# 智能指针的标准之争：Boost vs. Loki

**撰文/马维达 weida@flyingdonkey.com**

2001 年 10 月和 2002 年 4 月，在美国的华盛顿和荷兰的安的列斯群岛上分别召开了两次 C++标准会议。会议的内容之一是对一项新的 C++特性提议——智能指针（Smart Pointer）——进行讨论。本文将对可能成为 C++新标准的两种智能指针方案（Boost vs. Loki）进行介绍和分析，并给出了相应的使用实例。

**关键词：智能指针　C++　Boost　Loki**

在现在的标准 C++中，只有一种智能指针：std::auto_ptr。其原因并非是因为 auto_ptr 已足以应付所有相关的工作——实际上，auto_ptr 有一个重大的缺陷，就是它不能被用在 STL 容器中——而是因为现在的 C++标准在制定时并未能对智能指针进行全面的考察。按照 C++标准委员会成员 Herb Sutter 的说法，只有一种标准的智能指针是一件“可羞”的事情：首先，智能指针所能做的许多有用的事情，是可怜的 auto_ptr 不能完成的；其次，在有些情况下使用 auto_ptr 可能会造成问题，上面所说的不能在容器中使用就是一例。实际上，许多程序员已经开发了各种有用的智能指针，有些甚至在 auto_ptr 被定为标准之前就已存在，但问题是，它们不是标准的。在这样的情况下，C++标准委员会考虑引入新的智能指针，也就是自然而然的事情了。目前进入委员会视野的，主要有两种智能指针方案：Boost 智能指针和 Loki 智能指针。前者是由 C++标准委员会库工作组发起的 Boost 组织开发的，而后者由世界级的 C++专家 Andrei Alexandrescu 开发，并在他所著的“Modern C++ Design”一书中进行了详细的阐释。下面，让我们分别来看一看这两种方案各自的技术特点。

## 一、 Boost 智能指针

Boost 的智能指针方案实现了五种智能指针模板类，每种智能指针都用于不同的目的。这五种智能指针是：

```
template<typename T> class scoped_ptr;
template<typename T> class scoped_array;
template<typename T> class shared_ptr;
template<typename T> class shared_array;
template<typename T> class weak_ptr;
```

下面将分别介绍它们各自的特性，并给出相应的使用实例：

- **scoped_ptr**：意在用作指向自动（栈）对象的、不可复制的智能指针。该模板类存储的是指向动态分配的对象（通过 new 分配）的指针。被指向的对象保证会被删除，或是在 scoped_ptr 析构时，或是通过显式地调用 reset 方法。注意该模板没有“共享所有权”或是“所有权转让”语义。同时，它也是不可复制的（noncopyable）。正因为如此，在用于不应被复制的指针时，它比 shared_ptr 或 std:auto_ptr 要更安全。与 auto_ptr 一样，scoped_ptr 也不能用于 STL 容器中；要满足这样的需求，应该使用 shared_ptr。另外，它也不能用于存储指向动态分配的数组的指针，这样的情况应使用 scoped_array。

下面是使用 scoped_ptr 的一个简单实例：

```
class CTest
{
public:
 CTest() : m_id(0) {}
 CTest(int id) : m_id(id) {}
 ~CTest() { std::cout << "id: " << m_id << " - Destructor is being called\n"; }
```

```
 void SetId(int id) { m_id = id; }
 int GetId() { return m_id; }
 void DoSomething()
      { std::cout << "id: " << m_id << " - Doing something\n"; }

private:
 int m_id;
};

void main()
{
 boost::scoped_ptr<CTest> pTest(new CTest);
 pTest->DoSomething();
}
```

其运行结果为：

```
id: 0 - Doing something
id: 0 - Destructor is being called
```

（以下的几个例子所用的 CTest 类的定义完全相同，为节省篇幅，不再列出——作者）

显然，尽管我们自己没有调用 delete，pTest 仍然为我们正确地删除了它所指向的对象。看起来 scoped_ptr 的用途和 auto_ptr 十分类似，但实际上，scoped_ptr 类型的指针的所有权不可转让，这一点是和 auto_ptr 相当不同的。

- **scoped_array**：该模板类与 scoped_ptr 类似，但意在用于数组而不是单个对象。std::vector 可用于替换 scoped_array，并且远为灵活，但其效率要低一点。在不使用动态分配时，boost::array 也可用于替换 scoped_array。

下面是一个使用 scoped_array 的实例：

```
void main()
{
 boost::scoped_array<CTest> pTest(new CTest[2]);

 pTest[0].SetId(0);
 pTest[1].SetId(1);
 pTest[0].DoSomething();
 pTest[1].DoSomething();

 std::cout << '\n';
}
```

其运行结果为：

```
id: 0 - Doing something
id: 1 - Doing something

id: 1 - Destructor is being called
id: 0 - Destructor is being called
```

scoped_array 将负责使用 delete []，而不是 delete 来删除它所指向的对象。

- **shared_ptr**：意在用于对被指向对象的所有权进行共享。与 scoped_ptr 一样，被指向对象也保证会被删除，但不同的是，这将发生在最后一个指向它的 shared_ptr 被销毁时，或是调用 reset 方法时。shared_ptr 符合 C++标准库的“可复制构造”（CopyConstructible）和“可赋值”（Assignable）要求，所以可被用于标准的库容器中。另外它还提供了比较操作符，所以可与标准库的关联容器一起工作。shared_ptr 不能用于存储指向动态分配的数组的指针，这样的情况应该使用 shared_array。该模板的实现采用了引用计数技术，所以无法正确处理循环引用的情况。可以使用 weak_ptr 来“打破循环”。shared_ptr 还可在多线程环境中使用。

下面的例子演示怎样将 shared_ptr 用于 std::vector 中：

```
typedef boost::shared_ptr<CTest> TestPtr;

void PT(const TestPtr &t)
{
 std::cout << "id: " << t->GetId() << "\t\t"
           << "use count: " << t.use_count() << '\n';
}

void main()
{
 std::vector<TestPtr> TestVector;

 TestPtr pTest0(new CTest(0));
 TestVector.push_back(pTest0);

 TestPtr pTest1(new CTest(1));
 TestVector.push_back(pTest1);

 TestPtr pTest2(new CTest(2));
 TestVector.push_back(pTest2);

 std::for_each(TestVector.begin(), TestVector.end(), PT);
 std::cout << '\n';

 pTest0.reset();
 pTest1.reset();
 pTest2.reset();

 std::for_each(TestVector.begin(), TestVector.end(), PT);
 std::cout << '\n';
```

```
 TestVector.clear();
 std::cout << '\n';

 std::cout << "exiting...\n";
}
```

其运行结果为：

```
id: 0         use count: 2
id: 1         use count: 2
id: 2         use count: 2

id: 0         use count: 1
id: 1         use count: 1
id: 2         use count: 1

id: 0 - Destructor is being called
id: 1 - Destructor is being called
id: 2 - Destructor is being called

exiting...
```

运行结果中的“ use count ”是通过 shared_ptr 的 use_count()方法获得的“ 使用计数 ”，也就是，对所存储指针进行共享的 shared_ptr 对象的数目。我们可以看到，在通过 new 分配了 3 个 CTest 对象，并将相应的 shared_ptr 对象放入 TestVector 后，三个使用计数都为 2；而在我们使用 reset()方法复位 pTest0、pTest1 和 pTest2 后，TestVector 中的各个 shared_ptr 对象的使用计数变成了 1。这时，我们调用 TestVector 的 clear()方法清除它所包含的 shared_ptr 对象；因为已经没有 shared_ptr 对象再指向我们先前分配的 3 个 CTest 对象，这 3 个对象也随之被删除，并导致相应的析构器被调用。

- **shared_array**：该模板类与 shared_ptr 类似，但意在用于数组而不是单个对象。指向 std::vector 的 shared_ptr 可用于替换 scoped_array，并且远为灵活，但其效率也要低一点。

下面是使用实例：

```
void main()
{
 boost::shared_array<CTest> pTest1(new CTest[2]);

 pTest1[0].SetId(0);
 pTest1[1].SetId(1);

 std::cout << "use count: " << pTest1.use_count() << "\n\n";
 boost::shared_array<CTest> pTest2(pTest1);
 std::cout << "use count: " << pTest1.use_count() << "\n\n";
 pTest1.reset();
```

```
 pTest2[0].DoSomething();
 pTest2[1].DoSomething();
 std::cout << '\n';

 std::cout << "use count: " << pTest1.use_count() << "\n\n";
}
```

其运行结果为：

```
use count: 1

use count: 2

id: 0 - Doing something
id: 1 - Doing something

use count: 1

id: 1 - Destructor is being called
id: 0 - Destructor is being called
```

如此例所示，我们通过 new 所分配的数组只有在指向它的 pTest1 和 pTest2 都被销毁或复位后才被删除。

- **weak_ptr**：该模板类存储“已由 shared_ptr 管理的对象”的“弱引用”。要访问 weak_ptr 所指向的对象，可以使用 shared_ptr 构造器或 make_shared 函数来将 weak_ptr 转换为 shared_ptr。指向被管理对象的最后一个 shared_ptr 被销毁时将删除该对象，即使仍有 weak_ptr 指向它也是如此。与原始指针不同的是，届时最后一个 shared_ptr 会检查是否有 weak_ptr 指向该对象，如果有的话就将这些 weak_ptr 置为空。这样就不会发生使用原始指针时可能出现的“悬吊指针”（dangling pointer）情况，从而获得更高的安全水平。

  weak_ptr 符合 C++标准库的“可复制构造”（CopyConstructible）和“可赋值”（Assignable）要求，所以可被用于标准的库容器中。另外它还提供了比较操作符，所以可与标准库的关联容器一起工作。

```
void main()
{
 boost::shared_ptr<CTest> pTest(new CTest);
 boost::weak_ptr<CTest> pTest2(pTest);

 if(boost::shared_ptr<CTest> pTest3 = boost::make_shared(pTest2))
     pTest3->DoSomething();

 pTest.reset();

 assert(pTest2.get() == NULL);
}
```

其运行结果为：

```
id: 0 - Doing something
id: 0 - Destructor is being called
```

main 函数最后的断言确认了 pTest2 所存储的指针的确已被置为 NULL。

显然，Boost 的智能指针方案会让我们产生这样的疑问：如果我们还需要其他类型的智能指针（比如支持 COM 的智能指针），是否意味着我们必须在 C++中再增加智能指针类型，或是采用非标准的实现呢？在泛型技术已得到极大发展的今天，Boost 的“增加增加再增加”的思路是不能让人满意的。正是在这里，我们看到了下面将要介绍的 Loki Smart Pointer 的关键点：通过基于策略（policy-based）的设计来实现通用的智能指针模板。

## 二、 Loki 智能指针

按照美国传统辞典（双解）的解释，Loki 是“A Norse god who created discord, especially among his fellow gods.”(斯堪的纳维亚的一个制造混乱的神，尤其是在其同类之间)。就其给 Boost 智能指针带来的麻烦而言，Loki 智能指针倒真的当得起这个名字；而在另一方面，就其实现的优雅以及功能的强大（也就是说，它给开发者带来的好处）而言，它也的确属于“神族”。

上面已经说过，Loki 的智能指针方案采用了基于策略的设计。其要点在于把将各功能域分解为独立的、由主模板类进行混合和搭配的策略。让我们先来看一看 Loki 智能指针模板类 SmartPtr 的定义：

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

我们可以看到，除了 SmartPtr 所指向的对象类型 T 以外，在模板类 SmartPtr 中包括了这样一些策略：OwnershipPolicy（所有权策略）、ConversionPolicy（类型转换策略）、CheckingPolicy（检查策略）、StoragePolicy（存储策略）。正是通过这样的分解，使得 SmartPtr 具备了极大的灵活性。我们可以任意组合各种不同的策略，从而获得不同的智能指针实现。下面先对各个策略逐一进行介绍：

- **OwnershipPolicy**：指定所有权管理策略，可以从以下预定义的策略中选择：DeepCopy（深度复制）、RefCounted（引用计数）、RefCountedMT（多线程化引用计数）、COMRefCounted（COM 引用计数）、RefLinked（引用链接）、DestructiveCopy（销毁式复制），以及 NoCopy（无复制）。

- **ConversionPolicy**：指定是否允许进行向被指向类型的隐式转换。可以使用的实现有 AllowConversion 和 DisallowConversion。

- **CheckingPolicy**：定义错误检查策略。可以使用 AssertCheck、AssertCheckStrict、RejectNullStatic、RejectNull、RejectNullStrict，以及 NoCheck。

- **StoragePolicy**：定义怎样存储和访问被指向对象。Loki 已定义的策略有：DefaultSPStorage、ArrayStorage、LockedStorage，以及 HeapStorage。

除了 Loki 已经定义的策略，你还可以自行定义策略。实际上，Loki 的智能指针模板覆盖了四种基本的 Boost 智能指针类型：scoped_ptr、scoped_array、shared_ptr 和 shared_array；至于 weak_ptr，也可以通过定义相应的策略来实现其等价物。通过即将成为 C++标准（C++0x）的 typedef 模板特性，我们还可以利用 Loki 的 SmartPtr 模板来直接定义前面提到的 Boost 的前四种智能指针类型。举例来说，我们可以这样定义 shared_ptr：

```
template<typename T>    // typedef 模板还不是标准的
typedef Loki::SmartPtr
<
  T,
  RefCounted,         // 以下都是缺省的模板参数
  DisallowConversion,
  AssertCheck,
  DefaultSPStorage
>
shared_ptr;
```

下面是一个使用 Loki “ shared_ptr ”的实例：

```
typedef Loki::SmartPtr<CTest> TestPtr;

void PT(const TestPtr &t)
{
 std::cout << "id: " << t->GetId() << '\n';
}

void main()
{
 std::vector<TestPtr> TestVector;

 TestPtr pTest0(new CTest(0));
 TestVector.push_back(pTest0);

 TestPtr pTest1(new CTest(1));
 TestVector.push_back(pTest1);

 std::for_each(TestVector.begin(), TestVector.end(), PT);
 std::cout << '\n';

 Loki::Reset(pTest0, NULL);
 Loki::Reset(pTest1, NULL);

 std::for_each(TestVector.begin(), TestVector.end(), PT);
 std::cout << '\n';
```

```
 TestVector.clear();
 std::cout << '\n';

 std::cout << "exiting...\n";
}
```

其运行结果为：

```
id: 0
id: 1

id: 0
id: 1

id: 0 - Destructor is being called
id: 1 - Destructor is being called

exiting...
```

前面已经提到，要通过 Loki 定义与 Boost 的 shared_ptr 功能等价的智能指针，除了第一个模板参数以外，其他的参数都可以使用缺省值，所以在上面的例子中，我们直接使用“ typedef Loki::SmartPtr<CTest> TestPtr; ”就可以了。非常的简单！

为了进一步说明 Loki 的“ 基于策略的设计方法 ”，让我们再来看一个更为复杂的例子：通过 Loki::SmartPtr 实现线程专有存储（Thread-Specific Storage，TSS；又称线程局部存储，Thread Local Storage，TLS）。

所谓线程专有存储，是指这样一种机制，通过它，多线程程序可以使用一个逻辑上的全局访问点来访问线程专有的数据，并且不会给每次访问增加额外的锁定开销。举一个简单的例子，在 C 语言中，我们可以通过 errno 变量来获取错误代码；通常 errno 就是一个普通的全局变量——在单线程环境中，这当然没有什么问题，但如果是多线程环境，这个全局的 errno 变量就会给我们带来麻烦了。TSS 正是解决这一问题的有效方案。

显然，智能指针的语义能够很好地适用于 TSS。我们可以编写一种智能指针，使得所有对其所指向对象的访问都成为线程专有的——也就是说，每个线程访问的实际上是自己专有的对象，但从程序的外表来看，却都是对同一对象的访问。有了 Loki::SmartPtr，我们可以非常容易地实现这样的智能指针：如其名字所指示的，TSS 涉及的是存储问题，我们只要对 Loki::SmartPtr 的 StoragePolicy 进行定制就可以了，其他的工作可以交给 Loki::SmartPtr 去完成。

在 POSIX PThreads 库和 Win32 中都提供了用于线程专有存储的函数，它们分别是 pthread_key_create、pthread_setspecific、pthread_getspecific 和 pthread_key_delete（POSIX PThreads 库），以及 TlsAlloc、TlsSetValue、TlsGetValue 和 TlsFree（Win32）。关于这些函数的详细信息，请参阅相关的文档。

下面给出在 MSVC 6.0 下实现的用于 TSS 的 StoragePolicy，并通过注释逐行进行分析（这个实现使用了 PThreads-Win32 库，这是一个 Win32 上的 PThreads 实现。使用 Win32 的线程专有函数也可以实现类似的 StoragePolicy，但编写在线程退出时调用的 CleanupHook()却需要一点“ 窍门 ”。具体方法可参考 Boost 的 thread_specific_ptr 实现）：

```
namespace Loki
{
 // 实现 TSS 的 Loki 存储策略。改编自 Douglas C. Schmidt、Timothy H. Harrison
```

```
   // 和 Nat Pryce 的论文“ Thread-Specific Storage for C/C++ ”中的部分代码。
   // 使用了“ Loki VC 6.0 Port ”。
template <class T> class TS_SPStorage
{
public:
     typedef T* StoredType;    // 被指向对象的类型。
     typedef T* PointerType;   // operator->所返回的类型。
     typedef T& ReferenceType; // operator*所返回的类型。

public:
     // 构造器。对成员变量进行初始化。
     TS_SPStorage() : once_(0), key_(0), keylock_(NULL)
     { pthread_mutex_init(&keylock_, NULL); }

     // 析构器。释放先前获取的资源。
     ~TS_SPStorage()
     {
          pthread_mutex_destroy(&keylock_);
          pthread_key_delete(key_);
     }

     // 返回线程专有的被指向对象。
     PointerType operator->()  const { return GetPointee(); }
     // 返回线程专有的被指向对象的引用。
     ReferenceType operator*() { return *GetPointee(); }

          // Accessors。获取线程专有的被指向对象。
          friend inline PointerType GetImpl(TS_SPStorage& sp)
     { return sp.GetPointee(); }

     // 获取线程专有的被指向对象的引用。
        // 该函数没有实现。但 NoCheck 需要它才能正常工作。
          friend inline const StoredType& GetImplRef(const TS_SPStorage& sp)
     { return 0; }

protected:
          // 销毁所存储的数据。空函数。该工作将由 CleanupHook()在各个线程退出时完成。
          void Destroy() {}

     // 获取当前线程专有的数据。
     PointerType GetPointee()
     {
          PointerType tss_data = NULL;

          // 使用双重检查锁定模式来在除了初始化以外的情况下避免锁定。
          // 之所以在这里，而不是在构造器中对“ 专有钥 ”进行初始化及分配 TS 对象，
```

```
            // 是因为：(1) 最初创建 TS 对象的线程（例如，主线程）常常并不是使用
            // 它的线程（工作线程），所以在构造器中分配一个 TS 对象的实例常常并无
            // 好处，因为这个实例只能在主线程中访问。(2)在有些平台上，“ 专有
            // 钥 ” 是有限的资源，所以等到对 TS 对象进行第一次访问时再进行分配，有
            // 助于节省资源。
        // 第一次检查。
        if(once_ == 0)
        {
            // 加锁。确保访问的序列化。
            pthread_mutex_lock(&keylock_);

            // 双重检查。
            if(once_ == 0)
            {
                // 创建 “ 专有钥 ”。
                pthread_key_create(&key_, &CleanupHook);

                // 必须在创建过程的最后出现，这样才能防止其他线程在 “ 专有钥 ”
                 // 被创建之前使用它。
                once_ = 1;
            }

            // 解锁。
            pthread_mutex_unlock(&keylock_);
        }

        // 从系统的线程专有存储中获取数据。注意这里不需要加锁。
        tss_data = (PointerType)pthread_getspecific(key_);

        // 检查是否这是当前线程第一次进行访问。
        if (tss_data == NULL)
        {
            // 从堆中为 TS 对象分配内存。
            tss_data = new T;

            // 将其指针存储在系统的线程专有存储中。
            pthread_setspecific(key_, (void *)tss_data);
        }

        return tss_data;
    }

private:
    // 用于线程专有数据的 “ 专有钥 ”。
    pthread_key_t key_;
```

```
        // “第一次进入”标志。
        int once_;

        // 用于避免在初始化过程中产生竞态情况。
        pthread_mutex_t keylock_;

        // 清扫挂钩函数，释放为 TS 对象分配的内存。在每个线程退出时被调用。
        static void CleanupHook (void *ptr)
        {
            // 这里必须进行类型转换，相应的析构器才会被调用（如果有的话）
            delete (PointerType)ptr;
        }
    };

    // 用于模拟 typedef template 的结构。
       // 参见 Herb Sutter 的“Template Typedef”一文。
    struct TS_SPStorageWrapper
    {
        template <class T>
        struct In
        {
            typedef TS_SPStorage<T> type;
        };
    };
};
```

下面让我们来看一个使用实例。首先让我们先定义：

```
Loki::SmartPtr
<
    int,
    Loki::NoCopyWrapper,
    Loki::DisallowConversion,
    Loki::NoCheckWrapper,
    Loki::TS_SPStorageWrapper
> value;
```

其含义为：定义一种智能指针，被它指向的类型是 int，OwnershipPolicy 是 NoCopy，ConversionPolicy 是 DisallowConversion，CheckingPolicy 是 NoCheck（因为 TS 对象存储方式的限制，这是惟一能和 TS_SPStorage 一起使用的 CheckingPolicy。读者可自行尝试找到更好的解决方案），StoragePolicy 是 TS_SPStorage。

然后，编写这样一个程序：

```
pthread_mutex_t io_mutex = NULL; // iostreams 不一定是线程安全的!

void *thread_proc(void *param)
```

```
{
 int id = (int)param;

 *value = 0;
     for (int i = 0; i < 3; i++)
     {
          (*value)++;

     pthread_mutex_lock(&io_mutex);
     std::cout << "thread " << id << ": " << *value << '\n';
     pthread_mutex_unlock(&io_mutex);
     }

 return NULL;
}

void main(int argc, char* argv[])
{
 pthread_mutex_init(&io_mutex, NULL);
 pthread_t id[3];

 for(int i = 0; i < 3; i++)
     pthread_create(&id[i], 0, thread_proc, (void *)i);

 for(int i = 0; i < 3; i++)
     pthread_join(id[i], NULL);

 pthread_mutex_destroy(&io_mutex);
}
```

其运行结果为：

```
thread 0: 1
thread 0: 2
thread 1: 1
thread 2: 1
thread 1: 2
thread 2: 2
thread 1: 3
thread 2: 3
thread 0: 3
```

由此我们可以看出，尽管看起来在各个线程中访问的都是同样的*value，但实际上访问的却是各自的线程专有的对象。而且除了初始化以外，对这些对象的访问无需进行序列化。

因为 Loki::SmartPtr 为我们做了大量的工作，TS_SPStorage 的实现十分简洁明了。有兴趣的读者，可以对这里的实现与 Boost 的 thread_specific_ptr 进行比较。

像这样对 Loki::SmartPtr 的策略进行的定制，在理论上数目是无限的，也就是说，通过它我们可以拥有五花八门、“千奇百怪”的智能指针——了不起的 Loki，不是吗？有了这样的模板类，我们就再不需要一次一次地为标准 C++增加新的智能指针类型了。

## 三、 结束语

在推进器 Boost 和斯堪的纳维亚之神 Loki 的战争中，谁将胜出恐怕已不言而喻。倘若扮演上帝的 C++标准委员会起了偏心，硬要选中 Boost 的智能指针方案的话，那么在接下来的日子里，他们就将不再是 C++标准委员会，而是 C++智能指针救火委员会了。而即便那样，Loki，斯堪的纳维亚之神，也仍将矗立在 C++大陆上，为他的“子民”——C++程序员们——带来统一、秩序和和谐。

## 相关资源


1. Herb Sutter，The New C++: Smart(er) Pointers，见 C++ User Journal 网站：http://www.cuj.com/experts/2008/sutter.htm
2. Boost 文档：http://www.boost.org
3. Andrei Alexandrescu，Modern C++ Design 及 Loki，见 http://www.moderncppdesign.com 或 http://www.awprofessional.com/catalog/product.asp?product_id=%7B4ED3E6F3-371F-4ADC-9810-CC7B936164E3%7D。
4. Douglas C. Schmidt、Timothy H. Harrison 和 Nat Pryce，Thread-Specific Storage for C/C++,见 http://www.cs.wustl.edu/~schmidt/或 http://www.flyingdonkey.com/ace/(中文)。
5. Herb Sutter，Template Typedef，见 http://www.gotw.ca/gotw/079.htm。
6. PThreads-Win32 库，见 http://sources.redhat.com/pthreads-win32/。