C++ Core Guidelines 解析

[德] 赖纳 • 格林(Rainer Grimm) 著

吴咏炜 何荣华 张云潮 杨文波 译

清華大學出版社

北 京

北京市版权局著作权合同登记号　图字：01-2023-1360

图书在版编目(CIP)数据

C++ Core Guidelines 解析 / (德) 赖纳 • 格林著；吴咏炜等译. —北京：清华大学出版社，2023.6
书名原文：C++ Core Guidelines Explained: Best Practices for Modern C++
ISBN 978-7-302-63577-2

I. ①C… II. ①赖… ②吴… III. ①C++语言－程序设计 IV. ①TP312.8

中国国家版本馆 CIP 数据核字(2023)第 090897 号

责任编辑：王　军　刘远菁
装帧设计：孔祥峰
责任校对：马遥遥
责任印制：宋　林

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：小森印刷霸州有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：25.25　　**字　　数**：590 千字
版　　次：2023 年 7 月第 1 版　　**印　　次**：2023 年 7 月第 1 次印刷
定　　价：128.00 元

产品编号：098603-01

推荐序一

很多朋友都知道，这些年在给许多 C++ 研发团队提供培训和咨询时，我总爱提及 C++ Core Guidelines 开源文档。而最近，我正研究如何让 ChatGPT 写出更好的 C++ 代码，每当 ChatGPT 生成令我不满意的代码风格时，我都会首先找到 C++ Core Guidelines 开源文档的相关规则，并将其用作 ChatGPT 的提示词，结果往往相当不错。我与 C++ Core Guidelines 的结缘发生在几年前，当时，我邀请 C++ 之父 Bjarne Stroustrup 到上海参加 C++ 及系统软件技术大会，在和 Bjarne 交流时，我了解到他领衔的这个开源项目。在 Bjarne 的极力推荐下，我认识到它的分量不轻，此后，C++ Core Guidelines 开源文档便成了我的案头必备书籍。

C++ 语言之博大精深在所有编程语言中是出了名的。C++ 不仅支持面向过程、面向对象、泛型编程、函数式编程等多种编程范式，还融合了古典 C 语言、前现代 C++（C++03 之前的标准）以及现代 C++（C++11~23）的多种风格；在追求极致性能的同时，提供各种抽象逻辑来管理大规模软件的复杂性。凡此种种，使得在现代 C++ 中，同一个编程任务常常有 5～10 种实现选择，而每一种选择都有适合它的特定上下文。这就是 C++ 工程师经常面对的“心智负担”。如何消除这种“心智负担”？如何比较自如地选择最优的编程方式，同时确保团队有一致的编程规范？这不仅是 C++ 工程师个人的事情，也是一个 C++ 团队必须认真对待的课题。

在 C++ 技术社区，C++ 编程规范的相关研究与总结一直很活跃。历史上，*Effective C++*、*Exceptional C++* 等多部经典著作都诞生自 C++ 社区。而由 Bjarne Stroustrup 和 Herb Sutter 领衔编写的 C++ Core Guidelines 开源文档是当仁不让的集大成者，它在总体上汇聚了 C++社区多年来积累的宝贵经验，是一份难能可贵的资料。只可惜由于它的篇幅宏大，很多人很难完整通读，而容易将其束之高阁。

现在，由资深技术专家 Rainer Grimm 撰著的《C++ Core Guidelines 解析》，从内容上说，选取了现代 C++ 语言最核心的相关规则；从篇幅上说，对软件工程师非常友好。以“八二原则”看，这个精编解析版是一个非常聪明的选择。同时，Rainer Grimm 并没有简单照搬开源文档中的规则，而是结合自己丰富的咨询和培训经验，给出了非常翔实的解析，这自然为本书增色不少。最后，此书中文译本的质量让我非常放心。翻译团队非常强大，领衔的吴咏炜在 C++ 领域的功力自不必说，而且他在技术文本上字斟句酌的

认真劲可是出了名的，杨文波、张云潮和何荣华在 C++ 领域也都非常资深。非常开心 C++ 中文社区的好书越来越多，我相信《C++ Core Guidelines 解析》会给各位 C++ 工程师以及企业 C++ 研发团队带来长久的价值。

李建忠
Boolan 首席软件专家，C++ 及系统软件技术大会主席

推 荐 序 二

本书通过很多实例对 C++ Core Guidelines 进行了解说和阐释，书的译者之一杨文波（杨文波和我在庐山相识，是多年好友）邀请我写篇推荐序，盛情难却，我便也举个例子吧。

大约两年前的一个上午，我在北京大兴机场等航班。偌大的机场里人很少，我拉着行李箱在机场中闲逛。逛了一阵后，看时间差不多了，便准备先去洗手间，然后到登机口。当我走进洗手间时，电话铃声响了。通过来电提示，我看出这个电话来自我的客户，便立即接听了。

电话接通后，我感觉电话那端正在解决一个技术问题，有两个人——一位是开发工程师，另一位是技术带头人。

“张老师，我们遇到一个很妖的问题……”

这几年，我已经习惯于听到各种“很妖”的问题，于是平静地让对方描述一下细节。

“一个指针在函数内部是对的，但是返回到父函数里就不对了。”

听到这个描述，我大体猜到对方的问题了。但是为了避免猜错，我问道：“你们的程序是编译成 64 位的吧？”

得到的回答是：“对。”

得到这个确认后，我比较有把握了，于是给出了诊断意见和解决方案：

“很可能是因为调用这个函数的地方缺少原型声明，查一下是不是忘记包含头文件了。”

“哦，我们马上查一下。”

对方没有提出挂断电话，我也想听听结果，于是也没提出挂断电话。好在结果在 1 分钟之内就出来了。

“张老师，加了头文件之后就好了……”

就这样，我在北京大兴机场的某个洗手间里，帮客户解决了一个棘手问题。从接电话到问题解决，一共花了 2～3 分钟。

对于这个问题，有些读者可能已经看懂了，有些读者可能还有些糊涂。因为大家没有看到代码，我的描述也缺少一些信息。

就在前两天，这个“很妖”的问题也在格蠹现身了。小伙伴们急着发布一个软件，但是测试时，发现一个动态库存在随机性的问题，C++ 的主程序使用这个 DLL 时没问题，但是当它集成到较为复杂的前端程序中，被 Node.js 代码调用时，就有问题。

小伙伴们用经典的排除法试了一阵子，但没有效果。我看了一下他们的描述，以及

发给我的调用栈，感到确实有点复杂，需要静下心来慢慢调试。而且天色已晚，已到了下班时间，疲劳作战意义不大，我便让他们下班休息。

吃过晚饭，我自己挂上调试器，将问题复现出来。

是习以为常的访问违例，也称野指针，或者“段错误”。

但不寻常的是，这个指针并不为 0，也不是太小，看起来似乎还挺正常。

寻找这个指针的源头，发现它来自一条距离不远的 malloc 语句：

```
dap->bdata = malloc(sizeof(struct nkm_usb_backend_data));
if (dap->bdata == NULL) {
  LOG_ERROR("unable to allocate memory");
  return ERROR_FAIL;
}
```

这个 malloc 语句不应该失败啊。记得前些天，本书的另一位译者吴咏炜还特意做了一个测试 malloc 到底能分多少内存的小程序，并分享给我做试验。试验的结论是，malloc 可以分很多很多，与包括环境在内的多种因素有关，这里不展开了。

而且，如果 malloc 真的失败，那么 malloc 下面的检查语句应该起作用啊，不至于跑到再下面，出现野指针的问题。

顺着这个路线思考，我想到了大兴机场，意识到两年前帮客户解决的问题如今发生在自己身上。

对的，是缺函数原型声明，少头文件。

按 Ctrl + Home 到这个文件开头，扫视一圈，发现里头果然没有应该有的 malloc.h，于是按回车，增加一行：

```
#include <malloc.h>
```

再编译运行，“妖怪”不见了。

需要特别说明的是，发生问题的源代码是众多 C++ 源文件组成的一个较大项目中少有的几个.c 之一。如果将其换为.cpp，那么编译器会把这样的问题视为错误，无法构建成功。而如果是.c 后缀，那就应该符合 C 语言的规范，编译器会发出警告。

如果使用的是 GCC 编译器，那么报的警告信息是著名的“隐式函数声明警告”，大致如下：

```
warning: implicit declaration of function ‘malloc’;
```

并对赋值操作给出这样一个警告：

```
warning: initialization makes pointer from integer without a cast
```

如果使用的是微软的编译器，那么报的警告信息为：

```
D:\Work\nkm\apps\kiloeye\nkm\usb_hid.c(76,30): warning C4013
```

“malloc”未定义；假设外部返回 int。

如果仔细阅读两种编译器的警告信息，可以发现其中都包含了一个信息：对于这样

的隐式函数，编译器总是假设其返回 int 整数。int 整数是 32 位的，所以对于 malloc 这种返回指针的函数，在 64 位的情况下，就会出现 64 位指针被截去高 32 位，只剩低 32 位的情形，这足以让很多程序员困惑，不知这个看似正常的指针为啥不能访问。

这个例子说明好的编程习惯很重要，也说明要重视编译器的警告信息。但是，此话说着容易，实际执行却并不容易；在小项目中容易，在一些没有严格管控警告信息的大项目中就变得更困难。

治学难，治代码亦不易。感谢 C++ 之父在古稀之年仍笔耕不辍，亲自编辑 C++ Core Guidelines，字斟句酌，撰写每一则规范，指导 C++ 程序员养成良好的编程习惯。感谢这本书的原作者 Rainer Grimm，结合自身经验，用心阐释 C++ Core Guidelines。感谢本书的译者将这样的作品翻译为中文。序已经不短，词不达意，唯望能为读者增些许读书之乐。

张银奎
《软件调试》和《格蠹汇编》的作者

推荐序三

从很多维度看，C++ 语言都是程序设计语言中独树一帜的存在。其中最有趣的一点是，它的版本更新既非由语言发明者一人独断，又非以完全开放的方式由任何人提交更新并以复审加小步快跑的方式一点一滴地迭代。C++ 语言每三年发布一个包含相当数量、有时也相当激进的特性变更集的大版本，并且由一个标准委员会（包含 C++ 之父 Bjarne Stroustrup）决定每个大版本包含什么以及不包含什么。这种特别的语言发展模式对采用 C++ 语言的程序员们有着重大的影响：每当 C++ 语言发布新版本时，除了一些和标准委员会走得比较近的人外，大部分的 C++ 程序员往往都要接受一次带有断层的观念冲击。而 C++ 语言的多范型完全融合的特点，会让程序员在面对新特性或对旧特性的改变时，需要更多时间来了解某个看似人畜无害的细微变化在真正运用时会造成的旧代码到处炸裂的情形。C++ 语言是既灵活又重磅的全能武器，但对其使用者的要求也更高一筹。

所以，第一次听说 C++ Core Guidelines 这个项目（应该是某次和 Bjarne 本人还有标准委员会的一些前辈们在李建忠先生组织的一个饭局上）时我就意识到，这应该是个非同小可的神器。标准委员会的成员们终于脱下了长衫，开始亲近在一线苦干的程序员。然后，我就开始约稿，但是在我的稿还没约到时，一本新书横空出世，也就是这本《C++ Core Guidelines 解析》。看完以后，我觉得大概这本就可以了吧。Rainer 本人我并未见过，他可能至少在写这本书的时候，还并非 C++ 标准委员会成员。但高手在民间，这本书得到了 Bjarne 本人和标准委员会主席 Herb Sutter 等人的高度认可，就说明了问题。当然，吴咏炜和何荣华等中文译者做的工作，让这本书添色不少。翻译是再创作，技术翻译更是见功力的事情，既要在宏观和微观层面熟悉技术，又要完美传达作品原意，当然是一等一的工作。每个自觉技术不错的同行，都应该在人生的某个阶段挑战一本好作品的翻译工作。

话说回来，所有学习和使用 C++ 的人都必须读一读本书。现在完全可以说，学习 C++而不了解 C++ Core Guidelines 相当于盲人摸象。C++ 诞生于 20 世纪 70 年代，在当时它不可能了解现代计算系统的诸多特性，而现代 C++ 已经在语言层面上支持了大量的此类特性。但与此同时，C++ 语言仍然使用和早期版本并无太多不同的语法体系，只是这些语法中已悄然注入了崭新的语义。更重要的是，C++ 的设计哲学始终未曾动摇，而只有理解了这些哲学，在面对具体的新语法和新特性时，才能较快地准确理解其背后的设计意图并全面掌握它们。读一读 C++ Core Guidelines 的“P”字头部分吧，即使你在日常工作中并非以使用 C++ 语言为主，这些规则也会让你深刻地感受到这门语言的美不胜收。如果你可以举一反三，再回头看看你日常使用的语言，也一定会深受启发。我知道，

现在 C++ Core Guidelines 还没有中文版，也知道，标准委员会发表的原文可能有些晦涩难懂。但是，现在你没有借口了，《C++ Core Guidelines 解析》中文译本就在你面前，请打开它，今天就开始阅读！

高博

卷积传媒 CEO，*Effective Modern C++*的译者

推荐序四

C++ 历史悠久，从 C++11 发布之日到现在，它已经迭代了 4 个版本，新特性非常多，而 C++又是非常灵活的，达成同一目的的写法往往有多种。曾经见过一个搞笑的动图，里面描述了 C++ 的几十种初始化方式，这种夸张的灵活性往往会让 C++ 新手无所适从，到底应该选择哪种初始化方式呢？C++ Core Guidelines 给出了答案：总是通过花括号初始化，因为它更简单，语义更清晰，也更安全。

由此可见，C++ 开发者需要这些实用的规则来指导其开发工作，这也是 C++ Core Guidelines 诞生的背景。

C++ Core Guidelines 并非只适合 C++ 新手，对于一些有经验的 C++ 开发者来说，也很有参考价值。例如：实现一个重载语义的函数的时候，用默认参数好还是写一些重载函数好？用 char 还是 std::byte？用 char 数组还是 std::array？我相信很多人在写 C++ 代码的过程中经常会遇到这些选择上的疑惑，而 C++ Core Guidelines 可以给你很好的指导，告诉你怎么选择更好。

另外，C++ Core Guidelines 中有大量和 C 有关的建议：优先采用 STL 的 array 或 vector 而不是 C 数组；使用 std::string 而不是 C 字符串；不要使用 va_arg 参数；如果从 C 转到 C++ 开发的话，非常有必要看一下。C++ Core Guidelines 比较好的一点是，它并不是教条地说要这样或者不要那样，而是会告诉你为什么这样更好，例如为什么用 std::array 而不是 C 数组，因为 std::array 兼具了效率和安全性，访问越界的时候会抛异常，比 C 数组更安全。

C++ Core Guidelines 中有大量的建议是关于代码安全性的，例如使用条件变量的时候可能会发生虚假唤醒，应该使用谓词去 wait；全局的 lambda 不要按引用去捕获，以避免生命周期的问题；等等。这些对于实际的开发来说非常有指导价值。

当然，这些指导原则是有限定场景的，正如其作者所说：这些指导原则是有适用场景的，不要盲目地照搬。比如，有一条规则说不要使用 std::thread detach 方法，这就是一个典型的例子，它并不是说不能使用 detach，而是说线程函数如果按引用捕获变量，detach 之后变量生命周期就会出问题，那么，没有捕获引用的时候当然可以使用 detach。还有一条规则说用 std::string_view 引用字符序列，但这里并没有详谈字符串生命周期的问题，std::string_view 使用的时候需要关注这个问题。

总之，C++ Core Guidelines 是很多经验丰富的 C++ 开发者的经验总结，这些宝贵的经验确实值得 C++ 开发者（无论是否有经验）去参考、借鉴。深入理解这些经验背后的细节和原理，必能使 C++ 基本功变得更加扎实；将这些指导原则应用于实际开发，无疑

能提高代码的质量和安全性。另外，四位译者的翻译也是认真、细致的，准确翻译了英文的指导原则，我在这里力荐此书。

祁宇
purecpp 社区创始人，《深入应用 C++11》的作者

推荐序五

众所周知，C++ 是一门自由的语言，语言的设计哲学之一就是赋予程序员极大的自由度和灵活性，因此，使用 C++ 完成一个任务时，不同的程序员往往会有不同的实现方法，这真正阐释了什么叫条条大路通罗马。不过，这种自由和灵活的代价就是语言复杂度的提升，学习曲线也必然不会平滑。此外，C++ 语言特性也十分丰富，尤其从 C++11 标准开始，新的特性层出不穷，以至于曾有书友用《三国演义》中评价“八阵图”的一句话来评价现代 C++：“变化无穷，不能学也。”当然，这句话多少有些调侃和夸张的成分，但也说明了 C++的自由和灵活性多多少少地给学习者和从业者带来了一些困扰。从软件工程的角度出发，程序员也大都希望 C++ 在保持自由的同时能够有一些规则和惯用法，于是 C++ Core Guidelines 应运而生。

C++ Core Guidelines 非常全面地介绍了语言各个重要环节的规则和惯用法，如函数、类、错误处理、性能优化等。例如，我们常说尽量使用智能指针代替裸指针，从而避免内存的泄漏，但是这又造成另外一个问题——智能指针的过度使用。C++ Core Guidelines 中有这么一条建议——“对于一般用途，采用 T* 或 T& 参数而不是智能指针”，因为智能指针关注所有权和生命周期，在不操作生命周期的函数中应该使用原始指针或引用。

除了规则和惯用法，C++ Core Guidelines 还会解释一些容易引起争论的话题。比如错误处理中的异常，Guidelines 中非常客观地描述了异常的应用场景，例如只有在能精确预测从 throw 恢复的最长时间时，异常才适用于硬实时系统。此外，Guidelines 客观地提出，如果负担不起或者不喜欢基于异常的错误处理方式，那么也可以采取其他方案，但必须做充分的测试和测量，因为无论哪种错误处理方式都有其不同的复杂性和问题。

综上所述，C++ Core Guidelines 是一份非常有学习价值的指南，但遗憾的是，它的学习难度并不低。Guidelines 中有很多问题都只用一句话和一份简短的代码一笔带过，对于 C++新手来说不太友好。另外一个难点是，整个 Guidelines 是用英文编写的，这无疑让我们的学习难度又提高了一些。因此，当我看到《C++ Core Guidelines 解析》的中文版书稿时非常兴奋，因为终于有一本中文书籍将 C++ 规范精华的详细解析呈现在读者面前了。

最后，我诚心建议每一位从事 C++ 开发的朋友，在自己的办公桌上摆放这样一本指南，在遇到疑问和困惑的时候翻阅一下，很容易就能找到心中想要的答案。

谢丙堃

《现代 C++ 语言核心特性解析》的作者

推荐序六

不少人在学习 C++的时候都有一连串的疑问：为什么语法设计成这样？为什么 C++学起来如此复杂？为什么总感觉这门语言是个半成品？为什么 C++不能跟 C#、Python 和 TypeScript 一样开箱即用？在我看来，这跟 C++的定位有密切的关系。C++在设计之初有一种信念——对范式的偏好应该交给程序员，因此，凡是能写成库的概念，都尽量不要写进语法里。C++标准库正是作为语法的补充而建立起来的。C++20 引入的 coroutine 就是一个非常有 C++特色的例子。从业务需求的角度来看，coroutine 作为一个功能来讲是不完整的，它更像一套运算规则，不像可以开箱即用的 C#一上来就给你 IEnumerable<T>和 Task<T>。然而这种设计为库的作者提供了巨大的灵活度，你可以自由地把任何相关的类型都包装成 coroutine。库的使用者可以有充分的选择，甚至可以“自己来”，不必为了使用 coroutine 而依赖某个特定的多线程或者 IO 库。

如果我们孤立地看 C++的语法，可能会对它的存在感到困惑。比如 C++11 引入的右值引用、forward/move、完美转发以及它们与模板相关的一些类型的运算规则，甚至包括 lambda 表达式传参的一些手法，其实是一个整体，互相之间缺一不可。但是，如果单独地看右值引用，可能会对这个设计感到困惑。初学者有这种感觉是很正常的，然而抱着这些疑问去翻阅各种语法材料，哪怕材料说得很详细，可是如果对语法设计的动机缺乏了解，不能以全局的眼光来看待这些规定，就会觉得难以接受，甚至还会认为“右值引用类型的参数/变量其实是一个左值引用”这样的规定很费解。

因此在遇到困难的时候阅读一些书籍（比如本书），是十分有帮助的。本书并不照本宣科，而是会从各个角度告诉你，为什么要这么做，为什么不那么做，以及建议用什么手法做一件事，等等。在了解这些观点之后再审阅自己写过的 C++代码，或者在新的项目中实践新的最佳实践，不仅可以得到高质量的代码，而且能提高自己的技术。

工欲善其事，必先利其器，掌握扎实的基本功对一个优秀的程序员来说是很重要的。书中有一些内容并不局限于 C++，比如对性能和并发的阐述，都是一些通用的知识点。哪怕你以后不用 C++，这些知识在其他语言中依然会发挥重要的作用。这个领域往往由一些非常细致而且难度很高的知识构成，本书这方面的知识可以起到抛砖引玉的作用，程序员了解了相关基础知识之后，再去深入阅读其他材料，将事半功倍。

陈梓瀚 Vczh（轮子哥）

译 者 序

C++ 是一门博大精深的语言，其发展、演化历程也堪称波澜壮阔。它易学难用的特质劝退了不少人（入门即放弃）。从学会使用 C++ 到用好 C++ ，需要经过多年持续不断的学习和实践。自 C++11 以来，标准委员会每三年一更新，如今的“modern C++”相较于之前的 C++98 来说变化相当大。C++ 在成长和变化，因此，C++ 程序员需要跟紧脚步，不断学习，努力提高，这样才能在充满竞争和变化的世界中与 C++ 一起茁壮成长。

C++ Core Guidelines 与 C++ 语言本身一样，是由 Bjarne Stroustrup 领导的协作项目。该指南是许多组织耗费了大量时间共同探讨和设计的成果，旨在帮助人们有效地使用现代 C++。虽然其学习始于语法，但倘若拘泥于 C++ 语言自身，仅仅掌握语法和运用特性，一直纠结于字有几种写法，恐怕只能困于井底的囹圄，难以登高望远，胜任更高阶的架构和设计任务。而 C++Core Guidelines 聚焦于一些相对高层次的问题，例如接口、资源管理、内存管理以及并发等，可以帮助我们提升思想高度，学习业界行之有效的架构设计理念，避免浮沙筑高塔。

不过，C++ Core Guidelines 是按参考书的方式来组织的。它不是教程，不方便读者通过从头到尾的阅读方式来学习如何用好现代 C++。因此，Rainer Grimm 写了这本书，希望把 C++ Core Guidelines“编成可读的、可供消遣的故事，去除其中佶屈聱牙之处，必要时填补缺失的内容”。 Rainer 本人是德国的一位著名的 C++ 培训师兼咨询师。他不仅在本书中系统地描述了 C++ Core Guidelines，还在其中加入了很多个人的心得、示例和额外内容，大大提高了 C++ Core Guidelines 的可读性。比如，在第 10 章中，他介绍了 CppMem，在第 13 章中，他介绍了模板元编程的概念和由来，这些都是 C++ Core Guidelines 本身所没有的。因为这样，当接到本书的翻译任务时，我们几个译者也都非常高兴。对我们而言，这次的翻译也是一种很好的学习。

本书的翻译最令我们感到满意的地方是，经过与作者的多轮沟通，我们修正了英文原著中的许多问题——有些是英文版出版社的责任，有些是作者的疏忽，也有些是因为 C++ Core Guidelines 条款本身发生了重大变化。因此，除非我们在翻译中犯了更多的错误，否则我们可以高兴地说，这本中文译本不仅更方便中文读者，还比原版书更加“准确”。当然，中文图书在价格上的优势就更不用说了。

相比于在线的 C++ Core Guidelines，本书不可避免地发生了滞后，因为本书的英文原著从写作到出版已经有了很大的延迟，此外，还有翻译的延迟，以及中文书出版的延迟。幸好，在大部分情况下，条款本身不会发生质的变化，对于极小部分在翻译时已经

出现了重大变化的内容，我们也做了一点更新。毋庸置疑，这些变化不会影响本书的学习价值。

最后，我们提几点关于 C++ Core Guidelines 的建议：

- 按本书附录 A 的线索，在工作和学习环境中启用相关的自动检查，马上开始在代码设计和审核流程中使用。
- 花一两天的时间快速遍历 C++ Core Guidelines 的关键条目。该指南会一直动态更新，而本书提供了一个相对稳定的切片，更方便读者快速把握基本要点。
- 当需要细究具体条目或某方面的规则时，比如在代码审核中对同一条目有理解不一致的地方时，仍要参考 C++ Core Guidelines 的英文原文。
- C++ Core Guidelines 和本书并没有深入探讨条目背后的技术史和设计哲学。读者如果希望在这些方面进一步探究，可以参阅另一本“姊妹作品”——*Beautiful C++: 30 Core Guidelines for Writing Clean, Safe, and Fast Code* 及其作者 Kate Gregory 女士在 CppCon 2017 的演讲，也可参考 C++ Core Guidelines 列出的参考文献。
- C++ Core Guidelines 反映了 C++ 社区的公约数，但未必完美符合具体 C++ 团队在业务背景、技术选型和设计风格上的共识。读者在工程实践中，可以根据自己团队及项目的情况扩展或者改写某些条目，从而生成更适合自己和团队的技术指南。

本书的翻译是译者们的第二次合作（第一次是翻译 Bjarne 的 HOPL4 论文《在拥挤和变化的世界中茁壮成长：C++ 2006–2020》）。清华大学出版社给了我们这个机会，大家都非常高兴，也感谢编辑们的细心检查，正是因为各位的共同努力，本书才得以出版。同时，我们要感谢家人在各方面的支持和理解。当然，还有正在阅读本书的你，有了你的支持，本书才能真正发挥其价值。

由于时间仓促和能力所限，书中不免会有所错漏，如有任何意见和建议，请不吝指正。

译者

前　言

本前言只有一个目的：给你——亲爱的读者，提供必要的背景，以便你从本书中获得最大的收获。这包括我的技术细节、写作风格、写这本书的动机以及写这样一本书的挑战。

惯例

我保证，只有几个惯例。

规则还是指导原则

C++ Core Guidelines 的作者经常把这些指导原则称为规则。我也一样。在本书中，我使用的这两个术语可以互换。

特殊字体

粗体　有时我用粗体字强调重要的术语。

`Monospace`　代码、指令、关键词、类型、变量、函数和类的名称都用等宽字体显示。

方框

每一章的结尾处基本都有方框，里面用点列表进行总结。

相关规则

一个规则常常会与其他规则相关。如果有必要，我会在一章的末尾提供这些有价值的信息。

本章精华

重要
在每一章的结尾处获得基本信息。

源代码

我不喜欢 `using` 指令和声明，因为它们隐藏了库函数的来源。但由于页面的空间有限，有时我还得用一下它们。我使用它们时，总是可以从 `using` 指令（`using namespace std;`）或 `using` 声明（`using std::cout;`）中推断出来源。并非所有头文件都会在代码片段中标出来。布尔值会显示为 `true` 或 `false`，产生此输出所必需的输入/输出操作符 `std::boolalpha` 大多不放在代码片段中。

代码片段中的 3 个点（`...`）代表没写出的代码。

当我把完整的程序作为代码实例介绍时，你会在代码的第一行找到源文件的名称。假设你使用的是 C++14 编译器。如果这个例子需要 C++17 或 C++20 的支持，我会在文件名后面提到所需的 C++ 标准。

我经常在源文件中使用“`// (1)`”之类的标记，以便后续解释。如果可能的话，我把标记写在引用的那一行；如果不行，就写在前面一行。这些标记不是本书中一百多个源文件的一部分（源文件可通过扫描本书封底二维码获取）。由于排版上的原因，我经常会对本书中的源代码进行调整。

当我使用 C++ Core Guidelines 中的例子时，经常为了提高可读性而进行重写：如果缺少 `namespace std`，我会加上；我也会统一格式。

为什么需要指导原则

下面是些主观的结论，主要基于我超过 15 年的 C++、Python 和一般软件开发的培训师经验。在过去几年里，我负责有关除颤器的软件研发以及团队管理。我的职责包括我们设备的合规事务。为除颤器编写软件的任务极具挑战性，因为它们可能给病人和操作者带来死亡或严重伤害。

我心中有一个问题，它也是我们 C++ 社区需要回答的问题：为什么我们需要现代 C++ 的指导原则？下面是我的想法。为了简单起见，我的想法包括三个方面。

对新手来说很复杂

尤其对于初学者来说，C++ 是一种天然复杂的语言。这主要是因为我们要解决的问题本来就很棘手，而且往往很复杂。当你讲授 C++ 时，你应该提供一套规则，它们在至少 95% 的用例中对你的学员有效。我想到的规则包括：

- 让编译器推断你的类型。
- 用花括号初始化。
- 优先选择任务而不是线程。
- 使用智能指针而不是原始指针。

我在培训班上讲授诸如上面的规则。我们需要一个关于 C++ 的上佳实践或规则的全集。这些规则应该是正面表述的，而不是否定式的。它们得声明你应该如何写代码，而

不是应该避免什么问题。

对专业人士来说很困难

我并不担心每三年一次的新 C++ 标准所带来的大量新功能。我担心的是现代 C++ 支持的新思想。想想使用协程的事件驱动编程、惰性求值、无限数据流或用范围库进行函数组合。想想概念，它为模板参数引入了语义类别。向 C 程序员传授面向对象的思想的过程可能会充满挑战。因此，当你转向这些新的范式时，必须重新思考，你解决编程难题的方式也多半会改变。我想，过多的新思想尤其会让专业的程序员感到不知所措，他们习惯于用传统技术解决问题。他们很可能会落入“手里拿着锤子，所有问题都是钉子”这样的陷阱。

用在安全关键型软件中

最后，我有个强烈的担忧。在安全关键型软件的开发中，你经常必须遵守一些规则。最突出的是 MISRA C++。目前的 MISRA C++: 2008 指导原则是由汽车工业软件可靠性协会（MISRA）发布的。它们基于 1998 年的 MISRA C 指导原则，最初为汽车行业而设计，后来在航空、军事和医疗领域成为实施安全关键软件的事实标准。与 MISRA C 一样，MISRA C++ 描述了 C++ 的一个安全子集的指导原则。但是这里有个概念问题。MISRA C++ 并不是现代 C++ 软件开发的最先进技术，它落后了 4 个标准！举个例子：MISRA C++ 不允许运算符重载。我在培训班上讲，你应该使用用户定义字面量来实现类型安全的算术：`auto constexpr dist = 4 * 5_m + 10_cm - 3_dm`。为了实现这种类型安全的算术，你必须对算术运算符和后缀字面量运算符进行重载。说实话，我不相信 MISRA C++ 会与当前的 C++ 标准同步发展。只有社区驱动的指导原则，如 C++ Core Guidelines，才能面对这一挑战。

MISRA C++集成了 AUTOSAR C++14

不过，仍有希望。MISRA C++ 集成了 AUTOSAR C++14。AUTOSAR C++14 基于 C++14，应该会成为 MISRA C++ 标准的扩展。我非常怀疑由组织驱动的规则是否能够与现代 C++ 的动态发展保持同步。

我的挑战

回顾一下我在 2019 年 5 月与 Bjarne Stroustrup 和 Herb Sutter 讨论的电子邮件的基本内容，邮件里我告诉他们，我想写一本关于 C++ Core Guidelines 的书：“我是 C++ Core Guidelines 的绝对支持者，因为我坚信我们需要现代 C++ 的正确/安全的使用规则。我经常在我的 C++ 课程中使用 C++ Core Guidelines 中的例子或想法。Guidelines 的格式让我想起了 MISRA C++ 或 AUTOSAR C++14 的规则。这可能是有意为之，但对于广大受众来说，它并不是理想的格式。我认为，如果我们用第二份文件描述 Guidelines 的总

体思路，将会有更多的人阅读和讨论 Guidelines。”

我想对之前的这些对话补充一些说明。在过去的几年里，我在我的德语和英语博客上写了一百多篇关于 C++ Core Guidelines 的文章。此外，我还为德国的 *Linux-Magazin* 杂志写了一系列关于 C++ Core Guidelines 的文章。我这样做的原因有两个：首先，C++ Core Guidelines 应该被更多人所熟知；其次，我想以一种可读的形式介绍它们，如果有必要的话，提供更多的背景信息。

这是我的挑战：C++ Core Guidelines 由五百多条指导原则组成，很多时候直接称为规则。这些规则是在考虑静态分析的情况下设计的。许多规则对于专业的 C++ 软件开发者来说可以救命，但也有许多相当特殊的规则，往往不完整或多余，有时规则之间甚至相互矛盾。我的挑战是将这些有价值的规则编成可读的、可供消遣的故事，去除其中佶屈聱牙之处，必要时填补缺失的内容。说到底，这本书应该包含专业的 C++ 软件开发者必须遵守的规则。

万物流动，无物永驻

古希腊哲学家赫拉克利特有言：“万物流动，无物永驻。”这也代表了我在写这本书时面临的挑战。C++ Core Guidelines 是一个由 GitHub 托管的项目，有超过 200 个贡献者[1]。在我写这本书的时候，我所依据的原始条款可能已经发生改变了（见图 a）。

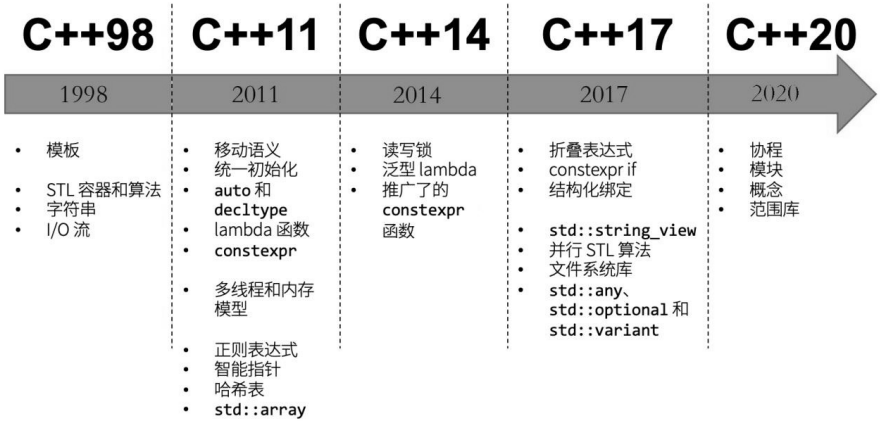

图 a　C++各版本新特性

Guidelines 已经包含了 C++ 的特性，这些特性可能会成为即将到来的标准的一部分，例如 C++23 中的契约[2]。为了反映这一挑战，我做了几个决定：

- 我将重点放在 C++17 标准上。在合适的场合，我会描述针对 C++20 标准的规则，如概念。

1 译者注：到 2022 年年底，贡献者已经超过了 300 人。

2 译者注：遗憾的是，契约这一特性没能进入 C++23。

- C++ Core Guidelines 在不断演进，特别是随着新 C++ 标准的发布而演进。本书也将如此。我计划对这本书进行相应的更新。

如何阅读本书

本书的结构代表了 C++ Core Guidelines 的结构。它有相应的主要章节和部分辅助章节。除了 C++ Core Guidelines 外，我还添加了附录，这些附录对缺失的主题进行了简明扼要的概述，包括 C++20 乃至 C++23 的特性。

至此，我仍然没有回答如何阅读本书的问题。当然，你应该从主要章节开始，最好从头到尾阅读。辅助章节提供了额外的信息，并特别介绍了 Guidelines 支持库。可将附录当作参考来获得所需的背景信息，以便理解主要章节。没有这些额外的信息，本书就不完整。

致　谢

首先，我必须感谢 C++ Core Guidelines 的所有贡献者。它是大约 250 位贡献者的工作成果；迄今为止，最多产的是 Herb Sutter、Bjarne Stroustrup、Gabriel Dos Reis、Sergey Zubkov、Jonathan Wakely 和 Neil MacIntosh（Guidelines 支持库）。

其次，我想好好感谢我的校对人员。没有他们的帮助，这本书就不会有现在的质量。此处列出他们的姓名，按字母顺序排列：Yaser Afshar、Nicola Bombace、Sylvain Dupont、Fabio Fracassi、Juliette Grimm、Michael Möllney、Mateusz Nowak、Arthur O' Dwyer 和 Moritz Strübe。

最后，非常感谢我的妻子 Beatrix Jaud-Grimm 为本书绘制插图。

目 录

第 1 章

简　　介

Cippi 初识字

在接下来的章节深入探讨 C++ Core Guidelines 的细节之前，此处先提供一个简短的介绍。

1.1 目标读者群

C++ Core Guidelines 的目标读者群是所有 C++ 程序员，包括可能考虑使用 C 语言的程序员。

1.2 目的

C++ Core Guidelines 的规则倡导现代 C++，旨在实现更统一的风格。当然，并不是所有规则都适用于老代码。这意味着，应该将这些规则应用于新的代码，但也应该将其应用于已经不工作或需要重构的老代码。重点在于类型安全和资源安全。这些规则不仅仅是说“不要那样做”；它们是规定性的，经常也是可检查的。规则的设计允许逐步采纳。

1.3 非目的

现在我们知道规则的目的是什么，“非目的”也很有趣。Guidelines 并不要求连续阅读，也不会替代教材。此外，没有提供将旧的 C++ 转换为现代 C++ 的现成方法，没有精确到允许你盲目遵循，也并非 C++ 的一个安全子集。

1.4 施行

如果没有施行（enforcement），这些 Guidelines 在大型代码库中是无法管理的。出于这个原因，每条规则都有一个施行部分。施行可以是代码审查、动态或静态代码分析。相关的规则被归类到规格配置（profile）里。C++ Core Guidelines 定义了防止类型违规、边界违规和生存期违规的规格配置。

1.5 结构

这些规则遵循一个典型的结构。

- 原因：规则的理由
- 例子：代码片段，显示有关该规则的好或坏的代码
- 替代方案：“不要这样做”规则的替代方案
- 例外：不使用该规则的原因
- 施行：如何检查该规则
- 参见：对其他规则的引用
- 注解：对一条规则的附加说明
- 讨论：对其他理由或例子的引用

1.6 主要部分

C++ Core Guidelines 由 16 个主要部分组成。下面列出它们，给大家一个概览。

- 简介
- 理念
- 接口
- 函数
- 类和类的层次结构
- 枚举
- 资源管理
- 表达式和语句

- 性能
- 并发性
- 错误处理
- 常量和不变性
- 模板和泛型编程
- C 风格编程
- 源文件
- 标准库

本章精华

重要

- 目标读者是所有 C++ 程序员。
- C++ Core Guidelines 的目的是采用更现代的 C++，实现一种普遍的风格。
- 这些规则不是教程，也没精确到允许盲目遵循。
- 每条规则都有一个施行部分。

第 2 章

理　　念

Cippi 在沉思

理念性规则强调一般性，因此，无法进行检查。不过，理念性规则为下面的具体规则提供了理论依据。由于一共只有 13 条理念性规则，本章将全面涵盖它们。

P.1　在代码中直接表达思想

程序员应该直接用代码表达他们的思想，因为代码可以被编译器和工具检查。下面的两个类方法使这一规则显而易见。

```
class Date {
    // ...
public:
    Month month() const;  // 这样做
    int month();          // 不要这样
    // ...
};
```

第二个成员函数 month() 既没有表示它不修改日期对象，也没有表示它返回一个月份。相对于标准模板库（STL）的算法，使用 for 或 while 等方式的手工循环通常也有类似的问题。接下来的代码片段说明了我的观点。

```
int index = -1;                          // 不好
for (int i = 0; i < v.size(); ++i) {
    if (v[i] == val) {
        index = i;
        break;
    }
}

auto it = std::find(begin(v), end(v), val); // 更好
```

一个专业的 C++ 开发者应该了解 **STL 算法**。使用它们的话，你就可以避免显式使用循环，你的代码也会变得更容易理解、更容易维护，因此，也更不容易出错。现代 C++ 中有一句谚语：“如果你显式使用循环的话，说明你不了解 STL 算法。”

P.2 用 ISO 标准 C++ 写代码

好吧，这条规则太简单了。要得到一个可移植的 C++ 程序，这条规则非常容易理解。使用当前的 C++ 标准，不要使用编译器扩展。此外，要注意未定义行为和实现定义行为。

- **未定义行为**：那什么都不用谈了。你的程序可能产生正确的结果，也可能产生错误的结果，可能在运行时崩溃，也可能连编译都过不去。当移植到一个新平台时，当升级到一个新编译器时，或者当一个无关的代码发生变化时，程序的行为都可能发生变化。
- **实现定义行为**：程序的行为可能因编译器实现而异。实现必须在文档里描述实际的行为。

当你必须使用没有写在 ISO 标准里的扩展时，可以用一个稳定的接口将它们封装起来。

会着火的语义

C++ 社区有一个描述未定义行为的谚语。当你的程序有未定义行为时，你的程序有“着火”的语义——你的计算机可能会着火。

P.3 表达意图

从以下的隐式循环和显式循环中，你可以看出什么意图？

```
for (const auto& v: vec) { ... }                    // (1)

for (auto& v: vec) { ... }                          // (2)
```

```
std::for_each(std::execution::par, vec, [](auto v) { ... });    // (3)
```

循环 (1) 并不修改容器 `vec` 的元素。这一点对于基于范围的 `for` 循环 (2) 并不成立。算法 `std::for_each` (3) 以并行方式（`std::execution::par`）执行。这意味着我们并不关心以何种顺序处理元素。

表达意图也是良好的代码文档的一个重要准则。文档应该说明代码会做什么，而不是代码会怎么做。

P.4　理想情况下，程序应该是静态类型安全的

C++ 是一种静态类型的语言。静态类型意味着编译器知道数据的类型，此外，还说明，编译器可以检测到类型错误。由于现有的问题领域，我们并非一直能够达到这一目标，但对于联合体、转型（cast[1]）、数组退化、范围错误或窄化转换，确实是有办法的。

- 在 C++17 中，可以使用 `std::variant` 安全地替代联合体。
- 基于模板的泛型代码减少了转型的需要，因此，也减少了类型错误。
- 当用一个 C 数组调用一个函数时，就会发生数组退化。函数需要用指向数组第一个元素的指针，另加数组的长度。这意味着，你从一个类型丰富的数据结构 C 数组开始，却以类型极差的数组首项指针结束。解决方法在 C++20 里：`std::span`。`std::span` 可以自动推算出 C 数组的大小，也可以防止范围错误的发生。如果你还没有使用 C++20，请使用 Guidelines 支持库（GSL）提供的实现。
- 窄化转换是对算术值的有精度损失的隐式转换。

```
int i1(3.14);
int i2 = 3.14;
```

如果你使用 `{}` 初始化语法，编译器就能检测到窄化转换。

```
int i1{3.14};
int i2 = {3.14};
```

P.5　编译期检查优先于运行期检查

如果可以在编译期检查，那就应该在编译期检查。这对 C++ 来说已经是惯用法了。从 C++11 开始，该语言就已经支持 `static_assert` 了。由于有 `static_assert`，编译器可以对 `static_assert(size(int) >= 4)` 这样的表达式进行求值，并在有问题时产生一个编译错误。此外，类型特征（type traits）库允许你表达强大的条件，如 `static_assert(std::is_integral<T>::value)`。当 `static_assert` 中的表达式求值为 `false` 时，编译器会输出一个可读的错误信息。

1　译者注：cast 也有“强制类型转换”“强制转换”等其他译法。

P.6 不能在编译期检查的事项应该在运行期检查

多亏有 dynamic_cast，你可以安全地将类的指针和引用沿着继承层次结构进行向上、向下以及侧向的转换。如果转型失败，对于指针，你会得到一个 nullptr；对于引用，则会得到一个 std::bad_cast 异常。请阅读第 5 章中“dynamic_cast”一节中的更多细节。

P.7 尽早识别运行期错误

可以采取很多对策来摆脱运行期错误。作为一名程序员，你应该管好指针和 C 数组，检查它们的范围。当然，对于转换，同样需要检查：如有可能，应尽量避免转换，对于窄化转换，尤其如此。检查输入也属于这个范畴。

P.8 不要泄漏任何资源

资源泄漏对于长期运行的程序来讲尤其致命。资源可以是内存，也可以是文件句柄或套接字。处理资源的惯用法是 RAII。RAII 是 Resource Acquisition Is Initialization（资源获取即初始化）的缩写，本质上意味着你在用户定义类型的构造函数中获取资源，在析构函数中释放资源。通过使对象成为一个有作用域的对象，C++ 的运行时会自动照顾到资源的生存期。C++ 大量使用 RAII：锁负责处理互斥量，智能指针负责处理原始内存，STL 的容器负责处理底层元素，等等。

P.9 不要浪费时间和空间

节省时间和空间都是一种美德。推理简明扼要：我们用的是 C++。你发现下面循环中的问题了吗？

```
void lower(std::string s) {
    for (unsigned int i = 0; i <= std::strlen(s.data()); ++i) {
        s[i] = std::tolower(s[i]);
    }
}
```

使用 STL 中的算法 std::transform，就可以把前面的函数变成一行。

```
std::transform(s.begin(), s.end(), s.begin(),
               [](char c) { return std::tolower(c); });
```

与函数 lower 相比，算法 std::transform 自动确定了字符串的大小。因此，你不需要用 std::strlen 指定字符串的长度。

下面是另一个经常出现在生产代码中的典型例子。为一个用户定义的数据类型声明

拷贝语义（拷贝构造函数和拷贝赋值运算符）会抑制自动定义的移动语义（移动构造函数和移动赋值运算符）。最终，编译器永远用不了廉价的移动语义（即使实际上移动是适用的），而只能一直依赖代价高昂的拷贝语义。

```
struct S {
    std::string s_;
    S(std::string s): s_(s) {}
    S(const S& rhs): s_(rhs.s_) {}
    S& operator = (const S& rhs) { s_ = rhs.s_; return *this; }
};

S s1;
S s2 = std::move(s1);                    // 进行拷贝，而不能从 s1.s_ 移动
```

P.10　不可变数据优先于可变数据

使用不可变数据的理由有很多。首先，当你使用常量时，你的代码更容易验证。常量也有更高的优化潜力。但最重要的是，常量在并发程序中具有很大的优势。不可变数据在设计上是没有数据竞争的，因为数据竞争的必要条件就是对数据进行修改。

P.11　封装杂乱的构件，不要让它在代码中散布开

混乱的代码往往是低级代码，易于隐藏错误，容易出问题。如果可能的话，用 STL 中的高级构件（如容器或算法）来取代你的杂乱代码。如果这不可能，就把那些杂乱的代码封装到一个用户定义的类型或函数中去。

P.12　适当使用辅助工具

计算机比人类更擅长做枯燥和重复性的工作。也就是说，应该使用静态分析工具、并发工具和测试工具来自动完成这些验证步骤。用一个以上的 C++ 编译器来编译代码，往往是验证代码的最简方式。一个编译器可能检测不到某种未定义行为，而另一个编译器可能会在同样情况下发出警告或产生错误。

P.13　适当使用支持库

这也很好解释。你应该去找设计良好、文档齐全、支持良好的库。你会得到经过良好测试、几乎没有错误的库，其中的算法经过领域专家的高度优化。突出的例子包括：C++ 标准库、Guidelines 支持库和 Boost 库。

本章精华

重要

- 理念性规则（或元规则）为具体规则提供理论依据。理想情况下，具体规则可以从理念性规则中推导出来。
- 在代码中直接表达意图和思想。
- 用 ISO 标准 C++ 编写代码，并使用支持库和辅助工具。
- 一个程序应该是静态类型安全的，因此，应该可在编译时进行检查。当这不可能时，要及早捕获运行期的错误。
- 不要浪费资源，如空间或时间。
- 将杂乱的构件封装在一个稳定的接口后面。