Enterprise Light Engine Next Generation

EIGHT 新一代企业应用平台

J. W.

2023年7月12日

- 什么是Eight
- 从单体应用到微服务
- Eight——超越之云
- Eight的应用场景
- Eight相关资料

什么是Eight

Eight是什么

- Eight是一种设计思想

- Eight是一种架构规范

- Eight是一种运行平台

- Eight是一种软件工程方法

Eight用来做什么

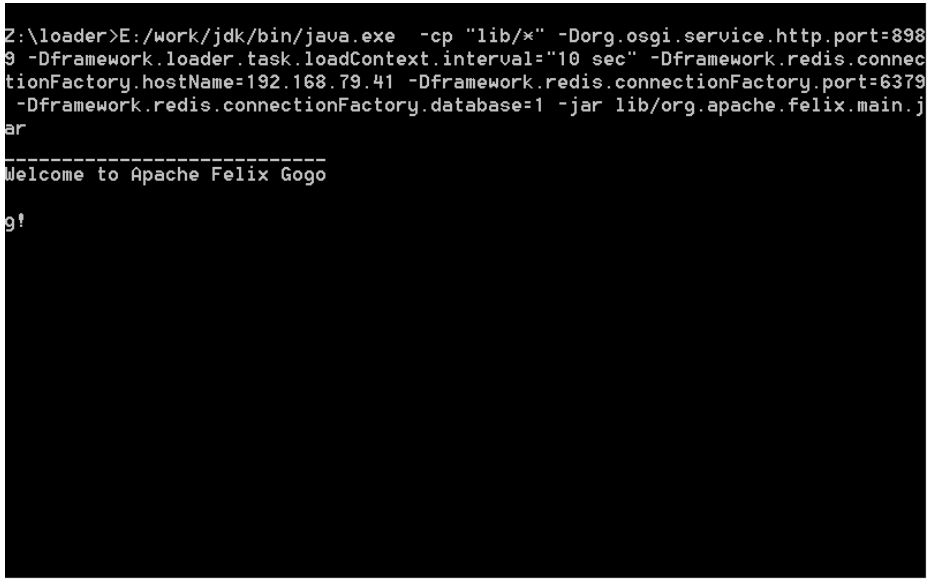
- Eight是为企业化应用开发而设计的，解决在企业内部运行的系统日益膨胀和复杂的状况下，企业私有化系统所遇到的各种复杂问题
  - 如何简单方便的部署和运行，对环境依赖和资源消耗尽可能少
  - 如何随业务的变化迅速的改变或调整系统，而对企业的影响尽可能少
  - 如何更好为企业堆积和积累业务和数据资产，使企业在构建新的业务系统时多快好省
  - 如何便捷有效的为企业监控、管理、升级和维护其私有系统，为企业节省信息化的人力物力投入
- Eight 应用领域广泛，相较于当前各类解决方案具有显著优势
  - 2B服务的私有化部署
  - 大型企业的广域应用发布和维护
  - 中小企业的轻量级云计算
  - 企业的离散节点与边缘节点管理
  - 企业的传统系统集成和数据集成
  - 云容器集成以提供业务层虚拟化
  - 业务组件化和组件仓库
- Eight是单体应用或微服务的替代解决方案

Eight初体验

- Eight线上的演示系统
  - https://www.yeeyaa.net/
- 注册账号并下载底座

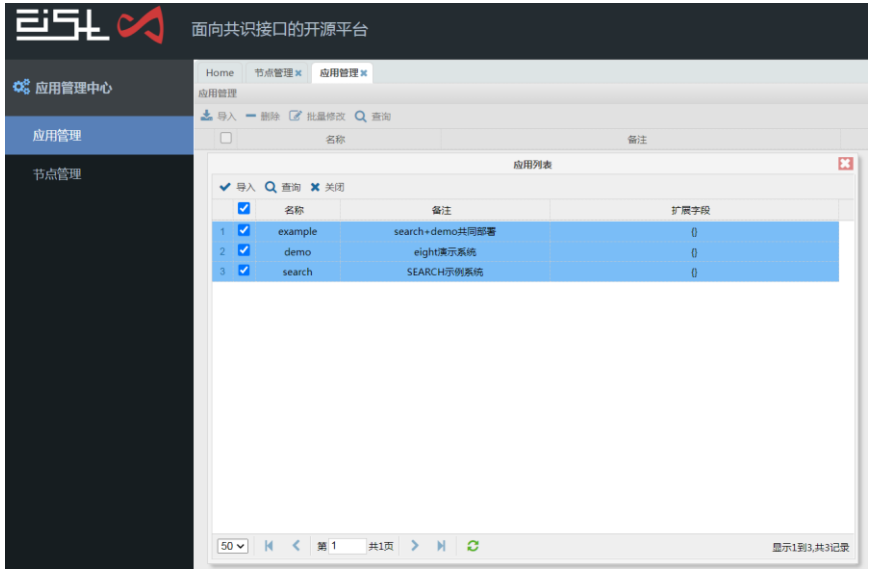
运行底座

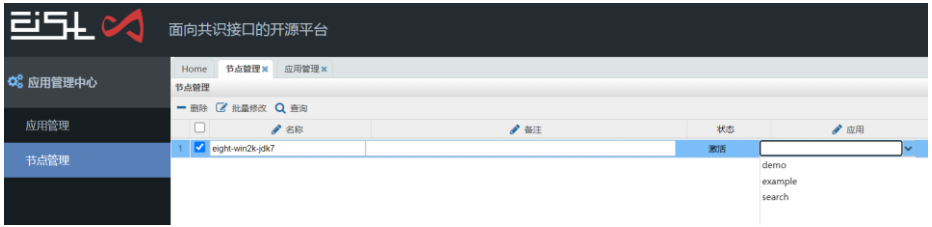
- Eight底座是一个jar包，需要jre 1.6（线上演示环境需要jre 7）以上运行环境
- 如果java版本在17以下，运行命令
  - java -Dframework.boot.scanner.node=你的节点名称 -Dfile.encoding=UTF8 -Dframework.web.user=你的用户名 -Dframework.web.password=你的密码 -Dframework.web.url=https://www.yeeyaa.net/api -jar eight-seat-1.0.0.jar
- 如果java版本在17及以上，运行命令
  - java –add-opens java.base/java.lang=ALL-UNNAMED -Dframework.boot.scanner.node=你的节点名称 -Dfile.encoding=UTF8 -Dframework.web.user=你的用户名 -Dframework.web.password=你的密码 -Dframework.web.url=https://www.yeeyaa.net/api -jar eight-seat-1.0.0.jar
- 注意红色字体部分填写你刚才注册的用户名和密码，以及你想设置的节点名称（默认是IP地址）



发布应用

- 登陆系统
- 在应用管理里导入几个预设的应用
- 在节点管理里找到刚才的节点，给它绑定一个应用

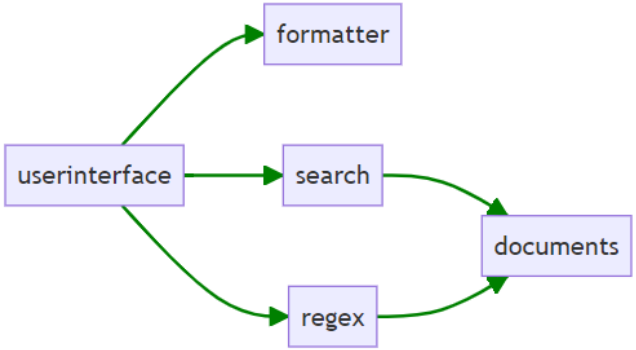




运行系统

- 默认设置下，发布的应用将会在最长5分钟（300秒）以内加载
- 先选择search应用
- 加载时容器会有相应输出
- 然后访问 http://localhost:7241/user/index.html
- 在运行目录下创建一个scores目录，然后放置几个txt文件进去
- 输入一个关键字搜搜

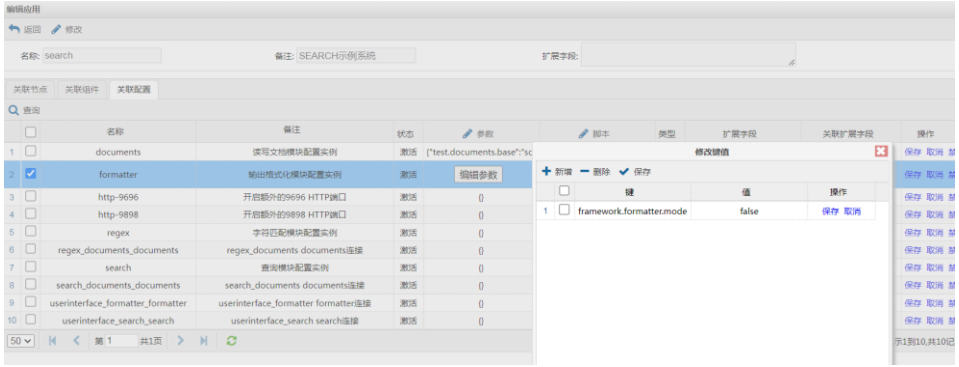
Search的组件结构

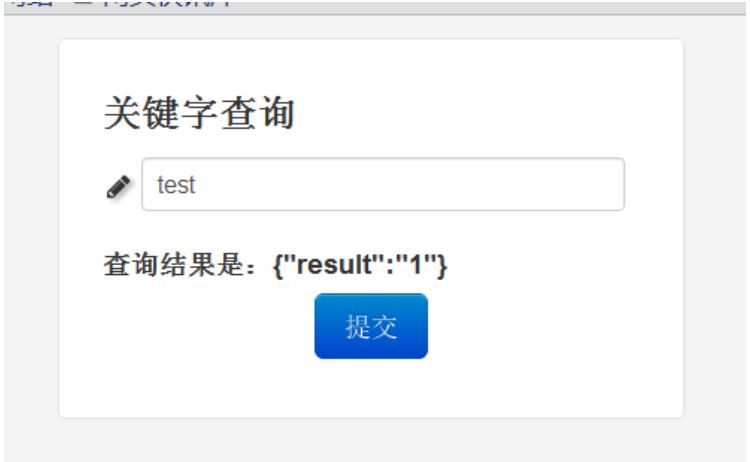


- userinterface组件是界面UI，也即上图我们看到的操作界面
- userinterface根据配置，选择使用search或是regex来进行字符串检索，两种算法匹配模式不同
- search和regex则共同连接documents模块，此模块负责提供前者需要的文本
- search和regex将检索的结果返回给userinterface
- userinterface将使用formatter对结果格式转化后，提供给用户

修改组件设置

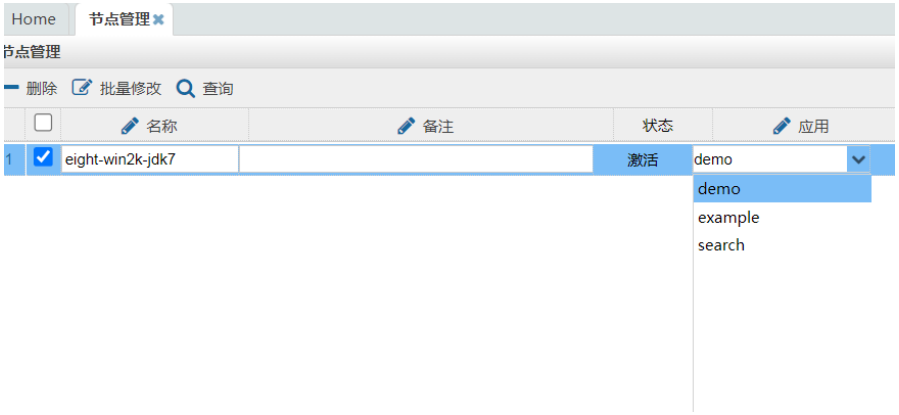
- 试着调整formatter组件参数
- formatter主管输出结果的格式
- framework.formatter.mode参数默认为false，输出为xml，当改成true时，输出为json格式
- 修改之后稍候，等待控制台有输出后，再执行相同的请求
- 可见输出结果不变，但是格式发生变化



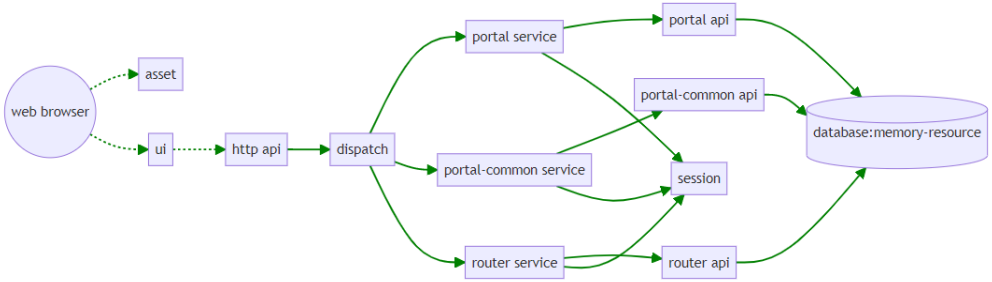


换一个系统

- 在节点管理里选择demo应用
- 加载时容器会有相应输出
- 随后原有页面无法打开，原有系统已经卸载
- 然后访问 http://localhost:7241/ui/index 新的系统已经加载



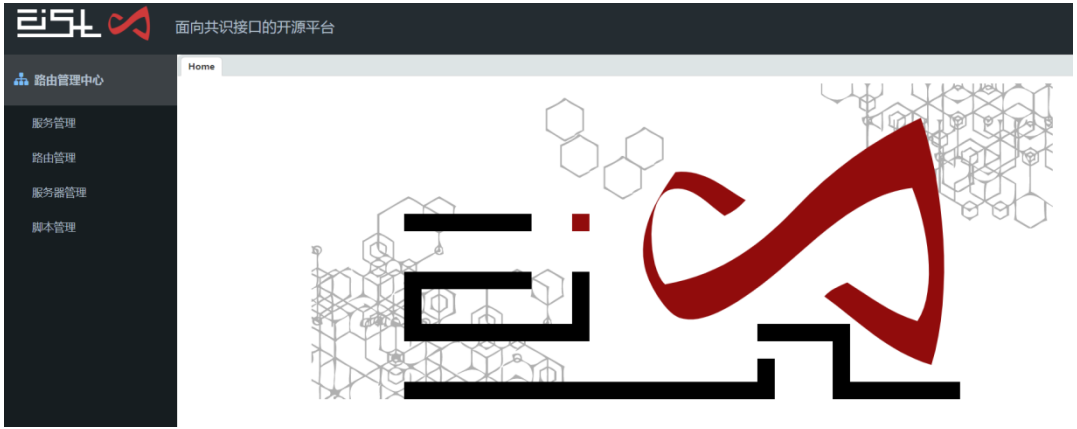
Demo的组件结构



- 用户浏览器访问前端，也即上图我们看到的操作界面
- 前端两个组件，其中asset组件是通用共享资源，包含大量前端常用静态资源（如js,css，图片等），与其它组件并无关联
- ui组件则是业务系统的ui界面，包含controller和view，以及本模块会用到的静态资源
- ui组件并不直接连接其它组件，而是通过http(s)调用，访问http api组件，该组件会开放一个http端口，向外提供api接口，以实现前后端分离
- http api组件下游是dispatch组件，它根据请求路由到不同的服务模块
- service组件是通用服务模块，它生成三个实例：portal service、portal-common service和router service，分别对应portal、portal-common和router三个业务组件
- router业务组件实现的就是路由管理中心这套业务，包含前述服务管理、路由管理、服务器管理和脚本管理等功能，而portal和portal-common则是进行用户、菜单和功能权限管理的业务（不提供操作界面）
- 各个service组件需要连接session组件来存取会话信息
- portal、portal-common和router需要对接memory-resource，memory-resource包含一些内存的存储结构，甚至还有一套内存数据库，用来提供一套完整的mvc业务系统
- 为了优化用户体验，DEMO不需要用户配置数据库，而是自行提供了一套内存数据库

关闭sql输出

- demo是一个标准的crud企业应用，可以试试增删改查
- 操作数据库时，控制台会有sql输出
- 数据库操作，由portal、portal-common和router-api这几个模块管理
- framework.router.api.sessionFactory.hibernate.showsql设置为false，则关闭sql输出



22	☐	demo-portal.ext_session_session	portal.ext和session的	激活
23	☐	demo-portal_redis.store_redisRes	portal和redis.store的	激活
24	☐	demo-portal_redis.userauth_stor	portal和redis.userau	激活
25	☐	demo-portal_service.portal_next	portal和service.port	激活
26	☐	demo-portal_session_session	portal和session的se	激活
27	☑	demo-router.api	路由管理api实例	激活

新增 删除 保存

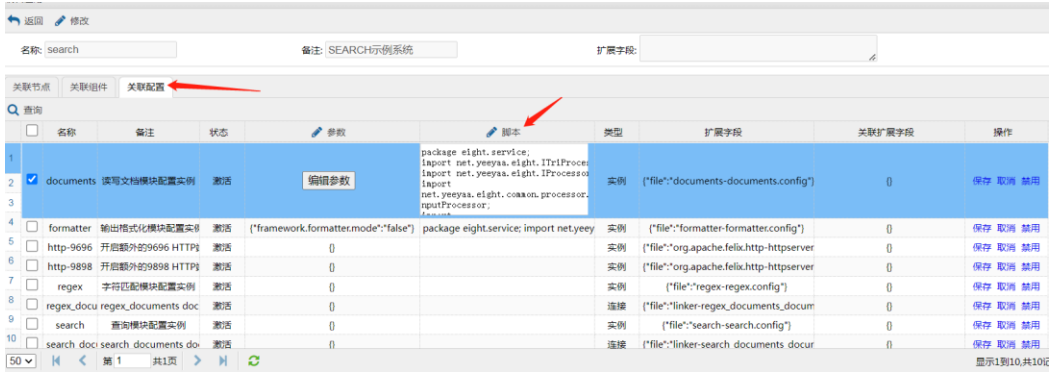
		键		
1	☐	framework.router.api.dataSource.url	jdbc:h2:mem	保存
2	☐	framework.router.api.server	mresource||	保存
3	☐	framework.router.api.dataSource.driver	org.h2.Drive	保存
4	☑	framework.router.api.sessionFactory.hibernate.showsql	false	保存

给dispatch组件附加脚本

- 可以在实例和连接的切入点上给系统附加groovy脚本，例如以下脚本会将组件间的调用输入输出打印出来

```
package eight.service;
import net.yeeyaa.eight.ITriProcessor;
import net.yeeyaa.eight.IProcessor;

class Proxy implements ITriProcessor {
   IProcessor context

                def operate(Object first, Object second, Object third) {
                        println "instance[" + context.process("hookid") + "] input: " + first + " " + second + " " + third;
                        def ret = ((ITriProcessor)context.process("next")).operate(first, second, third)
                        println "instance[" + context.process("hookid") + "] output: " + ret
                        ret
                }
}
```

给dispatch组件附加脚本




```
instance[null] input: preprocessor process {
   "name" : "portal@login",
   "content" : {
      "@type" : "ptUser",
      "name" : "admin"
   }
}
Hibernate: select ptuserenti0_.id as id1_9_0_, pttenanten1_.id as id1_8_1_, ptuseren
instance[null] output: {
   "name" : "portal@login",
   "token" : "5t5LU_dkS6uha6LcSSLsVg",
   "content" : {
      "@type" : "boolean",
      "value" : true
   }
}
```

增加功能模块

- 在DEMO中，已经配置了几个组件，但被禁用了。我们可以在组件管理中启动它
- 稍候底座会更新，系统结构也随之变化
- 新增的monitor的一组实例（monitor、monitor-ui、monitor-service、monitor-api）被加载到系统中

这组实例为系统带来了新的功能——监控模块

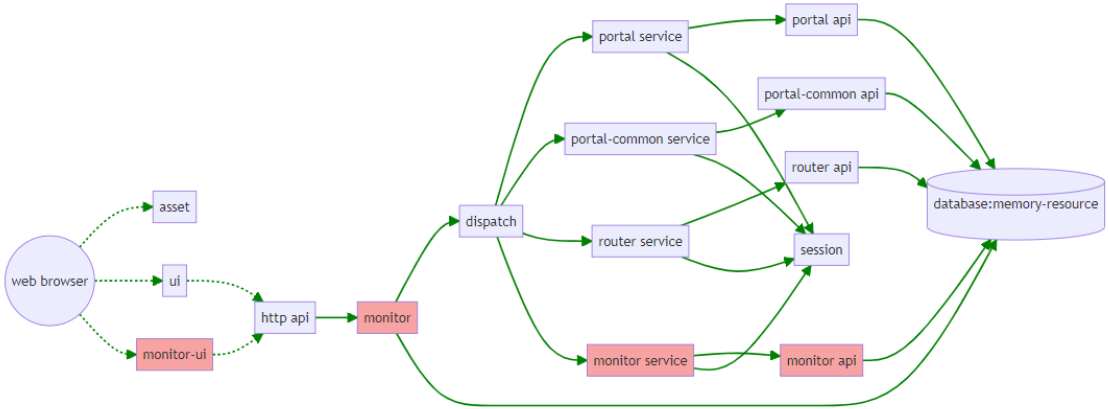
关联节点 | 关联组件 | 关联配置

查询

	名称	备注	状态	扩展字段	关联扩展字段	操作
1	demo-asset	前端共享静态资源库	激活	{"file":"eight-osgi-assets-1.0.0.jar"}	{}	禁用
2	demo-dispatch	服务分发模块	激活	{"file":"eight-osgi-dispatch-1.0.0.jar"}	{}	禁用
3	demo-http	后端api的http接入服务	激活	{"file":"eight-osgi-http-1.0.0.jar"}	{}	禁用
4	demo-memory-resou	内存数据库	激活	{"file":"eight-osgi-memory-resource-1.0.0.jar"}	{}	禁用
5	demo-monitor	接口监控服务	禁用	{"file":"eight-osgi-monitor-1.0.0.jar"}	{}	激活
6	demo-monitor-api	监控管理api	禁用	{"file":"eight-osgi-monitor-ext-1.0.0.jar"}	{}	激活
7	demo-monitor-ui	监控管理ui	禁用	{"file":"eight-osgi-monitor-ui-1.0.0.jar"}	{}	激活
8	demo-portal	portal基础服务api	激活	{"file":"eight-osgi-portal-1.0.0.jar"}	{}	禁用
9	demo-portal-commc	portal通用服务api	激活	{"file":"eight-osgi-portal-common-1.0.0.jar"}	{}	禁用
10	demo-router-api	路由管理api	激活	{"file":"eight-osgi-router-api-1.0.0.jar"}	{}	禁用
11	demo-service	api服务组件	激活	{"file":"eight-osgi-service-1.0.0.jar"}	{}	禁用
12	demo-session	会话组件	激活	{"file":"eight-osgi-session-memory-1.0.0.jar"}	{}	禁用
13	demo-ui	交互ui模块	激活	{"file":"eight-osgi-ui-1.0.0.jar"}	{}	禁用

50 第 1 共1页 显示1到13,共13记录

监控模块的组件结构



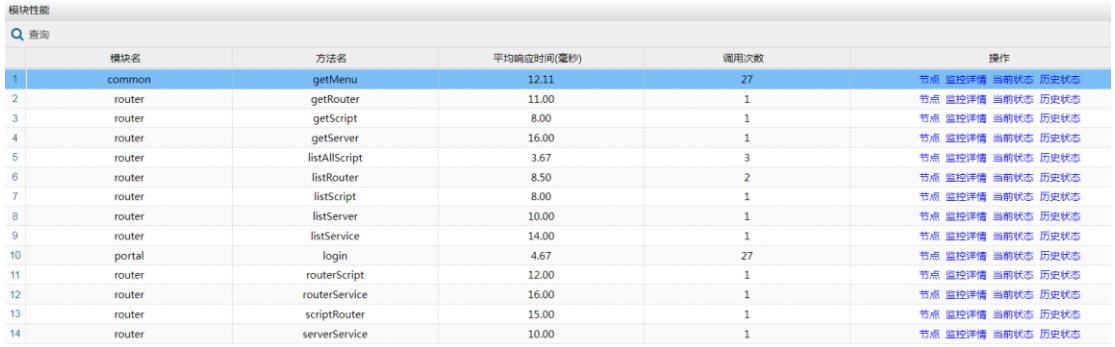
- 红色部分模块为新增组件，它们被嵌入到原先的系统各个部分
- 最核心的组件是monitor，它在系统中选取了一个最为合适的切入点：嵌入到http与dispatch之间，所有的系统调用均经过此处。它同时也连接database。它的功能是记录并缓存方法调用的状况，并将其批量写入数据库
- monitor-ui为监控管理前端，也即新出现在用户操作界面上的功能模块。
- monitor-service、monitor-api与router类似，也就是monitor模块进行增删查改的后台api

使用监控模块

◆系统升级完毕后，我们可以随便进行些操作，这些调用的次数、发生时间和持续时长会被记录下来。然后，我们可通过monitor界面查看其情况

◆如果不再需要哪些功能（router、monitor），可以将对应组件或实例禁用，则这些组件将会被系统移除，功能随之消失，资源也会释放

◆所有以上应用都是纯粹的本地应用，并不依赖网络而运行。不论是关闭java虚拟机、还是重启操作系统，再次启动节点时，节点仍维持最近的系统状态

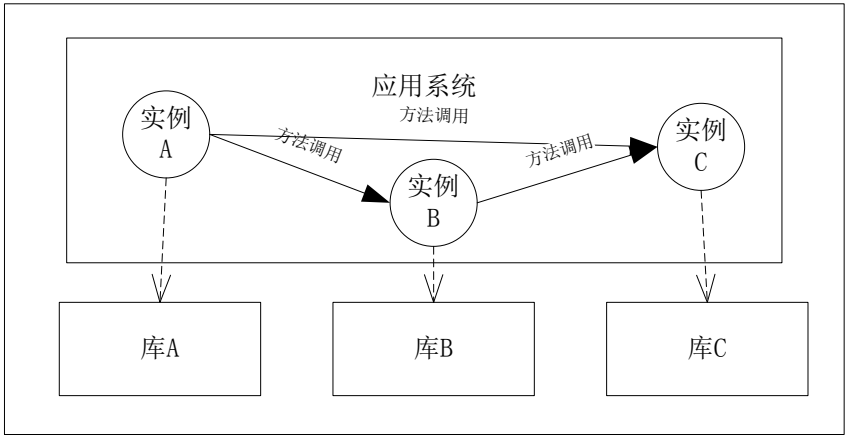


从单体应用到微服务

传统单体应用的弱点

Tomcat、Spring Boot

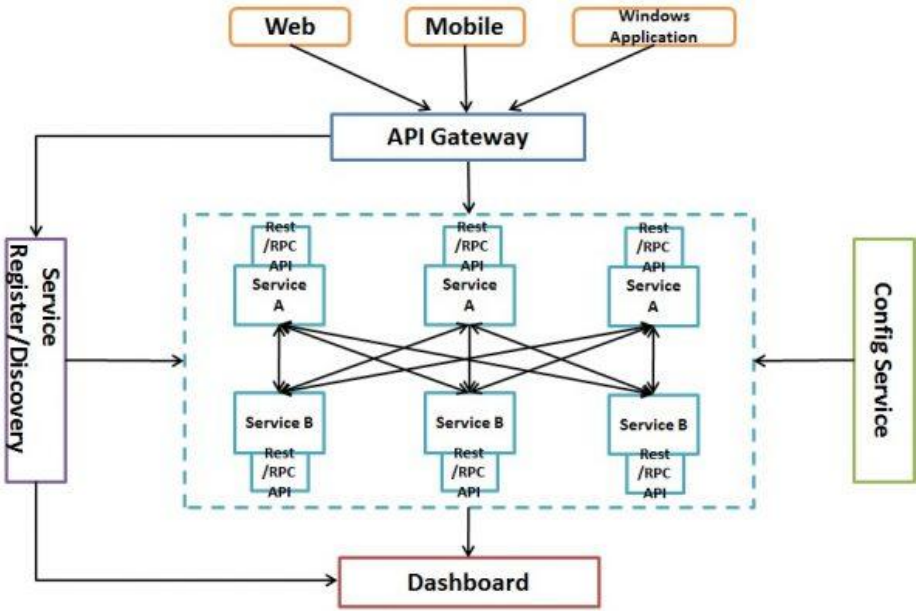
- 单体应用是最传统的应用架构
  - 开发简单，人员技术要求低
  - 部署和维护容易，单独一个进程启动即可
  - 环境和资源要求低，无须复杂的支撑环境，对网络、计算和存储资源要求低



- 痛点
  - 任何变更都需要在代码上进行，业务或依赖库的变化，都可能会导致代码本身的变化；每一行代码的变化，都需要重新集成、编译和部署
  - 库的依赖被集成到各个应用系统之中，导致库的升级和重构会影响到各个系统
  - 代码集成的依赖导致模块缺乏隔离，局部模块的问题影响到整体系统，并且提升了冲突的可能，系统日趋复杂和难以维护
  - 对于大型系统，单个模块变化需要整个开发组织沟通和同步，不便于分组开发和集成，难以提升研发并行度和开发效率
  - 当业务增长时，不易动态伸缩和扩容，而且部署和运维消耗大
  - 当需要远程大量部署和运维时尤为不便，每一次微小迭代都得全量发布，往往又应用体积庞大而发布节点太多（类似于OTA）而难以实施，并且需要长时间中断业务

微服务的优势

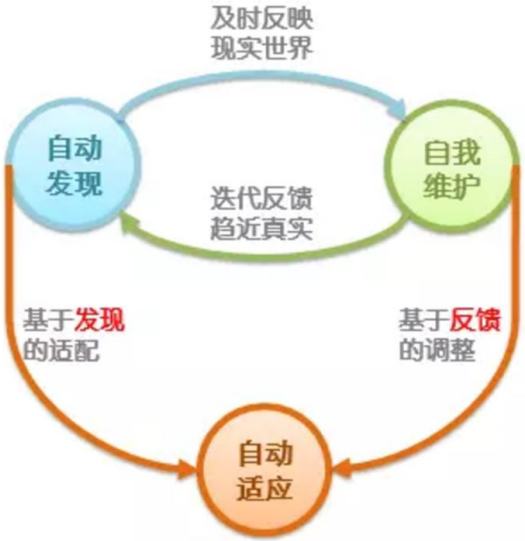
模块化



- 微服务的最大特征就是模块化（服务化）
  - 迫使开发者用一种模块化思想分解业务领域
  - 迫使开发者在设计之初就划清模块的边界
  - 迫使开发者规范各个模块的功能、接口定义和输入输出
  - 迫使开发者随着业务系统的发展不断的抽象和抽取新的公共模块
- 模块化的好处
  - 问题分而治之
  - 大型系统的分工与协作
  - 以及对互联网式开发模式有着天然的适应

微服务的优势

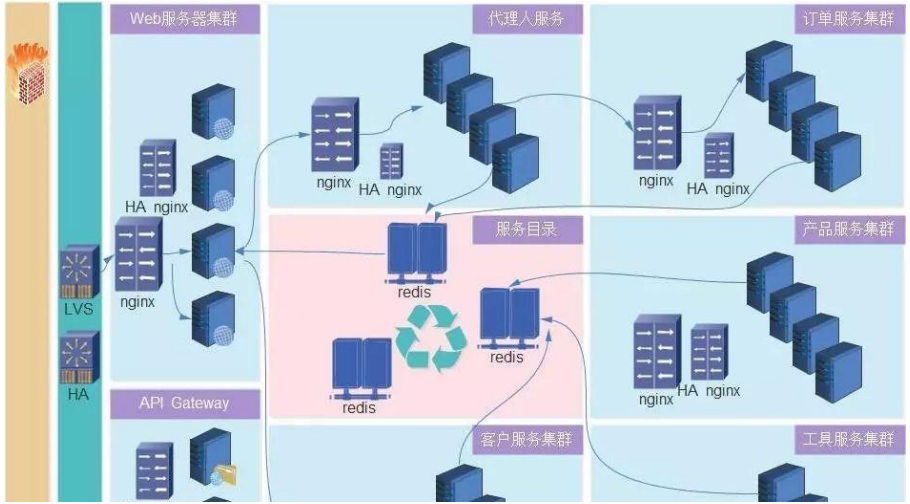
快速开发、局部部署与微创迭代



- 互联网行业软件的生命周期趋短，需求变化与迭代趋于频繁
  - 公用的服务组件能为新生业务快速搭建起基本框架
  - 不断变化的需求体现为某一局部模块的迭代需求
  - 微服务将变化隔离在某一局部模块内，对于快速响应提供便利
  - 微服务能在总体系统不变的情况下局部部署新的模块，可以把系统迭代造成的代价大大减小（分钟级更替）
  - 微服务的整个迭代过程，都是微创式的，造成的“伤害”也是相对微小的

微服务的优势

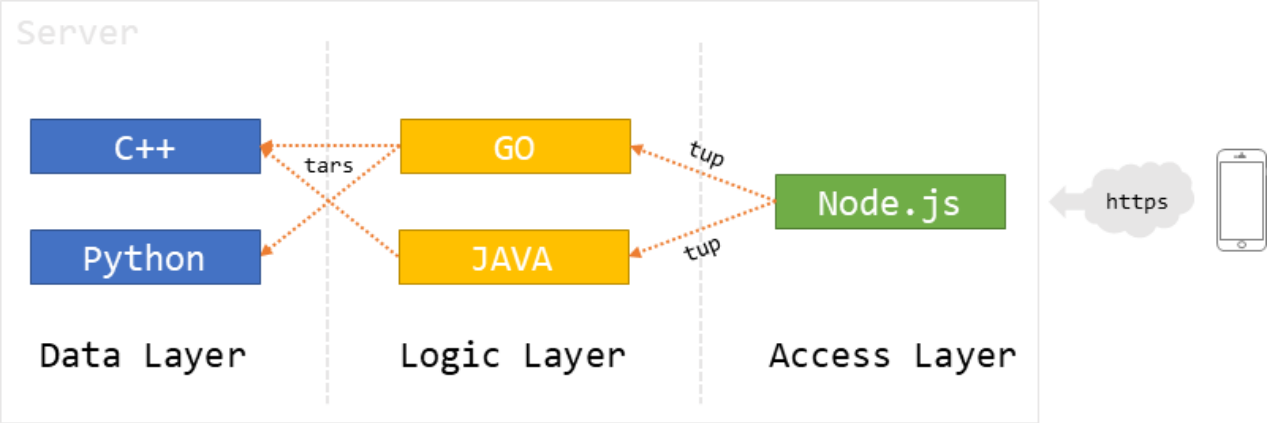
弹性伸缩、局部扩展与适应需求



- 互联网行业的业务密度与计算性能不可控的问题一直困扰着架构师们
  - 不同业务、不同发展阶段、不同时间点，不同模块的计算与响应的规模可能大不相同
  - 动态的应需增减集群能力显得非常重要
  - 微服务将业务划入不同的模块，当模块过热（持续高强度响应）时可以动态增加处理节点，当模块趋冷时可以回收计算资源。同时，由于服务模块相对较小，使得这些资源能有效应用在关键的计算逻辑上

微服务的优势

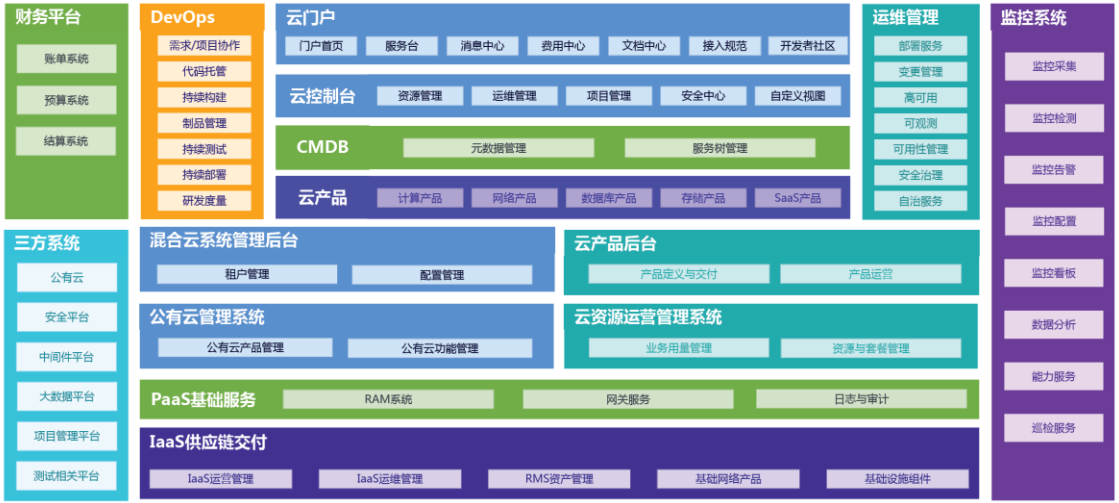
系统异构与引进先进技术



- 微服务的每个服务单元，只要规定了接口和输入输出，其技术实现可由任意的技术框架来完成
  - 很容易用更先进有效的技术手段来提升某个独立模块的服务能力，从而提升整体系统的服务指标
  - 很容易在其它环境不变的情况下，分析和比较不同技术实现在服务能力上的差异，从而对技术选择提供实践依据
  - 很容易在系统关键部位，置换上优势技术模块（如python之于数据分析、go之于网络通讯），从而提升整体系统的能力

微服务的缺陷

系统复杂化和零散化



某厂云原生架构图，感谢大牛朋友们的无私奉献

◆微服务的小粒度组件化和模块化，让一个系统变成了几十、几百甚至更多部件的系统。每一个需要独立配置、部署、运行，然后还要彼此关联

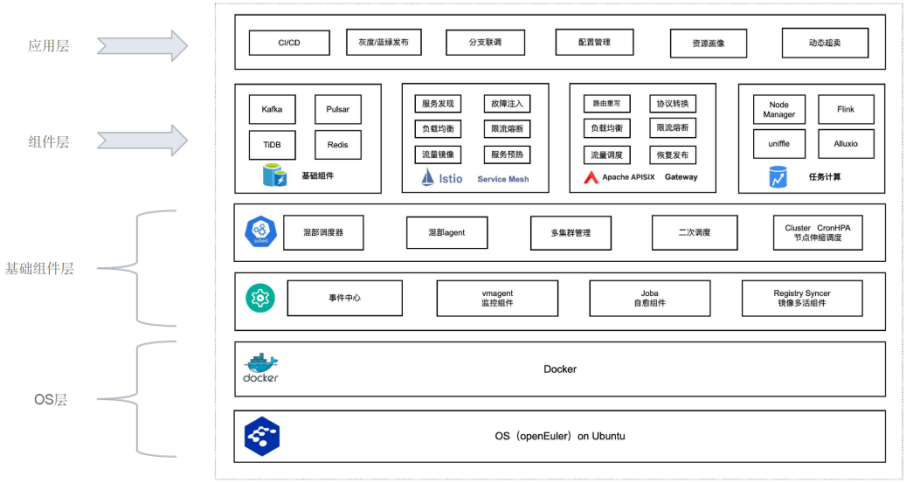
◆Docker，k8s，istio等技术能一定程度减少这种复杂度，智能运维能够一定程度协助排查故障，但配置与部署、服务关联、日常维护、故障处理仍需消耗大量人工

◆微服务的广泛依赖使得系统对外界环境变得敏感，很多确定性问题由于服务化变得不确定，同时，使得问题的追踪与定位变得复杂

◆大量异构子系统需要运维人员对各种技术都有一定程度的理解，对人员要求较高

微服务的缺陷

云环境本身会带来问题



某厂云原生架构图，感谢大牛朋友们的无私奉献

◆微服务本身就会带来大量额外的模块，如服务注册与发现模块，用于异步服务请求缓存的消息队列，服务链路分析模块，服务的熔断与降级模块，服务的观测与故障追踪模块，分布式事务调度器等等等等，这会更加加大系统复杂度

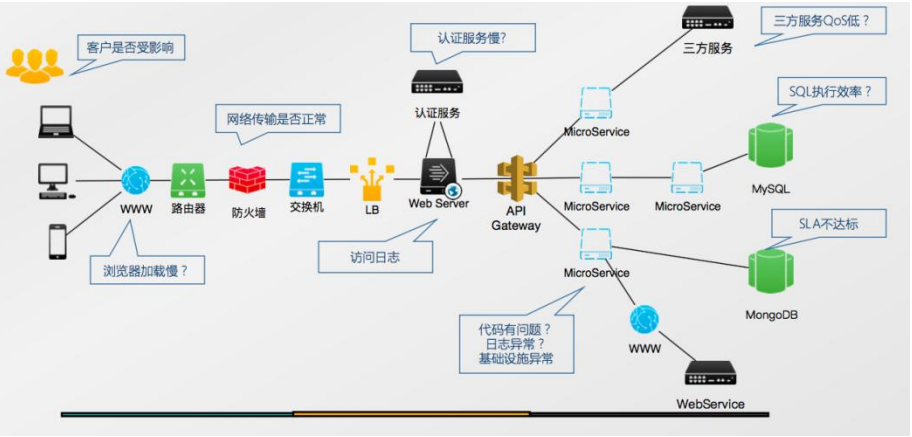
◆微服务的发展使得未来软件体系架构变成云原生（如Service Mesh，FaaS，Serverless），更加不能脱离云基础架构，难以复制迁移和广泛使用

◆云原生又更将带来额外的更多模块，如多云混合部署调度，FinOps成本管理，AIOps运维智能等等等等，这些东西原本是微服务本身带来的，并非业务需求，而他们本身的可用性和可靠性也需运维来保证，系统复杂度在云化过程中螺旋上升

◆而云基础架构建设、开发、维护、运行成本高昂，并且需要高质量团队支撑管理和持续优化，绝非大多数企业能够实施得起

微服务的缺陷

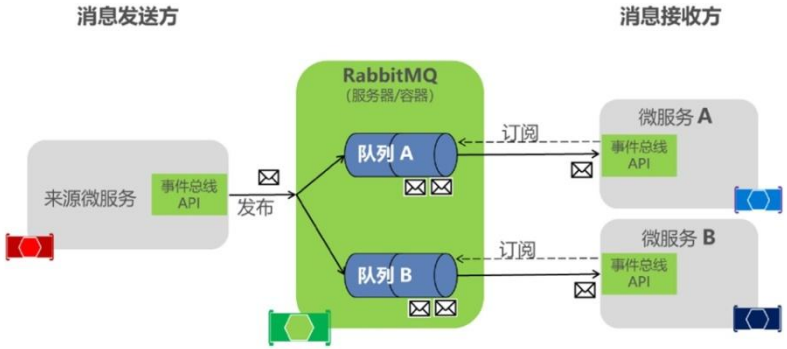
微服务并没有完全解耦依赖，反而使之更为复杂



- 过度服务化带来了系统结构的复杂联系，造成服务依赖与故障追踪难题
- 微服务初衷是通过服务化解耦模块关联，使得系统局部的调整不会影响整体，但实践中这种依赖往往会内化到服务的内部逻辑中（潜依赖），以至于影响范围难以评估，核心服务重要到了复杂而不能改的地步
- 原本无关的模块因为各种间接联系而彼此产生相关，原本用破除耦合的微服务架构却因为共同服务而耦合
- 服务的治理与故障的定位成为一个难题，对于模块问题的排查需要对其依赖链的进行追踪
- 广泛的依赖模式下，模块的影响范围难以评估，模块的修改、模块的故障等造成影响难以控制

微服务的缺陷

微服务对系统稳定性的冲击



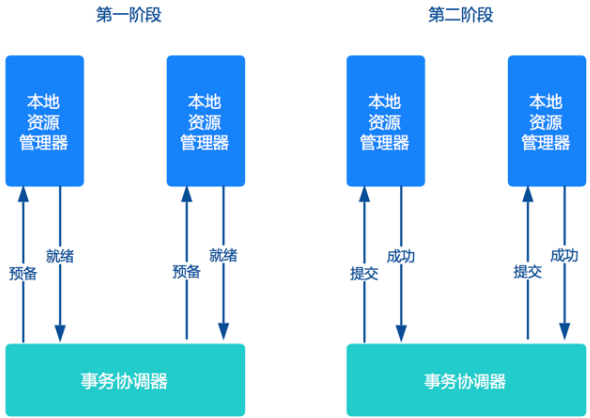
◆微服务的服务自治架构，导致任意时刻都需要处理服务的不可获得问题

◆通过消息队列进行业务请求的缓存，只在没有强事务要求和没有服务响应速度要求的场景下使用，不能保证服务质量和最终服务的达成

◆通过服务的熔断或降级，则要求所有的业务单元都需要处理相关业务逻辑，或载入处理模块，增加了开发和测试的复杂度。同时，熔断和降级并不解决服务不可获得的问题，只是提供了劣化的服务响应（或仅仅提供了服务异常的提示）

微服务的缺陷

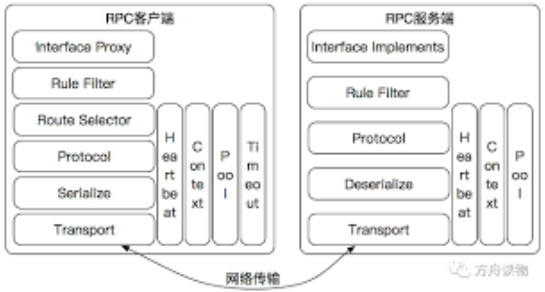
微服务对业务实现的冲击



- 微服务将一个业务分割成若干个服务阶段，数据与业务的分裂，事务一致性难题
  - 直到所有的服务完成，该业务才算完成
  - 任何一个服务不能够控制和管理其他服务的状态
  - 各个阶段的模块分别管理自身的数据
  - 要求这些数据在事务级别处于一致的状态
  - 结果是一个简单事务变成了分布式事务
  - 事务的完成或回滚需要除开业务模块外的其他模块（如事务协调器）来保证
  - 事务的多阶段提交提供了额外的复杂度和处理开销
  - 即便如此也不能保证事务能被顺利处理

微服务的缺陷

微服务对系统性能的冲击



◆微服务要求各模块使用进程间调用方式连接

◆实现上微服务基本上使用网络通讯协议进行互通（http restful，rpc等等）

◆Rpc过程（串行化、网络交互、故障重连、反串行化）带来大量的消耗

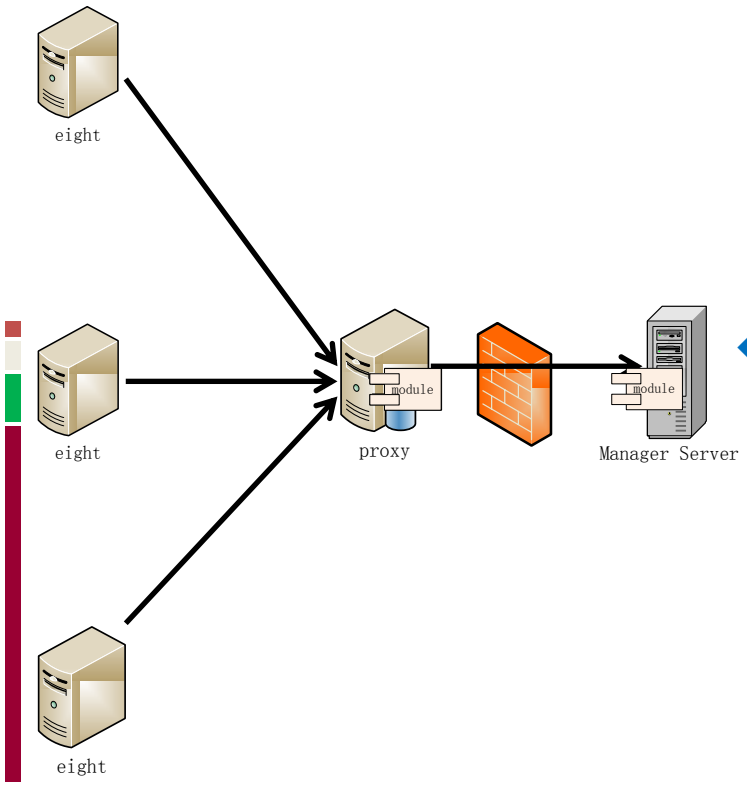
◆微服务将以前用内存进行传输的数据（单一系统的模块间调用）变更为网络间传输，导致微服务架构的系统对于网络带宽与网络稳定性要求飞快上升

◆网络本身的问题影响的范围不再局限于客户端与服务端之间的请求与响应。业务的每一个步骤，都有可能因为网络的故障带来各种失败

Eight——超越之云

Eight——更便捷的云

兼具微服务和单体应用的优点



◆相对于单体应用

◆系统任何一个局部都可以独立成组件，独立开发、部署、运维，能够局部更新

◆组件间完全解耦，毫无依赖，任何部分的变化不影响其他部分，能够快速迭代

◆弱化沟通，适应于大型系统的分组开发和分散开发，节省研发成本

◆容易扩展节点和部署、调整应用，适用于弹性扩容管控

◆动态在线更新，完全无缝切换，不存在服务离线状态

◆可轻易的将应用系统发布至成千上万节点上，且应用体积极轻，单个组件体积往往仅有几kb到几十kb

◆相对于微服务

◆非常轻量的集群节点设置，只需一行命令运行一个jar包，几乎没有技能要求

◆无须沉重的云平台环境，没有各种复杂的支撑服务，对于一般企业相当容易实施

◆比微服务更轻更动态，组件粒度比服务更细，能更好切割系统，系统可观测性更强，动态更新更快，资源消耗更少

◆面向共识接口开发，弱化沟通，能够更好的实现模块解耦，使系统更容易变化

◆与容器化云环境结合使用时，能够将大量以往切割成微服务的模块组合在一个节点中，减少微服务数量，简化系统结构和复杂度

◆具备单体应用的优势特征，不存在组件不可获得性问题，能够用统一的事务管理整个业务流程，不存在进程间调用，节省序列化和网络操作等消耗

Eight——更轻量的云

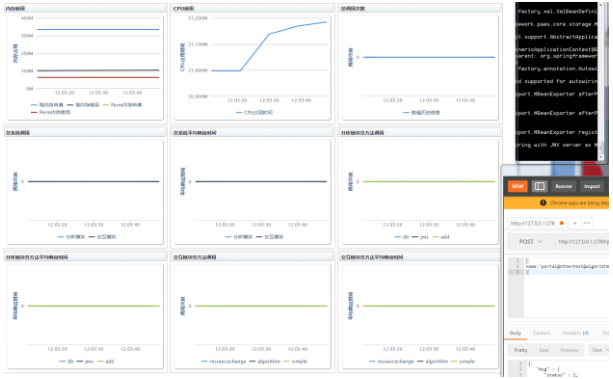
更细的粒度、更强的可观测性、更少的资源消耗

```
public class Formatter implements IProcessor<String, String>{
    protected volatile boolean mode = true;

    public void setMode(boolean mode) {
        this.mode = mode;
    }

    @Override
    public String process(String input) {
        if (mode) return Json.createObjectBuilder().add("result", input).build().toString();
        else try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = factory.newDocumentBuilder();
            Document document = db.newDocument();
            document.setXmlStandalone(true);
            Element result = document.createElement("result");
            result.setTextContent(input);
            document.appendChild(result);
            TransformerFactory tff = TransformerFactory.newInstance();
            Transformer tf = tff.newTransformer();
            tf.setOutputProperty(OutputKeys.INDENT, "no");
            StringWriter writer = new StringWriter();
            tf.transform(new DOMSource(document), new StreamResult(writer));
            return writer.toString();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

◆更细的粒度

◆代码级粒度，系统中任何一个模块，哪怕只有数行代码，都可以孤立出来作为组件，对其动态配置、修改和加载，或被多个组件共用

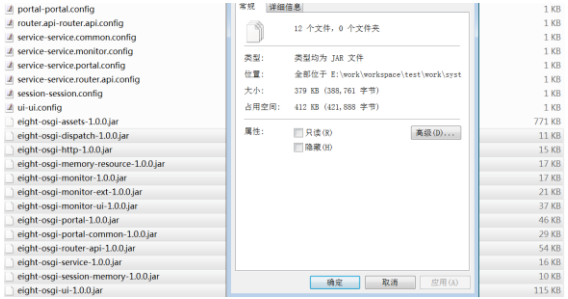
◆更能从业务角度切割，而非微服务那样将业务切割、性能切割、技术条件切割、资源环境切割混合在一起



◆更强的可观测性

◆观测能力渗透到运行时的代码内部，能够就任何一个组件观察其运行时状态和参数，每一次输入和输出

◆更能对运行时每一个组件进行日志记录、数据抓取、业务重放、在线调试等操作



◆更少的资源消耗

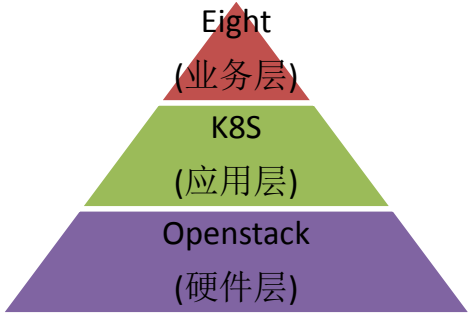
◆与微服务需要重启进程不同，Eight的动态加载是一个线程内的方法调用，组件切换是个毫秒操作，且新业务的加载不影响旧业务的执行，真正无缝更新

◆与微服务每次更新都需要打包和发布整个镜像不同，Eight的发布单元是组件，组件非常轻量，往往只有几KB至几十KB，节省网络带宽同时扩展了应用的可能性

◆更别提Eight无须微服务臃肿复杂的云环境，而Eight底座运行仅需要几十MB内存

Eight——云上之云

Eight与云架构结合



◆结合云环境运行

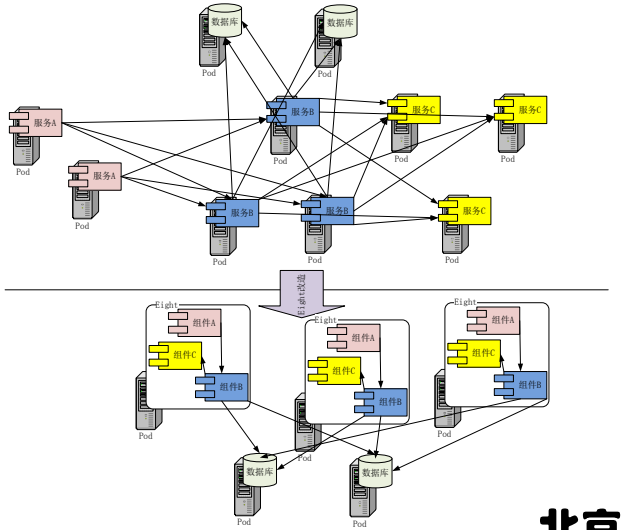
◆Eight天然云原生，一个固定不变的底座，一行命令传入参数即可启动节点，尤其适合在docker上部署，可以自然使用云环境扩展和伸缩节点

◆Eight与云结合之后，以更清晰的层次管理云上系统

- ◆硬件层将硬件虚拟化和单位化成可分配资源
- ◆应用层在硬件资源上启动各种应用服务（Eight、Kafka、ngnix）和数据服务（mysql、redis、hadoop），并弹性伸缩、观测监控
- ◆业务层将系统分割成细小组件，并行开发、局部更新、动态加载、观测监控

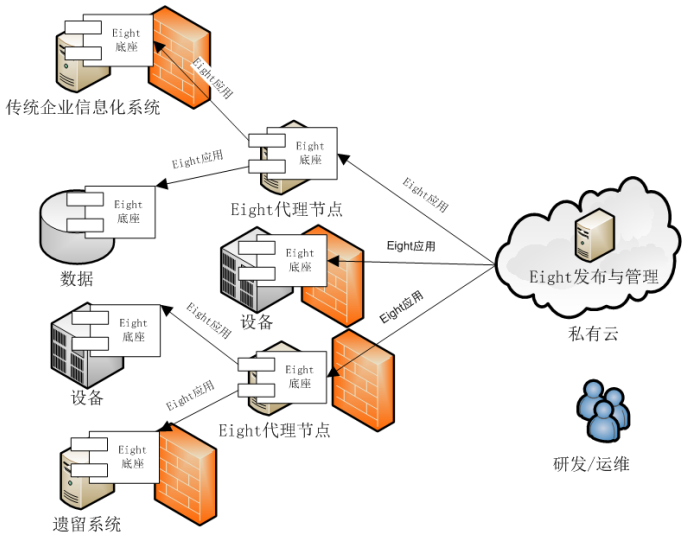
◆扩展云架构的能力

- ◆Eight代替微服务进行业务层切割，将微服务原先不必要的切割省略，大大精简微服务数量，简化系统服务依赖，增强系统的稳健性和可管理性
- ◆Eight给云系统提供了一个新的维度——业务层虚拟化，使得云的管理者可以深入到运行时的业务模块深处，对其进行观测、分析、调试和细粒度管理
- ◆Eight能够以更轻、代价更小的方式跟随业务而变化，并且处理事务、rpc等问题时能够得到更佳的选择



Eight——超越云的云

到达云所不能到达的地方



- Eight能够部署和管理前所未有的系统
  - 企业存在大量分支节点（银行、电信、超市网点，邮局、加油站等大型企业分支机构）
  - 企业存在大量的孤立节点、独立系统和传统应用（工业智能设备、OA办公节点、遗留传统系统）
  - 企业存在大量的边缘节点，分布广泛，难以触达（智能汽车、智能家居、物联网）
  - 这些节点和系统分处各处、本地运行、无法上云、不能集中管理
  - Eight兼容性极佳、资源消耗少、易于部署，能将各种节点纳入到整体环境之中
  - Eight能建立一种新形态的云，包含大量离散节点和边缘节点的云，补完企业的全局系统
  - Eight能对所有的节点，包括集群节点、离散节点、边缘节点，集中管理、发布、运行时更新、观测、运维和数据收集
  - Eight能实现超越现在所有云形态的环境，并孕育新的系统架构和业务方案，将企业的触角渗透到每一个角落

Eight的应用场景

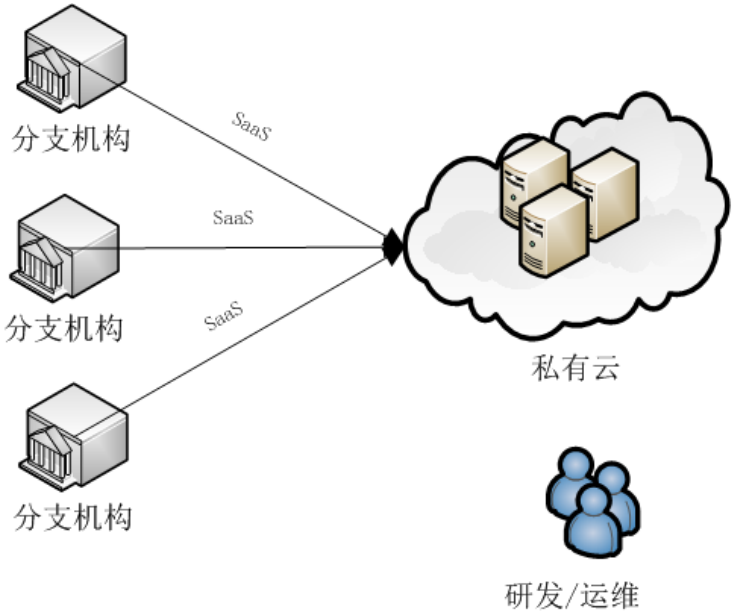
企业应用的常见问题

- 企业应用与互联网应用存在较大差异，企业应用的一些突出痛点
  - 企业应用的业务逻辑往往复杂而独特，且业务也会不断发生变化，对定制和维护的需求较多
  - 企业应用往往分布和扩散在企业的整个体系之中，在各个业务部门、分支机构甚至离散或边缘节点中存在，这与典型的互联网应用将系统集中架设于可控的核心设施（如公有云上）并不一致
  - 企业应用的标准化程度较低，复用程度较差，不能够像互联网应用那样依靠大量的使用者来摊薄开发和维护成本。往往企业应用被定制后唯一的使用方就是企业本身，企业对高昂的研发成本承担能力有限
  - 企业大多不具备先进的信息化技术使用条件（如平台虚拟化和私有云），也缺乏足够的技术人才队伍。一些先进的互联网技术难以落地，更难以管理和维护
  - 企业内部的计算基础设施环境复杂，存在各种边缘节点和信息孤岛，运行着大量的遗留系统和老旧设备，存在着不稳定的网络环境和重重阻隔的安全策略。企业应用系统需要在这样的环境中部署和延伸
- 在中国，因为国情、政策和长期形成的技术传统
  - 中国企业更倾向于使用私有化、本地化的解决方案，尤其大型企业和国企央企，由于政策法规的要求，业务和数据的安全和私有化是核心诉求。这意味着以上所有问题，都不能通过SaaS化的互联网服务而得到缓解
  - 中国地域发展不平衡，技术人员集中于少数几个城市，而企业、尤其是大型国企央企，业务则遍布全国各地。一般说来，企业难以建设技术队伍，更难于在自己的分支机构建设此类队伍

拥有庞大分支机构的全国性企业

超市、邮局、银行、电信、石油、电力

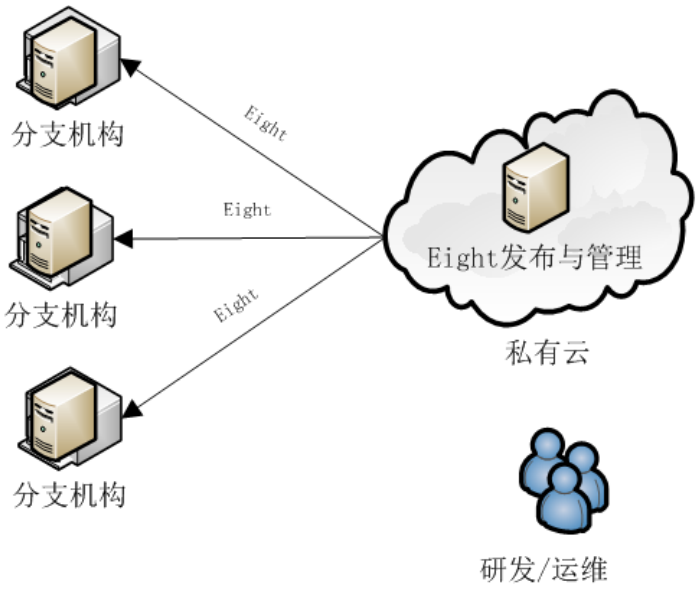
- 有私有云和优良的基础建设，主体系统集中建设在私有云上
- 在中心城市拥有一支有质量的研发队伍，能够研发和维护集中建设的系统
- 大量分布各地的分支机构主要使用私有云上的SaaS系统
- 仍有不少应用不能依赖于网络和SaaS（如各种离线业务系统），此类系统和终端需要管理、升级和维护，此类工作需要投入大量成本建设全国性的技术支撑队伍
- 痛点
  - 对于分支机构的系统维护成本偏高，尤其是当维护涉及软件和业务时
  - 少量的业务研发人员集中于核心城市，而问题却可能出现在各地
  - 中小城市缺乏人才储备，难以解决问题



使用Eight的解决方案后

超市、邮局、银行、电信、石油、电力

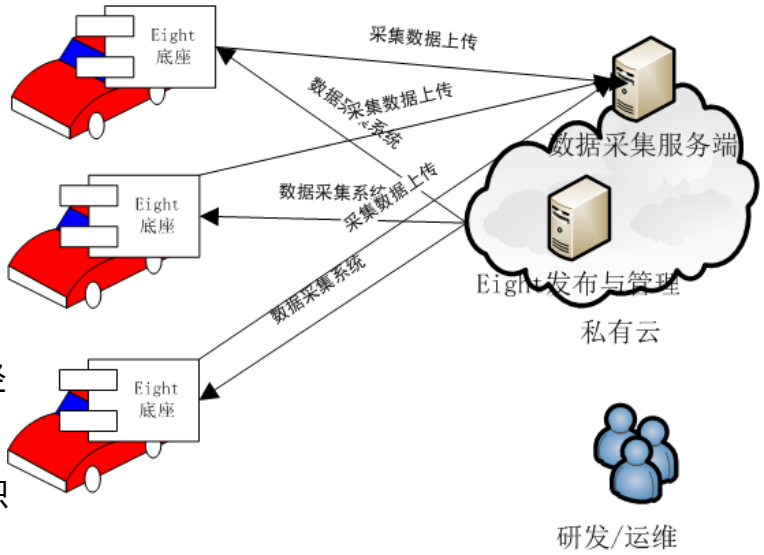
- 建设集中的应用分发和维护体系
- 将业务系统集中研发并部署于中央节点上，分支机构人员采用简单的方式对接中央节点的服务
- 中央节点即可部署和升级系统到本地
- 当出现运维需求时，分支节点连接到中央系统，则研发人员可以在线跟踪、调试和维护这些远程的本地系统
- 相比于以往方案，增加Eight基础设施之后的优势在于
  - 降低运维成本
  - 提高问题和故障响应效率
  - 更容易的接触现场，分析原因，查明故障，提高产品质量
  - 使大量业务不再依赖于SaaS成为可能



拥有大量边缘系统的企业

智能汽车、智能家居、物联网

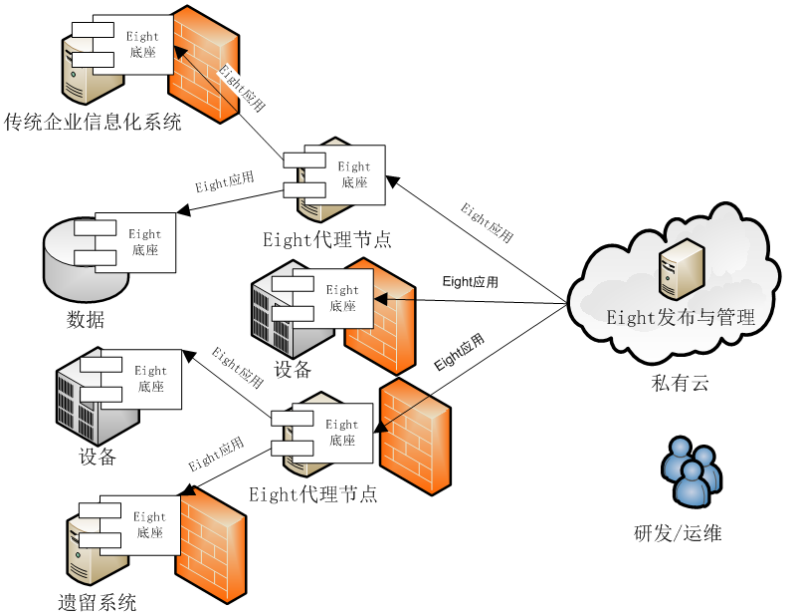
- 痛点
  - 随着工业互联网的发展，大量的传统或新兴企业开始拥有各种系统，这些系统往往分布范围广泛，基础环境复杂，系统的稳定性和可靠性都难以保证
  - 在弱网络条件下保证系统的及时更新和一致性非很困难
  - 企业对大量的边缘系统控制能力很弱
- 相比于以往方案，增加Eight基础设施之后的优势在于
  - Eight本身提供了在网即行应用部署能力，在不影响基础功能（也即无须OTA）的情况下，厂商可以获得一个可以任意支配的轻量级更新容器，将需要高度控制和管理的业务装载进去
  - 与OTA动辄数百MB乃至上G的更新包不同，Eight下的应用体积非常小，局部更新能力更能将更新粒度控制在代码级上
  - Eight应用的运行不一定依赖网络，在网络不稳定的边缘设备上运行稳定



拥有大量离散节点的企业

工业智能、OA办公、遗留系统集成

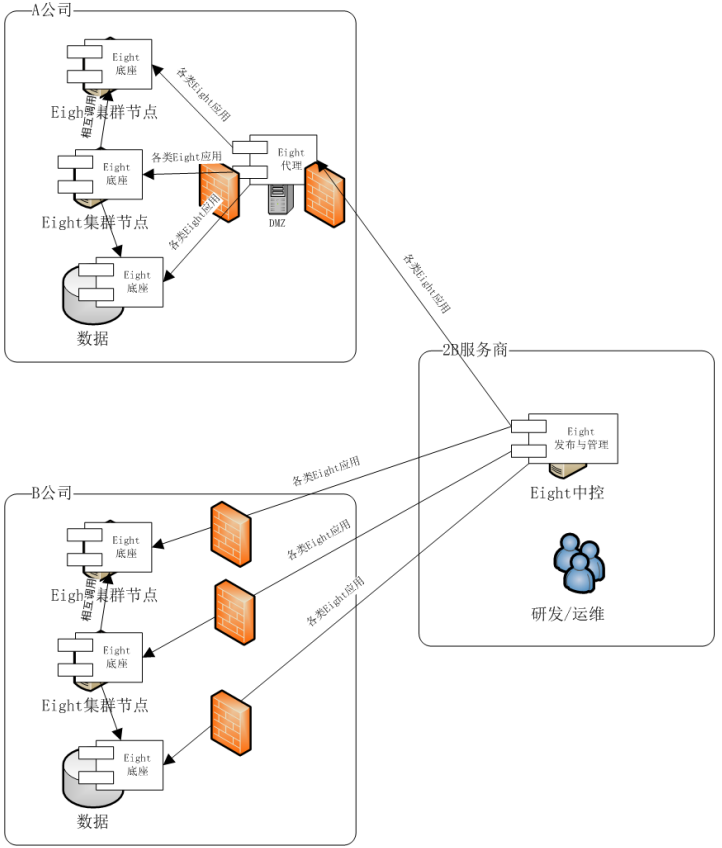
- 痛点
  - 有大量的离散系统和节点存在于企业内部，如分布于工厂车间的机械控制系统；分布于各个科室的OA办公系统；遗留下的古老的企业网站、ERP、SOA系统
  - 无论是云容器还是微服务，k8s或istio均不能应用在这种场景下
  - 如何将这些孤岛打通成整体系统，对它们进行集中的监控、管理、升级和维护
- Eight能够解决这类无解的问题
  - Eight的渗透能力强大，可以通过层层子网将节点连为一个整体
  - Eight具有强大的控制能力和可观测能力，可以将控制的触角延伸到每一个角落，对企业各处的系统进行监控、维护、升级和应用部署，可以将各种数据收集、整合、业务打通
  - Eight设计时充分考虑了传统系统的兼容问题，几乎可以运行在任何时代遗留的设备环境上
  - Eight部署方便，占用资源很少，适合在各种资源紧张的系统上持续运行
  - Java本身提供的丰富的库和扩展，能够方便的操控离散节点和应用



提供toB服务的企业

SaaS服务商、提供私有化部署方案企业

- 痛点
  - 在国内toB业务做成SaaS是没人用的，至少高净值客户是不喜欢SaaS的，他们需要私有化部署的解决方案
  - 私有化部署的痛点是成本高昂，特别涉及到定制化和二次开发
  - 私有化部署的产品更为高昂的成本体现在运维上，尤其是客户分布于全国各地时
  - 成本是toB服务商的生死线
- 相比于以往方案，增加Eight基础设施之后的优势在于
  - Eight能够运行于客户的本地环境，却又被远程集中监控和管理
  - Eight能够提供私有化部署的同时让运维SaaS化
  - toB服务商可以聚集少量优秀人才，就可以为大量分布于全国各地的客户提供服务，从而大大降低成本
  - 客户可以在满足安全性诉求的前提下，得到更为及时和精准的服务，且支付费用更少



对于所有企业

积累组件、积累业务、保留技术资产、降低研发成本

◆ 痛点

◆企业应用之所以成本高昂，核心问题在于企业业务的独特性带来的定制化需求。定制化导致现有软件不能完全被重用而必须进行二次开发，面向企业的非标准化开发投入的资源巨大而消费方却很狭窄，无法像互联网产品或标准化SaaS产品那样依靠大量用户的复用来摊薄成本

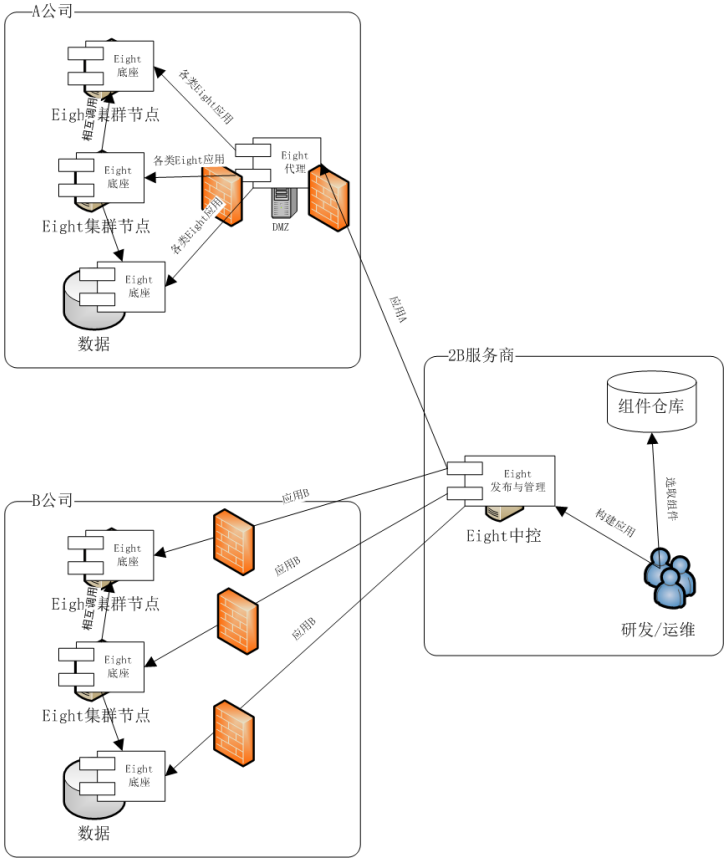
◆定制化开发缺乏长期大范围复用的考验，质量未必能够保证，其可能导致的运维问题将进一步恶化成本收益

◆ Eight独特的组件化、动态化系统架构

◆ 标准化的组件为基本单位则能够获取千变万化的能力。即便企业不断变化，业务不断演进，但其中绝大多数的是行业共性，这些共性容易抽取成为组件，在不同场景下配置不同参数按照不同方式拼装

◆ 在行业领域内纵深发展过程中，不断积累共性，积累标准化组件，应对改变，发展出更多新的业务组件，则不仅研发成本随着组件库的积累而不断降低，组件的质量也经历不断的历练而更趋稳定

◆ 同一行业的组件库可以在垂直领域内被广泛复用，以更广泛的使用来摊薄单次研发的成本，提升整体系统质量



Eight的相关资料

相关资料

- 演示系统地址
  - https://www.yeeyaa.net/
- 这里有更详细的文档，帮助你理解eight的核心思想
  - https://jekler.github.io/eight/
- Eight开源地址
  - https://github.com/jekler/eight
- 前端UI工具集EIS开源地址，这组工具包含支持IE早期版本的SVG渲染库、htc动态库和支持动态文字图标库
  - https://github.com/jekler/eis
- 欢迎使用eight做您企业应用的基础平台和解决方案
- 本公司承接企业应用项目，欢迎垂询

Enterprise Light Engine Next Generation

谢谢
THANKS

联系方式
Email： jekler@yeeyaa.net