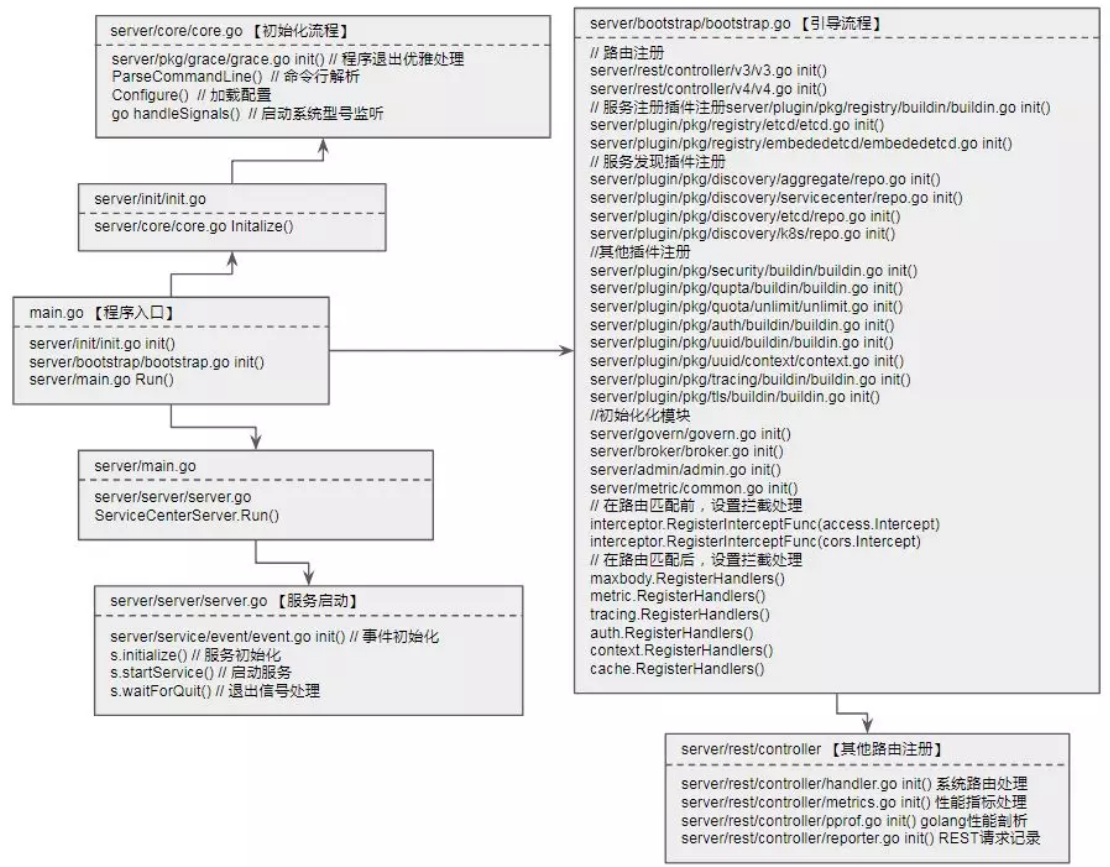
**[学习微服务第10天] Service-Center 启动流程分析**

上一篇我们介绍了Service-Center使用入门,本篇我们将介绍Service-Center 的启动流程分析。

简介

Service-Center的启动流程共分为初始化、启动引导、服务启动三个部分，以下是对启动流程的梳理：



正如所有的golang项目一样，运行流程是从main函数开始的。在main.go中，总共运行了三个方法（其中有两个init函数是通过import匿名导入包来完成的）。其中server/init/init.go init()执行了项目的初始化工作，server/bootstrap/bootstrap.go init()主要负责引导工作，server/main.go Run()才是具体的服务启动。下面让我们具体的分析一下各自工作。

**一. 初始化流程**

从以上流程图可以看出

负责初始化流程的server/init/init.go init()

实际工作是在server/core/core.go中完成的

**其主要做了以下三件事情：**

1.监听系统相关终止信号，用以实现程序优雅重启

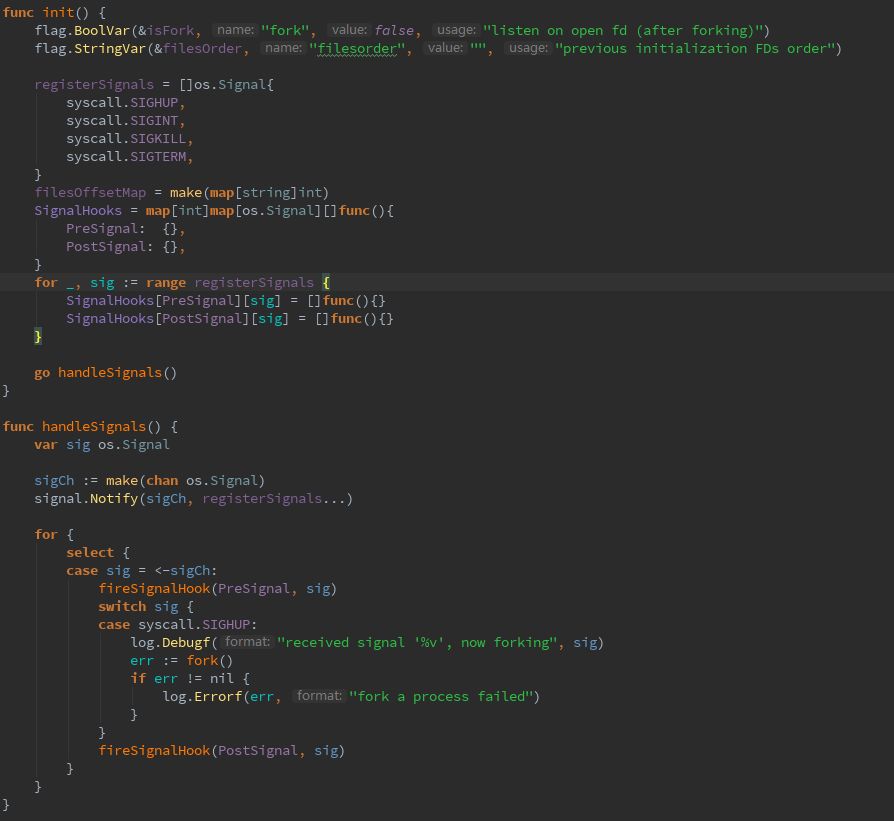
2.解析命令行参数，确认当前运行模式

3.加载配置文件，为后续启动流程提供参数

**1程序优雅重启**

**server/pkg/grace/grace.go**

在这里我们先来看看grace.go init()的源码：



此处总共做了一下三件事情：

1.从命令行中读取参数fork、filesorder

2.初始化系统信号监听集（SIGHUP、SIGINT、SIGKILL、SIGTERM）、并将其加入钩子队列

3.启动系统信号监听，接收到符合条件信号则进行前置、fork、后置等处理  
fork：SIGHUP信号处理函数，实际执行优雅重启。在接收到信号后，会在原始的命令行中增加-fork -filesorder=xxx的参数，组成新的命令行，在子进程中执行重启应用程序。

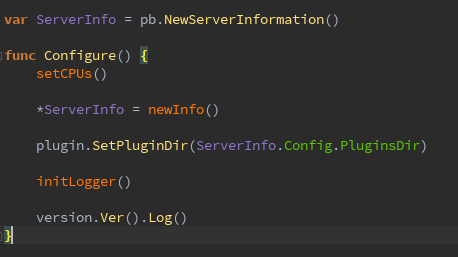
**2命令行参数解析和服务配置预载**

**server/core/core.go**

在server/core/core.go中，程序显式调用的有以下三个方法：

1．**ParseCommandLine(): 解析命令行参数，若命令行参数中-v的值为true，将打印版本信息，并退出**

2．**Configure()：服务启动配置**



•setCPUs(): 设置并发CPU数量

•newInfo()：从beego解析的配置文件中获取参数，并保存启动配置到全局变量ServerInfo中，以便后续的流程读取。（此处读取配置参数使用了beego组件，而在beego包中读取配置文件是在config.go init()方法中完成的，它会从默认文件位置:“./conf/app.conf”加载配置文件，该模式有些隐晦，大家需要注意一下）

•SetPluginDir(): 设置自定义插件目录（Service-Center很多功能是基于插件的形式设计的，支持用户使用自定义的插件，这个将会在后续的介绍文章中进行详细的分析）

•initLogger()：根据ServerInfo的配置，初始化日志工具

•version.Ver().Log()：打印版本信息

3．**go handleSignals(): 监听系统信号，在接收到信号时，延迟5s退出，并执行日志落盘（配合子进程重启应用）。**

**二. 启动引导流程**

启动引导流程主要负责服务的内部模块的注册与初始化，所有的工作均在server/bootstrap/bootstrap.go中进行，涉及以下几个方面：

1．路由注册

Service-Center的API基于RESTful进行构建的，需要启用http的服务监听，此处主要进行http路由的注册。由于涉及两个版本的兼容，所以注册了v3、v4两组API，具体实现是一致的，仅在URL路由上有差别。以下仅对v4代码进行分析，v3版本注册与其并无区别。

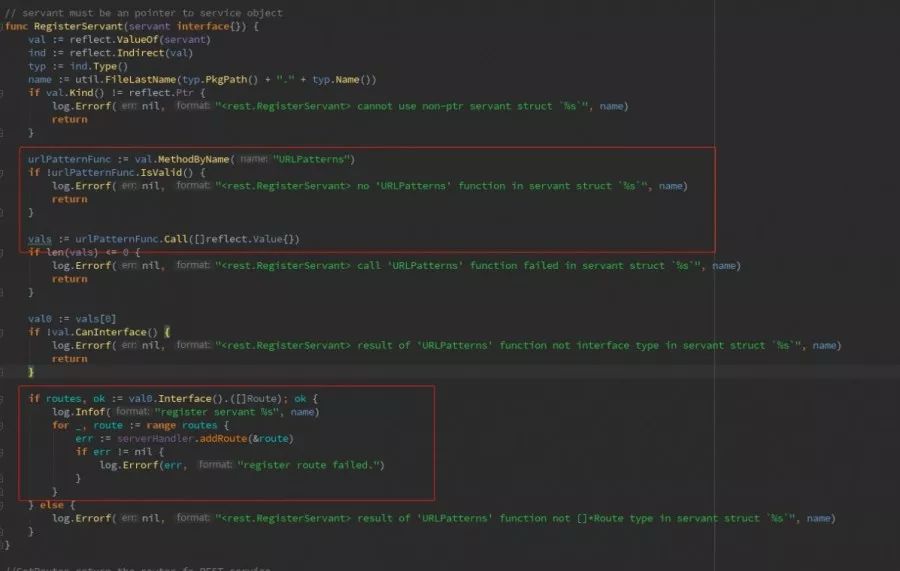
**server/rest/controller/v4/v4.go**

如图所示，在v4.go分别向roa中注册了多个RESTful处理对象。



**pkg/rest/roa.go**

接下来我们看看roa.RegisterServant()的实现：

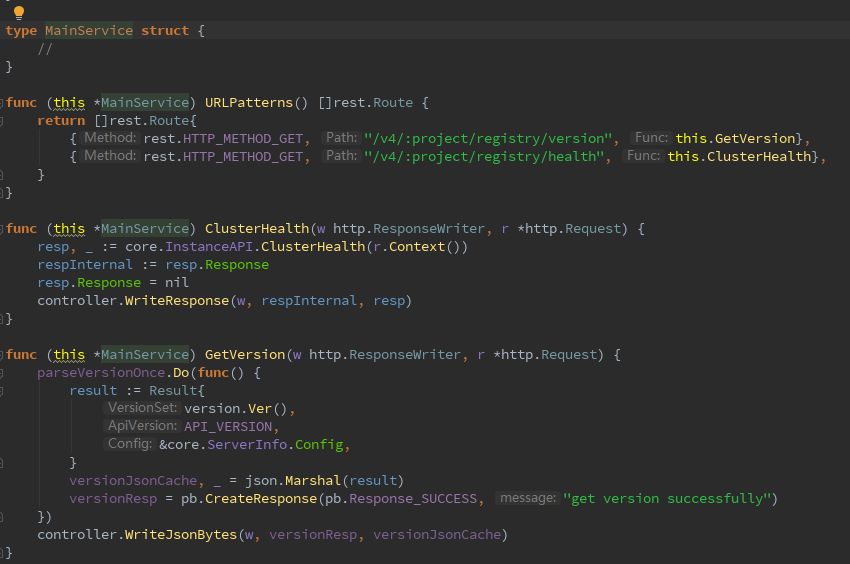


•在上图的第一个红框中，可以看到该方法通过反射的方式，获取并执行了输入对象的URLPatterns方法

•在第二个红框中，对上一步中返回的第一个参数进行[]Route切片的断言，如果类型符合，将其注册到http路由中（serverHandler.addRoute()）

**server/rest/controller/v4/main\_controller.go**

我们再来看看v4.go中所注册对象的具体实现，此处挑选了MainService，其他的不做赘述：

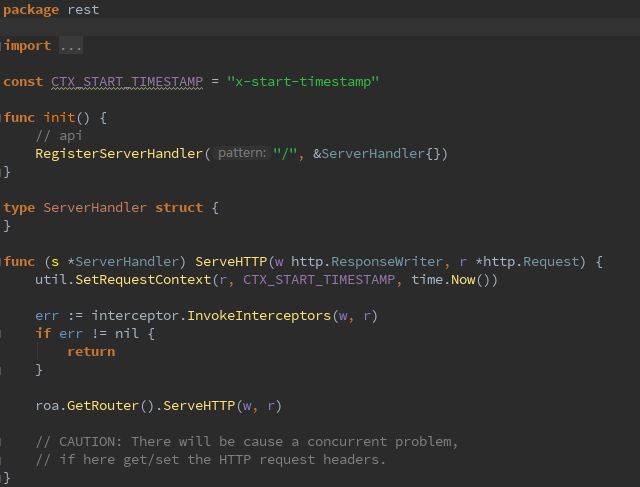


URLPatterns()方法返回了一个Route的切片，其中包括http method、路由和具体处理方法。当启动监听并匹配对应的method和路由后，会触发这里定义的方法。

2．其他路由注册（server/rest/controller）

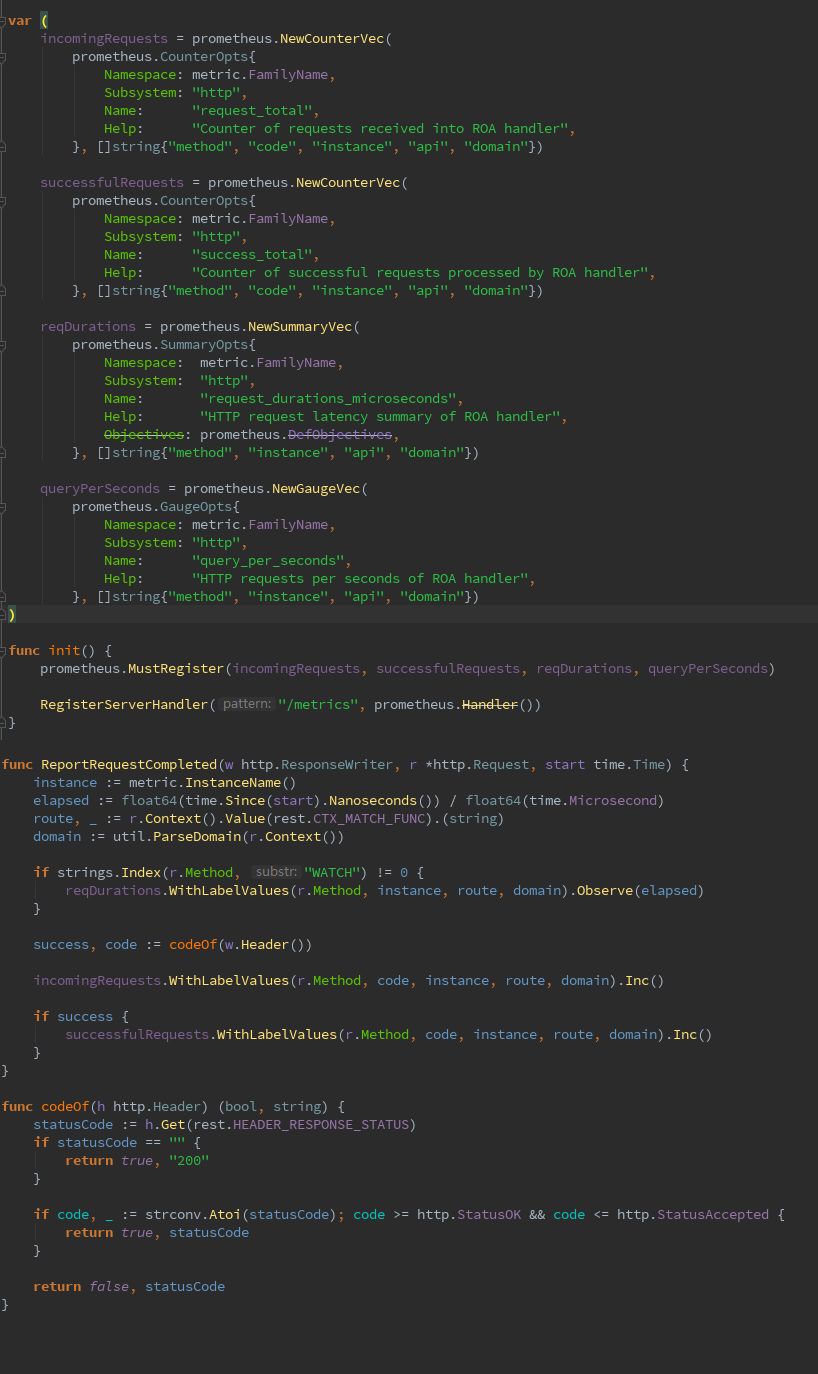
**server/rest/controller/handler.go**

在此文件中，Service-Center注册了服务总路由，所有的请求都将有此处进入服务的处理逻辑部分。在处理方法中，对进入的请求按序进行开始时间记录、拦截器调用、路由匹配。



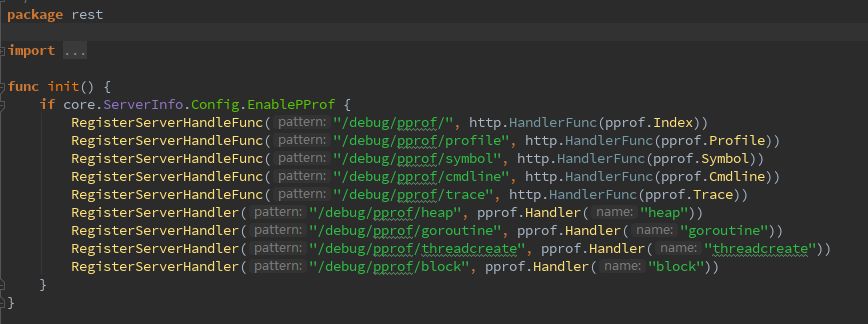
**server/rest/controller/metrics.go**

对接prometheus系统，向prometheus中注册进入请求计数器、成功请求计数器、请求计时器、预处理计时器，并向服务注册性能监控路由



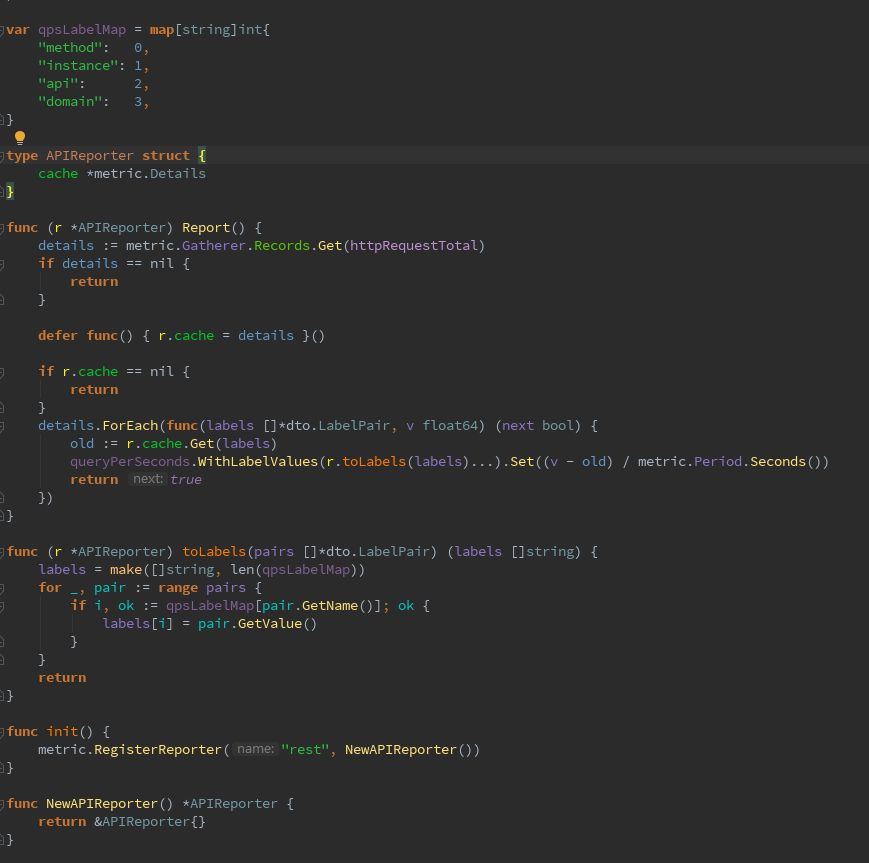
**server/rest/controller/pprof.go**

注册golang性能剖析路由



**server/rest/controller/reporter.go**

向性能监控模块注册REST请求记录器



3系统性能指标模块（metrics

**server/rest/controller/reporter.go**

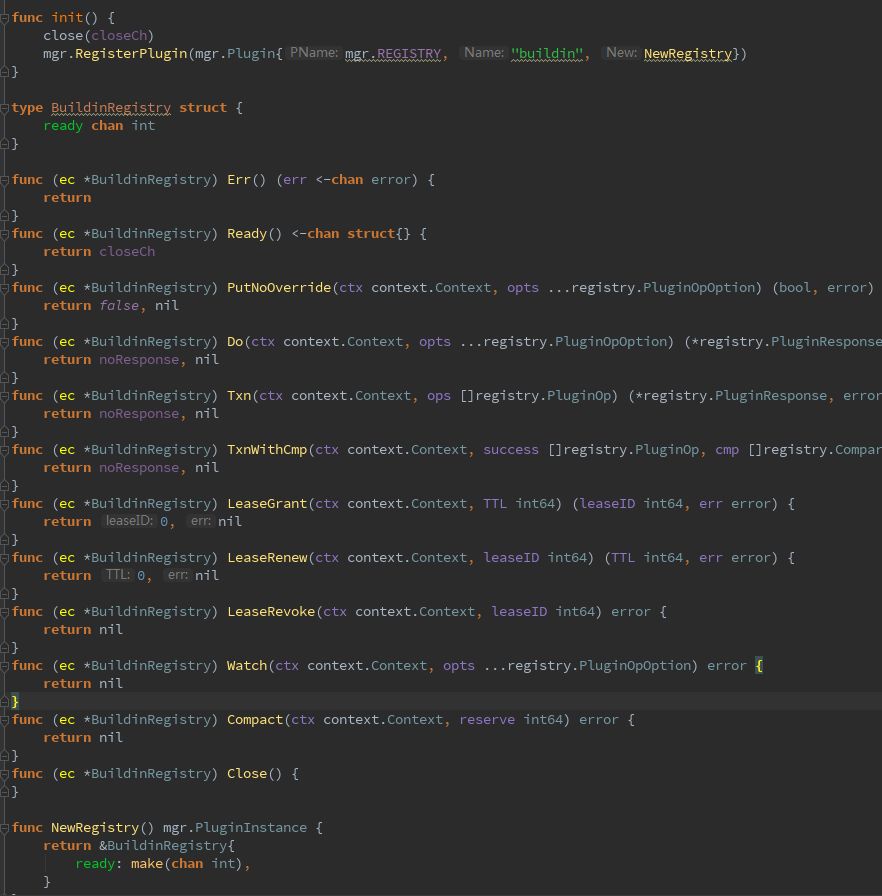
Service-Center性能指标模块对接了prometheus，在此文件中启动了一个指标收集者，定时收集指标并进行记录



**4.其他模块路由注册（govern、broker、admin）**

治理（govern）、请求代理（broker）、管理面（admin）的路由注册与前面提到的路由注册并无本质差异，此处就不做赘述了。  
**5.服务注册插件**

Service-Center将服务注册引擎进行了接口抽象，方便我们对接不同的注册系统，并提供了buildin（空实现仅对接口进行定义不做注册）、embededetcd（Service-Center内嵌的ETCD）、etcd（其他etcd或其集群）三个默认的注册系统。为了展示方便，以下仅展示buildin内容，其他类型原理相同：



**server/plugin/pkg/registry/buildin/buildin.go**

在init()方法中，向插件注册管理器注册了一个名为“buildin”的Registery实例；该实例实现了对应的仓库接口

**6.服务发现插件**

为了方便不同的服务发现引擎的对接（将在异构的场景中进行介绍），Service-Center还抽象了服务发现Discovey接口，也做了插件的支持，当前内置了aggregate、etcd、k8s、servicecenter四种类型服务发现系统的对接。此处我们仅以servicecenter为例，查看其初始化流程，具体实现大家可以参阅项目源码：



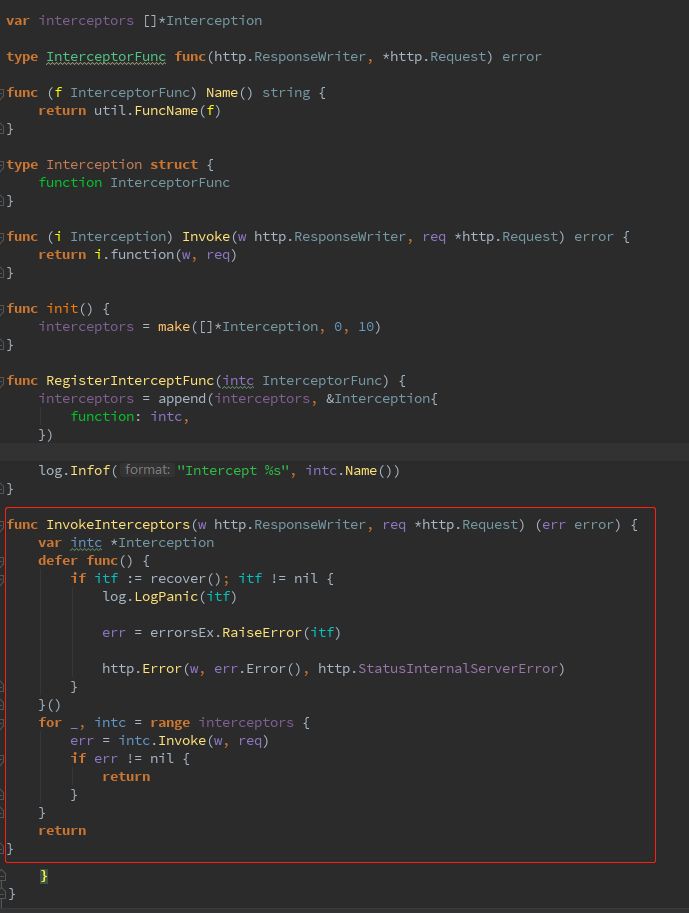
在init()方法中，向插件注册管理器注册了一个名为“servicecenter”的Discovery实例

**7.其他插件注册**

Service-Center还对加解密（cipher）、资源配额(quota)、认证（auth）、全局唯一ID（uuid）、链路跟踪（tracing）、安全传输协议（tls）进行了插件支持，并对其进行了基本的实现，用户可以通过自定义的插件进行替换。由于此部分代码与上面两个插件的注册方法类似，此处就不再展开说明了。

**8.请求拦截器设置**

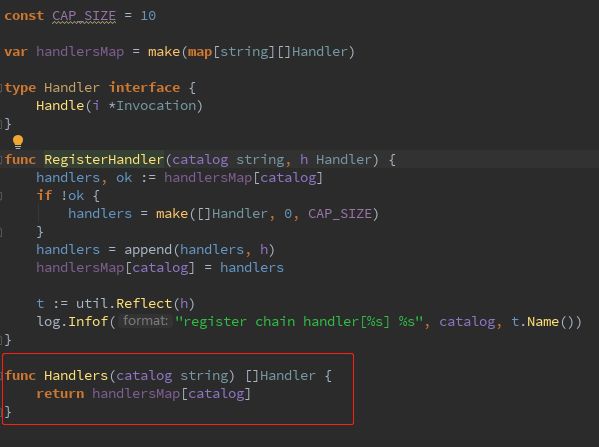
**路由匹配前：**  
通过interceptor.RegisterInterceptFunc()，将访问校验（access）、跨域支持（cors）注册到了访问之前的拦截处理中，具体实现位于server/interceptor/interceptor.go中，如图：



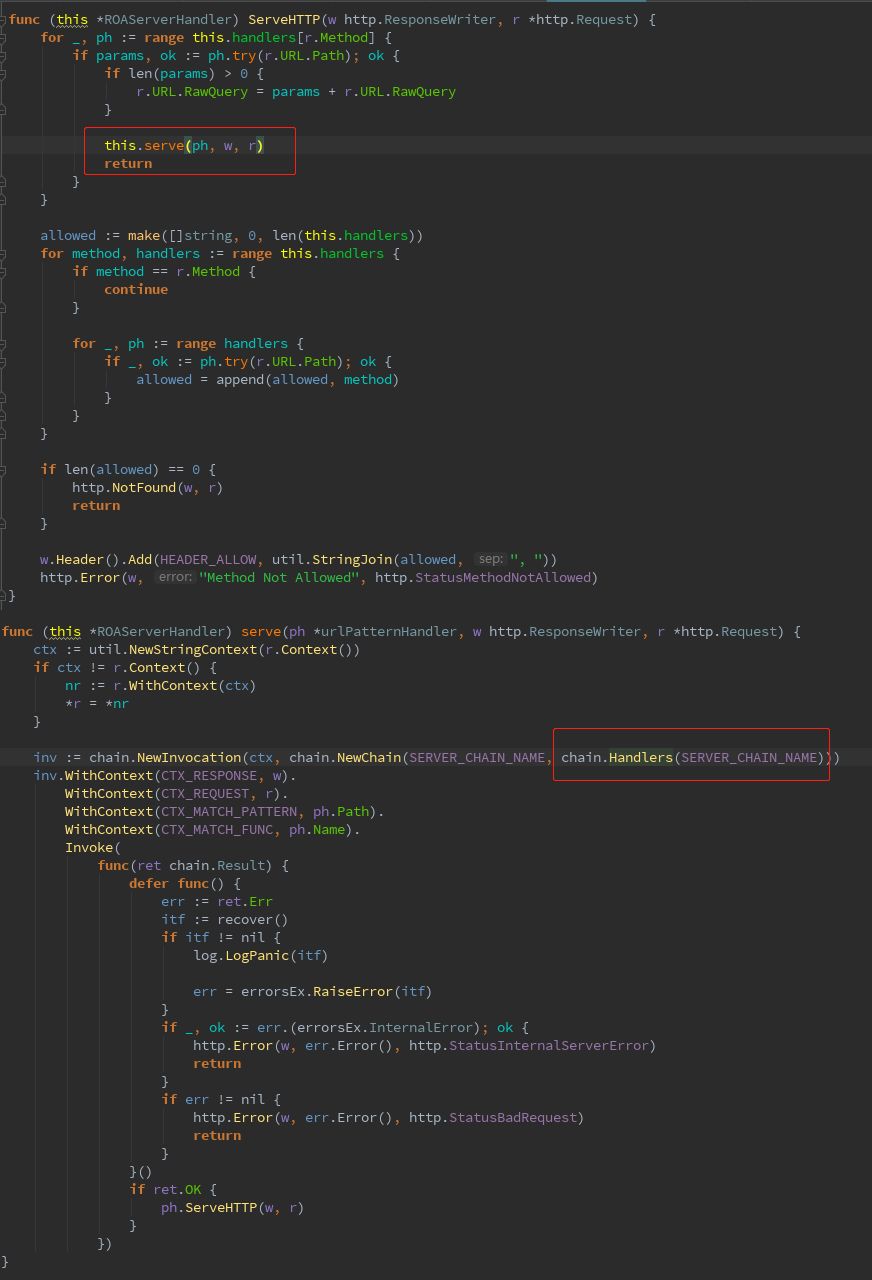
•在请求进入server/rest/controller/handler.go时，interceptor.InvokeInterceptors(w, r)方法，将在路由匹配之前调用上图红框中的方法，具体请见上面“其他路由注册-server/rest/controller/handler.go”的说明

**路由匹配后：**

在最大请求大小（maxbody）、性能指标（metric）链路追踪（tracing）、认证（auth）、上下文（context）、请求缓存（cache）各自模块的RegisterHandler()方法中，通过chain.RegisterHandler()方法将自身注册到了访问之后的拦截处理中： 文件位置：server/rest/handler.go



注册进来的处理将会被存储在变量handlersMap中，在红框中Handlers()方法被调用了被执行。  
通过跟踪可以发现Handlers()方法在server/rest/route.go中被调用：



在请求进入server/rest/controller/handler.go时，roa.GetRouter().ServeHTTP(w, r)将执行上图中的ServeHTTP()方法，具体请见上面“其他路由注册-server/rest/controller/handler.go”的说明

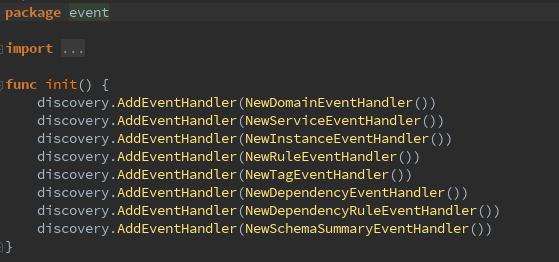
**三. 服务启动流程**

在进行完初始化与引导流程后，Service-Center将正式启动服务。

1服务发现事件监听

**server/service/event/event.go**

Service-Center中使用了事件，让注册与发现实现纯异步，在启动服务的时候，对服务发现引擎做了事件的监听：



上图中对Domain、Service、Instance、Rule、Tag、Dependency、DependencyRule、SchemaSummary这些资源进行了事件的监听，当watch到服务发现系统的变化，将在服务内部派发出对应的事件，以供事件监听器处理。

2服务启动

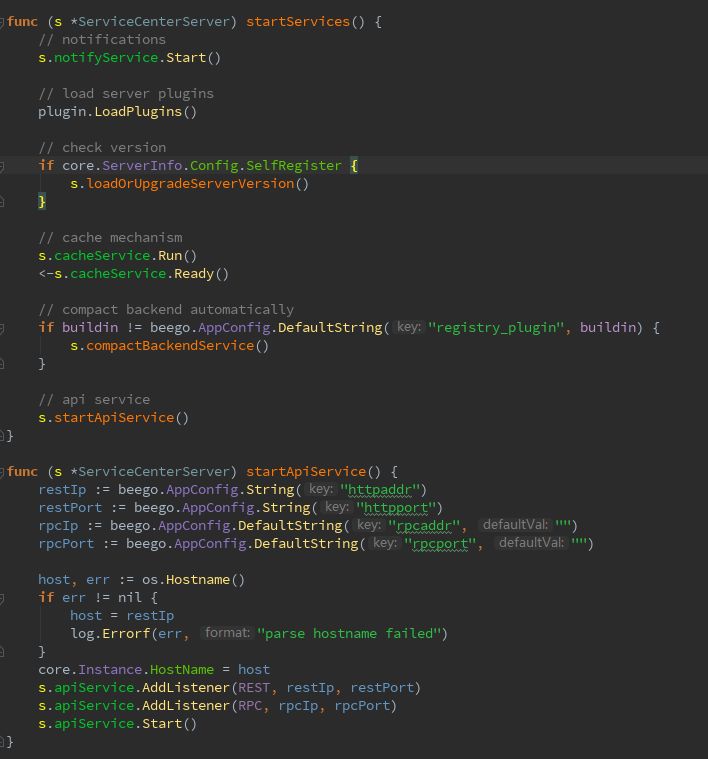
**server/server/server.go**

•initialize()



•初始化缓存服务（cacheService）、RESTful API服务（apiService）、通知服务（notifyService）、协程池（goroutine）

•startServices()



•启动通知服务、从配置的插件目录加载插件、按需注册自身服务、启动缓存服务、按需压缩后端存储数据、启动RESTful API服务

•waitForQuit()



等待服务结束，调用stop方法停止已开启的服务，关闭协程池、服务注册系统，并执行日志落盘。