第①章

新标准的诞生

从最初的代号 C++0x 到最终的名称 C++11，C++ 的第二个真正意义上的标准姗姗来迟。可以想象，这个迟来的标准必定遭遇了许多的困难，而 C++ 标准委员会应对这些困难的种种策略，则构成新的 C++ 语言基因，我们可以从新的 C++11 标准中逐一体会。而客观上，这些基因也决定了 C++11 新特性的应用范畴。在本章中，我们会从设计思维和应用范畴两个维度对所有的 C++11 新特性进行分类，并依据这种分类对一些特性进行简单的介绍，从而一览 C++11 的全景。

1.1 曙光：C++11 标准的诞生

1.1.1 C++11/C++0x（以及 C11/C1x）——新标准诞生

2011 年 11 月，在印第安纳州布卢明顿市，“八月印第安纳大学会议”（August Indiana University Meeting）缓缓落下帷幕。这次会议的结束，意味着长久以来以 C++0x 为代号的 C++11 标准终于被 C++ 标准委员会批准通过。至此，C++ 新标准尘埃落定。从 C++98 标准通过的时间开始计算，C++ 标准委员会，即 WG21，已经为新标准工作了 11 年多的时间。对于一个编程语言标准而言，11 年显然是个非常长的时间。其间我们目睹了面向对象编程的盛极，也见证了泛型编程的风起云涌，还见证了 C++ 后各种新的流行编程语言的诞生。不过在新世纪第二个 10 年的伊始，C++ 的标准终于二次来袭。

事实上，在 2003 年 WG21 曾经提交了一份技术勘误表（Technical Corrigendum，简称 TC1）。这次修订使得 C++03 这个名字已经取代了 C++98 成为 C++11 之前的最新 C++ 标准名称。不过由于 TC1 主要是对 C++98 标准中的漏洞进行修复，核心语言规则部分则没有改动，因此，人们还是习惯地把两个标准合称为 C++98/03 标准。

注意 在本书中，但凡是 C++98 和 C++03 标准没有差异时，我们都会沿用 C++98/03 这样的俗称，或者直接简写为 C++98。如果涉及 TC1 中所提出的微小区别，我们会使用 C++98 和 C++03 来分别指代两种 C++ 标准。

C++11 是一种新语言的开端。虽然设计 C++11 的目的是为了要取代 C++98/03，不过相比于 C++03 标准，C++11 则带来了数量可观的变化，这包括了约 140 个新特性，以及对

C++03 标准中约 600 个缺陷的修正。因此，从这个角度看来 C++11 更像是从 C++98/03 中孕育出的一种新语言。正如当年 C++98/03 为 C++ 引入了如异常处理、模板等许多让人耳目一新的新特性一样，C++11 也通过大量新特性的引入，让 C++ 的面貌焕然一新。这些全新的特性以及相应的全新的概念，都是我们要在本书中详细描述的。

1.1.2 什么是 C++11/C++0x

C++0x 是 WG21 计划取代 C++98/03 的新标准代号。这个代号还是在 2003 年的时候取的。当时委员会乐观地估计，新标准会在 21 世纪的第一个 10 年内完成。从当时看毕竟还有 6 年的时间，确实无论如何也该好了。不过 2010 新年钟声敲响的时候，WG21 内部却还在为一些诸如哪些特性该放弃，哪些特性该被削减的议题而争论。于是所有人只好接受这个令人沮丧的事实：新标准没能准时发布。好在委员会成员保持着乐观的情绪，还常常相互开玩笑说，x 不是一个 0 到 9 的十进制数，而应该是一个十六进制数，我们还可以有 A、B、C、D、E、F。虽然这是个玩笑，但也有点认真的意思，如果需要，WG21 会再使用“额外”的 6 年，在 2015 年之前完成标准。不过众所周知的，WG21“只”再花了两年时间就完成了 C++11 标准。

注意 C 语言标准委员会（C committee）WG14 也几乎在同时开始致力于取代 C99 标准。不过相比于 WG21，WG14 对标准完成的预期更加现实。因为他们使用的代号是 C1x，这样新的 C 标准完成的最后期限将是 2019 年。事实上 WG14 并没用那么长时间，他们最终在 2011 年通过了提案，也就是 C11 标准。

从表 1-1 中可以看到 C++ 从诞生到最新通过的 C++11 标准的编年史。

表 1-1 C++ 发展编年史

日 期	事 件
1990 年	*The Annotated C++ Reference Manual*，M.A.Ellis 和 B.Stroustrup 著。主要描述了 C++ 核心语言，没有涉及库
1998 年	第一个国际化的 C++ 语言标准：IOS/IEC 15882:1998。包括了对核心语言及 STL、locale、iostream、numeric、string 等诸多特性的描述
2003 年	第二个国际化的 C++ 语言标准：IOS/IEC 15882:2003。核心语言及库与 C++98 保持了一致，但包含了 TC1（Technical Corrigendum 1, 技术勘误表 1）。自此，C++03 取代了 C++98
2005 年	TR1（Technical Report 1，技术报告 1）：IOS/IEC TR 19768:2005。核心语言不变。TR1 作为标准的非规范出版物，其包含了 14 个可能进入新标准的新程序库
2007 年 9 月	SC22 注册（特性）表决。通过了 C++0x 中核心特性
2008 年 9 月	SC22 委员会草案（Committee Draft, CD）表决。基本上所有 C++0x 的核心特性都完成了，新的 C++0x 标准草稿包括了 13 个源自 TR1 的库及 70 个库特性，修正了约 300 个库缺陷。此外，新标准草案还包括了 70 多个语言特性及约 300 个语言缺陷的修正
2010 年 3 月	SC22 最终委员会草案（Final Committee Draft, FCD）表决。所有核心特性都已经完成，处理了各国代表的评议

（续）

日　期	事　件
2011 年 11 月	JTC1 C++11 最终国际化标准草案（Final Draft International Standard, FDIS）发布，即 IOS/IEC 15882:2011。新标准在核心语言部分和标准库部分都进行了很大的改进，这包括 TR1 的大部分内容。但整体的改进还是与先前的 C++ 标准兼容的
2012 年 2 月	在 ANSI 和 ISO 商店可以以低于原定价的价格买到 C++11 标准

注意 语言标准的发布通常有两种——规范的（Normative）及不规范的（Non-normative）。前者表示内容通过了批准（ratified），因此是正式的标准，而后者则不是。不过不规范的发布通常是有积极意义的，比方说 TR1，它就是不规范的标准，但是后来很多 TR1 的内容都成为了 C++11 标准的一部分。

图 1-1 比较了两个语言标准委员会（WG21，WG14）制定新标准的工作进程，其中一些重要时间点都标注了出来。

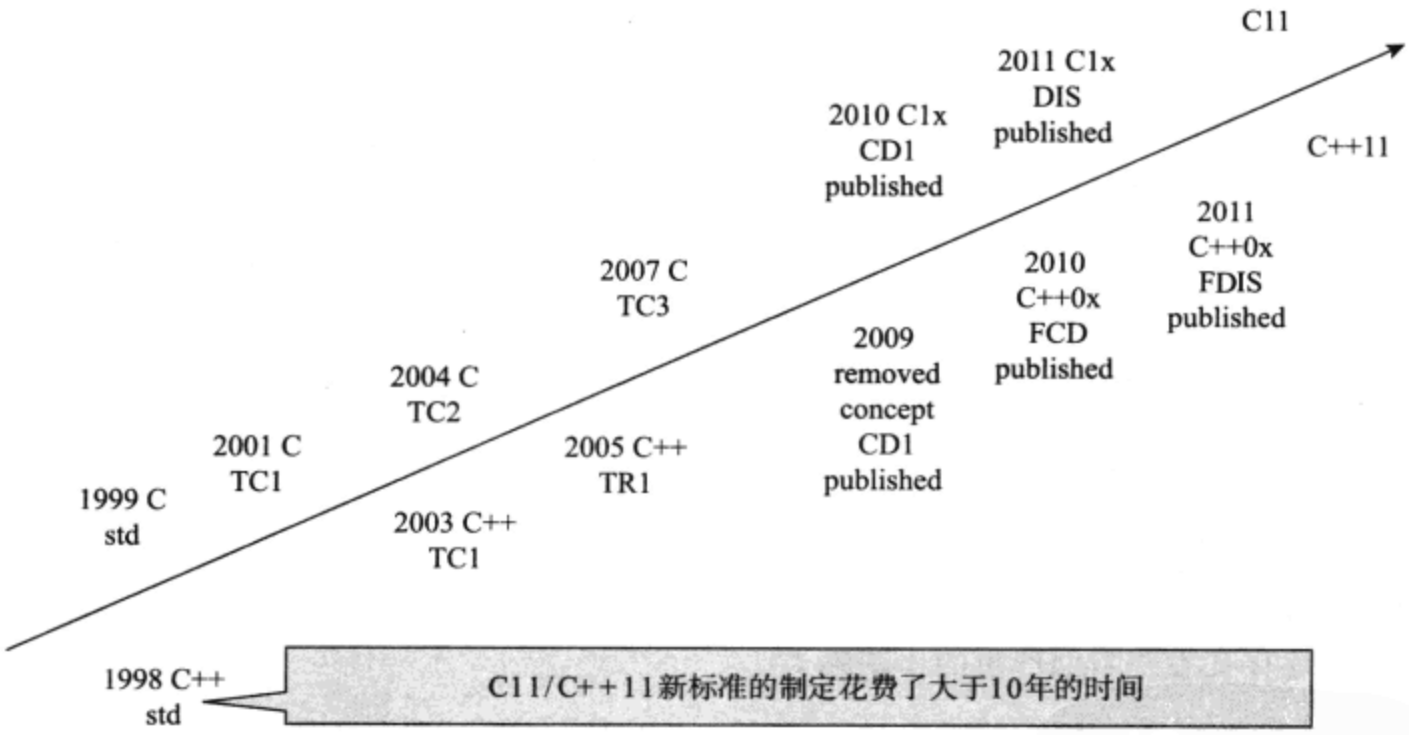

图 1-1　WG21 和 WG14 制定新语言标准的工作进程

1.1.3　新 C++ 语言的设计目标

如果读者已经学习过 C++98/03，就可以发现 C++98/03 的设计目标如下：

- 比 C 语言更适合系统编程（且与 C 语言兼容）。
- 支持数据抽象。
- 支持面向对象编程。
- 支持泛型编程。

这些特点使得面向对象编程和泛型编程在过去的 10 ~ 20 年内成为编程界的明星。不过从那时开始，C++ 的发展就不仅仅是靠学者的远见前瞻去推动的，有时也会借由一些“奇缘”而演进。比方说，C++ 模板就是这样一个“奇缘”。它使得 C++ 近乎成为了一种函数式编程语言，而且还使得 C++ 程序员拥有了模板元编程的能力。但是凡事有两面，C++98/03 中的一些较为激进的特性，比如说动态异常处理、输出模板，现在回顾起来则是不太需要的。当然，这是由于我们有了“后见之明”，或者由于这些特性在新情况下不再适用，又或者它们影响了 C++11 的新特性的设计。因此一部分这样的特性已经被 C++11 弃用了。在附录 B 中我们会一一列出这些弃用的特性，并分析其被弃用的原因。

而 C++11 的整体设计目标如下：

- 使得 C++ 成为更好的适用于系统开发及库开发的语言。
- 使得 C++ 成为更易于教学的语言（语法更加一致化和简单化）。
- 保证语言的稳定性，以及和 C++03 及 C 语言的兼容性。

我们可以分别解释一下。

首先，使 C++ 成为更好的适用于系统开发及库开发的语言，意味着标准并不只是注重为某些特定的领域提供专业化功能，比如专门为 Windows 开发提供设计，或者专门为数值计算提供设计。标准希望的是使 C++ 能够对各种系统的编程都作出贡献。

其次，使得 C++ 更易于教学，则意味着 C++11 修复了许多令程序员不安的语言“毒瘤”。这样一来，C++ 语法显得更加一致化，新手使用起来也更容易上手，而且有了更好的语法保障。其实语言复杂也有复杂的好处，比如 ROOTS、DEALII 等一些复杂科学运算的算法，它们的作者非常喜爱泛型编程带来的灵活性，于是 C++ 语言最复杂的部分正好满足了他们的需求。但是在这个世界上，新手总是远多于专家。而即使是专家，也常常只是精通自己的领域。因此语言不应该复杂到影响人们的学习。本书作者之一也是 WG21 中的一员，从结果上看，无论读者怎么看待 C++11，委员会大多数人都认同 C++11 达成了易于教学这个目标（即使其中还存在着些看似严重的小缺陷）。

最后，则是语言的稳定性。经验告诉我们，伟大的编程语言能够长期存活下来的原因还是因为语言的设计突出了实用性。事实上，在标准制定过程中，委员会承担了很多压力，这些压力源自于大家对加入更多语言特性的期盼——每一个人都希望将其他编程语言中自己喜欢的特性加入到新的 C++ 中。对于这些热烈而有些许盲目的期盼，委员会成员在 Bjarne Stroustrup 教授的引导下，选择了不断将许多无关的特性排除在外。其目的是防止 C++ 成为一个千头万绪的但功能互不关联的语言。而如同现在看到的那样，C++11 并非大而无序，相反地，许多特性可以良好地协作，进而达到“1 + 1 > 2”的效果。可以说，有了这些努力，今天的读者才能够使用稳定而强大的 C++11，而不用担心语言本身存在着混乱状况甚至是冲突。

值得一提的是，虽然在取舍新语言特性方面标准委员会曾面临过巨大压力，但与此同

时，标准委员会却没有收集到足够丰富的库的新特性。作为一种通用型语言，C++ 是否是成功，通常会依赖于不同领域中 C++ 的使用情况，比如科学计算、游戏、制造业、网络编程等。在 C++11 通过的标准库中，服务于各个领域的新特性确实还是太少了。因此很有可能在下一个版本的 C++ 标准制定中，如何标准化地使用库将成为热门话题，标准委员会也准备好了接受来自这方面的压力。

1.2 今时今日的 C++

1.2.1 C++ 的江湖地位

如今 C++ 依旧位列通用编程语言三甲，不过似乎没有以前那么流行了。事实上，编程语言排名通常非常难以衡量。比如，某位教授或学生用了 C++ 来教授课程应该被计算在内吗？在新的联合攻击战斗机（Joint Strike Fighter，JSF-35）的航空电子设备中使用了 C++ 编程应该计算在内吗？又或者 C++ 被用于一款流行的智能手机操作系统的编程中算不算呢？再或者是 C++ 被用于编写最流行的在线付费搜索引擎，或用于构建一款热门的第一人称射击游戏的引擎，或用于构建最热门的社交网络的代码库，这些都该计算在内吗？事实上，据我们所知，以上种种都使用了 C++ 编程。而且在构建致力于沟通软硬件的系统编程中，C++ 也常常是必不可少的。甚至，C++ 还常用于设计和编写编程语言。因此我们可以认为，编程语言价值的衡量标准应该包括数量、新颖性、质量，以及以上种种，都应该纳入“考核”。这样一来，结论就很明显了：C++ 无处不在。

1.2.2 C++11 语言变化的领域

如果谁说 C++11 只是对 C++ 语言做了大幅度的改进，那么他很可能就错过了 C++11 精彩的地方。事实上，读罢本书后，读者只需要看一眼代码，就可以说出代码究竟是 C++98/03 的，还是 C++11 的。C++11 为程序员创造了很多更有效、更便捷的代码编写方式，程序员可以用简短的代码来完成 C++98/03 中同样的功能，简单到你惊呼“天哪，怎么能这么简单”。从一些简单的数据统计上看，比起 C++98/03，C++11 大大缩短了代码编写量，依情况最多可以将代码缩短 30% ~ 80%。

那么 C++11 相对于 C++98/03 有哪些显著的增强呢？事实上，这包括以下几点：

- ❑通过内存模型、线程、原子操作等来支持本地并行编程（Native Concurrency）。
- ❑通过统一初始化表达式、auto、declytype、移动语义等来统一对泛型编程的支持。
- ❑通过 constexpr、POD（概念）等更好地支持系统编程。
- ❑通过内联命名空间、继承构造函数和右值引用等，以更好地支持库的构建。

表 1-2 列出了 C++11 批准通过的，且本书将要涉及的语言特性。这是一张相当长的表，而且一个个陌生的词汇足以让新手不知所措。不过现在还不是了解它们的时候。但看过这张

表，读者至少会有这样一种感觉：C++11 的确像是一门新的语言。如果我们将 C++98/03 标准中的特性和 C++11 放到一起，C++11 则像是个恐怖的“编程语言范型联盟”。利用它不仅仅可以写出面向对象语言的代码，也可以写出过程式编程语言代码、泛型编程语言代码、函数式编程语言代码、元编程编程语言代码，或者其他。多范型的支持使得 C++11 语言的“硬能力”几乎在编程语言中“无出其右”。

表 1-2 C++11 主要的新语言特性（中英文对照）

中文翻译	英文名称	备注
__cplusplus 宏	__cplusplus macro	
对齐支持	alignment support	
通用属性	general attribute	
原子操作	atomic operation	
auto 类型推导（初始化类型推导）	auto (type deduction from 18nitialize)	
C99 特性	C99	
强类型枚举	enum class (scoped and strongly typed enums)	
复制及再抛出异常	copy and rethrow exception	本书未讲解
常量表达式	constexpr	
decltype	decltype	
函数的默认模板参数	default template parameters for function	
显式默认和删除的函数（默认的控制）	defaulted and deleted functions (control of defaults)	
委托构造函数	delegating constructors	
并行动态初始化和析构	Dynamic Initialization and Destruction with Concurrency	本书未讲解
显式转换操作符	explicit conversion operators	
扩展的 friend 语法	extended friend syntax	
扩展的整型	extended integer types	
外部模板	extern templates	
一般化的 SFINAE 规则	generalized SFINAE rules	
统一的初始化语法和语义	Uniform initialization syntax and semantics	
非受限联合体	unrestricted union	
用户定义的字面量	user-defined literals	
变长模板	variadic templates	
类成员初始化	in-class member initializers	
继承构造函数	inherited constructors	
初始化列表	initializer lists	
lambda 函数	lambda	
局部类型用作模板参数	local classes as template arguments	
long long 整型	long long integers	
内存模型	memory model	
移动语义（参见右值引用）	move semantics（see rvalue references）	
内联名字空间	Inline namespace	

（续）

中文翻译	英文名称	备　注
防止类型收窄	Preventing narrowing	
指针空值	nullptr	
POD	POD（plain old data）	
基于范围的 for 语句	range-based for statement	
原生字符串字面量	raw string literals	
右值引用	rvalue reference	
静态断言	static assertions	
追踪返回类型语法	trailing return type syntax	
模板别名	template alias	
线程本地的存储	thread-local storage	
Unicode	Unicode	

而从另一个角度看，编程中程序员往往需要将实物、流程、概念等进行抽象描述。但通常情况下，程序员需要抽象出的不仅仅是对象，还有一些其他的概念，比如类型、类型的类型、算法，甚至是资源的生命周期，这些实际上都是 C++ 语言可以描述的。在 C++11 中，这些抽象概念常常被实现在库中，其使用将比在 C++98/03 中更加方便，更加好用。从这个角度上讲，C++11 则是一种所谓的“轻量级抽象编程语言”（Lightweight Abstraction Programming Language）。其好处就是程序员可以将程序设计的重点更多地放在设计、实现，以及各种抽象概念的运用上。

总的来说，灵活的静态类型、小的抽象概念、绝佳的时间与空间运行性能，以及与硬件紧密结合工作的能力都是 C++11 突出的亮点。而反观 C++98/03，其最强大的能力则可能是体现在能够构建软件基础架构，或构建资源受限及资源不受限的项目上。因此，C++11 也是 C++ 在编程语言领域上一次“泛化”与进步。

要实现表 1-2 中的各种特性，需要编译器完成大量的工作。对于大多数编译器供应商来说，只能分阶段地发布若干个编译版本，逐步支持所有特性（罗马从来就不是一天建成的，对吧）。大多数编译器已经开始了对 C++11 特性的支持。有 3 款编译器甚至从 2008 年前就开始支持 C++11 了：IBM 的 XL C/C++ 编译器从版本 10.1 开始。GNU 的 GCC 编译器从版本 4.3 开始，英特尔编译器从版本 10.1 开始。而微软则从 Visual Studio 2010 开始。最近，苹果的 clang/llvm 编译器也从 2010 年的版本 2.8 开始支持 C++11 新特性，并且急速追赶其他编译器供应商。在本书附录 C 中，读者可以找到现在情况下各种编译器对 C++11 的支持情况。

1.3　C++11 特性的分类

从设计目标上说，能够让各个特性协同工作是设计 C++11/0x 中最为关键的部分。委员会总希望通过特性协作取得整体大于个体的效果，但这也是语言设计过程中最困难的一点。

因此相比于其他的各种考虑，WG21 更专注于以下理念：

- ❑保持语言的稳定性和兼容性（Maintain stability and compatibility）。
- ❑更倾向于使用库而不是扩展语言来实现特性（Prefer libraries to language extensions）。
- ❑更倾向于通用的而不是特殊的手段来实现特性（Prefer generality to specialization）。
- ❑专家新手一概支持（Support both experts and novices）。
- ❑增强类型的安全性（Increase type safety）。
- ❑增强代码执行性能和操作硬件的能力（Improve performance and ability to work directly with hardware）。
- ❑开发能够改变人们思维方式的特性（Make only changes that change the way people think）。
- ❑融入编程现实（Fit into the real world）。

根据这些设计理念可以对新特性进行分类。在本书中，我们的核心章节（第 2 ~ 8 章）也会按照这样的方式进行划分。在可能的时候，我们也会为每个理念取个有趣一点儿的中文名字。

而从使用上，Scott Mayers 则为 C++11 创建了另外一种有效的分类方式，Mayers 根据 C++11 的使用者是类的使用者，还是库的使用者，或者特性是广泛使用的，还是库的增强的来区分各个特性。具体地，可以把特性分为以下几种：

- ❑类作者需要的（class writer，简称为“类作者”）
- ❑库作者需要的（library writer，简称为“库作者”）
- ❑所有人需要的（everyone，简称为“所有人”）
- ❑部分人需要的（everyone else，简称为“部分人”）

那么我们可以结合这种分类再来看一下可以怎样来学习所有的特性。下面我们通过设计理念和用户群对 C++11 特性进行分类，如表 1-3 所示。

由于 C++11 的新特性非常多，因此本书不准备涵盖所有内容。我们粗略地将特性划分为核心语言特性和库特性。而从 C++11 标准的章节划分来看（读者可以从网站上搜到接近于最终版本的草稿，正式的标准需要通过购买获得），本书将涉及 C++11 标准中第 1 ~ 16 章的语言特性部分（在 C++11 语言标准中，第 1 ~ 16 章涵盖了核心语言特性，第 17 ~ 30 章涉及库特性），而标准库将不在本书中描述。当然，这会导致许多灰色地带，因为如同我们提到的，我们总是倾向于使用库而不是语言扩展来实现一些特性，那么实际上，讲解语言核心特性也必然涉及库的内容。典型的，原子操作（atomics）就是这样一个例子。因此，在本书的编写中，我们只是不对标准库进行专门的讲解，而与核心内容相关的库内容，我们还是会有所描述的。

表 1-3 根据设计理念和用户群对 C++11 新特性进行划分

理　念	特性名称（中英文）	用户群
保持语言的稳定性和兼容性（Maintain stability and compatibility）	C99	部分人
	函数的默认模板参数 default template parameters for function	所有人
	扩展的 friend 语法 extended friend syntax	部分人
	扩展的整型 extended integer types	部分人
	外部模板 extern templates	部分人
	类成员的初始化 in-class member initializers	部分人
	局部类用作模板参数 local classes as template arguments	部分人
	long long 整型 long long integers	部分人
	__cplusplus	部分人
	noexcept	库作者
	override/final 控制 Override/final controls	部分人
	静态断言 static assertions	库作者
	类成员的 sizeof sizeof class data members	部分人
更倾向于通用的而不是特殊化的手段来实现特性（Prefer generality to specialization）	继承构造函数 Inherited constructor	类作者
	移动语义，完美转发，引用折叠 Move semantics, perfect forwarding, reference collapse	类作者
	委托构造函数 delegating constructors	类作者
	显式转换操作符 explicit conversion operators	库作者
	统一的初始化语法和语义，初始化列表，防止收窄 Uniform initialization syntax and semantics, initializer lists, Preventing narrowing	所有人
	非受限联合体 unrestricted unions (generalized)	部分人
	用户自定义字面量 UDL	部分人
	一般化 SFINAE 规则 generalized SFINAE rules	库作者
	内联名字空间 Inline Namespace	部分人
	PODs	部分人
	模板别名 template alias	所有人
专家新手一概支持（Support both experts and novices）	右尖括号 Right angle bracket	所有人
	auto	所有人
	基于范围的 for 语句 Ranged For	所有人
	decltype	库作者
	追踪返回类型语法（扩展的函数声明语法）Trailing return type syntax (extended function declaration syntax)	所有人
增强类型的安全性（Increase type safety）	强类型枚举 Strong enum	部分人
	unique_ptr, shared_ptr	类作者
	垃圾回收 ABI Garbage collection ABI	库作者

（续）

理　念	特性名称（中英文）	用户群
增强性能和操作硬件的能力（Improve performance and ability to work directly with hardware）	常量表达式 constexpr	类作者
	原子操作 / 内存模型 atomics/mm	所有人
	复制和再抛出异常 copying and rethrowing exceptions	所有人
	并行动态初始化和析构 Dynamic Initialization and Destruction with Concurrency	所有人
	变长模板 Variadic template	库作者
	线程本地的存储 thread-local storage	所有人
	快速退出进程 quick_exit Abandoning a process	所有人
开发能够改变人们思维方式的特性（Make only changes that change the way people think）	指针空值 nullptr	所有人
	显示默认和删除的函数（默认的控制）defaulted and deleted functions (control of defauls)	类作者
	lambdas	所有人
融入编程现实（Fit into the real world）	对齐支持 Alignments	部分人
	通用属性 Attributes [[carries dependency]] [[noreturn]]	部分人
	原生字符串字面量 raw string literals	所有人
	Unicode unicode characters	所有人

而之前我们提到过的“更倾向于使用库而不是扩展语言来实现特性”理念的部分，如果有可能，我们会在另一本书或者本书的下一个版本中来进行讲解。下面列出了属于该设计理念下的库特性：

- ❑算法增强 Algorithm improvements
- ❑容器增强 Container improvements
- ❑分配算符 Scoped allocators
- ❑std::array
- ❑std::forward_list
- ❑无序容器 Unordered containers
- ❑sts::tuple
- ❑类型特性 Type traits
- ❑std::function, std::bind
- ❑unique_ptr
- ❑shared_ptr
- ❑weak_ptr
- ❑线程 Threads
- ❑互斥 Mutex
- ❑锁 Locks

- ❑条件变量 Condition variables
- ❑时间工具 Time utilities
- ❑std::future, std::promises
- ❑std::async
- ❑随机数 Random numbers
- ❑正则表达式 regex

1.4 C++ 特性一览

接下来，我们会一窥 C++11 中的各种特性，了解它们的来历、用途、特色等。可能这部分对于还没有开始阅读正文的读者来说有些困难。如果有机会，我们建议读者在读完全书后再回到这里，这也是全书最好的总结。

1.4.1 稳定性与兼容性之间的抉择

通常在语言设计中，不破坏现有的用户代码和增加新的能力，这二者是需要同时兼顾的。就像之前的 C 一样，如今 C++ 在各种代码中、开源库中，或用户的硬盘中都拥有上亿行代码，那么当 C++ 标准委员会要改变一个关键字的意义，或者发明一个新的关键字时，原有代码就很可能发生问题。因为有些代码可能已经把要加入的这个准关键字用作了变量或函数的名字。

语言的设计者或许能够完全不考虑兼容性，但说实话这是个丑陋的做法，因为来自习惯的力量总是超乎人的想象。因此 C++11 只是在非常必要的情况下才引入新的关键字。WG21 在加入这些关键字的时候非常谨慎，至少从谷歌代码搜索（Google Code Search）的结果看来，这些关键字没有被现有的开源代码频繁地使用。不过谷歌代码搜索只会搜索开源代码，私人的或者企业的代码库（codebase）是不包含在内的。因此这些数据可能还有一定的局限性，不过至少这种方法可以避免一些问题。而 WG21 中也有很多企业代表，他们也会帮助 WG21 确定这些关键字是否会导致自己企业代码库中代码不兼容的问题。

C++11 的新关键字如下：

- ❑alignas
- ❑alignof decltype
- ❑auto（重新定义）
- ❑static_assert
- ❑using（重新定义）
- ❑noexcept
- ❑export（弃用，不过未来可能留作他用）
- ❑nullptr

- constexpr
- thread_local

这些新关键字都是相对于 C++98/03 来说的。当然，引入它们可能会破坏一些 C++98/03 代码，甚至更为糟糕的是，可能会悄悄地改变了原有 C++98/03 程序的目的。static_assert 就是这样一个例子。为了降低它与已有程序变量冲突的可能性，WG21 将这个关键字的名字设计得很长，甚至还包含了下划线，可以说命名丑得不能再丑了，不过在一些情况下，它还是会发生冲突，比如：

```
static_assert(4<=sizeof(int), "error:small ints");
```

这行代码的意图是确定编译时（不是运行时）系统的 int 整型的长度不小于 4 字节，如果小于，编译器则会报错说系统的整型太小了。在 C++11 中这是一段有效的代码，在 C++98/03 中也可能是有效的，因为程序员可能已经定义了一个名为 static_assert 的函数，以用于判断运行时的 int 整型大小是否不小于 4。显然这与 C++11 中的 static_assert 完全不同。

实际上，在 C++11 中还有两个具有特殊含义的新标识符：override、final。这两个标识符如何被编译器解释与它们所在的位置有关。如果这两个标识符出现在成员函数之后，它们的作用是标注一个成员函数是否可以被重载。不过读者实际上也可以在 C++11 代码中定义出 override、final 这样名称的变量。而在这样的情况下，它们只是标识了普通的变量名称而已。

我们主要会在第 2 章中看到相关的特性的描述。

1.4.2 更倾向于使用库而不是扩展语言来实现特性

相比于语言核心功能的稳定，库则总是能随时为程序员提供快速上手的、方便易用的新功能。库的能量是巨大的，Boost ⊖和一些公司私有的库（如 Qt、POOMA）的快速成长就说明了这一点。而且库有一个很大的优势，就是其改动不需要编译器实现新特性（只要接口保持一致即可），当然，更重要的是库可以用于支持不同领域的编程。这样一来，通常读者不需要非常精通 C++ 就能使用它们。

不过这些优点并不是被广泛认可的。狂热的语言爱好者总是觉得功能加入语言特性，由编译器实现了才是王道，而库只是第二选择。不过 WG21 的看法跟他们相反。事实上，如果可能，WG21 会尽量将一个语言特性转为库特性来实现。比较典型的如 C++11 中的线程，它被实现为库特性的一部分：std::thread，而不是一个内置的“线程类型”。同样的，C++11 中没有内置的关联数组（associative array）类型，而是将它们实现为如 std::unorder_map 这样的库。再者，C++11 也没有像其他语言一样在语言核心部分加入正则表达式功能，而是实现为

⊖ 在 C++ 的众多开源库，最为出名的应该是 Boost。Boost 是一个无限制的开源库，在设计和审阅的时候，都常常有 C++ 标准委员会的人参与。

std::regex 库。这样一来，C++ 语言可以尽量在保持较少的核心语言特性的同时，通过标准库扩大其功能。

从传统意义上讲，库可能是通过提供头文件来实现的。当然，有些时候库的提供者也会将一些实现隐藏在二进制代码库存档（archive）文件中。不过并非所有的库都是通过这样的方式提供的。事实上，库也有可能实现于编译器内部。比如 C++11 中的原子操作等许多内容，就通常不是在头文件或库存档中实现的。编译器会在内部就将原子操作实现为具体的机器指令，而无需在稍后去链接实实在在的库进行存档。而之所以将原子操作的内容放在库部分，也是为了满足将原子操作作为库实现的自由。从这个意义上讲，原子操作并非纯粹的“库”，因此也被我们选择性地纳入了本书的讲解中。

1.4.3 更倾向于通用的而不是特殊的手段来实现特性

如我们说到的，如果将无数互不相关的小特性加入 C++ 中，而且不加选择地批准通过，C++ 将成为一个令人眼花缭乱的“五金店”，不幸的是，这个五金店的产品虽然各有所长，凑在一起却是一盘散沙，缺乏战斗力。所以 WG21 更希望从中抽象出更为通用的手段而不是加入单独的特性来“练成” C++11 的“十八般武艺”。

显式类型转换操作符是一个很好的例子。在 C++98/03 中，可以用在构造函数前加上 explicit 关键字来声明构造函数为显式构造，从而防止程序员在代码中“不小心”将一些特定类型隐式地转换为用户自定义类型。不过构造函数并不是唯一会导致产生隐式类型转换的方法，在 C++98/03 中类型转换操作符也可以参与隐式转换，而程序员的意图则可能只是希望类型转换操作符在显式转换时发生。这是 C++98/03 的疏忽，不过在 C++11 中，我们已经可以做到这点了。

其他的一些新特性，比如继承构造函数、移动语义等，在本书的第 3 章中我们均会涉及。

1.4.4 专家新手一概支持

如果 C++ 只是适合专家的语言，那它就不可能是一门成功的语言。C++ 中虽然有许多专家级的特性，但这并不是必须学习的。通常程序员只需要学习一定的知识就可以使用 C++。而在 C++11 中，从易用的角度出发，修缮了很多特性，也铲除了许多带来坏声誉的“毒瘤”，比如一度被群起而攻之的“毒瘤”——双右尖括号。在 C++98/03 中，由于采用了最长匹配的解析规则（maximal munch parsing rule），编译器会在解析符号时尽可能多地“吸收”符号。这样一来，在模板解析的时候，编译器就会将原本是“模板的模板”识别为右移，并“理直气壮”地抛出一条令人绝望的错误信息：模板参数中不应该存在的右移。如今这个问题已经在 C++11 中被修正。模板参数内的两个右尖括号会终结模板参数，而不会导致编译器错误。当然从实现上讲，编译器只需要在原来报错的地方加入一些上下文的判断就可以避免

这样的错误了。比如：

```
vector<list<int>> veclist: //C++11 中有效，C++98/03 中无效
```

另一个 C++11 易于上手的例子则是统一初始化语法的引入。C++ 继承了 C 语言中所谓的“集合初始化语法”（aggregate initialization syntax，比如 a[] = {0, 1,};），而在设计类的时候，却只定义了形式单一的构造函数的初始化语法，比如 A a(0, 1)。所以在使用 C++98/03 的时候，编写模板会遇到障碍，因为模板作者无法知道模板用户会使用哪种类型来初始化模板。对于泛型编程来说，这种不一致则会导致不能总是进行泛型编程。而在 C++11 中，标准统一了变量初始化方法，所以模板作者可以总是在模板编写中采用集合初始化（初始化列表）。进一步地，集合初始化对于类型收窄还有一定的限制。而类型收窄也是许多让人深夜工作的奇特错误的源头。因此在 C++11 中使用了初始化列表，就等同于拥有了防止收窄和泛型编程的双重好处。

读者可以在第 4 章看到 C++11 是如何增进语言对新手的支持的。

1.4.5 增强类型的安全性

绝对的类型安全对编程语言来说几乎是不可能达到的，不过在编译时期捕捉更多的错误则是非常有益的。在 C++98/03 中，枚举类会退化为整型，因此常会与其他的枚举类型混淆。这个类型的不安全根源还是在于兼容 C 语言。在 C 中枚举用起来非常便利，在 C++ 中却是类型系统的一个大“漏勺”。因此在 C++11 中，标准引入了新的“强类型枚举”来解决这个问题。

```
enum class Color { red, blue, green };
int x = Color::red;     //C++98/03 中允许，C++11 中错误：不存在 Color->int 的转换
Color y = 7;     //C++98/03 中，C++11 中错误：不存在 int->Color conversion 的转换
Color z = red;     //C++98/03 中允许，C++11 中错误：red 不在作用域内
Color c = Color::red;     //C++98/03 中错误，C++11 中允许
```

在第 5 章中，我们会详细讲解诸如此类能够增强类型安全的 C++11 特性。

1.4.6 与硬件紧密合作

在 C++ 编程中，嵌入式编程是一个非常重要的领域。虽然一些方方圆圆的智能设备外表光鲜亮丽，但是植根于其中的技术基础也常常会是 C++。在 C++11 中，常量表达式以及原子操作都是可以用于支持嵌入式编程的重要特性。这些特性对于提高性能、降低存储空间都大有好处，比如 ROM。

C++98/03 中也具备 const 类型，不过它对只读内存（ROM）支持得不够好。这是因为在 C++ 中 const 类型只在初始化后才意味着它的值应该是常量表达式，从而在运行时不能被改变。不过由于初始化依旧是动态的，这对 ROM 设备来说并不适用。这就要求在动态初始化前就将常量计算出来。为此标准增加了 constexpr，它让函数和变量可以被编译时的常量取

代，而从效果上说，函数和变量在固定内存设备中要求的空间变得更少，因而对于手持、桌面等用于各种移动控制的小型嵌入式设备（甚至心率调整器）的 ROM 而言，C++11 也支持得更好。

在 C++11，我们甚至拥有了直接操作硬件的方法。这里指的是 C++11 中引入的原子类型。C++11 通过引入内存模型，为开发者和系统建立了一个高效的同步机制。作为开发者，通常需要保证线程程序能够正确同步，在程序中不会产生竞争。而相对地，系统（可能是编译器、内存系统，或是缓存一致性机制）则会保证程序员编写的程序（使用原子类型）不会引入数据竞争。而且为了同步，系统会自行禁止某些优化，又保证其他的一些优化有效。除非编写非常底层的并行程序，否则系统的优化对程序员来讲，基本上是透明的。这可能是 C++11 中最大、最华丽的进步。而就算程序员不乐意使用原子类型，而要使用线程，那么使用标准的互斥变量 mutex 来进行临界区的加锁和开锁也就够了。而如果读者还想要疯狂地挖掘并行的速度，或试图完全操控底层，或想找点麻烦，那么无锁（lock-free）的原子类型也可以满足你的各种“野心”。内存模型的机制会保证你不会犯错。只有在使用与系统内存单位不同的位域的时候，内存模型才无法成功地保证同步。比如说下面这个位域的例子，这样的位域常常会引发竞争（跨了一个内存单元），因为这破坏了内存模型的假定，编译器不能保证这是没有竞争的。

```
struct {int a:9; int b:7;}
```

不过如果使用下面的字符位域则不会引发竞争，因为字符位域可以被视为是独立内存位置。而在 C++98/03 中，多线程程序中该写法却通常会引发竞争。这是因为编译器可能将 a 和 b 连续存放，那么对 b 进行赋值（互斥地）的时候就有可能在 a 没有被上锁的情况下一起写掉了。原因是在单线程情况下常被视为普通的安全的优化，却没有考虑到多线程情况下的复杂性。C++11 则在这方面做出了较好的修正。

```
struct {char a; char b;}
```

与硬件紧密合作的能力使得 C++ 可以在任何系统编程中继续保持领先的位置，比如说构建设备驱动或操作系统内核，同时在一些像金融、游戏这样需要高性能后台守护进程的应用中，C++ 的参与也会大大提升其性能。

我们会在第 6 章看到相关特性的描述。

1.4.7 开发能够改变人们思维方式的特性

C++11 中一个小小的 lambda 特性是如何撬动编程世界的呢？从一方面讲，lambda 只是对 C++98/03 中带有 operator() 的局部仿函数（函数对象）包装后的“语法甜点”。事实上，在 C++11 中 lambda 也被处理为匿名的仿函数。当创建 lambda 函数的时候，编译器内部会生成这样一个仿函数，并从其父作用域中取得参数传递给 lambda 函数。不过，真正会改变人们思维方式的是，lambda 是一个局部函数，这在 C++98/03 中我们只能模仿实现该特性。

此外，当程序员开始越来越多地使用 C++11 中先进的并行编程特性时，lambda 会成为一个非常重要的语法。程序员将会发现到处都是奇怪的“lambda 笑脸”，即 ;}[⊖]，而且程序员也必须习惯在各种上下文中阅读翻译 lambda 函数。顺带一提，lambda 笑脸常会出现在每一个 lambda 表达式的终结部分。

另一个人们会改变思维方式的地方则是如何让一个成员函数变得无效。在 C++98/03 中，我们惯用的方法是将成员函数声明为私有的。如果读者不知道这种方法的用意，很可能在阅读代码的时候产生困惑。不过今天的读者非常幸运，因为在 C++11 中不再需要这样的手段。在 C++11 中我们可以通过显式默认和删除的特性，清楚明白地将成员函数设为删除的。这无疑改变了程序员编写和阅读代码的方式，当然，思考问题的方式也就更加直截了当了。

我们会在第 7 章中看到相关特性的描述。

1.4.8 融入编程现实

现实世界中的编程往往都有特殊的需求。比如在访问因特网的时候我们常常需要输入 URL，而 URL 通常都包含了斜线“/”。要在 C++ 中输入斜线却不是件容易的事，通常我们需要转义字符“\”的配合，否则斜线则可能被误认为是除法符号。所以如果读者在写网络地址或目录路径的时候，代码最终看起来就是一堆倒胃口的反斜线的组合，而且会让内容变得晦涩。而 C++11 中的原生字符串常量则可免除“转义”的需要，也可以帮助程序员清晰地呈现网络地址或文件系统目录的真实内容。

另一方面，如今 GNU 的属性（attribute）几乎无所不在，所有的编译器都在尝试支持它，以用于修饰类型、变量和函数等。不过 __attribute__((attribute-name)) 这样的写法，除了不怎么好看外，每一个编译器可能还都有它自己的变体，比如微软的属性就是以 __declspec 打头的。因此在 C++11 中，我们看到了通用属性的出现。

不过 C++11 引入通用属性更大的原因在于，属性可以在不引入额外的关键字的情况下，为编译提供额外的信息。因此，一些可以实现为关键字的特性，也可以用属性来实现（在某些情况下，属性甚至还可以在代码中放入程序供应商的名字，不过这样做存在一些争议）。这在使用关键字还是将特性实现为一个通用属性间就会存在权衡。不过最后标准委员会认为，在现在的情况下，在 C++11 中的通用属性不能破坏已有的类型系统，也不应该在代码中引起语义的歧义。也就是说，有属性的和没有属性的代码在编译时行为是一致的。所以 C++11 标准最终选择创建很少的几个通用属性——noreturn 和 carrier_dependency（其实 final、override 也一度是热门“人选”）。

属性的真正强大之处在于它们能够让编译器供应商创建他们自己的语言扩展，同时不

⊖ lambda 笑脸是一种编写 lambda 函数的编程风格，即在 lambda 函数结束时将分号与括号连写，看起来就是一个 ;} 形式的笑脸。而实际在本书第 7 章中没有采用 lambda 笑脸的编程风格。

会干扰语言或等待特性的标准化。它们可以被用于在一些域内加入特定的“方言”，甚至是在不用 pragma 语法的情况下扩展专有的并行机制（如果读者了解 OpenMP，对此会有一些体会）。

我们将在第 8 章中看到相关的描述。

1.5 本书的约定

1.5.1 关于一些术语的翻译

在 C++11 标准中，我们会涉及很多已有的或新建的术语。在本书中，这些术语我们会尽量翻译，但不求过度翻译。

在已有翻译且翻译意义已经被广为接受的情况下，我们会使用已有的翻译词汇。比如说将 class 翻译为“类”，或者将 template 翻译为“模板”。这样翻译已经为中文读者广为接受，本书则会沿用这样的译法。

而已有翻译但是意义并没有被广为接受的情况下，本书中则会考虑保留英文原文。比如说将“URL”翻译为“统一资源定址器”在我们看来就是一种典型的不良情况。通常将这样的术语翻译为中文会阻碍读者的理解。而大多数能够阅读本书的读者也会具有基本的英文阅读能力和一些常识性的计算机知识，因此本书将保留原文，以期望能够帮助读者更好地理解涉及术语的部分。

对于还没有广泛被认同的中文翻译的术语，我们会采用审慎的态度。一些时候，如果英文确实有利于理解，我们会尝试以注释的方式提供一个中文的解释，而在文中保持英文。如果翻译成中文非常利于理解，则会提供一个中文的翻译，在注释中留下英文。

1.5.2 关于代码中的注释

在本书中，如果可能我们会将一些形如 cout、printf 打印至标准输出 / 错误的内容放在代码的注释中，从读书的经验来看，我们认为这样是最方便阅读的。比如：

```
int a = 2012;
cout << "hello, world" << endl;     // hello, world
cout << a << " is doomed" << endl; // 2012 is doomed
```

同时，一些关键的、有助于读者理解代码的解释也会放在注释中。在通常情况下，注释中有了打印结果的语句不会再有其他的代码解释。如果有，我们将会以逗号将其分开。比如：

```
cout << "hello world" << endl;     // hello world, 打印 "hello world"
```

1.5.3 关于本书中的代码示例与实验平台

在本书的编写中，我们一共使用了 3 种编译器对代码进行编译，即 IBM 的 xlC++、GNU 的 g++，以及 llvm 的 clang++。我们使用的这 3 种编译器都是开发中的版本，其中 xlC++ 使用的是开发中的版本 13，g++ 使用的是开发中的版本 4.8，而 clang++ 则使用的是开发中的版本 3.2。

本书的代码大多数由作者原创，少量使用了 C++11 标准提案中的案例，以及一些网上资源。由于本书编写时，还没有编译器提供对 C++11 所有特性的完整支持，所以通常我们都会将使用的编译器、编译时采用的编译选项罗列在代码处。在本书的代码中，我们会以 g++ 编译为主，但这并不意味着其他编译器无法编译通过这些代码示例。从我们现在看到的结果而言，使用相同特性的代码，编译器的支持往往不存在很大的个体差别（这也是设立标准的意义所在）。而具体的编译器支持，读者则可以通过附录 C 获得相关的信息。

我们的代码运行平台之一是一台运行在 IBM Power 服务器上的 SUSE Linux Enterprise Server 11 (x86_64) 的虚拟机（从我们的实验看来，在该虚拟机上并没有出现与实体机器不一致之处，而不同的 Linux 也不会对我们的实验产生影响）。运行平台之二则是一台运行于 SUSE Linux Enterprise Server 10 SP2 (ppc) 的 IBM Power5+ 服务器。