C++11（及现代C++风格）和快速迭代式开发

By
刘未鹏
– August 27, 2012**Posted in:** 编程

过去的一年我在微软亚洲研究院做输入法，我们的产品叫“英库拼音输入法”（下载Beta版），如果你用过“英库词典”（现已更名为必应词典），应该知道“英库”这个名字（实际上我们的核心开发团队也有很大一部分来源于英库团队的老成员）。整个项目是微软亚洲研究院的自然语言处理组、互联网搜索与挖掘组和我们创新工程中心，以及微软中国Office商务软件部（MODC）多组合作的结果。至于我们的输入法有哪些创新的feature，以及这些feature背后的种种有趣故事… 本文暂不讨论。虽然整个过程中我也参与了很多feature的设想和设计，但90%的职责还是开发，所以作为client端的核心开发人员之一，我想跟大家分享这一年来在项目中全面使用C++11以及现代C++风格（Elements of Modern C++ Style）来做开发的种种经验。

我们用的开发环境是VS2010 SP1，该版本已经支持了相当多的C++11的特性：lambda表达式，右值引用，auto类型推导，static_assert，decltype，nullptr，exception_ptr等等。C++曾经饱受“学院派”标签的困扰，不过这个标签着实被贴得挺冤，C++11的新feature没有一个是从学院派角度出发来设计的，以上提到的所有这些feature都在我们的项目中得到了适得其所的运用，并且带来了很大的收益。尤其是lambda表达式。

说起来我跟C++也算是有相当大的缘分，03年还在读本科的时候，第一篇发表在程序员上面的文章就是Boost库的源码剖析，那个时候Boost库在国内还真是相当的阳春白雪，至今已经快十年了，Boost库如今已经是写C++代码不可或缺的库，被誉为“准标准库”，C++的TR1基本就脱胎于Boost的一系列子库，而TR2同样也大量从Boost库中取材。之后有好几年，我在CSDN上的博客几乎纯粹是C++的前沿技术文章，包括从06年就开始写的“C++0x漫谈”系列。（后来写技术文章写得少了，也就把博客从CSDN博客独立了出来，便是现在的mindhacks.cn）。自从独立博客了之后我就没有再写过C++相关的文章（不过仍然一直对C++的发展保持了一定的关注），一方面我喜欢关注前沿的进展，写完了Boost源码剖析系列和C++0x漫谈系列之后我觉得这一波的前沿进展从大方面来说也都写得差不多了，所以不想再费时间。另一方面的原因也是我虽然对C++关注较深，但实践经验却始终绝大多数都是“替代经验”，即从别人那儿看来的，并非自己第一手的。而过去一年来深度参与的英库输入法项目弥补了这个缺憾，所以我就决定重新开始写一点C++11的实践经验。算是对努力一年的项目发布第一版的一个小结。

09年入职微软亚洲研究院之后，前两年跟C++基本没沾边，第一个项目倒是用C++的，不过是工作在既有代码基上，时间也相对较短。第二个项目为Bing Image Search用javascript写前端，第三个项目则给Visual Studio 2012写Code Clone Detection，用C#和WPF。直到一年前英库输入法这个项目，是我在研究院的第四个项目了，也是最大的一个，一年来我很开心，因为又回到了C++。

这个项目我们从零开始，，而client端的核心开发人员也很紧凑，只有3个。这个项目有很多特殊之处，对高效的快速迭代开发提出了很大的挑战（研究院所倡导的“以实践为驱动的研究（Deployment-Driven-Research）”要求我们迅速对用户的需求作出响应）：

1. 长期时间压力：从零开始到发布，只有一年时间，我们既要在主要feature上能和主流的输入法相较，还需要实现我们自己独特的创新feature，从而能够和其他输入法产品区分开来。
2. 短期时间压力：输入法在中国是一个非常成熟的市场，谁也没法保证闷着头搞一年搞出来的东西就一炮而红，所以我们从第一天起就进入demo驱动的准迭代式开发，整个过程中必须不断有阶段性输出，抬头看路好过闷头走路。但工程师最头疼的二难问题之一恐怕就是短期与长远的矛盾：要持续不断出短期的成果，就必须经常在某些地方赶工，赶工的结果则可能导致在设计和代码质量上面的折衷，这些折衷也被称为Technical Debt（技术债）。没有任何项目没有技术债，只是多少，以及偿还的方式的区别。我们的目的不是消除技术债，而是通过不断持续改进代码质量，阻止技术债的滚雪球式积累。
3. C++是一门不容易用好的语言：错误的使用方式会给代码基的质量带来很大的损伤。而C++的误用方式又特别多。
4. 输入法是个很特殊的应用程序，在Windows下面，输入法是加载到目标进程空间当中的dll，所以，输入法对质量的要求极高，别的软件出了错误崩溃了大不了重启一下，而输入法如果崩溃就会造成整个目标进程崩溃，如果用户的文档未保存就可能会丢失宝贵的用户数据，所以输入法最容不得崩溃。可是只要是人写的代码怎么可能没有bug呢？所以关键在于如何减少bug及其产生的影响和如何能尽快响应并修复bug。所以我们的做法分为三步：1). 使用现代C++技术减少bug产生的机会。2). 即便bug产生了，也尽量减少对用户产生的影响。3). 完善的bug汇报系统使开发人员能够第一时间拥有足够的信息修复bug。

至于为什么要用C++而不是C呢？对于我们来说理由很现实：时间紧任务重，用C的话需要发明的轮子太多了，C++的抽象层次高，代码量少，bug相对就会更少，现代C++的内存管理完全自动，以至于从头到尾我根本不记得曾遇到过什么内存管理相关的bug，现代C++的错误处理机制也非常适合快速开发的同时不用担心bug乱飞，另外有了C++11的强大支持更是如虎添翼，当然，这一切都必须建立在核心团队必须善用C++的大前提上，而这对于我们这个紧凑的小团队来说这不是问题，因为大家都有较好的C++背景，没有陡峭的学习曲线要爬。（至于C++在大规模团队中各人对C++的掌握良莠不齐的情况下所带来的一些包袱本文也不作讨论，呵呵，语言之争别找我。）

下面就说说我们在这个项目中是如何使用C++11和现代C++风格来开发的，什么是现代C++风格以及它给我们开发带来的好处。

资源管理

说到Native Languages就不得不说资源管理，因为资源管理向来都是Native Languages的一个大问题，其中内存管理又是资源当中的一个大问题，由于堆内存需要手动分配和释放，所以必须确保内存得到释放，对此一般原则是“谁分配谁负责释放”，但即便如此仍然还是经常会导致内存泄漏、野指针等等问题。更不用说这种手动释放给API设计带来的问题（例如Win32 API WideCharToMultiByte就是一个典型的例子，你需要提供一个缓冲区给它来接收编码转换的结果，但是你又不能确保你的缓冲区足够大，所以就出现了一个两次调用的pattern，第一次给个NULL缓冲区，于是API返回的是所需的缓冲区的大小，根据这个大小分配缓冲区之后再第二次调用它，别提多别扭了）。

托管语言们为了解决这个问题引入了GC，其理念是“内存管理太重要了，不能交给程序员来做”。但GC对于Native开发也常常有它自己的问题。而且另一方面Native界也常常诟病GC，说“内存管理太重要了，不能交给机器来做”。

C++也许是第一个提供了完美折衷的语言（不过这个机制直到C++11的出现才真正达到了易用的程度），即：既不是完全交给机器来做，也不是完全交给程序员来做，而是程序员先在代码中指定怎么做，至于什么时候做，如何确保一定会得到执行，则交由编译器来确定。

首先是C++98提供了语言机制：对象在超出作用域的时候其析构函数会被自动调用。接着，Bjarne Stroustrup在TC++PL里面定义了RAII（Resource Acquisition is Initialization）范式（即：对象构造的时候其所需的资源便应该在构造函数中初始化，而对象析构的时候则释放这些资源）。RAII意味着我们应该用类来封装和管理资源，对于内存管理而言，Boost第一个实现了工业强度的智能指针，如今智能指针（shared_ptr和unique_ptr）已经是C++11的一部分，简单来说有了智能指针意味着你的C++代码基中几乎就不应该出现delete了。

不过，RAII范式虽然很好，但还不足够易用，很多时候我们并不想为了一个CloseHandle, ReleaseDC, GlobalUnlock等等而去大张旗鼓地另写一个类出来，所以这些时候我们往往会因为怕麻烦而直接手动去调这些释放函数，手动调的一个坏处是，如果在资源申请和释放之间发生了异常，那么释放将不会发生，此外，手动释放需要在函数的所有出口处都去调释放函数，万一某天有人修改了代码，加了一处return，而在return之前忘了调释放函数，资源就泄露了。理想情况下我们希望语言能够支持这样的范式：

```
void foo()
{
    HANDLE h = CreateFile(...);

    ON_SCOPE_EXIT { CloseHandle(h); }

    ... // use the file
}
```

ON_SCOPE_EXIT里面的代码就像是在析构函数里面的一样：不管当前作用域以什么方式退出，都必然会被执行。

实际上，早在2000年，Andrei Alexandrescu 就在DDJ杂志上发表了一篇文章，提出了这个叫做ScopeGuard 的设施，不过当时C++还没有太好的语言机制来支持这个设施，所以Andrei动用了你所能想到的各种奇技淫巧硬是造了一个出来，后来Boost也加入了ScopeExit库，不过这些都是建立在C++98不完备的语言机制的情况下，所以其实现非常不必要的繁琐和不完美，实在是戴着脚镣跳舞（这也是C++98的通用库被诟病的一个重要原因），再后来Andrei不能忍了就把这个设施内置到了D语言当中，成了D语言特性的一部分（最出彩的部分之一）。

再后来就是C++11的发布了，C++11发布之后，很多人都开始重新实现这个对于异常安全来说极其重要的设施，不过绝大多数人的实现受到了2000年Andrei的原始文章的影响，多多少少还是有不必要的复杂性，而实际上，将C++11的Lambda Function和tr1::function结合起来，这个设施可以简化到脑残的地步：

```
class ScopeGuard
{
public:
    explicit ScopeGuard(std::function<void()> onExitScope)
        : onExitScope_(onExitScope), dismissed_(false)
    { }

    ~ScopeGuard()
    {
        if(!dismissed_)
        {
            onExitScope_();
        }
    }

    void Dismiss()
    {
        dismissed_ = true;
    }

private:
    std::function<void()> onExitScope_;
    bool dismissed_;

private: // noncopyable
    ScopeGuard(ScopeGuard const&);
    ScopeGuard& operator=(ScopeGuard const&);
};
```

这个类的使用很简单，你交给它一个std::function，它负责在析构的时候执行，绝大多数时候这个function就是lambda，例如：

```
HANDLE h = CreateFile(...);
ScopeGuard onExit([&] { CloseHandle(h); });
```

onExit在析构的时候会忠实地执行CloseHandle。为了避免给这个对象起名的麻烦（如果有多个变量，起名就麻烦大了），可以定义一个宏，把行号混入变量名当中，这样每次定义的ScopeGuard对象都是唯一命名的。

```
#define SCOPEGUARD_LINENAME_CAT(name, line) name##line
#define SCOPEGUARD_LINENAME(name, line) SCOPEGUARD_LINENAME_CAT(name, line)

#define ON_SCOPE_EXIT(callback) ScopeGuard SCOPEGUARD_LINENAME(EXIT, __LINE__)(callback)
```

Dismiss()函数也是Andrei的原始设计的一部分，其作用是为了支持rollback模式，例如：

```
ScopeGuard onFailureRollback([&] { /* rollback */ });
... // do something that could fail
onFailureRollback.Dismiss();
```

在上面的代码中，“do something”的过程中只要任何地方抛出了异常，rollback逻辑都会被执行。如果“do something”成功了，onFailureRollback.Dismiss()会被调用，设置dismissed_为true，阻止rollback逻辑的执行。

ScopeGuard是资源自动释放，以及在代码出错的情况下rollback的不可或缺的设施，C++98由于没有lambda和tr1::function的支持，ScopeGuard不但实现复杂，而且用起来非常麻烦，陷阱也很多，而C++11之后立即变得极其简单，从而真正变成了每天要用到的设施了。

C++的RAII范式被认为是资源确定性释放的最佳范式（C#的using关键字在嵌套资源申请释放的情况下会层层缩进，相当的不能scale），而有了ON_SCOPE_EXIT之后，在C++里面申请释放资源就变得非常方便

```
Acquire Resource1
ON_SCOPE_EXIT( [&] { /* Release Resource1 */ })

Acquire Resource2
ON_SCOPE_EXIT( [&] { /* Release Resource2 */ })
…
```

这样做的好处不仅是代码不会出现无谓的缩进，而且资源申请和释放的代码在视觉上紧邻彼此，永远不会忘记。更不用说只需要在一个地方写释放的代码，下文无论发生什么错误，导致该作用域退出我们都不用担心资源不会被释放掉了。我相信这一范式很快就会成为所有C++代码分配和释放资源的标准方式，因为这是C++十年来的演化所积淀下来的真正好的部分之一。

错误处理

前面提到，输入法是一个特殊的东西，某种程度上他就跟用户态的driver一样，对错误的宽容度极低，出了错误之后可能造成很严重的后果：用户数据丢失。不像其他独立跑的程序可以随便崩溃大不了重启（或者程序自动重启），所以从一开始，错误处理就被非常严肃地对待。

这里就出现了一个两难问题：严谨的错误处理要求不要忽视和放过任何一个错误，要么当即处理，要么转发给调用者，层层往上传播。任何被忽视的错误，都迟早会在代码接下去的执行流当中引发其他错误，这种被原始错误引发的二阶三阶错误可能看上去跟root cause一点关系都没有，造成bugfix的成本剧增，这是我们项目快速的开发步调下所承受不起的成本。

然而另一方面，要想不忽视错误，就意味着我们需要勤勤恳恳地检查并转发错误，一个大规模的程序中随处都可能有错误发生，如果这种检查和转发的成本太高，例如错误处理的代码会导致代码增加，结构臃肿，那么程序员就会偷懒不检查。而一时的偷懒以后总是要还的。

所以细心检查是短期不断付出成本，疏忽检查则是长期付出成本，看上去怎么都是个成本。有没有既不需要短期付出成本，又不会导致长期付出成本的办法呢？答案是有的。我们的项目全面使用异常来作为错误处理的机制。异常相对于错误代码来说有很多优势，我曾经在2007年写过一篇博客《错误处理：为何、何时、如何》进行了详细的比较，但是异常对于C++而言也属于不容易用好的特性：

首先，为了保证当异常抛出的时候不会产生资源泄露，你必须用RAII范式封装所有资源。这在C++98中可以做到，但代价较大，一方面智能指针还没有进入标准库，另一方面智能指针也只能管内存，其他资源莫非还都得费劲去写一堆wrapper类，这个不便很大程度上也限制了异常在C++98下的被广泛使用。不过幸运的是，我们这个项目开始的时候VS2010 SP1已经具备了tr1和lambda function，所以写完上文那个简单的ScopeGuard之后，资源的自动释放问题就非常简便了。

其次，C++的异常不像C#的异常那样附带Callstack。例如你在某个地方通过.at(i)来取一个vector的某个元素，然后i越界了，你会收到vector内部抛出来的一个异常，这个异常只是说下标越界了，然后什么其他信息都木有，连个行号都没有。要是不抛异常直接让程序崩溃掉好歹还可以抓到一个minidump呢，这个因素一定程度上也限制了C++异常的被广泛使用。Callstack显然对于我们迅速诊断程序的bug有至关重要的作用，由于我们是一个不大的团队，所以我们对质量的测试很依赖于微软内部的dogfood用户，我们release给dogfood用户的是release版，倘若我们不用异常，用assert的话，固然是可以在release版也打开assert，但assert同样也只能提供很有限的信息（文件和行号，以及assert的表达式），很多时候这些信息是不足够理解一个bug的（更不用说还得手动截屏拷贝黏贴发送邮件才能汇报一个bug了），所以往往接下来还需要在开发人员自己的环境下试图重现bug。这就不够理想了。理想情况下，一个bug发生的时刻，程序应该自己具备收集一切必要的信息的能力。那么对于一个bug来说，有哪些信息是至关重要的呢？

1. Error Message本身，例如“您的下标越界啦！”少部分情况下，光是Error Message已经足够诊断。不过这往往是对于在开发的早期出现的一些简单bug，到中后期往往这类简单bug都被清除掉了，剩下的较为隐蔽的bug的诊断则需要多得多的信息。
2. Callstack。C++的异常由于性能的考虑，并不支持callstack。所以必须另想办法。
3. 错误发生地点的上下文变量的值：例如越界访问，那么越界的下标的值是多少，而被越界的容器的大小又是多少，等等。例如解析一段xml失败了，那么这段xml是什么，当前解析到哪儿，等等。例如调用Win32 API失败了，那么Win32 Error Message是什么。
4. 错误发生的环境：例如目标进程是什么。
5. 错误发生之前用户做了什么：对于输入法来说，例如错误发生之前的若干个键敲击。

如果程序能够自动把这些信息收集并打包起来，发送给开发人员，那么就能够为诊断提供极大的帮助（当然，既便如此仍然还是会有难以诊断的bug）。而且这一切都要以不增加写代码过程中的开销的方式来进行，如果每次都要在代码里面做一堆事情来收集这些信息，那烦都得烦死人了，没有人会愿意用的。

那么到底如何才能无代价地尽量收集充足的信息为诊断bug提供帮助呢？

首先是callstack，有很多种方法可以给C++异常加上callstack，不过很多方法会带来性能损失，而且用起来也不方便，例如在每个函数的入口处加上一小段代码把函数名/文件/行号打印到某个地方，或者还有一些利用dbghelp.dll里面的StackWalk功能。我们使用的是没有性能损失的简单方案：在抛C++异常之前先手动MiniDumpWriteDump，在异常捕获端把minidump发回来，在开发人员收到minidump之后可以使用VS或windbg进行调试（但前提是相应的release版本必须开启pdb）。可能这里你会担心，minidump难道不是很耗时间的嘛？没错，但是既然程序已经发生了异常，稍微多花一点时间也就无所谓了。我们对于“附带minidump的异常”的使用原则是，只在那些真正“异常”的情况下抛出，换句话说，只在你认为应该使用的assert的地方用，这类错误属于critical error。另外我们还有不带minidump的异常，例如网络失败，xml解析失败等等“可以预见”的错误，这类错误发生的频率较高，所以如果每次都minidump会拖慢程序，所以这种情况下我们只抛异常不做minidump。

然后是Error Message，如何才能像assert那样，在Error Message里面包含表达式和文件行号？

最后，也是最重要的，如何能够把上下文相关变量的值capture下来，因为一方面release版本的minidump在调试的时候所看到的变量值未必正确，另一方面如果这个值在堆上（例如std::string的内部buffer就在堆上），那就更看不着了。

所有上面这些需求我们通过一个ENSURE宏来实现，它的使用很简单：

```
ENSURE(0 <= index && index < v.size())(index)(v.size());
```

ENSURE宏在release版本中同样生效，如果发现表达式求值失败，就会抛出一个C++异常，并会在异常的.what()里面记录类似如下的错误信息：

```
Failed: 0 <= index && index < v.size()
File: xxx.cpp Line: 123
Context Variables:
    index = 12345
    v.size() = 100
```

（如果你为stream重载了接收vector的operator <<，你甚至可以把vector的元素也打印到error message里头）

由于ENSURE抛出的是一个自定义异常类型ExceptionWithMinidump，这个异常有一个GetMinidumpPath()可以获得抛出异常的时候记录下来的minidump文件。

ENSURE宏还有一个很方便的feature：在debug版本下，抛异常之前它会先assert，而assert的错误消息正是上面这样。Debug版本assert的好处是可以让你有时间attach debugger，保证有完整的上下文。

利用ENSURE，所有对Win32 API的调用所发生的错误返回值就可以很方便地被转化为异常抛出来，例如：

```
ENSURE_WIN32(SHGetKnownFolderPath(rfid, 0, NULL, &p) == S_OK);
```

为了将LastError附在Error Message里面，我们额外定义了一个ENSURE_WIN32:

```
#define ENSURE_WIN32(exp) ENSURE(exp)(GetLastErrorStr())
```

其中GetLastErrorStr()会返回Win32 Last Error的错误消息文本。

而对于通过返回HRESULT来报错的一些Win32函数，我们又定义了ENSURE_SUCCEEDED(hr)：

```
#define ENSURE_SUCCEEDED(hr) \
    if(SUCCEEDED(hr)) \
else ENSURE(SUCCEEDED(hr))(Win32ErrorMessage(hr))
```

其中Win32ErrorMessage(hr)负责根据hr查到其错误消息文本。

ENSURE宏使得我们开发过程中对错误的处理变得极其简单，任何地方你认为需要assert的，用ENSURE就行了，一行简单的ENSURE，把bug相关的三大重要信息全部记录在案，而且由于ENSURE是基于异常的，所以没有办法被程序忽略，也就不会导致难以调试的二阶三阶bug，此外异常不像错误代码需要手动去传递，也就不会带来为了错误处理而造成的额外的开发成本（用错误代码来处理错误的最大的开销就是错误代码的手工检查和层层传递）。

ENSURE宏的实现并不复杂，打印文件行号和表达式文本的办法和assert一样，创建minidump的办法（这里只讨论win32）是在__try中RaiseException(EXCEPTION_BREAKPOINT…)，在__except中得到EXCEPTION_POINTERS之后调用MiniDumpWriteDump写dump文件。最tricky的部分是如何支持在后面capture任意多个局部变量（ENSURE(expr)(var1)(var2)(var3)…），并且对每个被capture的局部变量同时还得capture变量名（不仅是变量值）。而这个宏无限展开的技术也在大概十年前就有了，还是Andrei Alexandrescu写的一篇DDJ文章：Enhanced Assertions 。神奇的是，我的CSDN博客当年第一篇文章就是翻译的它，如今十年后又在自己的项目中用到，真是有穿越的感觉，而且穿越的还不止这一个，我们项目不用任何第三方库，包括boost也不用，这其实也没有带来什么不便，因为boost的大量有用的子库已经进入了TR1，唯一的不便就是C++被广为诟病的：没有一个好的event实现，boost.signal这种非常强大的工业级实现当然是可以的，不过对于我们的项目来说boost.signal的许多feature根本用不上，属于杀鸡用牛刀了，因此我就自己写了一个刚刚满足我们项目的特定需求的event实现（使用tr1::function和lambda，这个signal的实现和使用都很简洁，可惜variadic templates没有，不然还会更简洁一些）。我在03年写boost源码剖析系列的时候曾经详细剖析了boost.signal的实现技术，想不到十年前关注的技术十年后还会在项目中用到。

由于输入法对错误的容忍度较低，所以我们在所有的出口处都设置了两重栅栏，第一重catch所有的C++异常，如果是ExceptionWithMinidump类型，则发送带有dump的问题报告，如果是其他继承自std::exception的异常类型，则仅发送包含.what()消息的问题报告，最后如果是catch(…)收到的那就没办法了，只能发送“unknown exception occurred”这种消息回来了。

```
inline void ReportCxxException(std::exception_ptr ex_ptr)
{
    try
    {
        std::rethrow_exception(ex_ptr);
    }
    catch(ExceptionWithMiniDump& ex)
    {
        LaunchProblemReporter(…, ex.GetMiniDumpFilePath());
    }
    catch(std::exception& ex)
    {
        LaunchProblemReporter(…, ex.what());
    }
    catch(...)
    {
        LaunchProblemReporter("Unknown C++ Exception"));
    }
}
```

C++异常外面还加了一层负责捕获Win32异常的，捕获到unhandled win32 exception也会写minidump并发回。

考虑到输入法应该“能不崩溃就不崩溃”，所以对于C++异常而言，除了弹出问题报告程序之外，我们并不会阻止程序继续执行，这样做有以下几个原因：

1. 很多时候C++异常并不会使得程序进入不可预测的状态，只要合理使用智能指针和ScopeGuard，该释放的该回滚的操作都能被正确执行。

2. 输入法的引擎的每一个输入session（从开始输入到上词）理论上是独立的，如果session中间出现异常应该允许引擎被reset到一个可知的好的状态。
3. 输入法内核中有核心模块也有非核心模块，引擎属于核心模块，云候选词、换肤、还有我们的创新feature：Rich Candidates（目前被译为多媒体输入，但其实没有准确表达出这个feature的含义，只不过第一批release的apps确实大多是输入多媒体的，但我们接下来会陆续更新一系列的Rich Candidates Apps就不止是多媒体了）也属于非核心模块，非核心模块即便出了错误也不应该影响内核的工作。因此对于这些模块而言我们都在其出口处设置了Error Boundary，捕获一切异常以免影响整个内核的运作。

另一方面，对于Native Language而言，除了语言级别的异常，总还会有Platform Specific的“硬”异常，例如最常见的Access Violation，当然这种异常越少越好（我们的代码基中鼓励使用ENSURE来检查各种pre-condition和post-condition，因为一般来说Access Violation不会是第一手错误，它们几乎总是由其他错误导致的，而这个“其他错误”往往可以用ENSURE来检查，从而在它导致Access Violation之前就抛出语言级别的异常。举一个简单的例子，还是vector的元素访问，我们可以直接v[i]，如果i越界，会Access Violation，那么这个Access Violation便是由之前的第一手错误（i越界）所导致的二阶异常了。而如果我们在v[i]之前先ENSURE(0 <= i && i < v.size())的话，就可以阻止“硬”异常的发生，转而成为汇报一个语言级别的异常，语言级别的异常跟平台相关的“硬”异常相比的好处在于：

1. 语言级别异常的信息更丰富，你可以capture相关的变量的值放在异常的错误消息里面。
2. 语言级别的异常是“同步”的，一个写的规范的程序可以保证在语言级别异常发生的情况下始终处于可知的状态。C++的Stack Unwind机制可以确保一切善后工作得到执行。相比之下当平台相关的“硬”异常发生的时候你既不会有机会清理资源回滚操作，也不能确保程序仍然处于可知的状态。所以语言级别的异常允许你在模块边界上设定Error Boundary并且在非核心模块失败的时候仍然保持程序运行，语言级别的异常也允许你在核心模块，例如引擎的出口设置Error Boundary，并且在出错的情况下reset引擎到一个干净的初始状态。简言之，语言级别的异常让程序更健壮。

理想情况下，我们应该、并且能够通过ENSURE来避免几乎所有“硬”异常的发生。但程序员也是人，只要是代码就会有疏忽，万一真的发生了“硬”异常怎么办？对于输入法而言，即便出现了这种很遗憾的情况我们仍然不希望你的宿主程序崩溃，但另一方面，由于“硬”异常使得程序已经处于不可知的状态，我们无法对程序以后的执行作出任何的保障，所以当我们的错误边界处捕获这类异常的时候，我们会设置一个全局的flag，disable整个的输入法内核，从用户的角度来看就是输入法不工作了，但一来宿主程序没有崩溃，二来你的所有键敲击都会被直接被宿主程序响应，就像没有打开输入法的时候一样。这样一来即便在最坏的情况之下，宿主程序仍然有机会去保存数据并体面退出。

所以，综上所述，通过基于C++异常的ENSURE宏，我们实现了以下几个目的：

1. 极其廉价的错误检查和汇报（和assert一样廉价，却没有assert的诸多缺陷）：尤其是对于快速开发来说，既不可忽视错误，又不想在错误汇报和处理这种（非正事）上消耗太多的时间，这种时候ENSURE是完美的方案。
2. 丰富的错误信息。
3. 不可忽视的错误：编译器会忠实负责stack unwind，不会让一个错误被藏着掖着，最后以二阶三阶错误的方式表现出来，给诊断造成麻烦。
4. 健壮性：看上去到处抛异常会让人感觉程序不够健壮，而实际上恰恰相反，如果程序真的有bug，那么一定会浮现出来，即便你不用异常，也并没有消除错误本身，迟早错误会以其他形式表现出来，在程序的世界里，有错误是永远藏不住的。而异常作为语言级别支持的错误汇报和处理机制，拥有同步和自动清理的特点，支持模块边界的错误屏障，支持在错误发生的时候重置程序到干净的状态，从而最大限度保证程序的正常运行。如果不用异常而用error code，只要疏忽检查一点，迟早会导致“硬”异常，而一旦后者发生，基本剩下的也别指望程序还能正常工作了，能做得最负责任的事情就是别导致宿主崩溃。

另一方面，如果使用error code而不用异常来汇报和处理错误，当然也是可以达到上这些目的，但会给开发带来高昂的代价，设想你需要把每个函数的返回值腾出来用作HRESULT，然后在每个函数返回的时候必须check其返回错误，并且如果自己不处理必须勤勤恳恳地转发给上层。所以对于error code来说，要想快就必须牺牲周密的检查，要想周密的检查就必须牺牲编码时间来做“不相干”的事情（对于需要周密检查的错误敏感的应用来说，最后会搞到代码里面一眼望过去尽是各种if-else的返回值错误检查，而真正干活的代码却缩在不起眼的角落，看过win32代码的同学应该都会有这个体会）。而只有使用异常和ENSURE，才真正实现了既几乎不花任何额外时间、又不至于漏过任何一个第一手错误的目的。

最后简单提一下异常的性能问题，现代编译器对于异常处理的实现已经做到了在happy path上几乎没有开销，对于绝大多数应用层的程序来说，根本无需考虑异常所带来的可忽视的开销。在我们的对速度要求很敏感的输入法程序中，做performance profiling的时候根本看不到异常带来任何可见影响（除非你乱用异常，例如拿异常来取代正常的bool返回值，或者在loop里面抛接异常，等等）。具体的可以参考GoingNative2012@Channel9上的The Importance of Being Native的1小时06分处。

C++11的其他特性的运用

资源管理和错误处理是现代C++风格最醒目的标志，接下来再说一说C++11的其他特性在我们项目中的使用。

首先还是lambda，lambda除了配合ON_SCOPE_EXIT使用威力无穷之外，还有一个巨大的好处，就是创建on-the-fly的tasks，交给另一个线程去执行，或者创建一个delegate交给另一个类去调用（像C#的event那样）。（当然，lambda使得STL变得比原来易用十倍这个事情就不说了，相信大家都知道了），例如我们有一个BackgroundWorker类，这个类的对象在内部维护一个线程，这个线程在内部有一个message loop，不断以Thread Message的形式接收别人委托它执行的一段代码，如果是委托的同步执行的任务，那么委托（调用）方便等在那里，直到任务被执行完，如果执行过程中出现任何错误，会首先被BackgroundWorker捕获，然后在调用方线程上重新抛出（利用C++11的std::exception_ptr、std::current_exception()以及std::rethrow_exception()）。BackgroundWorker的使用方式很简单：

```
bgWorker.Send([&]
{
.. /* do something */
});
```

有了lambda，不仅Send的使用方式像上面这样直观，Send本身的实现也变得很优雅：

```
bool Send(std::function<void()> action)
{
    HANDLE done = CreateEvent(NULL, TRUE, FALSE, NULL);

    std::exception_ptr  pCxxException;
    unsigned int        win32ExceptionCode = 0;
    EXCEPTION_POINTERS* win32ExceptionPointers = nullptr;
```

```
    std::function<void()> synchronousAction = [&]
    {
        ON SCOPE EXIT([&] {
            SetEvent(done);
        });

        AllExceptionsBoundary(
            action,
            [&](std::exception ptr e)
                { pCxxException = e; },
            [&](unsigned int code, EXCEPTION_POINTERS* ep)
                { win32ExceptionCode = code;
                  win32ExceptionPointers = ep; });
    };

    bool r = Post(synchronousAction);

    if(r)
    {
        WaitForSingleObject(done, INFINITE);
        CloseHandle(done);

        // propagate error (if any) to the calling thread
        if(!(pCxxException == nullptr))
        {
            std::rethrow_exception(pCxxException);
        }

        if(win32ExceptionPointers)
        {
            RaiseException(win32ExceptionCode, ..);
        }
    }
    return r;
}
```

这里我们先把外面传进来的function wrap成一个新的lambda function，后者除了负责调用前者之外，还负责在调用完了之后flag一个event从而实现同步等待的目的，另外它还负责捕获任务执行中可能发生的错误并保存下来，留待后面在调用方线程上重新raise这个错误。

另外一个使用lambda的例子是：由于我们项目中需要解析XML的地方用的是MSXML，而MSXML很不幸是个COM组件，COM组件要求生存在特定的Apartment里面，而输入法由于是被动加载的dll，其主线程不是输入法本身创建的，所以主线程到底属于什么Apartment不由输入法来控制，为了确保万无一失，我们便将MSXML host在上文提到的一个专属的BackgroundWorker对象里面，由于BackgroundWorker内部会维护一个线程，这个线程的apartment是由我们全权控制的。为此我们给MSXML创建了一个wrapper类，这个类封装了这些实现细节，只提供一个简便的使用接口：

```
XMLDom dom;
dom.LoadXMLFile(xmlFilePath);

dom.Visit([&](std::wstring const& elemName, IXMLDOMNode* elem)
{
    if(elemHandlers.find(elemName) != elemHandlers.end())
    {
        elemHandlers[elemName](elem);
    }
});
```

基于上文提到的BackgroundWorker的辅助，这个wrapper类的实现也变得非常简单：

```
void Visit(TNodeVisitor const& visitor)
{
    bgWorker .Send([&] {
        ENSURE(pXMLDom_ != NULL);

        IXMLDOMElement* root;
        ENSURE(pXMLDom_->get_documentElement(&root) == S_OK);

        InternalVisit(root, visitor);
    });
}
```

所有对MSXML对象的操作都会被Send到host线程上去执行。

另一个很有用的feature就是static_assert，例如我们在ENSURE宏的定义里面就有一行：

```
static_assert(std::is_same<decltype(expr), bool>::value, "ENSURE(expr) can only be used on bool expression");
```

避免调ENSURE(expr)的时候expr不是bool类型，确给隐式转换成了bool类型，从而出现很隐蔽的bug。

至于C++11的Move Semantics给代码带来的变化则是润物细无声的：你可以不用担心返回vector, string等STL容易的性能问题了，代码的可读性会得到提升。

最后，由于VS2010 SP1并没有实现全部的C++11语言特性，所以我们也并没有用上全部的特性，不过话说回来，已经被实现的特性已经相当有用了。

代码质量

在各种长期和短期压力之下写代码，当然代码质量是重中之重，尤其是对于C++代码，否则各种积累的技术债会越压越重。对于创新项目而言，代码基处于不停的演化当中，一开始的时候什么都不是，就是一个最简单的骨架，然后逐渐出现一点prototype的样子，随着不断的加进新的feature，再不断重构，抽取公共模块，形成concept和abstraction，isolate接口，拆分模块，最终prototype演变成product。关于代码质量的书很多，有一些写得很好，例如《The Art of Readable Code》，《Clean Code》或者《Implementation Patterns》。这里没有必要去重复这些书已经讲得非常好的技术，只说说我认为最重要的一些高层的指导性原则：

1. 持续重构：避免代码质量无限滑坡的办法就是持续重构。持续重构是The Boy Scout Rule的一个推论。离开一段代码的时候永远保持它比上次看到的时候更干净。关于重构的书够多的了，细节的这里就不说了，值得注意的是，虽然重构有一些通用的手法，但具体怎么重构很多时候是一个领域相关的问题，取决于你在写什么应用，有些时候，重构就是重设计。例如我们的代码基当中曾经有一个tricky的设计，因为相当tricky，导致在后来的一次代码改动中产生了一个很隐蔽的regression，这使得我们重新思考这个设计的实现，并最终决定换成另一个（很遗憾仍然还是tricky的）实现，后者虽然仍然tricky（总会有不得已必须tricky的地方），但是却有一个好处：即便以后代码改动的过程中又涉及到了这块代码并且又导致了regression，那么至少所导致的regression将不再会是隐蔽的，而是会很明显。
2. KISS：KISS是个被说烂了的原则，不过由于”Simple”这个词的定义很主观，所以KISS并不是一个很具有实践指导意义的原则。我认为下面两个原则要远远有用得多： 1) YAGNI：You Ain’t Gonna Need It。不做不必要的实现，例如不做不必要的泛化，你的目的是写应用，不是写通用库。尤其是在C++里面，要想写通用库往往会触及到这门语言最黑暗的部分，是个时间黑洞，而且由于语言的不完善往往会导致不完备的实现，出现使用上的陷阱。2) 代码不应该是没有明显的bug，而应该是明显没有bug：这是一条很具有指导意义的原则，你的代码是否一眼看上去就明白什么意思，就确定没有bug？例如Haskell著名的quicksort就属于明显没有bug。为了达到这个目的，你的代码需要满足很多要求：良好的命名（传达意图），良好的抽象，良好的结构，简单的实现，等等。最后，KISS原则不仅适用于实现层面，在设计上KISS则更加重要，因为设计是决策的第一环，一个设计可能需要三四百行代码，而另一个设计可能只需要三四十行代码，我们就曾遇到过这样的情况。一个糟糕的设计不仅制造大量的代码和bug（代码当然是越少越好，代码越少bug就越少），成为后期维护的负担，侵入式的设计还会增加模块间的粘合度，导致被这个设计拖累的代码像滚雪球一样越来越多，所以code review之前更重要的还是要做design review，前面决策做错了后面会越错越离谱。
3. 解耦原则：这个就不多说了，都说烂了。不过具体怎么解耦很多时候还是个领域相关的问题。虽然有些通用范式可循。
4. Best Practice Principle：对于C++开发来说尤其重要，因为在C++里面，同一件事情往往有很多不同的（但同样都有缺陷的）实现，而实现的成本往往还不低，所以C++社群多年以来一直在积淀所谓的Best Practices，其中的一个子集就是Idioms（惯用法），由于C++的学习曲线较为陡峭，闷头写一堆（有缺陷）的实现的成本很高，所以在一头扎进去之前先大概了解有哪些Idioms以及各自适用的场景就变得很有必要。站在别人的肩膀上好过自己掉坑里。

[我们在招人**]** 由于我们之前的star intern祁航同学离职去国外读书了，所以再次寻找实习生一枚，参与英库拼音输入法client端的开发，要求如下：

1. 扎实的win32系统底层知识。
2. 扎实的C++功底，对现代C++风格有一定的认识（了解C++11更好）。
3. 理解编写干净、可读、高效的代码的重要性。（最好读过clean code或implementation patterns）
4. 对新技术有热忱，有很强的学习能力；善于沟通，喜欢讨论。

有兴趣的请发简历至liuweipeng@outlook.com。此外，为了节省我们双方的时间，我希望你在发简历的同时回答以下两个问题：

1. 简要介绍一下你在大学里面学习技术的历程，例如看过那些书，经常上那些地方查资料，（如果有）参加过哪些开源项目，（如果有）写过哪些技术文章，等等。
2. 有针对性地对于上面的要求中提到的几点做简要的介绍：例如对win32有哪些了解，C++方面的技术储备，以及对高质量代码的认识，等等。

Tags: 编程

About 刘未鹏