**Cancellation 상속**

**부모-자식 관계에 있는 Task(연결된 Task)에서 부모 Task의 작업을 취소했을 경우 자식 Task에게 Cancellation이 상속된다는 특징을 가지고있습니다.** Cancellation 상속이란 부모 Task가 취소되어 canceled 상태가 되면 자식 Task 역시 취소되어 실행되지 않고 canceled 상태를 계승하는 것을 얘기합니다.

지난 강좌 ['[VC++] 병렬 프로그래밍 완전 정복 #2 - task (1)'](http://devmachine.blog.me/178262872) 편에서 value 기반의 연결과 task 기반의 연결의 차이점에 대하여 설명하다가 부모 task가 취소되었을 때의 자식 task의 동작에 대해 언급했던 내용을 기억하시나요? 역시나 기억이 나지 않으시겠죠..^^; 병렬 작업의 취소에 있어서도 value 기반의 연결과 task 기반의 연결은 커다란 차이점을 가지며 그 내용은 다음과 같이 정리해볼 수 있습니다.

* value 기반의 연결에서 자식 Task는 부모 Task의 Cancellation을 상속받는다.
* task 기반의 연결에서 자식 Task는 부모 Task의 Cancellation을 상속받지 않는다.
* 중첩된 Task 에서 자식 Task는 부모 Task의 Cancellation을 상속받지 않는다.

**cancel\_current\_task 함수를 이용하여 부모 Task의 작업을 취소하였을때 Cancellation이 상속되어 자식 Task가 실행되지 않고 canceled 상태가 되는 경우는 value 기반의 연결 뿐입니다.** 그 이외의 상황에서 자식 Task가 취소되게 하려면 cancellation\_token 을 명시적으로 생성하여 넘겨주고 cancellation\_token\_source::cancel 메서드로 취소 요청을 하여 자식 Task가 더이상 실행되지 않도록 하는 것이 유일한 방법입니다. 이제부터는 예제를 통해서 살펴보도록 하죠.

|  |
| --- |
| #include <ppltasks.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  // task\_status를 스트링으로 변환 std::wstring get\_status\_string(task\_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() {     auto t1 = create\_task([]() -> int     {         // 현재 작업 취소         **cancel\_current\_task();**     });      // task 기반의 연결     auto t2 = t1.then([](task<int> t)     {         try         {             **int n = t.get();**             wcout << L"The previous task returned " << n << L'.' << endl;         }         catch (const task\_canceled& e)         {             **wcout << L"The previous task was canceled." << endl;**         }     });      // 모든 Task 종료 대기     task\_status status = t2.wait();     **wcout << L"Task status of t2: " << get\_status\_string(status) << endl;** }  // 실행결과  The previous task was canceled. Task status of t2: completed |

부모 Task t1과 자식 Task t2가 Task 기반으로 연결되어있는 모습입니다. **부모 Task t1이 시작되면 cancel\_current\_task 함수를 호출하여 바로 작업을 취소하는데 task 기반의 연결이기 때문에 자식 Task가 Cancellation을 상속받지 않아 t1의 취소되더라도 t2 가 실행됩니다.** t2 내부에서는 파라미터로 넘어온 부모 Task의 get 메서드를 호출하는데 부모 Task는 canceled 상태이므로 task\_canceled 예외가 발생하게 되죠. 마지막으로 모든 Task 종료 이후의 자식 Task t2 의 상태가 completed 라는 것을 유념하시기 바랍니다. 그럼 이번엔 value 기반의 연결 예제를 살펴보도록 합시다.

|  |
| --- |
| auto t1 = create\_task([]() -> int {     // 현재 작업 취소    **cancel\_current\_task();** });  // value 기반의 연결 auto t2 = t1.then([](int n) {     wcout << L"The previous task returned " << n << L'.' << endl; });  try {     // 모든 Task 종료 대기     **t2.get();** } catch (const task\_canceled& e) {    **wcout << L"The task was canceled." << endl;** }  // status 출력 task\_status status = t2.wait(); **wcout << L"Task status of t2: " << get\_status\_string(status) << endl;**  // 실행결과  The task was canceled. Task status of t2: canceled |

이번엔 Task t1과 t2를 value 기반으로 연결한 다음 마찬가지로 부모 Task t1이 시작되면 바로 작업을 취소합니다.**value 기반의 연결이기 때문에 이전 예제와는 달리 자식 Task가 Cancellation을 상속받아 실행되지 않고 바로 canceled 상태로 변경됩니다.** 그 결과로 자식 task t2의 get 메서드 호출시 task\_canceled 예외가 발생하게 되며 상태를 출력해봐도 canceled 상태임을 알 수 있습니다.

**Task 그룹 작업 취소(Cancellation Token)**

Task 그룹 역시 Cancellation Token을 이용하여 실행중인 작업을 취소할 수 있으며 이는 Task를 취소하는 방식과 크게 다르지 않습니다. 이번엔 바로 예제를 보면서 설명을 드리겠습니다. 이 예제는 Cancellation Token을 이용하여 Task의 실행을 취소하는 이전 강좌의 예제를 Task 그룹으로 변경한 예제이니 비교하면서 보시면 더 도움이 되실겁니다.

|  |
| --- |
| #include <ppl.h> #include <iostream> #include <sstream>  using namespace concurrency; using namespace std;  // 작업 시뮬레이션 bool do\_work() {     wcout << L"Performing work..." << endl;     wait(250);     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() {     **cancellation\_token\_source cts;**     **auto token = cts.get\_token();**      // Task 그룹 생성     wcout << L"Creating task group..." << endl;     **structured\_task\_group tg(token);**      auto t = make\_task([&]     {         bool moreToDo = true;         while (moreToDo)         {             // 취소 요청 감시             **if (is\_current\_task\_group\_canceling())**             {                 // TODO: 이곳에서 리소스 해제 작업 등을 수행                  // 현재 실행중인 작업을 취소                 **break;**             }             else             {                 // 작업 수행                 moreToDo = do\_work();             }         }     });      // 자식 Task 실행     wcout << L"Run task..." << endl;     tg.run(t);      // 1초 대기 후 취소 요청     wait(1000);      wcout << L"Canceling task group..." << endl;     **cts.cancel();**      wcout << L"Waiting for task group to complete..." << endl;     **task\_group\_status status = tg.wait();**      wcout << L"Done." << endl;     wcout << L"Task group status: " << get\_status\_string(status) << endl; }  // 실행결과  Creating task group... Performing work... Performing work... Performing work... Performing work... Canceling task group... Waiting for task group to complete... Done.  Task group status: canceled |

cancellation\_token\_source 객체를 생성하고 structured\_task\_group 클래스의 생성자에 cancellation\_token 객체를 전달하여 해당 Task 그룹이 Cancellation Token을 구독하게 합니다. 그리고 자식 Task 실행 이후에 cancel 메서드를 호출하여 Cancellation Token을 구독하는 Task 그룹에게 작업 취소를 요청하는것 까지는 특별히 다른 점이 없습니다.

**하지만 Task 내부에서 취소 요청이 발생했는지 확인하기 위하여 is\_task\_cancellation\_requested 함수 대신 is\_current\_task\_group\_canceling 함수를 호출하여 감시하고 cancel\_current\_task 함수를 호출하는 대신 break로 루프를 빠져나와 작업을 종료하는 것을 볼 수 있습니다.** Cancellation Token을 이용한 Task 그룹의 취소의 경우 이러한 차이점만 유념하시면 될 것 같네요.