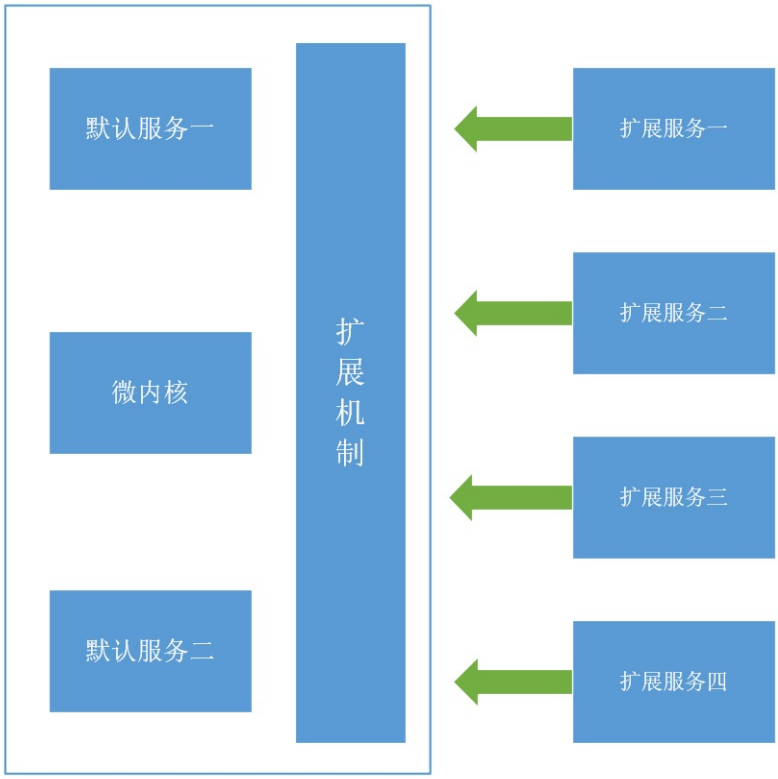
微内核架构浅析与实践

微内核架构(Microkernel Architecture)，也被称为插件化架构(Plug-in Architecuture)，是一种面向功能进行拆分的可扩展性架构。该架构通常用于实现基于产品（存在多个版本，需要下载安装才能使用，与Web-based相对应）的应用，例如Unix操作系统、Eclipse IDE、Maven、移动App客户端软件等，也有一些企业将业务系统设计成微内核架构，例如保险公司的保险核算逻辑系统，不同保险品种可以将逻辑封装成插件。

1. 基本架构与概念

微内核架构是一种设计范型（Design paradigm），其定义如下：

> The microkernel architecture pattern allows you to add additional application features as plug-ins to the core application, providing extensibility as well as feature separation and isolation.



微内核架构有两大模块组成: 核心系统与插件模块，核心系统负责和业务功能无关的通用功能,例如模块加载、模块间通信等; 插件模块负责实现具体的业务逻辑。微内核架构的本质是将变化封装在插件中，从而达到快速灵活扩展的目的，而又不影响整体系统的稳定。设计一个微内核体系的关键工作集中于核心系统的构建，插件的组装过程应基于统一的规则，比如基于setter注入，不应该对不同的插件进行硬编码组装，确保没有任何功能在内核中硬编码。微内核的核心系统设计的关键技术分为: 插件管理、插件链接和插件通信三个部分:

- **插件管理**
  包括插件的注册、插件的加载和插件的发现等，常用的实现方法是插件注册表机制，在核心系统中提供注册表（实现形式有配置文件、数据库甚至是代码），在插件注册表中保存插件的信息，包括名称、位置及加载时机等

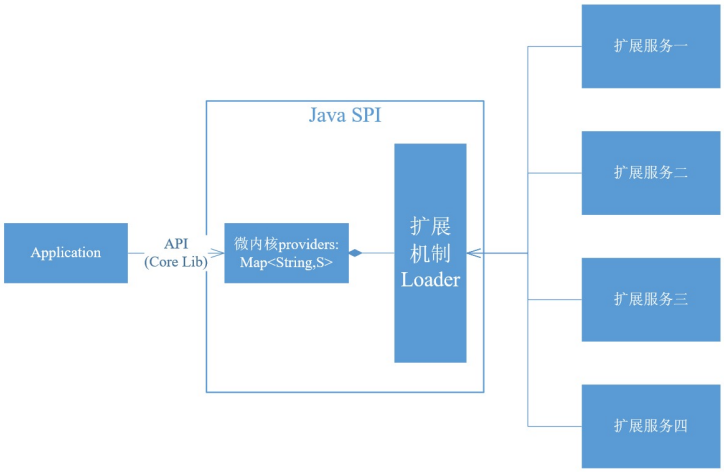
- **插件连接**
  插件如何连接到核心系统，通常来说在核心系统中指定连接规范，插件按照规范来实现，核心系统按照规范加载即可。常见的连接机制有OSGI、消息模式、依赖注入，甚至可以使用RPC、HTTP等分布式通信协议。

- **插件通信**
  核心系统提供插件通信机制，插件间的通信通过核心系统来完成，这种情况类似于计算机的cpu、硬盘、内存等配置通过主板的总线来完成通信

目前Spring, OSGI, JMX, ServiceLoader等都是常见的微核容器(Core System)，它们负责基于统一规则的组装，但不带功能逻辑，下面介绍这几种核心系统的实现

2 Java SPI

SPI全称为Service Provider Interface，是Java提供的一套用来被第三方实现或者扩展的API，可以用来启动框架扩展和替换组件，其基本原理如下图所示：



Java SPI实际上是基于接口的编程、策略模式及配置方法组合实现的动态加载机制，提供了通过接口寻找其实现的方法，类似于IOC的思想，将装配的控制权交给程序之外，从而实现解耦

2.1.1 使用入门

SPI是调用方制定的接口制定的接口规范，提供给外部来实现，调用方在调用时则选择自己需要的外部实现，从使用人员来看，SPI被框架扩展人员使用，其类似于IOC的思想，框架为某个接口寻找服务实现的机制。SPI的基本使用步骤如下：

> a.定义一组接口
> b.为接口实现一个或者多个实现
> c.在META-INF/services目录中，新增以接口命名的文件，内容是要应用的实现类
> d.使用ServiceLoader来加载配置文件中指定的实现

SPI的基本应用场景是可替换的插件机制，比如JDBC数据库驱动包、Spring框架及Dubbo等均通过SPI的方式实现了框架的扩展。

1) 定义接口及实现类

```
public interface IShout {
  void shout();
}

public class Cat implements IShout {
  @Override
  public void shout() {
    System.out.println("Cat MiaoMiao");
  }
}

public class Dog implements IShout {
    @Override
    public void shout() {
        System.out.println("Dog WangWang");
    }
}
```

定义了IShout接口，抽象方法为shout

2) 配置SPI装载 在META-INF.services目录下创建接口对应的文件：com.fys.javaspi.IShout，其内容如下：

> com.fys.javaspi.impl.Cat
> com.fys.javaspi.impl.Dog

3) 测试类及输出

```
public class SPIExamples {
    public static void main(String[] args) {
        ServiceLoader<IShout> shouts = ServiceLoader.load(IShout.class);
        for (IShout s : shouts) {
            s.shout();
        }
    }
}
```

执行后，在控制台输出如下:

> Cat Miao Miao
> Dog WangWang

2.1.2 原理浅析

Java SPI机制实现了接口与实现的解耦，第三方的服务模块的装配控制逻辑与调用者的业务代码相分离，而不是耦合在一起，从而使得应用程序可以根据实际的业务情况启用框架扩展或者替换框架中已有的组件。

```
ServiceLoader#
private S nextService() {
```

```
    String cn = nextName;
    nextName = null;
    Class<?> c = null;
    try {
        c = Class.forName(cn, false, loader);
        S p = service.cast(c.newInstance());
        providers.put(cn, p);
        return p;
     }  ......
}
```

程序通过META-INF/services中的配置，加载接口的实现插件（服务实现），实例化后存放在providers中

- 插件管理
  插件即服务的实现，存放在ServiceLoader#providers中，其核心为Map容器（Key为类名，value为插件对象）

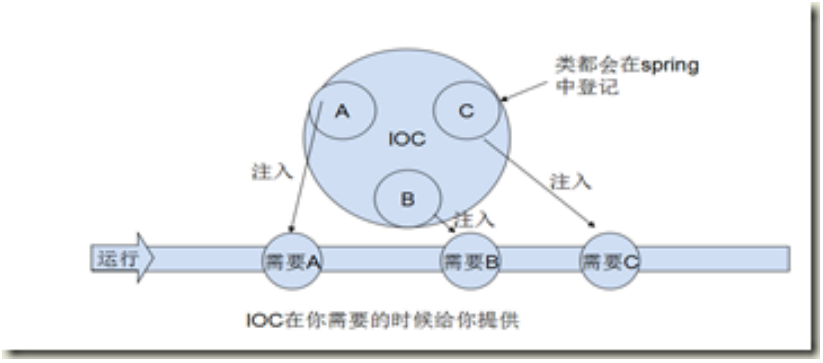
```
 // Cached providers, in instantiation order
 private LinkedHashMap<String,S> providers = new LinkedHashMap<>();
```

- 插件连接
  用户使用SPI中的插件，只需要根据插件名称去获取即可，以Apache Dubbo为例子，其对ServiceLoader进行了封装（ExtenstionLoader）

```
public T getExtension(String name) {
    .....
    final Holder<Object> holder = getOrCreateHolder(name);
    Object instance = holder.get();
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            if (instance == null) {
                instance = createExtension(name);
                holder.set(instance);
            }
        }
    }
    return (T) instance;
}
```

2.2 Spring IOC

Spring框架的核心逻辑采用微内核模型，在框架中用户定制了业务逻辑（通过Bean机制），Spring通过IOC及AOP来加载和组装用户的业务流程，如下图：



控制反转（IoC, Inversion of Control）的核心为依赖注入，在传统的应用程序中由开发者在类的内部主动创建依赖对象，从而导致类与类之间高耦合，难以测试。在Spring中通过IoC容器来维护对象生命周期和对象之间关系。在程序中创建和查询依赖对象的控制权有容器来完成，当使用依赖对象时容器将依赖对象注入业务对象，对象与对象之间松耦合，整个体系架构灵活且编译扩展。

> IoC对编程的改变不仅是从代码上，更多的是从思想上发生了“主从换位”的变化，应用程序原本是主导者，需要什么资源便主动的创建，但是在IoC/DI思想中，应用程序被动的等待容器来创建并注入需要的资源。

2.2.1 使用入门

IoC是Spring框架的核心内容，支持使用XML配置、注解及零配置等方式实现IoC。Spring容器在初始化时先读取配置文件，根据配置文件或元数据创建与组织对象存入容器中，程序使用时再从IoC容器中取出需要的对象。下面介绍采用XML方式进行Bean配置的使用方法

1.定义测试类

```
@Data  //lombok注解，简化类的定义
@ToString
public class User {
    private String name;
    private int age;
}
```

2.使用XML配置来实现IoC

在resources目录下创建applicationContext.xml配置文件，内容如下：

```
<?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 = "user" class="com.fys.microkernel.springioc.User">
        <property name="name" value="jeck"/>
        <property name="age" value="18" />
```

```
    </bean>

</beans>
```

在中定义了User的成员变量的赋值

3.测试类

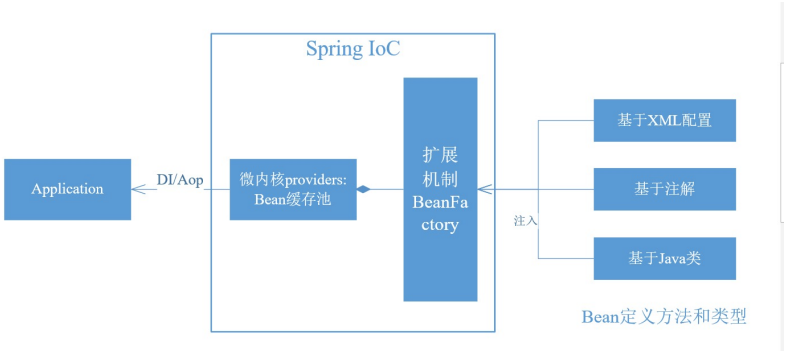
```
public class SpringIocExample {

    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        User user = (User)context.getBean("user");
        System.out.println(user);
    }

}
```

运行后，输出内容如下：

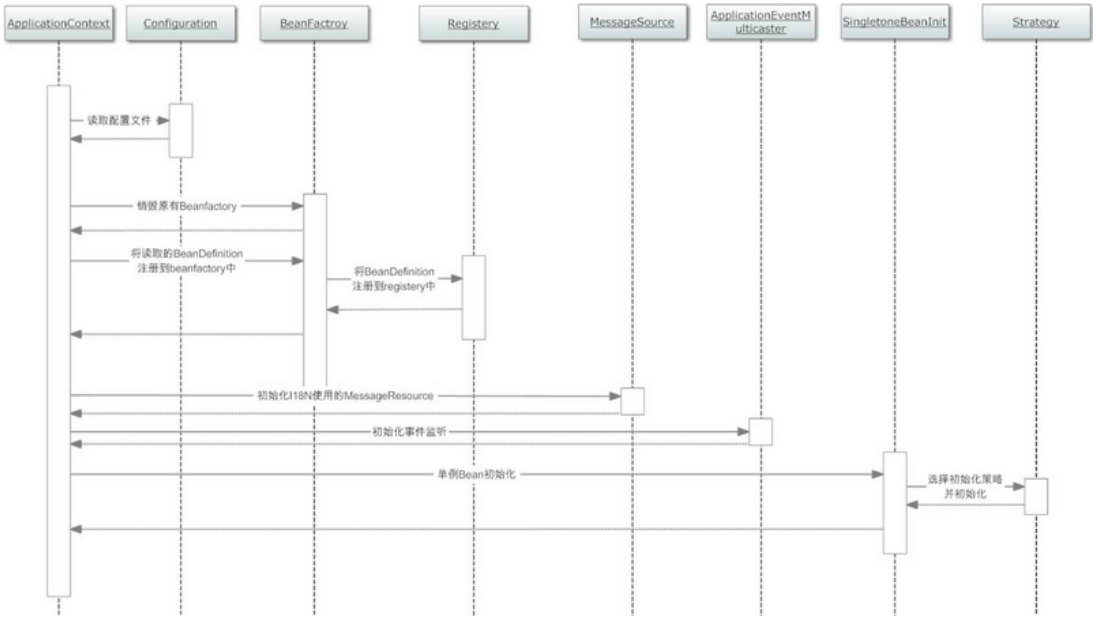
```
User(name=jeck, age=18)
```

2.2.2 原理解析

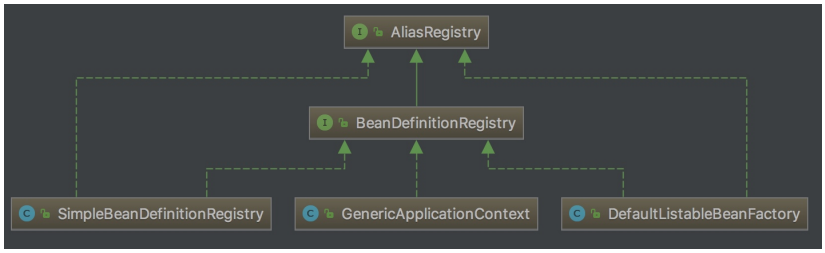
在Spring IoC中对象及之间的关系通过IoC Service Provider进行统一管理和维护，如下图：



在示例中Spring通过ClassPathXmlApplicationContext将配置文件定义的bean注入到Spring Ioc容器中，容器的初始化序列图如下所示：



- 容器初始化与管理 Spring IoC容器的入口ClassPathXmlApplicationContext中的构造器，在初始化过程中将定义的bean的资源文件解析成BeanDefinition后将其注入到容器中，这个过程由BeanDefinitionRegistry来完成，其类结构图如下所示：



BeanDefinitionRegistry继承了AliasRegistry接口，核心子类有三个：SimpleBeanDefinitionRestry、DefaultListableBeanFactory、GenericApplicationContext，其中Default则是一个具有注册功能的完成bean工厂，其核心变量与方法如下：

```
public class DefaultListableBeanFactory extends
AbstractAutowireCapableBeanFactory
        implements ConfigurableListableBeanFactory,
BeanDefinitionRegistry, Serializable {
  ....
  /** Map of bean definition objects, keyed by bean name. */
  private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<>(256);

  public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition) throws BeanDefinitionStoreException {
    ......
    this.beanDefinitionMap.put(beanName, beanDefinition);
    ......
  }

  @Override
  public BeanDefinition getBeanDefinition(String beanName) throws
NoSuchBeanDefinitionException {
```

```
    BeanDefinition bd = this.beanDefinitionMap.get(beanName);
    ......
    return bd;
  }
}
```

- 容器查询 在示例中，通过ApplicationContext#getBean(beanName)来查询Bean，其核心代码如下：

```
public <T> T getBean(Class<T> requiredType, @Nullable Object... args)
throws BeansException {
  Object resolved = resolveBean(ResolvableType.forRawClass(requiredType),
args, false);
  .....
  return (T) resolved;
}

private <T> T resolveBean(ResolvableType requiredType, @Nullable Object[]
args, boolean nonUniqueAsNull) {
  NamedBeanHolder<T> namedBean = resolveNamedBean(requiredType, args,
nonUniqueAsNull);
  if (namedBean != null) {
    return namedBean.getBeanInstance();
  }
  BeanFactory parent = getParentBeanFactory();
  if (parent instanceof DefaultListableBeanFactory) {
    return ((DefaultListableBeanFactory) parent).resolveBean(requiredType,
args, nonUniqueAsNull);
  }
  else if (parent != null) {
    ObjectProvider<T> parentProvider =
parent.getBeanProvider(requiredType);
    if (args != null) {
        return parentProvider.getObject(args);
    }
    else {
        return (nonUniqueAsNull ? parentProvider.getIfUnique() :
parentProvider.getIfAvailable());
    }
  }
  return null;
}
```

调用ObjectProvider生成Bean对象。

3.Spring Dataway

Dataway是基于微内核+插件的的设计思想，使用DataQL提供服务聚合能力框架，其为应用提供了一个接口配置工具，使得使用者无需开发任何代码就可以配置一个满足需求的接口。

> CoC(Convention over Configuration)，惯例由于配置的软件设计原则，将公认的配置方式和信息作为内部缺省的规则来使用，例如Hibernate的映射文件，约定字段名和类属性一致时不需要配置文件；MVN，执行compile时不需要指定源文件及结果输出目录

3.1.使用入门

Dataway是Hasor生态中的一员，在 Spring中使用Dataway首先要做的就是打通两个生态，根据官方文档中推荐的方式可以将 Hasor 和 Spring Boot 整合起来，创建spring boot应用后，开发流程如下

1.引入相关依赖

```
<dependency>
   <groupId>net.hasor</groupId>
   <artifactId>hasor-spring</artifactId>
   <version>4.1.6</version>
 </dependency>
 <dependency>
   <groupId>net.hasor</groupId>
   <artifactId>hasor-dataway</artifactId>
   <version>4.1.6</version>
 </dependency>
```

2.配置application.yml，增加参数

> HASOR_DATAQL_DATAWAY: true HASOR_DATAQL_DATAWAY_ADMIN: true HASOR_DATAQL_DATAWAY_API_URL: /api/ HASOR_DATAQL_DATAWAY_UI_URL: /interface-ui/ HASOR_DATAQL_FX_PAGE_DIALECT: mysql

配置数据源

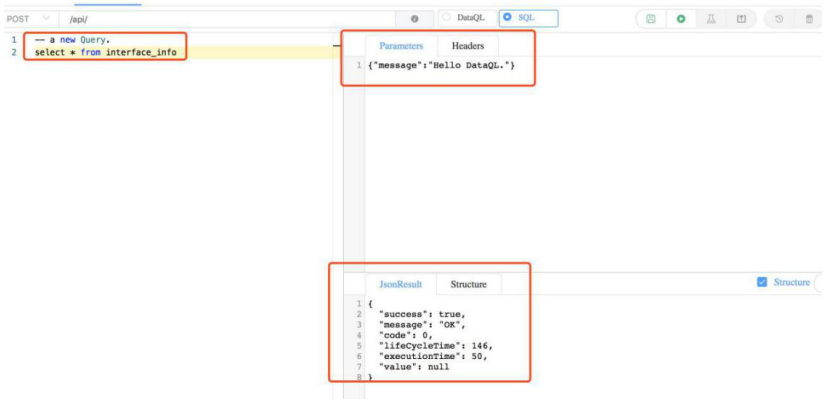
```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/test?
useSSL=false&useUnicode=true&amp&characterEncoding=utf-8
    username: root
    password: Root!!2020
    driver-class: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
    druid:
      initial-size: 3
      ax-active: 10
      max-wait: 60000
      stat-view-servlet:
        login-username: admin
        login-password: admin
      filter:
        stat:
          log-slow-sql: true
          slow-sql-millis: 1
```

3.通过注解启用Hasor，并配置数据源Module

```
@EnableHasor
@EnableHasorWeb
@SpringBootApplication(scanBasePackages = {"com.fys.springboot.dataway"})
public class SpringDatawayApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringDatawayApplication.class, args);
 }}

@DimModule
@Component
public class DatawayModule implements SpringModule {
  @Autowired
  private DataSource dataSource = null;
  @Override
  public void loadModule(ApiBinder apiBinder) throws Throwable {
    apiBinder.installModule(new JdbcModule(Level.Full, this.dataSource));
  }}
```

4.测试及使用

启动Spring Boot应用程序后，打开Dataway UI: http://localhost:8080/interface-ui/, 界面在Dataway中提供了DataQL和SQL语言。下面是SQL模式的使用模式，执行select查询，执行后可以看到这些结果



输出Qeury后，点击右上角发布按钮即可发布接口

3.2.原理解析

3.2.1 数据库schema

通过Dataway UI定义及发布接口，这些接口的调用信息，存储在数据库中，定义的schema如下表

```
CREATE TABLE `interface_info` (
    `api_id`           int(11)      NOT NULL AUTO_INCREMENT   COMMENT 'ID',
    `api_method`       varchar(12)  NOT NULL                  COMMENT
'HttpMethod: GET、PUT、POST',
    `api_path`         varchar(512) NOT NULL                  COMMENT '拦截
路径',
```

```
    `api_status`      int(2)       NOT NULL                  COMMENT '状
态：0草稿，1发布，2有变更，3禁用',
    `api_comment`     varchar(255)     NULL                  COMMENT '注
释',
    `api_type`        varchar(24)  NOT NULL                  COMMENT '脚本
类型：SQL、DataQL',
    `api_script`      mediumtext   NOT NULL                  COMMENT '查询
脚本：xxxxxxx',
    `api_schema`      mediumtext       NULL                  COMMENT '接口
的请求/响应数据结构',
    `api_sample`      mediumtext       NULL                  COMMENT '请求/
响应/请求头样本数据',
    `api_option`      mediumtext       NULL                  COMMENT '扩展
配置信息',
    `api_create_time` datetime     DEFAULT CURRENT_TIMESTAMP COMMENT '创建
时间',
    `api_gmt_time`    datetime     DEFAULT CURRENT_TIMESTAMP COMMENT '修改
时间',
    PRIMARY KEY (`api_id`)
) ENGINE=InnoDB AUTO_INCREMENT=0 DEFAULT CHARSET=utf8mb4 COMMENT='Dataway
中的API';

CREATE TABLE `interface_release` (
    `pub_id`          int(11)      NOT NULL AUTO_INCREMENT   COMMENT
'Publish ID',
    `pub_api_id`      int(11)      NOT NULL                  COMMENT '所属
API ID',
    `pub_method`      varchar(12)  NOT NULL                  COMMENT
'HttpMethod：GET、PUT、POST',
    `pub_path`        varchar(512) NOT NULL                  COMMENT '拦截
路径',
    `pub_status`      int(2)       NOT NULL                  COMMENT '状
态：0有效，1无效（可能被下线）',
    `pub_type`        varchar(24)  NOT NULL                  COMMENT '脚本
类型：SQL、DataQL',
    `pub_script`      mediumtext   NOT NULL                  COMMENT '查询
脚本：xxxxxxx',
    `pub_script_ori`  mediumtext   NOT NULL                  COMMENT '原始
查询脚本，仅当类型为SQL时不同',
    `pub_schema`      mediumtext       NULL                  COMMENT '接口
的请求/响应数据结构',
    `pub_sample`      mediumtext       NULL                  COMMENT '请求/
响应/请求头样本数据',
    `pub_option`      mediumtext       NULL                  COMMENT '扩展
配置信息',
    `pub_release_time`datetime     DEFAULT CURRENT_TIMESTAMP COMMENT '发布
时间（下线不更新）',
    PRIMARY KEY (`pub_id`)
) ENGINE=InnoDB AUTO_INCREMENT=0 DEFAULT CHARSET=utf8mb4 COMMENT='Dataway
API 发布记录';

create index idx_interface_release on interface_release (pub_api_id);
```

在示例中发布的接口，对应的底层信息

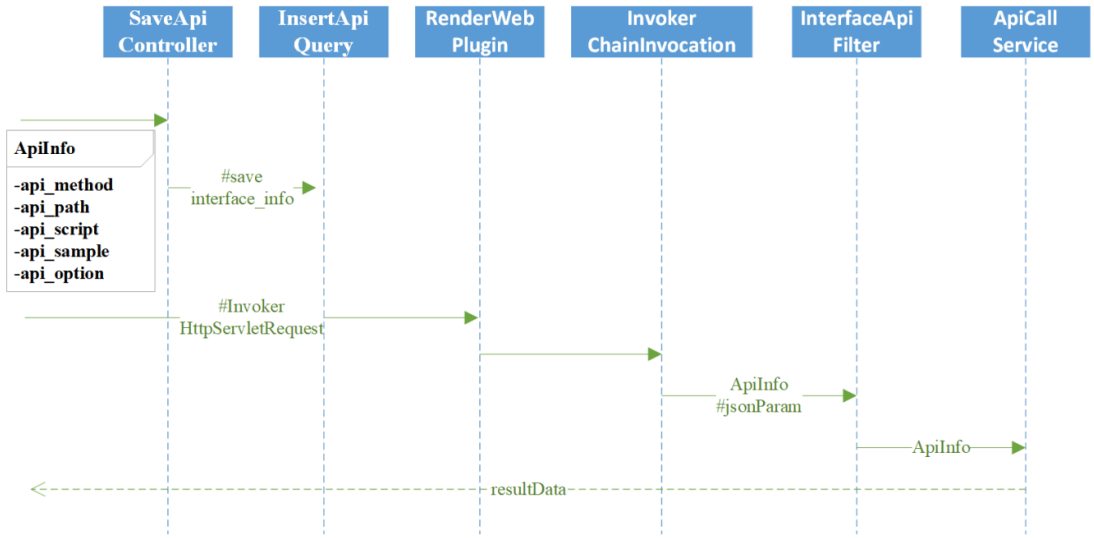

```
{
  "code": 200,
  "message": "OK",
  "result": {
    "id": 4,
    "select": "GET",
    "path": "/api/msg",
    "status": 1,
    "apiComment": "",
    "codeType": "DataQL",
    "codeInfo": {
      "codeValue": "// a new Query.\nreturn ${message};",
      "requestBody": "{\"message\":\"Hello DataQL.\"}",
      "headerData": []
    },
    "optionData": {
      "resultStructure": true,
      "responseFormat": "{\n  \"success\"      : \"@resultStatus\",\n
\"message\"      : \"@resultMessage\",\n  \"code\"         :
\"@resultCode\",\n  \"lifeCycleTime\": \"@timeLifeCycle\",\n
\"executionTime\": \"@timeExecution\",\n  \"value\"        :
\"@resultData\"\n}"
    }
  },
  "success": true
}
```


在保存的接口信息中，核心字段包括：

- path, 接口路径
- codeType，配置的信息获取类型，包括DataQL、SQL
- codeValue，具体执行的代码
- requestBody，接口请求中的传输的数据
- responseFormat，响应信息

3.2.2 接口注册及调用流程

接口发布后，就可以直接调用，流程如下：



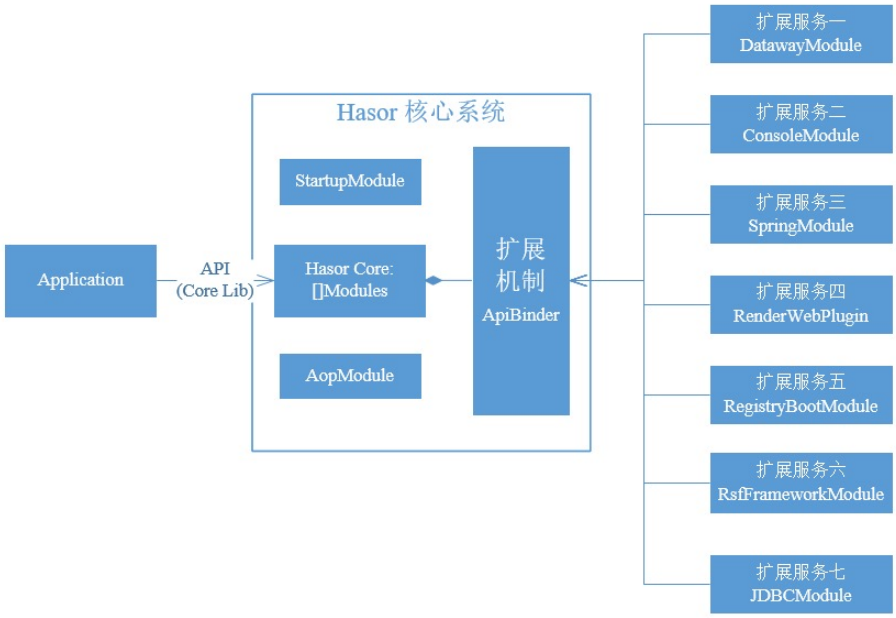
前端调用注册的接口，通过InterfaeApiFilter进行拦截，并调用ApiCallService从数据库中拉取执行代码并返回信息

```
private Object _doCall(ApiInfo apiInfo, QueryScriptBuild scriptBuild,
boolean needThrow)  {

   // .执行查询
   //  - 1.首先将 API 调用封装为 单例的 Supplier
   //  - 2.准备一个 Future 然后，触发 PreExecuteListener SPI
   //  - 3.如果 Future 被设置那么获取设置的值，否则就用之前封装好的 Supplier 中取值
   BasicFuture<Object> newResult = new BasicFuture<>();
   QueryResult execute = null;
   try {
      this.spiTrigger.chainSpi(PreExecuteChainSpi.class, (listener,
lastResult) -> {
        if (!newResult.isDone()) {
          listener.preExecute(apiInfo, newResult);
        }
        return lastResult;
      });
      ......
  else {
    // 4.编译DataQL查询，并执行查询
     final String scriptBody =
scriptBuild.buildScript(apiInfo.getParameterMap());
     final Set<String> varNames =
this.executeDataQL.getShareVarMap().keySet();
     final Finder finder = this.executeDataQL.getFinder();
    QIL compiler = this.spiTrigger.notifySpi(CompilerSpiListener.class,
(listener, lastResult) -> {
        return listener.compiler(apiInfo, scriptBody, varNames, finder);},
null);
    if (compiler == null) {
   compiler = CompilerSpiListener.DEFAULT.compiler(apiInfo, scriptBody,
varNames, finder);
    }
    execute =
```

```
this.executeDataQL.createQuery(compiler).execute(apiInfo.getParameterMap()
);
   }
}
   // 5.返回值
   try {
   Object resultData = execute.getData();
   resultData = this.spiTrigger.chainSpi(ResultProcessChainSpi.class,
(listener, lastResult) -> {
       if (lastResult instanceof DataModel) {
         lastResult = ((DataModel) lastResult).unwrap();
       }
       return listener.callAfter(newResult.isDone(), apiInfo, lastResult);
     }, resultData);
     return
DatawayUtils.queryResultToResultWithSpecialValue(apiInfo.getOptionMap(),
execute, resultData).getResult();
   }
}
```

3.2.3 Spring Dataway Module

Spring Dataway基于Hasor的微内核框架，涉及的各组件如下图所示：



用户通过Hasor.create调用，将Module添加到List中，并通过AppContext启动各个Module，在start中执行如下：

```
for (Module module : findModules) {
   if (module != null) {
        this.installModule(apiBinder, module); //module.onStart(eventData)
    }
}
```

支持多Module的启动，在示例中启动DatawayModule，在启动过程中绑定jdbc db来存储接口信息，并配置InterfaceUiFilter拦截REST 请求，根据注册的接口信息进行调用，过程不再详述。

4.Spring Cloud Fuction

Spring Cloud Fcuntion致力于促进函数作为主要的开发单元，该项目提供了在各个平台（Amazon AWS Lambda等FaaS函数即服务）上部署基于函数的软件的通用模型，主要目标如下：

- 在业务逻辑中函数作为基本元素
- 业务逻辑的开发生命周期与Runtime解耦，以此来实现代码可以作为Web Endpint、流数据或者Task来执行
- 支持独立运行及使用Serverless Provider平台提供的统一编程模型
- 在Serverless Provider上支持Spring Boot的功能，如auto-onfig、DI及Metric 在SCF中抽象了所有的运行基础设施及传输细节，开发者因此可以使用熟悉的工具及进程，从而把重心集中在业务逻辑的实现上

4.1 使用入门

Spring Cloud Function基于java.util.function包中定义的核心接口：Function、Consumer和Supplier来推广编程模型

1) 添加Maven依赖 新建maven项目，并在pom.xml中添加以依赖如下

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.0.RELEASE</version>
        <relativePath/>
    </parent>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-function-webflux</artifactId>
    </dependency>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-configuration-processor</artifactId>
    </dependency>
    <dependency>
       <groupId>org.json</groupId>
       <artifactId>json</artifactId>
       <version>20200518</version>
    </dependency
```

2）定义函数及启动程序

```
@SpringBootApplication
public class SpringCloudFunctionExample {
```

```
    @Bean
    public Function<String, String> uppercase() {
        return value -> value.toUpperCase();
    }
    public static void main(String[] args) {
        SpringApplication.run(SpringCloudFunctionExample.class, args);
    }
}
```

3）运行及测试

- 启动信息

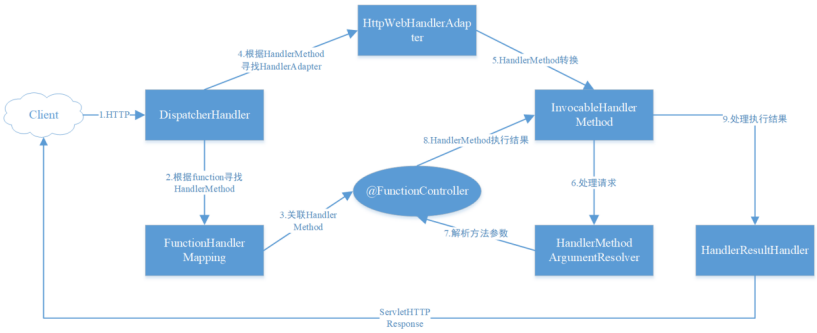
> FunctionCatalog: org.springframework.cloud.function.context.catalog.BeanFactoryAwareFunctionRegistry@1b28f282 Netty started on port(s): 8080

- 测试执行

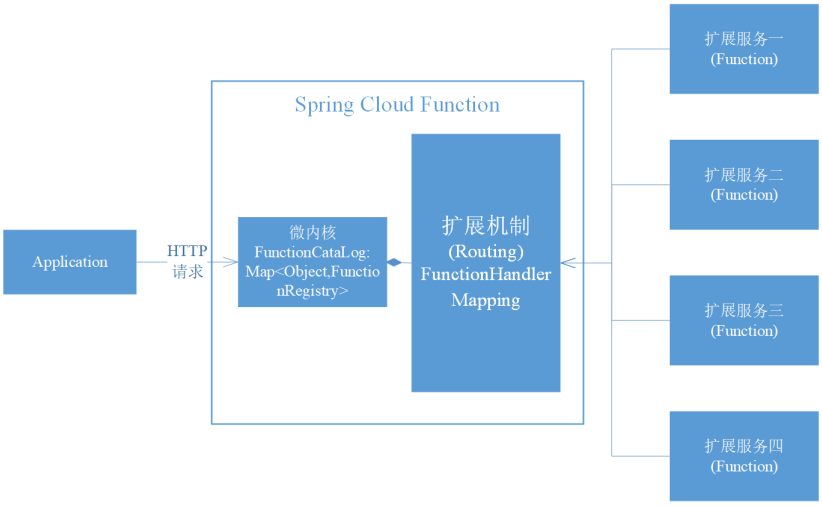
> $curl -H "Content-Type: text/plain" localhost:8080/uppercase -d 'hello' HELLO

4.2 原理分析

Spring Cloud Function的执行逻辑与Spring MVC类似，流程图如下所示：



在Spring MVC框架中根据url path从FunctionHandlerMapping获取对应的Function，然后通过InvocableHandlerMethod进行调用。Function通过@Bean注解标注，ApplicationContext启动时加载到FunctionMappingHandler中，Function实例对象存储在FunctionCatalog中。将系统架构转换为微内核框架如下图所示：



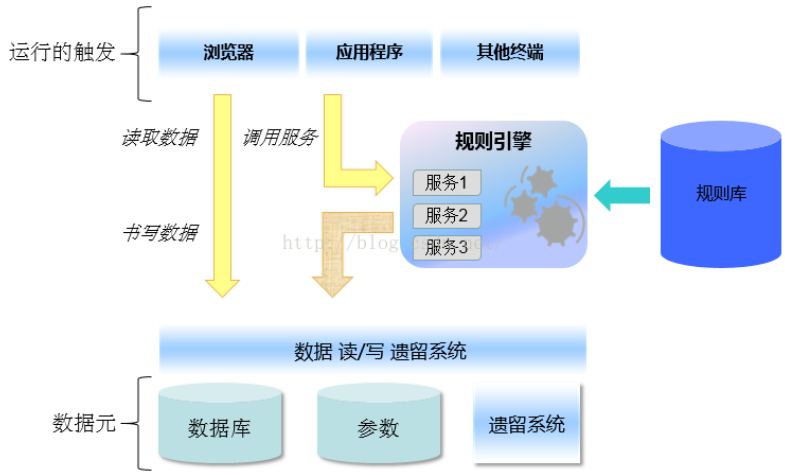
FunctionCatalog的核心逻辑如下所示：

```
public class SimpleFunctionRegistry implements FunctionRegistry,
FunctionInspector {

    ......
    private final Map<Object, FunctionRegistration<Object>>
registrationsByFunction = new HashMap<>();
    private final Map<String, FunctionRegistration<Object>>
registrationsByName = new HashMap<>();
    private List<String> declaredFunctionDefinitions;


    @Override
    @SuppressWarnings("unchecked")
    public <T> T lookup(String definition, String... acceptedOutputTypes)
{
        definition = StringUtils.hasText(definition) ?
definition.replaceAll(",", "|") : "";
        boolean routing =
definition.contains(RoutingFunction.FUNCTION_NAME)
                ||
this.declaredFunctionDefinitions.contains(RoutingFunction.FUNCTION_NAME);
        .....
        FunctionInvocationWrapper function = (FunctionInvocationWrapper)
this.compose(null, definition, acceptedOutputTypes);
        return (T) function;
    }

    private Function<?, ?> compose(Class<?> type, String definition,
String... acceptedOutputTypes) {
       definition = discoverDefaultDefinitionIfNecessary(definition);
       Function<?, ?> resultFunction = null;
       if (this.registrationsByName.containsKey(definition)) {
         Object targetFunction =
this.registrationsByName.get(definition).getTarget();
         Type functionType =
this.registrationsByName.get(definition).getType().getType();
```

```
            resultFunction = new FunctionInvocationWrapper(targetFunction,
functionType, definition, acceptedOutputTypes);
        }
        .....
       return resultFunction;
    }

    ......
}
```

5. Drools规则引擎

在很多企业的IT业务系统中会有大量业务规则配置，而且随着企业管理的决策变化，业务规则也会随之发生更改，为了适应这样的需求，通常是将业务规则的配置独立出来使之与业务系统保持低耦合，规则引擎是实现这一功能的核心组件，如下图所示：



规则引擎是一种相对简单的推理机，可以将规则引擎作为组件嵌入到应用系统中，从而将业务决策从应用程序代码中分离出来，并使用预定义的规则语言编写业务决策。规则引擎也属于微内核架构的一种实现，其中执行引擎可以看做是微内核，执行引擎通过解析、执行配置规则，来达到业务的灵活多变。 Drools是用Java语言编写的开发源码规则引擎，使用Rete算法对所编写的规则进行求值，其提供了核心业务规则引擎（BRE）、Web UI和规则管理应用程序（Drools Workbench），易于访问企业策略、调整和管理

5.1 使用入门

在服务端项目中使用drools的基本步骤如下

- 1）在服务端项目中使用drools的基本步骤如下

```
<dependency>
 <groupId>org.kie</groupId>
 <artifactId>kie-api</artifactId>
</dependency>
<dependency>
 <groupId>org.drools</groupId>
 <artifactId>drools-core</artifactId>
</dependency>
```

```
<dependency>
 <groupId>org.drools</groupId>
 <artifactId>drools-compiler</artifactId>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-decisiontables</artifactId>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-templates</artifactId>
</dependency>
<dependency>
  <groupId>org.kie</groupId>
  <artifactId>kie-internal</artifactId>
</dependency>
```

- 2）创建规则测试类及规则文件

```
@Getter
 @Setter
 public static class Message {
  public static final int HELLO   = 0;
  public static final int GOODBYE = 1;

  private String          message;
  private int             status;

  public static Message doSomething(Message message) {
      return message;
  }

  public boolean isSomething(String msg,
                             List<Object> list) {
      list.add( this );
      return this.message.equals( msg );
  }
}
规则文件
package org.drools.examples.helloworld

import org.drools.examples.helloworld.HelloWorldExample.Message;

global java.util.List list

rule "Hello World"
    dialect "mvel"
    when
        m : Message( status == Message.HELLO, message : message )
    then
        System.out.println( message );
        modify ( m ) { message = "Goodbye cruel world",
```

```
                    status = Message.GOODBYE };
end

rule "Good Bye"
    dialect "java"
    when
        Message( status == Message.GOODBYE, message : message )
    then
        System.out.println( message );
end
```

- 3）创建配置文件，在resources/META-INF中增加kmodule.xml，内容如下

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://www.drools.org/xsd/kmodule">

  <kbase packages="org.drools.examples.helloworld">
      <ksession name="HelloWorldKS"/>
  </kbase>
</kmodule>
```

- 4）测试类及执行

```
public class HelloWorldExample {

  public static final void main(final String[] args) {
    KieServices ks = KieServices.get();
    KieContainer kc = ks.getKieClasspathContainer();
    execute( ks, kc );
  }

  public static void execute( KieServices ks, KieContainer kc ) {
    KieSession ksession = kc.newKieSession("HelloWorldKS");
    ksession.setGlobal( "list", new ArrayList<Object>() );
    ksession.addEventListener( new DebugAgendaEventListener() );
    ksession.addEventListener( new DebugRuleRuntimeEventListener() );

    KieRuntimeLogger logger = ks.getLoggers().newFileLogger( ksession,
"./helloworld" );
    final Message message = new Message();
    message.setMessage( "Hello World" );
    message.setStatus( Message.HELLO );
    ksession.insert( message );

    ksession.fireAllRules();

    logger.close();
    ksession.dispose();
```

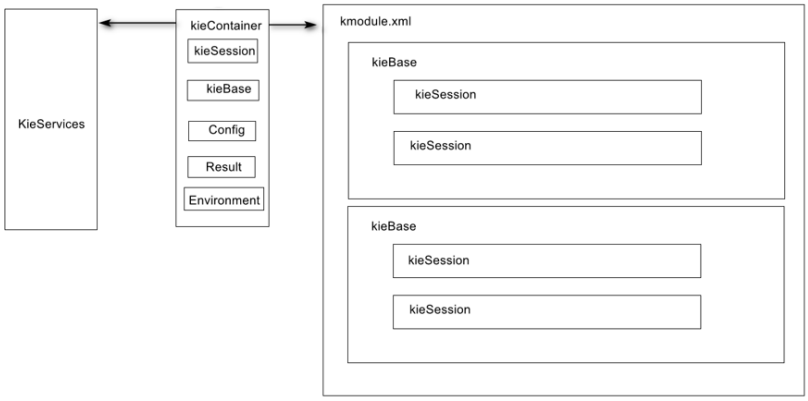
```
    }
}
```

执行测试后输出如下

- Hello World
- Goodbye cruel world

5.2 原理分析

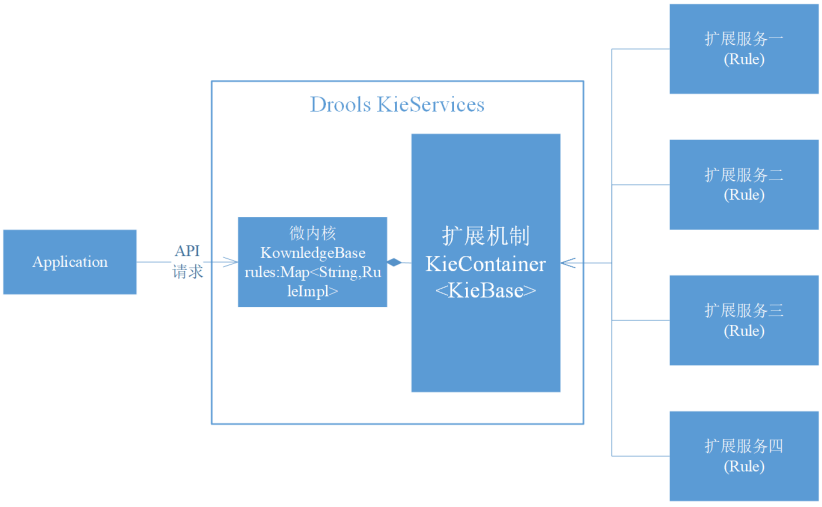
Drools涉及到的核心概念如下：



KieServices启动时从kmodule.xml中加载一个或者多个Kiebase，在Kiebase中可配置多个KieSession，通常动态生成

- KieService，可以访问所有的创建和运行时的对象，日志、资源、环境和容器等
- KieContainer，装载与引擎相关的组件，为其他kie工具提供服务
- KieSession，存储会话及执行运行时数据
- KieBase，在项目的KIE模块描述文件（kmodule.xml）中定义的存储库，但是不包括运行时的数据

在规则引擎的设计中采用了微内核的思想，其框架图如下所示：



KieService将kmodule.xml中配置的drl文件加载到KnowledgeBase的rules:Map<String,RuleImpl>中，Rule经过Agenda封装后由RuleExecutor来调用，最终调用MVELConsequence，核心代码如下：

```
public void evaluate(final KnowledgeHelper knowledgeHelper,
                         final WorkingMemory workingMemory) throws
Exception {
   ...
   InternalKnowledgePackage pkg =
workingMemory.getKnowledgeBase().getPackage( "MAIN" );
   CompiledExpression compexpr = (CompiledExpression) this.expr;
   ...
   MVEL.executeExpression( compexpr, knowledgeHelper, factory );
}
```

在MVEL执行过程中读取规则，将其编译成可执行文件。

6.参考链接

> https://www.jianshu.com/p/2f1a316185b1
> https://www.cnblogs.com/best/p/5727935.html
> https://www.cnblogs.com/best/p/5727935.html
> https://segmentfault.com/a/1190000017348726?utm_source=tag-newest http://cmsblogs.com/?p=4026