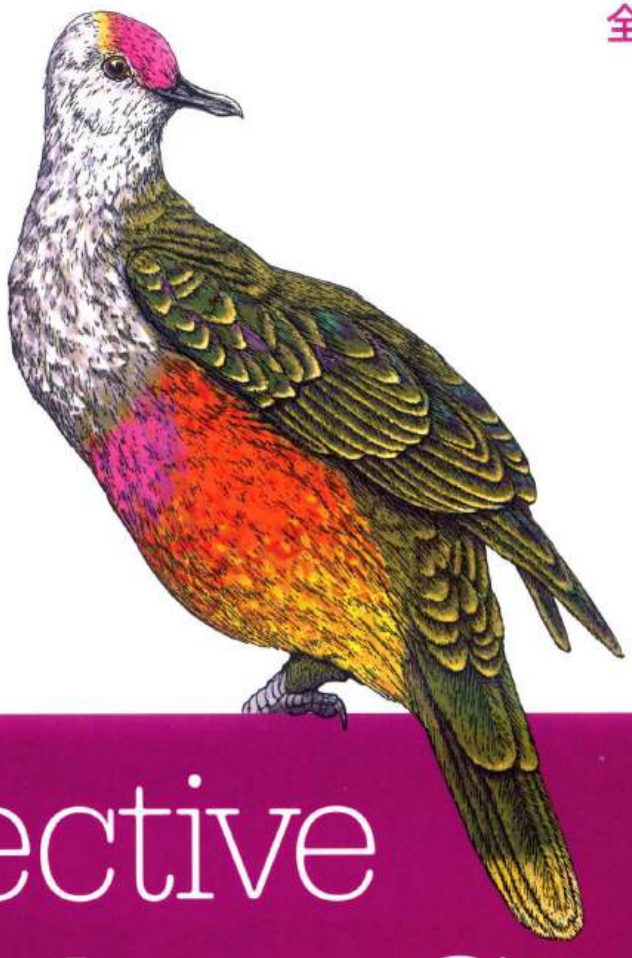
O'REILLY®

全彩印刷



Effective Modern C++

（中文版）

42招独家技巧助你改善C++11和C++14的高效用法

Scott Meyers 著
高博 译

中国电力出版社

O'REILLY®

Effective Modern C++ (中文版)

想要彻底理解C++11和C++14，不可止步于熟悉它们引入的语言特性（例如，auto型别推导、移动语义、lambda表达式，以及并发支持）。挑战在于高效地运用这些特性，从而使你的软件具备正确性、高效率、可维护性和可移植性。这正是本书意欲达成的定位。它描述的正是使用C++11和C++14（即现代C++）来编写真正卓越的软件之道。

涵盖以下主题：

- **大括号初始化、noexcept规格、完美转发，以及智能指针的make函数的优缺点。**
- **std::move、std::forward、右值引用和万能引用之间的联系。**
- **编写整洁、正确，以及高效的lambda表达式的方法。**
- **std::atomic和volatile有怎样的区别，它们分别用于什么场合，以及它们和C++的并发API有何联系。**
- **“旧”C++程序设计（即C++98）中的最佳实战要求在现代C++的软件开发中作出哪些修订。**

本书沿用了Scott Meyers早期作品中业已证明的基于指导原则和实例驱动的格式，但介绍的是全新材料。本书是所有C++软件开发工程师的必读之选。

“学会了C++基础知识以后，是Scott Meyers的*Effective C++*丛书教会了我如何在产品代码中运用C++。本书是最具重要性的一本学习手册，它给你关于核心指导原则、程序设计风格和习惯用法方面的建议，使你能够高效地、适当地使用现代C++。手头还没有一本这样的书吗？就买这本，就趁现在！”

——Herb Sutter
C++标准委员会主席，
微软公司C++软件架构师

20多年来，**Scott Meyers**的Effective C++丛书（包括《Effective C++》、《More Effecitve C++》和《Effective STL》）已经为C++程序设计指南的业界设立标杆。他清晰明了引人入胜的、对复杂技术材料进行条分缕析的阐释为他赢得了世界范围内的称誉，也使他成为一名广受欢迎的培训师、咨询顾问和会议讲师。他拥有布朗大学计算机科学专业的博士学位。

PROGRAMMING/C++

O'Reilly Media, Inc.授权中国电力出版社出版

Effective Modern C++

（中文版）

Scott Meyers 著
高博 译

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Effective Modern C++ / (美)斯科特·迈耶（Scott Meyers）著；高博译. — 北京：中国电力出版社，2018.4

书名原文：Effective Modern C++

ISBN 978-7-5198-1774-9

I. ①E… II. ①斯… ②高… III. ①C++语言－程序设计 IV. ①TP312.8

中国版本图书馆CIP数据核字(2018)第037670号

北京市版权局著作权合同登记 图字：01-2017-1820号

出版发行：中国电力出版社

地　址：北京市东城区北京站西街19号（邮政编码100005）

网　址：http://www.cepp.sgcc.com.cn

责任编辑：刘　炽（liuchi1030@163.com）

责任校对：朱丽芳

装帧设计：Ellie Volkhausen，张　健

责任印制：杨晓东

印　刷：北京盛通印刷股份有限公司

版　次：2018年4月第一版

印　次：2018年4月北京第一次印刷

开　本：750毫米×980毫米　16开本

印　张：19

字　数：367千字

印　数：0001—3000册

定　价：99.00元

对本书的赞誉

对 C++ 还有爱吗？理应如此！现代 C++（即 C++11/C++14）远不止是修修补补。考虑所有的新功能，这简直是一门语言的脱胎换骨。你在寻求指导和帮助吗？那么本书肯定就是你想找的。关于 C++，Scott Meyers 一直是精确、质量和惊喜的代名词。

——Gerhard Kreuzer

西门子股份公司研发工程师

精深的专业人士很难寻觅。完美主义的传道授业——讲求策略和言简意赅的作者也是一人难求。当你发现两者体现在同一个人身上时，你知道你找到的会是一种享受。《Effective Modern C++》是一位完美的技术作家高山仰止的成就。它在错综复杂、相互联系的话题之间游走，条分缕析地、意义明确地、井井有条地进行了阐明，而所有这些都在洗练的文笔中娓娓道来。你不太可能在《Effective Modern C++》中找到技术错误，枯燥段落，甚至偷懒的词句。

——Andrei Alexandrescu 博士，

Facebook 研究科学家，《Modern C++ Design》作者

作为拥有超过 20 年 C++ 经验的人，为了充分利用现代 C++（既要习得最佳实践，又要避免各种陷阱），我强烈建议你阅读本书、彻底阅读本书，并经常参考它！当然，我从本书中学到了很多新知识！

——Nevin Liber

DRW 交易集团公司高级软件工程师

C++ 的缔造者 Bjarne Stroustrup 如是说："C++11 感觉像是一种新的语言。"《Effective Modern C++》使得我们能够清楚地向日常使用 C++ 的软件工程师解释，如何从 C++11/C++14 的新特性和习惯用法中受益，并和他们产生共鸣。

Scott Meyers 出品，必属精品！

——Cassio Neri

FX 定量分析师，劳埃德银行集团

Scott 掌握了从复杂性中抽出容易理解的内核这个窍门。他的 Effective C++ 丛书有助于改善上一代 C++ 程序员的程序设计风格，而这本新书似乎定位于为使用现代 C++ 的人做同样的事情。

——Roger Orr

OR/2 有限公司，ISO C++ 标准委员会成员

《Effective Modern C++》是提高你的现代 C++ 技能的上佳工具。它不仅教会你如何使用、何时何地使用现代 C++，而且还是有效地使用。它还解释了背后的原因何在。毫无疑问，Scott 清晰而有见地的文字分布在 42 个经过精心思考的条款上，使得程序员能够更好地理解这门语言。

——Bart Vandewoestyne

研发工程师，C++ 爱好者

我喜欢 C++，几十年来它一直是我在工作中使用的工具。而且，它的最新特性比我以前想像的更强大、更富有表现力。但是，所有这些选择都带来了一个问题："何时以及如何应用这些特性呢？"如以往一样，Scott 的 Effective C++ 丛书是这个问题的明确答案。

——Damien Watkins

CSIRO 计算软件工程组组长

这是一本关于过渡到现代 C++ 的上佳读物，新的 C++11/14 语言特性被和 C++98 参照着描述，主题条款很容易参读，并且在每个部分末尾都给出了总结性建议。无论对于入门还是高级的 C++ 开发工程师，本书都既有娱乐性又有实用性。

——Rachel Cheng

F5 网络

如果你正在从 C++98/03 迁移至 C++11/14，则肯定需要 Scott 在《Effective Modern C++》中提供的极其实用而清晰的信息。如果你已经在撰写 C++11 代码，那么可能会通过 Scott 针对该语言的主要新功能的深入讨论来发现新功能的问题所在。无论你的情况属于哪一种，本书绝对值得你为阅读而付出的时间。

——Rob Stewart

Boost Steering 委员会成员（boost.org）

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——*Wired*

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

推荐序

一份宏大的作品，能够带动一个领域的蓬勃发展。C++ 社群在相对沉寂十数年之后迎来新一波热潮，原因是，作为一门编程工具的最核心，C++ 的语言和标准库都出现巨大的扩展和强化。这一番大变革始自 2011，并分别在 2014、2017 持续进化。业界习惯性地将这些新版本统称为 Modern C++，用以区别“传统”C++。

作为知名书系的最新作品，《Effective Modern C++》的佳质和佳评一如其早期同门作品《Effective C++》和《More Effective C++》。本书延续作者 Scott Meyers 的一贯风格和质量，其最大特质就是，不但告诉我们 How，更用巨大而精良的篇幅告诉我们 Why。作者穷追猛打讲究再三的劲儿，常让我筋疲力尽，痛并快乐地爬行于某个条款之际拍案而叹：“天啊，还有下一页！”

而我，是一个在 C++ 领域已经生活 25 年的老兵。

是的，我是一个在 C++ 领域生活了 25 年的老兵，这意味着我具备相当的 C++ 能力。尽管如此，面对这号称全新语言的 Modern C++，我时或也有力不能逮、掩卷长叹的焦躁，特别是面对 Rvalue Reference（右值引用）、Perfect Forwarding（完美转发）、Metaprogramming（元编程）、Type Deduction（类型推导）、Type Traits（型别特征）等艰涩主题的时候。然而正是在特别艰涩的主题上你可以领受本书的巨大价值：如果你想完善根基，本书是你的唯一选择。

这样一本好书引介到中国，需要一位好译者和一家好出版社。高博先生是非常用心的好译者，技术上和文字上学养俱佳。我和他结缘于多年前的 emails，因着他的用功和成果，深感此书所托得人。诚如各位所见，这是一本编排与细节俱皆上乘的出版物；我曾经亲

手编排超过 50 本书，完全知道这样的呈现需要多少细琐的步骤和细心的浇灌。本书的出版质量足以标示中国计算机图书的长足进步和精益求精。

侯捷

White Rock

译者序

Effective C++ 系列丛书对于 C++ 世界影响之深远，在每个 C++ 程序员的心中都清楚明了。如果说任何一门语言的重要世代升级都可以分为研究阶段、试用阶段和成熟阶段的话，那么无疑对于 C++ 语言来说，Effective C++ 系列丛书增加了新成员或是推出了新版本，都意味着相应的语言世代进入了成熟阶段。C++11 是 C++ 语言的一个划时代的版本，C++11 及其后续版本被统称为“Modern C++”，即“现代 C++”，而《Effective Modern C++》的问世，也就标志着这一世代的 C++ 语言系列版本进入了成熟阶段。一方面，如果在 C++11 推出之际，人们会对现代 C++ 有着“怕还是不成熟吧”的猜疑甚至惊惧，尚可说是情有可原，那么，如果在本书推出以后还要以任何借口不去热情拥抱现代 C++，恐怕就难免落伍了；另一方面，尽管“C++ 的版本升级已经进入了快车道”[注1]，在本书简体中文译本付梓之际，C++17 已被正式批准[注2]，内含有若干像带模板实参型别推导的构造函数（Constructor Template Argument Deduction，简称 CTAD）这样令人兴奋的新特性，甚至 C++20 也已经在积极迅速地推进之中，但它们都属于同一世代，因此从现在开始学习和掌握现代 C++ 可谓恰逢其时。

现代 C++ 在语言方面所进行的大刀阔斧、釜底抽薪式的变革，无须赘言。但是这些变革背后，更重要的反而是其保持不变者，即所谓 C++ 语言的精神，或曰设计哲学。例如，由实际问题驱动，并立刻可用于解决实际问题。现代 C++ 中提供了并发 API，在语言层面上支持并发程序设计，结束了在各种体系结构和操作系统之上存在很多互不兼容的第三方并发库的乱局，就是这种设计哲学的体现。程序员应该能够自由地选择自己的程序

注 1：见《程序员》杂志 2014 年 12 月刊，“C++ 之父 Bjarne Stroustrup 访谈录”。

注 2：见 C++ 标准委员会主席 Herb Sutter 在 2017 年 9 月 6 日发表的博客文章《C++17 is formally approved》。

设计风格，而语言应该为该风格提供完备的支持。现代C++引入了lambda表达式、返回值型别尾序语法等语言特性，提供了对函数式程序设计风格的支持，使得从Haskell甚至LISP开始学习程序设计者也能在C++世界中如鱼得水。不对型别系统开隐式的天窗。现代C++对于型别系统的改造是最彻底的，这也是现代C++的最高明之处。不存在的代码才是不会出错的代码，很多原先需要显式写出的型别，现在都可以使用auto来指定，甚至略去不写。但是C++并没有因此变成一种弱型别语言，静态型别检查仍然是代码在编译时就保证正确性的法宝。C++借助于型别推导机制来判定哪里应该使用什么型别，但是型别一旦判定，就固定下来。换言之，C++中未显式指定的型别并不是“动态型别”而只是“待定型别”，这种思想的来源是传统世代的模板。不为用不到的语言特性付出额外代价。现代C++中非常细心地引入了所谓的右值引用，实际上就是一种“人工右值”，尽可能地复用已经在内存中存在的值，以一次仅包含若干指针赋值的“移动”操作代替“先复制，再析构”这样两次甚至更多次代价高昂且有内存分配异常风险的操作。现代C++中还允许显式删除不使用的函数，从而在目标代码中完全不生成这些函数而不是通过访问层级机制等事后措施来阻断对业已生成的函数的访问，因为在后一种情况下内存足迹已至。可以说，每一个现代C++引入的新语言特性，都让现代C++变得“更加现代”的同时，也变得“更加C++”。核心语言特性，从noexcept、constexpr和“=default”这样的声明饰词，到充分利用多核CPU流水线尺寸的std::hardware_destructive_interference_size和std::hardware_constructive_interference_size常量[注3]，在被任何应用程序开发者使用之前首先会在标准模板库（Standard Template Library）中被彻底地使用。

C++语言之难，主要还是在于众多语言特性之间的综合交叉。尤其对于新的语言特性，掌握其本身往往并不很难，而要考虑到它与众多业已存在的语言特性之发生的相互作用，就不容易了。例如，统一初始化的大括号语法应该说不难理解，使用起来也很直观，但是将这种形式的字面量用作实参传递给函数的时候，却会造成完美转发的失败。再例如移动语义的引入使得标准模板库有机会实施一些优化，这样做的结果是只要符合一定条件的遗留代码在不作任何改动的前提下只要使用支持现代C++的编译器重新编译一遍就能获得更佳性能。但是移动语义同样也影响到了C++98中经典而简洁的“大三律”，在撰写构造和析构函数中时需要考虑的规则就更加复杂多变。举个类比，就好比在研究院里刚刚学完一套复杂的数学工具，内含数以百计的定理和公式，马上就要综合运用它们来解决现实的物理实验问题，甚至需要对这些定理和公式本身在深刻理解的基础上得出新的推论和公式，这样的难度可想而知。这就是本书最大的价值所在：Scott Meyers就

注3：该语言特性在C++17中被引入，意义在于首次从语言层面关注了比内存更加低级的硬件特性，以更好地支持并发程序设计。更多细节参见：http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0154r1.html。

是一名全球公认的“学霸”，他已经在吃透了标准并做了大量的实验基础之上，将自己的心经托出，这样所有的研发同仁都可以不必再从头去读标准文本（毕竟那是厚达1700多页的大部头），甚至也可以不必去读那些事无巨细、逐点讲解语言特性的现代C++教科书——本书不仅重点突出，而且案例丰富。一言以蔽之，本书的重点不在于讲解语言特性“是什么”，而在于讲解它们“如何高效运用”以及“背后的思想”，并且极尽深入浅出之能事，相信了解“Scott Meyers风格”的读者一定会再次大呼过瘾！

2017年12月，我作为出品人参加了在北京举办的C++ Summit大会[注4]。期间我和C++标准委员会成员Michael Spertus和Michael Wong交流时问他们：Effective C++系列丛书对于C++标准委员会来说，意味着什么？他们稍作思考后回答：Effective C++系列丛书就像是标准的问题发现者和描述者，在Effective C++系列丛书中提到的解决问题的技术和技巧，往往会在新版本的标准中以语言特性的引入或更新，或者库实现更新的形式予以吸纳。这个视角果然不同寻常！仔细想来，《Effective Modern C++》中的大量条款的确可以看作是对于该系列丛书旧有成员中的条款更新。行业的推动力，在于大量的一线工作者和理论研究者的共同努力和相互促进，C++语言在这方面确实可称典范。

本人从1998年开始接触C++语言，至今二十年矣。时至今日，首次阅读Effective C++时带来的心灵震撼仍历历如昨——世上竟有如此神作，把一门复杂而厚重的程序设计语言条分缕析、娓娓道来，把一本技术参考书写得比小说还可读耐看。这当然除了原作功力之外，也离不开它的译者侯捷先生。余生也幸，争取到了《Effective Modern C++》的翻译机会，披星戴月、呕心沥血两年有余，三易其稿，方得将成果献诸中国读者。感谢侯捷先生拨冗作推荐序，感谢杭州好友凌杰、上海好友李建忠和林应、北京好友何万青、深圳好友刘海平和伦敦好友吴天明审阅稿件并给予大量无价的意见。最后，家人在我译书过程中体谅照顾甚多，希望本书的出版，能给你们带来快乐。

高博

上海，阅读隧道

新浪微博：但以理_高博

注4：　会议官网：http://cpp-summit.org。

献给爱犬 Darla，
非凡的黑色拉布拉多寻回犬

目录

出版商声明

代码示例的使用

这本书是为了帮助你完成工作。一般来说，如果本书提供了示例代码，则可以在你的程序和文档中使用它。除非你直接复制本书的大部分代码，否则无需联系我们获得许可。例如，进行程序设计时使用本书中几个代码块的程序不需要获得许可。销售或分发 O'Reilly 书籍中的示例 CD-ROM 则需要获得许可。通过引用本书并引用示例代码来回答问题不需要获得许可。将本书中的大量示例代码合并到产品文档中则需要获得许可。

我们赞赏但不要求声明来源。如果声明，则请包括标题、作者、出版商和 ISBN。如："Effective Modern C++ by Scott Meyers(O'Reilly). Copyright 2015 Scott Meyers, 978-1-491-90399-5"。

如果你担心对代码示例的使用超出了合理使用范围或上述许可范围，请随时联系我们：*permissions@oreilly.com*。

Safari® Books Online

Safari® Books Online 是个按需提供的数字图书馆，以全球领先的技术和商业作者的书籍和视频形式提供专业内容。

技术专业人员、软件开发人员、网页设计师，以及业务和创意专业人员使用 Safari® Books Online 作为研究、解惑、学习和认证培训的主要资源。

Safari® Books Online 为企业、政府、教育机构和个人提供了一系列的产品组合和定价。

订阅者可以在一个快捷搜索的数据库中访问多家出版社提供的成千上万种图书、培训视频和正式出版前手稿，如 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他数百家出版公司。关于 Safari Books Online 的更多信息，请访问我们的在线网站。

如何联系我们

关于本书的评论和问题可以发给出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

要评论或询问有关本书的技术问题，请发送电子邮件到 *bookquestions@oreilly.com*。

有关我们的书籍、课程、会议和新闻的更多信息，请访问我们的网站 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*。

在 Twitter 上关注我们：*http://twitter.com/oreillymedia*。

在 YouTube 上观看我们的视频：*http://www.youtube.com/oreillymedia*。

致谢

我从 2009 年开始研究当时称为 C++0x 的语言（这是 C++11 的萌芽）。我在 Usenet 新闻组 comp.std.c++ 中发表了大量的问题帖子，我非常感谢该社区的成员（尤其是 Daniel Krügler）发表的非常有帮助的回帖。更晚近的几年来，当有关于 C++11 和 C++14 的问题时，我已经转向了 Stack Overflow，并且同样感谢该社区帮助我更好地理解现代 C++。

2010 年，我准备了关于 C++0x 的培训课程（最终整理成《Overview of the New C++》一书，Artima Publishing，2010 出版）。这些材料的成形和我的知识积累都极大地受益于 Stephan T. Lavavej、Bernhard Merkle、Stanley Friesen、Leor Zolman、Hendrik Schober 和 Anthony Williams 的技术审查。如果没有他们的帮助，我可能永远无法达到写作《Effective Modern C++》的高度。顺便说一下，本书书名是由几位读者在 2014 年 2 月 18 日发表的博客“帮我命名我的书”中以提名或点赞方式给出的。Andrei Alexandrescu（《Modern C++ Design》作者，Addison-Wesley，2001 出版）非常友好，本书书名没有从他浩如烟海的术语中随便选一个了事。

我无法确定本书中所有信息的来源，但是有些来源具有相对比较直接的影响。条款 4，使用未加定义的模板来诈取编译器给出的型别信息是由 Stephan T. Lavavej 提出的，而 Matt P. Dziu-binski 则引起了我对 Boost.TypeIndex 的注意。在条款 5 中，`unsigned std::vector<int>::size_type` 一例来自 Andrey Karpov 在 2010 年 2 月 28 日发表的文章《In what way can C++0x standard help you eliminate 64-bit errors》。同一条款中的 `std::pair <std::string, int>` 和 `std::pair <const std::string, int>` 一例来自 Stephan T. Lavavej 在 Going Native 2012 会议上的演讲“STL11: Magic && Secrets”。条款 6 受到了 Herb Sutter 在 2013 年 8 月 12 日的文章《GotW

#94 Solution: AAA Style (Almost Always Auto)》的启发。条款 9 的动机是 Martinho Fernandes 在 2012 年 5 月 27 日的博客“Handling dependent names”。条款 12 中演示依引用饰词重载的示例是基于 Casey 对 2014 年 1 月 14 日发布在 Stack Overflow 的“What's a use case for overloading member functions on reference”一问的回答。我在条款 15 中 C++14 对 constexpr 函数的扩展支持则包含了我从 Rein Halbersma 那里得到的信息。条款 16 是基于 Herb Sutter 在 C++ and Beyond 2012 会议的演讲“You don't know const and mutable”。条款 18 中建议工厂函数返回 `std::unique_ptr` 型别对象是基于 Herb Sutter 在 2013 年 5 月 30 日发表的文章《GotW# 90 Solution: Factories》。条款 19 中的 fastLoadWidget 源自 Herb Sutter 在 Going Native 2013 大会上的演讲“My Favorite C++ 10-Liner”。条款 22 中，我对 `std::unique_ptr` 和不完整型别的处理借鉴了 Herb Sutter 在 2011 年 11 月 27 日的文章《GotW #100: Compilation Firewalls》以及 Howard Hinnant 在 2011 年 5 月 22 日对 Stack Overflow“Is std::unique_ptr<T> required to know the full definition of T?”一问的回答。条款 25 中附加的矩阵例子基于 David Abrahams 的著作。JoeArgonne 在 2012 年 12 月 8 日对其 2012 年 11 月 30 日的博客文章发表评论的：“Another alternative to lambda move capture”是条款 32 中基于 `std::bind` 的方法来模拟 C++11 中的初始化捕获的来源。条款 37 中对 `std::thread` 的析构函数隐式分离的问题解释取自 Hans-J. Boehm 在 2008 年 12 月 4 日发表的文章《N2802: A plea to reconsider detach-on-destruction for thread objects》。条款 41 原本是由 David Abrahams 在 2009 年 8 月 15 日博客文章的讨论激发的，原题“Want speed? Pass by value”。只移型别应该进行特殊处理的想法来自 Matthew Fioravante，而基于赋值的复制分析则源自 Howard Hinnant 的评论。在条款 42 中，Stephan T. Lavavej 和 Howard Hinnant 帮助我了解了置入和插入函数的相对性能剖析，Michael Winterberg 提请我注意插入会如何导致资源泄漏（Michael Winterberg 将这个想法归功于 Sean Parent 在 Going Native 2013 大会上的演讲“C++ Seasoning”）。Michael 还指出置入函数如何使用了直接初始化，而插入函数则使用的是复制初始化。

审查技术图书草稿是一项艰巨的、耗时的，又是至关重要的任务，我很幸运，有那么多人愿意为我做这件事。《Effective Modern C++》全部或部分草稿的正式审查者有 Cassio Neri、Nate Kohl、Gerhard Kreuzer、Leor Zolman、Bart Vandewoestyne、Stephan T. Lavavej、Nevin “:-)” Liber、Rachel Cheng、Rob Stewart、Bob Steagall、Damien Watkins、Bradley E. Needham、Rainer Grimm、Fredrik Winkler、Jonathan Wakely、Herb Sutter、Andrei Alexandrescu、Eric Niebler、Thomas Becker、Roger Orr、Anthony Williams、Michael Winterberg、Benjamin Huchley、Tom Kirby-Green、Alexey A Nikitin、William Dealtry、Hubert Matthews 和 Tomasz Kamiński。我

还通过 O'Reilly 的早期发行电子书和 Safari 图书在线粗剪，我的博客“The View from Aristeia”和电子邮件等渠道收到了一些读者的反馈。我很感激这些人中的每一个。在他们的帮助下，这本书要比修改之前好太多了。我要特别感谢 Stephan T. Lavavej 和 Rob Stewart，他们的审稿意见是那样的详细和全面，我感觉他们在这本书上的投入不逊于我本人。我还要特别感谢 Leor Zolman，除了复印稿件之外，他还仔细检查了本书中所有的代码示例。

Gerhard Kreuzer、Emyr Williams 和 Bradley E. Needham 对本书的数字版本进行了专门审阅。

我之所以决定将代码显示的行长限制为 64 个字符（这个长度最可能在打印机以及各种数字设备的设备方向和字体配置中正确显示），是基于 Michael Maher 提供的数据。

Ashley Morgan Williamsd 使我在奥斯威戈湖的用餐成为了独一无二的享乐。谁要是想尝试下够一个大老爷们吃到饱的凯撒沙拉，她就是你应该找的妙龄巧手啦。

在好容易闯关成功抱回爱妻 Nancy L. Urbano 的 20 多年以后，她再次忍受了我长达数月说话心不在焉的状态，以及由辞职、暴怒和及时闪现的善解人意和体贴混合而成的鸡尾苦酒。与其同时我们的爱犬 Darla 在我盯着电脑屏幕一连数小时的时候，往往会心满意足地大睡特睡，但她却永远不会让我忘记键盘以外的精彩生活。

绪论

如果你是一名身经百战的C++程序员，又恰好和我有那么点儿气味相投，你会在和C++11最初打上交道时暗忖："嗯，我明白，这还是C++，锦上添花而已。"但随着学习的深入，你会对变化之纵深吃惊不小：auto声明式、基于范围的循环、lambda表达式，以及右值引用。这些已经改变了C++的面貌，这还没有把全新的并发特性计算以内。随之而来的还有诸多习惯用法的改变。0和typedef已经过时，nullptr和别名声明式大行其道。枚举量如今要限定作用域。相对于内建指针而言，智能指针成为优选。在正常情况下，对象的移动语义要好过复制语义。

C++11已有大量内容要学习，C++14自然更不必说。

更关键的是，想要高效地利用这些新能力，任重而道远。如果你仅仅想了解"现代"C++特性的基本信息，资料俯拾皆是。但如果你想要寻觅如何采用这些特性去创建正确、高效、易维护、可移植的软件，恐怕就没那么容易了。本书正是应此而生。本书的写作目的并非对于C++11和C++14特性的泛泛介绍，而是为了揭示它们的高效应用。

本书中的信息被分解成若干准则，称为条款。你想要理解型别推导的各种形式吗？又或者想要知道何时该使用（或不该使用）auto声明式？你是否有兴趣知道为何const成员函数应该保证线程安全，或如何使用std::unique_ptr实现Pimpl习惯用法，或为何在lambda表达式中应该避免默认捕获模式（default capture mode），或std::atomic和volatile的区别？答案统统在本书中。还不止如此，这些答案都与平台无关，且符合标准。本书是围绕着可移植的C++展开的。

本书中的条款都是准则，并非规则，因为准则允许有例外。条款给出的建议并非最要紧的部分，建议背后的原理才是精华。只有掌握了原理，你才能判定，你的项目面临

的具体情况是否真的违反了条款所指。本书的真正目标并不在于告诉你什么该做，什么不该做，而是想要传达对 C++11 和 C++14 运作原理的更深入理解。

术语和惯例

为保证理解的一致性，厘清对于一些术语的认识很有必要。首先要说清楚的就是“C++”这个词，是不是有点儿讽刺意味呢。C++ 有四个官方钦定版本，都是以 ISO 标准被接受的年份命名的，它们是 C++98、C++03、C++11 和 C++14。C++98 和 C++03 仅有一些技术细节上的不同，所以在本书中我将它们统称为 C++98。当提到 C++11 时，我的意思是 C++11 和 C++14 都适用，因为 C++14 实际上是 C++11 的超集。而如果写的是 C++14，我仅指 C++14。如果我说的是什么都不附加的 C++，意思就是适用一切语言版本。

术语	意指的语言版本
C++	所有版本
C++98	C++98 和 C++03
C++11	C++11 和 C++14
C++14	C++14

依照上面的约定，我可能会有这样的表述：C++ 非常重视运行效率（所有版本都成立），C++98 缺乏并发支持（仅对 C++98 和 C++03 成立），C++11 支持 lambda 表达式（对 C++11 和 C++14 成立），C++14 提供了广义返回值型别推导（仅对 C++14 成立）。

C++11 被最广泛接受的特性可能莫过于移动语义，而移动语义的基础在于区分左值表达式和右值表达式。因为，一个对象是右值意味着能够对其实施移动语义，而左值则一般不然。从概念上说（实践上并不总是成立），右值对应的是函数返回的临时对象，而左值对应的是可指涉的对象，而指涉的途径则无论通过名字、指针，还是左值引用皆可。

有一种甄别表达式是否左值的实用方法富有启发性，那就是检查能否取得该表达式的地址。如果可以取得，那么该表达式基本上可以断定是左值。如果不可以，则其通常是右值。这种方法之所以说富有启发性，是因为它让你记得，表达式的型别与它是左值还是右值没有关系。换言之，给定一型别 T，则既有 T 型别的左值，也有 T 型别的右值。这一点在处理右值引用型别的形参时尤其要注意，因为该形参本身是个左值。

```
class Widget {
public:
  Widget(Widget&& rhs);          // rhs 是个左值，
  ...                            // 尽管它具有右值引用型别
};
```

在 Widget 的移动构造函数内部对 rhs 取址完全没有问题，所以 rhs 是个左值，尽管它的型别属于右值引用（基于类似的理由，我们可以得知，任何形参都是左值）。

上述代码片断还演示了我通常会遵循的若干惯例：

- 类名是 Widget。每当我想要一个任意用户自定义型别时，我就会使用 Widget 这个名字。并且，除非我需要展示类的特殊细节，我可能会不经声明就使用 Widget 类。

- 我有时会采用 rhs（“right-hand side”，右侧）这个形参名，尤其是用作移动操作（即移动构造函数和移动赋值运算符），以及复制操作（即复制构造函数和复制赋值运算符）的形参名。这个名字我也会用于二元运算符的右侧形参：

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

  上面的 lhs 表示“left-hand side”（左侧），这很好理解吧。

- 我会在代码和注释的某些部分使用特殊的排版，以吸引你去注意它们。在上述 Widget 移动运算符中，我将 rhs 的声明，以及注释中强调其是一个左值的部分排成了突显格式。突显的代码既不是说明它好，也不是说明它坏，只是说明这部分代码需要引起重视。

- 我使用“…”来表示“这里可能有其他代码”，这种全角省略号和 C++11 中变长模板源码中所使用的半角省略号“...”是完全不同的。[译注 1] 听上去有点儿容易混淆，其实不然：

  ```
  template<typename... Ts>                  // 这两行中的省略号
  void processVals(const Ts&... params)     // 都是源码组成部分
  {

    ...                                     // 而这一行中的省略号
                                            // 则表示“这里可能有其他代码”
  }
  ```

译注 1：原文中是采用“窄省略号”和“宽省略号”来区分两者的；简体中文版中则采用全角和半角来区分，这样更易辨识。

在模板 processVals 的声明语句中，可以看到我使用的是 typename 来声明模板中的型别形参，这仅仅是我的个人偏好。此处使用关键字 class 也同样可以。但当我演示 C++ 标准中的代码片断时，我会使用 class 来声明型别形参，因为标准中采用的就是那样的做法。

若某对象是依据同一型别的另一对象初始化出来的，则该新对象称为提供初始化依据的对象的一个**副本**，即使该副本是由移动构造函数创建的。这样称呼情有可原，因为 C++ 中并无术语用以区分某对象到底是经由复制构造函数创建的副本，还是经由移动构造函数创建的副本。[译注 2]

```
void someFunc(Widget w);        // someFunc 的形参 w 按值传递

Widget wid;                     // wid 是 Widget 型别的某个对象

someFunc(wid);                  // 在这个对 someFunc 的调用中，
                                // w 是 wid 经由复制构造函数创建的副本

someFunc(std::move(wid));       // 在这个对 someFunc 的调用中，
                                // w 是 wid 经由移动构造函数创建的副本
```

右值的副本通常经由移动构造函数创建，而左值的副本通常经由复制构造函数创建。这也就是说，如果你仅仅了解到某个对象是另一对象的副本，则还不能判断构造这个副本要花费多少成本。例如，在上述代码中，如果不知道传入 someFunc 的究竟是右值还是左值，就无法计算创建形参 w 所花费的成本（你还需要知道移动或复制 Widget 型别对象的具体成本）。

在函数调用中，调用方的表达式，称为函数的实参。实参的用处，是初始化函数的形参。在上面 someFunc 的第一次调用中，实参是 wid。而在第二次调用中，实参则是 std::move(wid)。在两次调用中，形参都是 w。实参和形参有着重大的区别，因为形参都是左值，而用来作为其初始化依据的实参，则既可能是右值，也可能是左值。这一点在完美转发（perfect forwarding）的过程中尤其关系重大，在这样的一个过程中，传递给某个函数的实参会被传递给另一函数，并保持其右值性（rvalueness）或左值性（lvalueness）（完美转发会在条款 30 中详细讨论）。

设计良好的函数都是**异常安全**的，这意味着它们至少会提供基本异常安全保证（即**基本保证**）。提供了基本保证的函数能够向调用者确保即使有异常抛出，程序的不变量不会受到影响（即不会有数据结构被破坏），且不会发生资源泄漏。而提供了强异常安全保证（即**强保证**）的函数则能够向调用者确保即使有异常抛出，程序状态会在调用前后保持不变。

译注 2：在原文中，“副本”和“复制构造函数”中的“复制”是同一个词“copy”。

当提及**函数对象**时，我通常意指某个对象，其型别支持 operator() 成员函数。换言之，就是说该对象表现得像个函数。我偶尔也会在一个更加广义的情形下使用该术语，意指任何可以采用非成员函数语法［即形如“*function Name(arguments)*”］调用之物。这种更宽泛的定义不仅涵盖了支持 operator() 的对象，还有标准函数以及 C 风格的函数指针（狭义定义来自 C++98，而广义定义来自 C++11[译注 3]）。进一步泛化这一术语的话，就涵盖了指涉到成员函数的指针，从而得到了所谓的**可调用物**。一般情况下，你可以不用关心这些含义之前的细微差别，函数指针也好，可调用物也罢，你只需知道它们在 C++ 中表示某种函数调用语法加以调用就行了。

经由 lambda 表达式创建的函数对象称为闭包，将 lambda 表达式和它们创建的闭包区分开来，意义不大，所以我经常把它们统称为 lambda 式。相似地，我也很少区分函数模板（即用以生成函数的模板）和函数模板（即从函数模板生成的函数）。类模板和模板类的情形同上。

C++ 中有很多事物能够加以声明和定义。声明的作用是引入名字和型别，而不给出细节，如存储位置或具体实现：

```
extern int x;                  // 对象声明

class Widget;                  // 类声明

bool func(const Widget& w);    // 函数声明

enum class Color;              // 限定作用域的枚举声明（参见条款 10）
```

定义则会给出存储位置和具体实现的细节：

```
int x;                         // 对象定义

class Widget {                 // 类定义
    ...
};

bool func(const Widget& w)
{ return w.size() < 10; }      // 函数定义

enum class Color
{ Yellow, Red, Blue };         // 限定作用域的枚举定义
```

定义同时也可以当声明用。所以，除非某些场合非给出定义不可，我倾向于只使用声明。

译注 3：此句的意义并不是说 C++98 中没有标准函数和 C 风格的函数指针这两种语言特性，而是说在 C++98 的语境中使用函数指针这一术语时，一般不特指这两者。

我把函数声明的形参型别和返回值型别这部分定义为函数的签名，而函数名字和形参名字则不属于签名的组成部分。在上例中，func 的签名是 bool(const Widget&)。函数声明除形参型别和返回值型别的其他组成元素（即可能存在的 noexcept 或 constexpr）则被排除在外（noexcept 和 constexpr 的内容详见条款 14 和条款 15）。“签名”的官方定义与我给出的稍有不同，但在本书中，我的定义更加实用（官方定义有时会省去返回值型别）。

通常来说，在 C++ 旧标准下写就的代码在 C++ 新标准下仍然会保有合法性，但有时标准化委员会也会废弃某些特性。这些特性在标准化的前途上已是穷途末路，并有可能在未来的标准中被去除。编译器可能会，也可能不会对于在代码中使用这些已废弃特性的行为给出警告，但是你应该尽力避免使用它们。它们不仅会造成未来的可移植性问题，而且还不如那些替换掉它们的新特性好用。例如，在 C++11 标准中，std::auto_ptr 就是个被废弃的特性，因为 std::unique_ptr 可以完成同样的任务，并且使用起来更方便。

标准有时会把某个操作的结果说成是未定义行为。意思是，其运行期行为不可预测，你当然会对这样的不确定性敬而远之。未定义行为的例子有，在方括号（“[]”）内使用越界值作为 std::vector 的下标，对未初始化的迭代器实施提领操作，或者进入数据竞险（即两个或更多线程同时访问同一内存位置，且其中至少有一个执行写操作的情形）。

我将内建的指针，就是 new 表达式返回的那些指针，称为裸指针。而与裸指针形成对照的，则是智能指针。智能指针通常都重载了指针提领运算符（operator-> 和 operator*），不过条款 20 会解释为什么 std::weak_ptr 是个例外。

在源代码注释中，我有时会将“构造函数”简写为“*ctor*”，将“析构函数”简写为“*dtor*”。[译注 4]

提交缺陷报告和改进建议

我已尽力使得本书中的信息清晰、准确、实用，但是改进空间是一定少不了的。如果你在本书中发现任何谬误（无论是技术的、文字的，还是排版的），或有任何改进建议，请发电子邮件给我：*emc++@aristeia.com*。每次重印时我都有机会修订 Effective Modern C++，但如果我不知道哪里有问题，也无从着手。

译注 4：简体中文翻译时，不作此区别。“ctor”一律译为“构造函数”，“dtor”一律译为“析构函数”。

欲查看已知问题，请参考本书勘误页面：*http://www.aristeia.com/BookErrata/emc++-errata.html*。

第 1 章

型别推导

C++98 仅有一套型别推导规则，用于函数模板。C++11 对这套规则进行了一些改动，并且增加了两套规则，一套用于 auto，另一套用于 decltype。后来，C++14 又扩展了能够运用 auto 和 decltype 的语境。型别推导应用范围的不断普及，使得人们不必再去写下那些不言自明或是完全冗余的型别。它还让 C++ 软件获得更高的适应性，因为在源代码的一个地方对一个型别实施的改动，可以自动通过型别推导传播到其他地方。然而，它也有可能导致写出来的代码较难看懂，因为编译器推导出的型别，可能不像我们所认为的那样显而易见。

想要使用现代 C++ 高效编程，就离不开对于型别推导操作的坚实理解。型别推导涉及的语境实在不胜枚举：在函数模板的调用中，在 auto 现身的大多数场景中，在 decltype 表达式中，特别是在 C++14 中那个神秘莫测的 decltype(auto) 结构中。

本章讨论的是每个 C++ 开发工程师都需要了解的有关型别推导的知识。本章解释了模板型别推导如何运作，auto 的型别推导如何构建在此运作规则之上，以及 decltype 独特的型别推导规则。本章还教你如何迫使编译器来展示其型别推导的结果，从而让你确信该结果如你所愿。

条款 1：理解模板型别推导

如果一个复杂系统的用户对于该系统的运作方式一无所知，然而却对其提供的服务表示满意，这就充分说明系统设计得好。如果从这样的角度来看，C++ 的模板型别推导取得了极大的成功。成百上千的程序员都在向函数模板传递实参，并拿到了完全满意

的结果，而这些程序员中却有很多人对于这些函数使用的型别是如何被推导出来的过程连最模糊的描述都讲不出来。

若是你也在这样无知无畏的程序员之列，那么我这里有一个好消息，还有一个坏消息。好消息是，模板的型别推导，是现代 C++ 最广泛应用的特性之一——auto 的基础。也就是说，如果你对 C++98 推导模板型别的运作方式感觉满意，那么你也会自然而然地愉快接受 C++11 中推导 auto 型别的运作方式。而坏消息则是，当模板型别推导规则应用于 auto 语境时，它们不像应用于模板时那么符合直觉。出于这个缘故，了解作为 auto 基础的模板型别推导的方方面面就变得相当重要了。本条款说明了你需要了解的一切。

请允许我用一小段伪代码来说明，函数模板大致形如：

```
template<typename T>
void f(ParamType param);
```

而一次调用则形如：

```
f(expr);                // 以某表达式调用 f
```

在编译期，编译器会通过 *expr* 推导两个型别：一个是 T 的型别，另一个是 *ParamType* 的型别，这两个型别往往不一样。因为，*ParamType* 常会包含了一些饰词，如 const 或引用符号等限定词。例如，若模板声明如下：

```
template<typename T>
void f(const T& param);        // ParamType 是 const T&
```

而调用语句如下：

```
int x = 0;

f(x);                          // 以一个 int 调用 f
```

在此例中，T 被推导为 int，而 *ParamType* 则被推导为 const int&。

我们很自然地会认为，T 的型别推导结果和传递给函数的实参型别是同一的。换句话说，T 的型别就是 *expr* 的型别。在上例中，情况确乎如此：x 的型别是 int，T 的型别也推导为 int。但是，这一点并不总是成立。T 的型别推导结果，不仅仅依赖 *expr* 的型别，还依赖 *ParamType* 的形式。具体要分三种情况讨论：

- *ParamType* 具有指针或引用型别，但不是个万能引用（万能引用会在条款 24 中介绍，现在你只需知道有这么个东西，并且它们与左值引用和右值引用都有所区别即可）。

- *ParamType* 是一个万能引用。
- *ParamType* 既非指针也非引用。

这么一来，我们就有了三种型别推导场景进行分情况考察。在对它们逐一考察时，我们仍采用前述模板和调用的一般形式。

```
template<typename T>
void f(ParamType param);

f(expr);                // 从 expr 来推导 T 和 ParamType 的型别
```

情况 1：*ParamType* 是个指针或引用，但不是个万能引用

最简单的莫过于当 *ParamType* 是个指针或引用，但不是万能引用的情形了。在这种情况下，型别推导会这样运作：

1. 若 *expr* 具有引用型别，先将引用部分忽略。
2. 尔后，对 *expr* 的型别和 *ParamType* 的型别执行模式匹配，来决定 T 的型别。

例如，我们的模式如下：

```
template<typename T>
void f(T& param);               // param 是个引用
```

又声明了下列变量：

```
int x = 27;                     // x 的型别是 int
const int cx = x;               // cx 的型别是 const int
const int& rx = x;              // rx 是 x 的型别为 const int 的引用
```

在各次调用中，对 param 和 T 的型别推导结果如下：

```
f(x);       // T 的型别是 int，param 的型别是 int&

f(cx);      // T 的型别是 const int，param 的型别是 const int&

f(rx);      // T 的型别是 const int，param 的型别是 const int&
```

在第二个以及第三个调用语句中，请注意，由于 cx 和 rx 的值都被指明为 const，所以 T 的型别被推导为 const int，从而形参的型别就成了 const int&。这一点对于调用者来说至关重要。当人们向引用型别的形参传入 const 对象时，他们期望该对象保

持其不可修改的属性，也就是说，期望该形参成为 const 的引用型别。这也是为何向持有 T& 型别的模板传入 const 对象是安全的：该对象的常量性（constness）会成为 T 的型别推导结果的组成部分。

在第三个调用中，请注意，即使 rx 具有引用型别，T 也并未被推导成一个引用。原因在于，rx 的引用性（reference-ness）会在型别推导过程中被忽略。

尽量上述调用语句示例演示的都是左值引用形参，但是右值引用形参的型别推导运作方式是完全相同的。当然，传给右值引用形参的，只能是右值引用实参，但这个限制和型别推导无关。

如果我们将形参型别从 T& 改为 const T&，结果会有一点变化，但这些变化并没有什么出人意料之处。cx 和 rx 的常量性仍然得到了满足，但是由于我们现在会假定 param 具有 const 引用型别，T 的型别推导结果中包含 const 也就没有必要了。

```
template<typename T>
void f(const T& param);        // param 现在是个 const 引用了

int x = 27;                    // 同前
const int cx = x;              // 同前
const int& rx = x;             // 同前

f(x);                          // T 的型别是 int，param 的型别是 const int&

f(cx);                         // T 的型别是 int，param 的型别是 const int&

f(rx);                         // T 的型别是 int，param 的型别是 const int&
```

一如前例，rx 的引用性在型别推导过程中是被忽略的。

如果 param 是个指针（或指涉到 const 对象的指针）而非引用，运作方式本质上并无不同：

```
template<typename T>
void f(T* param);              // param 现在是个指针了

int x = 27;                    // 同前
const int *px = &x;            // px 是指涉到 x 的指针，型别为 const int

f(&x);                         // T 的型别是 int，param 的型别是 int*

f(px);                         // T 的型别是 const int，param 的型别是 const int*
```

读到这里，也许你已经开始呵欠连连，脑袋发沉。因为 C++ 的型别推导规则对于引用和指针形参的运作方式是如此自然，还要把它们一一写下真是无聊。一切都那么明显！这正符合你对于型别推导系统的期望。

情况 2：*ParamType* 是个万能引用

对于持有万能引用形参的模板而言，规则就不那么显明了。此类形参的声明方式类似右值引用（即在函数模板中持有型别形参 T 时，万能引用的声明型别写作 T&&），但是当传入的实参是左值时，其表现会有所不同。完整的过程将在条款 24 中揭示，但此处先给出一个大纲：

- 如果 *expr* 是个左值，T 和 *ParamType* 都会被推导为左值引用。这个结果具有双重的奇特之处：首先，这是在模板型别推导中，T 被推导为引用型别的唯一情形。其次，尽管在声明时使用的是右值引用语法，它的型别推导结果却是左值引用。
- 如果 *expr* 是个右值，则应用“常规”（即情况 1 中的）规则。

例如：

```
template<typename T>
void f(T&& param);              // param 现在是个万能引用

int x = 27;                     // 同前
const int cx = x;               // 同前
const int& rx = x;              // 同前

f(x);                           // x 是个左值，所以 T 的型别是 int&，param 的型别也是 int&

f(cx);                          // cx 是个左值，所以 T 的型别是 const int&，
                                // param 的型别也是 const int&

f(rx);                          // rx 是个左值，所以 T 的型别是 const int&，
                                // param 的型别也是 const int&

f(27);                          // 27 是个右值，所以 T 的型别是 int，
                                // 这么一来，param 的型别就成了 int&&
```

条款 24 详尽解释了为何上述例子会产生这样的结果。关键之处在于，万能引用形参的型别推导规则不同于左值引用和右值引用形参。具体地，当遇到万能引用时，型别推导规则会区分实参是左值还是右值。而非万能引用是从来不会作这样的区分的。

情况 3：*ParamType* 既非指针也非引用

当 *ParamType* 既非指针也非引用时，我们面对的就是所谓按值传递了：

```
template<typename T>
void f(T param);            // param 现在是按值传递
```

这意味着，无论传入的是什么，param 都会是它的一个副本，也即一个全新对象。param 会是个全新对象这一事实促成了如何从 *expr* 推导出 T 的型别的规则：

- 一如之前，若 *expr* 具有引用型别，则忽略其引用部分。
- 忽略 *expr* 的引用性之后，若 *expr* 是个 const 对象，也忽略之。若其是个 volatile 对象，同忽略之（volatile 对象不常用，它们一般仅用于实现设备驱动程序。欲知详情，参见条款 40）。

所以，

```
int x = 27;           // 同前
const int cx = x;     // 同前
const int& rx = x;    // 同前

f(x);                 // T 和 param 的型别都是 int

f(cx);                // T 和 param 的型别还都是 int

f(rx);                // T 和 param 的型别仍都是 int
```

请注意，即使 cx 和 rx 代表 const 值，param 仍然不具有 const 型别。这是合理的。param 是个完全独立于 cx 和 rx 存在的对象——是 cx 和 rx 的一个**副本**。从而 cx 和 rx 不可修改这一事实并不能说明 param 是否可以修改。正是由于这一原因，*expr* 的常量性以及挥发性（volatileness，若有）可以在推导 param 的型别时加以忽略：仅仅由于 *expr* 不可修改，并不能断定其副本也不可修改。

需要重点说明的是，const（和 volatile）仅会在按值形参处被忽略。正如此前所见，若形参是 const 的引用或指针，*expr* 的常量性会在型别推导过程中加以保留。但是，考虑这种情况：*expr* 是个指涉到 const 对象的 const 指针，且 *expr* 按值传给 param：

```
template<typename T>
void f(T param);              // param 仍按值传递

const char* const ptr =       // ptr 是个指涉到 const 对象的 const 指针
"Fun with pointers";

f(ptr);                       // 传递型别为 const char * const 的实参
```

这里，位于星号右侧的 const 将 ptr 声明为 cost：ptr 不可以指涉到其他内存位置，也不可被置为 null[位于星号左侧的 const 则将 ptr 指涉到的对象（那个字符串）为 const，即该字符串不可修改]。可 ptr 被传递给 f 时，这个指针本身将会按比特复制给 param。换言之，ptr 这个指针自己会被按值传递。依照按值传递形参的型别推导规则，ptr 的常量性会被忽略，param 的型别会被推导为 const char *，即一个可修改的、指涉到一个 const 字符串的指针。在型别推导的过程中，ptr 指涉到的对象的常量性会得到保留，但其自身的常量性则会在以复制方式创建新指针 param 的过程中被忽略。

数组实参

以上已经基本讨论完模板型别推导的主流情况，但还有一个边缘情况值得了解。这种情况是：数组型别有别于指针型别，尽管有时它们看起来可以互换。形成这种假象的主要原因是，在很多语境下，数组会退化成指涉到其首元素的指针。下面这段代码之所以能够通过编译，就是因为这种退化机制在发挥作用：

```
const char name[] = "J. P. Briggs";  // name 的型别是 const char[13]

const char * ptrToName = name;          // 数组退化成指针
```

这里，型别为 `const char *` 的指针 `ptrToName` 是通过 `name` 来初始化的，而后者的型别是 `const char[13]`。这两个型别（`const char *` 和 `const char[13]`）并不统一，但是因为数组到指针的退化规则地存在，上述代码能够通过编译。

但当一个数组传递给持有按值形参的模板时，又会怎样呢？

```
template<typename T>
void f(T param);       // 持有按值形参的模板
f(name);               // T 和 param 的型别会被推导成什么呢？
```

我们先是观察到，并没有任何的函数形参具有数组型别。没错，下面的语法是合法的：

```
void myFunc(int param[]);
```

但是既然数组声明可以按照指针声明方式加以处理，那就意味着 `myFunc` 可以等价地声明如下：

```
void myFunc(int* param);
```

这种数组和指针形参的等价性，是作为 C++ 基础的 C 根源遗迹，它使得“数组和指针型别是一回事”这一假象愈加扑朔迷离。

由于数组形参声明会按照它们好像是指针形参那样加以处理，按值传递给函数模板的数组型别将被推导成指针型别。也就是说，在模板 `f` 的调用中，其型别形参 `T` 会被推导成 `const char *`：

```
f(name);               // name 是个数组，但 T 的型别却被推导成 const char *
```

难点来了。尽管函数无法声明真正的数组型别的形参，它们却能够将形参声明成数组的引用！所以，如果我们修改模板 `f`，指定按引用方式传递其实参，

```
template<typename T>
void f(T& param);              // 按引用方式传递形参的模板
```

然后，向其传递一个数组，

```
f(name);                    // 向 f 传递一个数组
```

在这种情况下，T 的型别会被推导成实际的数组型别！这个型别中会包含数组尺寸，在本例中，T 的型别推导结果是 const char [13]，而 f 的形参（该数组的一个引用）型别则被推导为 const char (&)[13]。没错，语法看起来又臭又长，但了解到这个程度却会让你在面对极少数较真的人时，挣得好大一个面子。

有意思的是，可以利用声明数组引用这一能力创造出一个模板，用来推导出数组含有的元素个数：

```
// 以编译期常量形式返回数组尺寸
// （该数组形参未起名字，因为我们只关心其含有的元素个数）
template<typename T, std::size_t N>                    // 关于 constexpr
constexpr std::size_t arraySize(T (&)[N]) noexcept     // 和 noexcept
{                                                      // 的说明，
   return N;                                           // 请参考下文
}
```

如条款 15 所释，将该函数声明为 constexpr，能够使得其返回值在编译期就可用。从而就可以在声明一个数组时，指定其尺寸和另一数组相同，而后者的尺寸则从花括号初始化式（braced initializer）计算得出：

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };     // keyVals 含有 7 个元素

int mappedVals[arraySize(keyVals)];             // mappedVals 被指定与之相同
```

当然，身为一名现代 C++ 程序员，相对于内建数组，你自然会优先选用 std::array：

```
std::array<int, arraySize(keyVals)> mappedVals;  // mappedVals 也指定为 7 个元素
```

至于为何 arraySize 被声明为 noexcept，这为了帮助编译器生成更好的目标码。欲知详情，参见条款 14。

函数实参

数组并非 C++ 中唯一可以退化为指针之物。函数型别也同样会退化成函数指针，并且我们针对数组型别推导的一切讨论都适用于函数及其向函数指针的退化。所以结果如下：

```
void someFunc(int, double);          // someFunc 是个函数，
                                     // 其型别为 void(int, double)
```

```
template<typename T>
void f1(T param);              // 在 f1 中，param 按值传递

template<typename T>
void f2(T& param);             // 在 f1 中，param 按引用传递

f1(someFunc);                  // param 被推导为函数指针，
                               // 具体型别是 void (*)(int, double)

f2(someFunc);                  // param 被推导为函数引用，
                               // 具体型别是 void (&)(int, double)
```

在实践中，这些型别推导结果和前面讲过的没什么不一样。不过，如果你打算了解数组向指针的退化，那么就顺便也了解一下函数向针对的退化好了。

关于和 auto 相关的模板型别推导规则，内容就这么多了。在本条款一开始，我就强调过这些规则相当单刀直入，而事实上在大部分情况下它们也确实如此。特殊情况仅仅在于，在对万能引用进行推导时传入左值的情况下，会有一点小小的波澜。再有，就是数组和函数向指针退化的规则会将这小小的波澜搅成更大的漩涡。有时，你会想直接揪住编译器并命令道："告诉我，你推导出来的究竟是什么型别！"如果真是如此，请翻到条款 4，因为它专门讨论如何忽悠编译器去为你做这件事。

要点速记

- 在模板型别推导过程中，具有引用型别的实参会被当成非引用型别来处理。换言之，其引用性会被忽略。
- 对万能引用形参进行推导时，左值实参会进行特殊处理。
- 对按值传递的形参进行推导时，若实参型别中带有 const 或 volatile 饰词，则它们还是会被当作不带 const 或 volatile 饰词的型别来处理。
- 在模板型别推导过程中，数组或函数型别的实参会退化成对应的指针，除非它们被用来初始化引用。

条款 2：理解 auto 型别推导

如果你已经读完了有关模板型别推导的条款 1，那么你已经基本上了解有关 auto 型别推导的一切必要知识了。因为，除了一个奇妙的例外情况以外，auto 型别推导就是模板型别推导。这怎么可能？和模板型别推导打交道的是模板、函数和形参，而 auto 和它们秋毫无犯。

事实固然如此，但不影响上面的结论成立。在模板型别推导和 auto 型别推导可以建立起一一映射，它们之间也确实存在双向的算法变换。

条款 1 中，我们用来解释模板型别推导的函数模板形如：

```
template<typename T>
void f(ParamType param);
```

而一次调用则形如：

```
f(expr);      // 以某表达式调用 f
```

在 f 的调用语句中，编译器会利用 *expr* 来推导 T 和 *ParamType* 的型别。

当某变量采用 auto 来声明时，auto 就扮演了模板中的 T 这个角色，而变量的型别饰词则扮演的是 *ParamType* 的角色。说来话长，不如看看下面的例子：

```
auto x = 27;
```

在这个例子中，x 的型别饰词就是 auto 自身。而在下例的声明中，

```
const auto cx = x;
```

型别饰词成了 const auto，再看下例，

```
const auto& rx = x;
```

型别饰词又成了 const auto&。在上述这些例子中，若要推导 x、cx 和 rx 的型别，编译器的行为就仿佛对应于每个声明生成了一个模板和一次使用对应的初始化表达式针对该模板的调用似的：

```
template<typename T>            // 为推导 x 的型别而生成的概念性模板[译注 1]
void func_for_x(T param);

func_for_x(27);                 // 概念性调用语句：推导得出的 param 的型别
                                // 就是 x 的型别

template<typename T>            // 为推导 cx 的型别而生成的概念性模板
void func_for_cx(const T param);

func_for_cx(x);                 // 概念性调用语句：推导得出的 param 的型别
                                // 就是 cx 的型别

template<typename T>            // 为推导 rx 的型别而生成的概念性模板
void func_for_rx(const T& param);
```

译注 1：所谓概念性模板、概念性语句，意指仅仅是概念上等价，而并不是说编译器真的生成了该模板和语句。

```
func_for_rx(x);                    // 概念性调用语句：推导得出的 param 的型别
                                   // 就是 rx 的型别
```

如我所言，为 auto 推导型别，除了在一个例外情况下（马上就要讨论这种情况），和为模板推导型别一模一样。

条款 1 根据 *ParamType* 的特征，即一般形式的函数模板中 param 的型别饰词，将模板型别推导分成三种情况。而在采用 auto 进行变量声明中，型别饰词取代了 *ParamType*，所以也存在三种情况：

- 情况 1：型别饰词是指针或引用，但不是万能引用。
- 情况 2：型别饰词是万能引用。
- 情况 3：型别饰词既非指针也非引用。

情况 1 和情况 3，我们已经在前面的例子中见过：

```
auto x = 27;           // 情况 3 (x 既非指针也非引用 )

const auto cx = x;     // 情况 3 (cx 同样既非指针也非引用 )

const auto& rx = x;    // 情况 1 (rx 是个引用，但不是万能引用 )
```

情况 2 的运作也和你的期望一致：

```
auto&& uref1 = x;      // x 的型别是 int，且是左值，
                       // 所以 uref1 的型别是 int&

auto&& uref2 = cx;     // cx 的型别是 const int，且是左值，
                       // 所以 uref2 的型别是 const int&

auto&& uref3 = 27;     // 27 的型别是 int，且是右值，
                       // 所以 uref3 的型别是 int&&
```

条款 1 以一段数组和函数名字如何在非引用型别饰词的前提下退化成指针的讨论收官。同样的结论也适用 auto 型别推导：

```
const char name[] =            // name 的型别是 const char[13]
 "R. N. Briggs" ;

auto arr1 = name;              // arr1 的型别是 const char*

auto& arr2 = name;             // arr2 的型别是 const char (&)[13]

void someFunc(int, double);    // someFunc 是个函数，型别是 void(int, double)

auto func1 = someFunc;         // func1 的型别是 void (*)(int, double)
```

```
auto& func2 = someFunc;                    // func2 的型别是 void (&)(int, double)
```

如你所见，auto 型别推导和模板型别推导的运作是类似的，它们不过是硬币的正反两面。

只有一处不同。我们先观察一下，若要声明一个 int，并将其初始化为值 27，C++98 中有两种可选语法：

```
int x1 = 27;
int x2(27);
```

而 C++11 为了支持统一初始化（uniform initialization），增加了下面的语法选项：

```
int x3 = { 27 };
int x4{ 27 };
```

共计四种语法，然而结果却殊途同归：得到一个值为 27 的 int。

但正同条款 5 所述，采用 auto 声明变量，相比采用固定型别声明变量更具优势，所以将上面变量声明中的 int 替换成 auto 好了。直截了当的文本替换就得到了下面这段代码：

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

这些声明都能通过编译，但结果却与一开始并不全部相同。前面两个语句确实仍然声明了一个型别为 int、值为 27 的变量，而后面两个语句，却声明了这么一个变量，其型别为 std::initializer_list<int>，且含有单个值为 27 的元素！

```
auto x1 = 27;          // 型别是 int，值是 27

auto x2(27);           // 同上

auto x3 = { 27 };      // 型别是 std::initializer_list<int>，值是 { 27 }

auto x4{ 27 };         // 同上
```

这是有关 auto 的一条特殊的型别推导规则所致。当用于 auto 声明变量的初始化表达式是使用大括号括起时，推导所得的型别就属于 std::initializer_list。这么一来，如果型别推导失败（例如，大括号里的值型别不一），则代码就通不过编译：

```
auto x5 = { 1, 2, 3.0 };       // 错误！
                               // 推导不出 std::initializer_list<T> 中的 T
```

这是有关 auto 的一条特殊的型别推导规则所致。当用于 auto 声明变量的初始化表达式是使用大括号括起时，推导所得的型别就属于 std::initializer_list。这么一来，如果型别推导失败（例如，大括号里的值型别不一），则代码就通不过编译：正如注释所指出的那样，这种情况下型别推导会失败，但是重点在于，要意识到这里发生了两种型别推导。第一种源于 auto 的使用：x5 的型别需要推导。而 x5 的初始化表达式是用大括号括起的，所以 x5 必须推导为一个 std::initializer_list。但 std::initializer_list 是个模板，它要根据某个型别 T 产生实例型别 std::initializer_list<T>，而这就意味着 T 的型别也必须被推导出来。而这后一次推导就落入了第二种型别推导，即模板型别推导的范畴了。在本例中，该推导会失败，因为大括号括起的初始化表达式中的值型别不一。

对于大括号初始化表达式的处理方式，是 auto 型别推导和模板型别推导的唯一不同之处。当采用 auto 声明的变量使用大括号初始化表达式进行初始化时，推导所得的型别是 std::initializer_list 的一个实例型别。但是，如果向对应的模板传入一个同样的初始化表达式，型别推导就会失败，代码将不能通过编译：

```
auto x = { 11, 23, 9 };          // x 的型别是 std::initializer_list<int>

template<typename T>             // 带有形参的模板
void f(T param);                 // 与 x 的声明等价的声明式

f({ 11, 23, 9 });                // 错误！无法推导 T 的型别
```

所以，auto 和模板型别推导真正的唯一区别在于，auto 会假定用大括号括起的初始化表达式代表一个 std::initializer_list，但模板型别推导却不会。

不过，你若指定该模板中 param 的为 std::initializer_list<T>，则在 T 的型别未知时，模板型别推导机制会推导出 T 应有的型别：

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });                // T 的型别推导为 int
                                 // 从而 initList 的型别 std::initializer_list<int>
```

你可能会奇怪，为什么 auto 型别推导为大括号括起的初始化表达式设立了这么一条特殊规则，而模板型别推导却没有。我自己也奇怪。实话实说，对此我找不到一个有说服力的解释。但规则就是规则，它意味着当你采用 auto 来声明变量，而且初始化表达式是用大括号括起的话，推导得到的型别总是 std::initializer_list。极其重要的是，如果你想要拥抱统一初始化的哲学——就是说，会自然而然地把初始化值括在大括号里的话，那么务请牢记这条规则。C++11 程序设计中的一个经典错误，就是明明把变

量声明成了一个 std::initializer_list，其实本意却并非如此。正是由于这个陷阱的存在，很多程序员都只在必要的时候才会使用大括号括起的初始化表达式（至于什么是必要的时候，请参见条款 7）。

关于 C++11，上面所说的就是全部了。但是关于 C++14，还有别的话要说。C++14 允许使用 auto 来说明函数返回值需要推导（参见条款 3），而且 C++14 中的 lambda 式也会在形参声明中用到 auto。然而，这些 auto 用法是在使用模板型别推导而非 auto 型别推导。所以，带有 auto 返回值的函数若要返回一个大括号括起的初始化表达式，是通不过编译的：

```
auto createInitList()
{
   return { 1, 2, 3 };          // 错误：无法为 { 1, 2, 3 } 完成型别推导
}
```

同样地，用 auto 来指定 C++14 中 lambda 式的形参型别时，也不能使用大括号括起的初始化表达式：

```
std::vector<int> v;
…

auto resetV =
  [&v](const auto& newValue) { v = newValue; }; // C++14

…

resetV({ 1, 2, 3 });                          // 错误！无法为 { 1, 2, 3 } 完成型别推导
```

要点速记

- 在一般情况下，auto 型别推导和模板型别推导是一模一样的，但是 auto 型别推导会假定用大括号括起的初始化表达式代表一个 std::initializer_list，但模板型别推导却不会。
- 在函数返回值或 lambda 式的形参中使用 auto，意思是使用模板型别推导而非 auto 型别推导。

条款 3：理解 decltype

说起 decltype，这是个古灵精怪的东西。对于给定的名字或表达式，decltype 能告诉你该名字或表达式的型别。一般来说，它告诉你的结果和你预测的是一样的。不过，

偶尔它也会给出某个结果，让你抓耳挠腮，不得不去参考手册或在线 FAQ 页面求得一些启发。

先从一般案例讲起——就是那些不会引发意外的案例。与模板和 auto 的型别推导过程（参见条款 1 和条款 2）相反，decltype 一般只会鹦鹉学舌，返回给定的名字或表达式的确切型别而已：

```
const int i = 0;                    // decltype(i) 是 const int

bool f(const Widget& w);            // decltype(w) 是 const Widget&
                                    // decltype(f) 是 bool(const Widget&)

struct Point {
  int x, y;                         // decltype(Point::x) 是 int
};                                  // decltype(Point::y) 是 int

Widget w;                           // decltype(w) 是 Widget

if (f(w)) …                         // decltype(f(w)) 是 bool

template<typename T>                // std::vector 的简化版
class vector {
public:
  …
  T& operator[](std::size_t index);
  …
};

vector<int> v;                      // decltype(v) 是 vector<int>
…
if (v[0] == 0) …                    // decltype(v[0]) 是 int&
```

看到了吗？没有任何意外吧。

C++11 中，decltype 的主要用途大概就在于声明那些返回值型别依赖于形参型别的函数模板。举个例子，假设我们想要撰写一个函数，其形参中包括一个容器，支持方括号下标语法（即“[]”）和一个下标，并会在返回下标操作结果前进行用户验证。函数的返回值型别须与下标操作结果的返回值型别相同。

一般来说，含有型别 T 的对象的容器，其 operator[] 会返回 T&。std::deque 就属于这种情况，而 std::vector 也几乎总是属于这种情况。只有 std::vector<bool> 对应的 operator[] 并不返回 bool&，而返回一个全新对象。至于这样处理的原因和具体处理结果，在条款 6 中会有详细探讨。在此时此地，重要的在于容器的 operator[] 的返回型别取决于该容器本身。

而 decltype 使得这样的意思表达简单易行。下面是我们撰写该模板的首次尝试，其中

演示了使用 decltype 来计算返回值型别。这个模板还有改进空间，但我们后面再议此事：

```
template<typename Container, typename Index>    // 能运作，但是
auto authAndAccess(Container& c, Index i)       // 亟须改进
  -> decltype(c[i])
{
  authenticateUser();
  return c[i];
}
```

在函数名字之前使用的那个 auto 和型别推导没有任何关系。它只为说明这里使用了 C++11 中的**返回值型别尾序语法**（trailing return type syntax），即该函数的返回值型别将在形参列表之后（在“->”之后）。尾序返回值的好处在于，在指定返回值型别时可以使用函数形参。比如，在 authAndAccess 中，我们在指定返回值型别时就可以使用 c 和 i。如果我们还是使用传统的返回值型别先序语法，那 c 和 i 会由于还未声明，从而无法使用。

采用了这么一个声明形式以后，operator[] 返回值是什么型别，authAndAccess 的返回值就是什么型别，和我们期望的结果一致。

C++11 允许对单表达式的 lambda 式的返回值型别实施推导，而 C++14 则将这个允许范围扩张到了一切 lambda 式和一切函数，包括那些多表达式的。对于 authAndAccess 这种情况来说，这就意味着在 C++14 中可以去掉返回值型别尾序语法，而只保留前导 auto。在那样的声明形式中，auto 确实说明会发生型别推导。具体地说，它说明编译器会依据函数实现来实施函数返回值的型别推导：

```
template<typename Container, typename Index>    // C++14;
auto authAndAccess(Container& c, Index i)       // 不甚正确
{
  authenticateUser();
  return c[i];          // 返回值型别是根据 c[i] 推导出来
}
```

条款 2 解释说，编译器会对 auto 指定为返回型别的函数实现模板型别推导。而在上例中，这样就会留下隐患。一如前面讨论的那样，大多数含有型别 T 的对象的容器的 operator[] 会返回 T&，但是条款 1 解释说，模板型别推导过程中，初始化表达的引用性会被忽略。考虑一下，这会对客户代码产生怎样的影响：

```
std::deque<int> d;
...
authAndAccess(d, 5) = 10;       // 验证用户，并返回 d[5]，
                                // 然后将其赋值为 10；
                                // 这段代码无法通过编译
```

此处，d[5] 返回的是 int&，但是对 authAndAccess 的返回值实施 auto 型别推导将剥去引用饰词，这么一来返回值型别就成了 int。作为函数的返回值，该 int 是个右值，所以上述代码其实是尝试将 10 赋给一个右值 int。这在 C++ 中属于被禁止的行为，所以代码无法通过编译。

欲让 authAndAccess 如我们期望般运作，就要对其返回值实施 decltype 型别推导，即指定 authAndAccess 的返回值型别与表达式 c[i] 返回的型别完全一致。C++ 的监护人们，由于预见到在进行某些型别推导时需要采用 decltype 型别推导规则，在 C++14 中通过 decltype(auto) 饰词解决了这个问题。乍看上去自相矛盾（怎么会又是 decltype 又是 auto 呢），其实完全合情合理：auto 指定了欲实施推导的型别，而推导过程中采用的是 decltype 的规则。总而言之，我们可以这样撰写 authAndAccess：

```
template<typename Container, typename Index>    // C++14：能够运作，
decltype(auto)                                  // 但仍亟须改进
authAndAccess(Container& c, Index i)
{
  authenticateUser();
  return c[i];
}
```

现在，authAndAccess 的返回值型别真的和 c[i] 返回的型别一致了。具体地说，一般情况下 c[i] 返回 T&，authAndAccess 也会返回 T&。而对于少见情况，c[i] 返回一个对象型别，authAndAccess 也会亦步亦趋地返回对象型别。

decltype(auto) 并不限于在函数返回值型别处使用。在变量声明的场合上，若你也想在初始化表达式处应用 decltype 型别推导规则，也可以照样便宜行事：

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;                    // auto 型别推导：
                                        // myWidget1 的型别是 Widget

decltype(auto) myWidget2 = cw;          // decltype 型别推导：
                                        // myWidget2 的型别是 const Widget&
```

不过，我知道现在还有两个烦恼萦绕在你的脑际。一个是我前面说过的对 authAndAccess 的改进，这个我还一直憋着没说呢。现在，就说一说这个问题。

再看一遍 C++14 版本的 authAndAccess：

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

容器的传递方式是对非常量的左值引用（lvalue-reference-to-non-const），因为返回该

容器的某个元素的引用，就意味着允许客户对容器进行修改。不过这也意味着无法向该函数传递右值容器。右值是不能绑定到左值引用的（除非是对常量的左值引用，与本例情况不符）。

必须承认，向 authAndAccess 传递右值容器属于罕见情况。一个右值容器，作为一个临时对象，一般而言会在包含了调用 authAndAccess 的语句结束处被析构，而这就是说，该容器中某个元素的引用（这是 authAndAccess 一般情况下会返回的）会在创建它的那个语句结束时被置于空悬状态。但即使如此，向 authAndAccess 传递一个临时对象仍然可能是合理行为。客户可能就是想要制作该临时容量的某元素的一个副本，请看下例：

```
std::deque<std::string> makeStringDeque();      // 工厂函数

// 制作 makeStringDeque 返回的 deque 的第 5 个元素的副本
auto s = authAndAccess(makeStringDeque(), 5);
```

容器的传递方式是对非常量的左值引用（lvalue-reference-to-non-const），因为返回该容器的某个元素的引用，就意味着允许客户对容器进行修改。不过这也意味着无法向该函数传递右值容器。右值是不能绑定到左值引用的（除非是对常量的左值引用，与本例情况不符）。

如果要支持这种用法，就得修订 authAndAccess 的声明，以同时接受左值和右值。重载是个办法（一个重载版本声明一个左值引用形参，另一个重载版本声明一个右值引用形参），但这么一来就需要维护两个函数。避免这个后果的一个方法是让 authAndAccess 采用一种既能够绑定到左值也能够绑定到右值的引用形参，而条款 24 给出了解释，说明这正是万能引用大显身手之处。这么一来，authAndAccess 就可以这样声明：

```
template<typename Container, typename Index>    // c 现在是个
decltype(auto) authAndAccess(Container&& c,     // 万能引用了
                             Index i);
```

在本模板中，我们对于操作的容器型别并不知情，同时对下标对象型别也一样不知情。对未知型别的对象采用按值传递有着诸多风险：非必要的复制操作带来的性能隐患、对象截切（slicing）问题带来的行为异常（参见条款 41），还有同行的嘲笑等，但是在容器下标这个特定问题上，遵循标准库中给出的下标值示例（例如 std::string、std::vector 和 std::deque 的 operator[]）应该是合理的，所以这里坚持使用了按值传递。

不过，我们需要更新该模板的实现，以使它与条款 25 所教导我们的内容相符：对万能引用要应用 std::forward：

```
template<typename Container, typename Index>    // C++14 最终版
decltype(auto)
authAndAccess(Container&& c, Index i)
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

这个版本可以实现我们想要的一切，但它要求使用 C++14 编译器。如果你没有，就得使用本模板的 C++11 版本。它和 C++14 版本几乎一样，只是你需要自行指定返回值型别：

```
template<typename Container, typename Index>    // C++11 最终版
auto
authAndAccess(Container&& c, Index i)
->  decltype(std::forward<Container>(c)[i])
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

另一个你可能会有的烦恼就是我在本条款一开始指出的，decltype 几乎总会生成你期望的型别，但它偶尔也会吓到你。实在地讲，除非你是重度的库实现者，你一般不太会遭遇这些规则的例外情况。

想要彻底理解 decltype 的行为，就需要熟悉若干特殊情况。其中的大部分都太过艰涩，在本书中也不适合完全展开，但是只看一例则可以更深入地理解 decltype 及其使用。

将 decltype 应用于一个名字之上，就会得出该名字的声明型别。名字其实是左值表达式，但如果仅有一个名字，decltype 的行为保持不变。不过，如果是比仅有名字更复杂的左值表达式的话，dectype 就保证得出的型别总是左值引用。换言之，只要一个左值表达式不仅是一个型别为 T 的名字，它就得出一个 T& 型别。这种行为一般而言没有什么影响，因为绝大多数左值表达式都自带一个左值引用饰词。例如，返回左值的函数总是返回左值引用。

但这种行为还是会导致一个值得注意的后果，请看表达式：

```
int x = 0;
```

其中 x 是一个变量名字，所以 decltype(x) 的结果是 int。但是如果把名字 x 放入一对小括号中，就得到了一个比仅有名字更复杂的表达式“(x)”。作为一个名字，x 是个左值，而在 C++ 的定义中，表达式 (x) 也是一个左值，所以 decltype((x)) 的结果就成了 int&。仅仅是把一个名字放入一对小括号，就改变了 decltype 的推导结果！

在 C++11 中，知道了这个，其实也就是满足一下猎奇心理而已，但如果和 C++14 对 decltype(auto) 的支持这么联合一下，一个看似无关紧要的返回值写法上的小改动，就会影响到函数的型别推导结果：

```
decltype(auto) f1()
{
  int x = 0;
  …
  return x;            // decltype(x) 是 int，所以 f1 返回的是 int
}

decltype(auto) f2()
{
  int x = 0;
  …
  return (x);          // decltype((x)) 是 int&，所以 f2 返回的是 int&
}
```

请注意，问题不仅仅在于 f2 和 f1 有着返回值型别的不同，更重要的是 f2 返回了一个局部变量的引用！这种代码会把你送上未定义行为的快车——你永远不想乘坐的那种列车。

主要的教训在于，使用 decltype(auto) 时需要极其小心翼翼。看似是用以推导型别表达式的写法这样无关紧要的细节，却影响着 decltype(auto) 得出的结果。为了保证推导所得的型别和你期望的一致，请使用条款 4 讲述的技术。

同时，请勿因此丢掉了大局观。没错，decltype（无论是单独使用，还是和 auto 配合使用）时不时地会得出意外的型别推导结果，但那说到底并非正常情形。在正常情形下，decltype 产生的型别就和你期望的一致。

以上的说法在 decltype 应用于名字时尤其成立，因为在那种情况下，decltype 的行为可谓名副其实：它得出的就是该名字的声明型别（declared type）。

要点速记

- 绝大多数情况下，decltype 会得出变量或表达式的型别而不作任何修改。
- 对于型别为 T 的左值表达式，除非该表达式仅有一个名字，decltype 总是得出型别 T&。
- C++14 支持 decltype(auto)，和 auto 一样，它会从其初始化表达式出发来推导型别，但是它的型别推导使用的是 decltype 的规则。

条款 4：掌握查看型别推导结果的方法

采用何种工具来查看型别推导结果，取决于你在软件开发过程的哪个阶段需要该信息。我们将研究三个可能的阶段：撰写代码阶段、编译阶段和运行时阶段。

IDE 编辑器

IDE 中的代码编辑器通常会在你将鼠标指针悬停至某个程序实体，如变量、形参、函数等时，显示出该实体的型别。例如以下这段代码：

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

IDE 编辑器很可能会显示出，x 的型别推导结果是 int，而 y 则是 const int*。

要让这种方法奏效，代码就多多少少要处于一种可编译的状态。因为让 IDE 提供此类信息的工作原理是让 C++ 编译器（或至少也是其前端）在 IDE 内执行一轮。如果该编译器不能在分析你的代码时得到足够的有用信息，自然也就无法显示出推导出了何种型别。

对于像 int 这样的平凡型别，从 IDE 能得到的信息大体良好。不过你很快就会看到，一旦较为复杂的型别现身，IDE 显示的信息就不太有用了。

编译器诊断信息

想要让编译器显示其推导出的型别，一条有效的途径是使用该型别导致某些编译错误。而报告错误的消息几乎肯定会提及导致该错误的型别。

例如，我们想要查看上例中的 x 和 y 推导而得的型别。我们先声明一个类模板，但不去定义它。像这样的代码就能够很好地完成任务：

```
template<typename T>  // 只声明 TD 而不定义；
class TD;             // TD 是“型别显示类”(Type Displayer) 的缩写
```

只要试图具现该模板，就会诱发一个错误消息，原因是找不到具现模板所需要的定义。如果想查看 x 和 y 的型别，且试用 x 和 y 的型别去具现 TD 即可：

```
TD<decltype(x)> xType;        // 诱发包括 x 和 y 的型别的错误消息
TD<decltype(y)> yType;
```

这里我把变量的名字取为“变量名字 +Type”的形式，因为这么一来错误消息就更有可能产生有助于我找到所追寻的信息。对于上述代码，我使用的编译器之一给出的诊断信息包含以下部分：

```
error: aggregate 'TD<int> xType' has incomplete type and
    cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type
and cannot be defined
```

另一个编译器给出了同样的信息，只是形式不同：

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

除了格式各异以外，只要采用了这种手段，所有我测试过的编译器都产生出了包含有用型别信息的错误消息。

运行时输出

使用 printf 来显示型别信息（我可没有教你用 printf 哟），这种方法只有到了运行期才能使用，却可以对于型别输出的格式提供完全的控制。困难之处在于为你关心的型别创立一种适于显示的字符表示。你可能会想，“这有什么难的，不就是 typeid 和 std::type_info::name 吗？”沿着这个思路下去，查看 x 和 y 的推导型别的任务就落实成了以下代码：

```
std::cout << typeid(x).name() << '\n';          // display types for
std::cout << typeid(y).name() << '\n';          // x and y
```

这种手段依赖于以下事实：针对某个对象，如 x 和 y 调用 typeid，就得到了一个 std::type_info 对象，而后者拥有一个成员函数 name。该函数产生一个代表型别的 C-style 的字符串（即一个 const char*）。

对于 std::type_info::name 的调用不保证返回任何有意义的内容，但是各家的实现都会在这个返回内容上做些文章，使得它有一些用处。但这用处的大小就参差不齐了。例如，GNU 和 Clang 的编译器会报告结果说，x 的型别是“i”，而 y 的型别是“PKi”。如果你了解一下背景，那就可以读懂这样的结果：在这些编译器的输出中，“i”就表示“int”，“PK”就表示“pointer to konst const”（指涉到常量的指针）（这两种编译都支持一种叫做 c++filt 的工具，可以解读这种做过“混淆”的型别）。微软的编译器产生的输出就不那么神秘：x 的型别是“int”，而 y 的型别是“int const *”。

由于针对 x 和 y 的报告结果都正确，你可能会觉得型别报告的问题已经大功告成。不过且慢，再看一个较为复杂的例子：

```
template<typename T>
void f(const T& param);                  // f是打算调用的函数模板

std::vector<Widget> createVec();         // 工厂函数

const auto vw = createVec();             // 使用工厂函数返回值初始化vw

if (!vw.empty()) {
  f(&vw[0]);                             // 调用f
  ...

}
```

这段代码涉及一个用户自定义型别（Widget）、一个STL容器（std::vector），还有一个auto变量（vw），在需要查看编译器进行型别推导的需求方面比较有代表性。例如，如果能了解一下究竟传递给模板型别形参T和f的函数形参param的是什么型别，那将是十分理想的。

在这个问题上放手一搏使用typeid，十分直截了当。只需添加一些代码来显示你想查看的型别即可：

```
template<typename T>
void f(const T& param)
{
  using std::cout;
  cout << "T =     " << typeid(T).name() << '\n';        // 显示T

  cout << "param = " << typeid(param).name() << '\n';    // 显示param的型别
  ...
}
```

GNU和Clang编译器产生的可执行文件会输出以下结果：

```
T =     PK6Widget
param = PK6Widget
```

我们已经知道，在这些编译器的输出中，PK的意思是“指涉到常量的指针”，所以唯一神秘的部分就是数字6了。其实这个数字的意思是紧随其后的类名（Widget）的字符数。所以这个编译器告诉我们，T和param的型别都是const Widget*。

微软编译器的结果也一样：

```
T =     class Widget const *
param = class Widget const *
```

三个彼此独立的编译器不约而同地给出了相同结果，这给人以信息正确的印象。不过，再仔细观察一下就会发现问题。在模板f中，param被声明成型别const T&。如果是这样，

T 和 param 居然型别一样，这不奇怪吗？如果 T 是 int，那 param 的型别就该是 const int& 才是，完全不应该是同一个型别。

很不幸的是，std::type_info::name 并不可靠。就拿上例来说，三种编译器报告的型别都不正确。犹有进者，这种不正确的结果是符合标准要求的，因为标准规格上说，std::type_info::name 中处理型别的方式就仿佛是向函数模板按值传递形参一样。这么一来，根据条款 1，若该型别是个引用型别，其引用性将被忽略；而如果移除了引用性以后，该型别还带有 const（或 volatile）饰词，则其常量性和挥发性同样会被移除。这就是为什么 param 的型别本该是 const Widget * const &，却被报告成 const Widget*：首先，该型别的引用性被移除，尔后返回的指针型别的常量性也被消灭。

同样不幸的是，IDE 显示的型别信息也不可靠，至少在实用性方面不可靠。同样是上面这个例子，有一个 IDE 编辑器把 T 的型别显示成这样（我没有做任何修饰）：

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

同一个编辑器把 param 的型别显示成这样：

```
const std::_Simple_types<...>::value_type *const &
```

这个没有 T 的型别看起来那么吓人，但是中间的“...”意义你不会懂，直到有一天你意识到原来 IDE 通过它来表达的意思是“此处略去属于 T 型别的一部分的任何东西”。反正不管怎么样，在处理这样的代码方面，你的开发环境总是做得更出色。

如果你天生觉得依靠库比依靠运气来得好，那你会倍感欣慰，因为 std::type_info::name 和 IDE 跌倒的地方，Boost 的 TypeIndex 库（常写作 Boost.TypeIndex）的设计却能站稳。虽然说该库不是标准 C++ 的一部分，但 IDE 和像 TD 这样的模板也不是。Boost 库（可以从 boost.com 获取）的好处还不止于此，它跨平台、开源，并且其许可即使在有着最偏执的法务的团队也能照用不误。这么一来，Boost 库和那些依赖标准库的代码有几乎同样的可移植性。

以下介绍我们的函数 f 是如何在使用 Boost.TypeIndex 的条件下产生精确的型别信息的：

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
```

```
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // 显示T的型别
  cout << "T =          "
       << type_id_with_cvr<T>().pretty_name()
       << '\n';

  // 显示param的型别
  cout << "param = "
       <<  type_id_with_cvr<decltype(param)>().pretty_name()
       << '\n';
  ...
}
```

这种方法的工作原理是函数模板 boost::typeindex::type_id_with_cvr 接受一个型别实参（我们想要获取信息的型别），而且不会移除 const、volatile 和引用饰词（这也是该模板的名字中为什么含有 with_cvr 字样），该函数模板返回一个 boost::typeindex::type_index 对象，它利用成员函数 pretty_name 产生一个包含人类可读的型别表示的 std::string。

有了这样一个 f 的实现，再重新考量那个使用 typeid 时产生了错误的 param 型别信息的调用：

```
std::vector<Widget> createVec();          // 工厂函数

const auto vw = createVec();              // 使用工厂函数返回值初始化 vw

if (!vw.empty()) {
  f(&vw[0]);                              // 调用 f
  ...
}
```

在使用 GNU 和 Clang 编译器的情况下，Boost.TypeIndex 产生了以下的（精确）输出：

```
T =     Widget const*
param = Widget const* const&
```

微软编译器取得的结果本质上完全相同：

```
T =     class Widget const*
param = class Widget const* const&
```

这种近似稳定性固然美好，但是重要之处在于，请记住无论是 IDE 编辑器、编译器错误消息，还是像 Boost.TypeIndex 这样的库都仅仅是你弄明白你的编译器推导所得型别的辅助工具。它们都十分有用，但是说到底，理解条款 1~条款 3 的型别推导知识这一点无可替代。

要点速记

- 利用 IDE 编辑器、编译器错误消息和 Boost.TypeIndex 库常常能够查看到推导而得的型别。
- 有些工具产生的结果可能会无用，或者不准确。所以，理解 C++ 型别推导规则是必要的。

第 2 章

auto

如果单从概念上看，auto 真是简单到不能再简单，然而细究起来里面却大有文章。使用 auto 可以少打一些字，没错，但好处不止这些：它还能阻止那些由于手动指定型别带来的潜在错误和性能问题。还有，某些 auto 型别推导结果尽管是按部就班地符合标准规定的推导算法，然而从程序员的视角来看却是错误的。如果是这样的情况，那就很有必要知道如何去引导 auto 得出正确结果，因为退回手工指定型别的声明，是通常需要避免的途径。

这个小小的章节讨论了有关 auto 的一切。

条款 5：优先选用 auto，而非显式型别声明

啊，下面这个语句多么天真无邪：

```
int x;
```

等等，我忘记初始化 x 了，所以它的值是不确定的。不过也不一定，它也许被初始化为零了。到底怎样全看具体语境。

别烦恼。我们可以再看一个天真无邪的例子，使用迭代器的提领结果来初始化局部变量：

```
template<typename It>          // dwim（做我所想，do what I mean）算法
void dwim(It b, It e)          // 应用范围是从 b 到 e 的所有元素
{
  while (b != e) {
    typename  std::iterator_traits<It>::value_type
      currValue = *b;
```

```
    ...
  }
}
```

我的天啊。用“typename std::iterator_traits<It>::value_type”来表达迭代器所指涉到的值的型别？这样真的好吗？我真想忘记把这个型别写出来的“乐趣”。等等，我真的说过这么干有“乐趣”的话？

好吧，再来个天真无邪的（事不过三）：用闭包的型别来声明局部变量。不过，闭包的型别只有编译器晓得，所以写不出来呢。唉，我的天啊！

用 C++ 编程，享受不到应有的乐趣啊！

好吧，在已过时的旧标准下就是这样水深火热。可是，C++11 中我们迎来了 auto，一切烦恼都烟消云散了。用 auto 声明的变量，其型别都推导自其初始化物，所以它们必须初始化。所以，当你在现代 C++ 的高速公路上飞驰时，可以向那一系列由未初始化的变量带来的问题挥手告别了：

```
int x1;                          // 有潜在的未初始化风险

auto x2;                         // 编译错误！必须有初始化物

auto x3 = 0;                     // 没问题，x 的值有着合式定义
```

这条高速公路上可没有使用提领迭代器来声明局部变量时会遇到的那些坑：

```
template<typename It>            // 意义同前
void dwim(It b, It e)
{
  while (b != e) {
    auto currValue = *b;
    ...
  }
}
```

而且，由于 auto 使用了型别推导（参见条款 2），就可以用它来表示只有编译器才掌握的型别：

```
auto derefUPLess =                                  // Widget 对象的比较函数
  [](const std::unique_ptr<Widget>& p1,
     const std::unique_ptr<Widget>& p2)             // 使用 std::unique_ptr 来
  { return *p1 < *p2; };                            // 指涉到形参对象
```

这已经很酷了。但在 C++14 中，更是酷毙了，因为连 lambda 表达式的形参中都可以使用 auto：

```
auto derefLess =                    // C++14 中的比较函数
  [](const auto& p1,                // 可以应用于任何类似指针之物
     const auto& p2)                // 指涉到的值
  { return *p1 < *p2; };
```

撇开这样写有多酷不谈，也许你会感觉我们并不需要声明变量来持有闭包，因为我们可以使用 std::function 对象来完成这件事。这种说法是成立的，但是结果未必如你所想。好，你现在可能就在想，“啥是 std::function 对象啊？”那我们就先来搞清楚这个问题。

std::function 是 C++11 标准库中的一个模板，它把函数指针的思想加以推广。函数指针只能指涉到函数，而 std::function 却可以指涉任何可调用对象，即任何可以像函数一样实施调用之物。正如你若要创建一个函数指针就必须指定欲指涉到的函数的型别（即该指针指涉到的函数的签名），你若要创建一个 std::function 对象就必须指定欲指涉的函数的型别。这一步是通过 std::function 模板形参来完成的。举个例子，声明一个名为 func 的 std::function 对象，它可以指涉到任何能够以下述签名调用的对象。

```
bool(const std::unique_ptr<Widget>&,   // C++11 版本的
     const std::unique_ptr<Widget>&)   // std::unique_ptr<Widget>
                                       // 比较函数签名
```

这样来定义 func：

```
std::function<bool(const  std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)> func;
```

因为 lambda 表达式可以产生可调物对象，std::function 对象中就可以存储闭包。这就意味着，在 C++11 中，不用 auto 也可以声明 derefUPLess 如下：

```
std::function<bool(const  std::unique_ptr<Widget>&,
                   const  std::unique_ptr<Widget>&)>
  derefUPLess = [](const std::unique_ptr<Widget>& p1,
                   const std::unique_ptr<Widget>& p2)
                { return *p1 < *p2; };
```

值得一说的是，抛开词法上的啰嗦和需要指定重复的形参型别不谈，使用 std::function 和使用 auto 还是有所不同。使用 auto 声明的、存储着一个闭包的变量和该闭包是同一型别，从而它要求的内存量也和该闭包一样。而使用 std::function 声明的、存储着一个闭包的变量是 std::function 的一个实例，所以不管给定的签名如何，它都占有固定尺寸的内存，而这个尺寸对于其存储的闭包而言并不一定够用。如果是这样的话，std::function 的构造函数就会分配堆上的内存来存储该闭包。从结果上看，std::function 对象一般都会比使用 auto 声明的变量使用更多内

存。再有，编译器的实现细节一般都会限制内联。并会产生间接函数调用，把这些因素考虑在内的话，通过 std::function 来调用闭包几乎必然会比通过使用 auto 声明的变量来调用同一闭包要来得慢。换言之，std::function 手法通常比起 auto 手法来又大又慢，还可能导致内存耗尽异常。还有，在前述例子中，写一个“auto”可比写一整个 std::function 实例型别要省事多了。在持有闭包的这场发生在 auto 和 std::function 之间的较量中，auto 可谓大获全胜（如果再来一场较量，在 auto 和 std::function 之间比较持有 std:bind 的调用结果，则比分会是一样的。不过，在条款 34 中，我会竭尽全力地说服你去使用 lambda 式而非 std::bind，这是后话）。

auto 的优点还不止这些，除了避免未初始化的变量和啰嗦的变量声明，并且可以直接持有闭包外，它还可以避免一类我称为“型别捷径”的问题。下面的代码你可能曾经看到过，甚至曾经写出过：

```
std::vector<int> v;
…
unsigned sz = v.size();
```

标准规定，v.size() 的返回值型别应为 std::vector<int>::size_type，但只有很少一部分程序员会注意到，std::vector<int>::size_type 仅仅规定成一个无符号整型，所以很多程序员感觉 unsigned 足够了，于是就这样写了代码。但这会导致一些有意思的后果。比如说，在 32 位 Windows 上，unsigned 和 std::vector<int>::size_type 的尺寸是一样的，但在 64 位 Windows 上，unsigned 是 32 位，而 std::vector<int>::size_type 则是 64 位。这就意味着，在 32 位 Windows 上运行正常的代码可能在 64 位 Windows 上会表现异常，在将你的应用从 32 位移植到 64 位时，谁会愿意花费时间在解决这样的问题上呢？

而使用 auto 就可以保证你不会这样浪费时间：

```
auto sz = v.size();   // sz 的型别是 std::vector<int>::size_type
```

还是感觉对 auto 的大智慧有点拿不准？那就看看下面这段代码：

```
std::unordered_map<std::string, int> m;
…

for (const std::pair<std::string, int>& p : m)
{
  …                                  // 在 p 上实施某些操作
}
```

这段代码看起来完全合情合理，但其中暗藏隐患。你看出来了没有？

要想意识到缺失的是什么，就要记住 std::unordered_map 的键值部分是 const，所以哈希表中的 std::pair（也就是 std::unordered_map 本身）的型别并不是 std::pair<std::string, int>，而是 std::pair<const std::string, int>。可是，在上面的循环中，用以声明变量 p 的型别却并不是这个。结果编译器就要开足马力找到某种办法把 std::pair<const std::string, int> 对象（即哈希表中的元素）转换成 std::pair<std::string, int> 对象（声明 p 的型别）。这一步是可以成功的，方法是对 m 中的每个对象都做一次复制操作，形成一个 p 想要绑定的型别的临时对象，然后把 p 这个引用绑定到该临时对象。在循环的每次迭代结束时，该临时对象都会被析构一次。如果这个循环是你写的，你恐怕会惊异于其表现，因为你想要的效果几乎肯定只是想把引用 p 依次绑定到 m 中的每个元素而已。

这样的无心之错引发的型别不匹配可以轻松地使用 auto 化解：

```
for (const auto& p : m)
{
  …                              // 同前
}
```

这样做不仅运行效率能提升，而且打字也更少。犹有进者，这段代码还有一个极其诱人的特点，那就是如果对 p 取址，肯定会取得一个指涉到 m 中的某个元素的指针。而在那段没有使用 auto 的代码中，取得的则是一个指涉到临时对象的指针，而且这个对象会在循环迭代结束时被析构。

这最后两个例子（本来应该写成 std::vector<int>::size_type，却写成了 unsigned；本来应该写成 std::pair<const std::string, int>，却写成了 std::pair<std::string, int>）已经说明了，显式指定型别可能导致你既不想要，也没想到的隐式型别转换。如果你使用 auto 作为目标变量的型别，就完全没必要担心在用以声明变量的型别和它的初始化表达式的型别之间发生的不匹配了。

所以，有若干种理由使得 auto 相对于显式型别声明而言成为优选。话说回来，auto 也并不完美。每个 auto 变量的型别都是从它的初始化表达式推导出来的，而有些初始化表达式的型别既不符合期望也不符合要求。这样的情况在何时会出现，又该如何应对，这是条款 2 和条款 6 讨论的内容，在这里就不展开了。但我想把注意力放在人们可能在传统的型别声明之处开始改用 auto 时心存疑虑的问题上，即改用 auto 后写出来的源代码的可读性问题。

首先，深呼吸，放轻松。auto 是一个可选项，而不是一个必选项。如果，以你的专业判断之下，你的代码在使用显式型别声明的前提下更清晰、可维护性更高，或者有什么其他的好处，那你当然可以继续使用它们。但是要记住，C++ 中引入 auto 其实并

不是什么新鲜事，而只是采纳了在其他编程语言中称为型别推导的东西而已。其他的静态型别过程式语言（如C#、D、Scala、Visual Basic）中或多或少都有这么一个等价的特性，更不用说一系列静态型别函数式语言（如ML、Haskell、OCaml、F#等）。这些语言之所以要引入如此特性，部分原因在于动态型别语言的成功，而在后者中变量很少有显式型别。软件开发社区已经积累了丰富的型别推导方面的经验，而这也说明此类技术并不会与撰写和维护大型的、工业强度的基础代码这样的工作产生冲突。

有些程序员会觉得使用了auto以后，就不能一眼从源代码中看出对象的型别，因而感觉烦恼。但是，IDE的对象型别显示能力往往会缓和这个问题（即使考虑到条款4中提及的IDE的型别显示缺陷），并且，在很多情况下，对于对象型别的抽象理解可能和了解它的精确型别同等有用。比如说，知道某个对象是个容器、是个计数器，还是个智能指针就已经够用了，没有必要了解它具体是哪种型别的容器、计数器或智能指针。如果再取个好一点的变量名字，那么一些抽象型别信息就近乎总是唾手可得了。

事实上，显式地写出型别经常是画蛇添足，带来各种微妙的偏差，有些关乎正确性，有些关乎效率，或是两者都受影响。还有，auto型别可以随着其初始化表达式的型别变化而自动随之改变，这就意味着通过使用auto，有一些重构动作就被顺手做掉了。例如，假设有一个函数本来声明的返回型别是int，但后来又觉得long更合适一些，那么如果函数调用的结果是存储在auto变量中的，则调用它的代码在下次编译时就会自动更新自己。但如果调用的结果是存储在声明为int的变量中的话，就需要找到这个函数的所有调用点，才能更新它们了。

要点速记

- auto变量必须初始化，基本上对会导致兼容性和效率问题的型别不匹配现象免疫，还可以简化重构流程，通常也比显式指定型别要少打一些字。
- auto型别的变量都有着条款2和条款6中所描述的毛病。

条款6：当auto推导的型别不符合要求时，使用带显式型别的初始化物习惯用法

条款5解释了使用auto声明变量相对于显式指定型别而言，可以带来若干技术上的优势。但是有的情况下，你想往东，auto型别推导的结果偏偏向西。举个例子，假设我有一个函数接受一个Widget并返回一个std::vector<bool>，其中每一个bool元素都代表着Widget是否提供一种特定功能：

```
std::vector<bool> features(const Widget& w);
```

再假设，这里面的第 5 个比特代表的意思是：Widget 是否具有高优先级。所以我们会写出下面的代码：

```
Widget w;
...

bool highPriority = features(w)[5];     // w具有高优先级吗?
...

processWidget(w, highPriority);         // 按照w的优先级来处理之
```

这段代码没有什么问题，它会运行得很好。不过，只要我们做一个看似无害的改变，把 highPriority 从显式型别改成 auto，

```
auto highPriority = features(w)[5];     // w具有高优先级吗?
```

情况顿时急转直下。所有的代码仍然可以编译，但是其行为则变得不再可以预期了：

```
processWidget(w, highPriority);         // 未定义行为!
```

正如注释所言，这个对 processWidget 的调用现在会导致未定义行为。但为什么会这样呢？答案有点儿出乎意料。在那段改用了 auto 的代码中，highPriority 的型别不再是 bool 了。尽管从概念上说，std::vector<bool> 应该持有的是 bool 型别的元素，但 std::vector<bool> 的 operator[] 的返回值并不是容器中的一个元素的引用（对于其他所有形参型别而言，std::vector::operator[] 都返回这样的值，单单 bool 是个例外）。它返回的是个 std::vector<bool>::reference 型别的对象（这是个嵌套在 std::vector<bool> 里面的类）。

之所以要弄出个 std::vector<bool>::reference，是因为 std::vector<bool> 做过特化，用了一种压缩形式表示其持有的 bool 元素，每个 bool 元素用一个比特来表示。这种做法给 std::vector<bool> 的 operator[] 带来了一个问题，因为按理说 std::vector<T> 的 operator[] 应该返回一个 T&，然而 C++ 中却禁止比特的引用。既然不能返回一个 bool&，std::vector<bool> 的 operator[] 转而返回了一个表现得像 bool& 的对象。而这个把戏若要成功，std::vector<bool>::reference 型别的对象就要在所有能用 bool& 的地方保证它们也能用。实现这个效果的原理是，std::vector<bool>::reference 做了一个向 bool 的隐式型别转换（目标型别不是 bool&，而是 bool。如果要把 std::vector<bool>::reference 模拟 bool& 的全部技术都解释明白，哪怕是要偏题了。所以，这里我只给出一点说明，那就是这个隐式型别转换不过是一堵更大的马赛克墙上的一块砖而已）。

且先记着这些，然后再看一遍原始代码中的这一句：

```
bool highPriority = features(w)[5];  // 显式声明 highPriority 的型别
```

这里，features 返回了一个 std::vector<bool> 对象，然后针对该对象执行 operator[]。尔后，operator[] 返回一个 std::vector<bool>::reference 型别的对象，该对象紧接着被隐式转换为一个初始化 highPriority 所需的 bool 对象。所以，highPriority 的值最终被设定为 features 所返回的 std::vector<bool> 对象的第 5 个比特，一如所愿。

对比一下用 auto 来声明 highPriority 时所发生的事：

```
auto highPriority = features(w)[5];    // highPriority 的型别由推导而得
```

和前面一样，features 返回了一个 std::vector<bool> 对象，然后针对该对象执行 operator[]。尔后，operator[] 返回一个 std::vector<bool>::reference 型别的对象，可是从这里开始就不一样了，因为 auto 会把 highPriority 的型别推导成 std::vector<bool>::reference。这么一来，highPriority 的值就完全不可能会是 features 所返回的 std::vector<bool> 对象的第 5 个比特了。

在后一种情况下，highPriority 的值取决于 std::vector<bool>::reference 的实现。有一种实现让对象含有一个指针，指涉到一个机器字（word），该机器字持有那个被引用的比特，再加上基于那个比特对应的字的偏移量。考虑一下，如果 std::vector<bool>::reference 真的是这样实现的话，highPriority 的初始化将会如何完成。

对 features 的调用会返回一个 std::vector<bool> 型别的临时对象。该对象没有名字，但为讨论方便，我称之为 *temp*。针对 *temp* 执行 operator[]，返回一个 std::vector<bool>::reference 型别的对象，该对象含有一个指涉到机器字的指针，该机器字在一个持有 *temp* 所管理的那样比特的数据结构中，还要加上在第 5 个比特所对应的机器字的偏移量。由于 highPriority 是 std::vector<bool>::reference 对象的一个副本，所以 highPriority 也含有一个指涉到 *temp* 中的机器字的指针，加上还要加上在第 5 个比特所对应的机器字的偏移量。在表达式结束处，*temp* 会被析构，因为它是一个临时对象。结果，highPriority 会含有一个空悬指针（dangling pointer），最终导致调用 processWidget 时出现的未定义行为：

```
processWidget(w, highPriority);          // 未定义行为!
                                         // highPriority 含有空悬指针
```

std::vector<bool>::reference 是个代理类的实例。所谓代理类，就是指为了模拟或

增广其他型别的类。代理类的用途广泛。比如说，std::vector<bool>::reference 就是为了制造 std::vector<bool> 的 operator[] 返回了一个比特的引用的假象。再比如说，标准库中的智能指针（参见第 4 章）也是代理类，它们是为了将资源管理嫁接到裸指针之上。代理类的应用源远流长。事实情况是“代理”乃是软件设计模式万神殿中拥有最多信徒者之一。

有些代理类的设计让客户一望即知，这样的例子有 std::shared_ptr 和 std::unique_ptr。而其他一些代理类则设计得多少有些隐藏在背后的意思，std::vector<bool>::reference 就是这样的“隐形”代理，它的同胞兄弟，std::bitset 相对应的 std::bitset::reference 也一样。

同样属于代理类阵营的，还有另外一些 C++ 库中的类，它们采用了一种叫做表达式模板的技术。一开始开发这种库是为了提高数值计算代码的效率。例如，给定一个类 Matrix，以及 Matrix 型别的对象 m1、m2、m3 和 m4，

```
Matrix sum = m1 + m2 + m3 + m4;
```

如果 Matrix 对象的 operator+ 返回的是结果的代理，而非结果本身，则上述表达式的计算会高效得多。所以，两个 Matrix 对象的 operator+ 会返回一个代理类对象，比如 Sum<Matrix, Matrix>，而不是一个 Matrix 对象。和 std::vector<bool>::reference 与 bool 一样，从代理类到 Matrix 也会有一个隐式型别转换，从而允许 sum 使用位于“=”右侧的表达式生成的代理对象进行初始化（这种对象的型别往往会对整个初始化表达式进行编码，生成像 Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix> 这样的型别，这显然是需要对使用它的客户屏蔽的）。

一个普遍的规律是，“隐形”代理类和 auto 无法和平共处。这种类的对象往往会设计成仅仅维持到单个语句之内存在，所以，如果要创建这种类的变量，往往就是违反了基本的库设计的假定前提。这就是 std::vector<bool>::reference 的情形，而我们也看到了，违反这样的假定前提会导致未定义行为。

所以，你要防止写出这样的代码：

```
auto someVar = "隐形"代理型别表达式;
```

但如何才能知道代码中正在使用代理类呢？使用了它们的软件也未必会张扬其存在。它们天生隐形，至少概念上如此！还有，一旦发现了它们的踪影，是否真的就因此要放弃 auto 以及条款 5 中描述的那么多好处呢？

让我们先回答“如何发现”的问题。尽管“隐形”的代理类设计出来的本意是在日常

应用中进行低空飞行以避开程序员的雷达，使用它们的库往往会在文档中写明这一点。你对于自己使用的库作出的基础决定方面愈是熟悉，你就愈是不会对它们使用了代理的场合视而不见。

文档若不够，头文件来凑。源代码想要完全掩盖住代理对象是不太可能的。代理对象大多数是由客户意欲调用的函数所返回的，所以函数签名往往会反映出它们的存在。例如，以下是 std::vector<bool>::reference 的规格：

```
namespace std {                                    // 摘自 C++ 标准

  template <class Allocator>
  class vector<bool, Allocator> {
  public:
    ...
    class reference { ··· };

    reference operator[](size_type n);
    ...
  };
}
```

假设你已经知道 std::vector<T> 的 operator[] 在通常情况下返回的是 T&，本例中非同寻常的返回值型别就是一个此处使用了代理类的提醒。仔细观察你所用的接口，也可以揭示代理类的存在。

实际情况下，很多程序员只会在跟踪神秘的编译问题或调试不正确的单元测试结果时才会发现使用了代理类。无论代理类是如何被人发现的，只要是问题出在 auto 被决断成了代理型别，而非它意欲代理的那个型别，解决方案都不必放弃 auto。auto 本身并不是问题。问题在于 auto 没有推导成为你想推导出来的型别。解决方案应该是强制进行另一次型别转换。这种方法我称为带显式型别的初始化物习惯用法。

带显式型别的初始化物习惯用法要求使用 auto 声明变量，但针对初始化表达式进行强制型别转换，转换成你想要 auto 推导出来的型别。例如，下面是如何使用这种习惯用法来强制让 highPriority 成为一个 bool：

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

此处，features(w)[5] 仍然一如既往地返回一个 std::vector<bool>::reference 对象，但是强制型别转换将这个表达式的型别转换成了 bool，从而 auto 就将 highPriority 推导成了该型别。在运行时阶段，std::vector<bool>::operator[] 返回的 std::vector<bool>::reference，执行了其支持的向 bool 的强制型别转换，而作为强制型别转换过程的一部分，那个仍然有效的、指涉到 features 返回的 std::vector<bool> 的指针也被提领了。这就避免了此前我们遭遇的未定义行为。尔

后，下标 5 被应用到了该指针指涉到的比特，如此浮现出来的 bool 值就被用来初始化 highPriority。

在 Matrix 这个例子中，使用带显式型别的初始化物习惯用法以后，代码就会写成这样：

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

这种习惯用法的应用并不限于会产生代理型别别的初始化物。它同样可以应用于你想要强调你意在创建一个型别有异于初始化表达式型别的变量的场合，例如，假设你有一个计算某容差值的函数。

```
double calcEpsilon();          // 返回容量差
```

calcEpsilon 显然返回一个 double，但是假如你了解对于你的应用程序来说，float 的精度已经够用，另外你也在乎 float 和 double 之间的尺寸差异。你可以声明一个 float 变量来存储 calcEpsilon 的结果。

```
float ep = calcEpsilon();      // 进行从 double 到 float 的隐式型别转换
```

但这样的写法难以表明“我故意降低了函数的返回值精度”，而一个采用了带显式型别的初始化物习惯用法的声明却可以做到：

```
auto ep = static_cast<float>(calcEpsilon());
```

如果你需要故意把浮点表达式存储为整型，也可以采用相似的理念。假定你需要某个计算带有随机访问迭代器的容器（例如一个 std::vector、std::deque 或 std::array）中的某个元素的下标，而给到你的是一个 double 型别的、在 0.0 和 1.0 之间的值来表示和容器起始处有多远（0.5 表示容器的中间位置）。进一步假定你有信心计算出来的结果下标值在 int 范围内可以容纳。如果用 c 来表示容器，用 d 来表示这个 double 值，你可以这样来计算下标：

```
int index = d * c.size();
```

但这种写法会让你故意将右侧的 double 强制转换成 int 的事实显得含含糊糊。而采用带显式型别的初始化物习惯用法就可以让事情变得显而易见了：

```
auto index = static_cast<int>(d * c.size());
```

要点速记

- “隐形”的代理型别可以导致 auto 根据初始化表达式推导出“错误的”型别。
- 带显式型别的初始化物习惯用法强制 auto 推导出你想要的型别。

第 3 章

转向现代 C++

C++11 和 C++14 中“知名特性”可谓琳琅满目：auto、智能指针、移动语义、lambda 式、并发，不一而足。这些特性都很重要，我为它们中的每一个都分配了整整一章来讲述。掌握这些特性固然重要，但是要成为一名高效的现代 C++ 程序员还需要一系列的小步快跑。这每一小步都在回答从 C++98 转向现代 C++ 之旅中浮现的一些特定问题。在创建对象时，什么时候应该使用大括号而非小括号？为何别名声明优于 typedef？constexpr 和 const 有何不同？ const 成员函数和线程安全有何关系？这个问题列表会越来越长。而我们在本章内会把这些问题各个击破。

条款 7：在创建对象时注意区分 () 和 {}

取决于你怎么看了，C++11 中进行对象初始化的语法选择带来的或是选择太多的无所适从，或是眼花缭乱的左右为难。总而言之，指定初始化值的方式包括使用小括号、使用等号，或是使用大括号：

```
int x(0);              // 初始化物在小括号内
int y = 0;             // 初始化物在等号之后
int z{ 0 };            // 初始化物在大括号内
```

在很多情况下，使用一个等号和一对大括号也是可以的：

```
int z = { 0 };         // 使用等号和大括号来指定初始化物
```

在本条款其余部分，一般我将忽略这种“等号加大括号”语法，因为 C++ 通常会把它和只有大括号的语法同样处理。

“眼花缭乱”派会指出，使用一个等号来书写初始化语句往往会让 C++ 新手误以为这里面会发生一次赋值，但实际上却是没有的。对于像 int 这样的内建型别来说，初始化和赋值的区别不过是学术之争，但是对于用户定义的型别，把初始化和赋值彻底分开就不是可有可无了，这是因为两种行为背后调用的乃是不同的函数：

```
Widget w1;              // 调用的是默认构造函数

Widget w2 = w1;         // 并非赋值，调用的是复制构造函数

w1 = w2;                // 并非赋值，调用的是复制赋值运算符
```

尽管有了那么多初始化语法，但在 C++98 中却仍然没有办法来表达某些想要进行的初始化。比如，之前是没有办法直接指定一个 STL 容器在创建时持有一个特定集合的值的（比如持有 1、3 和 5）。

为了着手解除众多的初始化语法带来的困惑，也为了解决这些语法不能覆盖所有初始化场景的问题，C++11 引入了统一初始化：单一的、至少从概念上可以用于一切场合、表达一切意思的初始化。它的基础是大括号形式，因此我更喜欢称之为大括号初始化（braced initialization）。“统一初始化”是为其里，“大括号初始化”是为其表。

大括号初始化可以让你表达此前无法表达之事。使用大括号来指定容器的初始内容非常简单：

```
std::vector<int> v{ 1, 3, 5 };          // v 的初始内容为 1、3、5
```

大括号同样可以用来为非静态成员指定默认初始化值，这项能力（在 C++11 中新加入的能力）也可以使用“=”的初始化语法，却不能使用小括号：

```
class Widget {
  …

private:
  int x{ 0 };           // 可行，x 的默认值为 0
  int y = 0;            // 也可行
  int z(0);             // 不可行！
};
```

但话说回来，不可复制的对象（如 std::atomic 型别的对象，参见条款 40）可以采用大括号和小括号来进行初始化，却不能使用“=”：

```
std::atomic<int> ai1{ 0 };      // 可行

std::atomic<int> ai2(0);        // 可行

std::atomic<int> ai3 = 0;       // 不可行！
```

这么一来，就很容易理解为何大括号初始化被冠以“统一”之名了。在C++的三种初始化表达式的写法中，只有大括号适用于所有场合。

大括号初始化有一项新特性，就是它禁止内建型别之间进行隐式窄化型别转换（narrowing conversion）。如果大括号内的表达式无法保证能够采用进行初始化的对象来表达，则代码不能通过编译：

```
double x, y, z;

...

int sum1{ x + y + z };          // 错误！ double 型别之和可能无法用 int 表达
```

而采用小括号和“=”的初始化则不会进行窄化型别转换检查，因为如果那样的话就会破坏太多的遗留代码了：

```
int sum2(x + y + z);            // 没问题（表达式的值被截断为 int）

int sum3 = x + y + z;           // 同上
```

大括号初始化的另一项值得一提的特征是，它对于C++的**最令人苦恼之解析语法**（most vexing parse）免疫。C++规定：任何能够解析为声明的都要解析为声明，而这会带来副作用。所谓最令人苦恼之解析语法就是说，程序员本来想要以默认方式构造一个对象，结果却一不小心声明了一个函数。这个错误的根本原因在于构造函数调用语法。当你想以传参方式调用构造函数时，可以这样写：

```
Widget w1(10);          // 调用 Widget 的构造函数，传入形参 10
```

但如果你试图用对等语法来调用一个没有形参的`Widget`构造函数的话，那结果却变成声明了一个函数而非对象：

```
Widget w2();            // 最令人苦恼之解析语法现身！
                        // 这个语句声明了一个名为 w2、返回一个 Widget 型别对象的函数！
```

由于函数声明不能使用大括号来指定形参列表，所以使用大括号来完成对象的默认构造没有上面这个问题：

```
Widget w3{};            // 调用没有形参的 Widget 构造函数
```

关于大括号初始化，也说了不少了。这种语法可以应用的语境最为宽泛，可以阻止隐式窄化型别转换，还对最令人苦恼之解析语法免疫。那么，为什么本章的章名不干脆换成“优先选用大括号初始化语法”之类呢？

大括号初始化的缺陷在于伴随它有时会出现的意外行为。这种行为源于大括号初始化

物、std::initializer_list以及构造函数重载决议之间的纠结关系。这几者之间的相互作用可以使得代码看起来是要做某一件事，但实际上是在做另一件事。比如，条款2就说明了，如果使用大括号初始化物来初始化一个使用auto声明的变量，那么推导出来的型别就会成为std::initializer_list，尽管用其他方式使用相同的初始化物来声明变量就能够得出更符合直觉的型别。从结果上说，可能你越是喜爱使用auto，恐怕就越会对大括号初始化意兴阑珊。

在构造函数被调用时，只要形参中没有任何一个具备std::initializer_list型别，那么小括号和大括号的意义就没有区别：

```
class Widget {
public:
  Widget(int i, bool b);       // 构造函数的形参中没有任何一个具备
  Widget(int i, double d);     // std::initializer_list 型别
  ...
};

Widget w1(10, true);           // 调用的是第一个构造函数

Widget w2{10, true};           // 调用的还是第一个构造函数

Widget w3(10, 5.0);            // 调用的是第二个构造函数

Widget w4{10, 5.0};            // 调用的还是第二个构造函数
```

如果，有一个或多个构造函数声明了任何一个具备std::initializer_list型别的形参，那么采用了大括号初始化语法的调用语句会强烈地优先选用带有std::initializer_list型别形参的重载版本。真的非常强烈。编译器只要有任何可能把一个采用了大括号初始化语法的调用语句解读为带有std::initializer_list型别形参的构造函数，则编译器就会选用这种解释。举个例子，如果上述Widget类增加了一个带有std::initializer_list<long double>型别的形参：

```
class Widget {
public:
  Widget(int i, bool b);                                  // 同前
  Widget(int i, double d);                                // 同前
  Widget(std::initializer_list<long double> il);          // 增加的版本

...
};

Widget w1(10, true);           // 使用小括号，同前，调用的是第一个构造函数

Widget w2{10, true};           // 使用大括号，调用的是带有
                               // std::initializer_list 型别形参的构造函数
                               // （10和true被强制转换为long double）
```

```
Widget w3(10, 5.0);          // 使用小括号，同前，调用的是第二个构造函数

Widget w4{10, 5.0};          // 使用大括号，调用的是带有
                             // std::initializer_list 型别形参的构造函数
                             //  (10 和 5.0 被强制转换为 long double)
```

即使是平常会执行复制或移动的构造函数也可能被带有 std::initializer_list 型别形参的构造函数劫持：

```
class Widget {
public:
  Widget(int i, bool b);                                  // 同前
  Widget(int i, double d);                                // 同前
  Widget(std::initializer_list<long double> il);          // s 同前

  operator float() const;                                 // 强制转换成 float 型别
  ...
};

Widget w5(w4);          // 使用小括号，调用的是复制构造函数

Widget w6{w4};          // 使用大括号，调用的是带有
                        // std::initializer_list 型别形参的构造函数
                        // (w4 的返回值被强制转换成 float，随后 float 又被
                        // 强制转换成 long double)

Widget w7(std::move(w4));      // 使用小括号，调用的是移动构造函数

Widget w8{std::move(w4)};      // 使用大括号，调用的是带有
                               // std::initializer_list 型别形参的构造函数
                               // (和 w6 的结果理由相同)
```

编译器想要把大括号初始化物匹配带有 std::initializer_list 型别形参的构造函数的决心是如此的强烈，以至于最优选的带有 std::initializer_list 型别形参的构造函数无法被调用时，这种决心还是会占上风。举个例子：

```
class Widget {
public:
  Widget(int i, bool b);                          // 同前
  Widget(int i, double d);                        // 同前

  Widget(std::initializer_list<bool> il);         // 容器元素型别现在是 bool 了

  ...                                             // 并没有隐式强制型别转换函数
};

Widget w{10, 5.0};     // 错误！要求窄化型别转换
```

这里，编译器会忽略前两个构造函数（第二个构造函数的形参表和实参表的型别是精确匹配的），转而尝试带有一个 std::initializer_list<bool> 型别形参的构造函数。

而要调用该函数就要求把一个 int（10）和一个 double（5.0）强制转换成 bool 型别。而这两个强制型别转换都是窄化的（bool 无法精确表示这两个值中的任何一个），并且窄化型别转换在大括号初始化物内部是禁止的，所以这个调用不合法，导致整段代码通不过编译。

只有在找不到任何办法把大括号初始化物中的实参转换成 std::initializer_list 模板中的型别时，编译器才会退而去检查普通的重载决议。举例来说，如果我们把带有一个 std::initializer_list<bool> 型别形参的构造函数换成带有一个 std::initializer_list<std::string> 的话，那么形参中没有任何一个具备 std::initializer_list 型别的构造函数再次成为候选，因为找不到任何办法将 int 和 bool 转换成 std::string 型别：

```
class Widget {
public:
  Widget(int i, bool b);                    // 同前
  Widget(int i, double d);                  // 同前

  // std::initializer_list 模板的元素型别现在成为 std::string 了
  Widget(std::initializer_list<std::string> il);
  …                                         // 并没有隐式强制型别转换函数
};
Widget w1(10, true);  // 使用小括号，调用的仍是第一个构造函数

Widget w2{10, true};  // 使用大括号，变成调用第一个构造函数了

Widget w3(10, 5.0);   // 使用小括号，调用的仍是第二个构造函数

Widget w4{10, 5.0};   // 使用大括号，变成调用第二个构造函数了
```

以上讨论基本上就是考察大括号初始化物和构造函数重载的全部内容了，但有个有意思的边界用例需要特别提及。假定你使用了一对空大括号来构造一个对象，而该对象既支持默认构造函数，又支持带有 std::initializer_list 型别形参的构造函数。那么，这对空大括号的意义是什么呢？如果意义是“没有实参”，那就应该执行默认构造；但如果意义是“空的 std::initializer_list”，那就应该以一个不含任何元素的 std::initializer_list 为基础执行构造。

语言规定，在这种情形下应该执行默认构造。空大括号对表示的是“没有实参”，而非“空的 std::initializer_list”：

```
class Widget {
public:
  Widget();                                 // 默认构造函数

  Widget(std::initializer_list<int> il);    // 带有 std::initializer_list 型别形参的
                                            构造函数
```

```
  ...                                    // 并没有隐式强制型别转换函数
};

Widget w1;              // 调用的是默认构造函数

Widget w2{};            // 调用的仍是默认构造函数

Widget w3();            // 最令人苦恼的解析语法现身！变成函数声明语句了！
```

如果你的确想要调用一个带有 std::initializer_list 型别形参的构造函数，并传入一个空的 std::initializer_list 的话，你可以通过把空大括号对作为构造函数实参的方式实现这个目的，即把一对空大括号放入一对小括号或大括号的方式来清楚地表明你传递的是什么：

```
Widget w4({});          // 带有 std::initializer_list 型别形参的构造函数
                        // 传入一个空的 std::initializer_list

Widget w5{{}};          // 同上
```

话说至此，大括号初始化物、std::initializer_list、构造函数重载决议，这些貌似幻术的家伙开始在脑子里汩汩作响之时，你可能会想，这些知识和日常程序设计有什么关系。实际影响比你想象的要大，因为直接受到影响的一个类就是 std::vector。std::vector 有个形参中没有任何一个具备 std::initializer_list 型别的构造函数，它允许你指定容器的初始尺寸，以及一个初始化时让所有元素拥有的值，但它还有个带有一个 std::initializer_list 型别形参的构造函数，允许你逐个指定容器中的元素值。如果你要创建一个元素为数值型别的 std::vector（比如 std::vector<int>），并传递了两个实参给构造函数的话，你把这两个实参用小括号还是大括号括起来，结果会大相径庭：

```
std::vector<int>  v1(10, 20); // 调用了形参中没有任何一个具备
                              // std::initializer_list 型别的构造函数
                              // 结果是：创建了一个含有 10 个元素的 std::vector
                              // 所有的元素的值都是 20

std::vector<int>  v2(10, 20); // 调用了形参中含有
                              // std::initializer_list 型别的构造函数
                              // 结果是：创建了一个含有 2 个元素的 std::vector
                              // 元素的值分别为 10 和 20
```

但先暂时从 std::vector 还有大小括号、构造函数重载决议等细节中退一步。从以上讨论中，可以得到两个主要结论。首先，如果是一个类的作者，你需要有清醒的意识，了解自己撰写的一组重载构造函数中只要有一个或多个声明了任何一个具备 std::initializer_list 型别的形参，则使用了大括号初始化的客户代码可能会只发现那些具备 std::initializer_list 型别形参的重载版本。作为结论，你最好把构造

函数设计成客户无论使用小括号还是大括号都不会影响调用的重载版本才好。换言之，现在一般是把std::vector的接口设计视为败笔的，应该从中汲取教训，避免同类行为。

一个推论是，如果你有一个类本来所有的构造函数的形参中都没有任何一个具备std::initializer_list型别，而你却添加了一个带有std::initializer_list型别形参的构造函数的话，那就有可能在客户代码中使用大括号初始化的地方本来会决议到形参中没有任何一个具备std::initializer_list型别的构造函数的，现在却可能决议到一个新的函数去了。当然，这种事情在你往一组重载函数中添加新版本时随时都可能会出现：本来调用一个旧重载版本的语句，可能就开始调用一个新版本了。而带有std::initializer_list型别形参的构造函数所引发的重载有所不同，不同之处在于它不是仅仅和其他的重载版本发生竞争而已，它带来的阴影达到这样的程度：别的重载版本可能连露脸的机会都不给。所以，在添加像这样的重载版本时，一定要做到心中完全有数。

从以上讨论还可以学到的一件事情，那就是如果你是一个开发类客户代码的程序员，那你在创建对象时对于选用一对小括号还是大括号要三思而后行。大多数程序员最终会选用其中一种作为默认选用的，而另一种则只在必要时才选用。默认选用大括号的同仁是被其宽泛的应用语境、对隐式窄化型别转换的禁止，以及对最令人苦恼之解析语法的免疫所吸引。这些同仁理解在某些情况下（例如，在创建一个std::vector时指定容器的初始尺寸以及一个初始化时让所有元素拥有的值时），使用小括号就是必要的。而相应地，小括号的拥趸则坚守着这种与C++98语法传统的一致性，还可以免受auto型别推导意外错误之苦，在创建对象时也不会冷不丁地碰到带有std::initializer_list型别形参的构造函数设置的路障。他们也承认，有些场合非用大括号不可（例如，创建容器时逐个指定值时）。究竟选用哪一方更好，其实没有定论，所以我的建议是，你可以任选一方并坚持下去。

如果你是一位开发模板的程序员，那么在创建对象时是选用小括号还是大括号，会变成非常头疼的难题。因为，总体来说，你选择这个也不好，那个也不对。举例来说，如果你想以任意数量的实参来创建一个任意型别的对象，那么，一个可变模板可以让这种概念变得直截了当：

```
template<typename T,                    // 创建对象的型别
         typename... Ts>                // 一系列实参的型别
void doSomeWork(Ts&&... params)
{

  利用 params 创建局部对象 T
  ...

}
```

要把那一行伪代码变换成实际代码，有两种方式（有关 std::forward 的信息，参见条款 25）：

```
T localObject(std::forward<Ts>(params)...);      // 采用小括号
T localObject{std::forward<Ts>(params)...};      // 采用大括号
```

所以，考虑一下对应的调用代码：

```
std::vector<int> v;
...
doSomeWork<std::vector<int>>(10, 20);
```

如果 doSomeWork 在创建 localObject 使用了小括号，结果会得到一个包含 10 个元素的 std::vector。如果 doSomeWork 使用了大括号，结果会得到一个包含 2 个元素的 std::vector。哪个是对的呢？ doSomeWork 的作者不可能下这个判断，只有调用者才有决定权。

这正是标准库函数 std::make_unique 和 std::make_shared（参见条款 21）所面临的问题。而这些函数解决问题的办法是在内部使用了小括号，并把这个决定以文档的形式广而告之，作为其接口的组成部分。[注 1]

要点速记

- 大括号初始化可以应用的语境最为宽泛，可以阻止隐式窄化型别转换，还对最令人苦恼之解析语法免疫。
- 在构造函数重载决议期间，只要有任何可能，大括号初始化物就会与带有 std::initializer_list 型别的形参相匹配，即使其他重载版本有着貌似更加匹配的形参表。
- 使用小括号还是大括号，会造成结果大相径庭的一个例子是：使用两个实参来创建一个 std::vector< 数值型别 > 对象。
- 在模板内容进行对象创建时，到底应该使用小括号还是大括号会成为一个棘手问题。

注 1：更具弹性的设计，也就是允许调用者自行决定在从模板生成的函数内使用小括号还是大括号的设计，是可以实现的。具体的技术细节，参阅 Andrzej 的 C++ 博客 2013 年 6 月 5 日的文章“Intuitive interface — Part 1”。

条款 8：优先选用 nullptr，而非 0 或 NULL

开门见山摆事实吧：字面常量 0 的型别是 int，而非指针。当 C++ 在只能使用指针的语境中发现了一个 0，它也会把它勉强解释为空指针，但说到底这是一个不得已而为之的行为。C++ 的基本观点还是 0 的型别是 int，而非指针。

从实际效果来说，以上结论对于 NULL 也成立。虽然说 NULL 在技术细节上有一些不清不楚的成分，因为标准允许各个实现给予 NULL 非 int 的整型型别（如 long）。固然这样的实现不多，但其实也不重要，这里的关键并不在于 NULL 的精确型别，而在于 0 和 NULL 都不具备指针型别。

C++98 中，这样的基本观点可能在指针型别和整型之间进行重载时可能会发生意外。如果向这样的重载函数传递 0 和 NULL，是从来不会调用到要求指针型别的重载版本的。

```
void f(int);            // f 的三个重载版本
void f(bool);
void f(void*);

f(0);                   // 调用的是 f(int)，而不是 f(void*)

f(NULL);                // 可能通不过编译，但一般会调用 f(int)。从来不会调用 f(void*)
```

f(NULL) 的不确定性是 NULL 的型别在实现中的余地的一种反映。打个比方，假设 NULL 的定义为 0L（即 long 型别中的 0），那么这个调用就有多义性了。因为从 long 到 int、从 long 到 bool，还有从 0L 到 void* 的型别转换是被视为同样好的。这个调用有意思的地方在于源代码想要表达的本来意思（“我在调用 f 时传入了 NULL——空指针”）和它的真实含义（我在调用 f 时传入了某种整型——并不是空指针）之间的矛盾。这种违反直觉的行为导致了对于 C++98 的程序员来说，指导原则是不要在指针型别和整型之间做重载。当然这个原则在 C++11 中仍然成立，因为尽管有本条款的建议，有些程序员还是会继续使用 0 和 NULL，尽管 nullptr 是更好的选择。

nullptr 的优点在于，它不具备整型型别。实话实说，它也不具备指针型别，但你可以把它想成一种任意型别的指针。nullptr 的实际型别是 std::nullptr_t，并且，在一个漂亮的循环定义下，std::nullptr_t 的定义被指定为 nullptr 的型别。型别 std::nullptr_t 可以隐式转换到所有的裸指针型别，这就是为何 nullptr 可以扮演所有型别指针的原因。

调用重载函数 f 时传入 nullptr 会调用 void* 那个重载版本（带指针形参的重载版本），因为 nullptr 无法视作任何一种整型：

```
f(nullptr);             // 调用的是 f(void*) 这个重载版本
```

因此，使用nullptr而非0和NULL就避免了重载决议中的意外，但这不是它仅有的优点。它还可以提升代码的清晰性，尤其在涉及auto变量时。举个例子，假设你在代码仓库里发现了这段代码：

```
auto result = findRecord( /* 实参 */ );

if (result == 0) {
  ...
}
```

如果你刚好并不知道（或不容易得出）findRecord的返回值型别的话，那么result是指针型别还是整数型别就不清楚了。毕竟0（作为findRecord的检测值）两者都有可能。但如果你看到的是下面这段代码：

```
auto result = findRecord( /* 实参 */ );

if (result == nullptr) {
  ...
}
```

这回没有多义性了：result必然具备指针型别。

nullptr在有模板的前提下表现最亮眼。假设你有一些函数，它们仅在适当的互斥量被锁定的前提下才能被调用，而每个函数的形参都是不同型别的指针：

```
int f1(std::shared_ptr<Widget> spw);          // 仅当适当信息量被锁定
double f2(std::unique_ptr<Widget> upw);       // 才调用这几个函数
bool f3(Widget* pw);
```

调用这些函数并传入空指针的代码是这样的：

```
std::mutex f1m, f2m, f3m;      // f1、f2和f3对应的信息量

using MuxGuard =               // C++11中的typedef；参见条款9
  std::lock_guard<std::mutex>;
...

{
  MuxGuard g(f1m);             // 为f1锁定互斥量
  auto result = f1(0);         // 向f1传入0作为空指针
}                              // 解锁互斥量

...

{
  MuxGuard g(f2m);             // 为f2锁定互斥量
  auto result = f2(NULL);      // 向f2传入NULL作为空指针
}                              // 解锁互斥量

...
```

```
{
  MuxGuard g(f3m);              // 为 f3 锁定互斥量
  auto result = f3(nullptr);    // 向 f3 传入 nullptr 作为空指针
}                               // 解锁互斥量
```

在前两个调用中未能使用 nullptr 略有遗憾，但这代码还是能够运作的，也算是能交代了。反过来说，调用代码中的重复三部曲（锁定互斥量、调用函数、解锁信号）却是遗憾更大的。简直不忍卒视。这种源代码中的冗余正是模板这一语言特性设计出来所要避免的，所以我们把这个三部曲给模板化：

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr))
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

如果这个函数的返回值（auto …-> decltype(func(ptr)）让你开始挠头，那么请放过你的头，并翻回条款 3，它解释了这里的写法是什么意思。在那里你还能看到，在 C++14 中，返回值可以简化成为一个 decltype(auto)：

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
decltype(auto) lockAndCall(FuncType func,        // C++14
                           MuxType& mutex,
                           PtrType ptr)
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

给定 lockAndCall 模板（两个版本任一），则调用者可能会写出下面的代码：

```
auto result1 = lockAndCall(f1, f1m, 0);         // 错误！

...

auto result2 = lockAndCall(f2, f2m, NULL);      // 错误！

...
auto result3 = lockAndCall(f3, f3m, nullptr);   // 没问题
```

虽然想怎么写都可以，但是正如注释所表明的那样，三种情况中的两种是通不过编译的。第一个调用的问题在于，当 0 被传给 lockAndCall 时，模板型别推导启动。而 0

的型别当然永远是 int，所以在这次对 lockAndCall 调用时的具现里，形参 ptr 的型别就被定为 int。不幸的是，这意味着在 lockAndCall 内部的 func 在调用时传入的是一个 int，但这和 f1 期望的 std::shared_ptr<Widget> 并不兼容。传入 lockAndCall 的 0 本意是想代表一个空指针，结果实际传入时却是把它理解成一个 int。企图将 int 传入 f1 来充数 std::shared_ptr<Widget> 会导致型别错误。使用 0 来调用 lockAndCall 之所以会失败，是因为 int 被传给了一个要求 std::shared_ptr<Widget> 的函数。

对于采用 NULL 的调用，分析过程八九不离十。形参 ptr 的型别被推导为某种整型，然后在 ptr 这个具备 int 或相似于 int 的型别的实参被传递给 f2 时，型别错误就发生了，因为 f2 要求一个 std::unique_ptr<Widget>。

相比之下，采用 nullptr 的调用则毫无问题。当 nullptr 传给 lockAndCall 时，ptr 的型别被推导为 std::nullptr_t。当 ptr 被传递给 f3 时，从 std::nullptr_t 到 Widget* 存在隐式型别转换，因为 std::nullptr_t 可以隐式转换到所有指针型别。

模板型别推导会将 0 和 NULL 推导成“错误”型别（即它们的真实型别，而非退而求其次的表示空指针这个意义），这个事实构成了应该在表示空指针时使用 nullptr 而非 0 或 NULL 的压倒性理由。而使用 nullptr 的话，模板就不会带来特殊的麻烦。加上另一个事实：nullptr 不会造成 0 和 NULL 稍不留意就会遭遇的重载决议问题，这么一来整个结论就板上钉钉了：当你想要表示空指针时，使用 nullptr，而非 0 或 NULL。

要点速记

- 相对于 0 或 NULL，优先选用 nullptr。
- 避免在整型和指针型别之间重载。

条款 9：优先选用别名声明，而非 typedef

我可以肯定你会觉得 STL 容器好用，而我也希望条款 18 能让你觉得 std::unique_ptr 好用，但我猜想不会有人觉得一遍遍地写“std::unique_ptr<std::unordered_map<std::string,std::string>>”这样的型别是有意思的事情。想想这么做会增加腕管综合症的风险就够要命了。

避免此类医学灾难倒也不难。用个 typedef 就行：

```
typedef
std::unique_ptr<std::unordered_map<std::string, std::string>>
UPtrMapSS;
```

但 typedef 的 C++98 特征也太浓了。它在 C++11 中可以运作，这没问题，但 C++11 中还提供了别名声明（alias declaration）：

```
using UPtrMapSS =
  std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

既然 typedef 和别名声明完成的是一模一样的工作，那就有理由考虑是否有坚实的技术理由来说明应该优先选用其中一个，而非另一个。

技术理由是有的，但在说明这个理由之前，我想先指出一点：很多人发现别名声明在处理涉及函数指针的型别时，比较容易理解：

```
// FP 的型别是一个指涉到函数的指针，该函数形参包括
// 一个 int 和一个 const std::string&，没有返回值
typedef void (*FP)(int, const std::string&);    // 使用 typedef

// 和以上这句意义相同
using FP = void (*)(int, const std::string&);   // 使用别名声明
```

当然，这两种形式都不那么好理解，而且花大把时间和函数指针定义同义词的人毕竟有限。所以，以上情况很难构成要别名声明而不要 typedef 的压倒性理由。

但压倒性理由还是存在的，这就是模板。再说详细点，就是别名声明可以模板化（这种情况下它们被称为别名模板，alias template），typedef 就不行。它给予了 C++11 程序员一种直截了当的表达机制，用以表达 C++98 程序员不得不用嵌套在模板化的 struct 里的 typedef 才能硬搞出来的东西。比如，想要定义一个同义词，表达一个链表，它使用了一个自定义分配器 MyAlloc。在使用别名声明的前提下，这是小菜一碟：

```
template<typename T>                              // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>;     // 是 std::list<T, MyAlloc<T>>
                                                  // 的同义词

MyAllocList<Widget> lw;                           // 客户代码
```

如果使用 typedef 的话，你几乎肯定要从头自己动手了：

```
template<typename T>                        // MyAllocList<T>::type 是
struct MyAllocList {                        // std::list<T, MyAlloc<T>> 的同义词
  typedef std::list<T, MyAlloc<T>> type;
};

MyAllocList<Widget>::type lw;               // 客户代码
```

但还有更坏情况。如果你想在模板内使用 typedef 来创建一个链表，它容纳的对象型别由模板形参指定的话，那你就要给 typedef 的名字加一个 typename 前缀：

```
template<typename T>
class Widget {                          // Widget<T> 含一个
private:                                // MyAllocList<T> 型别的数据成员
  typename MyAllocList<T>::type list;
  ...
};
```

这里，MyAllocList<T>::type 代表一个依赖于模板型别形参（T）的型别，所以 MyAllocList<T>::type 称为带依赖型别，而 C++ 中众多讨喜的规则之一就是带依赖型别必须前面加个 typename。

如果 MyAllocList 是使用别名模板来定义的，那么要写 typename 的要求就消失了（一起消失的还有那个“::type”后缀）。

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;  // 同前

template<typename T>
class Widget {
private:
  MyAllocList<T> list;          // 不再有“typename”和“::type”
  ...
};
```

对你来说，MyAllocList<T>（即别名模板的运用）可能和 MyAllocList<T>::type（即嵌套 typedef 的运用）看上去同样程度地依赖模板形参 T，但你不是编译器。当编译器处理到 Widget 模板并遇见了 MyAllocList<T>（即别名模板的运用）时，它们知道 MyAllocList<T> 是一个型别的名字，因为 MyAllocList 是个别名模板：它必然命名了一个型别。综上，MyAllocList<T> 是个非依赖性型别，所以 typename 饰词既不要求也不允许。

但当编译器遇见 Widget 模板中的 MyAllocList<T>::type（即嵌套 typedef 的运用）时，就是另一回事。它们不可能确定 MyAllocList<T>::type 命名了一个型别，因为可能在 MyAllocList 的某个特化中，MyAllocList<T>::type 表示并非型别而是其他的什么东西。这听起来有点儿不可思议，但是不能怪编译器想得太多。人们真的可能写出这样的代码：

例如，有些神经不太正常的人可能会写出这样的代码：

```
class Wine { ... };

template<>                      // 在 T 取值为 Wine 型别时
class MyAllocList<Wine> {       // MyAllocList 的特化
private:
  enum class WineType           // “enum class”的相关信息参见条款 10
  { White, Red, Rose };
```

```
    WineType type;              // 在这个类中，type 是个数据成员
    ...
};
```

如你所见，MyAllocList<Wine>::type 表示的并不是一个型别。如果 Widget 模板采用 Wine 型别进行具现，则其中的 MyAllocList<T>::type 就表示一个数据成员而非型别。这就是说，Widget 模板中的 MyAllocList<T>::type 究竟是否表示一个型别的的确确依赖于 T 是什么，这也正是为何编译器坚持让你通过在前面加一个 typename 来断定这个意思。

如果你做过哪怕一点点模板元编程（template metaprogramming，TMP），那么你肯定有过从模板型别形参出发来创建其修正型别的需要。举例来说，给定某型别 T，你可能会需要去除 T 的所有 const 和引用饰词。例如，你可能需要将 const std::string& 变换成 std::string。又或者你可能想给一个型别加上 const 或将其变换成左值引用形式。例如，将 Widget 变换成 const Widget 或 Widget&（如果你从来没有做过 TMP，那可不太妙，因为如果你真的想成为一名高效的 C++ 程序员，你至少需要熟悉一下 C++ 这个方面的基础知识。你可以在条款 23 和条款 27 中看到 TMP 实战案例，包括刚才说的这几种型别变换）。

C++11 以型别特征（type trait）的形式给了程序员以执行此类变换的工具。型别特征是在头文件 <type_traits> 给出的一整套模板。该头文件中有几十个型别特征，它们并非都是执行型别变换功能的用途，但其中派此用途的部分则提供了可预测的接口。对给定待变换型别 T，其结果型别为 std::transformation<T>::type，例如：

```
std::remove_const<T>::type              // 由 const T 生成 T

std::remove_reference<T>::type          // 由 T& 或 T&& 生成 T

std::add_lvalue_reference<T>::type      // 由 T 生成 T&
```

以上注释只是给出这些变换的结论，所以不要太抠字眼。当你真的要把它们用于实际项目代码中时，肯定会查阅准确规格说明。

我的本意不是想要做一篇型别特征的概览介绍，而是想请你注意，这些变换的应用都以“::type”结尾。如果你想将它们应用于模板内的型别形参（在实际代码中你几乎总要这样来运用），那就还是要每次都在前面加上 typename。这两个词法臧速带存在的原因是，C++11 中的型别特征是用嵌套在模板化的 struct 里的 typedef 实现的。你没有看错，它们的对于型别同义词的实现手法是我前面劝你说不要用的，因为它不如别名模板。

C++11 之所以要这么做，有其历史原因，不过我这里不打算展开（那是一篇枯燥的长篇大论），反正由于标准委员会对于别名模板的好处后知后觉，所以他们到了 C++14 中才为 C++11 中的所有型别变换都加上了对应的别名模板。这些别名有一个公共形式：每个 C++11 中的变换 std::transformation<T>::type，都有一个 C++14 中名为 std::transformation_t 的对应别名模板。下面的例子能够清楚说明我想表达的意思：

```
std::remove_const<T>::type              // C++11: const T → T
std::remove_const_t<T>                  // C++14 中的等价物

std::remove_reference<T>::type          // C++11: T&/T&& → T
std::remove_reference_t<T>              // C++14 中的等价物

std::add_lvalue_reference<T>::type      // C++11: T → T&
std::add_lvalue_reference_t<T>          // C++14 中的等价物
```

C++11 的基础设施在 C++14 中仍然可用，但我不知道为什么你还会用它们。即使你还没用上 C++14，自行写出这些别名声明也是连小孩子都会的功夫。要用的只涉及 C++11 的语言特性，而且连小孩子也会照葫芦画瓢对吧？如果你有 C++14 标准的电子版，那就更简单，只需要复制粘贴就可以了。我在这里把你扶上马送一程吧：

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;
template <class T>
using add_lvalue_reference_t =
  typename add_lvalue_reference<T>::type;
```

看到了吧？没法更简单了。

要点速记

- typedef 不支持模板化，但别名声明支持。
- 别名模板可以让人免写“::type”后缀，并且在模板内，对于内嵌 typedef 的引用经常要求加上 typename 前缀。

条款 10：优先选用限定作用域的枚举型别，而非不限作用域的枚举型别

先说一个通用规则，如果在一对大括号里声明一个名字，则该名字的可见性就被限定在括号括起来的作用域内。但这个规则不适用于 C++98 风格的枚举型别中定义的枚

举量。这些枚举量的名字属于包含着这个枚举型别的作用域，这就意味着在此作用域内不能有其他实体取相同的名字：

```
enum Color { black, white, red };       // black、white、red 所在作用域
                                        // 和 Color 相同

auto white = false;                     // 错误！ white 已在范围内被声明过了
```

这些枚举量的名字泄漏到枚举型别所在作用域的这一事实，催生了此类枚举型别的官方术语：不限范围的（unscoped）枚举型别。它们在 C++11 中的对等物，限定作用域的（scoped）枚举型别，却不会以这样的方式泄漏名字：

```
enum class Color { black, white, red };         // black、white、red 所在作用域
                                                // 被限定在 Color 内

auto white = false;                             // 没问题，范围内并无其他 "white"

Color c = white;                        // 错误！范围内并无名为 "white" 的枚举量

Color c = Color::white;                 // 没问题

auto c = Color::white;                  // 同样没问题（还符合条款 5 的建议）
```

由于限定作用域的枚举型别是通过“enum class”声明的，所以有时它们也被称为枚举类。

限定作用域的枚举型别带来的名字空间污染降低，已经是“应该优先选择它，而不是不限范围的枚举型别”的足够理由。但是限定作用域的枚举型别还有第二个压倒性优势：它的枚举量是更强型别的（strongly typed）。不限范围的枚举型别中的枚举量可以隐式转换到整数型别（并能够从此处进一步转换到浮点型别）。这么一来，下面这样的语义怪胎却成为完全合法的了：

```
enum Color { black, white, red };       // 不限范围的枚举型别

std::vector<std::size_t>                // 函数，返回 x 的质因数
  primeFactors(std::size_t x);

Color c = red;
...
if(c < 14.5) {                  // 将 Color 型别和 double 型别值比较（！）

  auto factors =                // 计算一个 Color 型别的质因数（！）
    primeFactors(c);
  ...
}
```

仅仅是把一个简简单单的 `class` 加到 `enum` 之后，就完成了从不限范围的枚举型别到限定作用域的枚举型别的变换，整个语义一下子就变得完全不同了。从限定作用域的枚举型别到任何其他型别都不存在隐式转换路径：

```
enum class Color { black, white, red };          // 不限作用域的枚举型别

Color c = Color::red;                            // 同前，但现在要加上范围限定饰词
...

if (c < 14.5) {                  // 错误！不能将 Color 型别和 double 型别值比较
  auto factors =                 // 错误！不能将 Color 型别传入
    primeFactors(c);             // 要求 std::size_t 型别形参的函数
  ...
}
```

如果你真的想要实施一个从 Color 到另一型别的转换，那平时你想要怎么样扭曲型别来适应你的要求，就怎么做吧，实施一个强制型别转换即可：

```
if (static_cast<double>(c) < 14.5) {             // 不自然的代码，但合法

  auto factors =                                 // 合法性可疑，但是
    primeFactors(static_cast<std::size_t>(c));   // 能够通过编译
  ...
}
```

乍看之下，限定作用域的枚举型别相对于不限范围的枚举型别还有第三个优点，那就是限定作用域的枚举型别可以进行前置声明，亦即其型别名字可以比其中的枚举量先声明：

```
enum Color;           // 错误！

enum class Color;     // 没问题
```

这是一种误导了。C++11 中，不限范围的枚举型别也可以进行前置声明，但须得在完成一些额外工作之后。这些额外工作是由以下事实带来的：一切枚举型别在 C++ 里都会由编译器来选择一个整数型别作为其底层型别。对于像 Color 这样的不限范围的枚举型别：

```
enum Color { black, white, red };
```

编译器会选择 char 作为其底层型别，因为只有三个值需要表示。但有些枚举型别取值范围就大得多，例如：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

这里，需要表示的取值范围是从 0 到 0xFFFFFFFF。除非是在不常见的某些机型上（其中 char 包含至少 32 比特），编译器一般都会选择比 char 更大尺寸的整数型别来表示 Status 的取值。

为了节约使用内存，编译器通常会为枚举型别选用足够表示枚举量取值的最小底层型别。在某些情况下，编译器会用空间来换取时间，而在这样的情况下，它们可能会不选择只具备最小可容尺寸的型别，但是它们当然需要具备优化空间的能力。为了使这种设计成为可能，C++98 就只提供了枚举型别定义（即列出所有枚举量）的支持，枚举型别声明则不允许。这么一来，编译器就可能在枚举型别被使用前，逐个地确定其底层型别选择哪一种。

但是前置声明能力的缺失还是会造成一些弊端。其中最值得一提的大概就是它会增加编译依赖性了。再考虑一下 Status 枚举型别：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              audited = 500,
              indeterminate = 0xFFFFFFFF
            };
```

可能整个系统都会因此而需要重新编译，即使只有一个子系统（可能只是一个函数）用到了这个新的枚举量。这是人们讨厌的事情。而这种事情利用 C++11 中为枚举型别提供的前置声明能力即可破除。举例来说，下面是一个完全成立的限定作用域的枚举型别声明，还有一个形参取用的函数：

```
enum class Status;                          // 前置声明

void continueProcessing(Status s);          // 取用前置声明的枚举型别
```

若头文件中包含了这些声明，则在 Status 定义发生了修订时，就不会要求重新编译。犹有进者，即使 Status 被修改了（例如，加了一个 audited 枚举量），但是 continueProcessing 的行为未受影响（例如，由于 continueProcessing 并未使用 audited），则 continueProcessing 的实现也同样无须重新编译。

可是如果编译器需要在枚举型别被使用以前就知晓其尺寸，为什么 C++11 中的枚举型别就可以进行前置声明，C++98 中就不行呢？答案倒也简单：限定作用域的枚举型别的底层型别是已知的；而对于不限范围的枚举型别，你可以指定这个底层型别。

默认地，限定作用域的枚举型别的底层型别是 int：

```
enum class Status;              // 底层型别是 int
```

如果默认型别不合你意，你可以推翻它：

```
enum class Status: std::uint32_t;       // Status 的底层型别是 std::uint32_t
                                        // （该型别在 <cstdint> 中包含）
```

采用这两种方法中的任何一种，编译器都能知晓限定作用域的枚举型别中的枚举量尺寸。

如果要指定不限范围的枚举型别的底层型别，做法和限定作用域的枚举型别一样。这样做了以后，不限范围的枚举型别也能够进行前置声明了：

```
enum Color: std::uint8_t;       // 不限范围的枚举型别的前置声明
                                // 底层型别是 std::uint8_t
```

底层型别指定同样也可以在枚举型别定义中进行：

```
enum class Status: std::uint32_t {   good = 0,
                                     failed = 1,
                                     incomplete = 100,
                                     corrupt = 200,
                                     audited = 500,
                                     indeterminate = 0xFFFFFFFF
                                 };
```

说了这么多有关限定作用域的枚举型别的事实，比如避免名字空间污染，不会无意中进行隐式型别转换等，可能你听到这样的话会吃惊：在至少一种情况下，不限范围的枚举型别还是有用的：那就是当需要引用 C++11 中的 `std::tuple` 型别的各个域时。例如，假设我们要为一个社交网站准备一个元组来持有名字、电子邮件和声望值：

```
using UserInfo =                // 型别别名，参见条款 9
  std::tuple<std::string,       // 名字
             std::string,       // 电子邮件
             std::size_t> ;     // 声望值
```

注释说明了元组中的每个域代表着什么，但是如果你在另一个源文件中看到以下的代码，注释就没有什么太大作用：

```
UserInfo uInfo;                         // std::tuple 型别对象
...

auto val = std::get<1>(uInfo);          // 取用域 1 的值
```

作为程序员，你要跟进的事项很多。你真的应该记住域 1 对应用户的电子邮件吗？我并不这么认为。而采用一个不限范围的枚举型别和域序数关联就可以消解这种记忆需要：

```
enum UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                         // 同前
...

auto val = std::get<uiEmail>(uInfo);    // 啊！取得电子邮件对应的域了
```

以上代码能够运作，有赖于 UserInfoFields 向 std::size_t 的隐式型别转换，转换结果就是 std::get 要求的型别。

而采用限定作用域的枚举型别版本的对应代码就啰嗦多了：

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };

UserInfo uInfo;                                  // 同前
...

auto val =
 std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
          (uInfo);
```

要想不那么啰嗦，就得写个函数，以枚举量为形参并返回其对应的 std::size_t 型别的值，这样做会不太好办。std::get 是个模板，而你传入的值是一个模板形参（请注意这里用的是尖括号而非小括号），所以这个将枚举量变换成 std::size_t 型别值的函数必须在编译期就计算出结果。条款 15 会说明，这就意味着必须使用 constexpr 函数。

其实，光 constexpr 函数还不够，因为它需要配合任何枚举型别，因此还得用 constexpr 函数模板。还有，如果我们做了这样的泛化，那返回值也需要泛化才行。这样就不能返回 std::size_t，而需要返回枚举型别的底层型别才行。这个型别可以通过 std::underlying_type 型别特征取得（有关型别特征的信息，参见条款 9）。最后，我们还得把它声明为 noexcept（参见条款 14），因为我们知道它从不会生成异常。如此这般的结果是一个名为 toUType 的函数模板，取用任意枚举量并以编译期常量形式返回其值：

```
template<typename E>
constexpr typename std::underlying_type<E>::type
  toUType(E enumerator) noexcept
{
  return
    static_cast<typename
                std::underlying_type<E>::type>(enumerator);
}
```

C++14 中，toUType 可以作以下简化：把 typename std::underlying_type<E>::type 替换成时髦的 std::underlying_type_t（参见条款 9）：

```
template<typename E>                             // C++14
constexpr std::underlying_type_t<E>
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

更时髦的 auto 返回值（参见条款 3）在 C++14 中也合法：

```
template<typename E>  // C++14
constexpr auto
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

无论这个函数模板怎样写就，toUType 让我们可以用下面的方式来访问元组中的一个域：

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

即使这样写，使用限定作用域的枚举型别还是比不限范围的枚举型别打的字要多，可是，它还是避免了名字空间污染和无意进行的涉及枚举量的隐式型别转换。在众多情况下，你可能会同意，多打一些字尚算是一种合理的代价，付出这样的代价所获得的能力，是避免了枚举型别技术带来的陷阱。毕竟在这种技术刚发明出来的时代，最尖端的数据通信还在用 2400 波特率的调制解调器呢。

要点速记

- C++98 风格的枚举型别，现在称为不限范围的枚举型别。
- 限定作用域的枚举型别仅在枚举型别内可见。它们只能通过强制型别转换以转换至其他型别。
- 限定作用域的枚举型别和不限范围的枚举型别都支持底层型别指定。限定作用域的枚举型别的默认底层型别是 int，而不限范围的枚举型别没有默认底层型别。
- 限定作用域的枚举型别总是可以进行前置声明，而不限范围的枚举型别却只有在指定了默认底层型别的前提下才可以进行前置声明。

条款 11：优先选用删除函数，而非 private 未定义函数

如果你写了代码给其他程序员用，并且你想阻止他们调用某个特定函数的话，那你只需不要声明该函数即可。函数未经声明，不可调用。易如反掌。但是有时 C++ 会替你声明函数，而如果你要阻止客户调用这些函数，可就没有那么易如反掌了。

这种情况仅发生在“特种成员函数”身上，即 C++ 会在需要时自动生成的成员函数。条款 17 会讨论这些函数的细节，但是目前，我们只需考虑复制构造函数和复制赋值运算符。本章的主要篇幅都在讨论 C++98 中一些常用实践，但是它们已经被 C++11 中的更佳实践所取代。而在 C++98 中，如果你想要压制一个成员函数的使用，那多半会是复制构造函数、复制赋值运算符或它们两者。

C++98 中为了阻止这些函数被使用，采取的做法是声明其为 private，并且不去定义它们。举例来说，在接近 C++ 标准库中输入输出流库继承谱系基类部位的，是类模板 basic_ios，所有的输入流和输出流都继承自该类（可能是间接继承）。对输入流和输出流进行复制是不可取的。因为这种操作究竟要落实成什么样的行为，是不清楚的。例如，一个 istream 对象表示的是一个输入值流，其中可能有一部分已被读取，而有一部分未来有要读取的可能。如果要对 istream 对象进行复制，那是不是需要复制所有已经读取过的值以及所有未来要读的值呢？对于这一类问题，最好的处理方式就是定义它们不存在。而这正是禁止复制流这件事要做的。

为了让输入流和输出流类成为不可复制的，在 C++98 中的 basic_ios 是像下面这样规定的（连注释也是标准的一部分）：

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
...

private:
  basic_ios(const basic_ios& );                 // not defined
  basic_ios& operator=(const basic_ios&);       // not defined
};
```

译注 1

通过将这些函数声明为 private，就阻止了客户去调用它们。而故意不去定义它们，就意味着如果一段代码仍然可以访问它们（如成员函数，或类的友元）并使用了它们，链接阶段就会由于缺少函数定义而告失败。

在 C++11 中，有更好的途径来达成效果上相同的结果：使用“= delete”将复制构造函数和复制赋值运算符标识为**删除函数**（deleted function）。以下是 C++11 中关于 basic_ios 规定的同一片段：

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  ...
```

译注 1：标准文本，注释不在正文译出。“not defined”意为“未定义”。

```
    basic_ios(const basic_ios& ) = delete;
    basic_ios& operator=(const basic_ios&) = delete;
    ...
};
```

删除函数和将函数声明为 private 看起来只是风格不同的选择，可是还有更多值得思考的微妙之处。删除函数无法通过任何方法使用，所以即使成员和友元函数中的代码也会因试图复制 basic_ios 型别对象而无法工作。这对于 C++98 的行为是一种改进，因为在 C++98 中，后面这种不当使用直到链接阶段才能诊断出来。

习惯上，删除函数会被声明为 public，而非 private。这样做是有理由的。当客户代码尝试使用某个成员函数时，C++ 会先校验可访问性，后校验删除状态。这么一来，当客户代码试图调用某个 private 删除函数时，有些编译器只会抱怨该函数为 private，尽管函数的可访问性并不影响其是否可用。在修订遗留代码，把 private 未定义函数替换成删除函数时，尤其应该把这一点记牢，因为把新函数声明成 public，会得到较好的错误信息。

删除函数的一个重要优点在于，任何函数都能成为删除函数，但只有成员函数能声明成 private。举例来说，假定我们有一个非成员函数，取用一个整数，并返回其是否是个幸运数：

```
bool isLucky(int number);
```

C++ 的 C 渊源决定了把可以凑合看作是数值的型别，都可以隐式转型到 int，但有些调用尽管可以编译，但语义上却了无意义：

```
if (isLucky('a')) …          // 'a' 是个幸运数吗?
if (isLucky(true)) …         // "true" 又如何 ?
if (isLucky(3.5)) …          // 是不是应该先截断为 3 再检查是否为幸运数?
```

如果幸运数真的必须是整数，我们会想要阻止上面这样的调用通过编译。

而有一个办法就是为我们想要滤掉的型别创建删除重载版本：

```
bool isLucky(int number);          // 原始版本
bool isLucky(char) = delete;       // 拒绝 char 型别
bool isLucky(bool) = delete;       // 拒绝 bool 型别
bool isLucky(double) = delete;     // 拒绝 double 和 float 型别
```

在形参为 double 的重载版本后面的注释说，这个声明会同时拒绝 double 和 float 型别，

这可能会让你感觉意外。但这样说的话你就不会意外：如果你能想起，当 float 型别面临转型到 int 还是 double 型别的选择时，C++ 会优先转型到 double 型别。这么一来，对于 isLucky 的调用如果传入一个 float，则会调用 double 型别形参的重载版本，而不是 int 的那个。好吧，只能说它会尝试调用 double 型别形参的重载版本。但由于这个重载版本已经是个删除版本，所以编译就被阻止了。

尽管删除函数不可被使用，但它们还是程序的一部分。因此，它们在重载决议时还是会纳入考量。也正因此，如果删除函数按如上定义，对 isLucky 不可取的调用都会被拒绝编译：

```
if (isLucky('a')) …          // 错误！调用了删除函数

if (isLucky(true)) …         // 错误！

if (isLucky(3.5f)) …         // 错误！
```

还有一个妙处是删除函数能做到而 private 成员函数做不到的，那就是阻止那些不应该进行的模板具现。举例来说，假设你需要一个和内建指针协作的模板（这里暂时不去理会第 4 章中有关优先选用智能指针，而非裸指针的建议）：

```
template<typename T>
void processPointer(T* ptr);
```

而指针世界中有两个异类。一个是 void* 指针，因为无法对其执行提领、自增、自减等操作。还有一个是 char* 指针，因为它们基本上表示的是 C 风格的字符串，而不是指涉到单个字符的指针。而特殊情况往往需要特殊处理，在 processPointer 模板中，我们假定这样的特殊处理手法就是在采用这两个型别时拒绝调用。即，不可以使用 void* 和 char* 来调用 processPointer。

这个原则很好贯彻，只需删除这些具现：

```
template<>
void processPointer<void>(void*) = delete;

template<>
void processPointer<char>(char*) = delete;
```

那么，如果使用 void* 和 char* 来调用 processPointer 是非法的，那么很可能使用 const void* 和 const char* 来调用 processPointer 也是非法的，所以基本上这些具现也该删除：

```
template<>
void processPointer<const void>(const void*) = delete;
```

```
template<>
void processPointer<const char>(const char*) = delete;
```

如果你想来个斩草除根，那你还要删除 `const volatile void*` 和 `const volatile char*` 这些重载版本，还有那些指涉到其他标准字符型别的指针的重载版本，这些标准字符型别包括：`std::wchar_t`、`std::char16_t` 和 `std::char32_t`。

很有意思的是，如果是类内部的函数模板，并且你想通过 `private` 声明来禁用某些具现（C++98 中的老生常谈），这是做不到的，因为你不可能给予成员函数模板的某个特化以不同于主模板的访问层级。如果 `processPointer` 是在 `Widget` 内部的一个成员函数模板，而你想禁止使用 `void*` 指针来调用之，下面就是 C++98 的做法，尽管这是通不过编译的：

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }

private:
  template<>                                    // 错误!
  void processPointer<void>(void*);

};
```

问题在于，模板特化是必须在名字空间作用域而非类作用域内撰写的。这个毛病在删除函数身上压根就不会表现出来，因为一来它们根本不需要不同的访问层级，二来也因为成员函数模板可以在类外（即名字空间作用域）被删除：

```
class Widget {
public:
  …
  template<typename T>
  void processPointer(T* ptr)
  { … }
  …

};

template<>                                              // 仍然具备 public 访问层级
void Widget::processPointer<void>(void*) = delete;      // 但被删除了
```

事实上，C++98 中把函数声明为 `private` 并且不去定义，这样的实践想要的就是 C++11 中的删除函数实际达成的效果。前者作为后者的一种模拟动作，当然不如本尊来得好用。比如，前者无法应用于类外部的函数，也不总是能够应用于类内部的函数。就算它能用，也可能直到链接阶段才发挥作用。所以，请始终使用删除函数。

要点速记

- 优先选用删除函数，而非 private 未定义函数。
- 任何函数都可以删除，包括非成员函数和模板具现。

条款 12：为意在改写的函数添加 override 声明

C++ 中的面向对象编程世界是围绕着类、继承和虚函数的基础上演化出来的。在这个世界中最基本的理念就是：在派生类中虚函数实现，会改写基类中对应虚函数的实现。在这个前提下，一旦意识到虚函数改写这件事有多么容易出错，就不免令人沮丧。语言的这个部分设计的，怎么说呢，简直到了向墨菲定律致敬的地步。[译注 2]

由于“改写”（override）和“重载”（overload）读起来很像，尽管这是两个毫不相干的概念，我们还是要澄清，正是虚函数改写，使得通过基类接口调用派生类函数成为了可能：

```
class Base {
public:
  virtual void doWork();                // 基类中的虚函数
  ...
};

class Derived: public Base {
public:
  virtual void doWork();                // 改写了 Base::doWork
  ...                                   // （“virtual”在这里可写可不写）
};

std::unique_ptr<Base> upb =             // 创建基类指针，指涉到派生类对象
  std::make_unique<Derived>();          // std::make_unique 的相关信息
                                        // 请参见条款 21
...

upb->doWork();                          // 通过基类指针调用 doWork；
                                        // 结果是派生类函数被调用
```

而如果要这个改写动作真的发生，有一系列要求必须满足：

- 基类中的函数必须是虚函数。
- 基类和派生类中的函数名字必须完全相同（析构函数例外）。

译注 2：墨菲定律（Murphy's Law），意思是一件事情只要可能出错，那就一定会出错。

- 基类和派生类中的函数形参型别必须完全相同。
- 基类和派生类中的函数常量性（constness）必须完全相同。
- 基类和派生类中的函数返回值和异常规格必须兼容。

而除了这些限制，也就是 C++98 给出的限制，C++11 又加了一条。

- 基类和派生类中的函数引用饰词（reference qualifier）必须完全相同。成员函数引用饰词是 C++11 中鲜为人知的语言特性，所以如果你从来没有听说过这个概念也不必惊诧。它们是为了实现限制成员函数仅用于左值或右值。带有引用饰词的成员函数，不必是虚函数：

```
class Widget {
public:
  ...
  void doWork() &;          // 这个版本的 doWork 仅在 *this 是左值时调用

  void doWork() &&;         // 这个版本的 doWork 仅在 *this 是右值时调用
};

...

Widget makeWidget();        // 工厂函数（返回右值）

Widget w;                   // 普通对象（左值）

...

w.doWork();                 // 以左值调用 Widget::doWork
                            // （即 Widget::doWork &）

makeWidget().doWork();      // 以右值调用 Widget::doWork
                            // （即 Widget::doWork &&）
```

后面我会针对带有引用饰词的成员函数做更多讨论，但目前只需记住，如果基类中的虚函数带有引用饰词，则派生类要对该函数进行改写版本必须也带有完全相同的引用饰词。如果不然，那么这些声明了的函数在派生类依然存在，只是它们不会改写基类中的任何函数。

对于改写有着这么多的要求，这就意味着小的错误可以造成大的偏差。包含着改写方面错误的代码通常都是合法的，但表达的意思却并不符合程序员的初衷。这么一来，就不能依赖编译器来告知你做错事情了。举例来说，下面的代码完全合法，而且乍看之下也合情合理，但是它里面连一个虚函数改写动作都没有，没有任何一个派生类中的函数是和基类函数绑定的。你能找到每种情况下的问题所在吗？也就是说，这些派生类函数中的每一个，是由于什么原因没有达到改写基类中同名函数的目的？

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1();
  virtual void mf2(unsigned int x);
  virtual void mf3() &&;
  void mf4() const;
};
```

要不要一些提示？

- Base 中的 mf1 是声明为 const 的，而在 Derived 中的没有。
- Base 中的 mf2 的形参型别是 int，而 Derived 中的则是 unsigned int。
- Base 中的 mf3 带有左值引用饰词，而 Derived 中的则带有右值引用饰词。
- Base 中的 mf4 未声明为虚函数。

你可能会想，“嘿，在实践中，这些都会触发编译器警告的，所以我不用担心啊。”也许确实如此，但也许并非如此。在我用以校验的编译器中，有两个对于以上代码一声不吭地就接受了，并且这还是在所有警告选项都打开的前提下（其他编译器确实给出了其中一些问题的警告，但也不全）。

由于对于声明派生类中的改写，保证正确性很重要，而出错又很容易，C++11 提供了一种方法来显式地标明派生类中的函数是为了改写基类版本，为其加上 override 声明。如果把这一点应用于上例，就得到了下面这样的派生类：

```
class Derived: public Base {
public:
  virtual void mf1() override;
  virtual void mf2(unsigned int x) override;
  virtual void mf3() && override;
  virtual void mf4() const override;
};
```

这个版本显然不可能通过编译，因为一旦写成这样，编译器就会吹毛求疵地检视所有与改写相关的问题。而这正符合你的本意，也是为什么你把所有意在改写的函数加上 override 声明的理由。

加了 override 声明并且能够通过编译的代码，是下面这样的（假定目标是让 Derived 中的所有函数都能改写 Base 中的虚函数）：

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  virtual void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1() const override;
  virtual void mf2(int x) override;
  virtual void mf3() & override;
  void mf4() const override;            // 加个“virtual”没问题，但也没必要
};
```

请注意，在这个例子中，为了让代码运作，有一处是把 Base 中的 mf4 声明成虚函数。大多数改写相关的错误都是在派生类中，但在基类上出问题也是完全可能的。

为所有意在改写的派生类函数上加 override 声明，好处并不止让编译器在你想要改写的函数实际上并未改写时提醒你，它还可以在你打算更改基类中虚函数的签名时，衡量一下波及的影响面。如果派生类中到处都加了 override 声明，则你可以直接改掉签名并重新编译，你就能看到你自己造成了多大破坏（数一数多少派生类无法编译就知道了），从而决定是否在这样的麻烦之下值得去做这么一个签名更改。离开了 override 声明，你就只能指望拥有一个完备的单元测试集了。因为，正如我们所看到的那样，那些本意是要改写实际上却并未改写基类函数的派生类虚函数，不一定能触发编译器警告。

C++ 一向就有关键字，而 C++11 又加了两个**语境关键字**（contextual keyword）：override 和 final。[注2] 它们的特色是，语言保留这两个关键字，但仅在特定语境下保留。对于 override 的情况而言，它仅于出现在成员函数声明的末尾时才有保留意义。这就意味着，如果你有一些遗留代码，其中已经用过 override 这个名字的话，你不必为了升级到 C++11 而改名：

```
class Warning {                 // C++98 中可能的遗留代码
public:
  ...
  void override();              // 在 C++98 和 C++11 都合法（意义也相同）
```

注 2：将 final 应用于虚函数，会阻止它在派生类中被改写。final 也可以被应用于一个类，在这种情况下，该类会被禁止用作基类。

```
  ...
};
```

有关 override 的事情就说完了，但有关成员函数引用饰词的事情还没说完。我保证过后面提供有关它们的更多信息，这里就是后面。

如果我们想撰写一个函数，仅接受传入左值实参，则我们会声明一个非 const 左值引用形参：

```
void doSomething(Widget& w);          // 仅接受左值的 Widgets 型别
```

如果我们想撰写一个函数，仅接受传入右值实参，则我们会声明一个右值引用形参：

```
void doSomething(Widget&& w);         // 仅接受右值的 Widgets 型别
```

成员函数引用饰词的作用就是针对发起成员函数调用的对象，即 *this，加一些区分度。这和在成员函数声明末尾加一个 const 的情形一模一样：后者表明发起成员函数调用的对象，即 *this，应为 const。

对于带引用饰词的成员函数的需要并不常见，但也是有可能会出现的。举例来说，假设我们的 Widget 类中有个 std::vector 型别的数据成员，我们提供一个访问器函数让客户能对这个数据成员直接访问：

```
class Widget {
public:
  using DataType = std::vector<double>;       // 关于此处的 "using" 参见条款 9
  ...

  DataType& data() { return values; }
  ...

private:
  DataType values;
};
```

这个设计很难说得上什么封装性，但我们先抛开这个问题不谈，考虑一下客户代码那里会怎么做：

```
Widget w;
...

auto vals1 = w.data();          // 把 w.values 复制到 vals1
```

Widget::data 的返回值型别是一个左值引用（更准确地说，是个 std::vector<double>&），并且因为左值引用定义为左值，我们实际上是使用一个左

值来初始化 vals1 的。这么一来，vals1 就是以 w.values 为基础进行复制构造的，一如注释所言。

现在假设我们有个创建 Widget 型别对象的工厂函数，

```
Widget makeWidget();
```

而我们又想使用 makeWidget 返回的那个 Widget 里面的 std::vector 型别对象来初始化一个变量：

```
auto vals2 = makeWidget().data();      // 把 Widget 中的 values 复制到 vals2
```

再一次，Widget::data 返回一个左值引用；再一次，左值引用对应一个左值；所以，再一次，新对象（vals2）以 Widget 中的 values 为基础进行复制构造。不过，这一次，Widget 是 makeWidget 返回的一个临时对象（即一个右值），所以复制其中的 std::vector 型别对象纯属浪费时间之举。比较好的做法是移动而非复制，但是，因为 data 返回的是左值引用，C++ 语言规则会要求编译器生成执行复制的代码（这里有一点点可实施优化的回旋余地，称为“准规则”，但如果你指望编译器帮你找到利用这些旁门左道的路子，可不是高明的做法）。

那么，真正需要的其实是某种解决办法，可以指定让 data 在右值 Widget 上调用时，结果成为一个右值。而运用引用饰词来对 data 的 Widget 的左值和右值型别进行重载，就实现了这种解决办法：

```
class Widget {
public:
  using DataType = std::vector<double>;
  ...

  DataType& data() &           // 对于左值 Widgets 型别，返回左值
  { return values; }

  DataType data() &&           // 对于右值 Widgets 型别，返回右值
  { return std::move(values); }
  ...

private:
  DataType values;
};
```

请注意 data 的不同重载版本返回型别的不同。左值引用型别的重载版本，返回的是左值引用（即一个左值）；而右值引用型别的重载版本，则返回的是一个临时对象（即一个右值）。这就意味着，客户代码的行为现在符合我们的预期了：

```
auto vals1 = w.data();                    // 调用 Widget::data 的左值重载版本
                                          //  vals1 采用复制构造完成初始化

auto vals2 = makeWidget().data();         // 调用 Widget::data 的右值重载版本
                                          // vals2 采用移动构造完成初始化
```

这个结果固然让人欢欣鼓舞，但是我们不要让这个皆大欢喜的结果喧宾夺主，淡化了我们这个条款的真正主题。这个主题是：无论何时，只要你在派生类中声明了一个函数，并且该函数意在改写基类中的一个虚函数，请确保你给该函数加上 override 声明。

要点速记

- 为意在改写的函数添加 override 声明。
- 成员函数引用饰词使得对于左值和右值对象（*this）的处理能够区分开来。

条款 13：优先选用 const_iterator，而非 iterator

const_iterator 是 STL 中相当于指涉到 const 的指针的等价物。它们指涉到不可被修改的值。只要有可能就应该使用 const 的标准实践表明，任何时候只要你需要一个迭代器而其指涉到的内容没有修改必要，你就应该使用 const_iterator。

这一点对于 C++98 和 C++11 都成立，但在 C++98 中，const_iterator 得到的支持不够全面。建立它们不容易，而建立好了以后使用它们的方式也受限。比如，假设你想在 std::vector<int> 中搜索第一次出现的 1983（这一年，“C++”取代了“带类的 C”成为了该种程序设计语言的名字），然后在此位置插入值 1998（这一年，首个 ISO C++ 标准被接受）。如果 vector 中并无 1983，那么插入位置将是其末尾处。在 C++98 中使用 iterator 来实现，很容易：

```
std::vector<int> values;

...

std::vector<int>::iterator it =
  std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

但在这里使用 iterator 并非正确选择，因为代码中并无任何地方修改了 iterator 指涉到的内容。修订这段代码以使用 const_iterator 本应是举手之劳，可是在 C++98 中，要费很大工夫。下面是一种在概念上貌似站得住脚的途径，但其实并不正确：

```
typedef std::vector<int>::iterator IterT;       // 一些typedef
typedef std::vector<int>::const_iterator ConstIterT;

std::vector<int> values;

...

ConstIterT ci =
  std::find(static_cast<ConstIterT>(values.begin()),      // 强制型别转换
            static_cast<ConstIterT>(values.end()),        // 强制型别转换
            1983);

values.insert(static_cast<IterT>(ci), 1998);              // 可能无法通过编译，理由见下
```

这些 typedef 语句当然并非必要，但是它们使得代码中的强制型别转换更加容易书写（如果你奇怪我为什么会用 typedef 语句来做展示，而不采纳条款 9 中使用别名声明的建议，那是因为这段示例是 C++98 代码，而别名声明是 C++11 中的新特性）。

在 std::find 的调用语句中之所以要加强制型别转换，是因为 values 是 C++98 中的非 const 容器，而并没有什么简单的办法能从一个非 const 容器得到其对应的 const 容器。这些强制型别转换严格来说并非必须，因为还有一些别的办法能取得对应的 const 容器（例如，你可以将 values 绑定到一个引用到 const 的变量，再在代码中 values 的位置使用该变量），但是无论是采用这种办法还是那种办法，从一个非 const 容器得到其对应的 const 容器总是要费些周章才能办到。

而一旦得到了 const_iterator，往往事态会变得更糟。因为在 C++98 中，插入（或删除）的位置只能以 iterator 指定，而不接受 const_iterator。这就是为什么在上面的代码中，我对（那个我费了九牛二虎之力从 std::find 中取得的）const_iterator 实施了强制转型，让它变回一个 iterator，就是因为如果向 insert 传入一个 const_iterator，编译无法通过。

实话实说，这段演示代码还真可能使编译无法通过，因为从 const_iterator 到 iterator 并不存在可移植的型别转换，连 static_cast 也不行。即使用大招 reinterpret_cast 也无法完成任务（这并不是一个 C++98 的限制，在 C++11 中也一样。const_iterator 就是没法完成到 iterator 的型别转换，不管这种型别转换看起来是多么理所当然）。要生成 const_iterator 同样内容的 iterator，是存在一些可移植的方法的，但是这些方法并不显明，也不总是可用，在本书中不值得加以讨论。总之，希望现在我已经把观点表达清楚了：const_iterator 在 C++98 中是如此不好用，所以也很少有人愿意招惹这个麻烦。所以到头来，程序员遵循的原则不是只要有可能就使用 const，而是只在好用时才使用。而在 C++98 中，const_iterator 恰恰不好用。

C++11 中，这些现象彻底改观了。获取和使用 const_iterator 都变得容易了。容器的成员函数 cbegin 和 cend 都返回 const_iterator 型别，甚至对于非 const 容器也是如此，并且 STL 成员函数若要取用指示位置的迭代器（例如，作插入或删除之用），它们也要求使用 const_iterator 型别。要把原始的、使用 iterator 型别的 C++98 代码修订成使用 const_iterator 型别的 C++11 版本真是太简单了：

```
std::vector<int> values;                                    // 同前

...

auto it =                                                   // 使用 cbegin
  std::find(values.cbegin(),values.cend(), 1983);           // 以及 cend

values.insert(it, 1998);
```

这就是使用 const_iterator 的代码版本，非常方便好用！

只有一种情景下，C++11 对于 const_iterator 的支持显得不够充分，那就是你想撰写最通用化的库代码的时候。这些代码会考虑到，某些容器，或类似容器的数据结构会以非成员函数的方式提供 begin 和 end（还有 cbegin、cend 和 rbegin 等），而不是以成员函数方式。这就是内建数组的情况，也是某些仅以自由函数形式提供接口的第三方库的情况。所以，最通用化的代码会使用非成员函数，而不会假定其成员函数版本的存在性。

举例来说，刚才我们写的这段代码可以写成下面 findAndInsert 模板的通用形式：

```
template<typename C, typename V>
void findAndInsert(C& container,          // 在容器中查找 targetVal
                   const V& targetVal,    // 第一次出现的位置，
                   const V& insertVal)    // 然后在彼处插入 insertVal
{
  using std::cbegin;
  using std::cend;

  auto it = std::find(cbegin(container),          // 非成员函数版本的 cbegin
                      cend(container),            // 非成员函数版本的 cend
                      targetVal);

  container.insert(it, insertVal);
}
```

以上代码在 C++14 中可以完全正常运行，但可惜的是，在 C++11 中却不行。由于在标准化过程中的短视，C++11 仅添加了非成员函数版本的 begin 和 end，而没有添加 cbegin、cend、rbegin、rend、crbegin 和 crend。C++14 纠正了这种短视。

如果你使用的是 C++11，而且你想撰写最通用化的代码，但你使用的库中却都没有提

供非成员函数版本的 cbegin 系列这些缺失的模板，则你可以很容易地写出你自己的实现。举例来说，下面就是非成员函数版本的 cbegin 的一个实现：

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
  return std::begin(container);          // 请仔细看下文的解释
}
```

当你看到非成员函数版本的 cbegin 并没有调用成员函数版本的 cbegin 时，是不是吃了一惊？我其实刚看到时也吓一跳。但顺着逻辑推导一下。这个 cbegin 模板接受一个形参 C，实参型别可以是任何表示类似容器的数据结构，并通过其引用到 const 型别的形参 container 来访问该实参。如果 C 对应一个传统容器型别（例如，std::vector<int>），则 container 就是该容器型别的引用到 const 的版本（例如，const std::vector<int>&）。调用（C++11 提供的）非成员函数版本的 begin 函数并传入一个 const 容器会产生一个 const_iterator，而模板返回的正是这个迭代器。这样实现的好处是它对于那些只提供了 begin 成员函数（C++11 的非成员函数版本的 begin 调用的就是它）而未提供 cbegin 成员函数的容器也适用。这么一来，你就可以把这个非成员函数版本的 cbegin 用在那些仅直接支持 begin 的容器上了。

该模板在 C 是一个内建数组时也适用。在这种情况下，container 成为了一个 const 数组的引用。C++11 的非成员函数版本的 begin 为数组提供了一个特化版本，它返回一个指涉到数组首元素的指针。由于 const 数组的元素都是 const 的，所以若给非成员函数版本的 begin 传入一个 const 数组，则其返回的指针是个指涉到 const 的指针。而指涉到 const 的指针，实际上，就是数组意义下的 const_iterator（欲知模板如何为内建数组做特化，可以查阅条款 1 中有关模板型别推导中有关形参型别是数组引用的情况讨论部分）。

让我们回归初心。本章的要点是鼓励你只要在能使用 const_iterator 的场合下就去使用它。而它的源动力（只要有可能，就应该使用 const）是在 C++11 前就有的。但是在 C++98 中，迭代器实在不好用。在 C++11 中，它变得好用起来，而 C++14 则是把 C++11 中残留的一些任务死角清理并完成了。

要点速记

- 优先选用 const_iterator，而非 iterator。
- 在最通用的代码中，优先选用非成员函数版本的 begin、end 和 rbegin 等，而非其成员函数版本。

条款 14：只要函数不会发射异常，就为其加上 noexcept 声明

在 C++98 中，异常规格可谓喜怒无常的野兽。你必须梳理出一个函数可能发射的所有异常型别，所以，如果函数实现做了改动，那异常规格也难免要修订。而改动异常规格可能会破坏客户代码，因为调用方可能依赖于原先的异常规格。编译器通常不会在保持函数实现、异常规格和客户代码的一致性方面提供什么帮助。绝大多数程序员最终认定，C++98 中的异常规格还是不要招惹为妙。

而在 C++11 形成过程中，逐渐达成了一个共识，那就是关于函数发射异常这件事，真正重要的信息是它到底会不会发射。非黑即白，一个函数或者可能发射异常，或它保证自己不会。这种要么可能要么不可能的泾渭分明，形成了 C++11 异常规格的基础，也实质上把 C++98 异常规格替换掉了（C++98 风格的异常规格仍然成立，但已经它们已经被标为废弃特性了）。在 C++11 中，无条件的 noexcept 就是为了不会发射异常的函数而准备的。

函数是否要加上如此声明，事关接口设计。函数是否会发射异常这一行为，是客户方面关注的核心。调用方可以查询函数的 noexcept 状态，而查询结果可能会影响调用代码的异常安全性或运行效率。这么一来，函数是否带有 noexcept 声明，就和成员函数是否带有 const 声明是同等重要的信息。当你明明知道一个函数不会发射异常却未给它加上 noexcept 声明的话，这就是接口规格缺陷。

但是对不会发射异常的函数应用 noexcept 声明还有一个动机，那就是它可以让编译器生成更好的目标代码。为了理解这背后的理由，只需考察一下 C++98 和 C++11 在表达函数不会发射异常时的差异。考虑一个函数 f，欲向调用方保证它们不会接收到异常。两种表达方式如下：

```
int f(int x) throw();          // f 不会发射异常：C++98 风格

int f(int x) noexcept;         // f 不会发射异常：C++11 风格
```

如果，在运行期，一个异常逸出 f 的作用域，则 f 的异常规格被违反。在 C++98 异常规格下，调用栈会开解至 f 的调用方，然后执行了一些与本条款无关的动作以后，程序执行中止。而在 C++11 异常规格下，运行期行为会稍有不同：程序执行中止之前，栈只是可能会开解。

开解调用栈，和**可能**开解调用栈，这一点点区别对于代码生成造成的影响之大可能出乎人们的意料。在带有 noexcept 声明的函数中，优化器不需要在异常传出函数的前

提下，将执行期栈保持在可开解状态；也不需要在异常溢出函数的前提下，保证所有其中的对象以其被构造顺序的逆序完成析构。而那些以“throw()”异常规格声明的函数就享受不到这样的优化灵活性，和没有加异常规格声明的函数一样。这些可以总结为下述情况：

```
RetType function(params) noexcept;     // 最优化

RetType function(params) throw();      // 优化不够

RetType function(params);              // 优化不够
```

仅仅这个就已经构成充分理由，让你给任何已知不会产生异常的函数加上noexcept声明了。

对于某些函数来说，情况更加典型。移动操作就是个极好的例子。假设你有一段C++98代码，使用一个std::vector<Widget>型别对象，而且不时地会通过push_back向其中加入一些Widget型别对象：

```
std::vector<Widget> vw;

...

Widget w;

...                    // 使用w

vw.push_back(w);       // 将w加入vw

...
```

假定这段代码运作正常，而且你也没兴趣为了升级到C++11而修改它。不过，你的确想要在启用了移动能力的型别上利用C++11中的移动语义来提高遗留代码的性能。这么一来，你就需要保证Widget具有移动操作，不管是你自己撰写，还是去搞清楚其自动生成条件都已经满足（参见条款17）。

当向std::vector型别对象中添加新元素时，可能会空间不够，即std::vector型别对象的尺寸（size）和其容量（capacity）相等的时刻。当这件事发生时，std::vector型别对象会分配一个新的、更大的内存块（chunk）来存储其元素，然后它把元素从现存的内存块转移到新的。在C++98中，这种转移的做法是先把元素逐个地从旧内存复制到新内存，然后将旧内存中的对象析构。这个做法使得push_back能够提供强异常安全保证（strong exception safety guarantee）：如果在复制元素的过程中抛出了异常，则std::vector型别对象会保持原样不变，因为在旧内存中的元素直至所有的元素被成功复制入新内存以后，才会被执行析构。

而在 C++11 中，一个自然而然的优化，就是把针对 std::vector 型别对象元素的复制操作替换成移动操作。不幸的是，这样做是冒了违反 push_back 的强异常安全保证这一风险。如果 *n* 个元素已经从旧内存移出，而在移动第 *n*+1 个元素时抛出了异常，则 push_back 操作无法完成。不过，此时原始的 std::vector 型别对象已经被修改：*n* 个元素已经从其中移出。恢复到原始状态可能不行，因为想要把对象逐个地移回原始内存这个动作本身就有可能产生异常。

这是个严重的问题，因为遗留代码的行为可能会依赖于 push_back 的强异常安全保证。这么一来，C++11 的实现就不能一声不响地把 push_back 内部的复制操作采用移动替代，除非它知道移动操作不会发射异常。而在这种情况下，使用移动替代复制就是安全的，唯一的副作用就是顺带着提高了性能。

std::vector::push_back 利用了这种“能移动则移动，必须复制才复制”（move if you can, but copy if you must）策略，而这并不是标准库中唯一这样做的函数。C++98 中其他因为强异常安全保证而酷炫的函数（std::vector::reserve、std::deque::insert 等）的行为也是一样。所有这些函数都把其中 C++98 的复制操作替换成了 C++11 中的移动操作，但仅在已经移动操作不会发射异常的前提下。但是一个函数怎么能知道移动操作不会产生异常呢？答案显而易见：校验一下，看看这个操作是否带有 noexcept 声明即可。[注3]

swap 函数构成了极其需要 noexcept 声明的另一例子。swap 函数是许多 STL 算法实现的核心组件。在复制赋值运算符中，它也常常被调用。它的广泛使用昭示着针对其实施 noexcept 声明带来的收益是可观的。有意思的是，标准库中的 swap 是否带有 noexcept 声明，取决于用户定义的 swap 是否带有 noexcept 声明。例如，标准库为数组和 std::pair 准备的 swap 函数如下：

```
template <class T, size_t N>
void swap(T (&a)[N],
          T (&b)[N]) noexcept(noexcept(swap(*a, *b)));   // 解释见下

template <class T1, class T2>
struct pair {
  ...
```

注 3：此校验动作相当迂回。像 std::vector::push_back 这样的函数会调用 std::move_if_noexcept，后者是 std::move 的一个变体，它会在一定条件下强型转换至右值型别（参见条款 23），取决于型别的移动构造函数是否带有 noexcept 声明。接下来，std::move_if_noexcept 则会向 std::is_nothrow_move_constructible 求助，该模板特征（参见条款 9）的值由编译器根据移动构造函数是否指定了 noexcept（或 throw()）来设置。

```
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                                noexcept(swap(second,  p.second)));
    ...
};
```

这些函数带有条件式 *noexcept* 声明，它们到底是不是具备 noexcept 性质，取决于它的 noexcept 分句中的表达式是否结果为 noexcept。例如，给定两个 Widget 型别对象数组，要保证对其实施 swap，结果为 noexcept 的前提是对数组中的单个元素实施 swap 的结果为 noexcept，即 Widget 的 swap 需要为 noexcept。由于这个原因，Widget 的 swap 函数的作者就决定了 Widget 型别对象数组的 swap 行为是否为 noexcept。依此类推，它还决定了 Widget 型别对象数组型别的数组的 swap 行为是否为 noexcept，以及更多层的嵌套。相似地，两个含有 Widget 的 std::pair 的 swap 行为是否为 noexcept，也取决于 Widget 的 swap 是否为 noexcept。高阶数据结构的 swap 行为要 noexcept 性质，一般地，仅当构建它的低阶数据结构具备 noexcept 性质时才成立，这个事实应该可以促使你在只要有可能的情况下都让函数带有 noexcept 声明。

讲到这里，可能你会被 noexcept 声明带来的潜在优化机会弄得兴奋不已。不过，此时我必须给你泼泼冷水。优化诚可贵，正确价更高。在本章开始之处，我指出 noexcept 声明乃是函数接口的组成部分，所以你应该只在保证函数实现长期具有 noexcept 性质的前提下，才给予其 noexcept 声明。如果你先是为函数加上了 noexcept 声明，然后又反悔，那你就还没有拿定主意。你如果把 noexcept 从函数声明中移除（即改变了接口），那就会有破坏客户代码的风险。但你也可以变更实现，使得异常可能逸出，在这个前提下保留原始的（不过已经是错误的）异常规格。如果你采用后一种做法，那么如果异常企图脱离函数，程序就会被中止。又或者你也可以让自己屈从于既有实现，而从一开始就舍弃任何会激发你想要变更实现的因素。这几种做法中，没有一种是让人感觉舒服的。

事实是，大多数函数都是异常中立的（exception-neutral）。此类函数自身并不抛出异常，但它们调用的函数则可能会发射异常。当这种情况真的发生时，异常中立的函数会允许该发射的异常经由它传至调用栈的更深一层。异常中立函数永远不具备 noexcept 性质，因为它们可能会发射这种“路过”的异常。这么一来，大多数函数由于这个原因，就天生担不起 noexcept 的称号。

有些函数，怎么说呢，有着不发射异常的自然实现，而对于另一些函数（尤其是移动操作和 swap）具备 noexcept 性质的收益是如此之高，以至于只要有任何可能就应该将

它们的实现加上 noexcept 性质。[注4] 如果你确实能保证某个函数从不发射异常，你绝对应该为它加上 noexcept 声明。

请注意，我说的是，某些函数有着不发射异常的自然实现。如果你硬要扭曲函数的实现，使之符合 noexcept 性质，那你就是主次颠倒、轻重不分、只见树木不见森林……反正就用你爱用的比喻好了。[译注3] 如果一个直截了当的函数实现可能产生异常（例如，调用了一个会抛出异常的函数），而你需要一环扣一环地达成对调用者隐藏这一点（例如，捕获所有异常，将其替换成状态码，或是特殊返回值），这不仅会让你把函数实现弄得更复杂，也几乎肯定会把调用方代码弄得更复杂。例如，调用方可能不得不校验状态码或特殊返回值。这些复杂性带来的时间成本（例如，额外的分支、更大的函数体给指令缓存带来的压力等）会超过任何你希望通过 noexcept 声明实现的提速，更何况你还要承担更难理解和维护的源代码。这是差劲的软件工程实践。

对于某些函数来说，具备 noexcept 性质是如此之重要，所以它们默认就是这样的。在 C++98 中，允许内存释放函数（即 operator delete 或 operator delete[]）和析构函数发射异常，被认为是一种差劲的编程风格。而在 C++11 中，这种风格规则被升级成了一条语言规则。默认地，内存释放函数和所有的析构函数（无论是用户定义的，还是编译器自动生成的）都隐式地具备 noexcept 性质。这么一来，它们就无须加上 noexcept 声明了（这样做并无问题，因为这是无条件的）。析构函数未隐式地具备 noexcept 性质的唯一场合，就是所在类中有数据成员（包括继承而来的成员，以及在其他数据成员中包含的数据成员）的型别显式地将其析构函数声明为可能发射异常的[即为其加上“noexcept(false)”声明]。这样的析构函数很少见。标准库里一个也没有，而如果标准库使用了某个对象（例如，被包含在容器内，或被传递给某个算法），而其析构函数发射了异常，则该程序行为是未定义的。

值得指出的是，有些库的接口设计者会把函数区分成带有宽松契约（wide contract）和带有狭隘契约（narrow contract）的不同种类。带有宽松契约的函数，是没有前置

注4：标准库容器中，移动操作的接口规格并不带有 noexcept 声明。但是，各个实现者是允许为标准库函数加强异常规格的。而且，实践中，为至少一部分容器的移动操作加上 noexcept 声明是常见的。这样的实践印证了本条款的建议。只要发现容器的移动操作有可能写成不抛出异常的，实现者就往往会把这些操作加上 noexcept 声明，即使标准并不要求他们这样做。

译注3：“主次颠倒”原文是“tail wagging the dog”，直译是“尾巴摇狗”；“轻重不分”原文是“putting the cart before the horse”，直译是“货车拉马”。

条件的。要调用这样的函数，无须关心程序状态，对于调用方传入的实参也没有限制。[注5] 带有宽松契约的函数不可能展现出未定义行为。

而带有宽松契约的函数，就是带有狭隘契约的函数了。对于这些函数，如果前置条件被违反，则结果成为未定义的。

如果你在撰写的是个带有宽松契约的函数，并且你知道它不会发射异常，那么遵循本条款的建议，给它加个 noexcept 声明是容易的事。但对于带有狭隘契约的函数来说，情况就有那么点儿微妙。举例来说，假设你正在撰写一个函数 f，带有一个 std::string 型别的形参，并假设 f 的自然实现不会产生异常。这就说明 f 应该加上 noexcept 声明。

但再假设 f 有个前提条件：std::string 型别的形参不得超过 32 个字符。如果向 f 传入一个长度超过 32 的 std::string 型别对象，则行为未定义，因为按照定义，违反前置条件就是会导致未定义行为的。f 并无义务校验这个前置条件，因为函数会断言其前置条件一定能满足（调用方负责保证断言成立）。即使带有前置条件，为 f 加上 noexcept 声明看上去也合情合理：

```
void f(const std::string&  s) noexcept;// 前置条件：s.length() <= 32
```

但再假设 f 的实现者选择去校验前置条件违例的情形。这个校验动作并不是必须的，但也没有禁止，并且校验动作也是有其用途的，例如在做系统测试时就有用。针对被抛出的异常来做调试，通常比企图跟踪未定义行为的原因要容易。但是，一个前置条件违例要怎样报告，才能让测试装置或客户方的错误处理程序检测到呢？一个直截了当的做法是抛出一个“前置条件已违例”异常，但如果 f 已加上了 noexcept 声明，这就不好办了，因为这么一来异常会导致程序中止。也正因为如此，库的设计师们才会区分宽松的和狭隘的契约，并且一般地只把 noexcept 声明保留给那些带有宽松的契约的函数。

最后一点，请让我再尽心尽意地把我最早的观察讲完整，现在讨论一下编译器一般对于识别函数实现及其异常规格的一致性不予支持的问题。考虑以下代码，它完全合法：

```
void setup();                          // 函数在别处定义
void cleanup();
```

注5：“无须关心程序状态”和“没有约束”并不能让已经失去定义的程序重新合法化。例如，std::vector::size 带有宽松契约，但如果你是在让它针对一段强制转换成 std::vector 型别的随机内存块来调用，它也并不一定非要表现出合理行为。因为那个强制型别转换的结果就是未定义的，因而包含着这个强制型别转换动作的程序也就没有任何行为保证了。

```
void doWork() noexcept
{
  setup();                          // 建立要做的工作
  ...                               // 完成要做的工作
  cleanup();                        // 执行清理动作
}
```

这里 doWork 带有 noexcept 声明，尽管它调用了不带 noexcept 声明的函数 setup 和 cleanup。这看起来自相矛盾，但也有可能是 setup 和 clean 已经在文档中写明它们不会发射异常，尽管它们并没有加上这么一个声明。而不加这么一个声明，也可能有充分的理由。例如，它们可能来自于使用 C 语言撰写的库（即使是那些来自 C 标准库中被移入 std 名字空间的函数也都没有加上异常规格，例如，std::strlen 就不带 noexcept 声明）。又或者它们可能来自一个 C++98 库，当时决定不采用 C++98 异常规格，又还没来得及根据 C++11 标准做修订。

由于有着确实的理由使得带有 noexcept 声明的函数依赖于缺乏 noexcept 保证的代码，C++ 允许此类代码通过编译，并且编译器通常不会就此生成警告。

要点速记

- noexcept 声明是函数接口的组成部分，这意味着调用方可能会对它有依赖。
- 相对于不带 noexcept 声明的函数，带有 noexcept 声明的函数有更多机会得到优化。
- noexcept 性质对于移动操作、swap、函数释放函数和析构函数最有价值。
- 大多数函数都是异常中立的，不具备 noexcept 性质。

条款 15：只要有可能使用 constexpr，就使用它

如果 C++11 中引入的新词要评出一个“最令人困惑”大奖，那么 constexpr 很可能获此殊荣。当它应用于对象时，其实就是一个加强版的 const；但应用于函数时，却有着相当不同的意义。将其中的困惑之处彻底解释明白是值得的，因为如果 constexpr 真的反映了你想表达的内容，那么这就是你肯定需要使用它的场合。

表面上看，constexpr 表示的是这样的值：它不仅是 const，而且在编译阶段就已知。可是这种说法并不全面，因为当 constexpr 应用在函数上时，就比它的名字所示的内容更加微妙了。为了不剧透，现在我只能把话说到这儿：关于 constexpr 函数的结果，

你既不能断定它是 const，也不能假定其值在编译阶段就已知。最怪的是，这些都是有意设计的语言特性。也就是说，constexpr 函数不产生 const 或是编译阶段就已知的结果，这是一件好事！

但是，我们且先从 constexpr 对象讲起。这些对象，其实真的具备 const 属性；其实，真的是在编译阶段就已知（严格地说，它们的值是在翻译期间决定的，而翻译不仅包括编译，还包括链接。除非你是撰写 C++ 编译器或链接器的作者，否则这样的严格性对你而言并没有产生什么区别效应，也就是你可以毫无顾虑地认为 constexpr 对象是在编译期间决定的）。

在编译阶段就已知的值拥有种种特权。比如，它们可能被放置在只读内存里，尤其对于嵌入式系统开发工程师来说，这可是非常重要的语言特性。更广泛的应用场景里，在编译阶段就已知的常量整型值可以用在 C++ 要求整型常量表达式的语境中。这些语境包括数组的尺寸规格、整型模板实参（包括 std::array 型别对象的长度）、枚举量的值、对齐规格等。如果你需要使用一个变量来表示这些值，你当然会想把它们声明为，因为这么一来编译器就能保证它们具备一个编译期的值：

```
int sz;                                 // 非 constexpr 变量
...
constexpr auto arraySize1 = sz;         // 错误！ sz 的值在编译期未知
std::array<int, sz> data1;              // 错误！一样的问题
constexpr auto arraySize2 = 10;         // 没问题，10 是个编译期常量
std::array<int, arraySize2> data2;      // 没问题，arraySize2 是个 constexpr
```

请注意，const 并未提供和 constexpr 同样的保证，因为 const 对象不一定经由编译期已知值来初始化：

```
int sz;                                 // 仍是非 constexpr 变量
...
const auto arraySize = sz;              // 没问题，arraySize 是 sz 的一个 const 副本
std::array<int, arraySize> data;        // 错误！ arraySize 的值非编译期可知
```

一言以蔽之，所有 constexpr 对象都是 const 对象，而并非所有的 const 对象都是 constexpr 对象。如果你想让编译器提供保证，让变量拥有一个值，用于要求编译期常量的语境，那么能达到这个目的的工具是 constexpr，而非 const。

而 constexpr 对象的使用场景中如果涉及 constexpr 函数，那就更加有意思了。这样的函数在调用时若传入的是编译期常量，则产出编译期常量。如果传入的是直至运行期才知晓的值，则产出运行期值。按这样的说法，好像你无法预知它们的行为，但这么理解是不对的。正确的理解方式是以下这样：

- constexpr 函数可以用在要求编译期常量的语境中。在这样的语境中，若你传给一个 constexpr 函数的实参值是在编译期已知的，则结果也会在编译期间计算出来。如果任何一个实参值在编译期未知，则你的代码将无法通过编译。

- 在调用 constexpr 函数时，若传入的值有一个或多个在编译期未知，则它的运作方式和普通函数无异，亦即它也是在运行期执行结果的计算。这意味着，如果函数执行的是同样的操作，仅仅应用的语境一个是要求编译期常量的，一个是用于所有其他值的话，那就不必写两个函数。constexpr 函数就可以同时满足所有需求。

假设我们需要一种数据结构来持有某个实验的结果，而该实验可以采用多种途径进行。例如，在实验过程中，照明级别可以设定为高、低或关，风扇转速和温度等也一样。如果实验中有 n 种相关的环境条件，而每种有三个可能状态，那么一共会产生 3^n 种组合。这么一来，欲存储所有条件组合下的实验结果就需要足以放置 3^n 个值的空间。假定所有结果都是个 int，且 n 在编译期可知（或可计算），那么 std::array 就是一种合理的数据结构选择。不过，我们需要某种途径在编译期计算出 3^n。C++ 标准库提供了 std::pow，它确实在数学上完成了我们需要的功能，但是对于我们的目标来说，它还存在两个问题。首先，std::pow 的工作对象是浮点型别，而我们需要的是一个整型结果。其次，std::pow 并不是 constexpr 的（换言之，即使向它传入的是编译期的值，它也不保证返回编译期的结果），所以我们不能使用它来指定 std::array 的尺寸。

幸运的是，我们可以撰写出所需的 pow 函数。后面我会展示如何撰写，但现在我们先来看看它要如何被声明和使用：

```
constexpr                                          // pow 是个 constexpr 函数
int pow(int base, int exp) noexcept                // 且不会抛出异常
{
  ...                                              // 实现见后
}

constexpr auto numConds = 5;                       // 条件数量

std::array<int, pow(3, numConds)> results;         // results 有
                                                   // 3^numConds 个元素
```

回忆一下，pow 前面写的那个 constexpr 并不表明 pow 要返回一个 const 值，它表明的是如果 base 和 exp 是编译期常量，pow 的返回结果就可以当一个编译期常量使用；

如果 base 和 exp 中有一个不是编译期常量，则 pow 的返回结果就将在执行期计算。这么一来就意味着，pow 不仅可以用在诸如编译期计算之物 std::array 的尺寸这样的语境，还可以用于执行期语境，例如：

```
auto base = readFromDB("base");                    // 在执行期取得这些值
auto exp = readFromDB("exponent");

auto  baseToExp = pow(base, exp);                  // pow 函数在执行期被调用
```

由于 constexpr 函数必须在传入编译期常量时能够返回编译期结果，它们的实现就必须加以限制。而在 C++11 和 C++14 中，这样的限制还有所不同。

在 C++11 中，constexpr 函数不得包含多于一个可执行语句，即一条 return 语句。这个限制听上去限制极大，但其实没有那么大，因为我们还有两条技巧可以用来拓展 constexpr 函数的表达力。首先，条件运算符 ?: 可以用于需要使用 if-else 语句；其次，用到循环的地方可以用递归代替。所以，pow 可以像下面这样实现：

```
constexpr int pow(int base, int exp) noexcept

{
  return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

这么写虽然可以运作，但是除非是忠实的函数式程序员，不会有人觉得这样的写法很漂亮。C++14 中，限制条件大大地放宽了，所以下面这样的实现也成为可能了：

```
constexpr int pow(int base, int exp) noexcept    // C++14
{
  auto result = 1;
  for (int i = 0; i < exp; ++i) result *= base;

  return result;
}
```

constexpr 函数仅限于传入和返回字面型别（literal type），意思就是这样的型别能够持有编译期可以决议的值。在 C++11 中，所有的内建型别，除了 void，都符合这个条件。但是用户自定义型别同样可能也是字面型别，因为它的构造函数和其他成员函数可能也是 constexpr 函数。

```
class Point {
public:
  constexpr Point(double xVal = 0, double yVal = 0) noexcept
  : x(xVal), y(yVal)
  {}

  constexpr double xValue() const noexcept { return x; }
  constexpr double yValue() const noexcept { return y; }
```

```
  void setX(double newX) noexcept { x = newX; }
  void setY(double newY) noexcept { y = newY; }

private:
  double x, y;
};
```

此处，Point 的构造函数被声明成了 constexpr 函数，由于传入它的实参在编译期可知，构造出来的 Point 对象的数据成员，其值也是在编译期可知的。如此初始化出来的 Point 对象，也自然具备了 constexpr 属性。

```
constexpr Point p1(9.4, 27.7);          // 没问题，在编译期“运行”
                                        // constexpr 构造函数

constexpr Point p2(28.8, 5.3);          // 同样没问题
```

类似地，访问器 xValue 和 yValue 也可以声明为 constexpr，原因在于，若它们是通过一个在编译期已知其值的 Point 对象，即一个 constexpr Point 对象来调用的话，数据成员 x 和 y 的值就可以在编译期获知。从而，就能够撰写出这样的 constexpr 函数，它调用 Point 的访问器并使用其返回结果来初始化 constexpr 对象：

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
  return { (p1.xValue() + p2.xValue()) / 2,     // 调用 constexpr
           (p1.yValue() + p2.yValue()) / 2 };   // 成员函数
}

constexpr auto mid = midpoint(p1, p2);          // 使用 constexpr 函数的结果
                                                // 来初始化 constexpr 对象
```

这很让人激动。这就意味着，对象 mid，尽管在其初始化过程中涉及了构造函数、访问器、还有个非成员函数的调用，却仍可以在只读内存中得以创建！这就意味着，你可以将一个诸如 mid.xValue()*10 的表达式运用到模板形参中，或是指定枚举量的表达式中！[注6] 这就意味着，传统上那条划在编译期完成的工作和运行期完成的工作之间相当严格的界线已经开始变得模糊。并且，有些传统上会在运行期完成的工作已经可以迁至编译期完成。迁过去的代码越多，你的软件就会运行得越快（不过，编译会用更久）。

由于 constexpr 函数必须在传入编译期常量时能够返回编译期结果，它们的实现就必须加在 C++11 中，有两个限制使得 Point 的成员函数 setX 和 setY 无法声明为 constexpr。首先，它们修改了操作对象，可是在 C++11 中，constexpr 函数都隐式

注 6：因为 Point::xValue 返回的是 double，mid.xValue()*10 的型别也是 double。而浮点型别固然不能用于具现模板，或指定枚举值，但却可以用作一个更大表达式的一部分来生成整数型别。例如，static_cast<int>(mid.xValue()*10) 就可以用于具现模板、以及指定枚举值了。

地被声明为 const 的了。[译注4] 其次，它们的返回型别是 void，而在 C++11 中，void 并不是个字面型别。不过这两个限制在 C++14 中都被解除了，所以在 C++14 中，就连设置器也可以声明为 constexpr。

```
class Point {
public:
  ...

  constexpr void setX(double newX) noexcept     // C++14
  { x = newX; }

  constexpr void setY(double newY) noexcept     // C++14
  { y = newY; }

  ...

};
```

所以这就使得人们能够写出下面这样的代码：

```
// 返回 p 相对于原点的中心对称点 (C++14)
constexpr Point reflection(const Point& p) noexcept
{
  Point result;                                 // 创建一个非 const 的 Point 对象

  result.setX(-p.xValue());                     // 设置其 x 和 y 成员的值
  result.setY(-p.yValue());

  return result;                                // 返回 result 的副本
}
```

而客户代码可以是这样：

```
constexpr Point p1(9.4, 27.7);                  // 同上
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1, p2);

constexpr auto reflectedMid =           // reflectedMid 的值为
  reflection(mid);                      // (-19.1 -16.5)，
                                        // 且在编译期就已知
```

本条款给出的建议是：只要有可能使用 constexpr，就使用它。看到这里，我希望你已经清楚地了解了原因。比起非 constexpr 对象或 constexpr 函数而言，constexpr 对象或是 constexpr 函数可以用在一个作用域更广的语境中。通过在所有可能的情况下运用 constexpr，你也就将代码中对象和函数能够使用的情景范围拓展至最大了。

译注 4：这里说的并非函数返回值的 const 属性，而是指成员函数的 const 饰词，这意味着该成员函数不能修改其操作对象（严格地说是不能修改其非 mutable 数据成员）。

需要重点提请注意的是，constexpr 是对象和函数接口的组成部分。constexpr 实际上宣告的是："但凡任何 C++ 要求在此使用一个常量表达式的语境，皆可以用我。"一旦你把一个对象或函数声明成了 constexpr，客户就可以将其用于这种语境。而万一你后来又感觉你对 constexpr 的运用不当，然后移除了它，这个动作就可以导致无穷无尽的客户代码被拒绝编译（仅仅是向函数体里出于调试或性能调优的目的而增添一条 I/O 语句就可能导致这么个局面，因为通常来说，constexpr 函数里是不允许有 I/O 语句的）。"只要有可能使用 constexpr，就使用它"这句话中的"只要有可能"的含义就是你是否有一个长期的承诺，将由 constexpr 带来的种种限制施加于相关的函数和对象之上。

要点速记

- constexpr 对象都具备 const 属性，并由编译期已知的值完成初始化。
- constexpr 函数在调用时若传入的实参值是编译期已知的，则会产出编译期结果。
- 比起非 constexpr 对象或 constexpr 函数而言，constexpr 对象或是 constexpr 函数可以用在一个作用域更广的语境中。

条款 16：保证 const 成员函数的线程安全性

在数学领域中，使用一个类来表示多项式会非常方便。而在这个类中，若有一个函数能够计算多项式的根，即那些使得多项式求值结果为零的值，将会很有用。这样一个函数并不会造成多项式的值的改动，因此将它声明为 const 成员函数也很自然。

```
class Polynomial {
public:
  using RootsType =                    // 持有值的数据结构
    std::vector<double>;               // 这些值使得多项式求值结果为零
  …                                    // (using 关键字参见条款 9)

  RootsType roots() const;

  …

};
```

计算多项式的根也许代价高昂，所以我们不愿意执行这个计算，除非不得不做。即使不得不做，也当然不愿意做不止一次。这么一来，在不得不计算多项式的根时，我们就把这些根缓存起来，并以返回缓存值的手法来实现 roots。以下是一种基本的做法：

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    if (!rootsAreValid) {                // 如果缓存无效

      ...
      rootsAreValid = true;              // 则计算根，并将其存入 rootVals
    }

    return rootVals;
  }

  private:
    mutable bool rootsAreValid{ false };        // 关于初始化的信息
    mutable RootsType   rootVals{};             // 参见条款 7
  };
```

从概念上说，roots 不会改变它操作的 Polynomial 对象，然而作为缓存活动的组成部分，它可能需要修改 rootVals 和 rootsAreValid 的值。这是 mutable 的经典用例，也是它为何被加到数据成员声明中。

设想现在有两个线程同时在同一个 Polynomial 对象上调用 roots：

```
Polynomial p;

...

/*------  线程 1  ------ */     /*--------  线程 2  -------- */

auto rootsOfP = p.roots();      auto valsGivingZero = p.roots();
```

这段用户代码完全合理。roots 是一个 const 成员函数，这意味着它代表的是一个读操作。多个线程在没有同步的条件下执行读操作是安全的，至少会被认为是安全的。在本例中，却并不安全。因为在 roots 内部，这些线程中的一个或两个可能企图更改数据成员 rootsAreValid 和 rootVals，这就意味着这段代码可能有不同的多个线程在没有同步的情况下读写同一块内存，而这就是数据竞险（data race）。这段代码存在未定义行为。

问题就在于，roots 被声明成了 const 函数，但却并非线程安全的。在 C++11 中，const 声明的语义保持了和 C++98 一样的正确性（获取多项式的根并不会导致该多项式变更其值），因此，应该修正的是其线程安全性的缺失。

欲完成这个目标，最简单的办法也是最常见的：引入一个 mutex（意为互斥量，mutual exclusion 的简写）：

```
class Polynomial {
public:
  using RootsType = std::vector<double>;

  RootsType roots() const
  {
    std::lock_guard<std::mutex>  g(m);          // 加上互斥量

    if (!rootsAreValid) {                       // 若缓存无效则计算 / 存储 roots

      ...

      rootsAreValid = true;
    }

    return rootVals;
  }                                             // 解除互斥量

private:
  mutable std::mutex m;
  mutable bool rootsAreValid{ false };
  mutable RootsType rootVals{};
};
```

之所以要把 std::mutex m 声明为 mutable，是因为加锁与解锁都不是 const 成员函数所为，如果没有这么一个声明饰词，在 roots（这么个 const 成员函数）内，m 就会被当作是个 const 对象来处理了。

值得关注的是，由于 std::mutex 是个只移型别（move-only type）（即只能移动但不能复制的型别），将 m 加入 Polynomial 的副作用就是 Polynomial 失去了可复制性。不过，它仍然可移动。

就一些特定情况而言，引入互斥量是杀鸡用牛刀之举。例如，如果要计算一个成员函数被调用的次数，使用 std::atomic 型别的计数器（可以确保其他线程可以以不加分割的方式观察到其操作发生，参见条款 40）将会是一种成本较低的途径（是否真的成本较低，则取决于硬件以及互斥量在你使用标准库中的实现）。以下代码演示了如何使用 std::atomic 型别的对象来计算调用次数：

```
class Point {                                   // 表示 2D 点
public:
  ...

  double distanceFromOrigin() const noexcept
  {                                             // noexcept 的信息参见条款 14

    ++callCount;                                // 带原子性的自增操作

    return std::sqrt((x * x) + (y * y));
  }
```

```
private:
  mutable std::atomic<unsigned> callCount{ 0 };
  double x, y;
};
```

与 std::mutex 一样，std::atomic 也是只移型别。因此，Point 中 callCount 的存在会使得 Point 也变成只移型别。

由于对 std::atomic 型别的变量的操作与加上与解除互斥量相比，开销往往比较小，你也许应该尝试比惯常程度更重度地依靠 std::atomic 型别的对象，例如，如果某类需要缓存计算开销较大的 int 型别的变量，则应该尝试使用一对 std::atomic 型别的变量来取代互斥量。

```
class Widget {
public:
  ...

  int magicValue() const
  {
    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;                    // 第一部分
      cacheValid = true;                            // 第二部分
      return cachedValue;
    }
  }

private:
  mutable std::atomic<bool> cacheValid{ false };
  mutable std::atomic<int> cachedValue;
};
```

这样做可行，但有时会做一些不必要的工作。考虑以下情况：

- 一个线程调用 Widget::magicValue 时，观察到 cacheValid 值为 false，于是执行了两个大开销的计算，并将其和赋值给了 cacheValue。
- 与此同时，另一个线程也在调用 Widget::magicValue，也观察到 cacheValid 值为 false，于是也执行了第一个线程刚刚完成的两次同样的大开销运算（此处“另一个线程”实际上有可能是另外若干个其他线程）。

这种行为与缓存的目标南辕北辙。颠倒对 cacheValid 和 cacheValue 的赋值顺序可以消除该问题，但结果却更坏了：

```
class Widget {
public:
  ...
```

```
    int magicValue() const
    {
      if (cacheValid) return cachedValue;
      else {
        auto val1 = expensiveComputation1();
        auto val2 = expensiveComputation2();
        cacheValid = true;                          // 第一部分
        return cachedValue = val1 + val2;           // 第二部分
      }
    }

    ...

};
```

假设 cacheValid 的值为 false，接着：

- 一个线程调用 Widget::magicValue 并执行到了 cacheValid 值被置为 true 的时刻。
- 在这一时刻，另一个线程也在调用 Widget::magicValue 并检视 cacheValid 的值。观察到其值为 true 后，该线程就把 cacheValid 的值给返回了，即使此时第一个线程还没有执行对 cacheValid 的赋值。因此，返回值是不正确的。

这里我们学到的教益是：对于单个要求同步的变量或内存区域，使用 std::atomic 就足够了。但是如果有两个或更多个变量或内存区域需要作为一整个单位进行操作时，就要动用互斥量了。对于 Widget::magicValue 而言，代码应该是这样的：

```
class Widget {
public:
  ...

  int magicValue() const
  {
    std::lock_guard<std::mutex>  guard(m);          // 给 m 加上互斥量

    if (cacheValid) return cachedValue;
    else {
      auto val1 = expensiveComputation1();
      auto val2 = expensiveComputation2();
      cachedValue = val1 + val2;
      cacheValid = true;
      return cachedValue;
    }
  }                                                 // 为 m 解除互斥量
  ...

private:
  mutable  std::mutex m;
  mutable int cachedValue;                          // 不再具备原子性
  mutable bool cacheValid{ false };                 // 不再具备原子性
};
```

目前，本条款的陈述都基于这样的假设，即多个线程会同时调用同一对象的同一个const成员函数。如果你撰写的成员函数并不符合这种情况，也就是说，可以保证在同一对象上不会有多于一个线程来调用它，那么该函数的线程安全性也就可有可无了。例如，如果类本身就是为独占性的、单线程的用途而设计，则其成员函数是否具备线程安全性就不甚要紧。在这样的情况下，那些互斥量和std::atomic型别的对象所带来的开销都可以避免，同时还可以消除那些包含了它们的类变成只移类这样的副作用。不过，这些线程无关的场合正在变得越来越罕见，并且会变得更加罕见。可以肯定地说，const成员函数都会运行在并发执行的条件下，所以你应该保证const成员函数具备线程安全性。

要点速记

- 保证const成员函数的线程安全性，除非可以确信它们不会用在并发语境中。
- 运用std::atomic型别的变量会比运用互斥量提供更好的性能，但前者仅适用对单个变量或内存区域的操作。

条款17：理解特种成员函数的生成机制

在C++官方用语中，特种成员函数是指那些C++会自行生成的成员函数。C++98有四种特种成员函数：默认构造函数、析构函数、复制构造函数，以及复制赋值运算符。当然，看文档的时候可别漏了小字部分。这些函数仅在需要时才会生成，亦即，在某些代码使用了它们，而在类中并未显式声明的场合。仅当一个类没有声明任何构造函数时，才会生成默认构造函数（只要指定了一个要求传参的构造函数，就会阻止编译器生成默认构造函数）。生成的特种成员函数都具有public访问层级且是inline的，而且它们都是非虚的，除非讨论的是一个析构函数，位于一个派生类中，并且基类的析构函数是个虚函数。在那种情况下，编译器为派生类生成的析构函数也是个虚函数。

不过这些都是你已经知道的了。对，这些都是老生常谈，是像美索不达米亚文明、殷墟、FORTRAN和C++98一样的老古董。不过，斗转星移，C++的特种成员函数生成规则也已与时俱进。对新规则有个清醒意识很重要，因为了解编译器会如何默默地向你的类中插入成员函数可谓进行高效C++程序设计的核心。

在C++11中，特种成员函数俱乐部中加入了两位新会员：移动构造函数和移动赋值运算符，它们的签名如下：

```
class Widget {
```

```
public:
  ...
  Widget(Widget&& rhs);                    // 移动构造函数

  Widget& operator=(Widget&& rhs);         // 移动赋值运算符
  ...
};
```

这两个特种成员函数的生成规则和行为表现一如其复制版本。移动操作也仅在需要时才生成，而一旦生成，它们执行的也是作用于非静态成员的“按成员移动”操作。意思是，移动构造函数将依照其形参rhs的各个非静态成员对于本类的对应成员执行移动构造，而移动赋值运算符则将依照其形参rhs的各个非静态成员对于本类的对应成员执行移动赋值。移动构造函数同时还会移动构造它的基类部分（如果有的话），而移动赋值运算符则会移动赋值它的基类部分。

不过，当我提到移动操作在某个数据成员或基类部分上执行移动构造或移动赋值的时候，并不能保证移动操作真的会发生。“按成员移动”实际上更像是按成员的移动请求，因为那些不可移动的型别（即那些并未为移动操作提供特殊支持的型别，这包括了大多数C++98中的遗留型别）将通过其复制操作实现“移动”。每个按成员进行的“移动”操作，其核心在于把std::move应用于每一个移动源对象，其返回值被用于函数重载决议，最终决定是执行一个移动还是复制操作，这个流程将在条款23中详述。在本条款中，只需记住，按成员移动是由两部分组成的，一部分是在支持移动操作的成员上执行的移动操作，另一部分是在不支持移动操作的成员上执行的复制操作。

如果发生了复制操作的情况，移动操作就不会在已有声明的前提下被生成。这就是说，生成移动操作的精确条件，与复制操作有所不同。

两种复制操作是彼此独立的：声明了其中一个，并不会阻止编译器生成另一个。所以，如果你声明了一个复制构造函数，同时未声明复制赋值运算符，并撰写了要求复制赋值的代码，则编译器会为你生成复制赋值运算符。类似地，如果你声明了一个复制赋值运算符，同时未声明复制构造函数，而撰写了要求复制构造的代码，则编译器会为你生成复制构造函数。这在C++98中成立，在C++11中仍成立。

两种移动操作并不彼此独立：声明了其中一个，就会阻止编译器生成另一个。这种机制的理由在于，假设你声明了一个移动构造函数，你实际上表明移动操作的实现方式将会与编译器生成的默认按成员移动的移动构造函数多少有些不同。而若是按成员进行的移动构造操作有不合用之处的话，那么按成员进行的移动赋值运算符极有可能也会有不合用之处。综上，声明一个移动构造函数会阻止编译器去生成移动赋值运算符，而声明一个移动赋值运算符也会阻止编译器去生成移动构造函数。

犹有进者，一旦显式声明了复制操作，这个类也就不再会生成移动操作了。这样的判断的依据在于，声明复制操作（无论是复制构造还是复制赋值）的行为表明了对象的常规复制途径（按成员复制）对于该类并不适用。编译器从而判定，既然按成员复制不适用于复制操作，则按成员移动极有可能也不适用于移动操作。

反之亦然。一旦声明了移动操作（无论是移动构造还是移动赋值），编译器就会废除复制操作（废除的方式是删除它们，参见条款 11）。毕竟如果按成员移动不是对象认为适当的移动方式的话，也就没有理由期望按成员复制是对象认为适当的复制方式。乍听起来，这样的做法会对 C++98 的代码造成破坏，因为相比于 C++98 而言，C++11 中复制操作的生成条件的限制更多了，但实际情况并非如此。C++98 的代码中不可能存在移动操作，因为 C++98 中根本没有“可移动”对象。唯一往遗留代码中的类中加入用户声明的移动操作的可能，就是它是为了成为 C++11 的代码的一部分而被加入的，而若要对一个类加以修改以享用移动语义的好处，它们就得遵从 C++11 的特种成员函数的生成机制了。

你可能听说过一条指导原则叫做大三律（Rule of Three）。大三律是说，如果你声明了复制构造函数、复制赋值运算符，或析构函数中的任何一个，你就得同时声明所有这三个。它植根于这样的思想：如果有改写复制操作的需求，往往意味着该类需要执行某种资源管理，而这就意味着：①在一种复制操作中进行的任何资源管理，也极有可能在另一种复制操作中也需要进行；②该类的析构函数也会参与到该资源的管理中（通常是释放之）。需要施加管理的经典资源就是内存，这就是为什么标准库中用以管理内存的类（如执行动态内存管理的 STL 容器）都会遵从大三律，两种复制操作和析构函数它们都会声明全。

大三律的一个推论是，如果存在用户声明的析构函数，则平凡的按成员复制也不适于该类。根据这个推论，又能得出进一步的结论，如果声明了析构函数，则复制操作就不该被自动生成，因为它们的行为不可能正确。不过在 C++98 标准被接受的时代，这样的论证过程没有得到充分的重视，所以在 C++98 中，用户声明的析构函数即使存在，也不会影响编译器生成复制操作的意愿。这种情况在 C++11 仍然得到了保持，但原因仅仅在于，如果要对复制操作的生成条件施加更严格的限制，就会破坏太多的遗留代码了。

由于大三律背后的理由仍然成立，再结合声明了复制操作就会阻止隐式生成移动操作的事实，就推动了 C++11 中的这样一个规定：只要用户声明了析构函数，就不会生成移动操作。

这么一来，移动操作的生成条件（如果需要生成）仅当以下三者同时成立：

- 该类未声明任何复制操作。
- 该类未声明任何移动操作。
- 该类未声明任何析构函数。

总有一天，这样的机制也会延伸到复制操作，因为C++11标准规定，在已经存在复制操作或析构函数的条件下，仍然自动生成复制操作已经成为了被废弃的行为。这意味着，如果你有一些代码在已经存在任一复制操作或析构函数的条件下，仍然依赖复制操作的自动生成的话，你就得考虑升级这些类，以消除这样的依赖。假定编译器生成的这些函数有着正确的行为（即按成员复制类的非静态数据成员正是你所需要的行为），那事情就简单了，因为C++11可以通过“=default”来显式地表达这个想法：

```
class Widget {
public:
  ...
  ~Widget();                                    // 用户定义的析构函数

  ...
  Widget(const Widget&) = default;              // 默认复制构造函数的行为是正确的

  Widget&                                       // 默认复制赋值运算符的行为
    operator=(const Widget&) = default;         // 是正确的
  ...
};
```

这种手法往往对于多态基类会很有用，所谓多态基类，是指定义接口的类，而派生类的操作则通过这些接口来完成。多态基类往往会有虚析构函数，因为若不然，有些操作（比如通过基类指针或引用来对派生类执行的delete或typeid等）就会产生未定义的，或误导性的结果。而除非一个类继承而来的析构函数本身就是虚的，它也只能通过把析构函数声明成虚的来达到拥有虚析构函数的目的。通常情况下，虚析构函数的默认实现就是正确的，而“=default”则是表达这一点的很好方式。不过，一旦用户声明了析构函数，移动操作的生成就被抑制了，而如果可移动性是能够支持的，加上“=default”就能够再次给予编译器以生成移动操作的机会。声明移动操作又会废除复制操作，所以如果还要可复制性，就再加一轮“=default”来实现这个愿望吧：

```
class Base {
public:
  virtual ~Base() = default;                    // 使析构函数成为虚的
  Base(Base&&) = default;                       // 提供移动操作的支持
  Base& operator=(Base&&) = default;

  Base(const Base&) = default;                  // 提供复制操作的支持
  Base& operator=(const Base&) = default;

  ...
```

```
};
```

事实上，即使编译器能够为类生成复制和移动操作，并且这些生成的函数也符合需要，恐怕遵从这样一个策略也是更好的选择：你将这些函数自行声明，并以“=default”作为它们的定义。这样做是多费些功夫，但是可以更清晰地表明你的意图，并可以防止错过一些非常微妙的缺陷。举个例子，假设你有一个表示字符串表格的类，即一种允许通过整型 ID 来快速检索字符串值的数据结构：

```
class StringTable {
public:
  StringTable() {}
  ...                              // 实现插入、擦除、检索等操作的函数
                                   // 但没有函数来实现复制、移动或析构

private:
  std::map<int, std::string> values;
};
```

假定该类没有复制操作，没有移动操作，也没有析构函数，编译器将在这些函数有需要调用时自动生成它们。这挺方便的。

可是，假设过了一段时间，决定要把对象的默认构造和析构都记入日志。加上这样的功能也很容易：

```
class StringTable {
public:
  StringTable()
  { makeLogEntry( "Creating StringTable object" ); }    // 这是后加的

  ~StringTable()
  { makeLogEntry( "Destroying StringTable object" ); }  // 这也是后加的

  ...                                                   // 其他函数不变
private:
  std::map<int, std::string> values;                    // 数据成员也不变
};
```

这样的改动看起来合情合理，但是析构函数的声明有一个潜大的、可观的副作用：它阻止了移动操作的生成。当然，复制操作的生成并未受影响。所以，这段代码好像仍然正常地被编译、执行，并通过了功能测试。就连针对移动操作的测试也能够通过，因为即使该类不再可移动了，针对它进行移动操作的请求仍然可以编译和执行。而这些请求，一如本章前面所言，触发的则是复制操作。这就意味着，那些“移动”StringTable 对象的代码，实际上执行的是 StringTable 对象的复制，也就是底层的 std::map<int, std::string> 对象的复制。而 std::map<int, std::string> 对象的复制操作，可能会比其移动操作慢上若干个数量级。这么一来，向类中加入一个

析构函数的简单动作，就可能引发可观的性能问题！如果复制和移动操作都已经显式地用“=default”来定义了，这个问题就根本不会出现。

好啦，忍受了我啰嗦这么半天支配了C++11中复制和移动操作的机制，你可能会在想，我什么时候会开始将话题转向另外两个特种函数，即默认构造函数和析构函数呢。那就是现在了，不过就一句话，因为这两个成员函数的规则基本上没有任何变化：C++11中的机制和C++98如出一辙。

总而言之，C++11中，支配特种成员函数的机制如下：

- **默认构造函数**：与C++98的机制相同。仅当类中不包含用户声明的构造函数时才生成。
- **析构函数**：与C++98的机制基本相同，唯一的区别在于析构函数默认为noexcept（参见条款14）。与C++98的机制相同，仅当基类的析构函数为虚的，派生类的析构函数才是虚的。
- **复制构造函数**：运行期行为与C++98相同：按成员进行非静态数据成员的复制构造。仅当类中不包含用户声明的复制构造函数时才生成。如果该类声明了移动操作，则复制构造函数将被删除。在已经存在复制赋值运算符或析构函数的条件下，仍然生成复制构造函数已经成为了被废弃的行为。
- **复制赋值运算符**：运行期行为与C++98相同：按成员进行非静态数据成员的复制赋值。仅当类中不包含用户声明的复制赋值运算符时才生成。如果该类声明了移动操作，则复制构造函数将被删除。在已经存在复制构造函数或析构函数的条件下，仍然生成复制赋值运算符已经成为了被废弃的行为。
- **移动构造函数和移动赋值运算符**：都按成员进行非静态数据成员的移动操作。仅当类中不包含用户声明的复制操作、移动操作和析构函数时才生成。

请注意，这些机制中只字未提成员函数模板的存在会阻止编译器生成任何特种成员函数。这就意味着，如果Widget形如：

```
class Widget {
  ...
  template<typename T>
  Widget(const T& rhs);                  // 以任意型别构造 Widget

  template<typename T>
  Widget& operator=(const T& rhs);       // 以任意型别对 Widget 赋值
  ...
};
```

编译器会始终生成Widget的复制和移动操作（假定支配其生成的条件都得到了满足），即使这些模板的具现结果生成了复制构造函数或复制赋值运算符的签名（当T的值为Widget时就会发生这种情况）。十有八九，你会觉得我在螺蛳壳里做道场。但是我之所以要提起这个自然是有理由的。条款26会告诉你，这么一个边缘场景有着至关重要的推论。

要点速记

- 特种成员函数是指那些C++会自行生成的成员函数：默认构造函数、析构函数、复制操作，以及移动操作。
- 移动操作仅当类中未包含用户显式声明的复制操作、移动操作和析构函数时才生成。
- 复制构造函数仅当类中不包含用户显式声明的复制构造函数时才生成，如果该类声明了移动操作则复制构造函数将被删除。复制赋值运算符仅当类中不包含用户显式声明的复制赋值运算符才生成，如果该类声明了移动操作则复制赋值运算符将被删除。在已经存在显式声明的析构函数的条件下，生成复制操作已经成为了被废弃的行为。
- 成员函数模板在任何情况下都不会抑制特种成员函数的生成。

第 4 章

智能指针

诗人和作曲家往往对爱情情有独钟，有时又对计数偏爱有加，偶然也会把两者兼收并蓄。伊丽莎白·巴雷特·勃朗宁（Elizabeth Barrett Browning）有诗云，“何以爱汝，待我细数”（How do I love thee? Let me count the ways），保罗·西蒙（Paul Simon）亦有绝唱，“逃离爱娇娃，必有五十法”（There must be 50 ways to leave your lover），受此启发，我们也来把对于裸指针实在爱不起来的理由一一列举。

1. 裸指针在声明中并没有指出，裸指针指涉到的是单个对象还是一个数组。

2. 裸指针在声明中也没有提示在使用完指涉的对象以后，是否需要析构它。换言之，你从声明中看不出来指针是否拥有（own）其指涉的对象。

3. 即使知道需要析构指针所指涉的对象，也不可能知道如何析构才是适当的。是应该使用 delete 运算符呢，还是别有它途（例如，可能需要把指针传入一个专门的、用于析构的函数）？

4. 即使知道了应该使用 delete 运算符，参见理由 1，还是会发生到底应该用单个对象形式（“delete”）还是数组形式（“delete[]”）的疑问。一旦用错，就会导致未定义行为。

5. 即启用够确信，指针拥有其指涉的对象，并且也确知应该如何析构，要保证析构在所有代码路径上都仅执行一次（包括那些异常导致的路径）仍然困难重重。只要少在一条路径上执行，就会导致资源泄漏。而如果析构在一条路径上执行了多于一次，则会导致未定义行为。

6. 没有什么正规的方式能检测出指针是否空悬（dangle），也就是说，它指涉的内存是否已经不再持有指针本应该指涉的对象。如果一个对象已经被析构了，而某些指针仍然指涉到它，就会产生空悬指针。

可以肯定地说，裸指针是一种强力工具。但是几十年来的经验指出，只要稍微疏忽或放纵一下，这种工具就能把你忽悠得晕头转向。

智能指针（smart pointer）是解决这些问题的方法之一。智能指针对裸指针进行了包装，它们的行为很类似被包装起来的裸指针，但是却避免了很多在使用裸指针时会遭遇的陷阱。所以，相对于裸指针，应该优先选用智能指针。智能指针几乎可以做到任何裸指针能做到的事情，但是犯错的机会却大大减少了。

C++11 中共有四种智能指针：std::auto_ptr、std::unique_ptr、std::shared_ptr 和 std::weak_ptr。所有这些智能指针都是为管理动态分配对象的生命期而设计的，换言之，通过保证这样的对象在适当的时机以适当的方式析构（包括发生异常的场合），来防止资源泄漏。

std::auto_ptr 是个从 C++98 中残留下来的弃用特性，它是一种对智能指针进行标准化的尝试，这种尝试后来成为了 C++11 中的 std::unique_ptr。要正确地完成这个任务就需要移动语义，但在 C++98 中却并没有这样的语义。作为一种变通手段，std::auto_ptr 使用了它的复制操作来完成移动任务。这就导致了令人异常的代码（对 std::auto_ptr 对象执行复制操作会将其值置空）和让人烦恼的使用限制（例如，不能在容器中存储 std::auto_ptr 对象）。

std::unique_ptr 可以做 std::auto_ptr 能够做的任何事，并且不止于此。它执行的效率和 std::auto_ptr 一样高，而且不用扭曲其要表达的本意去复制任何对象。它从任何方面来看都比 std::auto_ptr 更好。只有一种 std::auto_ptr 的合理用例，那就是需要使用 C++98 编译器来编译代码的场合。除非有这样的限制，否则就应该用 std::unique_ptr 来替换 std::auto_ptr，不要再回头。

智能指针的 API 五花八门。唯一对于所有的智能指针都共同的功能就是默认构造。由于对于这些 API 的全面参考文档到处都有，我会把焦点放在 API 概述性的文档普遍缺失的那部分信息上，例如值得引起注意的用例、运行时成本分析等。能否掌握这些信息，就是仅仅一般性地使用智能指针和高效地使用智能指针的区别。

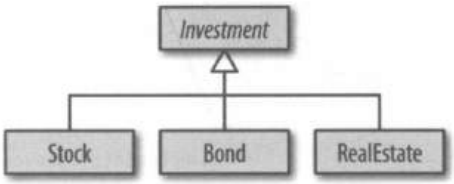
条款 18：使用 std::unique_ptr 管理具备专属所有权的资源

每当你需要使用智能指针时，std::unique_ptr 基本上应是手头首选。可以认为在默认情况下 std::unique_ptr 和裸指针有着相同的尺寸，并且对于大多数的操作（包括提领），它们都是精确地执行了相同的指令。这就意味着你甚至在内存和时钟周期紧张的场合下使用 std::unique_ptr。如果裸指针对你来说足够小并且足够快，那么 std::unique_ptr 几乎可以肯定也能满足你的要求。

std::unique_ptr 实现的是专属所有权语义。一个非空的 std::unique_ptr 总是拥有其所指涉到的资源。移动一个 std::unique_ptr 会将所有权从源指针移至目标指针（源指针被置空）。std::unique_ptr 不允许复制，因为如果复制了一个 std::unique_ptr，就会得到两个指涉到同一资源的 std::unique_ptr，而这两者都认为自己拥有（因此应当析构）该资源。因而，std::unique_ptr 是个只移型别。在执行析构操作时，由非空的 std::unique_ptr 析构其资源。默认地，资源的析构是通过对 std::unique_ptr 内部的裸指针实施 delete 完成的。

std::unique_ptr 的一个常见用法是在对象继承谱系中作为工厂函数的返回型别。假设我们有一个以 Investment 为基类的投资型别的继承谱系（例如股票、债券、不动产等）。

```
class Investment { ... };

class Stock:
  public Investment { ... };

class Bond:
  public Investment { ... };

class RealEstate:
  public Investment { ... };
```



这种继承谱系的工厂函数通常会在堆上分配一个对象并且返回一个指涉到它的指针，并当不再需要该对象时，由调用者负责删除之。这堪称 std::unique_ptr 的完美匹配，因为调用者需要对工厂函数返回的资源负责（亦即，对该资源拥有专属所有权），而当 std::unique_ptr 被析构时，又会自动对其所指涉到的对象实施 delete。Investment 继承谱系的工厂函数应声明如下：

```
template<typename... Ts>                          // 返回 std::unique_ptr
std::unique_ptr<Investment>                       // 指涉到根据指定实参创建的对象
makeInvestment(Ts&&... params);
```

调用者可以在单个作用域内像这样来使用返回的 std::unique_ptr：

```
{
  ...
  auto pInvestment =                    // pInvestment 的型别是
    makeInvestment( arguments );        // std::unique_ptr<Investment>
  ...
}                                       // *pInvestment 在此析构
```

然而，即使所有权不断流转，调用者也能使用 std::unique_ptr，例如当一个由工厂函数返回的 std::unique_ptr 被移动至一个容器中，该容器的元素又被移动到某对象的数据成员中，而稍后该对象又被析构这样的场景。在这种情况下，该对象的 std::unique_ptr 数据成员将随着对象主体的析构而被析构，从而引发工厂函数返回的资源也被析构。如果所有权链由于异常或其他非典型控制流（例如，函数提早返回或循环中出现了 break 语句）而中断时，具管托管资源所有权的 std::unique_ptr 最终将调用该资源的析构函数[注1]，因而，其托管资源终将被析构。

默认地，析构通过 delete 运算符实现，但是在析构过程中 std::unique_ptr 可以被设置为使用自定义析构器（custom deleter）：析构资源时所调用的任意函数（或函数对象，包括那些由 lambda 表达式产生的）。如果由 makeInvestment 创建的对象不应被直接删除，而是应该先写入一条日志，那么 makeInvestment 可以像下面这样实现（代码之后就会附上解释，所以你不必担心看上去有些部分的动机不够明显）。

```
auto delInvmt = [](Investment* pInvestment)          // 一个作为自定义析构器的
                {                                    // lambda 表达式
                  makeLogEntry(pInvestment);
                  delete pInvestment;
                };

template<typename... Ts>                             // 改进的返回型别
std::unique_ptr<Investment, decltype(delInvmt)>
makeInvestment(Ts&&... params)
{
  std::unique_ptr<Investment, decltype(delInvmt)>    // 待返回的指针
    pInv(nullptr, delInvmt);
```

注1：这条规则有若干例外情况，大部分源自非正常程序终止。如果一个异常传播开去，影响到某个线程的主函数（例如，初始化该程序的线程的 main 函数），或者违反了 noexcept 异常规格（参见条款 14），则局部对象可能不会析构。而如果调用了 std::abort 或某个退出函数（即 std::_Exit、std::exit 或 std::quick_exit）的话，局部对象肯定不会析构。

```
    if ( /* 应创建一个 Stock 型别的对象 */ )
    {
      pInv.reset(new Stock(std::forward<Ts>(params)...));
    }
    else if ( /* 应创建一个 Bond 型别的对象 */ )
    {
      pInv.reset(new Bond(std::forward<Ts>(params)...));
    }
    else if ( /* 应创建一个 RealEstate 型别的对象 */ )
    {
      pInv.reset(new RealEstate(std::forward<Ts>(params)...));
    }

    return pInv;
  }
```

我马上会解释这段代码如何运作，但首先考虑一下如果你是调用者，事情看起来会是怎样。假定你将 makeInvestment 的调用结果存储在 auto 变量中，则你可以欣然忽略正在使用的资源需要在析构时加以特殊处理这一事实。实际上，你真的应该喜出望外，因为通过使用 std::unique_ptr，你无需为资源何时被析构费心，更不必自行提供析构在程序的所有可能路径中都会执行的保证。std::unique_ptr 会自动完成所有工作。从客户角度来看，makeInvestment 的接口十分美好。

一旦理解了以下这些内容，就会发现这段代码不仅接口美好，实现也很不错：

- delInvmt 是由 makeInvestment 所返回的对象的自定义析构器。所有的自定义删除函数都接受一个指涉到欲析构对象的裸指针，然后采取必要措施析构该对象。本例中所采取的措施是调用 makeLogEntry 并实施 delete。使用 lambda 表达式来创建 delInvmt 很方便的，正如我们很快将看到的，它也比撰写一个常规函数效率更高。

- 当要使用自定义析构器时，其型别必须被指定为 std::unique_ptr 的第二个实参的型别。本例中即为 delInvmt 的型别，这也是为何 makeInvestment 的返回型别是 std::unique_ptr<Investment, decltype(delInvmt)> 的原因（关于 decltype，参考条款 3）。

- makeInvestment 的基本策略是创建一个空的 std::unique_ptr，使其指涉到适当型别的对象，然后将其返回。为了将自定义析构器 delInvmt 和 pInv 关联起来，我们将 delInvmt 作为传给构造函数的第二个实参。

- 将一个裸指针（例如，使用 new 运算符的结果）赋给 std::unique_ptr 的尝试，不会通过编译，因为这会形成从裸指针到智能指针的隐式型别转换。这种隐式型

别转换大有问题，因此 C++11 中的智能指针将这种操作禁止了。这就是为什么需要使用 reset 来指定让 pInv 获取从使用 new 对算符产生的对象的所有权。

- 对每一次 new 运算符的调用结果，我们都使用 std::forward 将实参完美转发给 makeInvestment（参见条款 25）。这可以使得所创建对象的构造函数能够获得调用者提供的所有信息。

- 自定义析构器接受一个型别为 Investment* 的形参，不管在 makeInvestment 内部实际创建的对象型别是什么（例如 Stock，Bond 或 RealEstate），它终究会在 lambda 表达式中作为一个 Investment* 对象被删除。这意味着我们会通过一个基类指针来删除一个派生类对象。为此，基类 Investment 必须具备一个虚析构函数：

```
class Investment {
public:
  ...
  virtual ~Investment();              // 必备的设计组件!
  ...
};
```

而在 C++14 中，由于有了函数返回型别推导（参见条款 3），就使得 makeInvestment 可以通过以下更加简单的、封装性更好的手法实现：

```
template<typename... Ts>
auto makeInvestment(Ts&&... params)                          // C++14
{
  auto delInvmt = [](Investment* pInvestment)
                  {                                          // 现在自定义析构器位于
                    makeLogEntry(pInvestment);               // makeInvestment 内部了
                    delete pInvestment;
                  };

  std::unique_ptr<Investment, decltype(delInvmt)>
    pInv(nullptr, delInvmt);                                 // 同前

  if ( ... )                                                 // 同前
  {
    pInv.reset(new Stock(std::forward<Ts>(params)...));
  }
  else if ( ... )                                            // 同前
  {
    pInv.reset(new Bond(std::forward<Ts>(params)...));
  }
    else if ( ... )                                          // 同前
  {
    pInv.reset(new RealEstate(std::forward<Ts>(params)...));
  }
  return pInv;                                               // 同前
}
```

我之前介绍过，在使用默认析构器（即delete运算符）的前提下，你可以合理地认为std::unique_ptr和裸指针尺寸相同。自定义析构器现身以后，情况便有所不同了。若析构器是函数指针，那么std::unique_ptr的尺寸一般会增加一到两个字长（word）。而若析构器是函数对象，则带来的尺寸变化取决于该函数对象中存储了多少状态。无状态的函数对象（例如，无捕获的lambda表达式）不会浪费任何存储尺寸。这意味着当一个自定义析构器既可以用函数，又可以用无捕获的lambda表达式来实现时，lambda表达式是更好的选择：

```
auto delInvmt1 = [](Investment* pInvestment)              // 使用无状态 lambda 表达式
                 {                                        // 作为自定义析构器
                   makeLogEntry(pInvestment);
                   delete pInvestment;
                 };

template<typename... Ts>                                  // 返回值尺寸
std::unique_ptr<Investment, decltype(delInvmt1)>          // 与 Investment* 相同
makeInvestment(Ts&&... args);

void delInvmt2(Investment* pInvestment)                   // 使用函数
{                                                         // 作为自定义析构器
  makeLogEntry(pInvestment);
  delete pInvestment;
}

template<typename... Ts>                                  // 返回值尺寸等于
std::unique_ptr<Investment,                               // Investment* 的尺寸
                void (*)(Investment*)>                    // 加上至少函数指针的尺寸！
makeInvestment(Ts&&... params);
```

析构器采用带有大量状态的函数对象实现，可能使得std::unique_ptr对象增加可观尺寸。因此，假如你发现某个自定义析构器使得std::unique_ptr大到不可接受，可能就是时候改变一下设计了。

工厂函数不仅是std::unique_ptr的惯常用例，它更广泛地用作实现Pimpl习惯用法的机制。后者代码并不复杂，但有时也不是十分直截了当，所以我推荐你参阅条款22，它专门讨论了这个主题。

std::unique_ptr以两种形式提供，一种是单个对象（std::unique_ptr<T>），另一种是数组（std::unique_ptr<T[]>）。这样区分的结果是，对于std::unique_ptr指涉到的对象种类不会产生二义性。std::unique_ptr的API也被设计成与使用形式相匹配。比如单个对象形式不提供索引运算符（operator[]），而数组形式则不提供提领运算符（operator*和operator->）。

std::unique_ptr数组形式的存在，作为一种知识上的了解就够了。因为，

std::array、std::vector 和 std::string 这些数据结构几乎总是比裸数组更好。我能想到的唯一 std::unique_ptr<T[]> 的合理使用场景是使用了一个 C 风格的 API，它返回了堆上的裸指针，且指定了其指涉对象的所有权。

std::unique_ptr 是 C++11 中表达专属所有权的方式，但它还有一个十分吸引人的特性，就是 std::unique_ptr 可以方便高效地转换成 std::shared_ptr：

```
std::shared_ptr<Investment> sp =        // 将 std::unique_ptr 型别的对象
  makeInvestment( arguments );          // 转换为 std::shared_ptr 型别
```

正是这一特性使得 std::unique_ptr 如此适合作为工厂函数的返回型别。工厂函数并不知道调用者是对其返回的对象采取专属所有权语义好，还是共享所有权（例如，std::shared_ptr）更合适。通过返回一个 std::unique_ptr，工厂函数就向调用者提供了最高效的智能指针，但它也不会阻止调用者把返回值转换成更具弹性的其他兄弟型别的智能指针（关于 std::shared_ptr，请参考条款 19）。

要点速记

- std::unique_ptr 是小巧、高速的、具备只移型别的智能指针，对托管资源实施专属所有权语义。
- 默认地，资源析构采用 delete 运算符来实现，但可以指定自定义删除器。有状态的删除器和采用函数指针实现的删除器会增加 std::unique_ptr 型别的对象尺寸。
- 将 std::unique_ptr 转换成 std::shared_ptr 是容易实现的。

条款 19：使用 std::shared_ptr 管理具备共享所有权的资源

使用带有垃圾回收的语言的程序员会指出并嘲笑 C++ 程序员为避免资源泄漏所作出的努力。“多么原始啊！”他们讥笑道，“你们难道没有读过上世纪 60 年代的 Lisp 语言备忘录吗？应当由机器来管理资源的生存期，而非人类。”C++ 开发者回敬了他们一个白眼，答道：“你说的那个备忘录成文的时代，难道不是唯一的所谓资源即为内存，并且资源回收的时序完全不确定吗？我们更喜欢析构函数所带来的通用性和可预测性，谢谢好意。”然而我们的振振有词有一部分是在夸口。垃圾回收的确很方便，而手动进行生存期管理也的确类似使用石刀和熊皮来构建一个记忆内存电路。我们为

什么不能鱼与熊掌兼得，拥有一个自动运作的系统（类似垃圾回收），但也能够应用到所有资源并且具备可预测时序（类似析构函数）呢？

std::shared_ptr 是 C++11 用来将这两个世界连通的方法。通过 std::shared_ptr 这种智能指针访问的对象采用共享所有权来管理其生存期。没有哪个特定的 std::shared_ptr 拥有该对象。取而代之的是，所有指涉到它的 std::shared_ptr 共同协作，确保在不再需要该对象的时刻将其析构。当最后一个指涉到某对象的 std::shared_ptr 不再指涉到它时（例如，由于该 std::shared_ptr 被析构，或使其指涉到另一个不同的对象），该 std::shared_ptr 会析构其指涉到的对象。正如垃圾回收一样，用户无须操心如何管理被指涉到对象的生存期，但又如析构函数一样，该对象的析构函数的时序是确定的。

std::shared_ptr 可以通过访问某资源的引用计数来确定是否自己是最后一个指涉到该资源的。引用计数是个与资源关联的值，用来记录跟踪指涉到该资源的 std::shared_ptr 数量。std::shared_ptr 的构造函数会使该计数递增（通常如此，参见下文），而其析构函数会使该计数递减，而复制赋值运算符同时执行两种操作（如果 sp1 和 sp2 是指涉到不同对象的 std::shared_ptr，则赋值运算“sp1 = sp2”将修改 sp1，使其指涉到 sp2 所指涉到的对象。该赋值的净效应是：最初 sp1 所指涉到的对象的引用计数递减，同时 sp2 所指涉到的对象的引用计数递增）。如果某个 std::shared_ptr 发现，在实施过一次递减后引用计数变成了零，即不再有 std::shared_ptr 指涉到该资源，则 std::shared_ptr 会析构之。

引用计数的存在会带来一些性能影响：

- **std::shared_ptr 的尺寸是裸指针的两倍。**因为它们内部既包含一个指涉到该资源的裸指针，也包含一个指涉到该资源的引用计数的裸指针[注 2]。
- **引用计数的内存必须动态分配。**从概念上来说，引用计数与被指涉到的对象相关联，然而被指涉到的对象对此却一无所知。因此它们没有存储引用计数的位置[令人愉快的是，这么一来任何型别的对象（甚至内建型别）都可以由 std::shared_ptr 托管]。条款 21 会解释，std::shared_ptr 若是由 std::make_ptr 创建，可以避免动态分配的成本。然而仍有一些场景下，不可以使用 std::make_ptr。但无论是不是使用 std::make_ptr，引用计数都会作为动态分配的数据来存储。
- **引用计数的递增和递减必须是原子操作，**因为在不同的线程中可能存在并发的读写器。例如，在某个线程中指涉到某资源的某个 std::shared_ptr 可能正在执行

注 2：这种实现方式并非标准所要求，然而我所熟悉的每一个标准库实现都采用了这种方式。

其析构函数（因此其所指资源的引用计数会递减），而在另一个线程中，一个指涉到同一对象的 std::shared_ptr 有可能同时正在被复制（因此该引用计数又会递增）。原子操作一般都比非原子操作慢，所以即使引用计数通常只有一个字长，也应当假设读写它们成本比较高昂的。

我前面说到，std::shared_ptr 的构造函数仅仅“通常”会增加其所指涉到的对象的引用计数时，是否激起了你的好奇心？创建一个指涉到某对象的 std::shared_ptr 总是会产生额外一个指涉到该对象的 std::shared_ptr，那么我们为什么不总是递增引用计数呢？

移动构造函数，这就是原因所在。从一个已有 std::shared_ptr 移动构造一个新的 std::shared_ptr 会将源 std::shared_ptr 置空，这意味着一旦新的 std::shared_ptr 产生后，原有的 std::shared_ptr 将不再指涉到其资源，结果是不需要进行任何引用计数操作。因此，移动 std::shared_ptr 比复制它们要快：复制要求递增引用计数，而移动则不需要。这一点对于构造和赋值操作同样成立，所以，移动构造函数比复制构造函数快，移动赋值比复制赋值快。

与 std::unique_ptr 类似（参见条款 18），std::shared_ptr 也使用 delete 运算符作为其默认资源析构机制，但它同样支持自定义析构器。然而这种支持的设计却与 std::unique_ptr 有所不同。对于 std::unique_ptr 而言，析构器的型别是智能指针型别的一部分。但对于 std::shared_ptr 而言，却并非如此：

```
auto loggingDel = [](Widget *pw)                    // 自定义析构器
                  {                                 // （同条款 18）
                    makeLogEntry(pw);
                    delete pw;
                  };

std::unique_ptr<                                    // 析构器型别是
  Widget, decltype(loggingDel)                      // 智能指针型别的一部分
  > upw(new Widget, loggingDel);

std::shared_ptr<Widget>                             // 析构器型别不是
  spw(new Widget, loggingDel);                      // 智能指针型别的一部分
```

std::shared_ptr 的设计更具弹性。考虑两个 std::shared_ptr<Widget>，各有一个不同型别的自定义析构器（例如，由于该自定义析构器是通过 lambda 表达式指定的）：

```
auto customDeleter1 = [](Widget *pw) { … };    // 自定义析构器
auto customDeleter2 = [](Widget *pw) { … };    // 各有不同型别

std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

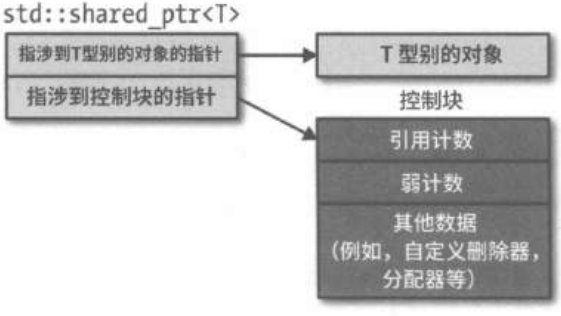
由于 pw1 和 pw2 具有同一型别，因此它们可以被放置在元素型别为该对象型别的容器中：

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

它们也可以互相赋值，并且都可以被传递至要求 std::shared_ptr<Widget> 型别形参的函数。然而对于具有不同自定义析构器型别的 std::unique_ptr 来说，以上这些均无法实现，因为自定义析构器的型别会影响 std::unique_ptr 的型别。

与 std::unique_ptr 的另一点不同，是自定义析构器不会改变 std::shared_ptr 的尺寸。无论析构器是怎样的型别，std::shared_ptr 对象的尺寸都相当于裸指针的两倍。这是个很棒的消息，但也可能会令你隐隐不安。自定义析构器有可能是函数对象，而函数对象可以包含任意数量的数据，这意味着它们的尺寸可能是任意大小。std::shared_ptr 如何能够在不使用更多内存的前提下，指涉到任意尺寸的析构器？

这并无可能，std::shared_ptr 不得不使用更多内存。然而，该部分内存却不属于 std::shared_ptr 对象的一部分。它位于堆上，又或是 std::shared_ptr 的创建者利用了 std::shared_ptr 对自定义内存分配器的支持的话，则它会在托管给该分配器的内存位置。在前文中，我曾提到一个 std::shared_ptr 对象包含了指涉到一个它所指涉到的对象的引用计数的指针。这一点没错，但是可能会有一点令人误解，因为该引用计数是控制块这样一个更大的数据结构的一部分。每一个由 std::shared_ptr 管理的对象都有一个控制块。除了包含引用计数之外，该控制块还包含自定义析构器的一个复制，如果该自定义析构器被指定的话。如果指定了一个自定义内存分配器，控制块也会包含一份它的复制。控制块还有可能包含其他附加数据，包括如条款 21 提到的一个被称为弱计数的次级引用计数，但我们在本条款中将忽略此类数据。我们可以想象与 std::shared_ptr<T> 对象相关的内存，如图所示。



一个对象的控制块由创建首个指涉到该对象的 std::shared_ptr 的函数来确定。至少，应该是这样运作。毕竟，正在创建指涉到某对象的 std::shared_ptr 的函数是无从得

知是否有其他的 std::shared_ptr 已经指涉到该对象的，因此，控制块的创建遵循了以下规则：

- **std::make_shared（参见条款 21）总是创建一个控制块。**std::make_shared 会生产出一个用以指涉到的新对象，因此在调用 std::make_shared 的时刻，显然不会有针对该对象的控制块存在。

- **从具备专属所有权的指针（即 std::unique_ptr 或 std::auto_ptr 指针）出发构造一个 std::shared_ptr 时，会创建一个控制块。**专属所有权指针不使用控制块，因此不应该存在所指涉到的对象来说不应存在控制块（作为构造过程的一部分，std::shared_ptr 被指定了其所指涉到的对象的所有权，因此那个专属所有权的智能指针会被置空）。

- **当 std::shared_ptr 构造函数使用裸指针作为实参来调用时，它会创建一个控制块。**如果想从一个已经拥有控制块的对象出发来创建一个 std::shared_ptr，你大概会传递一个 std::shared_ptr 或 std::weak_ptr（参见条款 20）而非裸指针作为构造函数的实参。如果给 std::shared_ptr 的构造函数传递 std::shared_ptr 或 std::weak_ptr 作为实参，则不会创建新的控制块，因为它们可以依赖传入的智能指针以指涉到任意所需的控制块。

这些规则会导致一个后果：从同一个裸指针出发来构造不止一个 std::shared_ptr 的话，简直如同免费搭乘了粒子加速器向未定义行为直奔而去。因为这么一来，被指涉到的对象将会有多重的控制块。多重的控制块意味着多重的引用计数，而多重的引用计数意味着该对象将被析构多次（每个引用计数会导致一次析构）。这意味着，如下所示的代码会是行不通的、行不通的、行不通的（重要的话说三遍）：

```
auto pw = new Widget;                              // pw 是个裸指针

…

std::shared_ptr<Widget> spw1(pw, loggingDel);      // 为 *pw 创建一个控制块

…

std::shared_ptr<Widget> spw2(pw, loggingDel);      // 为 *pw 创建了
                                                   // 第二个控制块！
```

创建裸指针 pw，使其指涉到一个动态分配对象，这本身就是不好的。因为它与本章背后的一个总体建议“更多地使用智能指针而非裸指针”背道而驰（如果你已经忘记该建议被提出的动因，参见条款 17 的相关内容）。不过此处先把这一点搁在一边，创建 pw 的这行代码固然没有很好的编程格调，但它至少尚未引发未定义的程序行为。

然后，spw1 的构造函数调用时传入了一个裸指针，因此它为所指涉到的对象创建一个控制块（并同时创建一个引用计数）。在这种情况下，就是 *pw（即被 pw 指涉到的对象）。如果仅就其自身而言，还没有出现问题，但是随着 spw2 的构造函数在调用时传入了同一个裸指针，它也为 *pw 创建了一个控制块（同时创建了又一个引用计数）。这么一来，*pw 就有了两个引用计数，而每个引用计数最终都会变为零，从而导致 *pw 被析构两次。第二次析构将会引发未定义行为。

从这段 std::shared_ptr 的用法代码片段，我们至少可以习得两个教训。首先，尽可能避免将裸指针传递给一个 std::shared_ptr 的构造函数。常用的替代手法，是使用 std::make_shared（参见条款 21）。然而在上述例子中，由于采用了自定义析构器，这么一来就无法使用 std::make_shared 了。其次，如果必须将一个裸指针传递给 std::shared_ptr 的构造函数，就直接传递 new 运算符的结果，而非传递一个裸指针变量。如果上例中第一部分的代码这样被改写：

```
std::shared_ptr<Widget> spw1(new Widget,        // 直接传递 new 表达式
                             loggingDel);
```

这将大大降低试图由相同的裸指针创建第二个 std::shared_ptr 的可能。取而代之的是，代码撰写者在创建 spw2 时会自然地使用 spw1 作为初始化实参（亦即，会调用 std::shared_ptr 的复制构造函数），这将不会产生任何诸如此类的问题：

```
std::shared_ptr<Widget>  spw2(spw1);    // spw2 使用的是和 spw1 同一个控制块
```

使用裸指针变量作为 std::shared_ptr 构造函数实参时，会有一种令人吃惊的方式导致涉及 this 指针的多重控制块。假设我们的程序使用 std::shared_ptr 来托管 Widget 对象，并且有个数据结构用来追踪被处理的 Widget：

```
std::vector<std::shared_ptr<Widget>>  processedWidgets;
```

又假设 Widget 有个成员函数用来做这种处理：

```
class Widget {
public:
  ...
  void process();
  ...
};
```

对 Widget::process 而言，有一种看似合理的方法来完成跟踪操作：

```
void Widget::process()
{
  ...                                   // 处理对象本身
```

```
    processedWidgets.emplace_back(this);          // 将处理完的 Widget 加入链表
}                                                 // 这种做法大错特错!
```

关于错误部分的注释说明了一切，或至少说明了大部分（错误的部分在于 this 指针的传递，而不是 emplace_back 的使用。如果不熟悉 emplace_back，参见条款 42）。该代码能够通过编译，然而它把一个裸指针（this）传入了一个 std::shared_ptr 容器。由此构造的 std::shared_ptr 将为其所指涉到的 Widget 型别的对象（*this）创建一个新的控制块。乍听之下，这似乎没有什么危害，直到你意识到，如果在已经指涉到该 Widget 型别的对象的成员函数外部再套一层 std::shared_ptr 的话，未定义行为就将取得彻头彻尾的胜利。

std::shared_ptr 的 API 为类似这种情况提供了一种基础设施，它可能有着一个 C++ 标准库中最古怪的名字：std::enable_shared_from_this。当你希望一个托管到 std::shared_ptr 的类能够安全地由 this 指针创建一个 std::shared_ptr 时，它将为你继承而来的基类提供一个模板。在我们的例子中，Widget 将继承自 std::enable_shared_from_this，如下所示：

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  ...
  void process();
  ...
};
```

正如我前面所言，std::enable_shared_from_this 是一个基类模板，其型别形参总是其派生类的类名，因此 Widget 继承自 std::enable_shared_from_this<Widget>。如果一个派生类的基类是用该派生类作为模板形参具现的型别，这样的概念令你感觉头疼的话，就不要再去想它就好了。该代码完全合法，其背后的设计模式也已得到普遍认可，还有一个标准的名字。尽管这个模式的名字听起来几乎和 std::enable_shared_from_this 一样古怪，叫作奇妙递归模板模式（The Curiously Recurring Template Pattern，CRTP）。欲了解更多相关知识，请打开搜索引擎吧，因为这里我们得把话题拉回到 std::enable_shared_from_this 了。

std::enable_shared_from_this 定义了一个的成员函数，它会创建一个 std::shared_ptr 指涉到当前对象，但同时不会重复创建控制块。这个成员函数的名字是 enable_shared_from_this，每当你需要一个和 this 指针指涉到相同对象的 std::shared_ptr 时，都可以在成员函数中使用它。Widget::process 的一个安全实现如下：

```
void Widget::process()
{
  // 同前，处理对象本身
  ...
```

```
    // 将指涉到当前对象的 std::shared_ptr 加入 processedWidgets
    processedWidgets.emplace_back(shared_from_this());
}
```

从内部实现的角度看，shared_from_this 查询当前对象的控制块，并创建一个指涉到该控制块的新 std::shared_ptr。这样的设计依赖于当前对象已有一个与其关联的控制块。为了实现这一点，就必须有一个已经存在的指涉到当前对象的 std::shared_ptr（例如，某个在成员函数外部并调用了 shared_from_this 的函数）。如果这样的 std::shared_ptr 不存在（即当前对象尚没有与之相关联的控制块），该行为未定义，尽管通常的结果是 shared_from_this 抛出异常。

为了避免用户在 std::shared_ptr 指涉到该对象前就调用了引发 shared_from_this 的成员函数，继承自 std::enable_shared_from_this 的类通常会将其构造函数声明为 private 访问层级，并且只允许用户通过调用返回 std::shared_ptr 的工厂函数来创建对象。例如，Widget 看起来可能会长成这样：

```
class Widget: public std::enable_shared_from_this<Widget> {
public:
  // 将实参完美转发给 private 构造函数的工厂函数
  template<typename... Ts>
  static std::shared_ptr<Widget> create(Ts&&... params);

  ...
  void process();     // 同前
  ...

private:
  ...                 // 构造函数
};
```

绕了这么大一个圈子以后，你也许只能模糊地回想起，我们关于控制块的讨论是由理解与 std::shared_ptr 相关的成本所引发的。现在，既然我们已经了解如何避免创建过多的控制块，就让我们回归最初的话题吧。

一个控制块的尺寸通常只有几个字长，尽管自定义析构器和内存分配器可能会使其变得更大。通常的控制块的实现比你预期的要更加复杂。它使用了继承，甚至会用到虚函数（用以确保所指涉到的对象被适当地析构）。这就意味着，使用 std::shared_ptr 也会带来控制块用到的虚函数带来的成本。

读了这么多关于动态分配控制块、任意尺寸的析构器和内存分配器、虚函数功能以及原子引用计数操作的内容，你对于 std::shared_ptr 的热情也许在某种程度上会有所消减。这没关系。它们不一定对于所有的资源管理问题都是最好的解决方案。然而，对应于其提供的功能，std::shared_ptr 要求的成本非常合理。在典型情况下，在使

用了默认析构器和默认内存分配器，并且 std::shared_ptr 是由 std::make_shared 创建的前提下，控制块的尺寸只有三个字长，并且分配操作实质上没有任何成本（这些成本被并入至所指涉到的对象的内存分配中去了。欲了解更多细节，参见条款 21）。提领一个 std::shared_ptr 并不比提领一个裸指针花费更多。进行一项要求引用计数的操作（例如，复制构造函数或复制赋值，析构）需要一个或两个原子化操作，但这些操作通常会映射到单个机器指令。尽管与非原子化指令相比可能成本要高些，但仍是单指令。控制块中的虚函数机制通常只被每个托管给 std::shared_ptr 的对象使用一次：在该对象被析构的时刻。

付出了这样相当温和的成本以后，得到的却是动态分配资源的自动生存期管理。多数时候，使用 std::shared_ptr 来试图手动管理一个具有共享所有权的对象的生存期是十分可取的。如果你发现自己在怀疑是否承担得起使用 std::shared_ptr 的成本，那么请重新考虑一下你是否真的需要共享所有权。如果专属所有权能够胜任，或甚至只是可能能够胜任，std::unique_ptr 会是一个更好的选择。它的表现与裸指针的非常类似，并且从 std::unique_ptr“升级”到 std::shared_ptr 也很容易，因为 std::shared_ptr 可以由 std::unique_ptr 创建而来。

反之则并不成立。一旦你将资源的生存期托管给了 std::shared_ptr，就不能再更改主意了。即便引用计数为一，你也不能回收该资源的所有权，并让一个 std::unique_ptr 来管理它。资源和指涉到它的 std::shared_ptr 之间的规约是“至死方休”型的。不能离异，不能废止，不能免除。

另一些 std::shared_ptr 不能做的事情，就包括处理数组。和 std::unique_ptr 的另一个不同是，std::shared_ptr 的 API 仅被设计用来处理指涉到单个对象的指针。并没有所谓的 std::shared_ptr<T[]>。有时“聪明的”程序员会误打误撞地发现可以使用 std::shared_ptr<T> 来指涉到一个数组，并通过指定一个自定义析构器来完成数组删除操作（即 delete[]）。这种做法能够通过编译，但却是个糟糕透顶的主意。一方面，std::shared_ptr 并未提供 operator[]，这么一来，要取得数组的下标，就要求基于指针算术的笨拙表达式。另一方面，std::shared_ptr 支持派生到基类的指针型别转换，这对单个对象而言是有意义的，但是当应用到数组时，它就会在型别系统上开天窗（也正因为如此，std::unique_ptr<T[]> 的 API 禁止此型别转换）。更为重要的是，考虑到 C++11 提供了丰富多彩的内置数组选择（例如，std::array，std::vector，std::string），在这样的前提下还要声明一个智能指针指涉到一个非智能的数组，通常总是标志着设计的拙劣。

要点速记

- std::shared_ptr 提供方便的手段，实现了任意资源在共享所有权语义下进行生命周期管理的垃圾回收。
- 与 std::unique_ptr 相比，std::shared_ptr 的尺寸通常是裸指针尺寸的两倍，它还会带来控制块的开销，并要求原子化的引用计数操作。
- 默认的资源析构通过 delete 运算符进行，但同时也支持定制删除器。删除器的型别对 std::shared_ptr 的型别没有影响。
- 避免使用裸指针型别的变量来创建 std::shared_ptr 指针。

条款 20：对于类似 std::shared_ptr 但有可能空悬的指针使用 std::weak_ptr

如果某种智能指针能够像 std::shared_ptr 一样方便（参见条款 19），但又无须参与管理所指涉到的对象的共享所有权的话，虽然矛盾，却挺方便。换言之，这种指针像 std::shared_ptr 那样运作，但又不影响其指涉对象的引用计数。这么一来，这种指针就需要处理一个对 std::shared_ptr 而言不是问题的问题：其所指涉到的对象有可能已被析构。既然称为真正的智能指针，就应该能够通过跟踪指针何时空悬，亦即判断其所指涉到的对象已不复存在，来处理这个问题。这些正好描述了 std::weak_ptr 这种智能指针。

你可能会对 std::weak_ptr 有何用武之地感觉疑惑，而在看过 std::weak_ptr 的 API 后可能会更加疑惑。它看上去半点也不智能。std::weak_ptr 不能提领，也不能检查是否为空。这是因为 std::weak_ptr 并不是一种独立的智能指针，而是 std::shared_ptr 的一种扩充。

这种关系在指针生成的时刻就已存在。std::weak_ptr 一般者是通过 std::shared_ptr 来创建的。当使用 std::shared_ptr 完成初始化 std::weak_ptr 的时刻，两者就指涉到了相同位置，但 std::weak_ptr 并不影响所指涉到的对象的引用计数：

```
auto spw =
 std::make_shared<Widget>();                    // spw 构造成完成后，
                                                // 指涉到 Widget 的引用计数置为 1
                                                // （有关 std::make_shared 的信息
                                                // 参见条款 21）
  ...
```

```
std::weak_ptr<Widget> wpw(spw);          // wpw 和 spw 指涉到同一个 Widget。
                                         // 引用计数保持为 1
...

spw = nullptr;                           // 引用计数变为 0，
                                         // Widget 对象被析构。
                                         // wpw 空悬
```

std::weak_ptr 的空悬，也被称作失效（expired）。可以直接测试：

```
if (wpw.expired()) ...                   // 若 wpw 不再指涉到任何对象
```

但通常你想要的效果是：校验一个 std::weak_ptr 是否已经失效，如果尚未失效（即，并非空悬），就访问它所指涉到的对象。这个想起来容易，做起来难。由于 std::weak_ptr 缺乏提领操作，撰写不出这样的代码。即便这样的代码能够撰写出来，将检验和提领分离也会带来竞险：在 expired 的调用和提领操作之间，另一个线程可能重新赋值或析构最后一个指涉到该对象的 std::shared_ptr，而这会导致该对象被析构。在此情况下，提领会引发未定义行为。

你需要的是一个原子操作来完成 std::weak_ptr 是否失效的校验，以及在未失效的条件下提供对所指涉到的对象的访问。这个操作可以通过由 std::weak_ptr 创建 std::shared_ptr 来实现。该操作有两种形式，选择哪一种取决于由 std::weak_ptr 来创建 std::shared_ptr 时若该 std::weak_ptr 失效，期望得到什么结果。一种形式是 std::weak_ptr::lock，它返回一个 std::shared_ptr。如果 std::weak_ptr 已经失效，则 std::shared_ptr 为空：

```
std::shared_ptr<Widget> spw1 = wpw.lock();       // 若 wpw 失效，则 spw1 为空

auto spw2 = wpw.lock();                          // 同上，但使用 auto
```

另一种形式是用 std::weak_ptr 作为实参来构造 std::shared_ptr。这样，如果 std::weak_ptr 失效的话，抛出异常：

```
std::shared_ptr<Widget> spw3(wpw);               // 若 wpw 失效，
                                                 // 抛出 std::bad_weak_ptr 型别的异常
```

但你可能依然会对于 std::weak_ptr 有何用武之地感觉困惑。考虑一个工厂函数，该函数基于唯一 ID 来创建一些指涉到只读对象的智能指针。根据条款 18 中针对工厂函数返回型别的建议，它返回一个 std::unique_ptr：

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

如果 loadWidget 是成本高昂（例如，因为其执行了文件或数据库的 I/O 操作），并且 ID 会被频繁地重复使用的话，一个合理优化是撰写一个能够完成 loadWidget 的工作，

但又能缓存结果的函数。然而用缓存所有用过的 Widget 造成缓存拥塞，可能本身就会引起性能问题，因此另一个合理的优化就是在缓存的 Widget 不再有用时将其删除。

对于这个带缓存的工厂函数而言，返回 std::unique_ptr 型别并不十分合适。调用者当然应该获取指涉到缓存对象的智能指针，调用者也当然应该决定这些对象的生存期，然而缓存管理器也需要一个指涉到这些对象的指针。缓存管理器的指针需要能够校验它们何时会空悬，因为工厂函数的用户用完由该工厂函数返回的对象后，该对象就将被析构，此时相应的缓存条目将会空悬。因此，应该缓存 std::weak_ptr，一种可以检测空悬的指针。这也意味着，该工厂函数的返回值应为 std::shared_ptr，因为只有当对象的生存期托管给 std::shared_ptr 时，std::weak_ptr 才能检测空悬。

这里是带缓存的 loadWidget 的一个快速而粗糙的实现版本：

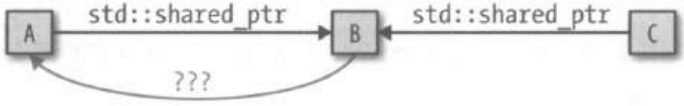
```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
  static std::unordered_map<WidgetID,
                            std::weak_ptr<const Widget>> cache;

  auto objPtr = cache[id].lock();       // objPtr 的型别是 std::shared_ptr
                                        // 指涉到缓存的对象
                                        // (如果对象不在缓存中，则返回空指针)

  if(!objPtr) {                         // 如果对象不在缓存中，
    objPtr = loadWidget(id);            // 则加载之
    cache[id] = objPtr;                 // 并缓存之
  }
  return objPtr;
}
```

该实现运用了 C++11 中的散列表容器（std::unorderer_map），尽管没有演示本应存在的 WidgetID 的散列值计算和相等性比较的函数。

fastLoadWidget 的实现忽略了一个事实，即由于相应的 Widget 不再使用（因此会被析构），缓存中的失效的 std::weak_ptr 可能会不断积累。这个实现可以被优化，不过与其花费时间在一个不会对 std::weak_ptr 提供额外洞见的议题上，我们不妨考虑第二种用例：观察者设计模式（Observer design pattern）。该模式的主要组件是主题（subject，可以改变状态的对象）和观察者（observer，对象状态发生改变后通知的对象）。在多数实现中，每个主题包含了一个数据成员，该成员持有指涉到其观察者的指针，这使得主题能够很容易地在其发生状态改变时发出通知。主题不会控制其观察者的生存期（即，不关心它们何时被析构），但需要确认的话当一个观察者被析构以后，主题不会去访问它。一种合理的设计就是让每个主题持有一个容器来放置指涉到其观察者的 std::weak_ptr，以便主题在使用某个指针之前，能够先确定它是否空悬。

关于 std::weak_ptr 的使用场合，再举最后一个例子，考虑一个包含 A、B、C 三个对象的数据结构，A 和 C 共享 B 的所有权，因此各持有一个指涉到 B 的 std::shared_ptr：

为了使用方便，假设有一个指针从 B 指回 A，那么该指针应该使用何种型别？



选择有三：

- **裸指针**。在此情况下，若 A 被析构，而 C 仍然指涉到 B，B 将保存着指涉到 A 的空悬指针。B 却检测不出来，所以 B 有可能无意中去提领这个空悬指针，就会产生未定义行为。

- **std::shared_ptr**。这种设计中，A 和 B 相互保存着指涉到对方的 std::shared_ptr，这种 std::shared_ptr 环路（A 指涉到 B 且 B 指涉到 A）阻止了 A 和 B 被析构。即使程序的其他数据结构已经不能再访问 A 和 B（例如，因为 C 不再指涉到 B），两者也会保待着彼此的引用计数为一。这样，实际上 A 和 B 已经发生了内存泄漏：因为程序已无法访问 A 和 B，但它们的资源也得不到回收。

- **std::weak_ptr**。这避免了上述两个问题。假如 A 被析构，那么 B 的回指指针将会空悬，但是 B 能够检测到这一点。更进一步，尽管 A 和 B 将指涉到彼此，但是 B 持有的指针不会影响 A 的引用计数，因此当 std::shared_ptr 不再指涉到 A 时，不会阻止 A 被析构。

显而易见，使用 std::weak_ptr 是其中最好的选择。然而值得指出的是，使用 std::weak_ptr 来打破 std::shared_ptr 引起的可能环路不是特别常见的做法。在类似树这种严格继承谱系式的数据结构中，子结点通常只被其父节点拥有，当父节点被析构后，子结点也应被析构。因此，从父节点到子节点的链接可以用 std::unique_ptr 来表示，而由子节点到父节点的反向链接可以用裸指针安全实现，因为子节点的生存期不会比父节点的更长，所以不会出现子节点去提领父节点空悬指针的风险。

当然，并非所有的基于指针的数据结构都是严格的继承谱系，在非严格继承谱系的数据结构中，以及缓存和观察者的列表实现等情况下，std::weak_ptr 就非常适用了。

从效率的角度来看，std::weak_ptr 和 std::shared_ptr 从本质上来说是一致的。std::weak_ptr 的对象和 std::shared_ptr 的对象尺寸相同，它们和 std::shared_ptr 使用同样的控制块（参见条款 19），其构造，析构，赋值操作都包含了对引用计数的原子操作。这种说法可能会令你惊讶，因为在本条款开头我曾提过，std::weak_ptr 不参与引用计数操作。其实我的说法与此有别，我说的是 std::weak_ptr 不干涉对象的共享所有权，因此不会影响所指涉到的对象的引用计数。实际上控制块里还有第二个引用计数，std::weak_ptr 操作的就是这第二个引用计数。更多细节请参考条款 21。

要点速记

- 使用 std::weak_ptr 来代替可能空悬的 std::shared_ptr。
- std::weak_ptr 可能的用武之地包括缓存，观察者列表，以及避免 std::shared_ptr 指针环路。

条款 21：优先选用 std::make_unique 和 std::make_shared，而非直接使用 new

让我们以把 std::make_unique 和 std::make_shared 放到同一起跑线作为本条款的开篇。std::make_shared 是 C++11 的一部分，但很可惜 std::make_unique 不是。它是在 C++14 中才加入标准库的。如果你在使用 C++11，不必担心，写出一个基础版本的 std::make_unique 易如反掌。请看这段代码：

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&...params)
{
  return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

如你所见，make_unique 仅仅是将其形参向待创建对象的构造函数作了一次完美转发，从一个 new 运算符产生的裸指针出发，构造了一个 std::unique_ptr，尔后返回了如此创建的这个 std::unique_ptr。这个形式的函数不支持数组和自定义析构器（参见条款 18），但它证明了只需通过一点点的努力就可以根据需要创建一个 make_unique。[注 3]

注 3：为使用最低成本创建一个全功能的 make_unique，可查阅创建它的标准化文档，然后复制找到的实现代码即可。目标文档编号是 N3656，作者 Stephan T. Lavavej，日期是 2013 年 4 月 18 日。

要记住的是，不要把你的版本放入 std 名字空间，因为当升级到 C++14 标准库实现时，你不会想让它和供应商提供的版本产生冲突。

std::make_unique 和 std::make_shared 是三个 make 系列函数中的两个。make 系列函数会把一个任意实参集合完美转发给动态分配内存的对象的构造函数，并返回一个指涉到该对象的智能指针。make 系列函数的第三个是 std::allocate_shared。它的行为和 std::make_shared 一样，只不过它的第一个实参是个用以动态分配内存的分配器对象。

即使对是否使用 make 系列函数来创建一个智能指针做最平凡的对比，也能揭示优先选用 make 系列函数的第一个原因。考虑如下代码：

```
auto upw1(std::make_unique<Widget>());          // 使用 make 系列函数

std::unique_ptr<Widget> upw2(new Widget);       // 不使用 make 系列函数

auto spw1(std::make_shared<Widget>());          // 使用 make 系列函数

std::shared_ptr<Widget> spw2(new Widget);       // 不使用 make 系列函数
```

我已经本质区别高亮标注了：使用了 new 的版本将被创建对象的型别重复写了两遍，但是 make 系列函数则没有。重复撰写型别，就违背了软件工程的一个重要原则：代码冗余应该避免。源代码中的重复会增加编译遍数，导致臃肿的目标代码，并且通常会产生更难上手的代码存根（code base）。它通常会演化成不一致的代码，而代码存根中的不一致性经常会导致代码缺陷。再说，输入两次要比输入一次更费工夫，谁不想减轻自己的打字负担呢？

优先使用 make 系列函数的第二个原因与异常安全有关。假设我们有一个函数依据某种优先级来处理一个 Widget 对象：

```
void processWidget(std::shared_ptr<Widget> spw, int priority);
```

对 std::shared_ptr 进行按值传递看起来颇为可疑，但是条款 41 解释了如果 processWidget 始终会构造 std::shared_ptr 的副本（例如，保存在一个数据结构里，该数据结构用来跟踪已经经过处理的 Widget 对象），这可能是个合理的设计选择。

现在假设有一个函数用来计算相对优先级：

```
int computePriority();
```

我们在 processWidget 的调用中用到该函数，并且在这次调用中，processWidget 使用了 new 运算符而非 std::make_shared：

```
processWidget(std::shared_ptr<Widget>(new Widget),        // 潜在的
              computePriority());                         // 资源泄漏
```

正如注释所指出的那样，这段代码可能会因为使用了new运算符而导致内存泄漏。这是怎么回事呢？发起调用的代码和接受调用的函数都在使用std::shared_ptr，而std::shared_ptr原本就是设计来避免资源泄漏的。当最后一个std::shared_ptr指涉到的对象消失时，它们会自动析构所指涉到的对象。人们在各处都使用了std::shared_ptr，代码在什么地方会发生资源泄漏呢？

答案与编译器从源代码到目标代码的翻译有关。在运行期，传递给函数的实参必须在函数调用被发起之前完成评估求值。因此，在processWidget的调用过程中，下列事件必须在processWidget开始执行前发生：

- 表达式“new Widget”必须先完成评估求值，即，一个Widget对象必须先在堆上创建。
- 由new产生的裸指针的托管对象std::shared_ptr<Widget>的构造函数必须执行。
- computePriority必须运行。

编译器不必按上述顺序来生成代码。“new Widget”必须在std::shared_ptr的构造函数得到调用前执行完毕，因为new表达式的结果将用作构造函数的实参之一。但是computePriority却可以在上述两个调用[译注1]之前、之后，甚至，在极端情况下，会在上述两个调用之间执行。也就是说，编译器可能会放出这样的代码，以按如下时序执行操作：

1. 实施“new Widget”。
2. 执行computePriority。
3. 运行std::shared_ptr构造函数。

如果生成了这样的代码，并且在运行期computePriority产生了异常，那么由第一步动态分配的Widget会被泄漏，因为它将永远不会被存储到在第三步才接管的std::shared_ptr中去。

使用std::make_shared可以避免该问题。调用代码如下：

```
processWidget(std::make_shared<Widget>(),        // 不会发生潜在的资源泄漏风险
              computePriority());
```

译注1：一个是new表达式的评估求值；另一个是std::shared_ptr的构造函数调用。

在运行期，std::make_shared 和 computePriority 中肯定有一个会首先被调用。如果是 std::make_shared 首先被调用，指涉到动态分配的 Widget 的裸指针会在 computePriority 被调用前被安全存储在返回的 std::shared_ptr 对象中。如果随后 computePriority 产生了异常，那么 std::shared_ptr 的析构函数也能够知道它所拥有的 Widget 已被析构。如果 computePriority 先被调用并产生了异常，std::make_shared 将不会被调用，因此也无须为动态分配的 Widget 担心。

如果我们把 std::shared_ptr 和 std::make_shared 分别替换成 std::unique_ptr 和 std::make_unique，则推理过程完全相同。欲编写异常安全的代码，则使用 std::make_unique 来代替 new 表达式和使用 std::make_shared 来代替 new 表达式道理是一样的。

std::make_shared 的另一个特色（与直接使用 new 表达式相比），是性能的提升。使用 std::make_shared 会让编译器有机会利用更简洁的数据结构产生更小更快的代码。考虑如下直接使用 new 的代码：

```
std::shared_ptr<Widget> spw(new Widget);
```

很显然这段代码会引发一次内存分配，但实际上会引发两次。条款 19 解释过，每个 std::shared_ptr 会指涉到一个控制块，除了其他东西之外，这个控制块包含了与所指涉到的对象相关联的引用计数。控制块的内存是 std::shared_ptr 的构造函数进行分配的。因此，直接使用 new 表达式的话，除了要为 Widget 进行一次内存分配，还要为与其相关联的控制块再进行一次内存分配。

如果使用 std::make_shared 来替代 new 表达式的话，

```
auto spw = std::make_shared<Widget>();
```

一次内存分配足矣。那是因为，std::make_shared 会分配单块（single chunck）内存既保存 Widget 对象又保存与其相关联的控制块。这种优化减小了程序的静态尺寸，因为代码只包含一次内存分配调用，同时还增加了可执行代码的运行速度，因为内存是一次性分配出来的。犹有进者，使用 std::make_shared 还能避免控制块中的一些簿记信息的必要性，这样也潜在地减少了程序的内存痕迹总量。

对 std::make_shared 的性能分析同样适用 std::allocated_shared，因此 std::make_shared 的性能优势也适用后者。

相对于直接使用 new 表达式，优先选用 make 系列函数，相关的论据都是十分强有力的。可是，尽管在软件工程、异常安全性以及效率方面皆有优势，本条款仍然主张的是优

先选用 make 系列函数，而非排他性地使用之。原因在于，还是有一些情景之下，不能或者不应使用 make 系列函数。

例如，所有的 make 系列函数都不允许使用自定义析构器（参见条款 18 和条款 19），但是 std::unique_ptr 和 std::shared_ptr 却都有着允许使用自定义析构器的构造函数。给定 Widget 对象的一个自定义析构器：

```
auto widgetDeleter = [](Widget* pw) { … };
```

欲创建一个使用该自定义析构器的智能指针，直接使用 new 表达式即可：

```
std::unique_ptr<Widget, decltype(widgetDeleter)>
  upw(new Widget, widgetDeleter);

std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

使用 make 系列函数，就无法做到相同的事情。

make 系列函数的第二个限制源于其实现的一个语法细节。条款 7 解释过，在创建对象时，如果其型别利用是否使用 std::initializer_list 型别的形参来重载构造函数，则如果在创建对象语句中使用大括号，会优先匹配形参型别为 std::initializer_list 的构造函数。反之，如果在创建对象语句中使用圆括号，则会非匹配形参型别为非 std::initializer_list 的构造函数。make 系列函数会向对象的构造函数完美转发其形参，但当它们到底是在使用圆括号的时候这样做，还是在使用大括号时这样做呢？对于某些型别而言，这个问题的答案会根据使用括号种类的不同而有很大不同。例如，在下述调用中：

```
auto upv = std::make_unique<std::vector<int>>(10, 20);

auto spv = std::make_shared<std::vector<int>>(10, 20);
```

作为运算结果的智能指针，到底指涉到的是个包含 10 个元素、每个元素值为 20 的 std::vector 呢，还是个包含 2 个元素、元素值分别是 10 和 20 的 std::vector 呢？还是结果未定呢？

好消息是，并非结果未定：两个调用都会创建出一个包含 10 个元素、每个元素值都是 20 的 std::vector。这说明，在 make 系列函数里，对形参进行完美转发的代码使用的是圆括号而非大括号。坏消息是，假如需要使用大括号初始化物来创建指涉到对象的指针，就必须直接使用 new 表达式了。使用 make 系列函数需要能够完美转发大括号初始化物的能力，但正如条款 30 中提及的那样，不能够完美转发大括号初始化物。但是，条款 30 也给出了一个变通方案：使用 auto 型别推导，从大括号初始化物出发，

创建一个 std::initializer_list 对象（参见条款 2），然后将 auto 创建的对象传递给 make 系列函数：

```
// 创建 std::initializer_list 型别的对象
auto initList = { 10, 20 };

// 利用 std::initializer_list 型别的构造函数创建 std::vector
auto spv = std::make_shared<std::vector<int>>(initList);
```

对于 std::unique_ptr 而言，其 make 系列函数仅在这两种场景下（自定义析构器和大括号初始化物）会产生问题。而对于 std::shared_ptr 和其 make 系列函数而言，还有其他两种场景。虽说都是边缘情况，但是有些程序员就是在边缘处求生存，你也许正是其中一员。

有些类会定义自身版本的 opeator new 和 operator delete，这些函数的存在意味着全局版本的内存分配和释放函数不适用于这种对象。通常情况下，类自定义的这两种函数被设计成仅用来分配和释放该类精确尺寸的内存块。例如，Widget 类的 operator new 和 operator delete 被设计用以处理尺寸恰好是 sizeof(Widget) 的内存块。这样的例程，就不适于用作 std::shared_ptr 所支持的那种自定义分配器（通过 std::allocate_shared）和释放器（通过自定义析构器）了。因为 std::allocate_shared 所要求的内存数量并不等于动态分配对象的尺寸，而是该尺寸的基础上加上控制块的尺寸。因此，使用 make 系列函数去为带有自定义版本的 operate new 和 operator delete 的类创建对象，通常并不是个好主意。

使用 std::make_shared 之所以相对于直接使用 new 表达式存在尺寸和速度上的优势，源于前者使得 std::shared_ptr 的控制块和托管对象在同一内存块上分配。当对象的引用计数变为零时，对象被析构（即，析构函数被调用）。然而，托管对象所占用的内存直到与其关联的控制块也被析构时才会被释放，因为同一动态分配的内存块同时包含了两者。

如前所述，控制块中除了引用计数本身还包含了其他一些信息。引用计数会跟踪有多少个 std::shared_ptr 指涉到该控制块，但控制块中还包含着第二个引用计数，它针对指涉到该控制块的 std::weak_ptr 进行计数。这第二个引用计数被称作弱计数。[注 4] std::weak_ptr 通过检查控制块里的引用计数（而非弱计数）来校验自己是否失效（参

注 4：实际上，弱计数的值并不始终等于指涉到控制块的 std::weak_ptr 的数量，因为库的实现者已经找到了某些方法向弱计数加入额外信息以促进更好的代码生成。但在本条款中，我们将忽略这一点，并假设弱计数的值始终等于指涉到控制块的 std::weak_ptr 的数量。

见条款 19）。假如引用计数为零（即，没有 std::shared_ptr 指涉到它指涉到的对象，从而该对象已经被析构），则 std::weak_ptr 就已失效，否则就未失效。

由于 std::weak_ptr 会指涉到某个控制块（即，弱计数大于零），该控制块肯定会持续存在。而由于控制块存在，包含它的内存肯定会持续存在。这么一来，通过对应于 std::shared_ptr 的 make 系列函数所分配的内存在最后一个 std::shared_ptr 和最后一个指涉到它的 std::weak_ptr 都被析构之前，无法得到释放。

假如对象型别的尺寸很可观，且最后一个 std::shared_ptr 被析构和最后一个 std::weak_ptr 析构之间的时间间隔不能忽略时，在对象的析构和内存的释放之间就会产生延迟。

```
class ReallyBigType { … };

auto pBigObj =                                  // 通过 std::make_shared
  std::make_shared<ReallyBigType>();            // 创建颇大的对象

… // 创建指涉到大对象的多个 std::shared_ptr 和 std::weak_ptr
  // 并使用这些智能指针来操作该对象

… // 最后一个指涉到该对象的 std::shared_ptr 在此被析构，
  // 但指涉到该对象的若干 std::weak_ptr 仍然存在

… // 在此阶段，前述大对象所占用的内存仍处于分配未回收状态


… // 最后一个指涉到该对象的 std::weak_ptr 在此被析构，
  // 控制块和对象所占用的同一内存块在此得到释放
```

而若直接使用 new 表达式，则 ReallyBigType 对象的内存可以在最后一个指涉到它的 std::shared_ptr 析构时就被释放：

```
class ReallyBigType { … };              // 同前

std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
                                        // 通过 new 表达式创建颇大的对象

…       // 同前，创建指涉到大对象的
        // 多个 std::shared_ptr 和 std::weak_ptr，
        // 并使用这些智能指针来操作该对象

…       // 最后一个指涉到该对象的 std::shared_ptr 在此被析构，
        // 但指涉到该对象的若干 std::weak_ptr 仍然存在，
        // 前述大对象所占用的内存在此被回收

…       // 在此阶段，仅控制块所占用的内存仍处于分配未回收状态

…       // 最后一个指涉到该对象的 std::weak_ptr 在此被析构，
        // 控制块所占用的内存在此得到释放
```

一旦当你发现自己处在一个不能够或者不适合使用 std::make_shared 的境地，你就要确保避免之前见过异常安全问题。最好的办法是确保你直接使用 new 表达式的时候，立即将该表达式的结果传递给智能指针的构造函数，并且在这样一条语句里不要做其他任何事。只有这样，才能阻止编译器在 new 表达式的评估求值和调用智能指针的构造函数并接管 new 表达式产生的对象这个过程之间放出异常来。

作为例子，我们对之前考察过的、那条非异常安全的调用语句所涉及的 processWidget 函数进行一个小修订。这次，我们指定一个自定义析构器：

```
void processWidget(std::shared_ptr<Widget> spw,          // 同前
                   int priority);

void cusDel(Widget *ptr);                                // 自定义析构器
```

下面就是那条非异常安全的调用语句，根据这次修订作了相应调整：

```
processWidget(                                           // 同前
  std::shared_ptr<Widget>(new Widget, cusDel),           // 潜在的
  computePriority()                                      // 资源泄漏!
);
```

回忆一下：假如 computePriority 的调用在“new Widget”之后，但在 std::shared_ptr 的构造函数之前，则若 computePriority 产生了异常，那么动态分配的 Widget 将会泄漏。

由于这次使用了自定义析构器，所以就无法使用 std::make_shared。因此，避免该问题的方法是把 Widget 的内存分配和 std::shared_ptr 的构造函数放到它们自己的一条单独语句中，然后再以作为结果的 std::shared_ptr 为实参去调用 processWidget。这就是该技巧的本质所在，尽管我们即将看到，通过少许修改就可以提升其性能：

```
std::shared_ptr<Widget> spw(new Widget, cusDel);

processWidget(spw, computePriority());          // 正确，但并非最优化，见下
```

由于 std::shared_ptr 拥有传递给其构造函数的裸指针的所有权，所以即使在其构造函数产生异常时，也不会发生问题。在本例中，如果 spw 的构造函数产生异常（例如，由于为控制块分配动态内存失败），它仍能保证 cusDel 针对“new Widget”返回的结果指针完成调用。

这里存在一处小小的性能隐患：在非异常安全的调用语句中，传递给 processWidget 的是个右值：

```
processWidget(
  std::shared_ptr<Widget>(new Widget, cusDel),           // 实参是个右值
```

```
    computePriority()
);
```

而在异常安全的调用语句中，传递的则是个左值：

```
processWidget(spw, computePriority()); // 实参是个左值
```

由于 processWidget 的 std::shared_ptr 形参是按值传递的，从右值出发来构造之仅仅需要一次移动，而从左值出发去构造之则需要一次复制。对于 std::shared_ptr 而言，这个区别可能会相当可观，因为复制一个 std::shared_ptr 要求对其引用计数进行一次原子的递增操作，而移动一个 std::shared_ptr 则不要求对其引用计数进行任何操作。如果我们希望这段异常安全的代码达到非异常安全的代码同样的性能，就需要对 spw 上实施 std::move，把它转化成一个右值（参见条款 23）：

```
processWidget(std::move(spw),          // 既有效率，又异常安全
              computePriority());
```

这一点很有意思，也值得了解，但通常也无关紧要，因为你很少会有理由不使用 make 系列函数。而除非有充分的理由不去使用 make 系列函数，否则都应该使用它们。

要点速记

- 相比于直接使用 new 表达式，make 系列函数消除了重复代码、改进了异常安全性，并且对于 std::make_shared 和 std::allcoated_shared 而言，生成的目标代码会尺寸更小、速度更快。
- 不适于使用 make 系列函数的场景包括需要定制删除器，以及期望直接传递大括号初始化物。
- 对于 std::shared_ptr，不建议使用 make 系列函数的额外场景包括：①自定义内存管理的类；②内存紧张的系统、非常大的对象、以及存在比指涉到相同对象的 std::shared_ptr 生存期更久的 std::weak_ptr。

条款 22：使用 Pimpl 习惯用法时，将特殊成员函数的定义放到实现文件中

假如曾经有过和过长构建时间的抗争经历，你就应该熟悉 Pimpl 习惯用法（“Pimpl”意为“pointer to implementation”，即指涉到实现的指针）。这种技巧就是把某类的数据成员用一个指涉到某实现类（或结构体）的指针替代，尔后把原来在主类中的数

据成员放置到实现类中，并通过指针间接访问这些数据成员。例如，考虑某 Widget 类如下：

```
class Widget {                    // 位于头文件 "widget.h" 内
public:
  Widget();
  ...
private:
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;              // Gadget 是某种用户自定义型别
};
```

因为 Widget 的数据成员属于 std::string、std::vector 和 Gadget 等多种型别，这些型别所对应的头文件必须存在，Widget 才能通过编译，这就说明 Widget 的客户必须 #include <string>、<vector>，以及 gadget.h。这些头文件增加了 Widget 的客户的编译时间，此外，它们也使得这些客户依赖于这些头文件的内容。假如某个头文件的内容发生了改变，则 Widget 的客户必须重新编译。标准头文件 <string> 和 <vector> 不会经常改变，但 gadget.h 却有可能会经常修订。

把 C++98 中的 Pimpl 习惯用法在这里实施，可以用一个指涉到已声明但未定义的结构的裸指针来替换 Widget 的数据成员：

```
class Widget {                    // 仍位于头文件 "widget.h" 内
public:
  Widget();
  ~Widget();                      // 析构函数变得必要，见下
  ...

private:
  struct Impl;                    // 声明实现结构体
  Impl *pImpl;                    // 以及指涉到它的指针
};
```

由于 Widget 不再提及 std::string，std::vector 和 Gadget 型别，Widget 客户不再需要 #include 这些型别的头文件。这会使编译速度提升，同时也意味着即使这些头文件的内容发生了改变，Widget 的客户也不会受到影响。

一个已声明但未定义的型别称为非完整型别。Widget::Impl 就是这样的型别。针对非完整型别，可以做的事情极其有限，但是声明一个指涉到它的指针就是其中之一。Pimpl 习惯用法正是利用了这一点。

Pimpl 习惯用法的第一部分，是声明一个指针型别的数据成员，指涉到一个非完整型别，第二部分是动态分配和回收持有从前在原始类里的那些数据成员的对象，而分配和回收代码则放在实现文件中。例如，对于 Widget 而言，这部分代码就位于 widget.cpp 内：

```
#include "widget.h"                    // 位于实现文件 "widget.cpp" 内
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {                  // Widget::Impl 的实现
  std::string name;                    // 包括此前在 Widget 中的数据成员
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                       // 为本 Widget 对象分配数据成员所需内存
: pImpl(new Impl)
{}

Widget::~Widget()                      // 为本 Widget 对象析构数据成员
{ delete pImpl; }
```

这里，我把 #include 指令展示出来，是为了说明对 std::string、std::vector 和 Gadget 所对应的头文件的总体依赖仍然存在。然而，这些依赖已经从 widget.h（对 Widget 客户可见并由他们使用）转移到了 widget.cpp 中（因而只对 Widget 实现者可见并被使用）。我也已经高亮标注了动态分配和回收 Impl 对象的代码。由于当 Widget 被析构时需要回收该对象，所以就要求 Widget 具备一个析构函数。

但是我展示的是 C++98 的代码，它们散发着上一个千年的腐朽气息。其中使用了裸指针、裸 new 运算符和裸 delete 运算符，一切都是如此地……赤裸裸。本章的主旨在于优先选用智能指针，而非裸指针。如果我们需要的是在 Widget 构造函数里动态分配一个 Widget::Impl 对象，同时在 Widget 析构时自动释放该对象，那么 std::unique_ptr（参见条款 18）正是我们所需要的工具。用 std::unique_ptr 替代指涉到 Impl 的裸指针以后，头文件的代码就会长成这样：

```
class Widget {                         // 位于头文件 "widget.h" 内
public:
  Widget();
  ...

private:
  struct Impl;
  std::unique_ptr<Impl> pImpl;         // 使用智能指针而非裸指针
};
```

而实现文件则如此这般：

```
#include "widget.h"                    // 位于实现文件 "widget.cpp" 内
#include "gadget.h"
#include <string>
#include <vector>
```

```
struct Widget::Impl {                    // 同前
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                         // 根据条款 21
: pImpl(std::make_unique<Impl>())        // 使用 std::make_unique
{}                                       // 创建 std::unique_ptr
```

你可能会注意到，Widget的析构函数不复存在了。那是因为，我们无须再为其撰写代码。当 std::unique_ptr 被析构时，它会自动析构它所指涉到的对象，因此我们不需要自己析构任何东西。这也正是智能指针吸引人的地方之一：它们消除了手动释放资源的要求。

这段代码本身能够通过编译，但遗憾的是甚至最平凡的客户代码都通不过编译：

```
#include "widget.h"

Widget w;                                // 错误!
```

具体产生的错误信息，依赖你使用的编译器种类。但错误信息文本通常会提及诸如在非完整型别上实施了 sizeof 或 delete 运算符，而这些属于不可以实施于该种型别的操作之列。

使用 std::unique_ptr 来实现 Pimpl 习惯用法会导致惨败，这让人十分震惊，原因是：① std::unique_ptr 号称可以支持非完整型别；② Pimpl 习惯用法是 std::unique_ptr 最广泛应用的场景之一。幸运的是，使这些代码正常运作起来十分容易，只需对于问题的产生原因有个基本的了解即可。

该问题是由 w 被析构时（例如，离开作用域时）所生成的代码引起的。在那一时刻，析构函数被调用，在使用了 std::unique_ptr 的类定义里，我们未声明析构函数，因为无须为其撰写代码。根据编译器生成特种成员函数的基本规则（参见条款 17），编译器为我们生成了一个析构函数。在该析构函数内，编译器会插入代码来调用 Widget 的数据成员 pImpl。pImpl 是个 std::unique_ptr<Widget::Impl> 型别的对象，即一个使用了默认析构器的 std::unique_ptr。默认析构器是在 std::unique_ptr 内部使用 delete 运算符来针对裸指针实施析构的函数。然而，在实施 delete 运算符之前，典型的实现会使用 C++11 中的 static_assert 去确保裸指针未指涉到非完整型别。这么一来，当编译器为 Widget w 的析构函数产生代码时，通常就会遇到一个失败的 static_assert，从而导致了错误信息的产生。这个错误信息和 w 被析构的位置有关，因为Widget的析构函数与其他编译器产生的特种成员函数一样，基本上隐式 inline 的。

这个编译错误通常会标示在w生成的那一代码行，因为正是这行源代码的显式创建对象导致了后来的隐式析构。

为解决这一问题，只需保证在生成析构std::unique<Widget::Impl>代码处，Widget::Impl是个完整型别即可。只要型别的定义可以被看到，它就是完整的。而Widget::Impl的定义位于widget.cpp中的。因此，成功编译的关键在于让编译器看到Widget的析构函数的函数体（即，编译器将要生成代码来析构std::unique_ptr型别的数据成员之处）的位置在widget.cpp内部的Widget::Impl定义之后。

这个实现起来就简单了。在widget.h内声明Widget的析构函数，但不要在那里定义它：

```
class Widget {                  // 同前，位于头文件 "widget.h" 内
public:
  Widget();
  ~Widget();                    // 仅声明
  ...

private:                        // 同前
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

在widget.cpp内定义之，位置在Widget::Impl定义之后：

```
#include "widget.h"             // 同前，位于实现文件 "widget.cpp" 内
#include "gadget.h"
#include <string>
#include <vector>

  struct Widget::Impl {         // 同前，Widget::Impl
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                // 同前
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget()               // ~Widget 的定义
{}
```

这段代码运作如期，打字也最少。但如果你想强调，由编译器生成的析构函数会做正确的事情，而声明它的唯一理由只是想要使得其定义出现在Widget的实现文件中，那么你可以在析构函数的函数体后写上“=default”来表达该定义：

```
Widget::~Widget() = default;  // 效果同上
```

使用了 Pimpl 习惯用法的类，自然支持移动操作，因为编译器生成的移动操作完全符合预期，即针对 std::unique_ptr 执行移动操作。可是条款 17 解释过，在 Widget 中声明析构函数的举动会阻止编译器产生移动操作，所以假如你需要支持移动操作，就必须自己声明该函数。既然编译器产生的版本是正确的，你很有可能会尝试如下实现：

```
class Widget {                                  // 仍位于头文件 "widget.h" 内
public:
  Widget();
  ~Widget();

  Widget(Widget&& rhs) = default;               // 想法正确
  Widget& operator=(Widget&& rhs) = default;    // 代码错误!

  ...

private:                                        // 同前
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

这种手法会导致和类中没有声明析构函数一样的问题，产生该问题的基本原因也相同。编译器生成的移动赋值操作需要在重新赋值前析构 pImpl 指涉到的对象，但在 Widget 的头文件里 pImpl 指涉到的是个非完整型别。move 构造函数出问题的原因有所不同，这里的问题在于，编译器会在 move 构造函数内抛出异常的事件中生成析构 pImpl 的代码，而对 pImpl 析构要求 Impl 具备完整型别。

由于产生原因一如此前，修复手法也是如法炮制，把移动操作的定义移入实现文件里：

```
class Widget {                          // 仍位于头文件 "widget.h" 内
public:
  Widget();
  ~Widget();

  Widget(Widget&& rhs);                 // 仅声明
  Widget& operator=(Widget&& rhs);

  ...

private:                                // 同前
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};

#include <string>                       // 仍位于实现文件 "widget.cpp" 内
...

struct Widget::Impl { ... };            // 同前

Widget::Widget()                        // 同前
```

```
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;                              // 同前

Widget::Widget(Widget&& rhs) = default;                   // 在这里
Widget& Widget::operator=(Widget&& rhs) = default;        // 放置定义
```

Pimpl 习惯用法是一种可以在类实现和类使用者之间减少编译依赖性的方法，但从概念上说，Plimpl 习惯用法并不能改变类所代表的事物。最初的 Widget 类包含了 std::string，std::vector 以及 Gadget 型别数据成员，如果假设 Gadget 类像 std::string 和 std::vector 一样可以复制，那么自然 Widget 类也支持复制操作。我们需要自己撰写这些函数，原因是：①编译器不会为像 std::unique_ptr 那样的只移型别生成复制操作；②即使编译器可以生成，其生成的函数也只能复制 std::unique_ptr（即，实施的是浅复制），而我们希望的则是复制指针所指涉到的内容（即，实施深复制）。

根据目前已经熟悉的规矩，我们在头文件里声明这些函数，并在实现文件里实现它们：

```
class Widget {                              // 仍位于头文件 "widget.h" 内
public:
  ...                                       // 其他函数，同前

  Widget(const Widget& rhs);                // 仅声明
  Widget& operator=(const Widget& rhs);

private:                                    // 同前
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};


#include "widget.h"                         // 同前，位于实现文件 "widget.cpp" 内
...

struct Widget::Impl { ... };                // 同前

Widget::~Widget() = default;                // 其他函数，同前

Widget::Widget(const Widget& rhs)                   // 复制构造函数
:  pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs)        // 复制赋值运算符
{
  *pImpl = *rhs.pImpl;
  return *this;
}
```

两种函数的实现都符合常规，在每一种情况中，每种情况下，我们都是从源对象（rhs）到目的对象（*this）简单地复制了Impl的结构体的内容。我们并未逐项地复制每个字段，而是利用了编译器为Impl类创建的复制操作会自动逐项复制这些字段的特性。总之，我们通过采用Widget::Impl的编译器生成的复制操作实现了Widget的复制操作。值得注意的是，在此复制构造函数中，我们还遵循了条款21的建议：尽量使用std::make_unique而非直接使用new表达式。

为达到实现Pimpl习惯用法的目的，应该选用的是std::unique_ptr智能指针，因为在对象内部（例如，在Widget内部）的pImpl指针拥有相应的实现对象（例如，Widget::Impl对象）的专属所有权。值得指出的是，如果我们在这里使用std::share_ptr而非std::unique_ptr来实现pImpl，则本条款的建议不再适用。在这个前提下，无须在Widget中声明析构函数。而由于没有用户自定义的析构函数，编译器会很乐意生成移动操作并精确地按我们想要的方式运作。给定widget.h中的代码如下：

```
class Widget {                    // 位于头文件 "widget.h" 内
public:
  Widget();
  …                               // 不再有析构函数或移动操作的声明

private:
  struct Impl;
  std::shared_ptr<Impl> pImpl; // 使用 std::shared_ptr
};                                // 而非 std::unique_ptr
```

使用#include指令包含widget.h以后的客户代码如下：

```
Widget w1;

auto w2(std::move(w1));           // 针对 w2 实施移动构造

w1 = std::move(w2);               // 针对 w2 实施移动赋值
```

一切编译和运行结果都如所期：w1会被默认构造，其值被移动到w2，然后该值又被移回w1，最后w1和w2都被析构（从而导致指涉到的Widget::Impl类对象被析构）。

std::unique_ptr和std::shared_ptr这两种智能指针在实现pImpl指针行为时的不同，源自它们对于自定义析构器的支持的不同。对于std::unique_ptr而言，析构器型别是智能指针型别的一部分，这使得编译器会产生更小尺寸的运行期数据结构以及更快速的运行期代码。如此高效带来的后果是，欲使用编译器生成的特种函数（例如，析构函数或移动操作），就要求其指涉到的型别必须是完整型别。而对于std::shared_ptr而言，析构器的型别并非智能指针型别的一部分，这就需要更大尺寸的运行时期

数据结构以及更慢一些的目标代码，但在使用编译器生成的特种函数时，其指涉到的型别却并不要求是完整型别。

就 Pimpl 习惯用法而言，并不需要在 std::unique_ptr 和 std::shared_ptr 的特性之间作出权衡，因为 Widget 和 Widget::Impl 这样的类之间的关系是专属所有权，所以在此处 std::unique_ptr 就是完成任务的合适工具。话说回来，还是需要了解，在其他情景下——存在共享所有权的情景下（从而 std::shared_ptr 成为合适的设计选项），就大可不必忍受 std::unique_ptr 所带来的必须自行撰写一系列函数定义的煎熬。

要点速记

- Pimpl 惯用法通过降低类的客户和类实现者之间的依赖性，减少了构建遍数。
- 对于采用 std::unique_ptr 来实现的 pImpl 指针，须在类的头文件中声明特种成员函数，但在实现文件中实现它们。即使默认函数实现有着正确行为，也必须这样做。
- 上述建议仅适用于 std::unique_ptr，但并不适用 std::shared_ptr。

第 5 章

右值引用、移动语义和完美转发

你在初次接触移动语义和完美转发这些概念的时候，它们显得非常名副其实：

- **移动语义**使得编译器得以使用不那么昂贵的移动操作，来替换昂贵的复制操作。同复制构造函数、复制赋值运算符给予人们控制对象复制的具体意义的能力一样，移动构造函数和移动赋值运算符也给予人们控制对象移动语义的能力。移动语义也使得创建只移型别对象成为可能，这些型别包括 `std::unique_ptr`、`std::future` 和 `std::thread` 等。
- **完美转发**使得人们可以撰写接受任意实参的函数模板，并将其转发到其他函数，目标函数会接受到与转发函数所接受的完全相同的实参。

右值引用是将这两个风马牛不相及的语言特性胶合起来的底层语言机制，正是它使得移动语义和完美转发成为了可能。

你和这两个语言特性打的交道越多，就越会发现你对它们的初始印象好像只是冰山一角。移动语义、完美转发和右值引用的世界比它们乍看上去要微妙得多。比如说，`std::move` 并没有移动任何东西。再比如说，完美转发并不完美。还有，移动操作的成本并不总是比复制低，即使低也不一定像你期望的那样低。并且，在移动操作能够成立的语境中，却并不一定能够调用到移动操作。形如“*type&&*”的结构，也不总是表示右值引用。

无论在这些特性上钻研多深，好像始终还有更多未知等待开启。幸运的是，它们深而有底，本章就会带你们到最底层的基石一游。而到达了这一层后，C++11 的这个部分就会豁然开朗。例如，你会理解 `std::move` 和 `std::forward` 的用法约定。你也会对

于“*type&&*”的天性多义感觉轻松。你还会懂得移动操作为何有如此丰富多样的行为表现形式。所有这些碎片化的知识都会逐一归位。到了那一刹那，你会回见初心，因为你会再一次地发现移动语义、完美转发和右值引用显得那样的名副其实。但这一次，这种感觉将维持不变。

在阅读本章中的条款时，一定要把这一点铭记在心：形参总是左值，即使其型别是右值引用。即，给定函数形如：

```
void f(Widget&& w);
```

形参 w 是个左值。即使它的型别是个指涉到 Widget 型别对象的右值引用（如果你对此感觉意外，请参阅本书第 1 章就写着的关于左值和右值的概述）。

条款 23：理解 std::move 和 std::forward

从讲述 std::move 和 std::forward 不做什么来开始这一章，不无裨益。须知，std::move 并不进行任何移动，std::forward 也不进行任何转发。[译注 1] 这两者在运行期都无所作为。它们不会生成任何可执行代码，连一个字节都不会生成。

std::move 和 std::forward 都是仅仅执行强制型别转换的函数（其实是函数模板）。std::move 无条件地将实参强制转换成右值，而 std::forward 则仅在某个特定条件满足时才执行同一个强制转换。就是这样。虽然这样的解释可能会引发一堆新的问题，但是，从根本意义上说，故事讲到这里就全部讲完了。

要把故事讲得更具体一些的话，请看看下面这个 C++11 中 std::move 的示例实现。它不完全符合标准的所有细节，但已经非常接近了：

```
template<typename T>                                // 位于名字空间 std 内
typename remove_reference<T>::type&&
move(T&& param)
{
  using ReturnType =                                // 别名声明
    typename remove_reference<T>::type&&;           // 参见条款 9

  return static_cast<ReturnType>(param);
}
```

我在这段代码中突出标注了两处。一处是函数名字，因为返回值的型别规格实在冗长，我不想让它喧宾夺主。还有一处就是那个强制型别转换语句，这构成了函数的实质部

译注 1：英文单词“move”的字面意义是“移动”，“forward”的字面意义是“转发”。此处强调的是函数的行为与其字面意义不符，这一点无法翻译，只能注出。下同。

分。如你所见，std::move的形参是指涉到一个对象的引用（准确地说，是万能引用——参见条款24），它返回的是指涉到同一个对象的引用。

函数返回值的“&&”部分，暗示着std::move返回的是个右值引用。但是，如条款28所言，如果T碰巧是个左值引用的话，那么T&&就成了左值引用。为了避免这种情况发生，它将型别特征（参见条款9）std::remove_reference应用于T，从而保证“&&”应用在一个非引用型别之上。这么一来，就可以确保std::move返回的是右值引用，而这一点十分重要，因为从该函数返回的右值引用肯定是右值。综上所述，std::move将实参强制转换成了右值，而这就是该函数全部的所作所为。

多说一句，如果采用C++14，std::move就可以以更简明扼要的方式实现。有了函数返回值型别推导（参见条款3）和标准库中的别名模板std::remove_reference_t（参见条款9），std::move就可以这样撰写：

```
template<typename T>                              // C++14 实现
decltype(auto) move(T&& param)                    // 仍在名字空间 std 内
{
  using ReturnType = remove_reference_t<T>&&;
  return static_cast<ReturnType>(param);
}
```

看起来省力多了，不是吗？

由于std::move只做一件事，就是把实参强制转换成右值，就有人曾经提过建议，说该函数更好的命名可能会是rvalue_cast之类。即便有过这样的建议，现在定下来的命名已经是std::move，我们就得记牢std::move要做什么，不做什么。它做的是强制型别转换，不做的是移动。

当然，右值是可以实施移动的，所以在一个对象上实施了std::move，就是告诉编译器该对象具备可移动的条件。这就是std::move得名的原因：它简化了对象是否可移动的表述。

说实话，右值也仅在通常情况下能够移动。假设你在撰写一个用以表示注解的类。该类的构造函数取用了一个std::string型别的形参来构成注解，并将该形参复制到一个数据成员中。参阅了条款41的内容以后，你声明了一个按值传递的形参：

```
class Annotation {
public:
  explicit Annotation(std::string text);          // 待复制的形参，
  …                                               // 根据条款 41 的说法，选择按值传递
};
```

但是Annotation的构造函数只需要读取text的值，而不需要修改。所以，遵循着

由来已久的传统“只要有可能使用 const 就使用”，你修订了代码，把 text 声明为 const：

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  …
};
```

为了避免付出将 text 复制入数据成员的过程产生的复制操作成本，你按照条款 41 的建议，对 text 实施了 std::move，从而产生了一个右值：

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  : value(std::move(text))              // 将 text “移动入” value；
  { … }                                 // 这段代码实际的所作所为和乍看之下有所不同！

  …
private:
  std::string value;
};
```

这段代码顺利地通过了编译。这段代码顺利地完成了链接。这段代码顺利地跑了起来。这段代码把数据成员 value 设置成了 text 的内容。这段代码近乎完美地实现了你想像中一切，唯一美中不足的是，text 并非是被移动，它还是被复制入 value 的。没错，text 已经被 std::move 强制转换成为一个右值。但是，text 是被声明为 const std::string 的，所以在强制转换之前，是个左值 const std::string，而强制转换的结果是个右值 const std::string，经过这番折腾以后常量性保留了下来。

考虑一下，当编译器需要决定调用哪一个 std::string 的构造函数时，会面临什么情景。选项有两个：

```
class string {                          // std::string 实际上是个 typedef，
public:                                 // 代表 std::basic_string<char>
  …
  string(const string& rhs);            // 复制构造函数
  string(string&& rhs);                 // 移动构造函数
  …
};
```

在 Annotation 的构造函数的成员初始化列表中，std::move(text) 的结果是个 const std::string 型别的右值。这个右值无法传递给 std::string 的移动构造函数，因为移动构造函数只能接受非常量 std::string 型别的右值引用作为形参。可是，这样一个右值却可以传递给复制构造函数，因为指涉到常量的左值引用允许绑定到一个常量右值型别的形参。因此，成员初始化最终会调用的是 std::string 的复制构造函数，即

使 text 已经被转换成了一个右值！这种行为对于维持常量正确性至关重要，因为将值移出对象通常会改动该对象，所以语言不应该允许常量对象传递到有可能改动它们的函数（例如移动构造函数）。

从本例中，可以习得两点经验：首先，如果想取得对某个对象执行移动操作的能力，则不要将其声明为常量，因为针对常量对象执行的移动操作将一声不响地变换成复制操作；其次，std::move 不仅不实际移动任何东西，甚至不保证经过其强制型别转换后的对象具备可移动的能力。关于针对任意对象实施过 std::move 的结果，唯一可以确定的是，该结果会是个右值。

std::forward 的身世与 std::move 类似，只是 std::move 会无条件地将其实参强制转换为右值型别不同，std::forward 仅在特定条件下才实施这样的强制型别转换。换言之，std::forward 是一个有条件强制型别转换。为了理解这强制型别转换何时实施，何时不实施，回忆一下 std::forward 的典型使用场景。其中最常见的，就是某个函数模板取用了万能引用型别为形参，随后将其传递给另一个函数：

```
void process(const Widget& lvalArg);          // 处理左值
void process(Widget&& rvalArg);               // 处理右值

template<typename T>                          // 把 param 传递给 process
void logAndProcess(T&& param)                 // 的函数模板
{
  auto now =                                  // 取得当前时间
    std::chrono::system_clock::now();

  makeLogEntry("Calling 'process'", now);
  process(std::forward<T>(param));
}
```

考虑两种调用 logAndProcess 的情形，一种向其传入左值，一种向其传入右值：

```
Widget w;

logAndProcess(w);              // 调用时传入左值
logAndProcess(std::move(w));   // 调用时传入右值
```

在 logAndProcess 内，形参 param 被传递给函数 process。而 process 依据形参是左值还是右值型别进行了重载，所以我们很自然地会期望，当调用 logAndProcess 时若传入的是个左值，则该左值可以在被传递给 process 函数仍被当作一个左值；而当调用 logAndProcess 时若传入的是个右值，则调用是 process 取用右值型别的那个重载版本。

但是，所有函数形参皆为左值，param 亦不例外。是故，所有 logAndProcess 内对 process 的调用都会是取用了左值型别的那个重载版本。为了避免这种结果，就需要一

种机制，当且仅当用来初始化 param 的实参（即传递给 logAndProcess 的实参）是个右值的条件下，把 param 强制转换成右值型别。这恰恰就是 std::forward 所做的一切。这就是为何说 std::forward 是有条件强制型别转换：仅当其实参是使用右值完成初始化时，它才会执行向右值型别的强制型别转换。

你可能会疑惑，std::forward 何以知晓其实参是否通过右值完成初始化。例如，在上述代码中，std::forward 是如何分辨 param 是通过左值还是右值完成了初始化的呢？一句话：该信息是被编码到 logAndProcess 的模板形参 T 中的。该形参被传递给 std::forward 后，随即由后者将编码了信息恢复出来。具体的原理细节参见条款 28。

既然 std::move 和 std::forward 归根结底都不过是强制型别转换，唯一不同就在于 std::move 始终实施强制型别转换，而 std::forward 仅仅有时会实施。这么一来，你不免会问，是否可以弃用 std::move 而只用 std::forward。从纯粹技术的视角，答案是肯定的：有 std::forward 就足够了，std::move 并不是必不可少的。当然如果这么说的话，这两个函数没有一个是必不可少的，因为我们可以在要用的地方自行撰写这些强制型别转换嘛。但是，我希望你们能够同意说如果真这么干，实在有点，呃，令人生厌。

std::move 的诱人之处在于：一是方便，二是减少错误可能，三是更加清晰。考虑一个类，我们打算用它进行移动构造函数的调用次数的跟踪。所需要的全部设施，就是一个随着移动构造操作而递增的静态计数器罢了。假设类中的唯一非静态数据成员是个 std::string 型别的对象，以下是个实现移动构造函数的经典途径（即利用 std::move）：

```
class Widget {
public:
  Widget(Widget&& rhs)
  : s(std::move(rhs.s))
  { ++moveCtorCalls; }

  ...
private:
  static std::size_t moveCtorCalls;
  std::string s;
};
```

如果采用 std::forward 来实现相同的行为，代码看起来会长成这样：

```
class Widget {
public:
  Widget(Widget&& rhs)                          // 离经叛道
  :  s(std::forward<std::string>(rhs.s))        // 结果错误
  { ++moveCtorCalls; }                          // 的实现
```

```
    ...
};
```

首先要注意，std::move 只取用一个函数实参（rhs.s），而 std::forward 则既须取用一个函数实参（rhs.s），又须取用一个模板型别实参（std::string）；其次要注意，传递给 std::forward 的实参型别应当是个非引用型别，因为习惯上它编码的所传递实参应该是个右值（参见条款 28）。这两项合起来，就意味着一是 std::move 在调用时比 std::forward 需要打的字更少；再是省去了需要传递一个型别实参且由它编码的须是个右值型别的麻烦。同时，采用 std::move 也消除了传递了错误型别的可能性（例如 std::string&，这会导致数据成员 s 被复制构造而非移动构造）。

更为重要的是，使用 std::move 所要传达的意思是无条件地向右值型别的强制型别转换，而使用 std::forward 则想说明仅仅对绑定到右值的引用实施向右值型别的强制型别转换。这是两个非常不同的行为。前者是典型地为移动操作做铺垫，而后者仅仅是传递（转发）一个对象到另一个函数，而在此过程中无论该对象原始型别具备左值性（lvalueness）和右值性（rvalueness），都保持原样。这两个行为是如此不同，因而最好使用两个不同函数（以及函数名字）来区分这两者。

要点速记

- std::move 实施的是无条件的向右值型别的强制型别转换。就其本身而言，它不会执行移动操作。
- 仅当传入的实参被绑定到右值时，std::forward 才针对该实参实施向右值型别的强制型别转换。
- 在运行期，std::move 和 std::forward 都不会做任何操作。

条款 24：区分万能引用和右值引用

谚云，真理给人自由。但若有机缘巧合，精心挑选的谎言也同样使人解脱。本条款就是这么一篇谎言。然而由于我们经手的是软件，还是讳去“谎言”一词，而说成是本条款包含了一个“抽象”好了。

欲声明指涉到某型别 T 的右值引用，就写作 T&&。于是理所当然地，当你在代码中看到“T&&”时，会把它当作一个右值引用。哎呀，其实并没有这么简单：

```
void f(Widget&& param);                    // 右值引用

Widget&& var1 = Widget();                  // 右值引用

auto&& var2 = var1;                        // 非右值引用

template<typename T>
void f(std::vector<T>&& param);            // 右值引用

template<typename T>
void f(T&& param);                         // 非右值引用
```

实际上，“T&&”有两种不同的含义。其中一种含义，理所当然，是右值引用。正如期望，它们仅仅会绑定到右值，而其主要的存在理由（raison d'être），在于识别出可移对象。

“T&&”的另一种含义，则表示其既可以是右值引用，亦可以是左值引用，二者居一。带有这种含义的引用在代码中形如右值引用（即 T&&），但它们可以像左值引用一样运作（即 T&）。这种双重特性使之既可以绑定到右值（如右值引用），也可以绑定到左值（如左值引用）。犹有进者，它们也可以绑定到 const 对象或非 const 对象，以及 volatile 对象或非 volatile 对象，甚至绑定到那些既带有 const 又带有 volatile 饰词的对象。它们几乎可以绑定到万事万物。这种拥有史无前例的灵活性的引用值得拥有一个独特的名字。我称之为万能引用（universal reference）。[注1]

万能引用会在两种场景下现身。最常见的一种场景是函数模板的形参，比如这个来自上方示例代码的例子：

```
template<typename T>
void f(T&& param);                         // param 是个万能引用
```

第二个场景是 auto 声明，包括这个来自上方示例代码的例子：

```
auto&& var2 = var1;                        // var2 是个万能引用
```

这两个场景的共同之处，在于它们都涉及型别推导。在模板 f 中，param 的型别是推导得到的；而在 var2 的声明语句中，var2 的型别也是推导得到的。相比之下，下面这些例子则不涉及型别推导（同样来自上面的示例代码）。如果你看到了“T&&”，却没有其涉及的型别推导，那么，你看到的就是个右值引用：

```
void f(Widget&& param);        // 不涉及型别推导；
                               // param 是个右值引用
```

注 1：条款 25 会解释万能引用几乎总是需要应用 std::forward，在本书即将出版时，C++ 委员会的一些成员开始用转发引用（forwarding references）来称呼万能引用。

```
Widget&& var1 = Widget();      // 不涉及型别推导；
                               // var1 是个右值引用
```

因为万能引用首先是个引用，所以初始化是必需的。万能引用的初始化物会决定它代表的是个左值还是右值引用：如果初始化物是右值，万能引用就会对应到一个右值引用；如果初始化物是左值，万能引用就会对应到一个左值引用。对于作为函数形参的万能引用而言，初始化物在调用处提供：

```
template<typename T>
void f(T&& param);             // param 是个万能引用

Widget w;
f(w);                          // 左值被传递给 f；
                               // param 的型别是 Widget&（即一个左值引用）

f(std::move(w));               // 右值被传递给 f；
                               // param 的型别是 Widget&&（即一个右值引用）
```

若要使一个引用成为万能引用，其涉及型别推导是必要条件，但还不是充分条件。引用声明的形式也必须正确无误，并且该形式被限定得很死：必须得正好形如“T&&”才行。再来温习一下前述示例代码：

```
template<typename T>
void f(std::vector<T>&& param); // param 是个右值引用
```

当 f 被调用时，型别 T 将被推导（除非调用者显式指明了型别，这是一种我们不必操心的边界情况），但 param 的型别声明的形式不是“T&&”，而是“std::vector<T>&&”，这就排除了 param 是个万能引用的可能性。因此，param 是个右值引用，这个事实在你试图传递一个左值给 f 时编译器会很乐意为你确认出来：

```
std::vector<int> v;
f(v);                          // 错误！不能给一个右值引用绑定一个左值
```

即使是一个 const 饰词的存在，也足以褫夺一个引用成为万能引用的资格：

```
template<typename T>
void f(const T&& param);       // param 是个右值引用
```

如果你在一个模板内看到一个函数形参的型别写作“T&&”，你可能会想当然地认为它肯定是个万能引用。不能想当然。这是因为，“位于模板内”并不能保证“一定涉及型别推导”。考虑下面这个在 std::vector 内的 push_back 成员函数：

```
template<class T, class Allocator = allocator<T>>      // 来自
class vector {                                  // C++ 标准
public:
```

```
  void push_back(T&& x);
  ...
};
```

push_back 的形参当然具备万能引用的正确形式，但在本例中，并不涉及型别推导。因为 push_back 作为 vector 的一部分，如果不存在特定的 vector 实例，则它也无从存在。该实例的具现型别完全决定了 push_back 的声明型别。即，给定：

```
std::vector<Widget> v;
```

会导致 std::vector 模板具现化为如下实例：

```
class vector<Widget, allocator<Widget>> {
public:
  void push_back(Widget&& x);                    // 右值引用
  ...
};
```

现在你可以看得非常清楚了：push_back 未涉及任何型别推导。这个 vector<T> 的 push_back 函数（实际上是两个函数，该函数是有重载版本的）的形参声明都是指涉到 T 型别的右值引用型别。

作为对比，同样是在 std::vector 内，和 push_back 概念上类似的成员函数 emplace_back 却实实在在地涉及型别推导：

```
template<class T, class Allocator = allocator<T>>        // 仍然来自
class vector {                                           // C++ 标准
public:
  template <class... Args>
  void emplace_back(Args&&... args);
  ...
};
```

其中的型别形参 Args 独立于 vector 的型别形参 T，所以，Args 必须在每次 emplace_back 被调用时进行推导（好吧，实际上 Args 是个形参包，而不是单个的型别形参，但是它符合我们讨论的目的，仍然可以把它当作单个的型别形参处理）。

emplace_back 的型别形参的名字是 Args，但它仍然是一个万能引用，这再次印证了我之前所说的话，即万能引用的形式必须是“T&&”，但没必要一定要取“T”这个名字。例如，下面的模板取用的是个万能引用，因为其形式（*type&&*）正确无误，并且 param 的型别乃是推导而来（再次强调，调用者显式指定型别的边界情况要予以排除）：

```
template<typename MyTemplateType>                // param 是个
void someFunc(MyTemplateType&& param);           // 万能引用
```

之前我提到过，auto 变量也可以作为万能引用。更准确地说，声明为 auto&& 型别的变量都是万能引用，因为它们肯定涉及型别推导并且肯定有正确的形式（“T&&”）。auto 万能引用不像用在函数模板形参的万能引用时那样普遍，但也会时不时在 C++11 中现身。在 C++14 中，它们现身的机会就更多得多了。这是因为 C++14 的 lambda 表达式中可以声明 auto&& 形参。例如，如果要撰写一个 C++14 的 lambda 表达式来记录任意函数调用所花费的时长，就可以像下面这样：

```
auto timeFuncInvocation =
  [](auto&& func, auto&&... params)            // C++14
  {

  计时器启动；
  std::forward<decltype(func)>(func)(          // 调用 func
    std::forward<decltype(params)>(params)..   // 取用 params
    );
  计时器停止并记录流逝的时间；
};
```

假如你看过代码“std::forward<decltype(blah blah blah)>”以后的反应是“这是啥玩意儿？”则很可能你尚未读过条款 33。不必担心，本条款的重点在于该 lambda 表达式所声明的 auto&& 型别的形参。func 是一个可以被绑定到任何可调用对象的万能引用，可以是左值或右值。params 是 0 个或更多的万能引用（亦即一个万能引用形参包），可以被绑定任何数量的任意型别的对象。多亏有了 auto 万能引用，才取得了“timeFuncInvocation 可以计算出大多数任何函数的执行时间”这样的成果（欲知“任何”和“大多数任何”的区别何在，参见条款 30）。

请牢记，本条款（万能引用的基础知识）是一篇谎言……哦不对，是一个“抽象”。而底层的真相，则被称为引用折叠（reference collapse），条款 28 会专题阐述这个主题。但了解真相以后，抽象并没有因此减少其有用价值。区分右值引用和万能引用有助于使你能够更精确地读懂代码（“我正在看的这个 T&& 是只会绑定到右值，还是可以绑定到任何东西？”）；它也可以避免你和同事的沟通中产生歧义（“我在这里使用的是万能引用，而非右值引用……”）。弄明白本条款可以让你更好地理解条款 25 和条款 26，这两个条款都是依赖于右值引用和万能引用的区别。所以，拥抱抽象并乐在其中吧。正如牛顿运动定律（其实是不正确的）比爱因斯坦的广义相对论（“真相”）往往同样有用，而且还更容易运用一样，掌握万能引用的概念通常也比完全了解引用折叠的技术细节是个更好的选项。

要点速记

- 如果函数模板形参具备 T&& 型别，并且 T 的型别系推导而来，或如果对象使用 auto&& 声明其型别，则该形参或对象就是个万能引用。
- 如果型别声明并不精确地具备 *type&&* 的形式，或者型别推导并未发生，则 *type&&* 就代表右值引用。
- 若采用右值来初始化万能引用，就会得到一个右值引用。若采用左值来初始化万能引用，就会得到一个左值引用。

条款 25：针对右值引用实施 std::move，针对万能引用实施 std::forward

右值引用仅会绑定到那些可供移动的对象上。如果形参型别为右值引用，则应该清楚地了解，它绑定的对象可供移动：

```
class Widget {
  Widget(Widget&& rhs);                    // rhs 确定无疑地会绑定到
  …                                        // 可以合于移动目的的对象
};
```

既然如此，你可能会希望把这些对象传递给其他函数时，使用一种允许该函数利用该对象的右值性的方式。手法就是，把绑定到了这些对象的形参转换为右值。正如条款 23 介绍的那样，这不仅仅是 std::move 能胜任的工作，这更是它被发明出来的动机：

```
class Widget {
public:
  Widget(Widget&& rhs)                     // rhs 是个右值引用
  : name(std::move(rhs.name)),
    p(std::move(rhs.p))
    { … }
  …

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

而万能引用，则有所不同（参见条款 24），它只是可能会绑定到可供移动的对象上。万能引用只有在使用右值初始化来初始化时才会强制转换成右值型别。条款 23 解释过，这正好就是 std::forward 的所作所为：

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)                    // newName 是个
  { name = std::forward<T>(newName); }         // 万能引用

  ...
};
```

简而言之，当转发右值引用给其他函数时，应当对其实施向右值的无条件强制型别转换（通过 std::move），因为它们一定绑定到右值；而当转发万能引用时，应当对其实施向右值的有条件强制型别转换（通过 std::forward），因为它们不一定绑定到右值。

条款 23 阐明了，针对右值引用实施 std::forward，也能硬弄出正确行为来，但是代码啰嗦、易错，且不符合习惯用法，所以应当避免针对右值引用实施 std::forward。而另一方面，针对万能引用使用 std::move 的想法更为糟糕，因为那样做的后果是某些左值会遭到意外改动（例如，某些局部变量）：

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)                    // 万能引用
  { name = std::move(newName); }               // 可以编译，但是
  ...                                          // 糟糕透顶！

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName();                   // 工厂函数

Widget w;

auto n = getWidgetName();                      // n 是个局部变量

w.setName(n);                                  // 将 n 移入了 w！

...                                            // n 的值变得未知
```

这里，局部变量 n 被传递给 w.setName，而调用者会合情合理地假定这是一个对 n 的只读操作。但由于 setName 函数内部使用了 std::move 把它的引用形参无条件地强制转换到右值，n 的值就会被移入 w.name。这么一来，调用完 setName 函数返回时，n 将变成一个不确定的值。这样的行为会让调用者绝望，甚至抓狂。

你可能会辩解道，setName 不应该将其形参声明成一个万能引用。这样的引用不能带有 const 饰词（参见条款 24），当然 setName 也不应该改动形参的值。你可能会指出，只要 setName 为常量左值和右值实现出不同的重载，整个问题就能得到解决。正如这样：

```
class Widget {
public:
  void setName(const std::string& newName)        // 从常量左值
  { name = newName; }                              // 取得赋值

  void setName(std::string&& newName)              // 从右值
  { name = std::move(newName); }                   // 取得赋值

  ...
};
```

这样的手法当然可以让本例得以正常运行，但是也带来了其他缺点。第一，需要编写和维护更多源代码（两个函数，而不是一个简单的模板）；第二，效率要打折扣。例如考虑下面这种 setName 的用法：

```
w.setName("Adela Novak");
```

在使用万能引用做形参的 setName 的版本里，字符串的字面值“Adela Novak”会被传递给 setName，然后再转手传递给 w 内部 std::string 的赋值运算符。这么一来，w 的数据成员 name 可以直接从字符串字面值得到赋值，而不会产生 std::string 型别的临时对象。然而，重载版本的 setName 就得创建 std::string 型别的临时对象以供其形参绑定，随后该临时对象才会移入 w 的数据成员。因此，一个对（重载版本的）setName 的调用会生成下面的执行序列：一次 std::string 的构造函数（以创建临时对象），一次 std::string 的移动赋值运算符（以移动 newName 到 w.name），还有一次 std::string 的析构函数（以销毁临时变量）。这个执行序列几乎肯定比仅仅调用一次 std::string 赋值运算符（通过使用 const char* 指针）要代价高昂。额外的开销和不同的实现有关，这种开销是否值得担忧也取决于不同的应用程序和库。但是事实就是，若用一对通过左值和右值引用的重载函数来替换使用万能引用形参的函数模板，很可能在某些情况下引发运行期效率问题。如果我们将讨论推广到一般情况，让这个例子中的 Widget 的数据成员声明成任意型别，而不是已知的 std::string 的话，则性能差距可能会显著增大，因为不是所有型别在执行移动时都像 std::string 一样成本低廉（参见条款 29）。

然而，依左值和右值的重载的最严重问题并不是代码膨胀或者对习惯用法的背离，甚至也不是运行期的效率折损，而是这种设计的可扩展性太差。Widget::setName 只有一个形参，因此两个重载就够了，但是对于那些有多个形参的函数，每个形参都需要一个左值和一个右值，从而重载函数的个数会呈几何级数增长：有 n 个形参，就需要 2^n 个重载函数。更糟糕的是有些函数（实际上是函数模板）会有无穷多个形参，而每个形参都可能是左值或是右值。这种函数的榜样就是 std::make_shared，如果谈到

C++14，那就还有 std::make_unique（参见条款 21）。看看它们最常用的重载版本的声明部分：

```
template<class T, class... Args>                    // 选自 C++11
shared_ptr<T> make_shared(Args&&... args);          // 标准

template<class T, class... Args>                    // 选自 C++14
unique_ptr<T> make_unique(Args&&... args);          // 标准
```

对于这样的函数，针对左值和右值进行重载并不可行：万能引用才是唯一的解决之道。可以保证的是，在这些函数内部，当万能引用的形参被传递给其他函数之时，会针对它们实施 std::forward。所以，这才是你应该做的。

好吧，只能说通常这是你应该做的。而且，你最终是会这么做的。但一开始这并不一定就是你想做的。有些情况下，你会想要在单一函数内将某个对象不止一次地绑定到右值引用或者万能引用，而且你想保证完成对该对象的其他所有操作之前，其值不被移走。在这种情况下，你就得仅在最后一次使用该引用时，对其实施 std::move（右值引用）或 std::forward（万能引用）。例如：

```
template<typename T>                                // text 是个
void setSignText(T&& text)                          // 万能引用
{
  sign.setText(text);                               // 使用 text，
                                                    // 但不改动其值

  auto now =                                        // 取得当前时间
    std::chrono::system_clock::now();

  signHistory.add(now,
                  std::forward<T>(text));           // 有条件地
}                                                   // 将 text 强制转换成右值型别
```

在这里，我们想要保证 text 的值不会被 sign.setText 修改，因为还要在调用 signHistory.add 时使用该值。因此，std::forward 仅仅实施在该万能引用最后一次使用的场合。

对 std::move，相同的想法也适用（亦即，在最后一次使用右值引用处实施 std::move）。但是应当注意到在很少数的情况下，你需要调用 std::move_if_noexcept 来代替 std::move。欲知何时以及为何这样做，请参阅条款 14。

在按值返回的函数中，如果返回的是绑定到一个右值引用或一个万能引用的对象，则当你返回该引用时，应该对其实施 std::move 或者 std::forward。为了明白为什么要这样做，考虑一个 operator+ 函数，用以实现矩阵加法。位于运算符左侧的矩阵已知是个右值（因此它的存储空间可以复用，以保存矩阵的和）：

```
Matrix                                          // 按值返回
operator+(Matrix&& lhs, const Matrix& rhs)
{
  lhs += rhs;
  return std::move(lhs);                        // 将 lhs 移入
}                                               // 返回值
```

通过在返回语句中把 lhs 强制转换成右值型别（经由 std::move），lhs 会被移入函数的返回值存储位置。如果把 std::move 的调用省去：

```
Matrix                                          // 同上
operator+(Matrix&& lhs, const Matrix& rhs)
{
  lhs += rhs;
  return lhs;                                   // 将 lhs 复制入
}                                               // 返回值
```

lhs 是个左值这一事实会强迫编译器将其复制入返回值存储位置。假定 Matrix 型别支持移动构造，这将比复制构造效率更高，从而在返回语句中使用 std::move 会产生更高效的代码。

假如 Matrix 不支持移动，将其转换为右值也并无大碍，因为右值也就是会通过 Matrix 的复制构造函数来完成复制而已（参见条款 23）。但若 Matrix 后来被修改为支持移动操作，则下次编译完成后，operator+ 就能自动获益。这个例子中，对函数按值返回的右值引用实施 std::move，不会付出任何代价（可能还会有不小收益）。

对于万能引用和 std::forward 来说，情况类似。考虑函数模板 reduceAndCopy，它接收一个可能未约分的 Fraction 对象，实施约分，然后返回一个约分后值的副本。如果原始对象是一个右值，它的值应当被移动到返回值上（这样可以避免制造副本的成本）。但如果原始对象是一个左值，那就必须创建出实实在在的副本来。因此：

```
template<typename T>
Fraction                                // 按值返回
reduceAndCopy(T&& frac)                 // 万能引用形参
{
  frac.reduce();
  return std::forward<T>(frac);         // 对于右值，是移入返回值
}                                       // 对于左值，是复制入返回值
```

如果省去对 std::forward 的调用，则 frac 会无条件地复制入到 reduceAndCopy 的返回值。

一些程序员根据上面的信息，并试图把它扩展到不适用的情况下。既然“假如针对被复制到返回值上的右值引用形参实施 std::move 会有机会把复制构造转换成移动构

造”，他们就推论到：“相同优化也可以用于欲返回的局部变量上”。换句话说，他们认为给定一个按值返回局部变量的函数，类似这样：

```
Widget makeWidget()                    // makeWidget 的“复制”版本
{
  Widget w;                            // 局部变量

  ...                                  // 操作 w

  return w;                            // 将 w “复制”入返回值
}
```

从而可以通过将“复制”转换为移动来“优化”：

```
Widget makeWidget()                    // makeWidget 的移动版本
{
  Widget w;
  ...
  return std::move(w);                 // 将 w 移入返回值
}                                      // （别这样做！）
```

我有意使用了引号，来说明这里的推理是错误的。但错在哪里呢？

根本原因在于，标准化委员已经领先于上例中的那些程序员很多年地在着手解决这里的优化问题。人们很久之前就认识到，makeWidget 的“复制”版本可以通过直接在为函数返回值分配的内存上创建局部变量 w 来避免复制之，这就是我们熟知的返回值优化（return value optimization，RVO），这是 C++ 标准一问世就有的、白纸黑字写下来的福音。

这句福音的措词十分严谨，这是因为仅当其不会影响软件的可观测行为时，你才会想要允许这种复制省略发生。细读这娓娓道来的（也有人说简直是有毒的）标准字句，就知道福音的内容具体是说：编译器若要在一个按值返回的函数里省略对局部对象[注2]的复制（或者移动），则需要满足两个前提条件：①局部对象型别和函数返回值型别相同；②返回的就是局部对象本身。据此，我们再重新审视下 makeWidget 的“复制”版本：

```
Widget makeWidget()                    // makeWidget 的“复制”版本
{
  Widget w;
```

注 2：满足条件的局部对象包括大多数局部变量（例如 makeWidget 中的 w）和作为返回语句的一部分被创建的临时对象。函数参数并不满足条件。有人对 RVO 实施在局部对象时，根据局部对象具名或不具名（亦即临时）的特性进行了区分，限制了 RVO 对不具名对象的使用，并将其对具名对象实施者，特别地称为具名返回值优化（named return value optimization，NRVO）。

```
  ...
  return w;                              // 将w“复制”入返回值
}
```

这里两个前提条件都满足了。要相信，对于这段代码，任何合格的C++编译器都会应用RVO来避免复制w。这也意味着，“复制”版本的makeWidget实际上并未复制任何东西。

移动版本的makeWidget做了名副其实的事情（假设Widget提供了移动构造函数）：它移动了w的内容到了makeWidget函数的返回值空间。但为什么编译器没有用RVO来消除移动，而是在函数返回值分配的内存里重建了一个w呢？答案很简单：它们做不到。前提条件2规定了RVO只能在返回值是一个局部对象时执行，然而移动版本的makeWidget里并非如此。重新审视一下移动版本的返回语句：

```
return std::move(w);
```

这里返回的不是局部对象w，而是w的引用，std::move(w)的结果。返回一个局部对象的引用并不满足实施RVO的前提条件，因此编译器必须把w移入函数的返回值存储位置。开发者本来企图通过对即将返回的局部变量实施std::move来帮助编译器进行优化，然而实际上却适得其反地限制了本来可用的编译优化选项！

但是RVO是种优化。编译器并无义务省略复制和移动操作，即便这是被允许的。也许你比较偏执，担心编译器会通过复制操作来惩罚自己，就因为它们这样做是合法的。或者也许你有足够洞见分辨出何种情况对于编译器来说执行RVO有困难，例如，当一个函数中不同的控制路径返回不同的局部变量时。（编译器不得不产生代码，以在分配给函数返回值存储位置中来构造合适的局部变量，但编译器如何确定哪个局部变量才合适呢？）这样的话，你可能认为付出移动的代价是一种为了避免复制而缴纳的保险。亦即，你也许仍然认为，针对欲返回的局部对象实施std::move是合理的。仅仅是因为这么一来你就可以放宽心，因为绝不会为复制付出成本。

即使在那种情况下，针对局部对象实施std::move仍然是个馊主意。因为标准中关于RVO的那条福音后面又接着说明，即使实施RVO的前提条件满足，但编译器选择不执行复制省略的时候，返回对象必须作为右值处理。这么一来，就等于标准要求：当RVO的前提条件允许时，要么发生复制省略，要么std::move隐式地被实施于返回的局部对象上。因此，在makeWidget的“复制”版本中：

```
Widget makeWidget()                      // 同上例
{
  Widget w;
  ...
```

```
    return w;
}
```

编译器必须要么省略 w 的复制操作，要么让函数进行特别处理，以与下面这样的代码等价：

```
Widget makeWidget()
{
  Widget w;
  ...
  return std::move(w);          // w 会作为右值处理，
}                               // 原因是复制省略没有实施
```

上述情况与按值传递的函数形参类似。它们作为函数返回值时，不适合实施复制省略，但编译器必须在其返回时作为右值处理。以结果论，如果你的代码看起来是这样的：

```
Widget makeWidget(Widget w)     // 按值传递的形参，
{                               // 与函数返回值型别相同
  ...
  return w;
}
```

在编译器必须处理上面这段代码，以使它们与以下代码等价：

```
Widget makeWidget(Widget w)
{
  ...
  return std::move(w);          // 将 w 作为右值处理
}
```

这意味着，针对函数中按值返回的局部对象实施 std::move 的操作，不能给编译器帮上忙（如果不执行复制省略，就必须将局部对象作为右值处理，效果一样），却可以帮倒忙（可能会排除掉 RVO 的实施机会）。的确存在适合于针对局部变量实施 std::move 的情况（亦即，将其传递给某个函数，并且你确定自己不再会使用该变量），但是其作为 return 语句的一部分时，要么适合 RVO，要么返回一个按值形参，从而并不属于以上的那些情况。

要点速记

- 针对右值引用的最后一次使用实施 std::move，针对万能引用的最后一次使用实施 std::forward。
- 作为按值返回的函数的右值引用和万能引用，依上一条所述采取相同行为。
- 若局部对象可能适用于返回值优化，则请勿针对其实施 std::move 或 std::forward。

条款 26：避免依万能引用型别进行重载

假定你需要撰写一个函数，取用一个名字作为形参，然后记录下当前日期和时间，再把该名字添加到一个全局数据结构中。可能你一开始拿出来的函数长得有点像下面这样：

```
std::multiset<std::string> names;       // 全局数据结构

void logAndAdd(const std::string& name)
{
  auto now =                            // 取得当前时间
    std::chrono::system_clock::now();

  log(now, "logAndAdd");                // 制备日志条目

  names.emplace(name);                  // 将名字添加到全局数据结构中，
}                                       // 关于 emplace 的更多信息参见条款 42
```

这段代码无可厚非，只是效率方面未能尽如人意。考虑以下三种可能的调用语句：

```
std::string petName("Darla");

logAndAdd(petName);                             // 传递左值 std::string

logAndAdd(std::string("Persephone"));           // 传递右值 std::string

logAndAdd("Patty Dog");                         // 传递字符串字面量
```

在第一个调用语句中，logAndAdd 的形参 name 绑定到了变量 petName。在 logAndAdd 内部，name 最终被传递给了 names.emplace。由于 name 是个左值，它是被复制入 names 的。没有任何办法避免这个复制操作，因为传递给 logAndAdd 的是个左值（petName）。

在第二个调用语句中，形参 name 绑定到了一个右值（从“Persephone”显式构造的 std::string 型别的临时对象）。name 自身是个左值，所以它是被复制入 names 的。但我们能够认识到，原则上该值是可以被移动入 names 的。所以在这个调用中，我们付出了一次复制的成本，但我们可以用一次移动来达成同样的目标。

在第三个调用语句中，形参 name 还是绑定到了一个右值，但这次这个 std::string 型别的临时对象是从“Patty Dog”隐式构造的。和第二个调用语句的情况一样，name 是被复制入 names 的，但在本语句中，传递给 logAndAdd 的实参是个字符串字面量。如果该字符串字面量被直接传递给 emplace，那就完全没有必要构造一个 std::string 型别的临时对象。emplace 完全可以利用这个字符串字面量在 std::multiset 内部直接

构造出一个 std::string 对象。在这第三个调用中，我们付出了复制一个 std::string 对象的成本，但实际上我们连一次移动的成本都没有必要付出，更别说复制了。

我们可以解决第二个和第三个调用语句的效率低下问题，只需重写 logAndAdd，让它接受一个万能引用（参见条款 24），并且根据条款 25，对该引用实施 std::forward 给到 emplace。重写结果不言自明：

```
template<typename T>
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}

std::string petName("Darla");                    // 一如此前

logAndAdd(petName);                              // 一如此前，将左值复制入 multiset

logAndAdd(std::string("Persephone"));            // 对右值实施移动而非复制

logAndAdd("Patty Dog");                          // 在 multiset 中直接构造一个
                                                 // std::string 对象，而非复制一个
                                                 // std::string 型别的临时对象
```

非常完美，效率达到极致了！

如果故事讲到这里就结束了，那我们就可以就此罢手荣誉收工了。但我并没有告诉过你，该函数的客户并不总能直接访问到 logAndAdd 所要求的名字。有些客户只能访问到一个索引，logAndAdd 需要根据该索引来查询一张表才能找到对应的名字。为了支持这样的客户，logAndAdd 提供了重载版本：

```
std::string nameFromIdx(int idx);                // 返回索引对应的名字

void logAndAdd(int idx)                          // 新的重载函数
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(nameFromIdx(idx));
}
```

调用时的重载决议符合期望：

```
std::string petName("Darla");            // 一如此前

logAndAdd(petName);                      // 一如此前，这三个调用语句
logAndAdd(std::string("Persephone"));    // 都调用了形参型别
logAndAdd("Patty Dog");                  // 为 T&& 的重载版本

logAndAdd(22);                           // 本句调用了形参型别为 int 的重载版本
```

实际上，说重载决议符合期望，只是在期望不能过高的前提之下。假设某个客户使用了 short 型别的变量来持有这个索引值，并将该变量传递给了 logAndAdd：

```
short nameIdx;
…                                   // 赋值给 nameIdx

logAndAdd(nameIdx);                 // 错误!
```

上述代码最后一行的注释并不能很好地说明问题，所以请容我解释这里到底发生了什么情况。

logAndAdd 有两个重载版本。形参型别为 T&& 的版本可以将 T 推导为 short，从而产生一个精确匹配。而形参型别为 int 的版本却只能在型别提升以后才能匹配到 short 型别的实参。按照普适的重载决议规则，精确匹配优先于提升后才能匹配。所以，形参型别为万能引用的版本才是被调用到的版本。

调用执行后，形参 name 被绑定到传入的 short 型别的变量上。然后，name 被 std::forward 传递给 names（一个 std::multiset<std::string> 型别的对象）的 emplace 成员函数，再然后，又被例行公事地被转发给 std::string 的构造函数。而 std::string 的构造函数中并没有形参为 short 的版本，所以，由 logAndAdd 的调用触发了 multiset::emplace 的调用，后者又触发了 std::string 的构造函数的调用，到了这一步失败了。这一切的原因归根结底在于，对于 short 型别的实参来说，万能引用产生了比 int 更好的匹配。

形参为万能引用的函数，是 C++ 中最贪婪的。它们会在具现过程中，和几乎任何实参型别都会产生精确匹配（条款 30 描述了几种不属于该情况的实参）。这就是为何把重载和万能引用这两者结合起来几乎总是馊主意：一旦万能引用成为重载候选，它就会吸引走大批的实参型别，远比撰写重载代码的程序员期望的要多。

填上这个坑的一个简单办法，是撰写一个带完美转发的构造函数。对 logAndAdd 这个示例作了一点点修改，就暴露了问题。我们先不去撰写一个自由函数来同时取用 std::string 或一个用以查表返回 std::string 的索引，而是先考虑一个 Person 类，它的构造函数有相同的功能：

```
class Person {
public:
  template<typename T>
  explicit Person(T&& n)                // 完美转发构造函数，
  : name(std::forward<T>(n)) {}         // 初始化了数据成员

  explicit Person(int idx)              // 形参为 int 的构造函数
  : name(nameFromIdx(idx)) {}
  …
```

```
private:
  std::string name;
};
```

在logAndAdd的情景中，传入一个非int型别的整型（例如std::size_t、short、long等）都会导致调用形参为万能引用的构造函数重载版本，从而引发编译失败。但是上例中的情景则要糟糕得多，因为在Person中还有比我们肉眼所见更多的重载版本。条款17解释了，在适当条件下，C++会同时生成复制和移动构造函数，并且这一点在即使类中包含着一个模板化的构造函数，且它可以具现出复制和移动构造函数的签名来的前提下也依然成立。假如Person中真的如此生成了复制和移动构造函数，那么它实际上是长成这样的：

```
class Person {
public:
  template<typename T>                    // 完美转发构造函数
  explicit Person(T&& n)
  : name(std::forward<T>(n)) {}
  explicit Person(int idx);               // 形参为int的构造函数

  Person(const Person& rhs);              // 复制构造函数（由编译器生成）

  Person(Person&& rhs);                   // 移动构造函数（由编译器生成）
  ...

};
```

只有花费了大量时间与编译器和写编译器的人打交道，才能形成对于程序行为的直觉，并忘记普通人的思维方式：

```
Person p("Nancy");

auto cloneOfP(p);                  // 从p出发创建新的Person型别对象；
                                   // 上述代码无法通过编译！
```

在这里我们尝试从一个Person出发创建另一个Person，看起来再明显不过会是调用复制构造的情况（p是个左值，这就足以打消一切会将“复制”通过移动来完成的想法）。但这段代码竟没有调用复制构造函数，而是调用了完美转发构造函数。该函数是在尝试从一个Person型别的对象（p）出发来初始化另一个Person型别的对象的std::string型别的数据成员。而std::string型别却并不具备接受Person型别形参的构造函数，你的编译器只能悲愤地举手投降，也许会丢出一堆冗长且无法理解错误信息作为惩罚和发泄。

你可能感觉莫名其妙，“这是怎么回事？怎么会调用的是完美转发构造函数而不是复制构造函数呢？这不是明明在用一个Person型别的对象初始化另一个Person型别的

对象吗！”确实是在做这件事，但是编译器是宣誓效忠于 C++ 规则的，而在这里用到的规则关乎调用重载函数时的决议。

编译器的推理过程如下：cloneOfP 被非常量左值（p）初始化，那意味着模板构造函数可以实例化来接受 Person 型别的非常量左值形参。如此实例化后，Person 的代码应该变换成下面这样：

```
class Person {
public:
  explicit Person(Person& n)                    // 从完美转发模板出发
  : name(std::forward<Person&>(n)) {}           // 实例化

  explicit Person(int idx);                     // 如前
  Person(const Person& rhs);                    // 复制构造函数（由编译器生成）
  ...

};
```

在下述语句中，

```
auto cloneOfP(p);
```

p 既可以传递给复制构造函数，也可以传递给实例化了的模板。但是，调用复制构造的话就要先对 p 添加 const 饰词才能匹配复制构造函数的形参型别，而调用实例化了的模板却不要求添加任何饰词。因而，模板生成的重载版本是更佳匹配，所以编译器的做法完全符合设计：它调用了符合更佳匹配原则的函数。这么一来，“复制”一个非常量的左值 Person 型别对象，会由完美转发构造函数而不是复制构造函数来完成。

如果我们稍微修改一下代码，使得所复制之物成为一个常量对象，反响就完全不同了：

```
const Person cp("Nancy");       // 对象成为常量了

auto cloneOfP(cp);              // 这回调用的是复制构造函数了！
```

因为欲复制的对象是个常量，就形成了对复制构造函数形参的精确匹配。另一方面，那个模板化的构造函数可以经由实例化得到同样的签名，

```
class Person {
public:
  explicit Person(const Person& n);     // 从模板出发实例化
                                        // 而得到的构造函数

  Person(const Person& rhs);            // 复制构造函数（由编译器生成）
  ...
};
```

但这不要紧，因为 C++ 重载决议规则中有这么一条：若在函数调用时，一个模板实例化函数和一个非函数模板（亦即，一个“常规”函数）具备相等的匹配程度，则优先

选用常规函数。根据这一条，在签名相同时，复制构造函数（它是个普通函数）就会压过实例化了的函数模板。

（如果你想知道，为什么明明实例化了的模板构造函数已经生成了和复制构造函数一模一样的签名，编译器还会生成复制构造函数，请参考条款 17。）

完美转发构造函数与编译器生成的复制和移动操作之间的那些错综复杂的关系，再加上继承以后就变得更让人眉头紧锁。特别的，派生类的复制和移动操作的平凡实现会表现出让人大跌眼镜的行为。请看好：

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs)      // 复制构造函数；
  : Person(rhs)                                // 调用的是
  { … }                                        // 基类的完美转发构造函数！

  SpecialPerson(SpecialPerson&& rhs)           // 移动构造函数；
  : Person(std::move(rhs))                     // 调用的是
  { … }                                        // 基类的完美转发构造函数！
};
```

注释说得很明白，派生类的复制和移动构造函数并未调用到基类的复制和移动构造函数，调用到的是基类的完美转发构造函数！要理解背后的原因，请注意，派生类函数把型别为 SpecialPerson 的实参传递给了基类，然后在 Person 类的构造函数中完成模板实例化和重载决议。最终，代码无法通过编译，因为 std::string 的构造函数中没有任何一个会接受 SpecialPerson 型别的形参。

我希望我现在已经说服了你去尽可能避免以把万能引用型别作为重载函数的形参选项。不过，如果使用万能引用进行重载是个糟糕的思路，而你又需要针对绝大多数的实参型别实施转发，只针对某些实参型别实施特殊处理，这时该怎么做呢？解决之道多种多样，由于实在方法太多了，所以我决定开辟一整个条款来讲述。这就是条款 27，即下一条款。请继续读下去，就一定能获得你想要的答案。

要点速记

- 把万能引用作为重载候选型别，几乎总会让该重载版本在始料未及的情况下被调用到。
- 完美转发构造函数的问题尤其严重，因为对于非常量的左值型别而言，它们一般都会形成相对于复制构造函数的更佳匹配，并且它们还会劫持派生类中对基类的复制和移动构造函数的调用。

条款 27：熟悉依万能引用型别进行重载的替代方案

条款 26 说过，依万能引用型别进行重载会导致形形色色的问题，独立函数和成员函数（构造函数尤其问题严重）。不过，该条款也展示多少了这种重载可能有用的若干用例。这些用例的行为要是能按我们想要的那样运作就好了！本条款进行了一番探索，目的就在于获得期望的行为，方法是或者通过依万能引用型别进行重载的设计手法，或者通过限制能匹配的实参型别。

接下来讨论是以条款 26 中构建的代码为基础的，如果你最近没读过该条款，就先复习一下再往下读。

舍弃重载

条款 26 的第一个例子，`logAndAdd`，可以作为很多函数的代表，这样的函数只需把本来打算进行重载的版本重新命名成不同的多个名字就可以避免依万能引用型别进行重载。以 `logAndAdd` 的两个重载版本为例，就可以分别改成 `logAndAddName` 和 `logAndAddNameIdx`。啊呀，但这种方法不适用于第二个例子——`Person` 类的构造函数，因为构造函数的名字是由语言固化的。还有，彻底放弃重载也不是长久之计呀！

传递 const T& 型别的形参

一种替代方法是回归 C++98，使用传递左值常量引用型别来代替传递万能引用型别。其实，这就是条款 26 做的第一种问题解决尝试。这种方法的缺点是达不到我们想要的高效率。不过，在已知依万能引用型别进行重载会带来的不良效应以后，放弃部分效率来保持简洁性不失为仍有一定吸引力的权衡结果。

传值

一种经常能够提升性能，却不用增加一点复杂性的方法，就是把传递的形参从引用型别替换成值型别，尽管这是反直觉的。这种设计遵守了条款 41 的建议——当你知道肯定需要复制形参时，考虑按值传递对象。我把关于运作原理和效率提升的细节推迟到那个条款里讨论，而在此我仅仅展示一下该技术如何在 Person 一例中运用：

```
class Person {
public:
  explicit Person(std::string n)          // 替换掉 T&& 型别的构造函数；
  : name(std::move(n)) {}                 // std::move 的用法参见条款 41
```

```
  explicit Person(int idx)              // 同前
  : name(nameFromIdx(idx)) {}
  ...

private:
  std::string name;
};
```

由于 std::string 型别并没有只接受单个整型形参的构造函数，所有 int 或者类 int 型别（例如，std::size_t，short，long）的实参都会汇集到接受 int 型别的那个构造函数重载版本的调用。类似地，所有 std::string 型别的实参（包括可以构造出 std::string 型别对象之物，例如字面量“Ruth”）都会被传递给接受 std::string 型别的那个构造函数。综上，不会有让调用者出乎意料的情况。不过也许你可能会说一些人会传递 NULL 和 0 来表示空指针，从而意外调用接受 int 型别的那个重载版本。但是，这些人应该参阅条款 8，而且要一遍遍地读，直到他们一想到要使用 NULL 和 0 表示空指针，就会立刻退缩为止。

标签分派

无论是传递左值常量还是传值，都不支持完美转发。如果使用万能引用的动机就是为了实施完美转发，那就只能采用万能引用，别无他途。然而，我们也不想舍弃重载。那么，如果一不想放弃重载，二不想放弃万能引用，那又该如何避免依万能引用型别进行重载呢？

其实解决之道并非那么难。重载函数在调用时的决议，会考察所有重载版本的形参，以及调用端传入的实参，然后选择全局最佳匹配的函数，这需将所有的形参 / 实参组合都考虑在内。一个万能引用形参通常会导致的后果是无论传入了什么都给出一个精确匹配结果，不过，如果万能引用仅是形参列表的一部分，该列表中还有其他非万能引用型别的形参的话，那么只要该非万能引用形参具备充分差（sufficiently poor）的匹配能力，则它就足以将这个带有万能引用形参的重载版本踢出局。这个想法就是标签分派（标签分派）手法的基础，为便于后续理解，先看一个例子。

我们对前面 logAndAdd 的例子实施标签分派的改造，这里把代码重列一次，不然你恐怕会翻页走神：

```
std::multiset<std::string> names;       // 全局数据结构

template<typename T>                    // 制备日志条目
void logAndAdd(T&& name)                // 将名字添加到数据结构中
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
```

```
    names.emplace(std::forward<T>(name));
}
```

就这个函数本身来说，一切运作如仪。可是一旦引入接受 int 的重载版本，以按索引查找对象时，就会重回条款 26 的麻烦境地。本条款的目的，就在于避免那些问题。方法是不再添加重载版本，而是重新实现 logAndAdd，把它委托给另外两个函数，一个接受整型值，另一个接受其他所有型别。而 logAndAdd 本身则接受所有型别的实参，无论整型和非整型都来者不拒。

这两个完成了实际工作的函数名字为 logAndAddImpl，亦即，我们重载的其实是 logAndAddImpl。两个函数中的一个，会接受万能引用型别的形参。所以我们就面对着既有重载，又有万能引用的情景了。不过，这两个函数都还有第二个形参，该形参用来判断传入的实参是否为整型。正是这第二个形参阻止了我们落入条款 26 所描述的陷阱里，因为经过我们的精心安排，第二个形参就会是选择了哪个重载版本的决定因素。

我知道，“少废话，上代码！”没问题啊，下面是一个几乎正确的 logAndAdd 更新版本：

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_integral<T>());          // 不够正确
}
```

上面这个函数把它的形参转发给了 logAndAddImpl，但它还传递了另一个实参，用来表示那个形参的型别（T）是否为整型。至少，它应该做到这件事情。若传给 logAndAdd 的实参是右值整型，它就做到了。但是，如条款 28 所言，如果传给万能引用 name 的实参是个左值，那么 T 就会被推导为左值引用。所以，如果传递给 logAndAdd 的是个左值 int，则 T 就会被推导为 int&。这不是个整型，因为引用型别都不是整型。这意味着 std::is_intergral<T> 在函数接受了任意左值实参时，会得到结果“假”，尽管这样的实参确实表示了一个整型值。

意识到问题所在，也就相当于已经解决了问题。这不是？无比趁手的 C++ 标准库中有个型别特征（参见条款 9），std::remove_reference，正如其名，也正如所需：它会移除型别所附的的一切引用饰词。因此，正确的 logAndAdd 如下：

```
template<typename T>
void logAndAdd(T&& name)
{
```

```
    logAndAddImpl(
      std::forward<T>(name),
      std::is_integral<typename std::remove_reference<T>::type>()
    );
}
```

这才算完成了整套戏法（C++14 中可以少敲几下键盘，因为可以把上面的突出标示部分用 std::remove_reference_t<T> 代替。欲知详情，参见条款 9）。

完成 logAndAdd 以后，就可以把注意力放到被调用的函数 logAndAddImpl 上了。它有两个重载版本，第一个是只针对于非整型实施的（亦即，std::is_integral<typename std::remove_reference<T>::type 的结果是“假”）：

```
template<typename T>
void logAndAddImpl(T&& name, std::false_type)   // 非整型实参：
{                                               // 将名字添加到全局数据结构中
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

只要你理解了突出标示的形参背后的运行机制，就能看出来这是段直截了当的代码。概念上，logAndAdd 会向 logAndAddImpl 传递一个布尔值，用以表示传递给 logAndAdd 的实参是否为整型。不过，true 和 false 都是运行期值，可是我们需要利用的是重载决议（一种编译期现象）来选择正确的 logAndAddImpl 重载版本。这就意味着我们需要一个对应于 true 的型别，和一个对应于 false 的不同型别。这个需求足够普适，所以 C++ 标准库提供了名为 std::true_type 和 std::false_type 的一对型别来满足之。若 T 是整型，则经由 logAndAdd 传递给 logAndAddImpl 的实参就会是个继承自 std::true_type 的对象。反之，若 T 不是整型，该实参就会是个继承自 std::false_type 的对象。总的结果是，只有当 T 不是整型时，logAndAdd 发起的调用才会从候选中选定上面这个 logAndImpl 重载版本。

第二个重载版本则包含了 T 是整型的相反情况。在此，logAndAddImpl 仅仅通过传入的索引查找到对应的名字，然后就把该名字传回给 logAndAdd：

```
std::string nameFromIdx(int idx);               // 同条款 26
void logAndAddImpl(int idx, std::true_type)     // 整型实参：
{                                               // 查找名字并用它调用
  logAndAdd(nameFromIdx(idx));                  // logAndAdd
}
```

通过让 logAndAddImpl 按索引查找对应名字，然后传递给 logAndAdd（在那里，它会经由 std::forward 转发到另一个 logAndAddImpl 重载版本），就可以避免在两个重载版本都放入记录日志的代码。[译注 2]

在上述设计中，型别 std::false_type 和 std::true_type 就是所谓“标签”，运用它们的唯一目的在于强制重载决议按我们想要的方向推进。值得注意的是，这些形参甚至没有名字。它们在运行期不起任何作用，实际上，我们希望编译器能够识别出这些标签形参并未使用过，从而将它们从程序的执行镜像中优化掉（某些编译器的确会这样做，至少有时会）。针对 logAndAdd 内的重载实现函数发起的调用把工作“分派”到正确的重载版本的手法就是创建适当的标签对象。这种设计因而得名：标签分派。它是模板元编程的标准构件，所以你越是深入地查看当下的 C++ 库代码，你也就越是会频繁地看到这种设计。

就目的而言，更重要的不在于标签分派的工作细节，而是它让我们得以将万能引用和重载加以组合却不会引发条款 26 所描述的问题的能力。分派函数 logAndAdd 接受的是个不受限制的万能引用形参，但该函数并未重载。实现函数 logAndAddImpl 则实施了重载，每个重载版本都接受一个万能引用形参，但重载决议却并不仅对这个万能引用形参有依赖，还对标签有依赖，而标签值则又加以设计以保证可以命中匹配的函数不会超过一个。这样设计的结果是，只有标签值才决定了调用的是哪个重载版本。万能引用形参总是给出精确匹配这个事实，也就无关紧要了。

对接受万能引用的模板施加限制

标签分派能够发挥作用的关键在于，存在一个单版本（无重载版本的）函数作为客户端 API。该单版本函数会把待完成的工作分派到实现函数。创建无重载的分派函数通常并不难，但条款 26 所关注的第二个问题，即关于 Person 类的完美转发构造函数的那个问题，却是个例外。编译器可能会自行生成复制和移动构造函数，所以如果仅仅撰写一个构造函数，然后在其中运用标签分派，那么有些针对构造函数调用就可能会由编译器生成的构造函数接手处理，从而绕过了标签分派系统。

实际上，真正的问题并不在于编译器生成的函数有时候会绕过标签分派设计，而在于编译器生成的函数并不能保证一定会绕过标签分派设计。当收到使用左值对象进行同

译注 2：此处的手法甚妙，需细看。logAndAdd 先是把工作委托给一个 logAndAddImpl 重载版本，第二个重载版本又将委托返还至委托源函数，并再次发起委托，结果是委托给了前一个 logAndAddImpl 重载版本。为什么要将委托返还至委托源函数而不直接转发？这个问题留给读者思考。

型别对象的复制请求时，你几乎总会期望调用到的复制构造函数。但是，就如条款 26 所演示的那样，只要提供了一个接受万能引用形参的构造函数，会导致复制非常量左值时总会调用到万能引用构造函数（而非复制构造函数）。同一条款也解释了，如果基类中声明了一个完美转发构造函数，则派生类以传统方式实现其复制和移动构造函数时，总会调用到该构造函数，尽管正确行为应该是调用到基类的复制和移动构造函数。

对于这样的情况，也就是接受了万能引用形参的重载函数比你想要的程度更贪婪，而却又未贪婪到能够独当一面成为单版本分派函数的程度，标签分派就不是你想寻找的好伙伴了。你需要是的另一种独门绝技，它可以让你把含有万能引用部分的函数模板被允许采用的条件砍掉一部分。朋友，你需要的这门绝技，名叫 std::enable_if。

std::enable_if 可以强制编译器表现出来的行为如同特定的模板不存在一般。这样的模板称为禁用的。默认地，所有模板都是启用的。可是，实施了 std::enable_if 的模板只会在满足了 std::enable_if 指定的条件的前提下才会启用。在我们讨论的情况下，仅在传递给完美转发构造函数的型别不是 Person 时才启用它。如果传递的型别是 Person，我们就会想要禁用完美转发构造函数（即，让编译器忽略它），因为这么一来，就会由类的复制或移动构造函数来接手处理这次调用了，而这才是在用一个 Person 型别的对象来初始化另一个 Person 型别的对象时应有的结果。

欲表示这种想法并非那么难，不过语法实在让人望而却步，尤其如果以前从没见过就更如此，所以我得循循善诱。std::enable_if 的条件部分可以套用公式，所以我们就从那里着手。以下是 Person 类的完美转发构造函数声明，仅展示了使得 std::enable_if 得以运作的够用程度。之所以仅展示该构造函数的声明，是因为 std::enable_if 对实现毫无影响，该函数的实现和条款 26 中的一样。

```
class Person {
public:
  template<typename T,
           typename = typename std::enable_if<condition>::type>
  explicit Person(T&& n);

  ...

};
```

欲知突出标示部分的代码到底运作机理是什么，我只能抱歉地建议，你去找找其他资料。因为解释起来太复杂，本书没有足够空间（在你的研究过程中，请不仅要看 std::enable_if，也要看“SFINAE”，因为 SFINAE 是使得 std::enable_if 得以运作的技术）。在这里，我会将焦点放在控制该构造函数是否被启用的条件表达式上。

我们想指定的条件是，T 不是 Person 型别，即，仅当 T 是 Person 以外的型别时，才启用该模板构造函数。正好有个型别特征能够判定两个型别是否同一（std::is_same），这么一来，我们想要的条件好像是 !std::is_same<Person, T>::value（注意表达式一开始的“!”。因为我们想要 T 和 Person 不相同）。这个表达式已经接近我们想要的，但不甚正确，原因在于，正如条款 28 所解释的，使用左值初始化万能引用时，T 的型别推导结果总是左值引用。这就意味着，对于这样的代码：

```
Person p("Nancy");

auto cloneOfP(p);                    // 从左值出发进行初始化
```

万能引用构造函数中的 T 的型别会被推导成 Person&。型别 Person 和 Person& 不相同，std::is_same 的结果会反映这一事实：std::is_same<Person, Person&>::value 的值是“假”。

如果我们更加精细地加以反思，我们说仅当 T 不是 Person 型别时才启用 Person 类中模板构造函数到底是什么意思的话，我们就会意识到，在审查 T 的时应该忽略：

- **它是否是个引用**。为判定万能引用构造函数是否应该被启用，型别 Person、Person& 和 Person&& 都应该与 Person 作相同处理。
- **它是否带有 const 和 volatile 饰词**。对目前关注的目的而言，const Person、volatile Person 和 const volatile Person 都应该与 Person 作相同处理。

这意味着在判定 T 是否与 Person 相同之间，需要一种手段来移除 T 型别带有的所有引用、const 和 volatile 饰词。再一次地，标准库以型别特征的形式赐予了我们所需之物，该特征叫做 std::decay。std::decay<T>::type 和 T 相同，区别在于它移除了 T 的引用和 cv 饰词（即 const 或 volatile 饰词）［这里我捏造了事实，因为 std::decay，顾名思义，也用于把数组和函数型别强制转型成指针型别（参见条款 1），但是对于这里的讨论，std::decay 的行为确实和我前面描述的一致］。这么一来，我们心心念念的判定构造函数是否启动的条件就成了这样：

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

即，忽略了 T 型别的引用和 cv 饰词后，Person 和 T 型别仍不同一（如条款 9 所言，std::decay 之前的“typename”不能省略，因为 std::decay<T>::type 对模板形参 T 有依赖）。

把上面得到的条件表达式套到上面的公式中去，再把得到的代码结果重排一下格式，

这样就能够更容易看清楚各个部分是如何就位的，最后得出 Person 类的完美转发构造函数声明：

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_same<Person,
                               typename std::decay<T>::type
                              >::value
               >::type
  >
  explicit Person(T&& n);

  ...

};
```

如果之前你从没看过这样的代码，也不必自惭。我把这个设计放到最后，自有原因。如果你有可能使用其他技术来避免万能引用和重载的混合（你几乎总是可以选择其他技术），你就应该使用之。然而，一旦习惯了函数式语法和无处不在尖括号，就会发现它也没那么糟糕。而且，在经历了千辛万苦以后，它确实实现了你期望的行为。给定了上述构造函数，当我们用一个 Person 型别的对象（无论它是左值还是右值，带不带 const 或 volatile 饰词）来构造另一个 Person 型别的对象时，将永远不会调用到接受万能引用的构造函数。

成功了对吧？完工啦！

唔，还没。现在还不到开香槟庆祝的时间。条款 26 的还有一个地方有点松动，我们得把这个地方扎牢了。

给定一继承 Person 的类，以传统的方式实现其复制和移动构造函数：

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs)      // 复制构造函数；
  : Person(rhs)                                // 调用的是
  { ... }                                      // 基类的完美转发构造函数！

  SpecialPerson(SpecialPerson&& rhs)           // 移动构造函数；
  : Person(std::move(rhs))                     // 调用的是
  { ... }                                      // 基类的完美转发构造函数！

  ...
};
```

以上和我在条款 26 展示的是同一段代码，连注释都未改一字，并且这注释仍然成

立。当复制或者移动一个 SpecialPerson 型别的对象时，我们会期望通过基类的复制或移动构造函数来完成该对象基类部分的复制或移动，不过在这些函数中，我们传递给基类构造函数的是 SpecialPerson 型别的对象，因为 SpecialPerson 和 Person 型别不同（实施过 std::decay 仍然不同），基类的万能引用构造函数会启用，后者很乐意在精确匹配了 SpecialPerson 型别的实参后执行实例化。原因在于，这个精确匹配比起 Person 类中的复制和移动构造函数所要求的从派生类到基类的强制转型才能把 SpecialPerson 型别的对象绑定到 Person 型别的形参来说，是更佳的匹配。所以，我们现有的代码在复制和移动 SpecialPerson 型别的对象时，会使用 Person 类的完美转发构造函数来复制和移动它们的基类部分。条款 26 中提及的有关问题，在此全部重现了一遍。

因为派生类只是遵循着正常规则在实现其复制和移动构造函数，所以问题的解决还须在基类中完成。更准确地说，就在那个决定了 Person 类的万能引用构造函数是否启用的条件中。我们现在认识到，我们想要的并非是为与 Person 不同一的实参型别启用模板构造函数，我们想要的是为与 Person 或继承自 Person 的型别都不同一的实参型别才启用模板构造。恼人的继承！

当你听说标准库中有个型别特征可以用来判定一个型别是否由另一个型别派生而来时，应该不会感到吃惊，该型别特征名叫 std::is_base_of。若 T2 由 T1 派生而来，std::is_base_of<T1, T2>::value 是“真”。所有型别都可以认为是从它自身派生而来，所以 std::is_base_of<T, T>::value 是“真”。这下就方便了，因为我们想要修改 Person 类的完美转发构造函数的启用条件，改成仅在型别 T 去除引用和 cv 饰词后，既非 Person 型别，亦非从 Person 派生的型别。以 std::is_base_of 代替 std::is_same 就可以得到想要的东西：

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_base_of<Person,
                                  typename std::decay<T>::type
                                 >::value
               >::type
  >
  explicit Person(T&& n);

  ...

};
```

到这里，才算终于完工了。前提是，我们在使用 C++11 撰写代码。如果在使用的是

C++14，上述代码仍然成立，不过我们可以利用别名模板来去掉 std::enable_if 和 std::decay 累赘的“typename”和“::type”部分，从而产生下面这段更加养眼的代码：

```
class Person {                                          // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<                        // 这里代码量更少
                !std::is_base_of<Person,
                                 std::decay_t<T> // 还有这里
                                >::value
               >                                        // 还有这里
  >
  explicit Person(T&& n);

  ...

};
```

好吧，我承认我在说谎。我们仍然不能说已经完工，但我们业已接近，接近得不能再接近了。我诚实着呢。

我们已经看到如何运用 std::enable_if 来为那些本来就想使用 Person 类的复制和移动构造函数的型别选择性地禁用 Person 类中接受万能引用的构造函数，但我们还没有看到如何运用这一技术来区分实参是整型还是非整型。而后者，毕竟才是我们最原始的目标。构造函数的多义词问题，一路上都在牵扯着我们注意力。

我们需要做的一切（这一回真的是所有工作了）就是：①为 Person 类添加一个处理整型实参的构造函数重载版本；②进一步限制模板构造函数，在接受整型实参时禁用之。把所有配料一股脑儿倒入我们先前所有讨论之锅，开文火炖煮，即可欣赏成功的芳香：

```
class Person {
public:
  template<
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)          // 接受 std::string 型别以及
  : name(std::forward<T>(n))      // 可以强制转型到 std::string 的实参型别的
  { ··· }                         // 构造函数

  explicit Person(int idx)        // 接受整型实参的构造函数
  : name(nameFromIdx(idx))
  { ··· }
```

```
  ...                                   // 复制和移动构造函数等
private:
  std::string name;
};
```

乌拉！美哉！不过，好吧，这句赞美更有可能是那些模板元编程迷发出的，但事实上这方法不仅能够运作，还表现出了一种独特的从容。这么说的理由，一来是它利用了完美转发，达成了最高效率，二来它又控制了万能引用和重载的组合，而非简单地禁用之。该技术可以实施于重载无法避免的场合（如构造函数）。

权衡

本条款关注的头三种技术（舍弃重载、传递 const T& 型别的形参和传值）都需要对待调用的函数形参逐一指定型别，而后两种技术（标签分派和对模板的启用资格施加限制）则利用了完美转发，因此无须指定形参型别。这个基础决定（指定，还是不指定型别）不无后果。

按理说，完美转发效率更高，因为它出于和形参声明时的型别严格保持一致的目的，会避免创建临时对象。在 Person 类的构造函数一例中，完美转发就允许把形如“Nancy”的字符串字面量转发给某个接受 std::string 的构造函数。而未使用完美转发的技术则一定得先从字符串字面量出发创建一个临时 std::string 对象，方能满足 Person 类的构造函数的形参规格。

但是完美转发亦有不足，首先是针对某些型别无法实施完美转发，尽管它们可以被传递到接受特定型别的函数，条款 30 探索了这些完美转发的失效案例。

其次是在客户传递了非法形参时，错误信息的可理解性。例如，假设在创建 Person 型别的对象时，传递的是个 char16_t 型别（C++11 引进的 16 比特字符型别）而非 char 型别的字符组成的字符串：

```
Person p(u "onrad Zuse");      // “Konrad Zusz”由 char16_t 型别的字符组成
```

如果是本条款介绍的前三种技术，编译器会发现，可用的构造函数只能接受 int 或 std::string。所以，它们会产生出多少算是直截了当的错误信息来解释：无法将 const char16_t[12] 强制转型到 int 或 std::string。

而如果使用的是基于完美转发的技术时，情况大不一样。const char16_t 型别的数组在绑定到构造函数的形参时，会一声不吭。接着，它又被转发到 Person 的 std::string 型别的成员变量的构造函数中。唯有在那里，调用者的传递之物（一个 const char16_t 型别的数组）与所要求的形参（std::string 的构造函数可接受的形

参型别）之间的不匹配才会被发现。如此产生的结果错误信息，很可能，嗯，会一眼难忘。在我使用的某个编译器上，它有 160 多行。

在本例中，万能引用只转发了一次（从 Person 类的构造函数到 std::string 类的构造函数）。但系统越是复杂，就越有可能某个万能引用要经由数层函数调用完成转发，才会抵达决定实参型别是否可接受的场所。万能引用转发的次数越多，某些地方出错时给出的错误信息就越让人摸不着头脑。许多程序员都发现，即使性能是首要的关注因素，在接口中也不去使用万能引用形参，根本原因就在于这一问题。

在 Person 一例中，我们了解转发函数的万能引用形参应该用作 std::string 型别的对象的初始化物，所以可以使用 static_assert 来验证其能够扮演这个角色。std::is_constructible 这个型别特征能够在编译期间判定具备某个型别的对象是否从另一型别（或另一组型别）的对象（或另一组对象）出发完成构造，所以这个断言很容易撰写：

```
class Person {
public:
  template<                                                    // 同前
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)
  : name(std::forward<T>(n))
  {
    // 断言可以从 T 型别的对象构造一个 std::string 型别的对象
    static_assert(
      std::is_constructible<std::string,  T>::value,
      "Parameter n can't be used to construct a std::string"
    );

    …                                  // 构造函数通常要完成的工作放在这里

  }

  …                                    // Person 类的其余部分（同前）

};
```

这会在客户代码尝试从一个无法构造出 std::string 型别的对象的型别出发来创建一个 Person 型别的对象时，产生出该指定错误信息。不幸的是，在本例中，static_assert 位于构造函数的函数体内，而转发代码属于成员初始化列表的一部分，位于它之前。在我使用的编译器中，产生自 static_assert 的漂亮、可读的错误信息仅会在通常的错误信息（那 160 多行）发生完之后才姗姗来迟。

要点速记

- 如果不使用万能引用和重载的组合，则替代方案包括使用彼此不同的函数名字、传递 const T& 型别的形参、传值和标签分派。
- 经由 std::enable_if 对模板施加限制，就可以将万能引用和重载一起使用，不过这种技术控制了编译器可以调用到接受万能引用的重载版本的条件。
- 万能引用形参通常在性能方面具备优势，但在易用性方面一般会有劣势。

条款 28：理解引用折叠

条款 23 曾经提及，实参在传递给函数模板时，推导出来的模板形参会将实参是左值还是右值的信息编码到结果型别中。但该条款未曾提及，这个编码操作只有在实参被用以初始化的形参为万能引用时才会发生。不过，有一个充分的理由来解释为什么当时不提这茬：万能引用是在条款 24 中才介绍。把万能引用和左右值编码信息的论述综合起来，意思就是，以下面这个模板为例：

```
template<typename T>
void func(T&& param);
```

模板形参 T 的推导结果型别中，会把传给 param 的实参是左值还是右值的信息给编码进去。

编码机制是直截了当的：如果传递的实参是个左值，T 的推导结果就是个左值引用型别；如果传递的实参是个右值，T 的推导结果就是个非引用型别（注意这里的非对称性：左值的编码结果为左值引用型别，但右值的编码结果却是非引用型别）。是故有下面的结果：

```
Widget widgetFactory();          // 返回右值的函数

Widget w;                        // 变量（左值）

func(w);                         // 调用 func 并传入左值；T 的推导结果型别为 Widget&

func(widgetFactory());           // 调用 func 并传入左值；T 的推导结果型别为 Widget
```

两个对 func 的调用，传递的实参型别都为 Widget。不同之处仅在于，一个是左值而另一个是右值，而这个不同之处却导致了针对模板形参 T 得出了不同的型别推导结果。这个机制，正如我们将很快看到的，就是决定了万能引用是变成左值引用还是右值引用的机制，也是 std::forward 得以运作的机制。

在我们以更深入细致的视角来观察 std::forward 和万能引用之前，我们必须注意一个事实，在 C++ 中，“引用的引用”是非法的。你如果胆敢尝试声明一个，则编译器必会对此大加痛斥：

```
int x;
...
auto& & rx = x;          // 错误！不可以声明“引用的引用”！
```

但如果仔细琢磨一下，当左值被传递给接受万能引用的函数模板时会发生下面的状况：

```
template<typename T>
void func(T&& param);          // 同前

func(w);                       // 调用 func 并传入左值；T 的推导结果型别为 Widget&
```

如果把 T 的推导结果型别（即 Widget&）代码实例化模板，不就得到下面的结果了吗：

```
void func(Widget& && param);
```

引用的引用！然而编译器却一声未吭。条款 24 告诉过我们，由于万能引用 param 是用左值初始化的，其推导结果型别理应是个左值引用，但编译器是如何取用了 T 的推导结果型别的，并代入模板，从而使它拥有了下面这个终极版函数签名呢？

```
void func(Widget& param);
```

答案就是引用折叠。是的，你是被禁止声明引用的引用，但编译器却可以在特殊的语境中产生引用的引用，模板实例化就是这样的语境之一。当编译器生成引用的引用时，引用折叠机制便支配了接下来发生的事情。

有两种引用（左值和右值），所以就有四种可能的引用——引用组合（左值－左值，左值－右值，右值－左值，右值－右值）。如果引用的引用出现在允许的语境（例如，在模板实例化过程中），该双重引用会折叠成单个引用，规则如下：

> 如果任一引用为左值引用，则结果为左值引用。否则（即两个皆为右值引用），结果为右值引用。

在上述例子中，将推导结果型别 Widget& 代入函数模板 func 后，产生了一个指涉到左值引用的右值引用。尔后，根据引用折叠规则，结果是个左值引用。

引用折叠是使 std::forward 得以运作的关键。如同条款 25 所解释的那样，std::forward 会针对万能引用实施，这么一来，就会出现如下的常见用例：

```
template<typename T>
void f(T&& fParam)
{
  ...                                          // 完成一些操作

  someFunc(std::forward<T>(fParam));    // 将 fParam 转发至 someFunc
}
```

由于 fParam 是个万能引用，我们就知道，传递给 f 的实参（即用以初始化 fParam 的表达式）是左值还是右值的信息会被编码到型别形参 T 中。std::forward 的任务是，当且仅当编码 T 中的信息表明传递给实参是个右值，即 T 的推导结果型别是个非引用型别时，对 fParam（左值）实施到右值的强制型别转换。

这里是 std::forward 的一种能够完成任务的实现：

```
template<typename T>                                  // 在名字空间 std 中
T&& forward(typename
              remove_reference<T>::type& param)
{
  return static_cast<T&&>(param);
}
```

这个实现和标准并不完全符合（我省去了若干接口细节），不过，那些有差异的部分对于理解 std::forward 的行为并无干系。

假设传递给函数 f 的实参的型别是个左值 Widget，则 T 会被推导为 Widget& 型别，然后对 std::forward 的调用就会实例化为 std::forward<Widget&>。而将 Widget& 插入 std::forward 的实现就会产生如下结果：

```
Widget& && forward(typename
                     remove_reference<Widget&>::type& param)
{ return static_cast<Widget& &&>(param); }
```

由于型别特征 remove_reference<Widget&>::type 的产生结果是 Widget 型别（参见条款 9），所以 std::forward 又变换成了下面的结果：

```
Widget& && forward(Widget& param)
{ return static_cast<Widget& &&>(param); }
```

引用折叠同样在返回值和强制型别转换的语境中得到了实施，导致实际调用结果是这样的终极版本 std::forward：

```
Widget& forward(Widget& param)                        // 仍在名字空间 std 中
{ return static_cast<Widget&>(param); }
```

如你所见，当左值实参被传递给函数模板 f 时，std::forward 实例化结果是：接受一个左值引用，并返回一个左值引用。而 std::forward 内部的强制型别转换未做任何事情，

因为 param 的型别已经是 Widget& 了，所以再要把它强制转换到 Widget& 型别不会产生什么效果。综上，被传递给 std::forward 的左值实参会返回一个左值引用。根据定义，左值引用是左值，所以传递左值给 std::forward 会导致返回一个左值，符合认知。

再假设传递给 f 的实参是右值 Widget 型别。在此情况下，f 的型别形参 T 的推导结果是个光秃秃的 Widget。因此，f 内部的 std::forward 就成了 std::forward<Widget>。在 std::forward 的实现中，在 T 之处用 Widget 代入，就得出下面的代码：

```
Widget&& forward(typename
                 remove_reference<Widget>::type& param)
{ return static_cast<Widget&&>(param); }
```

针对非引用 Widget 型别实施 std::remove_reference 会产生和起始型别相同的结果（Widget），所以 std::forward 又变成了这样：

```
Widget&& forward(Widget& param)
{ return static_cast<Widget&&>(param); }
```

这里没有发生引用的引用，所以也就没有发生引用折叠，所以这也就已经是本次 std::forward 调用的最终实例化版本了。

由函数返回的右值引用是定义为右值的，所以在此情况下，std::forward 会把 f 的形参 fParam（左值）转换成右值。最终的结果是，传递给函数 f 的右值实参会作为右值转发到 someFunc 函数，这也精确地符合认知。

在 C++14 中有了 std::remove_reference_t，从而 std::forward 的实现得以变得更加简明扼要：

```
template<typename T>                            // C++14
T&& forward(remove_reference_t<T>& param)       // 仍在名字空间 std 中
{
  return static_cast<T&&>(param);
}
```

引用折叠会出现的语境有四种。第一种，最常见的一种，就是模板实例化。第二种，是 auto 变量的型别生成。技术细节本质上和模板实例化一模一样，因为 auto 变量的型别推导和模板的型别推导在本质上就是一模一样的（参见条款 2）。重新反思下本条款前面出现过的一个例子：

```
template<typename T>
void func(T&& param);

Widget widgetFactory();         // 返回右值的函数

Widget w;                       // 变量（左值）
```

```
func(w);                    // 以左值调用函数，T 的型别推导结果为 Widget&
func(widgetFactory());      // 以右值调用函数，T 的型别推导结果为 Widget
```

这一切都能以 auto 形式模仿。下面这个声明：

```
auto&& w1 = w;
```

初始化 w1 的是个左值，因此 auto 的型别推导结果为 Widget&。在 w1 声明中以 Widget& 代入 auto，就产生了以下这段涉及引用的引用的代码，

```
Widget& && w1 = w;
```

引用折叠之后，又会变成

```
Widget& w1 = w;
```

这就是结果：w1 乃是左值引用。

再看一例，下述声明：

```
auto&& w2 = widgetFactory();
```

以右值初始化 w2，auto 的型别推导结果为非引用型别 Widget。将 Widget 代入 auto 就得到：

```
Widget&& w2 = widgetFactory();
```

这里并无引用的引用，所以到此结束：w2 乃是右值引用。

话说到这里，我们才真正地理解了条款 24 中介绍的万能引用。万能引用并非一种新的引用型别，其实它就是满足了下面两个条件的语境中的右值引用：

- **型别推导的过程会区别左值和右值**。T 型别的左值推导结果为 T&，而 T 型别的右值则推导结果为 T。
- **会发生引用折叠**。

万能引用的概念是有用的，有了这个概念以后，就避免了需要识别出存在引用折叠的语境，根据左值和右值的不同脑补推导过程，然后再脑补针对推导的结果型别代入引用折叠发生的语境后应用引用折叠规则。

我曾在前面提及有四种会发生引用折叠的语境，不过目前只讨论过两种：模板实例化和 auto 型别生成。第三种语境是生成和使用 typedef 和别名声明（参见条款 9）。如

果在 typedef 的创建或者评估求值的过程中出现了引用的引用，引用折叠就会出手消灭之。例如，假设我们有个类模板 Widget，内嵌一个右值引用型别的 typedef，

```
template<typename T>
class Widget {
public:
  typedef T&& RvalueRefToT;
...
};
```

再假设我们以左值引用型别来实例化该 Widget：

```
Widget<int&> w;
```

在 Widget 中以 int& 代入 T 的位置，则得到如下的 typedef：

```
typedef int& && RvalueRefToT;
```

引用折叠又将上述语句化简得到：

```
typedef int& RvalueRefToT;
```

这个结果显然表明，我们为 typedef 选择的名字也许有些名不符实：当以左值引用型别实例化 Widget 时，RvalueRefToT 其实成了个左值引用的 typedef。

最后一种会发生引用折叠的语境在于 decltype 的运用中。如果在分析一个涉及 decltype 的型别过程中出现了引用的引用，则引用折叠亦会介入并消灭之（有关 decltype 的信息，详见条款 3）。

要点速记

- 引用折叠会在四种语境中发生：模板实例化、auto 型别生成、创建和运用 typedef 和别名声明，以及 decltype。
- 当编译器在引用折叠的语境下生成引用的引用时，结果会变成单个引用。如果原始的引用中有任一引用为左值引用，则结果为左值引用。否则，结果为右值引用。
- 万能引用就是在型别推导的过程会区别左值和右值，以及会发生引用折叠的语境中的右值引用。

条款 29：假定移动操作不存在、成本高、未使用

移动语义可以说在 C++11 的所有语言特性中占据着首要中的首要地位。“移动容器现在和复制指针一样成本低廉了！”这是你很可能听说过的，类似说法还有“复制临时对象现在已经如此高效，如果刻意在撰写代码中避免它，就无异于犯了过早优化的禁忌！”这些情绪化的言辞不难理解。移动语义确实是个重要的语言特性。语言不只是允许编译器使用成本相对低廉的移动操作来代替昂贵的复制操作，实际上语言会要求编译器这样做（只要满足适当条件就必须这样做）。调出你的 C++98 版本的代码存根，然后只消一字不改地使用符合 C++11 标准的编译器和标准库重新编译一遍，叫一声“变！”，你的软件便应声增速。

移动语义确有此功，所以这个语言特性一传十、十传百，渐渐成了传奇。传奇嘛，你懂的，一般都是夸大其辞的结果。本条款就是想让你对这个语言特性的期望接上地气。

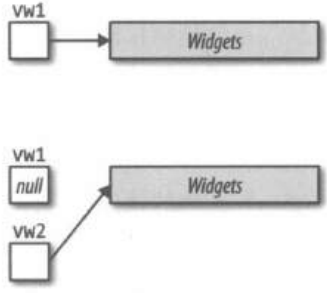
让我们从为何许多型别不能支持移动语义的观察开始。整个 C++ 98 标准库都已为 C++11 彻底翻修过，目的是为那些的型别移动的可以实现成比复制更快的型别增添移动操作，而且库组件的实现也已完成修订以充分利用这些移动操作，不过问题在于你有可能手上的代码存根并未完成修订以充分利用 C++11 的良好特性。若你的应用中的（或采用的库中的）型别没有为 C++11 做过专门修改，那么仅仅在编译器中有着对移动操作的支持也并不会给你带来什么明显好处。诚然，C++11 愿意为这些缺少移动操作的类生成移动操作，但这仅适用于那些未声明复制操作、移动操作以及析构函数的类（参见条款 17）。型别若是有数据成员或是基类禁用了移动（例如，通过将移动操作删除的方式参见条款 11），也将导致编译器生成的移动操作被抑制掉。如果型别并不提供对移动的显式支持，也不符合编译器生成移动操作的条件，当然也就没有理由期望它在 C++11 下比在 C++98 下有任何性能提升了。

即使对于那些显式支持移动操作的型别，也可能不会像人们希望的那样带来那么大的利好。举个例子，在标准的 C++11 库中，所有的容器都支持移动操作，但如果因此就断言所有的容器的移动都是成本低廉的，那就贻笑大方了。对于有些容器而言，根本没有什么成本低廉的途径来移动其内容。而对于另一些容器而言，它们虽然有着确实成本低廉的移动操作，但却又有一些附加条件造成其容器元素不能满足。

考虑一下 `std::array` 这个 C++11 中引入的新容器型别，它实质上就是带有 STL 接口的内建数组。这一点和其他标准容器存在根本差异，因为其他标准容器都是将其内容存放在堆上的，从而在概念上，只需（以数据成员的方式）持有一个指涉到存放容器

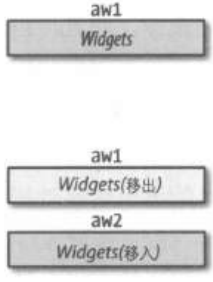
内容的堆内存的指针（实际情况当然比这更复杂些，但出于这里的分析目的，这些差异并不要紧）。由于该指针的存在，把整个容器的内容在常数时间内加以移动就成为了可能：仅仅把那个指涉到容器内容的指针从源容器复制到目标容器，尔后把源容器包含的指针置空即可：

```
std::vector<Widget> vw1;

// 将数据放入 vw1

...

// 移动 vw1 入 vw2。
// 完成执行仅须常数时间。
// 仅仅是包含在 vw1 和 vw2 中的指针被修改了
auto vw2 = std::move(vw1);
```



而 std::array 型别的对象则缺少这样一个指针，因为其内容数据是直接存储在对象内的。

```
std::array<Widget, 10000> aw1;

// 将数据放入 aw1

...

// 移动 aw1 入 aw2。
// 完成执行需要线性时间。
// 需要把 aw1 中的所有元素移动入 aw2
auto aw2 = std::move(aw1);
```



请注意，aw1 中的元素是被移动入 aw2 中的。假定 Widget 型别的移动比复制更快，则移动一个元素型别为 Widget 的 std::array 自然也会比复制一个同样的 std::array 更快。毫无疑问，std::array 当然提供移动支持。然而无论是移动还是复制 std:array 型别的对象都还是需要线性时间的计算复杂度，因为容器中的每个元素都必须逐一复制或移动。这可比人们有时会听说的“移动容器现在和赋值一堆指针一样成本低廉了”这类说法，要超出太多了。

作为对比，std::string 型别提供的是常数时间的移动和线性时间的复制。这听起来像是在说，它的移动比复制更快，但可能并非如此。许多 string 的实现都采用了小型字符串优化（small string optimization，SSO）。采用了 SSO 以后，“小型”字符串（例如，容量不超过 15 个字符的字符串）会存储在的 std::string 对象内的某个缓冲区内，而不去使用在堆上分配的存储。在使用了基于 SSO 的实现的前提下，对小型字符串

实施移动并不比复制更快，因为通常在移动的性能优于复制背后“仅复制一个指针”的把戏会失灵了。

有着发明 SSO 的动机，就充分表明短字符串才是许多应用程序的标配。使用一个内部缓冲区来存储这样的字符串，就消弭了为它们动态分配内存的需要，而这往往在效率上是一步胜手。而这样做是一步胜手，就隐含着移动并不比复制更快的意思，当然正好比半杯水既可以说是半空也可以说是半满，前面这句话同样可以说成复制并不比移动更慢。

即使是那些支持快速移动操作的型别，一些看似万无一失的移动场景还是以复制副本告终。条款 14 解释过，标准库中一些容器操作提供了强异常安全保证，并且为了确保依赖于这样的保证的那些 C++98 的遗留代码在升级到 C++11 时不会破坏这样的保证，底层的复制操作只有在已知移动操作不会抛出异常的前提下才会使用移动操作将其替换。这么做导致的一个后果是，即使某个型别的移动操作比对应的复制操作更高效，甚至在代码的某个特定位置，移动操作一般肯定不会有问题（例如，源对象是个右值的情况下），编译器仍会强制去调用一个复制操作，只要对应的移动操作未加上 noexcept 声明。

总而言之，在这样几个场景中，C++11 的移动语义不会给你带来什么好处：

- **没有移动操作**：待移动的对象未能提供移动操作。因此，移动请求就变成了复制请求。
- **移动未能更快**：待移动的对象虽然有移动操作，但并不比其复制操作更快。
- **移动不可用**：移动本可以发生的语境下，要求移动操作不可发射异常，但该操作未加上 noexcept 声明。

值得一提的是，还有另一种场景使得移动语义无法提供效率增益：

源对象是个左值：除了极少数例外（参见条款 25 中的例子），只有右值可以作为移动操作的源。

但本条款的标题是假定移动操作不存在、成本高、未使用。这比较典型地适合于通用代码的情形，例如，撰写模板的时候，因为你还不知道将要与哪些型别配合。在这种情况下，你必须像在使用 C++98 一样保守地去复制对象，正如移动语义尚不存在那样。这同样也符合“不稳定”代码的情形，即代码中所涉及型别的特征会比较频繁地加以修改。

然而，通常你已知代码中会使用的型别，也可以肯定它们的特性不会改变（例如，它们是否支持成本低廉的移动操作）。如果是这样的情形，你就不需要前面那些假定。你可以直接查阅所使用的型别对移动的支持细节。如果涉及的型别能够提供成本低廉的移动操作，并且是在这些移动操作会被调用的语境中使用对象，则可以放心大胆地依靠移动语义来将复制操作替换成相对不那么昂贵的对应移动操作。

要点速记

- 假定移动操作不存在、成本高、未使用。
- 对于那些型别或对于移动语义的支持情况已知的代码，则无需作以上假定。

条款 30：熟悉完美转发的失败情形

在 C++11 的宝箱上最引人注目的语言特性纹章之一，就是完美转发。完美转发，可是完美的哟！不过，揭开宝箱的表面，你才会发现会有这样的“完美”（理想版）和那样的“完美”（现实版）。C++11 的完美转发相当不错，但如果一定要说达到了真正的完美之境，那就不能拘泥于若干“小节”。该条款就致力让你熟悉这些“小节”。

在我们的“小节”探险启程之前，还是值得反思一下“完美转发”的确切含义。“转发”的含义不过是一个函数把自己的形参传递（转发）给另一个函数而已。其目的是为了让第二个函数（转发目的函数）接受第一个函数（转发发起函数）所接受的同一对象。这就排除了按值传递形参，因为它们只是原始调用者所传递之物的副本。我们想要转发目的函数能够处理原始传入对象。指针形参也只能出局，因为我们不想强迫调用者传递指针。论及一般意义上的转发时，都是在处理形参为引用型别的情形。

完美转发的含义是我们不仅转发对象，还转发其显著特征：型别、是左值还是右值，以及是否带有 const 或 volation 饰词等。结合前面的观察分析，即我们一般是会和引用形参打交道，这就是说，我们会运用万能引用（参见条款 24），因为只有万能引用形参才会将传入的实参是左值还是右值这一信息加以编码。

假设有某函数 f，尔后我们打算撰写一个函数（其实是函数模板）将 f 作为转发目标。欲达此目的，我们需要核心代码如下：

```
template<typename T>
void fwd(T&& param)              // 接受任意实参
{
  f(std::forward<T>(param));     // 转发该实参到 f
}
```

转发函数，天然就应该是泛型的。fwd 模板就是个例子，它接受任意型别的实参，然后无论接受了什么都加以转发。对于这样的泛型，一种符合逻辑的拓展就是使得转发函数不只是模板，而且是可变长形参模板，从而能够接受任意数量的实参，可变长形参形式的 fwd 长成这样：

```
template<typename... Ts>
void fwd(Ts&&... params)                    // 接受任意实参
{
  f(std::forward<Ts>(params)...);           // 转发所有实参到 f
}
```

这种形式你可以在很多地方见到，包括标准容器的置入函数（参见条款 42），以及智能指针的工厂函数 std::make_shared 和 std::make_unique（参见条款 21）。

给定目标函数 f 和转发函数 fwd，当以某特定实参调用 f 会执行某操作，而用同一实参调用 fwd 会执行不同的操作，则称完美转发失败：

```
f( expression );                            // 如果本语句执行了某操作，
fwd( expression );                          // 而本语句执行了不同的操作，
                                            // 则称 fwd 完美转发 expression 到 f 失败
```

有若干种实参会导致该失败。重要之处在于知道这几种实参是什么，以及如何绕过它们。现在就让我们开始逐一讲述这些不能实施完美转发的实参吧。

大括号初始化物

假设 f 声明如下：

```
void f(const std::vector<int>& v);
```

在此情况下，以大括号初始化物调用 f 可以通过编译：

```
f({ 1, 2, 3 });         // 没问题，
                        // "{1, 2, 3}" 会隐式转换为 std::vector<int>
```

但如果把同一大括号初始化物传递给 fwd 则无法通过编译：

```
fwd({ 1, 2, 3 });       // 错误！无法通过编译
```

原因在于，大括号初始化物的运用，就是一种完美转发失败的情形。

凡是归类于此的失败，原因都一模一样。在对 f 的直接调用中（如 f({1,2,3})），编译器先领受了调用端的实参型别，又领受了 f 所声明的形参型别。编译器会比较这两个型别来确定它们是否兼容，尔后，如有必要，会实施型隐式型别转换来使得调用得

以成功。在上面的例子中，编译器从 {1,2,3} 出发生成一个临时的 std::vector<int> 型别对象，从而 f 的形参就有了一个 std::vector<int> 对象得以绑定。

而经由转发函数模板 fwd 来对 f 实施间接调用时，编译器就不再会比较 fwd 的调用处传入的实参和 f 中所声明的形参了。取而代之的是，编译器会采用推导的手法来取得传递给 fwd 实参的型别结果，尔后它会比较推导型别结果和 f 声明的形参型别。完美转发会在下面两个条件中的任何一个成立时失败：

- **编译器无法为一个或多个 fwd 的形参推导出型别结果。**在此情况下，代码无法编译通过。
- **编译器为一个或多个 fwd 的形参推导出了"错误的"型别结果。**这里所谓"错误的"，既可以指 fwd 根据型别推导结果的实例化无法通过编译，也可以指以 fwd 推导而得的型别调用 f 与直接以传递给 fwd 的实参调用 f 行为不一致。这种分裂行为的源泉之一，可能在于 f 乃是个重载函数的名字，然后，依据"不正确的"推导型别，fwd 里调用到的 f 重载版本，就与直接调用 f 的版本有异。

在上述"fwd({1,2,3})"这句调用一例中，问题在于向未声明为 std::initializer_list 型别的函数模板形参传递了大括号初始化物，因为这样的语境按规定，用标准委员会的行话说，叫做"非推导语境"。通俗地说，这个词的意思是，由于 fwd 的形参未声明为 std::initializer_list，编译器就会被禁止在 fwd 的调用过程中从表达式 {1,2,3} 出发来推导型别。而既然从 fwd 的形参出发进行推导是被阻止的行为，所以编译器拒绝这个调用也是合情合理的。

有意思的事情来了，条款 2 曾经说明过，auto 变量在以大括号初始化物完成初始化时，型别推导可以成功。这样的变量会被视为 std::initializer_list 型别对象，这么一来，如果转发函数的形参的推导型别结果应为 std::initializer_list 的话，就有了一个简单易行的绕行手法——先用 auto 声明一个局部变量，然后将该局部变量传递给转发函数：

```
auto il = { 1, 2, 3 };          // il 的型别推导结果为
                                // std::initializer_list<int>

fwd(il);                        // 没问题，将 il 完美转发给 f
```

0 和 NULL 用作空指针

条款 8 曾经说明过，若尝试把 0 和 NULL 以空指针之名传递给模板，型别推导就会发生行为扭曲，推导结果会是整型（一般情况下会是 int）而非所传递实参的指针型别。结

论就是：0 和 NULL 都不能用作空指针以进行完美转发。不过，修正方案也颇简单：传递 nullptr，而非 0 或 NULL。欲知详情，请参阅条款 8。

仅有声明的整型 static const 成员变量

有这么个普适的规定：不需要给出类中的整型 static const 成员变量的定义，仅需声明之。因为编译器会根据这些成员的值实施常数传播，从而就不必再为它们保留内存。举个例子，考虑下面这段代码：

```
class Widget {
public:
  static const std::size_t MinVals = 28;          // 给出了 MinVals 的声明
  ...
};
...                                               // 未给出 MinVals 的声明

std::vector<int> widgetData;
 widgetData.reserve(Widget::MinVals);             // 此处用到了 MinVals
```

在这里，尽管 Widget::MinVals（以下简称 Minvals）并无定义，我们还是利用了 MinVals 来指定 widgetData 的初始容量。编译器绕过了 MinVals 缺少定义的事实（编译器的行为是这样规定的），手法是把值 28 塞到所有提及 MinVals 之处。未为 MinVals 的值保留存储这一事实并不会带来问题。如果产生了对 MinVals 实施取址的需求（例如，有人创建了一个指涉到 MinVals 的指针），MinVals 就得要求存储方可（因此指针才能够指涉到它），然后上面这段代码虽然仍能够通过编译，但是如果不为 MinVals 提供定义，它在链接期就会遭遇失败。

记住了上述预备知识，然后想象 f（fwd 转发实参的目的函数）声明如下：

```
void f(std::size_t val);
```

以 MinVals 直接调用 f 没问题，因为编译器会用 MinVals 的值来代替它自己：

```
f(Widget::MinVals);              // 没问题，当“f(28)”处理
```

哎呀，如果想经由 fwd 来调用 f，便会碰壁了：

```
fwd(Widget::MinVals);            // 错误！应该无法链接
```

上面的代码能够通过编译，却不能完成链接。如果这能提醒你想起在对 MinVals 实施取址所发生过的失败，这就对了，因为其底层原理相同。

尽管源代码看上去并没有对 MinVals 实施取址，但注意到 fwd 的形参是个万能引用，而引用这东西，在编译器生成的机器代码中，通常是当指针处理的。程序的二进制代

码中（从硬件视角来看），指针和引用在本质上是同一事物。在此层次，有一句老话说得对：引用不过是会提领的指针罢了。既然如此，MinVals 按引用传递和按指针传递结果也就没有什么区别了。基于同样的理由，也得准备某块内存以供指针去指涉。按引用传递整型 static const 成员变量通常要求其加以定义，而这个需求就会导致代码完美转发失败而等价的、未使用完美转发的代码却能成功。

你可能已经注意到，在先前的讨论中我在有些地方闪烁其词。代码“应该无法”链接，引用“通常”是当指针处理的，按引用传递整型 static const 成员变量通常要求其加以定义。仿佛我知道一些事情，但是不是很想一吐为快。

好吧，确有其事。依据标准，按引用传递 MinVals 时要求 MinVals 有定义，但并不是所有实现都服从了这个需求。因此，你可能会发现有时是能够完美转发未加定义的 static const 成员变量的，这取决于具体的编译器和链接器。如果真是这样，恭喜，不过没有理由期望这样的代码能够移植。若想添加可移植性，只需 static const 成员变量提供定义即可。对于 MinVals，定义如下：

```
const std::size_t Widget::MinVals;      // 在 Widget 的 .cpp 文件中
```

注意，定义语句没有重复指定初始化物（对于本例中的 MinVals，就是值 28），不过，该细节不用死记硬背。如果忘记了这一点，在两处都提供了初始化物，你的编译器肯定会发出控诉，从而提醒只在一处指定即可。[译注 3]

重载的函数名字和模板名字

假设 f（我们一直变着法子经由 fwd 转发各种东西的目标函数）想通过传入一个执行部分操作函数来自定义其行为。假定该函数接受并返回的型别 int，那么 f 可以声明如下：

```
void f(int (*pf)(int));          // pf 是 "processing function" 的简称
```

值得一提的是，f 也可以使用平凡的非指针语法来声明。这样的声明长成下面这样，尽管它与上面的声明含义相同：

```
void f(int pf(int));             // 声明与上面含义相同的 f
```

无论哪种方式声明都可以吧，再假设又有重载函数 processVal：

```
int processVal(int value);
int processVal(int value, int priority);
```

译注 3：此所谓唯一定义规则（one definition rule，ODR）。

processVal 就可以传递给 f，

```
f(processVal);        // 没问题
```

这样做居然没问题，有点意外吧。f 要求的实参是个指涉到函数的指针，可是 processVal 既非函数指针，甚至连函数都不是，它是两个不同函数的名字。无论如何，编译器还是知道它们需要的是哪个 processVal：匹配 f 形参型别的那个。总之，编译器会选择接受一个 int 那个版本的 processVal，尔后把那个函数地址传给 f。

这里之所以能够运作，就在于 f 的声明式使得编译器弄清楚了哪个版本的 processVal 是所要求的。fwd 就不行了，因为作为一个函数模板，它没有任何关于型别需求的信息，这也使得编译器不可能决议应该传递哪个函数重载版本：

```
fwd(processVal);      // 错误！哪个 processVal 重载版本？
```

光秃秃的 processVal 并无型别。没有型别，型别推导无从谈起；而没有型别推导，我们唯一还有拥有的就是另一种完美转发失败情形。

同一个问题，会出现在使用函数模板来代替（或附加于）重载函数名字的场合。函数模板不是只代表一个函数，而是代表着许许多多函数：

```
template<typename T>
T workOnVal(T param)          // 处理值的模板
{ … }

fwd(workOnVal);               // 错误！ workOnVal 的哪个实例？
```

欲让像 fwd 这种实施完美转发的函数接受重载函数名字或者模板名字，只有手动指定需要转发的那个重载版本或者实例。例如，可以创建一个与 f 的形参同一型别的函数指针，尔后用 processVal 和 workOnVal 初始化那个指针（这可以使得适当的 processVal 重载版本得以选择或适当的 workOnValue 实例得以生成），再后将指针传递给 fwd：

```
using ProcessFuncType =                          // 相当于创建一个 typede；
    int (*)(int);                                // 参见条款 9

ProcessFuncType processValPtr = processVal;      // 指定了需要的 processVal 签名

fwd(processValPtr);                              // 没问题

fwd(static_cast<ProcessFuncType>(workOnVal));    // 也没问题
```

当然，这要求你知道 fwd 转发的函数指针型别到底应该是什么。完美转发函数一般来说不会在文档中写明这个信息。毕竟，完美转发函数是被设计用来接受任何型别的，但这么一来，没有文档告知你要传递的型别，那你又如何知道呢？

位域

最后一种完美转发失败情形，是位域被用作函数实参。为考察在实践中如何表现，观察如下这个可以表示 IPv4 头部的模型：[注3]

```
struct IPv4Header {
  std::uint32_t version:4,
                IHL:4,
                DSCP:6,
                ECN:2,
                totalLength:16;
  ...
};
```

如果我们被虐了千百遍的函数 f（转发函数 fwd 万年不变的目标）的声明式中接受 std::size_t 型别的形参，然后用 IPv4Header 对象的，比如说，totalLength 字段来调用 f 吧，编译器会乖乖放行：

```
void f(std::size_t sz);        // 待调用的函数

IPv4Header h;
...
f(h.totalLength);              // 没问题
```

但是，如果是经由 fwd 把 h.totalLength 转发给 f，就是另一回事了：

```
fwd(h.totalLength);            // 错误!
```

问题在于 fwd 的形参是个引用，而 h.totalLength 是个非 const 的位域。乍听之下，这也没什么。但是 C++ 标准却对于这么个组合以异乎寻常的口吻严加禁止："非 const 引用不得绑定到位域"，该条禁令倒是有极其充分的理由。位域是由机器字的若干任意部分组成的（例如，32 位 int 的第 3 到第 5 个比特），但是这样的实体是不可能有办法对其直接取址的。我前面曾经提及，在硬件层次，引用和指针本是同一事物。这么一来，既然没有办法创建指涉到任意比特的指针（C++ 硬性规定，可以指涉的最小实体是单个 char），那自然也就没有办法把引用绑定到任意比特了。

要将完美转发位域的不可能化为可能，也简单不过。一旦你意识到接受位域实参的任何函数都实际上只会收到位域值的副本。毕竟，没有函数可以把位域绑定到引用，也不可能有函数接受指涉到位域的指针，因为根本不存在指涉到位域的指针。可以传递位域的仅有的形参种类就只有按值传递，以及，有点匪夷所思的常量引用（reference-

注3：这里假定了位域的内存布局方式是从最低有效位向最高有效位。C++ 并不保证一定如此布局，但编译器常常会给软件工程师提供某种机制以控制位域的内存布局。

to-const)。在按值传递的形参这种情况下，被调用的函数显然收到是位域内的值的副本；而在常量引用形形参这种情况下，标准要求这时引用实际绑定到存储在某种标准整型（例如 int）中的位域值的副本。常量引用不可能绑定到位域，它们绑定到的是“常规”对象，其中复制了位域的值。

这么一来，把位域传递给完美转发函数的关键，就是利用转发目的函数接收的总是位域值的副本这一事实。你可以自己制作一个副本，并以该副本调用转发函数。例如，在 IPv4Header 一例中，下述代码即演示了该技巧：

```
// 复制位域值，初始化形式参见条款 6
auto length = static_cast<std::uint16_t>(h.totalLength);

fwd(length);                    // 转发该副本
```

结语

在绝大多数情形下，完美转发就如规定所言的方式运作如仪。你很少需要特别留意什么。但当它无法运作时，亦即，当一些看上去合理的代码编译失败，或者更讨厌的情况，可以通过编译，行为却表现得和预料不同，重要的是要了解完美转发的不完美之所在，同样重要的是知道如何规避它们。在绝大多数情形下，这些规避手法都是直截了当的。

要点速记

- 完美转发的失败情形，是源于模板型别推导失败，或推导结果是错误的型别。
- 会导致完美转发失败的实参种类有大括号初始化物、以值 0 或 NULL 表达的空指针、仅有声明的整型 static const 成员变量、模板或重载的函数名字，以及位域。

第 6 章

lambda 表达式

lambda 表达式（简称 lambda 式）能成为 C++ 中极具颠覆性的语言特性，不免让人感觉有些不可思议，因为它们并没有给语言注入新的表达力。任何 lambda 式能做到的，你都能手工做到，无非要费力多打几个字。但是 lambda 式作为一种创建函数对象的手段，实在太过方便，所以才会对 C++ 日常软件开发产生极大的影响。如果不是有了 lambda 式，STL 中的“_if”族算法（例如，std::find_of、std::remove_if 和 std::count_if 等）恐怕只会使用最平凡的谓词来调用，但有了 lambda 式以后，使用复杂谓词来调用这些算法的应用便如雨后春笋般爆发了。这种情况同样发生在能够自定义比较函数的算法族（例如，std::sort、std::nth_element 和 std:;lower_bound 等）上。在 STL 之外，lambda 式也能够用来为 std::sort 和 std::sort（参见条款 18 和条款 19）快速创建自定义析构器，还能以同样直接的程度对谓词做特化处理来提供条件变量给线程 API（参见条款 39）。在标准库之外，lambda 式可以临时制作出回调函数、接口适配函数或是语境相关函数的特化版本以供一次性调用。有了 lambda 式的 C++ 语言，确实变得更富有亲和力了。

围绕着 lambda 式的术语词汇可能比较让人迷惑。这里给出一些提醒：

- lambda 表达式，顾名思义，是表达式的一种。它是源代码组成部分，比如在下面这段代码中高亮部分就是 lambda 式：

```
std::find_if(container.begin(),  container.end(),
             [](int val) { return 0 < val && val < 10; });
```

- 闭包是 lambda 式创建的运行期对象，根据不同的捕获模式，闭包会持有数据的副本或引用。在上面这个对 std::find_if 的调用中，闭包就是作为第三个实参在运行期传递给 std::find_if 的对象。

- 闭包类就是实例化闭包的类。每个 lambda 式都会触发编译器生成一个独一无二的闭包类。而闭包中的语句会变成它的闭包类成员函数的可执行指令。

lambda 式常用于创建闭包并仅将其用作传递给函数的实参。上述对 std::find_if 的调用就符合这个情形。不过，一般而言，闭包可以复制。所以，对应于单独一个 lambda 式的闭包型别可以有多个闭包。例如，在下面的代码中：

```
{
  int x;                          // x 是个局部变量
  ...

  auto c1 =                       // c1 是 lambda 产生的闭包的副本
    [x](int y) { return x * y > 55; };

  auto c2 = c1;                   // c2 是 c1 的副本

  auto c3 = c2;                   // c3 是 c2 的副本

  ...

}
```

c1、c2 和 c3 都是同一 lambda 式产生的闭包的副本。

在非正式场合，lambda 式、闭包和闭包类之间的界线大可以模糊一些。但在本章接下去的条款中搞清楚上面这几个概念哪些存在于编译期（lambda 式和闭包类），哪些存在于运行期（闭包），以及它们彼此之间有何联系，则常常是重要的。

条款 31：避免默认捕获模式

C++11 中有两种默认捕获模式：按引用或按值。按引用的默认捕获模式可能导致空悬引用，按值的默认捕获模式会忽悠你，好像可以对空悬引用免疫（其实并没有），你，让你认为你的闭包是独立的（事实上它们可能不是独立的）。

这些就是本条款的纲领性内容了。但如果你本性上更偏向于工程师而不是领导，你就会不仅要一个骨架，还得有血有肉。所以我们就从默认捕获模式的危害说起吧。

按引用捕获会导致闭包包含指涉到局部变量的引用，或者指涉到定义 lambda 式的作用域内的形参的引用。一旦由 lambda 式所创建的闭包越过了该局部变量或形参的生命期，那么闭包内的引用就会空悬。例如，我们有一个元素为筛选函数的容量，其中每个筛选函数都接受一个 int，并返回一个 bool 以表示传入的值是否满足筛选条件：

```
using FilterContainer =                          // 关于“using”，参见条款 9
  std::vector<std::function<bool(int)>>;         // 关于 std::function，参见条款 2

FilterContainer filters;                         // 元素为筛选函数的容器
```

我们可以像下面这样添加一个筛选 5 的倍数的函数：

```
filters.emplace_back(                    // 欲知 emplace_back 的详情，参见条款 42
    [](int value) { return value % 5 == 0; }
);
```

但是，我们可能需要在运行期计算出除数，而不是把硬编码的“5”写入 lambda 式中。所以，添加筛选器的代码多少可能与下面的代码相似：

```
void addDivisorFilter()
{
  auto calc1 = computeSomeValue1();
  auto calc2 = computeSomeValue2();

  auto divisor = computeDivisor(calc1, calc2);

  filters.emplace_back(                                    // 危险！
    [&](int value) { return value % divisor == 0; }       // 对 divisor
  );                                                       // 的指涉可能空悬！
}
```

这段代码随时会出错。lambda 式是指涉到局部变量 divisor 的引用，但该变量的在 addDivisorFilter 返回时即不再存在。换言之，该变量的销毁就是紧接着 filters.emplace_back 返回的那一时刻。所以这就等于说，添加到筛选器聚集的那个函数刚刚被添加完就消亡了。使用这个筛选器，从它刚被创建的那一刻起，就会产生未定义行为。

就算不这样做，换作以显式方式按引用捕获 divisor，问题依旧：

```
filters.emplace_back(
  [&divisor](int value)                  // 危险！
  { return value % divisor == 0; }       // 对 divisor
);                                       // 的指涉仍然可能空悬！
```

不过，通过显式捕获，确实较容易看出 lambda 式的生存依赖于 divisor 的生命期。而且，明白地写出名字“divisor”还提醒了我们，再次确认了 divisor 的至少和该 lambda 式的闭包具有一样长的生命期。比起“[&]”所传达的这种不痛不痒的“要保证没有空悬哟”式的劝告，显式指名更让人印象深刻。

如果你知道闭包会被立即使用（例如，传递给 STL 算法）并且不会被复制，那么引用比它持有的局部变量或形参生命期更长，就不存在风险。你可能会争论说，这样的情况下，既然没有引用空悬风险，也就没有理由要避免使用默认引用捕获模式。例如，

我们的筛选器 lambda 式仅用作 C++11 的 `std::all_of` 的实参，后者的作用是返回某作用域内的元素是否都满足某条件的判断：

```
template<typename C>
void workWithContainer(const C& container)
{
  auto calc1 = computeSomeValue1();              // 同上
  auto calc2 = computeSomeValue2();              // 同上

  auto divisor = computeDivisor(calc1, calc2); // 同上

  using ContElemT = typename C::value_type;      // 为实现泛型算法
                                                 // 取得容器中的元素型别，参见条款 13

  using std::begin;
  using std::end;

  if (std::all_of(                               // 如果所有
        begin(container), end(container),        // 容器中的元素值
        [&](const ContElemT& value)              // 都是 divisor
        { return value % divisor == 0; })        // 的倍数
     ) {
    ...                                          // 若全是，执行这里
  } else {
    ...                                          // 若至少有一个不是，执行这里
  }
}
```

不错，这样使用的确安全，但是这样的安全可谓朝不保夕。如果发现该 lambda 式在其他语境中有用（例如，加入到 `filters` 容器中成为一个函数元素），然后被复制并粘贴到其他闭包比 `divisor` 生命期更长的语境中的话，你就又被拖回空悬的困境了。这一回，在捕获语句中，可没有任何让你对 `divisor` 进行生命期分析的提示之物了。

从长远观点来看，显式地列出 lambda 式所依赖的局部变量或形参是更好的软件工程实践。

顺便说下，C++14 提供了在的 lambda 式的形参声明中使用 `auto` 的能力，这意味着上面的代码在 C++14 中可以简化，`ContElemT` 的声明可以删去，而 `if` 条件可以更改如下：

```
if (std::all_of(begin(container), end(container),
                [&](const auto& value)                    // C++14
                { return value % divisor == 0; }))
```

解决这个问题的一种办法是对 `divisor` 采用按值的默认捕获模式。即，我们这样向容器添加 lambda 式：

```
filters.emplace_back(
  [=](int value) { return value % divisor == 0; }        // 现在 divisor
);                                                       // 不会空悬
```

对于本例而言，这样做已经足够。但是，总的来说，按值的默认捕获并非你想像中能够避免空悬的灵丹妙药。问题在于，按值捕获了一个指针以后，在lambda式创建的闭包中持有的是这个指针的副本，但你并无办法阻止lambda式之外的代码去针对该指针实施delete操作所导致的指针副本空悬。

“这种事根本不会发生！”你抗议道，“我已经看完了第4章，智能指针是我的崇拜！只有没前途的C++98程序员才会使用裸指针和delete。”可能的确如此，但你很难脱离干系，因为事实上，有时你真的会使用裸指针，还有的时候，它们会在你眼皮底下实施delete操作。只不过现代C++编程风格中，在源代码中经常难觅其迹。

假设Widget类可以实施的一个操作是向筛选器容器中添加条目：

```
class Widget {
public:
  …                                 // 构造函数等
  void addFilter() const;           // 向filters添加一个条目

private:
  int divisor;                      // 用于Widget的filter元素
};
```

Widget::addFilter可能作如下定义：

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
  );
}
```

在满心欢喜的外行看来，这像是安全的代码。虽然该lambda式对divisor有依赖，但按值的默认捕获模式会确保divisor被复制到该lambda式创建的任何闭包里，对吗？

错。错得彻底。错得离谱。错得无可救药。

捕获只能针对于在创建lambda式的作用域内可见的非静态局部变量（包括形参）。在Widget::addFilter的函数体内，divisor并非局部变量，而是Widget类的成员变量。它压根无法被捕获。这么一来，如果默认捕获模式被消除，代码就不会通过编译：

```
void Widget::addFilter() const
{
  filters.emplace_back(                                  // 错误!
    [](int value) { return value % divisor == 0; }      // 没有可捕获的divisor
  );
}
```

而且，如果试图显式捕获 divisor（无论按值还是按引用，这无关紧要），这个捕获语句都不能通过编译，因为 divisor 既不是局部变量，也不是形参：

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [divisor](int value)                // 错误!
    { return value % divisor == 0; }    // 局部没有可捕获的 divisor
  );
}
```

所以如果在按值的默认捕获语句中捕获的并非 divisor，并且如果这句按值的默认捕获语句不存在，代码不能编译。那么到底实际发生了什么呢？

要解释这一现象，关键在于一个裸指针隐式应用，这就是 this。每一个非静态成员函数都持有一个 this 指针，然后每当提及该类的成员变量时都会用到这个指针。例如，在 Widget 的任何成员函数中，编译器内部都会把 divisor 替换成 this->divisor。在 Widget::addFiliter 的按值默认捕获版本中，

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
  );
}
```

被捕获的实际上是 Widget 的 this 指针，而不是 divisor。从编译器视角来看，上述代码相当于：

```
void Widget::addFilter() const
{
  auto currentObjectPtr = this;

  filters.emplace_back(
    [currentObjectPtr](int value)
    { return value % currentObjectPtr->divisor == 0; }
  );
}
```

理解了这一点，也就相当于理解了 lambda 闭包的存活与它含有其 this 指针副本的 Widget 对象的生命期是绑在一起的。特别地，考虑下面的代码，它掌握了第 4 章精髓，仅使用了智能指针：

```
using FilterContainer =                          // 同前
  std::vector<std::function<bool(int)>>;

FilterContainer filters;                         // as before
```

```
void doSomeWork()
{
  auto pw =                                  // 创建 Widget，关于
    std::make_unique<Widget>();              // std::make_unique，
                                             // 参见条款 21

  pw->addFilter();                           // 添加使用了 Widget::divisor 的筛选函数

  ...
}                                            // Widget 被销毁，filters 现在持有空悬指针
```

当调用 doSomeWork 时创建了一个筛选函数，它依赖于 std::make_unique 创建的 Widget 对象，即，一个含有指向 Widget 指针（Widget 的 this 指针的副本）的筛选函数。该函数被添加到 filters 中，不过当 doSomeWork 执行结束之后，Widget 对象即被管理着它的生命期的 std::unique_ptr 销毁（参见条款 18）。从那一刻起，filters 中就含有了一个带有空悬指针的元素。

这一特定问题可以通过将你想捕获的成员变量复制到局部变量中，尔后捕获该局部副本加以解决：

```
void Widget::addFilter() const
{
  auto divisorCopy = divisor;                // 复制成员变量

  filters.emplace_back(
    [divisorCopy](int value)                 // 捕获副本
    { return value % divisorCopy == 0; }     // 使用副本
  );
}
```

实话实说，如果你采用这种方法，那么按值的默认捕获也能够运作：

```
void Widget::addFilter() const
{
  auto divisorCopy = divisor;                // 复制成员变量

  filters.emplace_back(
    [=](int value)                           // 捕获副本
    { return value % divisorCopy == 0; }     // 使用副本
  );
};
```

但是为何要冒此不必要的风险？按值的默认捕获才是最开始造成意外地捕获了 this 指针，而不是期望中的 divisor 的始作俑者。

在 C++14 中，捕获成员变量的一种更好的方法是使用广义 lambda 捕获（generalized lambda capture，参见条款 32）：

```
void Widget::addFilter() const
{
  filters.emplace_back(                  // C++14:
    [divisor = divisor](int value)       // 将 divisor 复制入闭包
    { return value % divisor == 0; }     // 使用副本
  );
}
```

对广义 lambda 捕获而言，没有默认捕获模式一说。但是，就算在 C++14 中，本条款的建议（避免使用默认捕获模式）依然成立。

使用默认值捕获模式的另外一个缺点是，在于它似乎表明闭包是自洽的，与闭包外的数据变化绝缘。作为一般的结论，这是不正确的。因为 lambda 式可能不仅依赖于局部变量和形参（它们可以被捕获），它们还会依赖于静态存储期（static storage duration）对象。这样的对象定义在全局或名字空间作用域中，又或在类中、在函数中、在文件中以 static 饰词声明。这样的对象可以在 lambda 内使用，但是它们不能被捕获。但如果使用了默认值捕获模式，这些对象就会给人以错觉，认为它们可以加以捕获。思考下面这个前面见过的 addDivisorFilter 函数的修改版：

```
void addDivisorFilter()
{
  static auto calc1 = computeSomeValue1();      // 现在以 static 饰词声明
  static auto calc2 = computeSomeValue2();      // 现在以 static 饰词声明

  static auto divisor =                         // 现在以 static 饰词声明
    computeDivisor(calc1, calc2);

  filters.emplace_back(
    [=](int value)                              // 未捕获任何东西!
    { return value % divisor == 0; }            // 指涉到前述以 static 饰词声明的对象
  );

  ++divisor;                                    // 意外修改了 divisor
}
```

一目十行的读者在看到代码中有着“[=]”后，就会想当然地认为，“很好，lambda 式复制了它内部使用的对象，因此 lambda 式是自洽的。”这无可厚非，但该 lambda 式实在并不独立，因为它没有使用任何的非静态局部变量和形参，所以它没能捕获任何东西。更糟糕的是 lambda 式的代码中指涉了静态变量 divisor。因而，每次调用 addDivisorFilter 的最后 divisor 都会被递增，从而在把好多个 lambda 式添加到 filiters 时每个 lambda 式的行为都不一样（对应于 divisor 的新值）。从实际效果来说，这个 lambda 式实现的效果是按引用捕获 divisor，和按值默认捕获所暗示的含义有着直接的矛盾。如果从一开始就远离按值的默认捕获模式，也就能消除代码被如此误读的风险了。

要点速记

- 按引用的默认捕获会导致空悬指针问题。
- 按值的默认捕获极易受空悬指针影响（尤其是this），并会误导人们认为lambda式是自洽的。

条款32：使用初始化捕获将对象移入闭包

有时，按值的捕获和按引用的捕获皆非你所欲。如果你想要把一个只移对象（例如，std::unique_ptr或std::future型别的对象）放入闭包，C++11未提供任何办法做到此事。如果你有个对象，其复制操作开销昂贵，而移动操作成本低廉（例如，大部分标准库容器），而你又需要把该对象放入闭包，那么你肯定更愿意移动该对象，而非复制它。但是，C++11中也还是没有让你实现这一点的途径。

但那只是C++11，C++14则有云泥之别。它为对象移动入闭包提供了直接支持。如果你的编译器兼容C++14，则只需欢呼雀跃尔后继续阅读本书即可。但如果你还在使用C++11编译器，则你仍可欢呼尔后继续阅读本文，因为C++11中有近似达成移动捕获行为的做法。

移动捕获的缺失即使在C++11标准被刚接受时，也被视为一种缺憾。最直接的补救措施本是在C++14中添加这一特性，但标准委员会却另辟蹊径。委员们提出了一种全新的捕获机制，它是如此灵活，按移动的捕获只不过属于该机制能够实现的多种效果之一罢了。这种新能力称为初始化捕获（init capture）。实际上，它可以做到C++11的捕获形式能够做到的所有事情，而且还不止如此。初始化捕获不能表示者，则是默认捕获模式，但是条款31解释过，这是你无论何时都应该远离的一种模式（对于将C++11捕获可以实现的情况，若用初始化捕获语法则会稍显啰嗦，所以若使用C++11捕获已能解决问题，则大可以使用之）。

使用初始化捕获，则你会得到机会指定：

1. 由lambda生成的闭包类中的成员变量的名字。
2. 一个表达式，用以初始化该成员变量。

以下是如何使用初始化捕获将std::unique_ptr移动到闭包内：

```
class Widget {                                    // 一些有用的型别
public:
  ...

  bool isValidated() const;
  bool isProcessed() const;
  bool isArchived() const;

private:
  ...
};
auto pw = std::make_unique<Widget>();             // 创建 Widget
                                                  // 关于 std::make_unique，参见条款 21
...                                               // 配置 *pw

auto func = [pw = std::move(pw)]                  // 采用 std::move(pw)
            { return pw->isValidated()            // 初始化闭包类的数据成员
                     && pw->isArchived(); };
```

突出标示的那段代码就是初始化捕获，位于“=”左侧的，是你所指定的闭包类成员变量的名字，而位于其右侧的则是其初始化表达式。可圈可点之处在于，“=”的左右两侧处于不同的作用域。左侧作用域就是闭包类的作用域，而右侧的作用域则与lambda式加以定义之处的作用域相同。在上述例子中，“=”左侧的名字pw指涉的是闭包类的成员变量，而右侧的名字pw指涉的则在是在lambda式上面一行声明的对象，即经由调用make_unique所初始化的对象。所以，“pw=std::move(pw)”表达了“在闭包中创建一个成员变量pw，然后使用针对局部变量pw实施std::move的结果来初始化该成员变量。”

和平常一样，lambda式体内代码的作用域位于闭包类内，所以在那里用到的pw指涉的也是闭包类的成员变量。

该例中“配置*pw”这条注释表明，在Widget经由std::make_unique创建之后，并在指涉到该Widget的std::unique_ptr被lambda式捕获之前，该Widget会在某些方面加以修改。如果这样的配置并非必要动作，即，经由std::make_unique创建的Widget对象已具备被lambda式捕获的合适状态，则作为局部变量pw就亦非必要，因为闭包类成员变量可以径由std::make_unique实施初始化：

```
auto func = [pw = std::make_unique<Widget>()]     // 径以 make_unique 的调用结果
            { return pw->isValidated()            // 初始化闭包类的成员
                     && pw->isArchived(); };
```

这里，应该已经昭明，C++11的“捕获”概念在C++14中得到了显著的泛化，因为在C++11中不可能捕获一个表达式的结果。因此，初始化捕获还有另一美名，称为广义lambda捕获（generalized lambda capture）。

但如果你使用的编译器缺少对 C++14 初始化捕获的支持，又将如何是好？在不支持按移动捕获的语言中，又该如何实现按移动捕获呢？

回想一下，一个 lambda 表达式不过是生成一个类并且创建一个该类的对象的手法罢了。并不存在 lambda 能做，而你手工做不到的事情。以上面所见的 C++14 示例代码为例，就可以如下用 C++11 写作：

```
class IsValAndArch {                              // 表示“已校验并已归档”
public:
  using DataType = std::unique_ptr<Widget>;

  explicit IsValAndArch(DataType&& ptr)           // 条款 25 解释过
  : pw(std::move(ptr)) {}                         // std::move 的用法

  bool operator()() const
  { return pw->isValidated() && pw->isArchived(); }

private:
  DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

这比起撰写一个 lambda 式长了不少，但是它并未改变一个事实，即在 C++11 中，如果你想要一个支持对成员变量实施移动初始化的类，那么也只需多花些时间敲键盘，就能达到目的。

如果你非想要用 lambda 式（考虑到它们的便利性，你极有可能如此），按移动捕获在 C++11 中可以采用以下方法模拟，只需要：

1. **把需要捕获的对象移动到 std::bind 产生的函数对象中。**

2. **给到 lambda 式一个指涉到欲“捕获”的对象的引用。**

如果你本来就熟悉 `std::bind`，代码就显得十分直截了当。如果你还不熟悉 `std::bind`，则代码需要一些时间来习惯，但这些投入是值得的。

假如你想要创建一个局部的 `std::vector` 对象，向其放入适合的一组值，然后将其移入闭包。在 C++14，这是举手之劳：

```
std::vector<double> data;                 // 欲移入闭包的对象

...                                       // 灌入数据

auto func = [data = std::move(data)]      // C++14 的初始化捕获
            { /* 对数据加以运用 */ };
```

代码的关键部分已加以突显：欲移动的对象的型别（std::vector<double>）和该对象的名字（data），还为初始化捕获而准备的初始化表达式（std::move(data)）。使用 C++11 撰写的等价代码如下，我也把同样的关键部分加以突显：

```
std::vector<double> data;                          // 同前

...                                                // 同前

auto func =
  std::bind(                                       // C++11 中
    [](const std::vector<double>& data)            // 模拟初始化捕获的部分
    { /* 对数据加以运用 */ },
    std::move(data)
);
```

和 lambda 表达式类似地，std::bind 也生成函数对象。我称 std::bind 返回的函数对象为绑定对象（bind object）。std::bind 的第一个实参是个可调用对象，接下来的所有实参表示传给该对象的值。

绑定对象含有传递给 std::bind 所有实参的副本。对于每个左值实参，在绑定对象内的对应的对象内对其实施的是复制构造；而对于每个右值实参，实施的则是移动构造。在这个例子中，第二个实参是个右值（即 std::move 的结果，参见条款 23），所以 data 在绑定对象中实施的是移动构造。而该移动构造动作正是实现模拟移动捕获的核心所在，因为把右值移入绑定对象，正是绕过 C++11 无法将右值到闭包的手法。

当一个绑定对象被“调用”（即，其函数调用运算符被唤起）时，它所存储的实参会传递给原先传递给 std::bind 的那个可调用对象。在本例中，也就是当 func（绑定对象）被调用时，func 内经由移动构造出所得到的 data 的副本就会作为实参传递给那个原先传递给 std::bind 的 lambda 式。

这个 lambda 和 C++14 版本的 lambda 长得一样，但是多加了一个形参 data，它对应于我们的伪移动捕获对象。该形参是个指涉到绑定对象内的 data 副本的左值引用（而不是右值引用，因为虽然初始化 data 副本的表达式是 std::move(data)，但 data 的副本本身是一个左值）。这么一来，在 lambda 内对 data 做的操作，都会实施在绑定对象内移动构造而得的 data 的副本之上。

默认情况下，lambda 生成的闭包类中的 operator() 成员函数会带有 const 饰词。结果，闭包里的所有成员变量在 lambda 式的函数体内都会带有 const 饰词。但是，绑定对象里移动构造得到的 data 副本却并不带有 const 饰词。所以，为了防止该 data 的副本在 lambda 式内被意外修改，lambda 的形参就声明为常量引用。但如果 lambda 式的

声明带有 mutable 饰词，闭包里的 operator() 函数就不会在声明时带有 const 饰词，相应的适当做法，就是在 lambda 声明中略去 const：

```
auto func =
  std::bind(                                        // C++11 中针对可变 lambda 式
    [](std::vector<double>& data) mutable           // 模拟初始化捕获的部分
    { /* 对数据加以运用 */ },
    std::move(data)
  );
```

因为绑定对象存储着传递给 std::bind 所有实参的副本，在本例中的绑定对象就包含一份由作为 std::bind 的第一个实参的 lambda 式产生的闭包的副本。这么一来，该闭包的生命期和绑定对象就是相同的。这一点很重要，因为这意味着只要闭包还存在，则绑定对象内的伪移动捕获对象也存在。

如果这是你和 std::bind 的首次接触，那么在深究前面讨论的细节之前，你可能需要先垂询最喜欢 C++11 参考书。即使如此，现在你至少也已经弄清楚了以下基础知识：

- 移动构造一个对象入 C++11 闭包是不可能实现的，但移动构造一个对象入绑定对象则是可能实现的。
- 欲在 C++11 中模拟移动捕获包括以下步骤：先移动构造一个对象入绑定对象，然后按引用把该移动构造所得的对象传递给 lambda 式。
- 因为绑定对象的生命期和闭包相同，所以针对绑定对象中的对象和闭包里的对象可以采用同样手法加以处置。

关于使用 std::bind 模拟移动捕获，再举一例。下面是我们前面看过的，使用 C++14 在闭包内创建 std::unique_ptr 的代码：

```
auto func = [pw = std::make_unique<Widget>()]          // 同前，
            { return pw->isValidated()                 // 在闭包内创建 pw
                     && pw->isArchived(); };
```

以下是使用 C++11 撰写的模拟代码：

```
auto func = std::bind(
              [](const std::unique_ptr<Widget>& pw)
              { return pw->isValidated()
                       && pw->isArchived(); },
                std::make_unique<Widget>()
              );
```

我在这里展示的是如何使用 std::bind 来绕开 C++11 中 lambda 式语法的限制，这不免有些讽刺，因为在条款 34 中，我又在提倡优先选用 lambda 式，而非 std::bind。不过，

那个条款也解释说，C++11 中有一些情况下，std::bind 可能会是有用的，而以上就是这些情况之一（在 C++14 中，像初始化捕获和 auto 形参这些特性可以消除那些情况）。

要点速记

- 使用 C++14 的初始化捕获将对象移入闭包。
- 在 C++11 中，经由手工实现的类或 std::bind 去模拟初始化捕获。

条款 33：对 auto&& 型别的形参使用 decltype，以 std::forward 之

泛型 lambda 式（generic lambda）是 C++14 最振奋人心的特性之一——lambda 可以在形参规格中使用 auto。这个特性的实现十分直截了当：闭包类中的 operator() 采用模板实现。例如，给定下述 lambda 式：

```
auto f = [](auto x){ return func(normalize(x)); };
```

则闭包类的函数调用运算符如下所示：

```
class SomeCompilerGeneratedClassName {
public:
  template<typename T>                    // auto 型别的返回值
  auto operator()(T x) const              // 参见条款 3
  { return func(normalize(x)); }

  ...                                     // 闭包类的其他功能
};
```

在本例中，lambda 式对 x 实施的唯一动作就是将其转发给 normalize。如果 normalize 区别对待左值和右值，则可说该 lambda 式撰写的是有问题的，因为，lambda 总会传递左值（形参 x）给 normalize，即使传递给 lambda 式的实参是个右值。

该 lambda 式的正确撰写方式是把 x 完美转发给 normalize，这就要求在代码中修改两处。首先，x 要改成万能引用（参见条款 24）；其次，使用 std::forward（参见条款 25）把 x 转发给 normalize。概念上不难理解，这两处的修改都是举手之劳：

```
auto f = [](auto&& x)
         { return func(normalize(std::forward<???>(x))); };
```

遗憾的是，在概念和现实之间，横亘着一个问题，即，传递给 std::forward 的形参应该是何型别，也就是我在上面写着 ??? 的地方应该如何落实。

通常情况下，在使用完美转发时，你是在一个接受型别形参 T 的模板函数中，所以你写 std::forward<T> 就好。而在泛型 lambda 式中，却没有可用的型别形参 T。在 lambda 式生成的闭包内的模板化 operator() 函数中的确有个 T，但是在 lambda 式中无法指涉之，所以有也没用。

条款 28 解释过，如果把左值传递给万能引用的形参，则该形参的型别会成为左值引用；而如果把传递的是右值，则该形参会成为右值引用。那意味着在我们的 lambda 式中，我们可以通过探查 x 的型别，来判断传入的实参是左值还是右值。decltype 提供了实现这一点的一种途径（参见条款 3）。如果传入的是个左值，decltype(x) 将会产生左值引用型别；如果传入的是个右值，decltype(x) 将会产生右值引用型别。

条款 28 还解释了，使用 std::forward 时惯例是：用型别形参为左值引用表明想要返回左值，而用非引用型别时来表明想要返回的右值。再看我们的 lambda 式，如果 x 绑定了左值，decltype(x) 将产生左值引用型别。这符合惯例。不过，如果 x 绑定的是个右值，decltype(x) 将会产生右值引用惯例，而非符合惯例的非引用。

但是，再看下条款 28 中 std::forward 的 C++14 实现：

```
template<typename T>                                    // 在名字空间 std 中
T&& forward(remove_reference_t<T>& param)
{
  return static_cast<T&&>(param);
}
```

如果客户代码欲完美转发 Widget 型别的右值，按惯例它应该采用 Widget 型别（即非引用型别）来实例化 std::forward，然后 std::forward 模板会产生如下函数：

```
Widget&& forward(Widget& param)          // T 取值 Widget 时
{                                        // std::forward 的实例化结果
  return static_cast<Widget&&>(param);
}
```

但是，如果用户代码想要完美转发 Widget 的同一右值，但是这次没有遵从惯例将 T 指定为非引用型别，而是将 T 指定为右值引用，这会导致什么结果？这就是需要思考的问题，T 指定为 Widget&& 将会发生什么事情。在 std::forward 完成初步的实例化并实施了 std::remove_reference_t 之后，但在引用折叠（再参见条款 28）发生之前，std::forward 如下所示：

```
Widget&& && forward(Widget& param)                    // T取值Widget&&时
{                                                     // std::forward的实例化结果
  return static_cast<Widget&& &&>(param);             // (在引用折叠发生之前)
}
```

然后，应用引用折叠规则，右值引用的右值引用结果是单个右值引用，实例化结果为：

```
Widget&& forward(Widget& param)                       // T取值Widget&&时
{                                                     // std::forward的实例化结果
  return static_cast<Widget&&>(param);                // (在引用折叠发生之后)
}
```

如果你把这个实例化和在T为Widget时调用的std::forward那个实例化两相比较，你会发现它们别无二致。那就意味着，实例化std::forward时，使用一个右值引用型别和使用一个非引用型别，会产生相同结果。

这个结果非常不错，因为如果传递给我们的lambda式的形参x是个右值，decltype(x)产生的是右值引用型别。我们之前已经知道了，传递左值给我们的lambda式时，decltype(x)会产生传递给std::forward的符合惯例的型别，而现在我们又知道对于右值而言，虽然说decltype(x)产生的型别并不符合传递给std::forward的型别形参的惯例，但是产生的结果与符合惯例的型别殊途同归。所以，无论左值还是右值，把decltype(x)传递给std::forward都能给出想要的结果。是故，我们的完美转发lambda式可以编写如下：

```
auto f =
  [](auto&& param)
  {
    return
      func(normalize(std::forward<decltype(param)>(param)));
  };
```

在此基础上稍加改动，就可以得到可以接受多个形参的完美转发lambda式版本，因为C++14中的lambda能够接受可变长形参：

```
auto f =
  [](auto&&... params)
  {
    return
    func(normalize(std::forward<decltype(params)>(params)...));
  };
```

要点速记

- 对auto&&型别的形参使用decltype，以std::forward之。

条款 34：优先选用 lambda 式，而非 std::bind

std::bind 是 C++98 中 std::bind1st 和 std::bind2nd 的后继特性，但是，作为一种非标准特性而言，std::bind 在 2005 年就已经是标准库的组成部分了。正是在那时，标准委员会接受了名称 TR1 的文档，里面就包含了 std::bind 的规格（在 TR1 中，bind 位于不同的名字空间，所以是 std::tr1::bind 而非 std::bind，还有一些接口细节与现在有所不同）。这样的历史意味着，有些开发者已经有了十多年 std::bind 的开发经验。如果你是他们中的一员，那你可能不太情愿放弃这么一个运作良好的工具。这可以理解，但是对于这个特定的情况，改变是有收益的，因为在 C++11 中，相对于 std::bind，lambda 几乎总会是更好的选择。到了 C++14，lambda 不仅是优势变强，简直已成为不二之选。

该条款假设你熟悉 std::bind。如果你还不熟悉，那么在继续阅读之前，还是需要建立一个基本认识。这种认识在任何情况下都是值得的，因为你并不会知道，在哪个时刻，就会在你需要阅读或维护的代码中遭遇 std::bind。

和条款 32 一样，我称 std::bind 返回的函数对象为绑定对象。

之所以说优先选用 lambda 式，而非 std::bind，最主要原因是 lambda 式具备更高的可读性。举个例子，假设我们有个函数用来设置声音警报：

```
// 表示时刻的型别 typedef（语法参见条款 9）
using Time = std::chrono::steady_clock::time_point;

// 关于 "enum class"，参见条款 10
enum class Sound { Beep, Siren, Whistle };

// 表示时长的型别 typedef
using Duration = std::chrono::steady_clock::duration;

//  在时刻 t，发出声音 s，持续时长 d
void setAlarm(Time t, Sound s, Duration d);
```

进一步假设，在程序的某处我们想要设置在一小时之后发出警报并持续 30 秒。警报的具体声音，却尚未决定。这么一来，我们可以撰写一个 lambda 式，修改 setAlarm 的接口，这个新的接口只需指定声音即可：

```
// setSoundL（"L" 表示 "lambda"）是个函数对象，
// 它接受指定一个声音，
// 该声音将在设定后 1 小时发出，并持续 30 秒
auto setSoundL =
  [](Sound s)
  {
    // 使 std::chrono 组件不加限定饰词即可用
    using namespace std::chrono;
```

```
    setAlarm(steady_clock::now() + hours(1),     // 警报发出时刻为 1 小时后
             s,
             seconds(30));                       // 持续 30 秒
};
```

我将 lambda 式里对 setAlarm 的调用突显了出来，这是个观感无奇的函数调用，就算没有什么 lambda 经验的读者都能看得出来，传递给 lambda 的形参会作为实参传递给 setAlarm。

我们可以利用 C++14 所提供的秒（s）、毫秒（ms）时和小时（h）等标准后缀来简化上述代码，这一点建立在 C++11 中用户定义字面量这项能力的支持之上。这些后缀在 std::literals 名字空间里实现，所以上面的代码可以重写如下：

```
auto setSoundL =
  [](Sound s)
  {
    using namespace std::chrono;
    using namespace std::literals;              // 汇入 C++14 实现的后缀

    setAlarm(steady_clock::now() + 1h,          // C++14，
             s,                                 // 但和上一段代码
             30s);                              // 意义相同
};
```

下面的代码就是我们撰写对应的 std::bind 调用语句的首次尝试。这段代码里面包含一个错误，我们过会儿来修复它，修正后的代码会复杂很多。但是，即使是这个简化的版本可以让我们看到一些重要议题：

```
using namespace std::chrono;            // 同前
using namespace std::literals;

using namespace std::placeholders;      // 本句是因为需要使用 "_1"

auto setSoundB =                        // "B" 表示 "bind"
  std::bind(setAlarm,
            steady_clock::now() + 1h,           // 错误！详见下文
            _1,
            30s);
```

我也想和在 lambda 式里一样地把 setAlarm 的调用在这里突显出来，可惜这里没有调用可供我标出突显。在读这段代码时，只需了解，在调用 setSoundB 时，会使用在调用 std::bind 时指定的时刻和时长来唤起 setAlarm。固然对于初学者而言，占位符 "_1" 简直好比天书，但即使是行家也需要脑补出从占位符中数字到它在 std::bind 形参列表位置的映射关系，才能理解在调用 setSoundB 时传入的第一个实参，会作为第二个实参传递给 setAlarm。该实参的型别在 std::bind 的调用过程中未加识别，所以你还需要去咨询 setAlarm 的声明方能决定应该传递何种型别的实参给到 setSoundB。

但是，正如我前面提到的，这段代码不甚正确。在lambda式中，表达式“steady_clock::now() + 1h”是setAlarm的实参之一，这一点清清楚楚。该表达式会在setAlarm被调用的时刻评估求值。这样做合情合理：我们就是想要在setAlarm被调用的时刻之后的一个小时启动警报。但在std::bind的调用中，“steady_clock::now() + 1h”作为实参被传递给了std::bind，而非setAlarm。意味着表达式评估求值的时刻是在调用std::bind的时刻，并且求得的时间结果值会被存储在结果绑定对象中。最终导致的结果是，警报被设定的启动时刻是在调用std::bind的时刻之后的一个小时，而非调用setAlarm的时刻之后的一个小时！

欲解决这个问题，就要求知会std::bind以延迟表达式的评估求值到调用setAlarm的时刻，而实现这一点的途径，就是在原来的std::bind里嵌套第二层std::bind的调用：

```
auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<>(), steady_clock::now(), 1h),
            _1,
            30s);
```

如果你是从C++98的年代开始了解std::plus模板的，你有可能会感到一丝惊诧，因为代码中出现了在一对尖括号之间没有指定型别的写法，即，代码中有一处“std::plus<>”，而非“std::plus<type>”。在C++14中，标准运算符模板的模板型别实参大多数情况下可以省略不写，所以此处也就没有必要提供了。而C++11中则没有这样的特性，所以在C++11中，欲使用std::bind撰写与lambda式等价的代码，就只能像下面这样：

```
using namespace std::chrono;                    // 同前
using namespace std::placeholders;

auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<steady_clock::time_point>(),
                      steady_clock::now(),
                      hours(1)),
            _1,
            seconds(30));
```

如果到了这个份上，你还是看不出lambda式的实现版本更有吸引力的话，那你真的要去检查视力了。

一旦对setAlarm实施了重载，新的问题就会马上浮现。假如有个重载版本会接受第四个形参，用以指定警报的音量：

```
enum class Volume { Normal, Loud, LoudPlusPlus };
```

```
void setAlarm(Time t, Sound s, Duration d, Volume v);
```

之前那个 lambda 式会一如既往地运作如仪，因为重载决议会选择那个三形参版本的 setAlarm：

```
auto setSoundL =                                    // 同前
  [](Sound s)
  {
    using namespace std::chrono;

    setAlarm(steady_clock::now() + 1h,              // 没问题，调用的是
             s,                                     // 三形参版本的
             30s);                                  // setAlarm
};
```

不过，对 std::bind 的调用，现在可就无法通过编译了：

```
auto setSoundB =                                    // 错误！应该选择哪个
  std::bind(setAlarm,                               // setAlarm 呢？
            std::bind(std::plus<>(),
                      steady_clock::now(),
                      1h),
            _1,
            30s);
```

问题在于，编译器无法确定应该将哪个 setAlarm 版本传递给 std::bind。它拿到的所有信息就只有一个函数名，而仅函数名本身是多义的。

为使得 std::bind 的调用能够通过编译，setAlarm 必须强制转型到适当的函数指针型别：

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);

auto setSoundB =                                                    // 总算
  std::bind(static_cast<SetAlarm3ParamType>(setAlarm),              // OK 了
            std::bind(std::plus<>(),
                      steady_clock::now(),
                      1h),
            _1,
            30s);
```

但这么做，又带出来了 lambda 式和 std::bind 的另一个不同之处。在 setSoundL 的函数调用运算符中（即，lambda 式所对应的闭包类的函数调用运算符中）调用 setAlarm 采用的是常规的函数唤起方式，这么一来，编译器就可以用惯常的手法将其内联：

```
setSoundL(Sound::Siren);        // 在这里，setAlarm 的函数体大可以被内联
```

可是，std::bind 的调用传递了一个指涉到 setAlarm 的函数指针，而那就意味着在 setSoundB 的函数调用运算符中（即，绑定对象的函数调用运算符中），setAlarm 的调用是通过函数指针发生的。由于编译器不太会内联掉通过函数指针发起的函数调用，那也就意味着通过 setSoundB 调用 setAlarm 而被完全内联的几率，比起通过 setSoundL 调用 setAlarm 要低：

```
setSoundB(Sound::Siren);      // 在这里，setAlarm 的函数体被内联的可能性不大
```

综上所述，使用 lambda 式就有可能会生成比使用 std::bind 运行得更快的代码。

在 setAlarm 一例中，仅仅涉及了一个函数的调用而已。只要你想做的事情比这更复杂，使用 lambda 式的好处更会急剧扩大。例如，考虑下面这个 C++14 中的 lambda 式，它返回的是其实参是否在极小值（lowVal）和极大值（highVal）之间，而 lowVal 和 highVal 都是局部变量：

```
auto betweenL =
  [lowVal, highVal]
  (const auto& val)                                    // C++14
  { return lowVal <= val && val <= highVal; };
```

std::bind 也可以表达同样的意义，不过，要让它正常运作，必须用很晦涩的方式来构造代码：

```
using namespace std::placeholders;                     // 同前

auto betweenB =
  std::bind(std::logical_and<>(),                      // C++14
            std::bind(std::less_equal<>(), lowVal, _1),
            std::bind(std::less_equal<>(), _1, highVal));
```

如果是 C++11，还必须要指定待比较之物的型别，所以 std::bind 的调用会长成这样：

```
auto betweenB =                                  // C++11 版本
  std::bind(std::logical_and<bool>(),
            std::bind(std::less_equal<int>(), lowVal, _1),
            std::bind(std::less_equal<int>(), _1, highVal));
```

当然了，如果使用的是 C++11，就不能在 lambda 式中使用 auto 型别的形参，所以也必须固化到一个型别才行：

```
auto betweenL =                          // C++11 版本
  [lowVal, highVal]
  (int val)
  { return lowVal <= val && val <= highVal; };
```

不管是用 C++11 还是 C++14，我希望大家都能认同，lambda 式的版本不仅更加短小，还更易于理解和维护。

在前面，我曾提到，对于 std::bind 了无经验的程序员会感觉占位符（例如，_1，_2 等）看起来像天书一样。不过，可不仅仅只有占位符的行为如此佶屈聱牙。假设我们有一个函数用来制作 Widget 型别对象的压缩副本，

```
enum class CompLevel { Low, Normal, High };     // 压缩等级

Widget compress(const Widget& w,                // 制作 w 的
                CompLevel lev);                 // 压缩副本
```

然后我们想要创建一个函数对象，这样就可以指定特定的 Widget 型别对象 w 的压缩级别了。运用 std::bind，可以创建出这么一个对象：

```
Widget w;

using namespace std::placeholders;

auto compressRateB = std::bind(compress, w, _1);
```

这里，当我们把 w 传递给 std::bind 时，然后加以存储，以供未来让 compress 调用时使用。它存储的位置是在对象 compressRateB 内，但它以哪一种方式存储的：按值，还是按引用呢？这两者是泾渭分明的，因为如果 w 在对 std::bind 的调用动作与对 compressRateB 调用动作之间被修改了，如果采用按引用方式存储，那么存储起来的 w 的值也会随之修改，而如果采用按值存储，则存储起来的 w 值不会改变。

答案揭晓，w 是按值存储的。[注1] 可是，了解答案的唯一途径，就是牢记 std::bind 的工作原理。在 std::bind 调用中，答案是无迹可循的。对比之下，采用 lambda 式的途径，w 无论是按值或按引用捕获，在代码中都是显明的：

```
auto compressRateL =                    // w 是按值捕获
  [w](CompLevel lev)                    // lev 是按值传递
  { return compress(w, lev); };
```

同样显明的还有形参的传递方式。在这里，形参 lev 清清楚楚地是按值传递的。因此：

```
compressRateL(CompLevel::High);         // 实参是按值传递
```

但在 std::bind 返回的结果对象里，形参的传递方式又是什么呢？

```
compressRateB(CompLevel::High);         // 实参是采用什么方式传递的？
```

注 1：std:bind 总是复制其实参，但调用方却可以通过对某实参实施 std::ref 的手法达成按引用存储之的效果。下述语句：

```
auto compressRateB = std::bind(compress, std::ref(w), _1);
```

结果就是 compressRateB 的行为如同持有的是个指涉到 w 的引用，而非其副本。

还是那句话，欲知答案，唯一的途径是牢记 std::bind 的工作原理（答案是绑定对象的所有实参都是按引用传递的，因为此种对象的函数调用运算符利用了完美转发）。

总而言之，比起 lambda 式，使用 std::bind 的代码可读性更差、表达力更低，运行效率也可能更糟。在 C++14 中，根本没有使用 std::bind 的适当用例。而在 C++11 中，std::bind 仅在两个受限的场合还算有着使用的理由：

- **移动捕获。**C++11 的 lambda 式没有提供移动捕获特性，但可以通过结合 std::bind 和 lambda 式来模拟移动捕获。欲知详情，参见条款 32，同一条款还解释了 C++14 提供了初始化捕获的语言特性，从而消除了如此进行模拟的必要性。

- **多态函数对象。**因为绑定对象的函数调用运算符利用了完美转发，它就可以接受任何型别的实参（除了在条款 30 讲过的那些完美转发的限制情况）。这个特点在你想要绑定的对象具有一个函数调用运算符模板时，是有利用价值的。例如，给定一个类：

```
class PolyWidget {
public:
    template<typename T>
    void operator()(const T& param);
    ...
};
```

std::bind 可以采用如下方式绑定 polyWidget 型别的对象：

```
PolyWidget pw;

auto boundPW = std::bind(pw, _1);
```

这么一来，boundPW 就可以通过任意型别的实参加以调用：

```
boundPW(1930);        // 传递 int 给 PolyWidget::operator()

boundPW(nullptr);     // 传递 nullptr 给 PolyWidget::operator()

boundPW( "Rosebud" ); // 传递字符串字面量给 PolyWidget::operator()
```

在 C++11 中的 lambda 式，是无法达成上面的效果的。但是在 C++14 中，使用带有 auto 型别形参的 lambda 式就可以轻而易举地达成同样的效果：

```
auto boundPW = [pw](const auto& param)          // C++14
               { pw(param); };
```

这些都是边缘用例，而即使这些边缘用例也会转瞬即逝，因为支持 C++14 的 lambda 式的编译器已经日渐普及。

当 2005 年 bind 被非正式地加入 C++ 时，比起它在 1998 年的前身已经有了长足的进步。而 C++11 中加入的 lambda 式则使得 std::bind 相形见绌。到了 C++14 的阶段，std::bind 已经彻底失去了用武之地。

要点速记

- lambda 式比起使用 std::bind 而言，可读性更好、表达力更强，可能运行效率也更高。
- 仅在 C++11 中，std::bind 在实现移动捕获，或是绑定到具备模板化的函数调用运算符的对象的场合中，可能尚有余热可以发挥。

第 7 章

并发 API

C++11 的至伟功勋之一，就是将并发融入了语言和库中。熟悉其他线程 API（例如，pthread 或 Windows 线程库）的程序员有时会对 C++ 提供的相对斯巴达式地简练的特性集感觉惊讶，但这是因为，C++ 对并发的支持中有一大部分是以对编译器厂商实施约束的形式提供的。由此得到的一系列语言层面的保障，意味着在 C++ 的历史上，程序员首次可以跨越所有平台撰写具有标准行为的多线程程序。这就为建立富有表达力的库奠定了坚实的基础，而标准库中的并发元素（任务、期值、线程、互斥量、条件变量和原子对象等）仅仅是个开端，今后在开发并行 C++ 软件时，肯定会有一套越来越丰富的工具集。

在本章随后的条款中请牢记，标准库中为期值准备了两个模板：`std::future` 和 `std::shared_future`。在很多情况下，它们之间的区别并不重要，所以我经常仅仅谈论期值概念，意思是指可以对这两种都适用。

条款 35：优先选用基于任务而非基于线程的程序设计

如果你想以异步方式运行函数 `doAsyncWork`，有两种基本选择。你可以创建一个 `std::thread`，并在其上运行 `doAsyncWork`，因此这是基于线程（thread-based）的途径：

```
int doAsyncWork();

std::thread t(doAsyncWork);
```

抑或，你可以把 `doAsynWork` 传递给 `std::async`，这种策略叫做基于任务（task-based）：

```
auto fut = std::async(doAsyncWork);     // fut 是“期值”（future）的缩写
```

在这样的调用中，传递给 std::async 的函数对象（例如，doAsynWork）被看作任务（task）。

基于任务的方法通常比基于线程实现的对应版本要好，即使从刚才我们看过的这么几行代码中，已经展示了一些原因。请看，doAsyncWork 会产生一个返回值，我们有理由假定，调用 doAsyncWork 的代码会对该值感兴趣。在基于线程的调用中，没有什么直截了当的办法能够获取该值；而在基于任务的调用中，这很容易，因为 std::async 返回的期值提供了 get 函数。如果 doAsyncWork 函数发射了一个异常，get 函数就更重要了，因为它能访问到该异常。而如果采用了基于线程的途径，在 doAsyncWork 抛出异常时，程序就会死翘翘（经由调用 std::terminate）。

基于线程和基于任务的程序设计之间更基本的区别在于，基于任务的程序设计表现着更高阶的抽象。它把你从线程管理的细节中解放了出来，说到这里，我有必要概述一下“线程”在带有并发的 C++ 软件中的三种意义：

- 硬件线程是实际执行计算的线程。现代计算机体系结构会为每个 CPU 内核提供一个或多个硬件线程。
- 软件线程（又称操作系统线程或系统线程）是操作系统[注1]用以实施跨进程的管理，以及进行硬件线程调度的线程。通常，能够创建的软件线程会比硬件线程要多，因为当一个软件线程阻塞了（例如，阻塞在 I/O 操作上，或者需要等待互斥量或条件变量等），运行另外的非阻塞线程能够提升吞吐率。
- std::thread 是 C++ 进程里的对象，用作底层软件线程的句柄。有些 std::thread 对象表示为“null”句柄，对应于“无软件线程”，可能的原因有：它们处于默认构造状态（因此没有待执行的函数），或者被移动了（作为移动目的的 std::thread 对象成为了底层线程的句柄），或者被联结了（待运行的函数已运行结束），或者被分离了（std::thread 对象与其底层软件线程的连接被切断了）。

软件线程是一种有限的资源，如果你试图创建的线程数量多于系统能够提供的数量，就会抛出 std::system_error 异常。这一点无论如何都会成立，即使待运行函数不能抛出异常。例如，即使 doAsyncWork 带有 noexcept 声明饰词，

```
int doAsyncWork() noexcept;             // 关于 noexcept，参见条款 14
```

注 1：假定有操作系统。有些嵌入式系统中并无操作系统。

这条语句还是可能会抛出异常：

```
std::thread t(doAsyncWork);          // 若已无可用线程，则抛出异常
```

写得好的软件必须采取某种办法来处理这样的可能性，但如何解决呢？一个办法是在当前线程中运行 doAsyncWork，但这会导致负载不均衡，而且，如果当前线程是个 GUI 线程，会导致不能响应。另一个方法是等待某些已存在的软件线程完成工作，然后再尝试创建一个新的 std::thread 对象，但是有可能发生这种事情：已存在的线程在等待应该由 doAsyncWork 执行的某个动作（例如，产生返回值，或者向条件变量发送通知等）。

即使没有用尽线程，还是会发生超订（oversubscription）问题。也就是，就绪状态（即非阻塞）的软件线程超过了硬件线程数量的时候。这种情况发生以后，线程调度器（通常是操作系统的一部分）会为软件线程在硬件线程之上分配 CPU 时间片。当一个线程的时间片用完，另一个线程启动时，就会执行语境切换。这种语境切换会增加系统的总体线程管理开销，尤其在一个软件线程的这一次和下一次被调度器切换到不同的 CPU 内核上的硬件线程时会发生高昂的计算成本。在那种情况下，①那个软件线程通常不会命中 CPU 缓存（即，它们几乎不会包含对于那软件线程有用的任何数据和指令）；②CPU 内核运行的“新”软件线程还会“污染”CPU 缓存上为“旧”线程所准备的数据，它们曾经在该 CPU 内核上运行过，并且很可能再次被调度到同一内核运行。

避免超订是困难的，因为软件线程和硬件线程的最佳比例取决于软件线程变成可运行状态的频繁程度，而这是会动态地改变的，例如，当一个线程从 I/O 密集型转换为计算密集型的时候。软件线程和硬件线程的最佳比例也依赖于语境切换的成本，以及软件线程使用 CPU 缓存时的命中率。而硬件线程的数量和 CPU 缓存的细节（例如，缓存尺寸大小，以及相对速度）又取决于计算机体系结构，所以即使你在一个平台上调优好了你的应用，避免了超订（同时保持硬件满载工作），也无法保证在另一种机器上你的方案还是能高效工作。

如果把这些问题扔给别人去做，你的生活就可以轻松起来，而使用 std::async 正是做到了这一点：

```
auto fut = std::async(doAsyncWork);    // 由标准库的实现者负责线程管理
```

这句调用把线程管理的责任转交给了 C++ 标准库的实现者。例如，收到线程耗尽异常的可能性会大幅度地减小，因为这句调用可能从不产生该异常。“这怎么可能呢？”你可能会好奇，“如果我申请的软件线程数量多于系统可以提供的，使用 std::thread 和使用 std::async 有什么关系呢？”还真的有关系，因为当用这种形

式（即默认启动策略，参见条款 36）调用 std::async 时，系统不保证会创建一个新的软件线程。相反，它允许调度器把指定函数（本例中指 doAsyncWork）运行在请求 doAsyncWork 结果的线程中（例如，对 fut 调用了 get 或者 wait 的线程），如果系统发生了超订或线程耗尽，合理的调度器就可以利用这个自由度。

如果你自己来玩这个“在需求函数结果的线程上运行”的把戏，我曾说明过这会导致负载失衡，这问题并不会消失，只是由 std::async 和运行时调度器来代替你来面对它们。当谈到负载均衡时，运行时调度器很可能对于当前机器上正在发生什么比你有个更加全面的了解，因为它管理的是所有进程的线程，而非只是运着你的代码的那个。

即使使用 std::async，GUI 线程的响应性也会有问题，因为调度器没有办法知道哪个线程在响应性方面需求比较紧迫。在那种情况下，你可以把 std::lanuch::async 的启动策略传递给 std::async。那样做可以保证你想要运行的函数确实会在另一个线程中执行（参见条款 36）。

最高水平的线程调度器会使用全系统范围的线程池来避免超订，而且还会通过运用工作窃取算法来提高硬件内核间的负载均衡。C++ 标准库并未要求一定使用线程池或者工作窃取算法，而且，实话实说，C++11 并发规格的一些技术细节让我们不能像希望的程度那样去利用它们。但是，一些厂商还是会在它们的标准库实现中利用该技术，并且我们有理由期待这一领域会继续进步。如果你使用基于任务的方法进行编程，则在相关技术普及之时你会自动地享受到好处。如若不然，你直接使用了 std::thread 进行程序设计，则要自行承担处理线程耗尽、超订和负载均衡的重担，更不用提你在程序中的解决方案能否应用在同一台机器的另一个进程上，从而使问题变得雪上加霜了。

比起基于线程编程，基于任务的设计能够分担你手工管理线程的艰辛，而且它提供了一种很自然的方式，让你检查异步执行函数的结果（即，返回值或异常）。但是仍有几种情况下，直接使用线程会更适合，它们包括：

- **你需要访问底层线程实现的 API**：C++ 并发 API 通常会采用特定平台的低级 API 来实现，经常使用的有 pthread 或 Windows 线程库。它们提供的 API 比 C++ 提供的更丰富（例如，C++ 没有线程优先级和亲和性的概念）。为了访问底层线程实现的 API，std::thread 通常会提供 native_handle 成员函数，而 std::future（即 std::async 的返回型别）则没有该功能的对应物。

- **你需要且有能力为你的应用优化线程用法**。这也是有可能的，例如，你开发的是个服务器软件，执行时的性能剖析情况已知，并且作为唯一的主要进程部署在一种硬件特性固定的机器平台上。

- **你需要实现超越 C++ 并发 API 的线程技术。**例如，在 C++ 实现中未提供的线程池的平台上实现线程池。

无论如何，这些都是不常见的情况。大多数时候，你应该选择基于任务的设计，而非直接进行线程相关的程序设计。

要点速记

- std::thread的API未提供直接获取异步运行函数返回值的途径，而且如果那些函数抛出异常，程序就会终止。
- 基于线程的程序设计要求手动管理线程耗尽、超订、负载均衡，以及新平台适配。
- 经由应用了默认启动策略的std::async进行基于任务的程序设计，大部分这类问题都能找到解决之道。

条款 36：如果异步是必要的，则指定 std::launch::async

当调用 std::async 来执行一个函数（或可调用对象）时，你基本上都会想让函数以异步方式运行。但仅仅通过 std::async 来运行，你实际上要求的并非一定会达成异步运行的结果，你要求的仅仅是让该函数以符合 std::async 的启动策略来运行。标准策略有二，它们都是用限定作用域的枚举型别 std::launch 中的枚举量（关于限定作用域的枚举，参见条款 10）来表示的。假设函数 f 要传递给 std::async 以执行，则：

- **std::launch::async** 启动策略意味着函数 f 必须以异步方式运行，亦即，在另一线程之上执行。
- std::launch::deferred 启动策略意味函数 f 只会在 std::async 所返回的期值的 get 或 wait 得到调用时才运行。[注2] 亦即，执行会推迟至其中一个调用发生的时刻。

注 2：这种说法是对事实的简化。重点不在于那个调用了其 get 或 wait 的期值，而在于它指涉的共享状态（条款 38 讨论了期值和共享状态之间的关系）。由于 std::future 型别对象支持移动操作，也可以用以构造 std::shared_future 型别对象，而 std::shared_future 型别对象又可以复制，指涉到 f 传递给的那个 std::async 的调用所返回的共享状态期值对象很可能和 std::async 返回的期值对象并非同一个。这样说起来太啰嗦，所以通常不去说明这个事实，而只说在 std::async 返回的期值之上调用的 get 或 wait。

当调用 get 或 wait 时，f 会同步运行。即，调用方会阻塞至 f 运行结束为止。如果 get 或 wait 都没有得到调用，f 是不会运行的。

也许有点出人意料啦，std::async 的默认启动策略，也就是你如果不积极指定一个的话，它采用的并非以上两者中的一者。相反地，它采用的是对二者进行或运算的结果。下面两个调用有着完全相同的意义：

```
auto fut1 = std::async(f);                          // 采用默认启动策略运行 f

auto fut2 = std::async(std::launch::async |         // 采用或者异步
                       std::launch::deferred,       // 或者推迟的方式
                       f);                          // 运行 f
```

这么一来，默认启动策略就允许 f 以异步或同步的方式运行皆可，正如条款 35 所指出的那在，这种弹性使得 std::async 与标准库的线程管理组件能够承担得起线程的创建和销毁、避免超订，以及负载均衡的责任。这也是使用 std::async 来做并发程序设计如此方便的诸多因素中的一部分。

但以默认启动策略来使用 std::async，会触及一些意味深长的暗流。给定线程 t 执行一语句，

```
auto fut = std::async(f);                           // 采用默认启动策略运行 f
```

则：

- 无法预知 f 是否会和 t 并发运行，因为 f 可能会被调度为推迟运行。
- 无法预知 f 是否运行在与调用 fut 的 get 或 wait 函数的线程不同的某线程之上。如果那个线程是 t，那就是说无法预知 f 是否会运行在与 t 不同的某线程之上。
- 连 f 是否会运行这件起码的事情都是无法预知的，这是因为无法保证在程序的每条路径上，fut 的 get 或 wait 都会得到调用。

默认启动策略在调度上的弹性常会在使用 thread_local 变量时导致不明不白的混淆，因为这意味着如果 f 读或写此线程级局部存储（thread-local storage，TLS）时，无法预知会取到的是哪个线程的局部存储：

```
auto fut = std::async(f);       // f 的 TLS 可能是和一个独立线程相关，
                                // 但是也可能是和调用 fut 的 get 或 wait 的线程相关
```

它也会影响那些基于 wait 的循环中以超时为条件者，因为对任务（参见条款 35）调用 wait_for 或者 wait_until 会产生 std::launch::deferred 一值。这意味着以下循环虽然貌似会最终停止，但是，实际上，可能会永远运行下去：

```
using namespace std::literals;          // 关于 C++14 持续时长后缀，参见条款 34

void f()                                // f 睡眠 1 秒后返回
{
  std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);               // 以异步方式运行 f（说说而已）

while (fut.wait_for(100ms) !=           // 循环至
       std::future_status::ready)       // f 完成运行……
{                                       // 但此事可能永远不会发生！
  ...
}
```

如果 f 与调用 std::async 的线程是并发执行的（即选用了 std::launch::async 启动策略），这就没问题（假定 f 最终会完成执行）。但如果 f 被推迟执行，则 fut.wait_for 将总返回 std::future_status::deferred。而那永远也不会取值 std::future_status::ready，所以循环也就永远不会终止。

这一类缺陷在开发或单元测试中很容易被忽略，因为它只会在运行负载很重时才会现身。这样的负载是把计算机逼向超订或线程耗尽的条件，而那就是任务很可能会被推迟的时机了。毕竟，如果硬件层面没有面临超订或者线程耗尽的威胁，运行期系统并无理由不去调度任务以并发方式执行。

修正这个缺陷并不难：校验 std::async 返回的期值，确定任务是否被推迟，然后如果确实被推迟了，则避免进入基于超时的循环。不幸的是，没有直接的办法来询问期值任务是否被推迟了。作为替代手法，必须先调用一个基于超时的函数，例如 wait_for。在此情况下，你其实并不是要等待任何事情，而只是要查看返回值是否为 std::future_status::deferred，所以请搁置你的怀疑，径自调用一个超时为零的 wait_for 即可：

```
auto fut = std::async(f);                       // 同上

if (fut.wait_for(0s) ==                         // 如果任务
    std::future_status::deferred)               // 被推迟了……
{

...                     // ……则使用 fut 的 wait 或 get 以异步方式调用 f

} else {                // 任务未被推迟
  while (fut.wait_for(100ms) !=                 // 不可能死循环
         std::future_status::ready) {           // （前提假定 f 会结束）

    ...                 // 任务既未被推迟，也未就绪，
                        // 则做并发工作，直至任务就绪
  }
```

```
    ...                    // fut 就绪
}
```

将上述诸种因素纳入考量的总体结论是：以默认启动策略对任务使用 std::async 能正常工作需要满足以下所有条件：

- 任务不需要与调用 get 或 wait 的线程并发执行。
- 读 / 写哪个线程的 thread_local 变量并无影响。
- 或者可以给出保证在 std::async 返回的期值之上调用 get 或 wait，或者可以接受任务可能永不执行。
- 使用 wait_for 或 wait_unitil 的代码会将任务被推迟的可能性纳入考量。

只要其中一个条件不满足，你就很有可能想要确保任务以异步方式执行。实现这一点的手法，就是在调用时把 std::launch::async 作为第一个实参传递：

```
auto fut = std::async(std::launch::async, f);   // 以异步方式启动 f
```

其实，如果有个函数能像 std::async 那样运作，只是它会自动使用 std::launch::async 作为启动策略，那将是个趁手的工具，而且很不错的是撰写这个函数一点不难。这是 C++11 版本：

```
template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)              // 返回异步所需的期值
{
  return std::async(std::launch::async,         // 调用 f(params...)
                    std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

该函数接受一个可调用对象 f，以及零个或多个形参 params，并将后者完美转发（参见条款 25）给 std::async，同时传递 std::launch::async 作为启动策略。就像 std::async，它会返回一个型别为 std::future 的对象作为使用 params 调用 f 的结果。决定该结果的型别很容易，std::result_of 这个型别特征就把结果给到你了（有关型别特征的一般材料，参见条款 9）。

reallyAsync 的用法就像 std::async：

```
auto fut = reallyAsync(f);     // 以异步方式运行 f
                               // 如果 std::async 会抛出异常
                               // reallyAsync 也会抛出异常
```

在 C++14 中，对 reallyAsync 返回值进行推导型别的能力使函数声明得以简化：

```
template<typename F, typename... Ts>
inline
auto                                                    // C++14
reallyAsync(F&& f, Ts&&... params)
{
  return std::async(std::launch::async,
                    std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

这一版本清清楚楚地显示，reallyAsync 所做的一切，就是以 std::launch::async 启动策略来调用了 std::async。

要点速记

- std::async 的默认启动策略既允许任务以异步方式执行，也允许任务以同步方式执行。
- 如此的弹性会导致使用 thread_local 变量时的不确定性，隐含着任务可能永远不会执行，还会影响运用了基于超时的 wait 调用的程序逻辑。
- 如果异步是必要的，则指定 std::launch::async。

条款 37：使 std::thread 型别对象在所有路径皆不可联结

每个 std::thread 型别对象皆处于两种状态之一：可联结或不可联结。可联结的 std::thread 对应底层以异步方式已运行或可运行的线程。std::thread 型别对象对应的底层线程若处于阻塞或等待调度，则它可联结。std::thread 型别对象对应的底层线程如已运行至结束，则亦认为其可联结。

不可联结的 std::thread 的意思如你所想：std::thread 不处于以上可联结的状态。不可联结的 std::thread 型别对象包括：

- **默认构造的 std::thread**。此类 std::thread 没有可以执行的函数，因此也没有对应的底层执行线程。
- **已移动的 std::thread**。移动操作的结果是，一个 std::thread 所对应的底层执行线程（若有）被对应到另外一个 std::thread。

- **已联结的 std::thread**。联结后，std::thread 型别对象不再对应至已结束运行的底层执行线程。
- **已分享的 std::thread**。分离操作会把 std::thread 型别对象和它对应的底层执行线程之间的连接断开。

std::thread 的可联结性之所以重要的原因之一是：如果可联结的线程对象的析构函数被调用，则程序的执行就终止了。举个例子，假设我们有一个函数 doWork，它接受一个筛选器函数 filter 和一个最大值 maxVal 作为形参。doWork 会校验它做计算的条件全部成立，尔后会针对筛选器选出的 0 到 maxVal 之间的值实施计算。如果筛选是费时的，而条件检验也是费时的，那么并发地做这两件事就是合理的。

我们会优先选用使用基于任务的设计（参见条款 35），但是让我们假定会去设置实施筛选的那个线程的优先级。条款 35 解释过，这要求使用线程的低级句柄，从而只能通过 std::thread 的 API 来访问。基于任务的 API（如期值等）没有提供这个功能。因此，我们唯有采用基于线程一途，基于任务在此不可行。

我们可能会撰写出这样的代码：

```
constexpr auto tenMillion = 10000000;           // 关于 constexpr
                                                // 参见条款 15

bool doWork(std::function<bool(int)> filter,            // 返回值代表
            int maxVal = tenMillion)                    // 计算是否执行了
{                                                       // 关于 std::function
                                                        // 参见条款 2

  std::vector<int> goodVals;                            // 筛出的值

  std::thread t([&filter, maxVal, &goodVals]            // 遍历 goodVals
                {
                  for (auto i = 0; i <= maxVal; ++i)
                    { if (filter(i)) goodVals.push_back(i); }
                });

    auto nh = t.native_handle();                // 使用 t 的低值句柄设定 t 的优先级
    ...
    if (conditionsAreSatisfied()) {
      t.join();                                 // 让 t 结束执行
      performComputation(goodVals);
      return true;                              // 计算已实施
    }

    return false;                               // 计算未实施
}
```

在我解释这个代码为何有毛病之前，先说明 tenMillion 的初始值在 C++14 中可以更具可读性，这利用了 C++14 能把单引号作为数字分隔符的能力：

```
constexpr auto tenMillion = 10'000'000;          // C++14
```

我还想提一下在线程 t 开始执行之后才去设置它的优先级，这有点像老话说的，烈马已经脱缰跑走后才关上马厩的门。更好设计是以暂停状态启动线程 t（因而可以在它执行计算之前调整其优先级），但我不想在这里展示那部分代码来分散你的注意力。如果你感觉没看到这段代码更会分散注意力的话，请翻到条款 39，因为那里展示了如何启动处于暂停状态的线程。

回到 doWork，如果 conditionsAreSatisfied() 返回 true，则一切都好；但如果它返回 false 或者抛出了异常，那么在 doWork 的末尾调用 std::thread 型别对象 t 的析构函数时，它会是处于可联结状态，从而导致程序执行终止。

你可能想知道，std::thread 的析构函数为何会这样运作。那是因为，另外两种明显的选项可以说会更糟糕。它们是：

- **隐式 join**。在这种情况下，std::thread 的析构函数会等待底层异步执行线程完成。这听上去合理，但却可能导致难以追踪的性能异常。例如，如果 conditionAreSatisfied() 早已返回 false 了，doWork 却还在等待所有值上遍历筛选，这是违反直觉的。

- **隐式 detach**。在这种情况下，std::thread 的析构函数会分离 std::thread 型别对象与底层执行线程之间的连接。而该底层执行线程会继续执行。这听起来和 join 途径相比在合理性方面并不逊色，但它导致的调试问题会更加要命。例如，在 doWork 内 goodVals 是个通过引用捕获的局部变量，它也会在 lambda 式内被修改（通过对 push_back 的调用）。然后，假如 lambda 式以异步方式运行时，conditionsAreSatisfied() 返回了 false。那种情况下，doWork 会直接返回，它的局部变量（包括 goodVals）会被销毁，doWork 的栈帧会被弹出，可是线程却仍然在 doWork 的调用方继续运行着。

  在 doWork 调用方此后的语句中，在某个时刻，会调用其他函数，而至少会有一个函数可能会使用一部分或者全部 doWork 栈帧占用过的内存，不妨把这个函数称为 f。当 f 运行时，doWork 发起的 lambda 式依然在异步执行。该 lambda 式在原先的栈上对 goodVals 调用 push_back，不过那已是在 f 的栈帧中了。这样的调用会修改过去属于 goodVals 的内存，而那意味着从 f 的视角看，栈帧上的内存内容会莫名其妙地改变！想想看，你调试那样的问题时，会有多么酸爽。

标准委员会意识到，销毁一个可联结的线程实在太过可怕，所以实际上已经封印了这件事（通过规定可联结的线程的析构函数导致程序终止）。

这么一来，黑锅就甩给了你，如果你使用了 std::thread 型别对象，就得确保从它定义的作用域出去的任何路径，使它成为不可联结状态。但是覆盖所有路径是复杂的，这包括正常走完作用域，还有经由 return、continue、break、goto 或异常跳出作用域。路径何其多。

任何时候，只要想在每条出向路径上都执行某动作，最常用的方法就是在局部对象的析构函数中执行该动作。这样的对象称为 RAII 对象，它们来自 RAII 类（RAII 本身代表“Resource Acquisition Is Initialization”，资源获取即初始化，即使该技术的关键其实在于析构而非初始化）。RAII 类在标准库中很常见，例如 STL 容器（各个容器的析构函数都会析构容器内容并释放其内存）、标准智能指针（条款 18、条款 19 和条款 20 解释过，std::unique_ptr 的析构函数会对它指涉的对象调用删除器，而 std::shared_ptr 和 std::weak_ptr 的析构函数会对引用计数实施自减）、std::fstream 型别对象（其析构函数会关闭对应的文件），还有很多。然而，没有和 std::thread 型别对象对应的标准 RAII 类，可能是因为标准委员会把 join 或 detach 用作默认选项的途径都堵死了，这么一来也就不知道真有这样的类的话该如何运作。

幸运的是，自己写一个不难。例如，下面这个类就允许调用者指定 ThreadRAII 型别对象（它是个 std::thread 对应的 RAII 对象）销毁时是调用 join 还是 detach：

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };       // 关于枚举类，参见条款 10

  ThreadRAII(std::thread&& t, DtorAction a)     // 在析构函数中
  : action(a), t(std::move(t)) {}               // 在 t 上采取行动 a

  ~ThreadRAII()
  {                                             // 可联结性测试见下
    if(t.joinable()) {

      if(action == DtorAction::join) {
        t.join();
      } else {
        t.detach();
      }

    }
  }

  std::thread& get() { return t; }              // 见下
```

```
private:
  DtorAction action;
  std::thread t;
};
```

我希望这段代码基本上不言自明，但指出以下几点可能会有帮助：

- 构造函数只接受右值型别的 std::thread，因为我们想要把传入的 std::thread 型别对象移入 ThreadRAII 对象（提醒一下，std::thread 型别对象是不可复制的）。

- 读 / 写哪个线程的 thread_local 变量并无影响。或者可以给出保证在 std::async 返回的期值之上调用 get 或 wait，或者可以接受任务可能永不执行。使用 wait_for 或 wait_unitil 的代码会将任务被推迟的可能性纳入考量。构造函数的形参顺序的设计对于调用者而言是符合直觉的（指定 std::thread 作为第一个形参、而销毁行动作为第二个参数，比相反顺序更直观），但是，成员初始化列表的设计要求它匹配成员变量声明的顺序，而后者是把 std::thread 的顺序放到靠后的。在本类中，顺序不会导致不同，但作为一般讨论，一个成员变量的初始化有可能会依赖另一个成员变量，又因为 std::thread 型别对象初始化之后可能会马上用来运行函数，所以把它们声明在类的最后是个好习惯。这保证了当 std::thread 型别对象在构造之时，所有在它之前的成员变量都已经完成了初始化，因而 std::thread 成员变量对应的底层异步执行线程可以安全地访问它们了。

- ThreadRAII 提供了一个 get 函数，用以访问底层的 std::thread 型别对象。这和标准智能指针提供的 get 函数一样（后者用以访问底层裸指针）。提供 get 可以避免让 ThreadRAII 去重复 std::thread 的所有接口，也意味着 ThreadRAII 型别对象可以用于需要直接使用 std::thread 型别对象的语境。

- ThreadRAII 的析构函数在调用 std::thread 型别对象 t 的成员函数之前会先实施校验，以确保 t 可联结。这是必要的，因为针对一个不可联结的线程调用 join 或 detach 会产生未定义行为。用户有可能会构建了一个 std::thread 型别对象，然后从它出发创建一个 ThreadRAII 型别对象，再使用 get 访问 t，接着针对 t 实施移动或是对 t 调用 join 或 detach，而这样的行为会使 t 变得不可联结。

  如果你担心下面的代码会有竞险：

```
if (t.joinable()) {

  if (action == DtorAction::join) {
    t.join();
  } else {
    t.detach();
```

```
    }
  }
```

理由是，在 t.joinable() 的执行和 join 或 detach 的调用之间，另一个线程可能让 t 变得不可联结。你的直觉可圈可点，但你的担忧却是庸人自扰。一个 std::thread 型别对象只能通过调用成员函数以从可联结状态转换为不可联结状态，例如 join、detach 或移动操作。当 ThreadRAII 对象的析构函数被调用时，不应该有其他线程调用该对象的成员函数。如果同时发生多个调用，那的确会有竞险，但这竞险不是发生在析构函数内，而是发生在试图同时调用两个成员函数（一个是析构函数，一个是其他成员函数）的用户代码内。一般地，在一个对象之上同时调用多个成员函数，只有当所有这些函数都是 const 成员函数时才安全（参见条款 16）。

在我们的 doWork 一例中运用 ThreadRAII，代码会长成这样：

```
bool doWork(std::function<bool(int)> filter,              // 同前
            int maxVal = tenMillion)
{
  std::vector<int> goodVals;                              // 同前

  ThreadRAII t(                                           // 使用 RAII 对象
    std::thread([&filter, maxVal, &goodVals]
                {
                  for (auto i = 0; i <= maxVal; ++i)
                    { if (filter(i)) goodVals.push_back(i); }
                }),
                ThreadRAII::DtorAction::join              // RAII action
  );

  auto nh = t.get().native_handle();
  ...

  if(conditionsAreSatisfied()) {
    t.get().join();
    performComputation(goodVals);
    return true;
  }

  return false;
}
```

在该例中，我们选择在 ThreadRAII 析构函数中对异步执行线程调用 join。因为我们之前已经看到了，调用 detach 函数会导致噩梦般的调试。我们之前也看到过 join 会导致性能异常（实话实说，join 的调试也绝不令人愉悦），但在未定义行为（detach 导致的）、程序终止（使用裸 std::thread 产生的）和性能异常之间做出选择，性能异常也是权衡之下选取的弊端最小的一个。

哎呀，条款 39 会展示，使用 ThreadRAII 在 std::thread 析构中实施 join 不是仅仅会导致性能异常那么简单，而是会导致程序失去响应。这种问题的“合适的”解决方案是和异步执行的 lambda 式通信，当我们已经不再需要它运行，它应该提前返回，但 C++11 中并不支持这种可中断线程。我们可以手动实现它，但这个话题超出了本书的范围。[注3]

条款 17 解释过，因为 ThreadRAII 声明了析构函数，所以不会有编译器生成的移动操作，但这里 ThreadRAII 对象没有理由实现为不可移动的。如果编译器会生成这些函数，这些函数的行为就是正确的，所以显式地请求创建它们是适当的：

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };       // 同前

  ThreadRAII(std::thread&& t, DtorAction a)     // 同前
  : action(a), t(std::move(t)) {}

  ~ThreadRAII()
  {
    ...                                         // 同前
  }

  ThreadRAII(ThreadRAII&&) = default;           // 支持
  ThreadRAII& operator=(ThreadRAII&&) = default;          // 移动操作

  std::thread& get() { return t; }                        // 同前

private:                                                  // 同前
  DtorAction action;
  std::thread t;
};
```

要点速记

- 使 std::thread 型别对象在所有路径皆不可联结。
- 在析构时调用 join 可能导致难以调试的性能异常。
- 在析构时调用 detach 可能导致难以调试的未定义行为。
- 在成员列表的最后声明 std::thread 型别对象。

注 3：在 Anthony Williams 著的《C++ Concurrency in Action》（Manning 出版，2012）9.2 节可以找到一个精彩的实现。

条款 38：对变化多端的线程句柄析构函数行为保持关注

条款 37 解释过，可联结的线程对应着一个底层系统执行线程，未推迟任务（参见条款 36）的期值和系统线程也有类似关系。这么一来，std::thread 型别对象和期值对象都可以视作系统线程的句柄。

从这个视角来看，std::thread 对象和期值对象的析构函数表现出如此不同的行为值得深思。正如条款 37 所提及的，针对可联结的 std::thread 型别对象实施析构会导致程序终止，因为另外两个显而易见的选择（隐式 join 和隐式 detach）都被认为是更糟糕的选择。而期值的析构函数呢，有时候行为像是执行了一次隐式 join，有时候行为像是执行了一次隐式 detach，有时候行为像是二者都没有执行。它从不会导致程序终止。这套线程句柄行为的大杂烩，值得我们仔细品鉴一番。

我们的观察从这里开始：期值位于信道的一端，被调方把结果通过该信道传输给调用方。[注4] 被调方（通常以异步方式运行）把其计算所得的结果写入信道（通常经由一个 std::promise 型别对象），而调用方则使用一个期值来读取该结果。你可以把这个过程想像成下图，虚线箭头代表着从被调方向调用方的信息流：

但被调方的结果要存储在哪里呢？在调用方唤起对应期值的 get 之前，被调方可能已经执行完毕，因此结果不会存储在被调方的 std::promise 型别对象里。那个对象，对于被调方来说是个局部量，在被调方结束后会实施析构。

该结果也不能存储在调用方的期值中，因为（出于其他种种原因）可能会从 std::future 型别对象出发创建 std::shared_future 型别对象（因此把被调方结果的所有权从 std::future 型别对象转移至 std::shared_future 型别对象），而后者可能会在原始的 std::future 析构之后复制多次。如果被调方的结果型别不都是可复制的（即只移型别），而该结果至少生存期要延至和最后一个指涉到它的期值一样长。这么多个对应同一结果的期值中的哪一个，应该包含该结果呢？

既然与被调方相关联的对象和与调用方相关联的对象都不适合作为被调方结果的存储

注 4：条款 39 解释了，此类关联了期值的信道有其他用途。但在本条款中，我们仅考虑将其作为被调方将结果传达给调用方的机制这一用途。

之所，那就只能将该结果存储在位于两者外部的某个位置。这个位置称为共享状态。共享状态通常使用堆上的对象来表示，但是其型别、接口和实现标准皆未指定。标准库作者可以自由地用他们喜好的方法去实现共享状态。

我们可以把调用方、被调方和共享状态之间的关系使用下图来表示，虚线箭头仍然表示着信息流：

共享状态的存在很重要，因为期值析构函数的行为（这也是本条款的议题）是由与其关联的共享状态决定的。具体来说就是：

- **指涉到经由 std::aysnc 启动的未推迟任务的共享状态的最后一个期值会保持阻塞，直至该任务结束。**本质上，这样一个期值的析构函数是对底层异步执行任务的线程实施了一次隐式 join。

- **其他所有期值对象的析构函数只仅仅将期值对象析构就结束了。**对于底层异步运行的任务，这样做类似于对线程实施了一次隐式 detach。对于那些被推迟任务而言，如果这一期值是最后一个，也就意味着被推迟的任务将不会有机会运行了。

这些规则听上去复杂，其实不然。我们真正需要关心的，是一个平凡的“常规”行为外加一个不甚常见的例外而已。常规行为是指期值的析构函数仅会析构期值对象。就这样。它不会针对任何东西实施 join，也不会从任何东西实施 detach，也不会运行任何东西。它仅会析构期值的成员变量（好吧。实际上，它还多做了一件事。它针对共享状态里的引用计数实施了一次自减。该共享状态由指涉到它的期值和被调方的 std::promise 共同操纵。该引用计数使得库能够知道何时可以析构共享状态。关于引用计数的一般材料，参见条款 19）。

而相对于正常行为的那个例外，只有在期值满足以下全部条件时才会发挥作用：

- **期值所指涉的共享状态是由于调用了 std::async 才创建的。**

- **该任务的启动策略是 std::launch::async**（参见条款 36），这既可能是运行时系统的选择，也可能是在调用 std::async 时指定的。

- **该期值是指涉到该共享状态的最后一个期值。**对于 std::future 型别对象而言，这一点总是成立。而对于 std::shared_future 型别对象而言，在析构时如果不

是最后一个指涉到共享状态的期值，则它会遵循常规行为准则（即仅析构其成员变量）。

只有当所有条件都满足，期值的析构函数才会表现出特别行为。而行为的具体表现为阻塞直到异步运行的任务结束。从效果来看，这相当于针对正在运行 std::async 所创建的任务的线程实施了一次隐式 join。

经常会有人把这个例外和常规期望析构函数行为的差异说成是“来自 std::async 的期值会在其析构函数里被阻塞。”如果只是最粗略的近似，这种说法也不为错，但有时候你需要比最粗略的近似了解得更深一步。而现在，你已经知道了全面的真相。

抑或你的疑问又并不同，可能会是“为什么要为从 std::async 出发启动的非推迟任务相关联的共享状态专门制定一条规则？”问得合理。根据我所知道的，标准委员会想要避免隐式 detach 相关的问题（参见条款 37），但是他们又不想简单粗暴地让程序终止了事（他们针对可联结线程就是这样做的，参见条款 37），所以妥协结果就是实施一次隐式 join。这个决定并非没有争议，委员会也曾认真讨论过要在 C++14 中舍弃这样的行为。但是最后没有做出改变，所以期值析构函数的行为在 C++11 和 C++14 中是保持了一致的。

期值的 API 没有提供任何方法判断其指涉的共享状态是否诞生于 std::async 的调用，所以给定任意期值对象的前提下，它不可能知道自己是否会在析构函数中阻塞到异步任务执行结束。这个事实暗示着一些意味深长的推论：

```
// 该容器的析构函数可能会在其析构函数中阻塞，
//  因为它所持有的期值中可能会有一个或多个
// 指涉到经由 std::async 启动未推迟任务所产生的共享状态
std::vector<std::future<void>> futs;            // 关于 std::future<void>
                                                // 参见条款 39

class Widget {                                  // Widget 型别对象可能
public:                                         // 会在其析构函数中阻塞
  ...

private:
  std::shared_future<double> fut;
};
```

当然，如果有办法判定给定的期值不满足触发特殊析构行为的条件（例如，通过分析程序逻辑），即可断定该期值不会阻塞在其析构函数中。例如，只有因 std::async 调用而出现的共享状态才够格去展示特别行为，但是还有其他方法可以创建出共享状态。其中一个方法就是运用 std::packaged_task，std::packaged_task 型别对象会准备一个函数（或其他可调用的对象）以供异步执行，手法是将它加上一层包装，把

其结果置入共享状态。而指涉到该共享状态的期值则可以经由 std::packaged_task 的 get_future 函数得到：

```
int calcValue();                    // 待运行函数

std::packaged_task<int()>           // 给 calcValue 加上包装
  pt(calcValue);                    // 使之能以异步方式运行

auto fut = pt.get_future();         // 取得 pt 的期值
```

此时此刻，我们已知期值对象 fut 没有指涉到由 std::async 调用产生的共享状态，所以它的析构函数将表现出常规行为。

std::packaged_task 型别对象 pt 一经创建，就会运行在线程之上（它也可以经由 std::async 的调用而运行，但是如果你要用 std::async 运行任务，就没有很好的理由再去创建什么 std::packaged_task 型别对象，因为 std::async 能够在调度任务执行之前就做到 std::packaged_task 能够做到的任何事情）。

std::packaged_task 不能复制，所以欲将 pt 传递给 std::thread 的构造函数就一定要将它强制转型到右值（经由 std::move，参见条款 23）：

```
std::thread t(std::move(pt));           // 在 t 之上运行 pt
```

此例让我们能够隐约看出一些期值的常规析构行为，但如果把这些语句都放在同一代码块中，就可以看得更加清楚：

```
{                                   // 代码块开始

  std::packaged_task<int()>
    pt(calcValue);

  auto fut = pt.get_future();
  std::thread t(std::move(pt));

  …                                 // 见下

}                                   // 代码块结束
```

这里最值得探讨的代码是“…”部分，它位于 t 创建之后、代码块结束之前。值得探讨的是，在“…”中 t 的命运如何。基本存在三种可能：

- **未对 t 实施任何操作。**在这种情况下，t 在作用域结束点是可联结的，而这将导致程序终止（参见条款 37）。
- **针对 t 实施了 join。**在此情况下，fut 无须在析构函数中阻塞，因为在调用的代码已经有过 join。

- **针对 t 实施了 detach**。在此情况下，fut 无须在析构函数中实施 detach，因为在调用的代码已经做过这件事了。

换句话说，当你的期值所对应的共享状态是由 std::packaged_task 产生的，则通常无需采用特别析构策略。因为，关于是终止、联结还是分离的决定，会由操纵 std::thread 的代码作出，而 std::packaged_task 通常就运行在该线程之上。

要点速记

- 期值的析构函数在常规情况下，仅会析构期值的成员变量。
- 指涉到经由 std::aysnc 启动的未推迟任务的共享状态的最后一个期值会保持阻塞，直至该任务结束。

条款 39：考虑针对一次性事件通信使用以 void 为模板型别实参的期值

有时候，提供让一个任务通知另一个以异步方式运行的任务发生了特定的事件的能力，会是有用的，原因可能是第二个任务在事件发生之前无法推进。这事件也许是某个数据结构完成初始化了，也许是某个计算阶段结束了，又也许是某个重要传感器取值被检测到了等。在此情况下，何为线程间通信的最佳方式？

一种明显的途径是使用条件变量。若我们把检测条件的任务称为检测任务，把对条件作出反应的任务称为反应任务，则策略表述起来很简单：反应任务等待着条件变量，而检测任务则在事件发生时通知条件变量。给定：

```
std::condition_variable cv;            // 事件的条件变量
std::mutex m;                          // 在运用 cv 时给它加的互斥量
```

检测任务的代码可谓简单到不能再简单了：

```
…                          // 检测事件
cv.notify_one();           // 通知反应任务
```

如果有多个反应任务需要通知到，那么使用 notify_all 替换 notify_one 才合适，但是现在不妨假设只有一个反应任务。

反应任务的代码稍显复杂，因为在条件变量之上调用 wait 之前，必须通过 std::unique_lock 型别对象锁定互斥量（在等待条件变量之前锁定互斥量，对于线程库来说是典型操作。而通过 std::unique_lock 对象完成锁定互斥量的需求，是 C++11 在 API 中所提供的功能）。下面展示了概念上应该如何实现：

```
...                                          // 准备反应

{                                            // 临界区域开始

  std::unique_lock<std::mutex> lk(m);        // 为互斥量加锁

  cv.wait(lk);                               // 等待通知到来
                                             // 这里会出错!
  ...                                        // 针对事件作出反应
                                             // (m 被锁定)

}                                            // 临界区域结束
                                             // 通过 lk 的析构函数为 m 解锁

...                                          // 继续等待反应
                                             // (m 已解锁)
```

这种途径的第一个问题有时被称为代码异味（code smell）：即使代码能够一时运作，某些东西似乎也不太对劲。在本例中，异味源于需要使用互斥体。互斥体是用于控制共享数据访问的，但检测和反应任务之间大有可能根本不需要这种介质。例如，检测任务可能负责初始化一个全局数据结构，然后把它转交给反应任务使用。如果检测任务在初始化之后从不访问该数据结构，并且在检测任务指示它已就绪之前，反应任务从不访问它，那么根据程序逻辑，这两个任务将会互相阻止对方访问。根据不需要什么互斥量。采用条件变量这一途径却要求必须有个互斥量，这个事实就为设计留下了令人生疑和不安的气息。

即使对此视而不见，还有两个问题是无论如何都需要关切的：

- **如果检测任务在反应任务调用 wait 之前就通知了条件变量**，则反应任务将失去响应。为了实现通知条件变量唤醒另一个任务，该任务必须已在等待该条件变量。如果检测任务在响应任务执行 wait 之前就执行了通知动作，则反应任务就将错过该通知，并且将等待到地老天荒。

- **反应任务的 wait 语句无法应对虚假唤醒**。线程 API 的存在一个事实情况（很多语言中都如此，不仅仅是 C++），即使没有通知条件变量，针对该条件变量等待的代码也可能被唤醒。这样的唤醒称为虚假唤醒。正确的代码通过确认等待的条件确实已经发生，并将其作为唤醒后的首个动作来处理这种情况。C++ 的条

件变量 API 使得做到一点异常简单，因为它允许测试等待条件的 lambda 式（或其他函数对象）被传递给 wait。换言之，反应任务中调用 wait 时可以这样撰写：

```
cv.wait(lk,
        []{ return 事件是否确已发生；});
```

想要利用这项能力，就要求反应任务能够确认它所等待的条件是否成立。但是在我们考虑的上述场景中，是由检测线程负责识别它所等待的条件是否因为对应的事件发生导致的。反应线程可能无法确认它正在等待的事件是否已经发生。这也是为什么它等待的是个条件变量！

在许多情况下，使用条件变量进行任务间通信是对于所面对问题的适当解法，但我们现在看的这个问题似乎并非其中之一。

许多软件工程师的下一个锦囊妙计是使用共享的布尔标志位。该标志位的初始值是 false。当检测线程识别出它正在查找的事件时，会设置该标志位：

```
std::atomic<bool> flag(false);          // 共享的布尔标志位
                                        // 关于 std::atomic，参见条款 40

  …                                     // 检测事件
flag = true;                            // 通知反应任务
```

在这一途径中，反应线程只是会轮询标志位。一旦看到该标志被设置时，就知道它正在等待的事件已经发生了。

```
…                               // 准备反应

while (!flag);                  // 等待事件

…                               // 针对事件作出反应
```

这种方法没有任何基于条件变量的设计的缺点。不需要互斥体。如果检测任务在反应任务开始轮询之前就设置了标志位，也没有任何问题。并且虚假唤醒的毛病也不见了。这个办法好，就是好来就是好。

可是不那么好的地方在于，反应任务的轮询可能成本高昂。在任务等待标志位被设置的时候，它实质上应该被阻塞，但却仍然在运行。因此，它就占用了另一个任务本应该能够利用的硬件线程，而且在每次开始运行以及其时间片结束时，都会产生语境切换的成本。它还可能会让一颗硬件核心持续运行，而那颗核心本来可以关掉以节省电能。真正处于阻塞状态的任务不会耗用所有以上这些。这倒是基于条件变量的途径的一个优点，因为等待调用的任务会真正地被阻塞。

常用的手法是结合条件变量和基于标志位的设计。标志位表示是否发生了有意义的事件，但是访问该标志要通过互斥量加以同步。因为互斥锁会阻止并发访问该标志位，所以，如条款 40 所说，不需要该标志位采用 std::atomic 型别对象来实现，一个平凡的布尔量足矣。这么一来，检测任务会长成这样：

```
std::condition_variable cv;              // 同前
std::mutex m;

bool flag(false);                        // 非 std::atomic 型别对象

...                                      // 检测事件

{
  std::lock_guard<std::mutex>  g(m);     // 经由 g 的构造函数锁定 m

  flag = true;                           // 通知反应任务（第一部分）

}                                        // 经由 g 的析构函数为 m 解锁
cv.notify_one();                         // 通知反应任务（第二部分）
```

以下是反应任务的实现：

```
...                                      // 准备反应

{                                        // 同前
  std::unique_lock<std::mutex> lk(m);    // 同前

  cv.wait(lk, [] { return flag; });      // 使用 lambda 式应对虚假唤醒

  ...                                    // 针对事件作出反应
                                         // （m 被锁定）
}

...                                      // 继续等待反应
                                         // （m 已解锁）
```

采用这一途径，可以避免我们先前讨论过的问题。它能够运作、在检测任务通知之前响应任务就开始等待也没关系，在存在虚假唤醒的前提下也不影响，而且不需要轮询。然而，还是有一丝异味存在，因为探测任务和反应任务的沟通方式非常奇特。通知条件变量在这里的目的是告诉反应任务，它正在等待的事件可能已经发生了，然而反应任务必须检查标志位才能确定。设置标志位在这里的目的是告诉反应任务事件确确实实已经发生了，但是检测任务仍然需要通知条件变量才能让反应任务被唤醒并去检查标志位。这一途径是能够运作的，但是不够干净利落。

另一种方法是摆脱条件变量，互斥量和标志位，方法是让反应任务去等待检测任务设置的期值。这看似是一种怪异的想法。毕竟，条款 38 曾经解释说，期值代表了从被调者到（通常以异步方式运行的）调用者的信道接收端，在检测和反应任务之间并不

存在这种调用者和被调者的关系。不过，条款 38 又指出，发送端是 std::promise 型别对象，并且其接收端是期值的通信信道用途不止于调用者和被调者一种。这种信道可以用于任何需要将信息从一处传输到另一处的场合。在本例中，我们将使用它来将信息从检测任务传输到响应任务，传达信息则是有意义的事件已经发生。

这种设计简单易行。检测任务有一个 std::promise 型别对象（即，信道的写入端），反应任务有对应的期值。当检测任务发现它正在查找的事件已经发生时，它会设置 std::promise 型别对象（即，向信道写入）。与此同时，反应任务调用 wait 以等待它的期值。该 wait 调用会阻塞反应任务直至 std:promise 型别对象被设置为止。

在这里的 std::promise 和期值（即 std::future 和 std::shared_future）都是需要型别形参的模板。该形参表示的是要通过信道发送数据的型别。在本例中，却并没有数据要传送。对于反应任务有意义的唯一事情，就是它的期值有否已被设置。我们所需要的 std::promise 和期值模板是一种表示没有数据要通过信道传送的那么一种型别。那种型别就是 void。因此，检测任务将使用 std::promise<void>，并且反应任务将使用 std::future<void> 或 std::shared_future<void>。当有意义的事件发生时，检测任务将设置其 std::promise<void>，反应任务将等待其期值。即使反应任务不会接收任何来自检测任务的数据，信道也会允许反应任务通过在其 std::promise 型别对象上调用 set_value 来了解检测任务何时“写入”了其 void 型别的数据。

所以，给定

```
std::promise<void> p;                    // 信道的约值
```

检测任务的代码是平凡的：

```
…                                        // 检测事件
p.set_value();                           // 通知反应任务
```

而反应任务的代码也同样无奇：

```
…                                        // 准备反应
p.get_future().wait();                   // 等待 p 对应的期值
…                                        // 针对事件作出反应
```

就像使用标志位的途径一样，这个设计也不需要互斥量，检测任务是否在响应任务等待之前设置它的 std::promise 都可以，且对虚假唤醒免疫（只有条件变量会不能应对虚假唤醒）。也像基于条件变量的途径一样，在调用 wait 之后，反应任务真正被阻塞，所以在等待时不会消耗系统资源。完美，对不对？

不对。当然，基于期值的途径可以绕开前面那些险滩，但仍不免于其他一些陷阱。例如，条款 38 就解释过，std::promise 和期值之间是共享状态，而共享状态通常是动态分配的。因此，你就得假设这种设计会招致在堆上进行分配和回收的成本。

可能这一点是最重要的：std::promise 型别对象只能设置一次。std::promise 型别对象和期值之间的通信通道是个一次性机制：它不能重复使用。这是它基于条件变量和基于标志位的设计之间的显著差异，前两者都可以用来进行多次通信（条件变量可以被重复通知，标志位可以被清除并重新设置）。

一次性这一约束并不像你可能想像的限制那么大。假设你想创建一个暂停状态的系统线程。也就是说，你希望一开始就把与创建线程相关的所有开销都提前付清，尔后一旦要在线程上执行某些操作时即可避免常规的线程创建延迟了。又或者你可能想创建一个暂停的线程，以便在它运行之前先对其实施一些配置动作。这样的配置可能包括诸如设置其优先级或内核亲和性之类。C++ 并发 API 并未提供做这些事情的方法，但 std::thread 型别对象提供了 native_handle 成员函数，意在让你得以访问平台的底层线程 API（通常是 POSIX 线程或 Windows 线程）。低级 API 通常能够配置像优先级和亲和性这样的线程特征。

假定你只想暂停线程一次（在它创建之后，但在它运行其线程函数之前），使用 void 期值的设计就是合理的选择。下面是该技术的重要部分：

```
std::promise<void> p;

void react();                                   // 反应任务的函数

void detect()                                   // 检测任务的函数
{
  std::thread t([]                              // 创建线程
                {
                  p.get_future().wait();        // 暂停 t
                  react();                      // 直至其期值被设置
                });
  …                 // 在这里 t 处于暂停状态，在调用 react 之前

p.set_value();      // 取消暂停 t（调用 react）

  …                 // 做其他工作

  t.join();         // 置 t 于不可联结状态（参见条款 37）
}
```

因为使 t 在所有 detect 的出向路径上都置为不可联结这件事情很重要，所以使用条款 37 中像 ThreadRAII 那样的 RAII 类应该是可取的。于是，你可能会脑补出这么一段代码来：

```
void detect()
{
  ThreadRAII tr(                          // 使用 RAII 对象
    std::thread([]
                {
                  p.get_future().wait();
                  react();
                }),
    ThreadRAII::DtorAction::join          // 这里有风险！（见下）
  );

  ...                                     // tr 内的线程在此处被暂停

  p.set_value();                          // tr 内的线程在此处被取消暂停

  ...
}
```

这段代码不像看上去那么安全。问题在于第一个“…”区域（带有“tr 内的线程在此处被暂停”注释的那个），如果抛出异常的话，set_value 便永远不会在 p 上调用。这意味着，在 lambda 式内部调用的 wait 将永远不会返回。而这，反过来又意味着，运行 lambda 式的线程将永远不会完成，这是个问题，因为 RAII 对象 tr 已被配置为在 tr 的析构函数中针对该线程执行 join。换言之，如果从代码的第一个“…”区域抛出异常，这个函数将会失去响应，因为 tr 的析构函数将永远不会完成。

有多种方法可以解决该问题，但我将它留给读者作为一个宝贵的练习机会。[注5] 在这里，我想展示如何原始代码（即，不使用 ThreadRAII）加以扩充，使之可以针对不止一个，可以是很多个反应任务实施先暂停再取消暂停的功能。这个拓展不难，因为关键之处在于在 react 的代码中中使用 std::shared_futures 而非 std::future。一旦你了解到，std::future 的 share 成员函数是把共享状态的所有权转移给了由 share 生成的 std::shared_future 型别对象，代码也就自己呼之欲出了。唯一的微妙之处就是，每个反应线程都需要自己的那份 std::shared_future 副本去指涉到共享状态，所以，从 share 中获取的 std::shared_future 被运行在反应线程上的 lambda 式按值捕获：

```
std::promise<void> p;                     // 同前

void detect()                             // 现在可以处理多个反应任务了
{

  auto sf = p.get_future().share();       // sf 的型别是
                                          // std::shared_future<void>
```

注5：我在 2013 年 12 月 24 日发表的博客“The View From Aristeia, ThreadRAII + Thread Suspension = Trouble?”可以作为研究这一问题的出发点（此博客的译文可以关注国内的相关 C++ 论坛）。

```
  std::vector<std::thread> vt;          // 反应任务的容器

  for (int i = 0; i < threadsToRun; ++i) {
    vt.emplace_back([sf]{ sf.wait();    // sf 局部副本之上的 wait
                          react(); }); // 关于 emplace_back，参见条款 42
  }


  ...                                   // 若此 "..." 抛出异常，则 detect 会失去响应！

  p.set_value();                        // 让所有线程取消暂停

  ...

  for (auto& t : vt) {                  // 把所有线程置为不可联结状态
    t.join();                           // 欲知 "auto&" 详情，参见条款 2
  }
}
```

使用期值的设计能够实现这样的效果，此事实值得注意，这也是为何应该将一次性事件通信纳入考量。

要点速记

- 如果仅为了实现平凡事件通信，基于条件变量的设计会要求多余的互斥量，这会给相互关联的检测和反应任务带来约束，并要求反应任务校验事件确已发生。
- 使用标志位的设计可以避免上述问题，但这一设计基于轮询而非阻塞。
- 条件变量和标志位可以一起使用，但这样的通信机制设计结果不甚自然。
- 使用 std::promise 型别对象和期值就可以回避这些问题，但是一来这个途径为了共享状态需要使用堆内存，而且仅限于一次性通信。

条款 40：对并发使用 std::atomic，对特种内存使用 volatile

可怜的 volatile。被误解到如此地步。它甚至不应该出现在本章中，因为它与并发程序设计毫无关系。但是在其他程序设计语言中（例如 Java 和 C#），它还是会对并发程序设计有些用处。甚至在 C++ 中，一些编译器也已经把 volatile 投入了染缸，使得它的语义显得可以用于并发软件中（但是仅可用于使用这些编译器进行编译之时）。

因此，除了消除环绕在它周围的混淆视听外，没有什么其他的理由值得在关于并发的一章中讨论 volatile。

程序员有时会把 volatile 与绝对属于本章讨论范围的另一 C++ 特性混淆，那就是 std::atomic 模板。该模板的实例（例如，std::atomic<int>、std::atomic<bool> 和 std::atomic<Widget*> 等）提供的操作可以保证被其他线程视为原子的。一旦构造了一个 std::atomic 型别对象，针对它的操作就好像这些操作处于受互斥量保护的临界区域内一样，但是实际上这些操作通常会使用特殊的机器指令来实现，这些指令比使用互斥量来得更加高效。

考虑以下应用了 std::atomic 的代码：

```
std::atomic<int> ai(0);          // 将 ai 初始化为 0

ai = 10;                         // 将 ai 原子地设置为 10

std::cout << ai;                 // 原子地读取 ai 的值

++ai;                            // 原子地将 ai 自增为 11

--ai;                            // 原子地将 ai 自减为 10
```

在这些语句的执行期间，其他读取 ai 的线程可能只会看到它取值为 0、10 或 11，而不可能有其他的取值（当然，前提假设这是修改 ai 值的唯一线程）。

此例在两方面值得注意。首先，在“std::cout << ai;”这个语句中，ai 是 std::atomic 这一事实只能保证 ai 的读取是原子操作。至于整个语句都以原子方式执行，则没有提供如此保证。在读取 ai 的值和调用 operator << 将其写入标准输出之间，另一个线程可能已经修改了 ai 的值。这对语句的行为没有影响，因为整型的 operator << 会使用按值传递的 int 型别的形参来输出（因此输出的值会是从 ai 读取的值），重点在于了解这个语句中具备原子性的部分仅在于 ai 的读取而不涉及其余更多部分。

此例子第二个值得注意的方面是最后两个语句的行为——ai 的自增和自减。这两个都是读取—修改—写入（read-modify-write，RMW）操作，但皆以原子方式执行。这是 std::atomic 型别最棒的特性之一：一旦构造出 std::atomic 型别对象，其上所有的成员函数（包括那些包含 RMW 操作的成员函数）都保证被其他线程视为原子的。

对比之下，使用 volatile 的相应代码在多线程语境中几乎不能提供任何保证：

```
volatile int vi(0);              // 将 vi 初始化为 0

vi = 10;                         // 将 vi 设置为 10
```

```
std::cout << vi;                // 读取 vi 的值

++vi;                           // 将 vi 自增为 11

--vi;                           // 将 vi 自减为 10
```

在这段代码的执行期间，如果其他线程正在读取 vi 的值，它们可能会看到任何值，例如 -12、68、4090727，任何值！这样的代码会出现未定义的行为，因为这些语句修改了 vi，所以如果其他线程同时正在读取 vi，就会出现在既非 std::atomic，也非由互斥量保护的同时读写操作，这就是数据竞险的定义。

为了说明 std::atomic 型别对象和 volatile 的行为在多线程程序中会有怎样的差异，这里举个具体例子，考虑两者由多个线程执行自增的简单计数器。二者都初始化为 0：

```
std::atomic<int> ac(0);         // "ac" 是 "atomic counter" （原子计数器）缩写

volatile int vc(0);             // "vc" 是 "volatile counter" （挥发计数器）缩写
```

尔后，我们在两个同时运行的线程中将两者各自增一次：

```
/*-----  线程 1  ----- */      /*-----  线程 2  ----- */

        ++ac;                           ++ac;
        ++vc;                           ++vc;
```

当两个线程都完成后，ac 的值（即，std::atomic 型别对象的值）必定是 2，因为每次自增都是作为不可分割的操作出现的。另一方面，vc 的值则不一定是 2，因为它的自增可能会不以原子方式发生。每次自增包括读取 vc 的值、自增读取的值，并将结果写回 vc。但是这三个操作皆不能保证以原子方式处理 volatile 对象，所以可能两次 vc 自增的组成部分会交错进行，如下所示：

1. 线程 1 读取 vc 的值，即 0。
2. 线程 2 读取 vc 的值，仍为 0。
3. 线程 1 把读取的值 0 自增为 1，并将该值写入 vc。
4. 线程 2 把读取的值 0 自增为 1，并将该值写入 vc。

这么一来，vc 最终的值为 1，即使它被实施了两次自增操作。

这不是唯一可能的结果，vc 的最终取值一般说来是无法预测的，因为 vc 涉及数据竞险，而标准既然裁定数据竞险会导致未定义行为，意味着编译器可能会生成代码来做任何事情。当然，编译器一般不会利用这种保留余地来作什么恶。可是，它们会执行一些

在对于没有数据竞险的程序而言有效的优化，但这些优化在存在竞险的程序则会产生意想不到的、无法预测的行为。

RMW 操作的使用并不是唯一让 std::atomic 型别对象在并发条件下成功，而让 volatile 失败的情况。假设一个任务负责计算第二个任务所需的重要值。当第一个任务已经计算出该值时，它必须把这个值通信到第二个任务。条款 39 解释过，要使第一个任务将所需值的可用性传递给第二个任务，有一种方法就是使用 std::atomic<bool>。在负责计算的任务中，代码会长成差不多这样：

```
std::atomic<bool> valAvailable(false);

auto imptValue = computeImportantValue();        // 计算值

valAvailable = true;                             // 通知其他任务值已可用
```

当人类在阅读这段代码时，都会知道在为 valAvailable 赋值之前为 imptValue 赋值这一点至关重要，但是编译器所能看到的一切，不过是一对针对独立变量实施的赋值操作。一般地，编译器可以将这些不相关的赋值重新排序。换言之，给定下面的赋值序列（其中 a、b、x 和 y 对应于独立变量），

```
a = b;
x = y;
```

编译器可以径自将其重新排序成下面这样：

```
x = y;
a = b;
```

即使编译器未对它们进行重新排序，底层硬件也可能会这样做（或者可能会让其他内核将其视为重新排序后的样子），因为这样做有时会使代码运行得更快。

然而，std::atomic 型别对象的运用会对代码可以如何重新排序施加限制，并且这样的限制之一就是，在源代码中，不得将任何代码提前至后续会出现 std::atomic 型别变量的写入操作的位置（或使其他内核视作这样的操作会发生）。[注 6] 这意味着在我们的代码中：

注 6：这一点仅在 std::atomic 型别对象采用顺序一致性时才成立，这种一致性是默认采用的，也是本书中使用该语法时唯一采用的一致性模型。C++ 还支持另外的、在代码重新顺序方面更灵活的一致性模型。这样的弱化（也称松弛）模型使得创建在某些硬件体系结构上运行得更快的软件成为可能，但是运用这样的模型所产生的软件要想保证正确性、可理解性和可维护性，会困难得多。在松弛原子性中的微妙代码错误决不罕见，即使专家也会感觉棘手。所以但凡可能，你就应该抱紧顺序一致性不要放松。

```
auto imptValue = computeImportantValue();       // 计算值

valAvailable = true;                            // 通知其他任务值已可用
```

不仅编译器必须保留为 imptValue 和 valAvailable 的赋值顺序，它们还必须生成代码以确保底层硬件也保证这个顺序。因此，将 valAvailable 声明为 std::atomic 型别可以确保我们的关键顺序需求得到保证，imptValue 必须被所有线程看到，它是以不晚于 valAvailable 的时序被更改。

将 valAvailable 加上 volatile 声明饰词，不会给代码施加同样的重新排序方面的约束：

```
volatile bool valAvailable(false);

auto imptValue = computeImportantValue();

valAvailable = true;  // 其他线程可能将这个赋值操作视作
                      // 在 imptValue 之前!
```

在这里，编译器可能会将赋值顺序翻转为先 imptValue 后 valAvailable，即使它不这样做，也可能不会生成机器代码阻止底层硬件使其他内核上的代码看到 valAvailable 在 imptValue 之前发生改变。

这两个问题（无法保证操作的原子性，无法对代码重新排序施加限制）解释了为何 volatile 对于并发编程没用，但是并未解释它在什么情况下有用。简而言之，它的用处就是告诉编译器，正在处理的内存不具备常规行为。

“常规”内存的特征是：如果你向某个内存位置写入了值，该值会一直保留在那里，直到它被覆盖为止。所以，如果我有个常规的 int 变量：

```
int x;
```

且编译器看到了对其实施了以下序列的操作：

```
auto y = x;  // 读取 x
y = x;       // 再次读取 x
```

编译器可以通过消除对 y 的赋值操作来优化生成的代码，因为它和 y 的初始化形成了冗余。

常规内存还具有如下特征：如果向某内存位置写入某值，其间未读取该内存位置，然后再次写入该内存位置，则第一次写入可以消除，因为其写入结果从未使用过。所以，给定下面的两个相邻语句：

```
x = 10;       // 写入 x
x = 20;       // 再次写入 x
```

编译器就可以消除第一个操作，这意味着如果我们在源代码中有这样一段：

```
auto y = x;   // 读取 x
y = x;        // 再次读取 x
x = 10;       // 写入 x
x = 20;       // 再次写入 x
```

编译器可以径自把这段代码视作像长成下面这样一般：

```
auto y = x;   // 读取 x

x = 20;       // 写入 x
```

恐怕你会想，谁会撰写执行如此的冗余读取和多余写入的代码（术语是冗余加载和废弃存储）呢？答案是，人类不会直接撰写出如此代码，至少我们希望没人会这样做吧。但是，即使编译器接受的是看上去合情合理的源代码，对其执行模板实例化、内联以及各种常见的重新排序等优化后，结果中包含编译器能够消除的冗余加载和废弃存储的情况并不罕见。

此类优化仅在内存行为符合常规时才合法。“特种”内存就是另一回事。可能最常见的特种内存是用于内存映射 I/O 的内存。这种内存的位置实际上是用于与外部设备（例如，外部传感器、显示器、打印机和网络端口等）通信，而非用于读取或写入常规内存（即 RAM）。在此情况下，再次考虑看似冗余的代码：

```
auto y = x;   // 读取 x
y = x;        // 再次读取 x
```

如果 x 对应于，比如说，由温度传感器报告的值，则 x 的第二次读取操作并非多余，因为在第一次和第二次读取之间，温度可能已经改变。

看似多余的写入操作也有类似情形。比如，在这段代码中：

```
x = 10;       // 写入 x
x = 20;       // 再次写入 x
```

如果 x 对应于无线电发射器的控制端口，则有可能是代码在向无线电发出命令，并且值 10 对应于与值 20 不同的命令。如果把第一个赋值优化掉，就将改变发送到无线电的命令序列了。

而 `volatile` 的用处就是告诉编译器，正在处理的是特种内存。它的意思是通知编译器“不要对在此内存上的操作做任何优化”。所以，如果 x 对应于特种内存，则它应该加上 `volatile` 声明饰词：

```
volatile int x;
```

考虑这么一来，会对我们原先的代码序列产生什么影响：

```
auto y = x;  // 读取 x
y = x;       // 再次读取 x（不会被优化掉了！）

x = 10;      // 写入 x（不会被优化掉了！）
x = 20;      // 再次写入 x
```

如果 x 是内存映射的（或已映射到跨进程共享的内存位置等），这正是我们想要的效果。

测验时间！在上面最后一段代码中，y 应取什么型别：int 还是 volatile int？[注 7]

在处理特种内存时必须保留看似冗余加载和废弃存储这一事实，也顺便解释了为何 std::atomic 型别对象不适用于这种工作。编译器可以消除 std::atomic 型别上的冗余操作。代码的撰写方式与使用 volatile 时不尽相同，但是我们不妨暂时忽略这一点，而先关注编译器允许做的事情，我们可以这么说，从概念上说，编译器可能接受的是这样的代码：

```
std::atomic<int> x;

auto y = x;          // 概念上会读取 x（见下）
y = x;               // 概念上会再次读取 x（见下）

x = 10;              // 写入 x
x = 20;              // 再次写入 x
```

并优化成下面这样：

```
auto y = x;          // 概念上会读取 x（见下）
x = 20;              // 写入 x
```

这显然对于特种内存来说，是不可接受的行为。

无巧不成书，以下两个语句在 x 是 std::atomic 型别对象时都不能通过编译：

```
auto y = x;          // 错误！
y = x;               // 错误！
```

原因在于 std::atomic 的复制操作被删除了（参见条款 11）。而且这个删除是有充分

注 7：y 的型别是 auto 推导的结果，所以它使用了条款 2 中描述的规则。那些规则规定，声明非引用且非指针的型别时（y 符合该情况），const 和 volatile 饰词会被丢弃。是故，y 的型别就是 int。这意味着对 y 的冗余读写操作可以被消除。在本例中，编译器必须针对 y 实施初始化和赋值，而因为 x 的声明带有 volatile 饰词，所以第二次读取的 x 值有可能会产生与第一次不同的结果。

道理的。考虑如果从 x 出发来初始化 y 能够通过编译的话，会发生什么。由于 x 的型别是 std::atomic，所以 y 的型别也会被推导为 std::atomic（参见条款 2）。我之前说过，std::atomic 型别对象最好的一点，是它们的所有操作都是原子的。但是，为了使得从 x 出发来构造 y 的操作也成为原子的，编译器就必须生成代码来在单一的原子操作中读取 x 并写入 y。硬件通常无法完成这样的操作，所以 std::atomic 型别不支持复制构造。出于相同的原因，复制赋值也被删除，这就是为什么从 x 到 y 的赋值通不过编译的原因（由于移动操作没有在 std::atomic 中显式声明，因此，根据条款 17 中描述的编译器生成特种函数的规则，std::atomic 既不提供移动构造函数，也不提供移动赋值运算符）。

从 x 中取值并置入 y 是可以实现的，但是要求使用 std::atomic 的成员函数 load 和 store。load 成员函数以原子方式读取 std::atomic 型别对象的值，而 store 成员函数以原子方式写入之。如果想先用 x 初始化 y，尔后将 x 的值置入 y，代码必须如下撰写：

```
std::atomic<int> y(x.load());          // 读取 x

y.store(x.load());                     // 再次读取 x
```

这段代码可以通过编译，但是，读取 x（经由 x.load()）是个独立于初始化或存储到 y 的函数调用这一事实清楚地表明，没有理由去期望这两条语句中的任何一条可以整体作为单一的原子操作执行。

给定上述代码的前提下，编译器可以通过将 x 的值存储在寄存器中，而不是两次读取，以“优化”之：

```
register = x.load();                   // 将 x 读入寄存器

std::atomic<int> y(register);          // 以寄存器值初始化 y

y.store(register);                     // 将寄存器值存储入 y
```

结果正如你所见，x 的读取操作只执行了一次，这是在处理特种内存时必须避免的那种优化（该优化在 volatile 变量上不被允许）。

现在事情应该明确了：

- std::atomic 对于并发程序设计有用，但不能用于访问特种内存。
- volatile 对于访问特种内存有用，但不能用于并发程序设计。

由于 std::atomic 和 volatile 是用于不同目的，它们甚至可以一起使用：

```
volatile std::atomic<int> vai;          // 针对 vai 的操作是原子的，
                                        // 并且不可以被优化掉
```

如果 vai 对应于由多个线程同时访问的内存映射 I/O 位置，就可能会是有用的。

最后，有些开发人员更喜欢使用 std::atomic 的 load 和 store 成员函数，即使并非必要，因为这样做可以在源代码中明确地表示所涉及的变量并非“常规”。强调这一事实，也并非没有理由。访问 std::atomic 型别对象通常比访问非 std::atomic 型别对象慢得多，我们已经看到 std::atomics 型别对象在使用过程中会阻止编译器对某些类型的代码重新排序，而这样的重新排序在其他情况下是被允许的。召唤 std::atomic 型别对象的加载和存储有助于识别出阻碍潜在的可伸缩性之处。从正确性角度来看，如果本来想要通过某个变量将信息传达到其他线程，却未见它调用 store（例如，一个指示数据可用性的标志位），就可能意味着该变量本来应该声明为 std::atomic，却没有这样做。

这在很大程度上是一个代码风格问题，因此，这与在 std::atomic 和 volatile 之间进行的选择有着非常不同的性质。

要点速记

- std::atomic 用于多线程访问的数据，且不用互斥量。它是撰写并发软件的工具。
- volatile 用于读写操作不可以被优化掉的内存。它是在面对特种内存时使用的工具。

第8章

微调

C++中的每一项通用技术或特性，都会在某些情况下适用，而在另一些情况下则不适用。一般而言，描述通用技术或特性的适用情况才顺理成章，但本章却反其道而行之，描述了两种例外情况。通用技术指的是按值传递，而通用特性则是置入(emplacement)。欲作出是否采用它们的决定，需面对诸多影响因素，而我能够提供的最佳建议则是需要考虑它们其实是有适用场景的。话虽如此，这两者可都是高效现代C++程序设计的重要角色，下面的条款为在评价它们是否适用你的软件这方面提供了参考信息。

条款41：针对可复制的形参，在移动成本低并且一定会被复制的前提下，考虑将其按值传递

有些函数的形参本来就是打算拿来复制的。[注1] 例如，成员函数addName可能会将其形参复制入其私有容器。为效率计，这样的函数应该针对左值实参实施复制，而针对右值实参实施移动：

```
class Widget {
public:
  void addName(const std::string& newName)        // 接受左值
  { names.push_back(newName); }                    // 对其实施复制

  void addName(std::string&& newName)              // 接受右值
  { names.push_back(std::move(newName)); }         // 对其实施移动
  ...                                              // 关于std::move的用法，参见条款25
```

注1：本条款中，针对形参实施的“复制”，意思是将其用作复制或移动操作的源。回忆在本书前言提及的，经由复制操作而得到的副本，和经由移动操作而得到的副本，在C++中并没有不同的术语来区分这两者。

```
private:
  std::vector<std::string> names;
};
```

这样写也没有错，但是要求撰写本质上在做同一件事情的两个函数。这么一来可就有活干了：需要撰写两份函数声明、两份函数实现、两份函数文档、两份函数维护工作量。啊哟，要命。

犹有进者，在目标代码中将出现两个函数，如果你确实在意程序足迹[译注1]一事，这一情况就可能会让你忧心。在本例情况下，这两个函数都可能会被实施内联，而这很可能会消除与存在两个函数相关的任何膨胀问题，但是如果这些函数没有处处实施内联的话，那么你在目标代码中就真会看到有两个函数了。

另一种方法是把 addName 写成接受万能引用的函数模板（参见条款 24）：

```
class Widget {
public:
  template<typename T>                          // 接受左值
  void addName(T&& newName)                     // 也接受右值
  {                                             // 对左值实施复制
    names.push_back(std::forward<T>(newName)); // 对右值实施移动
  }                                             // 关于 std::forward 的用法，参见条款 25

  ...

};
```

这减少了你需要着手处理的源代码数量，但是万能引用的使用会导致其他方面的复杂性。作为模板，addName 的实现通常必须置于头文件中。它还可能在对象代码中产生好几个函数，因为它不仅针对左值和右值会产生不同的实例化结果，针对 std::string 和可以转型为 std::string 的型别（参见条款 25）也会产生不同的实例化结果。同时，有些型别不能按通用引用方式传递（参见条款 30），如果客户传入了不正确的实参型别，编译器错误消息可能会吓人一跳（参见条款 27）。

如果有一种方法来撰写像 addName 这样的函数，针对左值实施的是复制，针对右值实施的是移动，而且无论在在源代码和目标代码中只有一个函数需要着手处理，还能避免万能引用的怪癖，那该多妙！无巧不成书，方法还真有一个。你所需要做的事情只有一件，就是要放弃你可能身为 C++ 程序员学到的第一条规则。该规则说，要避免按值传递用户定义型别的对象。对于像 addName 这样的函数中的 newName 这样的形参，按值传递可能是个完全合理的策略。

译注 1：程序足迹（program footprint）通常是指目标代码经常性占用内存的尺寸（动态分配的内存、或是从外存加载到内存的部分往往不计在内）。

在我们讨论为什么按值传递可能非常适用于 newName 和 addName 之前，先看看实现长成什么样子：

```
class Widget {
public:
  void addName(std::string newName)              // 既接受左值
  { names.push_back(std::move(newName)); }       // 也接受右值，对后者实施移动

  ...

};
```

这段代码中唯一无法一望即明的部分，是针对形参 newName 实施 std::move。通常情况下，std::move 仅会针对右值引用实施。但在这种情况下，我们确知：①无论调用方传入什么，newName 都对它没有任何依赖，所以更改 newName 不会对调用方产生任何影响；②这次使用 newName 是对它的最后一次使用，所以移动它也不会对函数的其余部分产生任何影响。

只有单个 addName 函数的事实，就已经说明我们做到了避免源代码以及目标代码中的代码重复。我们没有使用万能引用，所以采用这种途径也不会导致头文件膨胀、怪异的失败情形或令人费解的错误消息。但是，这样设计会导致效率问题吗？毕竟我们可是按值传递的哟。会不会发生高昂的成本呢？

在 C++98 中，可以打保票的说，肯定会发生的。无论调用方传入的是什么，形参 newName 都会经由复制构造函数创建。不过，在 C++11 中，newName 仅在传入左值时才会被复制构造。而如果传入的是个右值，它会被移动构造。就像这样：

```
Widget w;

...

std::string name("Bart");

w.addName(name);                          // 在调 addName 时传入左值

...

w.addName(name + "Jenne");                // 在调 addName 时传入右值
                                          // （见下）
```

在 addName 的第一个调用中（即传入的是 name 时），用以初始化形参 newName 的是个左值。所以，对 newName 实施的是复制构造，一如在 C++98 中那样。在第二个调用中，用以初始化 newName 的，则是 std::string 的 operator+ 的调用产生的结果 std::string 型别对象（即字符串附加操作）。那样的对象是个右值，所以，对 newName 实施的是移动构造。

针对左值实施复制，针对右值实施移动，恰如所想。干净利落，对吧？

干净利落，诚哉。但有些指导原则，还是最好牢记心中。回顾一下我们考虑过的addName三版本，就能更容易地想到这个实现方案了：

```
class Widget {                                  // 途径一：
public:                                         // 针对左值和右值重载
  void addName(const std::string& newName)
  { names.push_back(newName); }

  void addName(std::string&& newName)
  { names.push_back(std::move(newName)); }
  ...

private:
  std::vector<std::string> names;
};

class Widget {                                  // 途径二：
public:                                         // 使用万能引用
  template<typename T>
  void addName(T&& newName)
  { names.push_back(std::forward<T>(newName)); }
  ...
};

class Widget {                                  // 途径三：
public:                                         // 按值传递
  void addName(std::string newName)
  { names.push_back(std::move(newName)); }
  ...
};
```

我把前两个版本称为“按引用的途径”，因为它们都是按引用传递形参的。

而下面则是我们考察过的两种调用场景：

```
Widget w;
...
std::string name("Bart");

w.addName(name);                // 传入左值
...
w.addName(name + "Jenne");      // 传入右值
```

现在考虑下成本问题，考察对象是复制和移动操作，考察两个调用场景在我们讨论过的三个addName实现中的每一个中添加一个名字会带来的成本是多少。这个成本会计过程将很大程度上忽视编译器优化掉复制和移动操作的可能性，因为这样的优化既依赖于语境，也依赖于编译器，实际上不会改变分析的本质结果。

- **重载**：无论传入左值还是右值，调用方的实参都会绑定到名字为 newName 的引用上。而这样做不会在复制或移动时带来任何成本。在接受左值的重载版本中，newName 被复制入 Widget::names；在接受左值的重载版本中，newName 被移入 Widget::names。成本合计：对于左值是一次复制，对于右值是一次移动。

- **使用万能引用**：与重载一样，调用方的实参会绑定到引用 newName 上。这是个无成本操作。由于使用了 std::forward，左值 std::string 实参被复制入 Widget::names 中，而右值 std:: string 实参则被移入。传入 std::string 实参的成本合计结果与重载相同：左值对应于一次复制，右值对应于一次移动。

  条款 25 解释过，如果调用方传递的实参并非 std::string 型别，它将被转发到适当的 std::string 构造函数。这么一来，会引发的复制和移动操作可能少到一次都没有。所以，接受万能引用的函数可能具备卓然不群的高效性。不过，这也并不影响本条款中的分析，所以为了简化，我们假定调用者传入的总是 std::string 型别的实参。

- **按值传递**：无论传入的是左值还是右值，针对形参 newName 都必须实施一次构造。如果传入的是个左值，成本是一次复制构造。如果传入的个右值，成本是一次移动构造。在函数体内，newName 需要无条件地移入 Widget::names。这么一来，就得到了成本合计的结果：对左值而言，是一次复制加一次移动；对右值而言，是两次移动。与按引用途径相比，无论是左值和右值，都存在一次额外的移动操作。

回顾一下本条款的标题：

> 对于可复制的形参，在移动成本低并且一定会被复制的前提下，考虑将其按值传递。

使用这样的措辞，其实是有理由的。说具体点儿，有四条理由：

1. 你只是要考虑按值传递。没错，它只要求撰写单个函数。没错，它在目标代码中只生成单个函数。没错，它能够避免万能引用带来的一系列毛病。但不要忘了，它的成本更高一些，并且，下面我们还会展示，在有些情况下，还会产生更多在此尚未讨论的成本。

2. 仅对于可复制的形参，才考虑按值传递。不符合这个要求的形参必然具备只移型别，因为如果它们本来不可复制，但函数却总会创建副本的话，那么就必须经由

移动构造函数来创建该副本。[注2] 回忆一下，按值传递相对于重载而言的优势，就在于若采用按值传递则只需撰写单个函数。但是只移型别却并不需要针对左值型别提供重载版本，因为复制左值需要调用复制构造函数，而只移型别的复制构造函数根本就已经被禁用了。这意味着，只需为右值型别的实参提供支持即可，所以在此情况下，所谓“重载”解决方案只要求一个重载版本：那个接受右值引用型别的重载版本。

考虑一个类，它含有一个 std::unique_ptr 型别的数据成员和一个针对它的设置器。 std::unique_ptr 是个只移型别，所以其设置器虽然采用了“重载”途径，却只由单个函数组成：

```
class Widget {
public:
  ...
  void setPtr(std::unique_ptr<std::string>&& ptr)
  { p = std::move(ptr); }

private:
  std::unique_ptr<std::string> p;
};
```

调用方可能以这样的方式使用它：

```
Widget w;
...

w.setPtr(std::make_unique<std::string>("Modern C++"));
```

在这里，从 std::make_unique（参见条款 21）返回的右值 std::unique_ptr<std::string> 会以右值引用方式传递给 setPtr，在那里它被移入数据成员 p。总成本是一次移动。

如果 setPtr 以按值传递方式来接受形参：

```
class Widget {
public:
  ...

  void setPtr(std::unique_ptr<std::string> ptr)
  { p = std::move(ptr); }

  ...
};
```

注 2：正是因为不得不写出这样的句子，才会感觉如果可以有术语用以区分经由复制操作和移动操作制作的副本，该有多好啊。

同一调用会导致针对形参 ptr 实施移动构造后，再将 ptr 移入数据成员 p。这么一来，总成本成了两次移动，比“重载”途径翻了一番。

3. 按值传递仅在形参移动成本低廉的前提下，才值得考虑。只有当移动成本低廉时，一次移动带来的额外成本才可能是可以接受的，但如果这个前提都不成立，那么执行不必要的移动就和执行不必要的复制没有区别了。而避免不必要的复制操作的重要性，也正是 C++98 中“要尽量避免按值传递”这条金科玉律的出发点。

   你应该只针对一定会被复制的形参才考虑按值传递。欲理解为何这一点很重要，假设在将其形参复制到容器 names 之前，addName 会先检查该新名字是否太短或太长。如果太短或太长，则忽略添加该名字的请求。按值传递的实现可能会写成这样：

```
class Widget {
public:
  void addName(std::string newName)
  {
    if ((newName.length() >= minLen) &&
        (newName.length() <= maxLen))
      {
        names.push_back(std::move(newName));
      }
  }
  ...

private:
  std::vector<std::string> names;
};
```

   即使没有向 names 添加任何内容，该函数也会招致构造和析构 newName 的成本。而如果采用了按引用途径，就不必为此买单。

即使你面对的函数确实是在针对可复制型别实施无条件复制，并且移动成本也低廉，还是存在不适合采用按值传递的一些情况。原因在于，函数可以经由两种方式来实施复制：经由构造（即复制构造或移动构造），以及经由赋值（即复制赋值或移动赋值）。addName 采用的是构造方式：其参数 newName 被传递给 vector::push_back，并且在该函数内，newName 被复制构造入 std::vector 末尾所创建的新元素中。对于使用构造来实施形参复制的函数，我们前面的分析已经是完整的了：使用值传递的话，无论传入的左值还是右值，都会招致一次额外移动所带来的成本。

如果采用赋值来实施形参复制的话，情况就更复杂了。例如，假定我们有个表示密码的类。由于密码可以更改，我们提供一个设置器函数 changeTo。在采用按值传递策略的前提下，我们可能会像这样实现 Password：

```
class Password {
public:
  explicit Password(std::string pwd)    // 按值传递
  : text(std::move(pwd)) {}             // 对 text 实施构造

  void changeTo(std::string newPwd)     // 按值传递
  { text = std::move(newPwd); }         // 对 text 实施赋值

  ...
private:
  std::string text;                     // 表示密文
};
```

以明文的形式存储密码会让软件安全特勤组暴怒，但先不说这个，考虑下面的代码：

```
std::string initPwd("Supercalifragilisticexpialidocious");

Password p(initPwd);
```

这段代码并无意外：`p.text` 采用给定密码构造，而在构造函数中采用按值传递会招致 `std::string` 的移动构造成本，该成本在采用重载或完美转发时不会发生。一切运行如仪。

不过，该程序的某个用户可能会感觉初始密码不尽如人意。因为“Supercalifragilistic-expialidocious”是可以在许多字典中直接找到的。因此，他或她可能会采取行动，造成等价于以下代码加以执行的结果：

```
std::string newPassword = "Beware the Jabberwock";

p.changeTo(newPassword);
```

新密码与旧密码孰优孰劣，这是个大可以留给用户去争论的问题。我们所面临的问题则是 `changeTo` 采用了赋值来对形参 `newPwd` 实施复制，这有可能导致该函数的按值传递策略带来爆发式的成本。

但在本例情况下，旧密码（“Supercalifragilisticexpialidocious”）比新密码（“Beware the Jabberwock”）更长，所以不需要实施任何内存分配和回收。如果采用重载途径，则很可能不会发生任何动态内存管理行为：

```
class Password {
public:
  ...

  void changeTo(const std::string& newPwd)       // 为左值而准备的重载
  {

    text = newPwd;     // 在下式成立的前提下，可以复用 text 的内存
                       // text.capacity() >= newPwd.size()
```

```
  }
...

private:
  std::string text;   // 同上
};
```

在此场景下，按值传递的代价会包括额外的内存分配和回本成本，该成本可能会比 std::string 移动操作的成本高出几个数量级。

有意思的是，如果旧密码比新密码短，在赋值过程中一般而言不可能避免分配—回收这对动作，在此情况下，按值传递可能与按引用传递有着大致相同的运行速度。也就是说，基于赋值的形参复制成本有可能取决于参与赋值的对象的取值！这一分析结果适用于可能在动态分配的内存中持有值的任何形参型别。不是所有的型别都符合这一特征，但很多确实符合，这其中就包括 std::string 和 std::vector。

这样的潜在成本增加通常只在传入左值实参时才会发生，因为实施内存分配和回收的需求通常仅在实施真正的复制操作（即，非移动操作）时才会发生。对于右值实参而言，几乎总是只要移动就足够了。

总而言之，采用赋值方式复制形参的函数，其按值传递带来的额外成本取决于传入的型别、左值和右值实参的占比、型别是否使用动态分配内存，还有，在确实使用了动态内存的前提下，该型别的赋值运算符如何实现，以及与赋值目标相关联的内存是否至少与赋值源相关联的内存尺寸相当的可能性高低。对于 std::string 而言，还要取决于实现是否使用了小型字符串优化（small string optimization，SSO，参见条款29），还有，如果确实使用了 SSO，所赋之值是否放得进 SSO 缓冲区。

所以，一如我前面说过的，当通过赋值实施形参复制时，进行按值传值的成本分析会是复杂的。通常情况下，最实用的途径是采取“无罪推定，除非证明有罪”的策略。亦即，总是采用重载或万能引用而非按值传递，除非已确凿地证明按值传递能够为所需的形参型别生成可接受效率的代码。

既然如此，对于那些必须运行得尽可能快的软件来说，按值传递可能并非可行的策略，因为也许即使移动成本低廉，可能避免它们仍然重要。更何况，其实会发生多少移动操作并不非常清晰。在 Widget::addName 例子中，按值传递只会招致一次额外的移动操作，但是假设 Widget::add 会先调用 Widget::validateName，并且调用后者时也按值传递（它可能会有个总是复制其形参的理由，例如，将其存储在某个所有待验证的值构成的数据结构中）。并假设 validateName 会调用第三个也要求按值传递的函数

……

你应该可以看出来，这样下去会往什么方向发展。当存在函数调用链时，如果每个函数都出于“只不过耗费一次低成本的移动”而选用了按值传递的话，则整个调用链的成本可能会超过你能容忍的范围。而如果使用的是按引用的形参传递，则调用链不会招致这种累积性的开销。

还有一个与效率无关，但仍需要之牢记的议题是，不同于按引用传递，按值传递较容易遭遇切片问题。这个问题在 C++98 中已是老生常谈，所以我不会展开，但是如果你有个函数被设计用以接受一个基类型别或从它派生的任何型别的形参，你肯定不会想要声明该型别的按值传递形参，因为传入的任何可能的派生型别对象的派生类特征都将被“截切”掉：

```
class Widget { ... };                          // 基类

class SpecialWidget: public Widget { ... };    // 派生类

void processWidget(Widget w);                  // 为任何 Widget 而设计的函数
                                               // 包括派生型别
                                               // 会受到切片问题的侵害
...

SpecialWidget sw;

...

processWidget(sw);                  // processWidget 看到的只是
                                    // 一个 Widget 而非 SpecialWidget 型别的对象！
```

如果你对切片问题尚不熟悉，搜索引擎和互联网是你的朋友，那上面的相关信息汗牛充栋。你会了解到，切片问题的存在是按值传递在 C++98 中如此声名狼藉的另一原因（比影响性能更甚）。为何你在初学 C++ 程序设计时就会被告知的事项之一，就是避免按值传递用户自定义型别对象，这也不无理由。

C++11 并没有从根本上颠覆 C++98 关于按值传递的智慧。一般地，按值传递仍会导致一些你本想避免的性能损失，按值传递还会导致切片问题。C++11 引入的新特性是左值和右值的区别对待。欲实现函数以利用可复制右值型别的移动语义，就需要重载或使用万能引用两者之一，但这两者都有一定的缺点。对于可复制的、移动成本低廉的型别，并且传入的函数总是对其实施复制这种特殊情况，在切片问题也无须担心的前提下，按值传递可以提供一个易于实现的替代方法，它和按引用传递的竞争对手效率相近，但是避免了它们的不足。

要点速记

- 对于可复制的、在移动成本低廉的并且一定会被复制的形参而言，按值传递可能会和按引用传递的具备相近的效率，并可能生成更少量的目标代码。
- 经由构造复制形参的成本可能比经由赋值复制形参高出很多。
- 按值传递肯定会导致切片问题，所以基类型别特别不适用于按值传递。

条款 42：考虑置入而非插入

如果你有个容器，持有一些，比如说，std::string 型别的对象吧，那么似乎合乎逻辑的做法是经由某个插入函数（insertion function）来向其添加新元素（比如 insert、push_front 和 push_back，又或者对于 std::forward_list 而言的 insert_after），而你传递给函数的元素型别将是 std::string。毕竟，那正是容器持有之物的型别。

说合乎逻辑倒也确实合乎逻辑，但却不一定合乎事实。考虑这段代码：

```
std::vector<std::string> vs;  // 持有 std::string 型别对象的容器

vs.push_back("xyzzy");        // 添加字符串字面量
```

这里，容器持有的乃是 std::string 型别对象，但是你手头上有的（你实际上想要实施 push_back 的）是个字符串字面量，即，一对引号内的一串字符。字符串字面量不是 std::string，也就是说，传递给 push_back 的实参并非容器持有之物的型别。

std::vector 的 push_back 针对左值和右值给出了不同的重载版本如下：

```
template <class T,                    // 引自 C++11 标准
          class Allocator = allocator<T>>
class vector {
public:
  ...
  void push_back(const T& x);         // 插入左值
  void push_back(T&& x);              // 插入右值
  ...
};
```

在下面的调用语句中：

```
vs.push_back("xyzzy");
```

编译器会看到实参型别（const char [6]）与 push_back（std::string 的引用型别）接受的形参型别之间的不匹配。而解决之道就是通过生成代码以从字符串字面量出发创建 std::string 型别的临时对象，并将该临时对象传递给 push_back。换言之，他们把这句调用看作下面这样：

```
vs.push_back(std::string("xyzzy"));     // 创建 std::string 型别的临时对象
                                        // 并将其传递给 push_back
```

这段代码顺利通过编译并且运行无误，所有人都开心地打卡下班了。所有人，除了那些性能狂人们，正是他们，认识到这段代码不像它应有的那样高效。

为了在持有 std::string 型别对象的容器中创建一个新的元素，他们明白，调用一次 std::string 的构造函数是应该的，但是上面的代码不会只发生一次构造函数调用。实际上，发生了两次。犹有进者，还发生了一次 std::string 的析构函数调用。以下是对 push_back 的调用在运行期执行的全部动作：

1. 从字符串字面量“xyzzy”出发，创建 std::string 型别的临时对象。该对象没有名字，我们称其为 *temp*。针对 *temp* 实施的构造，就是第一次的 std::string 构造函数的调用。因为是个临时对象，所以 *temp* 是个右值。

2. *temp* 被传递给 push_back 的右值重载版本，在那里它被绑定到右值引用形参 x。然后，会在内存中为 std::vector 构造一个 x 的副本。这一次的构造（已经是第二次了）结果就是在 std::vector 内创建了一个新的对象（用于将 x 复制到 std::vector 中的构造函数，是移动构造函数，因为作为右值引用的 x，在复制之前被转换成了右值。有关将右值引用形式强制转型到右值的相关信息，参见条款 25）。

3. 紧接着 push_back 返回的那一时刻，*temp* 就被析构，所以，这就需要调用一次 std::string 的析构函数。

性能狂人们肯定会不由自主地注意到，如果有什么办法能将字符串字面量直接传递给上述步骤 2 中那段在 std::vector 内构造 std::string 型别对象的代码，我们也就可以避免先构造再析构 *temp*。这么一来，就可以达到效率最大化，即便是性能狂人也能安心打卡下班了。

由于你是名 C++ 程序员，所以你比起其他人群成为性能狂人的概率更高。即使你不是个性能狂人，你也很可能会认同他们的观点（如果你对性能不以为意，难道你不该出门左转到 Python 之家去吗）。所以，我很乐意告诉你，确实有个办法能够使得 push_

back 调用过程的效率最大化。办法就是不要调用 push_back。调用 push_back 是弄错了函数。符合要求的函数是 emplace_back。

emplace_back 完全符合期望：它使用传入的任何实参在 std::vector 内构造一个 std::string。不会涉及任何临时对象：

```
vs.emplace_back("xyzzy");      // 直接从 "xyzzy" 出发在 vs 内
                               // 构造 std::string 型别对象
```

emplace_back 使用了完美转发，所以只要你没有遭遇完美转发的限制（参见条款 30），就可以通过 emplace_back 传递任意型别的任意数量和任意组合的实参。例如，如果你想通过 std::string 的那个接受一个字符及重复计数的构造函数重载版本来创建一个 std::string 型别对象，下面的代码就可以做到：

```
vs.emplace_back(50, 'x');      // 插入一个由 50 个 'x' 字符组成的 consisting
                               // std::string 型别对象
```

emplace_back 可用于任何支持 push_back 的标准容器。相似地，所有支持 push_front 的标准容器也支持 emplace_front。还有，任何支持插入操作（亦即，除了 std::forward_list 和 std::array 以外的所有标准容器）都支持置入操作。关联容器提供了 emplace_hint 来补充它们带有 "hint" 迭代器的 insert 函数，而 std::forward_list 也有着 emplace_after 与其 insert_after 一唱一和。

使得置入函数超越插入函数成为可能的，是置入函数更加灵活的接口。插入函数接受的是待插入对象，而置入函数接受的则是待插入对象的构造函数实参。这一区别就让置入函数得以避免临时对象的创建和析构，但插入函数就无法避免。

因为具备容器所持有之物型别的实参可以被传递给一个置入函数（嗣后该实参数会引发函数执行复制或者移动构造），所以即使在插入函数并不要求创建临时对象的情况下，也可以使用置入函数。在那种情况下，插入函数和置入函数本质上做的是同一件事。例如，给出下面的 std::string 型别对象：

```
std::string queenOfDisco("Donna Summer");
```

下面两个调用语句都成立，并且对于容器来说净效果相同：

```
vs.push_back(queenOfDisco);          // 在 vs 的尾部复制构造了 queenOfDisco

vs.emplace_back(queenOfDisco);       // 同上
```

这么一来，置入函数就能做到插入函数所能做到的一切。有时，他们可以比后者做得

更高效一些，而且，至少在理论上，它们应该不会比后者效率更低。既然如此，何不总是使用置入函数呢？

因为，从理论上说，理论和实践没有区别；但从实践上说，理论和实践是有区别的。从标准库的当前实现情况来看，在有些情况下，正如预期的那样，置入的性能优于插入，但是，不幸的是，还是存在插入函数运行得更快的情况。后面一种情况不太容易表征，因为具体来说，取决于传递的实参型别、使用的容器种类，请求插入或置入的容器位置，所持有型别构造函数的异常安全性，还有，对于禁止出现重复值的容器（亦即，std::set、std::map、std::unordered_set 和 std::unordered_map）而言，容器中是否已经存在要添加的值。所以，这里适用一般的性能调优建议：欲确定置入或插入哪个运行得更快，需对两者实施基准测试。

这样的说法当然不尽如人意，所以你肯定会很高兴地得知，有一种启发式思路可以帮助确定置入功能在哪些情况下极有可能值得一试。如果下列情况都成立，那么置入将几乎肯定会比插入更高效：

- **欲添加的值是以构造而非赋值方式加入容器。**本条款启首一例（即从值“xyzzy”出发创建 std::string 型别对象，并加入 std::vector），就能看出该是值被添加到了 vs 的结尾，该位置尚不存在对象。是故，新值必须以构造方式加入 std::vector。若我们修订此例，使得新的 std::string 型别对象去到某个已被某对象占据的位置，则完全变成另一回事了。考虑下面的代码：

```
std::vector<std::string> vs;        // 同前
…                                   // 向 vs 中添加元素
vs.emplace(vs.begin(), "xyzzy");    // 向 vs 的开头添加“xyzzy”
```

对于这段代码，很少会有实现是将待添加的 std::string 在由 vs[0] 占用的内存中实施构造。这里一般会采用不同的手法，即移动赋值的方式来让该值就位。但既然是移动赋值，总要有个作为源的移动对象，这也就意味着需要创建一个临时对象作为移动的源。由于置入相对于插入的主要优点就在于既不会创建也不会析构临时对象，那么当添加的值是经由赋值放入容器的时候，置入的边际效用也就趋于消失了。

唉，向容器中添加值究竟是通过构造还是赋值，这一般取决于实现者。但是，启发式思维会再一次派上用场。

基于节点的容器几乎总是使用构造来添加新值，而大多数标准容器都是基于节点的。仅有的一些例外是 std::vector，std::deque 和 std::string（std::array

也不基于节点，但它根本不支持插入或置入，所以和这里的讨论不相干）。在非基于节点的容器中，可以可靠地说 emplace_back 是使用构造非赋值来将新值就位的，而这一点对于 std::deque 的 emplace_front 来说也一样成立。

- 传递的实参型别与容器持有之物的型别不同。一次看出，置入相对于插入的优势通常植根于这样一个事实，即当传递的实参型别并非容器持有之物的型别时，其接口不要求创建和析构临时对象。当型别为T的对象被添加到container<T>中时，没有理由期望置入的运行速度会比插入快，因为并不需要为了满足插入的接口去创建临时对象。

- 容器不太可能由于出现重复情况而拒绝待添加的新值。这意味着或者是容器允许重复值，或者所添加的大部分值都满足唯一性。这个因素之所以值得一提，是因为，欲检测某值是否已经在容器中，置入的实现通常会使用该新值创建一个节点，以便将该节点的值与容器的现有节点进行比较。如果待添加的值尚不在容器中，则将节点链入该容器。但是，如果该值业已存在，则置入就会中止，节点也就会实施析构，这意味着其构造和析构的成本是被浪费掉了。这样的节点会更经常地为置入函数，而非插入函数创建。

下述前面已列出的调用语句满足所有的判断准则。它们就比对应的 push_back 调用要运行得更快。

```
vs.emplace_back("xyzzy");      // 在容器尾部以构造新值
                               // 传递和型别有异于容器持有之物的型别
                               // 未使用拒绝重复值的容器

vs.emplace_back(50, 'x');      // 同上
```

在决定是否选用置入函数时，还有其他两个问题值得操心。第一个和资源管理有关。假定你有个持有 std::shared_ptr<Widget> 型别对象的容器。

```
std::list<std::shared_ptr<Widget>> ptrs;
```

而你想向其添加一个需要通过自定义删除器来释放的（参见条款 19）std::shared_ptr。条款 21 解释过，只要可行，你就应该使用 std::make_shared 来创建 std::shared_ptr 型别的对象，但它同时也认可，在某些情况下你无法这样做。而其中之一，就是需要指定自定义删除器的情况。在此情况下，就必须直接使用 new 来获取裸指针以备 std::shared_ptr 加以管理。

如果自定义删除器就是下面这个函数：

```
void killWidget(Widget* pWidget);
```

选用了插入函数的代码就应该长成这样：

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

也可能长成这样，但意义相同：

```
ptrs.push_back({ new Widget, killWidget });
```

不管写成上面两者中的哪样，在调用 push_back 之前，都会创建一个 std::shared_ptr 型别的临时对象。push_back 的形参是个 std::shared_ptr 型别的引用，所以必须存在一个 std::shared_ptr 型别对象来让该形参指涉到。

如果选用了 emplace_back，本可以避免创建 std::shared_ptr 型别的临时对象，但是在本例的情况下，该临时对象带来的收益远超其成本。考虑下面这个有可能会发生的事件序列：

1. 上述两个调用语句无论哪个都会构造一个 std::shared_ptr<Widget> 型别的临时对象，用以持有从“new Widget”返回的裸指针。该对象暂称为 temp。

2. push_back 会按引用方式接受 *temp*。在为链表节点分配内存以持有 *temp* 的副本的过程中，抛出了内存不足的异常。

3. 该异常传播到了 push_back 之外，*temp* 被析构。作为给 Widget 兜底的、指涉到它并对其施加管理的 std::shared_ptr<Widget> 型别对象会自动释放该 Widget，在本例的情况下会调用 killWidget 达成该目的。

即使发生异常，也没有资源泄漏。在 push_back 的调用过程中，从“new Widget”出发构造的 Widget，会在为管理它创建的 std::shared_ptr（*temp*）的析构函数中得到释放。岁月静好。

现在考虑一下，如果调用的是 emplace_back，而不是 push_back，会发生什么：

```
ptrs.emplace_back(new Widget, killWidget);
```

1. 从“new Widget”返回的裸指针被完美转发，并运行到 emplace_back 内为链表节点分配内容的执行点。尔后该内存分配失败，并抛出了内存不足的异常。

2. 该异常传播到了 emplace_back 之外，作为唯一可以获取堆上 Widget 的抓手的裸指针，却丢失了。那个 Widget（连同他拥有的任何资源）都发生了泄漏。

在这种场景下，岁月不再静好，并且故障不能归罪于 std::shared_ptr。即使换用 std::unique_ptr 的自定义删除器，同样的问题仍然可能会现身。从根本上讲，像

std::shared_ptr和std::unique_ptr这样的资源管理类若要发挥作用，前提是资源（比如从new出发的裸指针）会立即传递给资源管理对象的构造函数。std::make_shared和std::make_unique这样的函数会把这一点自动化，这个事实正是为何它们如此重要的原因之一。

在调用持有资源管理对象的容器（例如，std::list<std::shared_prt<Widget>>）的插入函数时，函数的形参型别通常能确保在资源的获取（例如，运用new）和对资源管理的对象实施构造之间不再有任何其他动作。而在置入函数中，完美转发会推迟资源管理对象的创建，直到它们能够在容器的内存中构造为止。这开了一个“天窗”，其中就会因异常而导致资源泄漏。所有的标准容器对此都在劫难逃。在处理持有资源管理对象的容器时，必须小心确保在选用了置入而非插入函数时，不会在提升了一点代码效率的同时，却因异常安全性的削弱而赔得精光。

坦率地说，绝不应该把像“new Widget”这样的表达式传递给emplace_back、push_back或者大多数其他函数，因为正如条款21所解释过的，这可能会导致我们刚才所讨论的异常安全问题。要把这扇门关闭，就需要从“new Widget”中获取指针并将其在独立语句中转交给资源管理对象，然后将该对象作为右值传递给你最初想要向其传递“new Widget”的函数（条款21涵盖了该技术的更多细节）。所以，选用了push_back的代码应该这样写：

```
std::shared_ptr<Widget> spw(new Widget,          // 构造Widget并用spw管理它
                            killWidget);

ptrs.push_back(std::move(spw));                  // 以右值形式添加spw
```

选用了emplace_back的版本十分类似：

```
std::shared_ptr<Widget> spw(new Widget, killWidget);
ptrs.emplace_back(std::move(spw));
```

无论选用哪个，这个途径都会产生了构造和析构spw的成本。考虑选用置入而非插入的动机就在于避免容器所持有之物型别的临时对象的成本，而这成本正是spw所体现的概念，当你向容器中添加的是资源管理对象，并遵循了正确的做法以确保在资源的获取和将其移交给资源管理对象之间没有任何多余动作的话，置入函数的性能表现就不太会在此情形下仍然超越插入函数。

置入函数第二个值得一提的方面，是它们与带有explicit声明饰词的构造函数之间的互动。假设你为了表彰C++11对正则表达式的支持，构造了一个正则表达式对象的容器：

```
std::vector<std::regex> regexes;
```

由于被你的同事们为了“每天该上多少次社交网站才最理想”的争论而分了心，你无意间写下了下面这句看似无意义的代码：

```
regexes.emplace_back(nullptr);          // 向持有正则表达式的容器
                                        // 添加了一个 nullptr？
```

你在输入时没有注意到该错误，而编译器也一声不吭地接受了代码，最终你浪费了大把时间来调试。找了半天，你终于发现自己在正则表达式容器中插入了一个空指针。但这怎么可能呢？指针根本不是正则表达式啊，而且如果你试图这样做：

```
std::regex r = nullptr;         // 错误！无法通过编译
```

编译器会拒绝你的代码。有意思的是，如果你选用的是 push_back 而非 emplace_back，编译器同样也会拒绝你的代码：

```
regexes.push_back(nullptr);     // 错误！无法通过编译
```

这种令人好奇的行为源自 std::regex 对象可以从字符串出发来构造这一事实。这就使得下面这样有用的代码成为合法的：

```
std::regex upperCaseWord("[A-Z]+");
```

从字符串出发来构造 std::regex 型别对象，肯定会导致相对较高昂的运行期成本，因此，为了尽可能地减少无意间招致这种开销的可能性，接受 const char * 指针的 std::regex 构造函数以 explicit 饰词声明。这就是为何下面几个语句都通不过编译：

```
std::regex r = nullptr;         // 错误！无法通过编译

regexes.push_back(nullptr);     // 错误！无法通过编译
```

在两种情况下，我们都要求了一次从指针到 std::regex 的隐式型别转换，而由于该构造函数带有 explicit 声明饰词，这样的型别转换被阻止了。

然而，在 emplace_back 的调用过程中，我们却没有声称传递的是个 std::regex 对象。取而代之的是，我们向 std:regex 对象传递的是个构造函数实参。这不但不被视作隐式型别转换的请求，反而在编译器看来是等同于写了这样的代码：

```
std::regex r(nullptr);          // 能编译
```

如果简短的注释“能编译”暗示着一种无精打采，那是好事，因为这段代码虽然能通过编译，但却有着未定义行为。接受 const char * 指针的 std::regex 构造函数要求指涉到的字符串包含一个有意义的正则表达式，而空指针并不符合该要求。如果你编

写并编译这样的代码，你能指望的最好结果就是它在运行时崩溃。如果你不那么走运，你和你的调试器可能就得亲密好一阵子了。

先把 push_back、emplace_back 和亲密什么的按下不表，单注意下面几个非常相似的初始化语法如何产生了不同的结果：

```
std::regex r1 = nullptr;        // 错误！无法通过编译

std::regex r2(nullptr);         // 能编译
```

用标准中的官方术语来说，用于初始化 r1（采用等号）的语法对应于所谓复制初始化。相对地，用于初始化 r2 的语法（使用括号，尽量也可以使用花括号代替）会产生所谓的直接初始化。复制初始化是不允许调用带有 explicit 声明饰词的构造函数的，但直接初始化就允许。是故，对 r1 实施初始化的那一行通不过编译，但对 r1 实施初始化的那一行就可以通过编译。

但是，回过头来再说 push_back 和 emplace_back，或更一般地对插入函数和置入函数作对比。置入函数使用的是直接初始化，所以它们就能够调用带有 explicit 声明饰词的构造函数。而插入函数使用的是复制初始化，它们就不能调用带有 explicit 声明饰词的构造函数。因此：

```
regexes.emplace_back(nullptr);          //  能编译，直接初始化允许使用
                                        // 接受指针的、带有 explicit 声明饰词的
                                        // std::regex 构造函数

regexes.push_back(nullptr);             // 错误！复制初始化禁止使用
                                        // 那个构造函数
```

这里得到的教训是，在使用置入函数时，要特别小心去保证传递了正确的实参，因为即使是带有 explicit 声明饰词的构造函数也会被编译器纳入考虑范围，因为它会尽力去找到某种方法来解释你的代码以使得它合法。

要点速记

- 从原理上说，置入函数应该有时比对应的插入函数高效，而且不应该有更低效的可能。
- 从实践上说，置入函数在以下几个前提成立时，极有可能会运行得更快：①待添加的值是以构造而非赋值方式加入容器；②传递的实参型别与容器持有之物的型别不同；③容器不会由于存在重复值而拒绝待添加的值。
- 置入函数可能会执行在插入函数中会被拒绝的型别转换。