OSDT2018

浅谈LLVM的异常处理

史宁宁

2018年12月8日

01 LLVM异常处理简介

02 Itanium ABI Zero-cost Exception Handling

03 Setjmp/Longjmp Exception Handling

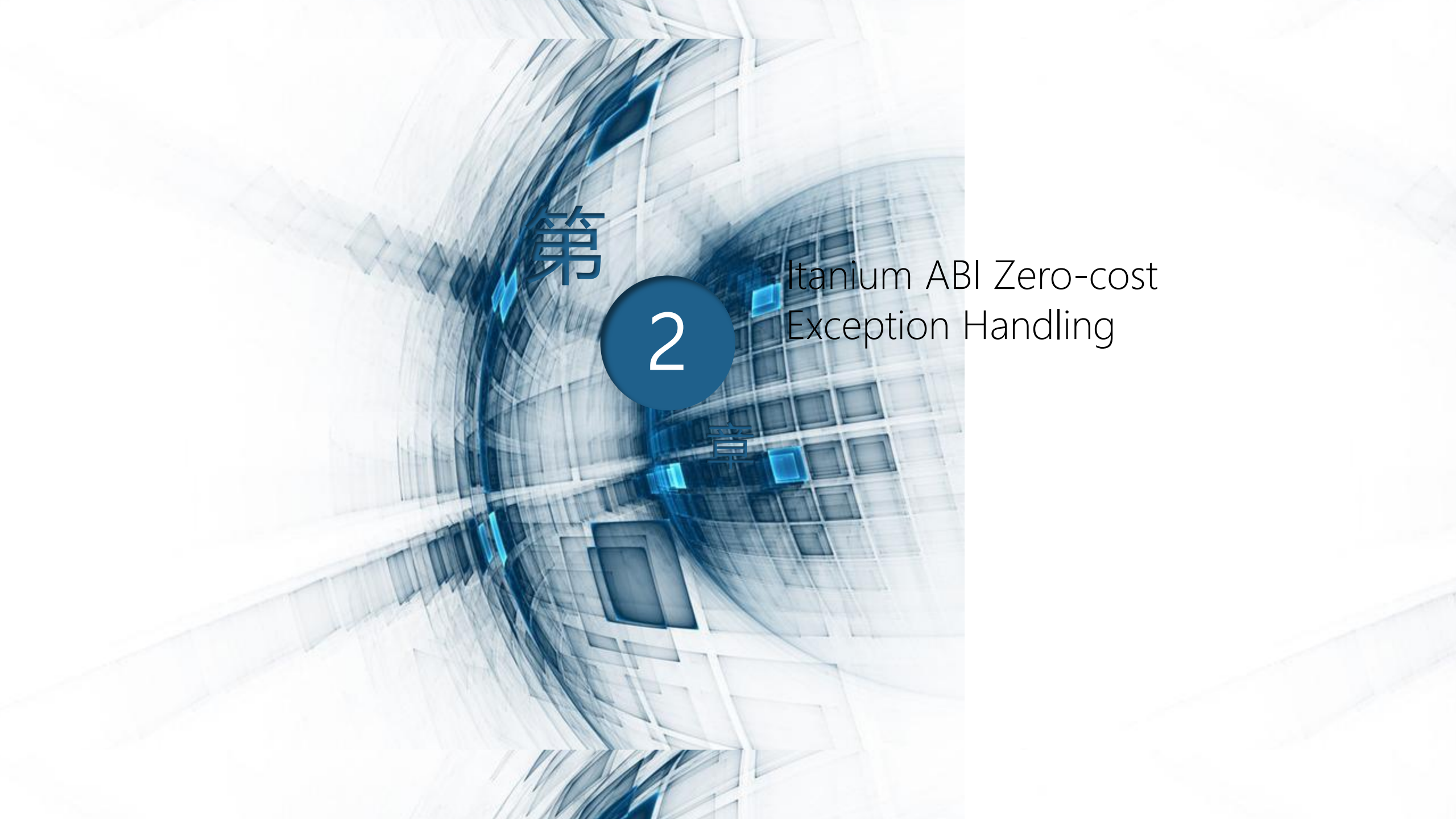
04 Windows Runtime Exception Handling

第1章 LLVM异常处理简介

简介

LLVM 异常处理

- Itanium ABI Zero-cost Exception Handling
- Setjmp/Longjmp Exception Handling
- Windows Runtime Exception Handling



Itanium ABI Zero-cost Exception Handling

Itanium C++ ABI

The Itanium C++ ABI is an ABI for C++.

As an ABI, it gives precise rules for implementing the language, ensuring that separately-compiled parts of a program can successfully interoperate.

Although it was initially developed for the Itanium architecture, it is not platform-specific and can be layered portably on top of an arbitrary C ABI.

Accordingly, it is used as the standard C++ ABI for many major operating systems on all major architectures, and is implemented in many major C++ compilers, including GCC and Clang.

Itanium C++ ABI: Exception Handling

Itanium C++ ABI : Exception Handling	libs
Level I: Base API This section defines the Unwind Library interface, expected to be provided by any Itanium psABI-compliant system. This is the interface on which the C++ ABI exception-handling facilities are built.	libunwind (LLVM) libgcc_s (GNU) libunwind (nongnu.org) libunwind (PathScale)
Level II: C++ ABI The second level of specification is the minimum required to allow interoperability in the sense described above.	libc++abi (LLVM) libsupc++ (GNU libcxxrt (PathScale)
Level III. Suggested Implementation The third level is a specification sufficient to allow all compliant C++ systems to share the relevant runtime implementation.	

libc++abi Specification

六个类别的函数
Memory management
Exception Handling
Guard objects
Vector construction and destruction
Handlers
Utilities

libc++abi常用的函数

void* __cxa_allocate_exception(size_t thrown_size) throw();

Effects: Allocates memory to hold the exception to be thrown. thrown_size is the size of the exception object. Can allocate additional memory to hold private data. If memory can not be allocated, call std::terminate().

Returns: A pointer to the memory allocated for the exception object.

void __cxa_throw(void* thrown_exception, struct std::type_info * tinfo, void (*dest)(void*));

libc++abi常用的函数

void* __cxa_begin_catch(void* exceptionObject) throw();

Effects:

Increment's the exception's handler count.
Places the exception on the stack of currently-caught exceptions if it is not already there, linking the exception to the previous top of the stack.
Decrements the uncaught_exception count.
If the initialization of the catch parameter is trivial (e,g., there is no formal catch parameter, or the parameter has no copy constructor), the calls to __cxa_get_exception_ptr() and __cxa_begin_catch() may be combined into a single call to __cxa_begin_catch().

When the personality routine encounters a termination condition, it will call __cxa_begin_catch() to mark the exception as handled and then call terminate(), which shall not return to its caller.

Returns: The adjusted pointer to the exception object.

libc++abi常用的函数

void __cxa_end_catch();

Effects: Locates the most recently caught exception and decrements its handler count. Removes the exception from the caughtÃ“exception stack, if the handler count goes to zero. Destroys the exception if the handler count goes to zero, and the exception was not re-thrown by throw. Collaboration between __cxa_rethrow() and __cxa_end_catch() is necessary to handle the last point. Though implementation-defined, one possibility is for __cxa_rethrow() to set a flag in the handlerCount member of the exception header to mark an exception being rethrown.

libc++abi常用的函数

void __cxa_rethrow();

Effects: Marks the exception object on top of the caughtExceptions stack (in an implementation-defined way) as being rethrown. If the caughtExceptions stack is empty, it calls terminate() (see [C++FDIS] [except.throw], 15.1.8). It then returns to the handler that called it, which must call __cxa_end_catch(), perform any necessary cleanup, and finally call _Unwind_Resume() to continue unwinding.

libc++abi常用的函数

```
_Unwind_Reason_Code __gxx_personality_v0 (int, _Unwind_Action,
                                          _Unwind_Exception_Class,
                                          struct _Unwind_Exception *,
                                          struct _Unwind_Context *);
```

Because different programming languages have different behaviors when handling exceptions, the exception handling ABI provides a mechanism for supplying personalities.

代码示例

```
class Ex1 {};
void throw_excepiton(int a, int b) {
  Ex1 ex1;
  if (a > b) {
    throw ex1;
  }
}
```

```
define void @_Z15throw_excepitonii(i32, i32) #0 {
 %3 = alloca i32, align 4
 %4 = alloca i32, align 4
.....

; <label>:9:                                       ; preds = %2
 %10 = call i8* @__cxa_allocate_exception(i64 1) #1
 %11 = bitcast i8* %10 to %class.Ex1*
 call void @__cxa_throw(i8* %10, i8* bitcast ({ i8*, i8* }*
@_ZTI3Ex1 to i8*), i8* null) #2
 unreachable
.....
```

代码示例

```
int test_try_catch() {
  try {
    throw_excepiton(2, 1);
  }
  catch(...) {
    return 1;
  }
  return 0;
}
```

```
define i32 @_Z14test_try_catchv() #0 personality i8* bitcast (i32 (...)*
@__gxx_personality_v0 to i8*) {
.....

 %6 = landingpad { i8*, i32 }          catch i8* null
.....

 %11 = call i8* @__cxa_begin_catch(i8* %10) #1
 store i32 1, i32* %1, align 4
 call void @__cxa_end_catch()
.....
```

第3章

Setjmp/Longjmp Exception Handling

SJLJ Exception Handling

- Setjmp/Longjmp (SJLJ) based exception handling uses LLVM intrinsics llvm.eh.sjlj.setjmp and llvm.eh.sjlj.longjmp to handle control flow for exception handling.
- In contrast to DWARF exception handling, which encodes exception regions and frame information in out-of-line tables, SJLJ exception handling builds and removes the unwind frame context at runtime.
- The llvm.eh.sjlj intrinsics are used internally within LLVM's backend. Uses of them are generated by the backend's SjLjEHPrepare pass.(llvm/lib/CodeGen/SjLjEHPrepare.cpp)

SJLJ Intrinsics

i32 @llvm.eh.sjlj.setjmp(i8* %setjmp_buf)

void @llvm.eh.sjlj.longjmp(i8* %setjmp_buf)

i8* @llvm.eh.sjlj.lsda()

void @llvm.eh.sjlj.callsite(i32 %call_site_num)

llvm.eh.sjlj.setjmp

i32 @llvm.eh.sjlj.setjmp(i8* %setjmp_buf)

The single parameter is a pointer to a five word buffer in which the calling context is saved.

The front end places the frame pointer in the first word, and the target implementation of this intrinsic should place the destination address for a llvm.eh.sjlj.longjmp in the second word. The following three words are available for use in a target-specific manner.

SjLj示例

```
typedef void *jmp_buf;
jmp_buf buf;

void do_jump(void) {
  __builtin_longjmp(buf, 1);
}
```

```
; Function Attrs: noinline nounwind optnone
define void @do_jump() #0 {
  %1 = load i8*, i8** @buf, align 8
  %2 = bitcast i8* %1 to i8**
  %3 = bitcast i8** %2 to i8*
  call void @llvm.eh.sjlj.longjmp(i8* %3)
  unreachable                                       ; No predecessors!
  ret void
}


; Function Attrs: noreturn nounwinddeclare void
@llvm.eh.sjlj.longjmp(i8*) #1
```

SjLj示例

```
void f(void);

void do_setjmp(void) {
  if (!__builtin_setjmp(buf))
    f();
}
```

```
; Function Attrs: noinline nounwind optnone
define void @do_setjmp() #0 {
  %1 = load i8*, i8** @buf, align 8
  %2 = bitcast i8* %1 to i8**
  %3 = call i8* @llvm.frameaddress(i32 0)
  store i8* %3, i8** %2, align 8
  %4 = call i8* @llvm.stacksave()
  %5 = getelementptr inbounds i8*, i8** %2, i64 2
  store i8* %4, i8** %5, align 8
  %6 = bitcast i8** %2 to i8*
  %7 = call i32 @llvm.eh.sjlj.setjmp(i8* %6)
  %tobool = icmp ne i32 %7, 0
  br i1 %tobool, label %9, label %8

; <label>:8:                                      ; preds = %0
  call void @f()
  br label %9; <label>:9:                                      ; preds = %8, %0
  ret void
}
```

SjLj VS Architectures

SUPPORT	UNSUPPORT
i386-unknown-unknown	aarch64-unknown-unknow
x86_64-unknown-unknown	mips-unknown-unknown
x86_64-windows	mips64-unknown-unknown
powerpc-unknown-unknown	
powerpc64-unknown-unknown	
sparc-eabi-unknown	

相关代码

SemaChecking.cpp

llvm/tools/clang/lib/Sema/
SemaChecking.cpp

CGBuiltin.cpp

llvm/tools/clang/lib/CodeGen/
CGBuiltin.cpp

SelectionDAGBuilder.cpp

llvm/lib/CodeGen/SelectionDAG/
SelectionDAGBuilder.cpp

SjLjEHPrepare.cpp

llvm/lib/CodeGen/SjLjEHPrepare.cpp

第4章 Windows Runtime Exception Handling

VS Itanium C++ abi

1.Interacting with exceptions on Windows is significantly more complicated than on Itanium C++ ABI platforms. The fundamental difference between the two models is that Itanium EH is designed around the idea of "successive unwinding," while Windows EH is not.

2.The Windows EH model does not use these successive register context resets.

3.LLVM supports handling exceptions produced by the Windows runtime, but it requires a very different intermediate representation. It is not based on the "landingpad" instruction like the other two models.

Windows Runtime Exception Handling

1.In the case of C++ exceptions, the exception object is allocated in stack memory and its address is passed to __CxxThrowException.

2. Each frame on the stack has an assigned EH personality routine, which decides what actions to take to handle the exception.

3. There are a few major personalities for C and C++ code: the C++ personality (__CxxFrameHandler3) and the SEH personalities (_except_handler3, _except_handler4, and __C_specific_handler).

4. General purpose structured exceptions (SEH) are more analogous to Linux signals, and they are dispatched by userspace DLLs provided with Windows.

Windows Runtime Exception Handling

1.In other words, the successive unwinding approach is incompatible with Visual C++ exceptions and general purpose Windows exception handling. Because the C++ exception object lives in stack memory, LLVM cannot provide a custom personality function that uses landingpads.

2.Similarly, SEH does not provide any mechanism to rethrow an exception or continue unwinding.

3.Therefore, LLVM must use the IR constructs to implement compatible exception handling.

示例代码

```
void f() {
  try {
    throw;
  } catch (...) {
    try {
      throw;
    } catch (...) {
    }
  }
}
```

```
define void @f() #0 personality i8* bitcast (i32 (...)*
@__CxxFrameHandler3 to i8*) {
entry:
  invoke void @_CxxThrowException(i8* null, %eh.ThrowInfo* null)
#1
          to label %unreachable unwind label %catch.dispatch

catch.dispatch:                                   ; preds = %entry
  %0 = catchswitch within none [label %catch] unwind to caller

catch:                                            ; preds = %catch.dispatch
  %1 = catchpad within %0 [i8* null, i32 64, i8* null]
  invoke void @_CxxThrowException(i8* null, %eh.ThrowInfo* null)
#1
          to label %unreachable unwind label %catch.dispatch2

catch.dispatch2:                                  ; preds = %catch
  %2 = catchswitch within %1 [label %catch3] unwind to caller
.....
```

参考文献

1、LLVM**异常处理官方文档**：http://llvm.org/docs/ExceptionHandling.html
2、LLVM**异常处理官方文档中文翻译**： https://blog.csdn.net/wuhui_gdnt/article/details/51859729
3、《LLVM Cookbook》**中文版** P265-270
4、libc++abi Specification： http://libcxxabi.llvm.org/spec.html
5、Itanium C++ ABI: Exception Handling： http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
6、Linux Standard Base Core Specification 3.0RC1（Chapter 8. Exception Frames）：
http://refspecs.linuxfoundation.org/LSB_3.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html
7、DWARF 4 Standard：http://llvm.org/docs/ExceptionHandling.html
8、Exception Handling Tables：http://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf
9、Unwind lib: https://clang.llvm.org/docs/Toolchain.html

OSDT2018

Thanks!

2018年12月8日