# 第21章 noexcept关键字

《现代C++语言核心特性解析》 谢丙堃

# 使用noexcept代替throw

- 不带参数语法

```
struct X {
    int f() const noexcept
    {
        return 58;
    }
    void g() noexcept {}
};
int foo() noexcept
{
    return 42;
}
```

- 带参数语法

```
template <class T>
T copy(const T &o)
noexcept(std::is_fundamental<T>::value) {
  ...
}
```

# 使用noexcept代替throw

- 作为运算符的情况

```
int foo() noexcept {
    return 42;
}
int foo1() {
    return 42;
}
int foo2() throw() {
    return 42;
}
int main() {
    std::cout << std::boolalpha;
    std::cout << "noexcept(foo())  = " << noexcept(foo()) << std::endl;
    std::cout << "noexcept(foo1()) = " << noexcept(foo1()) << std::endl;
    std::cout << "noexcept(foo2()) = " << noexcept(foo2()) << std::endl;
}
```

# 用noexcept来解决移动构造问题

- 阻止会抛出异常的移动

```
template<class T>
void swap(T& a, T& b)
noexcept(noexcept(T(std::move(a))) &&
noexcept(a.operator=(std::move(b))))
{
    static_assert(noexcept(T(std::move(a)))
        && noexcept(a.operator=(std::move(b))));
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
```

# 用noexcept来解决移动构造问题

- 让编译器自己选择更适合的版本

```
template<typename T>
void swap_impl(T& a, T& b, std::integral_constant<bool, true>) noexcept {
    T tmp(std::move(a));
    a = std::move(b);
    b = std::move(tmp);
}
template<typename T>
void swap_impl(T& a, T& b, std::integral_constant<bool, false>) {
    T tmp(a);
    a = b;
    b = tmp;
}

template<typename T>
void swap(T& a, T& b)
noexcept(noexcept(swap_impl(a, b,
    std::integral_constant<bool, noexcept(T(std::move(a)))
    && noexcept(a.operator=(std::move(b)))>())))
{
    swap_impl(a, b, std::integral_constant<bool, noexcept(T(std::move(a)))
        && noexcept(a.operator=(std::move(b)))>());
}
```

# noexcept 和 throw()

- C++11： 相同的结果，不同的机制
- C++17： 相同的结果和机制
- C++20： throw被移除

# 默认使用noexcept的函数

- 类型默认构造函数，默认拷贝构造函数，默认赋值函数，默认移动构造函数，默认移动赋值函数。有一个额外要求，对应的函数在类型的基类和成员中也具有noexcept声明。

```
struct X {};
#define PRINT_NOEXCEPT(x)    \
    std::cout << #x << " = " << x << std::endl

int main() {
    X x;
    PRINT_NOEXCEPT(noexcept(X()));
    PRINT_NOEXCEPT(noexcept(X(x)));
    PRINT_NOEXCEPT(noexcept(X(std::move(x))));
    PRINT_NOEXCEPT(noexcept(x.operator=(x)));
    PRINT_NOEXCEPT(noexcept(x.operator=(std::move(x))));
}
```

# 默认使用noexcept的函数

- 类型的析构函数以及delete运算符默认带有noexcept声明。

```
struct X {
};
struct X1 {
    ~X1() {}
};
#define PRINT_NOEXCEPT(x)    \
    std::cout << #x << " = " << x << std::endl
int main() {
    X *x = new X;
    X1 *x1 = new X1;
    PRINT_NOEXCEPT(noexcept(x->~X()));
    PRINT_NOEXCEPT(noexcept(x1->~X1()));
}
```

# 使用noexcept的时机

- 一定不会出现异常的函数
- 当我们的目标是提供不会失败或者不会抛出异常的函数时可以使用noexcept声明

# 将异常规范作为类型的一部分

- C++17之前存在的问题

```
void(*fp)() noexcept = nullptr;
void foo() {}

int main()
{
    fp = &foo;
}
```

# 感谢您的观看

# 欢迎关注