PostgreSQL

1. 장점

표준 SQL을 준수합니다. SQLite 또는 MySQL보다 PostgreSql은 표준에 좀 더 가깝게 구현하는 것을 목표로 하고 있습니다. 공식 PostgreSql 문서에 따르면 PostgreSql은 전체 핵심 SQL:2011 규정에 필요한 179개의 기능 중 160개를 지원하며 긴 목록의 선택적 기능도 지원합니다.

오픈소스 및 커뮤니티가 이끄는 데이터베이스입니다. 완전한 오픈소스 프로젝트인 PostgreSql의 소스코드는 대규모 헌신적인 커뮤니티에서 개발되었습니다. Postgres 커뮤니티는 DBMS로 작업하는 방법을 설명하는 공식문서, 위치, 온라인 포럼을 포함한 수많은 리소스를 유지 관리하고 기여합니다.

확장성이 뛰어납니다. 사용자는 카탈로그 기반 작업과 동적 로드 사용을 통해 PostgreSQL을 프로그래밍 방식으로 즉시 확장할 수 있습니다. 공유 라이브러리와 같은 객체 코드 파일을 지정할 수 있고 PostgreSQL은 필요에 따라 이를 로드합니다.

1. 단점

메모리 성능이 떨어집니다. 모든 새로운 클라이언트 연결에 대해 PostgresSQL은 새로운 프로세스를 포크(fork)합니다. 각각의 새로운 프로세스에는 약 10MB의 메모리가 할당되므로 많은 연결이 있는 경우 메모리가 빠르게 증가합니다. 따라서 읽기가 많은 간단한 작업의 경우 PostgreSQL은 일반적으로 MySQL과 같은 다른 RDBMS보다 성능이 떨어집니다.

인기도가 떨어집니다. 최근 몇 년 동안 더 널리 사용되고 있지만 PostgreSQL은 역사적으로 보았을 때 인기 측면에서 MySQL에 뒤쳐집니다. 그 결과 PostgreSQL 데이터베이스를 관리하는데 도움이 되는 타사 도구가 여전히 적습니다. 마찬가지로 MySQL은 경험이 있는 사람이 많은 것에 비해 Postgre 데이터베이스 관리 경험이 있는 사람들은 많지 않습니다.

**객체 지향 프로그래밍(OOP, Object-Oriented Programming)**

객체 지향 프로그래밍에서는 모든 데이터를 객체(object)로 취급하며, 이러한 객체가 바로 프로그래밍의 중심이 됩니다.

객체(object)란 간단히 이야기하자면 실생활에서 우리가 인식할 수 있는 사물로 설명할 수 있습니다.

이러한 객체의 상태(state)와 행동(behavior)을 구체화하는 형태의 프로그래밍이 바로 객체 지향 프로그래밍입니다.

이때 객체를 만들어 내기 위한 설계도와 같은 개념을 클래스(class)라고 합니다.

객체 지향 프로그래밍(OOP)이 뭐에요?

객체 지향 프로그래밍은 컴퓨터 프로그래밍 패러다임 중 하나로, 프로그래밍에서 필요한 데이터를 추상화시켜 상태와 행위를 가진 객체를 만들고 그 객체들 간의 유기적인 상호작용을 통해 로직을 구성하는 프로그래밍 방법이다.

**클래스(class)**

자바에서 클래스(class)란 객체를 정의하는 틀 또는 설계도와 같은 의미로 사용됩니다.

자바에서는 이러한 설계도인 클래스를 가지고, 여러 객체를 생성하여 사용하게 됩니다.

클래스는 객체의 상태를 나타내는 필드(field)와 객체의 행동을 나타내는 메소드(method)로 구성됩니다.

즉, 필드(field)란 클래스에 포함된 변수(variable)를 의미합니다.

또한, 메소드(method)란 어떠한 특정 작업을 수행하기 위한 명령문의 집합이라 할 수 있습니다.

**인스턴스(instance)**

자바에서 클래스를 사용하기 위해서는 우선 해당 클래스 타입의 객체(object)를 선언해야 합니다.

이렇게 클래스로부터 객체를 선언하는 과정을 클래스의 인스턴스 화라고 합니다.

또한, 이렇게 선언된 해당 클래스 타입의 객체를 인스턴스(instance)라고 합니다.

즉, 인스턴스란 메모리에 할당된 객체를 의미합니다.

자바에서는 하나의 클래스로부터 여러 개의 인스턴스를 생성할 수 있습니다.

이렇게 생성된 인스턴스는 독립된 메모리 공간에 저장된 자신만의 필드를 가질 수 있습니다.

하지만 해당 클래스의 모든 메소드(method)는 해당 클래스에서 생성된 모든 인스턴스가 공유하게 됩니다.

**메소드(method)**

자바에서 클래스는 멤버(member)로 속성을 표현하는 필드(field)와 기능을 표현하는 메소드(method)를 가집니다.

그중에서 메소드(method)란 어떠한 특정 작업을 수행하기 위한 명령문의 집합이라 할 수 있습니다.

**abstract 제어자**

자바에서 abstract 제어자는 '추상적인'이라는 의미로 사용됩니다.

선언부만 있고 구현부가 없는 메소드를 추상 메소드라 하며, 반드시 abstract 제어자를 붙여야 합니다.

또한, 하나 이상의 추상 메소드가 포함하고 있는 추상 클래스도 반드시 abstract 제어자를 붙여야 합니다.

자바에서 abstract 제어자를 사용할 수 있는 대상은 다음과 같습니다.

- 클래스, 메소드

**상속(inheritance)이란?**

상속(inheritance)이란 기존의 클래스에 기능을 추가하거나 재정의하여 새로운 클래스를 정의하는 것을 의미합니다.

이러한 상속은 캡슐화, 추상화와 더불어 객체 지향 프로그래밍을 구성하는 중요한 특징 중 하나입니다.

상속을 이용하면 기존에 정의되어 있는 클래스의 모든 필드와 메소드를 물려받아, 새로운 클래스를 생성할 수 있습니다.

이때 기존에 정의되어 있던 클래스를 부모 클래스(parent class) 또는 상위 클래스(super class), 기초 클래스(base class)라고도합니다.

그리고 상속을 통해 새롭게 작성되는 클래스를 자식 클래스(child class) 또는 하위 클래스(sub class), 파생 클래스(derived class)라고도 합니다.

**super 키워드**

super 키워드는 부모 클래스로부터 상속받은 필드나 메소드를 자식 클래스에서 참조하는 데 사용하는 참조 변수입니다.

인스턴스 변수의 이름과 지역 변수의 이름이 같을 경우 인스턴스 변수 앞에 this 키워드를 사용하여 구분할 수 있었습니다.

이와 마찬가지로 부모 클래스의 멤버와 자식 클래스의 멤버 이름이 같을 경우 super 키워드를 사용하여 구별할 수 있습니다.

이렇게 자바에서는 super 참조 변수를 사용하여 부모 클래스의 멤버에 접근할 수 있습니다.

this와 마찬가지로 super 참조 변수를 사용할 수 있는 대상도 인스턴스 메소드뿐이며, 클래스 메소드에서는 사용할 수 없습니다.

**super() 메소드**

this() 메소드가 같은 클래스의 다른 생성자를 호출할 때 사용된다면, super() 메소드는 부모 클래스의 생성자를 호출할 때 사용됩니다.

자식 클래스의 인스턴스를 생성하면, 해당 인스턴스에는 자식 클래스의 고유 멤버뿐만 아니라 부모 클래스의 모든 멤버까지도 포함되어 있습니다.

따라서 부모 클래스의 멤버를 초기화하기 위해서는 자식 클래스의 생성자에서 부모 클래스의 생성자까지 호출해야만 합니다.

이러한 부모 클래스의 생성자 호출은 모든 클래스의 부모 클래스인 Object 클래스의 생성자까지 계속 거슬러 올라가며 수행됩니다.

따라서 자바 컴파일러는 부모 클래스의 생성자를 명시적으로 호출하지 않는 모든 자식 클래스의 생성자 첫 줄에 자동으로 다음과 같은 명령문을 추가하여, 부모 클래스의 멤버를 초기화할 수 있도록 해줍니다.

**메소드 오버라이딩(method overriding)**

앞서 배운 오버로딩(overloading)이란 서로 다른 시그니처를 갖는 여러 메소드를 하나의 이름으로 정의하는 것이었습니다.

오버라이딩(overriding)이란 상속 관계에 있는 부모 클래스에서 이미 정의된 메소드를 자식 클래스에서 같은 시그니쳐를 갖는 메소드로 다시 정의하는 것이라고 할 수 있습니다.

자바에서 자식 클래스는 부모 클래스의 private 멤버를 제외한 모든 메소드를 상속받습니다.

이렇게 상속받은 메소드는 그대로 사용해도 되고, 필요한 동작을 위해 재정의하여 사용할 수도 있습니다.

즉, 메소드 오버라이딩이란 상속받은 부모 클래스의 메소드를 재정의하여 사용하는 것을 의미합니다.

**다형성(polymorphism)이란?**

다형성(polymorphism)이란 하나의 객체가 여러 가지 타입을 가질 수 있는 것을 의미합니다.

자바에서는 이러한 다형성을 부모 클래스 타입의 참조 변수로 자식 클래스 타입의 인스턴스를 참조할 수 있도록 하여 구현하고 있습니다.

다형성은 상속, 추상화와 더불어 객체 지향 프로그래밍을 구성하는 중요한 특징 중 하나입니다.

**추상 메소드(abstract method)**

추상 메소드(abstract method)란 자식 클래스에서 반드시 오버라이딩해야만 사용할 수 있는 메소드를 의미합니다.

자바에서 추상 메소드를 선언하여 사용하는 목적은 추상 메소드가 포함된 클래스를 상속받는 자식 클래스가 반드시 추상 메소드를 구현하도록 하기 위함입니다.

예를 들면 모듈처럼 중복되는 부분이나 공통적인 부분은 미리 다 만들어진 것을 사용하고, 이를 받아 사용하는 쪽에서는 자신에게 필요한 부분만을 재정의하여 사용함으로써 생산성이 향상되고 배포 등이 쉬워지기 때문입니다.

이러한 추상 메소드는 선언부만이 존재하며, 구현부는 작성되어 있지 않습니다.

바로 이 작성되어 있지 않은 구현부를 자식 클래스에서 오버라이딩하여 사용하는 것입니다.

**추상 클래스(abstract class)**

자바에서는 하나 이상의 추상 메소드를 포함하는 클래스를 가리켜 추상 클래스(abstract class)라고 합니다.

이러한 추상 클래스는 객체 지향 프로그래밍에서 중요한 특징인 다형성을 가지는 메소드의 집합을 정의할 수 있도록 해줍니다.

즉, 반드시 사용되어야 하는 메소드를 추상 클래스에 추상 메소드로 선언해 놓으면, 이 클래스를 상속받는 모든 클래스에서는 이 추상 메소드를 반드시 재정의해야 합니다.

**인터페이스(interface)란?**

자식 클래스가 여러 부모 클래스를 상속받을 수 있다면, 다양한 동작을 수행할 수 있다는 장점을 가지게 될 것입니다.

하지만 클래스를 이용하여 다중 상속을 할 경우 메소드 출처의 모호성 등 여러 가지 문제가 발생할 수 있어 자바에서는 클래스를 통한 다중 상속은 지원하지 않습니다.

하지만 다중 상속의 이점을 버릴 수는 없기에 자바에서는 인터페이스라는 것을 통해 다중 상속을 지원하고 있습니다.

인터페이스(interface)란 다른 클래스를 작성할 때 기본이 되는 틀을 제공하면서, 다른 클래스 사이의 중간 매개 역할까지 담당하는 일종의 추상 클래스를 의미합니다.

자바에서 추상 클래스는 추상 메소드뿐만 아니라 생성자, 필드, 일반 메소드도 포함할 수 있습니다.

하지만 인터페이스(interface)는 오로지 추상 메소드와 상수만을 포함할 수 있습니다.

**내부 클래스(inner class)**

내부 클래스(inner class)란 하나의 클래스 내부에 선언된 또 다른 클래스를 의미합니다.

이러한 내부 클래스는 외부 클래스(outer class)에 대해 두 개의 클래스가 서로 긴밀한 관계를 맺고 있을 때 선언할 수 있습니다.

**제네릭(generic)이란?**

자바에서 제네릭(generic)이란 데이터의 타입(data type)을 일반화한다(generalize)는 것을 의미합니다.

제네릭은 클래스나 메소드에서 사용할 내부 데이터 타입을 컴파일 시에 미리 지정하는 방법입니다.

이렇게 컴파일 시에 미리 타입 검사(type check)를 수행하면 다음과 같은 장점을 가집니다.

1. 클래스나 메소드 내부에서 사용되는 객체의 타입 안정성을 높일 수 있습니다

2. 반환값에 대한 타입 변환 및 타입 검사에 들어가는 노력을 줄일 수 있습니다.

JDK 1.5 이전에서는 여러 타입을 사용하는 대부분의 클래스나 메소드에서 인수나 반환값으로 Object 타입을 사용했습니다.

하지만 이 경우에는 반환된 Object 객체를 다시 원하는 타입으로 타입 변환해야 하며, 이때 오류가 발생할 가능성도 존재합니다.

하지만 JDK 1.5부터 도입된 제네릭을 사용하면 컴파일 시에 미리 타입이 정해지므로, 타입 검사나 타입 변환과 같은 번거로운 작업을 생략할 수 있게 됩니다.

**컬렉션 프레임워크(collection framework)란?**

자바에서 컬렉션 프레임워크(collection framework)란 다수의 데이터를 쉽고 효과적으로 처리할 수 있는 표준화된 방법을 제공하는 클래스의 집합을 의미합니다

즉, 데이터를 저장하는 자료 구조와 데이터를 처리하는 알고리즘을 구조화하여 클래스로 구현해 놓은 것입니다.

이러한 컬렉션 프레임워크는 자바의 인터페이스(interface)를 사용하여 구현됩니다.

**List 컬렉션 클래스**

List 인터페이스를 구현한 모든 List 컬렉션 클래스는 다음과 같은 특징을 가집니다.

1. 요소의 저장 순서가 유지됩니다.

2. 같은 요소의 중복 저장을 허용합니다.

대표적인 List 컬렉션 클래스에 속하는 클래스는 다음과 같습니다.

1. ArrayList<E>

2. LinkedList<E>

3. Vector<E>

4. Stack<E>

**ArrayList<E> 클래스**

ArrayList 클래스는 가장 많이 사용되는 컬렉션 클래스 중 하나입니다.

JDK 1.2부터 제공된 ArrayList 클래스는 내부적으로 배열을 이용하여 요소를 저장합니다.

ArrayList 클래스는 배열을 이용하기 때문에 인덱스를 이용해 배열 요소에 빠르게 접근할 수 있습니다.

하지만 배열은 크기를 변경할 수 없는 인스턴스이므로, 크기를 늘리기 위해서는 새로운 배열을 생성하고 기존의 요소들을 옮겨야 하는 복잡한 과정을 거쳐야 합니다.

물론 이 과정은 자동으로 수행되지만, 요소의 추가 및 삭제 작업에 걸리는 시간이 매우 길어지는 단점을 가지게 됩니다.

**LinkedList<E> 클래스**

LinkedList 클래스는 ArrayList 클래스가 배열을 이용하여 요소를 저장함으로써 발생하는 단점을 극복하기 위해 고안되었습니다.

JDK 1.2부터 제공된 LinkedList 클래스는 내부적으로 연결 리스트(linked list)를 이용하여 요소를 저장합니다.

배열은 저장된 요소가 순차적으로 저장됩니다.

하지만 연결 리스트는 저장된 요소가 비순차적으로 분포되며, 이러한 요소들 사이를 링크(link)로 연결하여 구성합니다.

다음 요소를 가리키는 참조만을 가지는 연결 리스트를 단일 연결 리스트(singly linked list)라고 합니다.

**Vector<E> 클래스**

Vector 클래스는 JDK 1.0부터 사용해 온 ArrayList 클래스와 같은 동작을 수행하는 클래스입니다.

현재의 Vector 클래스는 ArrayList 클래스와 마찬가지로 List 인터페이스를 상속받습니다.

따라서 Vector 클래스에서 사용할 수 있는 메소드는 ArrayList 클래스에서 사용할 수 있는 메소드와 거의 같습니다.

하지만 현재에는 기존 코드와의 호환성을 위해서만 남아있으므로, Vector 클래스보다는 ArrayList 클래스를 사용하는 것이 좋습니다.

**Stack<E> 클래스**

Stack 클래스는 List 컬렉션 클래스의 Vector 클래스를 상속받아, 전형적인 스택 메모리 구조의 클래스를 제공합니다.

스택 메모리 구조는 선형 메모리 공간에 데이터를 저장하면서 후입선출(LIFO)의 시멘틱을 따르는 자료 구조입니다.

즉, 가장 나중에 저장된(push) 데이터가 가장 먼저 인출(pop)되는 구조입니다.

**Queue<E> 인터페이스**

클래스로 구현된 스택과는 달리 자바에서 큐 메모리 구조는 별도의 인터페이스 형태로 제공됩니다.

이러한 Queue 인터페이스를 상속받는 하위 인터페이스는 다음과 같습니다.

1. Deque<E>

2. BlockingDeque<E>

3. BlockingQueue<E>

4. TransferQueue<E>

따라서 Queue 인터페이스를 직간접적으로 구현한 클래스는 상당히 많습니다.

그중에서도 Deque 인터페이스를 구현한 LinkedList 클래스가 큐 메모리 구조를 구현하는 데 가장 많이 사용됩니다.

큐 메모리 구조는 선형 메모리 공간에 데이터를 저장하면서 선입선출(FIFO)의 시멘틱을 따르는 자료 구조입니다.

즉, 가장 먼저 저장된(push) 데이터가 가장 먼저 인출(pop)되는 구조입니다.

**Set 컬렉션 클래스**

Set 인터페이스를 구현한 모든 Set 컬렉션 클래스는 다음과 같은 특징을 가집니다.

1. 요소의 저장 순서를 유지하지 않습니다.

2. 같은 요소의 중복 저장을 허용하지 않습니다.

대표적인 Set 컬렉션 클래스에 속하는 클래스는 다음과 같습니다.

1. HashSet<E>

2. TreeSet<E>

**HashSet<E> 클래스**

HashSet 클래스는 Set 컬렉션 클래스에서 가장 많이 사용되는 클래스 중 하나입니다.

JDK 1.2부터 제공된 HashSet 클래스는 해시 알고리즘(hash algorithm)을 사용하여 검색 속도가 매우 빠릅니다.

이러한 HashSet 클래스는 내부적으로 HashMap 인스턴스를 이용하여 요소를 저장합니다.

HashSet 클래스는 Set 인터페이스를 구현하므로, 요소를 순서에 상관없이 저장하고 중복된 값은 저장하지 않습니다.

만약 요소의 저장 순서를 유지해야 한다면 JDK 1.4부터 제공하는 LinkedHashSet 클래스를 사용하면 됩니다.

**TreeSet<E> 클래스**

TreeSet 클래스는 데이터가 정렬된 상태로 저장되는 이진 검색 트리(binary search tree)의 형태로 요소를 저장합니다.

이진 검색 트리는 데이터를 추가하거나 제거하는 등의 기본 동작 시간이 매우 빠릅니다.

JDK 1.2부터 제공되는 TreeSet 클래스는 NavigableSet 인터페이스를 기존의 이진 검색 트리의 성능을 향상시킨 레드-블랙 트리(Red-Black tree)로 구현합니다.

TreeSet 클래스는 Set 인터페이스를 구현하므로, 요소를 순서에 상관없이 저장하고 중복된 값은 저장하지 않습니다.

**Map 컬렉션 클래스**

Map 인터페이스는 Collection 인터페이스와는 다른 저장 방식을 가집니다.

Map 인터페이스를 구현한 Map 컬렉션 클래스들은 키와 값을 하나의 쌍으로 저장하는 방식(key-value 방식)을 사용합니다.

여기서 키(key)란 실질적인 값(value)을 찾기 위한 이름의 역할을 합니다.

Map 인터페이스를 구현한 모든 Map 컬렉션 클래스는 다음과 같은 특징을 가집니다.

1. 요소의 저장 순서를 유지하지 않습니다.

2. 키는 중복을 허용하지 않지만, 값의 중복은 허용합니다.

**HashMap<K, V> 클래스**

HashMap 클래스는 Map 컬렉션 클래스에서 가장 많이 사용되는 클래스 중 하나입니다.

JDK 1.2부터 제공된 HashMap 클래스는 해시 알고리즘(hash algorithm)을 사용하여 검색 속도가 매우 빠릅니다.

HashMap 클래스는 Map 인터페이스를 구현하므로, 중복된 키로는 값을 저장할 수 없습니다.

하지만 같은 값을 다른 키로 저장하는 것은 가능합니다.

**Hashtable<K, V> 클래스**

Hashtable 클래스는 JDK 1.0부터 사용해 온 HashMap 클래스와 같은 동작을 하는 클래스입니다.

현재의 Hashtable 클래스는 HashMap 클래스와 마찬가지로 Map 인터페이스를 상속받습니다.

따라서 Hashtable 클래스에서 사용할 수 있는 메소드는 HashMap 클래스에서 사용할 수 있는 메소드와 거의 같습니다.

하지만 현재에는 기존 코드와의 호환성을 위해서만 남아있으므로, Hashtable 클래스보다는 HashMap 클래스를 사용하는 것이 좋습니다.

**TreeMap<K, V> 클래스**

TreeMap 클래스는 키와 값을 한 쌍으로 하는 데이터를 이진 검색 트리(binary search tree)의 형태로 저장합니다.

이진 검색 트리는 데이터를 추가하거나 제거하는 등의 기본 동작 시간이 매우 빠릅니다.

JDK 1.2부터 제공된 TreeMap 클래스는 NavigableMap 인터페이스를 기존의 이진 검색 트리의 성능을 향상시킨 레드-블랙 트리(Red-Black tree)로 구현합니다.

TreeMap 클래스는 Map 인터페이스를 구현하므로, 중복된 키로는 값을 저장할 수 없습니다.

하지만 같은 값을 다른 키로 저장하는 것은 가능합니다.

**Iterator<E> 인터페이스**

자바의 컬렉션 프레임워크는 컬렉션에 저장된 요소를 읽어오는 방법을 Iterator 인터페이스로 표준화하고 있습니다.

Collection 인터페이스에서는 Iterator 인터페이스를 구현한 클래스의 인스턴스를 반환하는 iterator() 메소드를 정의하여 각 요소에 접근하도록 하고 있습니다.

따라서 Collection 인터페이스를 상속받는 List와 Set 인터페이스에서도 iterator() 메소드를 사용할 수 있습니다.

**Enumeration<E> 인터페이스**

Enumeration 인터페이스는 JDK 1.0부터 사용해 온 Iterator 인터페이스와 같은 동작을 하는 인터페이스입니다.

Enumeration 인터페이스는 hasMoreElements()와 nextElement() 메소드를 사용하여 Iterator와 같은 작업을 수행합니다.

하지만 현재에는 기존 코드와의 호환성을 위해서만 남아있으므로, Enumeration 인터페이스보다는 Iterator 인터페이스를 사용하는 것이 좋습니다.

**ListIterator<E> 인터페이스**

ListIterator 인터페이스는 Iterator 인터페이스를 상속받아 여러 기능을 추가한 인터페이스입니다.

Iterator 인터페이스는 컬렉션의 요소에 접근할 때 한 방향으로만 이동할 수 있습니다.

하지만 JDK 1.2부터 제공된 ListIterator 인터페이스는 컬렉션 요소의 대체, 추가 그리고 인덱스 검색 등을 위한 작업에서 양방향으로 이동하는 것을 지원합니다.

단, ListIterator 인터페이스는 List 인터페이스를 구현한 List 컬렉션 클래스에서만 listIterator() 메소드를 통해 사용할 수 있습니다.

**Comparable<T> 인터페이스**

Comparable 인터페이스는 객체를 정렬하는 데 사용되는 메소드인 compareTo() 메소드를 정의하고 있습니다.

자바에서 같은 타입의 인스턴스를 서로 비교해야만 하는 클래스들은 모두 Comparable 인터페이스를 구현하고 있습니다.

따라서 Boolean을 제외한 래퍼 클래스나 String, Time, Date와 같은 클래스의 인스턴스는 모두 정렬 가능합니다.

이때 기본 정렬 순서는 작은 값에서 큰 값으로 정렬되는 오름차순이 됩니다.

**Comparator<T> 인터페이스**

Comparator 인터페이스는 Comparable 인터페이스와 같이 객체를 정렬하는 데 사용되는 인터페이스입니다.

Comparable 인터페이스를 구현한 클래스는 기본적으로 오름차순으로 정렬됩니다.

반면에 Comparator 인터페이스는 내림차순이나 아니면 다른 기준으로 정렬하고 싶을 때 사용할 수 있습니다.

즉, Comparator 인터페이스를 구현하면 오름차순 이외의 기준으로도 정렬할 수 있게 되는 것입니다.

이때 Comparator 인터페이스를 구현한 클래스에서는 compare() 메소드를 재정의하여 사용하게 됩니다.

**오류(error)와 예외(exception)**

자바 프로그램을 작성할 때 자바 문법에 맞지 않게 코드를 작성하고 컴파일하려고 하면, 자바 컴파일러는 문법 오류(syntax error)를 발생시킵니다.

또한, 자바 문법에는 맞게 작성되었다 하더라도 프로그램이 실행되면서 예상하지 못한 오류가 발생할 수 있습니다.

이렇게 컴퓨터 시스템이 동작하는 도중에 예상하지 못한 사태가 발생하여 실행 중인 프로그램이 영향을 받는 것을 오류(error)와 예외(exception) 두 가지로 구분할 수 있습니다.

오류(error)는 시스템 레벨에서 프로그램에 심각한 문제를 야기하여 실행 중인 프로그램을 종료시킵니다.

이러한 오류는 개발자가 미리 예측하여 처리할 수 없는 것이 대부분이므로, 오류에 대한 처리는 할 수 없습니다.

하지만 예외(exception)는 오류와 마찬가지로 실행 중인 프로그램을 비정상적으로 종료시키지만, 발생할 수 있는 상황을 미리 예측하여 처리할 수 있습니다.

따라서 개발자는 예외 처리(exception handling)를 통해 예외 상황을 처리할 수 있도록 코드의 흐름을 바꿀 필요가 있습니다.

**Exception 클래스**

자바에서 모든 예외의 조상 클래스가 되는 Exception 클래스는 크게 다음과 같이 구분할 수 있습니다.

1. RuntimeException 클래스

2. 그 외의 Exception 클래스의 자식 클래스

RuntimeException 클래스를 상속받는 자식 클래스들은 주로 치명적인 예외 상황을 발생시키지 않는 예외들로 구성됩니다.

따라서 try / catch 문을 사용하기보다는 프로그램을 작성하면서 예외가 발생하지 않도록 주의를 기울이는 편이 좋습니다.

하지만 그 외의 Exception 클래스에 속하는 자식 클래스들은 치명적인 예외 상황을 발생시키므로, 반드시 try / catch 문을 사용하여 예외를 처리해야만 합니다.

따라서 자바 컴파일러는 RuntimeException 클래스 이외의 Exception 클래스의 자식 클래스에 속하는 예외가 발생할 가능성이 있는 구문에는 반드시 예외를 처리하도록 강제하고 있습니다.

만약 이러한 예외가 발생할 가능성이 있는 구문을 예외 처리하지 않았을 때는 컴파일 시 오류를 발생시킵니다.

**Throwable 클래스**

자바에서 Throwable 클래스는 모든 예외의 조상이 되는 Exception 클래스와 모든 오류의 조상이 되는 Error 클래스의 부모 클래스입니다.

Throwable 타입과 이 클래스를 상속받은 서브 타입만이 자바 가상 머신(JVM)이나 throw 키워드에 의해 던져질 수 있습니다.

**스트림(stream)**

자바에서는 파일이나 콘솔의 입출력을 직접 다루지 않고, 스트림(stream)이라는 흐름을 통해 다룹니다.

스트림(stream)이란 실제의 입력이나 출력이 표현된 데이터의 이상화된 흐름을 의미합니다.

즉, 스트림은 운영체제에 의해 생성되는 가상의 연결 고리를 의미하며, 중간 매개자 역할을 합니다.

**입출력 스트림**

스트림은 한 방향으로만 통신할 수 있으므로, 입력과 출력을 동시에 처리할 수는 없습니다.

따라서 스트림은 사용 목적에 따라 입력 스트림과 출력 스트림으로 구분됩니다.

자바에서는 java.io 패키지를 통해 InputStream과 OutputStream 클래스를 별도로 제공하고 있습니다.

즉, 자바에서의 스트림 생성이란 이러한 스트림 클래스 타입의 인스턴스를 생성한다는 의미입니다.

InputStream 클래스에는 read() 메소드가, OutputStream 클래스에는 write() 메소드가 각각 추상 메소드로 포함되어 있습니다.

사용자는 이 두 메소드를 상황에 맞게 적절히 구현해야만 입출력 스트림을 생성하여 사용할 수 있습니다.

**바이트 기반 스트림**

자바에서 스트림은 기본적으로 바이트 단위로 데이터를 전송합니다.

**보조 스트림**

자바에서 제공하는 보조 스트림은 실제로 데이터를 주고받을 수는 없지만, 다른 스트림의 기능을 향상시키거나 새로운 기능을 추가해 주는 스트림입니다.

**문자 기반 스트림**

자바에서 스트림은 기본적으로 바이트 단위로 데이터를 전송합니다.

하지만 자바에서 가장 작은 타입인 char 형이 2바이트이므로, 1바이트씩 전송되는 바이트 기반 스트림으로는 원활한 처리가 힘든 경우가 있습니다.

따라서 자바에서는 바이트 기반 스트림뿐만 아니라 문자 기반의 스트림도 별도로 제공합니다.

이러한 문자 기반 스트림은 기존의 바이트 기반 스트림에서 InputStream을 Reader로, OutputStream을 Writer로 변경하면 사용할 수 있습니다.

**프로세스(process)란?**

프로세스(process)란 단순히 실행 중인 프로그램(program)이라고 할 수 있습니다.

즉, 사용자가 작성한 프로그램이 운영체제에 의해 메모리 공간을 할당받아 실행 중인 것을 말합니다.

이러한 프로세스는 프로그램에 사용되는 데이터와 메모리 등의 자원 그리고 스레드로 구성됩니다.

**스레드(thread)란?**

스레드(thread)란 프로세스(process) 내에서 실제로 작업을 수행하는 주체를 의미합니다.

모든 프로세스에는 한 개 이상의 스레드가 존재하여 작업을 수행합니다.

또한, 두 개 이상의 스레드를 가지는 프로세스를 멀티스레드 프로세스(multi-threaded process)라고 합니다.

**스레드의 생성과 실행**

자바에서 스레드를 생성하는 방법에는 다음과 같이 두 가지 방법이 있습니다.

1. Runnable 인터페이스를 구현하는 방법

2. Thread 클래스를 상속받는 방법

두 방법 모두 스레드를 통해 작업하고 싶은 내용을 run() 메소드에 작성하면 됩니다.

**스레드의 우선순위**

자바에서 각 스레드는 우선순위(priority)에 관한 자신만의 필드를 가지고 있습니다.

이러한 우선순위에 따라 특정 스레드가 더 많은 시간 동안 작업을 할 수 있도록 설정할 수 있습니다.

getPriority()와 setPriority() 메소드를 통해 스레드의 우선순위를 반환하거나 변경할 수 있습니다.

스레드의 우선순위가 가질 수 있는 범위는 1부터 10까지이며, 숫자가 높을수록 우선순위 또한 높아집니다.

하지만 스레드의 우선순위는 비례적인 절댓값이 아닌 어디까지나 상대적인 값일 뿐입니다.

우선순위가 10인 스레드가 우선순위가 1인 스레드보다 10배 더 빨리 수행되는 것이 아닙니다.

단지 우선순위가 10인 스레드는 우선순위가 1인 스레드보다 좀 더 많이 실행 큐에 포함되어, 좀 더 많은 작업 시간을 할당받을 뿐입니다.

그리고 스레드의 우선순위는 해당 스레드를 생성한 스레드의 우선순위를 상속받게 됩니다.

**멀티 스레드(multi thread)**

일반적으로 하나의 프로세스는 하나의 스레드를 가지고 작업을 수행하게 됩니다.

하지만 멀티 스레드(multi thread)란 하나의 프로세스 내에서 둘 이상의 스레드가 동시에 작업을 수행하는 것을 의미합니다.

또한, 멀티 프로세스(multi process)는 여러 개의 CPU를 사용하여 여러 프로세스를 동시에 수행하는 것을 의미합니다.

멀티 스레드와 멀티 프로세스 모두 여러 흐름을 동시에 수행하다는 공통점을 가지고 있습니다.

멀티 프로세스는 각 프로세스가 독립적인 메모리를 가지고 별도로 실행되지만, 멀티 스레드는 각 스레드가 자신이 속한 프로세스의 메모리를 공유한다는 점이 다릅니다.

멀티 스레드는 각 스레드가 자신이 속한 프로세스의 메모리를 공유하므로, 시스템 자원의 낭비가 적습니다.

또한, 하나의 스레드가 작업을 할 때 다른 스레드가 별도의 작업을 할 수 있어 사용자와의 응답성도 좋아집니다.

**문맥 교환(context switching)**

컴퓨터에서 동시에 처리할 수 있는 최대 작업 수는 CPU의 코어(core) 수와 같습니다.

만약 CPU의 코어 수보다 더 많은 스레드가 실행되면, 각 코어가 정해진 시간 동안 여러 작업을 번갈아가며 수행하게 됩니다.

이때 각 스레드가 서로 교체될 때 스레드 간의 문맥 교환(context switching)이라는 것이 발생합니다.

문맥 교환이란 현재까지의 작업 상태나 다음 작업에 필요한 각종 데이터를 저장하고 읽어오는 작업을 가리킵니다.

이러한 문맥 교환에 걸리는 시간이 커지면 커질수록, 멀티 스레딩의 효율은 저하됩니다.

오히려 많은 양의 단순한 계산은 싱글 스레드로 동작하는 것이 더 효율적일 수 있습니다.

따라서 많은 수의 스레드를 실행하는 것이 언제나 좋은 성능을 보이는 것은 아니라는 점을 유의해야 합니다.

**스레드 그룹(thread group)**

스레드 그룹(thread group)이란 서로 관련이 있는 스레드를 하나의 그룹으로 묶어 다루기 위한 장치입니다.

자바에서는 스레드 그룹을 다루기 위해 ThreadGroup이라는 클래스를 제공합니다.

이러한 스레드 그룹은 다른 스레드 그룹을 포함할 수도 있으며, 이렇게 포함된 스레드 그룹은 트리 형태로 연결됩니다.

이때 스레드는 자신이 포함된 스레드 그룹이나 그 하위 그룹에는 접근할 수 있지만, 다른 그룹에는 접근할 수 없습니다.

이렇게 스레드 그룹은 스레드가 접근할 수 있는 범위를 제한하는 보안상으로도 중요한 역할을 하고 있습니다.

**데몬 스레드(deamon thread)**

데몬 스레드(deamon thread)란 다른 일반 스레드의 작업을 돕는 보조적인 역할을 하는 스레드를 가리킵니다.

따라서 데몬 스레드는 일반 스레드가 모두 종료되면 더는 할 일이 없으므로, 데몬 스레드 역시 자동으로 종료됩니다.

데몬 스레드의 생성 방법과 실행 방법은 모두 일반 스레드와 같습니다.

단, 실행하기 전에 setDaemon() 메소드를 호출하여 데몬 스레드로 설정하기만 하면 됩니다.

이러한 데몬 스레드는 일정 시간마다 자동으로 수행되는 저장 및 화면 갱신 등에 이용되고 있습니다.

**가비지 컬렉터(gabage collector)**

데몬 스레드를 이용하는 가장 대표적인 예로 가비지 컬렉터(gabage collector)를 들 수 있습니다.

가비지 컬렉터(gabage collector)란 프로그래머가 동적으로 할당한 메모리 중 더 이상 사용하지 않는 영역을 자동으로 찾아내어 해제해 주는 데몬 스레드입니다.

자바에서는 프로그래머가 메모리에 직접 접근하지 못하게 하는 대신에 가비지 컬렉터가 자동으로 메모리를 관리해 줍니다.

이러한 가비지 컬렉터를 이용하면 프로그래밍을 하기가 훨씬 쉬워지며, 메모리에 관련된 버그가 발생할 확률도 낮아집니다.

보통 가비지 컬렉터가 동작하는 동안에는 프로세서가 일시적으로 중지되므로, 필연적으로 성능의 저하가 발생합니다.

하지만 요즘에는 가비지 컬렉터의 성능이 많이 향상되어, 새롭게 만들어지는 대부분의 프로그래밍 언어에서 가비지 컬렉터를 제공하고 있습니다.

#### 람다 표현식(lambda expression)이란?

람다 표현식(lambda expression)이란 간단히 말해 메소드를 하나의 식으로 표현한 것입니다.

메소드를 람다 표현식으로 표현하면, 클래스를 작성하고 객체를 생성하지 않아도 메소드를 사용할 수 있습니다.

그런데 자바에서는 클래스의 선언과 동시에 객체를 생성하므로, 단 하나의 객체만을 생성할 수 있는 클래스를 익명 클래스라고 합니다.

따라서 자바에서 람다 표현식은 익명 클래스와 같다고 할 수 있습니다.

#### 함수형 인터페이스(functional interface)

람다 표현식을 사용할 때는 람다 표현식을 저장하기 위한 참조 변수의 타입을 결정해야만 합니다.

람다 표현식을 하나의 변수에 대입할 때 사용하는 참조 변수의 타입을 함수형 인터페이스라고 부릅니다.

함수형 인터페이스는 추상 클래스와는 달리 단 하나의 추상 메소드만을 가져야 합니다.

또한, 다음과 같은 어노테이션(annotation)을 사용하여 함수형 인터페이스임을 명시할 수 있습니다.

@**FunctionalInterface**

위와 같은 어노테이션을 인터페이스의 선언 앞에 붙이면, 컴파일러는 해당 인터페이스를 함수형 인터페이스로 인식합니다.

자바 컴파일러는 이렇게 명시된 함수형 인터페이스에 두 개 이상의 메소드가 선언되면 오류를 발생시킵니다.

#### 메소드 참조(method reference)

메소드 참조(method reference)는 람다 표현식이 단 하나의 메소드만을 호출하는 경우에 해당 람다 표현식에서 불필요한 매개변수를 제거하고 사용할 수 있도록 해줍니다.

(문법)

클래스이름**::**메소드이름

또는

참조변수이름**::**메소드이름

#### 생성자 참조

생성자를 호출하는 람다 표현식도 앞서 살펴본 메소드 참조를 이용할 수 있습니다.

즉, 단순히 객체를 생성하고 반환하는 람다 표현식은 생성자 참조로 변환할 수 있습니다.

**스프링**

**스프링 프레임워크**([영어](https://ko.wikipedia.org/wiki/%EC%98%81%EC%96%B4): Spring Framework)는 [자바 플랫폼](https://ko.wikipedia.org/wiki/%EC%9E%90%EB%B0%94_(%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4_%ED%94%8C%EB%9E%AB%ED%8F%BC))을 위한 [오픈 소스](https://ko.wikipedia.org/wiki/%EC%98%A4%ED%94%88_%EC%86%8C%EC%8A%A4) [애플리케이션 프레임워크](https://ko.wikipedia.org/wiki/%EC%95%A0%ED%94%8C%EB%A6%AC%EC%BC%80%EC%9D%B4%EC%85%98_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC)로서 간단히 **스프링**(Spring)이라고도 한다. 동적인 웹 사이트를 개발하기 위한 여러 가지 서비스를 제공하고 있다. 대한민국 [공공기관](https://ko.wikipedia.org/wiki/%EA%B3%B5%EA%B3%B5%EA%B8%B0%EA%B4%80)의 웹 서비스 개발 시 사용을 권장하고 있는 [전자정부 표준프레임워크](https://ko.wikipedia.org/wiki/%EC%A0%84%EC%9E%90%EC%A0%95%EB%B6%80_%ED%91%9C%EC%A4%80%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC)의 기반 기술로서 쓰이고 있다.

* 경량 컨테이너로서 자바 객체를 직접 관리한다. 각각의 객체 생성, 소멸과 같은 라이프 사이클을 관리하며 스프링으로부터 필요한 객체를 얻어올 수 있다.
* 스프링은 [Plain Old Java Object](https://ko.wikipedia.org/wiki/Plain_Old_Java_Object) 방식의 프레임워크이다. 일반적인 [J2EE](https://ko.wikipedia.org/wiki/J2EE) 프레임워크에 비해 구현을 위해 특정한 [인터페이스](https://ko.wikipedia.org/wiki/%EC%9D%B8%ED%84%B0%ED%8E%98%EC%9D%B4%EC%8A%A4)를 구현하거나 상속을 받을 필요가 없어 기존에 존재하는 [라이브러리](https://ko.wikipedia.org/wiki/%EB%9D%BC%EC%9D%B4%EB%B8%8C%EB%9F%AC%EB%A6%AC) 등을 지원하기에 용이하고 객체가 가볍다.
* 스프링은 [제어 반전](https://ko.wikipedia.org/wiki/%EC%A0%9C%EC%96%B4_%EB%B0%98%EC%A0%84)(IoC : Inversion of Control)을 지원한다. 컨트롤의 제어권이 사용자가 아니라 프레임워크에 있어서 필요에 따라 스프링에서 사용자의 코드를 호출한다.
* 스프링은 [의존성 주입](https://ko.wikipedia.org/wiki/%EC%9D%98%EC%A1%B4%EC%84%B1_%EC%A3%BC%EC%9E%85)(DI : Dependency Injection)을 지원한다. 각각의 계층이나 서비스들 간에 의존성이 존재할 경우 프레임워크가 서로 연결시켜준다.
* 스프링은 [관점 지향 프로그래밍](https://ko.wikipedia.org/wiki/%EA%B4%80%EC%A0%90_%EC%A7%80%ED%96%A5_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)(AOP : Aspect-Oriented Programming)을 지원한다. 따라서 [트랜잭션](https://ko.wikipedia.org/wiki/%ED%8A%B8%EB%9E%9C%EC%9E%AD%EC%85%98)이나 로깅, 보안과 같이 여러 모듈에서 공통적으로 사용하는 기능의 경우 해당 기능을 분리하여 관리할 수 있다.
* 스프링은 영속성과 관련된 다양한 서비스를 지원한다. [iBATIS](https://ko.wikipedia.org/wiki/IBATIS" \o "IBATIS)나 [하이버네이트](https://ko.wikipedia.org/wiki/%ED%95%98%EC%9D%B4%EB%B2%84%EB%84%A4%EC%9D%B4%ED%8A%B8) 등 이미 완성도가 높은 데이터베이스 처리 라이브러리와 연결할 수 있는 인터페이스를 제공한다.
* 스프링은 확장성이 높다. 스프링 프레임워크에 통합하기 위해 간단하게 기존 라이브러리를 감싸는 정도로 스프링에서 사용이 가능하기 때문에 수많은 라이브러리가 이미 스프링에서 지원되고 있고 스프링에서 사용되는 라이브러리를 별도로 분리하기도 용이하다.

### 제어 반전 컨테이너**[**[**편집**](https://ko.wikipedia.org/w/index.php?title=%EC%8A%A4%ED%94%84%EB%A7%81_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC&action=edit&section=4)**]**

[제어 반전](https://ko.wikipedia.org/wiki/%EC%A0%9C%EC%96%B4_%EB%B0%98%EC%A0%84)(IoC: Inversion of Control) 컨테이너는 스프링의 가장 중요하고 핵심적인 기능으로서 자바의 [반영](https://ko.wikipedia.org/wiki/%EB%B0%98%EC%98%81_(%EC%BB%B4%ED%93%A8%ED%84%B0_%EA%B3%BC%ED%95%99))(reflection)을 이용해서 객체의 생명주기를 관리하고 [의존성 주입](https://ko.wikipedia.org/wiki/%EC%9D%98%EC%A1%B4%EC%84%B1_%EC%A3%BC%EC%9E%85)(Dependency Injection)을 통해 각 계층이나 서비스들간의 의존성을 맞춰준다. 이러한 기능들은 주로 환경설정을 담당하는 XML 파일에 의해 설정되고 수행된다.

### 관점 지향 프로그래밍 프레임워크**[**[**편집**](https://ko.wikipedia.org/w/index.php?title=%EC%8A%A4%ED%94%84%EB%A7%81_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC&action=edit&section=5)**]**

스프링은 로깅이나 보안, 트랜잭션 등 핵심적인 비즈니스 로직과 관련이 없으나 여러 곳에서 공통적으로 쓰이는 기능들을 분리하여 개발하고 실행 시에 서로 조합할 수 있는 [관점 지향 프로그래밍](https://ko.wikipedia.org/wiki/%EA%B4%80%EC%A0%90_%EC%A7%80%ED%96%A5_%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D)(AOP)을 지원한다. 기존에 널리 사용되고 있는 강력한 관점 지향 프로그래밍 프레임워크인 [AspectJ](https://ko.wikipedia.org/wiki/AspectJ)도 내부적으로 사용할 수 있으며, 스프링 자체적으로 지원하는 실행시(Runtime)에 조합하는 방식도 지원한다.

### 데이터 액세스 프레임워크**[**[**편집**](https://ko.wikipedia.org/w/index.php?title=%EC%8A%A4%ED%94%84%EB%A7%81_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC&action=edit&section=6)**]**

스프링은 [데이터베이스](https://ko.wikipedia.org/wiki/%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%B2%A0%EC%9D%B4%EC%8A%A4)에 접속하고 자료를 저장 및 읽어오기 위한 여러 가지 유명한 라이브러리, 즉 [JDBC](https://ko.wikipedia.org/wiki/JDBC), [iBATIS](https://ko.wikipedia.org/wiki/IBATIS" \o "IBATIS)([MyBatis](https://ko.wikipedia.org/wiki/MyBatis" \o "MyBatis)), [하이버네이트](https://ko.wikipedia.org/wiki/%ED%95%98%EC%9D%B4%EB%B2%84%EB%84%A4%EC%9D%B4%ED%8A%B8) 등에 대한 지원 기능을 제공하여 데이터베이스 프로그래밍을 쉽게 사용할 수 있다.

### 트랜잭션 관리 프레임워크**[**[**편집**](https://ko.wikipedia.org/w/index.php?title=%EC%8A%A4%ED%94%84%EB%A7%81_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC&action=edit&section=7)**]**

스프링은 추상화된 [트랜잭션](https://ko.wikipedia.org/wiki/%ED%8A%B8%EB%9E%9C%EC%9E%AD%EC%85%98) 관리를 지원하며 XML 설정파일 등을 이용한 선언적인 방식 및 프로그래밍을 통한 방식을 모두 지원한다.

### 모델-뷰-컨트롤러 패턴**[**[**편집**](https://ko.wikipedia.org/w/index.php?title=%EC%8A%A4%ED%94%84%EB%A7%81_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC&action=edit&section=8)**]**

스프링은 웹 프로그램밍 개발 시 거의 표준적인 방식인 **Spring MVC**라 불리는 [모델-뷰-컨트롤러](https://ko.wikipedia.org/wiki/%EB%AA%A8%EB%8D%B8-%EB%B7%B0-%EC%BB%A8%ED%8A%B8%EB%A1%A4%EB%9F%AC)(MVC) 패턴을 사용한다. DispatcherServlet이 Controller 역할을 담당하여 각종 요청을 적절한 서비스에 분산시켜주며 이를 각 서비스들이 처리를 하여 결과를 생성하고 그 결과는 다양한 형식의 View 서비스들로 화면에 표시될 수 있다.

### 배치 프레임워크**[**[**편집**](https://ko.wikipedia.org/w/index.php?title=%EC%8A%A4%ED%94%84%EB%A7%81_%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC&action=edit&section=9)**]**

스프링은 특정 시간대에 실행하거나 대용량의 자료를 처리하는데 쓰이는 [일괄 처리](https://ko.wikipedia.org/wiki/%EC%9D%BC%EA%B4%84_%EC%B2%98%EB%A6%AC)(Batch Processing)을 지원하는 배치 프레임워크를 제공한다. 기본적으로 스프링 배치는 Quartz 기반으로 동작한다.

1. POJO(Plain Old Java Object)

(특정 규약이나 환경에 종속되지 않고 객체지향원리에 충실한) 평범한 구식 자바 객체

Spring이 POJO 기반이라는 말은, 특정한 API(예를 들어 Servlet)를 상속하거나 구현하지 않은 일반적인 자바 객체로 개발한다는 의미.

장점 : 1) 프레임워크 또는 클라이언트간의 이식성을 높일 수 있다. (servlet으로 개발하면 servlet으로만 사용할 수 있음)

2) 테스트 효율이 높아진다. (servlet의 경우 test하려면 request, response를 mock으로 구현해줘야 한다.

2. IoC(Inversion Of Control)

정의 : 프로그램의 제어권이 어플리케이션에서 외부로 역전된 것.(여기서 제어는 객체 생성, 객체간 의존관계 연결 등을 의미) 대부분의 프레임워크는 IoC가 도입되어 있음.

3. 컨테이너

정의 : 프로그래머가 작성한 코드의 처리(예: 메소드 호출, 객체 생성) 과정을 위임받은 대상.

역할 : 인스턴스의 생명주기 관리, 추가 기능(인스턴스 초기화, 의존성 주입, Security 관리 등) 제공

예: 서블릿 컨테이너(Tomcat, jetty, jboss 등), EJB 컨테이너, Spring

4. DI(Dependency Injection)

정의 : 객체 간의 의존관계를 외부에서 주입해주는 것.

장점 : 코드가 단순해짐. 컴포넌트간의 결합도가 제거됨. 테스트가 유리함. 객체 간의 의존관계를 한눈에 파악할 수 있음.

방법 : 1) constructor 2) setter

5. AOP(Aspect Oriented Programming)

기본 개념 : 횡단관심사(cross-concern)의 모듈화를 지향하는 프로그래밍 패러다임.

횡단관심사 : 비즈니스로직은 아니지만 없어서는 안되는 기능들 (예 : 로깅, 예외처리, 트랜잭션, 보안 등)

JoinPoint : 횡단관심 모듈의 기능을 적용할 수 있는 모든 위치

PointCut : 모든 JoinPoint중 횡단관심 모듈 기능을 적용할 위치를 선택하는 행위

Advice : JoinPoint에서 동작할 횡단관심 모듈 기능을 구현한 코드. (주로 메소드 단위) 언제(메소드 실행 전, 후, 전+후 등), 무슨 기능을 수행할지 결정함.

Weaving : PointCut에 의해 선택된 JoinPoint에 Advise를 삽입하는 과정.

Aspect : PointCut + Advice 를 정의한 것. (자바 클래스가 될 수도 있고, xml 파일에서 설정할 수도 있음) Aspect를 정의한다는 것은 결국 (전체 비즈니스 로직들 중) 어디에서, 언제, 무엇을 할 것인가(횡단관심사 중 어떤 기능을 적용할 것인가)를 결정하는 것이다.

- @Autowired 타입으로 결정

동일한 타입의 빈 객체를 찾아서 주입 대상으로 선택하는 것으로 생각할 수 있지만 더 정확하게 말하면 해당 타입에 할당할 수 있는 빈 객체를 찾아서 주입 대상으로 선택한다.

주입할 객체가 두 개일 경우 `@Qualifier` 로 지정

주입할 객체가 없으면 `(required=false)` 로 설정

1. 타입이 같은 빈 객체를 검색한다. 한 개면 그 빈 객체를 사용한다. `@Qualifier`가 명시되어 있을 경우, `@Qualifier`와 같은 값을 갖는 빈 객체여야 한다. (value)

2. 타입이 같은 빈 객체가 두 개 이상 존재하면, `@Qualifier`로 지정한 빈 객체를 찾는다. 존재하면, 그 객체를 사용한다.

`@Qualifier("dvdPlayter03")` (bean의 id)

3. 타입이 같은 빈 객체가 두 개 이상 존재하고 `@Qualifier`가 없을 경우, 이름이 같은 빈 객체를 찾는다. 존재하면, 그 객체를 사용한다.

위 경우에 해당되지 않으면 스프링 컨테이너는 익셉션을 발생시킨다.

`@Autowired`는 기본적으로 같은 타입의 빈이 여러개 있다면 그중에서도 같은 이름의 빈을 주입해줌. 같은 이름의 빈이 없다면 에러를 던진다.

- @Resources(name="memberDao") 이름으로 결정

- IOC 제어의 역행 제어권

개발자가 프로그램의 흐름을 주도하는 것이 아니라 객체의 생성 및 생명주기 관리 (등)을 프레임워크가 주도하는 것입니다 대표적으로 의존관계 주입이 있습니다

IOC란 인스턴스(객체)의 생성부터 소멸까지의 객체 생명주기 관리를 개발자가 하는 대신 스프링(컨테이너)이 대신 해주게 하는것을 말한다.

메소드나 객체의 호출작업을 개발자가 결정하는 것이 아니라, 외부에서 결정되는 것을 의미

스프링 IoC 기능의 대표적인 동작 원리는 주로 의존관계 주입이라고 불린다.

객체관리를 개발자가 하지 않고 프레임워크가 하는 것

프레임워크는 컨테이너를 두고 그 안에서 객체를 관리한다. 프레임워크 자체를 운영하기 위해 자신이 필요한 객체들은 자동으로 등록하고 추가적으로 개발자가 위탁관리를 요청한 클래스들을 대신 관리해 준다(빈 설정)

→ 변화가 있어도 개발자의 클래스 코드가 안 바뀌게 하고 싶어서

`@Component`라는 어노테이션이 붙은 클래스들은 Spring의 `Container`가 알아서 Spring Bean 객체로 등록하고 생성한다. 이렇게 생성된 객체는 `@Autowired`라는 어노테이션이 붙은 변수의 타입(타입이 같은 Bean 여러개 있다면 이름을 본다.)을 보고 해당 변수에 객체를 주입하게 된다.

스프링의 `Container`가 \*\*대신 객체를 생성해주고 알아서 객체를 주입해준다. 이렇게 생성된 객체는 자신이 어디에 쓰일지 알지 못한다.\*\* 이것이 \*\*제어 역전의 원칙\*\* 이며 스프링은 \*\*DI(의존성 주입)\*\* 라는 개념으로 구현하고 있다.

- 빈 컨테이너

스프링은 개발자 대신 객체를 관리하기 위해 '빈 컨테이너'를 갖고 있다. 개발자는 스프링의 빈 컨테이너에 등로갷 놓고 사용하고자 하는 클래스들을 스프링에게 미리 알려주어야 한다. → 빈 설정

1. XML

빈 관리가 한 곳에 모여 있다는 장점

2. Annotation

빠르게 개발할 수 있지만 빈 관리정보가 분산되어 있게 됨

3. Java-config

설정정보를 클래스에 바로 구현할 수 있다는 장점

- 컨테이너

Spring Container, DI Container, IoC Container, Bean Container 등.

스프링의 컨테이너는 프로그래머가 작성한 코드의 처리과정을 위임받아 독립적으로 처리하는 존재이다.

IoC방식으로 빈을 관리한다는 의미에서 어플리케이션 컨텍스트나 빈 팩토리를 컨테이너 또는 IoC 컨테이너하고도 한다

의존성 제어, 즉 객체 간의 의존성을 낮추기 위해 바로 Spring 컨테이너가 사용된다.

- `코드가 깔끔해지고 사용하기 쉽다.

- 재사용하기 좋다.

- 테스트하기 쉽다.`

- 어플리케이션 컨텍스트

빈 팩토리 보다 좀 더 확장 느낌으로 비는 생성 관계 설정 거 같은 어플리케이션 전반에 걸쳐 모든 구성요소의 재우 작업을 담당하는 IoC 컨테이너

빈 팩토리보다 좀 더 확장한 느낌

IoC 방식을 따라 만들어진 일종의 빈 팩토리

어플리케이션 전반에 걸쳐 모든 구성요소의 제어 작업을 담당하는 IoC 엔진이라는 의미가 좀 더 부각

별도의 정보를 참고해서 빈의 생성, 관계설정 들의 제어 작업을 총괄

IoC 컨테이너 빈 팩토리 라고 부름

스프링이 지원하는 애플리케이션 지원 기능을 모두 포함해서이야기

- 어노테이션

1. @Configuration

스프링 IOC Contatiner에게 해당 클래스를 `Bean`구성 Class임을 알려주는 것

어플리케이션 컨텍스트 또는 반 팩토리가 사용할 설정정보라는 표시

2. @Bean vs @Component

둘 다 Spring(IOC) Container에 Bean을 등록하도록 하는 메타데이터를 기입하는 어노테이션.

1. @Bean

개발자가 직접 제어가 불가능한 외부 라이브러리 등을 `Bean`으로 만들려할 때 사용

`@Bean(name=" ")` 이름을 직접 정해주지 않으면 메소드 이름을 CamelCase로 변경한 것이 id로 등록

오브젝트 생성을 담당하는 IoC용 메소드라는 표시

2. @Component

개발자가 직접 작성한 Class를 Bean으로 등록하기 위한

id 설정은 @Bean과 동일하지만 @Component(value=" ")로 이름을 지정

`<context:component-scan>`

3. @Value("22")

기본값 설정. 필드위에 선언

4. @Controller, @Service, @Repository

- DI 명시적 의존 주입 > 자동주입

- 정의

필요한 객체를 직접 생성하는 것이 아닌 외부로 부터 필요한 객체를 받아서 사용하는 것이다.

개발자가 객체를 직접 생성하는 게 아니라 외부에서 생성한 후 주입시켜주는 방식

객체 간의 의존성을 개발자가 설정하는 것이 아닌 스프링 컨테이너가 주입시켜 주는 기능

자신이 사용할 오브젝트에 대한 선택과 생성 제어권을 외부로 넘기고 자신은 수동적으로 주입받은 오브젝트를 사용한다는 점에서 IoC의 개념에 잘 들어맞는다

- 장점

이를 통해 객체간의 결합도를 줄이고 코드의 재활용성을 높여준다.

만약 직접 A객체를 생성하고 A를 의존하고잇는 B C클래스에서 A클래스가 바뀌게되면 A를 의존성을 갖는 모든 클래스를 변경해야한다.

코드를 수정해야하는 경우 직접 객체를 설정했을 경우에는 해당 객체를 선언한 코드를 직접 일일이 다 수정해야하지만 di로 주입한 방법은 설정 파일만 바꿔 주면 됩니다. 이는 특히, 프로젝트가 커지고, 소스가 길어질때는 결합도가 낮아지고 스프링에서 사용하는 DI가 확장성이나 유지보수에 용이

직접 객체를 설정했을 경우, 고양이의 정보가 아닌 강아지의 정보를 보고 싶을 때 메인 클래스의 소스를 싹 고쳐야 하지만, DI로 주입한 방법은 설정파일만 바꿔주면 됨. → 프로젝트가 커지고, 소스가 길어질때는 스프링에서 사용하는 DI가 확장성이나 유지보수에 용이

\*\*의존관계 검색 AnnitationConfigApplicationContext , getBean같은 코드를 서용함\*\*

\*\*setter -\*\* <property> 태그의 name 속성을 통해 어떤 의존 객체가 주입되는지 알 수 있다.

<property>방식은 setter()가 있어야함

\*\*JAVA\*\* → 잘 사용하지 않는다고 함. 근데 부트는 대부분 java파일에서 설정해줌

`@Configuration` - "이 클래스는 스프링 설정에 사용되는 클래스입니다"라고 명시해주는 어노테이션

`@Bean` - 객체를 생성하는 어노테이션

`AnnotationConfigApplicationContext`를 이용해서 context 불러옴

`GenericXmlApplicationContext` 클래스에서 파생된 클래스이기 때문에 JAVA파일도 결국 내부적으로 xml로 변환이 됨

\*\*생성자 -\*\* 빈 객체를 생성하는 시점에 모든 의존 객체가 주입된다.

- 장점 : 직관적 코딩, 이에따라 생산성 증대

- constructor-arg를 사용해서 값을 넘김. 순서대로 작성

- Set메소드보다 나은 점은 한 번에 여러 개의 파라미터를 받을 수 있다

- `NullPointerException` 을 방지할 수 있다.

- 주입받을 필드를  `final`로 선언 가능하다.

- 필드 주입이나, 수정자 주입은 객체 생성시점에는 순환참조가 일어나는지 아닌지 발견할 수 있는 방법이 없다.

- 단위테스트 작성하기가 좋아진다.

---

수정자 주입은 스프링 컨테이너가 아닌 외부에서 수정자를 호출해서 주입할 수 있는 방법이라도 열려있지만, 필드주입은 스프링 컨테이너 말고는 외부에서 주입할 수 있는 방법이 없다.

---

어노테이션 → 프로젝트가 거대해짐에 따라 XML을 이용하여 IOC컨테이너를 설정하는 것이 점점 어려워졌기 때문에 Annotation이 등장

```xml

<context:annotation-config/> 추가해야 autowired된 내용을 찾음

```

<property name="exam" ref="exam"/> → `@Autowired`만 입력하는것과 동일

`@Autowired`로 바인딩했을 때 생성된 객체의 클래스(class="...")들이 기준이 된다는 것이다.

```xml

<bean id="exam" class="entity.NewlecExam"/>

```

그래서 xml 내에 해당 객체가 매칭이될 때 굳이 id쓰지 않아도 된다.

```xml

<bean id="exam2" class="spring.di.entity.NewlecExam" p:kor="10" p:eng="20"/>

<bean id="exam1" class="spring.di.entity.NewlecExam" p:kor="30" p:eng="40"/>

```

위처럼 대응되는 객체가 여러 개 존재하면 `@Qualifier`를 사용. 클래스 뿐만 아니라 변수명에 해당하는 `id`를 비교함

`@Qualifier("exam1")`

---

`@Autowired` 위치 → 필드 위, 생성자 위, setter 위

required 옵션

`@Autowired` 기능은 xml에서 객체 생성이 이루어 진 뒤에 실행이 되어야 하는데 만약 생성이 되지 않은 상황에서 실행하게되면 `NullPointerException`이 발생하겠다. 따라서 이를 방지하기 위해 required 옵션을 사용

- DI (@Autowird로 하는 3가지) 의존 관계자동 주입

장점 : 스프링이 알아서 의존 객체를 찾아서 주입해준다. 예를 들어, 자동 주입을 사용하면 의존객체를 명시하지 않아도 스프링이 필요한 의존 빈 객체를 찾아서 주입해준다

1. 생성자 주입

```java

@Component

public class SampleController {

private SampleRepository sampleRepository;

@Autowired <----------

public SampleController(SampleRepository sampleRepository) {

this.sampleRepository = sampleRepository;

}

}

```

Spring4.3부터는 클래스의 생성자가 하나이고 그 생성자로 주입받을 객체가 `Bean`으로 등록되어 있다면 생성자 주입에서 @Autowired 생략가능

2. 필드 주입

```java

@Component

public class SampleController {

@Autowired <-----------

private SampleRepository sampleRepository;

}

```

3. Setter 주입

```java

@Component

public class SampleController {

private SampleRepository sampleRepository;

@Autowired <----------

public void \*\*set\*\*SampleRepository(SampleRepository sampleRepository) {

this.sampleRepository = sampleRepository;

}

}

```

4. 생성자를 사용하는 방법이 좋은 이유는 필수적으로 사용해야하는 의존성 없이는 인스턴스를 만들지 못하도록 강제할 수 있기 때문이다.

5. 필드, Setter 주입 방법의 필요성

순환참조 - A가 B를 참조하고 B가 A를 참조하는 상태

A클래스와 B클래스가 순환 참조 관계이고 둘 다 생성자 주입을 사용한다면 A와 B중 어떤 인스턴스도 생성할 수 없고 결과적으로 어플리케이션이 실행조차 되지 않음.

가급적이면 순환 참조를 피하는게 좋지만 어쩔 수 없는 상황이라면 필드나 Setter 주입방법을 사용할 수 있다. ?

- 순환참조

[https://yaboong.github.io/spring/2019/08/29/why-field-injection-is-bad/](https://yaboong.github.io/spring/2019/08/29/why-field-injection-is-bad/)

실제 코드가 호출이 되기 전까지는 아무것도 알지 못한다. 스프링 애플리케이션 구동도 너무나 잘된다. 여기서 궁금했던게 하나 있다.

> 왜 빈 생성이 잘 되는거지…?

>

수정자 주입이나 필드 주입시에 스프링 ApplicationContext 를 통해서 현재 로딩된 빈 목록을 출력하면 사이클 호출 로직을 가진 두개의 빈이 모두 떠있는 것을 확인할 수 있었다. 아니 사이클 호출을 하고 있는데 빈이 어떻게 생성될 수 있는거지? 생성은 안하고 빈 목록만 가지고 있다가 lazy 로딩하는 방식인건가? 근데 따로 lazy init 옵션을 주지 않으면 lazy 로딩은 적용 되지 않는다던데…?

여기저기 물어보니 한분이 명쾌한 답변을 주셨는데 ‘아 멍청이’ 하는 생각이 들었다. (혹시 보고 계신다면 다시한번 감사드립니다 ㅋ ㅋ) 객체생성시점에서 순환참조가 일어나는 것과 객체생성 후 비즈니스 로직상에서 순환참조가 일어나는 것은 완전히 다른 이야기인데, 하나로 묶어서 생각하고 있었기 때문에 이런 이상한 질문에 빠졌던 것이다.

> 필드 주입이나, 수정자 주입은 객체 생성시점에는 순환참조가 일어나는지 아닌지 발견할 수 있는 방법이 없다.

>

생성자 주입은 어플리케이션 구동실패한다

빈 생성시 아래와 같은 로직이 수행되면서 어떤 시점에 스프링이 그것을 캐치해서 순환참조라고 알려주는 것 같다.

```java

new CourseServiceImpl(new StudentServiceImpl(new CourseServiceImpl(new ...)))

```

이처럼 생성자 주입을 사용하면 객체 간 순환참조를 하고 있는 경우에 스프링 애플리케이션이 구동되지 않는다.

> 컨테이너가 빈을 생성하는 시점에서 객체생성에 사이클관계가 생기기 때문이다!

>

수정자 주입을 사용하면 아주 잘 구동되고 순환참조를 하고 있는 부분에 대한 호출이 이루어질 경우 StackOverflowError 를 뱉기 때문에, 오류를 뱉을 수 밖에 없는 로직을 품고 애플리케이션이 구동되는 것이다.

- 생성자주입을 사용해야하는 이유

[https://yaboong.github.io/spring/2019/08/29/why-field-injection-is-bad/](https://yaboong.github.io/spring/2019/08/29/why-field-injection-is-bad/)

[https://webdevtechblog.com/스프링-순환-참조-circular-reference-d01c6beee7e6](https://webdevtechblog.com/%EC%8A%A4%ED%94%84%EB%A7%81-%EC%88%9C%ED%99%98-%EC%B0%B8%EC%A1%B0-circular-reference-d01c6beee7e6)

[https://mangkyu.tistory.com/125](https://mangkyu.tistory.com/125)

- 빈

- 왜 Bean?

스프링은 객체를 다루기 위한 보조 기능들을 제공한다. 예를들어 트랜잭션처리, 객체의 라이프사이클 관리, 자동주입 등등... 정확히 어떠한 기능인지는 알 수 없을지 라도 많은 도움을 준다는 것은 알 수 있다. 이러한 기능을 제공받기 위해서 객체를 제어하기 위한 주도권을 스프링에게 넘겨주어야 하고, 그러기 위해서 객체를 Bean으로 생성 하는 것이다. 객체 제어에 대한 주도권이 개발을 하고 있는 '개발자'가 아닌 '스프링'에게 이전(Inversion)되었다고 하여, 이것을 IoC(Inversion of Control)이라고 한다.

Bean : 스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트

스프링 빈 : 스프링 컨테이너가 생성과 관계 설정 등을 제어

스프링이 IoC 방식으로 관리하는 오브젝트

스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트.

스프링 컨테이너가 생성과 관계설정, 사용 등을 제어해주는 제어의 역전이 적용된 오브젝트

Spring IOC 컨테이너가 관리하는 자바 객체

우리가 new 연산자로 어떤 객체를 생성했을 때 그 객체는 빈이 아니다

ApplicationContext.getBean()으로 얻어질 수 있는 객체는 빈이다

즉, Spring에서 빈은 ApplicationContext가 알고 있는 객체, 즉 ApplicationContext가 만들어서 그 안에 담고있는 객체

'스프링이 생성하고, 관리해주는 객체' 라고 할 수 있다. (그러므로 개발자가 직접 new연산자로 생성한다면 그것은 Bean이 아니다

빈 등록 방법

1. ComponentScan

@ComponentScan - 어느 지점부터 컴포넌트를 찾으라고 알려주는 역할

@Component - 실제로 찾아서 빈으로 등록할 클래스

2. 빈 설정파일에 직접 빈을 등록

```java

@Configuration

public class SampleConfiguration {

@Bean

public SampleController sampleController() {

return new SampleController;

}

}

```

@Configuration 애노테이션을 보면 이 애노테이션도 @Component를 사용하기 때문에 @ComponentScan의 스캔 대상이 되고 그에 따라 빈 설정파일이 읽힐때 그 안에 정의한 빈들이 IoC 컨테이너에 등록되는 것이다.

- 빈 팩토리

빈의 생성과 관계설정 같은 제어를 담당하는 IoC 오브젝트

빈을 생성하고 관계를 설정하는 IoC의 기본 기능에 초점을 맞춘 것

- Interceptor

URL 매핑기반으로 돌아가는 것이다. \*\*전/후 처리\*\* 다 된다.          / 공통적인 작업보다는 앞뒤에 작업을 분산해 놓는 것

클라이언트가 요청한 API를 찾은 뒤에 호출됨

Dispatcher Sevelt(front-controller)을 통해서 우리가만든 controller(back-end-controller)로 들어올 때 Intercepter를 통함

Spring에서 HTTP Request와  HTTP Response를  Controller 앞과 뒤에서 가로채는 역할을 한다.

Servlet의 앞과 뒤에서 HTTP Request와  HTTP Response를 가로채는 필터와 유사하다.

스프링 컨텍스트 내부에 존재

인터셉터는 여러개를 사용할 수 있으며, 실행 순서는 <mvc:interceptors> 에 나오는 순서 입니다. 인터셉터를 주로 사용하는 곳은 로그인 체크, 권한 체크, 프로그램 실행시간 계산 작업 로그 처리, 업로드 파일 처리 등에 많이 사용됨

---

\*\*1. preHandle()\*\* - 컨트롤러 메소드 실행 직전에 수행됩니다. true 를 반환하면 계속 진행이 되고  false 를 리턴하면 실행 체인(다른 인터셉터, 컨트롤러 실행)이 중지되고 반환 됩니다.  필터의 응답 처리가 있다면 그것은 실행이 됩니다.

\*\*2. postHandle()\*\* - 컨트롤러 메소드 실행 직후에 실행 됩니다. View 페이지가 렌더링 되기전에   ModelAndView 객체를 조작 할 수 있습니다.

\*\*3. afterCompletion()\*\* -  View 페이지가 렌더링 되고 난 후 에 실행됩니다.

\*\*4. afterConcurrentHandlingStarted()\*\* - Servlet 3.0 부터 비동기 요청이 가능해 졌습니다.   비동기 요청시 postHandle와 afterCompletion 은 실행되지 않고, 이 메소드가 실행됩니다.

- Filter

선처리만 된다.

DispatcherServlet 이전에 실행(Servlet 라이프사이클)

인증이 필요한 구간에 사용

브라우저에서 서블릿으로 들어올 때 DispatcherServlet전에 처리

스프링 컨텍스트의 바깥에 존재

- AOP

---

- AOP(Aspect Oriented Programming, 관점(관심) 지향적인 프로그래밍)

- OOP(Object Oriented Programming, 객체지향프로그래밍)를 보완하는 확장적인 개념

- Aspect(측면, 관점, 관심) : 핵심적인 비즈니스 로직은 아니지만 반드시 해야하는 작업들

- 관심의 분리(Separation of Concerns)를 통해 핵심관점(업무로직) + 횡단관점(트랜잭션, 로그, 보안, 인증 처리, 자원 정리, 에러 핸들, 커넥션 등)으로 관심의 분리를 실현

- 장점 : 중복되는 코드 제거, 효율적인 유지 보수, 높은 생산성, 재활용성 극대화, 횡단관점분리를 통해 핵심관심만 남게됨, 더욱 OOP답게 됨 / 변화

- java코드에 대한 전/후,전후 처리를 하는것이고 / \*\*공통적인 작업을 모아서 처리\*\*

@Around => ProceedingJoinPoint

@Before, @After => JoinPoint

실행순서 : Filter > DispatcherServlet > Interceptor > AOP

---

1. 핵심관심 : 시스템의 핵심 가치와 목적이 그대로 드러난 관심영역

2. 횡단관심: 핵심관심 전반에 걸쳐 반복적으로 나오게 되는 로깅, 트랜잭션, 보안, 인증, 리소스 풀링, 에러체크 등의 관심영역

3. 관심의 분리: 여러 핵심관심에 걸쳐 등장하는 횡단관심을 분리하여 독립적인 모듈로 만들고 핵심관심이 실행되는 동안 횡단관심을 호출하는 코드를 직접 명시하지 않고 선언적으로 처리

4. 핵심관심 모듈의 중간중간에서 필요한 횡단관심 모듈을 직접 호출하지 않고 위빙(Weaving)이라 불리는 작업을 이용하여 횡단관심 코드가 삽입되도록 만든다.

5. 핵심관심모듈에서는 횡단관심모듈이 무엇인지 조차 인식할 필요가 없음

1. JoinPoint(언제) - When

- 횡단관심모듈은 코드의 아무 때나 삽입이 되는 건 아니다.

- 조인포인트라 불리는 특정 시점에서만 삽입이 가능하다.

- 예)

메서드가 호출되거나 리턴 되는 시점

필드를 액세스 하는 시점

객체가 생성되는 시점

예외가 던져지는 시점

예외 핸들러가 동작하는 시점

클래스가 초기화되는 시점 등이다.

- AOP 프레임워크에 따라 제공되는 조인포인트는 다르며 스프링은 메소드 조인포인트 제공함

1. Before

2. After

3. Around

4. After Return

5. After Throwing

2. PointCut (어디에서) - Where

- 어느 조인포인트에 횡단관심모듈을 삽입할지를 결정하는 기능

- 횡단관심이 삽입될 특정 클래스의 특정 메소드를 선택하는 방법 정의

`@Before` 뒤에 (~~) 적어줌

3. Advice(or Interceptor, 무엇을 ) -

횡단관심모듈(로깅, 보안, 트랜잭션 등)

4. Weaving(위빙)

- 어드바이스(횡단관심)를 삽입하는 과정

- 위빙작업이 일어나는 시간

컴파일 시  - 특별한 컴파일러 필요

클래스 로딩시 - 특별한 클래스로더 필요

런타임시 - 프록시를 이용한 방법(스프링)

Advisor - 충고를 하는놈 `@Advisor`

5. Aspect( Advisor )

1. 어디에서 무엇을  언제  할 것인가?

2. PointCut + Advice를 정의

---

- 프레임워크 라이브러리

\*\*프레임워크\*\*는 전체적인 흐름을 스스로가 쥐고 있으며 사용자는 그 안에서 필요한 코드를 짜 넣으며 반면에 \*\*라이브러리\*\*는 사용자가 전체적인 흐름을 만들며 라이브러리를 가져다 쓰는 것이라고 할 수 있습니다.

다시 말해, 라이브러리는 라이브러리를 가져다가 사용하고 호출하는 측에 전적으로 주도성이 있으며 프레임워크는 그 틀안에 이미 제어 흐름에 대한 주도성이 내재(내포)하고 있습니다.

\*\*프레임워크는 가져다가 사용한다기보다는 거기에 들어가서 사용한다는 느낌/관점으로 접근할 수 있습니다\*\*

- Dispatcher-Servlet

Servlet Container에서 HTTP프로토콜을 통해 들어오는 모든 요청을 프레젠테이션 계층의 제일 앞에 둬서 중앙집중식으로 처리해주는 프론트

컨트롤러(Front Controller)

클라이언트로부터 들어오는 모든 요청을 제일 앞에서 처리해주는 프론트 컨트롤러

URL요청이 오면 이를 가로채서 HandlerMapping(@RequestMapping으로 봐도 무방)에게 해당 요청을 처리할 수 있는 Controller를 찾아내어 요청을 Controller에게 보내고 결과값을 리턴해준다.

모든 서블릿 요청은 DispatcherServlet에서 처리된다. 요청이 들어오면 DispatcherServlet이 servlet-context.xml을 읽어들인다. 여기서 읽어서 prefix, suffix를 읽고 base-package에서 클래스들을 읽어서 메모리에 올린다.

서버 startup

-> web.xml -> DispatcherServlet -> servlet-context.xml (컨트롤러 메모리에 등록(url mappin))

- 실행 순서

1. 클라이언트가 서버에 어떤 요청을 한다면 스프링에서 제공하는 DispatcherServlet(Front-Controller)가 요청을 가로챈다.

2. 요청을 가로챈 DispatcherServlet은 HandlerMapping(URL 분석 등..)에게 보내 해당 요청을 처리할 수 있는 Controller를 찾는다.

- 이 때, RequestMapping으로 구현한 API를 찾게 되는데, 이들은 RequestMappingHandlerAdapter가 모두 가지고 있음.

- 원하는 Mapping을 찾은 경우, 첫 번째로 Intercepter를 처리

- \*\*Argument Resolver\*\* 처리

- Message Converter 처리

3. Controller (Back-end-Controller) → Service→ DAO → DB처리 → DAO → Service → Controller

4. 결과를 받은 컨트롤러는 Model 객체에 결과물과 어떤 view(파일을 보여줄 것인지 등의 정보를 담아 DispatcherServlet에게 보낸다.

5. DispatcherServlet은 ViewResolver에게 받은 뷰의 대한 정보를 넘긴다.

6. ViewResolver는 해당 응답할 View를 찾아서DispatcherServlet에게 알려준다.

(servlet-context.xml에서 suffix, prefix를 통해 /WEB-INF/views/index.jsp 이렇게 만들어주는 것도 ViewResolver)

MessageConverter - `@ResponseBody`가 붙어있는경우 리턴된 값을 `MessageConverter`에 넘김. `@RequestMapping` 는 ViewResolver

7. DispatcherServlet은 응답할 View에게 Render를 지시하고 View는 응답 로직을 처리한다.

8. 결과적으로 DispatcherServlet이 클라이언트에게 렌더링 된 View를 응답한다.

---

1. 클라이언트가 Request 요청을 하면, \*\*DispatcherServlet\*\*이 요청을 가로챈다.

2. DispatcherServlet이 가로챈 요청을 \*\*HandlerMapping\*\*에게 보내 해당 요청을 처리할 수 있는 Controller를 찾는다.

3. 실제 로직 처리 (Controller -> Service -> DAO -> DB -> DAO -> Service -> Controller)

4. 로직 처리 후 \*\*ViewResolver\*\*를 통해 view 화면을 찾는다.

5. 찾은 view 화면을 \*\*View\*\*에 보내면 이 결과를 다시 DispatcherServlet에 보내고, \*\*DispatcherServlet\*\*는 최종 클라이언트에게 전송한다.

Request -> DispatcherServlet -> HandlerMapping -> (Controller -> Service -> DAO -> DB -> DAO -> Service -> Controller) -> DispatcherServlet -> ViewResolver -> View -> DispatcherServlet -> Response

- ViewResolver

View이름을 가지고 View오브젝트를 찾아줌.

InternalResourceViewResolver (기본 뷰 리졸버)

- 특정 뷰 리졸버를 빈으로 등록하지 않으면 DispatcherServlet는 이걸 사용함. JSP를 뷰로 사용하라 때 쓰임

- HandlerMapping

- URI, 컨트롤러의 메소드(핸들러)와의 매핑 정보를 가지고 있음

- HandlerMapping에서 정적 자원에 대한 URL은 DefaultServlet으로 위임할 수 있도록 설정 해주어야 한다 (spring-servlet.xml)

```java

<mvc:default-servlet-handler />

```

---

- @ResponseBody가 없을 경우 리턴타입은 `String` 또는 `ModelAndView`여야 함.

- 아무리 Annotation-config로 설정을해도 직접 XML Bean설정 할 줄 알아야함

- 익셉션 처리

적당한 위치에 Exception클래스?를 만들어서 한 곳에서 처리하는게 좋음(여기에 빨대처럼 다 받아서 처리) 컨트롤러에 익셉션처리를하는것도 그나마 낫지. Java, servlet은 우리가 건드는 부분이아니니까 우리가 건드는 Controller에서 처리

기존 레거시애들. 체크드익셉션애들은 런타임익셉션으로 포장을 하자. 런타임익셉션으로 고치고/보내고. Service인데 SQLException을 던진다던가하면 안됨 → AOP에서 처리하기(어노테이션)

런타임 익셉션으로 전환해라. 익셉션은 컨트롤러까지 보내기 서블릿까지는 X. 이런것도 있으니까 레이어 아키텍쳐. 레이어해놓으니까 익셉션처리를 한 곳에서 할 수 있는 방법이 생긴다. + AOP

- fuction(10) → Arguments / funtion(int i) → Parameter

- @Retention

어노테이션 타입을 어디까지 보유할지를 설정

default는 CLASS임

`RetentionPolicy`는 RetentionPolicy의 값을 넘겨주는 것으로 어노테이션의 메모리 보유 범위가 결정됨

@Retention(RetentionPolicy.RUNTIME / CLASS)

- \*\*SOURCE\*\* : 어노테이션을 사실상 주석처럼 사용하는 것. 컴파일러가 컴파일할때 해당 어노테이션의 메모리를 버립니다.

- \*\*CLASS\*\* : 컴파일러가 컴파일에서는 어노테이션의 메모리를 가져가지만 실질적으로 런타임시에는 사라지게 됩니다. 런타임시에 사라진다는 것은 리플렉션으로 선언된 어노테이션 데이터를 가져올 수 없게 됩니다. 디폴트값입니다.

- \*\*RUNTIME\*\* : 어노테이션을 런타임시에까지 사용할 수 있습니다. JVM이 자바 바이트코드가 담긴 class 파일에서 런타임환경을 구성하고 런타임을 종료할 때까지 메모리는 살아있습니다.

- CLASS, RUNTIME 둘 다 바이트 코드(.class)에 정의되어있지만 `CLASS`는 (RuntimeInVisibleAnnotation), `RUNTIME`은 (RuntimeVisibleAnnotation)로 정의되어있고, 버려지는 메모리가 됨. `SOURCE` .class 파일에서도 보이지 않습니다. 단지 개발자의 어떤 용도로만 사용되다가 컴파일할 때 버려지는 소스

- HandlerMethodArgumentResolver

Controller 메소드에서 특정 조건에 맞는 파라미터가 있을 때 원하는 값을 바인딩해주는 인터페이스.

스프링에서 @RequestBody를 사용해 Request의 Body값을 받아올때, @PathVariable을 통해 Request의 Path Parameter를 받아올 때 사용됨.

- `supportsParameter` 메서드는 현재 파라미터를 resolver이 지원하는지에 대한 boolean을 리턴합니다.

- 파라미터가 여러개 있으니까 여러번 실행됨.

- `resolveArgument` 메서드는 실제로 바인딩을 할 객체를 리턴합니다.

[https://zgundam.tistory.com/154](https://zgundam.tistory.com/154)

- Bean 주입방법

1. 컴포넌트 스캔을 이용한 등록방법

컴포넌트 스캔을 이용한 방법은 주로 실무에서 정형화된 컨트롤러, 서비스, 리포지토리 같은 곳에 사용한다.

\*\*즉, 고정화 되어 바뀔일이 없을 경우! 사용한다고 보면 된다. 무작정 쓰는거보다 상황에 맞게 사용하는것이 중요!!\*\*

- 컴포넌트 스캔방식 종류

- 필드주입방식

- setter를 사용한 주입방식

- 생성자를 이용한 주입방식 - \*\*대게 자주 사용하는 방식으로 권하는 방식이다!!!\*\*

2. 자바를 통한 직접 등록방법

주로 정형화 되지 않거나 상황에 따라 구현 클래스를 변경해야 할일이 생기게 되거나, 그럴 여지가 있는 경우 직접 설정을 통해 등록한다.

\*\*즉, 변경될 일 혹은 여지가 있는 경우에 직접 등록하게되면, 코드를 크게 수정하지않아도 쉽게 변경할 수 있는 장점이 있다!!\*\*

별도의 Config 파일을 생성하여 직접 주입하는 방법이다.

스프링이 시작될때 @Configuration을 찾아가게 된다.

- 빈 설정

```java

<bean id="bean1" class="soundsystem.MyBean"/> //기본 scope가 singleton 이다.

<bean id="bean2" class="soundsystem.MyBean"/>

// <bean id="bean3" class="soundsystem.MyBean"scope="prototype"/> //매번 요청시 새로운 객체 생성

```

ApplicationContext 에서 설정한 class들이 먼저 메모리에 올라가면서 생성자가 먼저 호출되고 나서 소스가 실행된다. bean1과 bean2과 참조가 같은 이유는 scope는 기본적으로 singleton 이기 때문에 컨테이너에 같은 class에 대한 Bean은 하나만 올라가 있기 때문이다.

하지만 scope 설정이 prototype 이면 해당 class 호출될때 메모리에 올라간며 호출시마다 새로운 객체를 생성한다.

- `@Qualifier` 파라미터에도 씀

- @Valid

- `@Valid`가 선언되어 있으므로 User 클래스의 account, password의 제약조건(`@NotEmpty`)에 따라 데이터 유효성을 검사한다.

- 모든 속성의 데이터가 유효하다면 아무일도 일어나지 않는다.

- 데이터가 유효하지 않는 속성이 있으면 그에 대한 에러 정보를 `BindingResult` 변수에 담는다.

- `BindingResult`에서 모든 에러 정보를 얻으려면 `getAllErrors()` 메소드를 호출해야 한다. (기타 메소드들은 공식 문서를 참고하자.)

- ObjectError의 List 타입으로 리턴하며, 각 ObjectError 오브젝트들은 속성 하나의 에러 정보를 담고 있으므로 에러 정보를 뽑아내어 상황에 맞게 코딩하면 된다.

```java

if (bindingResult.hasErrors()) {

List<ObjectError> errorList = bindingResult.getAllErrors();

for (ObjectError error : errorList)

System.out.println(error.getDefaultMessage());

}

```

- `context:annotation-config`

- ApplicationContext 안에 이미 등록된 bean들의 Annotation을 활성화 하기 위해 사용된다.

- bean들이 XML로 등록됐는지 혹은 패키지 스캐닝을 통해 등록됐는지는 중요하지 않다.

- 이미 스프링 컨텍스트에 의해 생성되어 저장된 bean들에 대해서 `@Autowired`와 `@Qualifier` Annotation을 인식할 수 있다.

- `component-scan` 또한 같은 일을 할 수 있는데, `context:annotation-config`는 bean을 등록하는 작업을 하지 않는다.

- 즉, bean을 등록하기 위해 패키지를 안의 클래스를 스캔할 수 없다.

- web.xml

- WAS(e.g. tomcat)는 Servlet을 생성하고 어떤 Servlet이 어떤 요청을 담당할 것인지(mapping), 어떤 요청이 인증과정을 거칠 것인지 등의 제어 기능을 지원해줍니다. 그러려면 \*\*WAS에게 Servlet에 대한 정보를 줘야하는데 이때 쓰이는 파일이 web.xml(Deployment Descriptor)\*\*입니다.

데이터베이스

- char varchar 차이

char(50) varchar(50) 하고 'ABC'를 넣으면

char는 고정길이 문자열타입이기 때문에 ABC넣고 남은 빈공간을 공백으로 다 채움. 그리고 헤더에는 레코드 길이에 대한 정보가 들어있지 않음

varchar 가변길이 문자열타입으로 이후공간은 공백을 채우지 않는다. 헤더에 레코드 길이에 대한 정보가 포함되어있음

- char(n) 에서 n은 바이트가 아니라 글자 수를 의미

최대길이 255

- char 장점

고정 사이즈라서 추후에 연산할 필요가 없기 때문에 검색속도 및 읽는 속도가 다른 타입보다 빠르다. (아이디, 주민등록번호..)

- varchar

65,535 까지 길이 가능

길이에 대한 정보를 저장하는 byte필요

255 이하는 1byte, 이상은 2byte의 추가 크기가 필요하다.

예를 들어 VARCHAR(10)에 ‘test’라는 4자짜리 문자열을 삽입하면 4byte + \*\*1byte(길이를 저장하기 위한 메모리)\*\* = 5byte가 소모된다.

고정공간을 만들어 놓았다가 빈 공간에 대한 비용이 더 큰 경우가 생길 수 있으므로 저는 VARCHAR

- Delete truncate drop

\*\*1.Delete\*\*

- 데이터만 삭제 되며 테이블 용량은 줄어 들지 않는다. 또한 삭제후 잘못 삭제한 것을 되돌릴 수 있다.

- TABLE이나 CLUSTER에 행이 많으면 행이 삭제될 때마다 많은 SYSTEM 자원이 소모 된다.

- Commit이전에는 Rollback이 가능하다.

- 롤백정보를 기록 하므로 Truncate에 비해서 느리다.전체 또는 일부만 삭제 가능 하다.

- 삭제 행수를 반환 한다.

- 데이터를 모두 Delete해도 사용했던 Storage는 Release 처리 되지 않는다.

\*\*2.Truncate - 테이블의 모든 로우를 제거하는 Truncate Table\*\*

테이블을 최초 생성된 초기상태로 만든다.

용량이 줄어들고, 인덱스 등도 모두 삭제 된다.

Rollback 불가능 하다.- 무조건 전체 삭제만 가능 하다.

삭제 행수를 반환 하지 않는다.

테이블이 사용했던 Storage중 최초 테이블 생성시 할당된 Storage만 남기고 Release 처리 된다.

\*\*3. Drop table - 테이블 구조를 제거\*\*

기존 테이블의 존재를 제한다. (테이블의 정의 자체를 완전히 삭제한다)

Rollback 불가능 하다.

테이블이 사용했던 Storage는 모두 Release 처리 된다.

- \*\*이상현상이란?\*\*

릴레이션에서 일부 속성들의 종속으로 인해 데이터 중복이 발생하는 것 (insert, update, delete)

- 조인

- 정의

2개 이상의 테이블을 서로 관계를 맺어서 데이터를 추출하는 방법

두 개 이상의 테이블을 서로 묶어서 하나의 결과 집합으로 만들어 내는 것 ㅡ 테이블 간의 연결 고리로 관계를 맺고 데이터를 추출 해야 하는 작업

- 장점

테이블을 여러 번 읽을 필요가 없고 트랜잭션 보호가 가능

- 동등 조인 EQUI JOIN

WHERE 절에서 = 연산자를 사용해 2개 이상의 테이블이나 피를 연결한 조인이다 집 = 연산자를 사용한 WHERE 절 조건을 만족 하는 데이터를 추출 하는 조인

- 세미 조인

서브 커리를 사용해서 보컬이 의 존재 하는 데이터만 메인거리에서 추출 하는 좋은 방법으로 IN 과 EXISTS 연산자를 사용한 조인

서브 쿼리에 있는 테이블을 B, 메인 쿼리에 사용된 테이블을 A 라고 한다면 세미 조인 B테이블에 존재하는 A테이블의 데이터를 추출 하는 조인

- INNER JOIN

정의 - 양쪽 테이블에 모두 내용이 있는 것만 조인되는 방식 (교집합)

- OUTER JOIN

정의 - 양쪽에 내용과 한쪽에만 내용이 있어도 결과가 표시되는 조인 방식 / 조인의 조건에 만족되지 않는 행까지도 포함시킴

- LEFT OUTER JOIN

정의 - 왼쪽 테이블(FROM 절의 테이블)의 것은 모두 출력 (조건에 만족되지 않는 결과도 출력)

```sql

SELECT U.userId, U.userName, B.prodName, U.addr, U.mobile1 AS "연락처"

FROM \*\*userTbl U\*\* <---- 테이블 기준

LEFT OUTER JOIN \*\*buyTbl B\*\*

ON U.userID = B.userID

ORDER BY U.userID;

------

FROM \*\*buyTbl B\*\*

RIGHT OUTER JOIN \*\*userTbl U\*\* <---- 테이블 기준

ON U.userID = B.userID

ORDER BY U.userID;

```

- RIGHT OUTER JOIN

정의 - 오른쪽 테이블의 것은 모두 출력

- FULL OUTER JOIN

정의 - LEFT, RIGHT OUTER JOIN이 합쳐진 것. 한쪽을 기준으로 조건과 일치하지 않는 것을 출력하는 것이 아니라, 양쪽 모두에 조건이 일치자히 않는 것을 모두 출력하는 개념

UNION과 같은 개념으로 중복데이터 삭제

- SELF JOIN (자체 조인)

동일 테이블 사이의 조인

자기 자신과 자기 자신이 조인. 하나의 테이블에 같은 데이터가 존재하되 의미는 다르게 존재하는 경우에는 두 테이블을 서로 SELF JOIN 시켜서 정보를 확인할 수 있다.

- NESTED LOOP JOIN

중첩 포문과 같은 원리.

IDOL\_GROUP IDOL\_MEMBER를 조인할때 GROUP\_NAME 컬럼 으로 연결

소녀시대 - 소녀시대 멤버

태연 찾고 써니찾고 티파니찾고 다 찾았다

2ne1 row 에 조인될만한 산다라박 박봄 .. 다 찾았다

또.. 다음 row 다음 그룹을 선택함

IDOL\_GROUP 이 OUTER TABLE IDOL\_MEMBER가 INNER TABLE

INNER TABLE이 JOIN 컬럼이 인덱스에 걸려있지 않으면 굉장한 비효일이 발생하게 된다. 왜냐면 한건한건 갈때마다 full scan을 해야한다. 소녀시대가서 full scan으로 찾아야하고 2ne1가서 full scan으로 찾아야하는 상황이 발생하기 때문에 Inner Table의 인덱스 구성 전력이 매우 중요한 요소가 됨.

건별로 계속 JOIN을 하는 방식이기 때문에 대량의 테이블을 JOIN하는 방식으로는 적절하지 않다. 보통은 OLTP성 환경의 쿼리가 적절. 아까의 예제 테이블은 1:M 관계였는데 1쪽에 해당하는 소량의 데이터를 가진 테이블이 OUTER TABLE로 설정되는것이 성능에 유리함.

- SORT MERGE JOIN

중첩 for문과 유사한 방식이지만 테이블과 테이블을 우선 JOIN 컬럼을 기준으로 sorting을하고 JOIN을함.

Innter table에 적적한 인덱스가 없어서 NESTED LOOP JOIN쓰기에 비효율적일 떄 쓰고 아니면 Equl JOIN이 아니라 범위로 JOIN을 하는 경우에 이런 쿼리에서도 적절한 수행원리라고할 수 있음

Table random access가 일어나지 않고 sorting 작업이PGA영역에서 수행되기 때문에 경합이 발생하지 않아서 성능에 유리한 이점이 있다.

- HASH JOIN

배치에 쓰면 좋은 수행 원리.

대용량 테이블을 JOIN할 떄 HASH JOIN을 이용하면 좋다.

IDOL\_GROUP IDOL\_MEMBER 있을떄 MEMBER테이블이 대용량테이블 일경우 GROUP 테이블을 Build Input으로 삼아서 HASH영역에 저장해놓는걱에요.

HASH영역은 아까 SORT MERGE JOIN에서 잠깐 얘기햇던 Sort Area처럼 PGA에 있기떄문에 처리 속도가 빠르고, IDOL\_GROUP 테이블이 HASH영역에 올라가고 IDOL\_MEMBER테이블이 읽히면서 JOIN이 되는 원리.

HASH영역으로 올라갈 떄는 JOIN 컬럼을 기준으로 HAsh Fuction이 적용이 되기때문에 이 Key값에 중복값이 없을수록 성능에 유리.

Equl JOIN만 가능하고 SORT MERGE JOIN처럼 Random Access부하가 없고 유의할점은 Hash영역에 들어갈 테이블의 크기가 충분히 작아야지 성능에 유리하다. 왜냐하면 Hash영역의 사이즈가 정해져잇는데 이 테이블의 용량이 초과해서 벗어나게 되면 오히려 디스크 영역을 사용하게 되기때문에 성능에 불리하게 됨. 그리고 수행빈도가 높은 OLTP환경에서 Hash JOIN으로 풀리게 되면 오히려 CPU나 메모리의 사용량이 늘어서 성능이 나빠질 수 있기때문에 이런 경우는 실행계획을 확인해보고 Hash JOIN을 풀리면 안되는데 HashJOIN으로 풀리고있다하면 힌트를 써서 다른 수행 방식으로 풀릴 수 있도록 유도를 하는게 효과적인 튜닝의 방법.

- 실행계획

내가 작성한 JOIN쿼리가 어떤 수행원리로 풀리고 있는지 궁금하면 실행계획을 보면됨

이 쿼리에는 이 수행원리가 적절하지 않는데 라고 판단이 되면 오라클 힌트를 써서 다른 수행방식으로 한번 유도를 해봐라 그리고 다시 수행계획을 보고 IO량이나 실행시간이 얼마나 변화하는지를 보고서 더 성능에 좋은 방향으로 튜닝을 하면 된다.

- 힌트

옵티마이저가 올바른 실행계획을 생성할 수 있도록 도움을 주는 것

쿼리 작성단계에서 힌투를 준다기 보다는 튜닝 단계에서 많이 힌트를 주게 되어있음

to 옵티마이저

실행계획에 영향을 미치게 됨

이 쿼리를 내부적으로 수행하는데 있어서 이런 방식으로 풀어라 이렇게 힌트를 주는거라서

힌트를 주기전에 지금 이 쿼리가 어떤 방식으로 실행이 되고있는지를 실행계획으로 확인한다. 그래서 만약에 이 쿼리의 테이블에 적절한 인덱스가 생성이 되어있는데도 불구하고 옵티마이저가 잘못된 통계 등으로 인해서 이게 인덱스를 타지 않고 풀스캔을 타고 있다. 라고 했을때는 힌트를 줘서 이 테이블에는 이 인덱스를 사용해라 라는 식의 가이드를 옵티마이저에게 줄 수가 있다. 약간 강제성을 띰 또 조인순서, 조인방식을 바꾸게 ㄱㄴ

- 유의해야하는점

기본적으로는 힌트를 사용하지 않는게 좋다. 왜냐하면 옵티마이저가 기본적으로 올바른 판단을 할 거라는 전제하에 쿼리를 작성해야 되기 때문에 굳이 처음부터 힌트를 줄 필요는 없고 쿼리를 작성하고 실행계획을 확인했을때 이게 예상치보다 너무나 많은 I/O가 생긴다던지 시간이 많이 걸린다던지 하는 경우에 실행계획의 어느 구간에서 부하가 걸리는지를 확인하고 왜 이렇게 느려졌는지 원인을 분석한 다음에 이에 맞는 힌트를 적절하게 사용하면 된다.

- 메모

NL은 인덱스 위주로 유도할때

이중포문 돌면서

조인한다고 보면 되고

Hash 조인은 인덱스 사용 못하는데 데이터 많을때

머 그런케이스에 쓰고

대신 메모리 많이씀

- 서브쿼리

- 정의

다른 쿼리 내부에 포함되어 있는 SELETE 문

하나의 SQL문 안에 포함되어 있는 또 다른 SQL문.

- 사용이유

여러번의 쿼리를 수행해야만 얻을 수 있는 결과를 하나의 중첩된 SQL문장으로 간편하게 결과를 얻을 수 있게 해줌

알려지지 않은 기준을 이용한 검색을 하기 위해서

- Scala subquery SELECT절에 사용함

스칼라 서브 쿼리는 하나의 레코드당 하나의 값을 리턴하는 서브쿼리 보통은 select 절에 많이 쓰지만 그 외에도 컬럼같이 올 수 있는 모든 자리에 쓸 수가 있습니다

- Inline view From절에 사용

- 중첩 서브쿼리 Where 절에 사용

- 단일 행 서브쿼리는 하나의 행만 반환되는 서브쿼리로, 비교연산자(=, <, <=, >=, <>)를 사용

- 다중 행 서브쿼리는 여러 행을 반환하는 서브쿼리로, IN, ANY, ALL, EXISTS를 사용

조인이 유리한지 서브쿼리가 유리한지는 옵티마이저가 알아서 판단해서 실행계획을 작성해주기때문에 서브쿼리는 성능과는 무관

장점

- 단점

연산 비용: 매번 SELECT 문이 실행되기 때문(내용이 복잡할수록 비용이 크다)

- 데이터 I/O 비용: 서브 쿼리의 결과 데이터 양이 크면 연산 결과를 저장소에 쓰기 때문

- 최적화 불가능: 서브 쿼리의 결과 집합에는 인덱스나 제약조건이 없기 때문

- 특징

- Subquery는 Main Query 이전에 한 번 실행

- 서브쿼리는 메인쿼리의 컬럼을 모두 사용할 수 있지만 메인쿼리는 서브쿼리의 컬럼사용 못함.

- 서브쿼리를 이용할 경우 성능이 좋아지는 SQL도 존재 하지만 조인으로 작성하게 되면 OPTIMIZER에게 정확한 판단을 할 수 있도록 도와성능 문제를 예방 할 수 있음

- SQL 작성이 쉽다는 이유 하나만으로 서브쿼리를 남용 할 경우, DB 성능 저하가 발생할 수 있음

- 서브쿼리 vs 조인

일반조인에서 두 테이블은 평등한 관계. 반면 서브쿼리에서는 서브쿼리가 메인 쿼리에 종속되는 관계

일반적으로는 JOIN 이 SUBQUERY 보다 성능이 좋습니당. DB 엔진이 JOIN 에 최적화가 더 잘되어 있거든용

- 인덱스

- 정의

데이터를 좀 더 빠르게 찾을 수 있도록 해주는 도구.

전체 중 특정한 부분을 바로 찾을 수 있는 목차나 색인의 개념

인덱스를 효과적으로 사용하려면 정규화가 되어 있어야한다. 정규화가 되어 있지 않은 테이블은 컬럼이 많으며, 이에 따라 조합할 수 있는 인덱스가 많아지게 된다. 인덱스가 많으면, 갱신 성능이 나빠지고 디스크 공간도 많아지므로 인덱스를 효과적으로 사용하려면 정규화가 잘 되어 있어야 한다.

- 장점

검색(특히 select) 은 속도가 무척 빨라질 수 있다./월등히 향상 (단, 항상 그런 것은 아니다)

- 왜 항상 그런게 아니지?

SELECT하는 경우에도 데이터 블록 수와 DB\_FILE\_MULTIBLOCK\_READ\_COUNT 값과 분포도 등에 따라 인덱스가 빠를 경우도 있고 full table scan보다 늦어지는 경우도 있음.

데이터의 종류가 적은 열에는 없는 편이 낫다 (비트리)

인덱스를 위한 쿼리가 거의 사용되지 않는다면 이 인덱스는 오히려 성능에 나쁜 영향을 줄 수 있다

인덱스를 탄다고 해서 무조건 빨라지는게 아니다. 인덱스 손익 분기점이라는게 있는데 테이블이 가지고 있는 전체 데이터양의 10에서 15프로 이내의 데이터가 출력이 될때만 인덱스를 타는게 효율적이고, 그 이상이 될때에는 오히려 풀스캔 하는게 더 빠르다.

그 결과 해당 쿼리의 부하가 줄어들어서, 결국 시스템 전체의 성능이 향상된다.

인덱스를 사용해서 검색을 하게되면 서버 입장에서는 적은 처리량으로 요청한 결과를 얻게 되므로, 다른 요청에 대해서도 많은 일을 할 수 있게 된다. 결과적으로 전체 시스템의 성능이 향상되는 효과도 얻게 된다. 하지만 항상 그런건 아닙니다. +단점

- 단점

INSERT경우에는 인덱스가 정렬이 된 상태에서 저장이 되어야하기때문에 어느 자리에 insert를 할지 찾아서 저장을 한다. 그리고 테이블에만 insert나 update를 하는게 아니라 인덱스도 똑같이 해줘야하기 때문에 두 군데에다 하니까 오히려 느려짐

인덱스가 데이터베이스 공간을 차지해서 추가적인 공간이 필요해지는데, 대략 데이터베이스 크기의 10% 정도의 추가 공간이 필요하다.

처음 인덱스를 생성하는데 시간이 많이 소요될 수 있다

INSERT할 경우, 이에 관계된 인덱스 테이블에서는 데이터를 정렬해야 하므로 전체적인 성능이 저하됨.

인덱스를 재작성해야 할 필요가 있기에 성능에 영향을 끼칠 수 있다.

데이터의 변경 작업이 자주 일어날 경우(UPDATE, DELETE, INSERT)에는 오히려 성능이 많이 나빠질 수 있다. 특히 INSERT 인덱스 분할로 인해 INSERT작업이 일어날 떄 성능이 느려질 수 있다. 비어있는 블록을 한 개 확보한 후에, 그 다음 리프 블록의 데이터를 적절히 나누게 됨. → 새로운 블록 할당+인덱스 분할.

- 왜죠?

인덱스가 있고, 데이터를 많이 삽입하게 되면 데이터 블록이 추가되고 인덱스 분할이 일어나기때문에 성능이 느려집니다

인덱스가 있으면 데이터 블록 추가+인덱스 분할로 인해 성능이 느림

- 인덱스 분할

인덱스는 정렬되어야 한다는 점 때문에 이미 생성된 구조에 새로운 로우가 삽입되면 기존의 위치에 파고 들어감

- 만드는 기준

테이블의 데이터 구성이 어떻게 되었는지, 어떠한 조회를 많이 사용하는지 등에 따라서 인덱스를 생성

어떤 컬럼을 조합해서 인덱스를 작성할 것인지, 즉 어떻게 컬럼을 조합해야 조회, 갱신의 모든 성능을 높일 수 있는지에 대한 논의

- 인덱스를 설계 한다는 것

- 인덱스 삭제 시

데이터를 삭제했을 때 테이블의 로우는 제거되지만 인덱스의 로우는 단지 삭제되었음을 표시하므로 그 자리에 인덱스 로우가 추가되지 않을 경우 저장공간의 낭비뿐 아니라 스캔해야 할 블록이 증가함

WHERE절에 사용되더라도 자주 사용해야 가치가 있다

- 특징

인덱스 컬럼을 기준으로 sorting이 되어서 저장이 되어있음

인덱스에서 먼저 데이터들을 찾은 다음에 그 테이블로 매핑된 곳을 가서 나머지 데이터들을 꺼내오는 방식.

인덱스가 해당 테이블의 데이터 블럭의 주소를 가지고 있음. 블럭에는 테이블의 데이터들이 로우 단위로 저장되어있음

특정 조건부터 스캔하기때문에 WHERE절에 자주 등장하는 컬럼을 인덱스로 설정해주면 효율적

order by절에 자주 등장하는 컬럼을 인덱스로 지정해놓으면 인덱스는 sorting이 되어서 저장되어 있다고 했기 때문에 별도로 order by를 수행할 필요 없이 인덱스에서 바로 꺼내서 출력을 하면 되니까 효율적

단일 컬럼으로 구성할 수도 있지만 여러 컬럼을 조합해서 결합 인덱스로 구성할 수도 있다. 이럴 경우에는 select절에 등장하는 컬럼들을 잘 조합을해서 인덱스로 구성을 해 놓으면 별도로 테이블에가서 데이터를 꺼내올 필요 없이 인덱스만 스캔을해서 바로 출력을 하면 되니까 되게 빠르다.

결합인덱스같은 경우에는 컬럼의 분별력과 선두 위치 여부는 상관이 없습니다. 오히려 분별력이 낮은 컬럼이 선두에 위치하는 것이 index skip scan 측면에선 더 유리

결합인덱스의 경우 상황에 따라 순서가 달라지지만 단일 인덱스는 확실히 분별력이 높은 컬럼으로 설정하는게 유리

- 애써 인덱스를 생성해놓고 SQL을 잘못 작성을해서 무용지물이 되는 경우도 있다.

1. 인덱스 컬럼올 가공

ex) WHERE SUBSTR([ORDER.NO](http://order.no/), 1, 4) = '2019'

> WHERE ORDER NO LIKE '2019%'

2. 인덕스 컬럼의 묵시적 형변환 (같은 타입으로 비교해야 함)

ex) WHERE REG.DATE= '20190730'

WHERE REG DATE= TO DATE('20190730'. 'YYYYMMDD ')

3. 인덕스 컬럼 부정형 비교

ex) WHERE MEM.TYPE ! = 'lO '

) WHERE MEM.TYPE IN ('20'. '30')

4. LIKE 연산자 샤용 시 %가 앞에 위지

5. OR 조건 사용→ UNION ALL로 대체

- 인덱스 스캔방식

index full scan - 어떤 시작점부터 특정범위를 스캔하는 방식

index skip scan

index fast full scan

인덱스는 열 단위에 생성된다

WHERE절에 사용되는 열에 인덱스를 만들어야 한다

데이터의 중복도가 높은 열은 인덱스를 만들어도 별 효과가 없다. 비트맵이냐 B-Tree에 따라 다름

JOIN에 자주 사용되는 열에는 인덱스를 생성해 주는 것이 좋다

INSERT UPDATE DELETE가 얼마나 자주 일어나는 지를 고려해야 한다

사용하지 않는 인덱스는 제거하자

오라클책

일반적으로 테이블 전체로서의 15% 의 데이터를 줘야 할 때 인덱스를 생성 한다

물론 15%는 정해진 것은 아니며 케이블 건수 데이터 분포 정도에 따라 달라진다

테이블 건수가 적다면 구지 인덱스를 만들 필요가 없다

데이터 추출을 위해 테이블이나 인덱스를 탐색하는 것을 스킨이라고 하는데 테이블 건수가 적으면 인덱스를 경리 하기 보다 테이블 전체를 스캔 하는 것이 빠르다

데이터에 유일성 정도가 좋거나 범위가 넓은 값을 가진 컬럼을 index로 만드는 것이 좋다

NULL이 많이 포함된 칼럼은 index 컬럼으로 만들기 적당치 않다

결합 인덱스를 만들 때는 칼럼의 순서가 중요하다

보통 자주 사용되는 컬럼을 순서상 앞에 두는 것이 좋다

테이블에 만들 수 있는 index 수의 제한은 없으나 너무 많이 만들면 오히려 성능 부하가 발생한다

- 결과

인덱스는 한 번 생성했다고 내버려 두는 것이 아니라 잘 활용되는 지를 살펴서 활용이 되지않으면 삭제하고, 사용하는 인덱스는 주기적으로 ALTER INDEX 인덱스이름 REBUILD로 인덱스를 재구성해서 시스템의 성능을 최상으로 유지하는 방법

- OLTP On Line Transaction Processing

INSERT UPDATE DELETE 가 실시간으로 자주 발생하므로 꼭 필요한 인덱스만 최소로 생성하는 것이 바람직하다

- OLAP On Line Analytical Processing

INSERT UPDATE DELETE 가 별로 사용될 일이 없으므로 되도록 인덱스를 많이 만들어도 별 문제가 되지 않는다

인덱스를 생성하고 인덱스를 사용하는 SQL을 만들어 사용한다면, 기존보다 아주 빠른 응답 속도를 얻을 수 있다. 또한 서버 입장에서는 적은 처리량으로 요청한 결과를 얻게 되므로, 다른 요청에 대해서도 많은 일을 할 수 있게 된다. 결과적으로 전체 시스템의 성능이 향상되는 효과도 얻게 된다.

테이블 생성 시에 제약 조건 Primary Key 또는 Unique를 사용하면 자동으로 인덱스가 자동 생성된다.

B-Tree 구조는 데이터를 검색(SELECT)할 때 아주 뛰어난 성능을 발휘 함.

인덱스는 여러 개 생성할 수가 있다. 하지만, 함부로 남용할 경우에는 오히려 시스템 성능을 떨어뜨리는 결과를 초래할 수 있으므로, 꼭 필요한 열에만 생성하는 것이 좋다.

디비에 인덱스 생성해도 인덱스사용하는게 빠른지 그냥 풀스캔이 빠를지 Oracle이 알아서 판단한다

데이터의 종류가 적은 열에는 인덱스를 생성하지 않는 편이 낫다.(선택도가 나쁜 데이터)

중복도가 높은 열에는 B-Tree 이ㄴ덱스를 만들어도 거의 사용하지 않는다 비트맵 인덱스는 비트라는 최소 딘위를 이용해서 데이터 깂을 저장하기 때문에 데이터의 중복도가 높은 경우에 사용하면 효율적이다

- 정규화

- 정의

데이터가 꼬이는 것을 막기 위해 테이블을 잘게 나누는 것

- 목적

중복을 최소화, 이상현상 제거 ( 삽입, 삭제, 변경 )

불필요한 데이터를 제거하고 데이터 저장을 논리적으로 한다.

- 제1정규형 원자성

원자성 - 모든 속성은 반드시 하나의 값만 가져야 한다

다중값을 가진다(sns컬럼에 인스타그램, 트위터, 페이스북, 유튜브) → 제 1 정규형 위배

반복 그룹을 가진다(SNS1 SNS2 SNS3 SNS4 SNS5) → 제 1 정규형 위배

- 제2정규형 (2NF) 부분 종속 기본키가 2개 이상의 속성으로 이루어진 경우

부분 종속 - 모든 속성은 반드시 모든 기본키에 종속되어야 한다. (기본키 일부에만 종속되어서는 안됨) → 테이블 분리

주문번호 | 음료코드 | 주문수량 | 음료명

음료명은 음료코드 속성에만 종속된다.

- 주문이 발생하지 않으면 음료 입력 불가 → 입력 이상

- 음료명이 변경될 경우 해당되는 주문 ROW UPDATE 필요 → 수정 이상

- 음료 삭제 시 주문까지 삭제 → 삭제 이상

→ 데이터 이상 발생

종속 관계가 있는 속성ㅁ나 추출해서 테이블 분리

결과 - 주문과 관계없이 음료 데이터 입력, 수정 삭제가능

- 제3 정규형(3NF) 이행 종속 기본키를 제외한 컬럼 간 종속성 제거

기본키가 아닌 모든 속성간에는 서로 종속될 수 없다. - 테이블 분리

직원번호 | 직원명 | 소속사번호 | 소속사명

소속사명은 소속사 번호 속성에 종속된다

→ 데이터 이상 현상 발생

2NF와 마찬가지로 똑같이 테이블 분리

- BCNF

3차 정규형을 만족하면서 모든 결정자가 후보키 집합에 속한 정규형

3차 정규형으로 해결할 수 없는 이상현상을 해결가능

즉, 함수 종속성이 모두 제거된 상태의 정규형. 이 이상에서 함수 종속성에 의한 무손실 분해를 할 수 없다.

- 반 정규화

- 정의

성능 향상을 위해 정규화를 포기하는 것. 그러나 데이터 무결성이 보장 되지 않으므로 제한적으로 사용해야 한다.

1. 테이블 반정규화 - 테이블 병합/분할/추가

테이블 병합 : 비즈니스 로직 상 JOIN되는 경우가 많아 통합하는 것이 성능 측면에서 유리할 경우 고려.

1. 1:1 관계 테이블 병합

2. 1:M 관계 테이블 병합

3. 슈퍼 서브 타입 테이블 병합

테이블 분할

1. 수직 분할 : 컬럼 단위로 테이블을 1:1 분리

2. 수평 분할 : ROW 단위로 테이블 분리

테이블 추가

1. 중복 테이블 추가

타 엄무 또는 타 서버에 있는 테이블과 동일한 구조의 테이블 추가, 원격 JOIN 방지

2. 통계 테이블 추가

통계값을 미리 계산해서 저장하는 테이블 추가

3. 이력 테이블 추가

마스터 테이블에 존재하는 ROW를 트랜잭션 발생 시점에 따라 복사해두는 테이블 추가

4. 부분 테이블 추가

자주 조회되는 컬럼들만 별도로 모아놓은 테이블 추가 → 인싸들만 모아놓은 테이블

2. 컬럼 반정규화 - 중복/파생/이력테이블 컬럼 추가

중복컬럼추가

- JOIN 프로세스를 줄이기 위해 중복 컬럼 추가

- SELECT 비용은 감소하나, UPDATE 비용은 증가

회원 | 주문 | 주문상품 | 배송

회원번호 | 주문번호 |주문번호 | 배송번호

회원명 | 회원번호 | 상품번호 | 주문번호

상품번호

회원번호

파생컬럼 추가

- 계산을 통해 얻어지는 결과값을 테이블에 컬럼으로 저장

\*\*주문 총금액\*\* | 상품번호+주문수량 | 단가

이력 테이블 컬럼 추가

- 이력 테이블에 기능성 컬럼 추가(최신 여부, 시작일/종료일 등)

3. 관계 반정규화 - 중복관계 추가

중복관계 추가

- 데이터 처리를 위해 여러 경로를 거쳐야 할 경우 관계를 중복시킴으로써 성능 개선

- 트랜잭션

- 정의

쿼리를 하나의 묶음 처리해서 만약 중간에 실행이 중단됐을 경우, 처음부터 다시 실행하는 ROLLBACK을 수행하고, 오류없이 실행을 마치면 COMMIT을 하는 실행 단위.

- 사용 이유

DB 서버에 여러 개의 클라이언트가 동시에 액세스 하거나 응용프로그램이 갱신을 처리하는 과정에서 중단될 수 있는 경우 등 \*\*데이터 부정합을 방지\*\*하고자 할 때 사용합니다.

부정합이 발생하지 않으려면 프로세스를 병렬로 처리하지 않도록 하여 한 번에 하나의 프로세스만 처리하도록 하면 되는데, 이는 효율이 너무 떨어집니다.

즉, 병렬로 처리할 수 밖에 없는 현실적인 상황으로 인해 부정합을 방지하고자 트랜잭션을 사용하는 것입니다.

- 원자성

트랜잭션이 데이터베이스에 모두 반영되던가, 아니면 전혀 반영되지 않아야 한다는 것

- 일관성

트랜잭션의 작업 처리 결과가 항상 일관성이 있어야 한다는 것.

트랜잭션이 진행되는 동안에 데이터베이스가 변경 되더라도 업데이트된 데이터베이스로 트랜잭션이 진행되는것이 아니라, 처음에 트랜잭션을 진행 하기 위해 참조한 데이터베이스로 진행되어야 한다.

- 독립성

둘 이상의 트랜잭션이 동시에 실행되고 있을 경우, 어떤 하나의 트랜잭션이라도 다른 트랜잭션의 연산에 끼어 들수 없다는 점

하나의 특정 트랜잭션이 완료될때까지, 다른 트랜잭션이 특정 트랜잭션의 결과를 참조할 수 없다.

- 지속성

트랜잭션이 성공적으로 완료되었을 경우에 결과는 영구적으로 반영되어야 한다는 점

- 옵티마이저(Optimizer)

SQL을 처리하는 최저비용의 경로를 생성해주는 DBMS 내부 핵심엔진

옵티마이저는 많은 조인 테크닉을 가지기 때문에 조인 형태로 변환했을 때 더 나은 실행계획을 찾을 가능성이 높다.

- SQL 개발자가 SQL을 작성하여 실행할 때, 옵티마이저는 SQL을 어떻게 실행할 것인지를 계획하게 된다.

- SQL 실행 계획(Execution Plan)을 수립하고 SQL을 실행한다.

- 옵티마이저는 SQL의 실행 계획을 수립하고 SQL을 실행하는 데이터베이스 관리 시스템의 소프트웨어이다.

- 동일한 결과가 나오는 SQL도 어떻게 실행하느냐에 따라서 성능이 달라지기에, SQL 성능에 옵티마이저는 아주 중요한 역할을 한다.

리액트

코드 스플릿

코드 스플릿팅이라고 하는 것은 webpack, rollup, browserify와 같은 모듈 번들러를 이용하여 만들어진 하나의 번들 파일을 여러 개의 번들 파일로 나누는 것을 의미합니다.

그리고 하나의 번들 파일을 여러 개의 번들 파일로 나누는 이유는 더 빠른 속도로 화면을 로드하기 위해서입니다.

Class component, Functional component

### 클래스형 :

* state, lifeCycle 관련 기능사용 가능하다.
* 메모리 자원을 함수형 컴포넌트보다 조금 더 사용한다.
* 임의 메서드를 정의할 수 있다.
* constructor 안에서 this.state 초기 값 설정 가능
* this.props로 통해 값을 불러올 수 있다.

### 함수형 :

* state, lifeCycle 관련 기능사용 불가능 [Hook을 통해 해결 됨]
* 메모리 자원을 함수형 컴포넌트보다 덜 사용한다.
* 컴포넌트 선언이 편하다.
* 함수형 컴포넌트에서는 useState 함수로 state를 사용한다.
* useState 함수를 호출하면 배열이 반환되는데 첫 번째 원소는 현재 상태
* 두 번째 원소는 상태를 바꾸어 주는 함수
* props를 불러올 필요 없이 바로 호출 할 수 있다

restful api

REST API(RESTful API, 레스트풀 API)란 REST 아키텍처의 제약 조건을 준수하는 애플리케이션 프로그래밍 인터페이스를 뜻합니다. REST는 Representational State Transfer의 줄임말입니다. Rest API의 개념 및 서버, 구현 등을 이 페이지에서 설명합니다.

API가 RESTful로 간주되려면 다음 기준을 따라야 합니다.

* 클라이언트, 서버 및 리소스로 구성되었으며 요청이 HTTP를 통해 관리되는 클라이언트-서버 아키텍처
* [스테이트리스(stateless)](https://www.redhat.com/ko/topics/cloud-native-apps/stateful-vs-stateless) 클라이언트-서버 커뮤니케이션: 요청 간에 클라이언트 정보가 저장되지 않으며, 각 요청이 분리되어 있고 서로 연결되어 있지 않음
* 클라이언트-서버 상호 작용을 간소화하는 캐시 가능 데이터
* 정보가 표준 형식으로 전송되도록 하기 위한 구성 요소 간 통합 인터페이스. 여기에 필요한 것은 다음과 같습니다.
  + 요청된 리소스가 식별 가능하며 클라이언트에 전송된 표현과 분리되어야 합니다.
  + 수신한 표현을 통해 클라이언트가 리소스를 조작할 수 있어야 합니다(이렇게 할 수 있는 충분한 정보가 표현에 포함되어 있기 때문).
  + 클라이언트에 반환되는 자기 기술적(self-descriptive) 메시지에 클라이언트가 정보를 어떻게 처리해야 할지 설명하는 정보가 충분히 포함되어야 합니다.
  + 하이퍼미디어: 클라이언트가 리소스에 액세스한 후 하이퍼링크를 사용해 현재 수행 가능한 기타 모든 작업을 찾을 수 있어야 합니다.
* 요청된 정보를 검색하는 데 관련된 서버(보안, 로드 밸런싱 등을 담당)의 각 유형을 클라이언트가 볼 수 없는 계층 구조로 체계화하는 계층화된 시스템.
* 코드 온디맨드(선택 사항): 요청을 받으면 서버에서 클라이언트로 실행 가능한 코드를 전송하여 클라이언트 기능을 확장할 수 있는 기능.

Babel

## Babel 이란 ?

* Babel is a Javascript compiler
* Babel은 Javascript 컴파일러이다.
* 정확히 Babel은 Javascript로 결과물을 만들어주는 컴파일러이다.

@babel/preset  
º Babel은 그 자체로는 사실상 하는 일이 없고 plugin에 의해서 실행된다.  
º 이른 위해 수많은 plugin들이 존재하는데 preset으로 plugin들을 묶어서 사용할 수 있다.  
º  @babel/preset-env  
  ■ ECMAScript2015+를 변환할 때 사용한다.  
  ■ 바벨 7 이전 버전에는 연도별로 각 프리셋을 제공하지만, 지금은 env 하나로 합쳐졌다

@babel/preset-react, @babel/preset-typescript  
- react, typescript 문법을 javascript로 바꾸기 위한 plugin들이 묶여있다.  
  
 targets: 프로젝트를 지원 / 대상으로하는 환경을 결정할 수 있다.

 useBuiltlns : poly-fill 을 처리하는 방법을 결정한다.

 corejs: @babel/polyfill이 deprecated 되면서 대체로 corejs 옵션을 사용한다.

 shippedProposals: 브라우저에 제공된 기능 제안에 대한 지원을 활성화

 modules: ES 모듈 구문을 다른 모듈 유형으로 변환 할 수 있다.

동기방식 (Synchronous)

요청을 보낸 후 응답(=결과)를 받아야지만 다음 동작이 이루어지는 방식이다.

어떠한 일을 처리할 동안 다른 프로그램은 정지한다.

실제 cpu가 느려지는 것은 아니지만 시스템의 전체적인 효율이 저하된다고 할 수 있다.

비동기 방식 (Asynchronous)

요청을 보낸 후 응답(=결과)와는 상관없이 다음방식이 동작하는 방식이다.

결과가 주어지는데 시간이 걸리더라도 그 시간 동안 다른 작업을 할 수 있으므로 자원을 효율적으로 사용할 수 있다.

비동기식은 비동기식 처리를 요청할 때 할일 이 끝난 후 처리결과를 알려주는 콜백이라는 함수를 함께 알려준다.

비동기식 처리를 요청하였을 때 호출받은 함수는 바로 응답(=확인)을 수행한다.

이 응답은 처리 결과에 대한 응답이 아니라 요청에 대한 확인 동작일 뿐이다.

호출받은 함수는 처리가 끝나면 요청한 함수를 호출하여 처리 결과를 전달하게 된다.

이러한 함수 호출의 흐름은 사용자가 아닌 일을 마친 시스템이 호출하는 형태이기 때문에 콜백이라고 불린다.

이미 응답을 했기 때문에 처리결과를 함수 호출이라는 형태로 전달하는 것이다.

비동기 방식은 DOS같은 단일 운영체제에서는 불가능하며 windows 같은 multitask 환경에서만 가능하다.

(ajax에서는 success, error, complete 을 콜백함수라고 할 수 있다.)

ssr

SSR 방식은 서버에서 렌더링을 마치고, 완전하게 만들어진 HTML 파일을 브라우저로 전달한 후 렌더링하는 방식이다. 새로운 페이지로 이동할 때마다 서버에 페이지를 요청하고 새로고침이 되어 깜빡이는 현상이 발생한다.

장점

CSR에 비해 다운 받는 파일이 많지 않아 초기 로딩 속도가 빠르기 때문에 사용자가 빨리 컨텐츠를 볼 수 있다.

HTML에 대한 정보가 처음부터 포함되어 있어서 모든 검색엔진에 대한 SEO(검색 엔진 최적화)가 가능하다.

단점

매번 페이지를 요청할 때마다 새로고침 되기 때문에 UX가 다소 떨어진다.

페이지를 이동할 때마다 매번 서버에 요청을 하기 때문에 서버의 부하가 커진다.

csr

전통적인 방식으로는 SPA가 사용하는 렌더링 방식이다. 최초 로딩 시 브라우저가 서버에 HTML을 비롯한 CSS, JavaScript 등 각종 리소스들을 받아오는 방식이 클라이언트 사이드 렌더링 방식이다.

각종 리소스들을 받아오면 사용자의 상호작용에 따라 JavaScript를 통해 동적으로 렌더링한다. 이후 필요에 따라 데이터를 서버에 요청해서 받아오게 된다.

장점

초기 요청을 제외하고는 SSR에 비해 빠른 페이지 전환 속도로 더 나은 UX를 제공한다.

서버에 요청하는 횟수가 적기 때문에 서버 부담이 SSR에 비해 적다.

단점

최초 로딩 시 모든 리소스들을 받아와야 하기 때문에 초기 로딩 속도가 느리다.

처음에는 HTML이 비어있어 크롤러가 데이터를 수집할 수 없기 때문에 SEO(검색 엔진 최적화) 문제가 발생한다.

쿠키나 localStorage에서 사용자에 대한 정보를 저장해야 하기 때문에 XSS 공격에 취약하다

프레임워크와 라이브러리의 차이점

라이브러리와 프레임워크의 차이는 제어 흐름에 대한 주도성이 누구에게/어디에 있는가에 있습니다.

즉, 어플리케이션의 Flow(흐름)를 누가 쥐고 있느냐에 달려 있습니다.

프레임워크는 전체적인 흐름을 스스로가 쥐고 있으며 사용자는 그 안에서 필요한 코드를 짜 넣으며 반면에 라이브러리는 사용자가 전체적인 흐름을 만들며 라이브러리를 가져다 쓰는 것이라고 할 수 있습니다.

다시 말해, 라이브러리는 라이브러리를 가져다가 사용하고 호출하는 측에 전적으로 주도성이 있으며 프레임워크는 그 틀안에 이미 제어 흐름에 대한 주도성이 내재(내포)하고 있습니다.

프레임워크는 가져다가 사용한다기보다는 거기에 들어가서 사용한다는 느낌/관점으로 접근할 수 있습니다.

라이브러리를 사용하는 애플리케이션 코드는 애플리케이션 흐름을 직접 제어합니다.

단지 동작하는 중에 필요한 기능이 있을 때 능동적으로 라이브러리를 사용할 뿐입니다.

반면에 프레임워크는 거꾸로 애플리케이션 코드가 프레임워크에 의해 사용되는 것입니다.

보통 프레임워크 위에 개발한 클래스를 등록해두고, 프레임워크가 흐름을 주도하는 중에 개발자가 만든 애플리케이션 코드를 사용하도록 만드는 방식입니다.

프레임워크에는 분명한 제어의 역전 개념이 적용되어 있어야 합니다.

애플리케이션 코드는 프레임워크가 짜놓은 틀에서 수동적으로 동작해야 합니다.

프레임워크의 장점

체계적인 코드관리로 유지보수가 용이하다.

기본설계 및 기능 라이브러리를 제공하여 개발 생산성이 높다

코드에 대한 재사용성이 높다

추상화된 코드 제공을 통해 확장성이 좋다

프레임워크의 단점

학습난이도가 높다.

기본 설계된 구조에 의해 자유로운 개발에 한계가 있다.

사용하지 않는 기능에 대한 라이브러리가 포함될 수 있다.

제공되는 기능만큼 프로젝트의 용량이 증가한다.

Spring boot 장점

1. 라이브러리 관리 자동화.

스프링부트의 Starter 라이브러리를 등록해서 라이브러리 의존성을 간단히 관리할 수 있다.

2. 라이브러리 버전 자동관리

기존 스프링 라이브러리는 버전을 직접 입력해야 했지만, 스프링 부트는 pom.xml에 스프링 부트 버전을 입력해놓으면 spring 라이브러리 뿐만 아니라 third party라이브러리들도 호환되는 버전으로 알아서 다운로드 및 관리해준다.

3. 설정 자동화

스프링 부트는 @EnableAutoConfiguration 어노테이션을 선언해서 스프링에서 자주 사용했던 설정들을 알아서 등록해준다. 이 기능이 스프링 부트의 마법이라고 불린다고 한다.

4. 내장Tomcat.

스프링 부트는 WAS인 Tomcat을 내장하고 있다. @SpringBootApplication 어노테이션이 선언되어 있는 클래스의 mail()메소드를 실행하는 것만으로 서버를 구동시킬 수 있다.

내장 톰캣을 사용하기 위한 별도 설정없이 web starter 의존성만 추가해주면 된다.

5. 독립적으로 실행가능함 JAR

웹 프로젝트라면 war파일로 패키징 해야하는데 스프링 부트는 내장 톰캩ㅅ을 지원하기 때문에 jar 파일로 패키징해서 웹 애플리케이션을 실행시킬 수 있다 !!!!

● 웹소켓(WebSocket)의 개념

웹소켓은 HTTP와 같이 약속입니다.

Transport protocol의 일종으로 서버와 클라이언트 간의 효율적인 양방향 통신을 실현하기 위한 구조입니다.

웹소켓은 단순한 API로 구성되어있으며, 웹소켓을 이용하면 하나의 HTTP 접속으로 양방향 메시지를 자유롭게 주고받을 수 있습니다.

위 배경에서 웹소켓이 나오기 이전에는 모두 클라이언트의 요청이 없다면, 서버로부터 응답을 받을 수 없는 구조였습니다.

웹소켓은 이러한 문제를 해결하는 새로운 약속이었습니다.

웹소켓에서는 서버와 브라우저 사이에 양방향 소통이 가능합니다. 브라우저는 서버가 직접 보내는 데이터를 받아들일 수 있고, 사용자가 다른 웹사이트로 이동하지 않아도 최신 데이터가 적용된 웹을 볼 수 있게 해줍니다. 웹페이지를 ‘새로고침’하거나 다른 주소로 이동할 때 덧붙인 부가 정보를 통해서만 새로운 데이터를 제공하는 웹서비스 환경의 빗장을 본질적으로 풀어준 셈입니다.

● 작동원리

서버와 클라이언트간의 웹소켓 연결을 HTTP프로토콜을 통해 이루어집니다.

연결이 정상적으로 이루어진다면 서버와 클라이언트 간에 웹소켓 연결(TCP/IP기반)이 이루어지고 일정 시간이 지나면HTTP연결은 자동으로 끊어집니다.

기본적으로 웹소켓 API는 아주 간단한 기능들만을 제공하기 때문에 대부분의 경우 SockJS나 Socket.IO같은 오픈 소스 라이브러리를 많이 사용하고 있으며 메시지 포맷 또한 STOMP같은 프로토콜을 같이 이용합니다.

스톰프(STOMP)

STOMP는 topic, subscribe 관계가 핵심철학이다. 즉 컨트롤러는 "/topic" prefix를 이용해서 명시적으로 브로커에게 알려주었고 그 뒤에 a라는 topic을 전달해주었다. 그러면 브로커는 a 토픽을 구독하고 있는 모든 클라이언트들에게 메세지를 전달하게 된다. 다시 말해 컨트롤러가 STOMP destination을 리턴했고 브로커는 이 destination을 구독하고 있는 구독자들에게 메세지를 전달하게 된다.

HTTP(HyperText Transfer Protocol, 문화어: 초본문전송규약, 하이퍼본문전송규약)는 W3 상에서 정보를 주고받을 수 있는 프로토콜이다. 주로 HTML 문서를 주고받는 데에 쓰인다. 주로 TCP를 사용하고 HTTP/3 부터는 UDP를 사용하며, 80번 포트를 사용한다. 1996년 버전 1.0, 그리고 1999년 1.1이 각각 발표되었다.

HTTP는 클라이언트와 서버 사이에 이루어지는 요청/응답(request/response) 프로토콜이다. 예를 들면, 클라이언트인 웹 브라우저가 HTTP를 통하여 서버로부터 웹페이지(HTML)나 그림 정보를 요청하면, 서버는 이 요청에 응답하여 필요한 정보를 해당 사용자에게 전달하게 된다. 이 정보가 모니터와 같은 출력 장치를 통해 사용자에게 나타나는 것이다.

[ HTTP(Hyper Text Transfer Protocol)란? ]

HTTP(Hyper Text Transfer Protocol)란 서버/클라이언트 모델을 따라 데이터를 주고 받기 위한 프로토콜이다.

즉, HTTP는 인터넷에서 하이퍼텍스트를 교환하기 위한 통신 규약으로, 80번 포트를 사용하고 있다. 따라서 HTTP 서버가 80번 포트에서 요청을 기다리고 있으며, 클라이언트는 80번 포트로 요청을 보내게 된다.

HTTP는 1989년 팀 버너스 리(Tim Berners Lee)에 의해 처음 설계되었으며, WWW(World-Wide-Web) 기반에서 세계적인 정보를 공유하는데 큰 역할을 하였다.

[ HTTP의 구조 ]

HTTP는 애플리케이션 레벨의 프로토콜로 TCP/IP 위에서 작동한다. HTTP는 상태를 가지고 있지 않는 Stateless 프로토콜이며 Method, Path, Version, Headers, Body 등으로 구성된다.

[ HTTPS(Hyper Text Transfer Protocol Secure)란? ]

HyperText Transfer Protocol over Secure Socket Layer, HTTP over TLS, HTTP over SSL, HTTP Secure 등으로 불리는 HTTPS는 HTTP에 데이터 암호화가 추가된 프로토콜이다. HTTPS는 HTTP와 다르게 443번 포트를 사용하며, 네트워크 상에서 중간에 제3자가 정보를 볼 수 없도록 공개키 암호화를 지원하고 있다.

[ 공개키/개인키 ]

HTTPS는 공개키/개인키 암호화 방식을 이용해 데이터를 암호화하고 있다. 공개키와 개인키는 서로를 위한 1쌍의 키이다.

공개키: 모두에게 공개가능한 키

개인키: 나만 가지고 알고 있어야 하는 키

공개키와 개인키로 암호화하면 다음과 같은 효과를 얻을 수 있다.

공개키 암호화: 공개키로 암호화를 하면 개인키로만 복호화할 수 있다. -> 개인키는 나만 가지고 있으므로, 나만 볼 수 있다.

개인키 암호화: 개인키로 암호화하면 공개키로만 복호화할 수 있다. -> 공개키는 모두에게 공개되어 있으므로, 내가 인증한 정보임을 알려 신뢰성을 보장할 수 있다.

[ HTTP와 HTTPS]

HTTP는 암호화가 추가되지 않았기 때문에 보안에 취약한 반면, HTTPS는 안전하게 데이터를 주고받을 수 있다. 하지만 HTTPS를 이용하면 암호화/복호화의 과정이 필요하기 때문에 HTTP보다 속도가 느리다.(오늘날에는 거의 차이를 못느낄 정도이다.) 또한 HTTPS는 인증서를 발급하고 유지하기 위한 추가 비용이 발생하다.

그렇다면 언제 HTTP를 쓰고, 언제 HTTPS를 쓰는 것이 좋겠는가?

개인 정보와 같은 민감한 데이터를 주고 받아야 한다면 HTTPS를 이용해야 하지만, 단순한 정보 조회 등만을 처리하고 있다면 HTTP를 이용하면 된다.

API란?

API는 애플리케이션 소프트웨어를 구축하고 통합하기 위한 정의 및 프로토콜 세트로, 애플리케이션 프로그래밍 인터페이스(Application Programming Interface)를 나타냅니다.

API를 사용하면 구현 방식을 알지 못해도 제품 또는 서비스가 서로 커뮤니케이션할 수 있으며 애플리케이션 개발을 간소화하여 시간과 비용을 절약할 수 있습니다. 새로운 툴과 제품을 설계하거나 기존 툴과 제품을 관리하는 경우 API는 유연성을 제공하고 설계, 관리, 사용 방법을 간소화하며 혁신의 기회를 제공합니다.

ORM이란

Object Relational Mapping, 객체-관계 매핑

객체와 관계형 데이터베이스의 데이터를 자동으로 매핑(연결)해주는 것을 말한다

MyBatis 장단점

장점

1) JPA에 비해 쉽다.

2) SQL 쿼리를 그대로 사용하기 때문에 복잡한 Join, 튜닝 등을 편하게 작성할 수 있다.

3) SQL 쿼리의 세부 내용을 변경 시 수월하다.

4) 동적 쿼리 사용 시 JPA보다 간편하게 구현 가능하다.

\*동적 쿼리: 상황에 따라 분기처리를 통해 SQL문을 동적으로 만드는 기법

단점

1) DB 설정 변경 시 수정할 부분이 많다.

2) Mapper 작성부터, 인터페이스, 모델 설계까지 JPA보다 많은 설계와 파일, 로직이 필요하다.

-> Mapper에서 select id로 사용할 인터페이스 설계, 쿼리에 파라미터로 # 혹은 $로 파라미터 바인딩을 위한 model 정의, resultType/ParameterType으로 전달할 model 정의가 필요하다.

3) 특정 DB에 종속적이다. DB가 바뀌면 DB 문법에 맞게 mapper를 전부 수정해야한다.

4) 쿼리에 로직을 녹여야 하는데 유지보수 하기가 힘들고 테스트도 까다롭다.

DDL(Data Definition Language)  - 데이터 정의어 란? 데이터베이스를 정의하는 언어이며, 데이터리를 생성, 수정, 삭제하는 등의 데이터의 전체의 골격을 결정하는 역할을 하는 언어 입니다.

DML(Data Manipulation Language) - 데이터 조작어란? 정의된 데이터베이스에 입력된 레코드를 조회하거나 수정하거나 삭제하는 등의 역할을 하는 언어를 말합니다.

DCL(Data Control Language) - 데이터베이스에 접근하거나 객체에 권한을 주는등의 역할을 하는 언어를 입니다.

저장 프로시저 (Stored Procedure)

프로시저란 SQL Server에서 제공하는 프로그래밍 기능입니다.

쿼리문을 마치 하나의 메서드 형식으로 만들고 어떠한 동작을 일괄적으로 처리하는 용도로 사용됩니다.

여러 개의 칼럼을 조회하고 여러 개의 테이블을 조인하고 거기에 조건까지 넣어줌으로써 하나의 쿼리를 만드는데

엄청나게 긴 쿼리문이 생기게 됩니다.

이러한 여러개의 쿼리를 사용할 때마다 긴 쿼리문을 써야 한다면 엄청나게 불편할 텐데 이를 프로시저에 저장하고

저장된 프로시저를 호출하여 프로그래밍을 하는것이 훨씬 효율적입니다.

저장 프로시저의 장점

저장 프로시저를 사용하게되면 여러 가지 장점들이 있습니다.

하나의 요청으로 여러 SQL문을 실행 할 수 있습니다.

네트워크 소요 시간을 줄일 수 있습니다.

만약 동일한 쿼리를 1000번 2000번 호출하는 것보다 SP를 이용해서 구현한다면 SP를 호출할 때 한 번만 네트워크를 경유하기 때문에 네트워크 소요시간을 줄이고 성능을 개선할 수 있습니다.

개발 업무를 구분해 개발 할 수 있습니다.

순수한 애플리케이션만 개발하는 조직과 DBMS 관련 코드를 개발하는 조직이 따로 있다면, DBMS 개발하는 조직에서는 데이터베이스 관련 처리하는 SP를 만들어 API처럼 제공하고 애플리케이션 개발자는 SP를 호출해서 사용하는 형식으로 역할을 구분하여 개발이 가능합니다.

저장 프로시저의 단점

저장 프로시저의 장점에 대해 알아보았다면 이제 단점에 대해서도 알아볼 필요가 있습니다.

처리 성능이 낮습니다.

문자나 숫자 연산에 저장 프로시저를 사용한다면 오히려 C나 JAVA보다 느린 성능을 보여줍니다.

디버깅이 어렵습니다.

DB 확장이 매우 힘듭니다.

서비스 사용자가 많아져 서버수를 늘려야할 때, DB 수를 늘리는 것이 더 어렵습니다.

서비스 확장을 위해 서버수를 늘릴경우 DB 수를 늘리는 것보다 WAS의 수를 늘리는 것이 더 효율적이기 때문에 대부분의 개발에서 DB에는 최소의 부담만 주고 대부분의 로직은 WAS에서 처리할 수 있게 합니다.