9

杂项讨论

Miscellany

欢迎来到大杂烩的一章。本章只有 3 个条款，但千万别被低微的数字或不迷人的布景愚弄了，它们都很重要！

第一个条款强调不可以轻忽编译器警告信息。至少，如果你希望你的软件有适当行为的话，别太轻忽它们。第二个条款带你综览 C++ 标准程序库，其中覆盖由 TR1 引进的重大新机能。最后一个条款带你综览 Boost，那是我认为最重要的一个 C++泛用型网站。如果你尝试写出高效 C++ 软件，却没有参考这些条款所提供的信息，那么充其量也只是一场事倍功半的恶战。

条款 53：不要轻忽编译器的警告

Pay attention to compiler warnings.

许多程序员习惯性地忽略编译器警告。他们认为，毕竟，如果问题很严重，编译器应该给一个错误信息而非警告信息，不是吗？这种想法对其他语言或许相对无害，但在 C++，我敢打赌编译器作者对于将会发生的事情比你有更好的领悟。举个例子，下面是多多少少都会发生在每个人身上的一个错误：

```
class B {
public:
  virtual void f( ) const;
};

class D: public B {
public:
  virtual void f( );
};
```

这里希望以 D::f 重新定义 virtual 函数 B::f，但其中有个错误：B 中的 f 是个 const 成员函数，而在 D 中它未被声明为 const。我手上的一个编译器于是这样说话了：

```
warning: D::f() hides virtual B::f()
```

太多经验不足的程序员对这个信息的反应是："噢当然，D::f 遮掩了 B::f，那正是想象中该有的事！"错，这个编译器试图告诉你声明于 B 中的 f 并未在 D 中被重新声明，而是被整个遮掩了（条款 33 描述为什么会这样）。如果忽略这个编译器警告，几乎肯定导致错误的程序行为，然后是许多调试行为，只为了找出编译器其实早就侦测出来并告诉你的事情。

一旦从某个特定编译器的警告信息中获得经验，你将学会了解，不同的信息意味什么——那往往和它们"看起来"的意义十分不同！尽管一般认为，写出一个在最高警告级别下也无任何警告信息的程序最是理想，然而一旦有了上述的经验和对警告信息的深刻理解，你倒是可以选择忽略某些警告信息。不管怎样说，在你打发某个警告信息之前，请确定你了解它意图说出的精确意义。这很重要。

记住，警告信息天生和编译器相依，不同的编译器有不同的警告标准。所以，草率编程然后倚赖编译器为你指出错误，并不可取。例如上述发生"函数遮掩"的代码就可能通过另一个编译器，连半句抱怨和抗议也没有。

请记住

- 严肃对待编译器发出的警告信息。努力在你的编译器的最高（最严苛）警告级别下争取"无任何警告"的荣誉。
- 不要过度倚赖编译器的报警能力，因为不同的编译器对待事情的态度并不相同。一旦移植到另一个编译器上，你原本倚赖的警告信息有可能消失。

条款 54：让自己熟悉包括 TR1 在内的标准程序库

Familiarize yourself with the standard library, including TR1.

C++ Standard ——定义 C++ 语言及其标准程序库的规范——早在 1998 年就被标准委员会核准了。标准委员会又于 2003 年发布一个不很重要的"错误修正版"，并预计于 2008 年左右发布 *C++ Standard* 2.0。日期的不确定性使得人们总是称呼下一版 C++ 为 "C++0x"，意指 200x 版的 C++。

C++0x 或许会覆盖某些有趣的语言新特性，但大部分新机能将以标准程序库的扩充形式体现。如今我们已经能够知道某些新的程序库机能，因为它被详细叙述于一份称为 TR1 的文档内。TR1 代表 "Technical Report 1"，那是 C++ 程序库工作小组对该份文档的称呼。标准委员会保留了 TR1 被正式铭记于 C++0x 之前的修改权，不过目前已不可能再接受任何重大改变了。就所有意图和目标而言，TR1 宣示了一个新版 C++ 的来临，我们可能称之为 *Standard C++ 1.1*。不熟悉 TR1 机能而却奢望成为一位高效的 C++ 程序员是不可能的，因为 TR1 提供的机能几乎对每一种程序库和每一种应用程序都带来利益。

在概括论述 TR1 有些什么之前，让我们先回顾一下 C++98 列入的 C++ 标准程序库有哪些主要成分：

- STL（Standard Template Library，标准模板库），覆盖容器（*containers* 如 `vector`, `string`, `map`）、迭代器（*iterators*）、算法（*algorithms* 如 `find`, `sort`, `transform`）、函数对象（*function objects* 如 `less`, `greater`）、各种容器适配器（*container adapters* 如 `stack`, `priority_queue`）和函数对象适配器（*function object adatpers* 如 `mem_fun`, `not1`）。
- Iostreams，覆盖用户自定缓冲功能、国际化 I/O，以及预先定义好的对象 `cin`, `cout`, `cerr` 和 `clog`。
- 国际化支持，包括多区域（multiple active locales）能力。像 `wchar_t`（通常是 16 bits/char）和 `wstring`（由 `wchar_t`s 组成的 `strings`）等类型都对促进 Unicode 有所帮助。
- 数值处理，包括复数模板（`complex`）和纯数值数组（`valarray`）。
- 异常阶层体系（exception hierarchy），包括 base class `exception` 及其 derived classes `logic_error` 和 `runtime_error`，以及更深继承的各个 classes。
- C89 标准程序库。1989 C 标准程序库内的每个东西也都被覆盖于 C++ 内。

如果你对上述任何一项不很熟悉，我建议你好好排出一些时间，带着你最喜爱的 C++ 书籍，把情势扭转过来。

TR1 详细叙述了 14 个新组件（components，也就是程序库机能单位），统统都放在 `std` 命名空间内，更正确地说是在其嵌套命名空间 `tr1` 内。因此，TR1 组件 `shared_ptr` 的全名是 `std::tr1::shared_ptr`。本书通常在讨论标准程序库组件时略而不写 `std::`，但我总是会在 TR1 组件之前加上 `tr1::`。

本书展示以下 TR1 组件实例：

- 智能指针（**smart pointers**）`tr1::shared_ptr` 和 `tr1::weak_ptr`。前者的作用有如内置指针，但会记录有多少个 `tr1::shared_ptrs` 共同指向同一个对象。这便是所谓的 *reference counting*（引用计数）。一旦最后一个这样的指针被销毁，也就是一旦某对象的引用次数变成 0，这个对象会被自动删除。这在非环形（acyclic）数据结构中防止资源泄漏很有帮助，但如果两个或多个对象内含 `tr1::shared_ptrs` 并形成环状（cycle），这个环形会造成每个对象的引用次数都超过 0——即使指向这个环形的所有指针都已被销毁（也就是这一群对象整体看来已无法触及）。这就是为什么又有个 `tr1::weak_ptrs` 的原因。`tr1::weak_ptrs` 的设计使其表现像是“非环形 `tr1::shared_ptr`-based 数据结构”中的环形感生指针（cycle-inducing pointers）。`tr1::weak_ptrs` 并不参与引用计数的计算；当最后一个指向某对象的 `tr1::shared_ptr` 被销毁，纵使还有个 `tr1::weak_ptrs` 继续指向同一对象，该对象仍旧会被删除。这种情况下的 `tr1::weak_ptrs` 会被自动标示无效。

  `tr1::shared_ptr` 或许是拥有最广泛用途的 TR1 组件。本书多次使用它，条款 13 解释它为什么如此重要。本书并未示范使用 `tr1::weak_ptr`，抱歉。

- **`tr1::function`**，此物得以表示任何 *callable entity*（可调用物，也就是任何函数或函数对象），只要其签名符合目标。假设我们想注册一个 callback 函数，该函数接受一个 `int` 并返回一个 `string`，我们可以这么写：

```
void registerCallback(std::string func(int));
                      //参数类型是函数，该函数接受一个 int 并返回一个 string
```

  其中参数名称 `func` 可有可无，所以上述的 `registerCallback` 也可以这样声明：

```
void registerCallback(std::string (int)); //与上同；参数名称略而未写
```

  注意这里的 `"std::string (int)"` 是个函数签名。

tr1::function使上述的RegisterCallback有可能更富弹性地接受任何可调用物（callable entity），只要这个可调用物接受一个 int 或任何可被转换为 int 的东西，并返回一个 string 或任何可被转换为 string 的东西。tr1::function 是个 template，以其目标函数的签名（target function signature）为参数：

```
void registerCallback(std::tr1::function<std::string (int)> func);
//参数 "func" 接受任何可调用物（callable entity）
//只要该“可调用物”的签名与 "std::string (int)" 一致
```

这种弹性真令人惊讶，我尽最大的努力在条款 35 示范了它的用法。

- **tr1::bind**，它能够做 STL 绑定器（***binder***s）bind1st 和 bind2nd 所做的每一件事，而又更多。和前任绑定器不同的是，tr1::bind 可以和 const 及 non-const 成员函数协同运作，可以和 ***by-reference*** 参数协同运作。而且它不需特殊协助就可以处理函数指针，所以我们调用 tr1::bind 之前不必再被什么 ptr_fun，mem_fun 或 mem_fun_ref 搞得一团混乱了。简单地说，tr1::bind 是第二代绑定工具（binding facility），比其前一代好很多。我在条款 35 示范过它的用法。

我把其他 TR1 组件划分为两组。第一组提供彼此互不相干的独立机能：

- Hash tables，用来实现 sets, multisets, maps 和 multi-maps。每个新容器的接口都以其前任（TR1 之前的）对应容器塑模而成。最令人惊讶的是它们的名称：tr1::unordered_set,tr1::unordered_multiset,tr1::unordered_map 以及 tr1::unordered_multimap。这些名称强调它们和 set, multiset, map 或 multimap 不同：以 hash 为基础的这些 TR1 容器内的元素并无任何可预期的次序。
- 正则表达式（Regular expressions），包括以正则表达式为基础的字符串查找和替换，或是从某个匹配字符串到另一个匹配字符串的逐一迭代（iteration）等等。
- Tuples（变量组），这是标准程序库中的 pair template 的新一代制品。pair 只能持有两个对象，tr1::tuple 可持有任意个数的对象。漫游于 Python 和 Eiffel 的程序员，额手称庆吧！你们前一个家园的某些好东西现在已经纳入 C++。

- **tr1::array**，本质上是个“STL 化”数组，即一个支持成员函数如 begin 和 end 的数组。不过 tr1::array 的大小固定，并不使用动态内存。
- **tr1::mem_fn**，这是个语句构造上与成员函数指针（member function pointers）一致的东西。就像 tr1::bind 纳入并扩充 C++98 的 bind1st 和 bind2nd 的能力一样，tr1::mem_fn 纳入并扩充了 C++98 的 mem_fun 和 mem_fun_ref 的能力。
- **tr1::reference_wrapper**，一个“让 references 的行为更像对象”的设施。它可以造成容器“犹如持有 references”。而你知道，容器实际上只能持有对象或指针。
- 随机数（random number）生成工具，它大大超越了 rand，那是 C++ 继承自 C 标准程序库的一个函数。
- 数学特殊函数，包括 Laguerre 多项式、Bessel 函数、完全椭圆积分（complete elliptic integrals），以及更多数学函数。
- C99 兼容扩充。这是一大堆函数和模板（templates），用来将许多新的 C99 程序库特性带进 C++。

第二组 TR1 组件由更精巧的 template 编程技术（包括 template metaprogramming，也就是模板元编程，见条款 48）构成：

- Type traits，一组 traits classes（见条款 47），用以提供类型（types）的编译期信息。给予一个类型 T，TR1 的 type traits 可以指出 T 是否是个内置类型，是否提供 virtual 析构函数，是否是个 empty class（见条款 39），可隐式转换为其他类型 U 吗……等等。TR1 的 type traits 也可以显现该给定类型之适当齐位（proper alignment），这对定制型内存分配器（见条款 50）的编写人员是十分关键的信息。
- **tr1::result_of**，这是个 template，用来推导函数调用的返回类型。当我们编写 templates 时，能够“指涉（*refer to*）函数（或函数模板）调用动作所返回的对象的类型”往往很重要，但是该类型有可能以复杂的方式取决于函数的参数类型。tr1::result_of 使得“指涉函数返回类型”变得十分容易。它也被 TR1 自身的若干组件采用。

虽然若干 TR1 成分（特别是 tr1::bind 和 tr1::mem_fn）纳入了某些“前 TR1”组件能力，但其实 TR1 是对标准程序库的纯粹添加，没有任何 TR1 组件用来替换既有组件，所以早期（写于 TR1 之前的）代码仍然有效。

TR1 自身只是一份文档[1]。为了取得它所规范的那些机能，你还需要取得实现代码。这些代码最终会随编译器出货。在我下笔的 2005 年此刻，如果你在你手上的标准程序库实现版本内寻找 TR1 组件，极可能有某些遗漏。幸运的是你可以补齐它们：TR1 的 14 个组件中的 10 个奠基于免费的 Boost 程序库（见条款 55），所以对 TR1-like 机能而言，Boost 是个绝佳资源。我说 "TR1-like" 是因为虽然许多 TR1 机能奠基于 Boost 程序库，但毕竟有些 Boost 机能并不完全吻合 TR1 规范。当你阅读这一段文字，说不定 Boost 已经不只提供与 TR1 一致的实现（对于那些奠基于 Boost 程序库的 10 个 TR1 组件），还供应 4 个不以 Boost 为基础的 TR1 组件实现。

在编译器附带 TR1 实现品的那一刻到来之前，如果你喜欢以 Boost 的 TR1-like 程序库作为一时权宜，或许你会愿意以一个命名空间上的小伎俩让自己将来好过些。所有 Boost 组件都位于命名空间 `boost` 内，但 TR1 组件都置于 `std::tr1` 内。你可以这样告诉你的编译器，令它对待 *references to* `std::tr1` 就像对待 *references to* `boost` 一样：

```
namespace std {
    namespace tr1 = ::boost;       //namespace std::tr1 是
}                                  //namespace boost 的一个别名
```

纯就技术而言，这简直是把你流放到“未定义行为”的国土去了，因为就如条款 25 所言，任何人不得加任何东西到 `std` 命名空间去。然而实际上你很可能不会有任何麻烦。一旦将来你的编译器提供它们自己的 TR1 实现品，你需要做的唯一事情就是消除上述的 namespace 别名，而后指涉 `std::tr1` 的代码继续生效，好极了。

非以 Boost 程序库为基础的那些 TR1 组件之中，最重要的或许是 hash tables。其实 hash tables 早已行之有年，分别以名称 `hash_set`, `hash_multiset`, `hash_map` 和 `hash_multimap` 为人熟知。也许你的编译器已经附带那些 templates 实现码。如果没有，请启动你最喜欢的查找引擎，查找那些名称（及其 TR1 称号），你一定可以找到若干来源，包括商业产品和免费产品。

---

1 在我下笔此刻的 2005 年初，这份文件尚未定稿，其 URL 常有变化。我建议你咨询 *Effective C++* TR1 信息网页，http://aristeia.com/EC3E/TR1_info.html。这个 URL 很稳定。

请记住

- C++ 标准程序库的主要机能由 STL、iostreams、locales 组成。并包含 C99 标准程序库。
- TR1 添加了智能指针（例如 `tr1::shared_ptr`）、一般化函数指针（`tr1::function`）、hash-based 容器、正则表达式（regular expressions）以及另外 10 个组件的支持。
- TR1 自身只是一份规范。为获得 TR1 提供的好处，你需要一份实物。一个好的实物来源是 Boost。

条款 55：让自己熟悉 Boost

Familiarize yourself with Boost.

你正在寻找一个高质量、源码开放、平台独立、编译器独立的程序库吗？看看 Boost 吧。有兴趣加入一个由雄心勃勃充满才干的 C++ 开发人员组成的社群，致力发展（设计和实现）当前最高技术水平之程序库吗？看看 Boost 吧！想要一瞥未来的 C++可能长相吗？看看 Boost 吧！

Boost 是一个 C++ 开发者集结的社群，也是一个可自由下载的 C++ 程序库群。它的网址是 `http://boost.org`。现在你应该把它设为你的桌面书签之一。

当然，世上多得是 C++ 组织和网站，但 Boost 有两件事是其他任何组织无可匹敌的。第一，它和 C++ 标准委员会之间有着独一无二的密切关系，并且对委员会深具影响力。Boost 由委员会成员创设，因此 Boost 成员和委员会成员有很大的重叠。Boost 有个目标：作为一个“可被加入标准 C++ 之各种功能”的测试场。这层关系造就的结果是，以 TR1（见条款 54）提案进入标准 C++ 的 14 个新程序库中，超过三分之二奠基于 Boost 的工作成果。

Boost 的第二个特点是：它接纳程序库的过程。它以公开进行的同僚复审（public peer review）为基础。如果你打算贡献一个程序库给 Boost，首先要对 Boost 开发者电邮名单（mailing list）投递作品，让他们评估这个程序库的重要性，并启动初步审查程序。然后开始这个网站所谓的“讨论、琢磨、再次提交”循环周期，直到一切都获得满足为止。

最后，你准备好你的程序库，要正式提交了。会有一位复审管理员出面确认你的程序库符合 Boost 最低要求。例如它必须通过至少两个编译器（以展现至此仍还微不足道的可移植性），你必须证明你的程序库在一个可接受的授权许可下是可用

的（例如这个程序库必须允许免费的商业化和非商业化用途）。然后你的提交正式进入 Boost 社群，等待官方复审。复审期间会有志愿者察看你的程序库各种素材（例如源码、设计文档、使用说明等等），并考虑诸如此类的问题：

- 这一份设计和实现有多好？
- 这些代码可跨编译器和操作系统吗？
- 这个程序库有可能被它所设定的目标用户——也就是在这个程序库企图解决问题的领域中工作的人们——使用吗？
- 文档是否清楚、齐备，而且精确？

所有批注都会被投寄至一份 Boost 邮件列表，所以复审者和其他人可以看到并响应其他人的评论。复审最后周期结束之后，复审管理员便表决你的程序库被接受、被有条件接受，或被拒绝。

同僚复审对于阻挡低劣的程序库很有贡献，同时也教育程序库作者认真考虑一个工业强度、跨平台的程序库的设计、实现和文档工程。许多程序库在被 Boost 接受之前，往往经历了一次以上的官方复审。

Boost 内含数十个程序库，而且还不断有更多添加进来。偶尔也会有程序库被从中移除，通常那是因为它们的机能已被新程序库取代，而新程序库提供了更多、更好的机能，或更好的设计（例如更弹性或更有效率）。

Boost 各程序库之间的大小和作用范围有很大变化。举一个极端例子，某些程序库概念上只需数行代码（但在加入错误处理和可移植性后往往变长很多）。例如 Conversion 程序库，提供较安全或较方便的转型操作符，其 `numeric_cast` 函数在将数值从某类型转换为另一类型而导致溢出（overflow）或下溢（underflow）或类似问题时会抛出异常。`lexical_cast` 则使我们得以将任何类型（只要支持 `operator<<`）转换为字符串，对程序的诊断和运转志记（logging）都十分有用。另一个极端例子是某些程序库提供大面积能力，甚至可以写成一整本书，这类程序库包括 Boost Graph Library（用于编写任意 graph 结构）和 Boost MPL Library（一个元编程程序库，metaprogramming library）。

Boost 程序库对付的主题非常繁多，区分数十个类目，包括：

- **字符串与文本处理**，覆盖具备类型安全（type-safe）的 printf-like 格式化动作、正则表达式（此为 TR1 同类机能的基础，见条款 54），以及语汇单元切割（tokenizing）和解析（parsing）。
- **容器**，覆盖“接口与 STL 相似且大小固定”的数组（见条款 54）、大小可变的 bitsets 以及多维数组。
- **函数对象和高级编程**，覆盖若干被用来作为 TR1 机能基础的程序库。其中一个有趣的程序库是 Lambda，它让我们得以轻松地随时随地创建函数对象，但是你颇有可能不太了解你正在做什么：

```
using namespace boost::lambda;      //让 boost::lambda 的机能曝光
std::vector<int> v;
...
std::for_each(v.begin(), v.end(),          //针对 v 内的每一个元素 x，
     std::cout << _1 * 2 + 10 << "\n"); //印出 x * 2+10;
                                        //其中 "_1" 是 Lambda 程序库
                                        //针对当前元素的一个
                                        //占位符号（placeholder）
```

- **泛型编程**（Generic programming），覆盖一大组 traits classes。关于 traits 请见条款 47。
- **模板元编程**（Template metaprogramming，TMP，见条款 48），覆盖一个针对编译期 assertions 而写的程序库，以及 Boost MPL 程序库。MPL 提供了极好的东西，其中支持编译期实物（compile-time entities）诸如 *types* 的 STL-like 数据结构，等等。

```
//创建一个 list-like 编译期容器，其中收纳三个类型：
//（float, double, long double），并将此容器命名为 "floats"
typedef boost::mpl::list<float, double, long double> floats;

//再创建一个编译期间用以收纳类型的 list，以 "floats" 内的类型为基础，
//最前面再加上 "int"。新容器取名为 "types"。
typedef boost::mpl::push_front<floats, int>::type types;
```

这样的“类型容器”（常被称为 *typelists*——虽然它们也可以以一个 mpl::vector 或 mpl::list 为基础）开启了一扇大门，通往大范围、火力强大且重要的 TMP 应用程序。

- **数学和数值**（Math and numerics），包括有理数、八元数和四元数（octonions and quaternions）、常见的公约数（divisor）和少见的多重运算、随机数（又一个影

响 TR1 内部相关机能的程序库）。

- **正确性与测试**（Correctness and testing），覆盖用来将隐式模板接口（implicit template interfaces，见条款 41）形式化的程序库，以及针对“测试优先”编程形态而设计的措施。
- **数据结构**，覆盖类型安全（type-safe）的 unions（存储各具差异之“任何”类型），以及 tuple 程序库（它是 TR1 同类机能的基础）。
- **语言间的支持**（Inter-language support），包括允许 C++ 和 Python 之间的无缝互操作性（seamless interoperability）。
- **内存**，覆盖 Pool 程序库，用来做出高效率而区块大小固定的分配器（见条款 50），以及多变化的智能指针（smart pointers，见条款 13），包括（但不仅仅是）TR1 智能指针。另有一个 non-TR1 智能指针是 `scoped_array`，那是个 `auto_ptr`-like 智能指针，用来动态分配数组；条款 44 曾经示范其用法。
- **杂项**，包括 CRC 检验、日期和时间的处理、在文件系统上来回移动等等。

请记住，这只是可在 Boost 中找到的程序库抽样，不是一份详尽清单。

Boost 提供的程序库可以做很多很多事，但它并未覆盖整套编程风光。例如其中就没有针对 GUI 开发而设计的程序库，也没有用以连通数据库的程序库——至少在我下笔此刻没有。然而当你阅读本书时就有了也说不定。到底有没有，唯一可以确定的办法是常常上网检核。我建议你现在就去访问：`http://boost.org`。纵使你没能找到刚好符合需求的作品，也一定会在其中发现一些有趣的东西。

请记住

- Boost 是一个社群，也是一个网站。致力于免费、源码开放、同僚复审的 C++ 程序库开发。Boost 在 C++ 标准化过程中扮演深具影响力的角色。
- Boost 提供许多 TR1 组件实现品，以及其他许多程序库。