C++异常机制的实现方式和开销分析

白杨

http://baiy.cn

在我几年前开始写《C++编码规范与指导》一文时，就已经规划着要加入这样一篇讨论 C++ 异常机制的文章了。没想到时隔几年以后才有机会把这个尾巴补完 :-)。

还是那句开场白："在恰当的场合使用恰当的特性" 对每个称职的 C++ 程序员来说都是一个基本标准。想要做到这点，就必须要了解语言中每个特性的实现方式及其时空开销。异常处理由于涉及大量底层内容，向来是 C++ 各种高级机制中较难理解和透彻掌握的部分。本文将在尽量少引入底层细节的前提下，讨论 C++ 中这一崭新特性，并分析其实现开销：

- 相关文档：
  - C++编码规范与指导
  - RTTI、虚函数和虚基类的实现方式、开销分析和使用指导
  - 多处理器环境和线程同步的高级话题
  - C++0x（C++11）新特性点评

关于线程

进程和线程的概念相信各位看官早已耳熟能详。在这里，我只想带大家回忆几点重要概念：

1. 一个进程中可以同时包含多个线程。

2. 我们通常认为线程是操作系统可识别的最小并发执行和调度单位（不要跟俺说还有 Green Thread 或者 Fiber，OS Kernel 不认识也不参与这些物件的调度）。

3. 同一进程中的多个线程共享代码段（代码和常量）、数据段（静态和全局变量）和扩展段（堆存储），但是每个线程有自己的栈段 。栈段又叫运行时栈，用来存放所有局部变量和临时变量（参数、返回值、临时构造的变量等）。这一条对下文中的某些概念来说是非常重要的 。但是请注意，这里提到的各个“段”都是逻辑上的说法，在物理上某些硬件架构或者操作系统可能不使用段式存储。不过没关系，编译器会保证这些逻辑概念和假设的前提条件对每个 C/C++ 程序员来说始终是成立的。

4. 由于共享了除栈以外的所有内存地址段，线程不可以有自己的“静态”或“全局”变量，为了弥补这一缺憾，操作系统通常会提供一种称为 **TLS**（Thread Local Storage，即：“线程本地存储”）的机制。通过该机制可以实现类似的功能。TLS 通常是线程控制块（TCB）中的某个指针所指向的一个指针数组，数组中的每个元素称为一个槽（Slot），每个槽中的指针由使用者定义，可以指向任意位置（但通常是指向堆存储中的某个偏移）。

函数的调用和返回

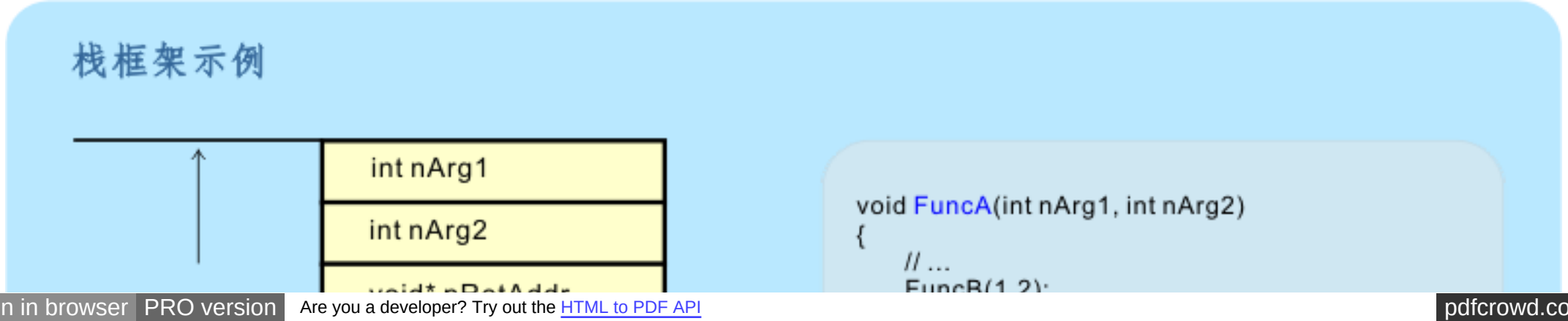
接着我们来回顾下一个预备知识：编译器如何实现函数的调用和返回。一般来说，编译器会为当前调用栈里的每个函数建立一个栈框架（Stack Frame）。“栈框架”担负着以下重要任务：

1. 传递参数：通常，函数的调用参数总是在这个函数栈框架的最顶端。
2. 传递返回地址：告诉被调用者的 return 语句应该 return 到哪里去，通常指向该函数调用的下一条语句（代码段中的偏移）。
3. 存放调用者的当前栈指针：便于清理被调用者的所有局部变量、并恢复调用者的现场。
4. 存放当前函数内的所有局部变量：记得吗？刚才说过所有局部和临时变量都是存储在栈上的。

最后再复习一点：栈是一种“后进先出”（LIFO）的数据结构，不过实际上大部分栈的实现都支持随机访问。

下面我们来看个具体例子：

假设有 FuncA、FuncB 和 FuncC 三个函数，每个函数均接收两个整形值作为其参数。在某线程上的某一时间段内，FuncA 调用了 FuncB，而 FuncB 又调用了 FuncC。则，它们的栈框架看起来应该像这样：



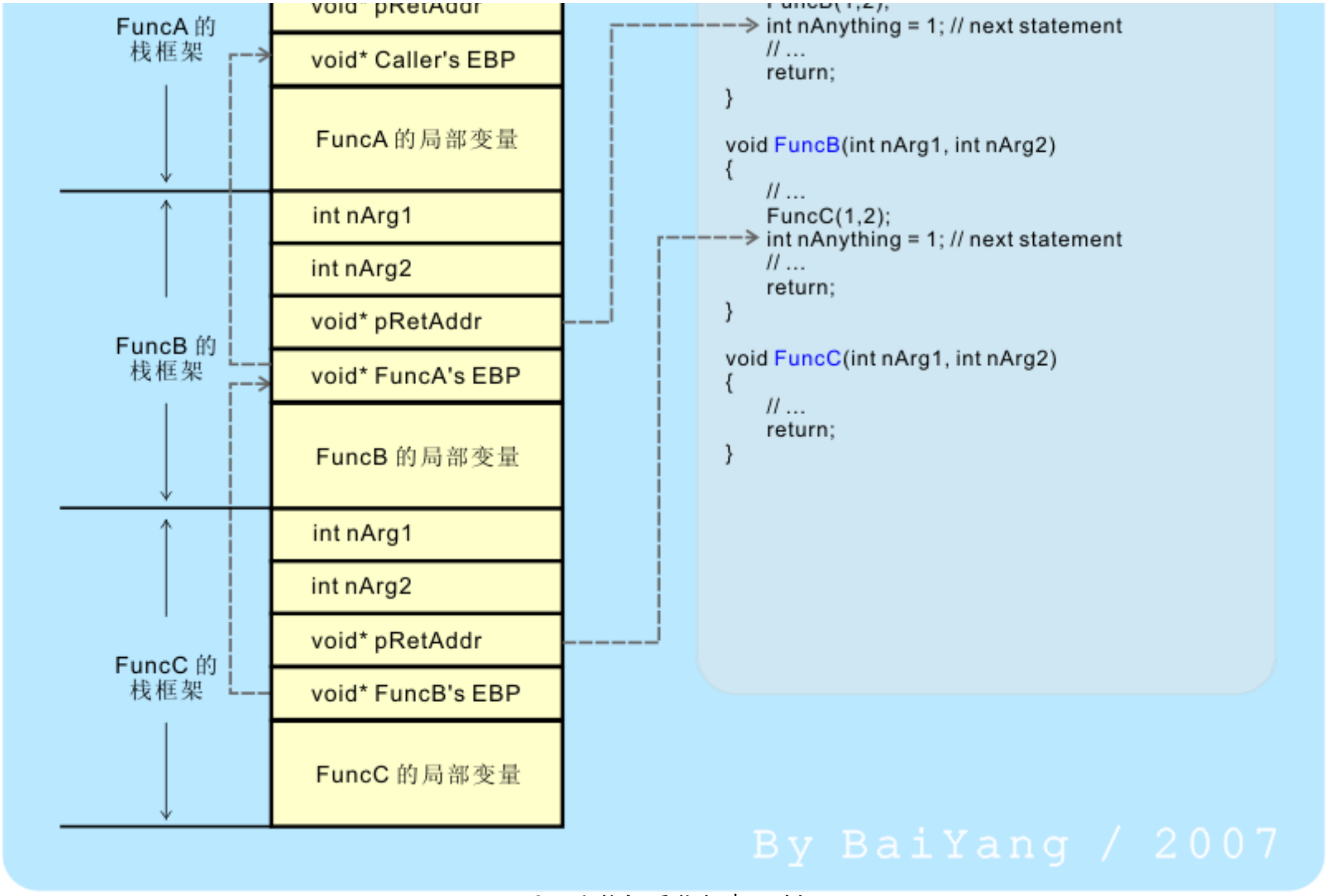


图1 函数调用栈框架示例

正如上图所示的那样，随着函数被逐级调用，编译器会为每一个函数建立自己的栈框架，栈空间逐渐消耗。随着函数的逐级返回，该函数的栈框架也将被逐级销毁，栈空间得以逐步释放。顺便说一句，递归函数的嵌套调用深度通常也是取决于运行时栈空间的剩余尺寸。

这里顺便解释另一个术语：调用约定（calling convention）。调用约定通常指：调用者将参数压入栈中（或放入寄存器中）的顺序，以及返回时由谁（调用者还是被调用者）来清理这些参数等细节规程方面的约定。

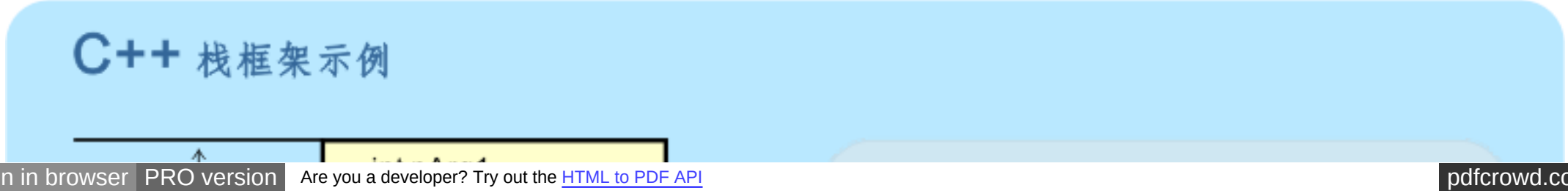
最后再说一句，这里所展示的函数调用乃是最“经典”的方式。实际情况是：在开启了优化选项后，编译器可能不会为一个内联甚至非内联的函数生成栈框架，编译器可能使用很多优化技术消除这个构造。不过对于一个 C/C++ 程序员来说，达到这样的理解程度通常就足够了。

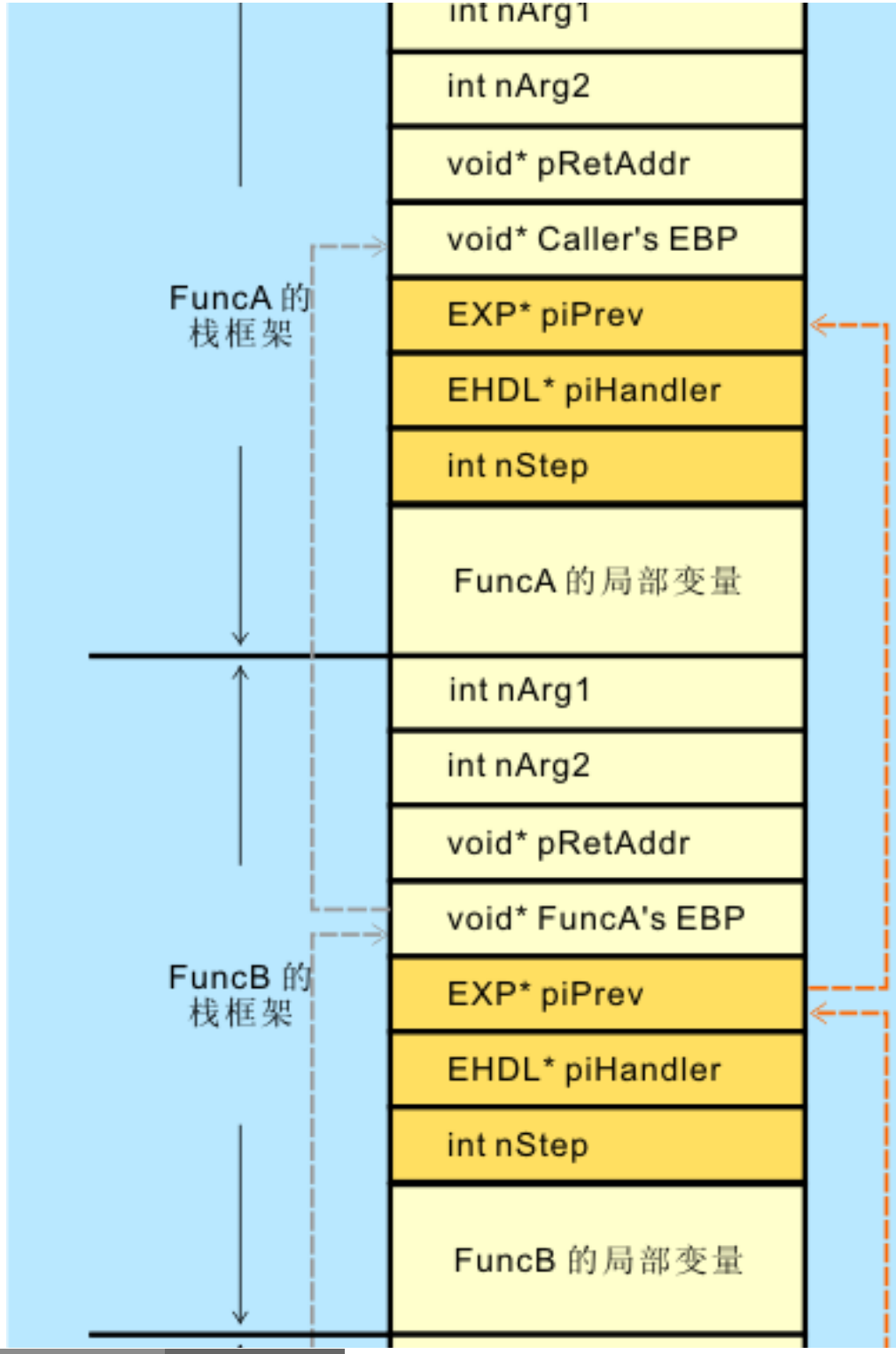
C++ 函数的调用和返回

首先澄清一点，这里说的 “C++ 函数”是指：

1. 该函数可能会直接或间接地抛出一个异常：即该函数的定义存放在一个 C++ 编译（而不是传统 C）单元内，并且该函数没有使用“throw()”异常过滤器。
2. 或者该函数的定义内使用了 try 块。

以上两者满足其一即可。为了能够成功地捕获异常和正确地完成栈回退（stack unwind），编译器必须要引入一些额外的数据结构和相应的处理机制。我们首先来看看引入了异常处理机制的栈框架大概是什么样子：






```
struct UNWINDTBL
{
    int nNextIdx;
    void (* pfnDestroyer)(void* this);
    void* pObj;
}

struct CATCHBLOCK
{
    // ...
    type_info* piType;
    void* pCatchBlockEntry;
}

struct TRYBLOCK
{
    // ...
    int nBeginStep;
    int nEndStep;
    CATCHBLOCK tblCatchBlocks[];
}

struct EHDL
{
    // ...
    UNWINDTBL tblUnwind[];
    TRYBLOCK tblTryBlocks[];
    // ...
}

struct EXP
{
    EXP*    piPrev;
    EHDL*   piHandler
    int     nStep;
}
```

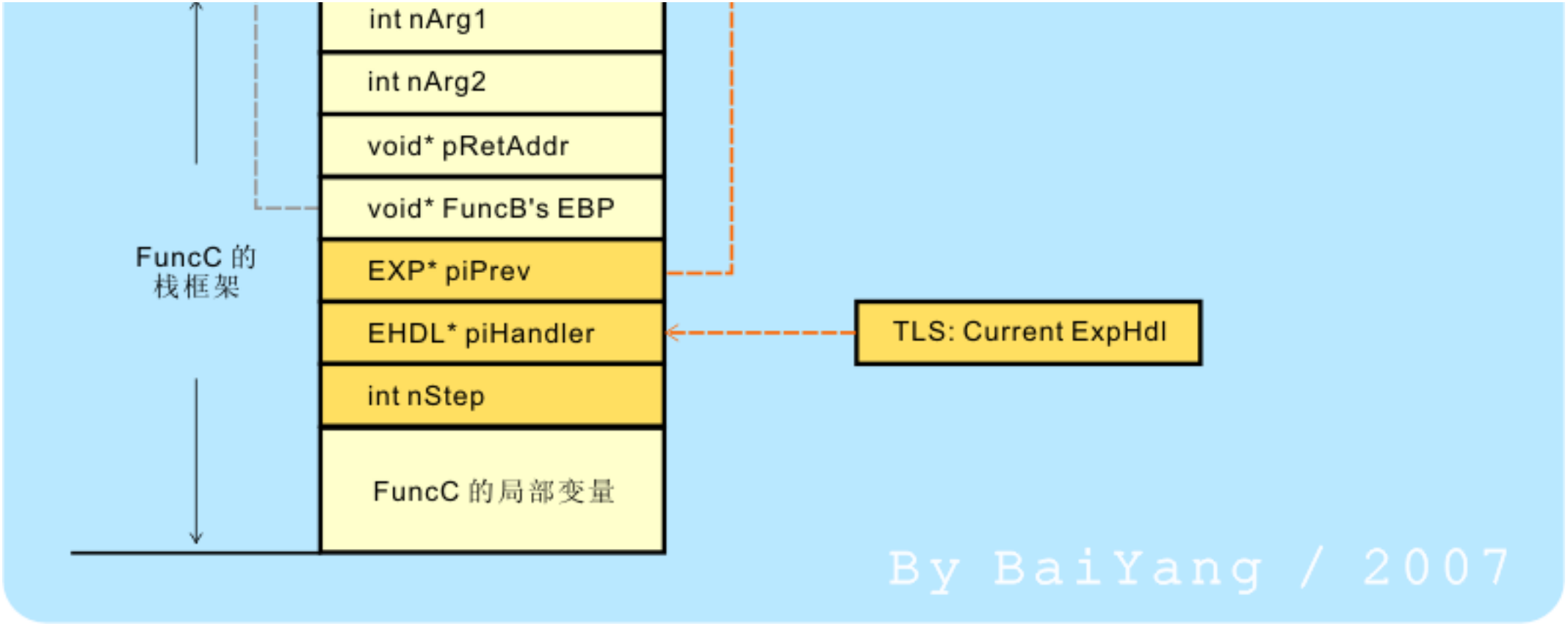


图2 C++函数调用栈框架示例

由图2可见，在每个 C++ 函数的栈框架中都多了一些东西。仔细观察的话，你会发现，多出来的东西正好是一个 EXP 类型的结构体。进一步分析就会发现，这是一个典型的单向链表式结构：

- piPrev 成员指向链表的上一个节点，它主要用于在函数调用栈中逐级向上寻找匹配的 catch 块，并完成栈回退工作。
- piHandler 成员指向完成异常捕获和栈回退所必须的数据结构（主要是两张记载着关键数据的表：“try”块表：tblTryBlocks 及“栈回退表”：tblUnwind）。
- nStep 成员用来定位 try 块，以及在栈回退表中寻找正确的入口。

需要说明的是：编译器会为每一个**“C++ 函数”**定义一个 **EHDL** 结构，不过只会为包含了**“try”**块的函数定

义 **tblTryBlocks** 成员。此外，异常处理器还会为每个线程维护一个指向当前异常处理框架的指针。该指针指向异常处理器链表的链尾，通常存放在某个 TLS 槽或能起到类似作用的地方。

最后，请再看一遍图2，并至少对其中的数据结构留下一个大体印象。我们会在后面多个小节中详细讨论它们。

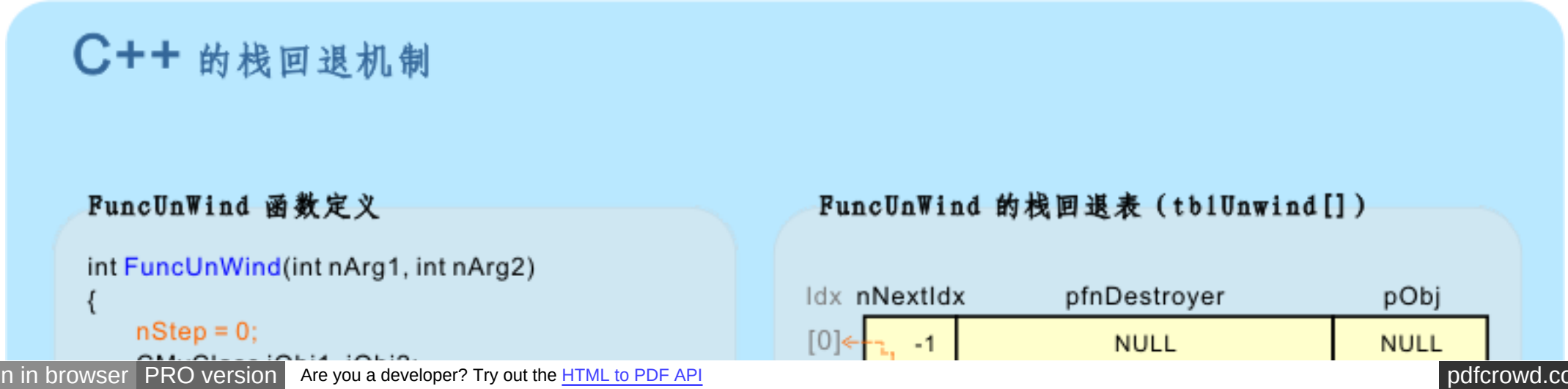
注意：为了简化起见，本文中描述的数据结构内，大多省略了一些与话题无关的成员。

栈回退（Stack Unwind）机制

“栈回退”是伴随异常处理机制引入 C++ 中的一个新概念，主要用来确保在异常被抛出、捕获并处理后，所有生命期已结束的对象都会被正确地析构，它们所占用的空间会被正确地回收。

受益于栈回退机制的引入，以及 C++ 类所支持的“资源申请即初始化”语意，使得我们终于能够彻底告别既不优雅也不安全的 setjmp/longjmp 调用，简便又安全地实现远程跳转了。我想这也是 C++ 异常处理机制在错误处理以外唯一一种合理的应用方式了。

下面我们就来具体看看编译器是如何实现栈回退机制的：



```
CMyClass iObj1, iObj2;
iObj1.CMyClass();  // constructor
nStep = 1;  // iObj1 constructed
iObj2.CMyClass();  // constructor
nStep = 2; // iObj2 constructed

if (0 == nArg1 || 1 == nArg2)
{
    CHisClass iObj3;
    iObj3.CHisClass(); // constructor
    nStep = 3; // iObj3 constructed

    nArg1 += nArg2
    // ...

    nStep = 2; // iObj3 will be destroyed
    iObj3.~CHisClass(); // destroy iObj3
}

CHerClass iObj4;
iObj4.CHerClass();  // constructor
nStep = 4;  // iObj4 constructed

// ...
if (2 == nArg1)
{
    throw myExp(ERR_VAL);
}

// ...

nStep = 2; // iObj4 will be destroyed
iObj4.~CHerClass(); // destroy iObj4
nStep = 1; // iObj2 will be destroyed
iObj2.~CMyClass(); // destroy iObj2
```

[1]	0	CMyClass::~CMyClass()	&iObj1
[2]	1	CMyClass::~CMyClass()	&iObj2
[3]	2	CHisClass::~CHisClass()	&iObj3
[4]	2	CHerClass::~CHerClass()	&iObj4

```
    nStep = 0; // iObj1 will be destroyed
    iObj1.~CMyClass(); // destroy iObj1
    return nArg1;
}
```

By BaiYang / 2007

图3 C++ 栈回退机制

图3中的“FuncUnWind”函数内，所有真实代码均以黑色和蓝色字体标示，编译器生成的代码则由灰色和橙色字体标明。此时，在图2里给出的 nStep 变量和 tblUnwind 成员作用就十分明显了。

nStep 变量用于跟踪函数内局部对象的构造、析构阶段。再配合编译器为每个函数生成的 tblUnwind 表，就可以完成退栈机制。表中的 pfnDestroyer 字段记录了对应阶段应当执行的析构操作（析构函数指针）；pObj 字段则记录了与之相对应的对象 this 指针偏移。将 pObj 所指的偏移值加上当前栈框架基址（EBP），就是要代入 pfnDestroyer 所指析构函数的 this 指针，这样即可完成对该对象的析构工作。而 nNextIdx 字段则指向下一个需要析构对象所在的行（下标）。

在发生异常时，异常处理器首先检查当前函数栈框架内的 nStep 值，并通过 piHandler 取得 tblUnwind[] 表。然后将 nStep 作为下标带入表中，执行该行定义的析构操作，然后转向由 nNextIdx 指向的下一行，直到 nNextIdx 为 -1 为止。在当前函数的栈回退工作结束后，异常处理器可沿当前函数栈框架内 piPrev 的值回溯到异常处理链中的上一节点重复上述操作，直到所有回退工作完成为止。

值得一提的是，nStep 的值完全在编译时决定，运行时仅需执行若干次简单的整形立即数赋值（通常是直接赋值给CPU里的某个寄存器）。此外，对于所有内部类型以及使用了默认构造、析构方法（并且它的所有成员和基类也使用了默认方法）的类型，其创建和销毁均不影响 nStep 的值。

注意：如果在栈回退的过程中，由于析构函数的调用而再次引发了异常（异常中的异常），则被认为是一次异常处理机制的严重失败。此时进程将被强行禁止。为防止出现这种情况，应在所有可能抛出异常的析构函数中使

用“std::uncaught_exception()”方法判断当前是否正在进行栈回退（即：存在一个未捕获或未完全处理完毕的异常）。如是，则应抑制异常的再次抛出。

异常捕获机制

一个异常被抛出时，就会立即引发 C++ 的异常捕获机制：

C++ 异常捕获机制

函数定义

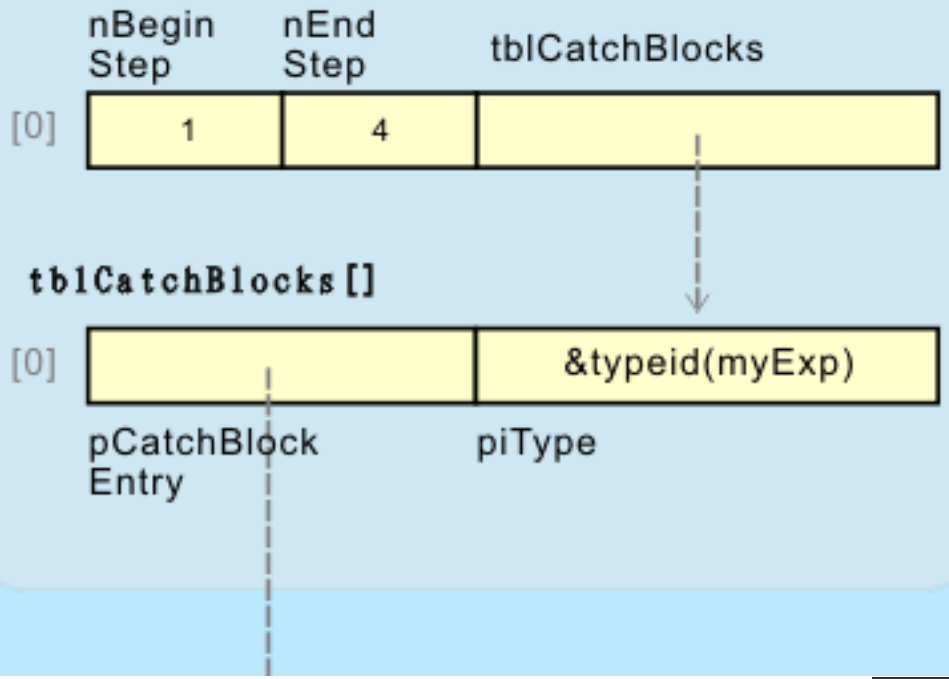
```
void FuncB(MyClass iObj1, MyClass iObj2);

void FuncA(int nArg1, int nArg2)
{
    nStep = 0;
    try
    {
        nStep = 1; // begin of try block
        CMyClass iObj1, iObj2;
        iObj1.CMyClass(); nStep = 2;
        iObj2.CMyClass(); nStep = 3;

        FuncB(iObj1, iObj2);

        nStep = 2; iObj2.~CMyClass();
```

FuncA 的 try 块表（tblTryBlocks[]）

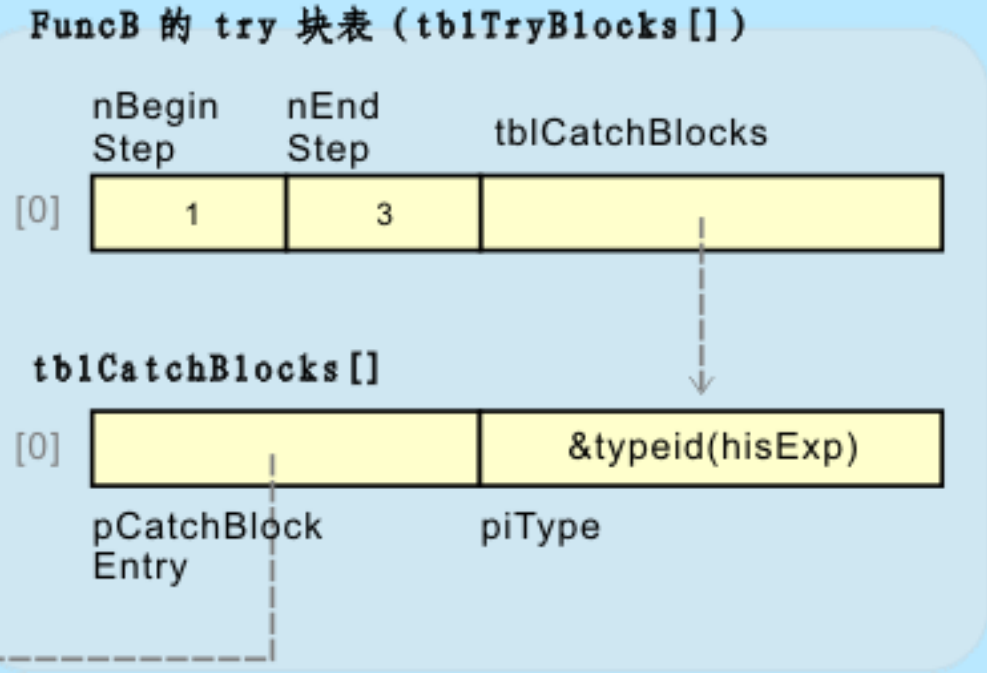


```
            nStep = 1; iObj1.~CMyClass();
        }
        nStep = 4; // end of try block
        catch (const myExp& err)
        {
            // ...
        }

        // ...
}

void FuncB(MyClass iObj1, MyClass iObj2)
{
        nStep = 0;
        try
        {
            nStep = 1; // begin of try block
            CHisClass iObj3
            iObj3.CHisClass(); nStep = 2;

            //...
            nStep = 1; iObj3.~CHisClass();
        }
        nStep = 3; // end of try block
        catch (const hisExp& err)
        {
            // ...
        }
```



By BaiYang / 2007

图4 C++ 异常捕获机制

在上一小节中，我们已经看到了 nStep 变量在跟踪对象构造、析构方面的作用。实际上 nStep 除了能够跟踪对象创建、销毁阶段以外，还能够标识当前执行点是否在 try 块中，以及（如果当前函数有多个 try 块的话）究竟在哪个 try 块中。这是通过在每一个 try 块的入口和出口各为 nStep 赋予一个唯一 ID 值，并确保 nStep 在对应 try 块内的变化恰在此范围之内来实现的。

在具体实现异常捕获时，首先，C++ 异常处理器检查发生异常的位置是否在当前函数的某个 try 块之内。这项工作可以通过将当前函数的 nStep 值依次在 piHandler 指向 tblTryBlocks[] 表的条目中进行范围为 [nBeginStep, nEndStep) 的比对来完成。

例如：若图4 中的 FuncB 在 nStep == 2 时发生了异常，则通过比对 FuncB 的 tblTryBlocks[] 表发现 $2 \in [1, 3)$，故该异常发生在 FuncB 内的第一个 try 块中。

其次，如果异常发生的位置在当前函数中的某个 try 块内，则尝试匹配该 tblTryBlocks[] 相应条目中的 tblCatchBlocks[] 表。tblCatchBlocks[] 表中记录了与指定 try 块配套出现的所有 catch 块相关信息，包括这个 catch 块所能捕获的异常类型及其起始地址等信息。

若找到了一个匹配的 catch 块，则复制当前异常对象到此 catch 块，然后跳转到其入口地址执行块内代码。

否则，则说明异常发生位置不在当前函数的 try 块内，或者这个 try 块中没有与当前异常相匹配的 catch 块，此时则沿着函数栈框架中 piPrev 所指地址（即：异常处理链中的上一个节点）逐级重复以上过程，直至找到一个匹配的 catch 块或到达异常处理链的首节点。对于后者，我们称为发生了未捕获的异常，对于 C++ 异常处理器而言，未捕获的异常是一个严重错误，将导致当前进程被强制结束。

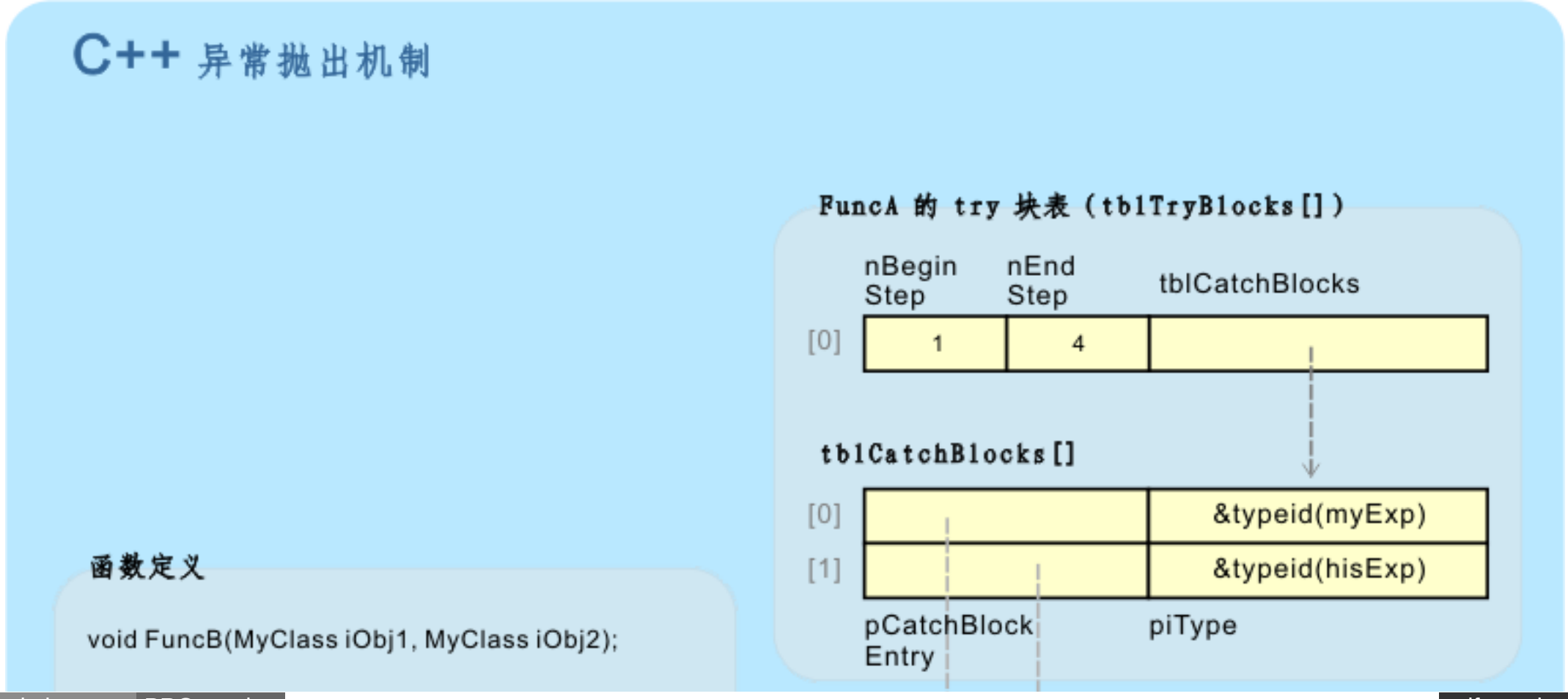
注意：虽然在图4示例中的 tblTryBlocks[] 只有一个条目，这个条目中的 tblCatchBlocks[] 也只有一行。但是在实际情况中，这两个表中都允许有多条记录。意即：一个函数中可以有多个 try 块，每个 try 块后均可跟随多个与之配套的 catch 块。

注意：按照标准意义上的理解，异常时的栈回退是伴随着异常捕获过程沿着异常处理链逐层向上进行的。但是有些编译器是在先完成异常捕获后再一次性进行栈回退的。无论具体实现使用了哪种方式，除非正在开发一个内存严格受限的嵌入式应用，通常我们按照标准语意来理解都不会产生什么问题。

备注：实际上 tblCatchBlocks 中还有一些较为关键但被故意省略的字段。比如指明该 catch 块异常对象复制方式（传值（拷贝构造）或传址（引用或指针））的字段，以及在何处存放被复制的异常对象（相对于入口地址的偏移位置）等信息。

异常的抛出

接下来讨论整个 C++ 异常处理机制中的最后一个环节，异常的抛出：



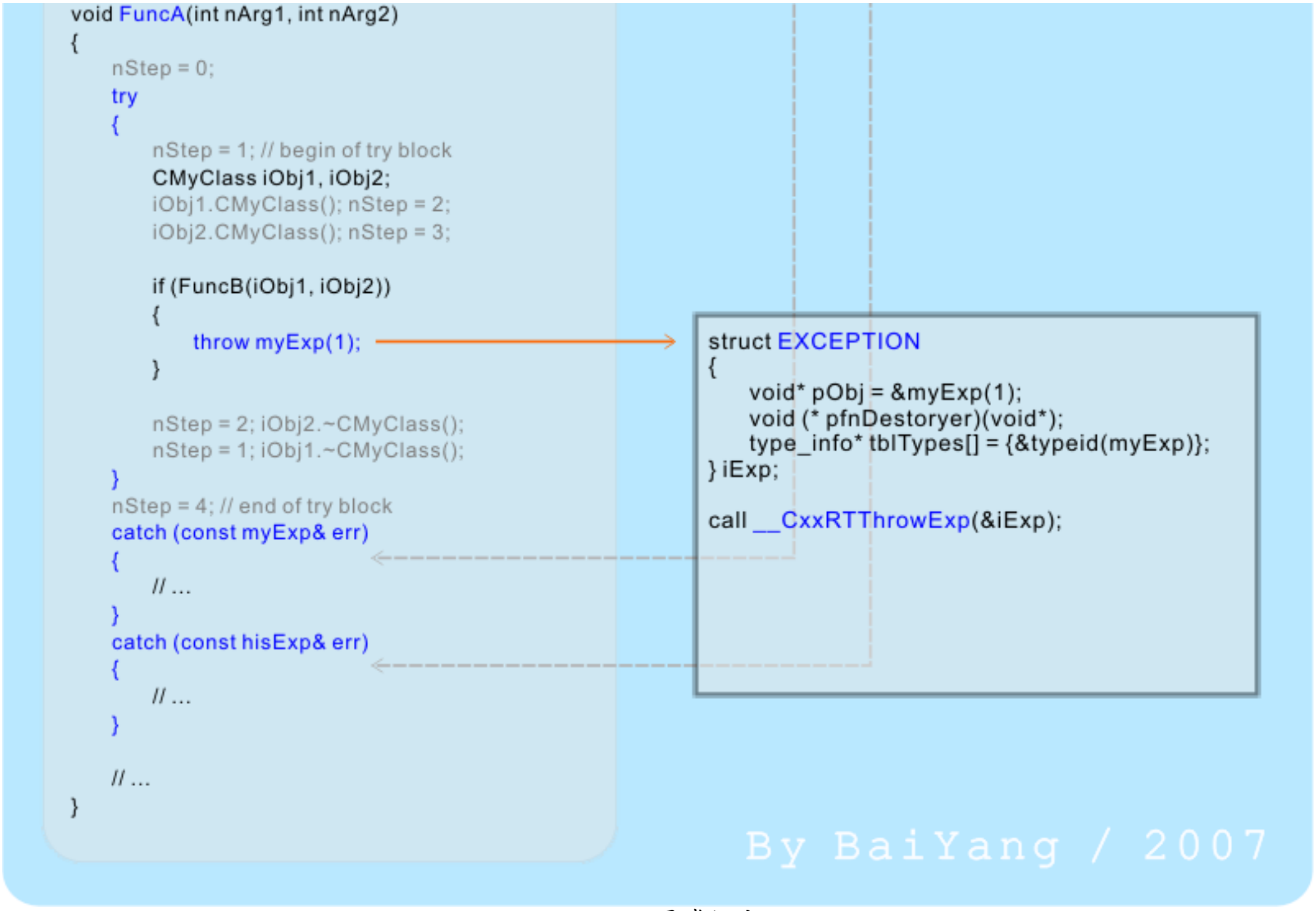


图5 C++ 异常抛出

在编译一段 C++ 代码时，编译器会将所有 throw 语句替换为其 C++ 运行时库中的某一指定函数，这里我们叫它 __CxxRTThrowExp（与本文提到的所有其它数据结构和属性名一样，在实际应用中它可以是任意名称）。该函数接收一个编译器认可的内部结构（我们叫它 EXCEPTION 结构）。这个结构中包含了待抛出异常对象的起始地址、用于销毁它的析构函数，以及它的 type_info 信息。对于没有启用 RTTI 机制（编译器禁用了 RTTI 机制或没有在类层次结构中使用虚表）的异常类层次结构，可能还要包含其所有基类的 type_info 信息，以便与相应的 catch 块进行匹配。

在图5中的深灰色框图内，我们使用 C++ 伪代码展示了函数 FuncA 中的 “throw myExp(1);” 语句将被编译器最终翻译成的样子。实际上在多数情况下，__CxxRTThrowExp 函数即我们前面曾多次提到的“异常处理器”，异常捕获和栈回退等各项重要工作都由它来完成。

__CxxRTThrowExp 首先接收（并保存）EXCEPTION 对象；然后从 TLS：Current ExpHdl 处找到与当前函数对应的 piHandler、nStep 等异常处理相关数据；并按照前文所述的机制完成异常捕获和栈回退。由此完成了包括“抛出”->“捕获”->“回退”等步骤的整套异常处理机制。

Windows 中的结构化异常处理

Microsoft Windows 带有一种名为“结构化异常处理”的机制，非常著名的“内存访问违例”出错对话框就是该机制的一种体现。Windows 结构化异常处理与前文讨论的 C++ 异常处理机制有惊人的相似之处，同样使用类似的链式结构实现。对于 Windows 下的应用程序，只需使用 SetUnhandledExceptionFilter API 注册异常处理器；用 FS:[0] 替代前文所述的 TLS: Current ExpHdl 等很少的改动，即可将此两种错误处理机制合而为一。这样做的优势十分明显：

- 由于可直接借助操作系统提供的机制，所以简化了 C++ 异常处理器的实现。
- 使“catch (...)” 块得以捕获操作系统产生的异常（如：“内存访问违例”等等）。
- 使操作系统的异常处理机制能够捕获所有 C++ 异常。

实际上，大多数 Windows 下的 C++ 编译器的异常机制均使用这种方式实现。

异常处理机制的开销分析

至此，我们已完整地阐述了整套 C++ 异常处理机制的实现原理。我在本文的开头曾提到，作为一名 C++ 程序员，了解其某一特性的实现原理主要是为了避免错误地使用该特性。要达到这个目的，还要在了解实现原理的基础上进行一些额外的开销分析工作：

特性	时间开销	空间开销
EHDL	无运行时开销	每“C++函数”一个 EHDL 对象，其中的 tblTryBlocks[] 成员仅在函数中包含至少一个 try 块时使用。典型情况下小于 64 字节。
C++栈框架	极高的 O(1) 效率，每次调用时进行3次额外的整形赋值和一次 TLS 访问。	每调用两个指针和一个整形开销。典型情况下小于 16 字节。
step 跟踪	极高的 O(1) 效率每次进出 try 块或对象构造/析构一次整形立即数赋值。	无（已记入 C++ 栈框架中的相应项目）。

异常的抛出、捕获和栈回退	异常的抛出是一次 O(1) 级操作。在单个函数中进行捕获和栈回退也均为 O(1) 操作。 但异常捕获的总体成本为 O(m)，其中 m 等于当前函数调用栈中，从抛出异常的位置到达匹配 catch 块之间所经过的函数调用中，包含 try 块（即：定义了有效 tblTryBlocks[]）的函数个数。 栈回退的总成本为 O(n)，其中 n 等于当前函数调用栈中，从抛出异常的位置到达匹配 catch 块之间所经过的函数调用数。	在异常处理结束前，需保存异常对象及其析构函数指针和相应的 type_info 信息。 具体根据对象尺寸、编译器选项（是否开启 RTTI）及异常捕获器的参数传递方式（传值或传址）等因素有较大变化。典型情况下小于 256 字节。

可以看出，在没有抛出异常时，C++ 的异常处理机制是十分有效的。在有异常被抛出后，可能会依当前函数调用栈的情形进行若干次整形比较（try块表匹配）操作，但这通常不会超过几十次。对于大多数 15 年前的 CPU 来说，整形比较也只需 1 时钟周期，所以异常捕获的效率还是很高的。栈回退的效率则与 return 语句基本相当。

考虑到即使是传统的函数调用、错误处理和逐级返回机制也不是没有代价的。这些开销在绝大多数情形下仍可以接受。空间开销方面，每“C++ 函数”一个 EHDL 结构体的引入在某些极端情形下会明显增加目标文件尺寸和内存开销。但是典型情况下，它们的影响并不大，但也没有小到可以完全忽略的程度。如果正在为一个资源严格受限的环境开发应用程序，你可能需要考虑关闭异常处理和 RTTI 机制以节约存储空间。

以上讨论的是一种典型的异常机制的实现方式，各具体编译器厂商可能有自己的优化和改进方案，但总体的出入不会很大。

小节

异常处理是 C++ 中十分有用的崭新特性之一。在绝大多数情况下，它们都有着优异的表现和令人满意的时空效率。异常处理本质上是另一种返回机制。但无论从软件工程、模块设计、编码习惯还是时空效率等角度来说，除了在有充分文档说明的前提下，偶尔可用来替代替代传统的 setjmp/longjmp 功能外，应保证只将其用于程序的错误处理机制中。

此外，由于长跳转的使用既易于出错，又难于理解和维护。在编码过程中也应当尽量避免使用。关于异常的一般性使用说明，请参考：代码风格与版式：异常。