Boost.Signals用法详解

当你面对需要用多段代码来处理一个事件的情况时，典型的解决方案有：用函数指针进行回调，或者直接对产生事件的子系统与处理事件的子系统之间的依赖性进行编码。这种设计常常会导致循环的依赖性。通过使用 Boost.Signals, 你将获得灵活性和解耦。要开始使用这个库，首先要包含头文件 "boost/signals.hpp".

[2] Boost.Signals 库和 Boost.Regex 库是本书所讨论的库中仅有的需要编译和链接才能使用的库。编译的过程很简单，在线文档中已有详尽的描述，这里我不再复述。

以下例子示范了 signals 和插槽(slots)的基本特性，包括如何连接它们以及如何产生一个 signal. 注意，插槽指的是由你提供的一个兼容于 signal 的函数签名的函数或函数对象。在以下代码中，我们既创建了一个普通函数，my\_first\_slot, 也创建了一个函数对象，my\_second\_slot; 它们两个都将连接到我们创建的一个 signal 上。

#include <iostream>  
#include "boost/signals.hpp"  
  
void my\_first\_slot() {  
std::cout << "void my\_first\_slot()\n";  
}  
  
class my\_second\_slot {  
public:  
void operator()() const {  
std::cout <<  
"void my\_second\_slot::operator()() const\n";  
}  
};  
  
int main() {  
boost::signal<void ()> sig;  
  
sig.connect(&my\_first\_slot);  
sig.connect(my\_second\_slot());  
  
std::cout << "Emitting a signal...\n";  
sig();  
}

我们首先声明一个 signal, 它所需的插槽为返回 void 且不带参数。然后，我们把两个兼容的插槽类型连接到该 signal. 对于第一个插槽，我们用普通函数 my\_first\_slot 的地址调用 connect。对于另一个插槽，我们缺省构造一个函数对象 my\_second\_slot 的实例并把它传给 connect。这些连接意味着当我们产生一个 signal (通过调用 sig)时，这两个插槽将被立即调用。

sig();

运行这个程序，输出信息如下：

Emitting a signal...  
void my\_first\_slot()  
void my\_second\_slot::operator()() const

但是，后两行的顺序不一定是这样的，因为属于同一个组的插槽会以不确定的顺序执行。没有办法确定哪一个插槽会先被调用。如果插槽的调用顺序事关紧要，你就必须把它们放入不同的组。

插槽分组

有时候，某些插槽需要在其它插槽之前调用，例如某些插槽会产生一些副作用而别的插槽需要依赖于这些副作用。分组就是支持这种需求的方法。signal 有一个模板参数，名为 Group，其缺省值为 int. Groups 缺省以 std::less<Group> 为排序标准，对于 int 就是 operator< 。换句话说，属于 group 0 的插槽会在 group 1 的插槽之前调用，等等。但是请注意，同一个组中的插槽的调用顺序是不确定的。要严格控制所有插槽的调用顺序，唯一的办法就是把每个插槽都安排到各自的组中。

把插槽指定到一个组的方法是，传递一个 Group 给 signal::connect. 一个已连接插槽不能改变其所属的组；要改变一个插槽所属的组，必须先断开它的连接，然后重新把它连接到 signal 上并同时指定新组。

作为例子，我们考虑两个插槽，它们带一个类型为 int& 的参数；第一个插槽将参数加倍，第二个插槽则把当前值加3。我们要求正确的语义是，先把该值加倍，然后再加3。如果不指定顺序，我们就不能确保按该语义执行。以下方法只能在某些系统的某些时候正确执行(可能是周一或周三而且月圆的时候)。

#include <iostream>  
#include "boost/signals.hpp"  
  
class double\_slot {  
public:  
void operator()(int& i) const {  
i\*=2;  
}  
};  
  
class plus\_slot {  
public:  
void operator()(int& i) const {  
i+=3;  
}  
};  
  
int main() {  
boost::signal<void (int&)> sig;  
sig.connect(double\_slot());  
sig.connect(plus\_slot());  
  
int result=12;  
sig(result);  
std::cout << "The result is: " << result << '\n';  
}

运行这段程序，可能产生以下输出：

The result is: 30

或者产生以下输出：

The result is: 27

不使用分组的方法就无法保证正确的行为。我们需要确保 double\_slot 总是在 plus\_slot 之前被调用。这就要求我们要指定 double\_slot 属于一个顺序在 plus\_slot 所属的组之前的组，即：

sig.connect(0,double\_slot());  
sig.connect(1,plus\_slot());

这样可以确保得到我们想要的(即 27)。再次提醒，对于同一个组中的插槽，它们被调用的顺序是不确定的。只要你需要插槽以特定的顺序来执行，就必须确保它们使用不同的组。

Groups 的类型是类 signal 的一个模板参数，所以它可以使用别的类型，如 std::string 。

#include <iostream>  
#include <string>  
#include "boost/signals.hpp"  
  
class some\_slot {  
std::string s\_;  
public:  
some\_slot(const std::string& s) : s\_(s) {}  
void operator()() const {  
std::cout << s\_ << '\n';  
}  
};  
  
int main() {  
boost::signal<void (),  
boost::last\_value<void>,std::string> sig;  
  
some\_slot s1("I must be called first, you see!");  
some\_slot s2("I don't care when you call me, not at all. \  
It'll be after those belonging to groups, anyway.");  
some\_slot s3("I'd like to be called second, please.");  
  
sig.connect(s2);  
sig.connect("Last group",s3);  
sig.connect("First group",s1);  
  
sig();  
}

首先我们定义一个插槽类型，它在执行时输出一个 std::string 到 std::cout 。然后，我们声明 signal. 因为 Groups 参数是在 Combiner 类型之后的，所以我们必须同时指定 Combiner (我们只是按缺省值来声明)。我们把 Groups 类型设为std::string 。

boost::signal<void (),boost::last\_value<void>,std::string> sig;

对于剩下的模板参数，我们接受缺省值就可以了。在连接到插槽 s1, s2, 和 s3 时，所创建的组是以字母顺序排序的(因为这是 std::less<std::string> 的行为)，因此 "First group" 先于 "Last group". 注意，由于字符串常量可以隐式转换为 std::string, 所以我们可以把它们直接传递给 signal 的 connect 函数。运行该程序可以告诉我们正确的结果。

I must be called first, you see!  
I'd like to be called second, please.  
I don't care when you call me, not at all.  
It'll be after those belonging to groups, anyway.

我们也可以在声明 signal 类型时选择别的排序方法，例如 std::greater.

boost::signal<void (),boost::last\_value<void>,  
std::string,std::greater<std::string> > sig;

如果我们把它用于前面的例子，输出将变为：

I'd like to be called second, please.  
I must be called first, you see!  
I don't care when you call me, not at all.  
It'll be after those belonging to groups, anyway.

当然，在这个例子中，std::greater 产生的顺序导致了错误的输出，但这是另一回事。分组非常有用，绝对必要，但是给组赋以正确的值并不总是那么简单的事，因为被连接的插槽并不需要在代码的同一个地方执行。弄清楚某个插槽所应该使用什么组号可能是个问题。有时，这个问题可以用规定来解决，即在代码中增加注释，确保每个人都能看到这些注释，但是这也只能在代码中不是很多地方要进行组号的赋值以及程序员不偷懒时有用。换句话说，这种方法也不一定管用。所以，你需要一个集中的产生组号的地方，它可以依据某个给定的值为每个插槽产生唯一的组号，或者如果相关的插槽相互了解，那么也可以由插槽提供它们自己的组号。

现在你已经知道如何解决按顺序调用插槽的问题了，让我们来看看如何让你的 signals 使用不同的签名。你常常需要传递额外的信息给你系统中的重要事件。

带参数的 Signals

通常会有一些额外的数据要传递给 signal. 例如，想象一个温度保护器，它报告温度的急剧变化。仅仅知道保护器发现了问题是不够的；插槽可能需要知道当前的温度。虽然保护器(一个 signal)和插槽都可以从一个公用的传感器去获取温度值，但是最简单的方式还是让保护器在调用插槽时把当前温度传递给插槽。还有一个例子，想象有多个插槽连接到多个 signal 上：插槽很可能需要知道是哪一个 signal 调用了它。有很多用例都需要从 signal 传递一些信息给插槽。插槽接受的参数是 signal 声明中的一部分。signal 类模板的第一个参数就是调用 signal 的函数签名，而且这个签名也用于 signal 调用那些被连接的插槽。如果我们想这个参数可以修改，我们就要确保它是通过非const 引用或指针来进行传递的，否则我们就可以通过值或 const 引用来传递它。注意，这个原始参数除了是可修改或不可修改这么明显的差异之外，对于 signal本身以及插槽可以接受的参数类型还有一些隐喻，如果 signal 接受一个传值或传 const 引用的参数，那么所有可以隐式转换为该参数类型的类型都可以用于产生一个 signal. 对于插槽也一样，如果插槽是通过传值或传 const 引用来接受参数的话，这就意味着允许从 signal 的真正参数类型隐式转换到这个类型。我们后面将讨论如果在处理信号时正确地传递参数，届时我们将看到更多关于这一点的详细讨论。

想象一个自动停车场监视器，一旦有车进入或离开停车场，监视器将收到一个通知。它需要知道一些关于这辆车的唯一信息，例如车的登记号码，这样它才可以跟踪每辆车的进入和离开。这个监视器有一个它自己的 signal，能够在有人试图进行欺骗时触发警报。这样就需要一些警卫监听这个 signal, 我们用一个名为 security\_guard 来对它们进行建模。最后，我们再增加一个 gate 类，它包含一个 signal 用于在一辆车进入或离开停车场时产生。( parking\_lot\_guard 显然需要知道这一点)。我们先来看看这个 parking\_lot\_guard 的声明。

class parking\_lot\_guard {  
typedef  
boost::signal<void (const std::string&)> alarm\_type;  
typedef alarm\_type::slot\_type slot\_type;  
  
boost::shared\_ptr<alarm\_type> alarm\_;  
  
typedef std::vector<std::string> cars;  
typedef cars::iterator iterator;  
  
boost::shared\_ptr<cars> cars\_;  
public:  
  
parking\_lot\_guard();  
boost::signals::connection  
connect\_to\_alarm(const slot\_type& a);  
void operator()(bool is\_entering,const std::string& car\_id);  
  
private:  
void enter(const std::string& car\_id);  
void leave(const std::string& car\_id);  
};

这里有三个特别重要的地方要认真看一下；第一个是警报，即一个返回 void 且接受一个 std::string (它用于标识一辆车)的 boost::signal 。这个 signal 的声明值得再好好看一次。

boost::signal<void (const std::string&)>

它就象是一个函数的声明，只是没有了函数名。如果有怀疑，请记住除此以外没有别的东西了！你可以从外部使用成员函数connect\_to\_alarm 连接这个 signal 。(我们将看到在实现这个类时，如何以及为何我们要发出警报)。下一个要留意的地方是，这个警报以及容纳车辆标识的容器(一个容纳 std::strings 的 std::vector )两者均保存于 boost::shared\_ptr 中。这样做的原因是，尽管我们只是打算声明一个 parking\_lot\_guard 实例，但是也可能变成多份拷贝；因为这个监视器类稍后还会连接到其它的 signal 上，这样就会创建多份拷贝(Boost.Signals 会复制插槽，所以需要正确地管理生存期)；而我们希望所有的数据都可用，因此我们就要共享它。虽然我们可以避免拷贝，例如通过使用指针或者把插槽的行为外部化，但是这样做可以发现一些容易掉进去的陷阱。最后还要留意的是，我们声明了一个调用操作符，其原因是我们将要在 gate 类(待会定义)中把 parking\_lot\_guard 连接到一个 signal in the class .

现在让我们把注意力放到 security\_guard 类。

class security\_guard {  
std::string name\_;  
public:  
security\_guard (const char\* name);  
  
void do\_whatever\_it\_takes\_to\_stop\_that\_car() const;  
void nah\_dont\_bother() const;  
  
void operator()(const std::string& car\_id) const;  
};

security\_guards 并不需要做太多事情。这个类有一个调用操作符，用作来自于 parking\_lot\_guard 的警报的一个插槽，另外还有两个函数：一个用于停住引发警报的车辆，另一个不做任何事。下面带来我们的 gate 类，它用于在有车辆到达停车场以及车辆离开时进行检查。

class gate {  
typedef  
boost::signal<void (bool,const std::string&)> signal\_type;  
typedef signal\_type::slot\_type slot\_type;  
  
signal\_type enter\_or\_leave\_;  
public:  
boost::signals::connection  
connect\_to\_gate(const slot\_type& s);  
void enter(const std::string& car\_id);  
void leave(const std::string& car\_id);  
};

你将留意到，gate 类包含一个 signal，它在有车辆进入或离开停车场时被触发。有一个公用成员函数(connect\_to\_gate)用于连接这个 signal, 另两个成员函数(enter 和 leave)用于在车辆进入或离开时被调用。

现在是时候来实现它们了。让我们从 gate 类开始。

class gate {  
typedef  
boost::signal<void (bool,const std::string&)> signal\_type;  
typedef signal\_type::slot\_type slot\_type;  
  
signal\_type enter\_or\_leave\_;  
public:  
boost::signals::connection  
connect\_to\_gate(const slot\_type& s) {  
return enter\_or\_leave\_.connect(s);  
}  
  
void enter(const std::string& car\_id) {  
enter\_or\_leave\_(true,car\_id);  
}  
  
void leave(const std::string& car\_id) {  
enter\_or\_leave\_(false,car\_id);  
}  
};

这个实现很简单。多数工作都前转到其它对象。函数 connect\_to\_gate 简单地把调用转为对 signal enter\_or\_leave\_的 connect 的调用。函数 enter 产生 signal, 传入一个 true (代表有车辆进入)和车辆的标识。leave 完成同样的工作，但是传入的是 false, 代表有车辆离开。简单的类做简单的事。security\_guard 类也不太复杂。

class security\_guard {  
std::string name\_;  
public:  
security\_guard (const char\* name) : name\_(name) {}  
  
void do\_whatever\_it\_takes\_to\_stop\_that\_car() const {  
std::cout <<  
"Stop in the name of...eh..." << name\_ << '\n';  
}  
  
void nah\_dont\_bother() const {  
std::cout << name\_ <<  
" says: Man, that coffee tastes f i n e fine!\n";  
}  
  
void operator()(const std::string& car\_id) const {  
if (car\_id.size() && car\_id[0]=='N')  
do\_whatever\_it\_takes\_to\_stop\_that\_car();  
else  
nah\_dont\_bother();  
}  
};

security\_guards 知道它们自己的名字，并且可以决定在警报发出时是否要做些事情(如果 car\_id 以字母 N 打头，它们就会有所动作)。调用操作符就是被调用的插槽函数，security\_guard 对象是一个函数对象，并且符合 parking\_lot\_guard 的 alarm\_type 信号的要求。parking\_lot\_guard 稍微复杂一些，但也不是很复杂。

class parking\_lot\_guard {  
typedef  
boost::signal<void (const std::string&)> alarm\_type;  
typedef alarm\_type::slot\_type slot\_type;  
  
boost::shared\_ptr<alarm\_type> alarm\_;  
  
typedef std::vector<std::string> cars;  
typedef cars::iterator iterator;  
  
boost::shared\_ptr<cars> cars\_;  
public:  
  
parking\_lot\_guard()  
: alarm\_(new alarm\_type), cars\_(new cars) {}  
  
boost::signals::connection  
connect\_to\_alarm(const slot\_type& a) {  
return alarm\_->connect(a);  
}  
  
void operator()  
(bool is\_entering,const std::string& car\_id) {  
if (is\_entering)  
enter(car\_id);  
else  
leave(car\_id);  
}  
  
private:  
void enter(const std::string& car\_id) {  
std::cout <<  
"parking\_lot\_guard::enter(" << car\_id << ")\n";  
  
// 如果车辆已经在这，就触发警报  
if (std::binary\_search(cars\_->begin(),cars\_->end(),car\_id))  
(\*alarm\_)(car\_id);  
else // Insert the car\_id  
cars\_->insert(  
std::lower\_bound(  
cars\_->begin(),  
cars\_->end(),car\_id),car\_id);  
}  
  
void leave(const std::string& car\_id) {  
std::cout <<  
"parking\_lot\_guard::leave(" << car\_id << ")\n";  
  
// 如果是未登记的车辆，就触发警报  
std::pair<iterator,iterator> p=  
std::equal\_range(cars\_->begin(),cars\_->end(),car\_id);  
if (p.first==cars\_->end() || \*(p.first)!=car\_id)  
(\*alarm\_)(car\_id);  
else  
cars\_->erase(p.first);  
}  
};

就是这样了！(当然，我们还没有把插槽连接到 signal 上，还要做一些事情。但是这些类对于所要做的事情而言还是非常地简单的)。 为了让警报和车辆标识的 shared\_ptr 有正确的行为，我们实现了缺省构造函数，在其中适当地分配了 signal 和 vector 。隐式创建的复制构造函数、析构函数以及赋值操作符都可以正确工作(这要归功于智能指针)。函数 connect\_to\_alarm 把调用转到所含的 signal 的 connect. 调用操作符则检查其布尔参数的值来看是否有车辆进入或离开，并且调用相应的函数 enter 或 leave. 在函数 enter 中，首先做的是在车辆标识的 vector 中进行查找。如果找到该标识则说明有问题；可能有人偷了车号牌。查找采用的是算法 binary\_search, 它要求容器是有序的(我们必须要确保它总是有序的)。如果我们发现标识已存在，就立即触发警报，即调用 signal 。

[3] binary\_search 的复杂度为 O(logN).

(\*alarm\_)(car\_id);

首先我们需要解引用 alarm\_ ，因为 alarm\_ 是一个 boost::shared\_ptr, 而在调用它时，我们传给它一个表示车辆标识的参数。如果我们没有找到该标识，则一切正常， 我们就把这个车辆标识插入到 cars\_ 的正确位置中。记住我们必须保证容器随时有序，最好的办法就是把元素插入到一个不会影响顺序的位置上。算法 lower\_bound 可以给我们指出这个位置(该算法同样要求有序序列)。最后一个是函数 leave, 它在有车辆离开停车场时被调用。leave 先确认车辆的标识是否已登记在我们的容器中。这是通过调用算法 equal\_range 来实现的，该算法返回一对迭代器，表示了一个元素可以插入且不影响有序性的范围。这意味着我们必须解引用这个返回的迭代器并确认它的值是否等于我们要查找的那个。如果我们没有找到，我们就要再一次触发警报，而如果我们找到了，就只需要简单地把它从 vector 中删掉。你也许留意到我们没有给出停车者交费的代码；这种有害的代码超出了本书的范围。

我们的停车场所需的各个参与者都已经定义好了，我们必须连接这些 signals 和这些插槽，否则不会发生任何事情！gate 类不知道任何关于 parking\_lot\_guard 类的东西，同样后者也不知道任何关于 security\_guard 类的东西。这就是本库的一个特性：产生事件的类型不需要对接收事件的类型有任何了解。回到这个例子上，我们来看看是否可以让这个停车场运作起来。

int main() {  
// 创建一些警卫  
std::vector<security\_guard> security\_guards;  
security\_guards.push\_back("Bill");  
security\_guards.push\_back("Bob");  
security\_guards.push\_back("Bull");  
// 创建两个门  
gate gate1;  
gate gate2;  
  
// 创建自动监视器  
parking\_lot\_guard plg;  
  
// 把自动监视器连接到门上  
gate1.connect\_to\_gate(plg);  
gate2.connect\_to\_gate(plg);  
  
// 把警卫连接到自动监视器上  
for (unsigned int i=0;i<security\_guards.size();++i) {  
plg.connect\_to\_alarm(security\_guards[i]);  
}  
  
std::cout << "A couple of cars enter...\n";  
gate1.enter("SLN 123");  
gate2.enter("RFD 444");  
gate2.enter("IUY 897");  
  
std::cout << "\nA couple of cars leave...\n";  
gate1.leave("IUY 897");  
gate1.leave("SLN 123");  
  
std::cout << "\nSomeone is entering twice - \  
or is it a stolen license plate?\n";  
gate1.enter("RFD 444");  
}

这就是你要的，一个具有完整功能的停车场。我们创建了三个 security\_guards, 两个 gates, 和一个 parking\_lot\_guard. 它们相互之间一无所知，但我们还是要通过正确的架构把它们联系起来，停车场中发生的重要事件才得以相互传递。这意味着要把 parking\_lot\_guard 连接到两个 gates 上。

gate1.connect\_to\_gate(plg);  
gate2.connect\_to\_gate(plg);

这样就确保了无论何时 gate 实例中产生了 signal enter\_or\_leave\_ 信号，parking\_lot\_guard 都可以收到这个事件通知。接着，我们再将 security\_guards 连接到 parking\_lot\_guard 中的警报 signal 上。

plg.connect\_to\_alarm(security\_guards[i]);

我们已经设法将这些类型相互之间进行了解耦，它们还是得到了执行它们的职责所需的适量的信息。在前面的代码中，我们让少量的车辆进入和离开，来测试这个停车场。这个真实世界的模拟显示了我们已经让各个模块按要求相互通信了。

A couple of cars enter...  
parking\_lot\_guard::enter(SLN 123)  
parking\_lot\_guard::enter(RFD 444)  
parking\_lot\_guard::enter(IUY 897)  
  
A couple of cars leave...  
parking\_lot\_guard::leave(IUY 897)  
parking\_lot\_guard::leave(SLN 123)  
  
Someone is entering twice - or is it a stolen license plate?  
parking\_lot\_guard::enter(RFD 444)  
Bill says: Man, that coffee tastes f.i.n.e fine!  
Bob says: Man, that coffee tastes f.i.n.e fine!  
Bull says: Man, that coffee tastes f.i.n.e fine!

可惜的是，拿着车牌 RFD 444 的骗子跑掉了，但是你能做的就是这些。

关于 signals 的参数已经讨论了很长一段篇幅，事实上我们更多是在讨论 Signals 的基本用法，即对产生 signals 的类型和监听它的插槽进行解耦。记住，任何类型的参数都可以传递，而 signal 类型的声明决定了插槽函数的签名，该声明看起来就象一个不带函数名的函数声明。我们根本没有提到返回类型，虽然它也是签名的一部分。这个疏忽的原因是返回类型可以有多种不同的处理方法，接下来我们将看到为什么会这样以及如何去做。

对结果进行组合

如果一个 signal 的签名以及它的插槽具有非void 的返回类型，显然对于插槽的返回值会有事发生，事实上，那个对 signal 的调用将产生某种结果。但是结果是什么呢？signal 类模板有一个参数名为 Combiner, 它就是负责组合并返回结果的一个类型。缺省的 Combiner 是 boost::last\_value, 它是一个类，只负责简单地返回所调用的最后一个插槽的返回值。那么，究竟是哪一个插槽呢？我们真的不知道，因为调用同一个组内的插槽的顺序是不确定的。我们从一个小例子来示范一下缺省的 Combiner 。

[4] 所以，假设最后一个组中只有一个插槽，我们就可以知道。

#include <iostream>  
#include "boost/signals.hpp"  
  
bool always\_return\_true() {  
return true;  
}  
  
bool always\_return\_false() {  
return false;  
}  
  
int main() {  
boost::signal<bool ()> sig;  
  
sig.connect(&always\_return\_true);  
sig.connect(&always\_return\_false);  
  
std::cout << std::boolalpha << "True or false? " << sig();  
}

有两个插槽，always\_return\_true 和 always\_return\_false, 被连接到 signal sig, 每个都返回一个 bool 且不带参数。调用 sig 的结果被输出到 cout. 它会是 true 还是 false? 不经测试的话，我们无法知道(我试了一上，结果是 false)。在实践中，你要么不关心调用 signal 所返回的值，要么你就要创建你自己的 Combiner 来提供有意义的、客户化的行为。例如，可能是对所有插槽返回的结果进行处理后得到调用 signal 的最终结果。另一种情况，也可能是在某一个插槽返回false 后就不再调用其它的插槽。一个定制的 Combiner 可以做到这些，甚至更多。这是因为 Combiner 可以对插槽进行逐个调用，并根据返回值来决定做什么。

想象一个初始化序列，其中任何失败都将中止整个序列。插槽可以根据它们被调用的次序来指定到组中。没有一个定制的 Combiner 的话，它看起来就象这样：

#include <iostream>  
#include "boost/signals.hpp"  
  
bool step0() {  
std::cout << "step0 is ok\n";  
return true;  
}  
  
bool step1() {  
std::cout << "step1 is not ok. This won't do at all!\n";  
return false;  
}  
  
bool step2() {  
std::cout << "step2 is ok\n";  
return true;  
}  
  
int main() {  
boost::signal<bool ()> sig;  
sig.connect(0,&step0);  
sig.connect(1,&step1);  
sig.connect(2,&step2);  
  
bool ok=sig();  
  
if (ok)  
std::cout << "All system tests clear\n";  
else  
std::cout << "At least one test failed. Aborting.\n";  
}

以上这段代码没有办法让代码知道其中有一个测试是失败的。你也记得，缺省的 combiner 是 boost::last\_value, 它只是简单地返回最后一个插槽的返回值，即调用 step2 的返回值。运行这个例子会给出一个令人失望的输出：

step0 is ok  
step1 is not ok. This won't do at all!  
step2 is ok  
All system tests clear

显然这不是正确的结果。我们需要一个 Combiner ，它应该在某个插槽返回 false 时中止处理，并把结果传回给 signal. 一个 Combiner 就是一个具有某些额外要求的函数对象。它必须有一个名为 result\_type 的 typedef，用于指定其调用操作符的返回类型。此外，调用操作符必须以它被调用的迭代器类型泛化。我们这里需要的 Combiner 非常简单，因此它恰好是一个好的例子。

class stop\_on\_failure {  
public:  
typedef bool result\_type;  
  
template <typename InputIterator>  
bool operator()(InputIterator begin,InputIterator end) const  
{  
while (begin!=end) {  
if (!\*begin)  
return false;  
++begin;  
}  
return true;  
}  
};

注意，公有的 typedef result\_type, 它定义为 bool. result\_type 的类型无需与插槽的返回类型相关。(在声明 signal 时，你指定了插槽的签名以及 signal 的调用操作符的参数。但是，Combiner 的返回类型决定了 signal 的调用操作符的返回类型。缺省情况下，它与插槽的返回类型相同，但这不是必须的)。stop\_on\_failure 的调用操作符以一个插槽迭代器类型所泛化，它对插槽进行逐个迭代并调用；直到我们遇到一个错误为止。对于 stop\_on\_failure, 我们不想在遇到错误的返回值后再继续调用插槽，因此我们对于每次调用都检查其返回值。如果返回值为 false, 该函数说立即返回，否则它继续调用下一个插槽。要使用这个 stop\_on\_failure, 我们只需在声明 signal 类型时指出即可：

boost::signal<bool (),stop\_on\_failure> sig;

如果我们在前面的例子中使用它，则输出的结果就会符合我们的要求了。

step0 is ok  
step1 is not ok. This won't do at all!  
At least one test failed. Aborting.

Combiner 的另一个常用类型是，返回所有被调用插槽的返回值中的最大或最小值。还有其它很多有趣的 Combiners，包括：将所有结果保存在一个容器中。本库的(优秀的)在线文档就有这么一个 Combiner 的例子，你应该去读一下！你并不是每天都需要编写自己的 Combiner 类，但偶尔在为一个复杂的问题给出一个漂亮的解决方案时可能会用到。

Signals 决不能复制

我已经提到过，signals 不能被复制，但是值得留意的是，应该怎样实现一个包含 signal 的类。这些类也都必须是不可复制的吗？不，它们不必，但必须手工实现其复制构造函数和赋值操作符。因为 signal 类将其复制构造函数和赋值操作符声明为私有的，所以一个聚合了 signals 的类必须实现其所需的语义。正确处理复制的一个方法是，在类的多个实例间共享 signals，我们在停车场的例子中就是这么做的。在那个例子中，每一个 parking\_lot\_guard 实例通过 boost::shared\_ptr 引向同一个 signal。对于其它类，可以在拷贝中缺省构造 signal，因为该复制语义不包含对插槽的连接。另一种情况是，复制一个含有 signal 的类是没有意义的，这种情况下你可以依赖所含 signal 的不可复制语义来确保复制与赋值是被禁止的。为了看得更清楚一点，考虑一个类 some\_class, 它的定义是：

class some\_class {  
boost::signal<void (int)> some\_signal;  
};

对于这个类，编译器生成的复制构造函数和赋值操作符都是不能使用的。如果代码企图去使用它们，编译器就会抗议。例如，以下例子试图从 sc1 复制构造 some\_class sc2 ：

int main() {  
some\_class sc1;  
some\_class sc2(sc1);  
}

编译这段程序时，编译器生成的复制构造函数试图对 some\_class 的成员进行逐个成员的复制。由于 signal 的私有复制构造函数，编译器会输出以下信息：

c:/boost\_cvs/boost/boost/noncopyable.hpp: In copy constructor `  
boost::signals::detail::signal\_base::signal\_base(const  
boost::signals::detail::signal\_base&)':  
c:/boost\_cvs/boost/boost/noncopyable.hpp:27: error: `  
boost::noncopyable::noncopyable(  
const boost::noncopyable&)' is private  
noncopyable\_example.cpp:10: error: within this context

所以，无论你的含有 signal 的类需要哪一种复制和赋值，你都必须确保其中不会有对 signal 的复制！