C++ 工程实践经验谈第 2 季

陈硕 (giantchen@gmail.com)

最后更新 2013-10-11

版权声明

内容一览

1 为什么多线程读写 shared_ptr 要加锁？

我在《Linux 多线程服务端编程：使用 muduo C++ 网络库》第 1.9 节“再论 shared_ptr 的线程安全”中写道：

> （shared_ptr）的引用计数本身是安全且无锁的，但对象的读写则不是，因为 shared_ptr 有两个数据成员，读写操作不能原子化。根据文档[1]，shared_ptr 的线程安全级别和内建类型、标准库容器、std::string 一样，即：
>
> • 一个 shared_ptr 对象实体可被多个线程同时读取（文档例 1）；
>
> • 两个 shared_ptr 对象实体可以被两个线程同时写入（例 2），“析构”算写操作；
>
> • 如果要从多个线程读写同一个 shared_ptr 对象，那么需要加锁（例 3~5）。
>
> 请注意，以上是 shared_ptr 对象本身的线程安全级别，不是它管理的对象的线程安全级别。

书中接下来（p.18）介绍如何高效地加锁解锁。本文则具体分析一下为什么“因为 shared_ptr 有两个数据成员，读写操作不能原子化”使得“多线程读写同一个 shared_ptr 对象需要加锁”。这个在我看来显而易见的结论似乎也有人抱有疑问，那将导致灾难性的后果，值得我写这篇短文。本文以 boost::shared_ptr 为例，与 std::shared_ptr 可能略有区别。

1.1 shared_ptr 的数据结构

shared_ptr 是引用计数型（reference counting）智能指针，几乎所有的实现都采用在堆（heap）上放个计数值（count）的办法（除此之外理论上还有用循环链表的办法，不过我没有见到实例）。具体来说，shared_ptr<Foo> 包含两个成员，一个是指向 Foo 的指针 ptr，另一个是 cnt 指针[2]，指向堆上的 ref_count 对象。ref_count 对象有多个成员，具体的数据结构如图 1 所示，其中 deleter 和 allocator 是可选的。ref_count 是个 class template，其模板类型参数决定了 ptr 的类型，图中黑框部分是其基类（ref_count_base），与模板类型参数无关。这里省略了这些细节，没有写成 ref_count<T>。

[1] http://www.boost.org/doc/libs/release/libs/smart_ptr/shared_ptr.htm#ThreadSafety
[2] 其类型不一定是原始指针，有可能是 class 类型，但不影响这里的讨论。

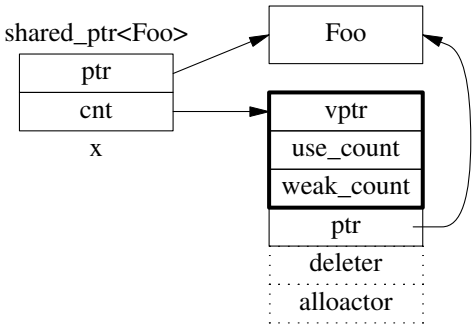
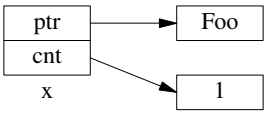


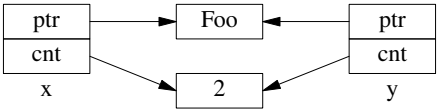
图 1 shared_ptr 的数据结构

为了简化并突出重点，后文只画出 use_count 的值：



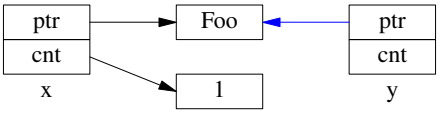
以上是 shared_ptr<Foo> x(new Foo); 对应的内存数据结构。

如果再执行 shared_ptr<Foo> y = x; 那么对应的数据结构如下。

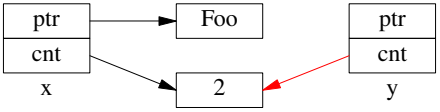


但是 y=x 涉及两个成员的复制，这两步拷贝不会同时（原子）发生[3]。

中间步骤 1，复制 ptr 指针（这一步是 atomic 的）：



中间步骤 2，复制 cnt 指针，导致引用计数加 1（这一步也是 atomic 的）：



步骤 1 和步骤 2 的先后顺序跟实现相关（因此步骤 2 里没有画出 y.ptr 的指向），我见过的都是先 1 后 2。

[3] 除非用一律用 atomic_load 和 atomic_store 来读写 shared_ptr，不过即便如此，Boost 目前（1.51）的实现也是用了 spinlock。

一般来说原子操作不具备可组合性（composable），两步原子操作合在一起就不再是原子（atomic）的了。既然 y=x 有两个步骤，如果没有 lock 保护，那么在多线程里就有 race condition。

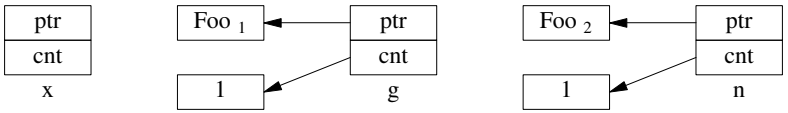
1.2 多线程无保护读写 shared_ptr 可能出现的 race condition

考虑一个简单的场景，有 3 个 shared_ptr<Foo> 对象 x、g、n：

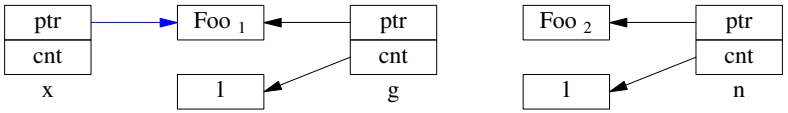
```
shared_ptr<Foo> g(new Foo); // 线程之间共享的 shared_ptr
shared_ptr<Foo> x; // 线程 A 的局部变量
shared_ptr<Foo> n(new Foo); // 线程 B 的局部变量
```

1.2.1 Race condition 1

一开始，各安其事。

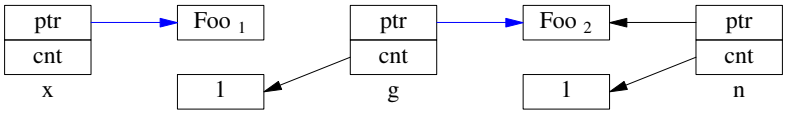


线程 A 执行 x = g; （即 read g），以下完成了步骤 1，还没来及执行步骤 2。这时切换到了 B 线程。

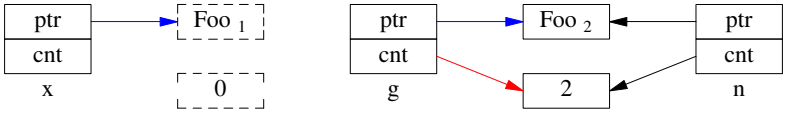


同时编程 B 执行 g = n; （即 write g），两个步骤一起完成了。

先是步骤 1：

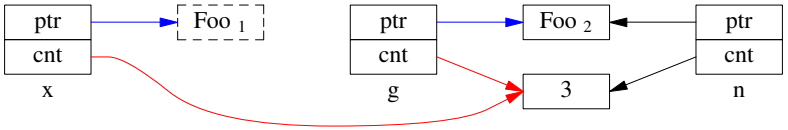


再是步骤 2：



这是 Foo1 对象已经销毁，x.ptr 成了空悬指针！

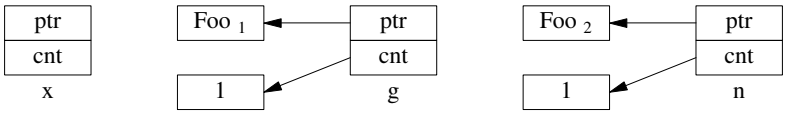
最后回到线程 A，完成步骤 2：



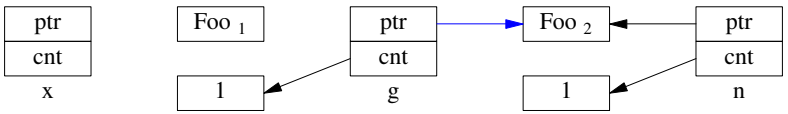
多线程无保护地读写 g，造成了“x 是空悬指针”的后果。这正是多线程读写同一个 shared_ptr 必须加锁的原因。

1.2.2 Race condition 2

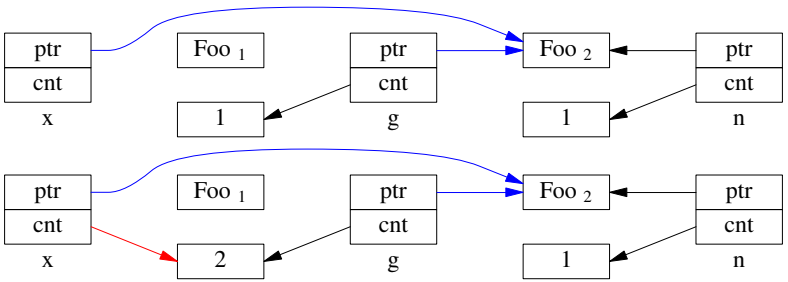
当然，race condition 远不止这一种，其他线程交织（interweaving）有可能会造成其他错误。例如：



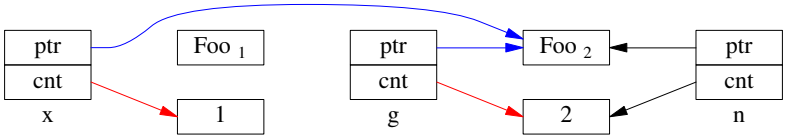
线程 B 先完成步骤 1：



然后切换到线程 A 完成步骤 1、2：



最后切回线程 B 完成步骤 2：



这造成了状态不一致，如果 g 和 n 先析构（或 reset() 或 assign），那么 x 就成了空悬指针。

思考，假如 shared_ptr 的 operator= 和/或 copy-ctor 实现是先复制 cnt（步骤 2）再复制 ptr（步骤 1），会有哪些 race condition？

1.2.3 复现 race condition 的代码

测试程序只能证明有 race condition，不能证明没有 race condition。万幸这个 race condition 比较容易复现，代码如下（https://gist.github.com/4688011）。

```
#include <muduo/base/Mutex.h>
#include <muduo/base/Thread.h>
#include <boost/shared_ptr.hpp>
#include <stdio.h>

boost::shared_ptr<int> g;
muduo::MutexLock mutex;

void read_g()
{
  boost::shared_ptr<int> x;
  long sum = 0;
  for (int i = 0; i < 1000 * 1000; ++i)
  {
    x.reset();
    {
      // muduo::MutexLockGuard lock(mutex); // 可以加上 mutex 以防止 race condition
      x = g;
    }
    sum += *x;
  }
  printf("sum = %ld\n", sum);
}

void write_g()
{
  for (int i = 0; i < 1000 * 1000; ++i)
  {
    boost::shared_ptr<int> n(new int(42));
    {
      // muduo::MutexLockGuard lock(mutex); // 可以加上 mutex 以防止 race condition
      g = n;
    }
  }
}

int main()
{
  g.reset(new int(42));
  muduo::Thread t1(read_g);
  muduo::Thread t2(write_g);
  t1.start();
  t2.start();
```

```
  t1.join();
  t2.join();
}
```

运行以上程序，十有九次会 core dump。把 mutex 加上去，就安全了。

1.3 杂项

1.3.1 shared_ptr 作为 unordered_map 的 key

如果把 boost::shared_ptr 放到 unordered_set 中，或者用于 unordered_map 的 key，那么要小心 hash table 退化为链表。[4]

直到 Boost 1.47.0 发布之前，unordered_set<shared_ptr<T> > 虽然可以编译通过，但是其 hash_value 是 shared_ptr 隐式转换为 bool 的结果。也就是说，如果不自定义 hash 函数，那么 unordered_set/map 会退化为链表。[5] [6]

Boost 1.51 在 boost/functional/hash/extensions.hpp 中增加了有关重载，现在只要包含这个头文件就能安全高效地使用 unordered_set<std::shared_ptr> 了。

这也是 muduo 的 examples/idleconnection 示例要自己定义 hash_value(const boost::shared_ptr<T>& x) 函数的原因（书第 7.10.2 节，p.255）。因为 Debian 6 Squeeze、Ubuntu 10.04 LTS 里的 Boost 版本都有这个 bug。

1.3.2 为什么图 1 中的 ref_count 也有指向 Foo 的指针？

shared_ptr<Foo> sp(new Foo) 在构造 sp 的时候捕获了 Foo 的析构行为。实际上 shared_ptr.ptr 和 ref_count.ptr 可以是不同的类型（只要它们之间存在隐式转换），这是 shared_ptr 的一大功能。下面分 3 点来说：

1. 无需虚析构；假设 Bar 是 Foo 的基类，但是 Bar 和 Foo 都没有虚析构。

```
shared_ptr<Foo> sp1(new Foo); // ref_count.ptr 的类型是 Foo*
shared_ptr<Bar> sp2 = sp1; // 可以赋值，自动向上转型（up-cast）
sp1.reset(); // 这时 Foo 对象的引用计数降为 1
```

此后 sp2 仍然能安全地管理 Foo 对象的生命期，并安全完整地释放 Foo，因为其 ref_count 记住了 Foo 的实际类型。

[4] http://stackoverflow.com/questions/6404765/c-shared-ptr-as-unordered-sets-key/12122314#12122314

[5] https://svn.boost.org/trac/boost/ticket/5216

[6] 这倒是一道很好的 Java 面试题，如果自己 override 的 Object.hashCode() 始终返回 1，会有什么后果？是否影响正确性？是否影响性能？

2. shared_ptr<void> 可以指向并安全地管理（析构或防止析构）任何对象；muduo::net::Channel class 的 tie() 函数就使用了这一特性，防止对象过早析构，见书 7.15.3 节。

```
shared_ptr<Foo> sp1(new Foo); // ref_count.ptr 的类型是 Foo*
shared_ptr<void> sp2 = sp1; // 可以赋值，Foo* 向 void* 自动转型
sp1.reset(); // 这时 Foo 对象的引用计数降为 1
```

此后 sp2 仍然能安全地管理 Foo 对象的生命期，并安全完整地释放 Foo，不会出现 delete void* 的情况，因为 delete 的是 ref_count.ptr，不是 sp2.ptr。

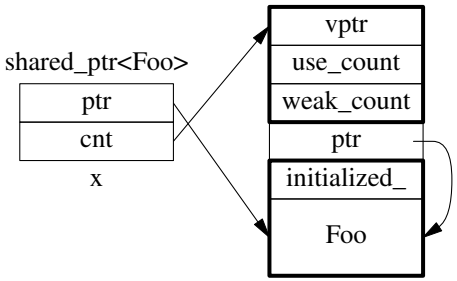
3. 多继承。假设 Bar 是 Foo 的多个基类之一，那么：

```
shared_ptr<Foo> sp1(new Foo);
shared_ptr<Bar> sp2 = sp1; // 这时 sp1.ptr 和 sp2.ptr 可能指向不同的地址，因为
                           // Bar subobject 在 Foo object 中的 offset 可能不为 0。
sp1.reset(); // 此时 Foo 对象的引用计数降为 1
```

但是 sp2 仍然能安全地管理 Foo 对象的生命期，并安全完整地释放 Foo，因为 delete 的不是 Bar*，而是原来的 Foo*。换句话说，sp2.ptr 和 ref_count.ptr 可能具有不同的值（当然它们的类型也不同）。

1.3.3 为什么要尽量使用 make_shared()？

为了节省一次内存分配，原来 shared_ptr<Foo> x(new Foo); 需要为 Foo 和 ref_count 各分配一次内存，现在用 make_shared() 的话，可以一次分配一块足够大的内存，供 Foo 和 ref_count 对象容身。数据结构是：



不过 Foo 的构造函数参数要传给 make_shared()，后者再传给 Foo::Foo()，这只有在 C++11 里通过 perfect forwarding 才能完美解决。

2 关于 std::set/std::map 的几个为什么

std::set/std::map （以下用 std::map 代表）是常用的关联式容器，也是 ADT (抽象数据类型)。也就是说，其接口（不是 OO 意义下的 interface）不仅规定了操作的功能，还规定了操作的复杂度（代价/cost)。例如 set::insert(iterator first, iterator last) 在通常情况下是 $O(N \log N)$，N 是区间 [first, last) 的长度；但是如果 [first, last) 已经排好序（在 key_compare 意义下），那么复杂度将会是 $O(N)$。

尽管 C++ 标准没有强求 std::map 底层的数据结构，但是根据其规定的时间复杂度，现在所有的 STL 实现都采用平衡二叉树来实现 std::map，而且用的都是红黑树。《算法导论（第 2 版)》第 12、13 章介绍了二叉搜索树和红黑树的原理、性质、伪代码，侯捷先生的《STL 源码剖析》第 5 章详细剖析了 SGI STL 的对应实现。本文对 STL 中红黑树（rb_tree）的实现问了几个稍微深入一点的问题，并给出了我的理解。本文剖析的是 G++ 4.7 自带的这一份 STL 实现及其特定行为，与《STL 源码剖析》的版本略有区别。为了便于阅读，文中的变量名和 class 名都略有改写（例如 _Rb_tree_node 改为 rb_tree_node)。本文不谈红黑树的平衡算法，在我看来这属于“旁枝末节”(见陈硕《谈一谈网络编程学习经验》对此的定义)，因此也就不关心节点的具体颜色了。

2.1 数据结构回顾

先回顾一下数据结构教材上讲的二叉搜索树的结构，节点（Node）一般有三个数据成员（left、right、data)，树（Tree）有一到两个成员（root 和 node_count)。

用 Python 表示：

```
class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

class Tree:
    def __init__(self):
        self.root = None
        self.node_count = 0
```

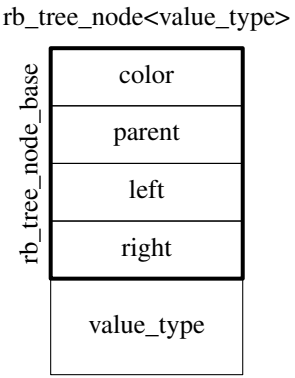
而实际上 STL rb_tree 的结构比这个要略微复杂一些，我整理的代码见 https://gist.github.com/4574621#file-tree-structure-cc。

2.1.1 节点

Node 有 5 个成员，除了 left、right、data，还有 color 和 parent。

```
C++ 实现，位于 bits/stl_tree.h
/**
 * Non-template code
 **/

enum rb_tree_color { kRed, kBlack };

struct rb_tree_node_base
{
  rb_tree_color       color_;
  rb_tree_node_base*  parent_;
  rb_tree_node_base*  left_;
  rb_tree_node_base*  right_;
};

/**
 * template code
 **/

template<typename Value>
struct rb_tree_node : public rb_tree_node_base
{
  Value value_field_;
};
```

见下图。



color 的存在很好理解，红黑树每个节点非红即黑，需要保存其颜色[7]；parent 的存在使得非递归遍历成为可能，后面还将再谈到这一点。

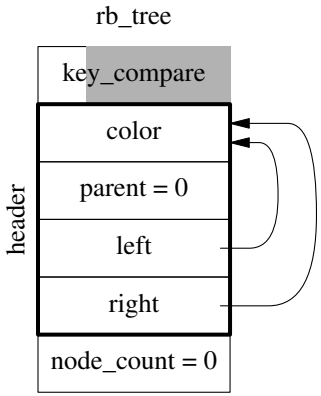
[7] 颜色只需要 1-bit 数据，一种节省内存的优化措施是把颜色嵌入到某个指针的最高位或最低位，Linux 内核里的 rbtree 是嵌入到 parent 的最低位。

2.1.2 树

Tree 有更多的成员，它包含一个完整的 rb_tree_node_base（color、parent、left、right），还有 node_count 和 key_compare 这两个额外的成员。

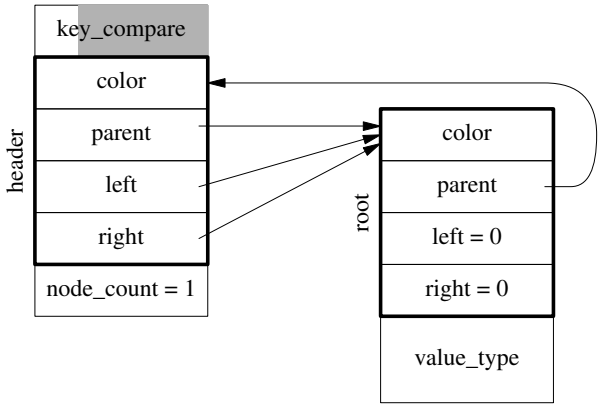
```
这里省略了一些默认模板参数，如 key_compare 和 allocator。
template<typename Key, typename Value> // key_compare and allocator
class rb_tree
{
 public:
  typedef std::less<Key> key_compare;
  typedef rb_tree_iterator<Value> iterator;
 protected:

  struct rb_tree_impl // : public node_allocator
  {
    key_compare       key_compare_;
    rb_tree_node_base header_;
    size_t            node_count_;
  };
  rb_tree_impl impl_;
};

template<typename Key, typename T> // key_compare and allocator
class map
{
 public:
  typedef std::pair<const Key, T> value_type;
 private:
  typedef rb_tree<Key, value_type> rep_type;
  rep_type tree_;
};
```

见下图。这是一颗空树，其中阴影部分是 padding bytes，因为 key_compare 通常是 empty class。（allocator 在哪里?）

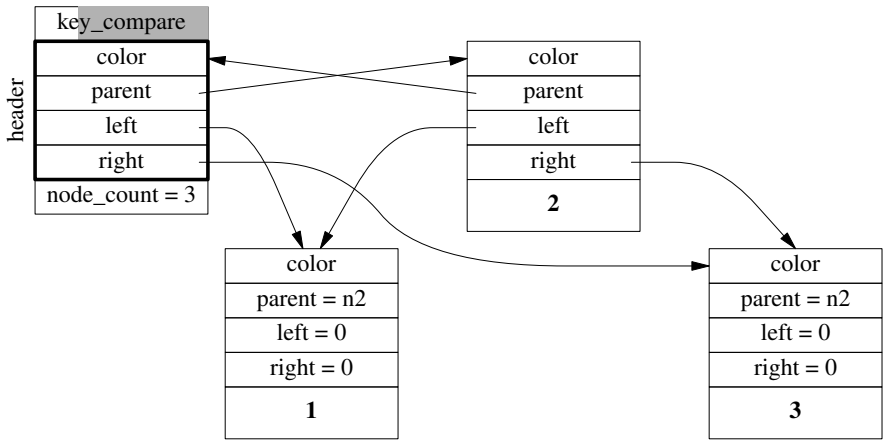


rb_tree 中的 header 不是 rb_tree_node 类型，而是 rb_tree_node_base。因此 rb_tree 的 size 是 6 * sizeof(void*)，与模板类型参数无关。在 32-bit 上是 24 字节，在 64-bit 上是 48 字节，很容易用代码验证这一点。另外容易验证 std::set 和 std::map 的 sizeof() 是一样的。

注意 rb_tree 中的 header 不是 root 节点，其 left 和 right 成员也不是指向左右子节点，而是指向最左边节点（left_most）和最右边节点（right_most），后面将会介绍原因，是为了满足时间复杂度。header.parent 指向 root 节点，root.parent 指向 header，header 固定是红色，root 固定是黑色。在插入一个节点后，数据结构如下图。



继续插入两个节点，假设分别位于 root 的左右两侧，那么得到的数据结构如下图所示（parent 指针没有全画出来，因为其指向很明显）。注意 header.left 指向最左侧节点，header.right 指向最右侧节点。



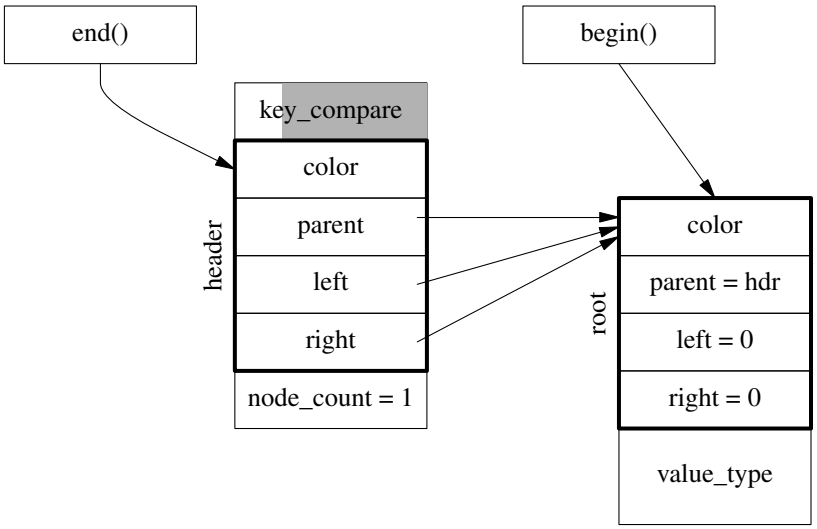
2.1.3 迭代器

rb_tree 的 iterator 的数据结构很简单，只包含一个 rb_tree_node_base 指针，但是其 ++/-- 操作却不见得简单（具体实现函数不在头文件中，而在 libstdc++ 库文件中）。

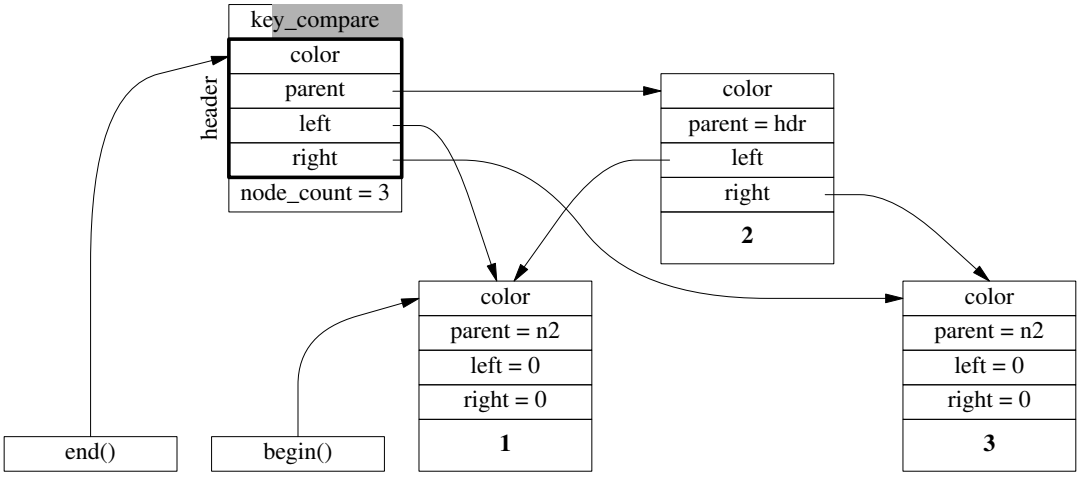
```
// defined in library, not in header
rb_tree_node_base* rb_tree_increment(rb_tree_node_base* node);
// others: decrement, reblance, etc.

template<typename Value>
struct rb_tree_node : public rb_tree_node_base
{
  Value value_field_;
};

template<typename Value>
struct rb_tree_iterator
{
  Value& operator*() const
  {
    return static_cast<rb_tree_node<Value>*>(node_)->value_field_;
  }

  rb_tree_iterator& operator++()
  {
    node_ = rb_tree_increment(node_);
    return *this;
  }

  rb_tree_node_base* node_;
};
```

end() 始终指向 header 节点，begin() 指向第一个节点（如果有的话）。因此对于空树，begin() 和 end() 都指向 header 节点（图略）。

对于 1 个元素的树，迭代器的指向如下。

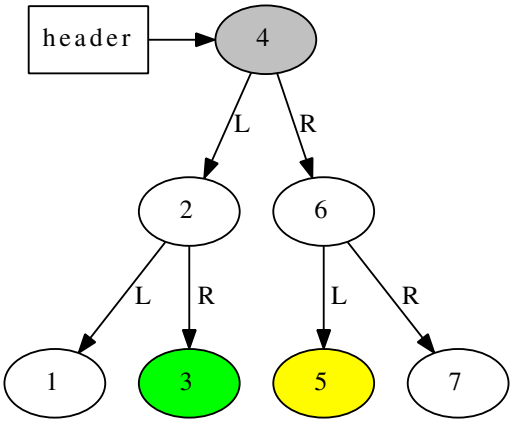


对于前面 3 个元素的树，迭代器的指向如下。

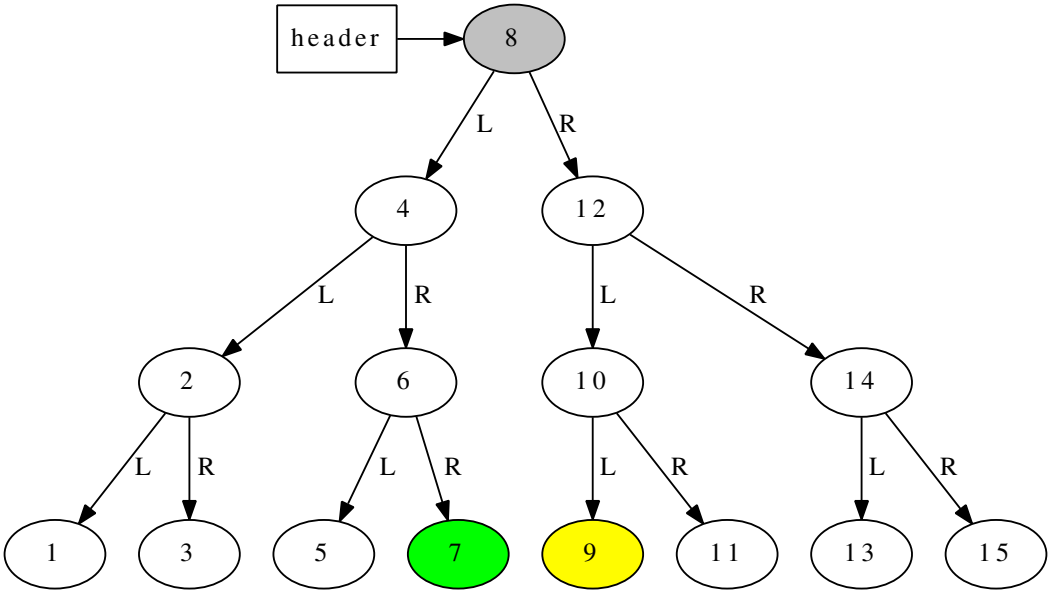


思考，对 std::set<int>::end() 做 dereference 会得到什么？（按标准，这属于 undefined behaviour，不过但试无妨。）

rb_tree 的 iterator 的递增递减操作并不简单。考虑下面这棵树，假设迭代器 iter 指向绿色节点 3，那么 ++iter 之后它应该指向灰色节点 4，再 ++iter 之后，它应该指向黄色节点 5，这两步递增都各走过了 2 个指针。



对于一颗更大的树（下图），假设迭代器 iter 指向绿色节点 7，那么 ++iter 之后它应该指向灰色节点 8，再 ++iter 之后，它应该指向黄色节点 9，这两步递增都各走过了 3 个指针。



由此可见，rb_tree 的迭代器的每次递增或递减不能保证是常数时间，最坏情况下可能是对数时间（即与树的深度成正比）。那么用 begin()/end() 迭代遍历一棵树还是不是 $O(N)$？换言之，迭代器的递增或递减是否是分摊后的（amortized）常数时间？

另外注意到，当 iter 指向最右边节点的时候（7 或 15），++iter 应该指向 header 节点，即 end()，这一步是 $O(\log N)$。同理，对 end() 做 --，复杂度也是 $O(\log N)$，后面会用到这一事实。

rb_tree 迭代器的递增递减操作的实现也不是那么一目了然。要想从头到尾遍历一颗二叉树（前序、中序、后序），教材上给出的办法是用递归（或者用 stack 模拟递归，性质一样），比如：（https://gist.github.com/4574621#file-tree-traversal-py）

```
def printTree(node):
    if node:
        printTree(node.left)
        print node.data
        printTree(node.right)
```

如果考虑通用性，可以把函数作为参数进去，然后通过回调的方式来访问每个节点，代码如下。Java XML 的 SAX 解析方式也是这样。

```
def visit(node, func):
    if node:
        printTree(node.left)
        func(node.data)
        printTree(node.right)
```

要想使用更方便，在调用方用 for 循环就能从头到尾遍历 tree，那似乎就不那么容易了。在 Python 这种支持 coroutine 的语言中，可以用 yield 关键字配合递归来实现，代码如下，与前面的实现没有多大区别。

```
在 Python 3.3 中还可以用 yield from, 这里用 Python 2.x 的写法。
def travel(root):
    if root.left:
        for x in travel(root.left):
            yield x
    yield root.data
    if root.right:
        for y in travel(root.right):
            yield y
```

调用方：

```
    for y in travel(root):
        print y
```

但是在 C++ 中，要想做到最后这种 StAX 的遍历方式，那么迭代器的实现就麻烦多了，见《STL 源码剖析》第 5.2.4 节的详细分析。这也是 node 中 parent 指针存在的原因，因为递增操作时，如果当前节点没有右子节点，就需要先回到父节点，再接着找。

2.1.4 空间复杂度

每个 rb_tree_node 直接包含了 value_type，其大小是 4 * sizeof(void*) + sizeof(value_type)。在实际分配内存的时候还要 round up 到 allocator/malloc 的对齐字节数，通常 32-bit 是 8 字节，64-bit 是 16 字节。因此 set<int> 每个节点是 24 字节或 48 字节，100 万个元素的 set<int> 在 x86-64 上将占用 48M 内存。说明用 set<int> 来排序整数是很不明智的行为，无论从时间上还是空间上。

考虑 malloc 对齐的影响，set<int64_t> 和 set<int32_t> 占用相同的空间，set<int> 和 map<int, int> 占用相同的空间，无论 32-bit 还是 64-bit 均如此。

2.2 几个为什么

对于 rb_tree 的数据结构，我们有几个问题：

1. 为什么 rb_tree 没有包含 allocator 成员？
2. 为什么 iterator 可以 pass-by-value？
3. 为什么 header 要有 left 成员，并指向 left most 节点？
4. 为什么 header 要有 right 成员，并指向 right most 节点？
5. 为什么 header 要有 color 成员，并且固定是红色？
6. 为什么要分为 rb_tree_node 和 rb_tree_node_base 两层结构，引入 node 基类的目的是什么？
7. 为什么 iterator 的递增递减是分摊（amortized）常数时间？
8. 为什么 muduo 网络库的 Poller 要用 std::map<int, Channel*> 来管理文件描述符？

我认为，在面试的时候能把上面前 7 个问题答得头头是道，足以说明对 STL 的 map/set 掌握得很好。下面一一给出我的解答。

2.2.1 为什么 rb_tree 没有包含 allocator 成员？

因为默认的 allocator 是 empty class（没有数据成员，也没有虚表指针 vptr），为了节约 rb_tree 对象的大小，STL 在实现中用了 empty base class optimization。具体做法是 std::map 以 rb_tree 为成员，rb_tree 以 rb_tree_impl 为成员，而 rb_tree_impl 继承自 allocator，这样如果 allocator 是 empty class，那么 rb_tree_impl 的大小就跟没有基类时一样。其他 STL 容器也使用了相同的优化措施，因此 std::vector

对象是 3 个 words，std::list 对象是 2 个 words。boost 的 compressed_pair 也使用了相同的优化。

我认为，对于默认的 key_compare，应该也可以实施同样的优化，这样每个 rb_tree 只需要 5 个 words，而不是 6 个。

2.2.2 为什么 iterator 可以 pass-by-value?

读过《Effective C++》的想必记得其中有个条款是 Prefer pass-by-reference-to-const to pass-by-value，即对象尽量以 const reference 方式传参。这个条款同时指出，对于内置类型、STL 迭代器和 STL 仿函数，pass-by-value 也是可以的，一般没有性能损失。

在 x86-64 上，对于 rb_tree iterator 这种只有一个 pointer member 且没有自定义 copy-ctor 的 class，pass-by-value 是可以通过寄存器进行的[8]，就像传递普通 int 和指针那样。因此实际上可能比 pass-by-const-reference 略快，因为 callee 减少了一次 dereference。

同样的道理，muduo 中的 Date class 和 Timestamp class 也是明确设计来 pass-by-value 的，它们都只有一个 int/long 成员，按值拷贝不比 pass reference 慢。如果不分青红皂白一律对 object 使用 pass-by-const-reference，固然算不上什么错误，不免稍微显得知其然不知其所以然罢了。

2.2.3 为什么 header 要有 left 成员，并指向 left most 节点?

原因很简单，让 begin() 函数是 $O(1)$。假如 header 中只有 parent 没有 left，begin() 将会是 $O(\log N)$，因为要从 root 出发，走 $O(\log N)$ 步，才能到达 left most。现在 begin() 只需要用 header.left 构造 iterator 并返回即可。

2.2.4 为什么 header 要有 right 成员，并指向 right most 节点?

这个问题不是那么显而易见。end() 是 $O(1)$，因为直接用 header 的地址构造 iterator 即可，不必使用 right most 节点。在 bits/stl_tree.h 源码中有这么一段注释：

```
// Red-black tree class, designed for use in implementing STL
// associative containers (set, multiset, map, and multimap). The
// insertion and deletion algorithms are based on those in Cormen,
// Leiserson, and Rivest, Introduction to Algorithms (MIT Press,
// 1990), except that
```

[8] 例如函数的头 4 个参数（by-value 返回对象算一个参数）。

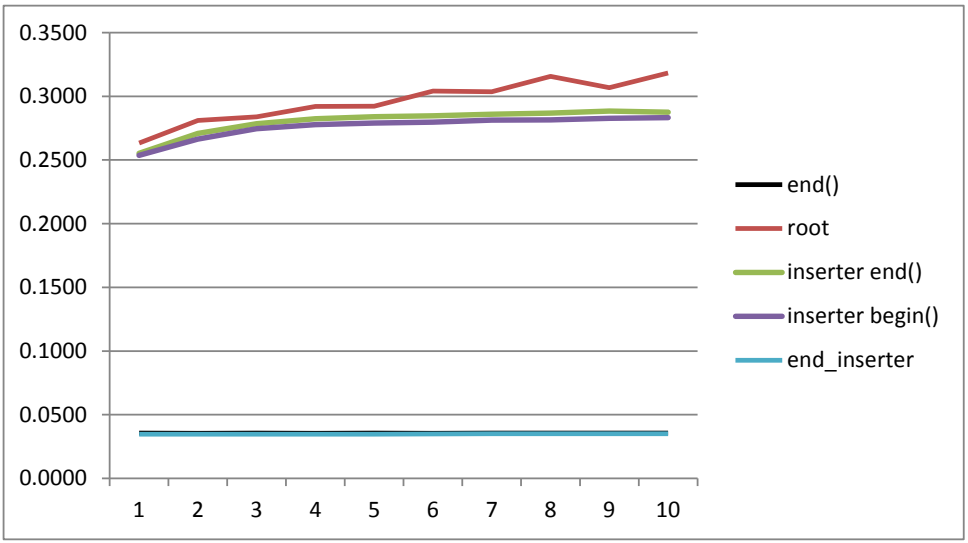
```
//
// (1) the header cell is maintained with links not only to the root
// but also to the leftmost node of the tree, to enable constant
// time begin(), and to the rightmost node of the tree, to enable
// linear time performance when used with the generic set algorithms
// (set_union, etc.)
//
// (2) when a node being deleted has two children its successor node
// is relinked into its place, rather than copied, so that the only
// iterators invalidated are those referring to the deleted node.
```

这句话的意思是说，如果按大小顺序插入元素，那么时间复杂度将会是 $O(N)$，而非 $O(N \log N)$。即下面这段代码的运行时间与 N 成正比：

```
// 在 end() 按大小顺序插入元素
std::set<int> s;
const int N = 1000 * 1000
for (int i = 0; i < N; ++i)
    s.insert(s.end(), i);
```

在 rb_tree 的实现中，insert(value) 一个元素通常的复杂度是 $O(\log N)$。不过，insert(hint, value) 除了可以直接传 value_type，还可以再传一个 iterator 作为 hint，如果实际的插入点恰好位于 hint 左右，那么分摊后的复杂度是 $O(1)$。对于这里的情况，既然每次都在 end() 插入，而且插入的元素又都比 *(end()-1) 大，那么 insert() 是 $O(1)$。在具体实现中，如果 hint 等于 end()，而且 value 比 right most 元素大，那么直接在 right most 的右子节点插入新元素即可。这里 header.right 的意义在于让我们能在常数时间取到 right most 节点的元素，从而保证 insert() 的复杂度（而不需要从 root 出发走 $O(\log N)$ 步到达 right most）。具体的运行时间测试代码 [9]，测试结果如下，纵坐标是每个元素的耗时（微秒），其中最上面的红线是普通 insert(value)，下面的蓝线和黑线是 insert(end(), value)，确实可以大致看出 $O(\log N)$ 和 $O(1)$ 关系。具体的证明见《算法导论（第 2 版）》第 17 章中的思考题 17-4。

[9] https://gist.github.com/4574621#file-tree-bench-cc



但是，根据测试结果，前面引用的那段注释其实是错的，std::inserter() 与 std::set_union() 配合并不能实现 $O(N)$ 复杂度。原因是 std::inserter_iterator 会在每次插入之后做一次 ++iter，而这时 iter 正好指向 right most 节点，其 ++ 操作是 $O(\log N)$ 复杂度（前面提到过 end() 的递减是 $O(\log N)$，这里反过来也是一样）。于是把整个算法拖慢到了 $O(N \log N)$。要想 set_union() 是线性复杂度，我们需要自己写 inserter，见上面代码中的 end_inserter 和 at_inserter，此处不再赘言。

2.2.5 为什么 header 要有 color 成员，并且固定是红色？

这是一个实现技巧，对 iterator 做递减时，如果此刻 iterator 指向 end()，那么应该走到 right most 节点。既然 iterator 只有一个数据成员，要想判断当前是否指向 end()，就只好判断 (node_->color_ == kRed && node_->parent_->parent_ == node_) 了。

2.2.6 为什么要分为 rb_tree_node 和 rb_tree_node_base 两层结构，引入 node 基类的目的是什么？

这是为了把迭代器的递增递减、树的重新平衡等复杂函数从头文件移到库文件中，减少模板引起的代码膨胀（set<int> 和 set<string> 可以共享这些的 rb_tree 基础函数），也稍微加快编译速度。引入 rb_tree_node_base 基类之后，这些操作就可以以基类指针（与模板参数类型无关）为参数，因此函数定义不必放在在头文件

中。这也是我们在头文件里看不到 iterator 的 ++/-- 的具体实现的原因，它们位于 libstdc++ 的库文件源码中。注意这里的基类和继承不是为了 OOP，而纯粹是一种实现技巧。

2.2.7 为什么 iterator 的递增递减是分摊（amortized）常数时间？

严格的证明需要用到分摊分析（amortized analysis），一来我不会，二来写出来也没多少人看，这里我用一点归纳的办法来说明这一点。考虑一种特殊情况，对前面图中的满二叉树（perfect binary tree）从头到尾遍历，计算迭代器一共走过多少步（即 follow 多少次指针），然后除以节点数 N，就能得到平均每次递增需要走多少步。既然红黑树是平衡的，那么这个数字跟实际的步数也相差不远。

- 对于深度为 1 的满二叉树，有 1 个元素，从 begin() 到 end() 需要走 1 步，即从 root 到 header。
- 对于深度为 2 的满二叉树，有 3 个元素，从 begin() 到 end() 需要走 4 步，即 $1 \to 2 \to 3 \to$ header，其中从 3 到 header 是两步。
- 对于深度为 3 的满二叉树，有 7 个元素，从 begin() 到 end() 需要走 11 步，即先遍历左子树（4 步）、走 2 步到达右子树的最左节点，遍历右子树（4 步），最后走 1 步到达 end()，4 + 2 + 4 + 1 = 11。
- 对于深度为 4 的满二叉树，有 15 个元素，从 begin() 到 end() 需要走 26 步。即先遍历左子树（11 步）、走 3 步到达右子树的最左节点，遍历右子树（11 步），最后走 1 步到达 end()，11 + 3 + 11 + 1 = 26。
- 后面的几个数字依次是 57、120、247

对于深度为 m 的满二叉树，有 $2^m - 1$ 个元素，从 begin() 到 end() 需要走 $f(m)$ 步。那么 $f(m) = 2f(m-1) + m$。

然后，用递推关系求出（推导过程留作练习）：

$$f(m) = \sum_{i=1}^{m} (i \times 2^{m-i}) = 2^{n+1} - n - 2$$

即对于深度为 m 的满二叉树，从头到尾遍历的步数小于 $2^{m+1} - 2$，而元素个数是 $2^m - 1$，二者一除，得到平均每个元素需要 2 步。因此可以说 rb_tree 的迭代器的递增递减是分摊后的常数时间。

似乎还有更简单的证明方法，在从头到尾遍历的过程中，每条边（edge）最多来回各走一遍，一棵树有 N 个节点，那么有 $N - 1$ 条边，最多走 $2(N-1) + 1$ 步，也就是说平均每个节点需要 2 步，跟上面的结果相同。

最后说一点题外话。

2.2.8 为什么 muduo 网络库的 Poller 要用 std::map<int, Channel*> 来管理文件描述符?

muduo 在正常使用的时候会用 EPollPoller，是对 epoll(4) 的简单封装，其中用 std::map<int, Channel*> channels_ 来保存 fd 到 Channel 对象的映射。我这里没有用数组，而是用 std::map，原因有几点：

- epoll_ctl() 是 $O(\log N)$，因为内核中用红黑树来管理 fd。因此用户态用数组管理 fd 并不能降低时间复杂度，反而有可能增加内存使用（用 hash table 倒是不错）。不过考虑系统调用开销，map vs. vector 的实际速度差别不明显。[10]
- channels_ 只在 Channel 创建和销毁的时候才会被访问，其他时候（修改关注的读写事件）都是位于 assert() 中，用于 debug 模式断言。而 Channel 的创建和销毁本身就伴随着 socket 的创建和销毁，涉及系统调用，channels_ 操作所占的比重极小。因此优化 channels_ 属于优化 nop，是无意义的。
- 既然 channels_ 只在 Channel 创建和销毁的时候才会被访问，那为什么不干脆去掉它？原因是为了 gdb 调试（包含检查 core dump）时能获知 epoll 的 watch list，因为 Linux 没有系统调用能查询这一信息（再说 core dump 里也没有内核数据）。

[10] 题外话：总是遇到有人说 epoll(4) 是 $O(1)$ 云云，其实 epoll_wait() 是 $O(N)$，N 是活动 fd 的数目。poll(2) 和 select(2) 也都是 $O(N)$，不过 N 的意义不同。仔细算下来，恐怕只有 epoll_create() 是 $O(1)$ 的。也有人想把 epoll 改为数组，但是被否决了，因为这是开历史倒车（https://lkml.org/lkml/2008/1/8/205）。

3 C++ 面试中 String class 的一种正确写法

先说说程序员（应届生）面试的一般过程，一轮面试（面对一到两个面试官）一般是四、五十分钟，面试官会问两三个编程问题（通常是两大一小），因此留给每个编程题的时间只有 20 分钟。这 20 分钟不光是写代码，还要跟面试官讨论你的答案并解答提问，比如面试官拿过你的答案纸，问某一行代码如果修改会有什么后果。因此真正留给在纸上或白板上写代码的时间也就 10 分钟上下，这是本文的适用场景。本文的配套代码位于 https://github.com/chenshuo/recipes/blob/master/string/StringTrivial.h。

C++ 的一个常见面试题是让你实现一个 String 类，限于时间，不可能要求具备 std::string 的功能，但至少要求能正确管理资源。具体来说：

- 能像 int 类型那样定义变量，并且支持赋值、复制。
- 能用作函数的参数类型及返回类型。
- 能用作标准库容器的元素类型，即 vector/list/deque 的 value_type。（用作 map 或 unordered_map 的 key_type 是更进一步的要求，本文从略）。

换言之，你写的 String class 能让以下测试程序编译运行通过，并且没有内存方面的错误。

测试

```
void foo(String x)
{ }

void bar(const String& x)
{ }

String baz()
{
  String ret("world");
  return ret;
}

int main()
{
  String s0;
  String s1("hello");
  String s2 = "hello";
  String s3(s0);
  String s4 = s1;
  s3 = s1;
  s3 = s3;
  s1 = "aewsome";
```

```
    foo(s1);
    foo("temporary");
    bar(s1);
    bar("temporary");
    String s5 = baz();
    s5 = baz();

    std::vector<String> svec;
    svec.push_back(s0);
    svec.push_back(s1);
    svec.push_back("good job");
  }
```

测试

本文给出我认为适合面试的答案：一个能用 10 分钟时间在纸上写出来且不会有错的 String class，强调正确性及易实现（白板上写也不会错），不强调效率与功能完备。某种意义上可以说是以时间（运行快慢）换空间（代码简洁）。

首先选择数据成员，最简单的 String 只有一个 char* 成员变量。好处是容易实现，坏处是某些操作的复杂度较高（例如 size() 会是线性时间），而且不能处理包含 '\0' 的二进制数据。为了面试时写代码不出错，本文设计的 String 只有一个 char* data_ 成员。而且规定 invariant 如下：一个有效的 String 对象的 data_ 保证不为 NULL，data_ 以 '\0' 结尾，以方便配合 C 语言的 strlen()/strcpy() 等函数。

其次决定支持哪些操作，构造、析构、拷贝构造、赋值这几样是肯定要有的（以前合称 big three，现在叫 copy control）。如果钻得深一点，C++11 的移动构造和移动赋值也可以有，见后文。为了突出重点，本文就不考虑 operator[] 之类的重载了。

这样 C++98/03 版的代码基本上就定型了：

面试手写版 String 实现

```
1  #include <utility>    // for std::swap
2  #include <string.h>
3
4  class String
5  {
6   public:
7    String()
8      : data_(new char[1])
9    {
10     *data_ = '\0';
11   }
12
13   String(const char* str)
14     : data_(new char[strlen(str) + 1])
15   {
16     strcpy(data_, str);
17   }
```

```
18
19    String(const String& rhs)
20      : data_(new char[rhs.size() + 1])
21    {
22      strcpy(data_, rhs.c_str());
23    }
24
25    ~String()
26    {
27      delete[] data_;
28    }
29
30    // In C++11, this is unifying assignment operator
31    String& operator=(String rhs) // yes, pass-by-value
32    {
33      swap(rhs);
34      return *this;
35    }
36
37    // Accessors
38
39    size_t size() const
40    {
41      return strlen(data_);
42    }
43
44    const char* c_str() const
45    {
46      return data_;
47    }
48
49    void swap(String& rhs)
50    {
51      std::swap(data_, rhs.data_);
52    }
53
54   private:
55    char* data_;
56  };
```

面试手写版 String 实现

注意代码的几个要点：

1. 只在构造函数里调用 `new char[]`，只在析构函数里调用 `delete[]`。
2. 赋值操作符采用了 copy-and-swap 这种现代写法。[11]
3. 每个函数都只有一两行代码，没有条件判断。
4. 析构函数不必检查 `data_` 是否为 `NULL`。

[11] http://en.wikibooks.org/wiki/More_C++_Idioms/Copy-and-swap

5. 构造函数 `String(const char* str)` 没有检查 `str` 的合法性，这是一个永无止境的争论话题。这里在初始化列表里就用到了 `str`，因此在函数体内用 `assert()` 是无意义的，google-glog 的 `CHECK_NOTNULL` 宏是个可行的办法。[12]
6. 第 2 行 `#include` 的头文件是 `string.h` 而非 `cstring`，这是我的个人习惯，理由见：http://blog.csdn.net/solstice/article/details/9923615。

这恐怕是最简洁的 `String` 实现了。

3.1 如果遇到知识老旧的面试官

如果面试官的知识比较陈旧，认为 copy-ctor 一定要以 const reference 为参数，不能像第 31 行那样 pass-by-value，那么在笔试（而非面试）时，copy-ctor 可以用下面这种传统的写法。

```
String& operator=(const String& rhs)
{
  String tmp(rhs);
  swap(tmp);
  return *this;
}
```

C++11 中更应该使用前面出现的 pass-by-value 做法，见下文。

其实 2005 年出版的《Effective C++ 第 3 版》和 2004 年出版的《C++ Coding Standards》上都介绍了 copy-and-swap 配合 pass-by-value 的做法，这两本书的中文版也早就出版了。

3.2 支持 C++11 的移动语意

在 C++11 中，这个 `String` 还可以改进一些。首先是 copy-ctor 不必重复 `String(const char* str)` 构造函数的代码，而是转发给后者。其次，实现 move-ctor 在配合前面已经实现的 unifying assignment operator 就能获得移动语意。最后，将析构函数声明为 noexcept，这样才算把活做全。

C++11

```
// Implement copy-ctor with delegating constructor in C++11
String(const String& rhs)
  : String(rhs.data_)
{
}
```

[12] http://google-glog.googlecode.com/svn/trunk/doc/glog.html#check

```
// move-ctor
String(String&& rhs) noexcept
  : data_(rhs.data_)
{
  rhs.data_ = nullptr;
}

~String() noexcept
{
  delete[] data_;
}
```

C++11

练习 1：增加 operator==、operator<、operator[] 等操作符重载。

练习 2：实现一个带 int size_; 成员的版本，以空间换时间。

练习 3：受益于右值引用及移动语意，在 C++11 中对 String 实施直接插入排序的性能比 C++98/03 要高，试编程验证之。（g++ 的标准库也用到了此技术。）