链滴

如何用 GraphQL+JavaPoet 把前端逼成瓶颈 (AKA: 如何炒了自己)

作者：crick77

原文链接：https://ld246.com/article/1608478325071

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

上篇文章中提到《DDD 探险——基于 GraphQL+Dgraph 实践》了一种假设

如果前端能通过领域模型进行数据操作，能否通过 GraphQL Schema同时描述领域模型、API接口以数据库结构?

基于此假设提供了 Arc框架，同时采用 Dgraph作为数据存储。

既然我们已经有了**Schema**，能否基于这个描述自动生成系统实现？通过降低开发成本、提升效率，成快速迭代、试错**领域模型**的目的。

尝试通过**代码生成器**来完成这个假设。

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

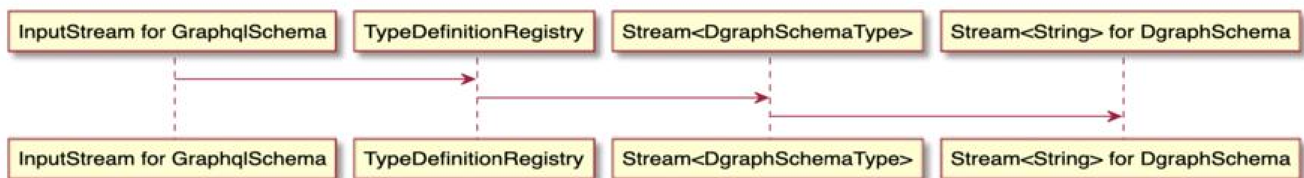
1. GraphQL Schema -> Dgraphql Schema

1.1 概念:

1. **Dgraph Schema:**Dgraph数据库结构化语句，类似mysql中的**DDL**
2. **predicate:**Dgraph的数据库字段，一个predicate可以被多个type使用
3. **domainClass:**Arc框架中定义javaBean类型，用于反序列化

1.2 方案

解析 GraphQL Schema后，根据转换逻辑生成 Dgraph Schema



注意 Arc框架限制

1. DB结构中拥有框架依赖通用 predicate，如domainClass
2. 为了解决 predicate跨type定义问题，dgraph中的predicate命名增加type为**前缀**

1.3 注意

1. 需要框架提供根据 DgraphSchema自动初始化数据库的能力

2. GraphQL Schema -> JavaCode

2.1 JavaPoet

JavaPoet is a Java API for generating .java source files.

通过 JavaPoet可以方便、详细的描述**Java源文件**并生成。

看官方issue发现很多人希望提供快速生成 getter、setter方法的方式，官方并未采纳，给出的回复是

Basically we're a tool for generating exactly what you tell us and not a tool for inferring code o generate. You can write a static method or helper class which can produce a field, getter, a d setting in one shot onto a TypeSpec.Builder.

这个回复也明确了 JavaPoet的定位。其实生成相关方法的方式非常简单

```
public class FieldSpecGenSetter implements Function<FieldSpec, MethodSpec> {

    @Override
    public MethodSpec apply(FieldSpec fieldSpec) {
        return MethodSpec.methodBuilder("set" + StringUtils.capitalize(fieldSpec.name))
                .addModifiers(Modifier.PUBLIC)
                .addParameter(fieldSpec.type, fieldSpec.name)
                .addStatement("this.$L = $L", fieldSpec.name, fieldSpec.name)
                .build();
    }

}

public class FieldSpecGenGetter implements Function<FieldSpec, MethodSpec> {

    @Override
    public MethodSpec apply(FieldSpec fieldSpec) {
        return MethodSpec.methodBuilder(GenSpecUtil.getGetterPrefix(fieldSpec) + StringUtils.c
pitalize(fieldSpec.name))
                .addModifiers(Modifier.PUBLIC)
                .addStatement("return $L", fieldSpec.name)
                .returns(fieldSpec.type)
                .build();
    }

}
```

提供一个生成 Builder的示例，基本涵盖了全部定义

```
public class TypeSpecGenBuilder implements UnaryOperator<TypeSpec> {

    @Override
    public TypeSpec apply(TypeSpec source) {
        if (CollectionUtils.isEmpty(source.fieldSpecs)) {
            return source;
        } else {
            TypeSpec target = TypeSpec.classBuilder(source.name + "Builder").build();
            return source.toBuilder()
                    .addMethod(getAllArgsConstructor(source.fieldSpecs))
                    .addMethod(getNoArgsConstructor())
                    .addMethod(getBuilderInstanceMethod(target))
                    .addType(getBuilderType(source, target))
                    .build();
        }
    }

    private MethodSpec getAllArgsConstructor(List<FieldSpec> fieldSpecs) {
```

```
        MethodSpec.Builder builder = MethodSpec.constructorBuilder()
                .addModifiers(Modifier.PUBLIC)
                .addParameters(fieldSpecs.stream().map(f -> ParameterSpec.builder(f.type, f.name).
uild()).collect(Collectors.toList()));
        fieldSpecs.forEach(fieldSpec -> builder.addStatement("this.$L = $L", fieldSpec.name, fiel
Spec.name));
        return builder.build();
    }

    private MethodSpec getNoArgsConstructor() {
        return MethodSpec.constructorBuilder().addModifiers(Modifier.PUBLIC).build();
    }

    private MethodSpec getBuilderInstanceMethod(TypeSpec target) {
        return MethodSpec.methodBuilder("builder")
                .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
                .returns(getType(target))
                .addStatement("return new $T()", getType(target))
                .build();
    }

    private TypeSpec getBuilderType(TypeSpec source, TypeSpec target) {
        return target.toBuilder()
                .addFields(source.fieldSpecs.stream().map(it -> FieldSpec.builder(it.type, it.name, M
difier.PRIVATE).build()).collect(Collectors.toList()))
                .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
                .addMethod(MethodSpec.constructorBuilder().addModifiers(Modifier.PRIVATE).buil
())
                .addMethods(getBuilderSetMethodSpecList(source, target))
                .addMethod(getSourceInstanceMethod(source))
                .build();
    }

    private List<MethodSpec> getBuilderSetMethodSpecList(TypeSpec source, TypeSpec targe
) {
        return source.fieldSpecs.stream()
                .map(fieldSpec -> MethodSpec.methodBuilder(fieldSpec.name)
                        .addModifiers(Modifier.PUBLIC)
                        .returns(getType(target))
                        .addParameter(ParameterSpec.builder(fieldSpec.type, fieldSpec.name).build())
                        .addStatement("this.$L = $L", fieldSpec.name, fieldSpec.name)
                        .addStatement("return this")
                        .build()).collect(Collectors.toList());
    }

    private MethodSpec getSourceInstanceMethod(TypeSpec source) {
        return MethodSpec.methodBuilder("build")
                .addStatement("return new $L($L)", source.name, source.fieldSpecs.stream().map(f -
 f.name).collect(Collectors.joining(",")))
                .addModifiers(Modifier.PUBLIC)
                .returns(ClassName.get("", source.name))
                .build();
    }
```

```
    private TypeName getType(TypeSpec typeSpec) {
        return ClassName.get("", typeSpec.name);
    }

}
```

2.2 实现

基于 graphql-java 解析 GraphqlSchema，通过 javapoet 按照 Arc约束生成相关**java文件**

- Graphql Enum ->dictionary
- Graphql input ->input
- Graphql type ->type、api、repository

2.3 注意

1. 通过只提供 interface方式，避免修改生成的代码。保障每次**schema**变更后都可以重新生成。
2. 如果存在需要修改生成代码的场景，通过 **配置**方式跳过相关**Java源文件**生成。

3. Maven Plugin

生成逻辑通过上述定义完成后，如何触发生成的动作？**Maven插件**是一个不错的选择。

只需要继承 AbstractMojo在 Override execute()中调用相关生成方法即可。这里暂不展开**如何开发aven插件**。

```
@Mojo(name = "generate")
public class GeneratorMojo extends AbstractMojo {

    @Parameter(defaultValue = "${project}", readonly = true, required = true)
    private MavenProject project;

    @Parameter(defaultValue = "${basedir}/src/main/resources/arc-generator.json")
    private File configJson;
    @Parameter(defaultValue = "${basedir}/src/main/resources/graphql/schema.graphqls")
    private File schemaPath;
    @Parameter(property = "target", defaultValue = "all")
    private String target;

    private static final String TARGET_ALL = "all";
    private static final String TARGET_JAVA = "java";
    private static final String TARGET_DGRAPH = "dgraph";

    @Override
    public void execute() throws MojoExecutionException, MojoFailureException {
        final CodeGenConfig config = getConfig();
        if (TARGET_ALL.equalsIgnoreCase(target) || TARGET_JAVA.equalsIgnoreCase(target)) {
            generateJava(config);
        }
        if (TARGET_ALL.equalsIgnoreCase(target) || TARGET_DGRAPH.equalsIgnoreCase(target)) {
            generateDgraph(config);
```

```
        }
    }
}
```

然后再命令行执行

```
mvn arc:generate
```

即可按照配置及规则生成相关代码。也可以通过参数决定只生成**java代码**，不生成 Dgraph Schema

```
mvn arc:generate -Dtarget=java
```

4. 效果

4.0 添加Maven插件依赖

```
<build>
    <plugins>
        <plugin>
            <groupId>com.github.yituhealthcare</groupId>
            <artifactId>arc-maven-plugin</artifactId>
            <version>1.4.0</version>
        </plugin>
    </plugins>
</build>
```

4.1 创建schema. 默认路径为 resources:graphql/schema.graphqs

```
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!
}
```

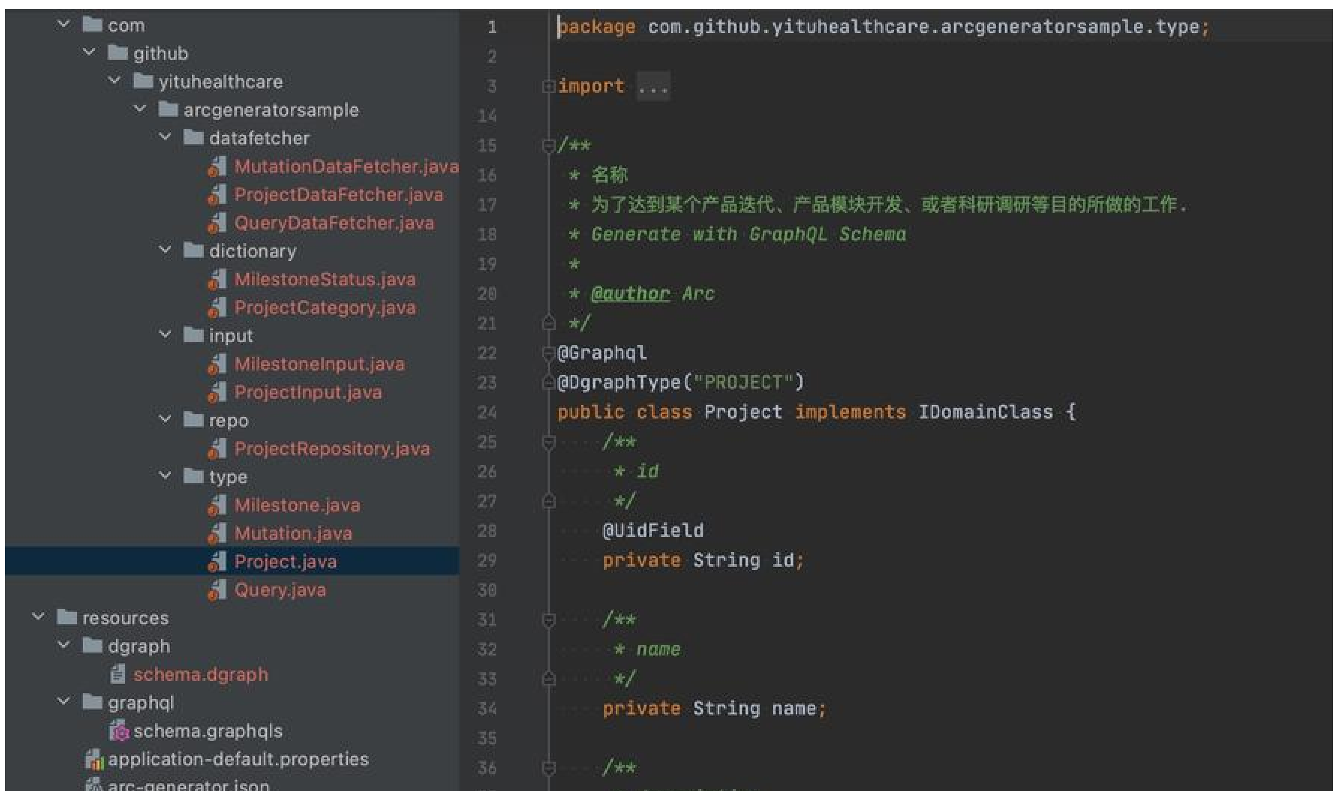
4.2 新建配置文件， 默认路径为 resources:arc-generator.json

```
{
  "basePackage": "com.github.yituhealthcare.arcgeneratorsample",
  "dropAll": false,
  "genStrategies": [
    {
      "codeGenOperation": "SKIP_IF_EXISTED",
      "codeGenType": "REPO"
    },
    {
      "codeGenOperation": "OVERRIDE",
      "codeGenType": "API"
    }
  ],
  "ignoreJavaFileNames": [
    "User"
  ],
  "dgraphPath": "dgraph/schema.dgraph"
}
```

4.3 执行命令行

```
$ mvn arc:generate
[INFO] Scanning for projects...
[INFO]
[INFO] -----------< com.github.yituhealthcare:arc-generator-sample >-----------
[INFO] Building arc-generator-sample 1.4.0
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- arc-maven-plugin:1.4.0:generate (default-cli) @ arc-generator-sample ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.922 s
[INFO] Finished at: 2020-12-20T20:04:38+08:00
[INFO] ------------------------------------------------------------------------
```

4.4 代码生成



在聊清楚schema后，只需要执行一行命令，然后编写一个实现类即可完成接口服务的开发。在schem修改后，重复执行这个过程。

5. 缺陷

5.1 GraphQL描述力不够，只描述来数据结构与类型，没有描述数存储可能会用到的主键、索引、缓存等信息，以及DDD中的限界上文(BoundedContext)、聚合根(AggregateRoot)等概念

原文链接：如何用 GraphQL+JavaPoet 把前端逼成瓶颈 (AKA: 如何炒了自己)

可以通过自定义directive实现相关扩展定义，比如:

```
scalar DateTime
directive @Cache on FIELD_DEFINITION
directive @Alloftext on FIELD_DEFINITION
directive @AggregateRoot on OBJECT

type Project @AggregateRoot {
    id: String!
    name: String! @Cache @Alloftext
    description: String!
    createTime: DateTime!
}
```

但是这个时候 QL已经变成了 DSL，需要考虑额外的学习、推广成本

5.2 生成的代码和实现代码混合，如何进行code review?

schema和生成的代码先提交一次PR，对schema进行review。之后的实现代码再单独提交PR

5.3 Dgraph更新时数据如何处理?

已经上线后无法自动迭代。数据版本迁移不应属于开发过程

5.4 仍需编写实现类(生成的代码是否允许编辑？)

理想情况是完全通过代码生成完成相关实现。但是实际业务并不是简单的crud，如果通过schema描复杂业务，就会发现本质上是通过schema来编写另一种java代码。所以只生成interface，需要开发补全相关实现。让schema专注于描述DDD。不排除提供生成简单crud实现的功能，在修改了相关实类后，再通过配置 SKIP_IF_EXISTED跳过相关实现

5.5 自动化生成后发现无事可做，不可替代性如何保障?

-_-#