链滴

DDD 探险——基于 GraphQL+Dgraph 实践

作者：crick77

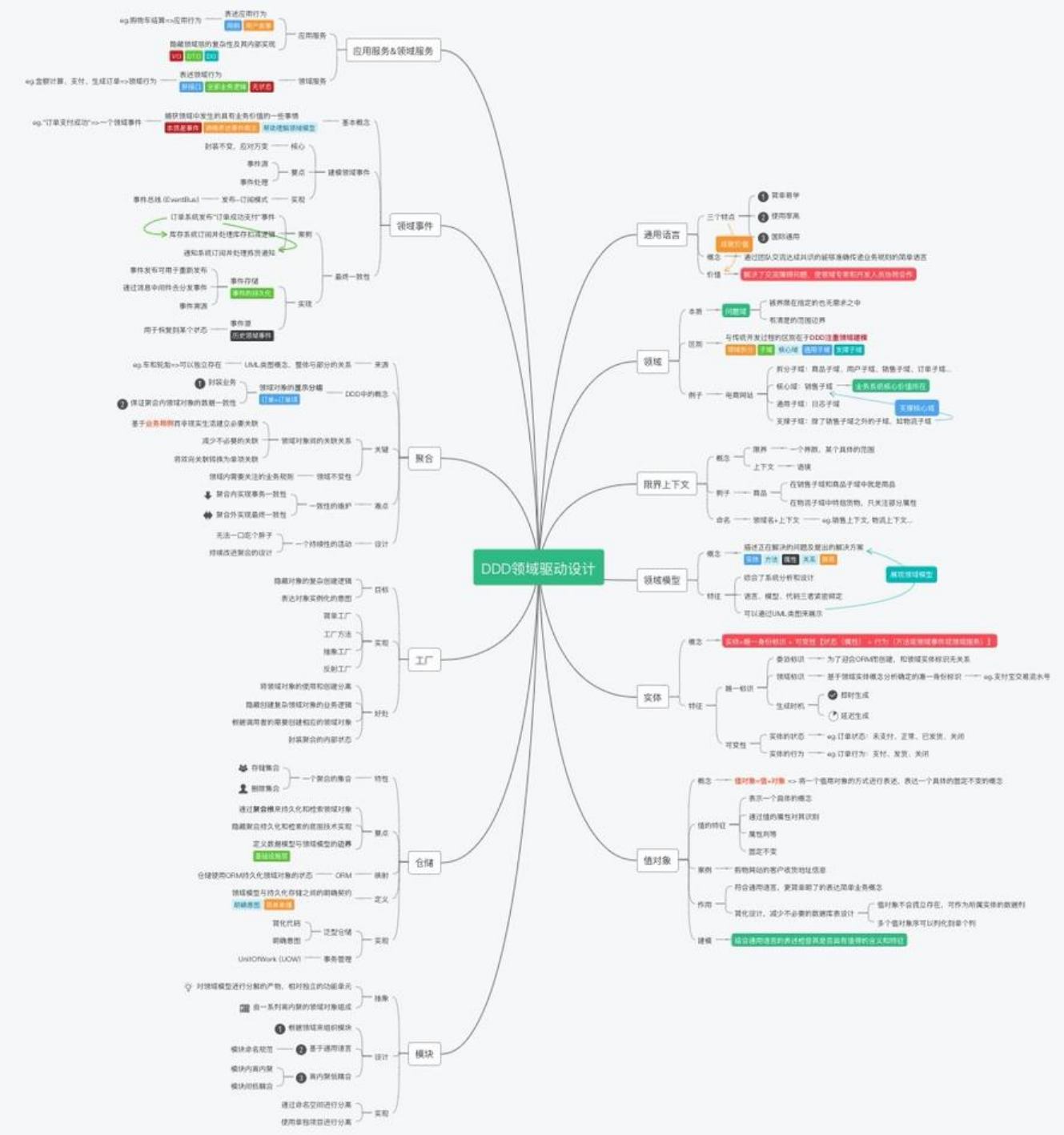
原文链接：https://ld246.com/article/1597402382777

来源网站：链滴

1. DDD

代码已开源: https://github.com/YituHealthcare/Arc

《DDD从入门到放弃》 只需要一张脑图



一个很奇怪的现象，介绍DDD的文章基本往往只聊思想和概念，不聊落地和代码。

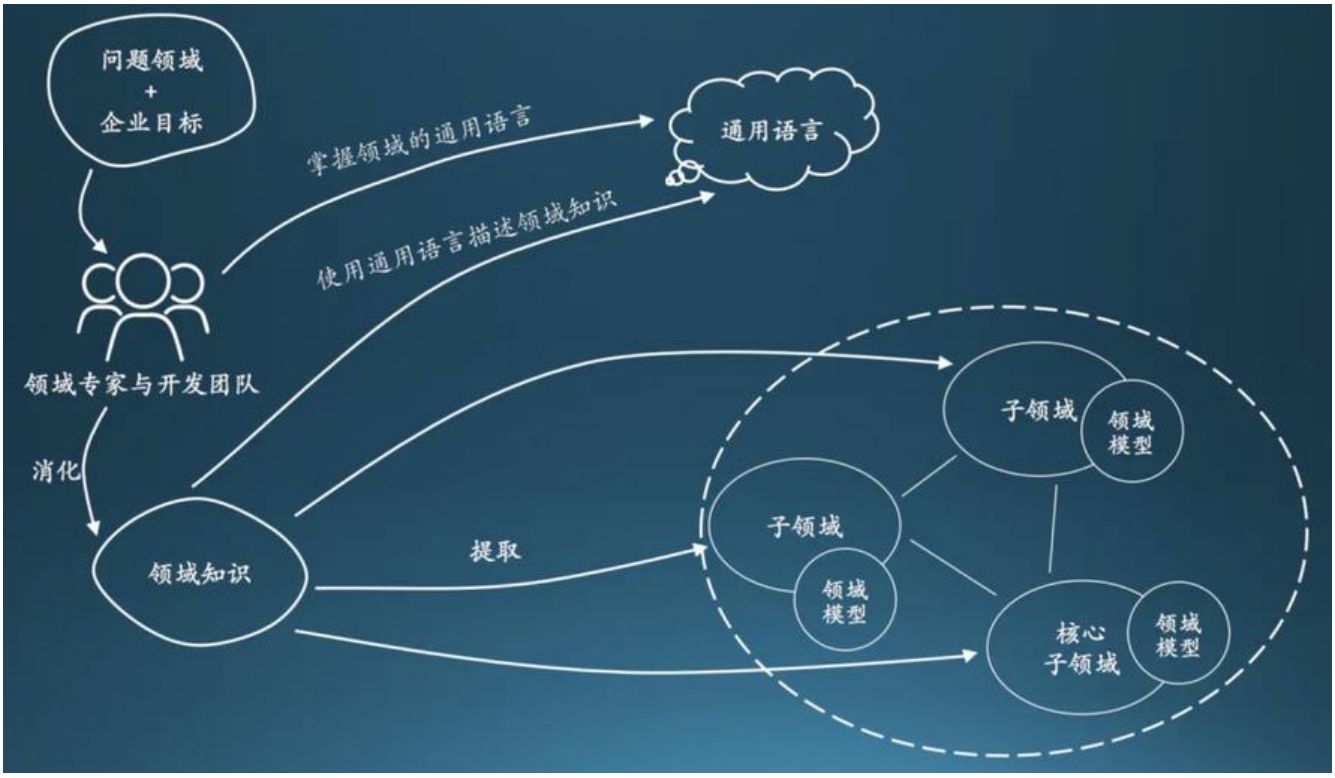
"思想，意会就好"，或者说是不是DDD只负责前期的设计。

因为不像设计模式一样有直观的代码用例，又拥有大量生涩难懂的概念，往往在落地时遇到了困境。
文基于笔者对DDD的理解、记录技术选型过程及激进的代码讲解，槽点过多，欢迎沟通。

原文链接：DDD 探险——基于 GraphQL+Dgraph 实践

1.1 DDD 是什么

DDD (Domain-Driven Design) 领域驱动设计。指对软件所涉及到的领域进行建模，以应对系统规过大时引起的软件复杂性的问题。



1.1.1 DDD 简单示例

尝试通过一个简单的示例来说明

如果我希望提供 **更新用户密码or用户姓名or年龄** 的功能，基于面向数据的角度，我会有一张user的，上述操作都是对user表的更新，我可以设计为 UserService.save(user) 来实现，并通过键值约束实一些重复限制。但是后续迭代时只看到这个方法，不了解被外部如何使用，这个方法很快就无法继续代。

我们尝试**不要专注在实体和值对象**，通过语义明确的API来实现。所以我可以提供N个方法

- UserService.updatePassword(password, userId);
- UserService.updateName(name, userId);
- UserService.updateAge(age, userId);

这种代码已经很符合基于Spring+贫血模型的风格

我们再重新审视一下最原始的需求，**更新用户密码or用户姓名or年龄**，这个操作是否应该由User来完？ 比如

- User.updatePassword(password);
- User.updateName(name);
- User.updateAge(age);

这就是DDD风格的代码，按照最贴合业务及正向思维的逻辑来编写。如果我们不考虑数据持久化、如配合Spring使用这些必须解决的问题的话，会更完美一些。当然，如果我们拥有一台内存无限、永不机的服务器，也可以实现。

1.1.2 DDD 不是什么

DDD不是银弹。

如果是银弹，或者说适用于大部分场景，那么DDD应该早就变成主流了(另一方面的原因是对工程人的要求较高)。

但是在某些特定场景下，DDD是很合适的一套方案。

1.2 为什么要用DDD

在一个创新探索领域、或者行业门槛较高的领域，只从工程角度太片面，需要引入领域专家一同工作。

这时有几个问题

1. 不同职业，使用不同工具，说着不同术语的人如何协同工作？如何确认彼此的想法已对齐？
2. 如何加深业务理解和更准确的定义，以便实现业务深耕和探索？
3. 建模知识如何沉淀？除了业务专家口口相传这种10bit/s的数据量传播方式外，是否有更通用高效方法？
4. 在业务探索期，如何降低频繁变更的成本？不需要在业务建模、架构设计与编码之间相互翻译？

也许通过DDD可以解决上述问题。

思考：如果一个行业相对成熟稳定，已经有知名的产品，是不是直接把对方的解决方案“借鉴”过来好一些？

1.3 怎么用DDD

```
usecase "讨论" as A
usecase "设计" as B
usecase "实践" as C
usecase "推翻" as D

A -right-> B
B -down-> C
C -left-> D
D -up-> A
```

1. 通过用户故事，每个人按照自己的理解定义相关"领域实践"、"行为操作"及"用户角色"，结构格式为
    1. actor->command->event，即:某人做了某个操作，产生了某个事件
2. 定义通用语言(团队自己创建的公用语言)，基于上述定义，通过归纳总结，明确常识及概念，并给定义，最终输出业务词典表。后续通过常识名沟通讨论
    1. 通用语言需要明确限界上下文(Bounded Context)，这个常识名只在对应的上下文内代表相应

含义。同一个概念在不同上下文内，代表着不同的含义，拥有不同的属性及行为

2. 对齐过程中包括但不限于

- 讨论
- 参考资料
- 引用标准
- 查阅词典

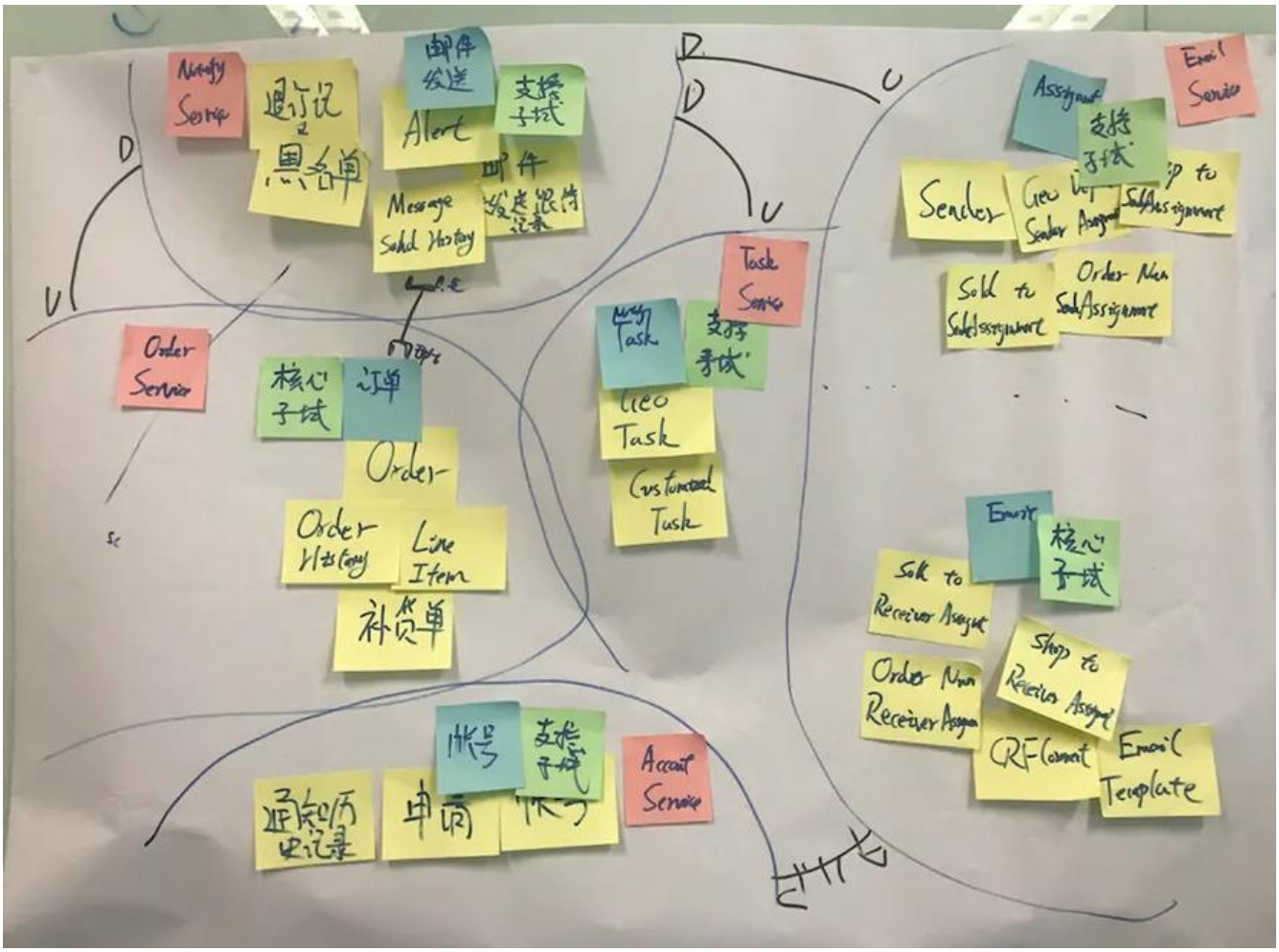
3. 示例

1. Project: 为了达到某个产品迭代、产品模块开发等目的所做的工作.

2. Milestone: 表述一个Project的某个时间阶段及阶段性目标. 一个Project可以同时拥有多个于相同或者不同阶段的Milestone.

3. 通过 **EventStorming**将用户故事抽象为"贴纸领域图"，并按照聚合根进行聚合

1. 此过程和通用语言定义可同时进行，在讨论时明确定义
2. **EventStorming**详情可以通过 https://www.eventstorming.com 了解，不再详细展开



上述过程相对通用，在我的实践过程中，又增加了一步

1. 按照聚合后的"贴纸领域图"编写 GraphQL Schema

2. GraphQL

2.1 GraphQL 是什么

https://graphql.org/

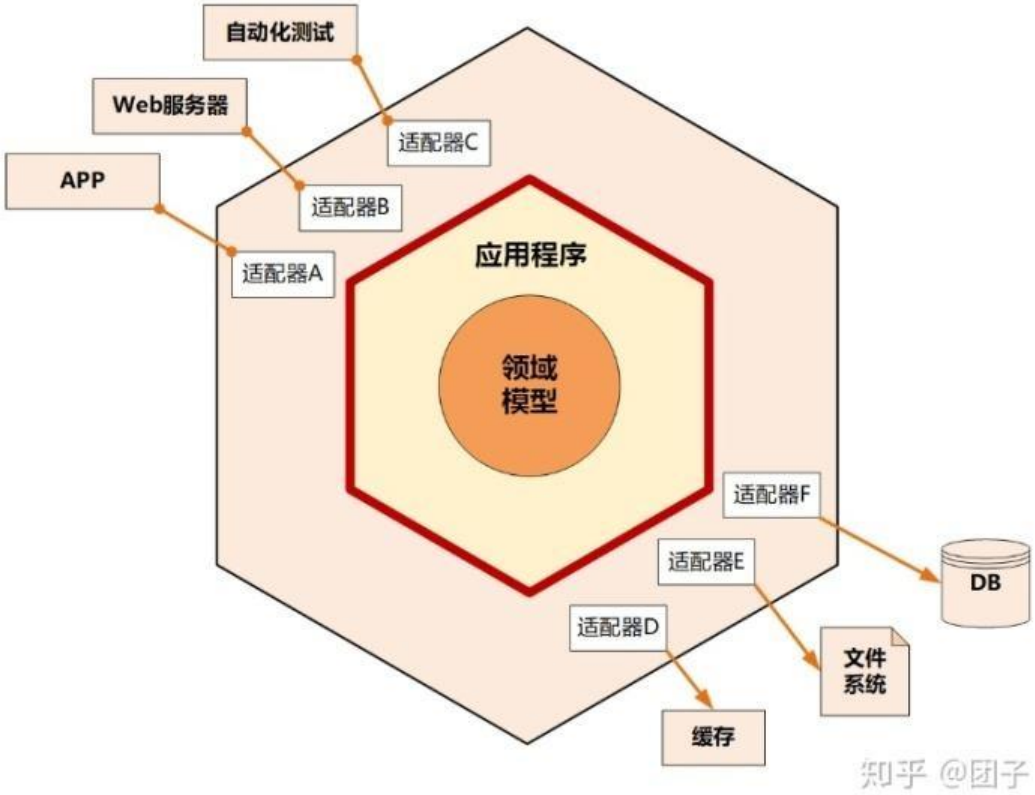
一种用于 API 的查询语言

GraphQL 既是一种用于 API 的查询语言也是一个满足你数据查询的运行时。 GraphQL 对你的 API 的数据提供了一套易于理解的完整描述，使得客户端能够准确地获得它需要的数据，而且没有任何冗，也让 API 更容易地随着时间推移而演进，还能用于构建强大的开发者工具

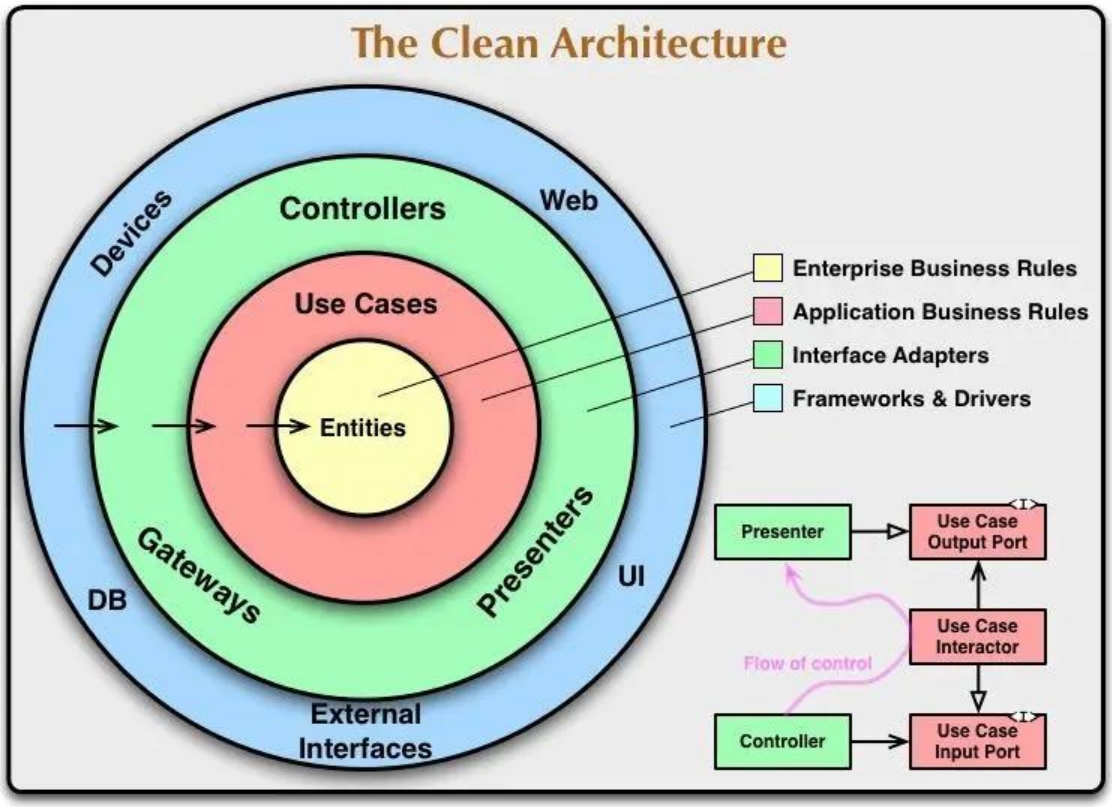
2.2 为什么要用GraphQL

如何保障领域模型和实践之间同时变动?

前文调侃说"如果我们拥有一台内存无限、永不宕机的机器"，我们可以很方便的开发出符合DDD的代，而这就是我们代码的领域层。但是实际工作中，我们不仅没有这台机器，而且不得不和各方进行交，比如说我们要和UI交互，提供相关数据。这时根据六边形orCA架构，我们拓展其中的一条边。



原文链接：DDD 探险——基于 GraphQL+Dgraph 实践



此时问题出现了。当我们画了实体、连了各种关系线之后，还要再去拉平，做各种转换(从领域层到AP交互层，前端也会做同样的各种 xxObject的转换，但我们明明是在同一套共同创建的领域模型下进开发)。这个过程叫做api定义。

如果前端能通过领域模型进行数据操作，是不是可以省略这一过程？那么在2者之间找到一个平衡点就是通过 GraphQL Schema描述领域模型，同时也描述了接口。

2.3 怎么用GraphQL

网上的例子大多是内存中的数据，而没有使用数据库的demo，无论是关系型数据库还是nosql。因实现起来不优雅，拉低了GraphQL的使用体验

```
@Component
public class GraphQLDataFetchers {

   private static List<Map<String, String>> authors = Arrays.asList(
         ImmutableMap.of("id", "author-1",
               "firstName", "Joanne",
               "lastName", "Rowling"),
         ImmutableMap.of("id", "author-2",
               "firstName", "Herman",
               "lastName", "Melville"),
         ImmutableMap.of("id", "author-3",
               "firstName", "Anne",
               "lastName", "Rice")
   );

   public DataFetcher getAuthorDataFetcher() {
      return dataFetchingEnvironment -> {
```

```
            Map<String,String> book = dataFetchingEnvironment.getSource();
            String authorId = book.get("authorId");
            return authors
                    .stream()
                    .filter(author -> author.get("id").equals(authorId))
                    .findFirst()
                    .orElse(null);
        };
    }
}
```

基于前面的推理，我们已经用schema描述了领域模型和api，那么能否再向下拓展一层，用来描述持化信息呢?

3. Dgraph

3.1 GraphQL 是什么

Dgraph is an open-source, scalable, distributed, highly available and fast graph database, des gned from the ground up to be run in production.

3.2 为什么要用Dgraph

1. 图数据库更直观，符合领域设计的过程，比如项目中添加用户，其实只是在项目和用户之间连接一条elong的线

2. 支持GraphQL

3.3 怎么用Dgraph

- https://github.com/dgraph-io/dgraph4j

```
// Query
String query =
"query all($a: string){\n" +
"  all(func: eq(name, $a)) {\n" +
"    name\n" +
 "}\n" +
"}\n";

Map<String, String> vars = Collections.singletonMap("$a", "Alice");

AsyncTransaction txn = dgraphAsyncClient.newTransaction();
txn.query(query).thenAccept(response -> {
    // Deserialize
    People ppl = gson.fromJson(res.getJson().toStringUtf8(), People.class);

    // Print results
    System.out.printf("people found: %d\n", ppl.all.size());
    ppl.all.forEach(person -> System.out.println(person.name));
});
```

我觉得官方示例唯一的作用是说明java不适合，快来学Go吧。。

4. 领域事件

通过消息队列订阅领域事件，进行异步交互即可。

5. Arc

上面吐槽和挖坑了这么多，是因为希望用一套框架解决问题，让DDD的落地实操更顺滑。

Arc集成&实现了很多功能

整体

- 集成zipkin链路追踪
- 接入Spring，通过配置+Annotation的方式简化操作

GraphQL 层面

- 内嵌了playground & voyager 方便调试及梳理领域关系
- 自定义异常处理
- intercept自定义拦截器
- 提供graphqlClient用于服务端之间GraphQL调用
- 自动解析schema定义的type、union type 及interface类型
- 封装controller层，扫描并解析@DataFetcherService及@GraphqlMutation、@GraphqlQuery法

Dgraph 层面

- intercept自定义拦截器
- 通过 properties 配置数据库信息
- 自动扫描xxxDgraph.xml， 支持编写复杂语句，并提供静态、动态变量处理
- 提供RDF处理工具
- SimpleRepository提供save、getOne、getAll、relationship、upsert等基本操作

mq

- 基于VM轻量级消息队列
- 拦截并基于订阅发送领域消息

5.1 开发流程

通过maven添加Arc框架后的整体开发流程:

start

: create graphql.schema & dgraph.schema;

```
: create javaBean with \n @DgraphType \n @UidField \n @RelationshipField;
: create @Repository class extends SimpleDgraphRepository;
: add xxDgraph.xml for complex persistence handle;
: create @DataFetcherService class include \n @GraphqlQuery \n @GraphqlMutation;
: browser http://localhost:${port}/playground for debug;

end
```

1. 定义领域模型，产生graphql.schema及dgraph.schema文件。
2. 创建javaBean并指定@DgraphType、@UidField、@RelationshipField
3. 创建 SimpleDgraphRepository 的子类声明为@Repository
4. 编写xxDgraph.xml实现query方法实现复杂DB操作
5. 创建 @DataFetcherService类及@GraphqlQuery、@GraphqlMutation 方法
6. 通过 http://localhost:${port}/playground 进行调试

5.2 Sample

graphql schema

```
scalar DateTime

schema{
    query: Query,
    mutation: Mutation
}

type Query{
    project(
        id: String
    ): Project
    users: [User]
}

type Mutation{
    createProject(
        payload: ProjectInput
    ): Project
    createMilestone(
        payload: MilestoneInput
    ): Milestone
}

"""
项目分类
"""
enum ProjectCategory {
    """
    示例项目
    """
    DEMO
```

```
    """
    生产项目
    """
    PRODUCTION
}

"""
名称
为了达到某个产品迭代、产品模块开发、或者科研调研等目的所做的工作.
"""
type Project{
    id: String!
    name: String!
    description: String!
    category: ProjectCategory
    createTime: DateTime!
    milestones(
        status: MilestoneStatus
    ): [Milestone]
}
"""
里程碑
表述一个Project的某个时间阶段及阶段性目标. 一个Project可以同时拥有多个处于相同或者不同阶段
Milestone.
"""
type Milestone{
    id: String!
    name: String!
    status: MilestoneStatus
}

type User {
    name: String!
}

"""
里程碑状态
"""
enum MilestoneStatus{
    """
    未开始
    """
    NOT_STARTED,
    """
    进行中
    """
    DOING,
    """
    发布
    """
    RELEASE,
    """
    关闭
    """
```

```
    CLOSE
}

input ProjectInput{
    name: String!
    description: String!
    vendorBranches: [String!]!
    category: ProjectCategory!
}


input MilestoneInput{
    projectId: String!
    name: String!
}
```

java 领域

```
@Slf4j
@DataFetcherService
@DgraphType("PROJECT")
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Project {

    private static final String RELATIONSHIP_HAS = "has";

    @Getter(AccessLevel.NONE)
    @Setter(AccessLevel.NONE)
    @Autowired
    private ProjectRepository projectRepository;
    @Getter(AccessLevel.NONE)
    @Setter(AccessLevel.NONE)
    @Autowired
    private MilestoneRepository milestoneRepository;


    @UidField
    private String id;
    private String name;
    private String description;
    private ProjectCategory category;
    private OffsetDateTime createTime;
    @RelationshipField(RELATIONSHIP_HAS)
    private List<Milestone> milestoneList;

    @GraphqlMutation
    public DataFetcher<Project> createProject() {
        return dataFetchingEnvironment -> {
            ProjectInput input = GraphqlPayloadUtil.resolveArguments(dataFetchingEnvironment.
etArguments(), ProjectInput.class);
            OffsetDateTime now = OffsetDateTime.now();
```

```
            this.name = input.getName();
            this.description = input.getDescription();
            this.category = input.getCategory();
            this.createTime = now;

            this.id = projectRepository.save(this);
            return this;
        };
    }

    @GraphqlMutation
    public DataFetcher<Milestone> createMilestone() {
        return dataFetchingEnvironment -> {
            MilestoneInput input = GraphqlPayloadUtil.resolveArguments(dataFetchingEnvironm
nt.getArguments(), MilestoneInput.class);
//可以正向通过project创建milestone，也可以先创建milestone，再关联project
//            this.id = input.getProjectId();
//            this.milestoneList = Collections.singletonList(new Milestone(input.getName()));
//            this.id = projectRepository.save(this);

            Milestone milestone = new Milestone(input.getName());
            milestone.setStatus(MilestoneStatus.NOT_STARTED);
            String milestoneId = milestoneRepository.save(milestone);
            milestone.setId(milestoneId);

            milestoneRepository.createRelationship(RelationshipInformation.builder()
                    .sourceList(Collections.singletonList(input.getProjectId()))
                    .relationship(RELATIONSHIP_HAS)
                    .targetList(Collections.singletonList(milestoneId))
                    .build());

            return milestone;
        };
    }


    @GraphqlQuery
    public DataFetcher<Project> project() {
        return dataFetchingEnvironment -> {
            String id = dataFetchingEnvironment.getArgument("id");
            return projectRepository.getOne(id)
                    .orElse(null);
        };
    }

    @GraphqlQuery(type = "Project")
    public DataFetcher<List<Milestone>> milestones() {
        return dataFetchingEnvironment -> {
            Project project = dataFetchingEnvironment.getSource();
            List<Milestone> milestones = milestoneRepository.listByProjectId(project.id);
            String status = dataFetchingEnvironment.getArgument("status");
            if (StringUtils.isEmpty(status)) {
                return milestones;
```

```
        } else {
            MilestoneStatus milestoneStatus = MilestoneStatus.valueOf(status);
            return milestones.stream()
                    .filter(m -> m.getStatus().equals(milestoneStatus))
                    .collect(Collectors.toList());
        }
    };
}

// 只是为了展示合并请求
@GraphqlQuery
public DataFetcher<List<User>> users() {
    return dataFetchingEnvironment -> Arrays.asList(
            User.builder().name("u1").build(),
            User.builder().name("u2").build(),
            User.builder().name("u3").build()
    );
}


@Consumer(topic = "users")
public void usersListener(Message<DomainEvent> record) {
    log.info("listen users event: {}", record);
}

}
```

repository

```
@Repository
public class MilestoneRepository extends SimpleDgraphRepository<Milestone> {

    public List<Milestone> listByProjectId(String projectId) {
        Map<String, String> vars = new HashMap<>();
        vars.put("projectId", projectId);
        return this.queryForList("milestone.listByProjectId", vars);
    }

}
```

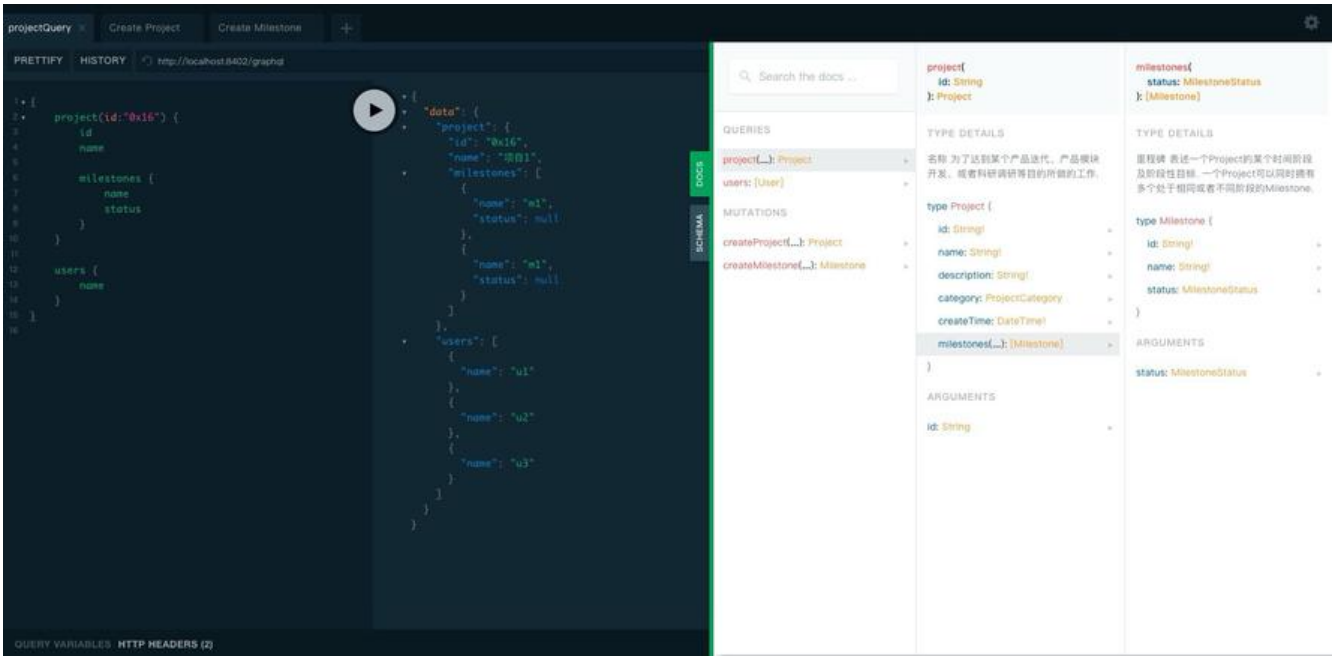
MilestoneDgraph.xml

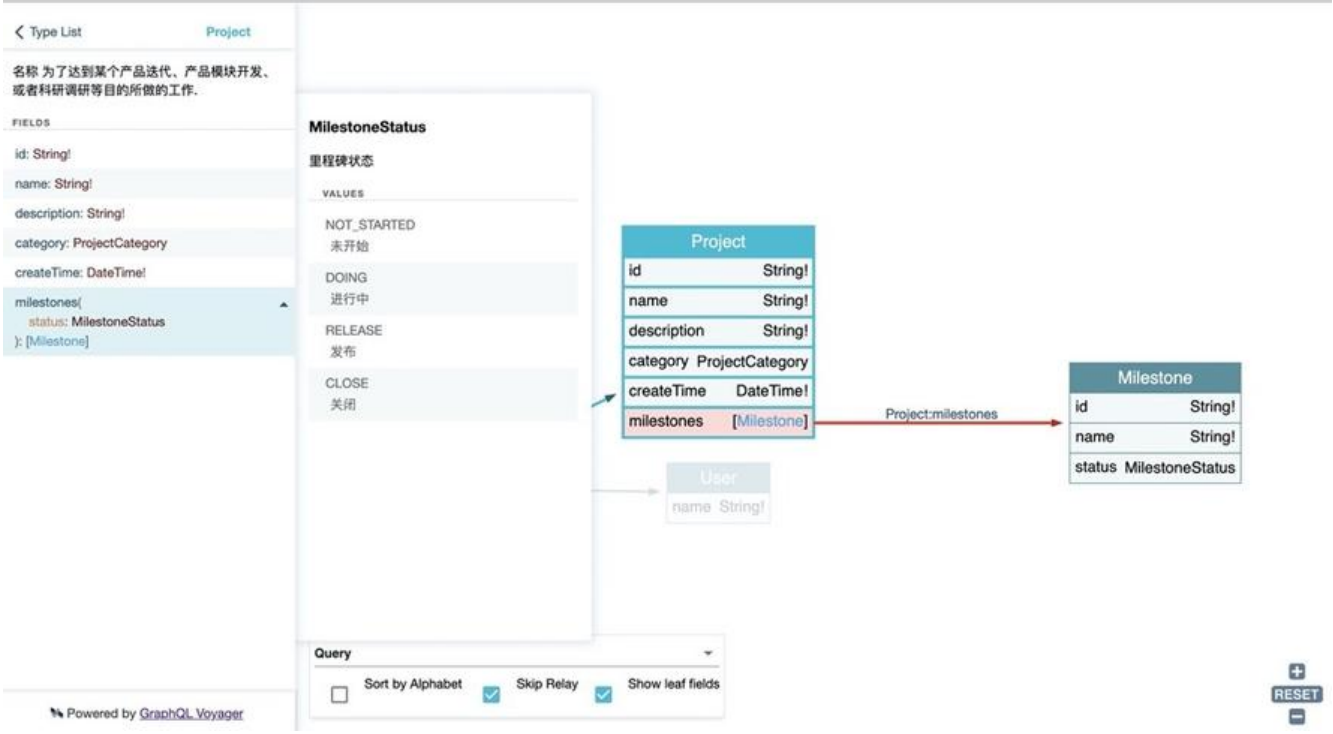
```
<dgraph>
    <var id="type">
        MILESTONE
    </var>
    <var id="common">
        uid
        expand(MILESTONE)
    </var>
    <query id="listByProjectId">
        query listByProjectId($projectId: string) {
            var(func:uid($projectId)) {
                has {
```

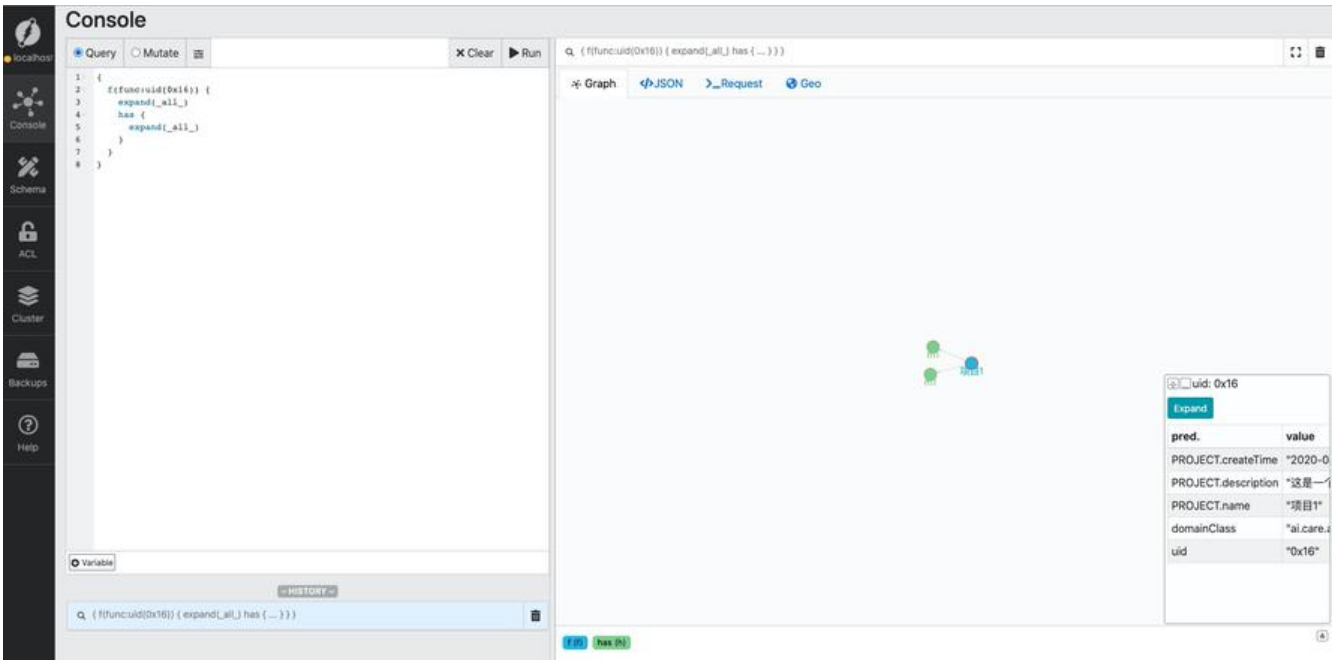
```
            var mids as uid
        }
    }
    listByProjectId(func:uid(mids)) {
        uid
        expand(_all_)
    }
  }
</query>
<mutation id="updateStatus">
  <![CDATA[
    <$id> <MILESTONE.status> "$status" .
  ]]>
</mutation>
</dgraph>
```

5.3 效果

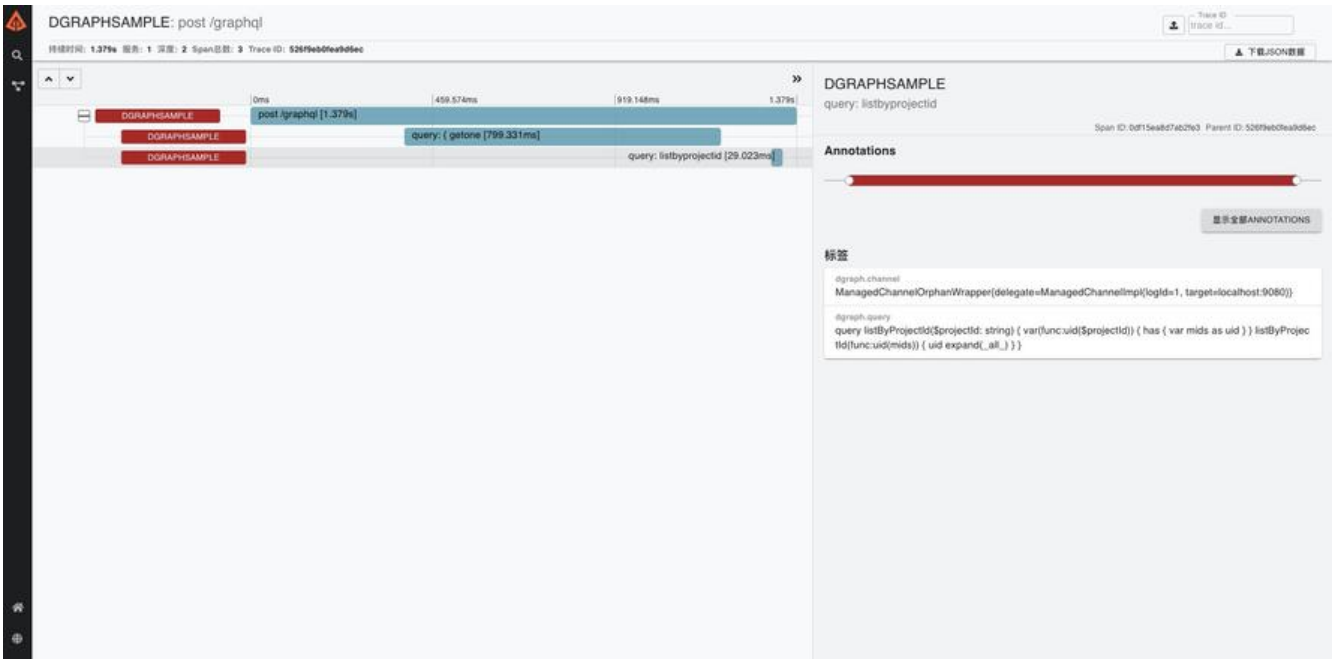
localhost:8080/playground







原文链接：DDD 探险——基于 GraphQL+Dgraph 实践



5.4 后续计划

1. 通过schema自动生成相关代码
2. 解析GraphQL，动态生成Dgraph查询

6. 弊端

1. 领域间的交互必须通过GraphQL，client封装不够优雅
2. Dgraph不擅长统计计算，比如统计类通过基于AOP+MQ通过mysql存储相关指标
3. 前后端技术体系同时变更

7. 相关工具推荐

7.1 工具

好用的工具已经集成到Arc中，而在schema确定时候的实现成本又很低，导致编写Mock的意义不大但仍列出部分开源代码，以作参考

1. **API Mock**
    1. graphql-faker
    2. graphql-tools
2. **SDL**
    1. graphql-editor
    2. voyager
3. **IDE**
    1. graphql-playground

原文链接：DDD 探险——基于 GraphQL+Dgraph 实践

2. graphiql
    3. graphqurl

7.2 最佳实践

1. 规范　https://spec.graphql.org/
2. 教程　https://www.howtographql.com/
3. 开放api示例　https://github.com/APIs-guru/graphql-apis
4. https://dgraph.io/tour