10.1 io_state_savers

C++标准库为输入输出提供了流操作功能，它代替了 C 的文件 handle 和相关的函数，使用起来灵活方便，而且支持国际化，是一个功能强大且富有弹性的处理框架。

流操作一般都很简单明了，但它的状态管理却是个麻烦的事情。如果执行了某些改变状态的操作，如流重定向、修改了输出格式标志（如 std::hex），则流的状态就会一直改变下去，而这通常不是程序的本意。我们必须手工保存流的状态，并在使用完后及时恢复，以避免流发生错误——这是个麻烦却又不得不做的工作。

io_state_savers 库可以简化恢复流状态的工作，它能够保存流的当前状态，自动恢复流的状态或者由程序员控制恢复的时机。

io_state_savers 库位于名字空间 boost::io，为了使用 io_state_savers 组件，需要包含头文件<boost/io/ios_state.hpp>，即：

```
#include <boost/io/ios_state.hpp>
using namespace boost::io;
```


10.1.1 类摘要

io_state_savers 库包含很多 saver 类，它们被细分用于保存流的不同状态，如数字精度、输出宽度、流缓冲、填充字符等，但基本的形式都差不多，大概是这样：

```
class ios_saver                                    //伪代码
{
    typedef std::ios_base           state_type;
    typedef implementation_defined  aspect_type;

    explicit  saver_class( state_type &s );
    saver_class( state_type &s, aspect_type const &new_value );
    ~saver_class();

    void  restore();
};
```

ios_saver 有两个内部类型定义:state_type 和 aspect_type，它们标记了 ios_saver 的基本属性:

- state_type 是 ios_saver 要保存的流类型，如 ios_base、basic_ios<>;
- aspect_type 是 ios_saver 要保存的具体流状态，如 fmtflags、iostate。

ios_saver 的构造函数接受一个流对象的引用，同时保存该流的 aspect_type 值。另一种形

式的构造函数则在保存的同时变更流的状态为 new_value。

ios_saver 在析构时会自动把保存的状态恢复到流对象，程序员也可以在任何时刻使用 restore()函数手工恢复流状态。

ios_saver 把赋值操作符 operator=声明为私有的，因此都是不可赋值的，但允许拷贝构造，可以在程序中拷贝一个 ios_saver 以备后用。

10.1.2 用法

io_state_savers 库里有很多用于保存并恢复流状态的类，它们被分成四组，分别是：

- 基本的标准属性保存器，如 ios_flags_saver、ios_width_saver；
- 增强的标准属性保存器，如 ios_iostate_saver、ios_rdbuf_saver；
- 自定义的属性保存器，如 ios_iword_saver、ios_pword_saver；
- 组合的属性保存器，如 ios_all_saver。

最简单也是最常用的类是 ios_all_saver 和 wios_all_saver，它们可以保存流的所有状态，让我们不必费心去决定保存流的哪个状态。wios_all_saver 是 ios_all_saver 的宽字符形式，用来配合宽字符流 wcout 使用。

假设我们有如下的日志函数 logging()，它用来向标准输出打印日志信息：

```
void logging(const char* msg)
{ cout << msg << endl;}
```

使用流的重定向功能，我们可以把日志的输出转到一个文件流中，这样可以把日志保存起来用于日后审查：

```
ofstream fs("logfile.log");                    //输出文件流
cout.rdbuf(fs.rdbuf());                        //cout 流重定向
logging("fatal msg");                          //向文件流输出日志
```

那么下面的代码将会发生崩溃的错误：

```
int main()
{
  string filename("c:/test.txt");
  cout << "log start" << endl;                 //开始
  if(!filename.empty())
  {
      ofstream fs(filename.c_str());           //打开一个文件流
      cout.rdbuf(fs.rdbuf());                  //标准输出重定向到文件流
```

```
        logging("fatal msg");                    //重定向输出日志
    }                                            //文件流被自动析构!!
    cout << "log finish" << endl;                //cout 无法输出，运行错误
}
```

出错的原因在于流的重定向。当文件流 fs 被设置为 cout 的缓冲区后，cout 将总向它输出数据。但 fs 是一个局部变量，当离开 if 语句的作用域后它被自动销毁，导致缓冲区失效。但 cout 对此并不知情，仍然向一个无效缓冲区写入数据，从而发生了严重的运行错误。

要修复这个 bug 非常容易，只需要在重定向或任何可能改变流状态的操作前用 ios_all_saver 保存 cout 的状态，那么在离开作用域时它就会自动把 cout 恢复到最初的状态，从而避免了灾难的发生：

```
{
    ...
    ios_all_saver ifs(cout);                 //保存流的所有状态
    cout.rdbuf(fs.rdbuf());
    ...
}                                            //离开作用域，导致保存器析构，自动恢复流的状态
```

10.1.3 简化 new_progress_timer

在 2.3.3 节我们实现了一个 progress_timer 的扩展类——new_progress_timer，它可以度量很高精度的时间，但析构函数的流状态的保存与恢复代码十分繁琐，现在到了使用 io_state_savers 库来改进它代码的时候了。

我们直接使用 ios_all_saver 来保存标准输出流 cout 的状态，将 new_progress_timer 的析构函数修改如下：

```
#include <boost/io/ios_state.hpp>
template<int N = 2 >
class new_progress_timer:public boost::timer
{
    ...
    ~new_progress_timer(void)                                    //析构函数
    {
        try
        {
            //保存流的状态
            boost::io::ios_all_saver ios(m_os);

            //设置输出格式
            m_os.setf( std::istream::fixed, std::istream::floatfield );
            m_os.precision( N );
```

```
            //输出时间
            m_os << elapsed() << " s\n" << std::endl;        //"s" 表示秒
        }                                                    //自动恢复流状态
}
```