声明，应该被放置在当前源文件开头处统一的地方，或选择放在单独的头文件中；
- 使用名字具有一定意义的不同变量来完成不同任务，且尽量使变量的声明处在恰好可以完成其“任务”的最小作用域中。除此之外，也不要声明会遮盖全局变量的局部变量。

在代码写法上，GNU 代码规范也作出了规定，比如：

- 不要将通过一个类型关键字声明的多个变量分散在多行；
- 可以将同类型的多个变量放在同一行声明，或选择在不同行使用独立的声明语句来声明多个变量；
- 当使用嵌套的流程控制语句（比如 `if` 语句）时，总是将内部的嵌套逻辑包裹在大括号中，以保证代码清晰可读；
- 尽量避免在 `if` 语句的条件判断处，也就是小括号内部做赋值操作（而在 `while` 语句中是可以的）。

命名

编码中的命名规范可以说是理解起来最简单，而行动上最复杂的部分。在 C 语言项目中，程序内使用到的全局变量和函数，它们的名称有时可以直接替代注释来说明它们的功能。因此，**需要确保对于它们的命名是具有一定意义的**。而相对地，局部变量由于仅作用在一个较短的上下文中，它的名字可以更短，只要能大概描述它的作用即可。

变量名一般采用下划线命名法，即通过下划线将其中不同的单词进行分割。变量名中可以使用缩写，但需要确保这些缩写是已经被人们熟知的，而不会产生任何歧义。并且，**尽量仅使用小写字母、数字和下划线来组成变量名，而把大写字母留给对宏常量以及枚举常量命名**。另外，当需要定义数字常量值时，请优先使用枚举常量而非宏常量，这将有利于调试器对程序执行逻辑的分析。比如下面这段代码：

复制代码

```
enum Settings {
  LIMITATION = 1000
};
const char* author_name = "Jason";
const int author_age = 28;
```

系统可移植性

GNU 规范中提到的“系统可移植性”主要是指将 C 应用移植到不同 Unix 版本系统上的能力。对于 GNU 程序来说，这种可移植性是可取的，但并非最重要的。

通常来说，你可以直接使用 GNU 旗下的 Autoconf 工具来部分实现这种能力。Autoconf 支持多种 Unix 系统，它可以根据当前所在宿主环境，自动生成进行程序编译所需的配置文件，比如 Makefile 与 configure 脚本文件。对于其他非 Unix 系统，则需要针对目标系统做特殊的兼容性开发。除此之外，在程序中应该尽量使用来自标准库（如 C 标准库、POSIX 标准接口等）中的函数，而非与特定系统强相关的底层能力。

最后，可以选择为 C 语言项目定义名为“_GNU_SOURCE” 的宏常量，当在 GNU 或 GNU/Linux 系统上编译定义有该宏常量的 C 项目时，预处理器会“启用”所有的 GNU 库扩展函数。这样，编译器在编译项目时，便会及时检查是否存在用户的自定义函数导致这些库函数被覆盖的问题。在保证程序系统兼容性方面，这是可以关注的一个点。

CPU 可移植性

不同 CPU 之间可能会存在着字节序、对齐要求等差异。因此，我们在编码时不应该假设一个 `int` 类型变量在内存中的起始地址与它的最低有效位（LSB）地址相同。同样地，**当使用 char 类型时，应显式指出它的符号性**。对于指针，应尽量避免出现将指针类型转换成数字值的过程，以保持程序最大的兼容性。

对于基本类型大小，GNU 标准不会处理 `int` 类型小于 32 位的情况。同样，也不会考虑 `long` 类型小于指针类型和 `size_t` 类型的情况。

系统函数

C 程序可以通过调用 C 标准库或 POSIX 库提供的函数，来使用相应的系统功能，但这些库函数在某些系统上可能存在着相应的可移植性问题。而借助 Gnulib，我们可以在一定程度上解决这个问题。

Gnulib 是 GNU 旗下的可移植性库，它为许多缺乏标准接口的系统提供了对标准接口的实现，其中包括对增强型 GNU 接口的可移植性实现等。Gnulib 可以直接与 GNU Autoconf 和 Automake 集成使用，来减少程序员编写可移植代码时的大部分负担。Gnulib 使得我们可以通过配置脚本，来自动确定缺少哪些功能，并使用 Gnulib 中的代码来“修复”缺少的部分。

国际化

GNU 规范也考虑到了 C 程序国际化的问题。同解决系统函数可移植性问题一样，GNU 也提供了名为 GNU gettext 的库，可以轻松地将程序中的消息翻译成各种语言。

字符集

在 GNU 源代码注释、文本文档，以及其他相关上下文中，请首选使用 ASCII 字符集中的字符。如果需要使用非 ASCII 字符，我们要确保文档的编码方式保持统一。通常来说，**UTF-8 应是首选的编码方式**。

总结

好了，讲到这里，今天的内容也就基本结束了。最后我来给你总结一下。

这一讲，我以 GNU 编码规范为例，介绍了在进行 C 项目编码时我们需要注意的几方面内容。从最基本的编码格式到可能影响程序正确运行的可移植性，这些内容涵盖了一个完整 C 工程项目在制定编码标准时需要考虑的绝大部分问题。

GNU 编码规范的出发点，是确保所有 GNU 旗下项目都能够采用统一的编码行为，以保证这些软件都能够在 GNU 操作系统上正确无误地运行。了解 GNU 规范的基本内容之后，你就能以此为基础，根据自身需求制定更加详细和个性化的 C 编码规范。

编码规范的制定，是项目从单人开发逐渐走向多人协作，让团队效率达到 1+1>2 效果的必经之路。因此，如何合理制定规范并加以落实并持续实践，是一个非常值得关注的问题。

思考题

最后想请你来聊一聊：你所在的团队是否制定了相应的编码规范？如果制定了，是怎样落实的？践行过程中又是否遇到哪些问题呢？欢迎在留言区告诉我你的想法。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 2　提建议

上一篇　19丨极致优化（下）：如何实现高性能的 C 程序？

下一篇　21丨生产加速：如何使用自动化测试确保 C 项目质量？

精选留言 (4)

ZR2021
2022-01-29

老师加油！！！

作者回复: 感谢支持！

  1

zxk

2022-04-03

个人主要是使用 Java 的，所在团队虽然有制定代码规范，但在实施过程中，由于时间精力问题，以及大量冗长的业务代码， Reviewer 往往没有时间精力去看的过于细致，导致代码质量逐步下降。虽然配置了一些代码检查工具，但提交者往往也不会去关注。

不知道老师的工作场景是如何落实的？有没有一些经验之谈。

作者回复: 我的建议是，这个问题可以从几个方面来考虑，首先如果配置了代码检查工具，那最好从代码审查流程上就严格把关。甚至可以将代码审查通过作为生产发布的强制性要求。其次，要明确代码审查的目的，有一部分代码问题是可以通过 linter 等工具来自动检查和修复的，而有关代码实现方式的部分，就需要代码审查者自己严格要求了。我们一般会通过不定期组织的团队代码 review 来和团队的其他成员一起过一下历次的 pull request，这样如果有问题也可以在这个过程中发现，同时也可以一定程度上约束 reviewer 的责任和态度。虽然代码审查不会纳入考核，但谁也不希望自己 review 的 PR 漏洞百出。最后，有关代码质量，通常来说定期的重构也是少不了。当然由于不算产出，怎样平衡 ROI 要具体看了。

 

dog_brother

2022-02-28

最近正在搞代码静态检查，还在摸索中

 

石天兰爱学习

2022-02-23

每天学一点，每天进步一点，奥力给

作者回复: 加油！