제12장

파이프라인 아키텍처 스타일

소프트웨어 아키텍처의 기본 스타일 중 하나는 파이프라인 아키텍처(파이프라인 및 필터 아키텍처라고도 함)입니다. 개발자와 아키텍트가 기능을 개별 부분으로 분리하기로 결정한 순간부터 이 아키텍처 스타일이 자리 잡았습니다.
대부분의 개발자는 이 아키텍처를 Bash 및 Zsh 와 같은 Unix 터미널 셸 언어의 기본 원리로 알고 있습니다 .

함수형 프로그래밍 언어를 사용하는 개발자라면 언어 구성 요소와 이 아키텍처의 요소 사이에 유사점을 발견할 수 있을 것입니다. 실제로 MapReduce 프로그래밍 모델을 사용하는 많은 도구들이 이 기본 구조를 따릅니다. 이 장의 예제는 파이프라인 아키텍처 스타일의 저수준 구현을 보여주지만, 고수준 비즈니스 애플리케이션에도 적용할 수 있습니다.

위상수학

파이프라인 아키텍처의 토폴로지는 파이프와 필터라는 두 가지 주요 구성 요소로 이루어져 있습니다. 필터는 시스템 기능을 포함하고 특정 비즈니스 기능을 수행하며, 파이프는 체인의 다음 필터(또는 필터들)로 데이터를 전달합니다. 이들은 특정 방식으로 상호 작용하며, 파이프는 그림 12-1에서 보는 바와 같이 필터 간에 단방향, 일반적으로 지점 간 통신을 형성합니다 .

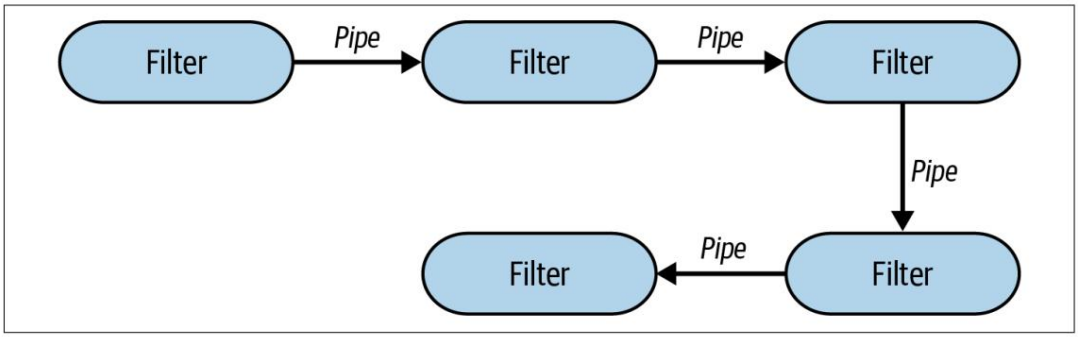

그림 12-1. 파이프라인 아키텍처의 기본 토폴로지

따라서 파이프라인 아키텍처의 동형적인 "형태"는 단일 배포 단위이며, 기능은 단방향 파이프로 연결된 필터 내에 포함됩니다.

스타일 사양

파이프라인 아키텍처의 대부분은 모놀리식 구조이지만, 각 필터(또는 필터 세트)를 서비스로 배포하여 각 서비스에 대한 동기 또는 비동기 원격 호출을 지원하는 분산 아키텍처를 구축할 수도 있습니다. 배포 토폴로지와 관계없이, 이 아키텍처는 필터와 파이프라는 두 가지 구성 요소로만 이루어져 있으며, 다음 섹션에서 자세히 설명합니다.

필터

필터는 다른 필터와 독립적인 자체 기능 단위입니다. 일반적으로 상태를 저장하지 않으며, 하나의 작업만 수행해야 합니다. 복합적인 작업은 일반적으로 단일 필터보다는 여러 필터를 순차적으로 사용하여 처리합니다.

필터는 여러 클래스 파일로 구현될 수 있기 때문에 아키텍처의 구성 요소로 간주됩니다(8 장 참조). 필터가 단순하고 단일 클래스 파일로만 구현되더라도 여전히 구성 요소입니다.

파이프라인 아키텍처 스타일에는 네 가지 유형의 필터가 있습니다.

프로듀서 필

터는 프로세스의 시작점으로, 출력 전용입니다. 소스라고도 합니다. 사용자 인터페이스와 시스템에 대한 외부 요청은 모두 프로듀서 필터의 예입니다.

트랜스포머 필터는

입력을 받아 선택적으로 데이터의 일부 또는 전체를 변환한 후, 변환된 데이터를 출력 파이프로 전달합니다. 함수형 프로그래밍을 선호하는 사람들은 이 기능을 맵(map)이라고 부릅니다. 트랜스포머 필터는 예를 들어 데이터를 개선하거나, 변환하거나, 특정 계산을 수행할 수 있습니다.

테스터

필터는 입력을 받아 하나 이상의 기준에 따라 테스트한 다음, 선택적으로 테스트 결과에 따라 출력을 생성합니다. 함수형 프로그래머라면 이를 reduce 명령어와 유사하다고 생각할 것입니다. 테스터 필터는 모든 데이터가 유효하고 올바르게 입력되었는지 확인하거나, 처리를 계속 진행할지 여부를 결정하는 스위치 역할을 할 수 있습니다(예: "주문 금액이 5달러 미만이면 다음 필터로 데이터를 전달하지 마십시오").

소비자

파이프라인 흐름의 종료 지점인 소비자 필터는 때때로 파이프라인 프로세스의 최종 결과를 데이터베이스에 저장하거나 UI 화면에 표시합니다.

파이프와 필터의 단방향적인 특성과 단순함은 구성 요소의 재사용을 용이하게 합니다. 많은 개발자들이 셸을 사용하면서 이러한 가능성을 발견했습니다. "More Shell, Less Egg" 블로그의 유명한 일화는 이러한 추상화의 강력함을 잘 보여줍니다.

도널드 크누스는 텍스트 파일을 읽고 가장 자주 사용되는 n개의 단어를 찾아 빈도수 순으로 정렬한 목록을 출력하는 텍스트 처리 프로그램을 작성해 달라는 요청을 받았습니다. 그는 10페이지가 넘는 파스칼 코드로 프로그램을 작성하면서 새로운 알고리즘을 설계하고 문서화했습니다. 그런데 더그 맥일로이가 소셜 미디어 게시물에 쉽게 들어갈 수 있을 만큼 작은 셸 스크립트를 만들어 이 문제를 훨씬 효율적으로 해결했습니다.

```
tr -cs A-Za-z '\n' | TR AZ az |
정렬 | 유니크 -c | 정
렬 -rn |
sed ${1}q
```

유닉스 셸을 설계한 사람들조차도 개발자들이 단순하지만 강력한 복합 추상화 구조를 얼마나 창의적으로 활용하는지 보고 놀라곤 합니다.

파이프

이 아키텍처에서 파이프는 필터 간의 통신 채널을 형성합니다. 각 파이프는 일반적으로 단방향이며 지점 간 연결 방식으로, 한 소스에서 입력을 받아 다른 소스로 출력을 전달합니다. 파이프의 페이로드는 모든 데이터 형식을 지원할 수 있지만, 일반적으로 설계자는 고성능을 위해 소량의 데이터를 선호합니다.

필터(또는 필터 그룹)가 분산 방식으로 별도의 서비스로 배포되는 경우, 파이프는 REST, 메시징, 스트리밍 또는 기타 원격 통신 프로토콜을 사용하여 단방향 원격 호출을 수행합니다. 배포 토폴로지가 단일 구조이든 분산 구조이든 관계없이 파이프는 동기식 또는 비동기식으로 작동할 수 있습니다. 단일 구조 배포에서 아키텍트는 필터와의 비동기 통신을 위해 스레드 또는 내장 메시징을 사용합니다.

데이터 토폴로지

대부분의 파이프라인 아키텍처는 모놀리식 형태로 배포되므로 단일 모놀리식 데이터베이스와 연결됩니다. 그러나 항상 그런 것은 아닙니다. 데이터베이스 토폴로지는 이 아키텍처 스타일에서 단일 데이터베이스부터 필터별 데이터베이스까지 매우 다양할 수 있습니다.

그림 12-2 의 예는 특정 운영 특성(예: 응답성 또는 확장성)을 분석하는 프로덕션 환경에서 실행되는 일반적인 연속 적합성 기능(아키텍처 테스트)에 대한 파이프라인 아키텍처를 보여줍니다. '원시 데이터 캡처' 필터는 원시 데이터가 포함된 별도의 데이터베이스를 로드합니다. 이 필터는 파이프를 통해 원시 데이터를 '시계열 선택기' 필터로 보내고, 시계열 선택기 필터는 별도의 데이터베이스에서 구성 정보(예: 분석 기간)를 읽어옵니다. 변환된 데이터는 다른 파이프를 통해 '추세 분석기' 필터로 보내져 데이터를 분석하고 분석 결과를 별도의 데이터베이스에 저장합니다. 마지막으로, 이 필터는 분석 데이터를 최종 파이프를 통해 '그래프 도구' 필터로 보내고, 그래프 도구 필터는 데이터의 그래프 보고서를 생성하여 파이프라인을 완료합니다.

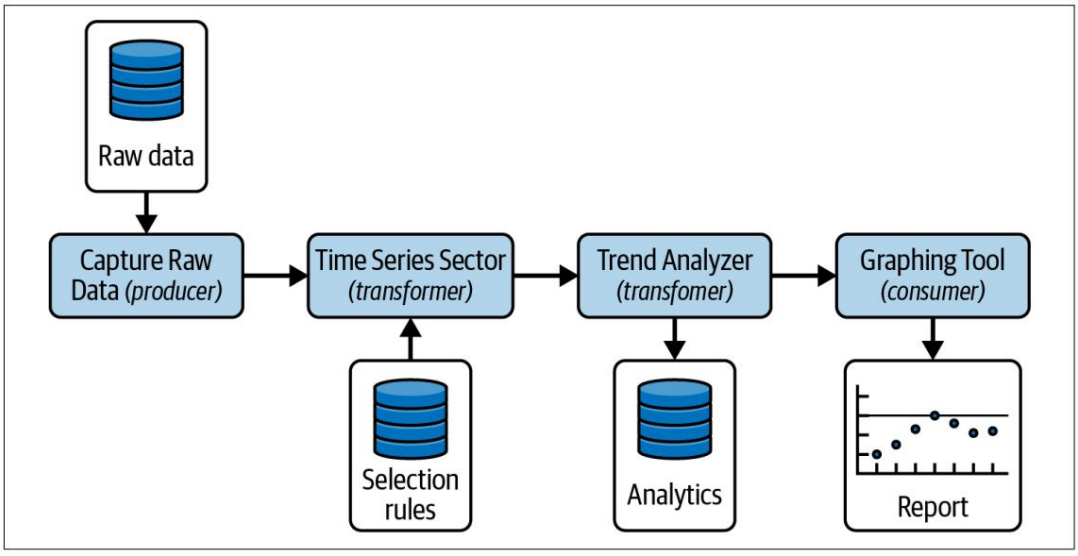

그림 12-2. 파이프라인 아키텍처는 단일 데이터베이스 또는 여러 개의 데이터베이스를 가질 수 있습니다.

클라우드 고려 사항

파이프라인 아키텍처 스타일은 높은 모듈성과 분리된 필터 유형 덕분에 클라우드 기반 배포에 매우 적합합니다. 대부분의 파이프라인 아키텍처는 지나치게 복잡하거나 방대하지 않기 때문에 모든 필터가 동일한 배포 단위에 배포되는 모놀리식 아키텍처로 배포하는 것이 효과적입니다.

하지만 필터는 클라우드 환경에서도 분산 함수로 잘 작동합니다. AWS에서는 파이프라인 아키텍처를 AWS Step Functions로 배포할 수 있으며, 각 함수는 다음과 같은 역할을 합니다 .

필터는 워크플로에서 별도의 Lambda 함수로 배포됩니다. AWS Step Functions는 Standard와 Express 두 가지 워크플로를 제공합니다. Standard 워크플로에서는 각 단계가 정확히 한 번만 실행되고, Express 워크플로에서는 각 단계가 여러 번 실행될 수 있습니다. 파이프라인 아키텍처 스타일은 두 워크플로 모두에서 작동합니다. 다음 코드는 그림 12-2 의 연속 적합도 함수 예제를 AWS Step Function으로 구현한 파이프라인 아키텍처의 일반적인 클라우드 배포 예시로 보여줍니다.

```
{
  "댓글": "확장성 추세를 측정하고 분석합니다.", "시작 위치": "원시 데이터 캡처", "상태": { "원시 데이터
  캡처": { "유형": "작업", "리소스":

  "arn:aws:lambda:region:account_id:function:raw_data_capture", "다음": "시계열 선택기" }, "시계열 선택기": { "유형": "작업", "리소
    스":

    "arn:aws:lambda:region:account_id:function:time_series_selector", "다음": "추세 분석기" }, "추세 분석기": { "유형": "작업", "리소스":

    "arn:aws:lambda:region:account_id:function:trend_analyzer", "다음": "그래프 도구" }, "그래프 도구": { "유형": "작업", "리소스":

      "arn:aws:lambda:region:account_id:function:graphing_tool", "끝": true }

  }
}
```

이 예시는 클라우드 환경에서 파이프라인 아키텍처를 사용하는 유일한 방법이 아닙니다! 필터는 서버리스 함수, 컨테이너화된 함수, 또는 네 가지 필터 구성 요소를 모두 포함하는 단일 서비스 형태의 모놀리식 배포로도 배포할 수 있습니다.

일반적인 위험

파이프라인 아키텍처의 주요 목표는 기능을 단일 목적 필터로 분리하는 것입니다. 각 필터는 데이터에 대해 특정 작업을 수행한 후 추가 처리를 위해 다른 필터로 결과를 전달합니다. 따라서 파이프라인 아키텍처의 가장 일반적인 위험 중 하나는 필터에 과도한 책임을 부여하는 것입니다. 다음 섹션에서 다룰 효과적인 거버넌스는 각 필터 구성 요소의 목적을 명확히 함으로써 팀이 이러한 위험을 완화하는 데 도움을 줍니다.

이러한 아키텍처 스타일에서 흔히 발생하는 또 다른 위험은 필터 간 양방향 통신을 도입하는 것입니다. 파이프라인은 단방향으로만 작동하도록 설계되어 필터 간의 명확한 기능 분리를 통해 필터 간의 협업을 방지합니다. 양방향 통신이 불가피한 경우, 이는 파이프라인 아키텍처가 적합하지 않거나 필터가 지나치게 복잡하여 기능이 제대로 구분되지 않았다는 것을 의미합니다.

오류 처리는 이러한 아키텍처 스타일에서 상당한 위험을 초래할 수 있는 또 다른 복잡한 문제입니다. 파이프라인 내에서 오류가 발생할 경우, 파이프라인이 시작된 후에는 파이프라인을 적절하게 종료하고 복구하는 방법을 파악하기가 어려운 경우가 많습니다. 따라서 아키텍트는 아키텍처를 정의하기 전에 파이프라인 내에서 발생할 수 있는 치명적인 오류 조건을 모두 파악하는 것이 중요합니다.

마지막 위험 영역은 필터 간의 계약 관리입니다. 각 파이프는 다음 필터로 전송하는 데이터(및 해당 데이터 유형)를 나타내는 계약을 가지고 있습니다. 필터 간의 계약을 변경하려면 엄격한 관리 및 테스트를 통해 해당 계약을 수신하는 다른 필터가 오류를 일으키지 않도록 해야 합니다.

통치

응답성, 확장성, 가용성 등과 같은 일반적인 운영 특성에 대한 거버넌스는 각 개별 사용 사례에 따라 다르며 크게 달라질 수 있습니다.
하지만 구조적인 관점에서 볼 때, 설계자들은 각 필터 유형의 역할과 책임을 활용하여 파이프라인 아키텍처를 구성합니다.

네 가지 기본 필터 유형(생산자, 변환기, 테스터, 소비자) 각각은 특정 역할을 수행합니다. 하지만 개발자들이 필터에 너무 많은 책임을 부여하여 파이프라인 아키텍처를 구조화되지 않은 모놀리식으로 만드는 경우가 너무나 흔합니다.

프로듀서 필터가 파이프라인의 시작점인지, 아니면 테스터 필터가 흐름을 계속 진행할지 종료할지 결정하는 조건 검사를 수행하는지 등을 자동화된 방식으로 판별하는 적합성 함수를 작성하는 것은 어렵습니다. 거버넌스 기법은 아키텍트가 개발팀이 각 필터 유형의 역할과 특정 필터 유형의 책임을 준수하도록 안내하는 데 도움이 됩니다.

그러한 기법 중 하나는 태그를 활용하는 것입니다. (자바에서는 태그를 어노테이션으로, C#에서는 사용자 정의 속성으로 구현합니다.) 태그 자체는 기능을 수행하지 않고, 컴포넌트 또는 서비스에 대한 메타데이터를 프로그래밍 방식으로 제공합니다. 필터 컴포넌트의 시작 클래스에 태그를 사용하면 개발자는 생성하거나 수정하는 필터의 유형을 알 수 있으므로, 해당 필터에 과도한 책임을 부여하거나 유형과 관련 없는 기능을 수행하는 것을 방지할 수 있습니다.

이 기법을 설명하기 위해 네 가지 기본 필터 유형에 대한 태그를 정의하는 다음 소스 코드를 살펴보겠습니다. 다음은 Java 태그 어노테이션 정의입니다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE) public
@interface 필터 {
    public FilterType[] value();

    public enum FilterType {
        생산자,
        시험 장치,
        변신 로봇,
        소비자
    }
}
```

다음은 C# 태그의 사용자 지정 속성 정의입니다.

```
[System.AttributeUsage(System.AttributeTargets.Class)] class Filter :
System.Attribute {

    public FilterType[] filterType;

    public enum FilterType {
        생산자,
        시험 장치,
        변신 로봇,
        소비자
    };
}
```

필터는 기본적으로 아키텍처 구성 요소이므로 여러 클래스 파일을 통해 구현될 수 있습니다. 예를 들어, 그림 12-2 의 연속 적합도 함수 예제에서 추세 분석기 필터는 상당히 복잡하며 여러 클래스 파일로 구성될 수 있습니다. 따라서 필터의 진입점을 나타내는 클래스를 식별하기 위해 추가 태그를 정의하여 다른 태그를 연결할 수 있도록 하겠습니다.

다음은 Java 진입점 태그 정의입니다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE) public
@interface FilterEntrypoint {}
```

그리고 C# 진입점 태그 정의는 다음과 같습니다.

```
[System.AttributeUsage(System.AttributeTargets.Class)] class FilterEntrypoint :
System.Attribute {}
```

이제 개발자는 Entrypoint 클래스 에 FilterType 태그를 추가하여 해당 유형과 아키텍처에서의 역할을 지정할 수 있습니다. 예를 들어, 다음 어노테이션은 트렌드 분석기 변환 필터의 유형과 역할을 나타냅니다. (Java 코드)

```
@FilterEntrypoint
@Filter(FilterType.TRANSFORMER) public
class TrendAnalyzerFilter {
    ...
}
```

C# 코드로는 다음과 같습니다.

```
[필터 진입점]
[필터(필터타입.TRANSFORMER)] 클래스
TrendAnalyzerFilter {
    ...
}
```

이 기법이 모든 개발자가 트랜스포머 필터 유형으로 테스트 로직을 수행하는 것을 막지는 못하더라도(테스트 로직은 테스터 필터에서 수행해야 함), 적어도 추가적인 맥락을 제공해 줄 수는 있습니다.

팀 토폴로지 고려 사항

이 책에서 설명하는 다른 아키텍처 스타일과는 달리, 파이프라인 아키텍처 스타일은 일반적으로 팀 구성에 구애받지 않으며 어떤 팀 구성에서도 작동합니다.

스트림 중심 팀 구성은 파이프

라인 아키텍처가 일반적으로 규모가 작고 독립적이며 시스템을 통한 단일 여정 또는 흐름을 나타내기 때문에 효과적입니다. 이러한 팀 구성에서 팀은 일반적으로 시스템의 흐름을 처음부터 끝까지 책임지며, 이는 파이프라인 아키텍처의 형태와 잘 맞아떨어집니다.

파이프라인 아키텍

처는 모듈화 수준이 높고 기술적 관심사별로 분리되어 있어, 다양한 팀 구성에 적합합니다. 전문가와 여러 분야의 팀 구성원은 파이프라인의 나머지 흐름에 영향을 주지 않고 추가 필터를 도입하여 제안을 하거나 실험을 수행할 수 있습니다. 예를 들어, 연속 적합도 함수 예시에서, 지원팀은 시계열 선택기 필터 다음에 변환 필터를 추가하여 다른 추세 분석을 수행할 수 있습니다. 이 새로운 필터는 기존 추세 분석기 필터와 동일한 데이터를 사용하므로 파이프라인의 정상적인 흐름을 방해하지 않습니다.

복잡한 하위 시스템 팀

각 필터가 매우 특정한 작업을 수행하기 때문에 파이프라인 아키텍처는 복잡한 하위 시스템 팀 구성에 적합합니다. 팀 구성원들은 서로 (그리고 다른 필터와도) 독립적으로 복잡한 필터 처리에 집중할 수 있습니다. 이러한 아키텍처에서 수반되는 단방향 핸드오프는 다음과 같은 이점을 제공합니다.

이 스타일은 복잡한 하위 시스템 팀의 구성원이 특정 필터 처리 내에 포함된 복잡성에만 집중할 수 있도록 해줍니다.

플랫폼 팀은 파이
프라인 아키텍처를 개발할 때 공통 도구, 서비스, API 및 작업을 활용하여 높은 수준의 모듈성을 활용할 수 있습니다.

스타일 특징

특성 평가표 (그림 12-3) 에서 별 1개는 특정 아키텍처 특성이 아키텍처에서 제대로 지원되지 않음을 의미하고, 별 5개는 해당 아키텍처 특성이 아키텍처 스타일에서 가장 강력한 기능 중 하나임을 의미합니다. 평가표에 포함된 특성은 4 장 에서 설명하고 정의합니다 .

파이프라인 아키텍처 스타일은 애플리케이션 로직이 필터 유형별로 분리되어 있기 때문에 기술적으로 파티션된 아키텍처입니다. 또한 파이프라인 아키텍처는 일반적으로 모놀리식 배포로 구현되므로 아키텍처 퀀텀은 항상 1입니다.

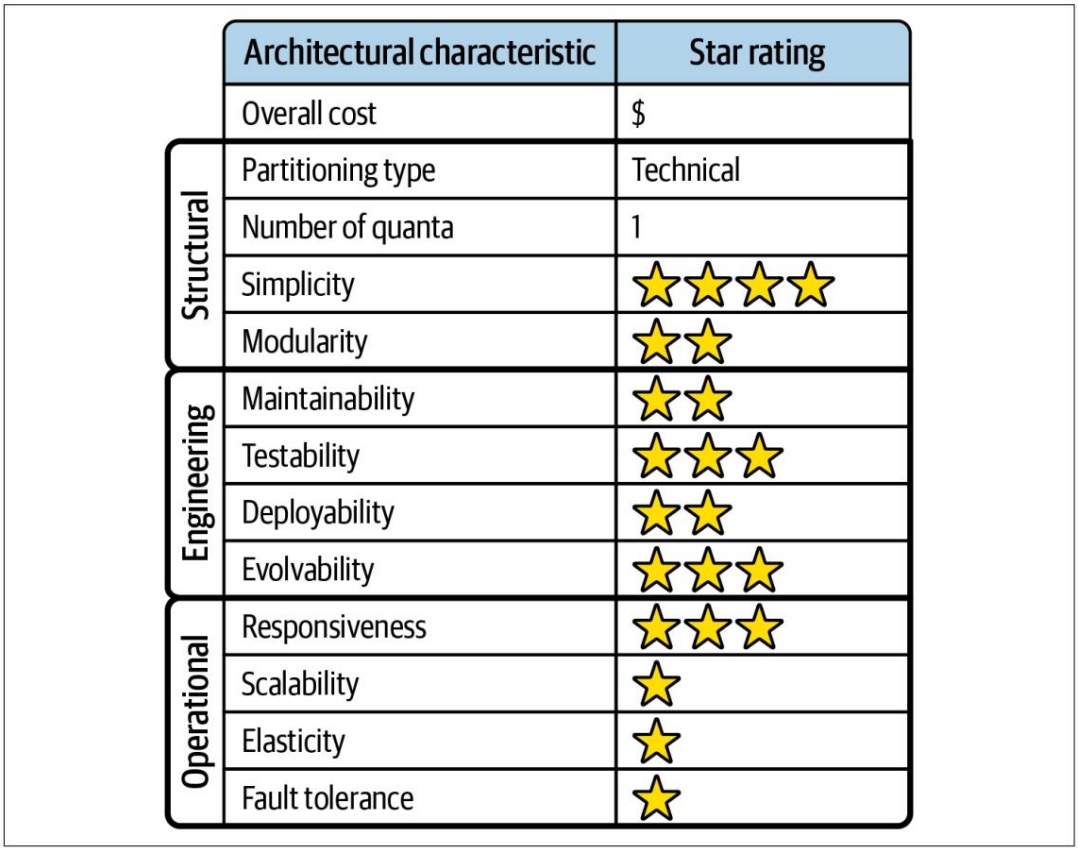

그림 12-3. 파이프라인 아키텍처 특성 등급

파이프라인 아키텍처 스타일의 주요 강점은 전반적인 비용 효율성, 단순성 및 모듈성입니다. 단일 구조인 파이프라인 아키텍처는 분산 아키텍처 스타일과 관련된 복잡성이 없으며, 단순하고 이해하기 쉬우며 구축 및 유지 관리 비용이 상대적으로 저렴합니다. 아키텍처의 모듈성은 다양한 필터 유형과 변환기 간의 관심사를 분리함으로써 달성됩니다. 즉, 어떤 필터든 다른 필터에 영향을 주지 않고 수정하거나 교체할 수 있습니다. 예를 들어, 그림 12-4 에 나타난 Kafka 예제에서 기간 계산기는 다른 필터를 변경하지 않고도 기간 계산 방식을 변경할 수 있습니다.

배포 용이성과 테스트 용이성은 평균 수준이지만, 필터가 달성할 수 있는 모듈화 수준 덕분에 계층형 아키텍처보다 약간 더 높습니다. 그러나 파이프라인 아키텍처는 여전히 일반적으로 단일 구조이므로, 절차적 부담, 위험, 잦은 배포, 그리고 완벽한 테스트의 어려움과 같은 단점이 있습니다.

파이프라인 아키텍처의 탄력성과 확장성은 매우 낮습니다(별 1개). 이는 주로 모놀리식 배포 방식 때문입니다. 비동기 통신을 사용하는 분산 아키텍처로 이 아키텍처 스타일을 구현하면 이러한 특성을 크게 개선할 수 있지만, 전체적인 비용과 단순성이 저하되는 단점이 있습니다.

파이프라인 아키텍처는 일반적으로 단일체 시스템으로 배포되기 때문에 내결함성을 지원하지 않습니다. 파이프라인 아키텍처의 작은 부분에서 메모리 부족 오류가 발생하면 전체 애플리케이션 단위에 영향을 미쳐 충돌이 발생합니다. 또한 대부분의 단일체 애플리케이션에서 흔히 나타나는 높은 평균 복구 시간(MTTR)으로 인해 전반적인 가용성이 저하되며, 시작 시간은 분 단위로 측정됩니다. 탄력성과 확장성과 마찬가지로, 이러한 아키텍처 스타일을 비동기 통신을 사용하는 분산 아키텍처로 구현하면 내결함성을 크게 향상시킬 수 있지만, 비용과 복잡성이 증가하는 단점이 있습니다.

앞서 언급했듯이, 점수가 낮은 운영 특성 대부분은 각 필터를 별도의 배포 단위로 하고 파이프라인을 원격 호출로 하는 비동기 통신 기반의 분산 아키텍처로 구현함으로써 개선할 수 있습니다. 그러나 이렇게 하면 단순성 및 비용과 같은 다른 속성에 부정적인 영향을 미치게 되는데, 이는 소프트웨어 아키텍처 설계에서 흔히 발생하는 상충 관계 중 하나를 보여줍니다.

사용 시점

파이프라인 아키텍처는 명확하고, 순서대로 진행되며, 결정론적인 단방향 처리 단계를 갖는 모든 복잡성의 시스템에 적합합니다. 또한 이러한 방식의 단순성 덕분에 시간적 제약이나 예산 제약이 있는 상황에도 매우 적합합니다.

사용하지 말아야 할 경우

이러한 아키텍처 스타일은 단일체적인 특성 때문에 높은 확장성, 탄력성 및 내결함성이 요구되는 시스템에는 적합하지 않습니다. 하지만 분산 아키텍처 방식을 사용하면 이러한 문제점을 완화할 수 있습니다.

파이프라인 아키텍처 방식은 파이프가 단방향으로만 설계되었기 때문에 필터 간의 양방향 통신이 필요한 시나리오에는 적합하지 않습니다.

파이프라인 전체에 걸쳐 다양한 테스터 필터를 활용하여 비결정적 워크플로우에도 이 아키텍처 스타일을 사용할 수 있지만, 권장하지 않습니다. 이는 비교적 간단한 아키텍처 스타일을 지나치게 복잡하게 만들어 유지보수성, 테스트 용이성, 배포 용이성, 그리고 결과적으로 전반적인 안정성에 부정적인 영향을 미칠 수 있습니다. 비결정적 워크플로우를 처리하는 상황에는 이벤트 기반 아키텍처(15장 참조)가 훨씬 더 적합합니다.

파이프라인 아키텍처 스타일은 특히 단순하고 단방향 처리가 필요한 작업에서 다양한 응용 분야에 나타납니다. 예를 들어, 많은 전자 데이터 교환(EDI) 도구는 파이프와 필터를 사용하여 한 문서 유형에서 다른 문서 유형으로의 변환을 구축하는 데 이 패턴을 사용합니다. 데이터베이스 추출, 변환 및 로드(ETL) 도구는 파이프라인 아키텍처를 활용하여 데이터를 수정하고 한 데이터베이스 또는 데이터 소스에서 다른 데이터베이스 또는 데이터 소스로 데이터를 전달합니다. Apache Camel 과 같은 오케스트레이터 및 미들웨어는 파이프라인 아키텍처를 사용하여 비즈니스 프로세스의 한 단계에서 다른 단계로 정보를 전달합니다.

예시 및 사용 사례

파이프라인 아키텍처를 사용하는 방법을 설명하기 위해 서비스 원격 측정 정보가 스트리밍을 통해 서비스에서 Apache Kafka 로 전송되는 다음 예제를 고려해 보겠습니다 (그림 12-4).

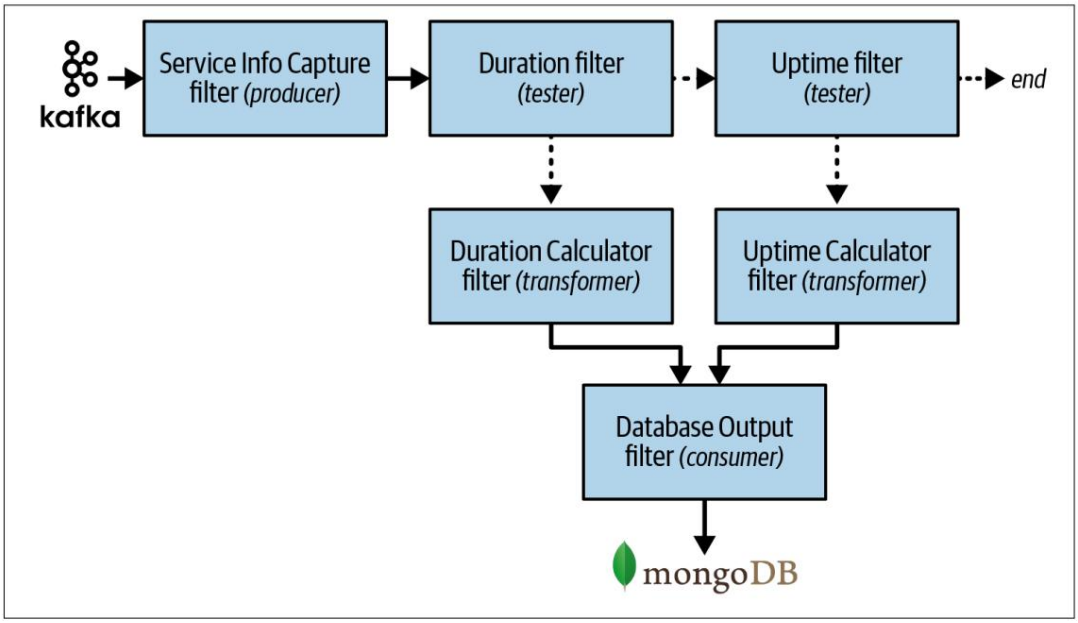

그림 12-4. 파이프라인 아키텍처 예시

그림 12-4 에 나타난 시스템은 파이프라인 아키텍처 스타일을 사용하여 Kafka로 스트리밍되는 다양한 종류의 데이터를 처리합니다. 서비스 정보 캡처 필터(프로듀서 필터)는 Kafka 토픽을 구독하고 서비스 정보를 수신합니다. 그런 다음 캡처된 데이터를 지속 시간 필터라는 테스터 필터로 보내 서비스 요청의 지속 시간(밀리초)과 관련이 있는지 여부를 판단합니다.

필터 간의 역할 분리에 주목하세요. 서비스 정보 캡처 필터는 Kafka 토픽에 연결하여 스트리밍 데이터를 수신하는 방법에만 관여하는 반면, 기간 필터는 데이터의 유효성을 검사하고 다음 파이프라인으로 전송할지 여부를 결정하는 데에만 관여합니다. 데이터가 서비스 요청 기간과 관련된 경우 기간 계산기 변환기 필터로 전달되고, 그렇지 않은 경우 가동 시간 테스터 필터로 전달되어 가동 시간 지표와 관련이 있는지 확인합니다.

그렇지 않으면 파이프라인이 종료됩니다. 해당 데이터는 이 특정 처리 흐름에 필요하지 않기 때문입니다. 만약 데이터가 가동 시간 지표와 관련이 있다면, 가동 시간 테스터 필터는 해당 데이터를 가동 시간 계산기로 전달하고, 가동 시간 계산기는 이를 사용하여 가동 시간 지표를 계산합니다. 그런 다음 가동 시간 계산기는 수정된 데이터를 데이터베이스 출력 소비자에게 전달하고, 데이터베이스 출력 소비자는 해당 데이터를 MongoDB 데이터베이스에 저장합니다.

이 예시는 파이프라인 아키텍처의 확장성을 보여줍니다. 예를 들어, 그림 12-4 에서 가동 시간 필터 다음에 새로운 테스터 필터를 쉽게 추가하여 데이터베이스 연결 대기 시간과 같은 새로운 메트릭에 사용할 데이터를 전송할 수 있습니다.

파이프라인 아키텍처 스타일은 일반적으로 단일체 구조이지만, 기술적으로 분할된 필터를 사용하여 높은 수준의 모듈성을 보여줍니다. 이는 시스템 내에서 데이터를 처리하는 워크플로 기반의 단계별 접근 방식이 필요한 상황에 적합합니다.

보다 복잡한 분산 아키텍처로 넘어가기 전에, 다음 장에서 플러그인 구성 요소를 사용하여 높은 수준의 모듈성을 지원하는 또 다른 단일체 아키텍처인 마이크로커널 아키텍처를 설명하겠습니다.