浅析Spring,框架设计与应用 讲义

报告流程

1.目的:

从Spring框架的IOC,DI,SpringBoot的自动装配原理产生原因,作用,实现原理入手,浅析框架的简化开发作用,以及框架的设计思路.

2.流程:

2.1 引入:

使用框架的感受?(SpringBoot框架)

方便快捷.快速开发,调试,部署.

编程语言从汇编语言到高级语言,程序员开发的工具越来越简单.而现在的主流开发,尤其是后端开发,越来越依赖于框架.框架的作用有很多,其核心目的就是简化开发.使编程人员能够更加专注于业务逻辑而最大程度上忽略其他的东西.

如何简化开发?不同的语言的框架,有很多不同的实现方式.接下来我们主要关注Java语言.

Java语言的核心是什么?

面向对象程序设计,一切皆对象,在所有的程序中都会出现的一点就是类和对象的管理.同时这也是java程序所面临的最大的问题.

对象管理包括:

1.对象的产生

2.对象的使用

3.对象的销毁

我们先来看对象的销毁.

首先,回答一下c语言和c++的对象的销毁方式,

是由程序主动关闭.

而Java程序对象是如何销毁的?是由JVM自动进行垃圾回收的.这就是简化开发的角度上java比c和c++先进的地方,但是现在,对象的销毁问题解决了,对象的创建和使用问题怎么办呢?这是单纯地Java所不能解决的.

2.2 DI:

先来看两段代码:

```
Class Car(){

    private Bottom bottom;

    public Car(){
    this.bottom = new Bottom();
    }

}
Class Bottom(){

    int a;

    public Bottom(){
    this.a = 1;
    }

}
```

```
Class Car(){

    private Bottom bottom;

    public Car(Bottom bottom){
    this.bottom = Bottom bottom;
    }

}
Class Bottom(){
    int a;

    public Bottom(){
    this.a = 1;
    }

}
```

这两种写法在程序的调用时候略有不同，第一种是在主程序中直接new一个car（）对象就完成了。而第二种则需要先new一个bottom对象，然后再new一个car对象。这样看来，似乎第一种写法更加简练一些，可是，事实真的是如此吗?

让我们考虑一种情况：

当bottom发生修改时，比如a的值通过传入一个变量来确定的时候，这样对于第一种写法，所有的构造函数全部都需要修改，而对于第二种方式则只需要修改一个构造函数即可。

对于第一种设计来说，**这样的设计基本是不可维护的**，因为在实际工程中，有些类会有几千个底层，如果要一一修改，所耗费的成本太大了。

而且，在第一种写法中，创建car对象必然会创建一个新的bottom对象，纵使会被销毁，但是也在一定程度上增加了开销。

所以现在我们知道，采用第二种写法是比较合理的，即：

所有的new对象的过程都写在程序运行时，而不是再类的构造方法中。构造方法中统一更改为传入所需要的类的对象这种格式。

这就是依赖注入（Dependency Injection）。

依赖注入的主谓宾补充完整，就是将调用者所依赖的类实例对象注入到调用者类。而在这个例子中，car依赖于bottom，car开放了一个接口（这里其实是构造方法）使它的依赖bottom对象可以注入到自己的对象中来。提高了系统的可维护性和可扩展性。

2.3 IOC：

好了，在有了上面的基础之后，我们再来看我们的程序，现在它已经变成了大概这样：

```
    public static void main(String[] args) {
        A a  = new A();
        //a发挥作用
        B b = new B();
        //b发挥作用
        C c = new C():
        //c发挥作用
        ......
        A a  = new A();
        //a发挥作用
    }
```

但是这样，仍然会出现很多问题.

如果a是某个接口的实现类，现在需要修改a的实现类，换成另一个类，这样就需要修改所有的new A的方法，这样仍然是十分复杂而不可维护的。

那么，能不能有一种机制来解决这个问题呢？这就体现出框架的作用了。

框架可以帮助我们创建对象.

只需要我们通过一些语法告诉框架我们需要什么样的对象，它的名字是什么，它的类在什么地方，程序就会自动的帮我们创建对象，我们只需要在需要这个对象的时候问框架要这个对象就可以了. 来看一个例子：

```
package com.spring.demo;

public class Person {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="person" class="com.spring.demo.Person">
        <property name="name" value="zje"></property>
        <property name="age" value="24"></property>
    </bean>

</beans>
```

```
package com.spring.demo;

import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class TextIoc {
    @Test
    public void textUser()
    {
        //1.获取spring配置文件
         ApplicationContext context = new
ClassPathXmlApplicationContext("Bean.xml");
         //2.由配置文件返回对象
         Person p = (Person)context.getBean("person");
         System.out.println(p);
         p.info();
    }
}
```

这种机制主要由三个组成:

1.框架读取配置文件(也有可能是某些注解),获取类的有关信息.

2.根据类文件生成对象(bean)

3.在程序中通过(bean)方法获取对象.

这样在发生修改时,只需要修改配置文件中的相关信息,就可以实现修改,而不需要修改代码中的所有new 语句.

在这种框架下,我们再也不需要手动的创建对象,而是由框架帮我们创建对象,换句话说,我们把对象的控制权完全的交给了程序(框架)(以前是销毁权利,现在是全部的权利).

这就是控制反转（Inversion of Control）.同样也是轻量级的Spring框架的核心。

不同的教程和文章对这两个词有很多种不同的解读,我认为,这样解读是最通俗易懂和符合常理的一种方式.

接下来我们主要关注Java语言,Java语言的核心是面向对象设计,在Java中,一切皆对象,在所有的程序中都会出现的一点就是类和对象的管理.同时这也是java程序所面临的最大的问题.

这样，我们就把java的对象创建和使用的问题，变成了bean的创建和获取的问题。

2.3.1 简单的手写一个IOC容器:

容器类:

```
package com;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.util.HashMap;
import java.util.Map;

public class IOCContainer {
    private Map<String, Object> beans; // 存储对象实例的 Map

    //初始化
    public IOCContainer() {
        beans = new HashMap<>();
    }


    // 获取 bean 对象
    public Object getBean(String name) {
        return beans.get(name);
    }

    // 注册 bean 到容器中
    private Object registerBean(String name,Class<?> clazz, Map<String, Object>
properties) throws Exception {
        try {
            // 获取无参数的构造函数
            Constructor<?>[] constructors = clazz.getDeclaredConstructors();
            Constructor<?> constructor = null;
            for (Constructor<?> c : constructors) {
                if (c.getParameterCount() == 0) {
                    constructor = c;
```

```
                    break;
                }
            }

            if (constructor == null) {
                throw new RuntimeException("No default constructor found for bean: " + clazz);
            }

            //生成实例
            Object bean = constructor.newInstance();

            // 获取变量属性，进行注入
            for (Field field : clazz.getDeclaredFields()) {
                if (properties.containsKey(field.getName())) {
                    field.setAccessible(true);
                    field.set(bean, properties.get(field.getName()));
                }
            }

            //注册到bean容器中
            beans.put(name,bean);
            return bean;

        } catch (Exception e) {
            throw new RuntimeException("Failed to create bean: " + clazz, e);
        }
    }
    public static void main(String[] args) throws Exception {
        IOCContainer iocContainer = new IOCContainer();

        // 注册 test到容器中
        Map<String, Object> testProperties = new HashMap<>();
        testProperties.put("name", "张三");
        iocContainer.registerBean("test", Test.class, testProperties);

        // 获取 test对象并调用方法
        Test test = (Test) iocContainer.getBean("test");
        test.Hello();
    }
}
```

测试类：

```
package com;

public class Test {

    private String name;
    public void Hello(){
```

```
        System.*out*.println("Hello!"+this.name);
    }

    public  Test(){

    }

}
```

2.4 @Autowried:

我们先来看bean的获取。

现在,我们的程序已经变成了这样:

```
   public static void main(String[] args) {
        A a  = （A） context.getBean("A");
        //a发挥作用
        B b = (B) context.getBean("A");
        //b发挥作用
        C c = (C) context.getBean("A"):
        //c发挥作用
        ......
        A a  = (A) context.getBean("A");
        //a发挥作用
    }
```

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MyApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        MyBean myBean = (MyBean) context.getBean("myBean");
        myBean.doSomething();
    }
}
```

这样手动获取bean的方式仍然是非常繁琐和容易出错的，因为我们需要同时指定bean的类型和名称。

那么可不可以自动的获取bean呢？只要我创建一个变量就好了。

使用自动装配（`@Autowired`注解）的好处包括：

1. 简化代码：自动装配可以大大减少手动装配的代码，特别是在大型应用程序中，手动装配可能非常繁琐和易错。

2. 更加可读性：通过使用自动装配，代码可以更加清晰和易于理解。不再需要在代码中指定显式的依赖注入，而是通过注解告诉Spring容器应该注入哪些依赖项。
3. 更加灵活：使用自动装配可以使代码更加灵活，因为它可以在运行时自动解析依赖关系。这使得代码更加易于维护和测试，因为您不需要手动维护依赖关系。
4. 可扩展性：使用自动装配可以使代码更加易于扩展。例如，如果您需要添加新的组件或服务，您可以使用自动装配来自动解析它们的依赖关系，而不需要手动修改现有代码。
5. 更高的生产率：使用自动装配可以提高生产率，因为它可以使您的代码更加简洁、易于理解和易于维护。这意味着您可以更快地编写代码，并且更容易开发和维护复杂的应用程序。

这就是@Autowried注解，它的意思是自动注入，补充完整是自动把IOC中的bean注入到它所需要的地方去。请注意这里的自动注入和下文的自动装配，和上文的依赖注入都有所区别，注意区分。

```
@Autowired
UserScoreRecordRepository userScoreRecordRepository;

@PostMapping("/addstrip")
public Result addScore(@RequestParam("name") String name,
                       @RequestBody() List<AddStrip> addStrips) {
    return activityService.addScore(name, addStrips);
}

@PostMapping("/searchActivity")
@RequestParam("status") Integer status) {
        return activityService.searchScore(status);
}
```

2.5 自动装配：

现在看来，似乎一切都非常完美……除了各种恶心的配置.

现在我们的开发过程是怎么样的呢? 新建项目,引入依赖,编写各种xml配置文件或者JavaConfig类,然后开始代码开发.

这是一个实际组件的例子:

```
@Configuration
public class CorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        UrlBasedCorsConfigurationSource urlBasedCorsConfigurationSource = new
                UrlBasedCorsConfigurationSource();
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        corsConfiguration.addAllowedOrigin("*");
        corsConfiguration.setAllowCredentials(false);
```

```
        urlBasedCorsConfigurationSource.registerCorsConfiguration("/**",
corsConfiguration);
        CorsFilter corsFilter = new CorsFilter(urlBasedCorsConfigurationSource);
        return corsFilter;
    }
}
```

但是,在实际的开发中,我们需要依赖的第三方组件是非常多的,比如tommat,mybatis等等,每当我引入这些依赖的时候,由于这协议来本身还依赖于其他的一些bean,我就需要写一大堆的配置才能完整配置,然后进行开发.

但是在很多时候,我们可以发现,在大多数情况下,所写出来的配置文件的细节都是差不多的,这时候就不禁引起我们思考了:能不能自动的导入这些配置文件呢?

这就是自动装配.简单来说,就是在约定大于配置的基础上,在第三方组件引入的时候,自动的带上一份配置文件,然后由框架自动的加载这些配置文件,自动的注入IOC容器生成bean.

这样就不需要我们为大多数组件手动的配置编写配置文件了。

这里就有同学要问了，那我在spingboot中为那些组件写的@bean注解是什么？不是已经有自动装配了吗?

但是在一些特定的情况下,仍然需要我们手动的配置文件.

1. 需要使用自定义的第三方组件，而该组件的配置无法通过自动配置来完成。
2. 需要使用一些较为复杂的配置，比如多数据源配置、分布式事务配置等。
3. 需要对一些组件的配置进行细粒度的控制，比如缓存的 TTL、连接池的大小等。
4. 需要对一些组件进行自定义扩展，比如自定义错误处理、自定义消息转换等。

就像在上文的跨域配置中，因为需要配置的东西特别多，而且在不同的应用场景下不太一样，所以仍然需要自己手动配置。

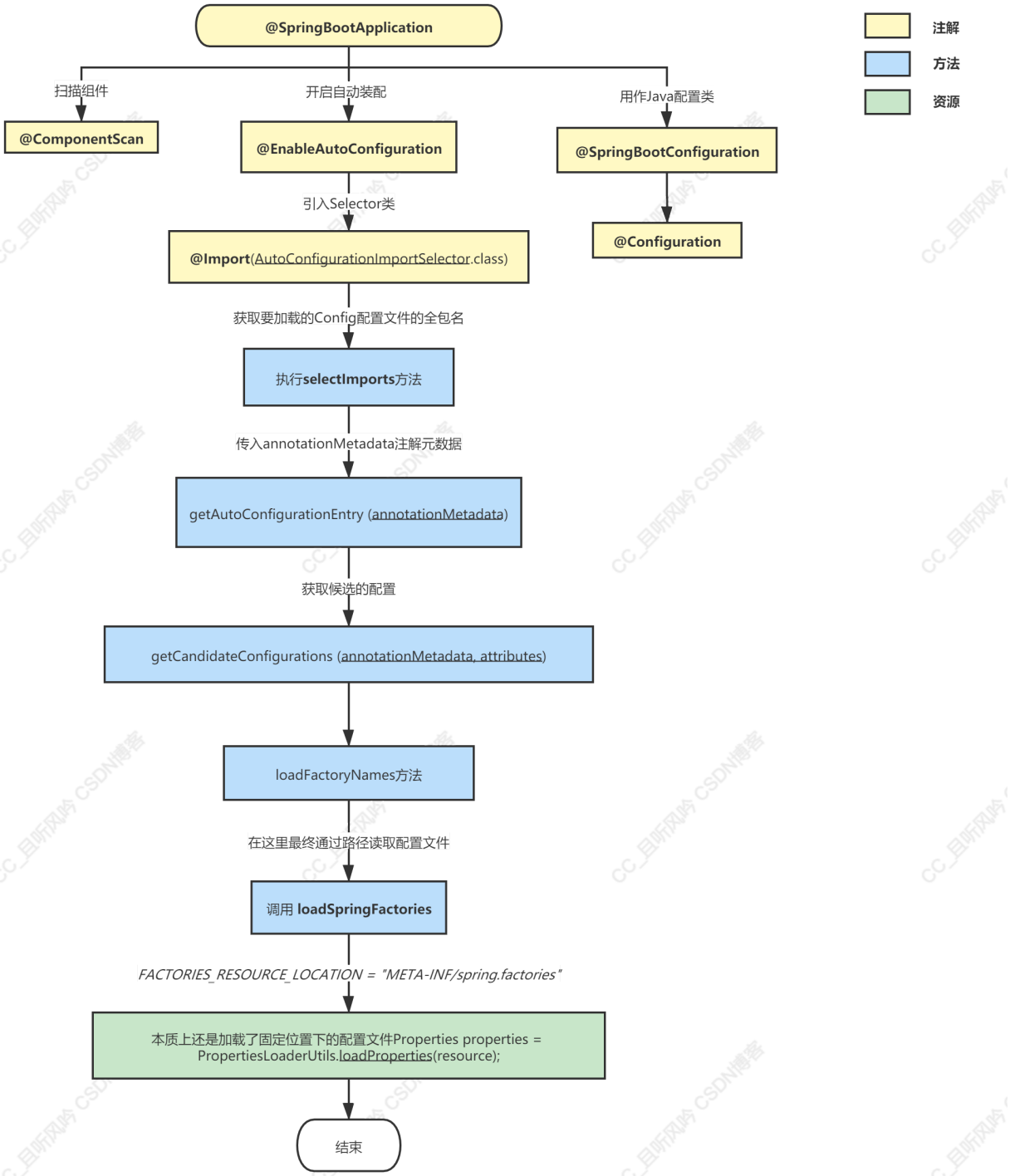
浅浅总结一下:

Spring Boot的自动装配和 `@Autowired` 注解都是基于Spring框架的IoC容器实现的，不同的是它们自动化的实现方式略有不同。

在Spring Boot中，自动装配的核心是自动读取并配置各种组件和依赖项。Spring Boot会自动扫描应用程序的类路径，查找并读取各种配置文件（例如 `application.properties` 或 `application.yml` ），并根据这些配置信息自动配置和初始化各种组件和依赖项。这样，我们就不需要手动配置和初始化每个组件和依赖项，从而简化了应用程序的开发和部署。

而 `@Autowired` 注解的自动则是指自动查找和注入符合类型要求的Bean对象。当我们在代码中使用 `@Autowired` 注解时，Spring IoC容器会自动查找并创建符合类型要求的Bean对象，并将它们注入到我们的类中。这样，我们就不需要手动创建和配置每个Bean对象，从而简化了依赖注入的实现。

总之，Spring Boot的自动装配和 `@Autowired` 注解都是基于Spring框架的IoC容器实现的，它们通过不同的自动化方式简化了应用程序的开发和部署。



2.6 代码生成：

(探讨部分)

现在，我们对框架的介绍已经进行到了实际开发的地步，日常开发中我们使用的spring boot基本上就是基于以上的原理和方法在进行简化开发的，但是框架的开发是没有尽头的，这不由得再次引发我们的思考，这就是最优秀的框架了吗?

事实上,虽然spring什么都能做,但是它最大的用处还是用来进行后端的那种crud开发,而这种开发一般都是遵循mvc模式,在这种情况下,有没有过这种疑惑?

来看一个service和controller:

```
   @Override
    public Result createActivity(String year, Integer type, String name, String
            explanation, String icon, String organization, String time, String site)
{
        try {
            if(activityRepository.findActivityByName(name)!=null){
                return Result.error(400,"活动名称已存在!");
            }
            Activity activity = new Activity();
            activity.setYear(year);
            activity.setActivityType(type);
            activity.setAname(name);
            activity.setExplanation(explanation);
            activity.setIcon(icon);
            activity.setOrganization(organization);
            activity.setTime(time);
            activity.setSite(site);
            activity.setDeleteStatus(1);
            activity.setStatus(0);
            SimpleDateFormat tempDate = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String datetime = tempDate.format(new java.util.Date());
            activity.setCreateTime(datetime);
            activityRepository.save(activity);
            return Result.success(200,"新建活动成功!");
        }catch (Exception e){
            return Result.error(500,"新建活动失败!");
        }
    }
```

```
    @PostMapping("/addactivity")
    @ApiOperation("创建活动")
    public Result createActivity(@ApiParam("活动年份形如2023春，2023秋")
@RequestParam("year") String year,
                                 @ApiParam("活动加分类型,1表示加日常行为分,2表示加个性发展
分,3表示加创新创业分,4表示加创新创业讲座学分,") @RequestParam("type") Integer type,
                                 @ApiParam("活动名称") @RequestParam("name") String
aname,
                                 @ApiParam("活动备注") @RequestParam("explanation")
String explanation,
                                 @ApiParam("活动图片的url") @RequestParam("icon")
String icon,
                                 @ApiParam("活动所属社团")
@RequestParam("organization") String organization,
                                 @ApiParam("活动进行时间") @RequestParam("time")
String time,
                                 @ApiParam("活动进行地点") @RequestParam("site")
String site){
        return
activityService.createActivity(year,type,aname,explanation,icon,organization,time,si
te);
    }
```

在这个例子中,controller其实只做了两件事,调用了service的方法,这一点是非常重复的,完全属于重复劳动.为方法所属的参数加上了注释.而这些个注释最后会生成接口文档.

有趣的是,在实际的项目开发中,有很多时候是先有接口文档,然后后端程序员根据接口文档去编写代码.那么在这些的基础上,有没有什么更好地编写mvc模式的crud的方式呢?没有多余的东西,只有业务逻辑

代码生成:根据某种接口定义,自动的生成出了业务逻辑值外的其他所有代码,而一个接口的业务逻辑被抽象成一个方法,程序员只需要关注这个方法即可.

下面展示的是hertz的代码生成:

接口定义:

```
struct RegisterRequest {
    1: string username
    2: string password
}
struct RegisterResponse {
    1: string code
    2: string msg
    3: i64 userid
    4: string token
}
```

生成代码:出了方法内部语句,其余都为框架自动生成.

```
```

```
func (s *UserImpl) Register(ctx context.Context, req *api.RegisterRequest) (resp
*api.RegisterResponse, err error) {
    //建立数据库连接，数据库名：数据库密码
    dsn := "root:123456@tcp(127.0.0.1:3306)/dbgotest"
    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{})
    //处理错误
    if err != nil {
        //控制台打印错误日志
        panic("数据库连接失败!")
    }

    var user User
    db.First(&user, "name = ?", req.Username)
    if db.RowsAffected != 0 {
        resp = &api.RegisterResponse{Code: "1", Msg: "用户名已经存在!"}
        return
    }

    //创建一条数据，传入一个对象
    db.Create(&User{Username: req.Username, Password: req.Password, FollowCount: 0,
FollowerCount: 0})
    resp = &api.RegisterResponse{Code: "0", Msg: "用户注册成功!", Token: req.Username}
    return

}
```

代码结构:

```
.
├── biz                             // business 层，存放业务逻辑相关流程
│   ├── handler                     // 存放 handler 文件
│   │   ├── hello                   // hello/example 对应 thrift idl 中定义的 namespace；而对于 protobuf idl，则是对应 go_package 的最后一级
│   │   │   └── example
│   │   │       ├── hello_service.go   // handler 文件，用户在该文件里实现 IDL service 定义的方法，update 时会查找 当前文件已有的 handler 在尾部追加新的
│   │   │       └── new_service.go     // 同上，idl 中定义的每一个 service 对应一个文件
│   │   └── ping.go                 // 默认携带的 ping handler，用于生成代码快速调试，无其他特殊含义
│   ├── model                       // IDL 内容相关的生成代码
│   │   └── hello                   // hello/example 对应 thrift idl 中定义的 namespace；而对于 protobuf idl，则是对应 go_package
│   │     └── example
│   │         └── hello.go          // thriftgo 的产物，包含 hello.thrift 定义的内容的 go 代码，update 时会重新生成
│   └── router                      // idl 中定义的路由相关生成代码
│       ├── hello                   // hello/example 对应 thrift idl 中定义的namespace；而对于 protobuf idl，则是对应 go_package 的最后一级
│       │   └── example
│       │       ├── hello.go        // hz 为 hello.thrift 中定义的路由生成的路由注册代码；每次 update 相关 idl 会重新生成该文件
│       │       └── middleware.go   // 默认中间件函数，hz 为每一个生成的路由组都默认加了一个中间件；update 时会查找当前文件已有的 middleware 在尾部追加
│       └── register.go             // 调用注册每一个 idl 文件中的路由定义；当有新的 idl 加入，在更新的时候会自动插入其路由注册的调用；勿动
├── go.mod                          // go.mod 文件，如不在命令行指定，则默认使用相对于GOPATH的相对路径作为 module 名
├── idl                             // 用户定义的idl，位置可任意
│   └── hello.thrift
├── main.go                         // 程序入口
├── router.go                       // 用户自定义除 idl 外的路由方法
└── router_gen.go                   // hz 生成的路由注册代码，用于调用用户自定义的路由以及 hz 生成的路由
```

这未必是最好的实践方式，只能说，框架的开发和使用是永无止境的，没有最好的框架，只有最合适的框架。

3.参考资料：

1.《Inversion of Control Containers and the Dependency Injection pattern》

IOC领域的元老级文章，首次提出了DI的确切概念。

2.掘金：

https://juejin.cn/post/6844904161775976456#heading-7

https://juejin.cn/post/7215507413779611707

https://juejin.cn/post/6844903793637720071

https://juejin.cn/post/7162568709955911717

3.CSDN:

https://blog.csdn.net/weixin_54514751/article/details/126055383?ops_request_misc=&request_id=&biz_id=102&utm_term=springioc&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-7-126055383.nonecase&spm=1018.2226.3001.4187

https://blog.csdn.net/weixin_52731337/article/details/119170446

https://blog.csdn.net/weixin_43826242/article/details/106005176?spm=1001.2014.3001.5506

https://blog.csdn.net/qq_24313635/article/details/109431324