架构分层模型适配

— 有效防止架构腐化实践

吴雪峰@201811

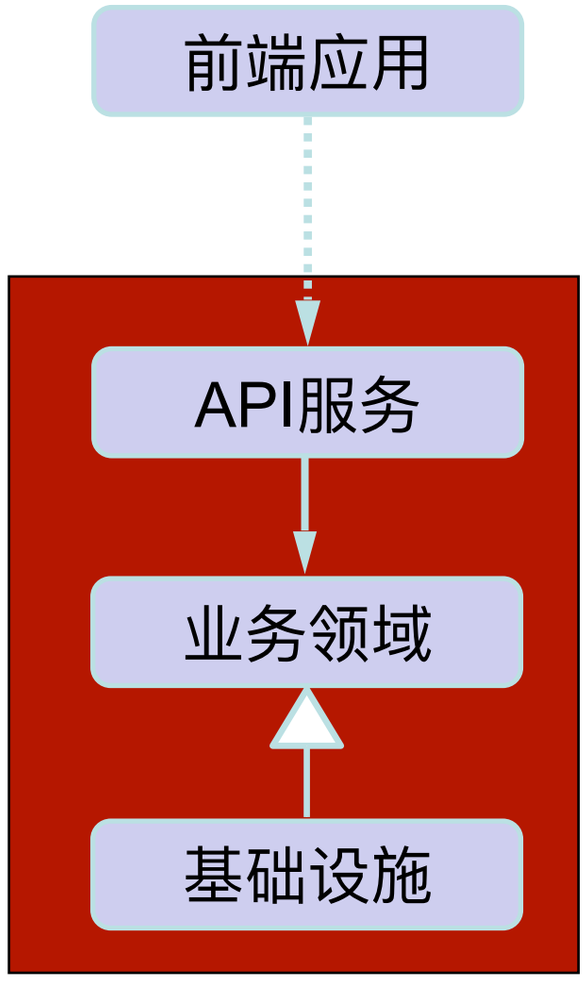
DDD CHINA

CONTENTS

DDD分层参考架构

DDD分层参考架构

前端应用



给用户提供界面，关注用户交互和体验

为前端应用提供API服务，关注事务和分布式等技术性问题

后台服务

领域模型和领域逻辑，关注业务概念。

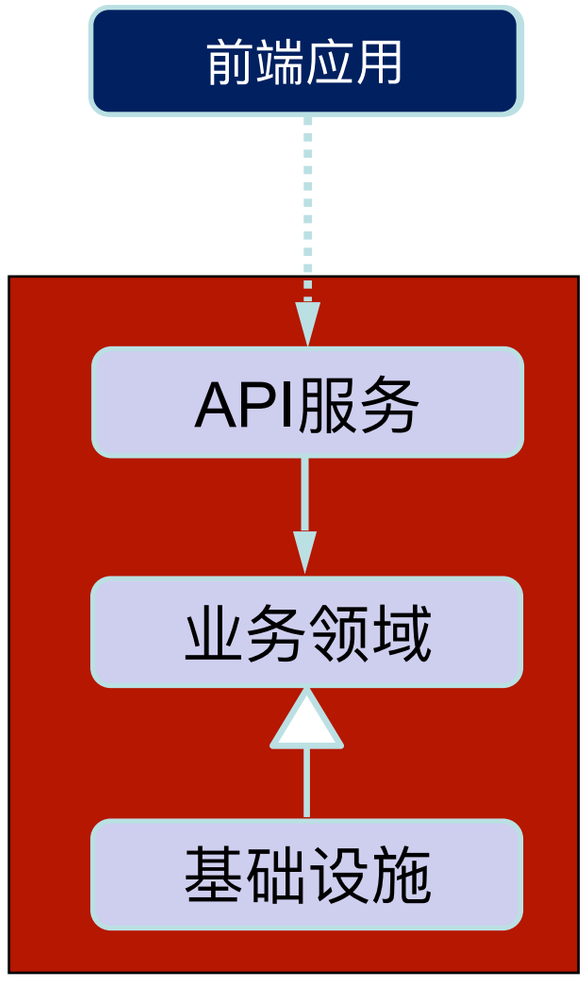
访问外界系统（调用外界系统）的技术相关实现。

分层依据：干系人和技术点

DDD分层参考架构 - 前端应用

前端应用

UX关注的层



DDD重点关注后台业务服务，不解决前端交互问题

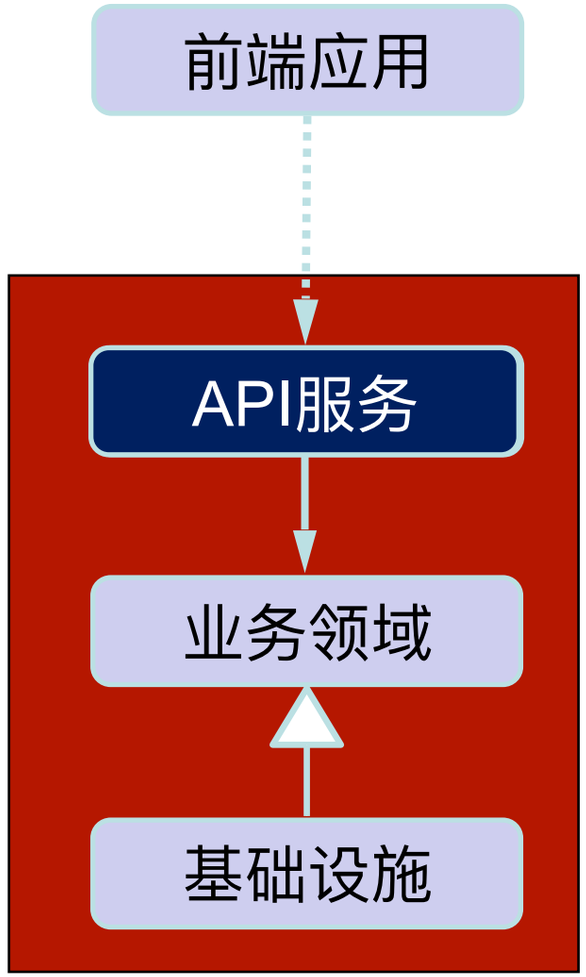
干系人: 终端用户

诉求: 良好的用户体验

技术点: 人机交互设计和实现

DDD分层参考架构

技术人员关注的层



为前端和第三方应用提供API服务，关注服务编排，事务和分布式等

干系人: 应用开发人员

诉求: 灵活易使用的API

技术点: 性能，事务，分布式，安全等非功能性需求

工作内容:

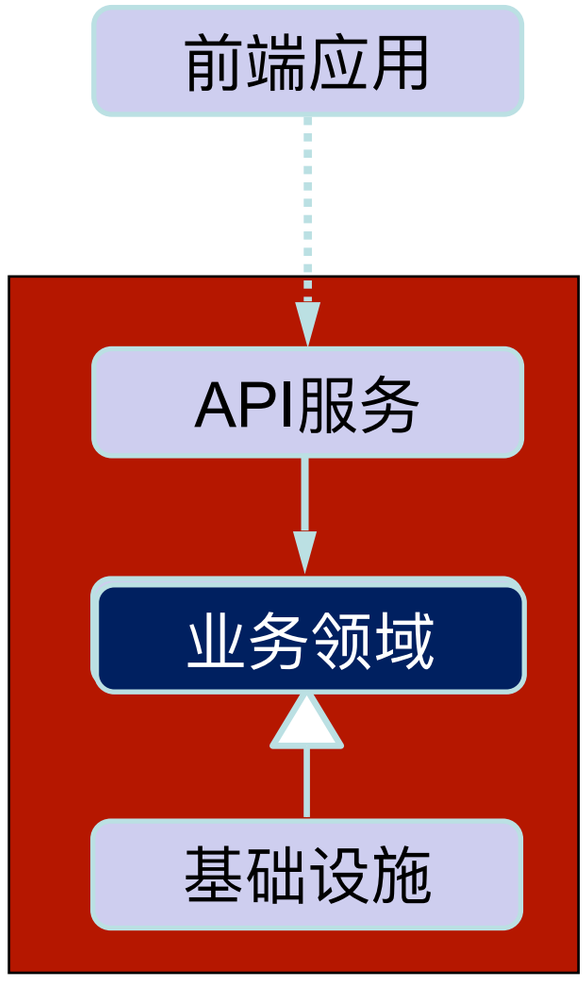
- 接收外部请求并响应：如HTTP请求, 消息处理
- 事务管理
- 认证
- 缓存
- 日志
- 异常处理
- 配置
- Session

模型:

- View Object
- Resource Model

腐化案例: 大量业务逻辑堆积

DDD分层参考架构



业务人员关注的层

领域模型和领域逻辑，关注业务概念。

干系人: 业务领域专家，业务领导

诉求: 表现业务概念和实现业务价值

要点: 业务建模和复杂性管理

工作内容:

- 建立业务模型，并体现在代码上
- 管理模型复杂度，适度拆分模块
- 实现业务逻辑

模型:

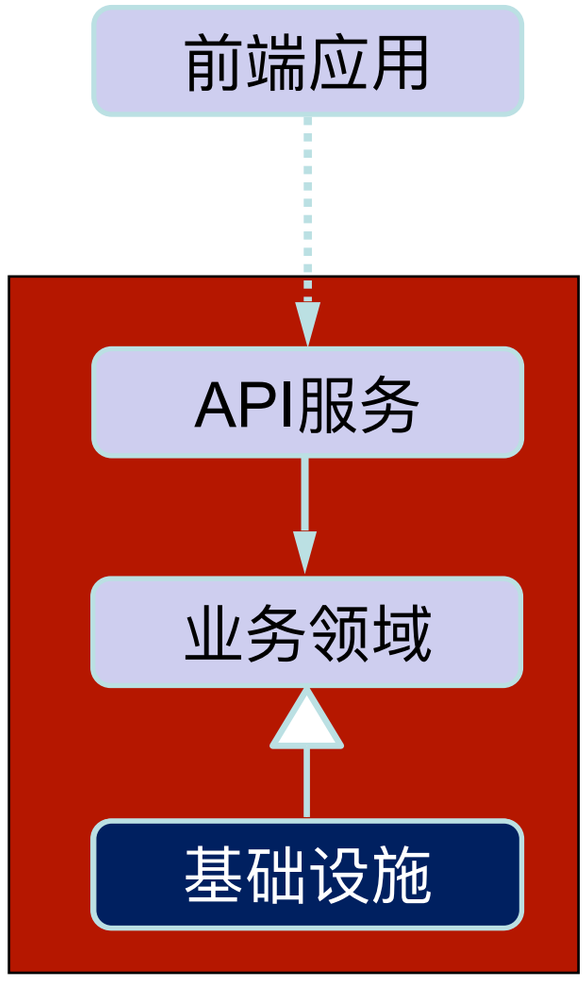
- 应用服务 — 跨Bond Context　DTO
- 领域服务 — 跨聚合
- 聚合 实体 仓库 事件

腐化案例:

亏空

大量技术术语业务人员完全看不懂

DDD分层参考架构



技术人员关注的层

访问外界系统（调用外界系统）的技术相关实现。

干系人: 外界系统

诉求: 稳定调用外部系统

技术点: 使用和适配外部系统模型，隔离和快速诊断错误

工作内容:

- 学习和使用外部系统，如数据库，邮件系统
- 适配外部系统模型，如SQL映射到模型对象

模型内容:

- PO
- 第三方Protobuffer
- 第三方SDK

腐化案例:

业务逻辑和外部调用逻辑混合

如一个方法里即处理业务逻辑又调用SQL

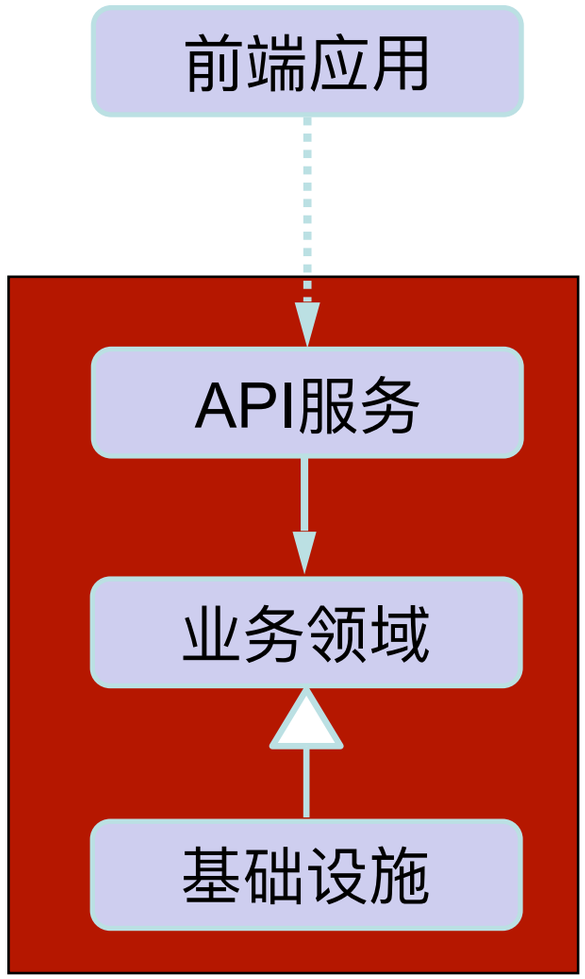
DDD分层参考架构 — Java技术视角

UX关注的层

技术人员关注的层

业务人员关注的层

技术人员关注的层



Web,Spring Boot, Kafka, Redis,
JTA,两阶段提交,SSO， 服务注册

尽量少依赖技术框架，
让业务人员也看得懂代码

ORM, SQL DB, NoSql,服务发现

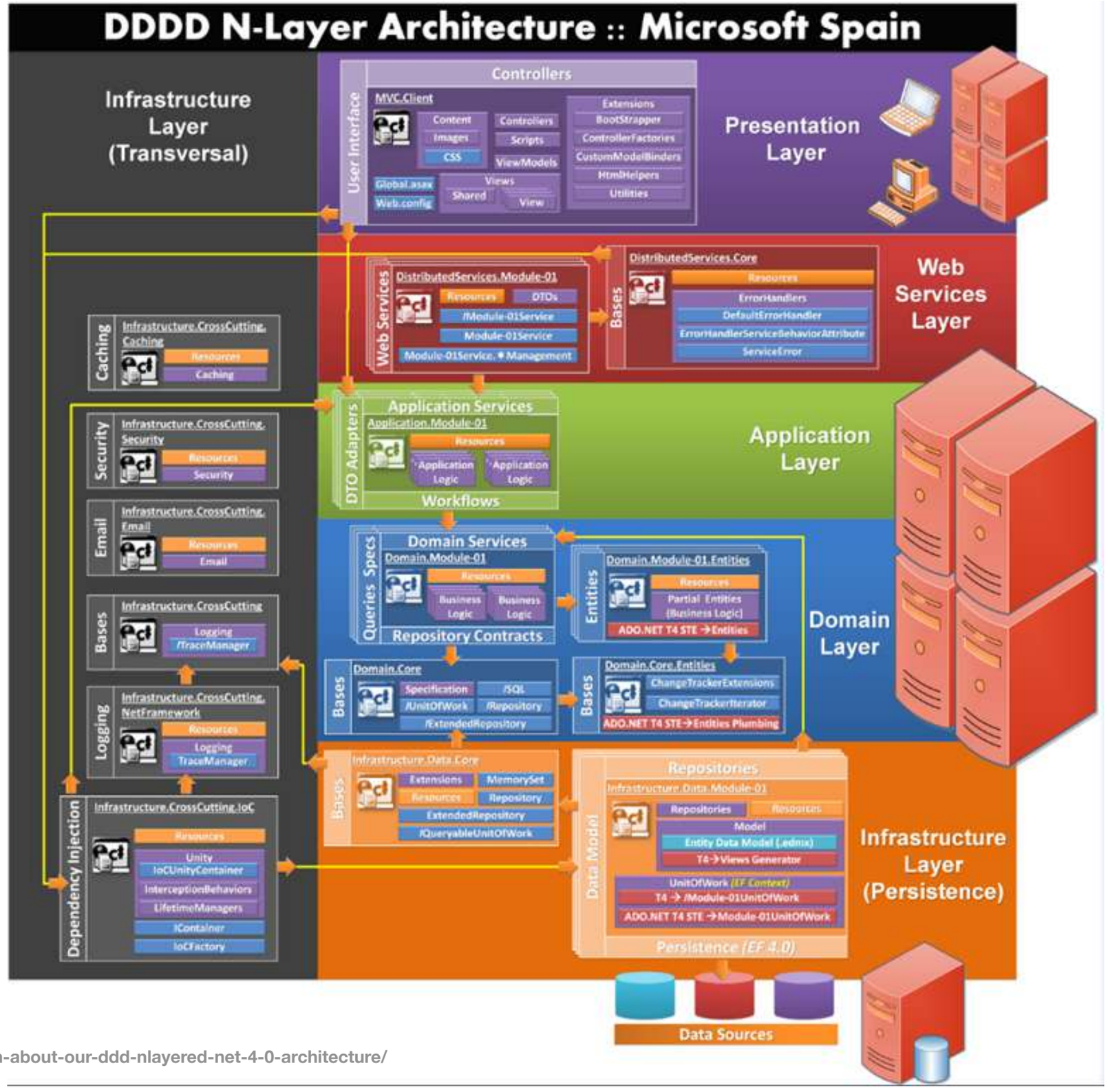
参考架构对比

前后端分离，完全剥离曾现层

Controller概念下移到Application层

为了简便，
合并Web Service和Application层

微软 DDD NLayered .NET 4.0 Architecture



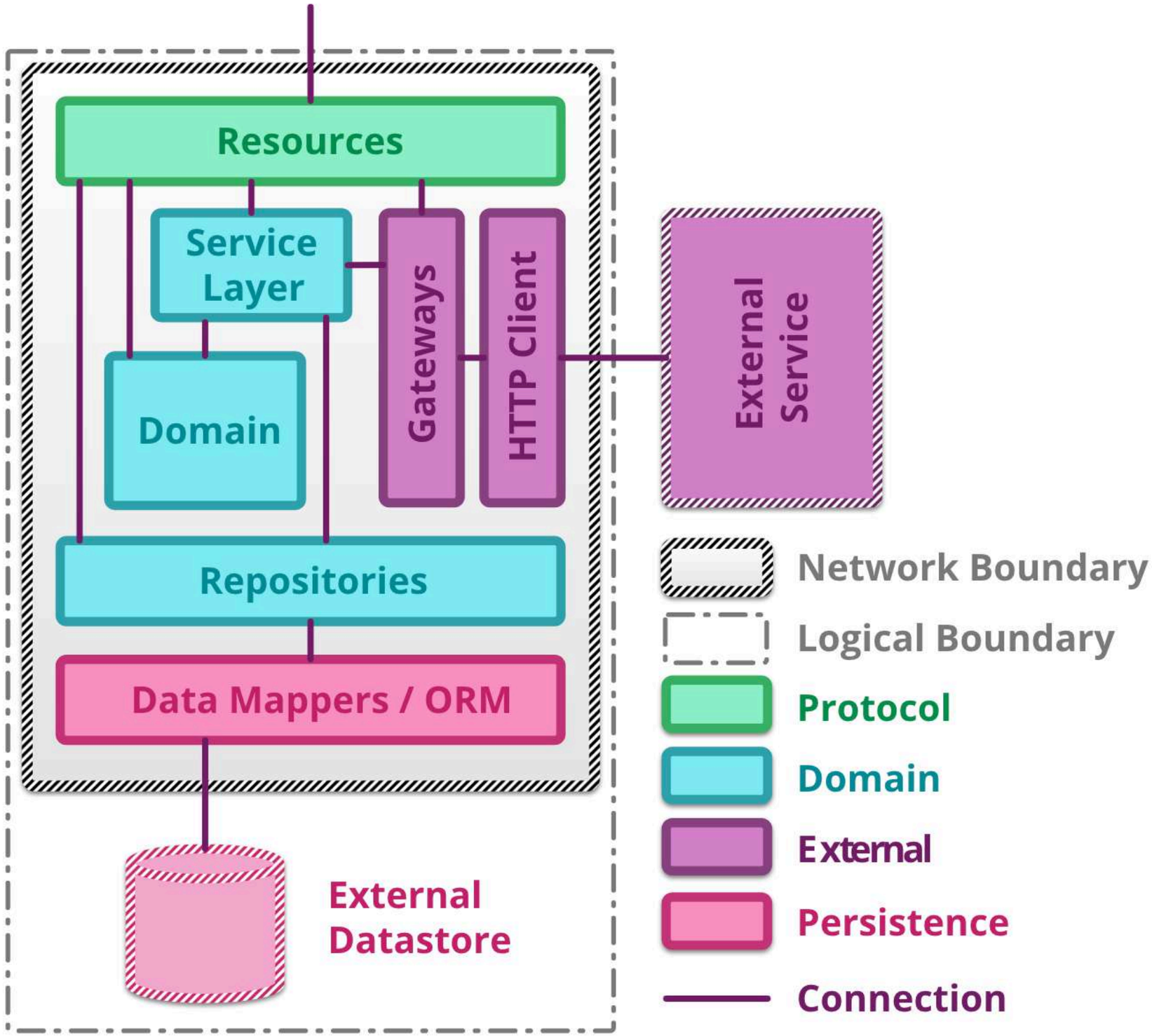
https://blogs.msdn.microsoft.com/cesardelatorre/2011/03/10/great-diagram-about-our-ddd-nlayered-net-4-0-architecture/

参考架构对比

应用层对应Resources

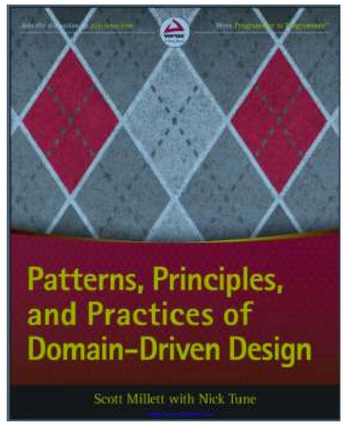
领域层对应
Service Layer,Domain和Repositories

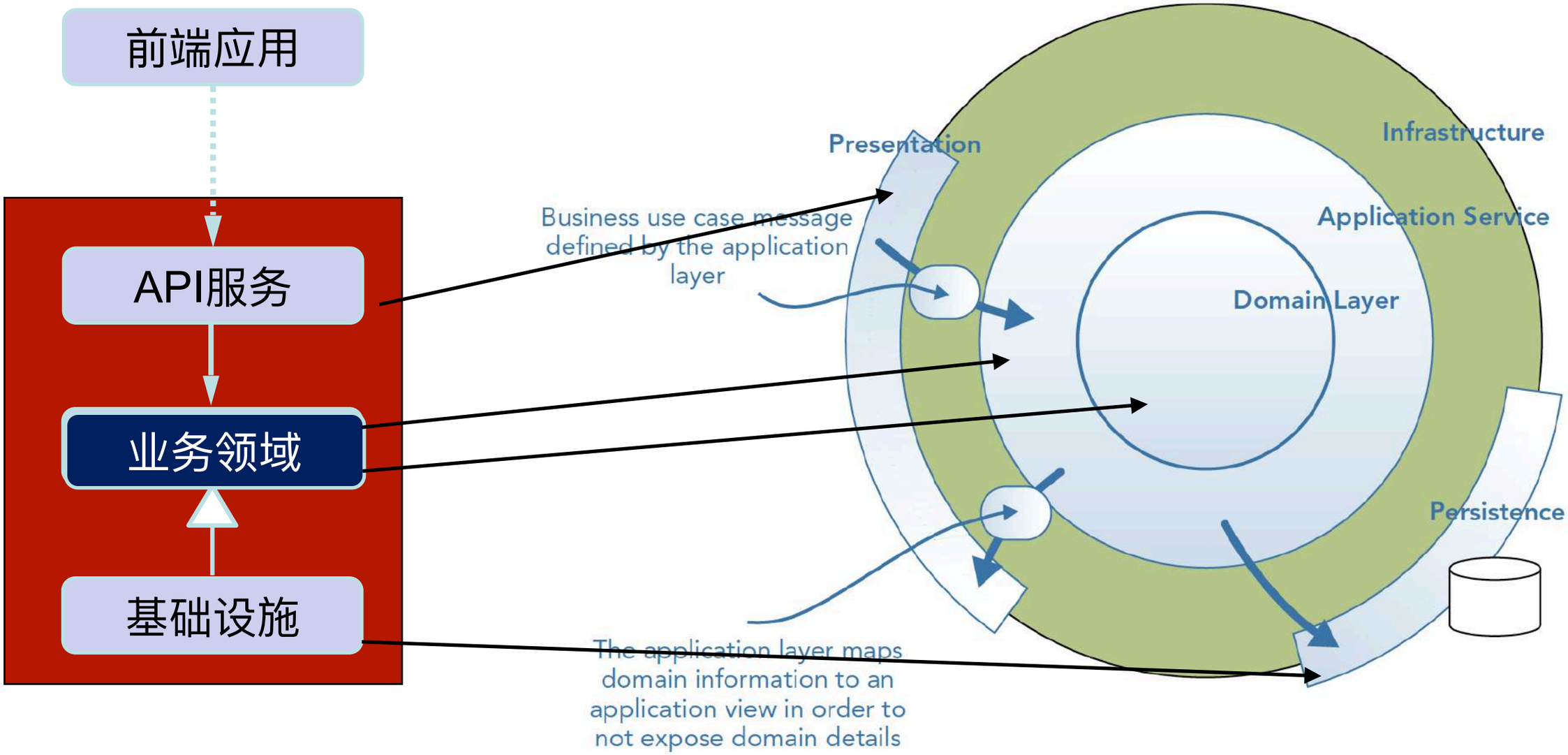
基础设施层对应
Gateways,HTTP Client和ORM



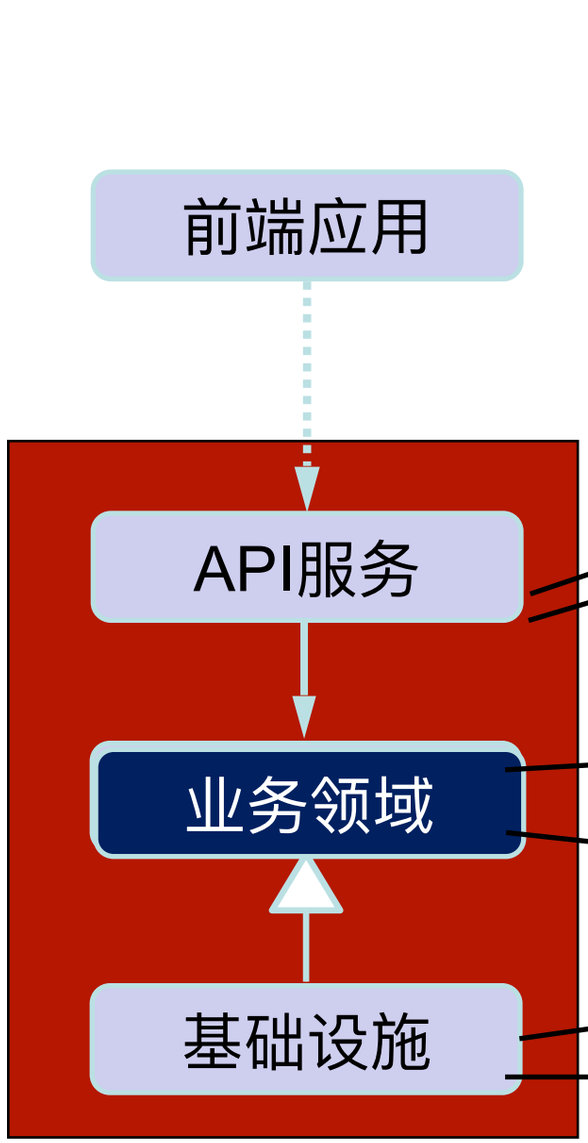
https://martinfowler.com/articles/microservice-testing/#anatomy-connections

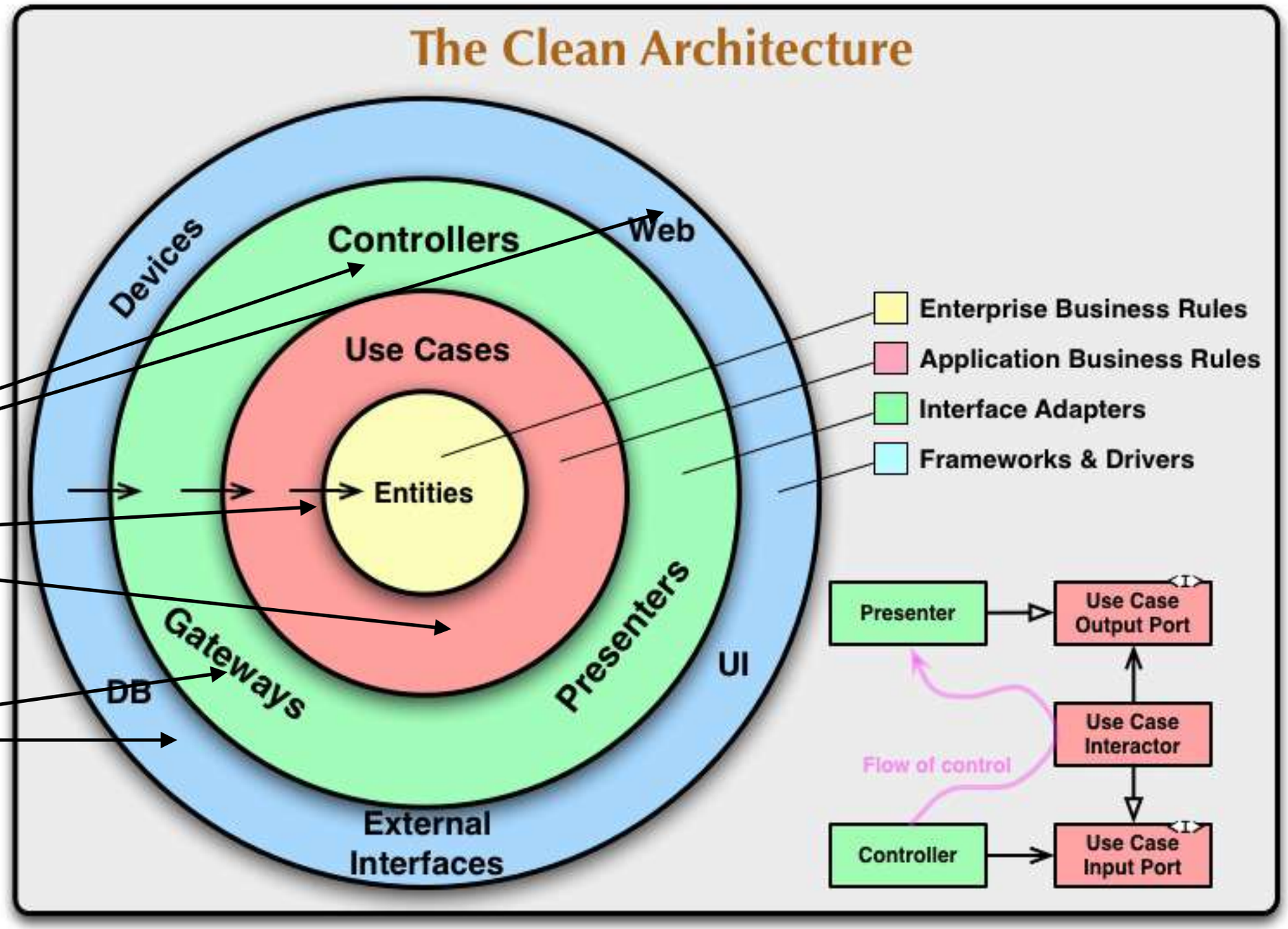
DDD分层架构对比


Patterns, Principles,
and Practices of
Domain-Driven Design
Scott Millett with Nick Tune


前端应用
API服务
业务领域
基础设施
Presentation
Infrastructure
Application Service
Domain Layer
Persistence
Business use case message defined by the application layer
The application layer maps domain information to an application view in order to not expose domain details

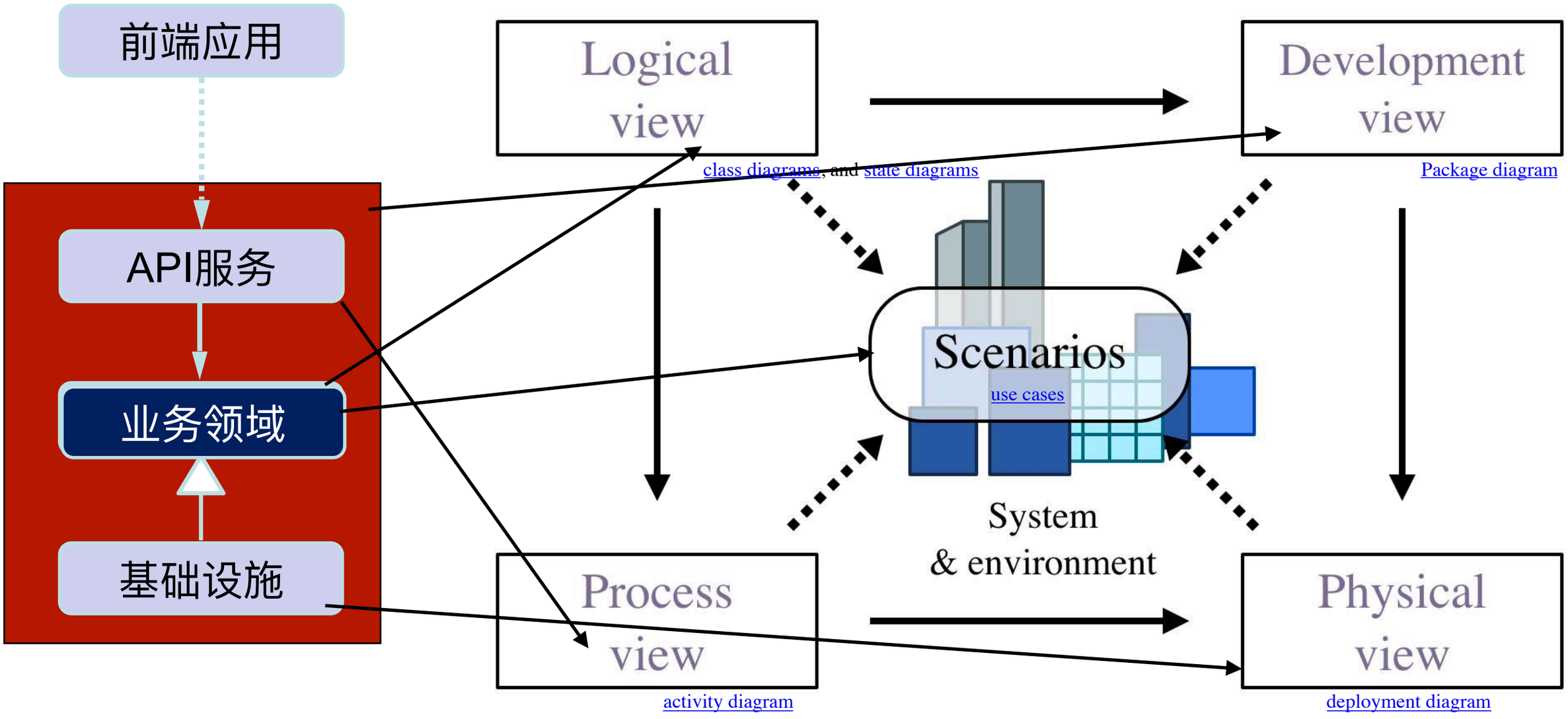
DDD分层架构对比Clean Architecture





https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

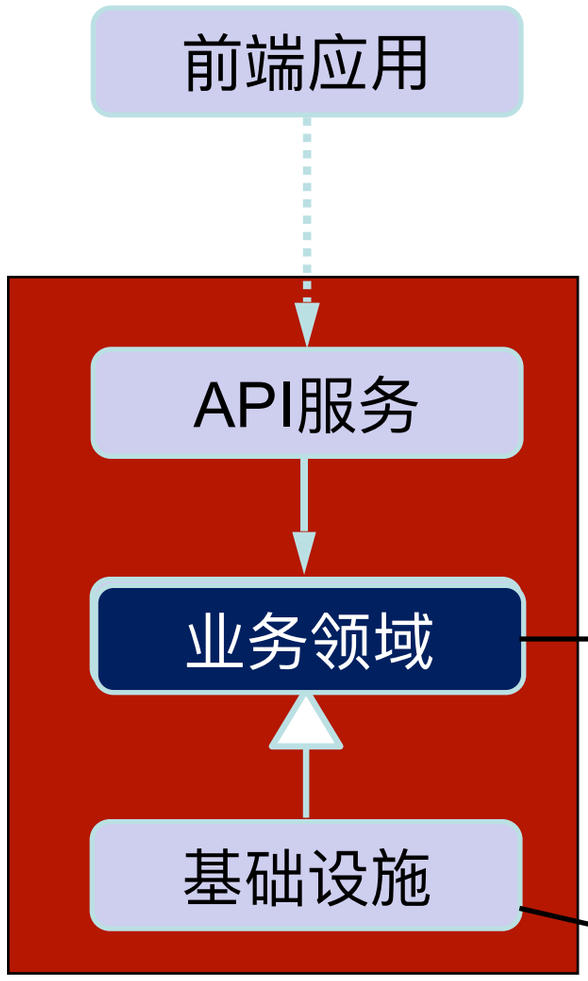
DDD分层架构对比Clean Architecture

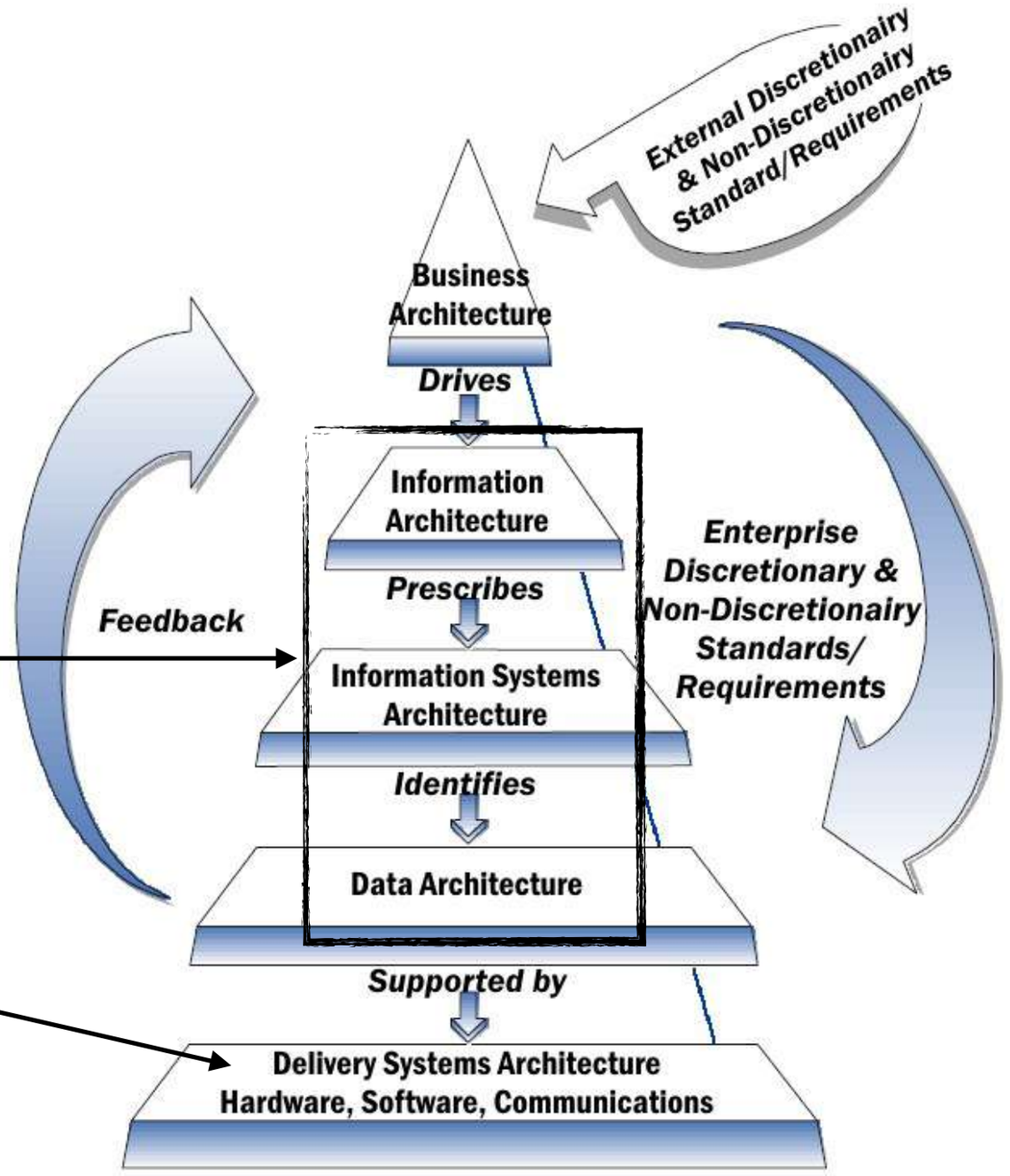


https://en.wikipedia.org/wiki/4%2B1_architectural_view_model

DDD分层架构对比

Enterprise architecture framework





如何有效防止架构腐化 分层模型适配

架构腐化是系统开发过程中非常头疼的事情。使用DDD分层架构参考实践，可以在系统初始设计的时候领域明确出来。

但问题是开发过程中不知不觉层次模糊，架构师有没有实践能有效阻止架构腐化?

如何从一盘散沙到百万雄师

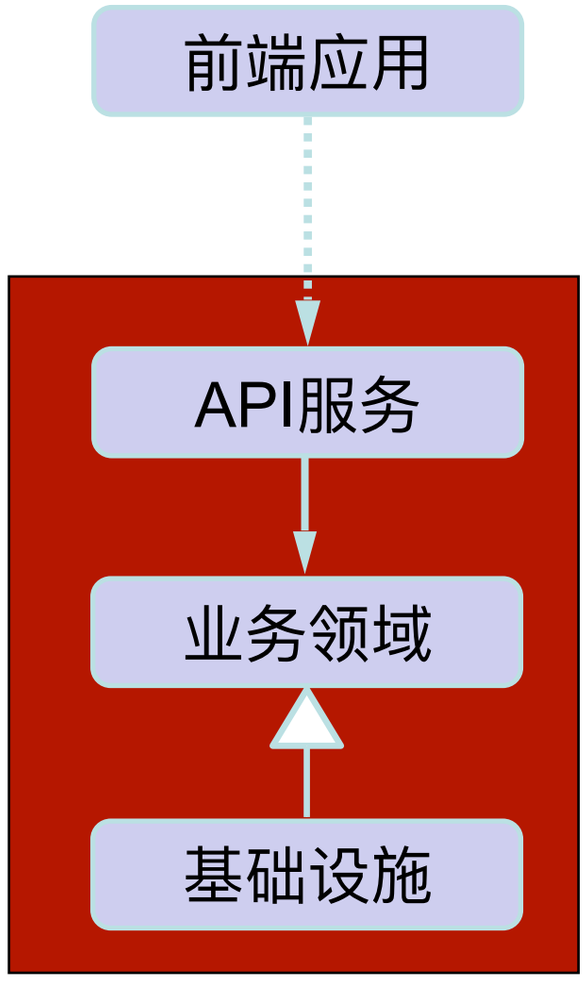
百万雄师的铸造秘密 — 踢正步 叠被子

知行合一

每个人日常就能做到

在日常最细微处抵抗懒散腐化 坚持塑造纪律

DDD分层架构 不同模型适配(名词)

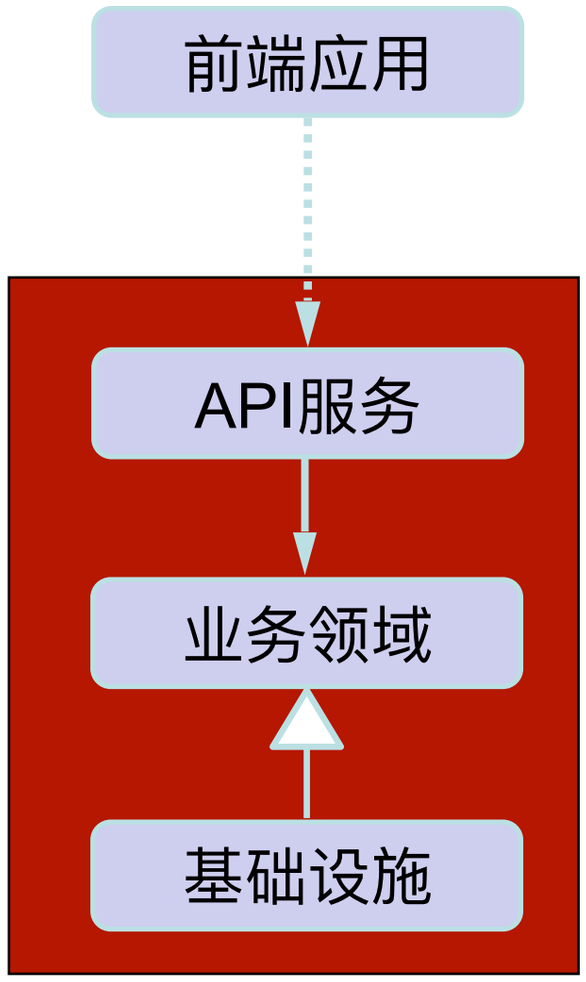


资源模型: 应用想要一把查询获取所有信息，一个操作做完业务

业务模型: 精确表达一个业务概念，分治管理复杂度

存储模型: 性能最优化

DDD分层架构 不同模型适配(动词)

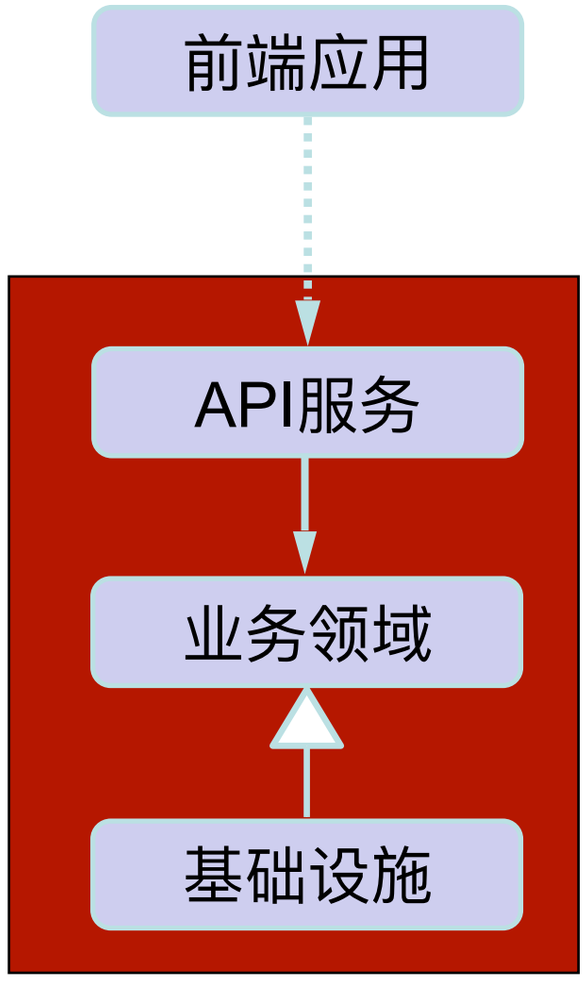


HTTP方法: POST GET PUT DELETE

业务操作: 注册 注销 登录 退出 修改 撤销

存储动作: INSERT UPDATE DELETE

DDD分层架构治理



不同层级的模型坚决隔离，

严格一对一翻译映射，不准复用

资源模型: 应用想要一把查询获取所有信息，一个操作做完业务

HTTP方法: POST GET PUT DELETE

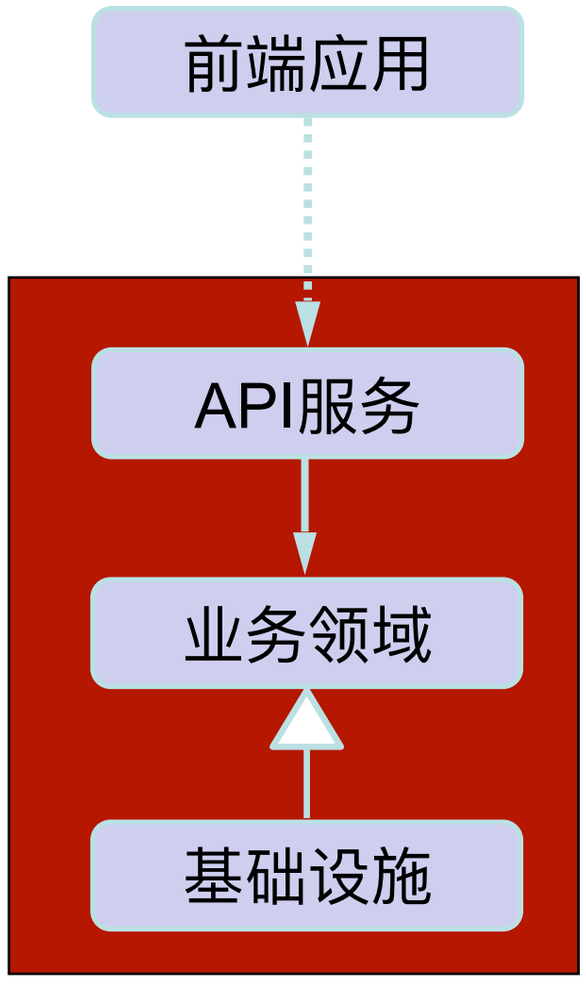
业务模型: 精确表达一个业务概念，分治管理复杂度

业务操作: 注册 注销 登录 退出 修改 撤销

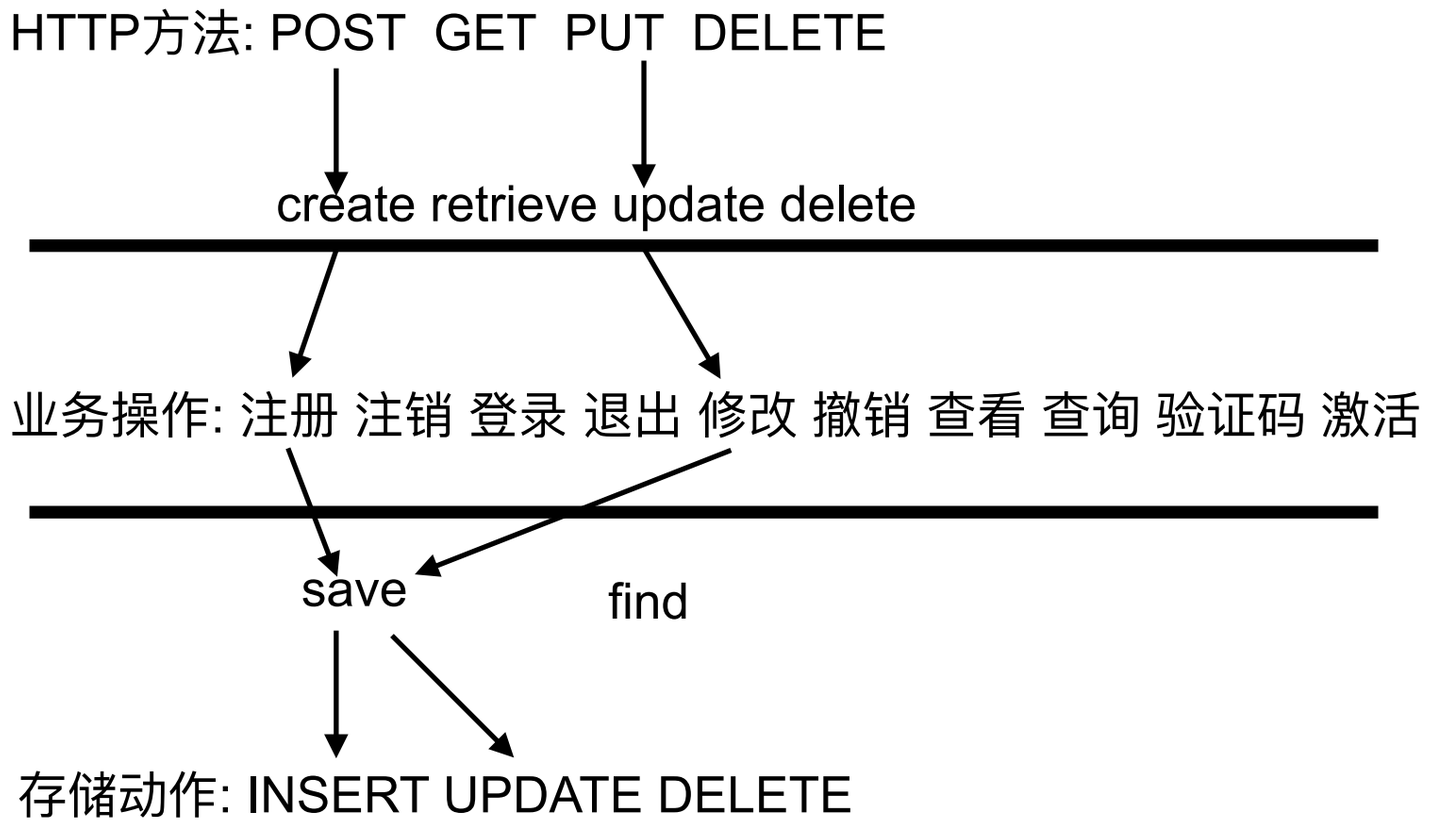
存储模型: 性能最优化

存储动作: INSERT UPDATE DELETE

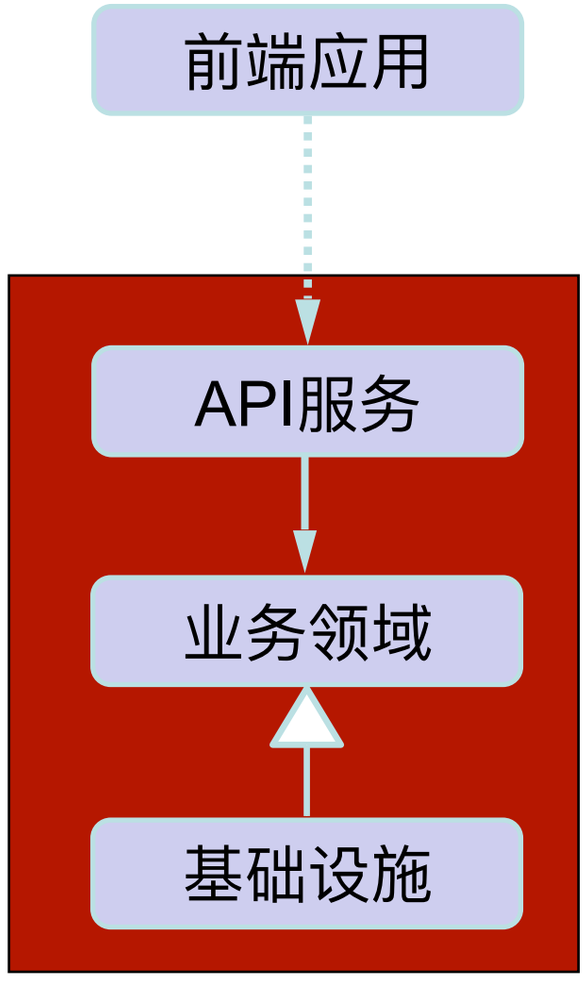
DDD分层架构治理



动词翻译映射



DDD分层架构治理



名词翻译映射

资源模型: 用户想要一把查询获取所有信息，一个操作做完业务

HTTP字符文本

Json

视图对象

一个用户请求一个视图对象

业务模型: 分解的业务概念

多个固定的业务概念表达业务规则

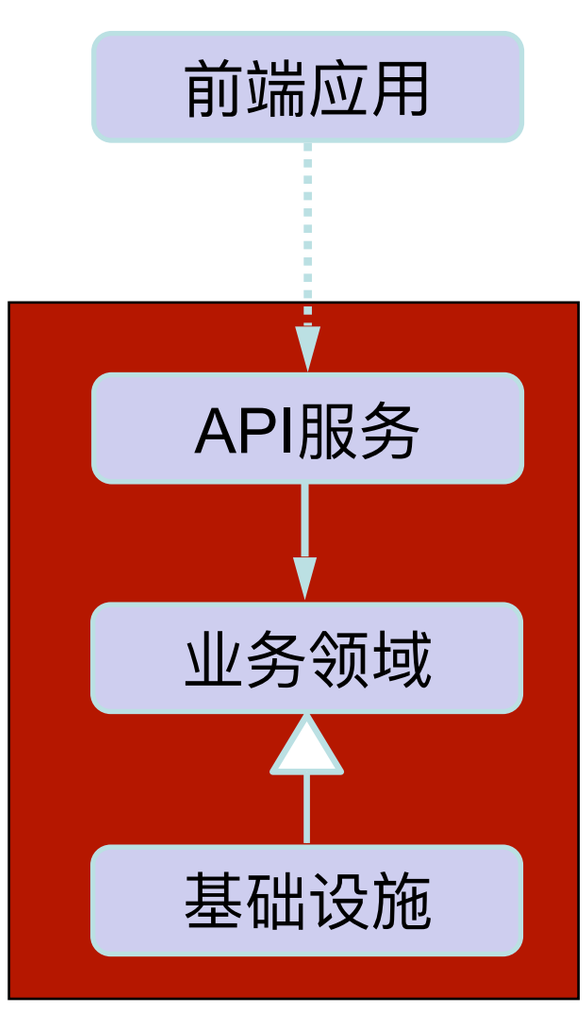
SQL: 无冗余字段设计, 主外键关联

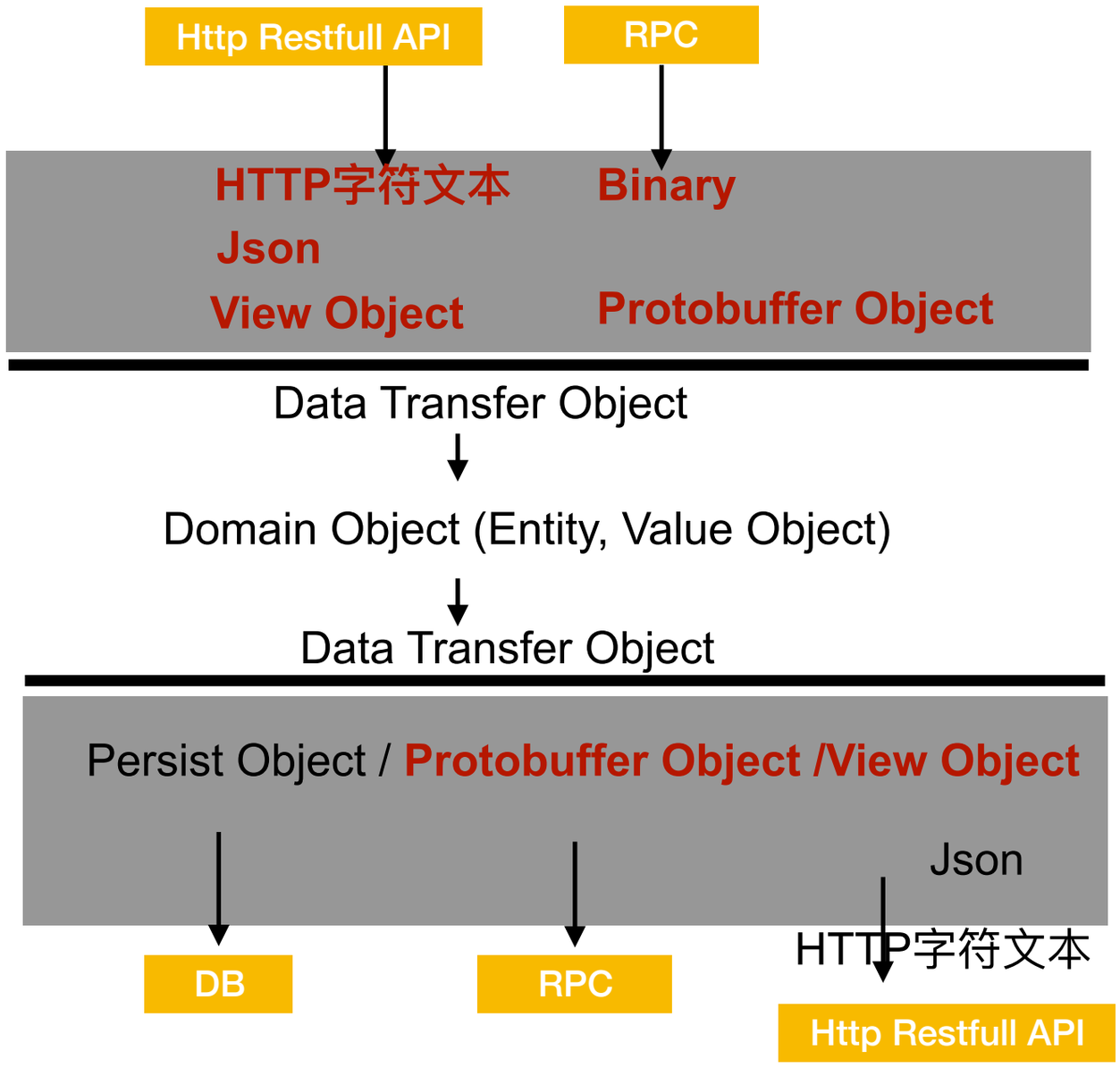
NoSQL: 冗余字段设计，无Schema宽列

存储模型: 性能最优化

DDD分层架构治理

名词翻译映射



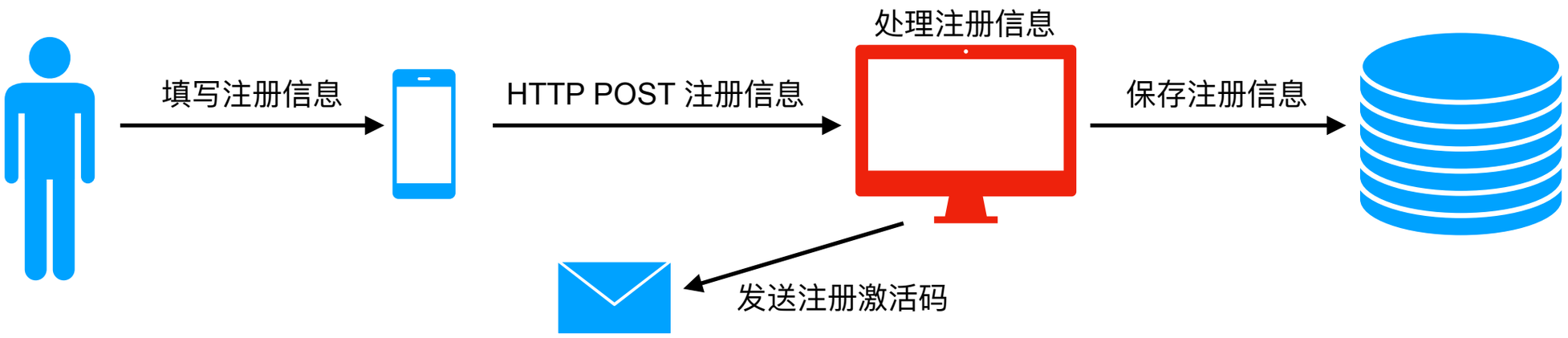


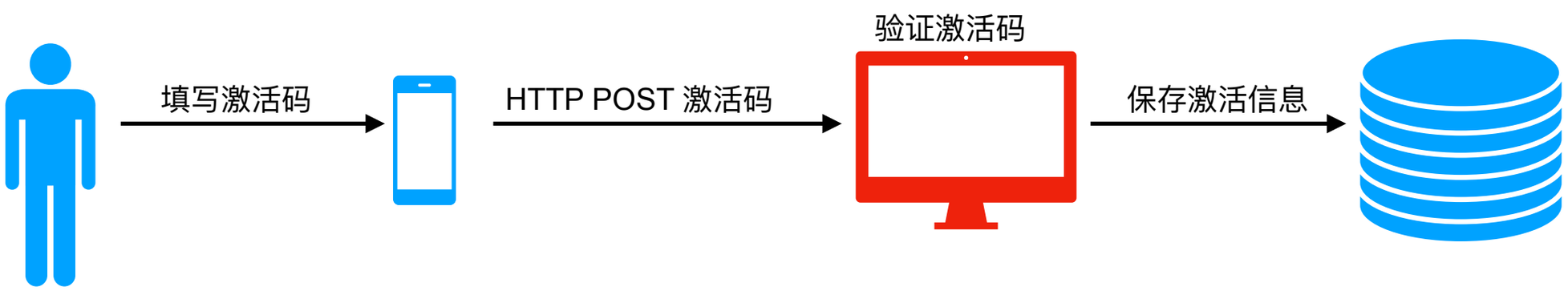
分层模型适配实例

用户账户注册和激活

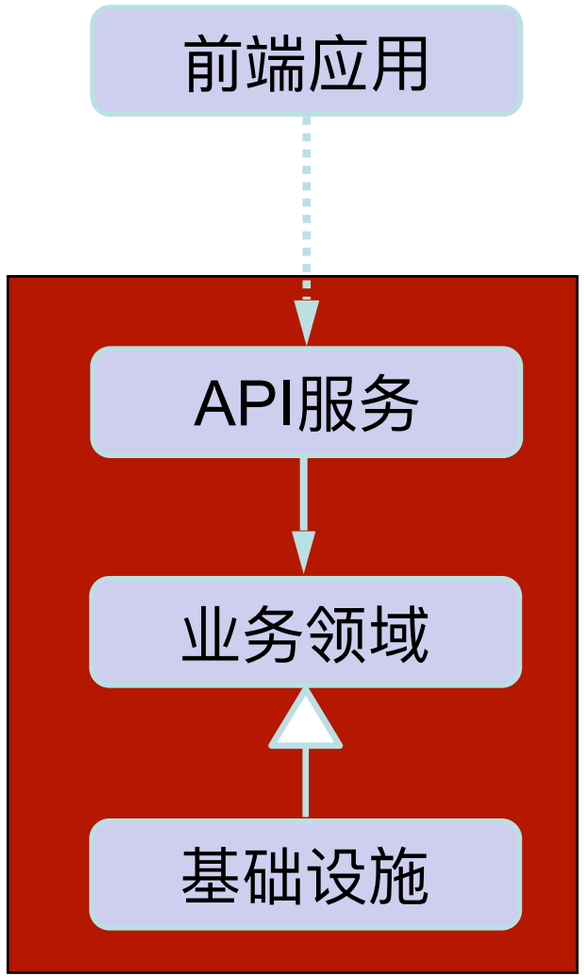
架构治理中的踢正步

案例-用户注册和激活


填写注册信息
HTTP POST 注册信息
处理注册信息
保存注册信息
发送注册激活码


填写激活码
HTTP POST 激活码
验证激活码
保存激活信息

案例-用户注册 -1



POST /users

```
{"mobile": “13520181212”,
"password": "29WLV74@2"}
```

```
createUser(registerInfo)
```

一对一字段映射

```
class RegisterInfo(mobile: String,
password: Option[String])
```

翻译到业务模型

```
register(account, parterId)
```

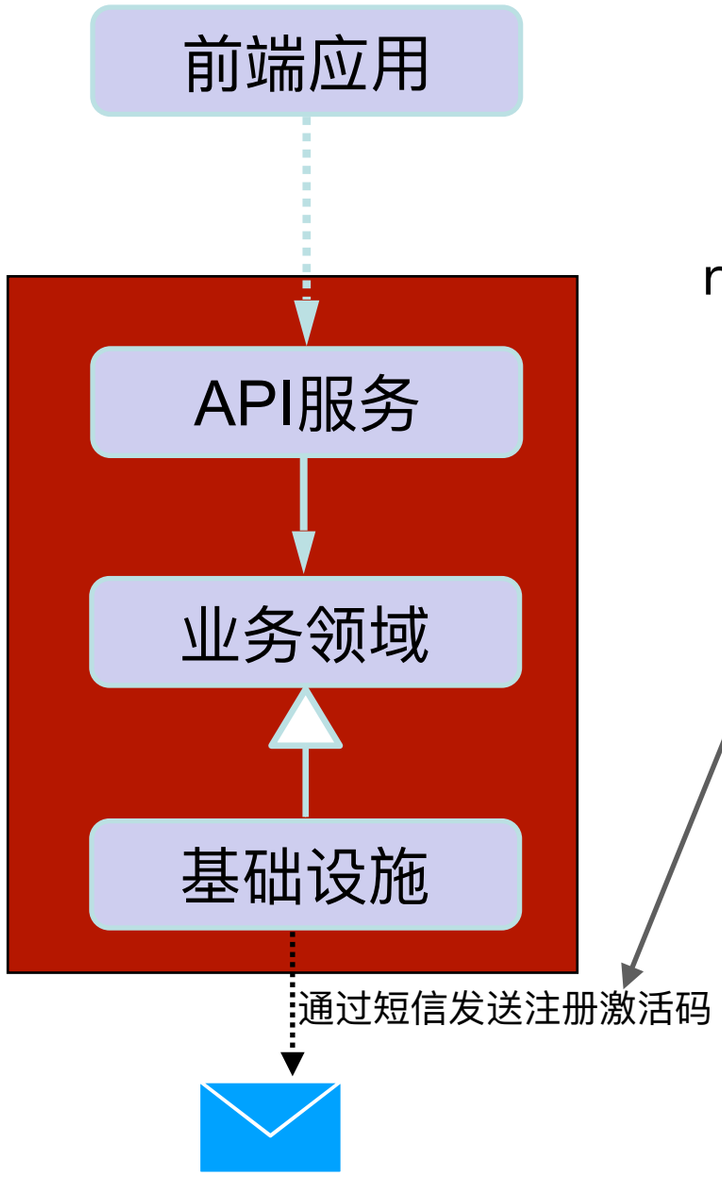
```
class Account(uid:Option[UUID], mobile: String,
password: Option[Password],activated:Boolean=False)

class Password(passwordPlain: String)
  def passwordHash() = encode(password)
  def passwordExpiration() =
```

```
save(account)
```

从业务业务模型
翻译出存储模型

```
CREATE TABLE `account` (
  `uid` varchar(32) NOT NULL,
  `login_id` varchar(128) DEFAULT NULL,
  `password_hash` varchar(128) DEFAULT NULL,
  `name` varchar(100) DEFAULT NULL,
  `activated` bit(1) DEFAULT NULL DEFAUT 0,
  `created_timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMES
  `password_expiration` timestamp DEFAULT NULL,
  PRIMARY KEY (`uid`)
);
```

案例-用户注册 -2




```
register(account, parterId)
    verify(account)
    save(account)
    notifyVerificationToken()
```

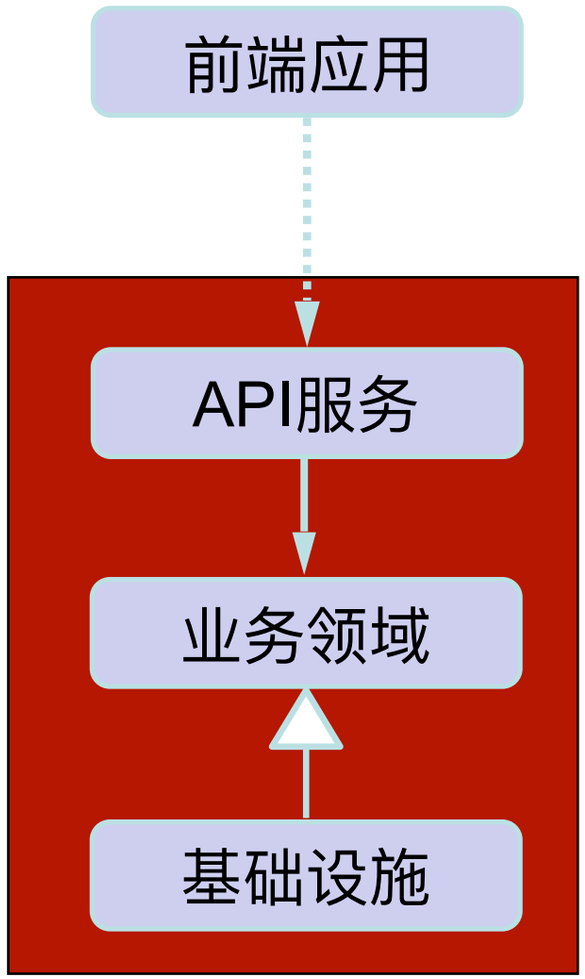
```
class VerificationToken(account: Account
expireToken: String,
expiryTime:Date,
verifyType: String)
```

```
save(verificationToken)
```

从业务业务模型
翻译出存储模型

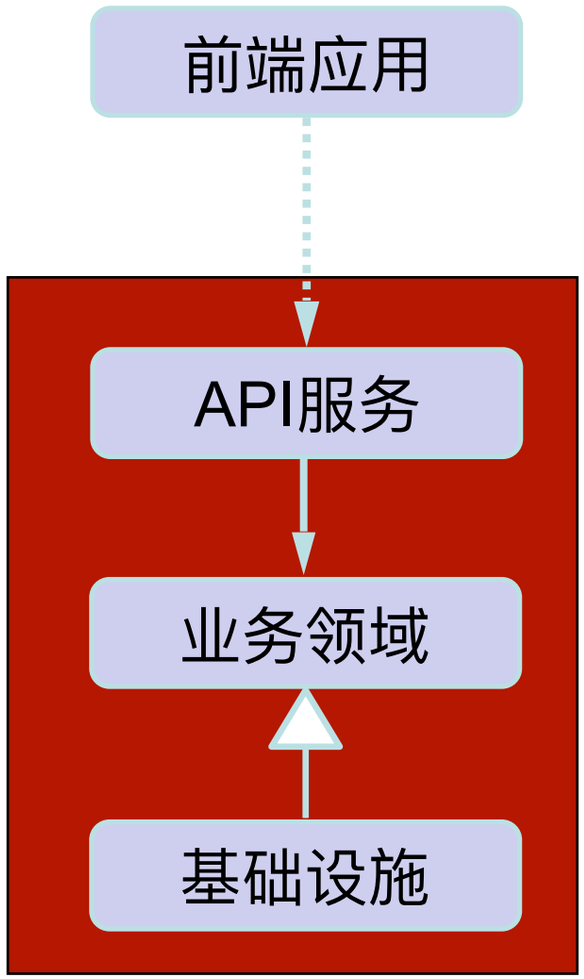
```
CREATE TABLE `verification_token` (
  `verification_id` bigint(20) unsigned NOT NULL AUT
  `account_uid` varchar(32) NOT NULL,
  `token` varchar(128) NOT NULL,
  `expiry_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `verify_type` varchar(32) NOT NULL,
  PRIMARY KEY (`verification_id`),
  CONSTRAINT `account_uid_fk` FOREIGN KEY (`account_uid`) REFERENCES `accou
);
```

案例-用户注册 - 动名词个数总结



	动词	名词
应用服务	1	1
领域模型	1	3
基础设施	3	2

案例-用户激活



PUT /users/activate

```
{"uid": “d219-xww0-13af-b021”,
"token": “23u3$z4o6799"}
```

一对一字段映射

```
updateActivate(activateInfo)
```

```
class ActivateInfo(uid: String,
token: String)
```

翻译到业务模型

```
activate(VerificationToken)
```

```
class VerificationToken(account: Account
expireToken: String,
expiryTime:Date,
verifyType: String)
```

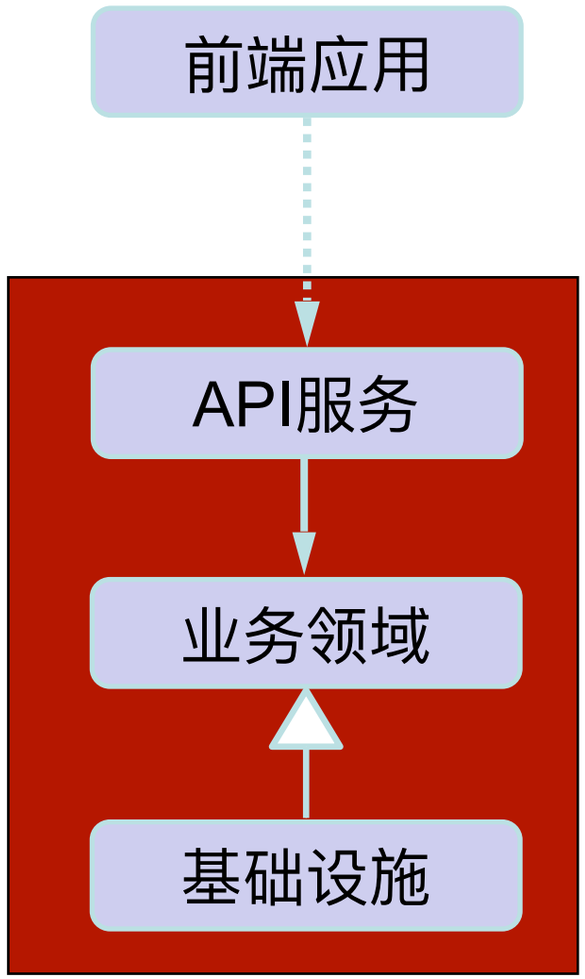
```
save(account)
```

```
UPDATE `account` SET `activated` =
{account.activated}
```

从业务业务模型
翻译出存储模型

```
delete(verificationToken)
```

案例-用户激活 - 动名词个数总结



	动词	名词
应用服务	1	1
领域模型	1	2
基础设施	2	2

原则

即使不同层的模型字段完全一样，也要求各自创建。

架构师职责

提炼概念

管理复杂性

坚持原则 但不增加工作量

```
public class SystemUser {
    private String firstName;
    private String lastName;
    private String phone;
    private String addressLine1;
    private String addressLine2;
    private String city;
    private String state;
    private String zip;

    // getters and setters
    // ...
}
```

```
public class User {
    private String first;
    private String last;
    private String address;
    private String city;
    private String state;
    private String zip;
    private String phone;

    // getters and setters
    // ...
}
```

```
import org.apache.commons.beanutils.BeanUtils;

UserBean newObject = new UserBean();
BeanUtils.copyProperties(newObject, oldObject);
```

坚持原则

看似工作量多 其实降低了复杂度(纠结)

```
systemUser.setFirstName(user.getFirst());
systemUser.setLastName(user.getLast());
systemUser.setPhone(user.getPhone());
systemUser.setAddressLine1(user.getAddress());
systemUser.setAddressLine2(user.getAddress());

systemUser.setCity(user.getCity());
systemUser.setState(user.getState());
systemUser.setZipcode(user.getZip());
```

如果从概念上User模型就是只有一个address

保持User模型的纯洁

降低了复杂度

守护原则-CI检查 分层架构单元测试

```
import com.tngtech.archunit.junit.AnalyzeClasses;
import com.tngtech.archunit.junit.ArchTest;
import com.tngtech.archunit.lang.ArchRule;

import static com.tngtech.archunit.library.Architectures.layeredArchitecture;

@AnalyzeClasses(packages = "com.tngtech.archunit.example")
public class LayeredArchitectureTest {
    @ArchTest
    static final ArchRule layer_dependencies_are_respected = layeredArchitecture()

            .layer(“Applications").definedBy("com.tngtech.archunit.example.applications..")
            .layer(“Domain").definedBy("com.tngtech.archunit.example.domain..")
            .layer("Infrastructure").definedBy("com.tngtech.archunit.example.infrastructure

            .whereLayer("Applications").mayNotBeAccessedByAnyLayer()
            .whereLayer("Services").mayOnlyBeAccessedByLayers("Applications")
            .whereLayer(“Infrastructure").mayOnlyBeAccessedByLayers("Services");
}
```

https://www.archunit.org/

守护原则-CI检查 分层模型单元测试

```
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.classes;
import static com.tngtech.archunit.lang.syntax.ArchRuleDefinition.noClasses;

@AnalyzeClasses(packages = "com.tngtech.archunit.example")
public class DaoRulesTest {
    @ArchTest
    static final ArchRule DAOs_must_reside_in_a_dao_package =
            classes().that().haveNameMatching(".*Dao").should().resideInAPackage("..dao..")
                    .as("DAOs should reside in a package '..dao..'");

    @ArchTest
    static final ArchRule entities_must_reside_in_a_model_package =

classes().that().resideInsideOfPackage("..domains.model..").should().resideInAPackage("..model..")
                    .as("Domain models should reside in package '..model..'");

    @ArchTest
    static final ArchRule only_DAOs_may_use_the_EntityManager =
            noClasses().that().resideOutsideOfPackage("..dao..")
                    .should().accessClassesThat().areAssignableTo(EntityManager.class)
                    .as("Only DAOs may use the " + EntityManager.class.getSimpleName());
}
```

https://www.archunit.org/

吴雪峰

Shanghai Pudong New Area

Scan the QR Code to add me on WeChat

THANK YOU

面对面建群：1201

欢迎一起探讨大规模系统架构设计、治理和演进

吴雪峰@ThoughtWorks 201811

DDD CHINA