

胡超的学习日志

[目录视图](#)[摘要视图](#)[RSS 订阅](#)

个人资料



xt_xiaotian

访问：226478次

积分：2006

等级：**BLOG > 5**

排名：第8347名

原创：29篇 转载：3篇

译文：1篇 评论：100条

[博客Markdown编辑器上线啦](#)[那些年我们追过的Wrox精品红皮计算机图书](#)[PMBOK第五版精讲视频教程](#)[火星人敏捷开发1001问](#)

C++ 智能指针详解

分类：[C++](#) [第三方开源库](#)

2010-07-05 20:19

34446人阅读

[评论\(15\)](#)[收藏](#)[举报](#)[c++](#)[delete](#)[reference](#)[编程](#)[架构设计](#)[vector](#)

C++ 智能指针详解

一、简介

由于 C++ 语言没有自动内存回收机制，程序员每次 new 出来的内存都要手动 delete。程序员忘记 delete，流程太复杂，最终导致没有 delete，异常导致程序过早退出，没有执行 delete 的情况并不罕见。

用智能指针便可以有效缓解这类问题，本文主要讲解参见的智能指针的用法。包括：`std::auto_ptr`、`boost::scoped_ptr`、`boost::shared_ptr`、`boost::scoped_array`、`boost::shared_array`、`boost::intrusive_ptr`。你可能会想，如此多的智能指针就为了解决 new、delete 匹配问题，真的有必要吗？看完这篇文章后，我想你心里自然会有答案。

文章搜索

文章分类

[C++ 第三方开源库](#) (8)[C/C++ 语言](#) (7)[Linux 开发应用技术](#) (5)[Windows 开发调试技术](#) (16)[互联网产品体验](#) (1)[其他](#) (2)[移动开发](#) (2)

文章存档

[2013年12月](#) (1)[2011年10月](#) (1)[2011年09月](#) (1)[2011年08月](#) (1)[2011年07月](#) (2)[展开](#)

阅读排行

[使用 C++ 处理 JSON 数据](#)
(41910)[C++ 智能指针详解](#)
(34442)

下面就按照顺序讲解如上 7 种智能指针（smart_ptr）。

二、具体使用

1、总括

对于编译器来说，智能指针实际上是一个栈对象，并非指针类型，在栈对象生命期即将结束时，智能指针通过析构函数释放有它管理的堆内存。所有智能指针都重载了“operator->”操作符，直接返回对象的引用，用以操作对象。访问智能指针原来的方法则使用“.”操作符。

访问智能指针包含的裸指针则可以用 get() 函数。由于智能指针是一个对象，所以 if (my_smart_object)永远为真，要判断智能指针的裸指针是否为空，需要这样判断：if (my_smart_object.get())。

智能指针包含了 reset() 方法，如果不传递参数（或者传递 NULL），则智能指针会释放当前管理的内存。如果传递一个对象，则智能指针会释放当前对象，来管理新传入的对象。

我们编写一个测试类来辅助分析：

```
class Simple {  
public:  
    Simple(int param = 0) {  
        number = param;  
        std::cout << "Simple: " << number << std::endl;  
    }  
  
    ~Simple() {  
        std::cout << "~Simple: " << number << std::endl;  
    }  
  
    void PrintSomething() {
```

[SendMessage、PostMe \(31754\)](#)
[让Visual Studio载入Sym \(8995\)](#)
[Linux下建立安装 smb, : \(8798\)](#)
[Windows 中 SQLite3 使 \(8271\)](#)
[Vista、Win7 网络共享访 \(8151\)](#)
[在 Win7 下使用 VS2005 \(7961\)](#)
[在Windows中编译Nginx \(7416\)](#)
[在Ubuntu10.04中建立Af \(7354\)](#)

评论排行

[使用 C++ 处理 JSON 数 \(23\)](#)
[SendMessage、PostMe \(23\)](#)
[C++ 智能指针详解 \(15\)](#)
[C/C++函数调用约定 \(6\)](#)
[Qualcomm ARM CPU与 \(4\)](#)
[在Ubuntu10.04中建立Af \(4\)](#)
[在 Win7 下使用 VS2005 \(4\)](#)
[深入解析SendMessage、 \(4\)](#)
[在Windows中编译Nginx \(3\)](#)
[Linux下建立安装 smb, : \(3\)](#)

推荐文章

```
std::cout << "PrintSomething: " << info_extend.c_str() << std::endl;
}

std::string info_extend;

int number;

};
```

2、std::auto_ptr

std::auto_ptr 属于 STL，当然在 namespace std 中，包含头文件 #include<memory> 便可以使用。std::auto_ptr 能够方便的管理单个堆内存对象。

我们从代码开始分析：

```
void TestAutoPtr() {
    std::auto_ptr<Simple> my_memory(new Simple(1)); // 创建对象，输出：Simple: 1
    if (my_memory.get()) {                          // 判断智能指针是否为空
        my_memory->PrintSomething();                 // 使用 operator-> 调用智能指针对象中的函数
        my_memory.get()->info_extend = "Addition";  // 使用 get() 返回裸指针，然后给内部对象赋值
        my_memory->PrintSomething();                 // 再次打印，表明上述赋值成功
        (*my_memory).info_extend += " other";        // 使用 operator* 返回智能指针内部对象，然后用“.”调用智能指针对象
    }
    my_memory->PrintSomething();                       // 再次打印，表明上述赋值成功
}
// my_memory 栈对象即将结束生命期，析构堆对象 Simple(1)
```

执行结果为：

```
Simple: 1
PrintSomething:
```

- * [纯CSS实现表单验证](#)
- * [段落文字彩条效果](#)
- * [Kepler性能分析之M2E调优](#)
- * [公共技术点之 Java反射 Reflection](#)
- * [QtAndroid详解\(2\): startActivity 和它的小伙伴们](#)
- * [Qt5官方demo解析集34——Concentric Circles Example](#)

最新评论

C++ 智能指针详解

LulinHsiao: 这样基类仅仅观察自己的 boost::weak_ptr 是否为空就知道子类有没对自己赋值了，这句话怎...

C++ 智能指针详解

krystal_wln: 最开始来看你写的时候完全看不懂，要有一点理解了再来看就觉得写的很好

C++ 智能指针详解

wbloodc: 讲解清楚，学习收藏了

C/C++函数调用约定

Nvidia_inside: 很好~~

深入解析SendMessage、PostW mingtiandexia: 学习了

C++ 智能指针详解

hb2008hahaha: 总结的很全面，例子也很好，顶一个

C++ 智能指针详解

huiliao: 不错

注册表 Run、RunOnce 键值解 natashage: 这个分享太好了

Qualcomm ARM CPU与Intel x8

PrintSomething: Addition

PrintSomething: Addition other

~Simple: 1

上述为正常使用 std::auto_ptr 的代码，一切似乎都良好，无论如何不用我们显示使用该死的 delete 了。

其实好景不长，我们看看如下的另一个例子：

```
void TestAutoPtr2() {  
  
    std::auto_ptr<Simple> my_memory(new Simple(1));  
  
    if (my_memory.get()) {  
  
        std::auto_ptr<Simple> my_memory2; // 创建一个新的 my_memory2 对象  
  
        my_memory2 = my_memory;          // 复制旧的 my_memory 给 my_memory2  
  
        my_memory2->PrintSomething();    // 输出信息，复制成功  
  
        my_memory->PrintSomething();    // 崩溃  
  
    }  
}
```

最终如上代码导致崩溃，如上代码时绝对符合 C++ 编程思想的，居然崩溃了，跟进 std::auto_ptr 的源码后，我们看到，罪魁祸首是“my_memory2 = my_memory”，这行代码，my_memory2 完全夺取了 my_memory 的内存管理所有权，导致 my_memory 悬空，最后使用时导致崩溃。

所以，使用 std::auto_ptr 时，绝对不能使用“operator=”操作符。作为一个库，不允许用户使用，确没有明确拒绝[1]，多少会觉得有点出乎预料。

看完 std::auto_ptr 好景不长的第一个例子后，让我们再来看一个：

```
void TestAutoPtr3() {  
  
    std::auto_ptr<Simple> my_memory(new Simple(1));
```

dalianzhipailiugang: 虽然数字上看ARM平台的性能差距很大,不过你们有没有感觉其实在PC上运行程序性能很差,当然主要是因为...

SendMessage、PostMessage
卿穹军: 顶

```
if (my_memory.get()) {  
    my_memory.release();  
}  
}
```

执行结果为:

Simple: 1

看到什么异常了吗? 我们创建出来的对象没有被析构, 没有输出“~Simple: 1”, 导致内存泄露。当我们不想让 my_memory 继续生存下去, 我们调用 release() 函数释放内存, 结果却导致内存泄露 (在内存受限系统中, 如果my_memory占用太多内存, 我们会考虑在使用完成后, 立刻归还, 而不是等到my_memory 结束生命期后才归还)。

正确的代码应该为:

```
void TestAutoPtr3() {  
    std::auto_ptr<Simple> my_memory(new Simple(1));  
    if (my_memory.get()) {  
        Simple* temp_memory = my_memory.release();  
        delete temp_memory;  
    }  
}
```

或

```
void TestAutoPtr3() {  
    std::auto_ptr<Simple> my_memory(new Simple(1));  
    if (my_memory.get()) {  
        my_memory.reset(); // 释放 my_memory 内部管理的内存  
    }  
}
```

```
}
```

原来 `std::auto_ptr` 的 `release()` 函数只是让出内存所有权，这显然也不符合 C++ 编程思想。

总结：`std::auto_ptr` 可用来管理单个对象的内存，但是，请注意如下几点：

- (1) 尽量不要使用“operator=”。如果使用了，请不要再使用先前对象。
- (2) 记住 `release()` 函数不会释放对象，仅仅归还所有权。
- (3) `std::auto_ptr` 最好不要当成参数传递（读者可以自行写代码确定为什么不能）。
- (4) 由于 `std::auto_ptr` 的“operator=”问题，有其管理的对象不能放入 `std::vector` 等容器中。
- (5)

使用一个 `std::auto_ptr` 的限制还真多，还不能用来管理堆内存数组，这应该是你目前在想的事情吧，我也觉得限制挺多的，哪天一个不小心，就导致问题了。

由于 `std::auto_ptr` 引发了诸多问题，一些设计并不是非常符合 C++ 编程思想，所以引发了下面 `boost` 的智能指针，`boost` 智能指针可以解决如上问题。

让我们继续向下看。

3、`boost::scoped_ptr`

`boost::scoped_ptr` 属于 `boost` 库，定义在 `namespace boost` 中，包含头文件

`#include<boost/smart_ptr.hpp>` 便可以使用。`boost::scoped_ptr` 跟 `std::auto_ptr` 一样，可以方便的管理单个堆内存对象，特别的是，`boost::scoped_ptr` 独享所有权，避免了 `std::auto_ptr` 恼人的几个问题。

我们还是从代码开始分析：

```
void TestScopedPtr() {  
    boost::scoped_ptr<Simple> my_memory(new Simple(1));  
    if (my_memory.get()) {  
        my_memory->PrintSomething();  
        my_memory.get()->info_extend = "Addition";  
        my_memory->PrintSomething();  
    }  
}
```

```

(*my_memory).info_extend += " other";

my_memory->PrintSomething();

my_memory.release();    // 编译 error: scoped_ptr 没有 release 函数

std::auto_ptr<Simple> my_memory2;


my_memory2 = my_memory;    // 编译 error: scoped_ptr 没有重载 operator=, 不会导致所有权转移
}
}

```

首先，我们可以看到，boost::scoped_ptr 也可以像 auto_ptr 一样正常使用。但其没有 release() 函数，不会导致先前的内存泄露问题。其次，由于 boost::scoped_ptr 是独享所有权的，所以明确拒绝用户写“my_memory2 = my_memory”之类的语句，可以缓解 std::auto_ptr 几个恼人的问题。

由于 boost::scoped_ptr 独享所有权，当我们真真需要复制智能指针时，需求便满足不了了，如此我们再引入一个智能指针，专门用于处理复制，参数传递的情况，这便是如下的 boost::shared_ptr。

4、boost::shared_ptr

boost::shared_ptr 属于 boost 库，定义在 namespace boost 中，包含头文件 `#include<boost/smart_ptr.hpp>` 便可以使用。在上面我们看到 boost::scoped_ptr 独享所有权，不允许赋值、拷贝，boost::shared_ptr 是专门用于共享所有权的，由于要共享所有权，其在内部使用了引用计数。boost::shared_ptr 也是用于管理 

我们还是从代码开始分析：

```

void TestSharedPtr(boost::shared_ptr<Simple> const reference)
{
    memory->PrintSomething();

    std::cout << "TestSharedPtr UseCour
}

```

```
void TestSharedPtr2() {  
    boost::shared_ptr<Simple> my_memory;  
    if (my_memory.get()) {  
        my_memory->PrintSomething();  
        my_memory.get()->info_extend = "Addition";  
        my_memory->PrintSomething();  
        (*my_memory).info_extend += " other";  
        my_memory->PrintSomething();  
    }  
  
    std::cout << "TestSharedPtr2 UseCount: " << my_memory.use_count() << std::endl;  
    TestSharedPtr(my_memory);  
    std::cout << "TestSharedPtr2 UseCount: " << my_memory.use_count() << std::endl;  
  
    //my_memory.release();// 编译 error: 同样, shared_ptr 也没有 release 函数  
}
```

执行结果为:

Simple: 1

PrintSomething:

PrintSomething: Addition

PrintSomething: Addition other

TestSharedPtr2 UseCount: 1

PrintSomething: Addition other

TestSharedPtr UseCount: 2

TestSharedPtr2 UseCount: 1

~Simple: 1

boost::shared_ptr 也可以很方便的使用。并且没有 release() 函数。关键的一点，boost::shared_ptr 内部维护了一个引用计数，由此可以支持复制、参数传递等。boost::shared_ptr 提供了一个函数 use_count()，此函数返回 boost::shared_ptr 内部的引用计数。查看执行结果，我们可以看到在 TestSharedPtr2 函数中，引用计数为 1，传递参数后（此处进行了一次复制），在函数 TestSharedPtr 内部，引用计数为 2，在 TestSharedPtr 返回后，引用计数又降低为 1。当我们需要使用一个共享对象的时候，boost::shared_ptr 是再好不过的了。

在此，我们已经看完单个对象的智能指针管理，关于智能指针管理数组，我们接下来讲到。

5、boost::scoped_array

boost::scoped_array 属于 boost 库，定义在 namespace boost 中，包含头文件

#include<boost/smart_ptr.hpp> 便可以使用。

boost::scoped_array 便是用于管理动态数组的。跟 boost::scoped_ptr 一样，也是独享所有权的。

我们还是从代码开始分析：

```
void TestScopedArray() {  
    boost::scoped_array<Simple> my_memory(new Simple[2]); // 使用内存数组来初始化  
    if (my_memory.get()) {  
        my_memory[0].PrintSomething();  
        my_memory.get()[0].info_extend = "Addition";  
        my_memory[0].PrintSomething();  
        (*my_memory)[0].info_extend += " other"; // 编译 error, scoped_ptr 没有重载 operator*  
        my_memory[0].release(); // 同上, 没有 release 函数  
        boost::scoped_array<Simple> my_memory2;  
        my_memory2 = my_memory; // 编译 error, 同上, 没有重载 operator=  
    }  
}
```

```
}
```

boost::scoped_array 的使用跟 boost::scoped_ptr 差不多，不支持复制，并且初始化的时候需要使用动态数组。另外，boost::scoped_array 没有重载“operator*”，其实这并无大碍，一般情况下，我们使用 get() 函数更明确些。

下面肯定应该讲 boost::shared_array 了，一个用引用计数解决复制、参数传递的智能指针类。

6、boost::shared_array

boost::shared_array 属于 boost 库，定义在 namespace boost 中，包含头文件 `#include<boost/smart_ptr.hpp>` 便可以使用。

由于 boost::scoped_array 独享所有权，显然在很多情况下（参数传递、对象赋值等）不满足需求，由此我们引入 boost::shared_array。跟 boost::shared_ptr 一样，内部使用了引用计数。

我们还是从代码开始分析：

```
void TestSharedArray(boost::shared_array<Simple> memory) { // 注意：无需使用 reference (或 const reference)
    std::cout << "TestSharedArray UseCount: " << memory.use_count() << std::endl;
}
```

```
void TestSharedArray2() {
    boost::shared_array<Simple> my_memory(new Simple[2]);
    if (my_memory.get()) {
        my_memory[0].PrintSomething();
        my_memory.get()[0].info_extend = "Addition 00";
        my_memory[0].PrintSomething();
        my_memory[1].PrintSomething();
        my_memory.get()[1].info_extend = "Addition 11";
        my_memory[1].PrintSomething();
    }
}
```

```
    //(*my_memory)[0].info_extend += " other"; // 编译 error, scoped_ptr 没有重载 operator*
}

std::cout << "TestSharedArray2 UseCount: " << my_memory.use_count() << std::endl;
TestSharedArray(my_memory);

std::cout << "TestSharedArray2 UseCount: " << my_memory.use_count() << std::endl;
}
```

执行结果为：

Simple: 0

Simple: 0

PrintSomething:

PrintSomething: Addition 00

PrintSomething:

PrintSomething: Addition 11

TestSharedArray2 UseCount: 1

TestSharedArray UseCount: 2

TestSharedArray2 UseCount: 1

~Simple: 0

~Simple: 0

跟 boost::shared_ptr 一样，使用了引用计数，可以复制，通过参数来传递。

至此，我们讲过的智能指针有

std::auto_ptr、boost::scoped_ptr、boost::shared_ptr、boost::scoped_array、boost::shared_array。这几个智能指针已经基本够我们使用了，90% 的使用过标准智能指针的代码就这 5 种。可如下还有两种智能指针，它们肯定有用，但有什么用处呢，一起看看吧。

7、boost::weak_ptr

boost::weak_ptr 属于 boost 库，定义在 namespace boost 中，包含头文件 `#include<boost/smart_ptr.hpp>` 便可以使用。

在讲 boost::weak_ptr 之前，让我们先回顾一下前面讲解的内容。似乎 boost::scoped_ptr、boost::shared_ptr 这两个智能指针就可以解决所有单个对象内存的管理了，这儿还多出一个 boost::weak_ptr，是否还有某些情况我们没纳入考虑呢？

回答：有。首先 boost::weak_ptr 是专门为 boost::shared_ptr 而准备的。有时候，我们只关心能否使用对象，并不关心内部的引用计数。boost::weak_ptr 是 boost::shared_ptr 的观察者（Observer）对象，观察者意味着 boost::weak_ptr 只对 boost::shared_ptr 进行引用，而不改变其引用计数，当被观察的 boost::shared_ptr 失效后，相应的 boost::weak_ptr 也相应失效。

我们还是从代码开始分析：

```
void TestWeakPtr() {  
  
    boost::weak_ptr<Simple> my_memory_weak;  
  
    boost::shared_ptr<Simple> my_memory(new Simple(1));  
  
  
    std::cout << "TestWeakPtr boost::shared_ptr UseCount: " << my_memory.use_count() << std::endl;  
  
    my_memory_weak = my_memory;  
  
    std::cout << "TestWeakPtr boost::shared_ptr UseCount: " << my_memory.use_count() << std::endl;  
  
}
```

执行结果为：

```
Simple: 1  
TestWeakPtr boost::shared_ptr UseCount: 1  
TestWeakPtr boost::shared_ptr UseCount: 1  
~Simple: 1
```

我们看到，尽管被赋值了，内部的引用计数并没有什么变化，当然，读者也可以试试传递参数等其

他情况。

现在要说的的问题是，`boost::weak_ptr` 到底有什么作用呢？从上面那个例子看来，似乎没有任何作用，其实 `boost::weak_ptr` 主要用在软件架构设计中，可以在基类（此处的基类并非抽象基类，而是指继承于抽象基类的虚基类）中定义一个 `boost::weak_ptr`，用于指向子类的 `boost::shared_ptr`，这样基类仅仅观察自己的 `boost::weak_ptr` 是否为空就知道子类有没对自己赋值了，而不用影响子类 `boost::shared_ptr` 的引用计数，用以降低复杂度，更好的管理对象。

8、`boost::intrusive_ptr`

`boost::intrusive_ptr` 属于 `boost` 库，定义在 `namespace boost` 中，包含头文件 `#include<boost/smart_ptr.hpp>` 便可以使用。

讲完如上 6 种智能指针后，对于一般程序来说 C++ 堆内存管理就够用了，现在又多了一种 `boost::intrusive_ptr`，这是一种插入式的智能指针，内部不含有引用计数，需要程序员自己加入引用计数，不然编译不过（☹~☹b汗）。个人感觉这个智能指针没太大用处，至少我没用过。有兴趣的朋友自己研究一下源代码哦J。

三、总结

如上讲了这么多智能指针，有必要对这些智能指针做个总结：

- 1、在可以使用 `boost` 库的场合下，拒绝使用 `std::auto_ptr`，因为其不仅不符合 C++ 编程思想，而且极容易出错[2]。
- 2、在确定对象无需共享的情况下，使用 `boost::scoped_ptr`（当然动态数组使用 `boost::scoped_array`）。
- 3、在对象需要共享的情况下，使用 `boost::shared_ptr`（当然动态数组使用 `boost::shared_array`）。
- 4、在需要访问 `boost::shared_ptr` 对象，而又不想改变其引用计数的情况下，使用 `boost::weak_ptr`，一般常用于软件框架设计中。

5、最后一点，也是要求最苛刻一点：在你的代码中，不要出现 delete 关键字（或 C 语言的 free 函数），因为可以用智能指针去管理。

[1]参见《effective C++（3rd）》，条款06。

[2]关于 boost 库的使用，可本博客另外一篇文章：《在 Windows 中编译 boost1.42.0》。

[3]读者应该看到了，在我所有的名字前，都加了命名空间标识符std::（或boost::），这不是我不想写 using namespace XXX 之类的语句，在大型项目中，有可能会用到 N 个第三方库，如果把命名空间全放出来，命名污染（Naming conflicts）问题很难避免，到时要改回来是极端麻烦的事情。当然，如果你只是写 Demo，可以例外。

[上一篇](#) [最快速度找到内存泄漏](#)

[下一篇](#) [手机屏幕解析](#)

主题推荐

[智能指针](#)

[C++](#)

[内存泄露](#)

[内存管理](#)

[架构设计](#)

猜你在找

[moto & google笔试题目-STLC++面试题](#)

[string的size和length](#)

[CC++2014年7月华为校招机试真题一](#)

[GDB 查看死锁](#)

[cs硕士妹子找工作经历阿里人搜等互联网](#)

[C++学习之深入理解虚函数--虚函数表解析](#)

[抄袭事件判决书](#)

[我的2012-分享我的四个项目经验](#)

[TCP拥塞控制算法内核实现剖析九](#)

[asserth头文件之断言](#)

准备好了么？**跳吧**

软件开发工程师（C++）

我要跳槽

更多职位尽在 **CSDN JOB**

苏州敏行医学信息技术有限公司

| 6-10K/月

c++开发工程师

我要跳槽

C++服务器

我要跳槽

京品高科信息科技（北京）有限公司

| 7-15K/月

北京乐动卓越科技有限公司

| 15-20K/月

C++软件开发工程师

我要跳槽

天津市努思企业服务有限公司

| 7-10K/月



查看评论

15楼 [LulinHsiao](#) 2015-01-10 10:03发表



这样基类仅仅观察自己的 boost::weak_ptr 是否为空就知道子类有没对自己赋值了，
这句话怎么理解呢？

14楼 [krystal_wln](#) 2015-01-05 17:48发表



最开始来看你写的的时候完全看不懂，要有一点理解了再来看就觉得写的很好

13楼 [wbloodc](#) 2014-08-26 10:38发表



讲解清楚，学习收藏了

12楼 [hb2008hahaha](#) 2014-05-16 15:44发表



总结的很全面，例子也很好，顶一个

11楼 [huiliao](#) 2014-04-20 23:30发表



不错

10楼 [驭风者_z](#) 2013-11-20 11:19发表



讲的好，学习了 谢谢

9楼 [chz429](#) 2013-08-20 21:18发表



学习了，谢谢楼主！

8楼 [ToConnection](#) 2013-07-19 15:19发表



很好

7楼 [qianwen36](#) 2013-06-27 09:09发表



总觉得你这么使用智能指针所谓，违背了cpp语言哲学，哪有这样搞的，只见new不见delete，神经般的设计啊？
cpp的信仰是尊崇对称完美的。要么new与delete都封装隐藏起来；不然像什么话。

个人觉得需要两类概念模板，一类是把new与delete都隐藏起来的动态构造器，一类是像智能指针这类的参数化指针。

像智能指针本身是对象销毁型概念模板

6楼 [benoit_fr](#) 2013-06-27 08:23发表



写的很好

5楼 [zhumeng1989](#) 2013-04-27 11:47发表



c++11 做了一些提高

4楼 [yingkongshi99](#) 2012-12-10 11:38发表



感谢楼主，学习了

3楼 [kai_ding](#) 2012-10-25 09:49发表



写的最清楚的智能指针专题了，谢谢

2楼 [qq349193851](#) 2012-07-01 23:38发表



学习学习啊。。应用很困难啊。。

1楼 [fengbangyue](#) 2012-06-09 12:12发表



讲的很好，很清楚。学习了

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap

[网站客服](#)

[杂志客服](#)

[微博客服](#)

webmaster@csdn.net

400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏乐知网络技术有限公司 提供商务支持

京 ICP 证 070598 号 | Copyright © 1999-2014, CSDN.NET, All Rights Reserved

