CNF

2025.12

엔터프라이즈 세션 관리, Redis로는 충분하지 않은 이유는?

“Redis는 정말 엔터프라이즈 세션을 ‘맡겨도 되는’ 기술인가?”
이 백서는 바로 그 지점에서, Redis 기반 세션 아키텍처가 왜 구조적 한계를 가질 수밖에 없는지를 기술적으로 설명하고, 그 대안으로 왜 In-Memory Data Grid(IMDG)가 등장했는지를 논리적으로 풀어냅니다.

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

Contents

제1장. 모든 엔터프라이즈 환경에서 세션 클러스터링이 중요한 이유

서론 (Introduction)

세션 클러스터링 기술을 선택하는 것은 단순히 고가용성을 확보하는 기술적 수단을 넘어, 시스템의 확장성, 운영 복잡성, 그리고 데이터 무결성에 대한 장기적인 방향을 결정하는 근본적인 아키텍처 분기점입니다. 이 기술은 최신 클라우드 네이티브 아키텍처의 전유물이 아니라, 지난 수십 년간 물리 서버, 가상화 환경에 이르기까지 모든 엔터프라이즈 IT 인프라에서 서비스 연속성을 담보해 온 핵심 기술이었습니다. 사용자가 시스템과 상호작용하며 남기는 세션 데이터는 단순한 기술적 부산물이 아닌, 비즈니스 흐름 그 자체를 담고 있는 귀중한 자산입니다.

본 장에서는 세션 데이터가 가진 본질적 가치와 그 관리가 왜 단순한 기술 선택을 넘어 비즈니스 연속성을 좌우하는 핵심적인 아키텍처 결정인지를 논증하고자 합니다. 이 장을 통해 IT 의사결정자는 세션 클러스터링 기술 선택이 고가용성을 넘어, 데이터 무결성을 보장하는 고신뢰 아키텍처와 단순하지만 운영적 한계를 가진 아키텍처 사이의 전략적 약속임을 명확히 이해하게 될 것입니다.

1.1 물리 서버·가상화·클라우드 네이티브를 관통하는 공통 과제

IT 인프라의 형태는 물리 서버 중심의 온프레미스(On-premise)에서 가상화 환경으로, 그리고 오늘날의 동적인 컨테이너 기반 클라우드 네이티브 환경으로 끊임없이 진화해왔습니다. 그러나 기술의 외형이 어떻게 변하든, '서비스의 중단 없는 제공'이라는 목표는 모든 시대와 환경을 관통하는 변치 않는 최우선 과제입니다. 이 섹션의 전략적 중요성은 세션 관리 문제가 특정 기술 트렌드에 국한된 것이 아니라, 모든 비즈니스 환경의 근본적인 고가용성(High Availability) 요구사항과 직결된다는 점을 명확히 하는 데 있습니다.

1.1.1 무중단 서비스와 세션 유지는 인프라 형태와 무관한 본질적 요구사항

사용자 경험의 관점에서 가장 중요한 것은 안정성입니다. 웹 애플리케이션 서버(WAS)의 예기치 않은 장애가 발생했을 때, 사용자는 시스템의 내부 문제를 인지하지 않고 하던 작업을 중단 없이 계속할 수 있어야 합니다. 만약 장애 발생과 동시에 장바구니가 비워지거나, 힘들게 작성하던 신청서 내용이 사라지거나, 강제로 로그아웃된다면 사용자의 신뢰는 크게 하락하며 이는 곧 비즈니스 손실로 직결됩니다.

이처럼 WAS 장애 시에도 사용자는 장애를 인지하지 않고 지속적인 서비스를 이용할 수 있도록 보장하는 것이 바로 고가용성 및 무정지 시스템 구현의 핵심 목표입니다. 이 목표는 시스템이 물리 서버 위에서 동작하든, 가상 머신(VM)이나 컨테이너(Container) 환경에서 운영되든 상관없이 동일하게 적용되는 본질적인 요구사항입니다.

1.1.2 Scale-out 이전에도 존재했던 세션 정합성과 장애 대응의 중요성

세션 관리의 중요성은 클라우드 시대에 갑자기 등장한 개념이 아닙니다. 수평적 확장(Scale-out)이 보편화되기 이전, 소수의 강력한 서버로 운영되던 전통적인 WAS 환경에서도 고가용성을 확보하기 위한 노력은 계속되었습니다.

실제로 사회보장정보원의 교육관리시스템과 같은 다수의 레거시 시스템들은 JEUS6와 같은 상용 WAS가 제공하는 내장 세션 클러스터링 기능을 활용하여 세션 복제(Replication) 구조를 설계했습니다. 이 방식은 클러스터 내의 모든 WAS 인스턴스가 서로 세션 정보를 실시간으로 복제하는 구조(All-to-All)로, 네트워크 부하가 심하고 동기화 이슈가 발생하는 등의 명백한 기술적 한계를 가지고 있었습니다.

여기서 중요한 것은 아키텍처적 트레이드오프(Trade-off)에 대한 이해입니다. 당시의 기업들은 이러한 성능 저하와 네트워크 오버헤드라는 비용을 감수하는 어려운 선택을 했습니다. 그 이유는 단 하나, 장애 발생 시 치명적인 세션 데이터 유실이라는 대안이 비즈니스적으로 도저히 용납할 수 없는 리스크였기 때문입니다. 이는 세션 데이터의 정합성 유지와 장애 대응이 이미 오래전부터 모든 엔터프라이즈 아키텍처의 핵심 고려사항이었음을 증명하며, 현대의 솔루션들이 어떻게 더 나은 트레이드오프를 제공하는지에 대한 논의의 장을 엽니다.

이처럼 모든 환경에서 세션의 연속성이 중요하게 다뤄지는 근본적인 이유는 세션 데이터가 단순한 임시 정보가 아닌, 비즈니스적으로 매우 중요한 '사용자 상태(State)' 그 자체이기 때문입니

다.

1.2 세션은 캐시가 아닌 “사용자 상태(State)”다

세션 데이터를 단순 캐시(Cache) 데이터와 동일하게 취급하는 것은 매우 위험한 발상입니다. 캐시는 성능 향상을 위해 사용되며, 최악의 경우 데이터가 유실되더라도 원본 데이터베이스 조회를 통해 복구할 수 있는 ‘소멸 가능한(disposable)’ 데이터입니다. 그러나 세션 데이터는 사용자의 로그인 상태, 장바구니, 업무 진행 단계 등 복구할 원본이 없는 유일한 정보를 담고 있습니다.

이 섹션의 전략적 목표는 세션 데이터가 가진 고유한 특성—고빈도 갱신, 강한 정합성 요구—을 명확히 정의하고, 이를 통해 왜 세션 관리에 단순 캐시 솔루션이 아닌 데이터베이스 수준의 데이터 무결성 보장 메커니즘이 필요한지를 IT 의사결정자에게 설득하는 데 있습니다.

1.2.1 세션 데이터의 특성: 고빈도 갱신·강한 정합성·유실 불가

데이터베이스 트랜잭션의 신뢰성을 보장하는 ACID 원칙은 세션 데이터가 요구하는 데이터 보증 속성을 이해하는 데 중요한 기준을 제시합니다. ACID는 각 작업이 단일 단위로 완전히 실행되거나 전혀 실행되지 않음을 보장하는 원자성(Atomicity), 데이터가 미리 정의된 규칙에 따라서만 변경됨을 보장하는 일관성(Consistency), 동시 트랜잭션이 서로 간섭하지 않도록 보장하는 고립성(Isolation), 그리고 성공한 트랜잭션은 시스템 장애에도 불구하고 영구적으로 저장됨을 보장하는 지속성(Durability)으로 구성됩니다.

세션 데이터 관리 역시 이러한 원칙에 준하는 엄격한 보증이 필요합니다. 예를 들어, 사용자의 장바구니 상품 추가는 하나의 트랜잭션으로 원자성을 요구합니다. 상품이 추가되고 총액이 업데이트되는 작업은 함께 성공하거나, 실패 시에는 아무 일도 없었던 것처럼 롤백되어야 합니다. 또한, 대규모 할인 행사 중에는 여러 사용자의 동시 주문이 서로의 재고 계산에 영향을 주지 않도록 고립성이 보장되어야 합니다.

- 강한 일관성 (Strong Consistency): 금융 거래나 좌석 예약과 같이 여러 사용자의 요청이 동시에 발생하는 환경에서 데이터 정합성은 비즈니스의 성패를 좌우합니다. 사용자가 조회하는 데이터는 언제나 가장 최신의 정확한 상태여야 합니다. 이러한 요구사항은 세션 관리 솔루션의 근본 아키텍처에 의해 결정됩니다.

  - In-Memory Data Grid (IMDG): P2P 데이터 그리드(Peer-to-Peer Data Grid) 아키텍처를 기반으로 합니다. 모든 노드가 동등한 역할을 수행하며, 데이터 변경 시 동기식 백업(synchronous backup)을 통해 여러 노드에 데이터 복제를 완료한 후에야 최종 응답을 반환합니다. 이 구조는 여러 노드에 걸쳐 데이터의 강한 일관성을 보장하는 근간이 됩니다.
  - Redis Cluster: Primary/Replica 샤딩(Primary/Replica Sharding) 모델을 사용합니다. 쓰기 작업은 Primary 노드에서 먼저 처리된 후, 비동기 복제(asynchronous replication)를 통해 Replica 노드로 전파됩니다. 이 방식은 Primary 노드 장애 시, 아직 Replica로 복제되지 않은 데이터가 유실될 수 있는 위험을 내포하며, 최종 일관성(Eventual Consistency) 모델을 따릅니다. 이 짧은 시간 동안의 데이터 불일치는 심각한 비즈니스 오류로 이어질 수 있습니다.

- 유실 불가 (Durability & Fault-Tolerance): WAS 인스턴스 장애, 네트워크 단절, 또는 예기치 않은 시스템 중단 상황에서도 세션 데이터는 절대 유실되어서는 안 됩니다. 순수 인메모리 저장소는 전원 공급이 중단되면 데이터가 소멸될 위험이 있으며, Redis의 경우에도 "서버가 충돌하거나 시스템 전원이 꺼지면 메모리의 모든 데이터가 손실될 수 있으며, 영속성 옵션을 활성화하더라도 스냅샷 사이에 데이터 유실 가능성이 존재한다"는 점을 명확히 인지해야 합니다. 이러한 위험을 극복하기 위해 엔터프라이즈급 IMDG는 아키텍처 수준의 내결함성을 제공합니다. P2P 아키텍처 덕분에 특정 노드에 장애가 발생하면, 시스템이 이를 자동으로 감지하여 데이터의 복제본을 즉시 새로운 Primary로 승격시키고, 부족해진 복제본을 다른 노드에 생성하는 자동 데이터 재조정(Rebalancing) 및 자가 복구(Self-Healing) 기능을 수행합니다. 이는 관리자가 `SETSLOT ... MIGRATING`과 같은 명령어로 수동 개입해야 하는 Redis의 리샤딩(resharding) 과정과 대조되며, 클라우드 네이티브 환경의 탄력적 확장에 필수적인 운영 자율성을 보장합니다. 나아가, 데이터센터 수준의 재해 복구를 위해 여러 지리적 위치에 걸쳐 데이터를 복제하는 WAN Replication 기능까지 제공합니다.

1.2.2 로그인·거래·업무 흐름에서 세션이 갖는 비즈니스적 의미

세션 데이터 유실은 단순한 기술적 오류나 사용자의 불편함을 넘어, 즉각적인 매출 하락과 생산성 감소로 직결되는 심각한 비즈니스 문제입니다.

- 사용자 식별: 세션은 익명의 HTTP(S) 접속을 특정 고객과 연결하는 핵심 정보입니다. 생성된 세션 ID는 시스템 내의 '사용자 식별인자(User Identifier)'와 매핑되어, "누가" 시스템을 사용하고 있는지를 식별하는 유일한 수단이 됩니다. 이 연결이 끊어지면 사용자는 신원을 잃고 모든 활동을 처음부터 다시 시작해야 합니다.
- 업무 흐름 연속성: 여러 페이지에 걸쳐 데이터를 입력하고 최종적으로 완료하는 다단계 업무 흐름에서 세션의 역할은 절대적입니다. 항공권 예매를 위한 다단계 결제 과정을 생각해 보십시오. 좌석 선택부터 탑승객 정보 입력, 부가 서비스 선택, 그리고 최종 결제에 이르기까지, 세션은 이 모든 미완의 거래 상태를 고스란히 담고 있습니다. 마지막 결제 단계에서 세션 장애가 발생한다면, 이는 단순히 사용자를 불편하게 만드는 것을 넘어 기업의 수익 창출을 직접적으로 막고 브랜드에 대한 신뢰를 심각하게 훼손하는 결과를 초래합니다.

결론적으로, 세션 데이터는 본질적으로 '파기 가능한 캐시'가 아닌 '보호해야 할 사용자 상태'입니다. 따라서 세션 클러스터링 기술을 선택하는 것은 단순히 빠른 응답 속도를 위한 기술 구현의 문제가 아닙니다. 이는 아키텍처의 근간을 이루는 전략적 결정입니다. 한쪽에는 비동기 복제의 성능적 이점을 위해 데이터 유실 위험을 감수하는 아키텍처가 있고, 다른 한쪽에는 동기식 복제를 통해 데이터 무결성을 절대적으로 보장하고 자동화된 확장과 복구를 지원하는 고신뢰 아키텍처가 있습니다. 이 선택은 예기치 않은 장애 상황에서도 비즈니스의 흐름을 중단 없이 이어가고 고객의 신뢰를 지키는, 비즈니스 연속성을 담보하는 핵심적인 아키텍처 결정입니다.

제2장. 전통적 WAS 세션 클러스터링의 진화와 구조적 한계

In-Memory Data Grid와 같은 현대적 세션 관리 솔루션의 필요성을 전략적으로 이해하기 위해서는, 그 이전에 존재했던 초기 웹 애플리케이션 서버(Web Application Server, WAS) 내장 세션 클러스터링 방식의 작동 원리와 본질적인 설계 결함을 먼저 파악하는 것이 필수적입니다. 이 방식들은 고가용성(High-Availability) 문제에 대한 최초의 해결 시도였으나, 세션 데이터의 생명주기를 WAS 인스턴스 자체에 깊숙이 결속시키는 구조적 한계를 지니고 있었습니다. 이러한 근본적인 한계는 이후 심각한 운영 리스크와 기술 부채로 이어졌습니다.

2.1 WAS 내부 세션 클러스터링의 작동 방식

2.1.1 In-Process Replication과 메모리 기반 세션 동기화 구조

전통적 WAS가 제공하는 내장 세션 클러스터링의 핵심은 In-Process Replication 방식에 기반합니다. 이는 별도의 세션 관리 서버나 인프라 없이, 클러스터에 속한 다른 WAS 인스턴스의 JVM 힙(Heap) 메모리에 세션 데이터를 복제하여 저장하는 모델입니다. 일반적으로 한 WAS 인스턴스가 세션 데이터의 주 서버(Primary) 역할을 하면, 지정된 다른 인스턴스가 보조 서버(Backup)가 되어 데이터를 복제하는 Primary/Backup 또는 Buddy Replication 구조를 따릅니다.

이 아키텍처의 가장 치명적인 설계 결함은 세션 데이터의 생명주기가 WAS 인스턴스의 생명주기에 완벽하게 종속된다는 점입니다. 세션 데이터가 각 서버의 JVM 힙 내에 직접 저장되므로, 다음과 같은 심각한 운영 문제를 야기했습니다.

- JVM 힙 메모리 사용량 급증: 활성 세션의 수가 증가할수록 각 WAS 인스턴스의 힙 메모리 사용량이 크게 증가합니다. 이는 애플리케이션 로직이 사용할 수 있는 메모리 공간을 잠식하는 결과를 낳습니다.
- 긴 Garbage Collection (GC) 지연: 세션 데이터로 인해 힙 메모리가 비대해지면, 불필요한 메모리를 정리하는 GC 작업에 소요되는 시간이 길어집니다. 이는 애플리케이션의 응답을 일시적으로 멈추게 하는 'Stop-the-World' 현상을 유발하여 서비스 전반의 성능 저하를 초래합니다.
- OutOfMemoryError (OOM) 장애 위험: 예기치 못한 트래픽 폭증으로 세션 데이터가 힙 메모리 용량을 초과하면, WAS 인스턴스는 OOM 오류를 발생시키며 비정상적으로 종료됩니다. 내장 클러스터링 환경에서는 하나의 인스턴스 장애가 다른 인스턴스로의 세션 복제 부하로 이어져 연쇄적인 장애를 일으킬 수 있습니다.

2.1.2 Sticky Session과 멀티캐스트 기반 복제의 기술적 제약

초기 세션 클러스터링은 기술적 한계를 극복하기 위해 여러 방안을 도입했으나, 이는 또 다른 문제를 낳았습니다.

첫째, 복제 부하를 줄이기 위한 방편으로 Sticky Session (또는 세션 고정) 방식이 널리 사용되었습니다. 이는 로드밸런서가 특정 사용자의 모든 요청을 항상 동일한 WAS 인스턴스로만 보내도록 고정하는 방식입니다. 하지만 이 구조는 부하 분산(Load Balancing)이라는 클러스터링의 근본적인 목적을 훼손하며, 해당 사용자가 접속 중인 서버에 장애가 발생할 경우 세션이 유실되는 단일 장애 지점(Single Point of Failure) 문제를 야기합니다.

둘째, 데이터 동기화를 위한 네트워크 모델의 비효율성입니다. 다수의 WAS 솔루션은 클러스터 내 모든 인스턴스가 서로에게 세션 정보를 전파하는 All-to-All 복제 방식을 채택했습니다. 이는 주로 멀티캐스트(multicast) 통신에 기반하는데, 클러스터에 WAS 인스턴스가 몇 대만 추가되어도 네트워크 트래픽이 기하급수적으로 증가하여 심각한 병목 현상을 유발했습니다. 결과적으로 진정한 의미의 수평적 확장(Scale-out)이 불가능해졌습니다.

이처럼 정적인(static) 구성과 예측 가능한 트래픽을 전제로 설계된 구시대적 세션 복제 모델은, 유동적이고 예측 불가능한(dynamic and unpredictable) 워크로드를 처리해야 하는 현대적인 클라우드 환경의 자동 확장(Scale-in/out) 요구사항과 근본적인 아키텍처 충돌을 일으켰습니다.

이러한 근본적인 기술적 제약은 단순히 성능 문제를 넘어, 기업의 운영 전반에 걸친 심각한 비즈니스 리스크를 초래했습니다.

2.2 고가 WAS 종속 구조가 만들어낸 운영 리스크

앞서 논의된 기술적 한계들은 단순히 독립된 엔지니어링의 문제가 아니었습니다. 이는 조직을 특정 고가 상용 WAS 벤더에 종속시키는 구조적 의존성을 강요했습니다. 이로 인해 기업은 한번 특정 솔루션을 선택하면 벗어나기 힘든 기술 종속성의 굴레에 갇히게 되었고, 이는 장기적인 관점에서 기업의 총소유비용(Total Cost of Ownership, TCO)을 급격히 증가시키고 기술 현대화를 저해하는 심각한 비즈니스 리스크로 이어졌습니다.

2.2.1 특정 WAS 벤더에 종속되는 세션 기술 스택

전통적 상용 WAS 시장을 주도했던 WebLogic, WebSphere, JEUS와 같은 제품들이 제공하는 세션 클러스터링 기능은 각 벤더의 고유 기술로 구현된 독점적(proprietary) 기능이었습니다. 이

는 서로 다른 벤더의 WAS 제품 간에는 세션 데이터가 호환되지 않음을 의미합니다.

이러한 구조는 조직의 전체 세션 관리 전략이 특정 WAS 벤더의 기술 로드맵과 라이선스 정책에 의해 좌우되는 Vendor Lock-in (벤더 종속성) 문제를 야기했습니다. 한 번 특정 WAS를 도입하면, 기업은 더 효율적이거나 비용 효과적인 다른 세션 관리 기술을 채택할 수 없게 되었습니다. 결국 애플리케이션, 미들웨어, 세션 관리가 하나의 거대한 모놀리식(Monolithic) 기술 스택으로 단단하게 결합되어, 변화와 혁신이 극도로 어려운 경직된 시스템 구조를 낳았습니다.

2.2.2 WAS 업그레이드·증설·전환 시 함께 따라오는 비용과 리스크

벤더 종속성은 기업의 IT 운영에 실질적인 비용과 리스크를 지속적으로 부과했습니다.

- 업그레이드 리스크 (Upgrade Risk) 세션 클러스터링은 WAS의 내장 기능이므로, WAS 버전 업그레이드 시 고가용성 보장을 위한 클러스터링 메커니즘 전체를 처음부터 다시 검증해야 했습니다. 이는 단순한 패치 적용을 넘어, 대규모 마이그레이션에 준하는 복잡하고 위험 부담이 큰 프로젝트로 변질되기 일쑤였습니다.
- 확장성 한계 (Scalability Limitations) 클러스터에 서버를 추가(증설)하는 것은 단순히 인스턴스를 늘리는 작업으로 끝나지 않았습니다. 앞서 언급한 All-to-All 복제 방식의 네트워크 오버헤드 때문에, 노드를 추가할수록 오히려 전체 클러스터의 성능이 저하되는 현상이 발생했습니다. 이는 예측 가능한 선형적 확장(Linear Scalability)을 불가능하게 만드는 근본적인 한계였습니다.
- 전환 비용 (Migration Costs) 다른 WAS 벤더의 제품으로 전환하는 것은 엄청난 비용과 복잡성을 수반했습니다. 애플리케이션 코드에 벤더 종속적인 세션 관리 API가 깊숙이 결합된 경우가 많아, 단순한 플랫폼 이전이 아닌 애플리케이션 재작성에 가까운 노력이 필요했습니다. 이는 사실상 다른 벤더로의 전환을 불가능하게 만드는 강력한 장벽으로 작용했습니다.

결론적으로, 전통적인 WAS 내장 세션 클러스터링은 기술적 설계의 한계와 특정 벤더에 대한 깊은 종속성이라는 두 가지 구조적 결함을 동시에 안고 있었습니다. In-Process Replication 방식은 JVM 불안정성과 OOM 장애를 야기했고, 비효율적인 All-to-All 복제 모델은 클라우드 환경의 탄력적 확장을 원천적으로 불가능하게 만들었습니다. 그리고 이 모든 기술적 제약은 특정 벤더의 독점적 기능 안에 갇혀 벗어날 수 없는 Vendor Lock-in으로 귀결되었습니다. 이러한 총체적인 문제들은 세션 상태를 WAS의 생명주기에서 분리하여, 확장 가능하고 벤더에 중립적인 외부

전용 플랫폼에서 관리하는 새로운 아키텍처 패러다임의 등장을 필연적으로 만들었습니다. 이는 다음 장에서 다룰 In-Memory Data Grid의 핵심 사상으로 이어집니다.

제3장. 왜 세션 클러스터링은 WAS에서 분리되어야 하는가

현대적인 IT 인프라는 Kubernetes를 중심으로 하는 클라우드 네이티브 환경으로 빠르게 전환하고 있습니다. 이러한 환경의 핵심 가치는 자동 확장(Auto-Scaling), 무중단 배포(Rolling Upgrade), 자동 복구(Self-Healing)와 같은 동적인 특성에 기반한 탄력성과 민첩성입니다. 그러나 이러한 패러다임은 과거의 안정적인 서버 환경을 전제로 설계된 전통적인 웹 애플리케이션 서버(WAS)의 세션 관리 방식과 근본적으로 충돌합니다.

과거에는 사용자의 세션 데이터를 WAS 인스턴스의 메모리에 직접 저장하고, 인스턴스 간에 데이터를 복제하여 고가용성을 유지하려 했습니다. 하지만 POD가 수시로 생성되고 소멸하는 동적인 클라우드 환경에서 이 방식은 더 이상 유효하지 않으며, 오히려 시스템 전체의 안정성과 확장성을 저해하는 기술 부채로 작용합니다.

따라서 세션 관리 계층을 애플리케이션 실행 환경(Runtime)으로부터 분리하는 '세션 외부화(Session Externalization)'는 더 이상 선택적 아키텍처 패턴이 아닙니다. 이는 Kubernetes, 자동 확장, CI/CD에 대한 막대한 투자가 온전한 비즈니스 가치로 이어지도록 보장하는 핵심적인 아키텍처 전제조건입니다. 예측 불가능한 장애 상황에서도 비즈니스 연속성을 담보하고 클라우드 네이티브의 잠재력을 최대한 활용하기 위한 이 필수적인 전략적 결정을 본 장에서 심층적으로 분석하고, 세션 계층 분리가 가져오는 명확한 아키텍처상의 이점을 구체적으로 증명하고자 합니다.

3.1 세션과 실행(Runtime)의 결합이 초래하는 문제

전통적인 아키텍처에서 사용자 세션은 애플리케이션이 실행되는 WAS 인스턴스의 메모리 내에 저장됩니다. 이러한 설계는 별도의 인프라 없이 세션 관리가 가능하다는 초기 구현의 편의성을 제공했지만, 세션 데이터의 생명주기를 WAS 인스턴스의 생명주기와 강하게 결합시키는 구조적 한

계를 내포합니다. 이 강한 결합은 시스템 전체의 유연성을 저해하고, 예측 불가능한 장애의 근본 원인이 됩니다. 이하에서는 이러한 결합 구조가 어떻게 단일 WAS 인스턴스의 문제를 서비스 전체의 장애로 확산시키고, 클라우드 네이티브 환경의 핵심적인 운영 모델과 충돌하는지 구체적으로 분석합니다.

3.1.1 WAS 장애가 곧 세션 장애로 확산되는 구조

WAS 인스턴스와 세션 데이터의 생명주기가 직접적으로 결합된 구조의 가장 큰 취약점은 WAS 메모리의 본질적인 '휘발성(Volatility)'에서 비롯됩니다. 이는 단순한 개념이 아닌 물리적 현실의 문제입니다. 세션 데이터가 저장되는 RAM(Random Access Memory)은 나노초 단위의 압도적인 속도를 제공하지만, 전원 공급이 중단되면 모든 데이터가 영구적으로 소멸하는 태생적 한계를 가집니다. 반면, 디스크와 같은 영구 저장소는 데이터가 물리적으로 보존됩니다. 이 근본적인 차이 때문에, 예측하지 못한 서버 장애나 정상적인 재기동만으로도 WAS 인스턴스에 저장된 모든 세션 데이터는 즉시 소멸됩니다.

이는 비즈니스에 직접적인 손실과 고객 신뢰도 하락으로 이어집니다. 사용자의 장바구니 정보나 로그인 상태가 순식간에 사라지는 현상은 바로 이 휘발성 때문에 발생합니다. 결국 사용자는 강제 로그아웃을 경험하거나 진행 중이던 작업을 모두 잃게 되어 서비스 만족도가 급격히 하락하고, 이는 장애 복구를 위한 평균 시간(MTTR) 증가와 비효율적 운영으로 인한 총소유비용(TCO) 상승으로 직결됩니다.

이처럼 단일 WAS 인스턴스의 장애가 곧바로 사용자 세션 데이터의 유실로 이어지는 구조는, 해당 인스턴스를 시스템 전체의 단일 장애 지점(Single Point of Failure, SPOF)으로 만듭니다. 이는 개별 서버의 문제가 전체 서비스의 안정성을 위협하는 심각한 아키텍처적 취약점입니다.

3.1.2 배포·확장·재기동 시 세션 안정성이 흔들리는 근본 원인

전통적인 WAS는 내장된 세션 복제(Replication) 기능을 통해 고가용성을 구현하려 시도합니다. 그러나 모든 인스턴스가 서로의 세션 정보를 동기화하는 'All-to-All' 복제 방식은 구조적인 모순을 내포합니다. 고가용성을 위해 설계된 이 메커니즘이 역설적으로 시스템 규모가 커질수록 가장 큰 병목점이자 장애 포인트가 되기 때문입니다. 클러스터 내 인스턴스가 증가할수록 세션 동기화를 위한 네트워크 트래픽은 기하급수적으로 증가하며, 각 WAS는 세션 객체의 직렬화(Serialization)

와 역직렬화(Deserialization)에 상당한 CPU 자원을 소모하여 애플리케이션 본연의 성능을 심각하게 저해합니다.

이러한 구조적 한계는 Kubernetes와 같은 최신 컨테이너 환경의 동적인 운영 방식과 정면으로 충돌합니다.

- 자동 확장(Auto-Scaling) 시의 문제: HPA(Horizontal Pod Autoscaler)에 의해 부하가 동적으로 조절되는 상황에서 심각한 문제가 발생합니다.
  - Scale-In: 부하 감소로 특정 POD가 종료되면, 해당 POD의 메모리에 저장된 모든 사용자 세션은 영구적으로 유실됩니다.
  - Scale-Out: 부하 증가로 새로운 POD가 생성될 때, 이 POD는 기존 세션 컨텍스트를 전혀 알지 못합니다. 세션 선호도(Session Affinity) 없이 동작하는 로드밸런서가 사용자의 다음 요청을 이 새로운 POD로 전달하면, 해당 POD에는 사용자의 세션 정보가 존재하지 않습니다. 결과적으로 사용자는 강제 재로그인을 하거나 장바구니 같은 상태 정보를 잃게 되어, 결국 사용자 여정(User Journey)이 실패로 끝나게 됩니다.
- 무중단 배포(Rolling Upgrade) 시의 문제: 새로운 버전의 애플리케이션을 배포할 때 구버전 POD가 순차적으로 종료되고 신규 버전 POD가 생성됩니다. 이 과정에서 구버전 POD가 종료되는 순간 해당 POD가 관리하던 세션 데이터는 함께 소멸하여 세션 일관성이 깨지게 됩니다.

결론적으로, WAS 내장 세션 관리 방식은 인스턴스의 수가 동적으로 변화하는 클라우드 네이티브 환경의 핵심 운영 모델을 지원할 수 없습니다. 이는 세션 안정성을 근본적으로 위협하며, 탄력적인 확장과 안정적인 배포라는 클라우드 환경의 핵심 가치를 무력화시킵니다.

3.2 세션 계층 분리가 주는 구조적 이점

앞서 설명한 아키텍처적 취약점들은 기존 모델을 최적화하는 방식으로는 해결할 수 없으며, 근본적인 패러다임의 전환—즉, 상태(State)와 실행(Runtime)의 분리—를 요구합니다. 세션 계층을 애플리케이션 런타임에서 분리하는 것은 단순히 기존의 문제를 해결하는 소극적인 조치를 넘어, 시스템 아키텍처를 현대화하고 미래의 기술적 요구사항에 유연하게 대응하기 위한 전략적 진화입니다.

세션을 WAS의 생명주기로부터 독립시켜 별도의 전용 데이터 그리드에서 관리함으로써, 시스템은 비로소 진정한 의미의 고가용성, 탄력적 확장성, 그리고 운영 유연성을 확보할 수 있습니다. 이는 각 구성 요소의 책임을 명확히 분리하고 결합도를 낮추는 현대적인 아키텍처 설계 원칙의 핵심적인 실현입니다.

3.2.1 WAS·프레임워크·인프라와 독립된 세션 수명주기 관리

세션 외부화가 제공하는 가장 근본적인 이점은 '독립된 세션 생명주기'를 확보하는 것입니다. 세션 데이터가 외부의 전용 데이터 그리드(Data Grid)—메모리 내 데이터를 여러 노드에 걸쳐 분산 및 복제하는 분산 시스템—에 저장되면, 개별 WAS 인스턴스의 생명주기와 세션 데이터의 생명주기가 완벽하게 분리됩니다.

이 아키텍처에서는 특정 WAS 인스턴스에 장애가 발생하거나, 애플리케이션 재배포 또는 서버 재기동이 수행되더라도 세션 데이터는 외부 데이터 그리드에 안전하게 보존됩니다. 사용자의 요청은 다른 정상적인 WAS 인스턴스로 즉시 라우팅되어 중단 없이 서비스를 이어갈 수 있습니다. 이는 개별 서버의 장애가 전체 서비스 중단으로 이어지는 것을 방지하고 비즈니스 민첩성을 보장하는 강력한 장애 극복(Failover) 메커니즘을 제공합니다.

또한, 이 구조는 WAS를 상태 정보(State)를 저장하지 않는 '무상태(Stateless)' 구조로 전환시킵니다. WAS는 더 이상 세션 데이터를 복제하고 동기화하는 무거운 부담을 지지 않아도 되므로, 예측 가능한 수평 확장(Scale-out)이 가능해집니다. 이는 애플리케이션의 JVM 힙(Heap)에서 세션 데이터 힙을 분리하는 결정적인 아키텍처 개선입니다. 과도한 세션 사용으로 인한 긴 가비지 컬렉션(GC) 시간과 메모리 부족(OOM) 장애의 근본 원인을 해결함으로써, 애플리케이션 성능의 예측 가능성을 극대화합니다.

3.2.2 이기종 WAS·MSA·Kubernetes 환경을 동시에 지원하는 기반

분리된 세션 계층은 IT 환경의 유연성과 상호운용성을 극대화하는 핵심 기반이 됩니다. 외부의 중앙 집중화된 세션 저장소는 시스템 전체의 단일 진실 공급원(Single Source of Truth) 역할을 수행합니다. 이는 단순히 기술적 우위를 넘어, 아키텍처를 단순화하고 데이터 중복성을 제거하며, 서비스 간의 복잡한 포인트-투-포인트(Point-to-Point) 통합을 제거하여 장기적인 유지보수 비용과 TCO를 절감하는 전략적 가치를 가집니다.

- 이기종 환경 지원: 이 구조를 통해 WebLogic, JBoss, Tomcat과 같이 서로 다른 종류의 WAS가 혼재된 이기종 환경에서도 모든 인스턴스가 동일한 세션 저장소를 참조하여 원활하게 세션을 공유할 수 있습니다. 이는 다양한 시스템이 복잡하게 얽혀있는 엔터프라이즈 환경에 최적화된 유연성을 제공합니다.
- 마이크로서비스 아키텍처(MSA) 지원: 인증(Auth), 결제(Payment), 프로필(Profile) 등 여러 마이크로서비스가 독립적으로 운영되는 환경에서 일관된 사용자 경험을 제공하기 위한 단일 로그인(Single Sign-On, SLO) 구현은 필수적입니다. 외부 세션 저장소는 모든 서비스가 공유하는 인증 컨텍스트를 제공함으로써, SLO를 구현하기 위한 가장 확실하고 검증된 기술적 전제 조건이 됩니다.
- Kubernetes 환경 최적화: POD가 동적으로 생성되고 소멸하는 Kubernetes 환경에서 세션의 영속성과 일관성을 보장하는 핵심은 상태를 POD 외부에서 관리하는 것입니다. 외부 세션 그리드는 POD의 생명주기와 무관하게 세션 데이터를 안정적으로 유지함으로써, 클라우드 네이티브 애플리케이션이 완전한 탄력성과 복원력을 갖추도록 하는 기반 기술로 작용합니다. 이는 클라우드 마이그레이션을 가속화하고 리스크를 줄이는 핵심 전략입니다.

제4장. Redis 세션 클러스터링이 선택된 이유와 그 한계

Spring Framework 기반의 애플리케이션에서 세션 클러스터링을 논의할 때, Redis가 거의 '기본값'처럼 선택되는 현상은 이제 하나의 관행에 가깝습니다. 특히 Spring Boot와 Spring Session을 사용하는 개발 환경에서는 Redis를 도입하지 않는 것이 오히려 예외처럼 보일 정도입니다.

이러한 선택은 우연이 아닙니다. Redis는 개발 초기 단계에서 압도적인 생산성과 즉각적인 성과를 제공하며, 복잡한 세션 클러스터링 문제를 매우 단순한 설정 문제로 환원시켜 줍니다. 이 시점에서 Redis는 "가볍고 빠르며 검증된 표준 솔루션"처럼 인식됩니다.

그러나 문제는 이 선택이 개발 환경의 논리에서 출발했다는 점입니다. 개발 단계에서의 단순함과 빠른 성공 경험은, 프로덕션 운영이라는 전혀 다른 차원의 현실과 마주하는 순간 전혀 다른 성격의 문제로 변모합니다. 많은 엔터프라이즈 시스템에서 Redis 기반 세션 클러스터링이 시간이 지날수록 운영 복잡성, 장애 전파 범위 확대, 그리고 예측 불가능한 리스크로 이어지는 이유가 바로 여기에 있습니다.

본 장에서는 Redis가 왜 개발자들에게 그토록 매력적인 선택이 되었는지를 먼저 짚어본 뒤,

그 선택이 장기 운영 관점에서 어떤 구조적 한계를 드러내는지 차분히 해부합니다. 이를 통해 개발 편의성과 운영 안정성 사이에 존재하는 본질적인 트레이드오프를 이해하고, 보다 성숙한 세션 아키텍처를 고민하기 위한 기술적 시야를 확보하고자 합니다.

4.2.1. Redis는 '데이터베이스'가 아니라 '캐시'로 태어났다

Redis를 세션 저장소로 쓸 때 겪는 모든 문제의 근원은 하나입니다. Redis의 DNA가 영구 저장을 위한 '데이터베이스'가 아니라, 휘발성을 전제로 한 '캐시'라는 점입니다. 이 차이는 단순한 용어의 차이가 아니라, 시스템이 위기 상황에서 어떤 가치를 포기하도록 설계되었는지를 결정합니다.

1. 설계 우선순위의 불일치: 속도(Speed) vs. 신뢰성(Reliability)
    - 캐시(Cache)의 철학: 캐시는 원본 데이터(Database)의 사본을 잠시 빌려 두는 공간입니다. 따라서 "데이터가 날아가도 원본 DB에서 다시 조회하면 된다"는 전제가 깔려 있습니다. Redis는 이를 위해 데이터의 안전한 저장보다 메모리 접근 속도를 극대화하는 방향으로 진화했습니다.
    - 세션 저장소(Session Store)의 요구사항: 세션은 사용자의 로그인 상태, 장바구니 품목, 결제 진행 단계 등 그 자체가 '유일한 진실의 원천(Source of Truth)'입니다. 세션 데이터 유실은 원본 DB에서 복구할 수 있는 성질의 것이 아닙니다. 이는 곧바로 사용자 강제 로그아웃, 결제 오류와 같은 치명적인 서비스 장애로 이어집니다. 즉, 세션 저장소는 속도보다 데이터가 절대 사라지지 않는 안정성이 제1원칙이어야 합니다.
2. 데이터 일관성 모델의 한계: 최종 일관성 vs. 강력한 일관성
    - 분산 시스템 이론인 CAP 이론 관점에서 볼 때, Redis는 AP(Availability + Partition tolerance) 시스템에 가깝습니다. 즉, 네트워크 단절 시 데이터 정합성을 희생하더라도 응답을 주는 것을 택합니다.
    - Redis의 복제(Replication)는 기본적으로 비동기(Asynchronous) 방식입니다. Primary 노드가 데이터를 받으면 Replica 노드에 전달하기도 전에 클라이언트에게 "성공(OK)" 응답을 먼저 보냅니다.
    - 위험 시나리오: 만약 Primary가 데이터를 받고 클라이언트에게 OK를 보낸 직후, Replica에 복제하기 전에 셧다운된다면? 시스템은 성공했다고 응답했지만, 실제 데

이터는 영원히 사라지는 상황이 발생합니다. 금융권이나 대형 커머스처럼 데이터 정합성(Strong Consistency)이 생명인 곳에서는 결코 용납될 수 없는 아키텍처입니다.

4.2.2. 세션과 캐시를 동일시할 때 발생하는 치명적 설계 오류

Redis의 '캐시 중심 설계'를 '세션 저장소'로 강행할 때, 운영 환경에서는 다음과 같은 구체적이고 심각한 문제들이 터져 나옵니다.

1) 장애 시 데이터 유실 위험 (ACID 중 Durability 위반)

데이터베이스 트랜잭션의 4원칙(ACID) 중 내구성(Durability)은 "성공한 트랜잭션은 시스템이 망가져도 보존되어야 한다"는 원칙입니다.

- 문제점: 앞서 언급한 Redis의 비동기 복제 특성 때문에, Failover(장애 조치) 과정에서 데이터 유실이 발생할 가능성이 상존합니다. Redis Sentinel이나 Cluster가 새로운 Master를 선출하는 과정에서, 기존 Master에만 존재하던 최신 세션 데이터는 증발합니다.
- 현실적 영향: 사용자는 분명 장바구니에 물건을 담았는데, 페이지를 새로고침 하자마자 장바구니가 비어버리는 경험을 하게 됩니다. 이는 단순 버그가 아니라 Redis 아키텍처가 가진 구조적 한계입니다.

2) 복잡한 확장성과 운영의 병목 (Sharding Complexity)

Kubernetes와 같은 클라우드 환경은 트래픽에 따라 서버가 자동으로 늘어나고 줄어드는 오토스케일링(Auto-scaling)이 핵심입니다. 하지만 Redis Cluster는 이러한 탄력성과 거리가 멉니다.

- 해시 슬롯(Hash Slot)의 제약: Redis Cluster는 데이터를 저장할 때 16,384개의 고정된 슬롯에 데이터를 나누어 담습니다. 노드를 추가하거나 제거하려면 이 슬롯들을 수동으로 재분배(Resharding)해야 합니다. 이는 운영자의 개입이 필수적이며, 자동화를 어렵게 만듭니다.
- 개발 복잡도 증가: 트랜잭션을 지원하기 위해 개발자는 `{user_id}`와 같은 해시 태그(Hash Tag)를 사용하여 강제로 같은 노드에 데이터가 저장되도록 키 설계를 해야 합니다. 이는 인프라의 제약 사항이 애플리케이션 코드에 침투하는 강한 결합(Tight Coupling)을 유발합니

다. 또한, 특정 노드에만 트래픽이 몰리는 'Hot Key' 문제가 발생할 경우 전체 클러스터의 성능을 떨어뜨리는 원인이 됩니다.

3) 예측 불가능한 성능 저하 (Latency Spike)

세션 저장소는 언제나 일정한 응답 속도를 보장해야 합니다. 하지만 Redis의 내부 동작 방식은 간헐적인 멈춤 현상을 유발합니다.

- **`fork()`와 Copy-on-Write의 비용:** Redis는 디스크에 데이터를 저장(RDB/AOF Rewrite)할 때 `fork()` 시스템 콜을 호출하여 자식 프로세스를 만듭니다. 이때 운영체제는 메모리 페이지 테이블을 복사하는데, 메모리 사용량이 클수록(예: 수십 GB) 이 작업은 수백 밀리초에서 수 초까지 메인 스레드를 멈추게(Blocking) 할 수 있습니다.
- **단일 스레드(Single-threaded)의 한계:** Redis는 명령어를 처리하는 스레드가 단 하나입니다. 이는 복잡한 연산(O(N) 명령어 등)이 하나라도 들어오면, 그 뒤에 줄 서 있는 수천 개의 간단한 세션 조회 요청들이 모두 대기해야 함(Head-of-Line Blocking)을 의미합니다. 64코어, 128코어 서버를 사용해도 단 하나의 코어만 사용하는 구조는 현대 하드웨어의 성능을 온전히 활용하지 못하는 비효율적인 구조입니다.

결론: Redis는 훌륭한 캐시지만, 위험한 세션 저장소다

"쓸 수 있음"과 "맡겨도 됨"은 전혀 다른 문제 입니다. Redis는 훌륭한 기술이며, 캐시·세션 보조·일시적 상태 관리에는 여전히 강력한 도구입니다.

하지만 미션 크리티컬한 세션 관리에 Redis를 사용하는 것은 "회피할 수 있는 아키텍처 리스크를 굳이 떠안는 것"과 같습니다. 개발 단계에서의 편리함은 잠깐이지만, 운영 단계에서의 데이터 유실, 확장 불가, 성능 지연은 지속적인 기술 부채가 됩니다.

데이터의 강력한 정합성(Strong Consistency)과 운영의 완전 자동화, 그리고 멀티 스레드 기반의 예측 가능한 성능이 필요한 조직이라면, Redis가 아닌 고가용성을 위해 설계된 In-Memory Data Grid (IMDG) 솔루션 도입은 단순한 대안이 아닌, 비즈니스 연속성을 위한 필수 전략입니다.

제5장. Redis 기반 세션 운영에서 드러나는 현실적 문제: 이상과 현실의 괴리

IT 인프라를 설계하는 초기 단계에서 Redis는 마치 '만능열쇠'처럼 매력적인 선택지로 다가옵니다. 디스크가 아닌 메모리에서 데이터를 처리하는 인메모리(In-Memory) Key-Value 저장소라는 특성 덕분에 압도적인 속도를 자랑하며, 개발자들은 복잡한 설정 없이도 웹 애플리케이션 서버(WAS) 간의 세션 공유(Clustering)를 손쉽게 구현할 수 있기 때문입니다. 실제로 스타트업이나 트래픽이 적은 초기 서비스 단계에서 Redis는 가성비가 뛰어난 훌륭한 대안이 됩니다.

하지만 서비스가 성장하여 엔터프라이즈급 대규모 트래픽 환경에 진입하는 순간, 초기의 그 '단순함'은 통제하기 힘든 '복잡성'으로 돌변합니다. 수십만 명의 동시 접속자, 사용자의 행동에 따라 예측 불가능하게 커지는 세션 데이터, 그리고 단 1초의 중단도 허용하지 않는 가용성(Availability) 요구사항 앞에서 Redis는 숨겨져 있던 아키텍처의 한계를 드러냅니다.

이러한 한계는 단순한 속도 저하에 그치지 않고, 시스템 전체의 병목(Bottleneck), 데이터의 영구적 유실, 그리고 운영 비용의 기하급수적 증가라는 심각한 경영 리스크로 이어집니다. 본 챕터에서는 Redis 기반 세션 운영이 실제 대규모 운영 환경(Production)에서 부딪히는 구조적 모순과, 왜 이것이 단순한 튜닝으로는 해결할 수 없는 아키텍처적 문제인지를 심층 분석합니다.

5.1 성능과 구조의 문제: 속도 뒤에 숨겨진 함정

이 섹션에서는 Redis의 핵심 아키텍처인 '싱글 스레드(Single-Threaded)' 모델과 '인메모리 데이터 구조'가 대규모 세션 처리 환경에서 어떻게 시스템의 발목을 잡는지 분석합니다. 작은 세션 데이터의 변경이 나비효과처럼 전체 시스템의 응답 속도를 떨어뜨리고, 메모리라는 한정된 자원이 어떻게 서비스 전체를 셧다운(Shutdown) 시킬 수 있는지 기술적 원리를 파헤칩니다.

5.1.1 전체 세션 직렬화(Serialization)와 재작성(Full-Rewrite)의 딜레마

Redis를 세션 저장소로 사용할 때 발생하는 가장 비효율적인 문제는 데이터가 처리되는 '단위'에 있습니다.

일반적인 WAS의 세션 관리자는 Redis의 `String` 타입을 사용하여 세션 객체를 저장합니다.

이때, 사용자가 쇼핑몰에서 장바구니에 양말 한 켤레를 추가했다고 가정해 봅시다. 세션 데이터 전체 크기가 10KB이고 변경된 데이터는 단 10Byte에 불과하더라도, 시스템은 10KB 전체를 다시 처리해야 합니다.

1. 직렬화(Serialization) 오버헤드: WAS는 변경된 10Byte를 포함한 세션 객체 전체를 바이트 배열(Byte Array)로 변환(직렬화)해야 합니다. 이 과정은 CPU 연산을 많이 소모하며, 세션 객체가 복잡하고 클수록(예: Java의 Nested Object 등) 소요 시간은 $O(N)$으로 증가합니다.
2. 전체 재작성(Full-Rewrite)과 네트워크 낭비: Redis는 구조상 객체 내부의 필드만 부분적으로 수정하는 것이 까다롭습니다(String 타입 사용 시). 따라서 WAS는 거대해진 직렬화 데이터를 네트워크를 통해 Redis로 전송하고, Redis는 기존 데이터를 지우고 전체를 다시 쓰는 작업을 반복합니다. 이는 네트워크 대역폭(Bandwidth)을 낭비할 뿐만 아니라, 패킷 분할(Fragmentation)로 인한 TCP/IP 처리 비용을 증가시킵니다.
3. 싱글 스레드(Single-Threaded)의 병목 현상: 이것이 가장 치명적인 문제입니다. Redis는 기본적으로 단 하나의 스레드가 모든 명령을 순차적으로 처리합니다. 만약 1MB짜리 거대 세션을 저장(SET 명령)하는 요청이 들어오면, Redis는 이 작업을 완료할 때까지 뒤에 줄 서 있는 다른 수천 개의 가벼운 조회 요청(GET)들을 전혀 처리하지 못하고 대기시킵니다. 이를 'Head-of-Line Blocking' 현상이라고 하며, 특정 사용자의 무거운 세션 하나가 전체 서비스의 응답 속도(Latency)를 급격히 떨어뜨리는 원인이 됩니다.

물론 여러 명령을 한 번에 보내는 파이프라이닝(Pipelining) 기술이 존재하지만, 이는 네트워크 왕복(RTT) 횟수를 줄일 뿐, "매번 전체 데이터를 다시 쓰고, 싱글 스레드를 점유한다"는 근본적인 직렬화 및 쓰기 오버헤드는 해결하지 못합니다. 즉, 서비스가 커질수록 CPU와 네트워크 비용은 비선형적으로 급증하게 됩니다.

5.1.2 메모리 의존성이 초래하는 '시한폭탄' 같은 장애 구조

Redis는 디스크(SSD/HDD)가 아닌 RAM(메모리)에 모든 데이터를 저장합니다. 이 구조는 빠른 읽기/쓰기를 보장하지만, 데이터 보존과 용량 관리 측면에서는 치명적인 리스크를 안고 있습니다.

1. 스와핑(Swapping)으로 인한 성능 추락: RAM은 용량 대비 가격이 비싸고 물리적 한계가 명확합니다. 세션 데이터가 급증하여 Redis가 할당된 물리 메모리를 초과해 사용하는 순

간, 운영체제(OS)는 부족한 메모리를 확보하기 위해 메모리 상의 데이터를 디스크의 스왑 영역(Swap area)으로 내리는 '스와핑'을 시작합니다.

- 메모리 접근 속도는 나노초(ns) 단위인 반면, 디스크 접근은 밀리초(ms) 단위입니다. 속도 차이가 수천 배에서 수십만 배까지 납니다.
- 한 번 스와핑이 발생하면 Redis의 응답 속도는 즉시 바닥으로 떨어지며, 이는 사실상 서비스 불능(Outage) 상태와 다름없습니다.

2. OOM Killer의 위협 (강제 종료): 상황이 더 악화되어 스왑 영역조차 부족해지면, 리눅스 커널의 보호 기제인 OOM(Out Of Memory) Killer가 작동합니다. OOM Killer는 시스템 전체의 붕괴를 막기 위해 메모리를 가장 많이 사용하는 프로세스를 강제로 사살(Kill)하는데, 그 1순위 타겟은 보통 Redis입니다.
    - 결과적으로 Redis 서버가 갑자기 꺼지며, 메모리에 있던 모든 로그인 정보와 장바구니 데이터가 순식간에 증발해 버립니다.
3. 데이터 영속성(Persistence) 확보 시 발생하는 '멈춤(Stall)': 데이터 유실을 막기 위해 디스크에 스냅샷을 저장하는 RDB(Redis Database) 방식을 사용해도 문제입니다.
    - RDB 저장을 위해 Redis는 `fork()`라는 시스템 호출을 사용하여 자식 프로세스를 생성합니다.
    - 이때 운영체제는 부모 프로세스의 메모리 페이지 테이블(Page Table)을 복사해야 합니다. Copy-on-Write(CoW) 메커니즘 덕분에 실제 데이터를 다 복사하진 않지만, 메모리 매핑 테이블을 복사하는 것만으로도 상당한 시간이 소요됩니다.
    - 예를 들어 24GB 크기의 Redis라면 페이지 테이블 크기만 약 48MB에 달할 수 있습니다. 특히 Xen이나 AWS EC2와 같은 가상화 환경에서는 이 메모리 할당 및 복사 작업이 물리 서버보다 훨씬 느려, 그 순간 동안 Redis가 멈추는 'Latency Spike(순간적인 지연 피크)'가 발생합니다.

결론적으로, 메모리 용량을 예측하기 힘든 세션 데이터를 Redis에 저장하는 것은, 언제 터질지 모르는 메모리 폭주와 서비스 중단의 위험(Technical Debt)을 안고 운영하는 것과 같습니다. 이는 안정성이 최우선인 엔터프라이즈 아키텍처에서는 결코 용납될 수 없는 리스크입니다.

다음 섹션에서는 이러한 장애 발생 시 데이터 유실 위험과 문제 해결의 어려움에 대해 더 깊이 분석하겠습니다.

5.2 장애 대응의 한계와 트러블슈팅의 복잡성: 숨겨진 운영 비용

안정적인 엔터프라이즈 시스템의 핵심은 단순히 평상시의 성능이 빠른 것에 그치지 않습니다. 진정한 안정성은 '장애가 발생했을 때 데이터가 안전한가(정합성)'와 '문제를 얼마나 빨리 파악하고 복구할 수 있는가(가시성)'에 달려 있습니다.

이 섹션에서는 Redis 기반 세션 클러스터가 가진 구조적 한계로 인해 장애 상황에서 데이터 정합성이 깨지는 매커니즘을 심층 분석하고, 문제 발생 시 원인을 규명하는 것이 왜 '모래사장에서 바늘 찾기'처럼 어려운지를 운영 관점에서 상세히 기술합니다. 이를 통해 단순히 라이선스 비용과 같은 눈에 보이는 비용(Hard Cost) 너머에 존재하는, 운영 인력의 시간과 비즈니스 리스크라는 막대한 숨겨진 비용(Hidden Cost)을 조명합니다.

5.2.1 구조적 한계: 비동기 복제와 필연적인 세션 유실 (CAP 이론 관점)

Redis는 분산 시스템 이론인 CAP 정리(Consistency, Availability, Partition Tolerance) 관점에서 볼 때, 강력한 일관성(C)보다는 가용성(A)과 분할 내성(P), 그리고 무엇보다 속도에 초점을 맞춘 솔루션입니다. 이러한 설계 철학은 Primary(마스터) – Replica(슬레이브) 구조와 비동기 복제(Asynchronous Replication) 방식에서 명확히 드러나며, 바로 이 지점이 엔터프라이즈 환경에서 데이터 유실의 근본 원인이 됩니다.

1) '응답 속도'를 위해 희생된 '데이터 안전성' (비동기 복제의 함정)

Redis는 쓰기 성능을 극대화하기 위해 Primary 노드가 클라이언트로부터 데이터를 받으면, Replica 노드에 복제가 완료되기를 기다리지 않고 즉시 "저장 성공(OK)" 응답을 보냅니다. 이를 비동기 복제라고 합니다.

- 정상 상황: Primary에 저장된 데이터는 밀리세컨드(ms) 단위의 시차를 두고 Replica로 복제됩니다.

- 장애 시나리오 (죽음의 틈, Gap of Death):

  1. 쓰기 요청: 사용자가 로그인하여 세션 생성 요청을 보냅니다. Primary는 메모리에 기록하고 “성공”을 응답합니다.
  2. 복제 지연 및 장애: 데이터가 네트워크를 타고 Replica로 넘어가기 직전(수 ms 사이), Primary 노드에 전원 차단이나 하드웨어 장애가 발생합니다.
  3. 데이터 증발: Redis Sentinel(감시자)이나 클러스터 관리 도구가 장애를 감지하고 Replica를 새로운 주인(Primary)으로 승격시키는 Failover(장애 조치)를 수행합니다. 하지만 승격된 Replica는 방금 기록된 로그인 정보를 받지 못한 상태입니다. 결과적으로 사용자의 로그인 정보는 그 누구도 가지고 있지 않은 상태가 되어 영구 유실됩니다.

2) 최종 일관성(Eventual Consistency) vs 강력한 일관성(Strong Consistency)

Redis의 이러한 특성은 “언젠가는 데이터가 같아진다”는 최종 일관성 모델입니다. 이는 SNS의 ‘좋아요’ 수치나 조회수 캐싱 등 약간의 오차가 허용되는 서비스에는 적합합니다.

하지만, 금융 거래, 결제, 장바구니, 로그인 세션과 같이 단 한 건의 데이터 유실도 치명적인(Mission Critical) 엔터프라이즈 업무는 모든 노드가 항상 동일한 데이터를 유지하는 강력한 일관성을 요구합니다.

> 비교: IMDG(In-Memory Data Grid)의 P2P 아키텍처 엔터프라이즈급 IMDG 솔루션은 Redis의 Primary/Replica 구조와 달리 P2P(Peer-to-Peer) 방식을 채택합니다. 모든 노드가 동등한 권한을 가지며, 데이터 저장 시 백업 노드에 복제가 완료된 것을 확인한 후 성공 응답을 보내는 옵션(Sync Replication)을 제공합니다. 또한 장애 발생 시 별도의 승격 절차 없이 살아있는 노드들이 데이터를 자동으로 재분배하여 단일 장애점(SPoF)을 원천적으로 제거합니다.

결론적으로 Redis의 HA 구조는 태생적으로 “장애 시, 운이 나쁘면 최근 데이터는 잃어버릴 수 있다”는 위험을 안고 있습니다. 이는 데이터 무결성이 생명인 엔터프라이즈 세션 관리에는 부적합한 특성입니다.

5.2.2 “원인이 어디죠?” 분산된 장애 포인트와 트러블슈팅의 늪

운영 중에 “로그인이 자꾸 풀려요” 혹은 “결제 중 세션이 만료되었습니다”라는 리포트가 접수되면 운영팀은 긴장하게 됩니다. Redis 기반 시스템은 애플리케이션 – 네트워크 – Redis 서버로 이어지는 경로가 복잡하여, 장애의 원인이 여러 계층에 흩어져(Distributed) 있기 때문입니다.

이를 추적하는 것은 마치 안개 속에서 범인을 찾는 것과 같으며, 각 계층별로 다음과 같은 복잡한 기술적 문제들이 숨어 있을 수 있습니다.

1. 애플리케이션 계층 (WAS): 직렬화와 로직의 문제

- 직렬화(Serialization) 오류: 자바 객체(Session Object)를 Redis에 저장하기 위해 바이트(Byte) 형태로 변환하거나 다시 복구하는 과정에서 버전 불일치나 포맷 오류가 발생할 수 있습니다. 이는 겉으로는 Redis 문제처럼 보이지만 실제로는 코드 레벨의 문제입니다.
- 커넥션 풀(Connection Pool) 고갈: WAS에서 Redis로 연결할 수 있는 연결(Connection) 개수가 꽉 차서, Redis는 멀쩡한데 애플리케이션이 접근을 못 해 타임아웃이 발생하는 경우입니다.

2. Redis 서버 계층: 싱글 스레드의 한계와 메모리 정책

- 싱글 스레드(Single Thread) 블로킹: Redis는 기본적으로 한 번에 하나의 명령어만 처리하는 싱글 스레드 아키텍처입니다. 만약 개발자가 실수로 `KEYS *` (모든 키 조회)와 같은 무거운 명령어를 실행하면, 그 작업이 끝날 때까지 수천, 수만 개의 다른 세션 요청(로그인 등)은 대기 상태에 빠지거나 타임아웃(Blocking) 됩니다.
- 메모리 축출(Eviction) 정책: 메모리가 가득 찼을 때 Redis는 설정된 정책(예: `allkeys-lru`)에 따라 기존 데이터를 강제로 삭제합니다. 사용자는 아직 로그아웃하지 않았는데 Redis가 메모리 확보를 위해 해당 세션을 지워버리는 상황이 발생하며, 이는 “원인 불명의 로그아웃”으로 나타납니다.

3. 네트워크 계층: 보이지 않는 유령

- TCP 재전송 및 패킷 유실: WAS와 Redis 사이의 네트워크 스위치나 방화벽 문제로 패킷이 유실되면, TCP 레벨에서 재전송이 일어나면서 응답 속도가 급격히 느려집니다.
- 마이크로버스트(Micro-burst): 순간적으로 트래픽이 폭주하여 네트워크 대역폭을 초과하는 현상으로, 모니터링 툴의 평균 그래프에는 잡히지 않지만 실제로는 세션 연결을 끊어버리는 원인이 됩니다.

결론: TCO(총소유비용)의 증가

이처럼 장애 원인이 애플리케이션 코드, Redis 내부 동작, 네트워크 환경 등 다각도에 걸쳐 있기에, 문제를 해결하기 위해서는 개발자, 시스템 엔지니어, 네트워크 관리자가 모두 모여 로그를 대조해야 합니다. 이는 평균 복구 시간(MTTR, Mean Time To Repair)을 기하급수적으로 늘리며, 결과적으로 기업의 총소유비용(TCO)을 증가시키는 주된 원인이 됩니다.

제6장. IMDG(In-Memory Data Grid): 클라우드 시대, 세션 관리의 필연적 선택

IT 인프라가 온프레미스(On-Premise)에서 클라우드 네이티브(Cloud-Native) 환경으로 급격히 이동하면서, 애플리케이션 아키텍처의 핵심 가치는 '유연한 확장성(Scalability)'과 '자동 복구(Resiliency)'로 바뀌었습니다.

과거에는 거대한 모놀리식 WAS(Web Application Server)가 고정된 서버에서 사용자 세션을 메모리에 꽉 쥐고 있었습니다(Sticky Session). 하지만 Kubernetes와 같은 컨테이너 환경에서는 Pod(애플리케이션 인스턴스)가 수시로 생겨나고 사라집니다. 만약 서버가 꺼질 때마다 사용자가 로그아웃되거나 장바구니가 비워진다면, 이는 치명적인 사용자 경험(UX) 저하로 이어집니다.

따라서 세션 데이터를 애플리케이션 서버 밖으로 빼내어 저장하는 '세션 외부화(Session Externalization)'는 선택이 아닌 필수 생존 전략이 되었습니다. 흔히 '빠르다'는 이유로 Redis

같은 Key-Value 스토어를 먼저 떠올리지만, 엔터프라이즈 환경에서 요구하는 데이터의 정합성(Consistency)과 운영 안정성을 고려할 때 단순 캐시 솔루션은 한계에 부딪힙니다.

이 챕터에서는 단순한 저장소를 넘어, 분산 환경에서 복잡한 데이터를 안전하게 다루기 위해 태어난 IMDG(In-Memory Data Grid)가 왜 현대적 세션 관리의 '표준(Standard)'이 될 수밖에 없는지, 그 기술적 우위를 심층 해부합니다.

6.1 IMDG의 세션 친화적 설계 철학: 속도를 넘어 '정확성'으로

IMDG가 엔터프라이즈 세션 관리에 최적화된 이유는 단순히 메모리를 사용해 빠르기 때문만이 아닙니다. 그보다는 "세션 데이터는 단순한 텍스트가 아니라, 상태(State)를 가진 객체(Object)다"라는 본질을 꿰뚫고 설계되었기 때문입니다. 이 섹션에서는 데이터 모델과 동시성 제어라는 두 가지 측면에서 IMDG와 일반적인 NoSQL(Redis 등)의 차이를 분석합니다.

6.1.1 Object-aware 모델: 데이터를 '이해'하는 저장소

IMDG와 단순 Key-Value 저장소(Redis 등)의 가장 결정적인 차이는 저장소가 보관하고 있는 데이터를 얼마나 이해하고 있느냐에 있습니다.

1) IMDG의 객체 인식(Object-Aware) 모델: "있는 그대로 저장한다"

Java의 `HttpSession` 객체 안에는 사용자 ID 같은 단순 문자열뿐만 아니라, `User` 객체, `List<Product>` 형태의 장바구니 등 복잡한 계층 구조를 가진 객체들이 들어있습니다.

- 작동 방식: IMDG(예: Hazelcast, Infinispan 등)는 Java 객체 그 자체를 1급 시민으로 대우합니다. 애플리케이션이 세션 객체를 넘기면, IMDG는 이를 별도의 복잡한 변환 과정 없이 직렬화(Serialization)하여 분산 메모리에 저장합니다. 필요할 경우 서버 측에서 코드(EntryProcessor)를 실행하여 데이터를 가져오지 않고도 그리드 내부에서 직접 수정할 수도 있습니다.
- 장점: 개발자는 "데이터를 어떻게 저장 포맷에 맞출까"를 고민할 필요 없이 비즈니스 로직에만 집중할 수 있습니다.

2) Redis의 Key-Value 모델: “블랙박스로 저장한다”

반면, Redis는 기본적으로 데이터를 `String`이나 `Byte Array`로 취급합니다. Redis 입장에서 세션 데이터는 그저 '의미를 알 수 없는 긴 문자열'일 뿐입니다.

- 번거로운 변환 과정: 개발자는 세션 객체를 Redis에 넣기 위해 JSON이나 별도의 바이너리 포맷으로 직접 변환(Marshalling)해야 하고, 읽을 때 다시 객체로 조립(Unmarshalling)해야 합니다. 이는 CPU 비용을 발생시킵니다.
- 운영상의 리스크 (Cluster 환경의 Hash Tag 문제): 만약 사용자의 '프로필 정보'와 '장바구니'를 트랜잭션으로 묶어 동시에 업데이트해야 한다고 가정해 봅시다. Redis 클러스터 환경에서는 데이터가 여러 노드(Shard)에 분산 저장되는데, 서로 다른 노드에 있는 키에 대해서는 원자적(Atomic) 연산이 불가능합니다 (`CROSSSLOT` 에러 발생).
  - 이를 해결하려면 개발자는 `user:{1234}:profile`, `user:{1234}:cart` 처럼 키 안에 **`{1234}`** 같은 해시 태그(Hash Tag)를 강제로 넣어, 두 데이터가 무조건 같은 노드에 저장되도록 유도해야 합니다.
  - 치명적 단점: 이렇게 되면 특정 '헤비 유저'의 모든 데이터가 단 하나의 노드에 몰리게 됩니다. 이는 '데이터 핫스팟(Hot Spot)'을 유발하여, 클러스터 전체의 밸런스를 무너뜨리고 해당 노드의 성능 병목을 초래합니다.

결론적으로, IMDG는 데이터의 구조를 이해하고 자동으로 분산 관리해 주는 '스마트한 창고'라면, Redis는 사용자가 직접 짐을 포장해서 지정된 위치에 넣어야 하는 '단순 물품 보관함'에 비유할 수 있습니다.

6.1.2 세션 단위 락(Lock)과 동시성 제어: 수천 명의 동시 접속도 안전하게

쇼핑몰의 '블랙 프라이데이' 이벤트를 상상해 보십시오. 수만 명의 사용자가 동시에 버튼을 누르고, 백그라운드에서는 결제, 재고 차감, 세션 갱신이 동시에 일어납니다. 이때 데이터가 꼬이지 않게 하는 동시성 제어(Concurrency Control) 능력은 시스템의 생명과도 같습니다.

1) IMDG의 정교한 제어: MVCC와 분산 락

IMDG는 관계형 데이터베이스(RDBMS)급의 데이터 무결성을 메모리 위에서 구현합니다.

- MVCC (Multi-Versioned Concurrency Control): 데이터베이스에서 주로 쓰이는 기술로, 데이터에 버전을 매겨 관리합니다. 누군가 세션을 수정하고 있어도(Write), 다른 사용자는 수정 전의 안전한 데이터를 즉시 조회(Read)할 수 있습니다. 즉, 읽기 작업이 쓰기 작업 때문에 기다리지 않아도 되어 높은 성능을 유지합니다.
- 세밀한 분산 락(Distributed Lock): 포인트 차감과 세션 변경이 동시에 성공해야 한다면? IMDG는 JTA(Java Transaction API)와 연동하여 트랜잭션을 지원합니다. `map.lock(sessionId)` 한 줄로 특정 세션 객체만 '잠금' 처리할 수 있습니다. 락의 범위가 객체 단위로 매우 좁기(Fine-grained) 때문에, 다른 사용자의 작업에는 전혀 영향을 주지 않습니다.

2) Redis의 아키텍처적 한계: 싱글 스레드의 위험성

Redis는 고성능을 위해 싱글 스레드(Single Threaded) 기반의 이벤트 루프 모델을 사용합니다. 이는 평소에는 매우 빠르지만, 복잡한 상황에서는 양날의 검이 됩니다.

- Stop-the-World 위험: 싱글 스레드 특성상, 하나의 명령어가 처리되는 동안 다른 모든 명령어는 대기해야 합니다. 만약 개발자가 디버깅을 위해 `KEYS *` 같이 전체 데이터를 훑는 명령어(O(N) 복잡도)를 실수로 실행하거나, 큰 데이터를 처리하느라 시간이 지체되면 해당 샤드(Shard)는 멈춰버립니다.
- 연쇄적 지연(Cascading Latency): 단 하나의 느린 명령이 뒤따라오는 수천 개의 가벼운 세션 조회 요청들을 줄줄이 지연시킵니다. 이는 전체 서비스의 응답 속도 저하로 직결됩니다.

요약하자면: Redis는 "최대한 빨리 처리한다"는 목표로 속도에 올인한 구조라면, IMDG는 "동시에 여러 요청이 와도 데이터가 깨지지 않게 안전하게 처리한다"는 안정성과 정확성(Consistency)에 방점을 둔 구조입니다. 금융 거래나 결제 정보가 포함된 엔터프라이즈 세션 관리에서 IMDG가 더 적합한 이유가 바로 여기에 있습니다.

제7장. 세션 관점에서 본 Redis와 IMDG의 본질적 차이 (심화 분석)

기업의 IT 환경에서 '세션(Session)'은 사용자의 로그인 상태, 장바구니 정보, 결제 진행 단계 등을 담고 있는 핵심 자산입니다. 이 세션을 어디에 저장하느냐는 단순한 저장소 선택의 문제를 넘어, 서비스의 생존성과 직결됩니다. 이번 장에서는 대표적인 인메모리 솔루션인 Redis와 IMDG(In-Memory Data Grid)가 세션을 다루는 방식에서 어떤 근본적인 차이가 있는지, 외부 기술 이론과 실제 운영 시나리오를 곁들여 상세히 분석합니다.

7.1 세션 저장소 평가 기준: 무엇을 보고 골라야 하는가?

미션 크리티컬(Mission Critical)한 서비스, 즉 금융 거래나 대규모 이커머스 트래픽을 처리하는 시스템에서는 단순한 '초당 처리량(TPS)'보다 더 중요한 가치들이 존재합니다. 바로 데이터가 사라지지 않는가(정합성), 시스템을 쉽게 키울 수 있는가(확장성), 장애 상황에서도 멈추지 않는가(안정성)입니다.

7.1.1 정합성·확장성·운영 안정성·장애 대응 상세 비교

Redis와 IMDG는 메모리를 사용한다는 점은 같지만, 탄생 배경과 설계 철학이 다릅니다. 이로 인해 아래 4가지 핵심 기준에서 뚜렷한 차이를 보입니다.

기업의 미션 크리티컬 서비스를 지탱하는 세션 저장소 기술을 선택할 때, 단순한 성능 벤치마크 수치를 넘어 아키텍처의 근본적인 특성을 평가하는 것은 매우 중요합니다. 데이터 정합성, 확장성, 운영 안정성, 그리고 장애 대응 능력은 시스템의 생존성과 직결되는 핵심 평가 지표이며, 이 기준들을 통해 각 기술이 제공하는 보증 수준의 차이를 명확히 이해할 수 있습니다.

아래 표는 이 네 가지 핵심 기준에 대해 Redis와 In-Memory Data Grid(IMDG)가 어떤 구조적 차이를 보이는지 비교 분석한 것입니다.

평가 기준	Redis의 접근 방식	In-Memory Data Grid(IMDG)의 접근 방식
데이터 정합성 (Consistency)	비동기(Asynchronous) 복제 방식을 기본으로 하여, 장애 발생 시 Primary(마스터) 노드의 최신 데이터가 Replica(슬레이브)로 전파되지 못해 데이터 유실이 발생할 수 있는 '최종 일관성(Eventual Consistency)' 모델을 따릅니다.	동기식(Synchronous) 복제를 지원하여 데이터 쓰기 작업이 클러스터 내 여러 노드에 성공적으로 복제되었음을 확인한 후에야 완료됩니다. 이는 장애 발생 시에도 데이터 유실 없는 '강력한 일관성(Strong Consistency)'을 보장합니다.
확장성 (Scalability)	16,384개의 고정된 해시 슬롯(Hash Slot)을 사용하는 Primary/Replica 샤딩 모델을 기반으로 합니다. 클러스터에 노드를 추가할 때, 관리자가 수동으로 슬롯을 재분배(Resharding)해야 하는 정적인 구조로, 운영 복잡성이 높고 확장 작업 중 휴먼 에러 발생 가능성이 내재되어 있습니다.	클러스터의 모든 노드가 동등한 역할을 하는 P2P(Peer-to-Peer) 아키텍처를 채택합니다. 신규 노드 추가 시 데이터 파티션이 자동으로 재조정(Rebalancing)되어 부하와 데이터를 분산시키므로, 탄력적이고 선형적인 확장이 가능합니다.
운영 안정성 (Operational Stability)	백그라운드 저장을 위한 `fork` 동작 시 시스템에 상당한 지연(latency)을 유발할 수 있습니다. 메모리 부족 시 eviction 정책이 없으면 쓰기 작업이 중단될 수 있고, eviction 발생 시 쓰기 지연이 증가합니다.	세션 데이터를 WAS의 생명주기에서 완전히 분리된 외부 메모리 공간에서 관리하여, WAS의 Full GC나 OOM(Out of Memory) 장애가 세션 데이터에 영향을 미치지 않습니다. 이는 예측 가능한 안정성을 제공합니다.
장애 대응 (Fault Tolerance)	Sentinel 또는 Cluster 모드를 통해 자동 Failover를 지원하지만, 비동기 복제 방식의 한계로 인해 Failover 과정에서 일부 세션 데이터가 유실될 수 있습니다.	P2P 아키텍처와 데이터 복제본을 통해 특정 노드에 장애가 발생해도 데이터 유실 없이 다른 노드가 즉시 서비스를 이어받는 Self-Healing 구조를 갖추고 있습니다. 이는 단일 장애 지점(SPOF)을 제거합니다.

표에서 드러나듯, 두 기술의 설계 철학은 근본적으로 다릅니다. Redis의 설계는 속도를 위해 데이터 정합성을 희생하는 전략적 트레이드오프입니다. 이는 캐시 용도로는 합리적일 수 있으나, 세션 데이터 유실은 곧 고객의 장바구니 소실, 로그인 재시도, 최악의 경우 결제 실패로 이어져 직접적인 매출 손실과 고객 신뢰도 하락을 유발하는 비즈니스 리스크입니다. 반면, IMDG는 이러한 리스크를 원천적으로 제거하는 데 설계 초점을 맞춘 것입니다.

이러한 구조적 차이는 자연스럽게 개발 및 운영의 책임 범위에도 영향을 미칩니다. 다음 섹션에서는 이 문제를 더 깊이 살펴보겠습니다.

7.1.2 개발 편의성과 운영 책임의 균형: 빙산의 일각

기술 도입 시, 개발 단계에서의 편의성("얼마나 빨리 만들 수 있는가")뿐만 아니라, 향후 5년 동안의 운영 비용("얼마나 편하게 유지할 수 있는가")을 고려해야 합니다. 이를 총소유비용(TCO, Total Cost of Ownership) 관점에서 비교해 봅니다.

Redis의 관점: "빠른 시작, 그러나 무거운 운영의 짐"

- 개발 단계 (Easy): Redis는 `String`, `List`, `Hash` 등 직관적인 자료구조와 수많은 클라이언트 라이브러리를 제공합니다. 개발자는 마치 로컬 변수를 다루듯 쉽게 코드를 짤 수 있어 초기 진입 장벽이 매우 낮습니다. 'Hello World' 수준의 구현은 몇 분이면 끝납니다.
- 운영 단계 (Hard): 하지만 서비스 규모가 커지면 숨겨진 비용이 드러납니다.
  - 노드 추가 시 관리자가 직접 슬롯을 계산하여 옮겨야 합니다.
  - 장애 발생 시 데이터 유실 여부를 사람이 직접 확인하고 복구 스크립트를 돌려야 할 수 있습니다.
  - 결론: 시스템이 해야 할 복잡한 일(확장, 정합성 보장)을 사람(운영자)의 시간과 노력으로 메우는 구조입니다. 이는 장기적으로 고급 엔지니어의 인건비 상승과 운영 피로도로 이어집니다.

IMDG의 관점: "초기 학습 비용, 그러나 자동화된 미래"

- 개발 단계 (Moderate): IMDG는 단순 캐시를 넘어 분산 컴퓨팅, 분산 락, 트랜잭션 처리 등 엔터프라이즈 기능을 제공하므로 초기 설정과 학습 곡선이 존재합니다. Redis보다 무겁게 느껴질 수 있습니다.
- 운영 단계 (Easy): 구축이 완료되면 IMDG는 '자율 주행' 모드에 가깝습니다.
  - 트래픽이 폭증해 서버를 늘리면 데이터가 알아서 분산됩니다.
  - 서버 한 대가 고장 나도 시스템이 알아서 복구하고 관리자에게는 "복구 완료" 알림만 보냅니다.
  - 결론: 초기 라이선스 비용이나 학습 비용이 들더라도, 운영의 책임을 시스템(플랫폼)에게 위임하는 구조입니다. 대규모 장애 상황에서도 운영팀은 패닉에 빠지지 않고 안

정적으로 대응할 수 있습니다.

요약: 단순한 '도구'인가, 지능형 '플랫폼'인가?

결국 Redis와 IMDG의 선택은 "사용하기 쉬운 도구를 쥐어주고 결과에 대한 책임은 사용자가 지게 할 것인가(Redis)" 아니면 "사용법은 조금 복잡해도 결과와 안정성을 시스템이 보장해 줄 것인가(IMDG)"의 차이로 귀결됩니다. 비즈니스의 중요도가 높고 데이터 유실이 치명적인 서비스라면, 초기 비용을 감수하더라도 운영 리스크를 시스템적으로 제거한 IMDG 방식이 합리적인 선택이 될 수 있습니다.

7.2 Redis vs IMDG 심층 비교: 엔터프라이즈 세션 관리를 위한 선택

7.2.1 Redis: 범용성 높은 '고속 캐시' 설계의 태생적 한계

Redis는 전 세계적으로 가장 사랑받는 인메모리 저장소이며, 단순한 데이터 조회 속도 면에서는 타의 추종을 불허합니다. 하지만 Redis의 탄생 목적과 핵심 설계 철학은 '최대한 빠르고 단순한 범용 Key-Value 캐시'에 맞춰져 있습니다.

속도를 위해 희생된 '데이터의 신뢰성'과 '복잡한 연산 능력'은, 엔터프라이즈 환경에서 세션(Session)이라는 '사용자의 현재 상태(State)'를 관리할 때 치명적인 아키텍처 불일치를 일으킵니다. 단순히 "빠르니까 쓴다"는 접근이 비즈니스에 어떤 리스크를 주는지 네 가지 관점에서 상세히 분석합니다.

- 데이터 모델의 한계: "내용을 모르는 블랙박스 저장소" (Opaque Data Store)
  - 문제점: Redis는 기본적으로 데이터를 `Key`와 `Value`로만 저장합니다. 이때 `Value`는 Redis 입장에서 '알 수 없는 거대한 문자열 덩어리(Blob)'일 뿐입니다. 세션 객체 안에 '장바구니 금액'이 있는지, '로그인 시간'이 있는지는 Redis는 전혀 모릅니다.

  - 기술적 부담: 따라서 애플리케이션(WAS)은 세션 객체를 저장할 때마다 이를 직렬화(Serialization, 예: Java Object → JSON/Byte)하여 텍스트로 바꾸고, 읽을 때 다시 역직렬화(Deserialization)해야 합니다.
  - 결과: 이 과정에서 막대한 CPU 자원이 소모되며(Serialization Penalty), 세션 구조가 조금만 바뀌어도 애플리케이션 코드를 모두 수정해야 하는 등 데이터 무결성 관리가 순수하게 개발자의 몫으로 남게 됩니다.

- 일관성 모델의 한계: "속도를 위해 데이터 유실을 허용하는 구조" (Eventual Consistency)
  - 문제점: Redis 클러스터의 기본 복제 방식은 '비동기(Asynchronous)'입니다. Master 노드는 데이터를 받자마자 클라이언트에게 "저장 성공(OK)"을 응답하고, 그 후에 뒤늦게 Replica(백업) 노드로 데이터를 보냅니다.
  - 비즈니스 리스크: 만약 Master가 "저장 성공"을 응답한 직후, Replica에게 데이터를 보내기 전에 장애로 다운된다면? 그 데이터는 영원히 사라집니다.
  - 결과: 쇼핑몰 로그인 풀림, 결제 중 세션 만료 등 사용자 경험을 해치는 장애가 발생합니다. 이는 CAP 이론에서 일관성(Consistency)보다 가용성/분할허용(AP)을 우선시한 Redis의 설계 특성 때문이며, 세션 관리에는 부적합한 모델입니다.

- 트랜잭션의 한계: "쪼개진 데이터 간의 원자성 보장 불가" (Sharding Limitations)
  - 문제점: Redis는 데이터를 여러 노드에 쪼개서 저장(Sharding)합니다. Redis의 트랜잭션 기능(`MULTI/EXEC`)은 데이터가 같은 해시 슬롯(Hash Slot), 즉 같은 물리적 서버에 있을 때만 작동합니다.
  - 기술적 부담: 분산 환경에서는 A 데이터는 1번 서버, B 데이터는 2번 서버에 저장될 확률이 높습니다. 이 경우 두 데이터를 동시에 수정하는 트랜잭션(예: 포인트 차감 후 아이템 지급)을 걸 수 없습니다.
  - 결과: '돈은 나갔는데 아이템은 안 들어온' 상태와 같은 데이터 불일치를 막기 위해 애플리케이션 레벨에서 매우 복잡한 보상 트랜잭션 로직을 구현해야 합니다.

- 아키텍처의 한계: "운영자가 개입해야 하는 관리 복잡성"
  - 문제점: Redis는 Master-Slave 구조를 가집니다. Master가 죽으면 Sentinel 같은 감시 도구가 새로운 Master를 선출해야 하는데, 이 과정(Failover)에서 수 초에서 수

십 초간의 '서비스 중단(Downtime)'이 발생할 수 있습니다.

- 결과: 세션 데이터를 '없어져도 되는 캐시'로 본다면 타협할 수 있지만, '서비스의 핵심 상태'로 본다면 이러한 불안정성은 운영팀에 큰 부담을 줍니다.

7.2.2 IMDG: 세션을 '시스템의 1급 자산'으로 다루는 전용 플랫폼

IMDG(In-Memory Data Grid)는 단순 저장소가 아닙니다. "여러 서버의 메모리를 합쳐 하나의 거대한, 절대 꺼지지 않는 슈퍼 컴퓨터의 메모리처럼 쓰자"는 개념에서 출발했습니다. Hazelcast, Infinispan, GridGain 같은 솔루션들이 이에 해당하며, 세션 데이터를 가장 중요하고 안전하게(Mission Critical) 다뤄야 할 자산으로 취급합니다.

- P2P 데이터 그리드 아키텍처: "주인과 하인이 없는, 완전한 평등 구조"
  - 특징: IMDG는 Master-Slave 개념이 없습니다. 모든 노드가 동등한 권한을 가진 P2P(Peer-to-Peer) 메시 구조로 연결됩니다.
  - 탄력성(Elasticity): 서버 한 대가 추가되면, 전체 클러스터가 이를 인지하고 데이터를 자동으로 재분배(Rebalancing)합니다. 반대로 서버가 장애로 다운되면, 즉시 다른 노드들이 백업 데이터를 활성화하여 서비스를 이어갑니다.
  - 장점: 단일 장애 지점(SPOF)이 원천적으로 제거되어 있으며, 운영자의 개입 없이 시스템이 스스로 치유(Self-Healing)합니다.
- 객체 지향 데이터 모델 (Object-Aware): "Java 객체를 그대로 이해하는 저장소"
  - 특징: IMDG는 Java(또는 다른 언어)의 객체를 변환 없이 바이너리 형태 그대로 관리합니다. IMDG 자체가 세션 객체의 구조를 이해하고 있습니다.
  - 장점: 불필요한 직렬화/역직렬화 비용이 획기적으로 줄어듭니다. 더 나아가, `SQL like` 쿼리를 통해 "현재 장바구니에 10만 원 이상 담은 모든 세션 객체 검색"과 같은 복잡한 조회가 가능합니다. Redis에서는 불가능한 기능입니다.
- 강력한 일관성과 트랜잭션: "금융권 수준의 데이터 정합성" (Strong Consistency)

  - 특징: IMDG는 데이터 쓰기 요청이 오면, 동기식 복제(Synchronous Replication)를 수행합니다. 즉, 원본뿐만 아니라 백업 노드에도 저장이 완료되었다는 확인(Ack)을 받아야만 클라이언트에게 "성공"을 응답합니다.
  - 신뢰성: 속도는 미세하게 느려질 수 있지만, "저장되었다고 응답받은 데이터는 지구가 멸망하지 않는 한 유실되지 않음"을 보장합니다. 또한 JTA/XA 같은 분산 트랜잭션 표준을 지원하여, 서로 다른 노드에 있는 데이터 간에도 완벽한 원자성(ACID)을 보장합니다.

- 데이터 중심의 추가 기능: "데이터가 있는 곳으로 코드를 보낸다" (Data Locality)
  - 철학: 보통은 데이터를 처리하기 위해 DB에서 데이터를 꺼내와서 애플리케이션 서버(WAS)에서 계산합니다(Data to Code). 하지만 IMDG는 데이터가 저장된 메모리 노드로 계산 로직(Code)을 전송하여 실행합니다(Code to Data).
  - 기능 (Distributed Executor / Entry Processor): 세션 값을 갱신하기 위해 데이터를 네트워크로 가져올 필요 없이, IMDG 내부에서 즉시 값을 수정합니다. 이는 네트워크 트래픽을 획기적으로 줄이고, 락(Lock) 경합 시간을 최소화하여 고성능 처리를 가능하게 합니다.

[결론 요약] 세션 관리 관점에서의 핵심 비교

아래 표는 단순한 스펙 비교가 아니라, "사용자의 세션 객체를 다룰 때 실제로 어떤 일이 벌어지는가"에 초점을 맞춘 비교입니다.

비교 항목	Redis (캐시 중심 접근)	IMDG (세션 플랫폼 중심 접근)
세션 데이터 인식	"내용을 모르는 블랙박스" 세션 객체를 단순 문자열(Blob)로 취급하므로, 애플리케이션이 매번 해석해야 함.	"구조를 아는 객체 저장소" Java/Object 형태를 그대로 인식하여 별도의 해석 없이 즉시 접근 가능.
세션 수정 방식 (예: 장바구니 추가)	"꺼내서 고치고 다시 넣기" (Read → Deserialize → Modify → Serialize → Write) 데이터 이동이 많고 CPU 부하가 큼.	"그 자리에서 즉시 수정" (Entry Processor) 데이터 이동 없이 메모리 내부에서 로직만 수행하여 수정 (고효율).

장애 시 사용자 경험 (Failover)	“로그인이 풀릴 수 있음” 비동기 복제 특성상, 장애 직전의 세션 변경 사항(결제, 장바구니 등)이 유실될 위험 존재.	“사용자는 장애를 눈치채지 못함” 동기식 복제로 데이터가 완벽히 보호되므로, 서버가 죽어도 세션은 그대로 유지됨.
트랜잭션 안전성	“제한적 보장” 분산된 노드 간의 복합적인 세션 변경(예: 포인트 차감+아이템 지급) 시 정합성 보장이 어려움.	“완벽한 보장 (ACID)” 분산 트랜잭션(JTA/XA)을 지원하여, 어떤 상황에서도 데이터가 꼬이지 않음.
세션 관리 추천 영역	비로그인 세션, 단순 조회용 데이터 (데이터가 날아가도 서비스에 치명적이지 않은 영역)	로그인 세션, 장바구니, 금융 거래 (데이터 유실이 곧 금전적 손실이나 컴플레인으로 이어지는 영역)

결론: 세션 스토리지는 ’임시 저장소’가 아니라 ’비즈니스 심장’입니다.

많은 조직이 단순히 “익숙하고 빠르다”는 이유로 Redis를 세션 저장소로 선택합니다. 하지만 세션 관리의 본질은 속도가 아니라 ‘사용자의 현재 상태(State)를 얼마나 안전하게 지키느냐’에 있습니다.

Redis는 데이터를 빠르게 ‘캐싱(Caching)’하는 데 최적화된 도구인 반면, IMDG는 데이터를 안전하게 ‘관리(Managing)’하고 ‘처리(Processing)’하는 데 특화된 플랫폼입니다. 특히 사용자의 로그인 상태, 장바구니 정보, 결제 진행 단계와 같이 단 1초의 데이터 유실도 허용되지 않는 엔터프라이즈 환경에서 세션 저장소를 선택해야 한다면, 답은 명확합니다.

데이터의 무결성(Integrity)과 서비스의 연속성(Continuity)을 최우선 가치로 둔다면, IMDG는 선택이 아닌 필수적인 아키텍처입니다.

결론적으로, 세션 클러스터링 기술을 선택하는 것은 단순한 ‘속도’ 경쟁이 아닙니다. 비즈니스의 연속성을 지키기 위해 데이터의 무결성과 고가용성을 얼마나 보장할 수 있는가를 선택하는 전략적 결정입니다. 이 기준에서 IMDG는 엔터프라이즈 환경이 요구하는 안정성을 충족하는 유일하고 검증된 아키텍처입니다.

제8장. OPENMARU Cluster의 IMDG 기반 세션 클러스터링 전략: 무중단 서비스를 위한 아키텍처의 진화

오늘날의 클라우드 네이티브(Cloud-Native) 환경에서 '세션 관리'는 단순한 로그인 유지 기능을 넘어, 비즈니스의 연속성을 담보하는 핵심 아키텍처 기술로 격상되었습니다. 과거와 달리 서버가 수시로 생성되고 사라지는 동적인 환경에서는, 사용자가 어떤 서버에 접속하든 끊김 없는 경험을 제공하는 것이 필수적입니다.

전통적인 방식인 웹 애플리케이션 서버(WAS) 내장 세션 관리는 서버 간 데이터를 복제하는 과정에서 성능 저하와 장애 전파라는 명백한 한계를 드러냈습니다. 이에 대한 해결책으로, OPENMARU Cluster는 애플리케이션의 상태(State) 정보를 서버 외부에서 안전하게 관리하는 세션 외부화(Session Externalization) 전략을 제시합니다. 이는 시스템의 확장성과 안정성을 동시에 확보하는 현대 미들웨어의 필수 선택입니다.

8.1. OPENMARU Cluster 세션 아키텍처 개요: 상태(State)와 로직(Logic)의 완벽한 분리

OPENMARU Cluster가 채택한 '세션 외부화' 아키텍처의 핵심은 "비즈니스 로직을 처리하는 WAS"와 "사용자 정보를 담고 있는 세션"을 물리적으로 분리하는 것입니다.

기존에는 WAS가 요리(로직 처리)도 하고 재료 보관(세션 저장)도 하는 구조였다면, OPENMARU Cluster는 WAS가 요리에만 집중하고 재료는 IMDG(In-Memory Data Grid)라는 거대하고 빠른 외부 냉장고에 보관하는 방식입니다. 이를 통해 개별 WAS 인스턴스는 상태 비저장(Stateless) 구조를 갖추게 됩니다.

- 상태 비저장(Stateless)의 이점: 특정 WAS 서버에 장애가 발생해 다운되더라도, 사용자 세션 데이터는 외부 IMDG에 안전하게 보관되어 있습니다. 사용자의 다음 요청은 정상적인 다른 WAS로 연결되어 IMDG에서 데이터를 불러오므로, 사용자는 서비스 중단을 전혀 느끼지 못합니다. 이것이 바로 진정한 의미의 고가용성(High Availability)입니다.

8.1.1. WAS에서 분리된 독립 세션 클러스터 구조: 기존 방식의 한계 극복

전통적인 WAS 클러스터링 방식, 특히 All-to-All 복제 방식은 현대적인 트래픽 규모에서 구조적인 한계를 가집니다. 이를 이해하기 위해서는 기존 방식이 시스템에 미치는 악영향을 구체적으로 살펴볼 필요가 있습니다.

[문제점] 전통적 WAS 내장 세션(In-Memory Replication)의 한계

과거에는 클러스터 내의 모든 WAS가 서로의 세션 정보를 실시간으로 복사했습니다. 서버가 2~3대일 때는 문제가 없지만, 서버가 늘어날수록 다음과 같은 심각한 문제가 발생합니다.

1. 네트워크 및 CPU 과부하 (Performance Overhead)
    - 서버 A에 세션이 생성되면 B, C, D 서버에도 똑같이 복제해야 합니다. 트래픽이 몰리면 비즈니스 로직을 처리해야 할 CPU와 네트워크 대역폭이 세션 복사(동기화) 작업에 잠식당해, 정작 중요한 서비스 처리가 느려집니다.
2. 치명적인 GC(Garbage Collection) 지연
    - Java 기반의 WAS는 힙(Heap) 메모리를 사용합니다. 세션 데이터가 WAS 힙 메모리에 쌓이면, JVM은 청소(GC)를 위해 더 자주, 더 오래 멈춰야 합니다(Stop-the-world). 세션 객체가 많아질수록 서비스가 멈칫하는 '렉' 현상이 빈번해집니다.
3. 도미노 식 OOM(Out of Memory) 장애
    - 접속자가 폭증하여 한 서버의 메모리가 가득 차면(OOM), 해당 서버는 다운됩니다. 문제는 이 서버가 처리하던 부하가 남은 서버들로 넘어가고, 남은 서버들도 세션 복제 부하까지 겹쳐 연쇄적으로 다운되는 '장애 전파'가 발생할 수 있습니다.

[해결책] OPENMARU Cluster의 독립형 IMDG 아키텍처

OPENMARU Cluster는 세션 데이터를 WAS의 메모리가 아닌, 독립된 IMDG 클러스터에서 전담 관리합니다.

- 리소스 효율 극대화: WAS는 세션 저장 부담에서 해방되어 오직 비즈니스 로직 처리에만 100% 자원을 활용할 수 있습니다.
- 장애 격리: 특정 WAS가 OOM으로 죽더라도, 세션 데이터는 별도의 IMDG 서버에 살아있으므로 데이터 유실이 없습니다.
- GC 성능 향상: WAS 힙 메모리에 세션 객체가 쌓이지 않으므로, Full GC 발생 빈도가 획기적으로 줄어들어 응답 속도가 일정하게 유지됩니다.

8.1.2. 물리·가상화·MSA·Kubernetes를 포괄하는 통합 세션 모델

IT 인프라는 물리 서버(On-Premise)에서 가상 머신(VM)을 거쳐, 이제는 컨테이너 기반의 Kubernetes(K8s) 환경으로 진화했습니다. OPENMARU Cluster는 이러한 인프라의 변화와 관계없이 일관된 세션 처리 모델을 제공합니다.

동적 환경(Kubernetes)에서의 세션 관리 필수성

Kubernetes와 같은 컨테이너 환경은 '서버(Pod)는 언제든 죽고 새로 태어날 수 있다(Ephemeral)'는 철학을 가집니다.

- 기존 방식의 붕괴: Pod는 수시로 IP가 바뀌고(Dynamic IP), 오토스케일링(Auto-scaling)에 의해 개수가 늘었다 줄었다 합니다. 특정 서버에 사용자를 묶어두는 '스티키 세션(Sticky Session)' 방식이나 서버 간 IP 지정 복제 방식은 이런 환경에서 사실상 불가능합니다. Pod가 축소(Scale-in)되어 사라지면 그 안의 세션도 함께 증발하기 때문입니다.

OPENMARU Cluster의 전략적 우위

OPENMARU Cluster의 외부 세션 아키텍처는 이러한 동적 환경에 최적화되어 있습니다.

1. 완벽한 오토스케일링(HPA) 지원: 트래픽이 늘어 Pod가 10개에서 100개로 늘어나거나, 다시 10개로 줄어들어도 상관없습니다. 모든 Pod는 외부의 IMDG를 바라보고 있기 때문에, Pod의 생성/소멸과 무관하게 사용자의 로그인 상태는 유지됩니다.
2. 이기종 WAS 간 세션 공유 및 SSO 구현:

- 서로 다른 기술 스택(예: Tomcat과 JBoss, 또는 Java 버전이 다른 레거시와 신규 시스템) 간에도 표준화된 프로토콜을 통해 세션을 공유할 수 있습니다.
- 동일 WAS 내의 서로 다른 웹 애플리케이션(예: 쇼핑몰 /shop과 결제 /pay) 간에도 세션을 통합하여, 사용자가 한 번의 로그인으로 여러 서비스를 이용하는 SSO(Single Sign-On) 환경을 손쉽게 구축할 수 있습니다.

결론적으로, OPENMARU Cluster의 세션 전략은 단순한 기술적 선택을 넘어, 복잡한 하이브리드 클라우드와 MSA 환경에서 서비스의 안정성을 지탱하는 가장 확실한 아키텍처 기반을 제공합니다.

8.2. 운영 환경에서의 차별화 요소: “이론적 성능”보다 중요한 “실제 운영의 안정성”

시스템 도입 초기에는 '최대 성능'이 중요해 보이지만, 실제 서비스가 오픈된 이후의 운영 단계에서는 '안정성'과 '관리 편의성'이 시스템의 성패를 가릅니다. 세션 저장소로 널리 알려진 Redis와 엔터프라이즈급 In-Memory Data Grid(IMDG) 솔루션은 겉보기에는 모두 '메모리 기반 저장소'라는 공통점이 있지만, 내부를 들여다보면 설계 철학부터가 다릅니다. 이 구조적 차이는 장애 발생 시 데이터가 유실되느냐, 안전하게 지켜지느냐를 결정짓는 핵심 요인이 됩니다.

8.2.1. 장애 시 세션 유실 없는 운영 경험: 비동기(Async) vs 동기(Sync)의 차이

금융, 공공, 커머스 등 엔터프라이즈 시스템에서 데이터 정합성은 타협할 수 없는 가치입니다. 이를 이해하기 위해 분산 시스템의 CAP 이론(Consistency, Availability, Partition Tolerance) 관점에서 두 기술의 차이를 비교해 볼 필요가 있습니다.

구분	IMDG (OPENMARU Cluster)	Redis Cluster
핵심 철학	데이터 정합성 우선 (CP/CA 지향)	성능 및 속도 우선 (AP 지향)
아키텍처	P2P (Peer-to-Peer) 그리드 구조	Primary-Replica (Master-Slave) 구조

복제 방식	동기식 (Synchronous) 복제	비동기식 (Asynchronous) 복제
일관성 모델	강력한 일관성 (Strong Consistency)	최종 일관성 (Eventual Consistency)
장애 시나리오	노드 장애 시에도 데이터 유실 0% 보장	Master 장애 시 데이터 유실 가능성 존재

Redis의 '최종 일관성'과 데이터 유실 위험

Redis는 압도적인 쓰기 속도를 위해 비동기 복제(Asynchronous Replication) 방식을 기본으로 사용합니다.

1. 사용자가 로그인을 하면 Master 노드에 세션이 기록되고, 즉시 "성공" 응답을 보냅니다.
2. 그 직후(수 밀리초 뒤), Master는 Replica(백업) 노드로 데이터를 복제합니다.
3. 문제 상황: 만약 Master에 데이터가 기록되었지만, Replica로 복제되기 직전의 아주 짧은 찰나에 Master 서버가 전원 차단 등으로 다운된다면 어떻게 될까요?
4. Replica가 새로운 Master로 승격되지만, 복제받지 못한 최신 데이터는 영구적으로 소멸됩니다. 이를 '최종 일관성(Eventual Consistency)' 모델의 한계라고 부르며, 사용자는 영문도 모른 채 로그아웃되는 경험을 하게 됩니다.

OPENMARU Cluster의 '강력한 일관성'과 데이터 보장

반면, OPENMARU Cluster의 IMDG는 데이터베이스 트랜잭션의 ACID(원자성, 일관성, 고립성, 지속성) 원칙을 메모리상에서 구현합니다.

1. 동기식 복제(Synchronous Replication): 세션 생성 요청이 오면, 주 노드뿐만 아니라 백업 노드에도 데이터가 안전하게 저장되었는지 확인합니다.
2. 모든 복제가 완료된 후에만 클라이언트에게 "성공" 응답을 보냅니다.
3. 이 과정은 P2P(Peer-to-Peer) 아키텍처 위에서 수행되어, 특정 노드가 죽더라도 다른 노드에 이미 완벽한 복제본이 존재함을 보장합니다. 이를 강력한 일관성(Strong Consistency)이라 하며, 결제 중이나 중요한 업무 처리 중에 서버 장애가 발생해도 세션은 절대 사라지지 않습니다.

8.2.2. 트러블슈팅 가능성과 운영 가시성의 확보: 자동화 vs 수동 관리

시스템 규모가 커질수록(Scale-out), 관리의 복잡도는 기하급수적으로 증가합니다. 운영자 입장에서는 서버를 추가하거나 뺄 때 얼마나 손이 덜 가느냐가 곧 비용(TCO) 절감으로 이어집니다.

Redis의 확장성 한계: '해시 슬롯(Hash Slot)'의 악몽

Redis Cluster는 데이터를 분산 저장하기 위해 전체 공간을 16,384개의 해시 슬롯(Hash Slot)이라는 고정된 구역으로 나눕니다.

- 수동 리샤딩(Resharding)의 고통: 서버를 증설해야 할 때, 기존 서버들이 가지고 있던 슬롯들을 새로운 서버로 옮겨주는 작업을 운영자가 명령어를 통해 수동으로 계산하고 실행해야 합니다.
- 운영 리스크: 이 과정은 복잡하고 실수하기 쉽습니다. 리샤딩 중에 시스템 부하가 급증하거나, 설정 실수로 데이터 접근이 불가능해지는 등 트러블슈팅이 어려운 장애 상황을 유발할 수 있습니다. 즉, 확장은 가능하지만 그 과정이 매끄럽지 않습니다.

OPENMARU Cluster의 자동화된 확장성: 진정한 '탄력성(Elasticity)'

IMDG 기반의 OPENMARU Cluster는 노드 추가/삭제 시 데이터 파티션을 자동으로 재분배(Automatic Rebalancing)합니다.

- P2P 자가 치유: 새로운 서버가 클러스터에 합류하면, 기존 서버들이 알아서 데이터를 나누어 줍니다. 운영자는 단순히 서버를 켜기만 하면 됩니다.
- 선형적 확장성(Linear Scalability): 사람의 개입 없이 시스템이 스스로 균형을 맞추므로, 서버 대수에 비례하여 정직하게 성능이 향상됩니다. 이는 예측 가능한 시스템 운영을 가능하게 하여 운영자의 스트레스를 획기적으로 줄여줍니다.

운영의 눈, 상세 모니터링 (Observability)

안정적인 운영의 마지막 퍼즐은 '가시성'입니다. OPENMARU Cluster는 자바 표준 모니터링 인터페이스인 JMX(Java Management Extensions)를 통해 블랙박스 같았던 세션 내부 정보를 투명하게 공개합니다.

단순히 “서버가 살아있다/죽었다”를 넘어, 다음과 같은 비즈니스 관점의 핵심 지표를 실시간으로 제공합니다:

- Active session count: 현재 실제 활동 중인 동시 접속자 수
- Session id count: 생성된 총 세션 객체 수 (메모리 점유율 분석 용도)
- Created/Destroyed session count: 트래픽 유입 및 이탈 추이 분석
- Duplicated login count: 중복 로그인 시도 횟수 (보안 위협 또는 사용자 패턴 분석)

이러한 지표들은 장애 발생 전 이상 징후를 미리 포착하거나, 마케팅 이벤트 시 트래픽 추이를 분석하는 데 있어 대체 불가능한 데이터를 제공합니다. 결론적으로 OPENMARU Cluster는 단순한 저장소를 넘어, 예측 가능하고 제어 가능한 운영 환경을 제공합니다.

제9장. 결론: 세션은 단순 기능이 아닌 ‘고객 신뢰의 뼈대(Trust Infrastructure)’입니다

지금까지 우리는 웹 애플리케이션의 성능과 안정성을 결정짓는 여러 기술적 요소들을 깊이 있게 파헤쳐 왔습니다. 이제 이 긴 여정을 마무리하며, IT 의사결정권자분들께 드리는 최종적인 제언을 정리하고자 합니다.

세션 클러스터링(Session Clustering) 기술을 선택하는 것은 단순히 “어떤 소프트웨어를 쓸까?”라는 취향의 문제가 아닙니다. 이것은 “우리 서비스가 장애 상황에서도 고객의 로그인을 유지시킬 수 있는가?”, “결제 도중 서버가 죽어도 고객의 장바구니는 안전한가?”라는 비즈니스 연속성의 핵심 질문에 답하는 과정입니다.

본 장에서는 세션 관리를 개발의 부산물인 ‘기능(Feature)’이 아니라, 고객과의 약속을 지키는 ‘신뢰 인프라(Infrastructure)’로 정의하고, 왜 엔터프라이즈 환경에서는 기술의 용도 변경이 아닌 ‘적재적소’의 원칙이 필요한지 명확히 결론 내리겠습니다.

9.1 핵심 메시지 정리

9.1.1 Redis는 '고속도로(Cache)'로, 세션은 '안전 금고(Dedicated Architecture)'로

[분석적 도입] 기술의 본질을 꿰뚫어야 리스크가 보입니다

엔터프라이즈 아키텍처를 설계할 때 가장 경계해야 할 것은 '만능 도구의 환상'입니다. 현재 많은 조직에서 Redis를 캐시 용도뿐만 아니라 세션 저장소로도 범용적으로 사용하고 있습니다. 물론 Redis는 훌륭한 솔루션입니다. 하지만 데이터가 날아가도 다시 조회하면 되는 '캐시(Cache)'와, 유실되면 고객의 연결이 끊기는 '세션(Session)'은 요구하는 기술적 DNA가 완전히 다릅니다.

외부 자료인 분산 시스템의 CAP 정리(CAP Theorem)에 빗대어 설명하자면, Redis는 일관성(Consistency)보다는 가용성(Availability)과 분할 내성(Partition Tolerance)에 치우친 AP 시스템에 가깝게 운영되는 경우가 많습니다. 반면 세션은 데이터의 정합성(Consistency)이 무엇보다 중요합니다. 이 섹션에서는 왜 Redis가 세션 관리의 '구조적 한계'를 가지는지, 그리고 왜 IMDG가 그 대안이 되어야 하는지 논리적으로 증명합니다.

1. Redis의 본질적 한계: 속도를 위해 희생한 안정성

Redis는 태생적으로 Key-Value 저장소이자, 디스크가 아닌 메모리에서 극강의 속도를 내기 위한 캐시 시스템으로 설계되었습니다. 이 설계 철학은 세션 관리에 치명적인 두 가지 약점을 만듭니다.

① 비동기 복제(Asynchronous Replication)의 위험: "데이터가 유실될 수 있다"

Redis는 기본적으로 '비동기' 방식을 사용합니다.

- 쉬운 설명: 주(Primary) 서버에 데이터가 들어오면, 주 서버는 보조(Replica) 서버에 데이터를 보내기도 전에 클라이언트에게 "저장 완료!"라고 응답합니다. 그 직후 주 서버가 다운된다면? 보조 서버로 데이터가 넘어가기 전이므로 그 데이터는 영원히 사라집니다.
- 기술적 심화: Redis 문서에서도 언급하듯, Redis는 대기 시간을 줄이기 위해 '최종 일관성(Eventual Consistency)' 모델을 따릅니다. 세션 정보가 1초라도 유실되면 사용자는 갑자

기 로그아웃되거나 장바구니가 비워지는 경험을 하게 됩니다. 금융이나 쇼핑몰 같은 엔터프라이즈 환경에서 이는 용납할 수 없는 리스크입니다.

② 정적인 샤딩(Static Sharding)의 족쇄: “자동 확장이 어렵다”

Redis 클러스터는 데이터를 저장할 방(Slot)을 미리 16,384개로 나누고, 이를 각 노드에 할당하는 방식을 씁니다.

- 쉬운 설명: 이사 갈 때 짐을 옮기는 것과 비슷합니다. 서버(노드)를 늘리려면, 기존 서버에 있던 짐(슬롯)을 일일이 계산해서 새 서버로 옮겨주는 작업(Resharding)이 필요합니다.
- 기술적 심화: Kubernetes(쿠버네티스)와 같은 클라우드 환경은 트래픽에 따라 서버가 수시로 늘어나고 줄어듭니다(Auto-scaling). 하지만 Redis의 구조는 운영자가 직접 개입하여 슬롯을 재분배해야 하는 구조적 경직성을 가집니다. 자동화된 인프라 위에서 수동적인 Redis 운영은 시스템 확장의 병목(Bottleneck)이 됩니다.

2. 세션을 위한 전용 아키텍처(IMDG): 태생부터 다른 ‘신뢰성’

반면, Infinispan이나 Hazelcast와 같은 IMDG(In-Memory Data Grid)는 단순히 데이터를 저장하는 것을 넘어, 분산 환경에서의 ’데이터 무결성’과 ’자동화’를 목표로 탄생했습니다. 세션 관리에 최적화된 DNA를 가지고 있습니다.

① 강력한 일관성(Strong Consistency): “단 하나의 세션도 잃지 않는다”

IMDG는 데이터 복제 시 동기식(Synchronous) 옵션을 강력하게 지원합니다.

- 쉬운 설명: 주 서버가 데이터를 받을 때, 백업 서버에도 안전하게 저장된 것을 확인한 후에야 비로소 “저장 완료”라고 응답합니다. 조금 더 꼼꼼하게 확인하느라 아주 미세한 시간이 더 걸릴 수 있지만, 서버가 폭발해도 데이터는 백업 서버에 완벽하게 살아있습니다.
- 기술적 심화: 이는 분산 트랜잭션의 원칙을 준수하여, 장애 조치(Failover) 상황에서도 데이터 정합성을 100% 보장합니다. 사용자는 서버 장애를 전혀 눈치채지 못하고 서비스를 계속 이용할 수 있습니다.

② P2P 아키텍처와 자동화된 확장: “스스로 치유하고 확장한다”

Redis가 주/종(Primary/Replica) 관계가 명확하다면, IMDG는 모든 노드가 평등한 P2P(Peer-to-Peer) 구조를 가집니다.

- 쉬운 설명: 원탁 테이블에 앉은 팀원들과 같습니다. 새로운 팀원이 들어오면(Scale-out), 기존 팀원들이 알아서 일감을 공평하게 나눠줍니다. 누군가 아파서 빠지면(Scale-in), 남은 사람들이 즉시 그 몫을 나눠 가집니다. 관리자가 일일이 지시할 필요가 없습니다.
- 기술적 심화: IMDG는 클러스터 토폴로지 변화를 자동으로 감지하고 데이터 리밸런싱(Rebalancing)을 수행합니다. 이는 클라우드 네이티브 환경의 오토스케일링(Auto-scaling)과 완벽하게 부합하며, 운영 비용을 획기적으로 낮춰줍니다.

[최종 결론] 올바른 도구를 올바른 곳에

정리하자면 다음과 같습니다.

- Redis: 조회수는 높지만 잃어버려도 다시 DB에서 가져오면 되는 ‘휘발성 캐시 데이터’ 처리에 세계 최고의 성능을 발휘합니다.
- IMDG (세션 클러스터링): 고객의 로그인 정보, 결제 상태처럼 비즈니스의 신뢰와 직결되는 ‘상태 정보(Stateful Data)’는 강력한 일관성과 자동 확장을 보장하는 전용 아키텍처에 맡겨야 합니다.

이러한 아키텍처의 분리는 단순한 기술 선택을 넘어, “우리는 어떤 상황에서도 고객의 경험을 끊어뜨리지 않겠다”는 기업의 기술적 성숙도와 의지를 보여주는 지표입니다.

9.2 IT 의사결정자를 위한 심층 제언: 신뢰할 수 있는 아키텍처를 향한 여정

클라우드와 마이크로서비스 아키텍처(MSA)가 표준이 된 지금, IT 리더에게는 과거의 관성을 끊어내는 결단이 필요합니다. 특히 애플리케이션의 상태를 관리하는 ‘세션(Session)’에 대한 접근

방식은 시스템의 전체 안정성을 좌우하는 핵심 변수입니다.

9.2.1 'WAS 중심'의 낡은 관행에서 '세션 중심 아키텍처'로의 대전환

패러다임 전환의 필연성: 왜 'WAS'에서 벗어나야 하는가?

과거 온프레미스 환경의 모놀리식(Monolithic) 아키텍처에서는 웹 애플리케이션 서버(WAS)가 비즈니스 로직 처리뿐만 아니라 사용자의 로그인 정보(세션) 보관까지 도맡아 했습니다. 하지만 클라우드 환경에서 이러한 방식은 더 이상 유효하지 않습니다.

서버가 수시로 생성되고 사라지는(Auto-scaling) 클라우드 환경에서, 세션을 WAS 내부에 가두는 것은 시스템의 유연성을 해치는 가장 큰 원인입니다. 이제 세션은 WAS의 부속품이 아니라, 모든 서비스가 공유하는 독립적이고 중립적인 '신뢰 계층(Trust Layer)'으로 격상되어야 합니다.

심층 분석: 'WAS 중심 사고(In-Memory Session)'의 치명적 한계

과거의 방식, 즉 WAS끼리 서로 세션 정보를 복제(Session Clustering)하는 방식은 현대적인 트래픽 규모에서 다음과 같은 구조적 모순을 갖습니다.

1. 성능 저하의 악순환 (The Network Storm):
    - 문제점: Tomcat 등 전통적인 WAS의 세션 클러스터링(예: DeltaManager)은 기본적으로 All-to-All 복제 방식을 사용합니다.
    - 상세: 서버가 2대일 때는 서로 1번씩만 통신하면 되지만, 서버가 10대가 되면 세션 하나가 변경될 때마다 나머지 9대 모두에게 데이터를 전송해야 합니다. 서버 수가 늘어날수록 복제 트래픽은 기하급수적으로 폭증($N(N-1)$)하며, 결국 네트워크 대역폭이 비즈니스 데이터가 아닌 세션 복제 데이터로 마비되는 현상이 발생합니다.
2. WAS 안정성 위협 (JVM Heap Contention):
    - 문제점: 세션 객체가 WAS의 JVM 힙(Heap) 메모리에 상주합니다.
    - 상세: 사용자가 몰리면 세션 객체가 메모리를 점유하고, 이는 비즈니스 로직 처리를 위한 메모리 공간을 침범합니다. 이로 인해 잦은 GC(Garbage Collection)가 발생하여 서버가 멈칫거리거나(Stop-the-world), 심한 경우 메모리 부족(OutOfMemoryError)으

로 WAS가 셧다운되는 장애를 유발합니다. 즉, 로그인 처리를 하느라 물건을 못 파는 상황이 벌어집니다.

3. 확장성의 족쇄 (Scalability Limit):
    - 문제점: 트래픽 대응을 위해 WAS를 늘리면(Scale-out), 오히려 세션 복제 부하 때문에 전체 성능이 떨어지는 역설이 발생합니다. 이는 진정한 의미의 클라우드 확장을 불가능하게 만듭니다.

'세션 중심 아키텍처(External Session Store)'의 전략적 가치

세션을 WAS에서 끄집어내어 전문적인 외부 세션 클러스터로 옮기는 것은 다음과 같은 확실한 이점을 제공합니다.

- 진정한 무상태(Stateless) 아키텍처 구현:
    - WAS는 이제 어떤 상태 정보도 저장하지 않습니다. 따라서 트래픽이 폭주할 때 WAS 인스턴스를 10개든 100개든 부담 없이 즉시 늘릴 수 있습니다. (The "Cattle, not Pets" 개념 실현)
- 완벽한 장애 격리 (Fault Isolation):
    - 특정 WAS가 다운되더라도 사용자의 로그인 정보는 외부 세션 저장소에 안전하게 보관되어 있으므로, 사용자는 아무런 끊김 없이 다른 WAS를 통해 서비스를 계속 이용할 수 있습니다. 비즈니스 로직의 오류가 데이터의 유실로 이어지지 않는 견고한 방화벽이 생기는 셈입니다.

9.2.2 "쉽게 붙이는 세션"의 유혹을 넘어 "끝까지 책임지는 세션"으로

의사결정 프레임워크: 개발 편의성 vs 운영 안정성

기술 도입 시 흔히 범하는 실수는 '개발자가 지금 당장 쓰기 편한가'에 매몰되는 것입니다. 하지만 경영진은 '장애가 발생했을 때 이 기술이 우리를 구해줄 수 있는가'를 물어야 합니다. 이제는 "쉽

게 붙이는 오픈소스"의 유혹을 넘어, "비즈니스를 끝까지 책임지는 엔터프라이즈급 기술"을 선택해야 할 시점입니다.

리스크 분석: '쉽게 붙이는 세션' (예: Redis)의 숨겨진 비용

Redis는 훌륭한 캐시(Cache) 솔루션이며 개발자들에게 친숙합니다. 하지만 이를 '세션 저장소'라는 핵심 시스템(System of Record)으로 사용할 때는 신중해야 합니다.

1. 데이터 정합성의 위기 (Eventual Consistency Risk):
    - 기술적 배경: Redis의 복제 방식은 기본적으로 비동기(Asynchronous)입니다. 마스터 노드에 데이터를 쓰고, 복제 노드로 전송하기 직전에 마스터가 다운되면 해당 세션 데이터는 영구히 유실됩니다.
    - 비즈니스 영향: 쇼핑몰 결제 중이던 고객이 갑자기 로그아웃되거나, 장바구니가 비워지는 경험을 하게 됩니다. 이는 단순한 불편을 넘어 고객 신뢰 하락으로 직결됩니다.
2. 운영의 복잡성과 기회비용 (Operational Overhead):
    - 기술적 배경: 클라우드 환경에서 트래픽 급증 시 Redis 클러스터를 확장(Scale-out)하려면, 데이터 샤딩(Sharding)을 수동으로 재조정하거나 복잡한 운영 절차를 거쳐야 합니다. 이 과정에서 'Blocking(일시적 멈춤)' 현상이 발생할 수 있습니다.
    - 비즈니스 영향: 블랙 프라이데이와 같은 대형 이벤트 시, 오토스케일링이 매끄럽게 이루어지지 않아 서비스 지연이 발생하고, 이는 곧바로 매출 손실이라는 기회비용으로 이어집니다.

가치 제안: '끝까지 책임지는 세션' (IMDG 기반 솔루션)

IMDG(In-Memory Data Grid) 기술을 기반으로 한 엔터프라이즈 세션 솔루션은 태생부터 '데이터 보호'를 최우선으로 설계되었습니다.

1. 데이터 무결성 보장 (Strong Consistency):
    - 데이터가 생성되는 즉시 백업 노드까지 동기화가 완료된 후 응답을 줍니다. 서버가 벼락을 맞아도 데이터는 절대 유실되지 않는 강력한 정합성을 보장합니다. 금융 거래나 중요 예약 시스템에서 필수적인 요소입니다.

2. 지능형 자동화 (Automated Rebalancing):
    - 노드가 추가되거나 제거될 때, 시스템이 알아서 데이터를 재분배(Rebalancing)하고 클러스터 상태를 최적화합니다. 운영자의 개입 없이도 시스템이 스스로 치유하고 확장하므로, 예측 불가능한 장애 상황에서도 비즈니스 연속성을 유지합니다.

최종 행동 촉구 (Call to Action)

화려한 UI나 기능은 고객을 끌어들일 수 있지만, 고객을 머무르게 하는 것은 '신뢰(Trust)'입니다.

IT 의사결정자 여러분, 초기 구축이 조금 더 쉽다는 이유로, 혹은 무료라는 이유로 비즈니스의 핵심인 '세션'을 불안정한 기반 위에 올리는 도박을 멈추십시오. 장애는 예고 없이 찾아옵니다. 그때 시스템이 멈추지 않고, 고객의 데이터가 단 하나도 유실되지 않도록 '끝까지 책임지는 아키텍처'*에 투자하십시오. 이것은 단순한 IT 인프라 구매가 아니라, 위기 상황에서도 기업의 신뢰를 지켜낼 '디지털 회복탄력성(Digital Resilience)'을 확보하는 리더의 전략적 결단입니다.

References & Links

- Redis Documentation – https://redis.io/docs/
- Martin Kleppmann, Designing Data-Intensive Applications
- Redis Persistence – https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/
- CNCF Cloud Native Principles – https://www.cncf.io/
- Redis replication | Docs – https://redis.io/docs/latest/operate/oss_and_stack/management/replication/
- Redis Cluster specification | Docs – https://redis.io/docs/latest/operate/oss_and_stack/reference/cluster-spec/
- Scale with Redis Cluster | Docs – https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/
- High availability with Redis Sentinel | Docs – https://redis.io/docs/latest/operate/oss_and_stack/management/sentinel/

- Redis persistence | Docs – https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/
- WAIT | Docs – https://redis.io/docs/latest/commands/wait/
- HttpSession Integration :: Spring Session – https://docs.spring.io/spring-session/reference/http-session.html
- Spring Session – Spring Boot :: Spring Session (Redis 가이드) – https://docs.spring.io/spring-session/reference/guides/boot-redis.html
- Spring Session Hazelcast | Hazelcast Documentation (튜토리얼) – https://docs.hazelcast.com/tutorials/spring-session-hazelcast
- Spring Session and Spring Security guide | Hazelcast Documentation – https://docs.hazelcast.com/hazelcast/5.5/spring/spring-session-guide
- Split-Brain Protection | Hazelcast Documentation – https://docs.hazelcast.com/hazelcast/5.6/network-partitioning/split-brain-protection
- Fault Tolerance | Hazelcast Documentation (Split-Brain 관련) – https://docs.hazelcast.com/hazelcast/5.1/fault-tolerance/fault-tolerance
- Infinispan Spring Boot Starter (Using Spring Session) – https://infinispan.org/docs/stable/titles/spring_boot/starter.html
- Oracle Coherence Session Management – https://www.oracle.com/middleware/technologies/coherence/session-management.html
- Spring for VMware GemFire is Now Available – Tanzu Blog – https://blogs.vmware.com/tanzu/spring-for-vmware-gemfire-is-now-available/
- Spring (GemFire QuickStart) – https://gemfire.dev/quickstart/spring/
- Apache Tomcat 9 – Clustering/Session Replication How-To – https://tomcat.apache.org/tomcat-9.0-doc/cluster-howto.html
- Apache Tomcat 7 – Clustering/Session Replication HOW-TO – https://tomcat.apache.org/tomcat-7.0-doc/cluster-howto.html
- DeltaManager (Tomcat 8.0.53 API) – https://tomcat.apache.org/tomcat-8.0-doc/api/org/apache/catalina/ha/session/DeltaManager.html
- DeltaManager (Tomcat 9.0.112 API) – https://tomcat.apache.org/tomcat-9.0

-doc/api/org/apache/catalina/ha/session/DeltaManager.html

- DeltaManager (Tomcat 9.0.113 API) – https://tomcat.apache.org/tomcat-9.0-doc/api/org/apache/catalina/ha/session/DeltaManager.html
- DeltaManager (Tomcat 10.0.27 API) – https://tomcat.apache.org/tomcat-10.0-doc/api/org/apache/catalina/ha/session/DeltaManager.html
- DeltaManager (Tomcat 10.1.42 API) – https://tomcat.apache.org/tomcat-10.1-doc/api/org/apache/catalina/ha/session/DeltaManager.html
- DeltaManager (Tomcat 11.0.13 API) – https://tomcat.apache.org/tomcat-11.0-doc/api/org/apache/catalina/ha/session/DeltaManager.html
- Virtual IPs and Service Proxies | Kubernetes (Session affinity: ClientIP) – https://v1-32.docs.kubernetes.io/docs/reference/networking/virtual-ips/
- Sticky sessions (cookie) – Ingress-NGINX Controller – https://kubernetes.github.io/ingress-nginx/examples/affinity/cookie/
- Annotations – Ingress-NGINX Controller (Session Affinity) – https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/annotations/

Contact Us

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

CNF Blog

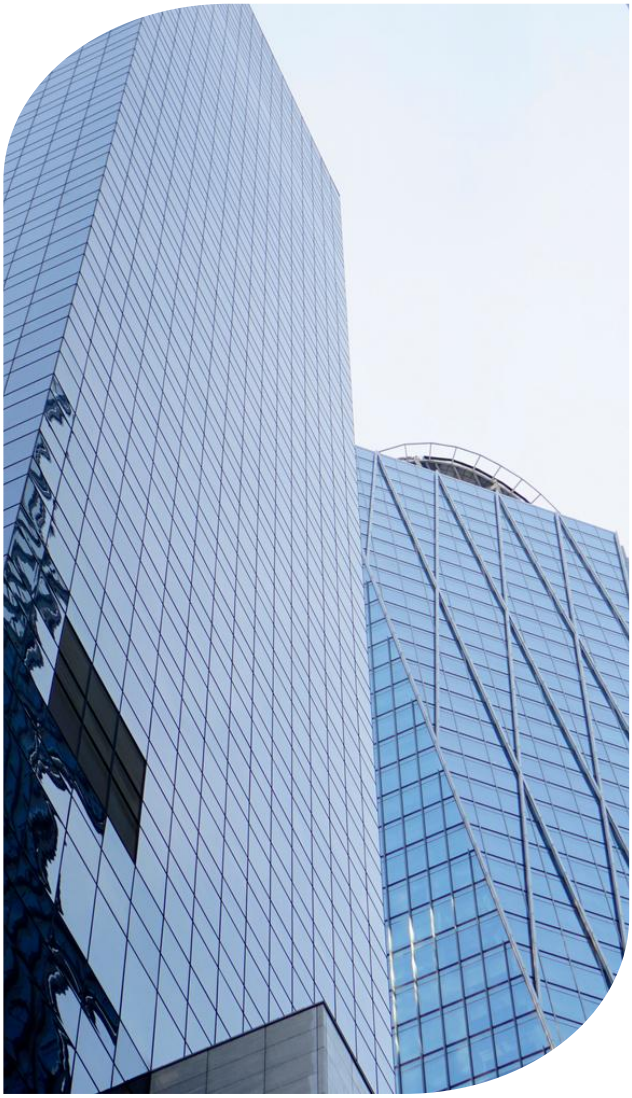
다양한 콘텐츠와 전문 지식을 통해
더 나은 경험을 제공합니다.

CNF eBook

이제 나도 클라우드 네이티브 전문가
쿠버네티스 구축부터 운영 완전 정복

CNF Resource

Community Solution의 최신 정보와
유용한 자료를 만나보세요.



씨엔에프 | CNF

전화 : (02) 469 - 5426
팩스 : (02) 469 - 7247
메일 : hello@cncf.co.kr