비동기 프로그래밍과 웹 프레임워크

Sol Kim

October 6, 2016

Abstract

이 글에서는 비동기 프로그래밍(Asynchronous)과 웹 프레임워크의 관계에 대해 알아보고, 향후 어떤 방향으로 발전할 것인지 살펴본다. 전반부에서는 비동기 프로그래밍과 웹 프로그래밍의 관계를 살펴보기에 앞서 비동기 프로그래밍과 관련있는 중요한 개념들을 살펴본다. 후반부에서는 비동기 프로그래밍을 적용한 웹 프레임워크란 무엇인지, 어떤 면에서 기존의 웹 프레임워크에 비해 이점이 있는지 살펴본다. 마지막으로 비동기 프로그래밍을 넘어서 웹 프레임워크들이 어떤 방향을 향해 발전해 가고 있는지, 어떤 작업들이 이와 밀접한 관련이 있는지 살펴본다.

Keywords: 웹 프레임워크, 비동기 프로그래밍, 동시성, 계산 모델, 추상화

1 앞서 짚고 넘어가야 할 중요한 개념들

이 장에서는 비동기 프로그래밍에 대해 자세히 알아보기 위해 이와 관련 있는 여러 추상적인 개념들 — 멈추어 기다리기(Blocking I/O)와 멈추지 않고 기다리기(Non-Blocking I/O), 병렬성(Parallelism)과 동시성(Concurrency), 비동기 프로그래밍(Asynchronous Programming) — 을 살펴본다. 각 개념들이 대표하고있는 문제 해결 방식이 풀고자 하는 문제는 무엇인지, 해결 방식과 문제를 추상화 하여 알아보는 것이 목적이다.

1.1 입출력을 기다리는 두 가지 방식, 멈추고 기다리기(Blocking I/O)와 멈추지 않고 기다리기(Non-Blocking I/O)

입출력을 기다리는 방식은 크게 멈추고 기다리는 방식(Blocking I/O)과 멈추지 않고 기다리는 방식(Non-Blocking I/O)이 있다. 프로그램이 외부와 데이터를 주고받을 때, 외부에서 데이터를 처리하는 동안 가만히 기다리는 것을 Blocking I/O, 다른 일을 하며 기다리는 것을 Non-Blocking I/O 이라고 한다. 멈추고 기다리기의 경우, 외부에서 작업을 진행하는 동안 프로그램은 자원을 외부 작업이 끝나기를 기다리는 데에 소비한다. 반면에 멈추지 않고 다른 일을 하면서 기다리기의 경우 외부에서 필요한 작업을 하는 동안 프로그램은 다른 일을 할 수 있다. 따라서 멈추지 않고 기다리는 방식을 이용하면 개발자의 역량에 따라 프로그램의 자원을 훨씬 효율적으로 사용할 수 있는 여지가 있다.

다만, 일을 멈추지 않고 기다리는 방식을 사용할 경우 프로그램과 외부 작업 사이에 데이터를 주고 받기 위한 약속이 필요하다. 멈추고 기다리는 방식은 외부 작업의 결과를 멈추어서 기다리기 때문에 그 자리에서 바로 받을 수 있다. 반면에 멈추지 않고 기다리는 경우, 프로그램은 결과를 얻어 오기 까지 멈추어 있는 것이 아니라 다른 일을 하다 나중에 결과를 사용하는

것이 가능해 졌을 때 결과를 받아서 사용하는 방식이다. 다만 멈추지 않고 기다리기를 사용하는 경우에는 프로그램은 자신의 현재 맥락과 상관없이 외부 작업과 데이터를 공유할 방법이 필요하고 이를 위한 추가 구현이 필요하다.

이 때 결과를 가져오는 방식에 따라 멈추지 않고 기다리기를 다시 두 분류로 나누기도 한다[15]. 입출력을 멈추지 않고 기다리는 방식 중, 주도적으로 데이터를 가져오는 방식(Pull Base)을 사용하는 경우를 멈추지 않고 기다리기(Non-Blocking I/O)라 부른다. 반대로 수동적으로 받아오는 방식(Push Base)을 사용하는 경우를 비동기 방식으로 입출력 기다리기(Asynchronous I/O)라 부른다. 일반적으로 수동적으로 받아오는 방식이 자원을 훨씬 효율적으로 사용할 수 지만, 주도적으로 가져오는 방식에 비해 구현해야 하는 코드가 많고 복잡한 단점이 있다. 결과를 가져오는 방식에 따라 다시 두 분류로 나누더라도, 본질은 여전히 입출력 결과를 수행하는 과정 동안 일하기를 멈추지 않고 기다린 다는 점이다. 따라서 이 글에서는 논의를 단순하게 하기 위해 입출력을 다루는 방식에 대한 논의는 처음과 마찬가지로 오로지 멈추고 기다리기와 멈추지 않고 기다리는 두 분류로만 나눌 것이다. 이는 이후 등장할 개념인 비동기 프로그래밍(Asynchronous Programming)의 비동기 기다리기(Asynchronous Wait)과 비동기 방식으로 입출력 기다리기 사이에 혼동을 줄이기 위한 것이기도 하다.

1.2 여러가지 일을 다루는 두 가지 방식, 병렬성(Parallelism)과 동시성(Concurrency)

병렬성(Parallelism)은 작업을 처리하는 일꾼을 물리적으로 여럿을 두어 같은 작업을 동시에 수행하는 것을 뜻하고, 동시성(Concurrency)은 일꾼이 여러가지 일을 번갈아 가며 수행하는 것을 뜻한다. 병렬성은 일꾼이 여럿이어야만 성립하는 개념이지만, 동시성은 일꾼의 수 와는 상관 없이 오로지 일꾼이 일하는 방식과 연관 있다. 동시성은 일꾼이 작업을 차례대로 끝내가며 진행하는 것이 아닌, 맡은 일이 끝나지 않았음에도 여러 일을 필요에 따라 번갈아 가면서 진행하는 방식으로 일하는 것을 뜻한다. 하나의 일꾼 만으로도 동시성을 확보할 수 있으며, 여러 일꾼을 사용하여 병렬성을 확보하고 이에 더해 동시성을 확보하는 것도 가능하다. 동시성은 병렬성과 시너지 효과가 있기 때문에 후자 방식을 통해 동시성을 확보하는 경우가 조금 더 일반적이다.

주의해야하는 점은 병렬성과 동시성을 같이 확보하는 경우 시너지 효과를 낼 수 있지만, 둘 사이에 의존성은 전혀 없다는 것이다. 동시성은 자원을 효율적으로 사용하는 것이 목적이고, 병렬성은 많은 자원을 투자해서 일의 처리량을 늘리는 것이 목적이다. 동시성을 통해 해결하고자 하는 문제와, 병렬성을 통해 해결하고자 하는 문제는 완전히 독립된 서로 다른 문제이다. 동시성을 통해 풀고자 하는 문제를 위해 병렬성을 도입이 항상 필수는 아니며, 반대로 병렬성을 통해 풀고자 하는 문제에 동시성 도입이 항상 필수는 아니다. 풀고자 하는 문제에 따라 하나만 적용하기도, 둘 다 적용하기도 할 뿐이다.

위와 마찬가지로 동시성과 멈추지 않고 기다리기(Non-Blocking I/O)를 같이 사용하면 시너지 효과를 낼 수 있지만, 둘 사이에 의존성은 전혀 없다. 동시성을 확보한 시스템에서 멈추지 않고 기다리기를 이용하여 작업의 효율을 높이는 방식은 일반적인 사용법이다. 그렇다고 하여 둘 사이에 의존 관계가 있다고 이야기 할 수는 없다. 멈추지 않고 기다리기는 외부 입출력과 프로그램 사이의 작업을 어떻게 할 것인지를 대표하는 개념이며, 동시성은 일꾼이 일하는 방식을 대표하는 개념이다 동시성을 확보한 프로그램이 멈추지 않고 기다리기를 중요한 요소로 사용할 수도 있고, 전혀 사용하지 않을 수도 있다. 일꾼이 하나라도 모든 작업에 대해 똑같이 1 초씩 작업을 진행하기로 하고 일을 수행하면 동시성을 확보했다고 할 수 있다. 풀고자 하는 문제에 따라 알맞은 추상적 개념을 재료로 삼아 조립 할 뿐이다.

병렬성과 동시성은 웹 프레임워크에서 매우 중요하게 다루고 있는 개념이다. 비동기 프로

그래밍(Asynchronous Programming) 방식을 사용하기 이전의 웹 프레임워크 에서는 병렬성만으로 성능 확보에 힘썼다. 하지만 비동기 프로그래밍 방식을 통해 동시성 확보하기 시작한 이후부터 단순히 같은 작업을 하는 일꾼을 늘리는 방식을 넘어서 일꾼들을 더 효율적으로 사용하는 방식을 사용하기 시작했다. 이는 웹 프레임워크의 핵심 기능인 요청을 처리하는 방식에 있어서 아주 큰 진화로, 병렬성을 이용한 물리적인 방식의 성능 향상을 넘어설 수 있는 중요한 발전이라고 할 수 있다. 이에 대한 자세한 내용은 2장에서 비동기 프로그래밍과 함께 더 자세히 살펴볼 것이다.

1.3 비동기 프로그래밍(Asynchronous Programming)

비동기 프로그래밍(Asynchronous Programming)은 실행 한 코드의 결과를 기다리지 않고 별도의 채널에 결과 처리를 맡긴 후 다음 작업을 바로 진행하는 방식의 프로그래밍을 뜻한다. 전통적인 절차 중심의 프로그래밍(Imperative Programming)에서는 코드를 실행 한 후 결과를 기다렸다 받아서 다음으로 진행하는 방식으로 코드를 실행한다. 비동기 프로그래밍은 이와 달리, 코드의 실행 결과 처리 및 활용을 별도의 채널에 맡겨둔 후 결과를 기다리지 않고 바로 다음 코드를 실행하는 방식으로 프로그램을 진행한다. 코드 실행의 결과가 다음 코드 들에서 직접 연관이 없는 경우 실행의 결과를 기다릴 이유가 없기 때문에 자원을 더 효율적으로 사용할 수 있는 여지가 생긴다.

비동기 프로그래밍에서는 크게 두 가지 방식 — 함수 전달을 통해 처리하기(Callback), 언어에서 지원하는 방식; Future, Promise, 등 — 으로 결과를 처리한다. 특정 함수를 호출하여 처리하는 방식은 함수를 값처럼 사용하기(First-class function)를 지원하는 언어에서 자연스럽게 사용하는 기법이다. 비동기 프로그래밍 방식으로 진행할 코드를 실행할 때 실행 하는 시점에 결과를 처리해 줄 함수를 일꾼에게 같이 넘기는 기법으로, 일꾼은 작업 후 실제 결과를 실행 시점에 받은 함수를 통해 처리한다. 이 방식은 직관적이고 일반적이지만, 실행 하는 시점에 매번 함수를 넘겨주어야 하는 점에서 코드의 실행 흐름과 변수의 범위를 눈으로 쫓기 힘들도록 만든다. 언어에서 지원하는 방식의 경우 비동기로 실행 한 코드의 결과를 언어에서 지원하는 방식으로 되돌려 받아 처리하는 방식이다. 이 방식을 이용하면 코드를 비동기로 실행할 때 해당 코드의 실제 결과가 아닌 언어에서 지정한 방식으로 결과를 즉시 되돌려 받는다. 언어적 방식을 통해 즉시 되돌려 받는 결과로는 Future, Promise 같은 객체 형태, Python의 코루틴(Coroutine)과 같은 언어의 문법을 이용하는 형태, 또는 Go와 같이 고루틴(Goroutine) 또는 결과를 주고받을 별도의 채널을 사용하는 형태 등이 있다. 즉시 되돌려 받은 결과를 이용하여 각 방식에 알맞게 추후에 비동기로 실행 한 코드의 실제 결과를 꺼내어 사용하는 형태이다. 객체 형태로 돌려받는 경우는 실행 흐름을 프로그램으로 구현하여 마음껏 다를 수 있기 때문에 높은 자유를 누릴 수 있지만, 이를 위해 필요한 코드를 더 구현해야 하는 단점이 있다. 반면 언어의 문법을 이용하는 형태는 언어가 제공하는 제어문을 통해 비동기 프로그래밍의 실행 흐름을 언어의 실행 흐름으로 가져와 코드를 작성하게 하지만, 비동기 실행을 언어의 특정 문법에 제한해 버린다는 한계가 있다.

비동기 프로그래밍에서 가장 중요한 요소 중 하나는 비동기 기다리기(Asynchronous Wait 또는 줄여서 Await)이다. 프로그램을 만들다 보면, 비동기 프로그래밍으로 실행 했던 작업의 결과를 다시 최초의 실행 흐름으로 되가져와 사용해야 하는 경우가 생긴다. 이 때 비동기 기다리기는, 기존의 실행 흐름에서 필요한 결과를 비동기 프로그래밍으로 실행한 작업에서 만들어낼 때 까지 기다려 두 작업 간 타이밍을 맞추어 주는 역할을 한다. 또한 비동기 프로그래밍으로 실행 중인 작업이 있으니 함수를 끝내지 않고 기다리는 방식으로 사용할 수도 있다. 비동기 기다리기는 비동기 프로그래밍으로 실행 한 작업과 원래 실행 흐름 묶어주는 중요한 기법이며, 대부분 비동기 프로그래밍 - 비동기 기다리기를 한 쌍으로 두고 당연히 따라오는

개념으로 생각한다. 특히 절차 중심의 언어에서는 비동기 기다리기를 통해 실행 흐름을 제어하는 것은 자연스러운 방식이다. 언어적 방식인 Future 객체, Promise 객체, 코루틴(Coroutine) 문법, 고루틴(Goroutine) 문법, 등을 이용하여 비동기 프로그래밍을 지원하는 언어들은 비동기 기다리기를 제공한다. **오로지 Java만 절차 중심의 프로그래밍이면서도 아직까지 비동기 기다리기가 없다.**

비동기 기다리기는 교착 상태(Deadlock) 문제가 발생할 여지가 있고, 뿐만 아니라 추상화의 수준을 절차 중심의 프로그래밍 언어 수준으로 낮추는 주된 요인이다. 비동기 프로그래밍을 지원하는 거의 모든 언어에서 비동기 기다리기가 필수인 것은 이견이 없는 사안이며 아주 중요한 핵심 요소 중 하나이다. 하지만 이는 사용하기에 따라 비동기 프로그래밍을 절차 중심의 프로그래밍 관점으로 되돌려 와서 실행 흐름 제어의 늪에 빠지게 만들기도 한다. 비동기 프로그래밍의 기능을 절차 중심의 프로그래밍이 가지는 추상화 수준에 맞추어 사용할 수 밖에 없게 되는 문제 이외에도, 사용자의 역량에 따라 자원을 효율적으로 사용할 수 있는 여지가 크게 달라진다는 문제가 있다.

비동기 프로그래밍의 추상화 수준을 높이는 계산 모델 또는 프로그래밍 모델을 사용하면 실행 흐름을 직접 제어하는 일 — 절차 중심의 프로그래밍 관점 — 에서 벗어나 비동기 프로그래밍의 장점만을 온전히 누릴 수 있다. 높은 수준의 추상화를 제공하는 기법을 이용하여 실행 흐름을 직접 제어하는 고통에서 벗어나면 교착 상태에 빠질 상황을 피할 수 있으며, 사용자의 역량과 상관없이 비동기 프로그래밍이 제공하는 효율적 자원 사용을 온전히 누릴 수 있다. Haskell, Scala, 또는 OCaml같은 함수 중심의 프로그래밍 언어에서는 비동기 프로그래밍의 흐름 제어를 모나드(Monad)를 이용하여 함수를 엮어나가는 방식으로 추상화 수준을 높여 사용할 수 있다. 또는 반응하는 프로그래밍(Reactive Programming)을 이용하여 프로그래밍의 관점을 절차 중심이 아니라 데이터의 흐름으로 바꾸어서 실행 흐름을 데이터 흐름에 맞추어 따라가도록 하는 방식으로 추상화 수준을 높여 사용할 수 있다. 이에 대한 내용은 3장에서 더 자세히 다룰 예정이다.

비동기 프로그래밍은 동시성(Concurrency), 멈추지 않고 기다리기(Non-Blocking I/O)와 매우 흡사한 개념 같아 보이지만 독립된 정의로 다루어야 한다. 비동기 프로그래밍은 '결과를 기다리지 않고 다음 코드를 실행하는 흐름' 을 대표하는 개념이다. 이를 구현하기 위해 여러 일꾼을 사용하여 실행하도록 하든 일꾼 하나에 작업의 순서를 조절하여 실행하도록 하든, 어떻게 구현할 지는 비동기 프로그래밍과 연관이 있을 지언정 정의의 구성 요소라고 볼 수는 없다. 물론 일반적으로 비동기 프로그래밍은 병렬성(Parallelism)과 동시성 모두를 확보한 구현체를 통해 코드를 진행하지만 그렇지 않고도 비동기 프로그래밍을 제공하는 것은 여전히 가능하다. 한 가지 예로, 이벤트 루프(Event Loop)를 사용하여 병렬성을 확보하지 않고 오직 동시성만으로 비동기 프로그래밍을 제공하는 방식이 있다. 또는, 함수를 전달하여 처리하기(Callback) 방식을 사용하여 동시성 없이 비동기 프로그래밍을 제공할 수도 있다. 하지만 그것은 비동기 프로그래밍의 코드 진행을 구현하는 구현 방식들 중 하나일 뿐이지 비동기 프로그래밍이 병렬성, 동시성 또는 멈추지 않고 기다리기에 종속된 개념은 아니다.

2 비동기 프로그래밍(Asynchronous Programming)과 웹 프레임워크

이 장에서는 비동기 프로그래밍(Asynchronous Programming) 기법과 웹 프레임워크 사이에 관계를 알아보고, 앞으로 웹 프레임워크가 어떻게 발전할 것인지에 대해 이야기 할 것이다. 거의 모든 주요 웹 프레임워크들은 개발 부터 비동기 프로그래밍을 지원하거나, 기존의 프레임워크에 비동기 프로그래밍을 더하여 지원하는 것을 중요한 과제로 삼고 있다 [10, 11, 12, 14, 13]. 비

동기 프로그래밍이 웹 프레임워크와 어떤 관계가 있는지, 어떤 영향을 주고 어떤 이득을 가져다 주는지 살펴봄을 통해 중요한 요소로 떠오른 이유를 알아볼 것이다. 또한 비동기 프로그래밍을 넘어서 다양한 시도를 하고 있는 프로그래밍 모델들을 살펴보고, 앞으로 웹 프레임워크가 어떤 방향으로 발전할 것인지 살펴볼 것이다.

2.1 비동기 프로그래밍(Asynchronous Programming)과 웹 프레임워크

웹 프레임워크에서 비동기 프로그래밍(Asynchronous Programming)은 요청 받아 이를 처리할 로직을 호출할 때 비동기 프로그래밍 방식을 사용하는 것을 뜻한다. 웹 프레임워크의 핵심 기능은 요청을 받아 알맞은 처리를 해 주도록 요청과 처리 로직을 연결해 주는 것이다. 고전적인 웹 프레임워크, 즉 비동기 프로그래밍을 지원하지 않는 웹 프레임워크의 경우 요청을 받아서 처리하기까지 연속된 작업으로 보고 일을 수행한다. 반면, 웹 프레임워크에 비동기 프로그래밍을 적용했다 함은 요청과 처리 로직을 연결하는 과정에서 비동기 프로그래밍 방식을 사용한다는 것을 뜻한다. 기존의 방식과 달리, 웹 프레임워크는 요청을 처리 로직에 맡겨두고 나서 결과를 별도의 채널을 통해 다루도록 떠넘긴 후 다른 처리 해야 할 요청으로 넘어가 같은 일을 반복한다.

비동기 프로그래밍의 추상적인 실행 흐름을 실제로 수행해 줄 구현체로 동시성(Concurrency)을 확보 한 구현체를 사용하면, 두 가지 관점에서 자원을 더 효율적으로 사용할 수 있게 된다. 한 가지 관점은, 동시성을 확보하는 것 자체로 자원을 효율적으로 사용할 수 있는 여지가 생긴다는 점이다. 앞서 동시성에 대해 살펴본 장에서 이야기 나눈 바와 같이 동시성을 사용하는 경우 자원을 효율적으로 사용할 수 있다. 이에 더불어 동시성과 시너지 효과를 내거나 동시성을 구현하기 위해 사용하기도 하는 개념인 병렬성(Parallelism)과 멈추지 않고 기다리기(Non-Blocking I/O)를 같이 사용한다면, 이 구현체를 사용하는 것 자체 만으로 자원을 효율적으로 사용할 수 있게 된다. 나머지 한 가지 관점은, 요청과 처리 로직을 연결해 주는 일과 처리 로직의 일을 분리함을 통해 자원을 효율적으로 사용할 수 있다는 점이다. 요청을 처리 로직으로 연결하는 일에 드는 프로그램의 자원은 비교적 적은 반면, 처리 로직이 사용해야 하는 자원은 천차만별이며 아주 많은 자원을 사용해야 하거나 외부 요인으로 많은 시간을 기다려야 하는 경우도 있다. 처리 로직을 연결하는 일과 처리 로직의 일을 비동기 프로그래밍을 이용하여 분리해 버리면 자원을 필요한 곳에 필요한 만큼 투입할 수 있는 여지가 생기고 이를 통해 자원을 효율적으로 사용할 수 있게 된다.

웹 프레임워크에 비동기 프로그래밍을 도입하면 자원을 효율적으로 사용하는 것을 넘어서 물리적 한계를 극복하는 것이 가능하다. 비동기 프로그래밍을 사용하지 않는 웹 프레임워크의 경우, 요청을 받아서 처리를 진행한 후 처리가 끝나면 결과를 되돌려 주는 일련의 일을 하나의 일꾼이 수행한다. 이 방식의 문제는 일꾼의 수가 서버의 수, 프로세스의 수, 스레드의 수 등 구현에 따라 어쩔 수 없이 물리적으로 제약이 생긴다는 점이다. 이로 인해 고전적인 방식의 웹 프레임워크, 즉 비동기 프로그래밍을 지원하지 않는 웹 프레임워크의 동시에 처리할 수 있는 요청의 수는 일꾼 수의 물리적 제한을 따르게 된다. 반면에 비동기 프로그래밍을 이용하여 요청을 처리 로직에 연결하는 일과 처리 로직의 일 사이의 관계를 떼어낸다면, 동시에 처리할 수 있는 요청의 수는 일꾼 수가 가지는 물리적 제한을 따르지 않도록 할 수 있다. 즉, 비동기 프로그래밍을 이용하여 요청을 처리하면 일꾼 수의 물리적 한계를 따르지 않을 수 있기 때문에, 기존의 웹 프레임워크가 가지고 있던 물리적 한계 중 한 가지를 극복할 수 있게 된다.

2.2 스레드 풀(Thread Pool)과 비동기 프로그래밍(Asynchronous Programming) 그리고 웹 프레임워크

대부분의 언어는 비동기 프로그래밍을 제공하기 위해 그 구현체로 스레드 풀(Thread pool)을 사용한다 [16, 17, 18]. 스레드 풀은 흔히 아는 바와 같이 일꾼의 단위를 스레드로 두고, 미리 만들어둔 스레드를 풀에서 필요할 때 하나씩 꺼내어 사용하고 일이 끝나면 다시 풀에 넣어 다른 일에 사용할 수 있는 상태로 두는 방식이다. 즉, 병렬성과 동시성을 확보하여 작업을 효율적으로 수행하기 위한 기법이다. 대부분의 언어에서 스레드 풀을 비동기 프로그래밍의 코드 실행 흐름을 수행하도록 하는 구현체로 사용한다. 일꾼, 다시 말해 스레드가 일을 진행하다가 비동기로 실행해야 할 코드를 만나면 스레드 풀에서 스레드를 하나 꺼내어 비동기로 진행 할 일을 맡기고 기존에 일을 하던 스레드는 비동기 실행에 관여하지 않고 나머지 일을 마저 진행하는 방식이다. 비동기로 진행 할 일을 맡은 스레드는 일이 끝나면 결과를 약속에 맞게 — 함수 전달을 통해 처리하기(Callback), 언어에서 지원하는 방식; Future, Promise, 등 — 처리하고 다시 스레드 풀로 돌아간다.

스레드 풀을 구현체로 사용하는 비동기 프로그래밍을 사용 할 때에는 오래 걸리는 작업으로 인해 스레드가 고갈되지 않도록 신경 써야 한다. 스레드 풀을 이용한다는 것은 미리 만들어 둔 스레드를 사용하여 자원이 필요한 일이 생기면 그 때 투입하여 일을 수행하겠다는 의미이다. 문제는 오래 걸리는 작업이 많아지는 경우인데, 오래 걸리는 작업을 수행하기 위해 스레드를 계속해서 투입 하다 보면 스레드 풀에 남아있는 스레드가 고갈될 수 있기 때문에 이를 주의해야 한다. 이 문제를 해결하기 위해 오래 걸리는 작업만을 처리하기 위한 별도의 스레드 풀을 사용하는 방식을 사용한다.

나머지 한 가지는, 비동기 작업 내부에서 일어나는 입출력을 되도록 멈추지 않고 기다리는 방식(Non-Blocking I/O)으로 처리해야 하는 점이다. 입출력을 처리할 때 되도록 멈추지 않고 기다리기 방식을 사용해야 하는 이유는 앞서 다룬 작업이 오래 걸리는 것을 피하고, 자원을 효율적으로 사용하기 위함이다. 입출력 결과를 기다리기 위해 멈추지 않고 기다리기를 사용한다면 일꾼은 스레드 풀로 다시 돌아가서 다른 일을 할 수 있다. 이로 인해 기다리기로 인해 작업이 길어지는 것을 피할 수 있고, 기다리는 동안 다른 일을 수행하여 자원을 효율적으로 사용할 수 게 된다. 다만 앞서 논의한 바와 같이 멈추지 않고 기다리는 방식은 결과를 되돌려 받기 위해 약속을 정하고 이를 위해 추가 구현이 필요하다는 단점이 있다. 하지만 멈추고 기다리는 방식(Blocking I/O)을 사용할 경우, 입출력 결과를 기다리는 동안 스레드를 점유하고 있어야 하기 때문에 첫 번째 문제와 같은 이유로 스레드가 고갈될 가능성이 생긴다. 따라서 스레드 풀을 사용하는 방식으로 비동기 프로그래밍을 지원하고 있다면 오래 걸리는 입출력 작업에 대해 되도록 멈추지 않고 기다리는 방식으로 처리하도록 해야한다.

웹 프레임워크에서 지원하는 비동기 프로그래밍은 대부분 언어를 따라 스레드 풀을 구현체로 사용하기 때문에 [10, 11, 12, 14, 13], 위와 마찬가지로 두 가지 요소를 주의해서 사용해야 한다. 즉, 이는 비동기 프로그래밍 방식으로 요청을 처리하기 시작했다면 처리 도중 발생하는 입출력 작업에 대해 되도록이면 멈추지 않고 기다리는 방식으로 수행해야 하는 것을 뜻한다. 처리 로직에서 요청을 처리하다 보면 필요에 따라 다른 웹 서비스와 요청을 주고받기, 데이터베이스와 데이터를 주고받기, 파일 시스템 접근 등 다양한 외부 입출력을 사용하게 된다. 요청을 비동기 프로그래밍 방식을 이용하여 처리하기로 했다면, 앞서 이야기 나눈 대로 되도록이면 멈추지 않고 기다리는 비동기 HTTP 클라이언트, 멈추지 않고 기다리는 데이터베이스 접근, 멈추지 않고 기다리는 파일 시스템 접근 등을 사용해야 한다. 멈추지 않고 기다리는 외부 입출력을 사용하다 보면 결과 처리 로직의 구현이 복잡해 지기 마련인데, 이를 극복하기 위해 더 높은 추상화 수준의 프로그래밍 모델을 사용하기도 하며, 다음 주제에서는 이런 프로그래밍 모델들에 대해 다룰 것이다.

3 관련 있는 작업들, 앞으로의 웹 프레임워크

비동기 프로그래밍(Asynchronous Programming)을 지원하기 이전의 웹 프레임워크 구현체는 추상화 된 공간이라고 이야기 하기 민망할 정도로 맨 바닥 위에 구현한 프로그램이었다. 최근 들어서는 비동기 프로그래밍을 필두로 다양한 프로그래밍 모델과 계산 모델을 도입하여 웹 프레임워크를 이루는 근간의 추상화 수준을 점차 높여나가고 있다.

절차 중심의 프로그래밍(Imperative Programming) 그리고 객체 중심의 프로그래밍(Object-Oriented Programming)은 오랜 기간동안 다양한 분야에서 사용해 왔지만, 추상화 수준을 높여나가는 일에는 근본적인 한계가 있다. 절차 중심의 프로그래밍은 비동기 프로그래밍 같이 실행 흐름을 복잡하게 제어하거나, 여러 일꾼을 통해 일을 동시에 처리하는 방식의 계산에는 적합하지 않다. 객체 중심의 프로그래밍 역시도 객체의 상태와 맥락이 객체의 값을 결정하기 때문에, 실행 흐름이 복잡해 지는 경우 객체가 가진 값을 유추하기 쉽지 않다. 또한 절차 중심의 프로그래밍의 큰 문제 중 하나는 비동기 프로그래밍을 절차 중심의 프로그래밍의 관점으로 사용하는 경우 사용자의 역량에 따라 자원을 사용하는 효율이 달라진다는 것이다. 위의 두 방식의 프로그래밍만을 가지고 비동기 프로그래밍을 더 직관적이고 더 효율적으로 동작하도록 구현하려면 엄청난 노력을 들여야 한다.

3.1 더 높은 추상화 수준을 제공하고자 하는 프로그래밍 모델, 계산 모델

이미 오래 전부터 병렬성 및 동시성을 높은 수준의 추상화를 통해 다룰 수 있도록 엄밀한 정의를 바탕으로 프로그래밍 모델 또는 계산 모델을 연구해 왔다. 대표적으로 파이 계산법(π calculus)[6], 액터 모델(Actor Model)[8], 반응하는 프로그래밍(Reactive Programming)[2, 3] 등이 있다. 파이 계산법의 경우 람다 계산법(λ calculus)과 대등한 층위에 있는 계산 모델로, 여러 일꾼들이 일을 진행하고 데이터를 주고받는 작업들에 대해 엄밀하게 정의한 계산 모델이다[6]. 이를 사용하기 좋은 형태로 재가공한 조인 계산법(join-calculus)등이 있으며[7], 파이 계산법과 매우 유사한 형태의 언어인 Erlang[9] 역시 이를 위한 계산 모델 위에서 프로그램을 실행한다. Erlang의 시카고 보스(Chicago Boss)[23], 액터 모델의 구현체인 아카(Akka)를 이용하는 플레이 프레임워크(Play Framework)[12] 등 병렬성과 동시성을 계산하기 위한 계산 모델에 기반한 웹 프레임워크들이 있다. 플레이 프레임워크는 링크드인(LinkedIn)에서 적극 활용하는 것으로 유명하다[19]. 반응하는 프로그래밍의 경우 웹 프레임워크의 형태는 아니지만 서버를 구현을 위해 넷플릭스(Netflix)에서 도입했으며, 스레드 안정성 및 데이터 동기화에 대한 걱정 없이 서버의 동시성을 높이는데 효과를 보았다고 이야기한다[25]. 또한 스프링 프레임워크(Spring Framework)[14]의 최신버전에서 반응하는 프로그래밍을 도입하고 있으며[20, 21], 플레이 프레임워크가 사용하는 하부 프레임워크인 아카 에서도 반응하는 프로그래밍을 도입하고 있다[24].

값 중심 프로그래밍(Value-Orient Programming)과 함수 중심 프로그래밍(Functional Programming)은 높은 수준의 추상화를 통해 프로그램을 작성하고 정확하게 이해 하는데에 도움을 주며, 위와 마찬가지로 웹 프레임워크에 많은 영향을 끼쳐왔다. 앞서 언급한 바와 같이 비동기 프로그래밍(Asynchronous Programming)의 흐름을 손쉽게 다루는 데에 값 중심의 프로그래밍과 함수 중심의 프로그래밍은 매우 중요한 역할을 한다. 값 중심 그리고 함수 중심으로 비동기 프로그래밍의 흐름을 다루면, 실행 흐름의 복잡한 논리를 함수를 조립하는 방식을 통해 절차 중심 또는 객체 중심의 방식에 비해 훨씬 직관적으로 구현할 수 있다. 특히 함수 중심의 프로그래밍은 프로그램을 실행 흐름 관점이 아닌, 함수 관점으로 구조적으로 짜올려서 추상화해 나가는 방식이기 때문이다[5]. 또한 엄밀하게 정의한 함수 중심의 언어를 사용하는 경우 그것들의 단단한 타입 시스템과 타입 시스템이 제공하는 모나드(Monad)를 활용하여 실행 흐

름을 손쉽게 다룰 수 있다. 즉, 비동기 프로그래밍을 오로지 함수들의 조합과 타입 시스템을 통해 표현할 수 있다. 반면 절차 중심 프로그래밍(Imperative Programming)과 객체 중심의 프로그래밍(Object-Orient Programming)을 사용하는 경우 계속해서 실행 흐름을 따라가고 복잡한 로직을 다루는 핵심 기능과 상관없는 불필요한 주변 코드들을 만들어야 하는 작업을 반복해야 한다. 이외에도 함수 중심의 프로그래밍 언어는 오래전부터 무한한 데이터 타입이라는 개념을, 늦게 계산하는 기법(Lazy Evaluation)을 통해 흐름(Stream)이라는 개념으로 만들어 사용하는 등[1], 최근 각광받는 다양한 기술들의 뿌리 역할을 하고 있다. 웹 프레임워크에서 값 중심, 함수 중심의 프로그래밍을 중요하게 생각하고 있는 것의 단적인 예로는, 함수 중심의 언어에 기반한 웹 프레임워크인 플레이 프레임워크[12]만 보아도 알 수 있으며, 함수 중심의 프로그래밍 기법을 도입하고 있는 스프링 프레임워크[22] 역시 좋은 예이다.

3.2 반응하는 프로그래밍(Reactive Programming)

이 중에서도 특히 반응하는 프로그래밍(Reactive Programming)은 많은 자원을 효율적으로 사용하는 웹 프레임워크의 진화에 있어서 가장 중요하고 영향력 있는 재료이다. 거의 모든 웹 프레임워크가 비동기 프로그래밍을 도입하듯이 앞으로 반응하는 프로그래밍을 도입할 것으로 예상하고 있으며, 앞서 언급한 바와 같이 이미 주요 웹 프레임워크들은 이미 반응하는 프로그래밍을 적극 도입하고 있다[24, 20, 21]. 비동기 프로그래밍(Asynchronous Programming)은 실행 결과를 가져오는 시점을 실행 흐름으로부터 분리해 내는 작업이라고 볼 수 있다. 반응하는 프로그래밍은 비동기 프로그래밍에서 하나씩 다루던 것을 한 번에 여러 개 씩 다룰 수 있도록, 프로그래밍의 관점을 실행 흐름에서 데이터의 흐름으로 옮기는 작업이다[4]. 다시 말하자면, 반응하는 프로그래밍 기법은 프로그래밍의 관점을 기존의 절차와 논리 그리고 실행 흐름에서부터 데이터 흐름 중심으로 옮기려는 시도이다. 반응하는 프로그래밍에서 모든 데이터는 절차와 로직을 통해 옮겨 다니는 것이 아니라, 데이터의 흐름을 통해서만 옮겨 다닌다. 데이터를 생성하면 생성한 데이터는 데이터의 흐름을 만들거나 또는 기존의 데이터의 흐름에 흘러 들어가게 되고, 데이터를 사용하고자 하면 데이터의 흐름을 구독하는 일을 통해서 해야 한다. 데이터를 만들고 사용하는 일련의 과정을 데이터의 흐름이라는 개념으로 뚝 떼어서 완전 분리했기 때문에 각 작업은 비동기 방식으로 수행하면서 동시에 서로의 실행 흐름에 관여할 필요가 전혀 없어진다.

반응하는 프로그래밍의 목표는 프로그래밍의 관점을 데이터 흐름으로 옮겨 비동기 프로그래밍과 이벤트 기반 프로그래밍(Event-Base Programming)들을 쉽게 조립하여 쓸 수 있도록 하는 것이다[3]. 데이터 흐름을 기준으로 해야 할 작업을 조각조각 나누어 수행하는 것이 가능하기 때문이다. 그 이유는 이전에 계산해온 역사와 맥락에 상관없이 데이터 흐름을 기준으로 필요한 일을 수행하도록 작성하고, 이후에 해야 할 일을 적당한 데이터 흐름에 던져주는 식으로 프로그램을 작성하도록 하기 때문이다. 프로그램을 작성함에 있어 중요하게 관심을 두고 작업하는 대상이 프로그램의 실행 흐름이 아니라 데이터의 흐름들을 어떻게 조작 하느냐로 바뀌게 된다. 비동기 프로그래밍을 위해 복잡한 실행 흐름 제어를 신경 써야 했던 기존의 프로그래밍 방식과 달리, 반응하는 프로그래밍을 이용할 경우 비동기 프로그래밍 코드 조각들의 조립만으로 프로그래밍을 할 수 있다. 즉 비동기 프로그래밍을 재료로 취급할 수 있도록 높은 추상화 수준을 제공하여 사용자에게 이를 노출하지 않고 자연스럽게 사용할 수 있도록 하는 기법이라고 이야기할 수 있다. 이는 웹 프레임워크의 현재 방향 — 비동기 프로그래밍의 도입과 적극 사용 — 과 잘 맞아 떨어지며, 단순히 비동기 프로그래밍만 가져다 쓰는 것이 아닌, 이에 대한 더 높은 추상화를 제공하고자 할 때 필히 마주치게 된다. 적어도 지금까지는 반응하는 프로그래밍을 도입하여 추상화 수준을 높이려는 움직임을 보이고 있다[24, 20, 21].

4 결론

이 글을 통해 비동기 프로그래밍을 사용하는 웹 프레임워크가 정확히 무엇을 의미하는지 살펴보고, 그 구현체를 들여다 보았으며 앞으로 어떤 방향으로 발전해 나갈 것인지 살펴 보았다. 웹 서비스는 현대 사회에서 사용자와 밀접하게 맞닿아 있는 중요한 요소로, 이에 따라 웹 프레임워크의 성능과 생산성은 아주 중요한 이슈로 다루고 있다. 따라서 웹 프레임워크의 진화 중 가장 중요한 요소 중 하나인 비동기 프로그래밍에 대해 살펴보고 요소들을 하나씩 들여다 보는 것은 큰 의미가 있다고 생각한다. 더 나아가 웹 프레임워크가 어떤 방향으로 발전하고 있는지, 웹 프레임워크 커뮤니티에서 성능과 추상화 수준을 높이기 위해 시도하고 시도들이 어떤 것이 있는지, 그 중에서 중요한 추상적 개념은 무엇이 있는지 살펴보았다. 이 글을 통해 웹 프레임워크에 대한 이해도를 높이는 데에 도움이 되었기를 바라며, 동시에 웹 프레임워크를 진화 시키기 위해 하고있는 다양한 시도들에 적극 참여하게 되는 계기가 되었으면 한다.

References

[1] Hinze, R., 2010. Reasoning about codata. *In Central European Functional Programming School* (pp. 42-93). Springer Berlin Heidelberg.

[2] Wan, Z. and Hudak, P., 2000, August. Functional reactive programming from first principles. In *Acm sigplan notices* (Vol. 35, No. 5, pp. 242-252). ACM.

[3] Meijer, E., 2012. Your mouse is a database. *Queue*, 10(3), p.20.

[4] Meijer, E., Millikin, K. and Bracha, G., 2015. Spicing up dart with side effects. *Queue*, 13(3), p.40.

[5] Hughes, J., 1989. Why functional programming matters. *The computer journal*, 32(2), pp.98-107.

[6] Milner, Robin. *Communicating and mobile systems: the pi calculus.* Cambridge university press, 1999.

[7] Fournet, C. and Gonthier, G., 2002. The join calculus: a language for distributed mobile programming. In *Applied Semantics* (pp. 268-332). Springer Berlin Heidelberg.

[8] Hewitt, C., Bishop, P. and Steiger, R., 1973, August. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence* (pp. 235-245). Morgan Kaufmann Publishers Inc.

[9] Armstrong, J., Virding, R., Wikström, C. and Williams, M., 1993. Concurrent programming in ERLANG.

[10] Netty. 2016. *Netty.* [ONLINE] Available at: `http://netty.io` [Accessed 4 October 2016].

[11] Apache Tomcat. 2016. *Advanced IO and Tomcat.* [ONLINE] Available at: `https://tomcat.apache.org/tomcat-8.0-doc/aio.html` [Accessed 4 October 2016].

[12] Play. 2016. *Handling asynchronous results.* [ONLINE] Available at: `https://www.playframework.com/documentation/2.5.x/ScalaAsync` [Accessed 4 October 2016].

[13] Tornado. 2016. *Tornado Web Server.* [ONLINE] Available at: http://www.tornadoweb.org/en/stable/ [Accessed 4 October 2016].

[14] Spring. 2016. *Web MVC framework.* [ONLINE] Available at: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[15] Max Bolingbroke. 2016. *Asynchronous and non-blocking IO* [ONLINE] Available at: http://blog.omega-prime.co.uk/?p=155 [Accessed 4 October 2016].

[16] Oracle. 2016. *Class ThreadPoolExecutor.* [ONLINE] Available at: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html [Accessed 4 October 2016].

[17] Scala. 2016. *Futures and Promises.* [ONLINE] Available at: http://docs.scala-lang.org/overviews/core/futures.html [Accessed 4 October 2016].

[18] Python. 2016. *Launching parallel tasks.* [ONLINE] Available at: https://docs.python.org/3/library/concurrent.futures.html [Accessed 4 October 2016].

[19] LinkedIn. 2013. *Play Framework: async I/O without the thread pool and callback hell.* [ONLINE] Available at: https://engineering.linkedin.com/play/play-framework-async-io-without-thread-pool-and-callback-hell [Accessed 4 October 2016].

[20] Spring. 2016. *Notes on Reactive Programming Part I: The Reactive Landscape.* [ONLINE] Available at: https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape [Accessed 4 October 2016].

[21] Spring. 2016. *Reactive Programming with Spring 5.0 M1.* [ONLINE] Available at: https://spring.io/blog/2016/07/28/reactive-programming-with-spring-5-0-m1 [Accessed 4 October 2016].

[22] Spring. 2016. *New in Spring5: Functional Web Framework.* [ONLINE] Available at: https://spring.io/blog/2016/09/22/new-in-spring-5-functional-web-framework [Accessed 4 October 2016].

[23] Chicago Boss. 2016. *Build your next website with Erlang.* [ONLINE] Available at: http://chicagoboss.org/ [Accessed 4 October 2016].

[24] Akka. 2016. *Akka Streams Kafka 0.11.* http://blog.akka.io/integrations/2016/09/10/akka-stream-kafka?_ga=1.155764729.396717306.1475666738 [Accessed 4 October 2016].

[25] Netflix. 2013. *Reactive Programming in the Netflix API with RxJava* http://techblog.netflix.com/2013/02/rxjava-netflix-api.html [Accessed 4 October 2016].