Table of Contents

智能指针探究

组员

Name	Student ID	Github ID
王若晖	PB15000142	noirgif
张一卓	PB15111610	eastOffice
钱劲翔	PB15151800	Qianjx

主要内容

智能指针是一种模拟指针的行为，但提供如自动内存管理功能的抽象数据类型。它对普通指针加了一层封装机制，其目的是为了使得智能指针可以方便的管理一个对象的生命期。正确使用智能指针，可以方便的防止指针的悬空引用，内存泄漏等问题。

本次项目尝试从C++的智能指针入手，探索并讨论如下内容：

- raw pointer 在应用中可能产生的问题
- C++ 中 smart pointer 的标准，实现方式和内存管理
- Rust 语言中的智能指针及其内存管理
- C++ shared_ptr 的简单实现

C++ 指针存在的缺陷

引言

C++语言与Java语言的不同之处，其中有一个一定要提到的就是指针了，Java是不存在指针的，而C++的指针操作反而是C++语言的一个特点(并且因为没有直接对内存的操作，Java又被称作是安全的编程语言)。为了探究C++智能指针的新特性，我们有必要先介绍一下一般的C++指针以及其可能引发的问题。

指针的定义

指针通常意义上指的是内存地址，这是它和C一致的定义的。www.cplusplus.com上对内存地址的解释就是一个编号的数值，这个数值代表了该变量起始地址在计算机内存中存储的位置。也就是说指针是指向存储特定内存空间的变量，指针的值是地址的编号。

指针的相关操作

- 取地址(&):得到一个指向变量的内存地址的指针。
- 得指针值(*) :得到一个指针变量指向的内存处所储存的值;此外这个运算符也可以定义指针。
- 指针的运算符(++,--)等:由于指针本身的值也是32位或者64位的值，这些整数操作符功能也基本相似。
- 赋值(=):当将指针理解为一个数值时，赋值操作也就很好理解了。但是有的时候你并不能保证指针指向的内存里时正确的对象，这就是我们接下来主要讨论的问题———C++ 指针存在的缺陷。

参考文献：


```
C Traps and Pitfalls
```

迷途指针

迷途指针，或称悬空指针、野指针，指的是不指向任何合法的对象的指针。

当所指向的对象被释放或者收回，但是对该指针没有作任何的修改，以至于该指针仍旧指向已经回收的内存地址，此情况下该指针便称迷途指针。若操作系统将这部分已经释放的内存重新分配给另外一个进程，而原来的程序重新引用现在的迷途指针，则将产生无法预料的后果。因为此时迷途指针所指向的内存现在包含的已经完全是不同的数据。通常来说，若原来的程序继续往迷途指针所指向的内存地址写入数据，这些和原来程序不相关的数据将被损坏，进而导致不可预料的程序错误。这种类型的程序错误，不容易找到问题的原因，通常会导致内存区块错误（Linux系统中）和一般保护错误（Windows系统中）。如果操作系统的内存分配器将已经被覆盖的数据区域再分配，就可能会影响系统的稳定性。

某些编程语言(C++)允许未初始化的指针的存在，而这类指针即为野指针。

下面是两种常见的悬空指针的情况：

1. 引用已经释放的空间

```
int *a=new int(3);
//do something
delete a;
cout<<a;
```

2. 引用未定义的空间

```
int *a=new int[3];
//do something
cout<<a[3];
delete a[];
```

迷途指针/野指针这类错误经常会导致安全漏洞。 例如，如果一个指针用来调用一个虚函数，由于vtable指针被覆盖了，因此可能会访问一个不同的地址（指向被利用的代码）。或者，如果该指针用来写入内存，其它的数据结构就有可能损坏了。一旦该指针成为迷途指针，即使这段内存是只读的，仍然会导致信息的泄露（如果感兴趣的数据放在下一个数据结构里面，恰好分配在这段内存之中）或者访问权限的增加（如果现在不可使用的内存恰恰被用来安全检测）。

内存泄漏：

内存泄漏是程序设计中一项常见错误，特别是使用没有内置自动垃圾回收的编程语言(C和C++)。一般情况下，内存泄漏发生是因为不能访问动态分配的内存。不过目前有相当数量的调试工具用于检测不能访问的内存，从而可以防止内存泄漏问题。同时垃圾回收机制则可以应用到任何编程语言，而C / C++也有此类库。

- 一个明显的内存泄漏C++程序段

```
void f(void)
{
     int *s = new int[50]; // 申请内存空间
     return;
     /*  s 指向新分配的堆空间。
     当此函数返回，离开局部变量s的作用域后将无法得知s的值，
     分配的内存空间不能被释放。
    */
 }
 int main(void)
 {
     while (true) f(); // new函数迟早会由于内存泄漏而返回NULL
     return 0;
 }
```

- 一个不太明显的内存泄漏C++程序段

```
typedef struct A{
    int a[1000];
    linkA B;
}A,*linkA

void f(A X,int b)
{
     b--;
     if(b==0) return;
     else f(A,b);
     /*这个函数是一个递归函数，
     虽然C中的函数的形式参数会在函数结束的时候释放，
     但是由于这个函数递归层数较多以及X占空间较大，应该很快能耗尽堆栈。
     这也是内存泄漏，实际上只要将参数设为A类型的指针就可以了
    */
 }
 int main(void)
 {
     linkA X= new A;
```

```
    int b=100;
    f(X,b);
    delete(X);
}
```

内存泄漏会因为减少可用内存的数量从而降低计算机的性能。在最糟糕的情况下，过多的可用内存被分配掉导致全部或部分设备停止正常工作，或者应用程序崩溃。不过内存泄漏带来的后果可能是不严重的，有时甚至能够被常规的手段检测出来。而且在现代操作系统中，一个应用程序使用的常规内存在程序终止时被释放。这表示一个短暂运行的应用程序中的内存泄漏不会导致严重后果。

在以下情况，内存泄漏导致较严重的后果：

- 程序运行后置之不理，并且随着时间的流逝消耗越来越多的内存（比如服务器上的后台任务，尤其是嵌入式系统中的后台任务，这些任务可能被运行后很多内都置之不理）
- 新的内存被频繁地分配，比如当显示电脑游戏或动画视频画面时
- 程序能够请求未被释放的内存,比如共享内存
- 泄漏在操作系统内部发生
- 泄漏在系统关键驱动中发生
- 内存非常有限，比如在嵌入式系统或便携设备中
- 当运行于一个终止时内存并不自动释放的操作系统（比如AmigaOS）之上，而且一旦丢失只能通过重启来恢复

引言

在没有接触智能之前，很本能的问题就是 C++ 设计它的动机是什么。我们看如下一段代码：

```
ObjectType* ptr = new ObjectType();
ptr->use_function();
delete ptr; // after using ptr
```

我们知道，使用如上的普通指针时，需要在使用完指针指向的对象后删除它。然而，如果我们忘记 `delete` ，那么会留下一个该指针的悬空引用，很容易导致内存泄漏。

那么，在每一次使用普通指针的时候别忘了 `delete` 不就行了吗？事实上，要做到这一点是很难的。如下面的情况：

```
ObjectType* ptr = new ObjectType();
if(something_not_right())
    throw_exception();    // need delete here
delete ptr;
```

如果 if 条件成立，那么指针将得不到 delete，所以要在 throw_exception() 前也加上delete语句。然而，在较大的工程中，这样的异常会有很多，很难一个一个维护，甚至还有没有捕获的异常发生，就为手动回收指针带来了很大的困难。基于这样的考虑，能够自动管理内存，确定变量生存期的智能指针就应运而生了。

从 C++ 98 讲起 —— auto_ptr

早在 c++ 98 就有一种智能指针—— auto_ptr，它基本上实现了对内存的自动管理，但是也有非常严重的缺陷，并且在 c++ 11 中被摒弃（虽然它还在标准库中，但是非常不推荐使用它）。在这里仍然要介绍它的原因是，它和 c++ 11 中的智能指针相比，概念和实现都更简单，容易上手，而且它是对智能指针的思想 RAII （Resource Acquisition Is Initialization）的一个实现，可以更好的理解智能指针的设计思路。

auto_ptr 使用

1. 声明与构造

```
template<class T> class auto_ptr;
template<> class auto_ptr<void>;     // 声明

explicit auto_ptr(X* p = 0);     // 指向 p 管理的对象
auto_ptr(auto_ptr& r);           // 接管 r 管理的对象，且 r 失去管理权

template<class Y>                // 针对能隐式转换为 T* 类型的 Y*
auto_ptr<auto_ptr<Y>& r);

template<class Y>                // 接管 auto_ptr_ref<Y> 类型的 m 管理的对象
auto_ptr(auto_ptr_ref<Y> m);
```

2. 析构函数

```
~auto_ptr(); // deprecated
```

3. copy 与 assign

```
auto_ptr& operator=(auto_ptr& r);

template<class Y>
auto_ptr& operator=(auto_ptr<Y>& r);

auto_ptr& operator=(auto_ptr_ref m);
```

   会让 rhs 失去对象的管理权。

4. 隐式转换

```
template<class Y>                // 转换为 auto_ptr_ref<Y>
operator auto_ptr_ref<Y>();
```

```
template<class Y>               // 转换为 auto_ptr<Y>
operator auto_ptr<Y>();
```

5. 其它函数

```
T* get() const;                  // 返回 *this 管理的对象指针

T& operator*() const;            // 返回 *this 管理的对象 （&）
T* operator->() const;           // 返回 *this 管理的对象指针 （*）

void reset(T* p = 0);            // reset *this，让它接管 p 指向对象

T* release();                   // 返回 *this 管理指针，交出它的管理权，并置nullptr
```

auto_ptr copy/assign 策略分析

auto_ptr 最大的问题就是在copy & assign 的时候会转交管理权（管理权 lhs -> rhs），这样就导致 lhs 变成了 nullptr。举例来说：

```
auto_ptr<int> p1(new int(1));
auto_ptr<int> p2(p1);
cout << *p1 << endl;

auto_ptr<int> p3=p1;
cout << *p1 << endl;
```

这是最明显的例子，两处 p1 在分别copy / assign 后变成了空指针。还有更不容易被发现的情况：

```
void func(auto_ptr<int> ap)
{
cout << *ap << endl;
}

auto_ptr<int> ap(new int(1));
func(ap);
cout << *ap1 << endl;
```

当 auto_ptr 作为函数参数值传递的时候，会新建一个 auto_ptr 进行拷贝构造，此时实参的管理权移交给了形参，当函数返回时，实参已经是空指针了。

如果引用传递 auto_ptr， 虽然没有上述的问题，但是函数就是一个不安全的过程，它有可能会用赋值等操作再次移交它的管理权。

所以，auto_ptr 还不能用作函数参数传递。auto_ptr 的所有局限性归根究底都是因为它的 copy / assign 策略导致的，我们不妨看一下它这一部分的源码实现：

```
 _Myt& operator=(_Myt& _Right)
  { // assign compatible _Right (assume pointer)
    reset(_Right.release());
    return (*this);
  }

template<class _Other>
_Myt& operator=(auto_ptr<_Other>& _Right)
{ // assign compatible _Right (assume pointer)
    reset(_Right.release());
    return (*this);
}

_Myt& operator=(auto_ptr_ref<_Ty> _Right)
{ // assign compatible _Right._Ref (assume pointer)
    _Ty *_Ptr = _Right._Ref;
    _Right._Ref = 0;    // release old
    reset(_Ptr);    // set new
    return (*this);
}
```

可以很明显的看到，`reset(_Right.release());` 和 `_Right._Ref = 0;` 都解除了 rhs 的管理权。

auto_ptr 对象生命周期的管理—— RAII

到这里，其实还没有讲到 auto_ptr 为什么可以自动管理对象的生命周期，可以不用手动 delete 呢？在源码中，没有特殊的成员/机制来管理，仅仅是把对象的指针放入 `_Myptr` 中，但是为什么用一个对象进行封装，就可以做到这一点呢？

这就在于之前提到的 RAII 思想，简单概括就是用对象来管理资源（Use objects to manage resources），它是由 Bjarne Stroustrup 首先提出的。我们知道在函数内部的一些成员是放置在栈空间上的，当函数返回时，这些栈上的局部变量就会立即释放空间。要想确保析构函数在对象释放前被执行，可以把对象放到这个程序的栈上。因为stack winding会保证它们的析构函数都会被执行。RAII就利用了栈里面的变量的这一特点。

RAII 的一般做法是这样的：在对象构造时获取资源，接着控制对资源的访问使之在对象的生命周期内始终保持有效，最后在对象析构的时候释放资源。借此，我们实际上把管理一份资源的责任托管给了一个存放在栈空间上的局部对象。

Unique_ptr

接下来，我们介绍 C++ 11 中引入的一种智能指针 `std::unique_ptr` 。它在很多方面和 `auto_ptr` 类似，可以说是淘汰 `std::auto_ptr` 的替代品。它并没有引入新的机制来实现智能内存的管理，仍然用指针来管理对象；同 `auto_ptr` 一样，它享有对象的独享权，不能用两个 `unique_ptr` 指向同一个对象。

unique_ptr 使用

1. 声明与构造

```
  constexpr unique_ptr();                  // 构造没有管理任何资源的空的指针
  constexpr unique_ptr(nullptr_t);          // nullptr_t 是 C++11 的新类型，nullptr
是它的字面值

  explicit unique_ptr(pointer p);          // 显式声明，构造管理 p 指向资源的对象

  unique_ptr(pointer p,                    // 构造管理 p 指向资源的对象
      typename conditional<is_reference<Deleter>::value,  // 同时把释放的函数设为 d
      Deleter, const Deleter&>::type d);

  unique_ptr(pointer p,                     // 和上一个声明类似
      typename remove_reference<Deleter>::type&& d);     // 主要针对 (3) 的 d 是右值
引用类型时的                                                    // 重载

  unique_ptr(unique_ptr&& u);               // 利用已有对象构造，注意，这是语义移动


  template<class U, class E>                // 和上面一样，主要针对隐式转化的构造
  unique_ptr(unique_ptr<U, E>&& u);

  template<class U>                         // 把 auto_ptr 变成 unique_ptr
  unique_ptr(auto_ptr<U>&& u);
```

补充：对第三个语法的解释

- conditional：

  template< bool B, class T, class F > struct conditional; 中， 如果 B 是 true, 则定义为 T 类型， 如果 B 是 false, 则定义为 F 类型。

- is_reference：在编译期，判断某个类型是否是引用类型。

- remove_reference：在编译期移除某个类型的引用符号。如：`remove_reference<int&>::type` 类型就是 `int` 类型。

2. 析构函数

```
~unique_ptr();
```

如果有管理的资源，那么使用 `deleter` 销毁管理的资源。

3. copy 与 assign

```
  unique_ptr& opertor=(unique_ptr&& r);           // 把 r 管理的资源交给 *this,右值引
用。
                                                 // 如果 *this 本身就有管理的资源，那么
先用
                                                 // deleter 释放
  template<class U, class E>                     // 针对隐式转换
  unique_ptr& operator=(unique_ptr<U, E>&& r);

  unique_ptr& operator=(nullptr_t);               // 相当于调用 reset，将管理资源的指
针置 nullptr
```

4. 其它函数

```
  pointer release();                             // 移交出 *this 管理资源的指针。如果
*this 没有管
                                                 //     理资源，则返回 nullpt
r

  void reset(pointer ptr = pointer());           // 设置 *this 管理 ptr 指向的资源，
如果 *this 本
                                                 // 身有管理的资源，则先用 de
leter 释放该资源

  void swap(unique_ptr& other);                  // 将 *this 管理的资源和 other管理的
资源进行交换

  pointer get() const;                           // 获取 *this 管理资源的指针，如果没
有则返回
                                                 // nullptr

  Deleter& get_deleter();                        // 获取 *this 绑定的 deleter
  const Deleter& get_deleter() const;

  explicit operator bool() const;                // 判断是否 *this 管理有资源

  typename std::add_lvalue_reference<T>::type    // 获取 *this 所管理的对象的左值引用
    operator*() const;
  pointer operator->() const;                    // 获取 *this 所管理的对象的指针
                                                 // 从这两个重载可以看出这个类表现的像个指
针
```

部分源码分析

1. `unique_ptr` 的成员

```
private:
    typedef std::tuple<typename _Pointer::type, _Dp>  __tuple_type;
    __tuple_type                                      _M_t;

public:
    typedef typename _Pointer::type   pointer;
    typedef _Tp                       element_type;
    typedef _Dp                       deleter_type;
```

成员中的核心就是一个tuple二元组，存放所管理的资源的指针以及 deleter 对象。

public 的成员有：所管理资源的指针的类型，所管理资源的了类型，以及负责销毁资源的 deleter 的类型。

2. 析构函数的实现

```
// Destructor, invokes the deleter if the stored pointer is not null.
  ~unique_ptr()
  {
    auto& __ptr = std::get<0>(_M_t);

    if (__ptr != nullptr)
      get_deleter()(__ptr);

    __ptr = pointer();
  }
```

根据上文可以知道， `get<0>(_M_t)` 得到的就是 `*this` 管理资源的指针的引用。析构函数中，如果管理资源的指针不为空，那么就调用绑定的 `deleter` 来销毁管理的资源。最后再重置管理的指针。

由此可以看到，uniqe_ptr 只要调用了析构函数，它管理的资源一定可以得到释放。

3. copy / assign 的实现

```
  // Assignment.
  /** @brief Move assignment operator.
   *
   * @param __u  The object to transfer ownership from.
   *
   * Invokes the deleter first if this object owns a pointer.
   */
  unique_ptr& operator=(unique_ptr&& __u)
```

```
  {
    reset(__u.release());
    get_deleter() = std::forward<deleter_type>(__u.get_deleter());
    return *this;
  }

  /** @brief Assignment from another type.
   *
   * @param __u  The object to transfer ownership from, which owns a
   *             convertible pointer to a non-array object.
   *
   * Invokes the deleter first if this object owns a pointer.
   */
  template<typename _Up, typename _Ep>
    typename enable_if< __and_<
    is_convertible<typename unique_ptr<_Up, _Ep>::pointer, pointer>,
    __not_<is_array<_Up>>
    >::value,
      unique_ptr&>::type operator=(unique_ptr<_Up, _Ep>&& __u)
  {
    reset(__u.release());
    get_deleter() = std::forward<_Ep>(__u.get_deleter());
    return *this;
  }

  /// Reset the %unique_ptr to empty,invoking the deleter if necessary.
  unique_ptr& operator=(nullptr_t)
  {
    reset();
    return *this;
  }
```

可以看到，在右值引用中，`reset(__u.release());`这一句话就实现了资源管理权的转换。

为了进一步说明`unique_ptr`赋值中右值引用和`auto_ptr`赋值中左值引用的区别，
从`cppreference`上改了一个小例子：

```
#include <iostream>
#include <memory>

struct Foo {
    Foo() { std::cout << "Foo\n"; }
    ~Foo() { std::cout << "~Foo\n"; }
};

int main()
{
    std::unique_ptr<Foo> p1;
```

```
    {
        std::cout << "Creating new Foo...\n";
        std::unique_ptr<Foo> p2(new Foo);

        // p1 = p2; // Error ! can't copy unique_ptr
        //unique_ptr& opertor=(unique_ptr&& r);
        p1 = std::move(p2);
        std::cout << "About to leave inner block...\n";

        // Foo instance will continue to live,
        // despite p2 going out of scope
    }

    // unique_ptr& operator=(nullptr_t);
    std::cout << "Creating new Foo...\n";
    std::unique_ptr<Foo> p3(new Foo);
    std::cout << "Before  p3 = nullptr...\n";
    p3 = nullptr;
    std::cout << "After  p3 = nullptr...\n";

    std::cout << "About to leave program...\n";
}
```

`unique_ptr` 不能直接使用 `p1 = p2` 这样的语句进行赋值，因为重载 `=` 时使用的是右值引用。要先用 `std::move(p2)` 来移动语义，得到 p2 的右值引用，才可以进行赋值。这就是和 `auto_ptr` 最大的不同之处。

上面程序的执行结果：

```
Creating new Foo...
Foo
About to leave inner block...
Creating new Foo...
Foo
Before  p3 = nullptr...
~Foo
After  p3 = nullptr...
About to leave program...
~Foo
```

这个例子还体现了两点：当 `unique_ptr` 过了作用域的时候，它所管理的资源仍然存活；当把 p3 置为 nullptr 的时候，它管理的资源得到了释放。

unique_ptr 的总结

总体上说， `std::unique_ptr` 和 `std::auto_ptr` 实现的功能是类似的，但是前者使用了 C++11 中的右值引用的特性，使得它们有如下的几点区别：

- auto_ptr 有拷贝语义，拷贝后源对象变得无效；unique_ptr 则无拷贝语义，但提供了移动语义。
- auto_ptr 不可作为容器元素，unique_ptr 可以作为容器元素
- unique_ptr 还可以设置销毁资源的 deleter，这就使得它不仅可以管理内存资源，也可以管理其它的需要释放的资源

Shared_ptr

简介

std::shared_ptr是通过指针保持对象共享所有权的智能指针，多个 shared_ptr 对象可占有同一对象。 下列情况之一出现时销毁对象并解分配其内存：

- 最后剩下的占有对象的 shared_ptr 被销毁；
- 最后剩下的占有对象的 shared_ptr 被通过 operator= 或 reset() 赋值为另一指针；
- 用 delete 或在构造期间提供给 shared_ptr 的定制删除器销毁对象。

shared_ptr 能在存储指向一个对象的指针时共享另一对象的所有权。此特性能用于在占有其所属对象时，指向成员对象。存储的指针为 get() 、解引用及比较运算符所访问。被管理指针是在 use_count 抵达零时传递给删除器者。 shared_ptr也可不占有对象，该情况下称它为空 (empty) (空 shared_ptr 可拥有非空存储指针，若以别名使用构造函数创建它)。

C++中的定义

shared_ptr需要在memory的头文件下才能调用，它的类型是template< class T > class shared_ptr;

memory中shared_ptr的定义

可以看出：shared_ptr和weak_ptr都由一个父类 _Ptr_base继承得到，_Ptr_base负责接收传入的对象；而shared_ptr内部包含两个指针，一个指向对象ptr，另一个指向控制块(control block)，控制块中包含一个引用计数和可选的deleter、allocator。

- 这是带了deleter和allocator的为_Ux类型对象生成shared_ptr的函数

```
template<class _Ux,
    class _Dx,
    class _Alloc>
        void _Resetp(_Ux *_Px, _Dx _Dt, _Alloc _Ax)
    {    // release, take ownership of _Px, deleter _Dt, allocator _Ax
        typedef _Ref_count_del_alloc<_Ux, _Dx, _Alloc> _Refd;
        typename _Alloc::template rebind<_Refd>::other _Al = _Ax;

        _TRY_BEGIN    // allocate control block and reset
            _Refd *_Ptr = _Al.allocate(1);
        new (_Ptr) _Refd(_Px, _Dt, _Al);
        _Resetp0(_Px, _Ptr);
        _CATCH_ALL    // allocation failed, delete resource
            _Dt(_Px);
        _RERAISE;
        _CATCH_END
    }
```

Shared_ptr的引用计数机制

由于shared_ptr的多个指针共同管理对象的机制，为了达到智能管理的效果，引入引用计数机制。

引用计数指的是，所有管理同一块内存的shared_ptr，都共享一个引用计数器，每当一个shared_ptr被赋值（或拷贝构造）给其它shared_ptr时，这个共享的引用计数器就加1，当一个shared_ptr析构或者被用于管理其它裸指针时，这个引用计数器就减1，如果此时发现引用计数器为0，那么说明它是管理这个指针的最后一个shared_ptr了，于是我们释放指针指向的资源。

- 例如：

```
int main()
{
    std::shared_ptr<int> p1(new int(1));// count(int(1))++
    std::shared_ptr<int> p2 = std::make_shared<int>(2);// count(int(2))++
    p1.reset(new int(3));// count(int(1))-- 析构int(1) 同时count(int(3))++
    std::shared_ptr<int> p3 = p1;//count(int(3))++
    p1.reset();//count(int(3))--
    p3.reset();//count(int(3))--,为0，析构int(3)
}
```

- 而在c++源码中，引用计数由各种各样参数的_Reset方法来维护

```
 template<class _Ty2>
    void _Reset(const _Ptr_base<_Ty2>& _Other)
    {    // release resource and take ownership of _Other._Ptr
        _Reset(_Other._Ptr, _Other._Rep, false);
    }

    template<class _Ty2>
    void _Reset(const _Ptr_base<_Ty2>& _Other, const _Static_tag&)
    {    // release resource and take ownership of _Other._Ptr
        _Reset(static_cast<_Elem *>(_Other._Ptr), _Other._Rep);
    }

    template<class _Ty2>
    void _Reset(const _Ptr_base<_Ty2>& _Other, const _Dynamic_tag&)
    {    // release resource and take ownership of _Other._Ptr
        _Elem *_Ptr = dynamic_cast<_Elem *>(_Other._Ptr);
        if (_Ptr)
            _Reset(_Ptr, _Other._Rep);
        else
            _Reset();
    }
```

Shared_ptr的赋值方法

1. 默认构造方法，最多三个参数：对象类型；自定义的delete方法，自定义的allocate方法

```
std::shared_ptr<int> p1;
std::shared_ptr<int> p2 (nullptr);
std::shared_ptr<int> p3 (new int);
std::shared_ptr<int> p4 (new int, std::default_delete<int>());
std::shared_ptr<int> p5 (p4);
std::shared_ptr<int> p6 (std::move(p5));
std::shared_ptr<int> p7 (std::unique_ptr<int>(new int));
std::shared_ptr<C> obj (new C);
std::shared_ptr<int> p8 (obj, obj->data);
```

2. = 和 make_shared 和 move 方法

```
std::shared_ptr<int> foo;
std::shared_ptr<int> bar (new int(10));
foo = bar;                  // foo和bar共享一个对象
bar = std::make_shared_ptr
std::unique_ptr<int> unique (new int(30));
foo = std::move(unique);  // foo改变为unique_ptr对象，原先的10被回收
```

3. static_pointer_cast 静态类型转换

```
struct A {
 static const char* static_type;
 const char* dynamic_type;
 A() { dynamic_type = static_type; }
};
struct B: A {
 static const char* static_type;
 B() { dynamic_type = static_type; }
};
const char* A::static_type = "class A";
const char* B::static_type = "class B";
int main () {
 std::shared_ptr<A> foo;
 std::shared_ptr<B> bar;
 foo = std::make_shared<A>();
 bar = std::static_pointer_cast<B>(foo);
}
```

4. dynamic_pointer_cast 动态类型转换

```
struct A {
```

```
 static const char* static_type;
 const char* dynamic_type;
 A() { dynamic_type = static_type; }
};
struct B: A {
 static const char* static_type;
 B() { dynamic_type = static_type; }
};

const char* A::static_type = "class A";
const char* B::static_type = "class B";

int main () {
 std::shared_ptr<A> foo;
 std::shared_ptr<B> bar;

 bar = std::make_shared<B>();
 foo = std::dynamic_pointer_cast<A>(bar);
}
```

5. const_pointer_cast 常量类型转换

```
int main () {
std::shared_ptr<int> foo;
std::shared_ptr<const int> bar;
foo = std::make_shared<int>(10);
bar = std::const_pointer_cast<const int>(foo);
}
```

Shared_ptr与unique_ptr,weak_ptr的不同

1. shared_ptr是引用计数的智能指针，而unique_ptr不是。可以有多个shared_ptr实例指向同一块动态分配的内存，当最后一个shared_ptr离开作用域时，才会释放这块内存。unique_ptr意味着所有权。单个unique_ptr离开作用域时，会立即释放底层内存。
2. c++默认的智能指针应该是unique_ptr。
3. weak_ptr是为了解决shared_ptr可能存在的循环计数的问题而提出的一种智能指针，通常与shared_ptr配合使用。如果将一个shared_ptr指针赋值给weak_ptr指针，对shared_ptr指针本身不会造成任何影响。对于weak_ptr指针来说，却可以通过一些方法来探测被赋值过来的shared_ptr指针的有效性，同时weak_ptr指针也可以间接操纵shared_ptr指针。

Shared_ptr 总结

shared_ptr关键在于共享的对象，只要对象还被引用就不会被释放；这比只能作用单一对象的unique_ptr来说便捷了许多。不过就算有了引用计数机制也不能保证没有内存泄漏，如果出现了循环引用，依然无法有效回收内存，这时候就需要weak_ptr的协助

Shared_ptr 的两种简单实现

在 C++ 标准库中的 `std:shared_ptr` 中，使用了引用计数来实现多个指针对同一个对象的共享。为了能更好的理解这个概念，我们用两种常用的方法来简单的实现 `shared_ptr` 。

在引用计数中，最基本的操作如下：

- 每次创建类的新对象时，初始化指针并将引用计数置为1
- 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
- 对一个对象进行赋值时，赋值操作符减少左操作数（lhs）所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数（rhs）所指对象的引用计数
- 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

对引用计数的实现由两种策略，一是引入辅助类，二是使用句柄类。

辅助类

```
// 使用辅助类实现引用计数
template <typename T>
class SmartPtr;

template <typename T>
class U_Ptr     //辅助类
{
private:
    friend class SmartPtr<T>;

    U_Ptr(T *ptr) :p(ptr), count(1) { }

    ~U_Ptr() { delete p; }
    int ref_count;   // 引用计数

    T *p;            // 对象指针
};

template <typename T>
class SmartPtr   //智能指针类
{
public:
    SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }
    SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) { ++rp->ref_count; }
    // 和 shared_ptr 一样，重载了 '='
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->ref_count;
        if (--rp->ref_count == 0)
```

```
                delete rp;
            rp = rhs.rp;
            return *this;
        }

        T & operator *()        //重载*操作符
        {
            return *(rp->p);
        }
        T* operator ->()       //重载->操作符
        {
            return rp->p;
        }

        ~SmartPtr() {
            if (--rp->ref_count == 0)
                delete rp;
            else
            std::cout << "ref_count = " << rp->ref_count << std::endl;
        }
    private:
        U_Ptr<T> *rp;  //辅助类对象指针
    };
```

辅助类 `U_ptr` 的所有成员皆为private，因为它只为智能指针所服务。这个辅助类含有两个数据成员：计数**ref_count**与对象指针。总的来说，就是用辅助类来以封装使用计数与基础对象指针。

这种方案的缺点是每个含有指针的类的实现代码中都要自己控制引用计数，比较繁琐。特别是当有多个这类指针时，维护引用计数比较困难。

句柄类

参考了 C++ Primer 中的实现：

```
template<class T> class Handle
{
public:
    Handle(T *p = 0):ptr(p), use(new size_t(1)){}
    T& operator*();
    T* operator->();

    const T& operator*()const;
    const T* operator->()const;

    Handle(const Handle& h):ptr(h.ptr), use(h.use)
    { ++*use; }
```

```
    Handle& operator=(const Handle&);
    ~Handle() { rem_ref(); }

private:
    T* ptr;             // 共享的对象指针
    size_t *use;        // ref_count
    void rem_ref()
    {
        if (--*use == 0)
        {
            delete ptr;
            delete use;
        }
    }
};

template<class T>
inline Handle<T>& Handle<T>::operator=(const Handle &rhs)
{
    ++*rhs.use;         //protect against self-assignment
    rem_ref();         //decrement use count and delete pointers if needed
    ptr = rhs.ptr;
    use = rhs.use;

    return *this;
}

template<class T>
inline T& Handle<T>::operator*()
{
    if(ptr) return *ptr;
    throw std::runtime_error
        ("dereference of unbound Handle");
}

template<class T>
inline T* Handle<T>::operator->()
{
    if(ptr) return ptr;
    throw std::runtime_error
        ("access through of unbound Handle");
}

template<class T>
inline const T& Handle<T>::operator*()const
{
    if(ptr) return *ptr;
    throw std::runtime_error
        ("dereference of unbound Handle");
```

```
}

template<class T>
inline const T* Handle<T>::operator->()const
{
    if(ptr) return ptr;
    throw std::runtime_error
        ("access through of unbound Handle");
}
```

与使用辅助类不同的是，句柄 Handle 只有一个类就实现了引用计数；并且，对于 `ref_count` ，在辅助类中的成员是 int / size_t 类型的，而在这里，是 int/ *size_t* 类型的，因为要保证多个指针共享对象的引用计数一致。

一个使用句柄类的例子：

```
int main()
{
    Handle<int> hp(new int(42));
    {
        //new scope
        Handle<int> hp2 = hp;
        std::cout<< *hp <<"  "<< *hp2 << std::endl;        //  输出两个42
        *hp2 = 10;                                  //  修改 hp2 的同时也修改了hp
    }                                              //  hp2 退出作用域，但是引用计数没到0，对象不释放

    std::cout<< *hp << std::endl;                  //  输出为 10
    return 0;
}
```

Weak_ptr

`std::weak_ptr` 是一种比较特殊的智能指针，它的名字也暗示这一点，它是一种弱指针：弱指针的引用并不能对它的对象进行内存管理。那么，它有什么用呢？

在上文的 `std::shared_ptr` 中，如果创建两个 shared_ptr 并且让它们互相引用，那么就会产生循环引用，根据引用计数的特性，它们的计数就永远不会为0，导致资源不能释放。weak_ptr 就可以解决这样的问题。

```
// 一个循环引用的例子
#include <memory>
#include <iostream>
using namespace std;

class Parent;
class Child;
typedef std::shared_ptr<Parent> parent_ptr;
typedef std::shared_ptr<Child> child_ptr;

class Child
{
public:
    Child()
    {
        cout << "Child ..." << endl;
    }
    ~Child()
    {
        cout << "~Child ..." << endl;
    }
    parent_ptr parent_;
};

class Parent
{
public:
    Parent()
    {
        cout << "Parent ..." << endl;
    }
    ~Parent()
    {
        cout << "~Parent ..." << endl;
    }
    child_ptr child_;
};
```

```
int main()
{
    parent_ptr parent(new Parent);
    child_ptr child(new Child);
    parent->child_ = child;
    child->parent_ = parent;

    return 0;
}
// 程序运行结束，但是两个类的析构函数都没有被调用
```

weak_ptr 使用

1. weak_ptr 的简单定义

```
    template<typename T> class weak_ptr
    {
    public:
        template <typename Y>
        weak_ptr(const shared_ptr<Y> &r);

        weak_ptr(const weak_ptr &r);            // weak_ptr 只能通过 shared_ptr 或者
                                                // 另一个 weak_ptr 来构造

        template<class Y>
        weak_ptr &operator=( weak_ptr<Y> && r );

        template<class Y>
        weak_ptr &operator=(shared_ptr<Y> const &r);
                                                // 支持拷贝和赋值，但是不会影响对应的 sh
ared_ptr                                                // 内部对象的计数
                                                // 没有重载 *，->，不能操作被管理的资源

        ~weak_ptr();

        bool expired() const;                   // 用于判断管理的资源是否已经被释放
        shared_ptr<T> lock() const;             // 用于返回强引用指针：shared_ptr
          int use_count() const;                // 用于返回与 shared_ptr 共享的对象的
引用计数
    };
```

2. weak_ptr 的弱引用机制

   上文的例子中，只需要把 parent 类中的成员改成：

```
class Parent
{
public:
    std::weak_ptr<parent> child_;
};
```

就可以不出现循环引用的问题了。那么，weak_ptr 是怎么样做到这一点的呢？

这就是它的弱引用机制：弱引用仅仅是对在生存期的对象的引用，不进行内存管理，不修改该对象的引用计数。它的构造和析构不会引起引用记数的增加或减少。

weak_ptr 在功能上类似于普通指针, 然而一个比较大的区别是, 弱引用能检测到所管理的对象是否已经被释放，当被释放时，会自动变成 nullptr ，从而避免访问非法内存。另一方面，它还可以通过 lock 函数来变成 强引用指针，从而操作内存，还不会引起计数的混乱。

C++ boost库中的智能指针

引言

boost是非C++标准库（STL），但boost库中也有着大量关于智能指针的定义和使用。在boost中的智能指针包括 `boost::scoped_ptr` ， `boost::scoped_array` ，`boost::shared_ptr` ，`boost::weak_ptr` ，`boost::intrusive_ptr` ，`boost::local_shared_ptr` ,而且其中的 `boost::shared_ptr` 和 `boost::weak_ptr` 已经加入了C++11标准。

相关指针的介绍

1. `boost::scoped_ptr`

   一个scoped_ptr独占一个动态分配的对象。这个类似于unique_ptr，它们都只能处理一个对象，一个unique_ptr不能传递它所包含的对象的所有权到另一个unique_ptr。不过scoped_ptr比起unique_ptr要简单，scoped_ptr只是简单保存和独占一个内存地址。一旦用另一个地址来初始化，这个动态分配的对象将在析构阶段释放。

   - 实例

   ```
   #include <boost/scoped_ptr.hpp>
   #include <iostream>
   int main()
   {
   boost::scoped_ptr<int> i(new int);
   *i = 1;
   *i.get() = 2;
   i.reset(new int);
   }
   ```

2. `boost::scoped_array`

   这个正如名称中的array，scoped_array是动态分配的数组对象，特点与scoped_ptr一样，都只是保存和占有内存地址。此外它重载了[]运算符，即确实可以像数组一样访问它的元素。

   - 实例

   ```
   #include <boost/scoped_array.hpp>
   #include <iostream>
   int main()
   {
     boost::scoped_array<int> i(new int[2]);
     *i.get() = 1;
     i[1] = 2;
   ```

```
    i.reset(new int[3]);
}
```

3. `boost::shared_ptr`

   见c++ shared_ptr介绍

4. `boost::weak_ptr`

   见c++ weak_ptr介绍

5. `boost::intrusive_ptr`

   类似于shared_ptr，intrusive_ptr也存在引用计数机制，实际上官网上表示，除 非shared_ptr不能满足你的需求，不然推荐使用shared_ptr。intrusive_ptr的特别之处在于，它的提供计数函数接口（intrusive_ptr_add_ref和intrusive_ptr_release），要求用户自定义。通常使用instrusive_ptr的情况有：

   - 已经写好了内部引用计数器的代码，而我们又没有时间去重写它(或者不能获得那些代码)。
   - 你需要把 this 当作智能指针来使用。
   - shared_ptr的引用计数器分配严重影响了程序的性能。

6. `boost::local_shared_ptr`

   local_shared_ptr和shared_ptr几乎一样，唯一不同的是它的引用计数是用非原子操作更新的。因此，一个local_shared_ptr的所有的副本必须驻留在本地的单个线程。

   local_shared_ptr可以转换为shared_ptr，反之亦然。可以从shared_ptr对象创建local_shared_ptr对象，并且创建新的引用计数，也就是说可以两个local_shared_ptr引用相同的对象，但不共享相同的计数，可以安全地由两个不同的线程使用（不用担心，当只有一个local_shared_ptr计数归0的时候，并不会销毁对象）。

Rust 中的智能指针

Rust在过去的版本中有managed pointer（用~和@表示，类似C++的 `std::unique_ptr` 和 `std::shared_ptr` ，在现在的版本中已经被移除，现在rust只有raw pointer和reference。

引言：Rust 的不同点

与其他语言相同， Rust 同样使用堆和栈来存储对象。和C/C++语言不同的是，Rust提供了一套机制，使其在内存安全的前提下得到接近C/C++的性能。

1. 编译时检查

Rust 能够处理对空指针、悬垂指针的解引用，但和 Java 等语言在运行时处理不同， Rust 会在编译时检查这些问题并报错。

```
{
    let s1 = String::from("hello")
    let s2 = s1; // string "hello"'s ownership transferred to s2
    s1; // error! s1 is not valid
}
```

如上的 Rust 程序会在编译时报错，因为 s1 已经不可用。RAII机制也保证了未初始化的值会被使用。

```
{
    std::string a = "hello";
    auto b = std::move(a); // transferred string "hello" to b
    a; // it's still valid to access a, though it's in an unspecified state
}
```

这段程序在 C++ 中能够运行，但是a的值未知。

1. 手动管理变量生存周期

和 Java 等语言不同，Rust 不使用垃圾收集。变量的存储空间被分配在栈上，其生命周期静态决定的。当出了定义的范围后，变量会自动销毁，其存储空间被释放。

```
{
    let a = 3;
}
a; // error: no 'a' in the scope
```

如上的程序会出错，因为a并不在范围内。

引用

Rust中由于每个value都拥有一个owner，reference是这样一种机制，它在不夺走值的使用权的同时获得其使用权。

```
{
  let s1 = String::from("hello");
  let s2 = s1; // s1's ownership is transferred to s2, since String is not Copy
  let s3 = &s2; // s3 got s2's string, without taking its ownership
}
```

如上的程序由于String并非可复制，因此s2会获取该 `String` 的所有权，之后将无法使用s1(提示 `use of moved value` ）。

这是一种借用，当原变量失效时，不能存在任何指向其的引用，以防止悬垂引用的产生，如下的程序会在编译时出错：

```
let b;
{
    let a = 3;
    b = &a;
} // error, a out of scope while still borrowed
```

对竞态的保护

Rust中，一个值最多同时只能够有一个 `&mut T` 引用（可更改值的引用）或若干个 `&T` 引用（只读的引用），这种限制避免了在一处使用引用值前被另一处的程序更改的情况。

对悬垂引用的保护

和C++不同，Rust在编译阶段保证引用的安全，一个引用会保证有效性，即使用一个引用时，其指向的一定是一个有效的值。

```
fn gen_int() -> &i32
{
  let i1 = 1;
  let i2 = &i1;

  i2
} // i1 is released returning a dangling reference

let ii = gen_int(); // error!
```

如上的程序会在编译时报错。

Box<T>

如其名所提示的，`Box<T>`提供的是对堆上分配内存的最简单实现，创立Box对象时，会分配堆上的空间当Box对象销毁时，会将带有的object一并销毁。

使用`Box::new`可以创建一个指针，用`*`来获得其指向的内容。当a的生命周期结束时，堆上的内存会被释放。

```
{
    let a = Box::new(5);
    println!("a is {}", a);
}
```

由于`Box`独占所指向的内存空间，因此`Box`不能复制，但是可以使用`clone`来获得另一个样本。

```
{
    let a = Box::new(5);
    let mut b = Box::clone(&a);
    *b = *b + 3;
    println!("A is {}", a);
}

会输出`A is 5`.

`Box::new`会在堆上分配能够保存接受该数据的空间，并返回指向该空间的指针(`std::boxed::Box<T>`)。由于历史原因（继承了pre 1.0的~指针），Box实际类似于基本类型，不需要Deref trait，而是可以直接用`*`解引用，Box在Rust源码中的Deref定义如下（可见[网页](https://doc.rust-lang.org/src/alloc/boxed.rs.html#623)）：

```rust
#[stable(feature = "rust1", since = "1.0.0")]
impl<T: ?Sized> Deref for Box<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &**self
    }
}

#[stable(feature = "rust1", since = "1.0.0")]
impl<T: ?Sized> DerefMut for Box<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut **self
    }
```
```

```
}
```

由于对一个类实例 `x` 调用 `*`，实际上是调用 `*(x.deref())`。但Box的特殊地位使其能够独立于deref trait而对 `Box<T>` 解引用。

如果对非Box的类使用如上的定义的话，会因为在deref函数中调用自身而导致无限循环，使编译器报错。

另外，在目前的 `Rust` 的实现中，Box的析构是由编译器完成而非定义。
在 `alloc/boxed.rs` 中 `Box` 的 `Drop` 实现为空：

```
#[stable(feature = "rust1", since = "1.0.0")]
unsafe impl<#[may_dangle] T: ?Sized> Drop for Box<T> {
    fn drop(&mut self) {
        // FIXME: Do nothing, drop is currently performed by compiler.
    }
}
```

如果 `box.rs` 内容如下：

```
fn main() {
    let a = Box::new(4);
}
```

则生成的LLVM IR 代码为：

```
; box::main
; Function Attrs: uwtable
define internal void @_ZN3box4main17h412e9b3ce7074bf7E() unnamed_addr #1 {
start:
  %a = alloca i32*
; call alloc::heap::exchange_malloc
  %0 = call i8* @_ZN5alloc4heap15exchange_malloc17hbdd98bb5621d389dE(i64 4, i64 4)
  %1 = bitcast i8* %0 to i32*
  store i32 4, i32* %1
  store i32* %1, i32** %a
  br label %bb1

bb1:                                              ; preds = %start
; call core::ptr::drop_in_place
  call void @_ZN4core3ptr13drop_in_place17h7cd262323c9f371cE(i32** %a)
  br label %bb2

bb2:                                              ; preds = %bb1
  ret void
}
```

而在该析构函数中：

```
; core::ptr::drop_in_place
; Function Attrs: uwtable
define internal void @_ZN4core3ptr13drop_in_place17h7cd262323c9f371cE(i32**) unnamed_ad
dr #1 personality i32 (i32, i32, i64, %"unwind::libunwind::_Unwind_Exception"*, %"unwin
d::libunwind::_Unwind_Context"*)* @rust_eh_personality {
start:
  %personalityslot = alloca { i8*, i32 }
  br label %bb3

bb1:                                              ; preds = %bb3
  ret void

bb2:                                              ; preds = %bb4
  %1 = load { i8*, i32 }, { i8*, i32 }* %personalityslot
  resume { i8*, i32 } %1

bb3:                                              ; preds = %start
  %2 = load i32*, i32** %0, !nonnull !3
; call alloc::heap::box_free
  call void @_ZN5alloc4heap8box_free17hdf1d9b7042aa931eE(i32* %2)
  br label %bb1

bb4:                                              ; No predecessors!
  %3 = load i32*, i32** %0, !nonnull !3
; call alloc::heap::box_free
  call void @_ZN5alloc4heap8box_free17hdf1d9b7042aa931eE(i32* %3) #7
  br label %bb2
}
```

调用了 `alloc::heap::box_free` ，该函数负责将其堆上的空间释放：

```
#[cfg_attr(not(test), lang = "box_free")]
#[inline]
pub(crate) unsafe fn box_free<T: ?Sized>(ptr: *mut T) {
    let size = size_of_val(&*ptr);
    let align = min_align_of_val(&*ptr);
    // We do not allocate for Box<T> when T is ZST, so deallocation is also not necessa
ry.
    if size != 0 {
        let layout = Layout::from_size_align_unchecked(size, align);
        Heap.dealloc(ptr as *mut u8, layout);
    }
}
```

Rc<T>

`Box<T>` 拥有对数据的所有权，因此同一个数据最多只能有一个box指向该数据。

```
let x = Box::new(5);
let y = x;
println!("{}", x); // error here
```

原因和之前的 `String` 相同， `Box` 为了保证对数据的所有权，因此不能复制。

这里就有 `Rc<T>` 来满足这种需求。rc表示reference counted。这和C的 `std::shared_ptr` 类似，每一个指向该数据rc会给计数器加1，当计数器归0时，该数据被释放。

`Rc` 位于 `std::rc` 下，可以在程序中使用 `use std::rc::Rc;` 来方便使用。

```
  let b;
  {
    let a = Rc::new(5);
    b = Rc::clone(&a);
  }
  println!("a is {}", b); // prints a is 5
```

在 `rc.rs` 中Rc是这样定义的：

```
struct RcBox<T: ?Sized> {
    strong: Cell<usize>,
    weak: Cell<usize>,
    value: T,
}


pub struct Rc<T: ?Sized> {
    ptr: Shared<RcBox<T>>,
    phantom: PhantomData<T>,
}
```

`ptr::Shared` 是对Rust中 `*mut T` (类似于C中的 `T*` ,mut表示可变)的封装，本身不做内存管理。

`PhantomData` 使得 `Rc<T>` 的行为和 `T` 实例的行为一致，即可以调用其指向对象的方法，为了不和 `Rc` 自身的方法冲突，应该使用类似 `Rc::clone(&a)` 而不是 `a.clone()` 的形式调用 `Rc` 的方法。

```
struct St {
}
```

```
impl St {
    fn hello(&self) {
        println!("Hello world!");
    }
}

fn main() {
    let a = Rc::new(St {});
    a.hello(); // calls St::hello(&a)
}
```

和C++的 `shared_ptr` 类似的，RcBox封装该数据和其strong引用和weak引用的计数值，Rc为指向该RcBox的指针，当clone时，strong引用的计数加1.

```
impl<T> Rc<T> {
    #[inline]
    fn clone(&self) -> Rc<T> {
        self.inc_strong();
        Rc { ptr: self.ptr }
    }
}

trait RcBoxPtr<T: ?Sized> {
    #[inline]
    fn inc_strong(&self) {
        self.inner().strong.set(self.strong().checked_add(1).unwrap_or_else(|| unsafe {
 abort() }));
    }
}
```

当 `Rc<T>` 对象销毁时，会调用drop（类似C++的析构函数），将对应的计数器减1，当计数器归0时，销毁对应RcBox的内容。因为该 `drop_in_place` 函数并不回收RcBox的空间（仅调用了 `T` 类型的 `drop` ，如果有)，因此weak和strong元素仍然可用。

```
#[stable(feature = "rust1", since = "1.0.0")]
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    fn drop(&mut self) {
        unsafe {
            let ptr = self.ptr.as_ptr();

            self.dec_strong();
            if self.strong() == 0 {
                // destroy the contained object
                ptr::drop_in_place(self.ptr.as_mut());
```

```
                // remove the implicit "strong weak" pointer now that we've
                // destroyed the contents.
                self.dec_weak();

                if self.weak() == 0 {
                    Heap.dealloc(ptr as *mut u8, Layout::for_value(&*ptr));
                }
            }
        }
    }
}
```

Weak<T>

Rust在和 `Rc<T>` 的同时，提供了 `Weak<T>` ， `Weak<T>` 和C++的 `std::weak_ptr` 功能相同。 `Weak<T>` 指示临时所有权，其所指向的对象可能在别处被释放。

Weak对象可以通过对Rc调用downgrade而得到:

```
// a is a
let a : Rc<i32> = Rc::new(4);
let b : Weak<i32> = Rc::downgrade(&a);
```

Weak本身不能直接使用指向元素的值，必须将其转换成 `Rc<T>` 才可以使用。但由于Weak指向的元素可能已经被释放，因此实际上使用 `Option<Rc<T>>` 来指示转换的结果。

```
let b : Weak<i32> = Weak::new();
let c = Weak::upgrade(&b);
match c {
    Some(x) => println!("B is {}", x),
    None => println!("B is empty"),
}
```

对每一个 `Option<T>` 有两种可能：一种是 `Some(T)` ，表示存在，并且有一个值；另一种是 `None` ，表示没有值。

这对应着试图将 `Weak<T>` 转换成 `Rc<T>` 的两种结果：一种是指向的值还有效，得到 `Some(Rc<T>)` ，另一种情况是已经被释放，得到的结果就是 `None` 。具体实现如下：

```
    #[stable(feature = "rc_weak", since = "1.4.0")]
    pub fn upgrade(&self) -> Option<Rc<T>> {
        if self.strong() == 0 {
            None
        } else {
            self.inc_strong();
            Some(Rc { ptr: self.ptr })
        }
    }
```

为了实现这个功能， `RcBox<T>` 会在释放完指向内存空间之后仍然维护指向它的 `Weak` 指针数，当该计数值也降为0后才释放堆上分配的空间。

Arc<T>

Rust还提供了一种支持多线程使用的Reference Counted，`Arc<T>`，表示"Atomic Reference Counted"。其主要原理是使用原子操作来加减引用计数器，但其代价是程序的效率变低，因为每次修改操作都需要互斥。

其使用和实现与`Rc<T>`基本相同，同样使用`strong`和`weak`来指示指针数，但改为原子类型。

```
struct ArcInner<T: ?Sized> {
    strong: atomic::AtomicUsize,

    // the value usize::MAX acts as a sentinel for temporarily "locking" the
    // ability to upgrade weak pointers or downgrade strong ones; this is used
    // to avoid races in `make_mut` and `get_mut`.
    weak: atomic::AtomicUsize,

    data: T,
}

#[stable(feature = "rust1", since = "1.0.0")]
pub struct Arc<T: ?Sized> {
    ptr: Shared<ArcInner<T>>,
}
```

```
#[stable(feature = "rust1", since = "1.0.0")]
unsafe impl<#[may_dangle] T: ?Sized> Drop for Arc<T> {
    #[inline]
    fn drop(&mut self) {
        // Because `fetch_sub` is already atomic, we do not need to synchronize
        // with other threads unless we are going to delete the object. This
        // same logic applies to the below `fetch_sub` to the `weak` count.
        if self.inner().strong.fetch_sub(1, Release) != 1 {
            return;
        }

        // make sure all references are dropped before dropping the data
        atomic::fence(Acquire);

        unsafe {
            self.drop_slow();
        }
    }
}

#[inline(never)]
```

```rust
unsafe fn drop_slow(&mut self) {
    let ptr = self.ptr.as_ptr();

    // Destroy the data at this time, even though we may not free the box
    // allocation itself (there may still be weak pointers lying around).
    ptr::drop_in_place(&mut self.ptr.as_mut().data);

    if self.inner().weak.fetch_sub(1, Release) == 1 {
        atomic::fence(Acquire);
        Heap.dealloc(ptr as *mut u8, Layout::for_value(&*ptr))
    }
}
```

Rust 使用 atomic fence 机制来确保不会使用已经释放的数据。