On catching exceptions in PPL tasks

devblogs.microsoft.com/oldnewthing/20230519-00

May 19, 2023

Raymond Chen

Say you want to catch exceptions that emerge from the Parallel Patterns Library (PPL) `task` class. Your initial inclination might be to write something like this:

```
extern concurrency::task<int> get_id_async();

void test()
{
  try {
    get_id_async().then([=](int id) {
      return get_name_from_id_async(id);
    }).then([=](std::string name) {
      update_name(name);
    });
  } catch (...) {
    /* deal with the exception */
  }
}
```

However, this only catches half of the exceptions (the rare half).

The code inside the `try` block builds a task chain:[1] It calls `get_id_async()` to obtain a `task`, and then calls the `then()` method on that `task` to register a continuation.

This means that the `try` statement will catch an exception that is thrown by the `get_value_async()` function before returning a `task`, and it will catch an exception that is thrown when attempting to register the continuation. But it doesn't catch the exceptions *thrown by the task itself*.

If an exception is thrown by the task, it is captured in the task, and the failed task is then passed to the `then` lambda, if possible.

In our example, the lambda is the `[=](int id)`. This does not accept a `task<int>`, so it cannot receive the failed task. The exception then propagates down the chain into the next task, which then tries to pass to its own lambda, `[=](int std::string name)`. This lambda does not accept a `task<std::string>`, so the exception tries to propagate into the next task, but that's the end of the chain.

If an exception falls off the end of a task chain, the Parallel Patterns Library (PPL) does what happens when an exception goes unhandled in a C++ program: It terminates the process.

Similarly, if an exception is thrown by any of the lambdas, or if `get_name_from_id_async` produces a failed task, then the exceptions from the lambda or the task propagate down the task chain and fall off the end, unhandled, leading to process termination due to an unhandled exception.

Remember, the lambdas are queued for execution when the task completes. They don't run immediately, so you can't catch them with a synchronous `try`/`catch` in the code that creates the task chain. Those exceptions are reported as the task chain proceeds.

To catch those exceptions, you need a `then` lambda that accepts a `task`.

```
void test()
{
  try {
    get_id_async().then([=](int id) {
      return get_name_from_id_async(id);
    }).then([=](std::string name) {
      update_name(name);
    }).then([=](concurency::task<void> precedent) {
      try {
        precedent.get();
      } catch (...) {
        /* deal with exceptions in the tasks */
      }
    });
  } catch (...) {
    /* deal with exceptions building the task chain */
  }
}
```

When you receive the `task`, you call `get()` to get the task result. If the task failed, then `get()` raises an exception, so you wrap the `get()` call inside a `try`/`catch` to deal with the exception.

In this case, we just let the exception propagate down the task chain, and we have a single "sink" that catches them all. If you wanted to catch each step separately, you could do that, too:

```
void test()
{
  try {
    get_id_async().then([=](concurrency::task<int> precedent) {
      try {
        int id = precedent.get();
        return get_name_from_id_async(id);
      } catch (...) {
        /* get_id_async failed; produce empty string */
        return concurrency::task_from_result(std::string());
      }
    }).then([=](concurrency::task<std::string> precedent) {
      try {
        update_name(precedent.get());
      } catch (...) {
        /* get_name_from_id_async() or update_name() failed */
      }
    });
  } catch (...) {
    /* deal with exceptions building the task chain */
  }
}
```

In practice, the exceptions when building the task chain come from `get_id_async()` (before it produces the task) or `std::bad_alloc` (unable to allocate memory for the task chain).

Bonus chatter: Coroutines let you write more natural-looking code.

```
concurrency::task<void> test()
{
  try {
    auto id = co_await get_id_async();
    auto name = co_await get_name_from_id_async(id);
    update_name(name);
  } catch (...) {
    /* deal with exceptions */
  }
}
```

Bonus reading: Windows Runtime asynchronous operations can fail in two different ways, so make sure you get them both, which is the reverse scenario, in which someone was careful to catch the asynchronous failure but forgot to catch the synchronous failure.

[1] Building task chains is basically a practical test to see how closely you paid attention to SICP. Writing loops in terms of continuations is particularly interesting.