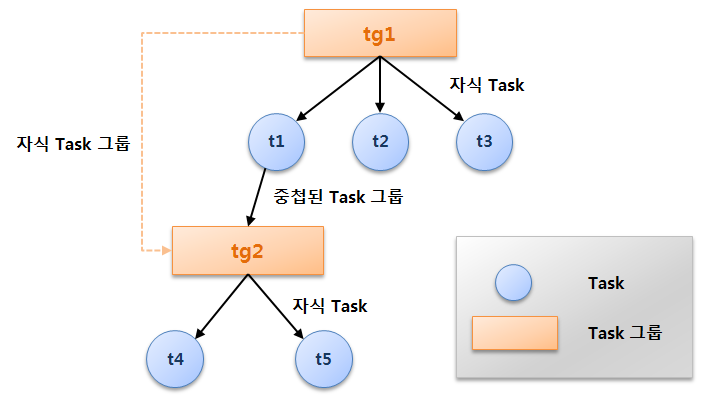
**병렬 작업 트리**

지난 강좌까지는 병렬 작업을 취소하기 위한 첫번째 방법인 Cancellation Token을 이용한 방법에 대하여 설명드렸었고 이번 강좌에서는 병렬 작업을 취소할 수 있는 나머지 두 가지 방법에 대하여 설명하려합니다. 이후 설명에 대한 이해를 돕기 위해 먼저 Task와 Task 그룹을 이용하여 병렬 작업 트리를 구성하는 방법을 살펴보도록 하죠. PPL에서는 자식 Task 내부에 Task 그룹을 포함(중첩)시키는 방식으로 다음과 같은 작업 트리를 구성할 수 있습니다.

 Task 그룹 tg1은 3개의 Task t1, t2, t3 를 실행하며 Task t1의 내부에서는 또 다른 Task 그룹인 tg2를 생성하여 Task t4와 t5를 실행합니다. **이렇게 자식 Task 내부에 Task 그룹을 중첩시키는 방식으로 tg2를 tg1의 자식 Task 그룹으로 구성할 수 있습니다.** tg1과 tg2를 structured\_task\_group 객체이고 t1, t2, t3, t4, t5가 task\_handle 객체라고 가정하면 위의 병렬 작업 트리는 다음과 같이 구현될 수 있습니다.

|  |
| --- |
| #include <ppl.h>  using namespace concurrency; using namespace std;  int wmain() {     // 최상위 Task 그룹 생성     **structured\_task\_group tg1;**      // 중첩된 Task 그룹을 포함하는 Task 생성     **auto t1 = make\_task([&] {**          // 중첩된 Task 그룹 생성         **structured\_task\_group tg2;**          // 자식 Task 생성         auto t4 = make\_task([&] {             // TODO: 작업 수행         });          // 자식 Task 생성         auto t5 = make\_task([&] {             // TODO: 작업 수행         });          // 자식 Task를 실행하고 종료 대기         tg2.run(t4);         tg2.run(t5);         tg2.wait();     });      // 자식 Task 생성     auto t2 = make\_task([&] {         // TODO: 작업 수행     });      // 자식 Task 생성     auto t3 = make\_task([&] {         // TODO: 작업 수행     });      // 자식 Task를 실행하고 종료 대기     tg1.run(t1);     tg1.run(t2);     tg1.run(t3);     tg1.wait(); } |

예제 코드에 대하여 따로 설명드리지 않아도 어떻게 구성되는지 이해가 되셨으리라 생각합니다. 위 예제에서는 structured\_task\_group을 사용하였지만 그 대신 task\_group을 사용할 수 있습니다. 그리고 비슷한 작업 트리를 구성하기 위하여 task 클래스를 사용할 수도 있지만 task 작업 트리는 부모 Task가 종료된 이후에 자식 Task가 실행되는 의존성을 가지는 작업 트리라는 점이 다르다는 것을 유념하시기 바랍니다. 이후 나오는 병렬 작업 취소 예제는 위와 같은 병렬 작업 트리를 구성하는 상태라고 가정하고 진행하겠습니다.

**Task 그룹 작업 취소(cancel 메서드)**

**병렬 작업을 취소하는 두번째 방법은 Task 그룹의 cancel 메서드를 이용한 방법입니다.** 지난 강좌에서는 Task 그룹의 작업을 취소하기 위해 Cancellation Token을 이용하였는데, Task 그룹의 cancel 메서드를 이용하면 좀 더 직관적이고 간단하게 Task 그룹의 작업을 취소할 수 있습니다. PPL에서는 Task 그룹을 구현한 두 가지 클래스, task\_group 클래스와 structured\_task\_group 클래스의 cancel 메서드를 제공합니다.**Task 그룹의 작업 취소 요청을 위해 cancel 메서드를 호출하면 Task 그룹은 canceled 상태가 되고 아직 시작되지 않은 작업이 있다면 더이상 실행되지 않습니다.**

PPL에서는 내부적으로 Exception을 발생시키고 발생된 Exception를 처리하는 방식으로 작업 취소를 구현하고 있습니다. **만약 Task 내부에서 알려지지 않은 Exception을 처리할 경우 예기치 못한 상황이 발생할 수 있으므로 절대로 임의의 Exception을 처리하지 않도록 구현해야 합니다.**

Cancellation Token을 이용한 작업 취소와 마찬가지로 Task 그룹의 cancel 메서드는 어디까지나 작업 취소 요청이기 때문에 실행중인 작업이 강제로 중단되지 않습니다. **그러므로 Task 내부에서는 Task 그룹의 is\_canceling 메서드를 주기적으로 호출하여 작업 취소 요청이 발생했는지 감시하고 true 값을 리턴할 경우 task body를 종료하도록 구현해야합니다.** 아래 예제는 병렬 작업 트리의 일부로 Task 그룹의 cancel 메서드를 이용한 작업 취소 방식을 보여줍니다.

|  |
| --- |
| #include <ppl.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  // 작업 수행후 결과값 반환 bool work(int index) {     if (index == 100)         return false;      return true; }  // task\_group\_status를 스트링으로 변환 std::wstring get\_status\_string(task\_group\_status status) {     switch (status)     {     case not\_complete:         return L"not\_complete";      case completed:         return L"completed";      case canceled:         return L"canceled";     }     return L""; }  int wmain() {     **structured\_task\_group tg2;**      // 자식 Task 생성     auto t4 = make\_task([&] {         // 반복 작업 수행         for (int i = 0; i < 1000; ++i)         {             // 작업을 수행하고 결과값으로 false 반환시 작업 취소 후 루프를 빠져나옴             **bool succeeded = work(i);**             if (!succeeded)             {                 wcout << L"Canceling task group..." << endl;                 **tg2.cancel();**                 break;             }         }     });      // 자식 Task 생성     auto t5 = make\_task([&] {         // 반복 작업 수행         for (int i = 0; i < 1000000; ++i)         {             // 오버헤드를 줄이기 위해서 가끔 취소 요청 감시             **if ((i % 100) == 0)**             {                 **if (tg2.is\_canceling())**                 {                     wcout << L"The task was canceled." << endl;                     **break;**                 }             }              // TODO: 작업 수행         }     });      // 자식 Task를 실행하고 종료 대기     tg2.run(t4);     tg2.run(t5);     **task\_group\_status status = tg2.wait();**     wcout << L"Task group status: " << get\_status\_string(status) << endl; }  // 실행결과  Canceling task group... The task was canceled. Task group status: canceled |

이번 예제는 제 강좌를 쭉 보셨다면 바로 이해가 되실만한 내용이네요. 루프 안에서 100번째 반복마다 취소 요청 여부를 감시하도록 구현되어있는데 이 값은 작업의 소요시간에 따라서 적절하게 조절하면 됩니다. 그리고 Task 내부에서 Task 그룹 객체에 접근할 수 없는 경우에는 is\_canceling 메서드 대신 is\_current\_task\_group\_canceling 함수를 사용하여 부모 Task가 취소되었는지 감시할 수 있으며 이 방식은 Cancellation Token을 이용했던 방식과 같습니다.

위 예제가 병렬 작업 트리를 사용하는 코드의 일부이고 두개의 Task 그룹(tg1, g2)과 다섯개의 Task(t1, t2, t3, t4, t5)가 존재한다고 가정해봅시다. **cancel 메서드는 Task 그룹에 속한 자식 Task 에게만 영향을 미치기때문에 tg2의 cancel 메서드를 호출하면 자식 Task인 t4와 t5가 취소됩니다.** tg1은 취소되지 않았기 때문에 t1, t2, t3는 영향이 없겠죠. **그리고 부모 Task 그룹이 취소되면 자식 Task 그룹도 취소되기 때문에 아래 예제처럼 tg1의 cancel 메서드를 호출하면 병렬 작업 트리의 모든 Task(t1, t2, t3, t4, t5)가 취소됩니다.**

|  |
| --- |
| // 자식 Task 생성 auto t4 = make\_task([&] {     // 반복 작업 수행     for (int i = 0; i < 1000; ++i)     {         // 작업을 수행하고 결과값으로 false 반환시 작업 취소 후 루프를 빠져나옴         bool succeeded = work(i);         if (!succeeded)         {             wcout << L"Canceling task group..." << endl;             **tg1.cancel();**             break;         }     } }); |

4장 Task 그룹 강좌에서 설명드렸듯이 structured\_task\_group 클래스는 thread-safe 하지 않기 때문에 자식 Task에서 부모 Task 그룹의 메서드 호출시 예기치 않은 오류가 발생할 수 있습니다. 하지만 cancel 메서드와 is\_cancelling 메서드는 이 규칙을 적용받지 않는 예외 사항이기 때문에 자식 Task에서 자유롭게 호출할 수 있습니다.

**Task 그룹 작업 취소(Exception)**

**병렬 작업을 취소하는 마지막 세번째 방법은 Exception을 이용한 방법입니다.** Cancellation Token 또는 cancel 메서드의 경우 취소 상태를 부모에서 자식들에게 전파하는 하향식 방법이라고 한다면 반대로 Exception을 이용한 방식은 자식들에게서 각각 독립적으로 발생한 Exception을 모아서 부모에게 전파하는 상향식 방법이라고 얘기할 수 있습니다. **일반적으로 Exception을 이용한 작업 취소는 기존에 설명했던 방식에 비해 비효율적이고 오류를 발생시키기 쉬우므로 가능하면 Cancellation Token 또는 cancel 메서드를 이용하시길 권장합니다.**

비록 권장하는 방법은 아니지만 어떻게 구현해야 하는지 정도는 알아야겠죠. 그럼 예제를 통해서 Exception을 이용한 작업 취소를 어떻게 작성해야 하는지 알아보도록 하겠습니다. 이 예제는 위에서 설명했던 cancel 메서드를 이용한 작업 취소 예제와 같은 역할을 하므로 두 예제를 비교하면서 살펴보시기 바랍니다.

|  |
| --- |
| #include <ppl.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  // 작업 수행후 결과값 반환 bool work(int index) {     if (index == 100)         return false;      return true; }  int wmain() {     structured\_task\_group tg2;      // 자식 Task 생성     auto t4 = make\_task([&] {         // 반복 작업 수행         for (int i = 0; i < 1000; ++i)         {             // 작업을 수행하고 결과값으로 false 반환시 작업 취소 후 루프를 빠져나옴             bool succeeded = work(i);             if (!succeeded)             {                 wcout << L"Canceling task group..." << endl;                 **throw exception("The task failed");**             }         }     });      // 자식 Task 생성     auto t5 = make\_task([&] {         // 반복 작업 수행         for (int i = 0; i < 1000000; ++i)         {             // TODO: 작업 수행         }     });      // 자식 Task 실행     tg2.run(t4);     tg2.run(t5);      // 자식 Task 종료 대기 후 발생된 Exception 출력     **try     {         tg2.wait();     }     catch (const exception& e)     {         wcout << e.what() << endl;     }** }  // 실행결과  Canceling task group... The task failed |

**이번 예제에서는 Task 그룹의 작업을 취소하기 위하여 cancel 메서드를 호출하는 대신 Exception을 발생시킵니다.**Task 그룹에 포함된 자식 Task 내부에서 Exception을 발생시키게 되면 런타임은 발생된 예외를 저장해놓고 Task 그룹의 모든 작업이 종료될 때까지 대기한 후 모든 작업이 종료되면 wait 메서드 내부에서 저장해놓았던 Exception을 다시 발생시킵니다. **그러므로 wait 메서드 호출 부분을 try-catch 블럭으로 묶어줌으로서 Exception에 의한 작업 취소를 처리하도록 구현할 수 있습니다.**

Exception을 이용한 작업 취소는 cancel 메서드와 마찬가지로 아직 실행되지 않은 작업이 있다면 더이상 실행하지 않습니다. 하지만 이미 실행중이던 Task의 작업은 중단되지 않고 이 경우 is\_canceling 메서드를 사용할 수도 없기 때문에 t5는 모든 작업을 수행한 뒤 종료됩니다.

아까와 마찬가지로 이 예제가 병렬 작업 트리를 사용하는 코드의 일부라고 가정하면 tg2에 대한 wait 메서드를 try-catch 블럭으로 묶어주었기 때문에 tg2만 취소되고 tg1은 취소되지 않습니다. **만약 tg1과 tg2 를 모두 취소하려면 다음과 같이 부모 Task 그룹인 tg1의 wait 메서드를 try-catch 블럭으로 묶어주면 됩니다.**

|  |
| --- |
| // 자식 Task 실행 tg1.run(t1); tg1.run(t2); tg1.run(t3);  // 자식 Task 종료 대기 후 발생된 Exception 출력 try {     **tg1.wait();** } catch (const exception& e) {     wcout << e.what() << endl; } |

지금까지 Task 그룹의 작업 취소를 위해 cancel 메서드를 이용한 방식과 Exception을 이용한 방식까지 모두 살펴보았습니다.