# Pin编译使用手册

## 1 基本概念

Pin可以用于对可执行程序进行运行时动态插桩来分析程序的运行信息。它支持Linux，macOS，Windows等多个平台和IA-32，Intel(R) 64等指令架构。在官方文档中Pin可以理解一种“just in time”编译器，只不过编译的是常规的可执行文件，而不是字节码。

**Pintool**

一般来说，要通过插桩完成对目标程序的分析，需要了解插桩的位置以及插入的代码，也就是插桩部分和分析代码部分。这两部分在Pin中被集成到了一类可执行程序中，也就是Pintool。可以这么理解，Pin通过调用执行Pintool来完成对目标程序的分析，Pintool是用户根据分析需求所编写的调用Pin库里的函数来完成插桩分析需求的工具。

**Instrumentation**

Pin的插桩主要有四种模式（即粒度），分别是instruction instrumentation, trace instrumentation, routine instrumentation和image instrumentation。其中，最细粒度的就是instruction instrumentation即指令级插桩，它可以遍历程序执行的指令，很适合初学者使用。routine instrumentation可以理解为函数级插桩，可以对目标程序所调用的函数进行遍历，并在每一个routine中对instruction做更细粒度的遍历。image instrumentation可以理解为对整个程序映像做插桩，Pintool通过这种插桩，可以对section进行遍历。

## 2 Pin的使用

在source/tools/ManualExamples文件夹下，已经有了一些写好的Pintool，可以直接通过make进行编译使用，其中，要指定编译的对象和机器架构类型。例如，对目录下所有工具进行编译，所用机器指令架构为intel64，

$ cd source/tools/ManualExamples

$ make all TARGET**=**intel64

也可以指定某个工具，如inscount0:

$ cd source/tools/ManualExamples

$ make inscount0.test TARGET**=**intel64

之后，可以通过执行命令：

$./pin -t ./source/tools/ManualExamples/obj-intel64/xxxx.so -- target command

来进行动态插桩，并得到输出信息文件。这里的xxxx.so指代所要使用的Pintool，如inscount0.so，--代表之后要输入需要运行的目标程序（target）和执行命令（command）。我们这里尝试使用inscount0.so来进行一些分析。

ubuntu@ubuntu:~/moonlight/pin-3.11-97998-g7ecce2dac-gcc-linux$ ./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so -o ../inscount0.log -- ../programs/binutils-2.34/nor-ins/binutils/objdump -d ../queue/id\:001500\,src\:000536\,op\:havoc\,rep\:2

ubuntu@ubuntu:~/moonlight/pin-3.11-97998-g7ecce2dac-gcc-linux$ cat ../inscount0.log

Count 23657578

执行上述命令，可以使用inscount0.so来对objdump -d file的执行过程进行分析，并将结果输出到文件inscount0.log中。这里inscount0.so是统计执行指令数量的Pintool

## 3 PIN插桩分析基本流程

插桩(Instrumentation)就是在程序运行时在程序自身代码中插入一定分析代码的过程，在Manual提到从概念上来说插桩的流程包含两个部分：

* 确定需要插桩的代码的机制
* 插桩之后需要执行的分析代码

## 最基础的例子：程序计数

我们可以看一下inscount0.cpp这个程序的内容

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 | #include <iostream> #include <fstream> #include "pin.H"  ofstream OutFile;  // The running count of instructions is kept here // make it static to help the compiler optimize docount static UINT64 icount = 0;  // This function is called before every instruction is executed VOID docount() { icount++; }  // Pin calls this function every time a new instruction is encountered VOID Instruction(INS ins, VOID \*v) {  // Insert a call to docount before every instruction, no arguments are passed  INS\_InsertCall(ins, IPOINT\_BEFORE, (AFUNPTR)docount, IARG\_END); }  KNOB<string> KnobOutputFile(KNOB\_MODE\_WRITEONCE, "pintool",  "o", "inscount.out", "specify output file name");  // This function is called when the application exits VOID Fini(INT32 code, VOID \*v) {  // Write to a file since cout and cerr maybe closed by the application  OutFile.setf(ios::showbase);  OutFile << "Count " << icount << endl;  OutFile.close(); }  /\* ===================================================================== \*/ /\* Print Help Message \*/ /\* ===================================================================== \*/  INT32 Usage() {  cerr << "This tool counts the number of dynamic instructions executed" << endl;  cerr << endl << KNOB\_BASE::StringKnobSummary() << endl;  return -1; }  /\* ===================================================================== \*/ /\* Main \*/ /\* ===================================================================== \*/ /\* argc, argv are the entire command line: pin -t <toolname> -- ... \*/ /\* ===================================================================== \*/  int main(int argc, char \* argv[]) {  // Initialize pin  if (PIN\_Init(argc, argv)) return Usage();   OutFile.open(KnobOutputFile.Value().c\_str());   // Register Instruction to be called to instrument instructions  INS\_AddInstrumentFunction(Instruction, 0);   // Register Fini to be called when the application exits  PIN\_AddFiniFunction(Fini, 0);   // Start the program, never returns  PIN\_StartProgram();   return 0; } |

这个程序给出了一般pintool的基本框架，在main函数中首先调用PIN\_Init初始化，之后就可以使用INS\_AddInstrumentFunction注册一个插桩函数，在原始程序的每条指令被执行前，都会进入Instruction这个函数中，其第2个参数为一个额外传递给Instruction的参数，即对应VOID \*v这个参数，这里没有使用。而Instruction接受的第一个参数为INS结构，用来表示一条指令。

最后又注册了一个程序退出时的函数Fini，接着就可以使用PIN\_StartProgram启动程序了。

## 回调函数模式

可以看到，上面inscount0.cpp这个pintool插桩的对象就是所有指令。pintool在编写中将比较多的使用回调函数的机制，譬如在每条指令之前回调Instruction函数。而在Instruction函数的内部又使用INS\_InsertCall注册了一个函数docount，意为在指令执行之前插入一个对docount函数的调用。注意INS\_InsertCall是一个变参函数，前3个参数分别为指令，插入的时机（这里IPOINT\_BEFORE表示之前）以及函数指针（转为AFUNPTR类型），在之后就可以指定传给函数的参数，并以IARG\_END结尾，这里没有指定参数，直接调用。而docount的作用即是将一个全局变量加1，以达到统计执行指令条数的目的。

故此处插桩的分析代码即是将指令数加1.

# 指令(Instruction)级别的插桩

我们可以在inscount0的基础上，慢慢扩展出更加复杂的插桩分析程序

## 指定插桩的位置

最简单的情况是直接针对所有指令插桩，INS模块中提供了很多API来判断当前指令的类型

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | INS\_IsMemoryRead (INS ins) INS\_IsMemoryWrite (INS ins) INS\_IsLea (INS ins) INS\_IsNop (INS ins) INS\_IsBranch (INS ins) INS\_IsDirectBranch (INS ins) INS\_IsDirectCall (INS ins) INS\_IsDirectBranchOrCall (INS ins) INS\_IsBranchOrCall (INS ins) INS\_IsCall (INS ins) INS\_IsRet (INS ins) ... |

一般看到API的名字就可以明白其作用了，如果有不明白则可以去查API的手册，或者还有种更加直接、具体的方法

|  |  |
| --- | --- |
| 1 2 3 4 | if (INS\_Opcode(ins) == XED\_ICLASS\_MOV &&  INS\_IsMemoryRead(ins) &&  INS\_OperandIsReg(ins, 0) &&  INS\_OperandIsMemory(ins, 1)) |

上面的代码来自safecopy.cpp，直接通过Opcode来识别mov指令，并且是一条内存读指令，并且指令的第一个操作数是寄存器，并且指令的第二个操作数是内存。通过组合这些API就可以非常精确地筛选出想要插桩的指令了。

## 插桩分析代码

inscount0中的分析代码写的非常简略，再之后还有一个例子itrace

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 | // This function is called before every instruction is executed // and prints the IP VOID printip(VOID \*ip) { fprintf(trace, "%p\n", ip); }  // Pin calls this function every time a new instruction is encountered VOID Instruction(INS ins, VOID \*v) {  // Insert a call to printip before every instruction, and pass it the IP  INS\_InsertCall(ins, IPOINT\_BEFORE, (AFUNPTR)printip, IARG\_INST\_PTR, IARG\_END); } |

在这里传递给printip的是一个IARG\_INST\_PTR参数，实际对应的类型是VOID \*，指示了当前指令的位置，而printip则是把它输出出来，所以itrace的作用即是输出所有指令的地址。

实际来说[Instrumentation arguments](https://software.intel.com/sites/landingpage/pintool/docs/97438/Pin/html/group__INST__ARGS.html)中给出了很多可以传递给回调函数的参数，包括当前指令读取的有效内存地址、相关寄存器的值等等，能够对程序的运行状态有很全面的描述，便于回调函数的进一步分析。

# 程序运行状态监控 & 修改

## 寄存器

想要获得当前某个寄存器的值，可以传递...IARG\_REG\_VALUE, REG\_RAX...参数，实际对应的类型是ADDRINT，将寄存器当前的值传给回调函数。或者可以通过INS\_OperandReg函数首先提取出指令中的寄存器操作数，然后再用IARG\_REG\_VALUE传递给回调函数。

想要修改寄存器的值，可以传递...IARG\_REG\_REFERENCE, REG\_RAX...这种参数，实际对应的类型是PIN\_REGISTER \*指针，指向一个表示寄存器值的union类型，在64位中，可以使用reg->qword[0]来访问RAX，reg->dword[0]来访问EAX，以达到修改寄存器值的目的。

## 内存

关于内存数据的获取和写入，可以参考safecopy，其中使用到了PIN\_SafeCopy函数

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 | //======================================================= // Analysis routines //=======================================================  // Move from memory to register ADDRINT DoLoad(REG reg, ADDRINT \* addr) {  \*out << "Emulate loading from addr " << addr << " to " << REG\_StringShort(reg) << endl;  ADDRINT value;  PIN\_SafeCopy(&value, addr, sizeof(ADDRINT));  return value; }  //======================================================= // Instrumentation routines //=======================================================  VOID EmulateLoad(INS ins, VOID\* v) {  // Find the instructions that move a value from memory to a register  if (INS\_Opcode(ins) == XED\_ICLASS\_MOV &&  INS\_IsMemoryRead(ins) &&  INS\_OperandIsReg(ins, 0) &&  INS\_OperandIsMemory(ins, 1))  {  // op0 <- \*op1  INS\_InsertCall(ins,  IPOINT\_BEFORE,  AFUNPTR(DoLoad),  IARG\_UINT32,  REG(INS\_OperandReg(ins, 0)),  IARG\_MEMORYREAD\_EA,  IARG\_RETURN\_REGS,  INS\_OperandReg(ins, 0),  IARG\_END);   // Delete the instruction  INS\_Delete(ins);  } } |

safecopy实际模拟了mov指令内存读的过程，将寄存器和指令操作的内存地址传递给分析函数DoLoad，并在最后用IARG\_RETURN\_REGS指定将分析函数的返回值写入到指令的操作寄存器中，实际指令的语义没有改变。

而在DoLoad函数中，实际调用了PIN\_SafeCopy(&value, addr, sizeof(ADDRINT));将对应地址的内容模拟装载并返回。由此就可以看出在程序实际运行时pintool和原始程序位于同一地址空间，因而PIN\_SafeCopy既可以从内存中读取数据，亦可以写入数据。

# 更粗粒度的插桩

有时我们并不需要在指令级的插桩，pin也可以实现基于Basic Block，Routine或Image的插桩函数，以例子中的malloctrace来说

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 | ... VOID Image(IMG img, VOID \*v) {  // Instrument the malloc() and free() functions. Print the input argument  // of each malloc() or free(), and the return value of malloc().  //  // Find the malloc() function.  RTN mallocRtn = RTN\_FindByName(img, MALLOC);  if (RTN\_Valid(mallocRtn))  {  RTN\_Open(mallocRtn);   // Instrument malloc() to print the input argument value and the return value.  RTN\_InsertCall(mallocRtn, IPOINT\_BEFORE, (AFUNPTR)Arg1Before,  IARG\_ADDRINT, MALLOC,  IARG\_FUNCARG\_ENTRYPOINT\_VALUE, 0,  IARG\_END);  RTN\_InsertCall(mallocRtn, IPOINT\_AFTER, (AFUNPTR)MallocAfter,  IARG\_FUNCRET\_EXITPOINT\_VALUE, IARG\_END);   RTN\_Close(mallocRtn);  }   // Find the free() function.  RTN freeRtn = RTN\_FindByName(img, FREE);  if (RTN\_Valid(freeRtn))  {  RTN\_Open(freeRtn);  // Instrument free() to print the input argument value.  RTN\_InsertCall(freeRtn, IPOINT\_BEFORE, (AFUNPTR)Arg1Before,  IARG\_ADDRINT, FREE,  IARG\_FUNCARG\_ENTRYPOINT\_VALUE, 0,  IARG\_END);  RTN\_Close(freeRtn);  } } ... |

使用IMG\_AddInstrumentFunction来注册一个在Image载入时插桩的函数，随后在Image里面使用RTN\_FindByName来找到模块里的malloc和free两个符号，注意在pintool开头除了PIN\_Init之外还要用PIN\_InitSymbols来初始化symbol manager。在找到相应的函数之后，可以使用RTN\_InsertCall来插入分析代码Arg1Before，并将此时函数的参数传递给分析函数。最后这个pintool完成的作用就是追踪malloc/free的调用，并输出它们的参数与返回值。