（译自“Modern C++ Design”。仅限个人使用，不得转载传播）

第 7 章　智能指针（Smart Pointers）

作者：Andrei Alexandrescu　　译者：马维达

智能指针已经成为全世界的程序员和作家所编写的无数代码和所消耗的墨水之河的主题。作为或许是最为流行、复杂和强大的 C++习语（idiom），其有趣之处在于它们结合了许多语法和语义上的论题。这一章将讨论智能指针，从最为简单的方面到最为复杂的方面，从实现它们时最为明显的错误到最为微妙的错误——其中一些错误碰巧也是最为可怕的。

简而言之，智能指针是通过实现 operator->和一元 operator*来模拟简单指针的 C++对象。除了表面的指针语法和语义，智能指针还常常在底层执行一些有用的任务，比如内存管理或锁定，从而将应用从“对所指向对象的生存期的小心管理”中解放出来。

本章不仅将讨论智能指针，还将实现一个 SmartPtr 类模板。SmartPtr 是围绕策略（policy。见第 1 章）来设计的，所得到的是这样的智能指针：它正好有着你想要的安全、效率和易用性。

读过本章后，你将成为智能指针问题的专家，比如：

- 智能指针的优点和缺点
- 所有权管理方略（strategy）
- 隐式转换
- 测试与比较
- 多线程问题

本章实现了一个泛型 SmartPtr 类模板。每一部分分别介绍一个实现问题。在结束时，该实现将所有的片段放在一起。除了理解 SmartPtr 的设计原理，你还将学会怎样对其进行使用、调整和扩展。

7.1　智能指针 101（Smart Pointers 101）

那么什么是智能指针？智能指针是在语法和某些语义上模拟普通指针，但所做更多的 C++类。因为指向不同类型的对象的智能指针往往有着许多相同代码，几乎所有现有的、质量好的智能指针都是通过被指向者类型来进行模板化的，如下所示：

```
template <class T>
class SmartPtr
{
public:
  explicit SmartPtr(T*pointee):pointee_(pointee);
  SmartPtr&operator=(const SmartPtr&other);
  ~SmartPtr();
  T&operator*()const
  {
      ...
      return *pointee_;
  }
```

```
  T*operator->()const
  {
      ...
      return pointee_;
  }

private:
  T*pointee_;
  ...
};
```

SmartPtr<T>在其成员变量 pointee_聚合了一个指向 T 的指针。这是大多数智能指针所做的事情。在有些情况下，智能指针可以聚合一些指向数据的句柄，并在运行中对指针进行计算。

两个操作符给了 SmartPtr 类似指针的语法和语义。也就是，你可以编写：

```
class Widget
{
public:
  void Fun();
};

SmartPtr<Widget>sp(new Widget);
sp->Fun();
(*sp).Fun();
```

除了 sp 的定义，没有什么显示其不是指针。这是智能指针的曼陀罗（mantra）：你可以用智能指针定义来取代指针定义，而不需要对你的应用代码进行大的变动。这样你很容易就得到了额外的糖果。要让大型应用使用智能指针，微量的代码变动是非常吸引人和至关紧要的。但是，如你很快就会看到的，智能指针不是免费午餐。

7.2 交易（The Deal）

但与智能指针的交易是什么？你可能会问。用智能指针来取代简单的指针，你可以得到什么？解释很简单。智能指针有值语义（value semantic），而某些简单指针则没有。

具有值语义的对象是你可以进行*复制*和*向其赋值*的对象。int 类型是第一类对象的完美实例。你可以自由地创建、复制和改变整数值。你用于在缓冲区中进行迭代的指针也有值语义——你进行初始化，使其指向缓冲区的开头，并且增大它，直到到达末尾。在此过程中，你可以将它的值复制到其他变量中，以保存临时的结果。

但是，对于通过 new 分配的持有值的指针，其故事极其不同。例如：

```
Widget*p =new Widget;
```

变量 p 不仅指向、而且也*拥有*为 Widget 对象所分配的内存。这是因为随后你必须发出 delete p 确保 Widget 对象的销毁及其内存的释放。如果在刚才所示的代码行后面你写下：

```
p = 0; // assign something else to p
```

你就失去了先前由 p 所指向的对象的所有权，而且你也完全没有机会再获得对它的掌握。你所得到的是资源泄漏，而资源泄漏从不是好事。

更进一步，在你将 p 复制到另一变量中时，编译器不会自动地管理指针所指向的内存的所有权。你所得到的只是两个指向同一对象的原始指针，而且你还必须更为小心地跟踪它们，因为重复删除比起没有删除更是灾难性的。所以，指向被分配对象的指针*没有*值语义——你不能随便对其进行或赋值。

在这一区域智能指针可以起到极大的作用。除了类似指针的行为，大多数智能指针还提供*所有权管理*。智能指针可以分析出所有权是怎样变化的，并且它们的析构器可以根据良好定义的方略来释放内存。许多智能指针拥有足够的信息来在释放所指向对象的问题上采取完全的主动。

智能指针可以通过不同的方式来管理所有权，每一种方式适用于不同的问题范畴。某些智能指针自动转移所有权：在你复制指向某对象的智能指针后，源智能指针变为空，而目的指针则指向该对象（并且持有其所有权）。这是标准所提供的 std::auto_ptr 所实现的行为。其他一些智能指针实现引用计数：它们跟踪指向同一对象的智能指针的总数，并在此计数降为零时，删除（delete）所指向的对象。最后，还有一些智能指针在你复制（copy）它们时使其所指向的对象变为两份（duplicate）。

简言之，在智能指针的世界里，所有权是一项重要话题。通过提供所有权管理，智能指针能够对完整性保证和完全的值语义进行支持。因为所有权与智能指针的构造、复制和销毁有着莫大的关联，很容易推断出它们就是智能指针的最为重要的功能。

下面的一些部分将讨论智能指针设计和实现的各个方面。其目标是使智能指针尽可能地接近原始指针，直到不能再接近为止。该目标是自相矛盾的：毕竟，如果你的智能指针的行为*完全*像是哑指针（dumb pointer），那它们*就是*哑指针。

实现智能指针与原始指针的兼容性时，在“令人满意地满足兼容性需求”与“铺设通向混乱的道路”之间有一条细细的分界线。你将会发现，增加看上去值得去实现的特性可能会使客户面临代价昂贵的风险。实现好的智能指针的许多工艺都是由对它们的特性集的小心平衡组成的。

7.3 智能指针的存储（Storage of Smart Pointers）

让我们问一个关于智能指针的基础性问题作为开始。pointee_的类型必然是 T*吗？如果不是，它还可以是什么类型？在泛型编程中，你应该总是问你自己这样的问题。硬编码进泛型代码中的每一种类型都会降低代码的普泛性。硬编码的类型对于泛型代码，就像是魔术常数对于普通代码一样。

在若干情形中，允许定制被指向者的类型是值得的。一种情形是在你处理非标准的指针修饰符时。在 16 位 Intel 80x86 的日子里，你可以通过像__near __far、和__huge 这样的修饰符来限定指针。其他的分段架构也使用类似的修饰符。

另一种情形是在你想要对智能指针进行分层时。如果你拥有别人实现的一个 LegacySmartPtr<T>智能指针，而你想要对其进行增强，应该怎么办？你会对它进行继承吗？那是一个危险的决定。最好是将遗留的智能指针包装进你自己的智能指针中。这是可能的，因为内部的智能指针支持指针语法。从外层的智能指针的视点来看，被指向者的类型不是 T*而是 LegacySmartPtr<T>

智能指针分层有着有趣的应用，主要是因为 operator->的语义。当你将 operator->应用于不是内建指针的类型时，编译器会去做有趣的事情。在查找用户定义的 operator->、并将其应用于那个类型后，它还会再次将 operator->应用于所得结果。编译器递归地进行这样的操作，直到它到达指向内建类型的指针为止，并且直到这时才处理成员访问。所以，智能指针的 operator->不一定要返回指针，它可以返回依次实现了 operator->的对象，而不用改变使用语法。

这样的语义产生了非常有趣的习语：前函数和后函数调用（Stroustrup 2000）。如果你通过传值从 operator->返回 PointerType 类型的对象，其执行序列如下所示：

1. PointerType 的构造器
2. PointerType::operator->被调用；很可能返回指向类型为 PointeeType 的对象
3. 访问 PointeeType 的成员——很可能是函数调用
4. PointerType 的析构器

简而言之，你有了一种实现加锁的函数调用的漂亮方法。这一习语在多线程和加锁的资源访问方面有着广泛的用途。你可以让 PointerType 的构造器锁定资源，随后你就可以访问资源；最后，PointerType 的析构器解除资源锁定。

泛化并未停止于此。与包含在智能指针中的强大的资源管理技术相比，面向语法的“指针”部分有时会变得苍白无力。所以，在一些罕见的情况下，智能指针可以放弃指针语法。没有定义 operator->和 operator*的对象违反了智能指针的定义，但是也有一些这样的对象很应该得到像智能指针一样的对待，尽管它们，严格地讲，不是智能指针。

看一看真实世界的 API 和应用，许多操作系统采用*句柄*（handle）作为特定内部资源——比如窗口、互斥体或设备——的访问符。句柄是被有意模糊化的指针；其目的之一是防止它们的用户直接操作关键的操作系统资源。大多数时候，句柄是一些整数值，是隐藏的指针表的索引。这张表提供了额外的间接层次，以防止应用程序员破坏内部系统。尽管它们没有提供 operator->，句柄在其语义和它们被管理的方式上与指针是类似的。

对于这样的智能资源，提供 operator->和 operator*是没有意义的。但是，你可以利用所有智能指针特有的资源管理技术。

为泛化智能指针的类型世界，我们区分了在智能指针中三种潜在的不同类型：

- 存储类型（storage type）。这是 pointee_的类型。“缺省”地——在普通的智能指针中——它是原始指针。
- 指针类型（pointer type）。这是 operator->所返回的类型。它可以与存储类型不同，如果你想要返回代理对象、而不是指针的话。（在这一章的后面，你将会发现使用代理对象的实例。）
- 引用类型（reference type）。这是 operator*所返回的类型。

如果 SmartPtr 以一种灵活的方式支持这一概括，那将会是有益的。因而，这里提到的三种类型应当被抽象进叫做 Storage 的策略中。

总之，智能指针能够、而且应该泛化被指向者的类型。为做到这一点，SmartPtr 在 Storage 策略中抽象了三种类型：被存储的类型、指针类型，以及引用类型。对于特定的 SmartPtr 实例化，不是所有的类型都必然是有意义的。因此，在一些罕见的情况下（句柄），一种策略可以使对 operator->或\和 operator*的访问失效。

7.4 智能指针成员函数（Smart Pointer Member Functions）

许多现有的智能指针实现允许通过成员函数进行操作，比如用于访问被指向对象的 Get，用于改变它的 Set，以及用于放弃所有权的 Release。这是明显的而自然的封装 SmartPtr 的功能的途径。

但是，经验已证明成员函数不是特别适合智能指针。原因是“对智能指针的成员函数的调用”与“对所指向对象的成员函数的调用”之间的相互干扰可能会极其让人迷惑。

例如，假设你有一个 Printer 类，它有着像 Acquire 和 Release 这样的成员函数。通过 Acquire 你可以接管打印机的所有权，这样别的应用就不能再向其进行打印；而通过 Release 你可以放弃所有权。因为你使用了指向 Printer 的智能指针，你可能会注意到语义上非常不同的事物，却有着奇怪的语法近似性。

```
SmartPtr<Printer>spRes =...;
```

```
spRes->Acquire();//acquire the printer
...print a document ...
spRes->Release();//release the printer
spRes.Release();//release the pointer to the printer
```

SmartPtr 的用户现在可以访问两个完全不同的世界：所指向对象的成员世界和智能指针的成员世界。一个点或一个箭头的事情脆弱地分割了这两个世界。

在表面上，C++的确强制你经常性地观察语法上特定的细微差别。学习 C++的 Pascal 程序员可能会觉得&和&&之间细微的语法差别是可恶的事情，然而 C++程序员甚至不会朝它眨一下眼睛。他们所受的训练使他们习惯于毫不费力地区分这样的语法问题。

但是，智能指针成员函数会使习惯性训练变得无效。原始指针没有成员函数，所以 C++程序员不习惯检测和区分点调用和箭头调用。编译器则工作良好：如果你在原始指针后使用点调用，编译器就会报告错误。因此，很容易想象，而且经验也证明了，即使是老到的 C++程序员也会发现这一事实是极其烦扰人的：sp.Release()和 sp->Release()都可以无错编译，但所做的却是非常不同的事情。处方很简单：智能指针不应该使用成员函数。SmartPtr 只使用非成员函数。这些函数成了智能指针类的朋友。

重载函数也可以如智能指针的成员函数一样让人迷惑，但是却有着重要的区别。C++程序员已经使用了重载函数。重载是 C++语言的重要组成部分，并且例行公事般地被用于库和应用开发中。这意味着 C++程序员在编写和复查代码时，*肯定*会注意函数调用语法中的不同，比如像 Release(*sp) vs. Release(sp)

仅有的 SmartPtr 必须保留的成员函数是构造器、析构器、operator= operator->，以及一元 operator* SmartPtr 的所有其他的操作都是通过指定的非成员函数来提供的。

为了清晰性的缘故，SmartPtr 没有指定任何成员函数。仅有的访问被指向对象的函数是 GetImpl GetImplRef Reset，以及 Release；它们是在名字空间级被定义的。

```
template <class T>T*GetImpl(SmartPtr<T>&sp);
template <class T>T*&GetImplRef(SmartPtr<T>&sp);
template <class T>void Reset(SmartPtr<T>&sp,T*source);
template <class T>void Release(SmartPtr<T>&sp,T*&destination);
```

- GetImpl 返回由 SmartPtr 所存储的指针对象。
- GetImplRef 返回对“由 SmartPtr 所存储的指针对象”的*引用* GetImplRef 允许你改变底层指针，所以在使用时需要非常小心。
- Reset 将底层指针重置为其他值，并释放先前的值。
- Release 释放智能指针的所有权，赋予它的用户管理被指向对象的生存期的责任。

在 Loki 中实际上这四个函数的声明要更为精细一些。它们没有假定由 SmartPtr 所存储的指针对象的类型是 T*。如 7.3 部分所讨论的，Storage 策略定义了指针类型。大多数时候，它是直接的指针，而在 Storage 的特殊实现中，它可能是句柄或详细制作的类型。

7.5 所有权处理策略（Ownership-Handling Strategies）

所有权处理常常是智能指针最为重要的存在理由。通常，从它们的客户的视点来看，智能指针拥有它们所指向的对象。智能指针是第一类的值，负责在底层删除所指向的对象。客户可以通过发出对助手管理函数的调用来干预被指向对象的生存期。

为实现自所有权（self-ownership），智能指针必须小心地跟踪被指向对象，特别是在复制、赋

值以及析构的过程中。这样的跟踪带来了一些空间和/或时间上的开销。应用应该采用最能解决手上的问题、而且开销不大的策略。

下面的小节讨论最为流行的所有权管理方略以及 SmartPtr 是怎样实现它们的。

7.5.1 深度复制（Deep Copy）

可应用的最为简单的策略是只要你复制智能指针，就复制被指向对象。如果你确保这一点，每个被指向对象就只有一个智能指针。因此，智能指针的析构器可以安全地删除（delete）被指向对象。图 7.1 描述了在你使用深度复制智能指针的情况下的事态。

初看上去，深度复制策略有点无趣，好像智能指针并没有在普通的 C++值语义之上增加任何价值。如果被指向对象的简单传值工作得也一样好，你又为什么要费功夫去使用智能指针呢？

答案是*对多态的支持*。智能指针是安全地传送多态对象的交通工具。你持有一个指向基类的智能指针，它实际上指向的可能是派生类。当你复制智能指针时，你也想要复制它的多态行为。有趣的是你并不明确地知道你正在处理什么样的行为和状态，但你却又的确需要复制该行为和状态。

因为深度复制往往要处理多态对象，下面的对复制构造器的天真实现是错误的：

```
template <class T>
class SmartPtr
{
public:
  SmartPtr(const SmartPtr&other)
  :pointee_(new T(*other.pointee_))
  {
  }
  ...
};
```

假定你复制一个类型为 SmartPtr<Widget>的对象。如果 Other 指向派生自 Widget 的类 ExtendedWidget 的实例，上面的复制构造器仅仅复制了 ExtendWidget 对象的 Widget 部分。这样的现象被称为“*切断*”（slicing）——所复制的只是可能更大的 ExtendedWidget 对象的 Widget“断片”。切断往往是不合需要的。遗憾的是在 C++中很容易发生切断——一个简单的传值调用就会切断对象，而不会产生任何警告。

第 8 章将深入地讨论克隆(cloning)。如那里所示，获取层次结构的多态克隆的经典办法是这样来定义虚拟 Clone 函数并实现它：

```
class AbstractBase
{
  ...
  virtual Base*Clone()=0;
};

class Concrete :public AbstractBase
{
  ...
  virtual Base*Clone()
  {
      return new Concrete(*this);
```

```
  }
};
```

在所有的派生类中 Clone 实现都必须遵循同样的模式；尽管它有着重复性的结构，并不存在有效的自动定义 Clone 成员函数的方法（更确切地说，除了宏以外）。

泛型智能指针不能依赖于克隆成员函数的确切名称——也许它是 Clone，又或许是 MakeCopy 因此，最为灵活的途径是通过针对克隆的策略来参数化 SmartPtr

7.5.2 Copy on Write

Copy on write（COW，如它的爱好者所亲切地称呼的）是一种避免不必要的对象复制的优化技术。在 COW 之下的想法是：在第一次试图修改被指向对象时对其进行克隆；在此之前，若干指针可以共享同一对象。

但是，智能指针不是实现 COW 的最佳场所，因为智能指针无法区分对被指向对象的 const 和非 const 成员函数的调用。这里有一个例子:

```
template <class T>
class SmartPtr
{
public:
  T*operator->(){return pointee_;}
...
};

class Foo
{
public:
  void ConstFun()const;
  void NonConstFun();
};

...
SmartPtr<Foo>sp;
sp->ConstFun();//invokes operator->,then ConstFun
sp->NonConstFun();//invokes operator->,then NonConstFun
```

同样的 operator->被用于调用两个函数；因此，智能指针没有任何应否进行 COW 的线索。对被指向对象的函数的调用发生在智能指针所不能及的某处（7.11 部分将解释 const 是怎样与智能指针及其所指向的对象相互作用的。）

总之，COW 在作为全特性类的实现优化时最为有效。智能指针处在过低的层面上，无法有效地实现 COW 语义。当然，在为某个类实现 COW 时，智能指针可以是良好的构件。

本章中的 SmartPtr 实现没有提供对 COW 的支持。

7.5.3 引用计数（Reference Counting）

引用计数是最为流行的用于智能指针的所有权策略。引用计数跟踪指向同一对象的智能指针的

数目。当该数目到达零时，删除被指向的对象。如果你不破坏某些规则的话，这一策略工作得非常好——例如，你不应该同时保留指向同一对象的哑指针和智能指针。

实际的计数器必须在智能指针对象之间共享，从而产生了图 7.2 所描述的结构。除了指向对象自身的指针，每个智能指针还持有指向引用计数器的指针（图 7.2 中的 pRefCount_）。这通常会使智能指针的大小加倍；取决于你的需要和限制，这样的开销可能是、也可能不是可以接受的。

另外还有一个更微妙的开销问题。引用计数的智能指针必须在自由存储区（free store）中存储引用计数器。问题是在许多实现中，如第 4 章中所讨论的，缺省的 C++自由存储区分配非常慢，并且在分配小对象时很浪费空间（显然，引用计数通常占用 4 个字节，肯定具备小对象资格）。速度方面的开销源于查找可用内存块所用的缓慢的算法，而大小方面的开销是由分配器为每个内存块所保留的簿记信息引起的。

如图 7.3 所示，相对的大小开销可以部分地通过将指针和引用计数器放置在一起来加以降低。图 7.3 中的结构将智能指针的大小减为一个指针的大小，但却是以访问速度的损失为代价的：需要一个额外的间接层次才能对被指向对象进行访问。这是值得考虑的缺点，因为你通常需要多次使用智能指针，而你显然只要构造和销毁它一次。

最为高效的解决方案是在被指向对象里面保持引用计数器，如图 7.4 所示。这样，SmartPtr 就只有一个指针大小，而且完全没有额外的开销。该技术被称为“侵入式引用计数”，因为引用计数器对被指向者来说是一个“侵入者”——在语义上它属于智能指针。这个名称还给出了对该技术的阿喀琉斯脚踵的提示：你必须预先设计、或修改被指向者的类，以支持引用计数。

泛型智能指针应该在情况允许时使用侵入式引用计数，并且实现非侵入式引用计数方案作为合乎要求的替换。为实现非侵入式引用计数，第 4 章介绍的小对象分配器可以起到极大的作用。使用非侵入式引用计数的 SmartPtr 实现有效地利用了小对象分配器，因而极大地减少了引用计数器所带来的性能开销。

7.5.4 引用链接（Reference Linking）

引用链接建立在这样的观察之上：你并不真的需要指向某个对象的智能指针对象的实际计数；你需要的只是检测计数何时变为零。这样就引发了保持“所有权列表”的想法，如图 7.5 所示[1]

所有指向特定的被指向者的 SmartPtr 对象形成了一个双向链表。当你从已有的 SmartPtr 那里创建新的 SmartPtr 时，新对象被追加到表中；SmartPtr 的析构器负责将被销毁的对象从表中移除。当表变为空时，删除被指向对象。

双向链表结构像手套（适合手）一样适合引用跟踪。你不能使用单链表，因为在这样的表中进行移除需要线性的时间。你不能使用向量，因为 SmartPtr 对象不是连续的（而在向量中进行移除总是需要线性的时间）。你需要能表现出恒定的追加时间、恒定的移除时间，以及恒定的空检测时间的结构。双向链表正好能够独占这份清单。

在引用链接实现中，每个 SmartPtr 对象持有两个额外的指针——一个指向下一个元素，一个指向前一个元素。

相对于引用计数，引用链接的优点是无需使用额外的自由存储区，从而使得它更为可靠：创建引用链接的智能指针不会失败。缺点是引用链接需要更多的内存来用于管理（三个指针 vs. 一个指针加上一个整数）。而且，引用计数应该要更快一些——当你复制智能指针时，只需要进行一次间接和一次增值。表管理要更为费事一点。总之，只有当自由存储区不足时你才应该使用引用链接。否则的话，使用引用计数更好。

为了结关于引用管理策略的讨论，让我们注意一下它们所具有的显著的缺点。引用管理——无论是计数还是链接——都是叫做*循环引用*（cyclic reference）的资源泄漏的牺牲品。想象对象 A 持有指向对象 B 的智能指针，而对象 B 又持有指向 A 的智能指针。这两个对象形成了循环引用；即使你再也不使用它们，它们也在相互使用。引用管理策略无法检测这样的循环引用，而这两个对象

[1] 1995 年 11 月 Risto Lankinen 在 Usenet 上描述了引用链接机制。

永远也不会被释放。循环可以跨越多个对象；闭合的圆环常常揭示了未曾料想的依赖，非常难以调试。

无论如何，引用管理是一种健壮、快速的所有权处理策略。如果使用得当，引用管理可以显著地使应用开发变得更为容易。

7.5.5 销毁式复制（Destructive Copy）

销毁式复制所做的恰如你所想的那样：在复制过程中，它销毁正在被复制的对象。在智能指针的案例中，销毁式复制通过接管源智能指针所指向的对象、并将其传递给目的智能指针来销毁源智能指针。std::auto_ptr 类模板采用了销毁式复制。

除了对所采取的动作作出提示，“销毁式”还生动地描述了与这一方略相关联的危险。误用销毁式复制会对你的程序数据、你的程序正确性，以及你的脑细胞产生毁灭性的效果。

智能指针可以使用销毁式复制来确保任何时候只有一个智能指针指向给定对象。在将一个智能指针复制或赋值到另一个的过程中，“活着的”指针被传递到复制的目的地，而源智能指针的 pointee_ 会变为零。下面的代码演示了一个简单的、采用销毁式复制的 SmartPtr 的复制构造器和赋值操作符。

```
template <class T>
class SmartPtr
{
public:
  SmartPtr(SmartPtr&src)
  {
      pointee_=src.pointee_;
      src.pointee_=0;
  }

  SmartPtr&operator=(SmartPtr&src)
  {
      if (this !=&src)
      {
          delete pointee_;
          pointee_=src.pointee_;
          src.pointee_=0;
      }

      return *this;
  }

  ...
};
```

C++成规要求复制构造器和赋值操作符的右手边为对 const 对象的引用。采用销毁式复制的类为了一些明显的原因打破了这一惯例。因为成规的存在总是有理由的，如果你打破它的话你就应该预见到负面的后果。确实，这就是它们：

```
void Display(SmartPtr<Something>sp);
```

```
...
SmartPtr<Something>sp(new Something);
Display(sp);//sinks sp
```

尽管 Display 对它的参数没有恶意（是通过传值接收它的），它的行为对智能指针来说却像是一场灾难：它使所有传递给它的智能指针失效。在 Display(sp)被调用后，sp 持有的是空指针。

因为采用销毁式复制的智能指针不支持值语义，它们不能被存储在容器中，而且一般而言，对它们的处理几乎必须像对原始指针一样小心。

能够在容器中存储智能指针是非常重要的。原始指针容器使得手工的所有权管理十分棘手，有那么多指针的容器可以使用智能指针来获取好处。但是，采用销毁式赋值的智能指针却不能与容器混合使用。

而在光明的一面，采用销毁式赋值的智能指针有着显著的优点：

- 它们几乎不会带来开销。
- 它们有利于实施所有权转移语义。在这种情况下，你可以使用先前描述的“灾难效应”来获取好处：你将明确，你的函数会接管传入的指针。
- 它们可以很好地作为函数的返回值。如果智能指针实现使用了特定的技巧[2]，你可以从函数中返回采用销毁式复制的智能指针。这样，你就可以明确，如果调用者不使用返回值的话，被指向的对象会被销毁。
- 在有多条返回路径的函数中它们和栈变量一样出色。你无需记住手工删除被指向的对象——智能指针会为你照管这个。

标准所提供的 std::auto_ptr 使用了销毁式复制策略。这带给销毁式复制另一项重要的优点：

- 采用销毁式复制语义的智能指针是标准所提供的惟一一种智能指针，这意味着许多程序员迟早会习惯它们的行为。

因为这些原因，SmartPtr 实现应该对销毁式复制语义提供可选的支持。

智能指针使用各种所有权语义，每一种都有着它们自己的折衷权衡。最为重要的技术是深度复制、引用计数、引用链接，以及销毁式复制。SmartPtr 通过 Ownership 策略实现了所有这些方略，从而允许它的用户选择最适合应用需要的一种。缺省的策略是引用计数。

7.6 Address-of 操作符（The Address-of Operator）

在尽可能地使智能指针与它们的“土产”对应物变得不可分辨的斗争中，设计者被绊倒在可重载操作符列表中的一个含混的操作符上：一元的 operator&，也就是 address-of 操作符。[3]

智能指针的实现者可能会选择这样来重载 address-of 操作符：

```
template <class T>
class SmartPtr
{
public:
    T**operator&()
    {
```

[2] 由 Greg Colvin 和 Bill Gibbons 发明并用于 std::auto_ptr

[3] 一元 operator&与二元 operator&有区别，后者是按位与操作符。

```
        return &pointee_;
    }
    ...
};
```

毕竟，如果智能指针要模拟指针，那么它的地址就必须可用于替换普通指针的地址。通过这样的重载可以编写这样的代码:

```
void Fun(Widget**pWidget);
...
SmartPtr<Widget>spWidget(...);
Fun(&spWidget);   //okay,invokes operator& and obtains a
                  //pointer to pointer to Widget
```

看上去拥有这样的智能指针和哑指针的兼容性是十分合乎需要的，但是重载一元 operator&是这样一些聪明的技巧中间的一个: 它们带来的危害要多于好处。有两个理由说明重载一元 operator&不是一个非常好的主意。

一个理由是暴露所指向对象的地址意味着放弃任何自动的所有权管理。当客户自由地访问原始指针的地址时，智能指针所持有的任何助手结构，比如引用计数，会全面地失效。在客户直接处理原始指针的地址的同时，智能指针会完全“不省人事”。

第二个更为实际的理由是重载一元 operator&会使得智能指针无法与 STL 容器一起使用。实际上，为某种类型重载一元 operator&会使得你几乎无法对该类型进行泛型编程，因为一个对象的地址是过于基础性的属性，以致于不能“天真地”进行摆弄。大多数泛型代码都假定将&应用到类型 T 的对象会返回类型为 T*的对象——你瞧，address-of 是一个基础性的概念。如果你向这一概念挑战，泛型代码将会在编译时——或更糟，在运行时——表现出奇怪的行为。

因而，不推荐为智能指针或其他一般的对象重载一元 operator& SmartPtr 没有重载一元 operator&

7.7 向原始指针类型的隐式转换（Implicit Conversion to Raw Pointer Types）

考虑这样的代码:

```
void Fun(Something*p);
...
SmartPtr<Something>sp(new Something);
Fun(sp);//OK or error?
```

这样的代码应否通过编译？遵循“最大限度兼容性”的思路，答案为“是”。

技术上，通过引入用户定义的转换，要使前面的代码可编译是非常容易的:

```
template <class T>
class SmartPtr
{
public:
  operator T*()//user-de .ned conversion to T*
  {
```

```
        return pointee_;
    }
    ...
};
```

但是，这不是故事的结尾。

在 C++中，用户定义的转换有着有趣的历史。回到 1980 年代，当用户定义的转换被引入时，大多数程序员认为它们是了不起的发明。用户定义的转换允诺了更为统一的类型系统、有表现力的语义，以及定义与内建类型不可区分的新类型的能力。但是，与时间一道，用户定义的转换揭示出它们自身是难以使用的、有着潜在的危险——特别是在暴露了内部数据的句柄时（Meyers 1998a 29 则）；这正好也是先前的代码中 operator T*的情况。这就是为什么在允许对你设计的智能指针进行自动转换之前，你应该好好想一想的缘故。

给予用户不受看管地访问智能指针所包装的原始指针的权利有着固有的潜在危险。四处传递原始指针使智能指针的内部工作归于失败。一旦从它的包装者的限制中被释放出来，原始指针就会很容易地再次变成对程序健全的威胁，就好像它在智能指针被引入之前那样。

另一种危险是用户定义的转换会出乎意料地冒出来，即使你并不需要它们。考虑下面的代码:

```
SmartPtr<Something>sp;
...
//A gross semantic error
//However,it goes undetected at compile time
delete sp;
```

编译器将操作符 delete 与用户给 T*定义的转换相匹配。在运行时，operator T*被调用，并且 delete 被应用到它的结果。这肯定不是你想对智能指针做的，因为它试图自己管理所有权。一个额外的、不知不觉发生的 delete 调用会将智能指针在底层进行的所有小心的所有权管理扔出窗外去。

有相当一些方法可以防止 delete 调用被编译。其中有一些非常富有独创性（Meyers 1996）。一种非常高效而易于实现的方法是故意使对 delete 的调用变得*有歧义*（ambiguous）。你可以通过给易受 delete 调用影响的类型提供*两种*自动转换来实现这一点。一种类型是 T*自身，另一种可以是 void*

```
template <class T>
class SmartPtr
{
public:
  operator T*()//User-de .ned conversion to T*
  {
      return pointee_;
  }

  operator void*()//Added conversion to void*
  {
      return pointee_;
  }
  ...
};
```

针对这样的智能指针对象进行 delete 调用是有歧义的。编译器无法决定应用哪一种转换，而上面的技巧有效地利用了这一“优柔寡断”。

不要忘了使 delete 操作符失效只是问题的一部分。是否提供向原始指针的自动转换仍然是实现智能指针时的重要决策。让它进来过于危险，而赶它出去又过于省事。最终的 SmartPtr 实现将会提供给你一种选择。

然而，禁止隐式转换没有必要消除所有对原始指针的访问；获得这样的访问常常是有必要的。因此，通过调用下面的函数，所有的智能指针都提供了对其所包装的指针的显式访问：

```
void Fun(Something*p);
...
SmartPtr<Something>sp;
Fun(GetImpl(sp));//OK,explicit conversion always allowed
```

这并不是一个你是否能访问被包装的指针的问题；它太简单了。这看起来像是一个小区别，但实际上却非常重要。隐式转换的发生并不会通知程序员或维护者，甚至也不为他们所知。显式转换——如对 GetImpl 的调用——会经过程序员的头脑、理解以及手指，会写在代码中，任何人都可以看见它。

从智能指针类型向原始指针类型的隐式转换是合乎需要的，但有时也是危险的。SmartPtr 提供了这样的隐式转换作为选择。缺省行为则是在安全的一边——没有隐式转换。通过 GetImpl 函数，显式访问总是可用的。

7.8 相等与不等（Equality and Inequality）

C++教导它的用户，任何聪明的技巧，比如像前面的部分中介绍的“故意的歧义性”，都会确立新的语境，并可能带来料想不到的“波动”。

考虑一下智能指针的相等和不等测试。智能指针应该支持与原始指针所支持的比较语法相同的语法。程序员希望下面的测试能够编译并运行，就像他们对原始指针所做的一样：

```
SmartPtr<Something>sp1,sp2;
Something*p;
...
if (sp1)//Test 1:direct test for non-null pointer
  ...
if (!sp1)//Test 2:direct test for null pointer
  ...
if (sp1 ==0)//Test 3:explicit test for null pointer
  ...
if (sp1 ==sp2)//Test 4:comparison of two smart pointers
  ...
if (sp1 ==p)//Test 5:comparison with a raw pointer
  ...
```

如果你考虑对称和 operator!=的话，还有比这里所描述的更多的测试。如果我们解决了相等测试，我们可以很容易地定义相应的对称和不等测试。

在前一问题（阻止 delete 的编译）的解决方案和这一问题的可能的解决方案之间有着一种不凑巧的冲突。对于某种被指向的类型，在只有一个用户定义的转换时，大多数测试表达式（除了测试

4）都可以成功编译并正确运行。不好的方面是你可能会偶然地对智能指针调用 delete 操作符。在有两个用户定义的转换时（故意的歧义性），你可以检测到不正当的 delete 调用，但这些测试却再也不能够编译通过——它们也变成有歧义的了。

再加上一个用户定义的向 bool 类型的转换可以有所帮助，但是，没有人会觉得奇怪，这又会引起新的麻烦。给出这样的智能指针:

```
template <class T>
class SmartPtr
{
public:
operator bool()const
{
return pointee_!=0;
}
...
};
```

四个测试都可以编译通过，但是下面的无意义的操作也可以:

```
SmartPtr<Apple>sp1;
SmartPtr<Orange>sp2;  //Orange is unrelated to Apple
if (sp1 ==sp2)        //Converts both pointers to bool
                      //and compares results
  ...
if (sp1 !=sp2)//Ditto
  ...
bool b =sp1;          //The conversion allows this,too
if (sp1 *5 ==200)     //Ouch!SmartPtr behaves like an integral
                      //type!
  ...
```

如你所能看到的，不是一点没有就是太多了：一旦你增加了用户定义的向 bool 类型的转换，你就使得 SmartPtr 会在许多超出你设想的情况下像 bool 类型一样工作。实际上，为智能指针定义 operator bool 不是聪明的解决方案。

这一两难局面的真正、完全、像石头一样坚固的解决方案是走遍所有的路径并分别重载每一个操作符。这样，对裸指针有意义的任何操作也对智能指针有意义，就再也不会有什么问题了。这里是实现此想法的代码:

```
template <class T>
class SmartPtr
{
public:
  bool operator!()const //Enables "if (!sp)..."
  {
      return pointee_==0;
  }
```

```
  inline friend bool operator==(const SmartPtr&lhs,
      const T*rhs)
  {
      return lhs.pointee_==rhs;
  }

  inline friend bool operator==(const T*lhs,
      const SmartPtr&rhs)
  {
      return lhs ==rhs.pointee_;
  }

  inline friend bool operator!=(const SmartPtr&lhs,
      const T*rhs)
  {
      return lhs.pointee_!=rhs;
  }

  inline friend bool operator!=(const T*lhs,
      const SmartPtr&rhs)
  {
      return lhs !=rhs.pointee_;
  }
  ...
};
```

是的，这是让人痛苦的事情，但是这一方法解决了几乎所有的比较问题，包括针对直接的零的测试。在此代码中的各转发操作符所做的是将客户代码应用于智能指针的操作符应用到智能指针所包装的原始指针上。没有什么模拟能比这更现实主义了。

我们仍然没有完全解决问题。即使你提供向被指向者类型的自动转换，仍然存在发生歧义的危险。假设你有一个 Base 类和一个继承自 Base 的 Derived 类。下面的代码有着实际的意义，但由于歧义性的缘故，其形式仍然是有问题的。

```
SmartPtr<Base>sp;
Derived*p;
...
if (sp ==p){} //error!Ambiguity between:
              //'(Base*)sp ==(Base*)p'
              //and 'operator==(sp,(Base*)p)'
```

确实，智能指针开发不适合虚弱的心脏。

然而，我们的弹药还未用尽。除了定义 operator==和 operator!=，我们可以增加它们的*模板化*版本，如你在下面的代码中可看到的：

```
template <class T>
```

```
class SmartPtr
{
public:
  ...as above ...
  template <class U>
  inline friend bool operator==(const SmartPtr&lhs,
      const U*rhs)
  {
      return lhs.pointee_==rhs;
  }

  template <class U>
  inline friend bool operator==(const U*lhs,
      const SmartPtr&rhs)
  {
      return lhs ==rhs.pointee_;
  }
  ...similarly de .ned operator!=...
};
```

模板化的操作符在这样的意义上是“贪婪的”：它们无论怎样都会将比较与任何一种指针类型进行匹配，从而吃掉歧义性。

如果情况是这样，我们为何还要保留那些取被指向者类型为参数的、非模板化的操作符呢？它们决不会得到匹配的机会，因为模板匹配所有的指针类型，包括被指向者类型自身。

“决不会”实际上意味着“几乎不会”的法则在这里也适用。在测试 if(sp == 0)中，编译器会尝试下面的匹配。

- 模板化操作符。它们不匹配，因为零不是指针类型。直接的零可被隐式地转换为指针类型，但是模板匹配不包括转换。
- 非模板化操作符。在排除了模板化操作符之后，编译器尝试各个非模板化操作符。通过从直接的零隐式地转换到被指向者类型，非模板化操作符中的一个被匹配出来。如果非模板化操作符不存在的话，该测试就将是一个错误。

总之，非模板化和模板化操作符，我们*两者*都需要。

让我们看一看，如果我们比较两个通过不同的类型实例化的 SmartPtr，会发生什么。

```
SmartPtr<Apple>sp1;
SmartPtr<Orange>sp2;
if (sp1 ==sp2)
...
```

因为有歧义性，编译器会在此比较上“噎住”：两个 SmartPtr 实例化都定义了 operator==，而编译器不知道选择哪一个。我们可以通过定义“歧义清扫机”来避开这一问题，如下所示：

```
template <class T>
class SmartPtr
```

```
{
public:
  //Ambiguity buster
  template <class U>
  bool operator==(const SmartPtr<U>&rhs)const
  {
      return pointee_==rhs.pointee_;
  }

  //Similarly for operator!=
  ...
};
```

新增加的操作符是专门用于比较 SmartPtr<…>对象的成员。这个歧义清扫机的美在于它使得智能指针的比较像原始指针比较一样进行。如果你比较两个指向 Apple 和 Orange 的智能指针，代码基本上与比较两个指向 Apple 和 Orange 的原始指针是等价的。如果比较有意义的话，代码就能编译；否则，就是一个编译时错误。

```
SmartPtr<Apple>sp1;
SmartPtr<Orange>sp2;
if (sp1 ==sp2)    //Semantically equivalent to
                  //sp1.pointee_==sp2.pointee_
...
```

还剩下一个未被满足的语法制品，也就是，直接的测试 if(sp)。这里生活真的变得很有趣。If 语句只能应用于算术和指针表达式。因此，要使 if(sp)得以编译，我们必须定义向算术或指针类型的自动转换。

因为早先的 operator bool 带来的经验的见证，不推荐向算术类型的转换。指针不是算术类型，句号。向指针类型的转换有着大得多的意义，而由此问题也开始分岔。

如果你想要提供向被指向者类型的自动转换（见前面的部分），那么你有两个选择：你不是冒不小心调用操作符 delete 的危险，就是放弃 if(sp)测试。加时赛在不便和危险的生活之间进行。胜者是安全，所以你不能编写 if(sp)。相反，你可以在 if(sp != 0)和更为巴洛克式的 if(!!sp)之间进行选择。故事结束。

如果你不想提供向被指向者类型的自动转换，有一个你可以用于使 if(sp)成为可能的有趣技巧。在 SmartPtr 类模板的内部，定义一个 Tester 类，并定义向 Tester*的转换，如下面的代码所示：

```
template <class T>
class SmartPtr
{
  class Tester
  {
      void operator delete(void*);
  };

public:
  operator Tester*()const
```

```
    {
        if (!pointee_)return 0;
        static Tester test;
        return &test;
    }
    ...
};
```

现在如果你编写 if(sp)　operator Tester*开始活动。当且仅当 pointee_为空时，该操作符返回空值。Tester 自身使操作符 delete 失效，所以如果某人调用 delete sp，会发生编译时错误。有趣的是，Tester 的定义自身驻留在 SmartPrt 的私有部分中，所以客户代码不能再用它做其他事情。

SmartPtr 是这样来解决相等和不等测试的问题的：

- 以两种风格（模板化的和非模板化的）定义 operator==和 operator!=
- 定义 operator!
- 如果你允许进行向被指向者类型的自动转换，就定义另外的向 void*的转换，以故意地使对 delete 操作符的调用有歧义；否则，定义私有内部类 Tester，声明一个私有的 operator delete，并为 SmartPtr 定义向 Tester*的转换——当且仅当被指向对象为空时它才返回空指针。

7.9　次序比较（Ordering Comparisons）

次序比较操作符是 operator<　operator<=　operator>和 operator>=。你可以根据 operator<来全部实现它们。

是否允许确定智能指针的次序是其自身中的（in itself）、同时也是其自身的（of itself）一个有趣的问题，它与指针的总是让人迷惑的双重本性有关。指针是二而一的概念：迭代器（iterator）和标记（moniker）。指针的迭代本性允许你使用指针来遍历对象数组。指针运算，包括比较，支持指针的这一迭代本性。与此同时，指针也是标记——可以快速传递并即时访问对象的、并不昂贵的对象表示。解除引用操作符*和->支持标记概念。

指针的两种本性有时会让人迷惑，特别是在你只需要其中之一时。为了操作向量，迭代和解除引用，你可以两者都使用，而遍历链表或操作独立的对象，你只使用解除引用。

指针的次序比较仅在多个指针属于同一连续内存时被定义。换言之，只有对指向同一数组的指针，你才可以使用次序比较。

为智能指针定义次序比较可以说是这样一个问题：指向同一数组中的对象的智能指针有意义吗？表面上，答案是否。智能指针的主要特性是管理对象所有权，而有着不同的所有权的对象通常并不属于同一数组。因此，允许用户进行无意义的比较是危险的。

如果你真的需要次序比较，你总是可以使用对原始指针的显式访问。这里的问题同样是找到在大多数情况下、最安全和最有表现力的行为。

前面的部分总结说，向原始指针的隐式转换是可选的。如果 SmartPtr 的客户选择允许进行隐式转换，下面的代码就可以编译通过：

```
SmartPtr<Something>sp1,sp2;
if (sp1 <sp2)     //Converts sp1 and sp2 to raw pointer type,
                  //then performs the comparison
...
```

这意味着如果我们想要使次序比较失效，我们必须采取主动，明确地禁止它们。做到这一点的一种

方法是声明它们，但却不定义它们；这意味着对它们的任何使用都将触发链接时错误。

```
template <class T>
class SmartPtr
{...};
template <class T,class U>
bool operator<(const SmartPtr<T>&,const U&);//Not defined
template <class T,class U>
bool operator<(const T&,const SmartPtr<U>&);//Not defined
```

但是，相对于不定义它们，更聪明的办法是根据 operator<来定义所有其他的操作符。这样，如果 SmartPtr 的用户认为最好引入智能指针定序，他们只需要定义 operator<就可以了。

```
//Ambiguity buster
template <class T,class U>
bool operator<(const SmartPtr<T>&lhs,const SmartPtr<U>&rhs)
{
  return lhs <GetImpl(rhs);
}
//All other operators
template <class T,class U>
bool operator>(SmartPtr<T>&lhs,const U&rhs)
{
  return rhs <lhs;
}
...similarly for the other operators ...
```

注意歧义清扫机的再次出现。现在如果有些库用户认为 SmartPtr<Widget>应该被排序，下面的代码就是技巧所在：

```
inline bool operator<(const SmartPtr<Widget>&lhs,
const Widget*rhs)
{
  return GetImpl(lhs)<rhs;
}
inline bool operator<(const Widget*lhs,
const SmartPtr<Widget>&rhs)
{
  return lhs <GetImpl(rhs);
}
```

遗憾的是用户必须定义两个操作符，而不是一个，但是比起定义八个来，这要好得多了。

由此可以推断出定序的问题是它的细节并不有趣。有时确定多个位置任意的对象的次序是非常有用的。例如，你可能需要对辅助的、各个对象自己的信息进行存储，而且你需要快速地访问该信息。对于这样的任务，由对象的地址确定次序的映射是十分有效的。

标准的 C++有助于实现这样的设计。尽管没有定义位置任意的对象的指针比较，对于任何两个

同一类型的指针，标准确保 std::less 产生的结果是有意义的。因为标准的关联容器(associative container)使用 std::less 作为缺省的次序关系，你可以安全地使用用指针作为关键字的映射。

SmartPtr 也应该支持这一习语；因此，SmartPtr　std::less 进行了专门化，简单地将调用转发给用于普通指针的 std::less

```
namespace std
{
  template <class T>
  struct less<SmartPtr<T>>
      :public binary_function<SmartPtr<T>,SmartPtr<T>,bool>
  {
      bool operator()(const SmartPtr<T>&lhs,
          const SmartPtr<T>&rhs)const
      {
          return less<T*>()(GetImpl(lhs),GetImpl(rhs));
      }
  };
}
```

总而言之，SmartPtr 在缺省情况下没有定义定序操作符。它声明了——但没有实现——两个泛型 operator<，并根据 operator<实现了所有其他的定序操作符。用户可以定义 operator<的专门的或泛型的版本。

SmartPtr　std::less 进行了专门化，以提供任意的智能指针的次序。

7.10 检查和错误报告（Checking and Error Reporting）

从智能指针那里，应用需要不同程度的安全性。有些程序是计算密集型的，必须对速度进行优化，而另外一些（实际上，是大多数）是输入/输出密集型的，允许进行更好的运行时检查，而又不会降低性能。

大多数时候，在应用中你可能正好需要两种模型：某些关键区域中的低安全性/高速度，以及在别处的高安全性/较低的速度。

我们可以将智能指针的检查问题划分为两个范畴：初始化检查和解除引用之前的检查。

7.10.1 初始化检查（Initialization Checking）

智能指针应该接受空（零）值吗?

保证智能指针不能为空是容易的，在实践中也可能非常有用。这意味着任何智能指针都总是有效的（除非你通过使用 GetImplRef 来瞎摆弄原始指针）。通过在传入空指针时扔出异常的构造器的帮助，很容易实现这样的想法。

```
template <class T>
class SmartPtr
{
public:
  SmartPtr(T*p):pointee_(p)
  {
```

```
        if (!p)throw NullPointerException();
    }
    ...
};
```

另一方面，空值又是方便的“不是有效指针”占位符，这常常也是有用的。

是否允许空值还影响到缺省构造器。如果智能指针不允许空值，那么缺省构造器怎样去初始化原始指针呢？缺省构造器可以缺席，但那将使得指针更难处理。例如，你有一个 SmartPtr 成员变量，但在构造时却没有适当的初始化值，你应该怎么办？总之，定制初始化涉及到提供适当的缺省值。

7.10.2 解除引用之前的检查（Checking Before Dereference）

因为对空指针解除引用造成不确定的后果，在解除引用之前进行检查是很重要的。对于许多应用来说，不确定的行为是不可接受的，所以在解除引用之前检查指针的有效性是正确的方法。在解除引用之前的检查归属于 SmartPtr 的 operator->和一元 operator*

与初始化检查相反，解除引用之前的检查可能成为你的应用中的主要性能瓶颈，因为比起创建智能指针对象，通常应用使用（解除其引用）智能指针要频繁得多。因此，你应该在安全和速度之间保持平衡。一个好的经验法则是从严格检查过的指针开始，并在剖析显示出了相关需要时从所选择的智能指针那里移除检查。

初始化检查和解除引用之前的检查可以在概念上进行分离吗？不行，因为在它们之间存在着关联。如果你在初始化时实施严格的检查，那么解除引用之前的检查就变得多余了，因为指针总是有效的。

7.10.3 错误报告（Error Reporting）

惟一明智的报告错误的方式是扔出异常。

你可以为避免错误而做某些事情。例如，如果指针在解除引用时是空的，你可以在运行中对其进行初始化。这是有效、有价值的策略，叫做*懒情初始化*（lazy initialization）——你可以仅仅在你第一次需要构建其值。

如果你只想在调试过程中进行检查，你可以使用标准的 assert 或类似的、更为成熟的宏。编译器会在发布模式中忽略测试，所以，假定你在调试过程中消除了所有的空指针错误，你就收获了检查和速度这两种好处。

SmartPtr 将检查移往专门的 Checking 策略。该策略实现检查函数（它们可以提供可选的懒情初始化）和错误报告方略。

7.11 指向 const 的智能指针和 const 智能指针（Smart Pointers to const and const Smart Pointers）

原始指针提供两种恒常性（constness）：所指向对象的恒常性和指针自身的恒常性。下面是对这两种属性的演示：

```
const Something*pc =new Something;//points to const object
pc->ConstMemberFunction();//ok
pc->NonConstMemberFunction();//error
delete pc;//ok (surprisingly)[4]
```

[4] 有时候，“为什么你可以对指向 const 的指针应用 delete 操作符？”会在 comp.std.c++新闻组引发一场激烈的辩论。事实是，无论好坏，语言允许这样的语义。

```
Something*const cp =new Something;//const pointer
cp->NonConstMemberFunction();//ok
cp =new Something;//error,can't assign to const pointer
const Something*const cpc =new Something;//const,points to const
cpc->ConstMemberFunction();//ok
cpc->NonConstMemberFunction();//error
cpc =new Something;//error,can't assign to const pointer
```

相应的 SmartPtr 用法看起来是这样:

```
//Smart pointer to const object
SmartPtr<const Something>spc(new Something);
//const smart pointer
const SmartPtr<Something>scp(new Something);
//const smart pointer to const object
const SmartPtr<const Something>scpc(new Something);
```

SmartPtr 类模板可以检测所指向对象的恒常性，无论是通过部分专门化，还是通过使用第 2 章中定义的 TypeTraits 模板。后一种方法更为可取，因为它不会像部分专门化那样带来源代码的重复。

Smart 模拟了指向 const 对象的指针、const 指针，以及相应的两者的结合。

7.12 数组（Arrays）

在大多数情况下，除了处理堆分配的数组和使用 new[]及 delete[]，你最好使用 std::vector。标准的 std::vector 类模板提供了动态分配数组所提供的一切，以及其他许多东西。所带来的额外开销在大多数情况下都是微不足道的。

但是，“大多数情况”并不是“总是”。有许多情形你不需要也不想要功能完备的向量；动态分配的数组就完全足够了。在这样的情况下不能利用智能指针是不方便的。在成熟的 std::vector 和动态分配的数组之间肯定存在着裂缝。如果用户需要的话，智能指针可以通过提供数组语义来填补这道裂缝。

从指向数组的智能指针的视点来看，惟一重大的问题是在它的析构器中调用 delete[] pointee_ 而不是 delete pointee_。这个问题已经由 Ownership 策略处理了。

第二个问题是通过为智能指针重载 operator[]来提供索引化访问。这在技术上是可行的；实际上，SmartPtr 的初始版本的确提供了单独的用于可选的数组语义的策略。但是，仅在非常罕见的情况下智能指针才指向数组。在那样的情况下，如果你使用 GetImpl 的话，已经有了提供索引化访问的途径:

```
SmartPtr<Widget>sp =...;
//Access the sixth element pointed to by sp
Widget&obj =GetImpl(sp)[5 ];
```

看起来，以引入新策略的代价来努力提供额外的语法方便是一项糟糕的决定。

SmartPtr 通过 Ownership 策略来支持定制的析构。因此你可以通过 delete[]来进行数组特定的析构。但是，SmartPtr 不提供指针运算。

7.13 智能指针和多线程（Smart Pointers and Multithreading）

智能指针往往会用于共享对象。多线程问题会影响到对象共享。因此，多线程问题也会影响智能指针。

智能指针和多线程之间的交互发生在两个层面上。一个是被指向对象级，另一个是簿记数据级。

7.13.1 在被指向对象级的多线程（Multithreading at the Pointee Object Level）

如果多个线程访问同一对象，而你是通过智能指针来访问该对象的，那么在通过 operato->调用函数的过程中锁定对象可能是合乎需要的。通过让智能指针返回代理对象、而不是原始指针，可以做到这一点。代理对象的构造器锁定被指向对象，而它的析构器对其进行解锁。在 Stroustrup（2000）中阐述了这一技术。在这里提供了演示这一方法的代码。

首先，让我们考虑类 Widget，它有两个锁定原语：Lock 和 Unlock。在调用 Lock 后，你可以安全地访问对象。任何其他调用 Lock 的线程都会阻塞。调用 Unlock 使得其他线程能够锁定此对象。

```
class Widget
{
  ...
  void Lock();
  void Unlock();
};
```

其次，我们定义类模板 LockingProxy。它的任务是在其生存期内锁住对象（使用 Lock/Unlock 方法）。

```
template <class T>
class LockingProxy
{
public:
  LockingProxy(T*pObj):pointee_(pObj)
  {pointee_->Lock();}
  ~LockingProxy()
  {pointee_->Unlock();}
  T*operator->()const
  {return pointee_;}
private:
  LockingProxy&operator=(const LockingProxy&);
  T*pointee_;
};
```

除了构造器和析构器，LockingProxy 还定义了 operator->，返回指向被指向对象的指针。

尽管 LockingProxy 看起来有点像是智能指针，它的层次要更多一层——SmartPtr 类模板自身。

```
template <class T>
class SmartPtr
{
  ...
```

```
  LockingProxy<T>operator->()const
  {return LockingProxy<T>(pointee_);}
private:
  T*pointee_;
};
```

回忆 7.3 部分，在其中解释了 operator->的机制，即编译器可以将 operator->多次应用于一个->表达式，直到获得“土产”指针为止。现在设想你发出下面的调用（假设 Widget 定义了函数 DoSomething）：

```
SmartPtr<Widget>sp =...;
sp->DoSomething();
```

其技巧是这样的：SmartPtr 的 operator->返回临时的 LockingProxy<T>对象。编译器继续应用 operator-> LockingProxy<T>的 operator->返回 Widget*。编译器使用这个指向 Widget 的指针来发出对 DoSomething 的调用。在调用过程中，临时对象 LockingProxy<T>是活着的，并锁住了对象；这意味着对象是被安全地锁定的。一旦对 DoSomething 的调用返回，临时的 LockingProxy<T>对象被销毁，于是 Widget 对象被解锁。

自动的锁定是智能指针分层的良好应用。你可以通过改变 Storage 策略来对智能指针进行分层。

7.13.2 在簿记数据级的多线程（Multithreading at the Bookkeeping Data Level）

有时除了被指向对象，智能指针还操作各种数据。如你在 7.5 部分读到的，引用计数智能指针在底层共享某些数据——也就是引用计数。如果你从一个线程将引用计数智能指针复制到另一个线程，你最后就会拥有两个指向同一引用计数器的智能指针。当然，它们也指向同样的被指向对象，但那对用户来说是可访问的，用户可以锁定它。相反，引用计数对用户来说是不可访问的，所以对它进行管理完全是智能指针的责任。

不仅仅是引用计数指针面临着与多线程有关的危险。引用跟踪智能指针（7.5.4 部分）在内部持有互指的指针，它们也是共享数据。引用连接产生出智能指针的“共同体”，它们并不必然都属于同一线程。因此，每次你复制、赋值或销毁引用跟踪智能指针，你都必须进行适当的锁定；否则的话，双向链表可能会被破坏。

总之，多线程问题绝对会影响智能指针的实现。让我们看一下怎样来解决引用计数和引用链接中的多线程问题。

7.13.2.1 多线程化引用计数（Multithreaded Reference Counting）

如果你在线程间复制智能指针，你最后会从不同的线程增加引用计数，其次数无法预知。

如附录所解释的，增值操作不是原子操作。要在多线程化环境中增加或减少整形值，你必须使用 ThreadingModel<T>::IntType 和 AtomicIncrement 和 AtomicDecrement 函数。

这里事情变得有点棘手。或者最好说，如果你想要使引用计数与线程分离，事情就会变得棘手。

基于策略的类设计指示你将类分解为基本的行为元素，并将各个元素设计成单独的模板参数。在理想的世界里，SmartPtr 会指定 Ownership 策略和 ThreadingModel 策略，并使用两者来进行正确的实现。

但是，在多线程化引用计数的情形下，事情过于紧密地绞在一起。例如，计数器必须是 ThreadingModel<T>::IntType 类型的。于是，你必须使用 AtomicIncrement 和 AtomicDecrement，而不是使用 operator++和 operator--。线程和引用计数融合在一起；要分开它们难以说明地困难。

最好是在 Ownership 策略中结合多线程。于是你可以有两个实现：RefCounting 和 RefCountingMT

7.13.1.2 多线程化引用链接（Multithreaded Reference Linking）

考虑引用链接智能指针的析构器。它看起来很可能是这样:

```
template <class T>
class SmartPtr
{
public:
  ~SmartPtr()
  {
      if (prev_==next_)
      {
          delete pointee_;
      }
      else
      {
          prev_->next_=next_;
          next_->prev_=prev_;
      }
  }
  ...
private:
  T*pointee_;
  SmartPtr*prev_;
  SmartPtr*next_;
};
```

析构器中的代码执行经典的双向链表删除。为使实现更简单快速，链表是循环的——最后的节点指向第一个节点。这样对于任何智能指针，我们都不必测试 prev_和 next_是否为零。只有一个元素的循环链表，其 prev_和 next_都等于 this

如果多个线程对互相链接的智能指针进行销毁，析构器显然必须是原子的（不能被其他线程中断）。否则的话，另外的线程可以中断 SmartPtr 的析构器——例如，在更新 prev_->next_和更新 next_->prev_之间。随后那个线程将在已被破坏的表上工作。

类似的原因也适用于 SmartPtr 的复制构造器和赋值操作符。这些函数必须是原子的，因为它们对所有权表进行了操作。

很有意思的是，我们不能在这里应用对象级锁定语义。附录将锁定策略划分为*类级*（class-level）和*对象级*(object-level)方略。类级锁定操作在操作过程中锁定给定类的所有对象。对象级锁定操作只锁定从属于此操作的对象。前面的技术所占用的内存较少（每个类只有一个互斥体），但是容易变成性能瓶颈。后面的技术更“重”（每个对象一个互斥体），但可能会更快。

我们不能将对象级锁定应用于智能指针，因为操作最多会涉及到三个对象：当前的正被增加或移除的对象、前面的对象，以及所有权表中的下一个对象。

如果我们想要引入对象级锁定，起初的印象是每个被指向对象必须有一个互斥体——因为每个被指向对象有一个表。我们可以为每个对象动态地分配互斥体，但是这取消了引用链接相对于引用

计数的优点。仅仅因为引用链接不使用自由存储区，它才是更为吸引人的。

另外，我们也可以使用侵入式方法：被指向对象持有互斥体，而智能指针操作该互斥体。但是另外一种稳固的、有效的方法——引用计数式智能指针——消除了提供这一特性的动机。

总之，多线程问题会影响使用引用计数或引用链接的智能指针。线程安全的引用计数需要整数原子操作。线程安全的引用链接需要互斥体。SmartPtr 只提供了线程安全的引用计数。

7.14 大融合（Putting It All Together）

不用再走多远了！这里我们到达了有趣的部分。到目前为止我们是单独处理各个问题的。现在是时候把所有的决定集中进单一的 SmartPtr 实现中了。

我们将要使用的策略是第 1 章中所描述的：基于策略的类设计。没有惟一的解决方案的各个设计方面被转换为策略。SmartPtr 类模板将各个策略作为单独的模板参数进行接收。SmartPtr 继承所有这些模板参数，允许相应的策略对状态进行存储。

让我们通过列举 SmartPtr 的变化点来扼要地重述前面的部分。每种变化点都被转译为一种策略

- *Storage 策略（7.3 部分）*。缺省所存储的类型是 T*（T 是 SmartPtr 的第一个模板参数），指针类型也是 T*，而引用类型是 T&。销毁被指向对象的手段是 delete 操作符。
- *Ownership 策略（7.5 部分）*。深度复制、引用计数、引用链接，以及销毁式复制是流行的实现。注意 Ownership 与析构自身的机制无关；这是 Storage 的任务。Ownership 控制销毁的*时刻*
- *Conversion 策略（7.7 部分）*。有些应用需要向底层的原始指针类型的自动转换；有些不需要。
- *Checking 策略（7.10 部分）*。这一策略控制 SmartPtr 的初始化值的有效性，以及 SmartPtr 是否可以进行有效的解除引用。

其他一些问题无需用专门的策略来处理它们，或是有着最为适宜的解决方案：

- address-of 操作符（7.6 部分）最好不要重载。
- 相等和不等测试是通过 7.8 部分所示的技巧处理的。
- 次序比较（7.9 部分）没有实现；但是，Loki 为 SmartPtr 对象专门化了 std::less。用户可以定义 operator<，而 Loki 可以根据 operator<定义所有其他的次序比较。
- Loki 为 SmartPtr 对象和/或被指向对象定义了能正确处理 const 的实现。
- 对数组没有特别的支持，但是已“罐装”的 Storage 实现之一可以通过使用 operator delete[]来处理数组。

对围绕智能指针的设计问题所作的陈述使得这些问题更易于理解和可管理，因为各个问题是分别讨论的。于是，如果实现能够对问题各个击破、并分别进行处理，而不是同时与所有的复杂性作斗争，将会是非常有益的。

即使是在今天，在处理智能指针时，尤利乌斯．凯撒所铸造的古老格言“分而治之” Divide et Impera 仍然是有用的（我敢打赌他并没有预见到这一点）。我们将问题划分进小的组件类中，称为*策略*。每种策略只处理一种问题。SmartPtr 继承了所有这些类，因而就继承了它们所有的特性。如你很快就会看到的，它是那样的简单——又让人难以置信地灵活。每个策略也是一个模板参数，这意味着你可以混合和搭配已有的库存策略类，或是构建你自己的策略类。

首先是被指向类型，接着是其他的各个策略。下面是我们所得到的 SmartPtr 的声明：

```
template
<
  typename T,
```

```
  template <class>class OwnershipPolicy =RefCounted,
  class ConversionPolicy =DisallowConversion,
  template <class>class CheckingPolicy =AssertCheck,
  template <class>class StoragePolicy =DefaultSPStorage
>
class SmartPtr;
```

策略在 SmartPtr 的声明中出现的次序是按照策略定制的频度从高到低排列的。

下面的四小节讨论我们所定义的四个策略的需求。一条适用于所有策略的规则是它们必须具有值语义；也就是说，它们必须定义适当的复制构造器和赋值操作符。

7.14.1 Storage 策略（The Storage Policy）

Storage 策略抽象了智能指针的结构。它提供类型定义并存储实际的 pointee_对象。

如果 StorageImple 是 Storage 策略的实现，而 storageImpl 是 StorageImpl<T>类型的对象，那么表 7.1 中的构造就是适用的。

下面是缺省的 Storage 策略实现:

```
template <class T>
class DefaultSPStorage
{
protected:
  typedef T*StoredType;//the type of the pointee_object
  typedef T*PointerType;//type returned by operator->
  typedef T&ReferenceType;//type returned by operator*
public:
  DefaultSPStorage():pointee_(Default())
  {}
  DefaultSPStorage(const StoredType&p):pointee_(p){}
  PointerType operator->()const {return pointee_;}
  ReferenceType operator*()const {return *pointee_;}
  friend inline PointerType GetImpl(const DefaultSPStorage&sp)
  {return sp.pointee_;}
  friend inline const StoredType&GetImplRef(
      const DefaultSPStorage&sp)
  {return sp.pointee_;}
  friend inline StoredType&GetImplRef(DefaultSPStorage&sp)
  {return sp.pointee_;}
protected:
  void Destroy()
  {delete pointee_;}
  static StoredType Default()
  {return 0;}
private:
  StoredType pointee_;
};
```

除了 DefaultSPStorage　Loki 还定义了：

- ArrayStorage，它在 Release 中使用了 operator delete[]
- LockedStorage，它使用分层来提供在解除引用时对数据进行锁定的智能指针（见 7.13.1 部分）
- HeapStorage，它使用显式析构器调用（std::free 跟随其后）来释放数据

表 7.1　Storage 策略构造

表达式	语义
StorageImpl<T>::StoredType	由实现实际存储的类型。缺省：T*
StorageImple<T>::PointerType	由实现定义的指针类型。这是 SmartPtr 的 operator->所返回的类型。缺省：T*。在你使用智能指针分层时，可与 StorageImpl<T>::StoredType 不同(见 7.3 和 7.13.1 部分）。
StorageImpl<T>::ReferenceType	引用类型。这是 SmartPtr 的 operator*所返回的类型。缺省：T&
GetImpl(storageImpl)	返回 StorageImpl<T>::StoredType 类型的对象。
GetImplRef(storageImpl)	返回 StorageImpl<T>::StoredType&类型的对象，如果 storageImple 是 const 的话，则限定有 const
storageImpl.operator->()	返回 StorageImpl<T>PointerType 类型的对象。用于 SmartPtr 自己的 operator->
storageImpl.operator*()	返回 StorageImpl<T>::ReferenceType 类型的对象。用于 SmartPtr 自己的 operator*
StorageImpl<T>::StoredType p; p =storageImpl.Default();	返回缺省值（通常为零）。
storageImpl.Destroy()	销毁被指向对象。

7.14.2　Ownership 策略（The Ownership Policy）

Ownership 策略必须支持侵入式以及非侵入式的引用计数。因此，和 Koenig（1996）所做的一样，它使用了显式的函数调用，而不是构造器/析构器技术。其原因是你可以在任何时候调用成员函数，而构造器和析构器则只在特定的时候被自动调用。

Ownership 策略实现需要一个模板参数，即相应的指针类型。SmartPtr 传递 StoragePolicy<T>::PointerType 给 OwnershipPolicy。注意 OwnershipPolicy 的模板参数是指针类型，而不是对象类型。

如果 OwnershipImpl 是 Ownership 的实现，而 ownershipImpl 是 OwnershipImpl<P>的对象，那么表 7.2 中的构造就是适用的。

表 7.2　Ownership 策略构造

表达式	语义
P val1 P val2 =OwnershipImplImpl; Clone(val1);	克隆对象。它可以修改源始值，如果 OwnershipImpl 使用销毁式复制的话。

Const P val1 P val2 =ownershipImpl; Clone(val1);	克隆对象。
P val; bool unique =ownershipImpl; Release(val);	释放对某个对象的所有权。如果对此对象的最后的引用已被释放，返回真。
bool dc =OwnershipImpl<P>::destructiveCopy;	规定 OwnershipImpl 是否使用销毁式复制。如果是的话，SmartPtr 使用已被用于 std::auto_ptr 中的 Colvin/Gibbons 技巧（Meyers 1999）。

支持引用计数的 Ownership 实现如下所示:

```
template <class P>
class RefCounted
{
  unsigned int*pCount_;
protected:
  RefCounted():pCount_(new unsigned int(1)){}
  P Clone(const P &val)
  {
      ++*pCount_;
      return val;
  }
  bool Release(const P&)
  {
      if (!--*pCount_)
      {
          delete pCount_;
          return true;
      }
      return false;
  }
  enum {destructiveCopy =false };//see below
};
```

为其他的引用计数方案实现策略是非常容易的。让我们为 COM 对象编写一个 Ownership 策略。COM 对象拥有两个函数：AddRef 和 Release。发生最后一个 Release 调用时，对象会销毁自身。你只需要直接针对 AddRef 定义 Clone，针对 COM 的 Release 定义 Release 就可以了:

```
template <class P>
class COMRefCounted
{
public:
  static P Clone(const P&val)
  {
      val->AddRef();
```

```
        return val;
    }
    static bool Release(const P&val)
    {
        val->Release();
        return false;
    }
    enum {destructiveCopy =false };//see below
};
```

Loki 定义了下列 Ownership 实现:

- DeepCopy，在 7.5.1 中描述。DeepCopy 假定被指向类实现了成员函数 Clone
- RefCounted，在 7.5.3 及这一部分中描述。
- RefCountedMT　RefCounted 的多线程版。
- COMRefCounted　在这一部分中描述的侵入式引用计数的变种。
- RefLinked，在 7.5.4 部分中描述。
- DestructiveCopy，在 7.5.5 部分中描述。
- NoCopy，没有定义 Clone，从而取消了任何形式的复制。

7.14.3　Conversion 策略（The Conversion Policy）

Conversion 是一种简单的策略：它定义了一个布尔类型的编译时常数，说明 SmartPtr 是否允许进行向底层指针类型的隐式转换。

如果 ConversionImpl 是 Conversion 的实现，那么表 7.3 中的构造就是适用的。

SmartPtr 底层的指针类型由它的 Storage 策略规定，其类型为 StorageImpl<T>::PointerType

如你可能会预期的，Loki 正好定义了两种 Conversion 实现:

- AllowConversion
- DisallowConversion

表 7.3　Conversion 策略构造

表达式	语义
bool allowConv =ConversionImpl<P>::allow;	如果 allow 为真，SmartPtr 允许进行向其底层指针类型的显式转换。

表 7.4　Checking 策略构造

表达式	语义
S value; checkingImpl.OnDefault(value)	SmartPtr 在缺省构造器调用中调用 OnDefault。如果 CheckingImpl 没有定义此函数，它就使缺省构造器在编译时失效。
S value; checkingImpl.OnInit(value);	SmartPtr 在进行构造器调用时调用 OnInit
S value;	SmartPtr 在从 operator->和 operator*返回之前调

checkingImpl.OnDereference(value);	用 OnDereference
const S value; checkingImpl.OnDereference(value);	SmartPtr 在从 operator->和 operator*的 const 版本返回之前调用 OnDereference

7.14.4 Checking 策略（The Checking Policy）

如 7.10 部分所讨论的，检查 SmartPtr 对象的一致性主要在两处地方：在初始化过程中和在解除引用之前。检查自身可以使用 assert、异常，或懒惰初始化，或什么也不做。

Checking 策略是在 Storage 策略的 StoredType、而不是 PointerType 上进行操作（Storage 的定义见 7.14.1 部分）

如果 S 是 Storage 策略实现所定义的被存储类型，且 CheckingImpl 是 Checking 的实现，且 checkingImpl 是 CheckingImpl<S>类型的对象，那么表 7.4 中的构造就是适用的。

Loki 定义了 Checking 的下列实现:

- AssertCheck，它使用 assert 来在解除引用之前对值进行检查。
- AssertCheckStrict，它使用 assert 来在初始化时对值进行检查。
- RejectNullStatic，它没有定义 OnDefault。因此，任何对 SmartPtr 的缺省构造器的使用都会产生编译时错误。
- RejectNull，它在你试图对空指针解除引用时扔出异常。
- RejectNullStrict，它不接受用空指针来作初始化（同样，也扔出异常）。
- NoCheck，它沿袭了伟大的 C 和 C++传统的处理错误的方式——也就是，根本不进行检查。

7.15 总结（Summary）

祝贺你！你刚刚读完了本书中最长、最疯狂的一章——我们希望你的努力已经得到了回报。现在你知道了许多关于智能指针的事情，并且装备了相当全面的、可配置的 SmartPtr 类模板。

智能指针在语法和语义上模拟内建指针。此外，它们还执行许多内建指针无法完成的任务。这些任务可以包括所有权管理和针对无效指针的检查。

智能指针的概念超出了实际的指针行为；它们可被推广进智能资源中，比如标记（moniker 没有指针语法的句柄，而通过它们进行资源访问的方式又与指针行为类似）。

智能指针是成功、健壮的应用的一项基本要素，因为它们能够很好地自动完成手工非常难以管理的事情。尽管它们那么小，它们可以影响项目的成败——或者，更为常见的，成为程序是正确、还是像筛子一样泄漏资源的关键。

那也是为什么智能指针实现者应该在此项任务中尽可能地多加注意和努力的原因；所作的投入很可能会得到长期的回报。类似地，智能指针用户应该理解智能指针所确立的约定，并根据这些约定来使用它们。

这里所介绍的智能指针的实现的着重点是将各功能域分解为独立的、由主类模板 SmartPtr 进行混合和搭配的策略。这之所以可能，是因为每个策略都实现了定义良好的接口。

7.16 SmartPtr 速写（SmartPtr Quick Facts）

- SmartPtr 声明

```
template
<
  typename T,
```

```
    template <class>class OwnershipPolicy =RefCounted,
    class ConversionPolicy =DisallowConversion,
    template <class>class CheckingPolicy =AssertCheck,
    template <class>class StoragePolicy =DefaultSPStorage
>
class SmartPtr;
```

- T 是 SmartPtr 所指向的类型。T 可以是基本类型或用户定义的类型。void 类型是允许的。
- 对于剩下的类模板参数（OwnershipPolicy ConversionPolicy CheckingPolicy，以及 StoragePolicy），你可以实现你自己的策略或从 7.14.1 到 7.14.4 部分中所提到的缺省策略中选择。
- OwnershipPolicy 控制所有权管理策略。你可以从 7.14.2 部分中描述的以下预定义类中选择: DeepCopy RefCounted RefCountedMT COMRefCounted RefLinked DestructiveCopy 及 NoCopy
- ConversionPolicy 控制是否允许进行向被指向类型的隐式转换。缺省值是禁止隐式转换。无论哪种方式，你都仍然可以通过调用 GetImpl 来访问被指向对象。你可以使用 AllowConversion 和 DisallowConversion 实现（7.14.3 部分）。
- CheckingPolicy 定义错误检查策略。所提供的缺省策略是 AssertCheck AssertCheckStrict RejectNullStatic RejectNull RejectNullStrict，以及 NoCheck（7.14.4 部分）。
- StoragePolicy 定义怎样存储和访问被指向对象的细节。缺省策略是 DefaultSPStorage，它在通过类型 T 实例化时，将引用类型定义为 T&，将存储类型定义为 T*，将从 operator->返回的类型也定义为 T*。其他由 Loki 定义的存储类型是 ArrayStorage LockedStorage，以及 HeapStorage（7.14.1）。