Ada 95 品质和风格

专业程序员的准则

原著: (美) 国防部 Ada 联合项目办公室
翻译: 朱群英

中文修订版: 0.2, 2013 年 3 月 25 日

本书使用 XeTeX 和 xeCJK 进行排版。LaTeX 源码可从 http://code.google.com/p/ada95qs-zh/获得。

Ada 95 Quality and Style:
Guidelines for Professional Programmers

SPC-94-03-CMC

Version 0.1.00.10

October 1995

Prepared for
Department of Defense Ada Joint Program Office

Poduced by the
SOFTWARE PRODUCTIVITY CONSORTIUM

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

译者序

第一次知道 Ada 这个程序语言是在大学读书时，那时对各种计算机语言感兴趣，在图书馆里找各种语言类的书看。Ada 进入了我的视野，她为我打开了另一扇窗。虽然其后的工作中主要都是用 C，Ada 的一些精神也影响了我的 C 程序，从中借鉴了一些好的方法。尽管 Ada 的功能很强大，因为各种原因一直没有被广泛的使用，希望通过这个翻译可以引起更多人的兴趣。

由于翻译中难免有些词不达意的地方，欢迎各位读者指正。请把您的批评和建议发到电邮：zhu.qunying@gmail.com。

朱群英
2009 年于温哥华

前言

目的

本书[1]的目的是帮助电脑专业人员确立一套风格准则从而写出更高好的 Ada 95 程序。这套风格准则将会直接影响他们程序的代码质量。本风格指南并不打算代替 Ada 参考手册 (1995) 或原理 (1995)，也不是 Ada 95 程序语言的教程和从 Ada 83 过渡到 Ada 95 的指南。关于这些问题，请读者查阅相关的资料。

本指南根据程序员为了写出高质量、可靠、可重用、可移植的 Ada 程序，所做出的各个重要决定，进行分章描述。章节中某些部分会有所重复，因为不是所有的决定都可以独立做出。本书对源代码的呈现、可读性、程序结构、并行处理、可移植性、可重用性、性能以及面向对象进行了分章阐述。

每一章都分为若干准则，内容规约而具可塑性，可广泛的应用。每个准则都简明叙述了应遵守的规则以及从原理上解释其重要性。准则也提供了范例，以及可能的异例。许多准则都非常明确，可以作为公司或者项目的编程标准。其他的需要管理上对于某个具体事例决定是否可作为标准，这种情况，书中都会用一个样板的具体事例来说明和使用。

背景

Ada 联合项目办公室 (Ada Joint Program Office, AJPO) 资助了本风格指南。对原来支持 Ada 83 的《Ada 品质和风格：专业程序员的准则》版本 02.01.01 (AQ&S 83) (软件生产力协会，Software Productivity consortium 1992) 进行了修改和增加了使用 Ada 95 的准则。Ada 95 的准则产生于从 Ada 9X 项目、AJPO 图书馆以及整个 Ada 社区得到的丰富数据。软件生产力协会的技术人员发起了这次更新，高等研究计划署参与了这次努力。

之前的 AQ&S 83 提供了一套准则，帮助程序员依据这些准则来使用 Ada 的各项功能。1992 年，协会在 AJPO 的合约下完成了版本 2.1 的更新。AJPO 这么提及它“所有国防部项目的推荐风格指南。”

公众评论

这本新风格指南期望给新手和有经验的 Ada 程序员提供一个工具。为了到达这个目的，协会直接让公众和 Ada 社区中可联系到的最好的专家参与其中。为了保证这种参与度，本风格指南的完成通过了一个三阶段的过程：完成一个基本的草稿，举行公众和专家的审核，完成最后的风格指南。

协会欢迎对本书的评论，以继续提高其品质和有用性。作者会仔细考虑对现存准则的建议和未来需要扩充的地方的建议。有突出某个问题的例子会更有帮助。

本书的电子版本可以从 Ada 信息交换所 (Ada Information Clearinghouse) 下载 (电话：1 (800) 232-4211；电邮：adainfo@sw-eng.falls-church.va.us)。[2]

请直接发评论给以下地址：

---

[1] 部分内容来自国防部 Ada 联合项目办公室 (Department of Defense Ada Joint Program Office) 资助的研究结果，资金由高等研究计划署 (Advanced Research Projects Agency) 通过奖助金 #MDA972-92-J-1018 提供。本书的内容并不一定代表美国政府的立场和政策，不能因此而推论受到官方的认可。

[2] 译者注：AJPO 已于 1998 年 10 月 1 日关闭。英文资料可由 http://www.adaic.org/docs/95style/ 获得。

Christine Ausnit
Software Productivity Consortium
2214 Rock Hill Road
Herndon, VA 22070
e-mail: ausnit@software.org
fax: (703) 742-7200
或
Kent A. Johnson
Software Productivity Consortium
2214 Rock Hill Road
Herndon, VA 22070
e-mail: johnson@software.org
fax: (703) 742-7200
请在评论中包含您的联系方式。

作者和感谢

协会希望感谢上一版 Ada 83 风格指南中近 100 人的贡献者，包括作者、编辑和卓越的审核者。本书的贡献者有作者：Ms. Christine Ausnit-Hood, Mr. Kent A. Johnson, Mr. Roger G. Pettis, Mr. Steven B. Opdahl, 以及以下的卓越审核者、专家审核者和技术顾问。

卓越审核者： Mr. Bill Beckwith, Objective Interface Systems, Inc.; Dr. Norman H. Cohen, IBM TJWatson Research Center; Dr. Robert Dewar, New York University; Dr. Charles B. Engle, Jr., Department of Computer Science, Florida Institute of Technology; Mr. Jay Ferguson, NSA, Department of Defense; Mr. Ken Garlington, Lockheed Martin, Fort Worth Company; Mr. Tim Harrison, ParcPlace-Digitalk Inc.; Mr. Ed Seidewitz, NASA, Goddard Space Flight Center; Mr. S. Tucker Taft, Intermetrics Inc.

专家审核者： Mr. Brad Balfour, CACI; Dr. Bryce Bardin, Ada Consulting and Training; Mr. Philip Brashear, CTA Inc.; Dr. Ben Brosgol, Brosgol Consulting and Training; Dr. Michael B. Feldman, Department of Electrical Engineering and Computer Science, George Washington University; Mr. Gil Myers, NOSC ; Mr. Jim Moore, The MITRE Corporation; Ms. Eileen S. Quann, FASTRAK Training, Inc.; Mr. Richard Riehle, AdaWorks; Dr. Tim Teitelbaum, GrammaTech; Dr. Joyce Tokar, Tartan.

技术顾问： Mr. John Barnes, John Barnes Informatics; Mr. Leslie Dupaix, OO-ALC/TISE, Hill AFB; Mr. Dave Emery, The MITRE Corporation; Mr. Magnus Kempe, Kempe Software CE; Ms. Judy Kerner, The Aerospace Corporation; Mr. Alexander Miethe, CCI; LtCol Pat Lawlis, AFIT/ENG, Wright-Patterson AFB.

感谢其他转来评论、准则、和范例的贡献者：Lisa Chan, Bo Sanden, Wesley Groleau, Terry D. Humphrey, Pascal Leroy, Gilles Demailly, Philippe Kipfer, Tomas Peterson, Ted Baker, Mike Dingas, Willem Treurniet, T. A. Vo 和 Dave Weller.

特别感谢： Ed Seidewitz, Tim Harrison, Bill Beckwith, Ken Garlington, Tucker Taft, Chuck Engle, 和 Don Reifer, 在百忙中抽出时间参加卓越审核者技术交流会。

陆军研究实验室 (Army Research Lab) 的 Mike Evans 和 Dan Hocking 为卓越审核者技术交流会提供电子支持。

Philip Brashear 重写了第**??**章。

Mike Feldman 和 Brian Kallberg 更新了哲学家吃晚餐的问题。

John Barnes 提供了他新书的摘要。

GrammaTech, Inc., 提供了新一版的 Ada-ASSURED 产品. 大部分范例都是用他们提供的工具进行了格式化。

另外 Bobbie Troy 和 Mary Mallonee 提供了技术编辑; Debbie Morgan 和 Lisa Smith 提供了文档处理；Bobbie Troy 进行了校对。

目录

列表目录

第一章

导言

风格这个写作的重要属性往往被忽略，而写作的风格直接影响最后作品的可读性和可理解性。编程的风格，即使用某个电脑编程语言书写源程序，也受到相当的忽视。程序不仅仅需要被机器读懂，还要被人读并易于理解。这个需求对生产高质量的产品同样重要，即在预算内按时按质完成符合客户要求的产品。本书试图帮助电脑专业人员写出更好的 Ada 程序。本书呈现给读者一套有针对性的风格准则，从而 Ada 95 [ARM 1995] 的强大功能可以在遵守一些规则下使用。

每个**准则**都简明叙述了应遵守的规则以及从**原理**上解释其原因。大多数情况下，都会有一个使用该准则的**范例**，某些情况下，还会有进一步的范例来显示不遵守准则的后果。所有可能的使用上的**异例**都会特别指出，如果可能，会有进一步的**注解**。某些情况下，会使用一个 **具体事例**来进一步呈现更加精确的准则，从而可以作为标准实施。某些特别选出来的情况，**自动化注解**会探讨怎样自动实施这个准则。

Ada 当初被设计为支持开发高质量、可靠、可重用、可移植的软件。由于各种原因，没有程序语言可以只凭自己本身来保证达到这些目的。例如，编程必须嵌入在一个有序的开发流程，从而分析需求，设计，实现，查验，确认和维护得以井然有序。从获得好评的软件工程规约中总结出的好的编程实践在语言的使用中必须遵守。本书试图在工程规约和 Ada 实际编程中的差异架起桥梁。

本书中的许多准则都为促进源代码的清晰而设。这些准则的目标是提高程序的演变、改写和维护的容易度。可理解的源代码，使其看起来正确和可靠的可能性更高。代码的改写要求对程序有完整的理解，清晰的代码对于理解有相当的帮助。有效的代码改写是代码重用不可缺的条件，而代码的重用有大幅降低系统开发成本的潜力。最后，系统使用期间的维护 (实际上是演变) 是个持续的昂贵过程，清晰的代码在维持较低的维护成本上起了主要作用。在整个系统的生命周期中，代码被阅读和理解的次数大大多于编写，所以投资在代码的可读性和可理解性上是值得的。

在本章剩下的小节中，总结了一下本书的结构和如何把呈现的内容给不同的角色使用，如新的 Ada 程序员，有经验的 Ada 程序员，面向对象的程序员，项目经理，承包商，标准制定组织，从现有的 Ada 83[ARM 1983] 程序过渡到 Ada 95 的计划者。

1.1 本书的结构

本书的格式遵循受到广泛欢迎的 《Ada 品质和风格：专业程序员的准则》版本号 02.01.01 (AQ&S 83) (软件生产力协会 1992) 中所用的格式。本指南根据程序员为了写出高质量、可靠、可重用、可移植的 Ada 程序，所做出的各个重要决定，进行分章描述。章节中某些部分会有所重复，因为不是所有的决定都可以独立做出。

本书对源代码的呈现、可读性、程序结构、并行性、可移植性、可重用性、性能以及面向对象进行了分章阐述。每一章都以一个本章准则的**总结**来结束。最后一章呈现了一个完整的哲学家吃晚餐的实现，这个范例由 Michael B. Feldman 博士和 Bjorn Kallberg 先生提供。这个范例中使用了本书中的许多准则。附录给出了一个 Ada 参考手册 (1995) 和本书中的准则之间的交互参照矩阵。

本书的用词是过去 20 年软件工程中发展出的通用词语。软件工程是个快速发展的领域，有许多相对新的概念和术语。为了有个共通的参考框架，需要的定义会从 Ada 参考手册 (1995) 和原理 (1995) 中取用。

本书多处引用了其他有关 Ada 风格和问题的文献和资料。相关的参考资料列表在书的末尾。书末也提供了参考文献列表。

“Ada” 这个专有名词在本书中是指在 1995 年二月公布的最新 Ada 标准 [1] (有时也用 Ada 95 来表示)。引用早前的 Ada 标准都会清楚的注明 “Ada 83”。

1.1.1 源代码的呈现和可读性

第2章和第3章直接针对创作清晰、可读、可理解源代码中遇到的问题。第2章集中讨论代码的格式，第3章解决运用注释、命名规则和类型的问题。

代码的清晰有两个主要的方面：(1) 第2章涵盖代码在页面和屏幕上谨慎而一致的**版面规划**，可大幅提高代码的可读性；(2) 第3章涵盖谨慎对待代码的**结构**，令代码易于理解。这两方面即适合于小规模的应用 (例如小心挑选变量名或按规约使用循环)，也适合于大规模的应用 (例如封装包的适当使用)。

代码格式和命名规则的偏好是很个人的选择。你必须平衡好自己和项目中其他工程师的喜好和厌恶，以确立一套全项目的人都要一致遵守的规约。自动代码格式器有助于实施这类的规约以保持代码的一致。

1.1.2 程序结构

第**??**章主讲程序的整体结构。适当的结构会提高程序的清晰度。这相当于在底层的可读性上再包含高层结构的内容，特别是封装包 (package)、子封装包 (child package) 的应用，可见度和异常。本章大部分的准则都是有关于软件工程中健全的原则，例如信息的隐藏、抽象、封装和关系的分离。

1.1.3 编程实践

第**??**章的准则定义了语言功能使用上的一致性和逻辑性。这些准则针对语法、类型、数据结构、表达式、语句、可见度、异常和执行错误中的可选部分。

1.1.4 并行性

第6章定义了并行性的正确使用，从而开发出可预料、可靠、可重用、可移植的软件。主题包括任务 (tasking)、被保护单元 (protected unit)、通信以及终止。本版 Ada 语言的主要增强的一个方面就是更好的支持数据共享。以前，唯一保护共享数据的方法是通过任务的机制。本章的准则支持使用保护类型来封装和同步共享数据的访问。

1.1.5 可移植性和可重用性

第**??**章和第**??**章讨论因着眼点的稍微不同而引起的设计改变的问题。第**??**章探讨可移植性的基础，即软件很容易从一个电脑系统或环境改到另一个系统或环境，以及某些特定功能的使用对可移植性的影响。第**??**章讨论代码的重用，即代码以最少改动可以使用的范围。

要特别注意第**??**章中讨论的准则。即使现在看不到软件产品的移植需要，遵守这些准则还是很有必要的，这提高了软件在其它使用不同 Ada 实现的项目中重用的可能。当在某个项目中，某些准则必须被放松，你应该坚持把不可移植部分的功能代码显著的标示出来。

第**??**章中有关重用性的准则，是在封装和为变化设计的原则基础上得出来的。即使并不预期会重用，这些准则强调了理解性和清晰，健壮性，适应性和独立性是有益而最希望的，因为这样的代码更能对应计划和非计划的变化。

[1] 译者注：现在最新的标准是 Ada 2005 了。

1.1.6 面向对象的功能

第**??**章用通用的面向对象的术语定义了一套准则，来开发 Ada 95 中的某些新功能。这些准则讨论了 Ada 新功能的使用，包括类型扩展 (标签类型，tagged)、抽象标签 (abstract tagged) 类型、实现单一继承、多重继承和多形的抽象子程序。

1.1.7 性能

第**??**章定义了一套旨在提高性能的准则。大家都公认某些提高性能的方法是和可维护性、可移植性相冲的。本章大多数的准则都含有这样的句子“当性能的测量指出。”“指出”意味着，在你的系统中，应用程序性能的提高带来的好处超出其负面影响，即降低了源码的可理解性、可以维护性和可移植性。

1.2 怎样使用本书

本书适用于实际使用 Ada 进行软件系统开发的相关人员。下面的段落中讨论如合最有效的利用本书中的内容。有不同程度的 Ada 经验的读者或软件项目中的不同角色要以不同的方式使用本书。

本书有好几种使用的方法：作为好的 Ada 风格的指南；为更好的 Ada 程序作出贡献的全面的准则列表；或者作为探讨语言中某些特定功能的使用范例或着设计折衷的参考资料。本书包含了许多准则，某些还十分复杂。一般不大可能同时使用语言中的所有功能，所以同时学习所有的准则并不是必须的。但是，推荐所有的程序员 (如果可能，所有 Ada 项目的员工) 都应该尽力去读懂第2、**??**、**??**章，以及直到第**??**章的第**??**节。某些内容比较困难 (例如第 **??**节，讨论可见度)，但是它涵盖了有效使用 Ada 的基础，这对任何构建 Ada 系统的相关专业人员很重要。

本书不是 Ada 语言的入门介绍，也不是全面的手册。本书假定您已经知道 Ada 的语法以及对语义的基本理解。在这样的前提下，您将发现这些准则是有益，有教育性，通常还具启发性的。

如果您在学习 Ada，您最好让自己对语言的有个全面的入门了解。有两本不错的 Ada 83 入门书 [Barnes 1989] 和 [Cohen 1986]。两位作者都出版了涵盖 Ada 95 的新书 ([Barnes 1996]、[Cohen 1996])。一旦你熟悉了这些内容，推荐您和 [原理 1995] 一起使用。《Ada 95 参考手册》[ARM 1995] 应当被视为这些书的姊妹篇。大多数准则都会把《Ada 95 参考手册》中相关语言功能定义的段落和讨论关联起来。附录 [**?**] 列出了《Ada 95 参考手册》和本书中的准则之间的交互参照。

1.3 给新 Ada 程序员

第一眼，Ada 提供的各类功能让人迷惑。这个强大的工具是用来解决困难问题的，几乎每个功能都有在某些场合下使用的合理性。这使得在一个规约而有组织的情况下使用 Ada 的这些功能，变得特别重要。遵循这些准则让学习 Ada 变得简单些，还能帮助你掌握她的复杂的功能。从一开始，你就可以用写程序来熟悉语言中的最好功能，就像语言设计者预期的那样。

有使用其它语言经验的程序员经常陷入把 Ada 当成以前熟悉的语言来使用的情况，只是用不同的恼人的语法。应该不惜代价来避免这个陷阱；它可能导致错综的代码而破坏了让 Ada 恰恰成为适合建造高质量系统的方面。你必须学会用“Ada 思考”。遵循本书的准则，参看范例的使用，会帮助你尽可能快而不痛苦的学会用 Ada 思考。

从某些角度来说，无经验的程序员学习 Ada 有优势。从一开始就遵循这些准则，有助于养成清晰的编程风格和有效的活用语言。如果你属于这个类别，推荐你在学习 Ada 做练习时采用这些准则。开始的时候，集中精力在准则本身和它的范例，以养成好的编程习惯比理解每个准则的原理更重要。

准则的原理帮助有经验的程序员理解和接受准则中的建议。某些准则是为有经验的程序员而写的，他们必须作出某些工程折衷。这在可移植性、可重用性和性能方面特别突出。这些比较困难的准则和原理让你注意到影响编程决定的问题。当你成为经验的 Ada 程序员的时候，你会注意到并作出工程折衷。

1.4 给有经验的 Ada 程序员

作为一个有经验的 Ada 程序员，你写的代码已经符合了许多本书的准则。但是，在某些方面，你可能已经形成了自己的编程风格，与本书提到的准则不符，而你可能不愿去改。仔细的回顾那些和你的风格不一致的准则，确认你了解它的原理，并考虑采用。本书的准则汇集了一套可靠的方法以完成高质量的程序，太多的异例会削弱它。

一致性是另一个全面采用通用准则的重要原因。如果项目中所有的人都用同一风格来写代码，项目中许多重要的活动会变得容易。一致的代码，令正式、非正式的代码复核、系统整合、项目中代码的重用、提供和使用支持工具变得简化。实践中，公司或项目的标准也许有和准则不符的地方，这需要特别列出，所以采用非标准的方法需要更多的工作。

本书中的某些准则集中在设计中的折衷上，特别是有关并行性、可移植性、可重用性、面向对象的功能和性能的章节。这些准则要求你考虑，在应用中使用某个 Ada 功能是否是合适的设计决定。通常会有几种方法来实现某个特定的设计决定，你在做决定的时候要考虑这些准则讨论的各种折衷方案。

1.5 给有经验的面向对象的程序员

作为一个有经验的向对象的程序员，你将会欣赏为简洁的扩展 Ada 语言，使其包含强大的面向对象的功能，所做的努力。这些新功能和现有的语言功能和语汇紧密的整合在一起。本书特意从风格的角度来写，所以 Ada 面向对象的功能在全书中都有使用。规约的使用这些功能有助于程序更容易的阅读和修改。这些功能让你可以灵活的构造可重用构件。第**??**章针对面向对象的编程，以及继承和多形中的问题。其他章节会互相对照第**??**章的准则。

如果你做过面向对象的设计，你将更容易的利用第**??**章中的许多概念。一个面向对象的设计包含了一套有意义的抽象和分级结构的类。抽象应该包括设计对象的定义，即结构和状态、对象的操作以及每个对象的封装。如何设计这些抽象和分级结构的类超出了本书的范围。有许多很好的文献讨论这个问题，例如：[Rumbaugh, 等人 1991]、[Jacobson, I. 等人 1992]、《ADARTS 指南》[软件生产力协会 1993] 和 [Booch 1994]。

1.6 给软件项目经理

技术的管理在保证生产的软件的正确性、可靠性、可维护性和可移植性中具有重要的地位。管理层必须为生产高质量的产品建立一套项目范围的约定：定义针对项目的代码标准和准则；让所有人理解为什么保持选定标准对产品品质的重要性；同时建立政策和程序来检查和实施这些标准。本书中的准则有助于这方面的努力。

经理的一个重要工作就是定义一个公司或项目的代码标准。这些准则本身并不构成一套完整的标准，但是它们可以作为标准的基础。有几个准则指出了好几个决定，但没有指定其中的某个。例如，本书中的**准则 [?]** 提倡使用一致的定数空格来缩进代码，在原理中指出两个或四个空格都可以。和你的资深技术员工一起回顾这些准则，以确定公司或项目标准中所用的例示。

指定标准中，还有两个方面需要管理层的决定。**准则 [?]** 建议你为对象或单元命名时，避免使用任意的缩略语。你应当为此准备一个词汇表，列出项目中允许使用的专有缩略语，例如 FFT 是 Fast Fourier Transform (快速傅利叶变换)，SPN 是 Stochastic Peri Net (随机佩特里网)。这个词汇表应当尽可能的短小，只包含要经常在名字中出现的术语。如果需要经常的使用大规模的词汇表，那令代码难以阅读。

第 [?] 章有关可移植性的准则需要格外的注意。即使现在看不到软件产品的移植需要，遵守这些准则还是很重要的。这提高了软件在其它使用不同 Ada 实现的项目中重用的可能。当在某个项目中，某些准则必须被放松，你应该坚持把不可移植部分的功能代码显著的标示出来。第 [?] 章中的准则要求在项目或公司中定义和标准化项目或公司特有的数字类型，来代替语言中预定义的数字类型 (有不可移植的可能)。

你在标准化中遇到的问题和决定，应该记录在项目或公司的代码标准文档中。有了恰当的标准，你要保证大家都遵守。取得编程人员诚心的承诺去使用标准非常的关键。有了这个承诺和程序员写的高质量的 Ada 程序范例，进行有效的针对项目标准符合性的正式代码审核会容易很多。

一些有关 Ada 项目管理的普遍问题在 [Hefley, 等人, 1992] 中有讨论。

1.7 给承包商和标准制定组织

本书中的许多准则都很有针对性，可以采用作为公司或项目的编程标准。其它的需要管理上决定应用某个实例才能采用，这种情况下，会有一个样本实例，并在范例中使用。这样的实例应当看作比准则要弱。某些情况下，例子是从公开的资料中取得的，作者的风格被保留而没有修改。

书中的某些准则特意从设计选择的角度来写。这些准则不能被直接变成项目的规约来实施。例如**准则 [?]** 和 **[?]** 不能被认为项目中不能用任务。反而，这些准则有助于设计者在使用被保护对象还是任务中作出折衷，从而作出更有依据的选择。

本书中的准则并不能只凭自己而成为标准。某些情况下，某个准则能不能实施还是个问题，因为它只是要引起工程师注意其中的折衷。其它情况，准则中还有需要选择的地方，如每一级的缩进需要多少空格。

当一个准则太概略而不能给出范例，它的**实例**部分含有更有针对性的准则。标准中可以采用这些实例，因为它们更容易被实施。任何试图从本书中摘录出内容作为标准的组织应该衡量所有的上下文。只有和其相关的准则一起使用，才能发挥每条准则的最大功效。孤立出来，一条准则也许只有很少甚至没有效用。

1.8 给从 Ada 83 过渡到 Ada 95 的计划者

过渡的问题主要有两类：语言的不兼容，特别是向上兼容，以及语言新功能的利用。

向上兼容是 Ada 95 设计时的主要目标。实践中很可能遇到一些 Ada 83 和 Ada 95 不兼容的情况，这些情况很容易就可以克服 (参见 [原理 1995] 的 附录 X：向上兼容[2])。兼容问题的进一步信息还可参考 [Taylor 1995] 和 [Intermetrics 1995]。

过渡计划者可以从两方面深入了解如何利用语言功能。首先列表1.1 列出 Ada 95 的新功能对风格指南章节的影响。另外附录**??** 作了 [ARM 1995] 中的章节到本书中准则的映射。

1.9 排版规约

有待翻译结束再更新。

[2] http://http://www.adaic.com/standards/95rat/RAThtml/rat95-p4-x.html

列表 1.1: Ada 95 的新功能对风格指南章节的影响

Ada 95 功能和增强	代码	可读	结构	实践	并行	移植	重用	面向对象	性能
面向对象功能									
类型扩展 (标签类型)	✓	✓		✓			✓	✓	
管制类型							✓	✓	
多形				✓			✓	✓	✓
多重继承								✓	
抽象类型和子程序								✓	
程序结构和编译									
子单元库		✓	✓						
通用模板			✓				✓		
任务模型修订									
被保护类型		✓	✓		✓	✓			
同步机制	✓		✓		✓	✓			
说明和类型									
子程序访问类型						✓	✓		
访问类型				✓					
其它变动									
异例		✓	✓	✓		✓			
类型使用和重命名			✓	✓					
与其它语言的接口						✓			
专门的附录									
系统编程					✓	✓			
实时系统					✓	✓			✓
分布式系统					✓	✓			✓
信息系统		✓		✓		✓			
数值		✓		✓		✓			
安全和防备		✓		✓	✓	✓			✓

第 二 章

源代码的呈现

代码在纸张和屏幕上的版面规划对于代码的可读性起了很大的作用。本章包含了令代码更可读的呈现准则。

在通用的准则之外，在**具体事例**部分，还有特别的推荐。如果你不同意这些特别的推荐，你可以采用自己的规约，同时也遵守了准则。总之，保持全项目的一致。

完全一致的版面规约很难靠人工去完成和检查，所以，对于版面规范的自动化，你可能比较细欢使用工具根据不同参数来格式化代码，或者把准则整合到自动代码模块。某些准则和特别的推荐不能用自动格式化工具完成，因为它们是基于 Ada 语句的语义而非语法。**自动化注解**会给出更多的详细内容。

2.1 代码格式化

代码的格式化影响代码的观感，而不是代码的功用。这里讨论的问题包括：水平间隔、缩进、对齐、分页和行长度。最重要的准则是在编译单元和整个项目中保持一致。

2.1.1 水平间隔

准则

- 分割符之间使用固定的间隔。
- 使用和写一般文章一样的空格。

具体事例

特别指出，在下列地方，保留至少一个空格，本书中的范例也使用一样的准则。紧跟的准则会为了对齐的原因，可能需要更多的纵向空间。

- 在下列分隔符和二进制操作符的前后：

```
+       -       *       /       &
<       =       >       /=      <=   >=
:=      =>      |       ..
:
<>
```

- 除了禁止的地方，在字符串和字符引号 (") 和 (') 之外的文字。
- 在括号的外面，不是里面。
- 在逗号 (,) 和分号 (;) 之后。

在下列的地方不要有任何空格，即使和上面的建议相背。

- 当加号 (+) 和减号 (–) 作为一元操作符时，它们的后面。

- 函数调用的后面。
- 在标签分隔符内 (<< >>)。
- 在幂操作符 (**)、撇号 (') 和句号 (.) 的前后。
- 在多个连续的开启或终止括号中间。
- 在逗号 (,) 和分号 (;) 之前。

当多余的括号因为操作优先的规则被忽略，表达式中围绕最高优先级操作符的空格可选择性的去除。

范例

```
Default_String : constant String :=
      "This is the long string returned by" &
      " default. It is broken into multiple" &
      " Ada source lines for convenience.";

type Signed_Whole_16 is range -2**15 .. 2**15 - 1;
type Address_Area  is array (Natural range <>) of Signed_Whole_16;

Register : Address_Area (16#7FF0# .. 16#7FFF#);
Memory   : Address_Area (       0 .. 16#7FEC#);

Register(Pc) := Register(A);

X := Signed_Whole_16(Radius * Sin(Angle));

Register(Index) := Memory(Base_Address + Index * Element_Length);

Get(Value => Sensor);

Error_Term := 1.0 - (Cos(Theta)**2 + Sin(Theta)**2);

Z      := X**3;
Y      := C * X + B;
Volume := Length * Width * Height;
```

原理

分隔符和操作符通常只有一到两个字符的宽度，很容易在比较长的关键字和变量名中迷失，所以在它们周围加上空格是很好的注意。这让它们突出。代码中一致的空格也有助于视觉上的审视。

但是，许多分隔符 (逗号、分号、括号，等) 是我们熟悉的普通标点符号。如果电脑程序代码中这些符号和一般的文本不一样，这很容易分散注意力。因此，使用和普通文本一样的空格规则 (在逗号、分号前没空格，括号内内空格，等等)。

异例

唯一引人注目的异例是冒号 (:)。在 Ada 中，有时用它来做定位符或纵列分隔符 (参见准则2.1.4)，在这种情形下，在它的前面和后面加入空格而不只是象普通文本那样加在后面，就显得有意义。

自动化注解

本节里的准则很容易通过一个自动的代码格式器来实施。

2.1.2 缩进

准则

- 缩进和对齐嵌套控制结构、连续行和嵌入单元，并保持一致。
- 区分连续行和嵌套控制结构的缩进。
- 使用空格作为缩进符，不要使用标记符 (tab 字符 [Nissen 和 Wallis 1984] §2.2)。

具体事例

特别说明，推荐使用以下的缩进约定，本书中的范例都会使用。注意说明的最小缩进空间。紧跟的准则可能需要更多的纵向空间。

- 使用 [ARM 1995] §1.1.4 中推荐的分段法 [1]。
- 用 3 个空格作为嵌套的缩进的单位。
- 用 2 个空格作为连续行的缩进单位。

标签用 3 个空格来缩进：

```
begin
<<lable>>                       | <long statement with line break>
   <statement>                  |   <trailing part of same statement>
end;
```

if 和简单循环语句：

```
if <condition> then             | <name>:
   <statements>                 |    loop
elsif <condition> then          |       <statements>
   <statements>                 |       exit when <condition>;
else                            |       <statements>
   <statements>                 |    end loop <name>;
end if;
```

for 和 while 循环：

```
<name>:                         | <name>:
   for <scheme> loop            |    while <condition> loop
      <statemetns>              |       <statements>
   end loop <name>;             |    end loop <name>;
```

语句段落块和 case 语句，正如 [ARM 1995] 中所推荐的：

```
<name>:                         | case <expression> is
   declare                      |    when <choice> =>
      <declarations>            |       <statements>
   begin                        |    when <choice> =>
      <statements>              |       <statements>
   exception                    |    when others =>
      when <choice> =>          |       <statements>
         <statements>           | end case; -- <comment>
      when others =>            |
         <statements>           |
   end <name>;                  |
```

下面的 case 语句比 [ARM 1995] 的推荐节省一些空间，它依赖于较短的语句。无论你选择那种方式，要保持一致：

[1] http://www.adaic.com/standards/95lrm/html/RM-1-1-4.html

```
case <expression> is
when <choice> =>
     <statements>
when <choice> =>
     <statements>
when others =>
     <statements>
end case;
```

```
case <expression> is
   when <choice> => <statements>
                    <statements>
   when <choice> => <statements>
   when others   => <statements>
end case;
```

各种形式的可选 accept、计时和有条件导入 (entry) 调用:

```
select
   when <guard> =>
      <accept statement>
      <statements>
or
   <accept statement>
   <statements>
or
   when <guard> =>
      delay <interval>;
      <statements>
or
   when <guard> =>
      terminate;
else
   <statements>
end select;
```

```
select
   <entry call>;
   <statements>
or
   delay <interval>;
   <statements>
end select;

select
   <entry call>;
   <statements>
else
   <statements>
end select;

select
   <triggering alternative>
then abort
   <abortable part>
end select;
```

accept 语句:

```
accept <specification> do
   <statements>
end <name>;
```

```
separate (<parent  unit>)
<proper body>
```

一个子单元:

```
separate (<parent unit>)
<proper body>
end <name>;
```

程序单元的适当形式:

```
procedure <specification> is
   <declarations>
begin
   <statements>
exception
   when <choice> =>
      <statements>
end <name>;

function <specification>
  return <type name> is
   <declarations>
begin
   <statements>
exception
   when <choice> =>
      <statements>
end <name>;
```

```
package body <name> is
   <declarations>
begin
   <statements>
exception
   when <choice>=>
      <statements>
end <name>;

task body <name> is
   <declarations>
begin
   <statements>
exception
   when <choice>=>
      <statements>
end <name>;
```

编译单元的语境语句使用列表形式。通用 (generic) 行参不可以模糊单元本身。对函数、封包 (pacakage) 和任务 (task) 的规约使用标准缩进:

```
with <name>; use <name>;        | function <specification>
with <name>;                    |   return <type>;
with <name>;                    |
                                | package <name> is
                                |    <declarations>
                                | private
<compilation unit>              |    <declarations>
                                | end <name>;
                                |
generic                         | task type <name> is
   <formal parameters>          |    <entry declarations>
<compilation unit>              | end <name>;
```

通用单元的实现和结构的缩进:

```
procedure <name> is                |type ... is
   new <generic name> <actuals>    |   record
                                   |      <component list>
function <name> is                 |      case <discriminant name> is
  new <generic name> <actuals>     |          when <choice> =>
                                   |             <component list>
package <name> is                  |          when <choice> =>
   new <generic name> <actuals>    |             <component list>
                                   |      end case;
                                   |   end record;
```

for 结构的对齐缩进:

```
for <name> use
   record <mod clause>
      <component clause>
   end record;
```

标签类型和类型的扩展:

```
type ... is tagged
   record
      <component list>
   end record;

type ... is new ... with
   record
      <component list>
   end record;
```

范例

```
Default_String : constant String :=
     "This is the long string returned by" &
     " default. It is broken into multiple" &
     " Ada source lines for convenience.";
  if Input_Found then
     Count_Characters;
  else -- not Input_Found
     Reset_State;
     Character_Total :=
       First_Part_Total * First_Part_Scale_Factor +
       Second_Part_Total * Second_Part_Scale_Factor +
       Default_String'Length + Delimiter_Size;
```

```
   end if;
end loop;
```

原理

缩进是程序结构的可见标示，从而提高了代码的可读性。嵌套的层数由缩进可清晰的分辨出来，结构中开始和结束的关键字可视觉的匹配。

尽管有许多关于用多少空格来缩进的讨论，缩进的目的是令代码清晰。代码缩进的一致比使用多少空格来缩进更重要。

另外，[ARM 1995]§1.1.4 陈述推荐使用手册的例子和语法规则中用的版面规划作为 Ada 程序的代码版面规划:“语法规则中描述的结构的构建，使用的是推荐的分段法……如果语法规则中的构建需要在不同的行上，构建会分行处理……推荐所有的缩进使用基本缩进单元的倍数(基本单元中空格的数目没有定义)。”

特别注意对连续行使用和嵌套控制结构不一样的缩进，这样它们会凸显出来，从而在审视代码时，不会朦胧了代码结构。

语境语句列在不同的行使维护更加容易; 改变语境语句也不容易出错。

使用空格来缩进比用标记符来得易移植，因为标记符在不同的终端和打印机可能显示不同。

异例

如果你用变宽的字体，标记符比空格对齐的好。但是，依赖于你的标记符设置，相继的缩进行可能只留给你很短的长度。

自动化注解

本节里的准则很容易通过一个自动代码格式器来实施。

2.1.3 操作符的对齐

准则

- 竖向对齐操作符，以突出当前程序结构和语意。

范例

```
if Slot_A >= Slot_B then
   Temporary := Slot_A;
   Slot_A    := Slot_B;
   Slot_B    := Temporary;
end if;
```

```
Numerator   := B**2 - 4.0 * A * C;
Denominator := 2.0 * A;
Solution_1 := (B + Square_Root(Numerator)) / Denominator;
Solution_2 := (B - Square_Root(Numerator)) / Denominator;
```

```
X := A * B +
     C * D +
     E * F;
Y := (A * B + C) +  (2.0 * D - E) -  -- basic equation
     3.5;                            -- account for error factor
```

原理

对齐使得操作符的位置更容易被看到，从视觉上强调了代码在干什么。

在长表达式中换行和间隔，突出了条件、操作符的优先级和其它语意。它也为在表达式中标示注释留出了空间。

异例

如果竖向对齐使得一个语句被迫变成两行，特别是如果换行的地方并不合适，可以适当的放宽这条对齐的准则。

自动化注释

前面的范例中，呈现一种“语意对齐”，通常自动代码格式器并不能做到或者保留。如果你根据语意把一个表达式分拆成多个部分，每个部分各占一行，以后使用自动代码格式器是要注意。整个表达式有可能变成一行，而注释都集中在后面。但是，如果那一行含有注释，某些足够聪明的格式器会保留换行。一个好的格式器可以识别出前面的范例没有违反准则，从而保留原样。

2.1.4 声明的对齐

准则

- 用竖向对齐来使声明更可读。
- 每行最多只提供一个声明。
- 缩进在同一层声明部分的声明。

具体事例

对于没有被空行分隔的声明，请遵守下面的规则：

- 对齐冒号分隔符。
- 对齐初始化分隔符 :=。
- 有句末注释时，对齐注释符。
- 当声明超出了一行，注意换行以及缩进被换行。换行的地方依次为：(1) 注释符; (2) 初始化分隔符; (3) 冒号分隔符。
- 对于无法在一行内完成的枚举类型，让每一个字面值占一行并缩进一级。当合适的时候，语意相关的字面值可以按行和列进行布置成为表格。

范例

变量和常量的声明可以按:、:= 和 --进行分隔列表

```
Prompt_Column : constant         := 40;
Question_Mark : constant String := " ? "; -- prompt on error input
Prompt_String : constant String := " ==> ";
```

如果这样令行太长，可以让每以部分分占一行，同时有自己唯一的缩进级别：

```
subtype User_Response_Text_Frame is String (1 .. 72);
-- If the declaration needed a comment, it would fit here.
Input_Line_Buffer : User_Response_Text_Frame
       := Prompt_String &
          String'(1 .. User_Response_Text_Frame'Length -
                      Prompt_String'Length => ' ');
```

枚举类型的声明，字面值可以用单列或多列成表：

```
type Op_Codes_In_Column is
      (Push,
       Pop,
       Add,
       Subtract,
       Multiply,
       Divide,
       Subroutine_Call,
       Subroutine_Return,
       Branch,
       Branch_On_Zero,
       Branch_On_Negative);
```

或节省空间:

```
type Op_Codes_Multiple_Columns is
      (Push,            Pop,               Add,
       Subtract,        Multiply,          Divide,
       Subroutine_Call, Subroutine_Return, Branch,
       Branch_On_Zero,  Branch_On_Negative);
```

或突出相互关系的组别:

```
type Op_Codes_In_Table is
      (Push,            Pop,
       Add,             Subtract,          Multiply,          Divide,
       Subroutine_Call, Subroutine_Return,
       Branch,          Branch_On_Zero,    Branch_On_Negative);
```

原理

许多编程标准的文档，要求在单元前的注释中用表格的形式重复名字、类型、初始值和意义。这些注释是多余的，而且会变得和代码不符。将声明本身对齐为表格形式 (参见之前的范例)，为编译器和读者提供了同样的信息；执行每行最多只有一个声明；为初始化加空间和必要的注释以方便维护。表格化的版面增强可读性，避免名字在大量的声明中“隐藏”起来。这些对所有的声明都有效: 类型、子类型、对象、异常、数字命名等等。

自动化注解

本小节中的大部分准则都可以用自动代码格式器来实施。唯一的异例是范例中的枚举类型，它是根据字面值的语意来换行排版的。一个自动代码格式器不能做到，很可能会把枚举类型的字面值换到不同的行。但是，只是检查是否有违规的工具应该接受这种表格形式的枚举类型声明。

2.1.5 更多对齐

准则

- 竖向对齐形参模式和括号。

具体事例

特别推荐:

- 每行只放一个形式参数的规约。
- 竖向对齐形参的名字、冒号、保留字 in、保留字 out 和形参的类型。
- 把第一个形参放在子程序或入口名字的同一行。如果任何一个形参的类型超过了行长度限制，把第一个形参放在新的一行，并使用和连续行一样的缩进。

范例

```
procedure Display_Menu (Title   : in      String;
                        Options : in      Menus;
                        Choice  :    out  Alpha_Numerics);
```

下面的两个例子用本准则的另外的具体事例:

```
procedure Display_Menu_On_Primary_Window
      (Title   : in      String;
       Options : in      Menus;
       Choice  :    out  Alpha_Numerics);
```

或:

```
procedure Display_Menu_On_Screen (
       Title   : in      String;
       Options : in      Menus;
       Choice  :     out Alpha_Numerics
    );
```

对齐括号令复杂的关系表达式更加清晰:

```
if not (First_Character in Alpha_Numerics and then
        Valid_Option(First_Character))        then
```

原理

这个准则促进可读性和理解性，也很容易由自动化工具达到。形参模式的对齐，产生了一个列表的效果，形参的名字、模式、类型、如果由必要，针对形参的注释。跨越同一编译单元内子程序的竖向对齐进一步增进了可读性。

注解

字程序的版面布局有许多可用的选择。上面的第二个例子对齐了所有子程序的形参名字。这样有个占用不需要行的缺点，如果子程序的名字比较短，只有一个形参，那看齐来难看，

第三个例子的格式，通常用来减少形参被增加、删除或重组时需要的编辑。国号不需要随着行来变化。但是，最后一个形参行是唯一没有冒号的。

异例

当一个操作符函数由两个以上同类型形参，在一行里列表声明形参比把形参分到不同行里更易读。

```
type Color_Scheme is (Red, Purple, Blue, Green, Yellow, White, Black, Brown,
                      Gray, Pink);
function "&" (Left, Right : Color_Scheme) return Color_Scheme;
```

自动化注解

本小节中的大部分准则都可以用自动代码格式器来实施。唯一的异例是最后一个范例中，用竖向对齐括号来突出表达式的条件。用自动代码格式器很难做到，除非相关的条件可以严格按照操作符的优先级来决定。

2.1.6 空行

准则

- 用空行来办逻辑上相关的文字组在一起 ([NASA 1987])。

范例

```
if ... then

   for ... loop
      ...
   end loop;

end if;
```

这个例子把不同的类型声明用空行分割开来:

```
type Employee_Record is
   record
      Legal_Name     : Name;
      Date_Of_Birth  : Date;
      Date_Of_Hire   : Date;
      Salary         : Money;
   end record;

type Day is
  (Monday,    Tuesday,    Wednesday, Thursday, Friday,
   Saturday,  Sunday);

subtype Weekday is Day range Monday    .. Friday;
subtype Weekend is Day range Saturday  .. Sunday;
```

原理

当空行在深思熟虑以及一致的情况下用时，相关的代码部分对于读者更明显。

自动化注解

自动格式器并不能很好的实施这个准则，因为在决定在哪里插入空行和语意相关。怠驶，许多格式器能够保留以存在的空行，所以你可以手动加入空行，当你运行那样的格式器的时候，不会丢失你的改动。

2.1.7 分页

准则

- 标示每一个封包或任务规约的顶端，每个程序单元主体的顶端，和每个程序单元的末段。

具体事例

特别推荐:

- 用文件序言，规约标题，和体标题来标示那些**??**中推荐的结构。
- 用当前缩进同一列开始的划行，来标示在声明部分嵌套的单元定义。在定义之前和之后立即插入划行。
- 如果两个换行相邻，忽略较长的那个。

范例

```
with Basic_Types;

package body SPC_Numeric_Types is

...

   ------------------------------------------------------------------------
   function Max
         (Left  : in    Basic_Types.Tiny_Integer;
          Right : in    Basic_Types.Tiny_Integer)
         return Basic_Types.Tiny_Integer is
   begin
      if Right < Left then
         return Left;
      else
         return Right;
      end if;
   end Max;

   ------------------------------------------------------------------------
   function Min
         (Left  : in    Basic_Types.Tiny_Integer;
          Right : in    Basic_Types.Tiny_Integer)
         return Basic_Types.Tiny_Integer is
   begin
      if Left < Right then
         return Left;
      else
         return Right;
      end if;
   end Min;

   ------------------------------------------------------------------------

   use Basic_Types;

begin -- SPC_Numeric_Types
   Max_Tiny_Integer := Min(System_Max, Local_Max);
   Min_Tiny_Integer := Max(System_Min, Local_Min);
   -- ...
end SPC_Numeric_Types;
```

原理

不在当前页或屏幕的程序单元很容易被忽略。呈现用的硬件和软件之间，页长度变化很大。清楚的标示出程序逻辑页的边界 (例如用划行)，读者可以快速的检查是否整个程序单元都可见。这样的分页，也更容易在大文件中，快速扫描到某个程序单元。

注解

本准则不是针对物理的“页”, 因为那样的页的尺寸变化很多，没有一个单一准则可以适用。

自动化注解

本节里的准则很容易通过一个自动代码格式器来实施。

2.1.8 每一行里的语句个数

准则

- 用新的一行开始每个语句。
- 每行不要些超过一个简单语句。
- 断开复合语句到多行。

范例

用：

```
if End_Of_File then
   Close_File;
else
   Get_Next_Record;
end if;
```

不要用：

```
if End_Of_File then Close_File; else Get_Next_Record; end if;
```

异例情况：

```
Put("A="); Natural_IO.Put(A); New_Line;
Put("B="); Natural_IO.Put(B); New_Line;
Put("C="); Natural_IO.Put(C); New_Line;
```

原理

每行一个语句增强了读者找到某些语句的能力，有助于避免漏了某些语句。同样的，如果复合语句的构件在不同的行上，它的结构更清晰。

注解

如果一个语句比一行里剩下的空间要长，在下一行继续。本准则适用于声明、语境子句和子程序形参。

据 [ARM 1995]§1.1.4 描述“最好在分号之后换行。”

自动化注解

本节里的准则很容易通过一个自动代码格式器来实施。唯一的异例是最后那个例子，呈现了多个语句在语意组合成一行。

异例

Put 和 New_Line 语句的例子，呈现了一个合理的异例。把这些有紧密关系的语句组合在一行，使组之间的结构关系更清晰。

2.1.9 源代码行长度

准则

- 坚持一个源代码的最大行长度 ([Nissen 和 Wallis 1984] §2.3)。

具体事例

特别推荐：

- 用 72 个字符作为行长度的最大值。

原理

当把 Ada 程序从一个系统移植到另一个系统，对于源代码的语句行可能有记录的限制，可能的原因有：某些操作系统的磁带输入输出可能不支持变长的记录，或者某些 80 列的打印机或终端不支持换行。更多的原理参见准则 **??**。

由于各种原因，有可能需要发行源代码，纸张对于可用的列没有电脑那样宽容。

另外，阅读代码时，人本身对理解视野内的代码也有宽度的局限，这个局限大概在 70 到 80 列之间。

异例

也可以用 79 个字符作为代码的行长度限制。79 个字符宽度使得代码和 FORTAN 的 72 个字符限制区分开来。它也避免了某些 80 列的终端对最后一列的显示问题。

自动化注解

本节里的准则很容易通过一个自动代码格式器来实施。

2.2 总结

代码格式化

- 分割符之间使用固定的间隔。
- 使用和写一般文章一样的空格。
- 缩进和对齐嵌套控制结构、连续行和嵌入单元，并保持一致。
- 区分连续行和嵌套控制结构的缩进。
- 使用空格作为缩进符，不要使用标记符 (tab 字符 [Nissen 和 Wallis 1984]
- 竖向对齐操作符，以突出当前程序结构和语意。
- 用竖向对齐来使声明更可读。
- 每行最多只提供一个声明。
- 缩进在同一层声明部分的声明。
- 竖向对齐形参模式和括号。
- 用空行来办逻辑上相关的文字组在一起 ([NASA 1987])。
- 标示每一个封包或任务规约的顶端，每个程序单元主体的顶端，和每个程序单元的末段。
- 用新的一行开始每个语句。
- 每行不要些超过一个简单语句。
- 断开复合语句到多行。
- 坚持一个源代码的最大行长度 ([Nissen 和 Wallis 1984] §2.3)。

第 三 章

可读性

本章推荐如何使用 Ada 的功能，使得代码更易阅读和理解。有许多有关注释和可读性的迷思。命名和代码结构比注释对真正的可读性承担更多的责任。拥有和代码一样多的注释并不代表可读性；它更可能是作者不理解什么是需要沟通的重要部分。

3.1 拼写

源代码中的拼写规则包括大写、下划线的应用、数字和缩写。如果你一致遵循这些规则，那写出来的代码就会更清晰和易读。

3.1.1 下划线的应用

准则

- 用下划线分隔复合名字中的词。

范例

```
Miles_Per_Hour
Entry_Value
```

原理

当一个标识符包括多个词，如果用下划线分隔词，那会更容易阅读。事实上，英语中也有这样的惯例，复合词用练字符或空格分隔。另外，为了促进代码的可读性，如果在名字中用下划线，代码格式器对于大小写有更多的控制。参见准则3.1.3。

3.1.2 数字

准则

- 使用一致的风格来呈现数字。
- 使用对问题适当的根植来呈现字面值。
- 使用下划线来分隔数字，就像在普通文字中用逗号或句号 (或对于非十进制用空格)。
- 当使用科学标示法，用一致的大写或小写的 E。
- 在非十进制数字中，要么使用全大写或小写来标示其中的字母。

具体事例

- 十进制或八进制数字，在小数点的左边，每三个数字一组，在右边，每五个数字一组。
- 科学标示中，总用大写字母 E。
- 用大写字母表示在十以上进制的数字。
- 十六进制数字，小数点两边都是每四个数字一组。

范例

```
type Maximum_Samples      is range            1 .. 1_000_000;
type Legal_Hex_Address    is range    16#0000# ..   16#FFFF#;
type Legal_Octal_Address  is range 8#000_000# .. 8#777_777#;

Avogadro_Number : constant := 6.02216_9E+23;
```

用一下方式表示常量 $1/3$:

```
One_Third : constant := 1.0 / 3.0;
```

或

```
One_Third_Base_3 : constant := 3#0.1#;
```

不要用:

```
One_Third_As_Decimal_Approximation : constant := 0.33333_33333_3333;
```

原理

大写或小写字母的一致使用，有助于搜索数字。用下划线来把数字组成熟悉的模式。与平时使用模式的一致是可读性的一大要素。

注解

如果一个分数用一个它可以终止的基数进制来表示，而不用一个重复的表示法，就象上面例子中的 `3#0.1#`，在转换到机器的基数时，有可能更精确。

3.1.3 大写

准则

- 让保留字和程序的其他组件从视觉上分开。

具体事例

- 所有保留字用小写 (当用做保留字时)。
- 所有其他标识符混用大小写，每个词用一个大写字母开始，并用下划线分开。
- 用大写字母表示缩略语或首写字母缩写 (见自动化注释)。

范例

```
...

type Second_Of_Day       is range 0 .. 86_400;
type Noon_Relative_Time  is (Before_Noon, After_Noon, High_Noon);

subtype Morning   is Second_Of_Day range 0 .. 86_400 / 2 - 1;
subtype Afternoon is Second_Of_Day range Morning'Last + 2 .. 86_400;

...

Current_Time := Second_Of_Day(Calendar.Seconds(Calendar.Clock));

if Current_Time in Morning then
    Time_Of_Day := Before_Noon;
elsif Current_Time in Afternoon then
    Time_Of_Day := After_Noon;
else
    Time_Of_Day := High_Noon;
end if;

case Time_Of_Day is
    when Before_Noon =>   Get_Ready_For_Lunch;
    when High_Noon   =>   Eat_Lunch;
    when After_Noon  =>   Get_To_Work;
end case;

...
```

原理

视觉上把保留字突出来，让你集中精力在程序结构上，同时有助于扫描某个标识符。

这里选择的具体事例对于有经验的 Ada 程序员更容易阅读，他不需要保留字在页面上的突出。语言的初学者常常发觉保留字的突出有助于他们更容易的理解控制结构。因此，学习班的教师和介绍 Ada 语言的书籍可以采用另一种实现。[ARM 1995] 选用粗体小写字母来书写所有的保留字。

自动化注解

Ada 的名字是忽略大小写的。所以象 `max_limit`、`MAX_LIMIT`、`Max_Limit` 表示一样的对象或实体。只要词之间用下划线分隔，一个好的代码格式器可以自动从一种风格转换到另一种。

正如**??** 中所推荐，缩写应该是项目层的。自动化的工具应该允许一个项目列出它的缩写并格式他们。

3.1.4 缩写

准则

- 如果一个短小的同义词存在，不要使用长词语的缩写。
- 用一致的缩写策略。
- 不要用有多种解释的缩写。
- 缩写必须比原词节省了许多字符，以认可它的使用。
- 使用应用领域通用的缩写。
- 维护一个缩写列表，只使用列表中的缩写。

范例

用 Time_Of_Recipt 比用 Recd_Time 或 R_Time 要好。但是，对于经常需要处理达到军事标准要求的信息格式，DOD_STD_MSG_FMT 是 Department_Of_Defense_Standard_Message_Format [1] 可接受的缩写。

原理

许多缩写需要结合上下文才不会模糊和不明确。例如，Temp 可以是临时 (temporary) 或温度 (temperature)。因此，当你需要用到缩写时，要小心的选择。准则 **??** 的原理中就上下文如何影响缩写的使用有更加彻底的讨论。

因为非常长的变量名会遮蔽程序结构，特别是在缩进的深层嵌套的控制结构中，所以要尽量让变量名短而有意义。如果可能，尽量的使用短而不用缩写的名字。如果没有合适的短变量名，那么众所周知的缩写是下一个最好的选择，特别是整个项目都在使用的标准缩写列表中的名字。

使用 rename 语句，你可以对一个全名建立缩写。这是个很有用的功能，当在一段本地代码中，一个出现多次的非常长的全名因而可以简化 (参见准则 **[?]**)。

项目中，一个可接受的缩写列表提供了使用这些缩写的标准语境。

3.2 命名约定

选择一个可以有助表达对象或实体运用的名字。Ada 允许任意长度的标志符，只要标志符所有的有效字符 (含下划线) 可以在一行内。标志符就是是变量、常量、程序单元或程序中其他实体的名字。

3.2.1 名字

准则

- 尽量选择可以自我说明的名字。
- 使用短的同义词而不用缩写 (参见准则3.1.4)。
- 使用应用的名词，但是不要用偏僻的术语。
- 避免使用同一个名字来声明不同类型的标识符。

范例

在一个树遍历中，用 Left 在这个语境中已经足够表达它的意思，不用 Left_Branch。然而，要用 Time_Of_Day 而不是 TOD。

数学公式往往用单个字符来命名变量。在程序中，对于数学公式继续这个约定，这样它们可以提示公式，例如：

```
A * (X**2) + B * X + C
```

当使用子包的时候，如果包、子单元或标志符的名字选的不好，有可能导致与子单元的可见性冲突。[原理 1995] (§8.1) 有个因此而变得很晦涩的代码。

原理

遵循这些准则的程序很容易被理解。自我说明的名字需要更少的注释说明。经验表明，如果你的变量名没有过度的长，你可以进一步增加程序的可理解性 ([Schneiderman 1986], 7)。上下文和应用域可以提供很多帮助。数字实体的测量单位是一个子类型名字的来源。

要尽量避免在不同的声明中用同一个名字作为标志符，例如一个对象和一个子包。在表面上看起来不同的名字空间过度使用同一个标识符，如果这些程序单元要在一些工作，实际上会导致可见性冲突。

[1] 国防部信息格式标准

注解

参见准则**??** 有个关于如何用应用领域作为选择缩写的指南。

3.2.2 子类型名字

准则

- 使用单数、泛指的名词作为子类型标志符。
- 选用能说明子类型值的标志符。
- 对定义可见的访问类型、子界、数组，考虑在子类型标志符中用后缀。
- 对私有类型，不要使用针对子类型标志符的构造规则 (例如加后缀)。
- 不要使用预定义包中的子类型名字。

范例

```
type Day is
   (Monday,    Tuesday,   Wednesday, Thursday, Friday,
    Saturday,  Sunday);
type Day_Of_Month    is range      0 ..    31;
type Month_Number    is range      1 ..    12;
type Historical_Year is range −6_000 .. 2_500;

type Date is
   record
      Day   : Day_Of_Month;
      Month : Month_Number;
      Year  : Historical_Year;
   end record;
```

特别指出，Day 应该优先于 Days 或 Day_Type 的使用。

标志符 Historical_Year 也许看起来象特指，但实际是泛指的，形容词 historical 说明了范围的限制。

```
procedure Disk_Driver is
   -- In this procedure, a number of important disk parameters are
   -- linked.
   Number_Of_Sectors  : constant :=      4;
   Number_Of_Tracks   : constant :=    200;
   Number_Of_Surfaces : constant :=     18;
   Sector_Capacity    : constant := 4_096;

   Track_Capacity   : constant := Number_Of_Sectors  * Sector_Capacity;
   Surface_Capacity : constant := Number_Of_Tracks   * Track_Capacity;
   Disk_Capacity    : constant := Number_Of_Surfaces * Surface_Capacity;

   type Sector_Range  is range 1 .. Number_Of_Sectors;
   type Track_Range   is range 1 .. Number_Of_Tracks;
   type Surface_Range is range 1 .. Number_Of_Surfaces;

   type Track_Map   is array (Sector_Range)  of ...;
   type Surface_Map is array (Track_Range)   of Track_Map;
   type Disk_Map    is array (Surface_Range) of Surface_Map;
begin -- Disk_Driver
   ...
end Disk_Driver;
```

后缀 _Capacity、_Range 和 _Map 帮助定义了上述子类型的目的，避免了寻找抽象 sector, track 和 surface 的同义词。没有了这些后缀，对每一个抽象你需要三个不同的名字来说明每个简洁后缀名中所描述的概念。这个建议只对那些可见的子类型。对于私有类型，应该被赋予一个能反应抽象代表的好名字。

原理

当使用这个风格和建议的对象标志符风格时，程序代码更相似于英文 (参见准则**??**)。另外，这个风格和语言预定义的标志符名字保持了一致。它们并没有被命名为 Integers, Booleans, Interger_Type 或 Boolean_Tyoe。

但是，如果使用预定义包中的子类型名，当子类型出现在某处而没有包限定的话，一定会弄糊涂程序员。

注解

这个风格准则试图和 [ARM 1995] 在对术语 “type (类型）” 和 “subtype (子类型) 名字” 的使用上保持一致。通常 “类型” 指一个抽象的概念，如在类型声明中，而 “子类型” 指在实际声明中赋予抽象概念的名字。所以在 [ARM 1983] 中的类型现在是子类型。

3.2.3 对象名字

准则

- 对于逻辑对象使用判断语或从属的名字。
- 使用单数、泛指的名词作为对象标志符。
- 选用能说明对象执行时的值的标识符。
- 使用单数、泛指的名词作记录的组件。

范例

非逻辑对象:

```
Today            : Day;
Yesterday        : Day;
Retirement_Date  : Date;
```

逻辑对象:

```
User_Is_Available : Boolean; -- predicate clause
List_Is_Empty     : Boolean; -- predicate clause
Empty             : Boolean; -- adjective
Bright            : Boolean; -- adjective
```

原理

使用针对对象的名词建立了理解对象值的语境，这正是有关子类型命名的一点 (参见准则3.2.2)。这种风格下，对象的声明很像英文。例如，上面的第一个声明可读为着者 “Today is Day”。

泛指的名词用在记录的组件命名，因为一个记录的对象名会提供理解组件的语境。因此，下面的组件可理解为 “the year of retirement (退休的年份)”:

```
Retirement_Date.Year
```

把对象类型和语言联系起来，可以是代码看起来更像文本。例如，因为选取了恰当的名字，下面的代码不需要注释:

```
if List_Is_Empty then
   Number_Of_Elements := 0;
else
   Number_Of_Elements := Length_Of_List;
end if;
```

注解

如果很难找到一个合适的名词来声明一个对象在整个执行期间的值，这个对象很可能服务于多个目的。这种情况下，应该用多个对象。

3.2.4 标签类型和相对应包的命名

准则

- 使用一致的命名约定来对标签类型和相对应包命名。

具体事例

命名约定点燃了“信仰战争”；因此，这里展现展了两种具体实例。第一种实例集成了面向对象的使用特性。除了两种特殊情况，它在声明种使用同一命名约定，不管是否用到了面向对象的特性：

- 对标签类型和子类型的命名一致 (参见准则3.2.2)。
- 对于导出会有多个实现的抽象 (参见准则9.1.1) 的包，使用前缀 `Abstract_`。
- 对于提供可以和主要抽象“mixed in (混合)”的功能单元的包，用后缀 `_Mixin`。

第二种实现用特殊的名字和后缀来突出面向对象的特性的使用：

- 用代表的对象来命名类包，不用后缀 ([Rosen 1995])。
- 用混合包所代表的一面类命名，用后缀 `_Facet` ([Rosen 1995])。
- 用 `Instance` 命名主标签类型。
- 沿用某个类型的声明，对应的类程子类型用 `Class`。

范例

下面的两组从 [原理 1995] (§4.4.4, §4.6.2) 中选出的例子，使用了第一种命名约定。

在第一组的例子种，假设 `Set_Element` 已经在别的地方声明过了：

```
package Abstract_Sets is
   type Set is abstract tagged private;
   -- empty set
   function Empty return Set is abstract;
   -- build set with 1 element
   function Unit (Element: Set_Element) return Set is abstract;
   -- union of two sets
   function Union (Left, Right: Set) return Set is abstract;
   -- intersection of two sets
   function Intersection (Left, Right: Set) return Set is abstract;
   -- remove an element from a set
   procedure Take (From    : in out Set;
                   Element :    out set_Element) is abstract;
   Element_Too_Large : exception;
private
   type Set is abstract tagged null record;
end Abstract_Sets;
```

```
with Abstract_Sets;
package Bit_Vector_Sets is      -- one implementation of set abstraction
   type Bit_Set is new Abstract_Sets.Set with private;
   ...
private
   Bit_Set_Size : constant := 64;
   type Bit_Vector is ...
   type Bit_Set is new Abstract_Sets.Set with
       record
          Data : Bit_Vector;
       end record;
end Bit_Vector_Sets;
with Abstract_Sets;
package Sparse_Sets   -- alternate implementation of set abstraction
   type Sparse_Set is new Abstract_Sets.Set with private;
   ...
private
   ...
end Bit_Vector_Sets;
```

第二组例子是个支持视窗系统的混合包，应用了混合包的命名约定:

```
-- assume you have type Basic_Window is tagged limited private;
generic
   type Some_Window is abstract new Basic_Window with private;
package Label_Mixin is
   type Window_With_Label is abstract new Some_Window with private;
   ...
private
   ...
end Label_Mixin;
generic
   type Some_Window is abstract new Basic_Window with private;
package Border_Mixin is
   type Window_With_Label is abstract new Some_Window with private;
   ...
private
   ...
end Border_Mixin;
```

下面的例子应用了第二种命名约定，是 [Rosen 1995] 中讨论的:

```
package Shape is
   subtype Side_Count is range 0 .. 100;
   type Instance (Sides: Side_Count) is tagged private;
   subtype Class is Instance'Class;
   . . .
   -- operations on Shape.Instance
private
   . . .
end Shape;

with Shape; use Shape;
package Line is
   type Instance is new Shape.Instance with private;
   subtype Class is Instance'Class;
   . . .
   -- Overridden or new operations
private
   . . .
end Line;
with Shape; use Shape;
generic
```

```
   type Origin is new Shape.Instance;
package With_Color_Facet is
   type Instance is new Origin with private;
   subtype Class is Instance'Class;
   -- operations for colored shapes
private
   . . .
end With_Color_Facet;
with Line; use Line;
with With_Color_Facet;
package Colored_Line is new With_Color_Facet (Line.Instance);
```

声明看起来会是这样:

```
Red_Line : Colored_Line.Instance;
procedure Draw (What : Shape.Instance);
```

上述方案在用全名或用 use 语句都工作。只要你用同样的名字来表示所有的类型和类程类型，没有确认的名字总是相互隐藏。因此，编译器会要求你使用全名称来解决用 use 语句所带来的多义性 [Rosen 1995]。

原理

你应采用一个一致、可读、传达抽象意图的命名方案。理想状况下，命名方案应统一处理从不同方法用标签类型生成类别。如果命名约定太死板，从可读性的角度看，那你写的代码片段会不自然。对于推导或通用混合的 (9.4.1 类型夸展使用相近的命名约定，对象和过程的声明变得可读。

注解

类的命名约定在面向对象的抽象和其他抽象间画了一条分界线。工程师们已经用 Ada 83 ([ARM 1983]) 定义抽象数据类型有十多年了，你并不想只是为了使用某个类型的类型扩展而改变命名约定。你必须考虑所有在程序里使用的抽象中，继承有多重要。一般情况下，如果你喜欢突出抽象，而不是实现抽象的机制 (例如：继承、类型夸展以及多态)，也许你并不想强加这么一个苛刻的命名约定。一个平滑的从开发没有继承的抽象过渡到开发有继承的抽象的命名约定，不会影响质量。

如果你选用了突出使用面向对象特性的命名约定，而之后有想改变声明到不用面向对象特性，这个改变也许会代价高昂。你必须更换所有出现的名字，并且不引入别的错误。如果你选用了一个命名约定，禁止使用前、后缀来描述声明的特性，你失去了一个传递声明使用意图的机会。

3.2.5 程序单元的名字

准则

- 用行动的动词来命名过程和入口。
- 用判断式的子句命名逻辑函数。
- 用名词命名非逻辑函数。
- 给包的名字要暗含比子程序高级的组构。通常来说，这都是描述供应的抽象的名词。
- 给任务命名隐含活动的实体。
- 用描述被保护数据的名词来命名保护单元。
- 把通用子程序当做非通用子程序来命名。
- 把通用包当做非通用包来命名。
- 通用名字应该比例示的名字更广泛。

范例

下面是些一个 Ada 程序的构成元素的名字样本:

过程名字样本:

```
procedure Get_Next_Token — get is a transitive verb
procedure Create         — create is a transitive verb
```

逻辑函数名字样本:

```
function Is_Last_Item — predicate clause
function Is_Empty     — predicate clause
```

非逻辑函数名字样本:

```
function Successor — common noun
function Length    — attribute
function Top       — component
```

包名字样本:

```
package Terminals is      — common noun
package Text_Routines is — common noun
```

被保护对象样本:

```
protected Current_Location is — data being protected
protected type Guardian is    — noun implying protection
```

任务名字样本:

```
task Terminal_Resource_Manager is — common noun that shows action
```

下面这段代码样本，因为使用语言部分命名约定而呈现的清晰结果:

```
Get_Next_Token(Current_Token);
case Current_Token is
   when Identifier =>          Process_Identifier;
   when Numeric    =>          Process_Numeric;
end case; — Current_Token
if Is_Empty(Current_List) then
   Number_Of_Elements := 0;
else
   Number_Of_Elements := Length(Current_List);
end if;
```

当包和她的子程序一起被指定，能得到非常有描述性的代码:

```
if Stack.Is_Empty(Current_List) then
   Current_Token := Stack.Top(Current_List);
end if;
```

原理

当使用这些命名约定，代码会易于理解，读起来也象自然语言。当动词用于行动，如子程序，名词用于对象，例如子程序处理的数据，写出的代码更容易阅读和理解。这里模仿了读者熟悉的交流媒体。一段程序模拟了现实中的情况，使用这样的约定，减少了阅读和理解程序种的翻译次数。某种意义上，你选择的名字反应了从电脑硬件到应用要求的抽象级别。

有关标签类型相关的包中，特殊意义的后缀使用，参见准则3.2.4 。

注解

当前，对于任务的进入的命名，有些相互矛盾的命名约定。有些程序员和设计员提倡任务的进入使用和子程序一样的命名约定，模糊了程序中包含了任务的事实。他们的理由是如果任务用包重新实现或反之，名字不需要被替换。其他人则喜欢尽可能的突出任务存在的事实，以保证能分辨出任务的存在和它所带来的假设额外开销。可以根据项目侧重点的不同来选择使用适当的命名约定。

3.2.6 常量和命名的数字

准则

- 尽量用符号的名字而不是文字。
- 对于数学常量 Pi 和 e, 用预定义的常量 Ada.Numerics.Pi 和 Ada.Numerics.e。
- 用常量而不是变量来表示常数。
- 当一个值是某个类型所特有的或这个值必须静止，用常量。
- 尽量用命名的数字，而不是常量。
- 用命名的数字来代替文字的数字，它的类型或属性是真正的通用的。
- 对于对象的值在确立之后不能改变的，用常量 ([United Technologies 1987])。
- 用静止表达式来表示符号间的关系。
- 用线性的独立的文字组。
- 尽量用属性 'First 或者 'Last 而不是文字。

范例

```
3.14159_26535_89793                                  -- literal
Max_Entries : constant Integer       := 400;         -- constant
Avogadros_Number : constant := 6.022137 * 10**23;    -- named number
Avogadros_Number / 2                                 -- static expression
Avogadros_Number                                     -- symbolic value
```

声明 Pi 作为一个被命名的数字 (假设有对 [ARM 1995][§A.5] 中的预定义包 Ada.Numerics 使用 with 语句), 它可以被下面的赋值语句用符号引用:

```
Area := Pi * Radius**2; -- if radius is known.
```

而不用这样:

```
Area := 3.14159 * Radius**2; -- Needs explanatory comment.
```

另外， Ada.Characters.Latin_1.Bel 比 Character'Val(8#007#) 更具表达性。
常量和被命名数字的清晰度，可以因使用其他的常量和被命名数字而提高。例如:

```
Bytes_Per_Page    : constant := 512;
Pages_Per_Buffer  : constant := 10;
Buffer_Size       : constant := Pages_Per_Buffer * Bytes_Per_Page;
```

比以下的代码更能自我说明和维护:

```
Buffer_Size : constant := 5_120; -- ten pages
```

下面的文字应该用常量:

```
if New_Character = '$' then -- "constant" that may change
...
if Current_Column = 7 then  -- "constant" that may change
```

原理

用标识符而不是文字，使表达式的目的明确，也减少了注释的需要。含有数字文字表达式的常量更安全，因为他们不需要人工去计算。他们也比单独的数字文字有启发作用，因为有更多的机会植入解释性的名字。常量声明的清晰度可以因为在静态表达式中使用其他相关的常量来定义新的常量而提高。这并不会降低效率，因为被命名数字的静态表达式在编译时就被计算出来了。

一个常量是有类型的。一个被命名数字只能是通用类型: universal_integer 或 universal_real。强类型只对常量起作用，对被命名数字或文字不起作用。被命名数字允许编译器产生比常量更

有效的代码，编译时做更完全的错误检查。如果文字包含了许多个数字 (如上面例子中的 Pi)，使用标志符减少了击键错误。如果有击键错误，他们也更容易在检阅或编译时被指出。

文字的独立意味着少数几个文字的使用不会互相依赖，而任何常量和被命名值之间的关系由静态表达式表现。线性独立的文字值，给出这样一个机会，如果这个文字值变了，所有依赖于这个文字的被命名数字的值都会自动被改变。

参见准则**??**中更多有关选择无参数函数和常量的准则。

注解

有某些灰色地带，数字本身比名字更能自我说明。这是某些特殊的应用，通常在数字是大家都熟悉的，值不会变的情况下发生:

```
Fahrenheit := 32.0 + (9.0 * Celsius) / 5.0;
```

3.2.7 异常

准则

- 用异常所代表的问题种类命名。

范例

```
Invalid_Name: exception;
Stack_Overflow: exception;
```

原理

用检测到的问题作异常的名字，提高了代码的可读性。你应该尽量精确的命名你的异常，这样代码的维护者会理解为什么异常会发生。一个正确命名的异常，应该对使用客户有意义。

3.2.8 构造器

准则

- 使用前缀 New, Make 或 Create 命名构造器 (这里指创造或初始化对象的操作)。
- 对于含有构造器的子包，用可以标示内容的名字。

具体事例

- 命名一个含有构造器的子包 <whatever>.Constructor。

范例

```
function Make_Square (Center : Cartesian_Coordinates;
                      Side   : Positive)
  return Square;
```

原理

在构造器名字中使用前缀 New, Make 或 Create，令它的目的清晰。你也许希望限制前缀 New 只用于返回存取值的构造器，因为这个前缀暗示内部使用了分配符。

将所有的构造器都放在一个子包中是很有用的组织原则，即使他们返回存取值。

有关 Ada 的构造器的信息，参见准则9.2.3。

3.3 注释

源代码中的注释是个有争论的议题。对于注释是否会提高阅读性，反对和赞成的都有依据。实践中，注释最大的问题是，当代码改变时，人们经常忘记改变对应的注释，这导致注释会误导读者。注释应该保留给代码无法表达的内容，以及突出某些准则被违背的原因。如果可能，代码中的对象和单元应该使用可以自我说明的名字，简单、容易理解的程序结构也应该被使用，这样就需要非常少的注释。为了选择 (和键入) 合适的名字所做的额外努力，以及对设计干净和可理解的程序结构的额外思考是完全应该的。

使用注释来陈述代码的意图。概述代码的注释有助于维护的程序员看到整体意图。代码本身是如何实现的详细内容，不应该在注释中改述。

注释应该减至最少。他们应该提供 Ada 程序中无法表述的必须内容，侧重于代码结构，以及引起注意到故意和必须违反准则的地方。注释的出现应该让人注意到例示的真正问题或者补充示例程序中的不足。

维护的程序员要知道不相邻代码间的因果交互关系，以便对程序有一个差不多的全局观。他们一般都是通过人工模拟部分代码来得到这些信息。注释应该足够支持这个过程 ([Soloway et al. 1986])。

本节呈现了有关怎样写好的注释的通用准则。之后又定义了几个不同类型的注释，每种类型都有对应的使用准则。这些类型有文件头、程序单元规约头、程序单元主体头、数据注释、语句注释和标记注释。

3.3.1 一般注释

准则

- 尽可能写清晰的代码来减少注释的需求。
- 绝不要在注释里重复可以从代码中轻易获得的信息。
- 在需要注释的地方，要写得简明和完整。
- 注释中要用正确的语法和拼写。
- 注释要在视觉上和代码区分出来。
- 把注释放在数据头上，这样相关的信息可以被自动工具提取出来。

原理

结构和函数写得好的代码，没有注释也很清楚。对于晦涩的或结构糟糕的代码，无论有没有注释都很难令人理解、维护或重用。*坏代码应该被改进而不是解释*。阅读代码本身是唯一可以绝对确定代码做了些什么的方法，所以代码应该尽可能写得可读。

在注释中重复代码中的信息是个很不好的主意。有几个原因，首先，这不必要，而且降低效率。第二，代码在不断的改动，很难正确地维护重复的信息。当代码被改动时，它需要通过编译和测试来确定其正确性。但是，没有一种自动的机制可以保证注释能正确反映代码的改变。经常，注释中重复的信息在首次代码改动后就变得过时，但一直存在于软件的生命周期中。第三，当有关整个系统的注释用一个子系统的作者的有限眼光来写，这个注释通常从一开始就不正确。

注释在揭示从代码中很难或不可能得到的信息时很有必要。本书后续的章节中会有这样的注释。完全而简明的呈现所要求的信息。

注释的目的是帮助读者理解代码。拼写错误、不合语法、歧义或不完整的注释违背了这个目的。如果一个注释值得加入，那就值得正确的加入来增大它的有用性。

用缩进、和头组合在一起或者用虚线来突出注释，使得注释从视觉上和代码区分出来，这很有用，因为它使代码变得容易阅读。本书的后续章节会详细说明这点。

自动化注解

有关在注释中存放多余信息的准则是只在人工注释中应用。有些工具可以自动维护代码的相关信息 (例如，调用的单元、被调用的单元、交叉引用信息和修订历史等等)，这些信息存放在和代码同一文件的注释中。其他工具可以读取注释，但不更新它们，从注释中的信息可以自动生成详细的设计文档或其他报告。

这些工具的使用，鼓励或要求你构造头注释，这样它们才可以被自动提取或更新。注意，自动改变文件中的注释的工具，只有足够的经常运行才有用。自动生成的过时信息比人工的过时信息更危险，因为读者更相信它们。

配置管理工具对修订历史的维护更加准确和完整。在没有工具的支持下，工程师修改了代码而忘了更新修订历史是很常见的事。如果你的配置管理工具可以在源文件中以注释形式维护修订历史，不管在格式或位置上如何折衷，都要利用这个功能。在文件尾部附加一个完整的修订历史，要比在文件头有一个格式很好但不完整的历史要好。

3.3.2 文件头

准则

- 在每一个源文件中放一个文件头。
- 在文件头中，放入所有者、责任和历史信息。

具体事例

- 在文件头中放版权通告。
- 在文件头中放作者名字和部门。
- 在文件头中放修订历史，包括每个改动的概要，日期和修改的人。

范例

```
-------------------------------------------------------------------------
--      Copyright (c) 1991, Software Productivity Consortium, Inc.
--      All rights reserved.
-- Author: J. Smith
-- Department:System Software Department
-- Revision History:
--   7/9/91 J. Smith
--     - Added function Size_Of to support queries of node sizes.
--     - Fixed bug in Set_Size which caused overlap of large nodes.
--   7/1/91 M. Jones
--     - Optimized clipping algorithm for speed.
--   6/25/91 J. Smith
--     - Original version.
-------------------------------------------------------------------------
```

原理

如果你想要确定保护你对软件的权利，每一个源文件都应该有所有者信息。另外，为了有更高的可见性，它应该是文件中的第一项。

责任和修订信息也应该在每个文件中，这方便未来的维护者。这些头信息是维护者最相信的信息，因为它们累积。它们不会演变。没有修改文件作者名字或修订历史的需要，修订历史应该反映每次改动而更新。最糟的，也就是它不完整，它很少会出错。而且，一个单元的历史上修改的次数和频率，修改的人的个数都是很好的关于设计的实现完整性的指示器。

除了作者名字，有关如何找到最初作者的信息也应该包含在文件头里，如果有问题产生，这有助于维护者找到原作者。但是，详细的信息，象电话号码、邮政地址、办公室号码、电脑帐号用户名则太易变动而不太有用。最好记录下作者写代码时工作的部门。这个信息在作者欢办公室、部门甚至离开公司后还是有用的，因为这个部门很有可能还对原来版本负责。

注解

现代的配置管理系统，直接从文件头注释中截取版本历史是多余的。管理系统维护着一个从内容上看更可靠和一致的修改历史。某些系统可以重现一个单元的早期版本。

3.3.3 程序单元规约头

准则

- 在每一个程序单元规约都放一个头。
- 在程序单元规约头中放上用户使用要求信息。
- 除了单元名字，不要在头中重复规约中的信息。
- 解释单元做些什么，而不是怎样和为什么这么做。
- 描述这个程序单元的完整接口，包括它能抛出的任何异常和任何可以造成全局影响的地方。
- 不要包括这个单元如何适用于包含的软件系统的信息。
- 描述单元的性能特性 (时间、空间)。

具体事例

- 在头中放入程序名。
- 简单说明程序单元的目的。
- 对于包，描述可见子程序相互之间的影响和应当怎样一起使用它们。
- 列出单元抛出的所有异常。
- 列出单元所有的全局影响。
- 列出单元的前置条件和后置条件。
- 列出单元的激活的隐藏任务。
- 不要列出子程序的参数名字。
- 不要只为列出而列出子程序的名字。
- 不要列出单元中使用的其它单元的名字。
- 不要列出使用本单元的其它单元的名字。

范例

```
------------------------------------------------------------------------------
-- AUTOLAYOUT
-- Purpose:
--     This package computes positional information for nodes and arcs
--     of a directed graph. It encapsulates a layout algorithm which is
--     designed to minimize the number of crossing arcs and to emphasize
--     the primary direction of arc flow through the graph.
-- Effects:
--     - The expected usage is:
--       1. Call Define for each node and arc to define the graph.
--       2. Call Layout to assign positions to all nodes and arcs.
--       3. Call Position_Of for each node and arc to determine the
--          assigned coordinate positions.
--     - Layout can be called multiple times, and recomputes the
--       positions of all currently defined nodes and arcs each time.
--     - Once a node or arc has been defined, it remains defined until
--       Clear is called to delete all nodes and arcs.
```

```
-- Performance:
--     This package has been optimized for time, in preference to space.
--     Layout times are on the order of N*log(N) where N is the number
--     of nodes, but memory space is used inefficiently.
-------------------------------------------------------------------------
package Autolayout is
   ...
   -------------------------------------------------------------------------
   -- Define
   -- Purpose:
   --     This procedure defines one node of the current graph.
   -- Exceptions:
   --     Node_Already_Defined
   -------------------------------------------------------------------------
   procedure Define
         (New_Node : in       Node);
   -------------------------------------------------------------------------
   -- Layout
   -- Purpose:
   --     This procedure assigns coordinate positions to all defined
   --     nodes and arcs.
   -- Exceptions:
   --     None.
   -------------------------------------------------------------------------
   procedure Layout;
   -------------------------------------------------------------------------
   -- Position_Of
   -- Purpose:
   --     This function returns the coordinate position of the
   --     specified node. The default position (0,0) is returned if no
   --     position has been assigned yet.
   -- Exceptions:
   --     Node_Not_Defined
   -------------------------------------------------------------------------
   function Position_Of (Current : in Node)
         return Position;
   ...
end Autolayout;
```

原理

程序单元规约注释头的目的是帮助用户理解怎样使用本单元。从阅读规范和注释头中，用户应给知道有关如何使用单元的所有必须的信息。它不应该还需要阅读单元主体。所以，每一个单元规约都应该有一个注释头，每个头应该包含所有没有在规范中表达的使用信息。这些信息应包括单元之间的相互影响，对共享资源的影响，异常的抛出，以及时间、空间的特性。这些信息都不可能从 Ada 的程序单元规约中得到。

当你在注释头中重复可以从规约中读得的信息，这些信息在维护过程中会趋于不准确。例如，当定义一个过程时，列出所有的参数名字、模式或子类型并无意义，这些信息在规约中都有。同样的，不要在注释头中列出包中所有的子程序，除非这是有关子程序的重要声明中必须的。

不要在注释头中加入用户不需要的信息。特别是不要包含程序单元如何完成它的功能或为什么运用了某个特定的算法。这些信息应该藏在程序体内，以保持单元的抽象性。如果用户知道这些信息，并由此作出一些决定，当这些信息改变后，程序也许会因此而出错。

当叙述单元的目的时，避免提及系统内其它部分。“本单元做... ...” 比 “本单元由 Xyz 调用做... ...” 要好。程序单元应该编写为不必知道和搭理谁调用它，这使得单元更具广泛性和重用性。另外，有关其它单元的信息很容易在维护过程中变得过时和不正确。

单元的性能特性 (时间和空间) 应该包含在性息中。许多这些信息并没有在 Ada 规约中出现，但是用户需要。要整合单元到系统中，用户需要知道资源 (CPU、内存，等等) 的使用状况。

特别重要的是当一个子程序调用会激活一个程序体中隐藏的任务，这个任务有可能在子程序结束后还在继续消耗资源。

注解

某些项目把大多数的注释都放在程序单元结尾而不是开头。他们的依据是单元只编写一次，而被阅读许多次，长长的头注释让规约的开始很难被找到。

异例

当一组程序单元相互有紧密联系或者很容易理解，可以用一个头注释来解释这一组单元。例如，emph 最大和最小函数;Sin, Cos 和 Tan 函数; 或者一个包中的针对某一对象的一组查询属性的函数。当某组中的每一个函数都可以抛出相同的异常，这就更加需要一个注释。

3.3.4 程序单元主体注释头

准则

- 单元维护者需要的信息。
- 解释单元如何以及为什么进行它的功能，而不是单元做什么。
- 不要重复从代码中就能轻易得到的信息 (除了单元名字)。
- 不要重复规约头中有的信息 (除了单元名字)。

具体事例

- 放上程序单元名字。
- 记录移植问题。
- 概述复杂的算法。
- 记录重大或有争议的实现决定原因。
- 记录丢弃的替代实现，和丢弃的原因。
- 记录预期的改变，特别是部分工作已经完成，以便改变更容易完成。

范例

```
------------------------------------------------------------------------
-- Autolayout
-- Implementation Notes:
--    - This package uses a heuristic algorithm to minimize the number
--      of arc crossings.  It does not always achieve the true minimum
--      number which could theoretically be reached.  However it does a
--      nearly perfect job in relatively little time.  For details about
--      the algorithm, see ...
-- Portability Issues:
--    - The native math package Math_Lib is used for computations of
--      coordinate positions.
--    - 32-bit integers are required.
--    - No operating system specific routines are called.
-- Anticipated Changes:
--    - Coordinate_Type below could be changed from integer to float
--      with little effort.  Care has been taken to not depend on the
--      specific characteristics of integer arithmetic.
------------------------------------------------------------------------
package body Autolayout is
   ...
```

```
   ------------------------------------------------------------------
   -- Define
   -- Implementation Notes:
   --    - This routine stores a node in the general purpose Graph data
   --      structure, not the Fast_Graph structure because ...
   ------------------------------------------------------------------
   procedure Define
         (New_Node : in      Node) is
   begin
      ...
   end Define;
   ------------------------------------------------------------------
   -- Layout
   -- Implementation Notes:
   --    - This routine copies the Graph data structure (optimized for
   --      fast random access) into the Fast_Graph data structure
   --      (optimized for fast sequential iteration), then performs the
   --      layout, and copies the data back to the Graph structure.  This
   --      technique was introduced as an optimization when the algorithm
   --      was found to be too slow, and it produced an order of
   --      magnitude improvement.
   ------------------------------------------------------------------
   procedure Layout is
   begin
      ...
   end Layout;
   ------------------------------------------------------------------
   -- Position_Of
   ------------------------------------------------------------------
   function Position_Of (Current : in      Node)
         return Position is
   begin
      ...
   end Position_Of;
   ...
end Autolayout;
```

原理

单元主体注释头的目的是帮助维护者理解单元的实现，包括各种不同技术的替代妥协。要保证记录实现过程中所有的决定，避免维护者犯同样的错误。对维护者最有用的一种注释是有关某种改变为什么不行的描述。

注释头也是记录移植性考虑的好地方。维护者也许需要移植程序到不同的环境，一个不可移植的特性列表对此帮助很多。另外，收集和记录移植性问题本身，令注意力集中在这些问题上，那从一开始就会产生可移植性的更高的代码。

如果复杂的算法很难从阅读代码中理解，在注释中加入算法的概述，但不要仅仅改写一下代码。那样的重复的内容没有必要也很难维护。同样的，不要重复规约头中的信息。

注解

经常的，一个程序单元可以自我解释，不需要一个解释头。这样的情况，完全乎略掉解释头，就像上面的 Position_Of。记住要确定你乎略的信息真的不包含信息。例如，看看下面两个头的不同：

```
-- Implementation Notes: None.
```

和：

```
-- NonPortable Features: None.
```

第一条信息是作者对维护者说:“我想不到任何其他东西告诉你。”而第二条信息意思是“我保证本单元完全可移植性。”

3.3.5 数据注释

准则

- 除非他们的名字能自我解释，否则注释所有的数据类型，对象和异常。
- 从语意上注释复杂的指针类的数据结构。
- 注释数据对象之间存在的关系。
- 乎略名字中包含的信息。
- 对于标签类型，如果你期望它的特殊化 (例如衍生类型) 复写某些重分派操作，注释重分派的信息。

范例

可以用目的来对对象分组，那注释为：

```
...
-------------------------------------------------------------------------
-- Current position of the cursor in the currently selected text
-- buffer, and the most recent position explicitly marked by the
-- user.
-- Note:  It is necessary to maintain both current and desired
--        column positions because the cursor cannot always be
--        displayed in the desired position when moving between
--        lines of different lengths.
-------------------------------------------------------------------------
Desired_Column  : Column_Counter;
Current_Column  : Column_Counter;
Current_Row     : Row_Counter;
Marked_Column   : Column_Counter;
Marked_Row      : Row_Counter;
```

应该注释异常被抛出的条件：

```
-------------------------------------------------------------------------
-- Exceptions
-------------------------------------------------------------------------
Node_Already_Defined : exception; -- Raised when an attempt is made
                                  --|   to define a node with an
                                  --|   identifier which already
                                  --|   defines a node.
Node_Not_Defined     : exception; -- Raised when a reference is
                                  --|   made to a node which has
                                  --|   not been defined.
```

下面是个更加复杂的例子，包括了多个记录，访问类型来组成一个复杂的数据结构：

```
-------------------------------------------------------------------------
-- These data structures are used to store the graph during the
-- layout process. The overall organization is a sorted list of
-- "ranks," each containing a sorted list of nodes, each containing
-- a list of incoming arcs and a list of outgoing arcs.
-- The lists are doubly linked to support forward and backward
-- passes for sorting. Arc lists do not need to be doubly linked
-- because order of arcs is irrelevant.
-- The nodes and arcs are doubly linked to each other to support
-- efficient lookup of all arcs to/from a node, as well as efficient
-- lookup of the source/target node of an arc.
-------------------------------------------------------------------------
type Arc;
type Arc_Pointer is access Arc;
```

```
type Node;
type Node_Pointer is access Node;
type Node is
   record
      Id        : Node_Pointer;— Unique node ID supplied by the user.
      Arc_In    : Arc_Pointer;
      Arc_Out   : Arc_Pointer;
      Next      : Node_Pointer;
      Previous  : Node_Pointer;
   end record;
type Arc is
   record
      ID      : Arc_ID;         — Unique arc ID supplied by the user.
      Source  : Node_Pointer;
      Target  : Node_Pointer;
      Next    : Arc_Pointer;
   end record;
type Rank;
type Rank_Pointer is access Rank;
type Rank is
   record
      Number     : Level_ID;  — Computed ordinal number of the rank.
      First_Node : Node_Pointer;
      Last_Node  : Node_Pointer;
      Next       : Rank_Pointer;
      Previous   : Rank_Pointer;
   end record;
    First_Rank : Rank_Pointer;
Last_Rank  : Rank_Pointer;
```

原理

用注释来解释目的、结构、数据结构的语义对理解代码很有帮助。许多维护者在试图理解单元实现时，都会先看数据结构。理解数据的储存，同时还有不同数据之间的联系和数据在单元中的流程是理解单元内容的第一步。

在上面的第一个例子中，变量名 Current_Column 和 Current_Row 相对是自我注释的。变量名 Desired_Column 是个好名字。但他让读代码的人疑惑 Current_Column 和 Desired_Column 之间的关系。注释还解释了要有这两个变量的原因。

另一个注释数据声明的好处是一组数据声明注释可以代替可能分布在在代码中各个数据被操作的地方。在上面的第一个例子中，注释简单扩展了 “current” 和 “marked” 的意义。它陈述了当前 (current) 的方位是光标的位置。当前方位在当前缓冲中，而标记 (marked) 方位是由用户标记的。这些注释，和辅助的变量名，大幅减少了代码中需要注释的各个单独语句。

记录异常的完全意思及其被抛出的条件是很重要的，这在上面第二个例子中可看到，特别是当异常是在规约包中声明的。如果没有读包体中的代码，读者没有其他方法得到异常的准确意思。

象第二个例子中那样，把所有的异常都组在一起，就象给读者提供了一份特殊情况的词汇表。当包中不同的子程序可以抛出同样的异常时，这个很有用。如果包中每个异常只能被一个子程序抛出，那把异常和相关的子程序组在一起比较好。

当注释异常时，用一般用语来描述异常的意思比列出所有会抛出此异常的子程序更好；那样的列表很难维护。当一个新例程加入后，这个列表很有可能没有跟着更新。另外，这些信息在子程序的注释中已经有了，所有子程序能抛出的异常都应该在注释中列出。列出子程序的异常比列出抛出异常的子程序要有用和更容易维护。

第三个例子中，记录的域名短而易记，但他们并不能完全自述。这通常在包含访问类型的复杂的数据结构中发生。无论给记录和域起什么名字，都没有办法完全解释整个记录和指针结构成为一个嵌套和排序的链表。例子中的注释就是这样，很有用。没有了注释，读者不会知道哪些链表已经排序，哪些是双向链表和为什么。注释解释了作者对这个复杂数据结构的意图。

如果维护者想确认双向链表实现正确，还是必需要阅读代码。有着这样的准备，当维护者阅读代码是，就比较容易找到一个臭虫：一个指针改变了，但另一个方向的没有。

有关注释重分派操作的原理，参见9.2.2。(重分派是指把一个基元操作的参数转化为一个类宽类型，然后分派到另一个基元操作。) 在准则9.2.2的原理中，讨论了这样的文档应该放在规约文件还是体文件中。

3.3.6 语句注释

准则

- 尽量减少语句中包含的注释。
- 只注释代码不够清晰的部分。
- 注释代码中有意省略的地方。
- 不要用注释来改写代码。
- 不要解释别的地方的代码，例如被本单元调用的子程序。
- 当需要注释的时候，要让注释跟代码明显区分出来。

范例

下面是个注释的很糟糕的例子：

```
...
-- Loop through all the strings in the array Strings, converting
-- them to integers by calling Convert_To_Integer on each one,
-- accumulating the sum of all the values in Sum, and counting them
-- in Count.  Then divide Sum by Count to get the average and store
-- it in Average. Also, record the maximum number in the global
-- variable Max_Number.

for I in Strings'Range loop
   -- Convert each string to an integer value by looping through
   -- the characters which are digits, until a nondigit is found,
   -- taking the ordinal value of each, subtracting the ordinal value
   -- of '0', and multiplying by 10 if another digit follows.  Store
   -- the result in Number.
   Number := Convert_To_Integer(Strings(I));
   -- Accumulate the sum of the numbers in Total.
   Sum := Sum + Number;
   -- Count the numbers.
   Count := Count + 1;
   -- Decide whether this number is more than the current maximum.
   if Number > Max_Number then
      -- Update the global variable Max_Number.
      Max_Number := Number;
   end if;
end loop;
-- Compute the average.
Average := Sum / Count;
```

下面的例子改正了上面的错误：不在注释中重复代码中明显的内容；不描述 Convert_To_Integer 内部的详细内容；删除错误的代码 (在和的累加那行语句)；让剩下的少数注释更加明显从代码中区分出来。

```
Sum_Integers_Converted_From_Strings:
   for I in Strings'Range loop
      Number := Convert_To_Integer(Strings(I));
      Sum := Sum + Number;
```

```
            Count := Count + 1;
            -- The global Max_Number is computed here for efficiency.
            if Number > Max_Number then
               Max_Number := Number;
            end if;
         end loop Sum_Integers_Converted_From_Strings;

      Average := Sum / Count;
```

原理

范例中注释的改善不单单是减少了注释的数量，而是减少了无用注释的数量。

改写或解释明显的代码的注释没有任何价值。他们浪费了作者的工夫去写，也浪费了维护者的工夫去更新。所以，他们常常在后来变成不准确的。这样的注释也让代码显得杂乱，隐藏了少数重要的注释。

描述别的单元内部的信息，违反了信息隐藏的原则。有关 Convert_To_Integer 的具体内容跟调用的单元无关，这些内容最好一直隐藏着，以免算法改变了。试图解释别的地方的代码很难维护，而且几乎总是在第一次代码修正时就变得不正确了。

让注释从代码中明显的显现出来，这样的好处是代码更容易阅读，少数重要的注释也能突显出来。突显不寻常或特殊的代码的特性，以说明他们是有意的。这帮助维护者在维护或移植时，把注意力集中到那些容易产生问题的代码部分。

注释应该记录不可移植的，实现依赖性的，环境依赖性的，或者棘手的代码。他提醒读者这些不寻常的地方是有原因的。象解释针对编译器错误的变通方法是个有用的注释。如果你用了比较低级别的解决方法 (在软件工程角度上看不是“完美”的)，请注释它。注释包含的信息要陈述你选用某个构造的原因。还要包含那些失败的尝试，例如，用高级别的构造。这些信息对维护者在历史维护上有帮助。你对读者展现了在选择这个构造是进行了相当的思考。

最后，注释应该用来解释代码中所没呈现和存在的东西。如果你作了一个刻意的决定不去进行某些动作，例如在使用结束后，释放某个数据结构，一定要记得注释为什么不释放。否则，维护者可能注意到这个明显的乎略而“改正”了，也就引入了错误。

在准则9.2.1中，讨论了对标签类型和重分派该准备什么样的文档。

注解

对于上面的例子，还可以改进，把变量 Count 和 Sum 在本地的块声明，这样他们的作用域就局限在本地，而且他们的初始化也靠近使用的地方。例如，命名块为 Compute_Average 或把代码转移到一个名为 Average_Of 的函数里。Max_Number 的计算也可以从 Average 中分离出来。但是这些改变有别的准则来参考，这里的例子只是用来示范注释的使用。

3.3.7 标示注释

准则

- 用分页标示来标识单元边界 (参见准则2.1.7)。
- 如果包程序体、子程序体、任务体、或块的开始之前有声明部分，那在注释里重复单元名字来标示他们的开始。
- 对于很长或嵌套得很厉害的 if 或 case 语句，在结束的地方用一个总结语句控制条件的注释来标示结束。
- 对于很长或嵌套得很厉害的 if 语句，用一个总结 else 部分语句控制条件的注释来标示 else 部分。

范例

```
if     A_Found then
   ...
elsif B_Found then
   ...
else  -- A and B were both not found
   ...
   if Count = Max then
      ...
   end if;
   ...
end if;  -- A_Found
```

```
package body Abstract_Strings is
   ...
   procedure Concatenate (...) is
   begin
      ...
   end Concatenate;
   ...
begin  -- Abstract_Strings
   ...
end Abstract_Strings;
```

原理

标示注释突出了代码的结构，使得阅读起来更容易。他们可以是分隔代码段的线或构造的说明标签。他们帮助读者了解在代码中的当前位置。这对大型的单元比小单元更有用。一个小巧的标示可以和被注释的内容在一行内，因此他增加了信息而没有引起混乱。

`if`，`elsif`，`else` 和 `end if` 经常被长序列的语句分离，有时还包含其他的 `if` 语句。如上面第一个范例中，标示注释突出了同个语句中和相距比较远的对应关键词间的联系。对于块语句和循环语句，标签注释不是必须的，因为这些语句的语法允许命名并且在结束时重复。用名字比用标示注释更好，因为编译器在开始还和结束时会检验名字。

包体内的语句序列通常离包的第一行比较远。许多子程序体会先出现，每个都可能包含许多行。在第二个范例中，标签注释突出了和包开始时的对应关系。

注解

如果过分重复名字和加注条件表达式会让代码很混乱。标示注释的好处在于他的视觉距离，特别是分页。

3.4 使用类型

强类型促进软件的可靠性。一个对象的类型定义了所有合法的值和操作，这允许编译器在编译时检验和识别潜在的错误。另外，类型的规则允许编译器生成在执行时检验违反类型约束的代码。运用这些 Ada 编译器的特性，比其他没那么强类型的语言更早以及更完整的检测出错误。

3.4.1 类型声明准则

准则

- 尽量限制标量类型的值域。
- 从应用中查找可能值的信息。

- 不要重用标准 (Standard) 包中的子类型名称。
- 用子类型声明来促进程序的可读性 ([Booch 1987])。
- 协调使用导出类型和子类型 (5.1.1)。

范例

```
subtype Card_Image is String (1 .. 80);
Input_Line : Card_Image := (others => ' ');
-- restricted integer type:
type    Day_Of_Leap_Year     is                    range 1 .. 366;
subtype Day_Of_Non_Leap_Year is Day_Of_Leap_Year range 1 .. 365;
```

下面的声明，程序员的意思是:“我不知道有多少。”但是实际的基本值域会掩埋在代码中或作为系统参数:

```
Employee_Count : Integer;
```

原理

把无意义的数值从合法的范围内剔除，增加了编译器检测到错误的能力。这也提高了程序的可读性。另外，当对象声明为这些子类型时，这也让你仔细考虑每次对象的使用。

不同的实现对于大多数预定义类型提供了不同的一套数值。读者不能从预定义的名字知道预期的值域。当预定义名字被重载这样的情形会更恶化。

对象和子类型的名字可以阐明他们预期的使用和作为底层设计决定的文档。上面的范例中，记录了限制软件的设计决定：只能用在物理参数符合穿孔卡特性的设备。这些信息很容易找到，因此提高了程序的可维护性。

可以通过声明一个没有限制的子类型来实现类型的重命名 ([ARM 1995] §8.5)。子类型的名字不可以被重载；重载只能用在客调用的实体。枚举的实字被当作无参数的函数，所以包含在这个规则里。

类型可以是有高度约束的一组值，同时没有排除掉有用的值。在5.1.1中描述的用法在执行语句中排除了许多标志变量和类型转换。这令程序更易读，同时也允许编译器强化强类型约束。

注解

子类型声明没有定义一新类型，只是在现有的类型上加了限制。

任何与本准则的偏差，都会导致对 Ada 语言中强类型机制的损伤。

异例

有些情况下，你并不需要依赖某个特定的数值范围。这些情况确实存在，例如数组的索引(如一个列表的大小没有任何语义上的限制)。参见准则 7.2.1 有关正确使用预定义类型的讨论。

3.4.2 枚举类型

准则

- 用枚举类型替代数字码。
- 只有在绝对必要的时候，才用表示语句来匹配外部设备的需求。

范例

用:

```
type Color is (Blue, Red, Green, Yellow);
```

来代替:

```
Blue   : constant := 1;
Red    : constant := 2;
Green  : constant := 3;
Yellow : constant := 4;
```

如果学要才加入以下的内容:

```
for Color use (Blue   => 1,
               Red    => 2,
               Green  => 3,
               Yellow => 4);
```

原理

枚举比数字码更健壮。在遇到不正确的解释或在维护中增加或删除数值的时候，因此而引起的潜在错误会少。数字码是从那些没有用户类型定义的语言遗留下来的。

另外，Ada 提供了一系列属性操作给枚举类型 ('Pos, 'Val, 'Succ, 'Pred, 'Image 和 'Value)，用起来比用户自己写的编码操作更可靠。

一开始的时候，数字码也许看起来适合用在匹配外部数值。这种情况就需要枚举类型的表示语句。表示语句记录了“编码”。如果程序结构能正确分隔和封装硬件依赖 (参见准则 **[?]**)，数字代码就会封装在接口包中，如果要求改变了，这很容易被找到和取代。

通常，尽量避免使用枚举类型的表示语句。当枚举文字没有明显的排列顺序，枚举的表示语句可能会引起移植性问题。例如由于新平台的表示顺序的改变，枚举的顺序必须跟着改变。

3.5 总结

拼写

- 用下划线分隔复合名字中的词。
- 使用一致的风格来呈现数字。
- 使用对问题适当的根植来呈现字面值。
- 使用下划线来分隔数字，就像在普通文字中用逗号或句号 (或对于非十进制用空格)。
- 当使用科学标示法，用一致的大写或小写的 E。
- 在非十进制数字中，要么使用全大写或小写来标示其中的字母。
- 让保留字和程序的其他组件从视觉上分开。
- 如果一个短小的同义词存在，不要使用长词语的缩写。
- 用一致的缩写策略。
- 不要用有多种解释的缩写。
- 缩写必须比原词节省了许多字符，以认可它的使用。
- 使用应用领域通用的缩写。
- 维护一个缩写列表，只使用列表中的缩写。

命名约定

- 尽量选择可以自我说明的名字。
- 使用短的同义词而不用缩写。
- 使用应用的名词，但是不要用偏僻的术语。
- 避免使用同一个名字来声明不同类型的标识符。

- 使用单数、泛指的名词作为子类型标志符。
- 选用能说明子类型值的标志符。
- 对定义可见的访问类型、子界、数组，考虑在子类型标志符中用后缀。
- 对私有类型，不要使用针对子类型标志符的构造规则 (例如加后缀)。
- 不要使用预定义包中的子类型名字。
- 对于逻辑对象使用判断语或从属的名字。
- 使用单数、泛指的名词作为对象标志符。
- 选用能说明对象执行时的值的标识符。
- 使用单数、泛指的名词作记录的组件。
- 使用一致的命名规则来对标签类型和相对应包命名。
- 用行动的动词来命名过程和入口。
- 用判断式的子句命名逻辑函数。
- 用名词命名非逻辑函数。
- 给包的名字要暗含比子程序高级的组构。通常来说，这都是描述供应的抽象的名词。
- 给任务命名隐含活动的实体。
- 用描述被保护数据的名词来命名保护单元。
- 把通用子程序当做非通用子程序来命名。
- 把通用包当做非通用包来命名。
- 通用名字应该比例示的名字更广泛。
- 尽量用符号的名字而不是文字。
- 对于数学常量 Pi 和 e, 用预定义的常量 Ada.Numerics.Pi 和 Ada.Numerics.e。
- 用常量而不是变量来表示常数。
- 当一个值是某个类型所特有的或这个值必须静止，用常量。
- 尽量用命名的数字，而不是常量。
- 用命名的数字来代替文字的数字，它的类型或属性是真正的通用的。
- 对于对象的值在确立之后不能改变的，用常量 ([United Technologies 1987])。
- 用静止表达式来表示符号间的关系。
- 用线性的独立的文字组。
- 尽量用属性 'First 或者 'Last 而不是文字。
- 用异常所代表的问题种类命名。
- 使用前缀 New, Make 或 Create 命名构造器 (这里指创造或初始化对象的操作)。
- 对于含有构造器的子包，用可以标示内容的名字。

注释

- 尽可能写清晰的代码来减少注释的需求。
- 绝不要在注释里重复可以从代码中轻易获得的信息。
- 在需要注释的地方，要写得简明和完整。
- 注释中要用正确的语法和拼写。
- 注释要在视觉上和代码区分出来。
- 把注释放在数据头上，这样相关的信息可以被自动工具提取出来。
- 在每一个源文件中放一个文件头。

- 在文件头中，放入所有者、责任和历史信息。
- 在每一个程序单元规约都放一个头。
- 在程序单元规约头中放上用户使用要求信息。
- 除了单元名字，不要在头中重复规约中的信息。
- 解释单元做些什么，而不是怎样和为什么这么做。
- 描述这个程序单元的完整接口，包括它能抛出的任何异常和任何可以造成全局影响的地方。
- 不要包括这个单元如何适用于包含的软件系统的信息。
- 描述单元的性能特性 (时间、空间)。
- 单元维护者需要的信息。
- 解释单元如何以及为什么进行它的功能，而不是单元做什么。
- 不要重复阅读代码就能轻易得到的信息 (除了单元名字)。
- 不要重复规约头中有的信息 (除了单元名字)。
- 除非他们的名字能自我解释，否则注释所有的数据类型，对象和异常。
- 从语意上注释复杂的指针类的数据结构。
- 注释数据对象之间存在的关系。
- 乎略名字中包含的信息。
- 对于标签类型，如果你期望它的特殊化 (例如衍生类型) 复写某些重分派操作，注释重分派的信息。
- 用分页标示来标识单元边界 (参见准则2.1.7)。
- 如果包程序体、子程序体、任务体、或块的开始之前有声明部分，那在注释里重复单元名字来标示他们的开始。
- 对于很长或嵌套得很厉害的 `if` 或 `case` 语句，在结束的地方用一个总结语句控制条件的注释来标示结束。
- 对于很长或嵌套得很厉害的 `if` 语句，用一个总结 `else` 部分语句控制条件的注释来标示 `else` 部分。

类型的使用

- 尽量限制标量类型的值域。
- 从应用中查找可能值的信息。
- 不要重用标准 (Standard) 包中的子类型名称。
- 用子类型声明来促进程序的可读性 ([Booch 1987])。
- 协调使用导出类型和子类型 (5.1.1)。
- 用枚举类型替代数字码。
- 只有在绝对必要的时候，才用表示语句来匹配外部设备的需求。

第 四 章

程序结构

适当的结构提高程序的清晰度。这好比底层的可读性并促进可读性准则 (第3章) 的使用。Ada 提供的各种程序结构设施是为了增进设计的整体清晰而设计的。本章准则会展示这些设施的目的。

子包的概念支持子系统的概念，在 Ada 中，子系统由分级结构的库单元表示。通常，大型系统应由一系列子系统构建。子系统应该用来表示逻辑上相关的库单元，这些单元集合起来实现了一个高层抽象或架构。

包的概念和私有类型支持抽象和封装的概念。相关的数据和子程序可以组在一起，这样高层可以把他们看成单一实体。强类型，包和子程序的规约和实体的分离加强了信息隐藏。异常和任务是会影响程序结构的另外两个 Ada 语言元素。

4.1 高级结构

构建良好的程序很同意被理解、提高和维护。构建糟糕的程序在维护时，经常需要重新构建以方便工作。下面列出的许多准则，是作为通用程序设计准则给出的。

4.1.1 分别编译能力

准则

- 把每一个库单元的规约放在独立于实体的文件中。
- 避免定义不是主程序的库单元子程序。如果定义了这样的子程序，对每一个库单元子程序创建一个明确的单独的规约文件。
- 把亚单元的使用降到最低。
- 对于亚单元而言，用子库单元来把一个子系统构建为可管理的多个单元。
- 每一个亚单元存放在独立的文件中。
- 使用一致的文件命名约定。
- 对于包主体的嵌套，用私有子包并在父包中引用。
- 对于 (其他) 子单元用于扩展父单元的抽象或服务需要的数据和子程序，用私有子单元规约。

范例

下面的文件名说明了一种可能的文件组织以及相随的一致命名约定。库单元的名字用后缀 adb 来表示单元主体，后缀 ads 表示单元规约，任何其他包含亚单元的文件的名字，由体名字和亚单元名字以下划线分隔:

```
text_io.ads                  — the specification
text_io.adb                  — the body
text_io_integer_io.adb       — a subunit
text_io_fixed_io.adb         — a subunit
text_io_float_io.adb         — a subunit
text_io_enumeration_io.adb   — a subunit
```

依赖于文件系统允许文件名中可以用什么字符，你可以在文件名字中用更清晰的方式来区分父和亚单元的名字。如果文件系统允许使用 "#" 字符，那可以用 # 来分隔主体和亚单元的名字:

```
text_io.ads                  -- the specification
text_io.adb                  -- the body
text_io#integer_io.adb       -- a subunit
text_io#fixed_io.adb         -- a subunit
text_io#float_io.adb         -- a subunit
text_io#enumeration_io.adb   -- a subunit
```

有些操作系统区分大小写，尽管 Ada 本身是不区分大小写的语言。这时，可以选择全用小写来表示文件名。

原理

本准则侧重在分散文件上的主要原因，是最小化每次改动后需要重新编译的文件数。通常，在软件开发过程中，主体的改动要比规约要频繁得多。如果规约和主体在同一个文件中，那么每次主体的编译也会编译一次规约，虽然规约并没有改变。因为规约定义了单元和其所有使用者之间的接口，每次规约的重编译都会导致所有使用者必须重编译以确认符合规约。如果使用者的规约和主体也在一个文件里，那任何使用这些单元的使用者也要重编译，以此类推。这个涟漪效应，会强制大量本可以避免的重编译，严重拖慢了项目的开发和测试阶段。这就是为什么你要把所有库单元 (非嵌套单元) 中的规约和其主体分别存放在不同的文件中。

库单元中子程序要尽量短小。库单元子程序的真正运用只有作为主子程序时。几乎在其他任何情况下，把子程序嵌套在包中会更好些。这为子程序在主体中提供了一个本地化所需数据的场所。而且，这也减低了系统中单独模块的数量。

通常，对于任何用 `with` 语句引用的库中的子程序，都应有一个单独的规约。这让使用的单元只依赖于库子程序的规约，而不是其主体。

亚单元的使用也应该尽量的少，因为他们产生维护问题。父主体中的声明在亚单元中是可见的，这增加了亚单元中的全局数据，因此提高了改动带来的潜在涟漪效应。亚单元阻碍了代码的重用，因为他提供了把可能重用的代码直接放在亚单元的便利，而不是把他们放在一个通用的例程以供其他子程序调用。

Ada 95 提供了子库单元的功能，可以避免大多数亚单元的使用。例如，用子库单元和对应的语境语句来替代亚单元庞大而嵌套的主体。子库单元主体的改变不会引起子系统中其他单元的重编译。

另一个使用多个独立文件的好处是，不同的程序员可以使用普通的编辑器同时修正系统的不同部分，一般的编辑器不允许同时对一文件进行多处修改。

最后，将规约和主体分离可以实现一个规约对应多个主体或多个规约对应一个主体。虽然 Ada 要求在一个系统中的任何时间，一个主体只能有一个规约，但是维护多主体或多规约对于系统的不同构建仍然有好处。例如，一个规约可以有多个主体，每个主体都实现相同的功能，但是在时间和空间的折衷不同，或者是依赖机器的代码，每一个目标机器都有一个对应的主体。维护多规约在开发和调试阶段也很有用。例如，一个规约提供给客户，另一个给单元测试。前一个只导出在正常操作下外部可以使用的子程序，后一个导出所有的子程序，以便独立测试每一个子程序。

一致的文件命名约定是值得推荐的，这有利于管理由于遵循本准则而产生的大量文件。

在实现包规约中的抽象时，通常需要写一些支持的子程序来操纵内部数据。不要在接口中导出这些子程序。可以选择把他们放在父包的主体中，或者父包主体中语境语句引用的子包中。当放在父包的主体中时，所有使用父包的客户都不能访问他们，包括扩展父包的子包也不可以。如果实现父包抽象的扩展时，要用到这些子程序，父包的规约和主体都将被迫修改，因为扩展必须要在父包规约中声明。这个方法就会引起整个包 (规约和主体) 和所有客户的重编译。

也可以用私有子包来实现这些支持子程序。因为父单元的规约没有更改，他和他的客户都不需要重编译。本来要在父单元主体中声明的数据和子程序，改在私有子单元的规约中声明，这样他们在父单元主体和任何扩展父单元服务和抽象的子单元中都可见。(参见准则4.1.6 和 **??**)。私有子单元的使用通常会将本单元家族及其客户重编译的次数变得最少。

当声明子包为私有时，相当于在父包主体中声明的效果，父包的客户不可以用语境语句来指定私有子包。这么做很有灵活性，用子包来扩展父包的抽象时，如果没有修改父包的规约和主体，那就不用重编译父包。这个灵活性通常会用来抵消因此而增加的单元之间的依赖性，在这里指父主体或其他子包主体中用于指定私有子包的多出来的语境语句。

4.1.2 配置编译指示

准则

- 尽可能通过编译器参数或其他不需要修改代码的形式来配置编译指示。
- 当配置编译指示必须放在代码中时，考虑在每一个分区中把他们隔离在一个编译单元里，如果要指定，推荐用分区的主子程序。

原理

编译指示一般是作为分区或系统范围的参数。通常，他们反应高级别软件架构的决策 (例如：pragma Task_Dispatching_Policy) 或者在某特定应用领域之使用 (例如：安全关键软件)。如果编译指示内嵌在某个软件组件中，当组件在配置并不适用的环境中重用，就会在新软件中产生问题。这包括被编译系统拒绝或运行时有意外的行为。由于编译指示的影响域比较大，这些问题可能会很显著。另外，系统维护可能需要修改某些系统范围的配置。如果这些配置散落在系统中，将很难找出需要修改的地方。

因此，推荐尽可能的把所有编译指示放在一个编译单元中，以便于查找和修改。如果这个编译单元不大可能被重用 (例如：主子程序)，那么和将来重用时的冲突随之减少。最后，如果系统范围的指示可以不必内嵌在代码中，如通过编译器参数，那以上所述问题发生的可能性就更少了。

异例

某些编译指示 (例如：pragma Suppress) 有不同形式的使用，包括作为一个编译指示。本准则不适用于那些不作为指示时的使用。

4.1.3 子程序

准则

- 用子程序来增加抽象度。
- 限制每个子程序只执行单一动作。

范例

作为一个菜单推动的用户界面包中的一部分，要求画出一个菜单的用户选项。菜单的内容会根据用户状态而改变，此时用一个子程序来画比较恰当。这样，输出的子程序只有一个目的，而决定菜单内容的逻辑在别的地方。

```
...
-------------------------------------------------------------------------------
procedure Draw_Menu
      (Title   : in     String;
       Options : in     Menu) is
   ...
begin  -- Draw_Menu
   Ada.Text_IO.New_Page;
   Ada.Text_IO.New_Line;
   Ada.Text_IO.Set_Col (Right_Column);
```

```
   Ada.Text_IO.Put_Line (Title);
   Ada.Text_IO.New_Line;
   for Choice in Alpha_Numeric loop
     if Options (Choice) /= Empty_Line then
         Valid_Option (Choice) := True;
         Ada.Text_IO.Set_Col (Left_Column);
         Ada.Text_IO.Put (Choice & " -- ");
         Ada.Text_IO.Put_Line (Options (Choice));
     end if;
     ...
   end loop;
end Draw_Menu;
```

原理

子程序是非常高效和已知的抽象技巧。子程序通过隐藏某个活动的细节，提高了程序的可读性。子程序并不是需要被调用一次以上才能存在。

注解

准则10.1.1 探讨子程序调用的额外开销。

4.1.4 函数

准则

- 当子程序的主要目的是提供单一值时，用函数。
- 最小化函数的副作用。
- 当值不需要是静态时，考虑用无参数函数。
- 当导出类型需要继承基类型的值时，用无参数函数 (不要用常数)。
- 如果值本身有可能被改变，用无参数函数。

范例

虽然从一个文件中读取一个字符会改变下一个将要读取的字符，和下面函数的主要目的比起来，这是可接受的副作用:

```
function Next_Character return Character is separate;
```

然而，如此使用函数可能会导致隐蔽的问题。任何时候，当计算顺序未定，那函数返回值的顺序就未定。在以下这个例子中，字符放入 Word 中的顺序和接下来的两个给 Suffix 参数的字符顺序未定。没有一个 Next_Character 函数的实现可以保证那个字符赋值给谁:

```
   Word : constant String := String'(1 .. 5 => Next_Character);
begin  -- Start_Parsing
   Parse(Keyword => Word,
         Suffix1 => Next_Character,
         Suffix2 => Next_Character);
end Start_Parsing;
```

当然，如果顺序并不重要 (如随机数生成器)，那计算顺序也不重要。

下面的例子展现了用无参数函数代替常量:

```
type T is private;
function Nil return T;        -- This function is a derivable
function Default return T;    -- operation of type T. Also derivable,
                              -- and the value can be changed by
                              -- recompiling the body of the function
```

也可以用常量来写:

```
type T is private;
Nil : constant T;
Default : constant T;
```

原理

改变任何非子程序内的变量都是副作用。这包括函数本身调用别的子程序或入口时改变的变量在返回时改变被保持了。副作用很难被理解和维护，因此要反对。另外，Ada 语言没有定义表达式或作为子程序参数的函数计算顺序，因此，依赖于函数副作用计算顺序的的程序是错误的。避免在任何地方使用副作用。

4.1.5 包

准则

- 用包来实现信息隐藏。
- 用含有标签类型和私有类型的包来实现抽象数据类型。
- 用包把相关的类型和对象定义组在一起 (例如，两个或更多库单元的相同声明部分)。
- 将机器依赖封装在包中。将某个设备的软件接口放在一个包中，以便更换到另一种设备。
- 将底层实现的决策或接口放在包中的子程序。
- 用包和子程序来封装和隐藏可能改变的的程序细节 ([Nissen 和 Wallis 1984])。

范例

读取文件名和外部文件的其他属性非常依赖于系统。一个名为 Directory 的包，可包含了支持对外部目录及其文件的通用操作的类型和子程序。他的内部则依次依赖于其他更加针对硬件或操作系统的包:

```
package Directory is

   type Directory_Listing is limited private;

   procedure Read_Current_Directory (D : in out Directory_Listing);

   generic
      with procedure Process (Filename : in String);
   procedure Iterate (Over : in Directory_Listing);

   ...

private

   type Directory_Listing is ...

end Directory;
```

```
package body Directory is

   -- This procedure is machine dependent
   procedure Read_Current_Directory (D : in out Directory_Listing) is separate;
```

```
   procedure Iterate (Over : in Directory_Listing) is
      ...
   begin
      ...

      Process (Filename);

      ...
   end Iterate;

   ...

end Directory;
```

原理

包是 Ada 中主要的构造设施。可直接支持抽象、信息隐藏和模块化。例如，为了辅助可移植性，而把系统依赖部分封装的时候很有用。一个规约可有多个主体，以隔离特定实现的信息，这样其的代码就不需要改变了。

把可能会有变化的地方封装起来，去掉了系统无关部分对其的不必要依赖，从而最小化了需要实现改变的精力。

注解

对这个准则最大反对声通常和性能受损有关。参见准则 10.1.1 有关子程序开销的讨论。

4.1.6 子库单元

准则

- 如果一个新库单元是某个初始抽象的逻辑上的扩展，那定义为子库单元。
- 如果一个库单元是独立的 (例如，导入新的抽象，只部分依赖于现存抽象)，那么封装在一个分离的库单元中。
- 用子包来实现子系统。
- 用公共子单元来实现那些子系统中客户可见部分。
- 用私有子单元来实现那些子系统中客户不可见部分。
- 只有在实现包规约时，用私有子单元来实现局部声明。
- 用子包来实现构造器，即使他们的返回值是访问类型。

范例

下面这个有关视窗系统的例子，是从 [Cohen 等 1993] 中抽取出来的，他说明了在设计子系统时子单元的使用。父 (根) 包声明了客户和子系统需要的类型，子类型和常量。每个子包则提供了视窗系统中某个特定的抽象部分，例如原子、字体、图形输出、光标和键盘信息:

```
package X_Windows is
   ...
private
   ...
end X_Windows;


package X_Windows.Atoms is
   type Atom is private;
   ...
```

```
private
   ...
end X_Windows.Atoms;


package X_Windows.Fonts is
   type Font is private;
   ...
private
   ...
end X_Windows.Fonts;


package X_Windows.Graphic_Output is
   type Graphic_Context is private;
   type Image is private;
   ...
private
   ...
end X_Windows.Graphic_Output;


package X_Windows.Cursors is
   ...
end X_Windows.Cursors;


package X_Windows.Keyboard is
   ...
end X_Windows.Keyboard;
```

原理

用户在需要时用子库单元来扩展接口，能创造出更严谨和不杂乱的接口。父包只有相关的功能。父包提供了通用的接口，而子单元在提供更完整的编程接口，是对他们扩展或定义的抽象量身定做的。

子包是建立在 Ada 模块化的优势上的，“分开的规约和主体，分隔了用户对包的使用接口 (规约) 和包的实现 (主体) ” ([原理 1995], §II.7)。子包增加了既能扩展一个父包又不需要重编译父包或父包的客户的能力。

逻辑上不同的包可通过子包共享私有类型。可见度规则允许子规约的私有部分和子主体可见父包的私有部分。因此，可避免仅仅为了开发抽象需要共享一部分私有类型及其表征而创建一个整体的包。包的客户看不到私有的表征，因此其子包和本身的抽象得以维持。

私有子包可以用作局部声明，这让你在需要实现父包及其扩展包时，拥有支持性的声明。使用一套通用的支持声明 (数据结构、数据操作子程序)，提高了程序的可维护性。你可以修改内部的表征和支持子程序的实现而不需要修改或重编译其他子系统，因为这些支持性的子程序在私有子包的主体中实现。参见准则 4.1.1、**??**、8.1.1 和 8.1.2。

另参见准则9.3.1 有关用子库单元来生成标签类型分层结构的讨论。

4.1.7 内聚性

准则

- 让每个包只用于一个目的。
- 用包把相关的数据、类型和子程序组在一起。
- 避免收集没有关系的对象和子程序 ([NASA 1987], [Nissen 和 Wallis 1984])。

- 考虑系统的重构，把两个紧密相关的单元合并到一个包或包的分层结构中，又或者把相对独立的单元分到不同的包里。

范例

这是一个反面的例子，一个名为 Project_Definitions 的包，明显是某个项目中包罗所有的包，也可能很混乱。这可能是为了让项目成员在他们的程序中只用一个 with 语句而形成的。

好的例子，名为 Display_Format_Definitions 的包，包含了针对某个特定显示的特定格式的所有类型和常量，名为 Cartridge_Tape_Handler 的包，包含了针对某特殊用途设备的所有类型、常量和子程序。

原理

包内实体间关系的深浅程度，对于包及其组成的程序是否易于理解有直接的影响。组合的条件有很多，某些条件没有另一些条件有效。按照类的数据、活动 (例如初始化模块) 的时间特性来分组，就没有按他们的功能或数据交流的需求分来得有效 (参加 [Charette 1986])。

“正确”的系统结构对于系统的维护性有巨大的影响。如果初始的结构不是很理想，结构重组是很重要的，虽然当时会比较痛苦。

有关异构数据参见5.2.1。

注解

传统的子例程库通常把不同功能的子例程组在一起，即使这些库应该分成包的组合，每个包在包含一套逻辑上内聚的子例程。

4.1.8 数据耦合

准则

- 避免在包规范中定义变量。

范例

这个例子是编译器中的一部分。处理错误信息的包和含有代码生成器的包都需要知道当前的行数。这里并没有定义一个 Natural 的变量来分享这一信息，而是把具体实现隐藏在包中，并提供了一个访问函数:

```
package Compilation_Status is
   type Line_Number is range 1 .. 2_500_000;
   function Source_Line_Number return Line_Number;
end Compilation_Status;
```

```
with Compilation_Status;
package Error_Message_Processing is
   -- Handle compile-time diagnostic.
end Error_Message_Processing;
```

```
with Compilation_Status;

package Code_Generation is
   -- Operations for code generation.
end Code_Generation;
```

原理

紧密耦合的程序单元非常难调试和维护。用访问函数来保护共享的数据，耦合度就减少了。这避免了对于数据结构的依赖，也控制了数据访问。

注解

最普遍的反对声音通常都牵涉到性能损失。当把一个变量搬到包实体中，就必须提供访问子程序，每次调用都是额外开销。有关子程序调用的额外开销讨论，参见准则10.1.1。

4.1.9 任务

准则

- 在问题域中用任务来对抽象、异步的实体建模。
- 用任务定义多处理器架构中的并行算法。
- 用任务执行并行、循环或排序的事件 ([NASA 1987])。

原理

这些准则的原理在准则6.1.2中。第6章会对任务进行更深入的讨论。

4.1.10 保护类型

准则

- 用保护类型来控制或同步对数据或设备的访问。
- 用保护类型来实现任务间的同步，如被动的资源监视。

范例

参见准则6.1.1。

原理

这些准则的原理在准则6.1.1中。第6章对并行性和保护类型有更深入的讨论。

4.2 可见度

Ada 的可见度控制功能是 Ada 语言最重要的优势之一，特别是当“分开开发一个大型系统的各部分”时。她强制信息的隐藏和分离利害关系的能力就来源于此。假如没有这些功能，而过度依赖于 `use` 语句，是很浪费而危险的。参见准则5.3 和 9.3.1。

4.2.1 接口的最小化

准则

- 只把使用包需要的信息放在规范中。
- 最小化包规范中的声明数量。
- 不要因为容易建立而包含额外的操作。
- 在包规范中，最小化语境语句 (`with`)。

- 重新考虑有大量参数的子程序。
- 不要为了限制参数的数量而在子程序或包中操纵全局数据。
- 避免不必要的可见度，对用户隐藏程序的具体实现。
- 用子库单元来控制子系统接口的可见度。
- 用私有子库单元来声明外部用不到的信息。
- 用子库单元来对不同客户呈现某一实体的不同视点。
- 在找出各样预期客户的接口逻辑后，设计 (及重设计) 接口。

范例

```
package Telephone_Book is
   type Listing is limited private;
   procedure Set_Name (New_Name : in      String;
                       Current  : in out Listing);
   procedure Insert (Name    : in      String;
                     Current : in out Listing);
   procedure Delete (Obsolete : in      String;
                     Current  : in out Listing);
private
   type Information;
   type Listing is access Information;
end Telephone_Book;
```

```
package body Telephone_Book is
   -- Full details of record for a listing
   type Information is
      record
         ...
         Next : Listing;
      end record;
   First : Listing;
   procedure Set_Name (New_Name : in      String;
                       Current  : in out Listing) is separate;
   procedure Insert (Name    : in      String;
                     Current : in out  Listing) is separate;
   procedure Delete (Obsolete : in      String;
                     Current  : in out Listing) is separate;
end Telephone_Book;
```

原理

规范中的每一个实体，都要认真考虑其转移到子包或父包实体中的可能。越少的冗余细节，程序、包或子程序就越容易理解。对于维护者而言，知道准确的包接口很重要，以便于理解改动的效果。子程序的接口不仅限于参数。任何在包或子程序中修改全局变量的行为，都可被外界视为未公开的接口。

4.3 总结

高级结构

- 把每一个库单元的规约放在独立于实体的文件中。
- 避免定义不是主程序的库单元子程序。如果定义了这样的子程序，对每一个库单元子程序创建一个明确的单独的规约文件。

- 把亚单元的使用降到最低。
- 对于亚单元而言，用子库单元来把一个子系统构建为可管理的多个单元。
- 每一个亚单元存放在独立的文件中。
- 使用一致的文件命名约定。
- 对于包主体的嵌套，用私有子包并在父包中引用。
- 对于 (其他) 子单元用于扩展父单元的抽象或服务需要的数据和子程序，用私有子单元规约。
- 尽可能通过编译器参数或其他不需要修改代码的形式来配置编译指示。
- 当配置编译指示必须放在代码中时，考虑在每一个分区中把他们隔离在一个编译单元里。如果要指定，推荐用分区的主子程序。
- 用子程序来增加抽象度。
- 限制每个子程序只执行单一动作。
- 当子程序的主要目的是提供单一值时，用函数。
- 最小化函数的副作用。
- 当值不需要是静态时，考虑用无参数函数。
- 当导出类型需要继承基类型的值时，用无参数函数 (不要用常数)。
- 如果值本身有可能被改变，用无参数函数。
- 用包来实现信息隐藏。
- 用含有标签类型和私有类型的包来实现抽象数据类型。
- 用包把相关的类型和对象定义组在一起 (例如，两个或更多库单元的相同声明部分)。
- 将机器依赖封装在包中。将某个设备的软件接口放在一个包中，以便更换到另一种设备。
- 将底层实现的决策或接口放在包中的子程序。
- 用包和子程序来封装和隐藏可能改变的的程序细节 ([Nissen 和 Wallis 1984])。
- 让每个包只用于一个目的。
- 用包把相关的数据、类型和子程序组在一起。
- 避免收集没有关系的对象和子程序 ([NASA 1987], [Nissen 和 Wallis 1984])。
- 考虑系统的重构，把两个紧密相关的单元合并到一个包或包的分层结构中，又或者把相对独立的单元分到不同的包里。
- 避免在包规范中定义变量。
- 在问题域中用任务来对抽象、异步的实体建模。
- 用任务定义多处理器架构中的并行算法。
- 用任务执行并行、循环或排序的事件 ([NASA 1987])。
- 用保护类型来控制或同步对数据或设备的访问。
- 用保护类型来实现任务间的同步，如被动的资源监视。

可见度

- 只把使用包需要的信息放在规范中。
- 最小化包规范中的声明数量。
- 不要因为容易建立而包含额外的操作。
- 在包规范中，最小化语境语句 (`with`)。
- 重新考虑有大量参数的子程序。
- 不要为了限制参数的数量而在子程序或包中操纵全局数据。

- 避免不必要的可见度，对用户隐藏程序的具体实现。
- 用子库单元来控制子系统接口的可见度。
- 用私有子库单元来声明外部用不到的信息。
- 用子库单元来对不同客户呈现某一实体的不同视点。
- 在找出各样预期客户的接口逻辑后，设计 (及重设计) 接口。

第 五 章

编程实践

第 六 章

并行性

6.1 并行的选项

6.1.1 被保护对象

6.1.2 任务

第 七 章

可移植性

7.1 基础

7.1.1 封装实现依赖

7.2 数字类型和表达式

7.2.1 预定义数字类型

第 八 章

复用性

8.1 独立性

8.1.1 子系统设计

8.1.2 标签类型分层结构

第 九 章

面向对象的特性

第 十 章

提高性能

10.1 编译指示

10.1.1 内联 (Pragma Inline)

文献

[ARM 1983] 《Reference Manual for the Ada Programming Language》. Department of Defense, ANSI/MIL-STD-1815A. 1983

[ARM 1995] 《Ada 95 Reference Manual》, ISO/8652-1995 Cambridge, Massachusetts: Intermetrics, Inc. 1995

[Barnes 1989] Barnes, J.G.P.《Programming in Ada》, 3d ed. Reading, Massachusetts: Addison-Wesley. 1989.

[Barnes 1996] Barnes, J.G.P.《Programming in Ada 95》, Reading, Massachusetts: Addison-Wesley. 1996.

[Booch 1987] Booch, G. 《Software Componets With Ada-Structures, Tools and Subsystems》, Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc.

[Booch 1994] Booch, G.《Object-Oriented Analysis and Design》, 2d ed. Menlo Park, California: The Benjamin/Cummings Publishing Company, Inc. 1994.

[Charette 1986] Charette, R. N.《Software Engineering Environments Concepts and Technology》. Intertext Publications Inc. New York, New York: McGraw-Hill Inc. 1986

[Cohen 1986] Cohen, N.H.,《Ada as a Second Language》. New York, New York: McGraw-Hill Inc. 1986

[Cohen 等 1993] Cohen, N., M. Kamrad, E. Schonberg, 和 R. Dewar, 《Ada 9X as a Second Ada》, TRI-Ada '93 Tutorial Proceedings, pp. 1115-1196. 1993.

[Cohen 1996] Cohen, N.H.,《Ada as a Second Language》, 2nd edition, New York, New York: McGraw-Hill Inc. 1996.

[Hefley, 等人, 1992] Hefley, W., J. Foreman, C. Engle, and J. Goodenough, 《Ada Adoption Handbook: A Program Manager's Guide》 http://archive.adaic.com/docs/adopt-hbk/95adopt/html/toc.html, version 2.0. CMU/SEI-92-TR-29. Pittsburgh, Pennsylvania: Software Engineering Institute. 1992.

[Intermetrics 1995] 《Changes to Ada-1987 to 1995》, version 6.0. ISO/IEC 8652:1995(E), 1995.

[Jacobson, I. 等人 1992] 《Object-Oriented Software Engineering》. Reading, Massachusetts: Addison-Wesley. 1992.

[Nissen 和 Wallis 1984] 《Portability and Style in Ada》. Cambridge University Press.

[NASA 1987] 《Ada Style Guide》, Version 1.1, SEL-87-002. Greenbelt, Maryland: NASA, Goddard Space Flight Center.

[Schneiderman 1986] Empirical Studies of Programmers: The Territory, Paths and Destinations. Empirical Studies of Programmers, edited by E. Soloway and S. Iyengar. Norwood, NJ: Ablex Publishing Corp. pp. 1-12.

[Soloway et al. 1986] Soloway, E., J. Pinto, S. Fertig, S. Letovsky, R. Lampert, D. Littman, and K. Ewing, Studying Software Documentation From a Cognitive Perspective: A Status Report. Proceedings of the Eleventh Annual Software Engineering Workshop. Report SEL-86-006, Software Engineering Laboratory. Greenbelt, Maryland: NASA, Goddard Space Flight Center.

[Taylor 1995] Taylorm, B.《Ada Compatibility Guide》, version 6.0. United Kingdom: Transition Technology Limited. 1995.

[软件生产力协会 1992] 《Ada Quality and Style: Guidelines for Professional Programmers》, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium. 1992.

[软件生产力协会 1993] 《ADARTS Guidebook》, SPC-91104-MC, version 03.00.09. 2 vols. Herndon, Virginia: Software Productivity Consortium. 1993.

[Rosen 1995] 《A Naming Convention for Classes in Ada 9X》, ACM Ada Letters. VXV(2):54-58.

[Rumbaugh, 等人 1991] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen,《Object-Oriented Modeling and Design》。Englewood Cliffs, New Jersey: Prentice-Hall. 1991

[原理 1995] 《Ada 95 Rationale》, Cambridge, Massachusetts: Intermetrics, Inc. 1995

[United Technologies 1987] CENC Programmer's Guide. Appendix A: Ada Programming Standards。