第四章：Architecture
架構風格與 DDD 的協奏曲

2025 年 5 月 17 日

目錄

1. 開場
2. SaaSOvation 演進地圖
3. Layered Architecture
4. Hexagonal Architecture
5. SOA Architecture
6. REST Architecture
7. CQRS Pattern
8. Event-Driven Styles
9. Data Fabric
10. 結語

開場

開場引言

"Architecture should speak of its time and place, but yearn for timelessness."
- Frank Gehry

核心觀點

- DDD 是一套以 **Bounded Context** 劃界的思維框架，而非單一實作架構。
- 架構 = **品質屬性** + **功能需求**的平衡藝術。
- 架構應服務於 Domain 模型；勿讓技術層凌駕業務語言。
- 追求可演進，抵禦未知需求，才能保持系統韌性。

SaaSOvation 演進地圖

SaaSOvation 架構演進

階段	架構	驅動	收穫	風險
Startup	Monolith (Layered)	MVP 速度	快速迭代	技術債
Scale-up	Hexagonal + CQRS	可測試性	關注點分離	邊界模糊
Enterprise	多 Context + SOA	團隊協作	去耦	協調成本

啟示

沒有一步到位的架構：**需求** × **風險** × **演進式重構 才是真正的長青之道**。

Layered Architecture

Layered 四層模型

- UI → Application → Domain → Infrastructure
- 優點：結構清晰、好上手
- 缺陷：若缺乏 DDD 思維，易淪為"Controller + Service + DAO" 技術債
- 解方：**DIP + Ports/Adapters**、在層內劃分 Bounded Context

Layered Java (DIP)

```java
// Domain Layer (Port)
public interface OrderRepository {
    Order findById(String id);
    void save(Order order);
}

public class Order {
    private final String id;
    private OrderStatus status = OrderStatus.CREATED;
    public Order(String id) { this.id = id; }
    public void complete() { status = OrderStatus.COMPLETED; }
    public OrderStatus status() { return status; }
}

// Application Layer
public class OrderService {
    private final OrderRepository repo; // 依賴抽象
    public OrderService(OrderRepository repo) { this.repo = repo; }
    public void completeOrder(String id) {
        Order o = repo.findById(id);
        o.complete(); repo.save(o);
    }
}
```

Hexagonal Architecture

Hexagonal 重點

1. 所有依賴指向 Domain；外界透過 Port 呼叫
2. Adapter 隔離協定／格式，方便測試替身
3. 高可維護、技術棧可替換

Hexagonal Java 範例

```java
// Port
public interface PaymentPort {
    boolean pay(double amount);
}

// Domain
public class Checkout {
    private final PaymentPort port;
    public Checkout(PaymentPort port) { this.port = port; }
    public Receipt process(double amt) {
        if (port.pay(amt)) return new Receipt("OK");
        throw new PaymentFailed();
    }
}

// Adapter
public class PaypalAdapter implements PaymentPort {
    private final PaypalApi api;
    public PaypalAdapter(PaypalApi api) { this.api = api; }
    public boolean pay(double amt) { return api.execute(amt); }
}
```

SOA Architecture

SOA 架構精要

- 每個 Bounded Context → 獨立服務 (Service)
- 通訊：REST / gRPC (同步)，Event Bus (非同步)
- 服務治理：Registry、Contract-First、Versioning、Policy Enforcement
- 韌性：Circuit Breaker、Retry、Bulkhead、Timeout

REST Architecture

Contract-First Design 範例

```java
@Path("/products")
public interface ProductService {
    @GET @Path("/{id}")
    ProductDTO get(@PathParam("id") String id);
}
```

REST 架構風格

- 資源導向，使用 HTTP 動詞 (GET/POST/PUT/DELETE)
- 無狀態、統一介面、可快取、分層系統
- HATEOAS：在回應中提供 Link 進行導覽
- Aggregate Resource，API 不可破壞 Context 邊界

REST Java 範例

```java
@RestController
@RequestMapping("/orders")
class OrderController {
    private final OrderService svc;
    OrderController(OrderService svc) { this.svc = svc; }

    @GetMapping("/{id}")
    RepresentationModel<OrderDTO> get(@PathVariable String id) {
        Order o = svc.findById(id);
        OrderDTO dto = new OrderDTO(o.status().name());
        dto.add(linkTo(methodOn(OrderController.class).get(id)).withSelfRel());
        return dto;
    }
}
```

CQRS Pattern

CQRS 核心概念

- Command / Query 分離；寫入模型　讀取模型
- 為寫入一致性與讀取效能分別優化
- 常與 Event Sourcing 搭配；需要處理最終一致性

CQRS Java - Command Side

```java
public record CreateUserCommand(String id, String name) {}

public class UserCommandHandler {
    private final UserRepository repo;
    public UserCommandHandler(UserRepository repo) { this.repo = repo; }
    public void handle(CreateUserCommand cmd) {
        repo.save(new User(cmd.id(), cmd.name()));
    }
}
```

CQRS Java - Query Side

```java
public class UserProjection {
    @EventListener
    public void on(UserCreated e) {
        // 寫入投影 DB
    }
}

public class UserQueryService {
    private final UserReadRepo repo;
    public UserQueryService(UserReadRepo r) { this.repo = r; }
    public UserDTO fetch(String id) { return repo.find(id); }
}
```

Event-Driven Styles

事件驅動三風格

1. Pipes & Filters —流式轉換
2. Sagas —分散式交易協調 $+$ 補償
3. Event Sourcing —狀態 $=$ 事件折疊

Saga 編排範例

```java
class ShippingSaga {
    @SagaEventHandler
    void on(OrderCreated e) {
        send(new ReserveInventory(e.id()));
    }
    @SagaEventHandler
    void on(InventoryReserved e) {
        send(new ArrangeShipment(e.orderId()));
    }
    @SagaEventHandler
    void on(ShipmentArranged e) {
        send(new MarkOrderShipped(e.orderId()));
        end();
    }
}
```

Event Sourcing 範例

```java
public interface DomainEvent { Instant occurredAt(); }

public record OrderCreated(String id, Instant occurredAt) implements DomainEvent {}

public class OrderAggregate {
    private String id;
    private OrderStatus status;

    public static OrderAggregate reconstitute(List<DomainEvent> history) {
        OrderAggregate agg = new OrderAggregate();
        history.forEach(agg::apply);
        return agg;
    }
    private void apply(DomainEvent e) {
        if (e instanceof OrderCreated oc) {
            this.id = oc.id();
            this.status = OrderStatus.CREATED;
        }
    }
}
```

Data Fabric

Data Fabric 架構要點

- 統一資料平面：整合 OLTP / OLAP / Streams / Cache
- Smart Cache、Federated Query、Consistency Policy、Observability
- 與 DDD 聚合分片結合，確保資料局部性與效能

Hazelcast 快取範例

```java
Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
IMap<String, OrderSummary> orders = hz.getMap("orders");

OrderSummary summary = new OrderSummary("ID-123", 1023, Instant.now());
orders.set(summary.id(), summary, 30, TimeUnit.MINUTES);

Collection<OrderSummary> highValue = orders.values(
   Predicates.greaterThan("total", 1000));
```

Apache Ignite 分散運算範例

```java
Ignition.start();
Ignite ignite = Ignition.ignite();
IgniteCache<Integer, Tick> cache = ignite.getOrCreateCache("ticks");

IgniteCallable<Double> task = () -> {
    List<List<?>> rows = cache.query(
      new SqlFieldsQuery("SELECT price FROM Tick WHERE pid = ?")
        .setArgs(portfolioId)).getAll();
    List<Double> prices = rows.stream()
        .map(r -> (Double) r.get(0))
        .sorted()
        .collect(Collectors.toList());
    return prices.get((int)(prices.size() * 0.05)); // Value-at-Risk
};

Double var = ignite.compute().call(task);
```

Data Fabric 風險與治理建議

- **記憶體壓力**：熱資料量估錯 → OOM，建議 TTL + LRU Eviction
- **Schema 演進**：需有 Registry + 相容性驗證
- **Split-Brain**：多區部署需啟用 CP 模式或資料同步機制
- **安全**：敏感資料須加密（靜態/傳輸/使用中）與審計紀錄

結語

結語：架構即演進

- 沒有銀彈架構，DDD 幫助你因應變化、協作清晰
- 每種架構風格皆服務於 Domain 模型的演進
- 避免技術導向的「錯配式設計」；堅持語言一致性與 Context 純度
- 架構設計的重點不是「選哪一種」，而是「如何隨著需求演進」

謝謝收看！

Slides by