eight 平台
V1.0
设计和使用说明

版权所有 © 2021 北京漫汇为都科技有限公司 Yeeyaa.net

文档修订记录

版本编号	修订时间	修订说明	作者	负责人
1.0	**2021-09-30**	**新的创建**	J.W.	D.D.

北京漫汇为都科技有限公司（版权所有）

目 录

目 录 3

1. 总体设计 5

1.1. 两种常见的应用系统架构 5

1.2. 基于 osgi 容器的架构及其优缺点 7

1.3. 耦合与依赖的实质 8

1.4. eight 约定的共识接口模型 9

2. eight 的开发方法 13

2.1. eight 的元件开发 13

2.2. 元件的关联 19

2.3. 模块在运行时动态关联 20

2.3.1. 模块项目结构 20

2.3.2. 模块的运行环境 23

2.3.3. 系统的动态变化 27

3. eight 的应用 30

3.1. eight 提供的基础元件库 33

3.1.1. ChainXProcessor 链式过程 33

3.1.2. ProxyProcessor 代理过程 35

3.1.3. ProcessorPResource 转换加工 resource 37

3.1.4. CascadeLPResource 嵌套代理 resource 39

3.1.5. 用元件组合来实现复杂元件 41

北京漫汇为都科技有限公司（版权所有）

3.2. eight 的监控管理 42

3.2.1. 通过 iPojo 查看系统结构 42

3.2.2. 基于 jmx 监控实例 47

北京漫汇为都科技有限公司（版权所有）

1. 总体设计

eight 平台为本公司核心技术平台，为应用层基础架构，是本公司各个业务系统和应用软件开发的基础，是本公司模块化元件库和软件工程管理的基石。

eight 平台为应用层容器，同时也是系统开发模式和模块化工程方案。旨在提供以下的能力和服务：

1） 统一的对象建模方式和标准的接口规范，归一化管理各种软件模块及对应的对外接口；
2） 组装与配置元件和组件的规范和方法，提供根据已有的元件和组件组建应用系统的能力；
3） 模块的运行时容器，提供模块的运行环境和对模块进行监控、管理的能力，提供系统的动态部署和更新的能力；
4） 软件的开发方法，提供分组开发、局部隔离、运行时组建、局部更新和维护的开发/运维手段，同时持续进行标准化的模块生产，积累模块仓库，提高软件资产的复用度；
5） 新的软件构建方式和 2B 服务形态，将系统的私有化部署及运行和在 saas 上进行系统组建、更新、监控和运维管理相结合。

1.1. 两种常见的应用系统架构

eight 设计初衷是用来解决应用系统的频繁变化和模块的复用这两个相互关联的问题。

传统的模块复用方式，以代码调用为主。通用的代码，被封装在基础库或包中以供调用，由包依赖和版本控制工具将其集成到应用系统之中。这些通用模块的使用，是通过应用系统的显式调用来实现的。对于应用系统的变化，是通过更改代码逻辑或调整依赖库来实现的。

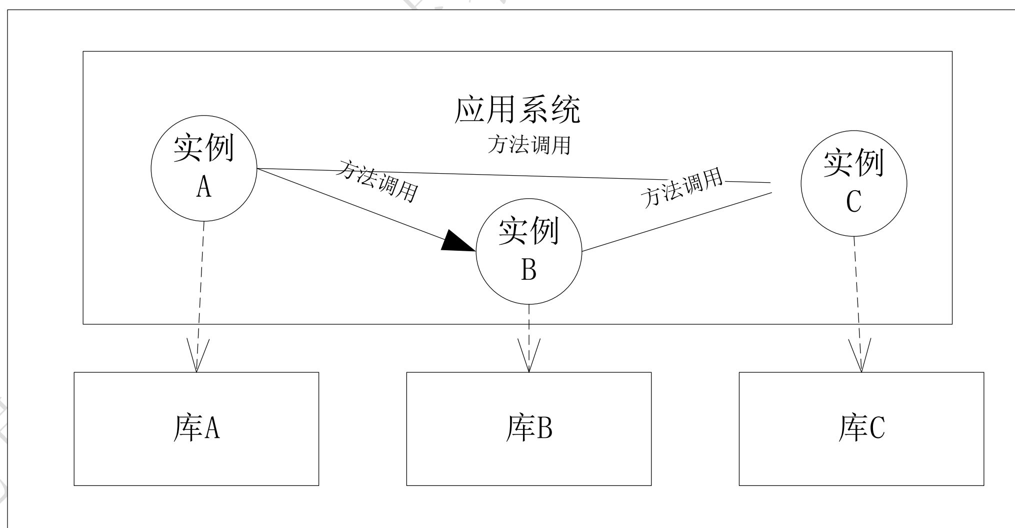


图 1.1 传统的应用系统模块调用方式

传统的应用系统为单一应用，运行在一个独立的环境之中。这种软件开发与运行方式的优势在于系统简洁，运行和维护简单。但是随着应用系统日趋庞大复杂，存在诸多问题：

1） 代码级依赖意味着任何变更都需要在代码上进行，一切业务或依赖库的变化，都会导致代码本身的变化，需要重新集成、编译和部署；
2） 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；
3） 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，并且提升冲突的可

能，系统日趋复杂和难以维护；

4） 对于大型系统，不便于分组开发和集成，难以提升研发并行度和开发效率；

5） 当应用变得庞大时，就不易扩容了，而且部署和运维消耗很大。

微服务架构则是另一种体系架构，它将一个应用系统根据功能的不同，划分为多个可独立运行的服务，这些服务在运行环境和部署上相互没有关联。功能的集成与依赖，是通过服务间的协议调用来完成的。一个服务可以被多个其他服务调用，也可以同时用于多个系统，以此来完成模块的复用。对于应用系统的变化，也就归结为服务的变化。可以单独对某个服务进行升级和更新，对变化的部分进行局部调整。

相较于复用静态代码库的方式，微服务复用的是动态运行的服务，复用方式是通过进程间甚至跨节点的协议调用来执行。

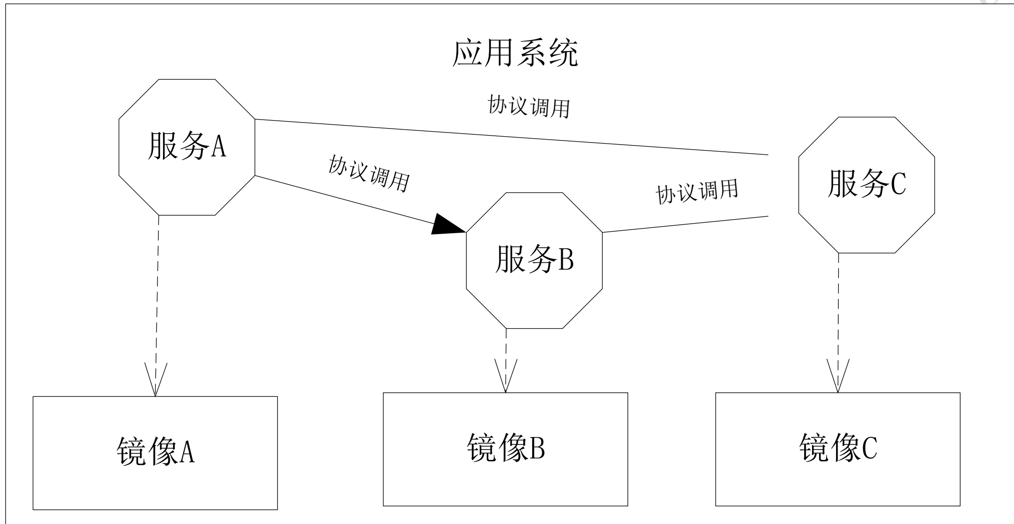


图 1.2 基于微服务架构的系统模块调用方式

微服务很大程度上缓解了传统应用架构面对的问题。

1） 将一个大型系统拆解成多个模块，模块以服务的方式独立运行，分而治之，以解决系统的复杂性问题；

2） 不同服务间相互独立，其开发、测试、部署、更新均相互隔离，互不影响，对于某一部分的调整不会影响到整体系统，便于快速开发和局部迭代；

3） 微服务以服务方式分治，每个服务均可以根据实际使用需求进行弹性伸缩和扩容，而单个服务占用资源较少，可以更为合理的扩展和利用资源；

4） 微服务可以被多个系统所复用，既节省资源，也提升了复用功能的一致性。

但是，微服务本身同样存在诸多问题：

1） 微服务带来了服务不可获得问题，引入了服务治理、追踪、监控等一系列问题；

2） 微服务将业务拆解成服务，导致了分布式事务与数据一致性等一系列问题；

3） 微服务的进程间调用方式带来额外的系统消耗，使用微服务常需要更多的硬件设备和网络带宽，以支持大型系统在诸多节点上的运行、数据传输和协同；

4） 微服务带来大量的运行部件，每一个都需要独立配置、部署、运行、维护，还需要彼此关联。其本身为了维护运行健康，也带来大量额外的部件，问题的追踪和定位也变得更为复杂。实质是把复杂性外化了；

微服务架构普遍用于庞大复杂的系统，其对基础设施和实施成本要求很高，需要有较为强大的基础设施环境、运维团队和管理手段来维护应用系统的运行。

1.2. 基于 osgi 容器的架构及其优缺点

eight 平台是一种在同一运行环境中（一般是指单一进程内）即可动态加载、运行、更新、卸载多个模块的容器。其复用方式与微服务相类似，其运行状态则类似于传统的单一应用。

eight 平台基于 spring 与 osgi 技术，使用 osgi 容器来实现模块的动态部署和运行，但对 osgi 的模块（bundle）做出重要改进。

先看看 osgi 的特点。就理念上说，osgi 提供了在单一进程内进行模块化系统开发的技术能力，但就实际使用状况上看，应用得并不普及。

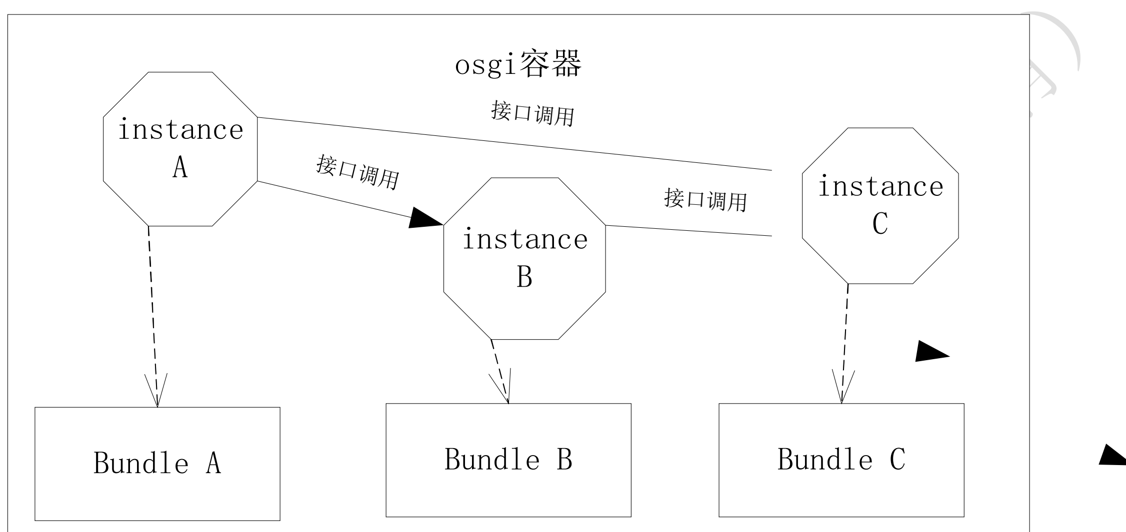


图 1.3 基于 osgi bundle 架构的系统调用方式

理想状态下的 osgi 架构，兼具了传统单一应用与微服务架构的特征，积聚了两者的优点。

1） 其运行、使用和维护上，是一个单一的进程，类似于传统单一应用，有着方便简洁的特点；
2） 它的调用模式类似于传统的方法调用，在一个进程内的方法间进行（但使用接口进行了隔离），不会产生大量的进程间和节点间网络调用，不存在额外的序列化和网络消耗；
3） 不存在服务获得性问题，由于模块实例（instance）都在同一进程内，所以其存在性是确定的。不会因为外部依赖的服务不在线出现各种异常，更无须配置各种工具对其进行监控和跟踪；
4） 同一进程内的多个模块实例，执行在单一线程上，其事务和一致性更容易保障；
5） 同微服务类似，容器内的任一局部的模块及其对应实例可以被加载、更新和卸载，随之引起整体系统的变化。而且，其动态装卸的效率要比微服务高很多；
6） 同微服务类似，不同模块相互独立，其开发、测试、部署、更新等相互隔离，对于某一部分的调整不会影响到整体系统，适用于快速开发和局部迭代的复杂系统；
7） 系统的弹性扩容更为简单，无须借助于 k8s 这类复杂的容器平台，可以在简单部署了 osgi 容器的裸系统上进行；
8） 模块可以积累成组件和元件仓库，其复用粒度是代码级的，可复用性更强。

但事实上，尽管 osgi 技术历史悠久，却一直未能得到广泛应用，实际使用中还是存在不少问题：

1） osgi 存在复杂而隐晦的包依赖和频繁的包冲突，概念体系也很复杂，在使用上存在门槛；
2） osgi 的优势在于模块化和动态性，强调模块复用和系统组建式开发。然而由于模块间暴露的 service 接口各异，以至于将各种模块拼装在一起并不容易。事实上，很多模块要使用其他模块的服务，就需要知道被调用方的接口特征和使用方式，这样固化的组合模式让模块的标准化和可复用性打了折扣，使得 osgi 的分模块加载失去意义；
3） 动态加载则由于模块间存在显示或隐式的代码依赖（如引用了其他 bundle 所 export 的包、

类或方法），导致用 classloader 加载的模块不能完全卸载。这样的系统无法顺利更新，可能带来新旧代码共存的系统故障，甚至内存泄漏等问题。所以，osgi 架构提供的模块加载方案在实践中也是问题重重。

如果能够克服 osgi 的弊端，则可以基于 osgi 技术提供的能力设计出一种优于传统的单一应用和微服务架构，兼具二者之长而避其之短，并且具备二者皆不能及的优良特征的架构。

1.3. 耦合与依赖的实质

传统的 osgi 开发，将接口暴露成服务供其它实例调用，代码实现被封装在模块内部，外界只需要知道其接口调用方法即可使用。模块可以切换更新内部实现，只要维持接口不变即可。

理想上这就实现了模块隔离，但是事实上，由于各种迥异的对外接口的存在，使得要使用一个模块就不能不了解其定义。最终这种封装并没能完全起到隔离的效果，开发者不得不为使用某个模块去专门对其接口开发代码，这与传统的代码调用差异不大了。带来的后果是模块的复用性不高，关联的模块间绑定的很深，系统的动态性更像是画蛇添足。而一旦被调用的模块发生变化，导致接口或其使用方式变化，则依赖它的所有模块的代码均需要调整。

这种问题的表象，是各种纷纭复杂的接口定义带来的特异性问题，而实质是各个模块的概念不一。模块对外提供了不同的调用界面（接口），本质是其开发者在开发模块时所使用的各自不同的概念体系和思维方式。这就意味着使用者在需要使用这些模块时，形式上是需要了解这些接口的定义和使用方法，实质是需要了解模块开发者的设计思想。

这样的结果，即使形式上看似进行了隔离，实质上模块间仍然是耦合的。

模块隔离与解耦，需要模块的开发者与调用者不必形成依赖关系，模块的内容，须不以外物的存在为前提。但如果从开始，开发者就需要去理解其所依赖的模块（服务、库），则耦合在开发过程中就被显示或隐式的埋藏在模块的基因之中。从这个意义上说，现有的一切应用系统，无论其基于传统的代码依赖，还是采用了时兴的微服务，抑或是使用了 osgi 框架，其实都没有摆脱模块间的耦合。以至于局部的模块变化会扩散至系统的其他部分；而软件的研发和系统的不断演进也并不那么独立，受限于各种依赖的次序；至于模块、微服务间的组合，则往往是早已母胎决定，没有那么多灵活度和复用性可言。

这是现今应用系统依然难以大规模使用既有模块组建的原因，也是在软件工程上影响着软件开发、演化和重用的难题。解决上述问题，是软件系统突破耦合边界的束缚，进入新的开发和应用形态的关键。

如何能够达成这样的目标，首先是需要找到问题的本质。仔细分析上述过程：代码的耦合是现象，而思维的依赖是本质。开发者在开发过程中对当前模块所需使用的边界外的“物”（模块、服务、库或其他），需要有一定程度（在不同的架构下其程度有所差异）的认知，这是必须的，而这种认知就是依赖的本质。

如果不存在这种需要认知的依赖，则模块间自然就不存在这种耦合。但这却难以设想，如何可能在一个模块开发和调用其它模块时不去认识其它模块的接口和使用方式，而在完全无知的情况下调用又何以能得实行？

这是一个关于认识的问题。然而问题并不在于如何在无知的状态下去进行调用和协同，而在于如何在并不需要学习与理解的情况下，消除这种无知的状态，至少要能做到对其外部界面（接口）有所认识。这很矛盾，对于一个陌生的外“物”，何以能够毋须学习和理解，就能对其拥有认识呢？

这看似无法达成的设想，其实有着相对积极的答案。既然模块的依赖和耦合是思维依赖的具象体现，那么对不同思维个体所创造的“物”，是否必须先进行足够的学习才能使用呢？情况并非如此，至少在现实世界里，并非任意一种“物”的使用都伴随着学习和理解这一先决条件。当先民会用石块砸开核桃时，给他们一把锤子他们多半不必去研究如何使用。若是在一条河边放上一个桶，

北京漫汇为都科技有限公司（版权所有）

人们大多会明白这桶是用来装水的，而无需额外的解释。

这样不言自明的认识来自于何处？按理性主义的观点，这些知识是先验的，与生俱来的，如同“洞穴外的真相”、“前定和谐”或“先验综合判断”；按经验主义的观点，这些知识是人们在共同或类似的经验之中习得的，依靠“逻辑实证”而升华，成为很多人所共有的知识。

不管这种知识源于何处，总之，这种被多数人所拥有的具有共同性的知识，提供了一条消除或弱化模块间耦合和依赖的道路。假如模块能够按照共有的知识（共识）来自我陈述，则学习和理解就不再是必要的过程，至少会相对容易很多。共识先于外“物”而存在于各个开发者思维之中，则开发过程也就不必依赖于对外“物”的学习与掌握，甚至不必依赖于外“物”的存在。这样、真正独立的、能够并行的模块开发将成为可能，而模块则能摆脱其对外界的依赖和耦合，成为独存的个体，却又能在必要时与外“物”发生关联和协同，构造出各异的系统。软件的动态组装和模块的高度复用将不再是空中楼阁。

这就有赖于开发者去建立“共识”。

1.4. eight 约定的共识接口模型

软件开发的代码，是思维对客观事物的描述，“物”的外部特征，就体现为“物”（模块）对外暴露的接口。不建立在共识基础上的迥异接口，是模块间存在耦合与依赖的病因所在。而要求思维以共识来对“物”进行表述，也就是约束模块对外暴露出共同的接口。

这里存在一个似乎怪异的前提，要求用代码表述的万“物”，对外呈现出共同的接口。那么是否可能存在一种共识，能够用来表达所有的“物”的对外特征呢？

这将涉及到三个相互关联的问题：完备性，是否存在这种表述的可能，以及这种表述是否能覆盖所有涉及到的问题的领域；趋同性，是否不同的思维主体，在约定的表述方式下，对其表述内容的实现具有一致的趋向，这是消除或弱化耦合的关键；适应性，如果对“物”的思维不能取得完全一致，则共识多少存有差异，作为存有差异的思维所物化结晶的各种模块，是否还能够基于共识进行关联和协同，以构造出整体的系统。

完备性的问题，实质上是在探寻认识一切事物的方式，能否都按照约定的模式进行，或者换一个说法，一切事物是否都能用约定的方式进行解构。尽管对此并没有科学的方法去求证，但历史上却不乏这样的尝试，无论是德谟克利特的“原子”、莱布尼兹的“单子”还是维特根斯坦的“事实”，都在追溯事物的基本结构，找寻世界本质的统一。

就软件开发而言，在特定应用领域或某些场景也有类似思路。如 hadoop 将各种大数据处理过程抽象为 map 和 reduce，由此提供了 mapper、reduecer 等少数几种接口。而 spring 则用于协助开发者生成对象，它在 IoC 时将对象视为 bean，以 bean 的 setter 为其共识界面，以注入为通用过程。

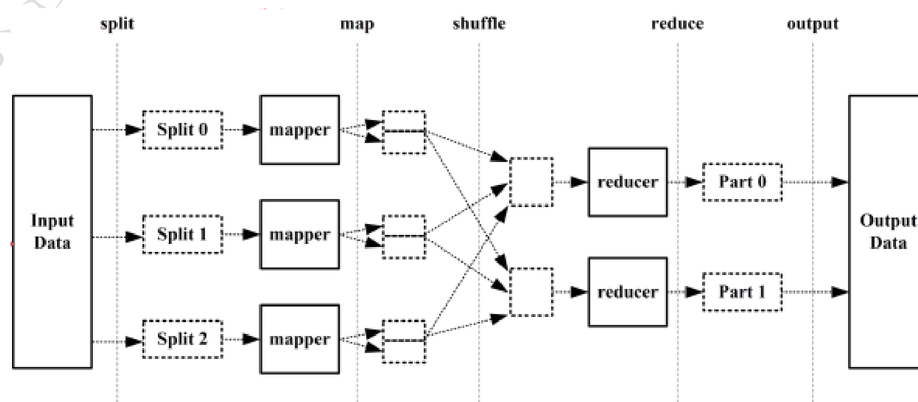


图 1.4 hadoop 的 MR 模型

北京漫汇为都科技有限公司（版权所有）

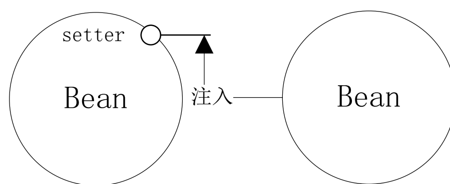


图 1.5 spring 的 Bean 模型

尽管这些模型仅用于其针对的局部领域或方面，并且更多的是从传统视角出发，为设定好的框架提供了标准化的扩展接口，而非以统一的接口规约所有的领域和场景。但不容置疑的是这些模型也都很好的覆盖了其局部问题领域，而这些模型通常具备简洁的形式。简洁的形式、少量的概念、直观的过程往往更接近于事物的基本结构，也更有普适性；而复杂的接口，则往往意味着太多属于个体的特异性。所以，能够提供高完备性的共识，其形式势必是相对简单的。

关于趋同性的问题则是另一个方面。形态过于简单的模型，其自由度也更高，对思维的约束也就更低，形成的不须阐明的共识也就更少。例如 spring 的模型，对 bean 的认知，除了其开放了注入界面外，其它是一无所知。这样的高度自由，尽管其普适性几乎不受约束，但却无法单纯的仅靠共识去使用。事实是，spring 只是用于创建 bean，它并不能告知这些 bean 的作用，对 bean 的调用依然需要去了解 bean 所实现的具体方法或接口。所以，能够引导开发者趋同化思维的共识，其形式也需要有相对的约束力。

如果能在完备性与趋同性上取得合理的尺度，则由共识建立的模块也就具备了独存和关联的可能。然而涉及到关联，尽管有共识约束存在，但开发者之间的思维差异也不可避免。此时如何去协调模块之间的关联，使其能够协同构建出整体系统呢？一方面，这需要合适的共识形式，能够在不同开发者中引导出相同或相近的思维；另一方面，则有赖于支撑模块的运行时容器，能够提供简单方便的方式，用尽量少的成本来实现模块间的适配。后一个方面往往借助于前一个方面，模块开发者对共识的思维越接近，则适配就越是简单。

基于以上设计思想，eight 约定以下的共识。

eight 将一切事物视为资源（存储“物”信息之实体）与过程（处理“物”信息之操作），资源代表“物”的静态，过程代表“物”的动态。eight 认为系统的一切计算和处理，都是过程在资源上运行的结果。

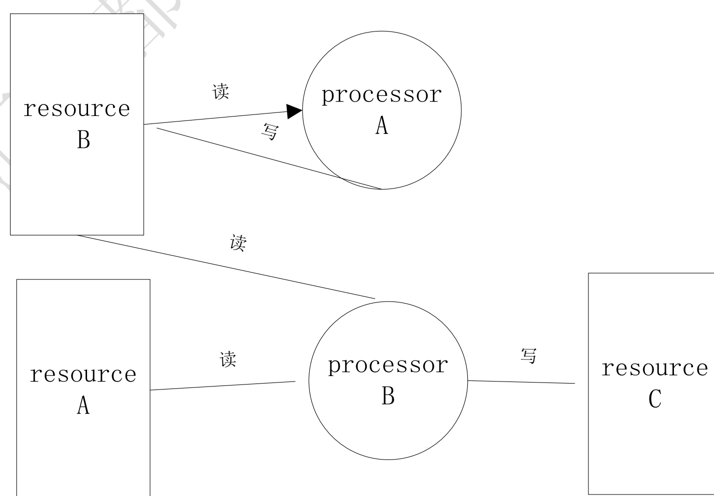


图 1.6 eight 的共识模型

由此，eight 的共识接口总体上分为两族：代表资源的 resource 和代表过程的 processor。其中 resource 存储数据，遵循 key-value 结构。

代表过程的 processor 族有以下接口：

1）一元 processor，定义如下：

北京漫汇为都科技有限公司（版权所有）

```java
public interface IProcessor<T,R> {
        public R process(T object);
}
```

2）二元 processor，定义如下：

```java
public interface IBiProcessor<K, V, R> {
        public R perform(K first, V second);
}
```

3）三元 processor，定义如下：

```java
public interface ITriProcessor<T, K, V, R> {
        public R operate(T first, K second, V third);
}
```

代表资源的 resource，有以下接口：

4）只读 resource，定义如下（其中 key 为数组，表示多级目录）：

```java
public interface IInputResource<K, V> {
        public V find(K ... paras);
}
```

5）只写 resoruce，定义如下：

```java
public interface IOutputResource<K, V> {
        public <P> P store(V value, K ... paras);
        public <P> P discard(K ... paras);
        public <P> P empty(K ... paras);
}
```

6）可列举 resource，定义如下：

```java
public interface IListable<K, V> {
        public Collection<K[]> keys(K ... paras);
        public Map<K[], V> all(K ... paras);
}
```

7）由只读 resource 和只写 resoruce 组合构成可读写 resource，定义如下：

```java
public interface IResource<K, V> extends IInputResource<K, V>, IOutputResource<K, V> {}
```

8）由只读 resource 和可列举 resoruce 组合构成可列举只读 resource，定义如下：

```java
public interface IReadonlyListable<K, V> extends IInputResource<K, V>, IListable<K, V> {}
```

9）由可读写 resource 和可列举 resoruce 组合构成可列举读写 resource，定义如下：

```java
public interface IListableResource<K, V> extends IResource<K, V>, IReadonlyListable<K, V> {}
```

对于资源存取数据时，如果存在原子性的事务需求，则需要实现事务接口：

10）事务接口需要实现 R execute(IProcessor<U, R> processor)方法，其方法接受一个 IProcessor 过程，其过程以 U extends IResource<K, V>为输入参数，也即接受一个 IResource<K, V>为参数，这个传入的 resource 是对要被实际运行的 resource 的封装，其应确保 processor 过程对其的操作，要么全部成功，要么全部不被执行（传入的 resource 事务的具体实现方式以及事务级别可能存有差异，这些差异如果对执行过程存在影响，则属于适配问题，应该在模块组装时加以区别）。定义如下：

```java
public interface ITransaction<K, V, U extends IResource<K, V>, R> {
        public R execute(IProcessor<U, R> processor);
}
```

北京漫汇为都科技有限公司（版权所有）

11）由可读写 resource 和事务 resource 组合构成可读写事务 resource，定义如下：

```
public interface ITransactionResource<K, V, U extends IResource<K, V>, R> extends IResource<K, V>, ITransaction<K, V, U, R> {}
```

12）由可列举读写 resource 和事务 resoruce 组合构成可列举读写事务 resource，定义如下：

```
public interface IListableTransaction<K, V, U extends IResource<K, V>, R> extends IListableResource<K, V>, ITransactionResource<K, V, U, R> {}
```

除此之外，考虑到资源与过程模型可能难以覆盖所有的场景，由此增加了扩展模型。扩展模型采用名实论世界观，将一切“物”视为一组“名”的“组合”，而每一个“名”又代表一个“资源”或“过程”，构成“物”的局部，可以在该“物”上按名获取这个局部的“资源”或“过程”。这些“资源”或“过程”代表可以对该物进行的操作，而其本身也可能是可扩展的“物”。如图：

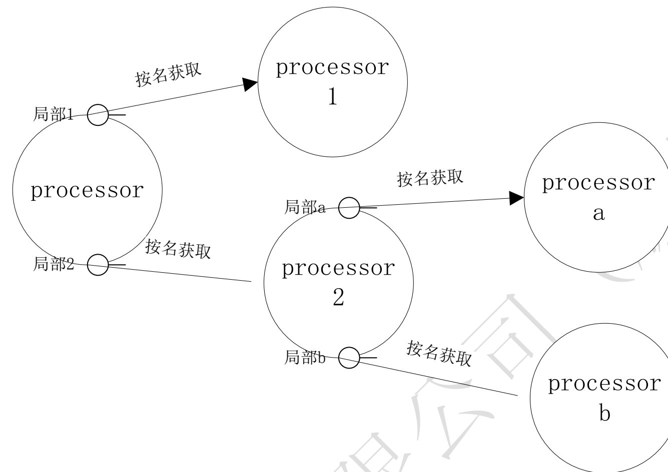


图 1.7 eight 的扩展共识模型

13）扩展接口 IExtendable，该接口可以附加于任何“物”（包括资源和过程），为该物增加任意的可操作的“局部”，定义如下：

```
public interface IExtendable<M> {
        public Collection<M> methods();
        public <N> N extend(M method);
}
```

14）IThing 接口则是另一种扩展，它代表一个“物”，该物可以是任意接口（资源或过程）的“综合”，通过 present 呈现出“物”的不同“方面”。定义如下：

```
public interface IThing {
        public <L> L present(Class<L> clazz);
}
```

扩展共识作为一种辅助模型，在增强“物”的完备性的同时，也大大增加了其自由度。结果是对其表述的“物”的约束力降低，带来趋同性低下和适配困难。所以，扩展共识应严格控制使用范围，仅用于必要的场景。

最后一种接口，是实现了所有接口的接口，常用作代理。

15）由所有接口组合构成的接口，其有一个 realObject 方法用于获取其代理的“实物”：

```
public interface IUniversal <K, V, U extends IListableResource<K, V>, T, R> extends IProcessor<T,R>, IBiProcessor<K, V, R>, ITriProcessor<T, K, V, R>, IListableTransaction<K, V, U, R>, IExtendable<T>, IThing {
   public <O> O realObject();
}
```

eight 的共识，由此 15 个接口构成。其血缘关系如图：

北京漫汇为都科技有限公司（版权所有）

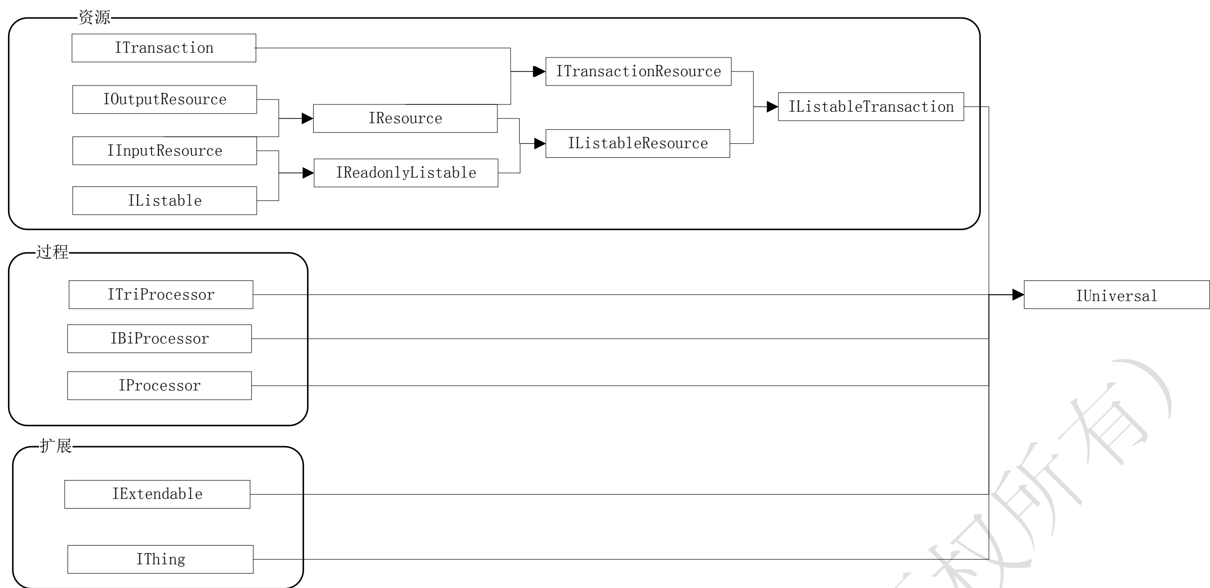


图 1.8 eight 的共识接口的血缘关系

2. eight 的开发方法

eight 的开发方法涉及元件（element）、组件(component)和系统三个层次，与之对应的是代码、模块（bundle）和容器。以代码开发元件，由元件组合成组件模块打包，有不同的组件模块在容器中实例化和关联组合成系统。其中元件组合的过程采用了 spring 框架，组件组合的系统过程，采用了 osgi 框架（felix iPojo）。系统运行于 felix（osgi）容器之中。

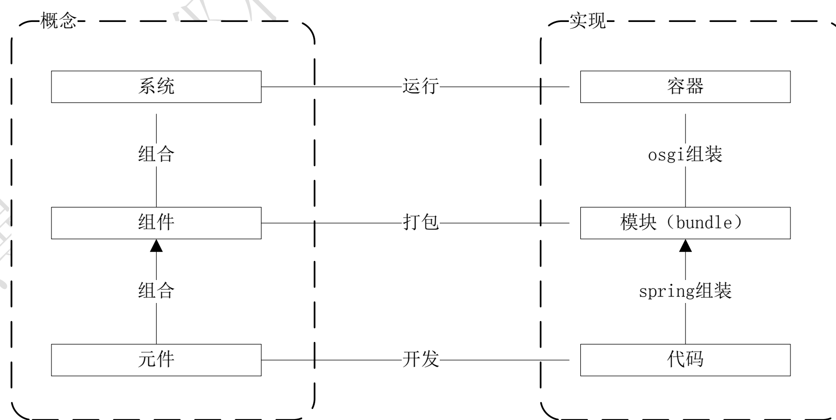


图 2.1 eight 的概念层级和实现方式

2.1. eight 的元件开发

eight 元件开发的方式，就是按照 eight 的共识模型开发“物”的过程。每一个“物”由一个

北京漫汇为都科技有限公司（版权所有）

或多个类（class）构成，并且服从以下约定：

1） 这一个或多个类构成一个整体的元件，该元件对外提供的接口应包含且仅包含共识；

2） 元件对其它元件的指代（reference）遵循且仅遵循共识。

一般而言，建议一个元件仅由一个类构成，代表一个唯一的“物”，较复杂的“物”需要多个类来实现的也可以考虑使用内部类以保持外观上的原子性。一个元件，应在概念上等同于一个独立的“物”，该“物”用以处理某一特定过程或是某一方面问题的特定组成部分。“物”具有抽象性和普遍性，可以为这一类问题提供通用的处理。而不包含在此“物”概念范畴之内，却又需要赖以发生关联的应以他“物”来描述。此“物”对他“物”的调用，也应遵循共识的约束。任意的“物”不应该用共识之外的方式去与他“物”发生关联。

在此做一个简单的系统示例，假设有如下功能需求：给定一个关键词，对一个目录下的所有文件，统计该关键词出现的总次数。

将该功能划分为三个彼此独立的元件：

1） 一个可列举的存储，用于检索和列举目录下的文件，以 String 形式返回文件内容；

2） 一个执行关键词匹配的处理过程，在 String 中发现指定关键词并计数；

3） 一个用以累加所有文件中出现关键词的总次数的处理过程，对所有文件进行统计。

其中，1 是资源，因为存在列举，所以应该使用 IListable（或继承了它的子接口）这种 resource；而 2，3 则是过程，由 2 的功能定义看来，2 需要两个参数：关键词和文件内容，所以 2 的实现是一个 IBiProcessor；而 3 则是进行统计的过程，它只需要一个参数：关键词，所以 3 的实现是一个 IProcessor。

它们三者之间的关联方式如图：

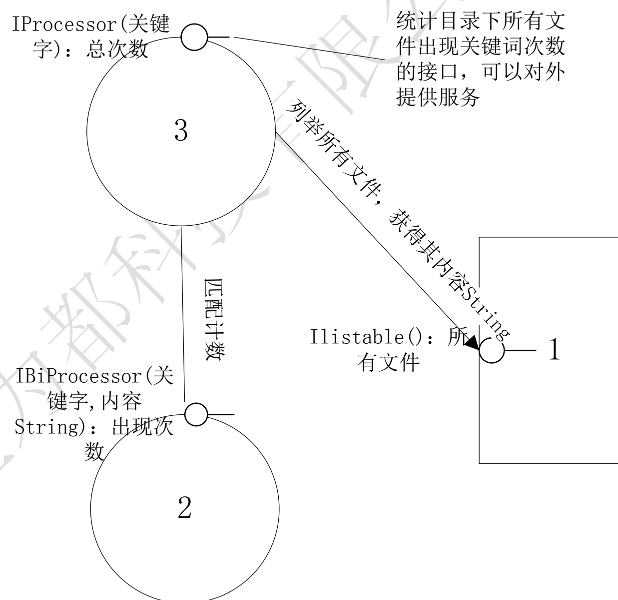


图 2.2 关键词计数服务的资源、过程及其接口与关联结构

对于 1，它是被调用方，它的功能主要是确定一个文件目录并对目录下的文件进行获取或列举（如果实现了 IOutputResource 接口，则还可以对文件进行增删改），将文件内容按 String 方式输出。这样的文件处理过程并没有与当前业务存在必然联系，因而这个元件具备普适性。

这个“物”代码如下：

```
public class FileStringLResource<K> implements IListableResource<K, String> {
    protected String base;
    protected FilenameFilter filter;
    protected Boolean dir;
    protected Integer buffer = 8192;
```


北京漫汇为都科技有限公司（版权所有）

```
	protected String charset;
	protected Boolean auto = false;
	protected Long max = 0L;

	public void setMax(Long max) {
		if(max != null && max > 0) this.max = max;
	}

	public void setBuffer(Integer buffer) {
		if(buffer != null && buffer > 0) this.buffer = buffer;
	}

	public void setCharset(String charset) {
		this.charset = charset;
	}

	public void setAuto(Boolean auto) {
		if(auto != null) this.auto = auto;
	}

	public void setDir(Boolean dir) {
		this.dir = dir;
	}

	public void setFilter(FilenameFilter filter) {
		this.filter = filter;
	}

	public void setBase(String base) {
		this.base = base;
	}

	@Override
	public String find(K ... paras) {
		try{
			StringBuilder p = new StringBuilder();
			if(base != null) p.append(base);
			if(paras != null) for (K para : paras) if (para != null)
p.append(para.toString());
			if(auto) return
CommonUtil.urlToStr(TypeConvertor.strToUrl(p.toString()), buffer, max);
			else return
TypeConvertor.bytesToStr(TypeConvertor.urlToBytes(TypeConvertor.strToUrl(p.toString()),
buffer, max), charset);
		} catch(Exception e){
			log.error(paras[0] + ":find resource error", e);
```

北京漫汇为都科技有限公司（版权所有）

```
		}
		return null;
	}

	@Override
	public <P> P store(String resource, K ... paras) {
		if(resource != null) try{
			StringBuilder p = new StringBuilder();
			if(base != null) p.append(base);
			if(paras != null) for (K para : paras) if (para != null) 
p.append(para.toString());
			TypeConvertor.bytesToFile(TypeConvertor.strToBytes(resource, 
charset), p.toString());
		} catch(Exception e){
			log.error(paras[0] + ":find resource error", e);
		}
		return null;
	}

	@Override
	public <P> P discard(K ... paras) {
		try{
			StringBuilder p = new StringBuilder();
			if(base != null) p.append(base);
			if(paras != null) for (K para : paras) if (para != null) 
p.append(para.toString());
			String path = p.toString();
			PlatformUtil.deleteFiles(new File(path));
		} catch(Exception e){
			log.error(paras[0] + ":find resource error", e);
		}
		return null;
	}

	@Override
	public <P> P empty(K... paras) {
		StringBuilder p = new StringBuilder();
		if(base != null) p.append(base);
		if(paras != null) for (K para : paras) if (para != null) 
p.append(para.toString());
		PlatformUtil.deleteFiles(new File(p.toString()));
		return null;
	}

	@Override
	public Collection<K[]> keys(K... paras) {
```

北京漫汇为都科技有限公司（版权所有）

```
            StringBuilder path = new StringBuilder();
            ArrayList ret = new ArrayList<K[]>(0);
            if(base != null) path.append(base);
            if(paras != null && paras.length > 0 && paras[0] != null)
path.append(paras[0].toString());
            if(path.length() > 0) try{
                  File f = new File(path.toString());
                  File[] files;
                  if(filter == null) files = f.listFiles();
                  else files = f.listFiles(filter);
                  if (files != null && files.length > 0) {
                        ret = new ArrayList<K[]>(files.length);
                        for(File file : files) try{
                              if(dir == null || dir.equals(file.isDirectory())){
                                    StringBuilder sb = new StringBuilder();
                                    if(paras != null && paras.length > 0 &&
paras[0] != null) {
                                          sb.append(paras[0].toString());
                                          if(sb.charAt(sb.length() - 1) !=
File.separatorChar) sb.append(File.separatorChar);
                                    }
                                    String[] name = new String[1];
                                    name[0] =
sb.append(file.getName()).toString();
                                    ret.add(name);
                              }
                        }catch(Exception e){
                              log.error(paras[0] + ":list resource error", e);
                        }
                  }
            }catch(Exception e){
                  log.error(paras[0] + ":list resource error", e);
            }
            return ret;
      }

      @Override
      public Map<K[], String> all(K... paras) {
            Collection<K[]> keys = keys(paras);
            Map<K[], String> map = new HashMap<K[], String>(keys.size());
            for(K[] key : keys) map.put(key, find(key));
            return map;
      }
}
```

这个类实现了 IListableResource（IListable 的子接口），可用于需要进行写操作的 resource 场景中。它带有几个成员变量，用来为后续操作提供参数。如 base 指定根目录，filter 过滤文件

北京漫汇为都科技有限公司（版权所有）

名，charset 指定文件的编码，max 指定最大允许读写的文件大小等。这些参数有些有默认值，而有些没有，该 resource 提供了一系列 setter 方法，需要像 spring 或 guice 这样的 IoC 框架来完成其参数配置和实例化。这是开发元件的常见写法。

该类并没有调用其它的 processor 或 resource，它仅在其自身涉及的范畴内（文件读写）调用了一些库（如 java 的 io 库，这些库在被元件调用时，是元件内部实现的一部分，并不需要实现共识接口）来实现功能，但对外提供了共识接口。这个类向外界宣布了一个 resource，这个 resource 是基于 IListableResource 接口进行文件读写的，其 value 是文件内容，具备 find\store\discrad\empty\keys\all 等几个操作方法，分别执行查\增改\删\清\列举\获取所有的操作过程。这样的共识是遵循直觉的，当另一个开发者需要一个这样的 resource 时，这个实现往往与开发者需要的使用方式一致。也即，该实现在不同的开发者思维中趋同。

对于 2 这个过程，其实现了一个简单的匹配逻辑。

```
public class KeywordMatcher implements IBiProcessor<String, String, Integer>{
   @Override
   public Integer perform(String keyword, String content) {
          Pattern p = Pattern.compile(keyword, Pattern.LITERAL );
          return p.split(content, -1).length -1;
   }
}
```

同样的，该实现也具备独立性，接口也符合直觉，从而具备关联和复用的能力。

3 这个过程相对特殊一些，它本身会关联到 1 与 2，它的代码如下：

```
public class Counter implements IProcessor<String, Integer>{
   protected IListable<Object, String> documents;
   protected IBiProcessor<String, String, Integer> processor;

   public void setDocuments(IListable<Object, String> documents) {
          this.documents = documents;
   }

   public void setProcessor(IBiProcessor<String, String, Integer> processor) {
          this.processor = processor;
   }

   @Override
   public Integer process(String keyword) {
          Map<Object[], String> docs = documents.all();
          int sum = 0;
          for (String content : docs.values())
                sum += processor.perform(keyword, content);
          return sum;
   }
}
```

可见，该元件对于需要调用但尚且无知的外“物”，定义了两个共识指代。一个用以指代存放这些文件的存储，它至少应该是可列举的。另一个是对该文件进行处理的过程，它需要传入文件内容和输入的关键字作为参数，结果必须返回一个整数。有趣的是，尽管 2 的需求是计算关键字匹配的次数，但在 3 的代码里只是单纯的将 2 其描述为一个过程处理（processor），并没有暗示其是匹

北京漫汇为都科技有限公司（版权所有）

配器（matcher）。换言之，在 3 的开发者的视角里，可能并不在意这个处理过程具体的作用，只需要知道它对文件内容和关键字做过处理后，返回了一个整数处理结果。这个处理过程可能是任意的，例如在文件中找到关键字第一次出现的位置，或是得到匹配字串最大的长度。

所以，尽管 3 这个过程对外“物”存有关联，但其仍是在按开发者自己建立的共识指代在进行开发。这个过程是独立的，无须去学习和理解其关联的他“物”，所以这个元件本身也是独存的，有其固有意义且不依赖于外“物”。

一旦这些共识指代被确定，这三个元件关联起来，本系统的功能也就实现了。

2.2. 元件的关联

前面的示例完成了元件的代码，这些元件本身是独立的，对外“物”的使用是抽象的，如果不加关联，则无法组合成可用的系统。若要进行关联，可以采用依赖注入的 spring 或是 guice 来实现元件的实例化、参数注入和共识指代的注入。

以下以 spring 为例介绍元件的关联。对于前述三个元件，可以按功能需求将其组装起来，其配置如下：

```
<bean class="net.yeeyaa.test.matcher.KeywordMatcher" id="matcher"/>

<bean class="net.yeeyaa.eight.common.resource.FileStringLResource" id="documents">
        <property name="dir" value="false"/>
       <property name="charset" value="utf-8"/>
       <property name="base" value="/tmp/docs/"/>
</bean>

<bean class="net.yeeyaa.test.count.Counter" id="counter">
        <property name="documents" ref="documents"/>
       <property name="processor" ref="matcher"/>
</bean>
```

对 FileStringLResource 配置的 base 是/tmp/docs 目录，dir 为 false 表示不会搜索其子目录，字符集则选定了 utf-8。这样，这个系统直到运行时才由配置文件将元件关联起来，其执行细节也随着参数的确定而固化下来。

如果需要分析的文件并没有存储在文件系统上，而是存在数据库中，只需要更换对应的 resource 就可以。如下：

```
<bean class="net.yeeyaa.test.matcher.KeywordMatcher" id="matcher"/>

<bean class="net.yeeyaa.eight.data.resource. DbLResource" id="documents">
       <property name="platformDao" value="dao"/>
</bean>

<bean class="net.yeeyaa.test.count.Counter" id="counter">
        <property name="documents" ref="documents"/>
       <property name="processor" ref="matcher"/>
</bean>
```

DbLResource 也是实现了 IListable 接口的 resource。它需要配置一个数据库的 dao，默认会访问 ResourceEntity 表，列举其中的文件并将内容传回。

北京漫汇为都科技有限公司（版权所有）

可见，采用了共识接口后的系统，其环境适应性可以大大扩展。

spring 还可以参数化配置，如下：

```
<bean class="net.yeeyaa.test.matcher.KeywordMatcher" id="matcher"/>

<bean class="net.yeeyaa.eight.common.resource.FileStringLResource" id="documents">
        <property name="dir" value="false"/>
        <property name="charset" value="utf-8"/>
        <property name="base" value="${test.documents.base:/tmp/docs/}"/>
</bean>

<bean class="net.yeeyaa.test.count.Counter" id="counter">
        <property name="documents" ref="documents"/>
        <property name="processor" ref="matcher"/>
</bean>
```

将文件目录作为一个参数暴露出来，则在运行该系统时，传入 test.documents.base=/xxx/yyy，就可以在/xxx/yyy 目录下检索文件。每次加载时都可以设置不同目录。这个特征后续将用于参数化实例的生成。

spring 的更多用法可以参考其官方文档：

https://docs.spring.io/spring-framework/docs/current/reference/html/

2.3. 模块在运行时动态关联

前述的关联方式已经能将元件组合起来构成一个整体的系统，采用 spring（或其他类似的注入框架）是将关联延迟到配置时，这显然比开发时确定关联的灵活性提高很多。但是一旦配置完成，其关联也就固化下来，必须要到运行结束后修改配置才能进行变化。

如果需要在运行时对系统进行更新和变化，就需要打包成模块后用 osgi 框架支持。

2.3.1. 模块项目结构

对于 osgi 的框架，eight 采用了基于 felix 的 iPojo 容器，也可以采用其它框架（如 equinox，knopflerfish，springDM 等）或容器（如 scr 等）。关于 iPojo 的介绍可参阅：

http://svn.apache.org/repos/asf/felix/releases/org.apache.felix.ipojo-1.0.0/doc/ipojo-in-10-minutes.html

一个或多个元件可以被打包成一个模块（bundle），在一个模块中，可以定义一个或多个组件（component），建议一个模块（bundle）仅定义一个组件。一个组件可以由一个或多个元件的组合而成，组合方式可以是写一个类进行拼装，更常见的是采用 spring 等注入框架来组装元件。组件服从以下约定：

1） 一个或多个元件构成一个整体的组件，该组件对外提供的接口应包含共识；

2） 组件对其它组件的指代（reference）遵循且仅遵循共识。

注意，组件并不限定对外接口仅限于共识，是因为 osgi 容器本身也会约定一些接口提供服务。

继续前述示例，如果计划在运行时动态的关联 matcher、counter 和 documents 这三个元件，则应当将它们分别打包成模块，并作为组件发布。原则上，可以将系统中存在变化的元件孤立出来打

北京漫汇为都科技有限公司（版权所有）

包，稳定的关联关系则组装为一个整体的组件。也即，组装组件的原则是固化内聚性和孤立耦合性。

例如，若在系统之中，matcher 与 counter 的搭配是稳定的，而 documents 可能发生变化（如从扫描文件系统变更为检索数据库），则应将 documents 独立打包，而 matcher 与 counter 组合成组件打包。而假如 matcher 的算法可能变化（如从关键词匹配变化为正则表达式匹配），则应当把 matcher 也拆分开来。在后续的例子里，将这三部分均拆解为独立的组件。

eight 为组件打包提供了标准结构，定义一个组件，可以在指定的包目录下放置 spring 的 xml（或 groovy）配置。如图所示：

```
> osgi-matcher [osgi-matcher master]
  > src/main/java
    > net.yeeyaa.test.matcher
      KeywordMatcher.java
  > src/main/resources
    > META-INF
      > spring
        > test_matcherContext.xml
      > metadata.xml
      > spring.handlers
      > spring.schemas
    > OSGI-INF
```

图 2.3 matcher 模块的项目结构

一个模块（bundle）项目中，可以包含代码（如实现了关键词匹配的 KeywordMatcher），组件定义（如 META-INF/metadata.xml 是用来定义和发布组件的），以及配置信息（如 META-INF/spring 目录下的 xml 配置，是用来定义和配置元件及其关联的）。

在这个模块中只定义了一个元件，定义在 test_matcherContext.xml 中，这是一个 spring 配置：

```
<bean class="net.yeeyaa.test.matcher.KeywordMatcher" id="matcher"/>
```

而 metadata.xml 则是 iPojo 的配置，用来定义组件（component），matcher 的定义如下：

```
<ipojo xmlns:jmx="org.apache.felix.ipojo.handlers.jmx">
    <component classname="net.yeeyaa.eight.osgi.runtime.BundleCenter" name="matcher">
        <controller field="state"/>
        <callback transition="validate" method="validate" />
        <callback transition="invalidate" method="invalidate" />
        <requires id="executor" optional="true" nullable="false">
            <callback type="bind" method="bindExecutor"/>
            <callback type="unbind" method="unbindExecutor"/>
        </requires>
        <requires id="proxy" aggregate="true" optional="true">
            <callback type="bind" method="bindProxy"/>
            <callback type="unbind" method="unbindProxy"/>
        </requires>
        <properties updated="updated" propagation="false">
            <property name="wait" method="setWait"/>
            <property name="config" method="setConfig"/>
            <property name="permit" method="setPermit"/>
            <property name="path" method="setPath" value="OSGI-INF/blueprint"/>
            <property name="pattern" method="setPattern" value="*Context.xml"/>
            <property name="recurse" method="setRecurse" value="false"/>
            <property name="trace" method="setTrace"/>
```

北京漫汇为都科技有限公司（版权所有）

```
                    <property name="clone" method="setClone"/>
                    <property name="mode" method="setMode"/>
                    <property name="thread" method="setThread"/>
                    <property name="resource" method="setResource"/>
                    <property name="holder" method="setHolder"/>
                    <property name="log" method="setLog"/>
                    <property name="logger" method="setLogger"/>
                    <property name="context" method="setContext"/>
                    <property name="begin" method="setBegin" value="begin"/>
                    <property name="close" method="setClose" value="close"/>
                    <property name="reload" method="setReload"/>
                    <property name="readonly" method="setReadonly"/>
                    <property name="hookid" method="setHookid"/>
                    <property name="key" method="setKey" value="service.description"/>
             </properties>
             <provides           specifications="net.yeeyaa.eight.osgi.IBundleService"
post-unregistration="unregistered"  post-registration="registered">
                 <property name="service.description" type="string" value="matcher"/>
             </provides>
             <jmx:config>
                 <jmx:method name="setConfig"/>
                 <jmx:method name="setPermit"/>
                 <jmx:method name="setLogger"/>
                 <jmx:method name="setTrace"/>
                 <jmx:method name="setClone"/>
                 <jmx:method name="control"/>
                 <jmx:method name="state"/>
                 <jmx:method name="trace"/>
                 <jmx:method name="debug"/>
                 <jmx:method name="info"/>
             </jmx:config>
        </component>
        <instance name="matcher" component="matcher">
              <property name="requires.filters">
             <property name="proxy" value="(service.description=matcher_*)"/>
              </property>
              <property name="context" value="context"/>
              <property name="close" value="close"/>
              <property name="log" value="log"/>
              <property name="logger" value="log.test|this.one"/>
              <property name="holder" value="beanHolder"/>
              <property name="hookid" value="matcher"/>
              <property name="reload" value="true"/>
           <property name="service.ranking" value="2100"/>
           <property name="service.description" value="matcher"/>
        </instance>
```

北京漫汇为都科技有限公司（版权所有）

```
</ipojo>
```

该配置定义了一个名为matcher的component，并为这个component创建了一个实例(instance)。组件是由 eight 提供的 net.yeeyaa.eight.osgi.runtime.BundleCenter 来创建的，该类会加载本项目中 spring 定义的各种元件，并为组件内外提供双向的代理。BundleCenter 的配置参数较多，涉及配置文件的信息、操作特征、监控和跟踪、日志、组件的类别与名称、暴露的服务接口等。Component 预定义的参数，在 instance 实例化时可以修改。

模块项目使用 maven 打包后，放置到容器中，就可以生成对应的组件实例。一般是放置在指定目录下，eight 平台的加载模块，会定时扫描指定目录以动态加载、运行、更新和卸载模块。

同样的方式可以定义 counter 和 documents。值得注意的是，因为 counter 需要共识指代，而这些共识又都在组件外部，不能直接引用，此时需要用代理来占位。代理是 eight 提供的用以跨组件边界进行关联的元件，它实现了 IUniversal 接口。counter 元件的定义如下：

```
<bean class="net.yeeyaa.test.count.Counter" id="counter">
        <property name="documents">
                <bean class="net.yeeyaa.eight.core.PlatformUniversal$$Proxy">
                        <property name="beanHolder" ref="centerHolder"/>
                        <property name="bean"
value="${framework.test.counter.documents:documents}"/>
                </bean>
        </property>
        <property name="processor">
                <bean class="net.yeeyaa.eight.core.PlatformUniversal$$Proxy">
                        <property name="beanHolder" ref="centerHolder"/>
                        <property name="bean"
value="${framework.test.counter.processor:processor}"/>
                </bean>
        </property>
</bean>
```

可见 counter 使用了 net.yeeyaa.eight.core.PlatformUniversal 代理，将共识指代代理到组件外部，这个代理定义的 name 默认为 documents 和 processor，在运行时，eight 容器将这些指代通过 linker 连接到另一个组件内部的元件上（由 spring 配置的 bean）。

2.3.2. 模块的运行环境

所有的组件开发完毕后，可以放置到指定的目录下去加载。基本的容器目录结构如下：

```
> runtime
    > bundle
    > conf
        config.properties
        logback.xml
        system.properties
    > lib
        jta-1.1.jar
        org.apache.felix.main.jar
    > work
        > system
            eight-osgi-counter-1.0.0.jar
            eight-osgi-documents-1.0.0.jar
            eight-osgi-matcher-1.0.0.jar
```

图 2.4 eight 容器的目录结构

北京漫汇为都科技有限公司（版权所有）

eight 容器基于 felix osgi 容器扩展而成。

bundle 目录下放置了大量的符合 osgi 规范的库包（lib），其中有一部分是 eight 平台提供的元件库。这些库用以实现各种功能和组建各种组件。这些基础库稳定而不易变化。

conf 目录下则是各种配置，包括配置 felix 环境的 config.properties，配置日志的 logback.xml，配置 eight 容器以及一些核心组件的 system.properties，还有其他一些基础配置也可能放置于此（例如提供 https 服务时的 ca 证书等）。

lib 目录下是 felix 容器的核心运行环境，是固定的两个包。

work 目录就是加载组件的目录，其中动态组件和 config 配置（可视为 instance，如果配置多个 config，则可以为组件生成多个实例，这些实例配置参数可以不同）都放置在 system 目录下。前面定制的三个模块就已经放置就绪。

eight 需要在 java 1.6 以上的环境中运行，执行 java -cp lib/org.apache.felix.main.jar org.apache.felix.main.Main 命令即可启动容器和加载组件。

启动完毕后使用 components 命令可以查看发布的组件，instances 命令可以查看运行的实例。

```
g! components
Factory linker (VALID)
Factory boot (VALID)
Factory file (VALID)
Factory center (VALID)
Factory org.apache.felix.ipojo.webconsole.IPOJOPlugin (VALID)
Factory documents (VALID)
Factory matcher (VALID)
Factory counter (VALID)
Factory org.apache.felix.ipojo.arch.gogo.Arch (UNKNOWN) - Private
```

图 2.5 eight 容器中定义的组件

```
g! instances
Instance boot_file_store -> valid
Instance org.apache.felix.ipojo.arch.gogo.Arch-0 -> valid
Instance boot -> valid
Instance file -> valid
Instance org.apache.felix.ipojo.webconsole.IPOJOPlugin-0 -> valid
Instance documents -> valid
Instance matcher -> valid
Instance counter -> valid
```

图 2.6 eight 容器中运行的实例

可见，前面定义的 matcher、counter 和 documents 组件已经被成功发布并且已经生成了对应的实例。另外一些组件和实例则是 eight 预定义的，提供平台核心服务。

此时虽然三个实例已经在运行，但是并没有关联起来，counter 所指代的共识并没有与 matcher 和 documents 相关联。注意到 eight 系统提供了一个 linker 组件，这个组件就用于指代关联。可以用 config 生成它的一个实例。

定义一个文件名为 linker-counter_matcher_processor.config 的 config 文件，其中第一部分 linker 表示生成 linker 组件的一个实例，后面的 counter_matcher_ processor 遵从约定，分别为指代实例名，被指代实例名，以及指代共识的名称，这里是：

<property name="name" value="${framework.test.counter.processor:processor}"/>

在这个 config 内部是一个 properties 配置文件，配置如下：

bean="matcher"

service.description="counter_matcher_processor"

name="processor"

requires.filters=["service","(service.description\=matcher)"]

rank="999"

service.description 标识了这个 linker 实例的名称，counter_matcher_processor 名称，匹配了 counter 配置里 service.description=counter_*的约定，可以关联到 counter 上。

name="processor"则宣称 linker 是用以代替一个名为 processor 的指代的。requires.filters 表示这个 lnker 连接实例的过滤条件，这里可以看到它需要 service.description=matcher，也即对接到 matcher 这个实例上。bean 表示连接到实例的哪一个元件上，这里是名为 matcher 的 bean。rank 则表示优先级，当有多个符合条件的 linker 时，优先级高的将替换优先级低的。

linker 配置过后，可以用 instance counter 命令查看 counter 实例的当前状况：

instance name="counter" state="valid" bundle="144" component.type="counter"
handler name="org.apache.felix.ipojo:requires" state="valid"
requires specification="java.util.concurrent.ExecutorService" id="executor" optional="true" aggregate="false" proxy="true" binding-policy="dynamic" state="resolved"
uses service.id="87"
selected service.id="87"
matches service.id="87"
requires specification="net.yeeyaa.eight.osgi.IBundleProxy" id="proxy" filter="(service.description=counter_*)" optional="true" aggregate="true" proxy="true" binding-policy="dynamic" state="resolved"
uses service.id="120" instance.name="counter_matcher_processor"
selected service.id="120" instance.name="counter_matcher_processor"
matches service.id="120" instance.name="counter_matcher_processor"

可见，net.yeeyaa.eight.osgi.IBundleProxy 代理的 service.description=counter_*的服务，已经动态绑定到 counter_matcher_processor 这个 linker 上。

再查看 counter_matcher_processor 这个 instance，如下：

instance name="counter_matcher_processor" state="valid" bundle="33" component.type="linker"
handler name="org.apache.felix.ipojo:requires" state="valid"
requires specification="net.yeeyaa.eight.osgi.IBundleService" id="service" filter="(service.description=matcher)" optional="true" nullable="true" aggregate="false" proxy="true" binding-policy="dynamic-priority" comparator="org.apache.felix.ipojo.util.ServiceReferenceRankingComparator" state="resolved"
uses service.id="103" instance.name="matcher"
selected service.id="103" instance.name="matcher"
matches service.id="103" instance.name="matcher"

可见这个 linker 已经关联到了 matcher 这个组件上了，由此，counter 的 processor 指代，通过 linker 的配置，指向了另一个 matcher 组件里的 matcher 元件。如图：

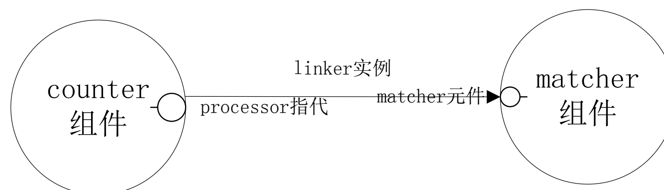


图 2.7 linker 链接 counter 和 matcher

同样的方式，将 counter 于 documents 链接起来。为了能够调用 counter 提供的统计功能，还配置几个额外的实例用以通过 http 请求来接受参数和返回结果，最后的系统结构如下：

北京漫汇为都科技有限公司（版权所有）

```
◢ > system
    documents-documents.config
    eight-osgi-counter-1.0.0.jar
    eight-osgi-documents-1.0.0.jar
    eight-osgi-jsp-1.0.0.jar
    eight-osgi-matcher-1.0.0.jar
    linker-counter_documents_documents.config
    linker-counter_matcher_processor.config
    linker-jsp_counter_calculator.config
```

图 2.8 配置完成的 counter 系统

```
instances
Instance org.apache.felix.ipojo.arch.gogo.Arch-0 -> valid
Instance boot_file_store -> valid
Instance file -> valid
Instance boot -> valid
Instance org.apache.felix.ipojo.webconsole.IPOJOPlugin-0 -> valid
Instance counter_documents_documents -> valid
Instance counter_matcher_processor -> valid
Instance jsp_counter_calculator -> valid
Instance jsp -> valid
Instance documents -> valid
Instance matcher -> valid
Instance counter -> valid
```

图 2.9 counter 系统的运行实例

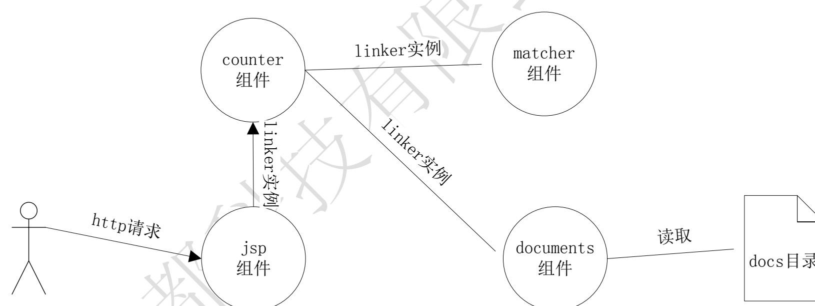


图 2.10 counter 系统组件结构图

在运行环境的 docs 目录下放置几个测试文件：

```
◢ > runtime
  › > bundle
  › > conf
  ◢ > docs
      a.txt
      b.txt
```

图 2.11 在指定目录下放置测试文件

查询的结果如下(其中一个文件出现 2 次关键词，另一个出现 1 次，共 3 次)：

北京漫汇为都科技有限公司(所有)

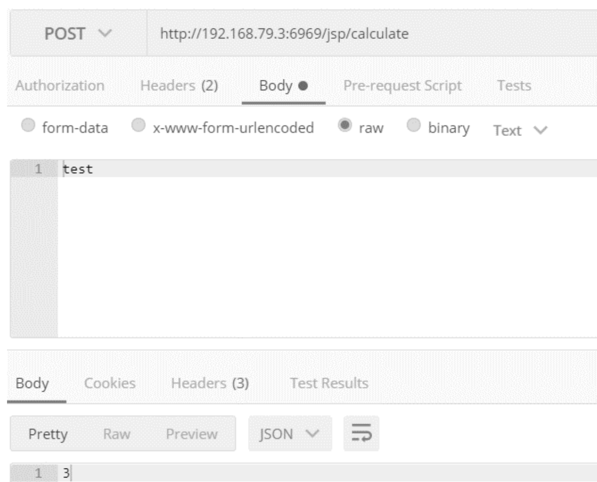


图 2.12 查询关键字出现的次数

2.3.3. 系统的动态变化

假设业务需求发生了变化，现在要求查找关键词出现最多的文件里出现了多少次关键词。这部分功能由 counter 负责，则修改其逻辑并打包：

```
public Integer process(String keyword) {
        Map<Object[], String> docs = documents.all();
        int max = 0;
        for (String content : docs.values()) {
                int count = processor.perform(keyword, content);
                if (count > max) max = count;
        }
        return max;
}
```

打包后的模块放置到 work/system 目录下，counter 组件会自动更新，此时同样的请求发送过去，该系统返回的是出现关键词最多的次数（2 次），counter 更新不影响系统其他部分。

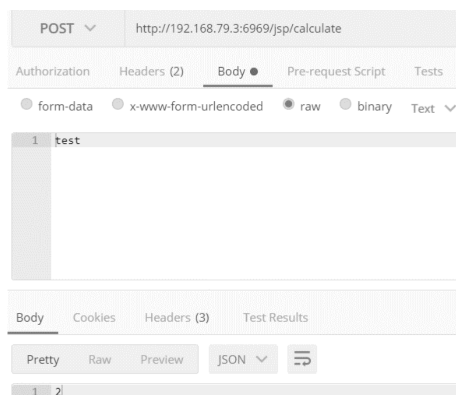


图 2.13 局部功能发生变化，则处理结果随之改变

前面提到，counter 这个组件的视角里，并不约定 processor 的处理逻辑，仅认为它是一个对文本内容和关键字进行统计的处理过程。

北京漫汇为都科技有限公司（版权所有）

现在，假设系统的功能发生如下变化：

1）在目录 scores 下放置着每个班的各科目成绩，一班一个文件。以 csv 格式存放，结构如下：

name	english	math	physical
张三	90	68	72
李四	77	100	95

2）matcher 的处理逻辑修改为根据关键词（科目名称）找到当前班级该科目最高分；

3）要求传入一个科目名称，则查找该科目全校最高分。

此时，counter 的逻辑不需要修改。需要修改 matcher 的处理逻辑，并把 documents 的路径设置为 scores/。之前在配置 documents 时已经提供了 test.documents.base 参数，用于设置文件路径，在 documents 实例的配置文件 documents-documents.config 对其加以调整：

config="test.documents.base#scores/"

设置 config 的参数 test.documents.base#scores/，则将运行实例的 test.documents.base 修改为了 scores/。此时 scores 目录下有 class 1.csv 和 class 2.csv 两个班级的成绩，查询各科目成绩：

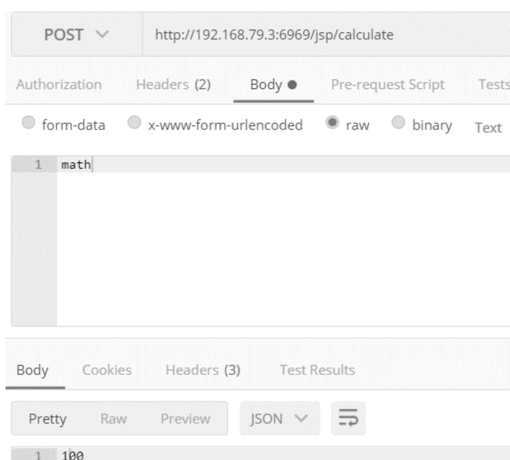


图 2.14 数学成绩的最高分

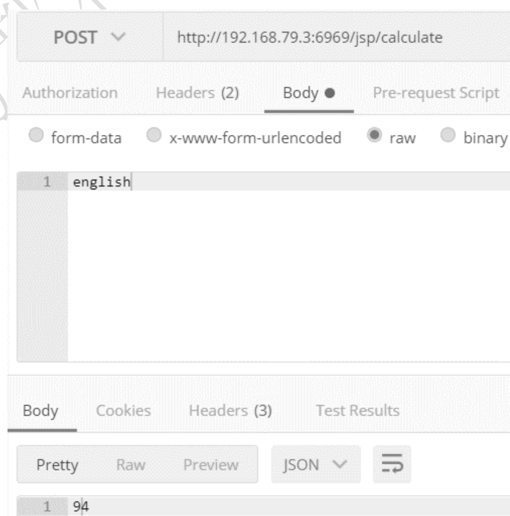


图 2.15 英语成绩的最高分

由此，整个系统的功能由统计关键词出现的次数变为了计算科目成绩的全校最高分。

假如需要调整一下输入输出的格式，将输出改为 json 样式，可以在 counter 前增加一个 formatter 组件。

北京漫汇为都科技（版权所有）

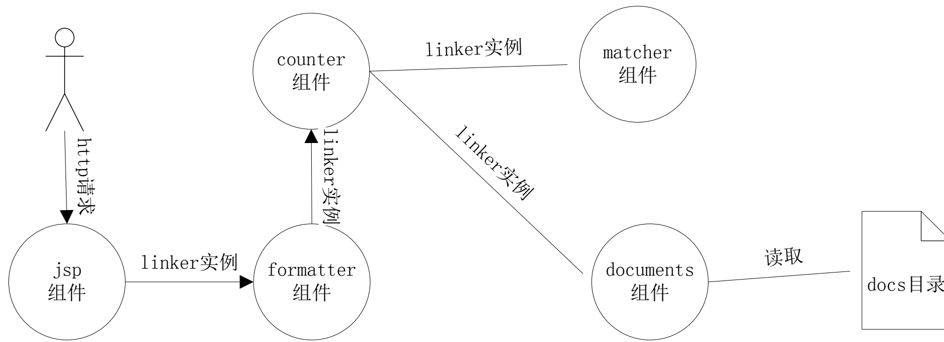


图 2.16 处理流程中嵌入新的处理步骤

formatter 模块：

```
public class Formatter implements IProcessor<String, String>{
   protected IProcessor<String, Object> processor;

   public void setProcessor(IProcessor<String, Object> processor) {
          this.processor = processor;
   }

   @Override
   public String process(String input) {
          Object ret = processor.process(input);
          return                          Json.createObjectBuilder().add("result",
ret.toString()).build().toString();          }
}
```

将 formatter 实例优先级设置的高于 counter，则优先被 jsp 组件使用，同时其自身连接到 counter 上，这样当存在 formatter 组件，返回结果就是 json 格式，当将 formatter 组件卸载时，就恢复到原始格式。

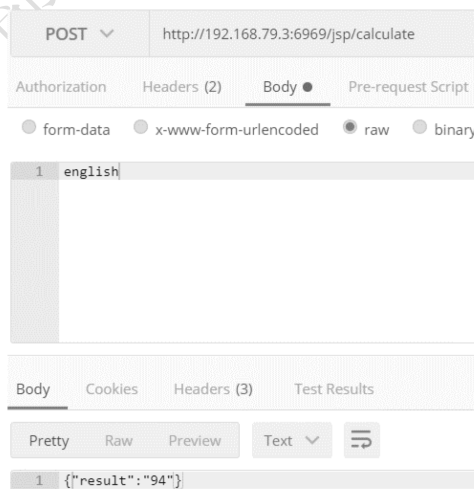


图 2.17 修改输出样式为 json

同样的，如果想将输出设置为 xml 样式，只需要更新 formatter 即可。

北京漫汇为都科技有限公司（版权所有）

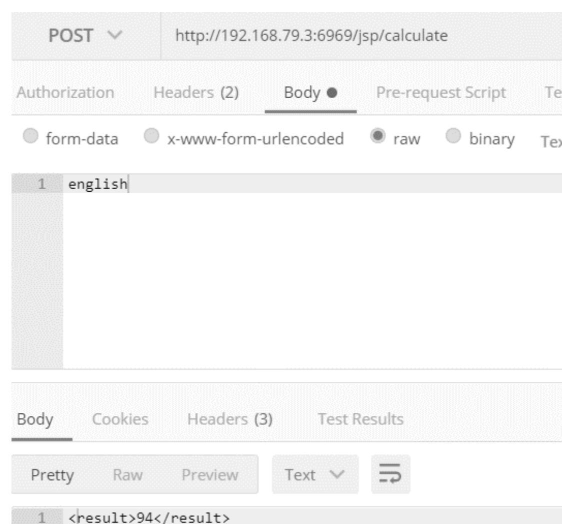


图 2.18 修改输出样式为 xml

由上可见，eight 是一个动态的系统，无论是系统配置还是功能逻辑，乃至系统结构都能够在运行时不断变化，这种变化可以发生于任意的局部或是整体系统中。一个系统完全可以在 eight 的一次运行过程中组装、运行、变更、卸载乃至变化为另一种新系统。

3. eight 的应用

eight 较之传统的单一应用以及当下盛行的微服务架构，有着明显的优势。eight 也克服了 osgi 框架的弱点，比之有着明显的进步。

其具备的优势与微服务类似且更为进步：

1） 将一个大型的系统划分为多个模块，模块之间无耦合（或弱耦合），各个模块能够独立开发，在运行时进行组装。
相较于微服务，模块之间的耦合更弱，相互衔接的模块通过共识接口而进行，独立性更强。而微服务则存在上下游依赖关系，下游服务必须充分了解上游服务的接口，而这意味着下游对上游的依赖，这些依赖将一直体现在开发、更新、运维等过程中。
相较于微服务，eight 的组件粒度更细。微服务还是以服务为基本单元，其运行服务的资源代价相对高，决定了它不宜将服务拆解得更为细碎。而对 eight 而言，其粒度为代码级。任何一部分的代码如果有其独立性（即便仅有一个类），与其它代码块的关联具有可变性，均可以切割为组件。eight 划分组件的原则是孤立耦合性，只要存在不确定依赖的关联，均可以划分开来。这样的系统在未来发生变化时，就能做到何处改变、何处切换；

2） 各个模块生命周期与迭代相对独立，可以独立的开发、测试、部署和更新，单一局部的调整不影响整体，可以快速开发和迭代。
相较于微服务，eight 的组件粒度更小，其调整涉及的系统部位更精确，其开发与迭代的代价与周期也更小。eight 的运行环境更为简单（仅需要用 java 运行一个容器），模块的启动与加载更为方便迅捷（仅需要向工作目录拖放组件），则其进行测试与部署更为容易。eight 对于软件开发的支持成本更低，效用更佳；

3） 系统可以根据使用需求进行局部扩容，以满足业务增长的需要。
相较于微服务，eight 在系统弹性上更具备独特的优势。对于一个 eight 容器而言，其系统从组装到卸载都是动态的。一个容器内运行着什么样的组件、多少组件，都是随时可变化

北京漫汇为都科技有限公司（版权所有）

的。需要扩容的组件，可以随时部署到某个容器中，并且与其他组件服务并存，这与 docker 这类的微服务独占式占用容器资源是不同的。而在容器内资源是共享的，不频繁使用的组件不会占用（或少量占用）cpu 和内存，而活跃组件则自然会使用到更多的计算资源。由此，也无需对运行单元进行频繁调度，减轻了运维成本；

4） 模块的复用性和组建模块仓库。

对于 eight 而言，其强大的模块化能力来源于基于共识的组件隔离，组件间的耦合是比微服务更为薄弱的。这样的组件其复用性更高，也更适于构建组建仓库，成为积累软件资本、迅速搭建系统的重要手段。

同时，eight 也消除了微服务的很多缺陷。

5） eight 为单体应用，独立运行于单一进程之中，无须复杂的支撑框架，其开发与运维对使用方的软硬件资源和人员管理水平要求不高。总体而言，运行 eight 不会比运行 spring boot 更复杂；

6） eight 为单体应用，不涉及进程间通信或跨网通信，也不存在依赖的服务不可获得的问题。占用资源少，对系统的监控、管理要求低，不会带来额外的消耗；

7） eight 元件高度可复用，意味着更少的类和更好的代码加速。更少的类带来更少的内存占用，更好的代码加速带来更高效的 cpu 使用，使得资源能更佳的使用；

8） 单体应用也易于实现数据处理的事务性和一致性；

9） eight 的组件加载和更新，其响应比微服务更迅速（亚秒级），基本上不存在局部服务中断导致的 stw 问题。可以在业务无感的情况下，实现系统更新（通常在系统更新前执行的业务会依旧使用旧组件，直到执行完毕，旧组件才被释放；同时新组件已经开始为后续业务服务，此动态升级过程不会影响到业务执行）。

实际上，正如 openstack 的硬件虚拟化之于 docker，eight 与容器也位于不同的层面上，eight 本身可以与微服务容器很好的结合。eight 可以基于微服务平台（如 k8s）迅速部署到成千上万个容器上，而 eight 的局部可变性和可配置性，以及系统配置管理集中化（都在一个目录下），可以大大提升微服务集群的可变性，允许微服务在启动 eight 容器后，系统进行动态变化，大大增加了灵活性，降低了运维成本。

eight 也存在一些缺陷，eight 比之微服务，主要问题就是组件在同一进程中运行，则组件无法有效隔离。一旦某些组件出现问题，可能导致其他组件的运行受到影响。

eight 也克服了 osg 的弱点。osgi 并没有能有效的降低模块间的依赖，模块间的依赖也导致了动态性的减弱。osgi 模块间存在着大量的直接关联，导致模块无法有效卸载和释放，一次强制更新，可能导致大量的模块被同步卸载，影响范围难以控制，消耗成本也高。这也让其动态性优势难以发挥。

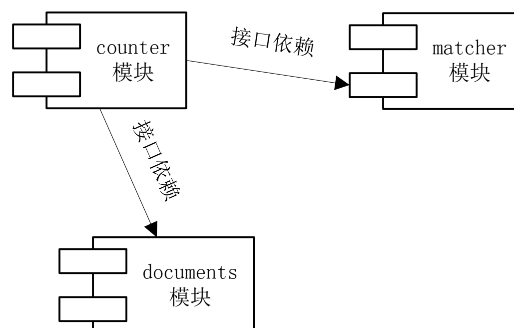


图 3.1 osgi 基于接口依赖的模块架构

由上图可见，osgi 的开发，模块间的依赖是通过接口进行的，存在开发时与运行时的双重耦合。由于开发时 counter 依赖于 matcher 和 documents 提供的接口，所以，counter 开发者必须了解这些接口才能进行开发。而运行时，一旦 matcher 和 documents 模块进行了修改需要重新载入，则 counter 也需要随之重新载入。即便将 matcher 和 documents 的接口独立打包，matcher 和 docuemnts

北京漫汇为都科技有限公司（版权所有）

的功能逻辑修改自然不必影响 counter，但是接口本身的耦合仍然存在，一旦接口的样式和使用方式发生变化，则 counter 的修改也将是不可避免的。

相对而言，eight 则是一个层次结构，如图：

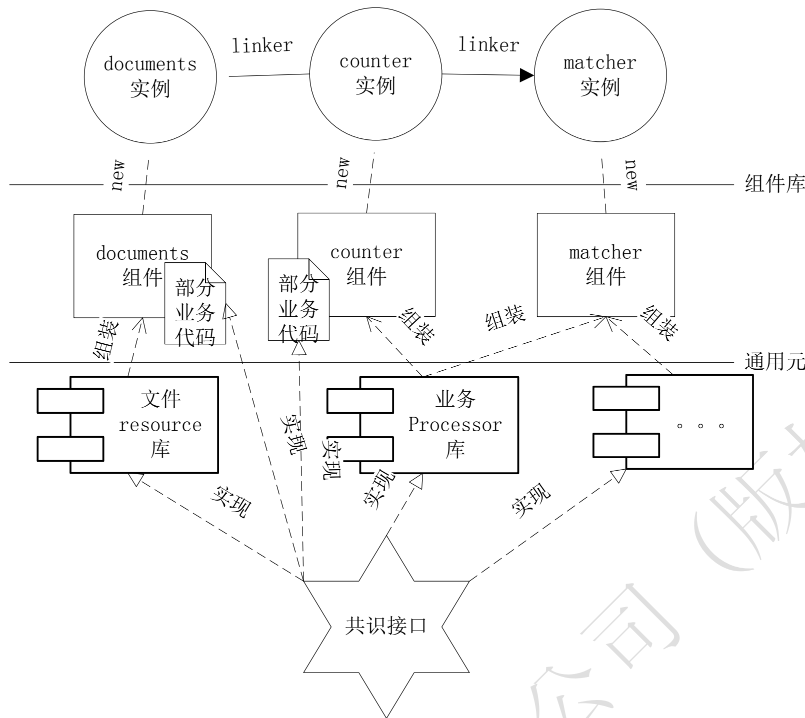


图 3.2 eight 基于组件连接的模块架构

eight 采用了高度抽象的共识世界观，其能够开发大量的相互独立的元件。这些元件组成基础的元件库，具备高度复用的能力。这一部分的代码库是很少需要变动的。

其上是涉及具体功能的组件，这些组件可能是由一个或多个基础元件组装而成，当然其本身也可能会包含一些特定的代码实现。这些组装（通过 spring 之类的组装框架）形成一个个独立的模块，这些模块对外界的引用都通过共识接口来指代。此时，这些组件仍是静态的，彼此无关的。

在动态的运行环境里，组件将生成实例。组件生成实例的方式与类通过构造函数传入参数生成对象相似，组件也是可以参数化的，不同参数导致不同实例的性状不同。实例之间再通过 linker 将共识指向其它实例。此时，一个运行时系统才真正搭建起来。一个典型的 eight 服务系统如下：

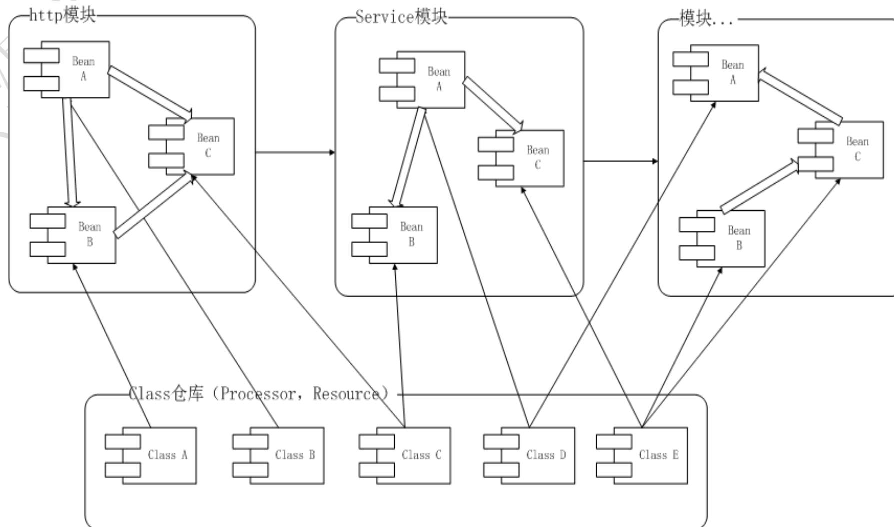


图 3.3 eight 运行实例与元件仓库的关系

北京漫汇为都科技有限公司（版权所有）

无论是元件库还是组件都是独立可复用的模块。元件库会更为通用，并且更稳定，通用元件库一般放在 bundle 目录下随系统运行，而组件则随 config 实例一起放置于 work/system 目录下进行动态加载和卸载。

由此可见，eight 的依赖是垂直的，实例之间是没有依赖的，任何一个实例的变化（如参数调整）不会导致其他部分的改动。组件之间也没有关联，当组件本身的代码逻辑或组装方式发生变化时，影响的仅仅是其实例。只有当最底层的元件库发生变化时才可能影响较多模块。而 eight 定义里的元件库是形如 java 基础库一样的基本件，其变化频率很低。元件库变化需要容器重新发布和重启。所以通常而言，变化集中在 work/system 目录下的组件和实例中，这部分是高度动态并且影响受限的。由此，eight 的组件独立性和系统的灵活度要远高于通常 osgi 框架。

3.1. eight 提供的基础元件库

eight 提供了大量的基础元件以供使用，主要的库包括：

1）eight-base：包含了 eight 的共识接口和异常定义；

2）eight-core：包含了 eight 大量常见的过程（processor）和资源（resource）、工具类等；

3）eight-common：包含了 eight 大量常见的过程和资源、工具类等，这些元件除了 java 核心库外，还依赖 spring、groovy、moxy 等第三方库；

4）eight-access：主要涉及与外界进行交互的元件，如 http、jms、websocket、webservice 等等；

5）eight-osgi：eight 在 osgi 框架下的支持元件，涉及到 osgi 容器的运行环境和诸多服务；

6）eight-data：eight 进行数据访问的元件，涉及数据库、redis、memcache 等访问元件；

7）eight-service：eight 提供功能服务的元件，包括服务发布与检索，过滤器，会话管理等元件；

8）eight-client：eight 访问服务的客户端，包括登录验证、数据的串行化和反串行化等；

9）eight-share：客户端与服务端共享的一些元件；

10）eight-annotation、eight-annotation-tag：用来使用 annotation 方式加载 eight 的服务，用 annotation 写的类无须实现 eight 的共识接口就可以被 eight 平台加载和使用；

11）eight-agent：eight 使用 java agent 的一些元件；

12）eight-aspect、eight-ss：eight 对切面以及 spring security 等的一些支持。

以下选择几个常见示例介绍 eight 元件的设计、开发方式以及元件的组装方法。

3.1.1. ChainXProcessor 链式过程

链式过程代表一个处理过程，其处理是一系列处理过程的顺序组合。此 processor 内置一个 processors 的 list 列表。列表中 processor 将依次执行，前一输出为后一输入。这个过程元件的输入为第一个 processor 的输入，输出为最后一个 processor 的输出。如图：

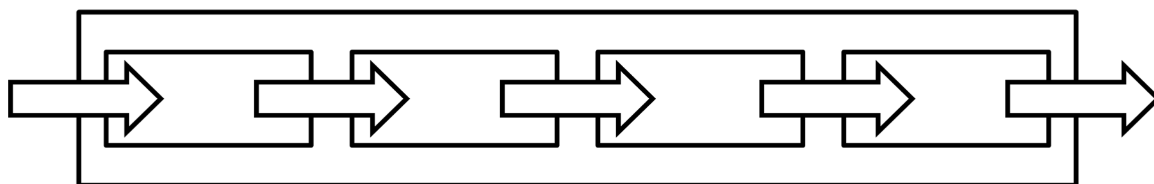


图 3.4 ChainXProcessor 链式过程

北京漫汇为都科技有限公司（版权所有）

这种元件常用于将一系统简单元件组合起来定制一个复杂的处理过程。实现代码：

```
public class ChainXProcessor<T> implements IProcessor<T, T>{
	protected final Logger log;
	protected Collection<IProcessor<T, T>> processors;
	protected Boolean ignoreError = false;;
	protected Boolean nullable = false;
	protected Boolean sync;

	public ChainXProcessor() {
		this.log = LoggerFactory.getLogger(ChainXProcessor.class);
	}

	public ChainXProcessor(Logger log) {
		this.log = log == null ? LoggerFactory.getLogger(ChainXProcessor.class) :
log;
	}

	public void setSync(Boolean sync) {
		this.sync = sync;
	}

	public void setNullable(Boolean nullable) {
		if(nullable != null) this.nullable = nullable;
	}

	public void setIgnoreError(Boolean ignoreError) {
		if(ignoreError != null) this.ignoreError = ignoreError;
	}

	public void setProcessors(Collection<IProcessor<T, T>> processors) {
		this.processors = processors;
	}

	@Override
	public T process(T in) {
		if(processors != null && processors.size() > 0) if (sync == null)
for(IProcessor<T, T> processor : processors)  synchronized(this) {
				try{
					in = processor.process(in);
					if(in == null && !nullable) return null;
				}catch(Exception e){
					log.error("ChainXProcessor: processor failed.", e);
					if(!ignoreError) if (e instanceof PlatformException) throw
(PlatformException) e;
					else                    throw                    new
PlatformException(PlatformError.ERROR_OTHER_FAIL, e);
```


北京漫汇为都科技有限公司（版权所有）

```
                        else if(!nullable) return null;
                    }
            } else if(sync) synchronized(this) {
                    for(IProcessor<T, T> processor : processors)  try{
                        in = processor.process(in);
                        if(in == null && !nullable) return null;
                    }catch(Exception e){
                        log.error("ChainXProcessor: processor failed.", e);
                        if(!ignoreError) if (e instanceof PlatformException) throw
(PlatformException) e;
                        else                    throw                    new
PlatformException(PlatformError.ERROR_OTHER_FAIL, e);
                        else if(!nullable) return null;
                    }
            } else for(IProcessor<T, T> processor : processors)  try{
                    in = processor.process(in);
                    if(in == null && !nullable) return null;
            }catch(Exception e){
                    log.error("ChainXProcessor: processor failed.", e);
                    if(!ignoreError)  if  (e  instanceof  PlatformException)  throw
(PlatformException) e;
                    else  throw  new  PlatformException(PlatformError.ERROR_OTHER_FAIL,
e);
                    else if(!nullable) return null;
            }
            return in;
        }
}
```

主要的参数是一个 Collection<IProcessor<T, T>>的 processors 列表；ignoreError 表示当其中某一环发生异常时是否继续执行；nullable 表示是否允许上游环节返回空值并作为参数传递给下游，如果不允许则返回空退出；sync 表示整个执行过程是否是同步的（线程安全）。

由上可见，ChainXProcessor 是一个适应性较强的元件，为各种应用场景提供了可调整的参数。

ChainXProcessor 的实现代码是典型的 eight 元件的模式：

1） 用单一的类表达一个完整的元件；
2） 为该类涉及的领域设置参数，以供 spring 注入；
3） 除了明确使用到的类，其它的指代均以共识接口提供；
4） 本身实现了某个或多个共识接口，可供其它元件使用。

3.1.2. ProxyProcessor 代理过程

代理过程将代理另一个过程，它可以配置 preProcessor（预处理）、postProcessor（后处理）、aroundProcessor（中间处理），用法类似于 aop。如图：

北京漫汇为都科技有限公司（版权所有）

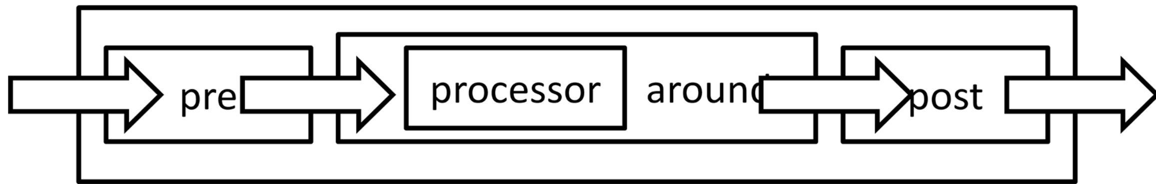


图 3.5 ProxyProcessor 代理过程

```
public class ProxyProcessor<T, R> implements IProcessor<T, R> {
        protected final Logger log;
        protected IProcessor<T, T> preProcessor;
        protected IProcessor<T, R> processor;
        protected IProcessor<R, R> postProcessor;
        protected IProcessor<Object[], R> aroundProcessor;

        public ProxyProcessor() {
                this.log = LoggerFactory.getLogger(ProxyProcessor.class);
        }

        public ProxyProcessor(Logger log) {
                this.log = log == null ? LoggerFactory.getLogger(ProxyProcessor.class) :
log;
        }

        public void setAroundProcessor(IProcessor<Object[], R> aroundProcessor) {
                this.aroundProcessor = aroundProcessor;
        }

        public void setPreProcessor(IProcessor<T, T> preProcessor) {
                this.preProcessor = preProcessor;
        }

        public void setProcessor(IProcessor<T, R> processor) {
                this.processor = processor;
        }

        public void setPostProcessor(IProcessor<R, R> postProcessor) {
                this.postProcessor = postProcessor;
        }

        @Override
        public R process(T instance) {
                try{
                        R ret = null;
                        if(preProcessor != null) instance = preProcessor.process(instance);
                        if(aroundProcessor != null) ret = aroundProcessor.process(new
Object[]{processor, instance});
```

北京漫汇为都科技有限公司（版权所有）

```
                else if(processor != null) ret = processor.process(instance);
                if(postProcessor != null) ret = postProcessor.process(ret);
                return ret;
            }catch(Exception e){
                log.error("ProxyProcessor: processor failed.", e);
            }
            return null;
        }
    }
```

ProxyProcessor 同样是典型的 eight 元件开发方式。

3.1.3. ProcessorPResource 转换加工 resource

ProcessorPResource 是一种带上了输入输出转化器的 resource，它代理另一个 resource。能够用配置好的转化器（一种 processor）对 resource 的输入（key）和输出（value）进行转化，以适配调用方的需求。

该 resource 需要配置 in，out，paraConvertor 三个 processor，用以对 key 和 value 进行进一步加工和转换。in 可以对 find 得到的 value 进行处理。out 可以对 put 输出的 value 进行处理。paraConvertor 可以对 key 进行处理。如图：

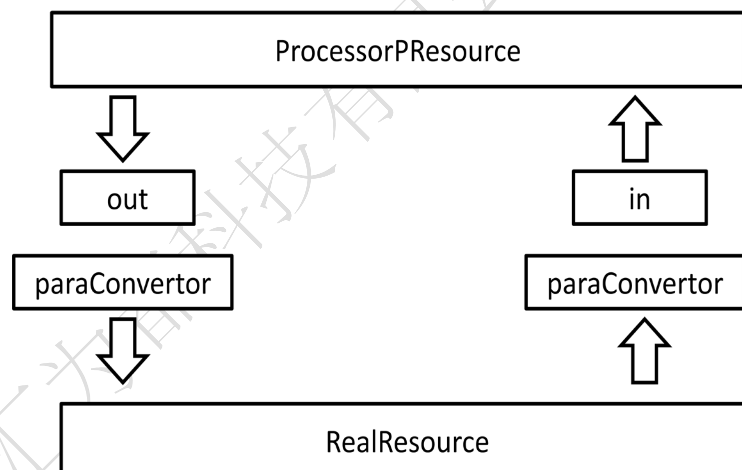


图 3.6 ProcessorPResource 转换加工 resource

代码实现如下：

```
public class ProcessorPResource<K, V, U extends IResource<K, V>> extends ProxyResource<U>
implements IResource<K, V> {
    protected IProcessor<V, V> in;
    protected IProcessor<V, V> out;
    protected IProcessor<Object[], K[]> paraConvertor;

    public void setIn(IProcessor<V, V> in) {
        this.in = in;
    }
```

```
	public void setOut(IProcessor<V, V> out) {
		this.out = out;
	}

	public void setParaConvertor(IProcessor<Object[], K[]> paraConvertor) {
		this.paraConvertor = paraConvertor;
	}

	@Override
	public V find(K ... paras) {
	      if(paraConvertor != null) paras = paraConvertor.process(new Object[]{"find",
paras});
		V ret = resource.find(paras);
		if(in != null) ret = in.process(ret);
		return ret;
	}

	@Override
	public <P> P store(V value, K ... paras) {
		if(paraConvertor   !=   null)   paras   =   paraConvertor.process(new
Object[]{"store", paras});
		if(out != null) value = out.process(value);
		return resource.<P>store(value, paras);
	}

	@Override
	public <P> P discard(K ... paras) {
		if(paraConvertor   !=   null)   paras   =   paraConvertor.process(new
Object[]{"delete", paras});
		return resource.discard(paras);
	}

	@Override
	public <P> P empty(K... paras) {
		if(resource instanceof IOutputResource) {
			if(paraConvertor  !=  null)  paras  =  paraConvertor.process(new
Object[]{"empty", paras});
			return ((IOutputResource<K, V>)resource).<P>empty(paras);
		}
		return null;
	}
}
```

北京漫汇为都科技有限公司（版权所有）

3.1.4. CascadeLPResource 嵌套代理 resource

CascadeLPResource 是一个嵌套 resource，该 resource 代理另一个 resource，要求被代理的 resource 的 value 必须是一个 map。对于这个 map，此 resource 将之视为一个下级 resource，并按 key 进行存取。

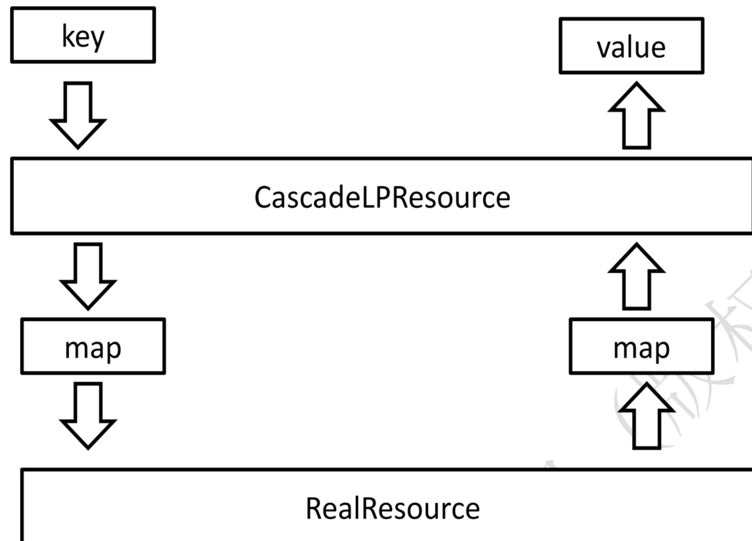


图 3.7 CascadeLPResource 嵌套代理 resource

```
public class CascadeLPResource<K, V, U extends IInputResource<K, V>> extends
ProxyResource<U> implements IListableResource<K, V> {
	protected Boolean setBack = true;

	public void setSetBack(Boolean setBack) {
		if(setBack != null) this.setBack = setBack;
	}

	@Override
	public V find(K ... paras) {
		if(paras == null || paras.length == 0 || resource == null) return null;
		K[] reparas = PlatformUtil.newArrayOf(paras, paras.length - 1);
		for(int i =0; i < reparas.length; i++) reparas[i] = paras[i+1];
		Object hm = resource.find(reparas);
		if(Map.class.isInstance(hm)) return ((Map<K, V>)hm).get(paras[0]);
		return null;
	}

	@Override
	public <P> P store(V value, K ... paras) {
		if(paras == null || paras.length == 0 || resource == null) return null;
		K[] reparas = PlatformUtil.newArrayOf(paras, paras.length - 1);
		for(int i =0; i < reparas.length; i++) reparas[i] = paras[i+1];
		Object hm = resource.find(reparas);
```


北京漫汇为都科技有限公司（版权所有）

```
			if(!Map.class.isInstance(hm) && resource instanceof IOutputResource) hm =
new ConcurrentHashMap<String, Object>();
			if(Map.class.isInstance(hm)){
				((Map)hm).put(paras[0], value);
				if(setBack && resource instanceof IOutputResource)
((IOutputResource)resource).store(hm, reparas);
			}
			return null;
		}

		@Override
		public <P> P discard(K ... paras) {
			if(paras == null || paras.length == 0 || resource == null) return null;
			K[] reparas = PlatformUtil.newArrayOf(paras, paras.length - 1);
			for(int i =0; i < reparas.length; i++) reparas[i] = paras[i+1];
			Object hm = resource.find(reparas);
			if(Map.class.isInstance(hm)&&((Map)hm).containsKey(paras[0])){
				((Map<K, V>)hm).remove(paras[0]);
				if(setBack && resource instanceof IOutputResource)
((IOutputResource)resource).store(hm, reparas);
			}
			return null;
		}

		@Override
		public <P> P empty(K... paras) {
			if(paras == null || paras.length == 0 || resource == null) return null;
			K[] reparas = PlatformUtil.newArrayOf(paras, paras.length - 1);
			for(int i =0; i < reparas.length; i++) reparas[i] = paras[i+1];
			Object hm = resource.find(reparas);
			if(Map.class.isInstance(hm)){
				((Map)hm).clear();
				if(setBack && resource instanceof IOutputResource)
((IOutputResource)resource).store(hm, reparas);
			}
			return null;
		}

		@Override
		public Collection<K[]> keys(K... paras) {
			if(paras != null && paras.length > 0 && resource != null) {
				K[] reparas = PlatformUtil.newArrayOf(paras, paras.length - 1);
				for(int i =0; i < reparas.length; i++) reparas[i] = paras[i+1];
				Object hm = resource.find(reparas);
				if(Map.class.isInstance(hm)){
					ArrayList<K[]> ret = new ArrayList<K[]>(((Map)hm).size());
```

北京漫汇为都科技有限公司（版权所有）

```
                    List<K> ls = Arrays.asList(paras);
                    for(K o : ((Map<K, V>)hm).keySet()){
                        ls.set(0, o);
                        ret.add(ls.toArray(PlatformUtil.newArrayOf(paras,
ls.size())));
                    }
                    return ret;
                }
            }
            return null;
        }

        @Override
        public Map<K[], V> all(K... paras) {
            if(paras != null && paras.length > 0 && resource != null) {
                K[] reparas = PlatformUtil.newArrayOf(paras, paras.length - 1);
                for(int i =0; i < reparas.length; i++) reparas[i] = paras[i+1];
                Object hm = resource.find(reparas);
                if(Map.class.isInstance(hm)){
                    Map<K[], V> ret = new HashMap<K[], V>(((Map)hm).size());
                    List<K> ls = Arrays.asList(paras);
                    for(Entry<K, V> o : ((Map<K, V>)hm).entrySet()){
                        ls.set(0, o.getKey());
                        ret.put(ls.toArray(PlatformUtil.newArrayOf(paras,
ls.size())), o.getValue());
                    }
                    return ret;
                }
            }
            return null;
        }
}
```

有一个 setBack 存回参数，表示当 map 内的数据已经被修改时，是否要把 map 本身存回到原来的 reosource 中，一般引用式的不需要存回，拷贝式的需要存回。

3.1.5. 用元件组合来实现复杂元件

假设需要一个元件，该元件是一个 resource，作用是从一个目录下的指定 properties 文件里进行 value 读取。其 key 的第一个参数是 properties 文件名，第二个参数是 properties 里的 key 名称。用前面的元件，组装方式如下：

北京漫汇为都科技有限公司（版权所有）

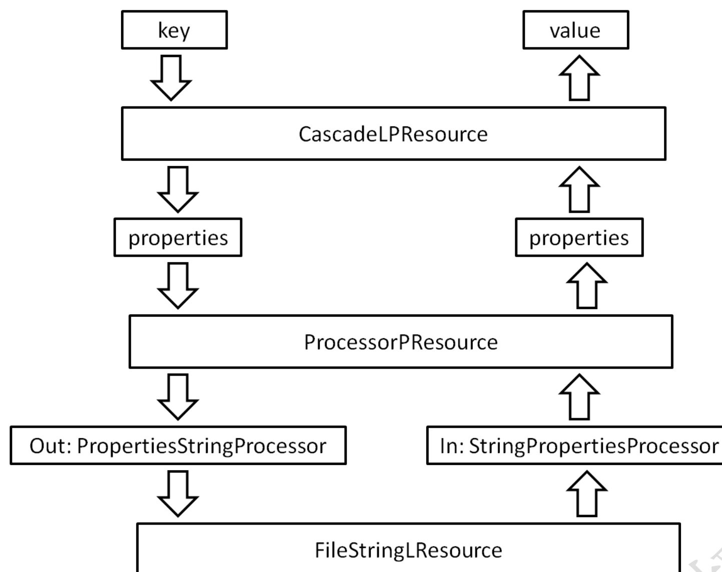


图 3.8 多个元件组装目录下读写 properties 文件的 resource

该元件由 CascadeLPResource、ProcessorPResource、FileStringLResource 以及两个过程 PropertiesStringProcessor 和 StringPropertiesProcessor 组成。

处理流程如下：

1）CascadeLPResource 接受参数，第一个参数是 properties 文件名，第二个参数是 properties 里的 key；

2）CascadeLPResource 将第一个参数传下去，预期返回一个 map；

3）ProcessorPResource 接受第一个参数，将其代理给 FileStringLResource；

4）FileStringLResource 接受第一个参数，从文件目录下读取这个文件并作为 String 返回；

5）ProcessorPResource 的 in 处理器对返回的 String 进行加工，由 StringPropertiesProcessor 将其处理成一个 properties，返回给 CascadeLPResource；

6）CascadeLPResource 接受 properties（实现了 Map 接口），将其作为一个 map，用第二个参数取其值；

7）CascadeLPResource 将 properties 中第二个参数对应的 value 返回给调用方。

同样的，向该 resource 写入数据，也将写入到第一个参数对应的 properties 里的 key-value 中。具体方式可以参考上图实现过程。

3.2. eight 的监控管理

eight 容器里的每一个组件的实例都可以有效的进行监控和管理。主要的方式有通过 jmx 监控管理和通过 iPojo 监控管理两种。这两种方式作用不同，互为补充。

3.2.1. 通过 iPojo 查看系统结构

iPojo 提供了一系列的命令方式，可以用于查看当前运行环境中的组件（component）、实例（instance）以及关联关系。

使用 components 命令可以列出当前所有的已加载组件，加载成功的会显示 VALID。如：

北京漫汇力都科技有限公司（版权所有）

```
components
Factory linker (VALID)
Factory boot (VALID)
Factory center (VALID)
Factory file (VALID)
Factory org.apache.felix.ipojo.webconsole.IPOJOPlugin (VALID)
Factory counter (VALID)
Factory jsp (VALID)
Factory matcher (VALID)
Factory documents (VALID)
Factory formatter (VALID)
Factory org.apache.felix.ipojo.arch.gogo.Arch (UNKNOWN) - Private
```

图 3.9 组件列表

除了 counter、matcher 等几个业务模块外，还有一些系统提供的组件。linker 是连接器，boot、center 和 file 是 eight 用于启动和加载系统的基础组件，其余的则是 iPojo 自带的基础服务。

使用 component <<组件名称>>则可以查看每一个组件的设置。例如 component counter:

factory name="counter" bundle="147" state="valid" implementation-class="net.yeeyaa.eight.osgi.runtime.BundleCenter"

requiredhandlers list="[org.apache.felix.ipojo:requires, org.apache.felix.ipojo:properties, org.apache.felix.ipojo:callback, org.apache.felix.ipojo:provides, org.apache.felix.ipojo:controller, org.apache.felix.ipojo.handlers.jmx:config, org.apache.felix.ipojo:architecture]"

missinghandlers list="[]"

provides specification="net.yeeyaa.eight.osgi.IBundleService"

property name="wait" type="java.lang.Integer" value="null"

property name="config" type="java.lang.String" value="null"

property name="permit" type="java.lang.String" value="null"

property name="path" type="java.lang.String" value="OSGI-INF/blueprint"

property name="pattern" type="java.lang.String" value="*Context.xml"

property name="recurse" type="java.lang.Boolean" value="false"

property name="trace" type="java.lang.Integer" value="null"

property name="clone" type="java.lang.Boolean" value="null"

property name="mode" type="java.lang.Integer" value="null"

property name="thread" type="java.lang.Boolean" value="null"

property name="resource" type="java.lang.String" value="null"

property name="holder" type="java.lang.String" value="null"

property name="log" type="java.lang.String" value="null"

property name="logger" type="java.lang.String" value="null"

property name="context" type="java.lang.String" value="null"

property name="begin" type="java.lang.String" value="begin"

property name="close" type="java.lang.String" value="close"

property name="reload" type="java.lang.Boolean" value="null"

property name="readonly" type="java.lang.Boolean" value="null"

property name="hookid" type="java.lang.String" value="null"

property name="key" type="java.lang.String" value="service.description"

property name="service.description" type="string" value="counter"

inherited interfaces="[net.yeeyaa.eight.osgi.IBundleService, net.yeeyaa.eight.IListableResource, net.yeeyaa.eight.IReadonlyListable, net.yeeyaa.eight.IExtendable, net.yeeyaa.eight.IProcessor,

北京漫汇为都科技有限公司（版权所有）

org.osgi.framework.BundleReference, net.yeeyaa.eight.IInputResource, net.yeeyaa.eight.IListable, net.yeeyaa.eight.IResource, org.osgi.framework.BundleContext, net.yeeyaa.eight.IOutputResource]" superclasses="[]"

这里可以看见 counter 这个组件的配置参数，配置的值以及其实现的接口等。这些参数都是 eight 用于管理组件的。这些参数在生成实例时，可以在 config 文件中设置以覆盖默认值。

instances 命令则用于列出当前运行的实例。如：

```
g! instances
Instance boot_file_store -> valid
Instance org.apache.felix.ipojo.arch.gogo.Arch-0 -> valid
Instance boot -> valid
Instance file -> valid
Instance org.apache.felix.ipojo.webconsole.IPOJOPlugin-0 -> valid
Instance formatter_counter_processor -> valid
Instance counter_documents_documents -> valid
Instance jsp_formatter_calculator -> valid
Instance counter_matcher_processor -> valid
Instance counter -> valid
Instance documents -> valid
Instance jsp -> valid
Instance matcher -> valid
Instance formatter -> valid
```

图 3.10 实例列表

此时，除了各个组件外，linker 也生成了多个实例，用来将各实例连接起来。

对于某个实例，则可以通过 instance <<实例名称>>来查看其运行时参数以及连接状态。如 instance counter：

```
'instance name="counter" state="valid" bundle="147" component.type="counter"
    handler name="org.apache.felix.ipojo:requires" state="valid"
        requires specification="java.util.concurrent.ExecutorService" id="executor" optional="true" aggregate="false" proxy="true" binding-policy="dynamic" state="resolved"
            uses service.id="87"
            selected service.id="87"
            matches service.id="87"
        requires specification="net.yeeyaa.eight.osgi.IBundleProxy" id="proxy" filter="(service.description=counter_*)" optional="true" aggregate="true" proxy="true" binding-policy="dynamic" state="resolved"
            uses service.id="104" instance.name="counter_matcher_processor"
            uses service.id="100" instance.name="counter_documents_documents"
            selected service.id="104" instance.name="counter_matcher_processor"
            selected service.id="100" instance.name="counter_documents_documents"
            matches service.id="104" instance.name="counter_matcher_processor"
            matches service.id="100" instance.name="counter_documents_documents"
    handler name="org.apache.felix.ipojo:properties" state="valid"
        property name="wait" value="UNVALUED"
        property name="config" value="UNVALUED"
        property name="permit" value="UNVALUED"
        property name="path" value="OSGI-INF/blueprint"
        property name="pattern" value="*Context.xml"
```


北京漫汇为都科技有限公司（版权所有）

```
            property name="recurse" value="false"
            property name="trace" value="UNVALUED"
            property name="clone" value="UNVALUED"
            property name="mode" value="UNVALUED"
            property name="thread" value="UNVALUED"
            property name="resource" value="UNVALUED"
            property name="holder" value="beanHolder"
            property name="log" value="log"
            property name="logger" value="log.test|this.one"
            property name="context" value="context"
            property name="begin" value="begin"
            property name="close" value="close"
            property name="reload" value="true"
            property name="readonly" value="UNVALUED"
            property name="hookid" value="counter"
            property name="key" value="service.description"
        handler name="org.apache.felix.ipojo:callback" state="valid"
        handler name="org.apache.felix.ipojo:provides" state="valid"
            provides         specifications="[net.yeeyaa.eight.osgi.IBundleService]"
state="registered" service.id="109"
                property name="service.ranking" value="2100"
                property name="service.description" value="counter"
                property name="instance.name" value="counter"
                property name="factory.name" value="counter"
        handler name="org.apache.felix.ipojo:controller" state="valid"
        handler     name="org.apache.felix.ipojo.handlers.jmx:config"     state="valid"
registered="true"
objectname="net.yeeyaa.eight.osgi.runtime:type=net.yeeyaa.eight.osgi.runtime.BundleCen
ter,instance=counter"
        handler name="org.apache.felix.ipojo:architecture" state="valid"
        object name=net.yeeyaa.eight.osgi.runtime.BundleCenter@6de672a1
```

可见一部分 property 参数与 component 设置并不一致，如 logger 就是 log.test|this.one。uses service.id 部分以及 instance.name 则指出这个实例的引用指向哪个实例。

如果使用 webconsole，还可以从 web 页面上获得这些信息。system/console/iPOJO/factories 可以查看 components，如图：

Factory Name	Bundle
linker	net.yeeyaa.eight.osgi (33)
boot	net.yeeyaa.eight.osgi-boot (34)
center	net.yeeyaa.eight.osgi (33)
file	net.yeeyaa.eight.osgi-file (36)
org.apache.felix.ipojo.webconsole.IPOJOPlugin	org.apache.felix.ipojo.webconsole (90)
counter	net.yeeyaa.eight.osgi-counter (147)
jsp	net.yeeyaa.eight.osgi-jsp (145)
matcher	net.yeeyaa.eight.osgi-matcher (146)
documents	net.yeeyaa.eight.osgi-documents (144)
formatter	net.yeeyaa.eight.osgi-formatter (143)

图 3.11 管理控制台的组件列表及所属的模块（bundle）

同样可以看到每个组件的详情。

北京漫汇为都科技有限公司（版权所有）

Factory Name	counter
State	valid
Bundle	net.yeeyaa.eight.osgi-counter (147)
Provided Service Specifications	net.yeeyaa.eight.osgi.IBundleService
Configuration Properties	(see table below)

Name	Type	Mandatory	Immutable	Default Value
wait	java.lang.Integer	false	false	*No default value*
config	java.lang.String	false	false	*No default value*
permit	java.lang.String	false	false	*No default value*
path	java.lang.String	false	false	OSGI-INF/blueprint
pattern	java.lang.String	false	false	*Context.xml
recurse	java.lang.Boolean	false	false	false
trace	java.lang.Integer	false	false	*No default value*
clone	java.lang.Boolean	false	false	*No default value*
mode	java.lang.Integer	false	false	*No default value*
thread	java.lang.Boolean	false	false	*No default value*
resource	java.lang.String	false	false	*No default value*
holder	java.lang.String	false	false	*No default value*
log	java.lang.String	false	false	*No default value*
logger	java.lang.String	false	false	*No default value*
context	java.lang.String	false	false	*No default value*
begin	java.lang.String	false	false	begin
close	java.lang.String	false	false	close
reload	java.lang.Boolean	false	false	*No default value*
readonly	java.lang.Boolean	false	false	*No default value*
hookid	java.lang.String	false	false	*No default value*
key	java.lang.String	false	false	service.description
service.description	string	false	false	counter

Required Handlers	org.apache.felix.ipojo:requires org.apache.felix.ipojo:properties org.apache.felix.ipojo:callback org.apache.felix.ipojo:provides org.apache.felix.ipojo:controller org.apache.felix.ipojo.handlers.jmx:config org.apache.felix.ipojo:architecture
Missing Handlers	*No missing handlers*
Created Instances	counter

图 3.12 管理控制台的组件详情

管理控制台详细列出组件名称、状态、模块、参数、依赖以及生成的实例。这些信息能有效追踪组件与实例的关系。

system/console/iPOJO/instances 用于查看 isntances。如图：

Instance Name	Factory
boot_file_store	linker
boot	boot
file	file
org.apache.felix.ipojo.webconsole.IPOJOPlugin-0	org.apache.felix.ipojo.webconsole.IPOJOPlugin
org.apache.felix.ipojo.arch.gogo.Arch-0	org.apache.felix.ipojo.arch.gogo.Arch
formatter_counter_processor	linker
counter_documents_documents	linker
jsp_formatter_calculator	linker
counter_matcher_processor	linker
counter	counter
documents	documents
jsp	jsp
matcher	matcher
formatter	formatter

图 3.13 管理控制台的实例与对应的组件

同样可以查看每一个 instance 的当前状态。

Instance Name	counter
State	valid
Factory	counter

Provided Services

Specifications	State	Service Id	Service Properties
net.yeeyaa.eight.osgi.IBundleService	registered	109	service.ranking = 2100 service.description = counter instance.name = counter factory.name = counter

Required Services

Specification	State	Filter	Binding Policy	Optional	Aggregate	Matching Services	Used Services
java.util.concurrent.ExecutorService	resolved	*no filter*	dynamic	true	false	87	87
net.yeeyaa.eight.osgi.IBundleProxy	resolved	(service.description=counter_*)	dynamic	true	true	counter_matcher_processor [104] counter_documents_documents [100]	counter_matcher_processor [104] counter_documents_documents [100]

图 3.14 管理控制台的实例详情

这里列出的信息包括实例名称、状态、所属的组件名称、对外提供的服务以及编号、当前连接上的服务类型及实例。

以上这些均可以通过 restful 调用得到 json 格式的信息。由此，eight 可以通过标准的 http 协议或基于 telnet 的命令行方式，获取当前运行的一切组件和实例的状态信息，并以此构建出当前的系统结构图以及各个节点的配置详情。

北京漫汇为都科技有限公司（版权所有）

3.2.2. 基于 jmx 监控实例

对于系统中运行的实例，不仅仅可以获取其运行状态和关联关系，还可以对每一个实例进行信息采集和控制。eight 平台提供了相关的控制方式，这些控制是通过 jmx 协议提供的。

eight 在打包 bundle 时，会用 BundleCenter 对外提供出 jmx 接口，这些接口可以用来：

1） 观测当前实例的变化；

2） 修改当前实例的配置参数（这与修改 config 有所不同）；

3） 对当前实例进行调试。

对于前面的系统，可以使用 jconsole 来看看其结构状态：

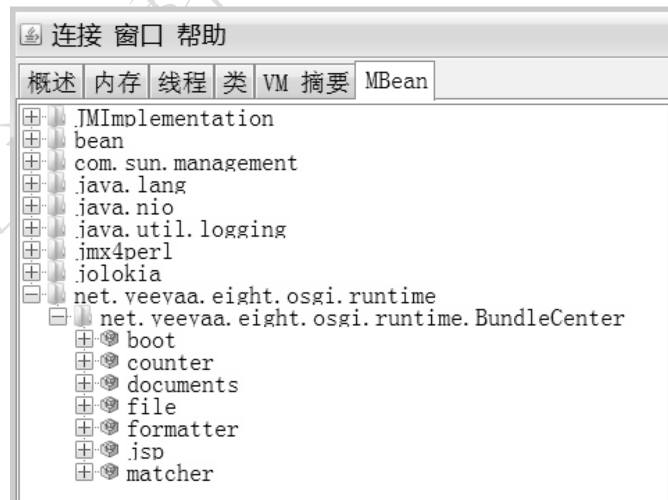


图 3.15 jconsole 中的 MBean 列表

可见在 net.yeeyaa.eight.osgi.runtime.BundleCenter 下的每一个实例均有一个对应的 MBean。可以展开看一个 MBean 的详情：

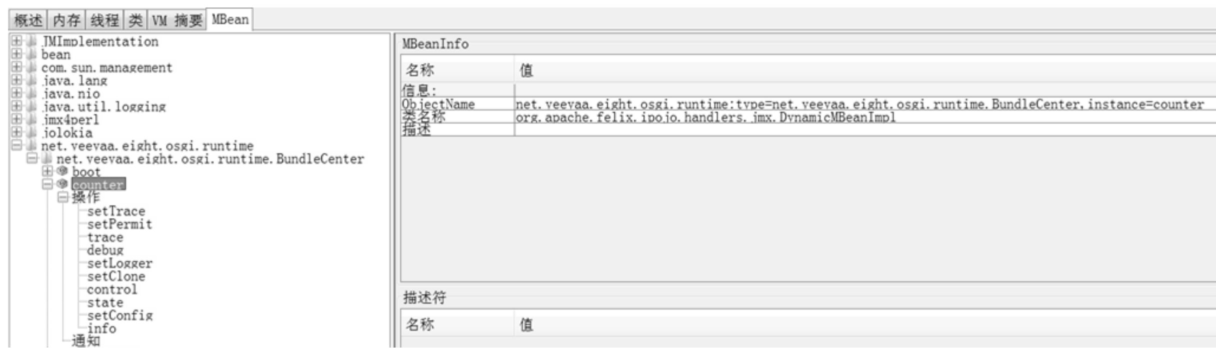


图 3.16 jconsole 中的 MBean 列表

对于每一个实例，都可以在 ObjectName 中看到其唯一位址和类名等信息。而操作里则能看见对该实例能够执行的管理操作。主要涉及到实例的运行调整、信息采集、配置变更等：

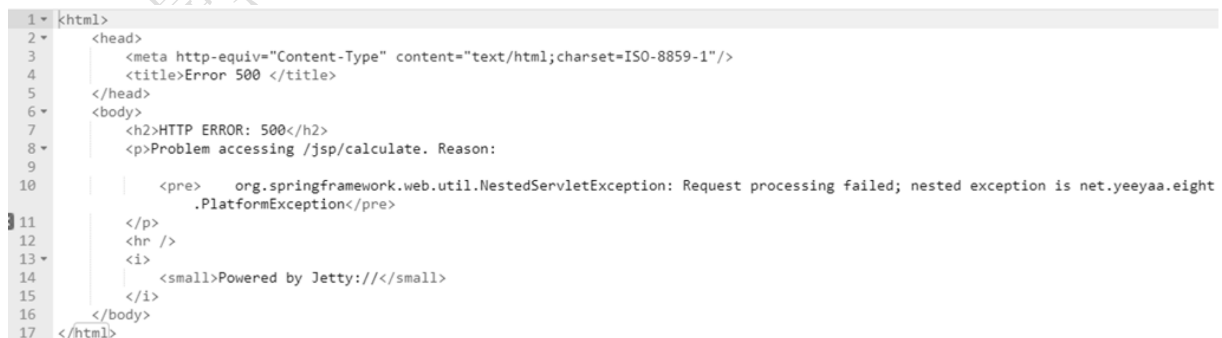
1）setTrace：调整 trace 参数，这是一个整数，用于开关数据缓存，此参数需要与 setClone 以及 trace 配合使用。当为 0 时（默认值），关闭掉当前实例（指当前实例通过共识接口向其它实例提供服务时的输入输出）的缓存。当大于 0 时，设置的是缓存出错信息的容量。当小于 0 时，设置的是对所有调用都进行缓存的容量；

2）setClone：设置的是缓存原始输入输出还是对其进行拷贝，默认（null）不拷贝，true 是拷贝输入输出，false 拷贝输入但不拷贝输出；

3）trace：从当前实例获取缓存下来的结果集（下面的例子是将 setTrace 设置为-5，进行几次调用后，trace 的结果如图）；

图 3.17 jconsole 中进行 trace 观测实例接口调用

4）setPermit：设置白名单列表，当前实例允许哪些内部的 bean 通过共识接口访问，只有白名单内的 bean 及其方法允许访问(下面的例子是设置 counter 不允许访问后，调用出现异常)；



```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;charset=ISO-8859-1"/>
        <title>Error 500 </title>
    </head>
    <body>
        <h2>HTTP ERROR: 500</h2>
        <p>Problem accessing /jsp/calculate. Reason:

            <pre>    org.springframework.web.util.NestedServletException: Request processing failed; nested exception is net.yeeyaa.eight
                .PlatformException</pre>
        </p>
        <hr />
        <i>
            <small>Powered by Jetty://</small>
        </i>
    </body>
</html>
```

图 3.18 禁止实例的共识接口访问

5）debug：调试接口，直接调用实例内的 bean 对应的共识接口，并返回处理结果；

6）setLogger：设置打印日志时的附加信息，由于 eight 是一个松散组合系统，每一行日志从哪个实例输出的必须要进行明确的标识。这些额外信息可以通过 logger 设置注入。默认情

况下 eight 会输出当前实例的 bundle id、所属的模块名称、版本、组件名称和实例名称，如图：

```
2022-01-31_20:55:16.507 ERROR [qtp860174630-44] id[143] name[net.yeeyaa.eight.osgi-formatter] version[1.0.0] component[formatter]  instance[formatter]  - BundleCenter: performing fail
net.yeeyaa.eight.PlatformException: null
	at net.yeeyaa.eight.osgi.runtime.BundleCenter.aaaaaaaaaa(Unknown Source) [eight-osgi-1.0.0.jar:na]
	at net.yeeyaa.eight.osgi.runtime.BundleCenter.service(Unknown Source) [eight-osgi-1.0.0.jar:na]
	at net.yeeyaa.eight.osgi.runtime.BundleLinker.aaaaaaaaaa(Unknown Source) [eight-osgi-1.0.0.jar:na]
```

图 3.19 日志中的实例标识

7）control：用来控制当前实例。有几个控制参数：mode：null 时重启实例，true 为启动，false 为关闭实例。status：null 为挂起当前实例（不再响应调用）直到重启并进入之前的状态，true 是挂起当前实例并重启，始终进入启动状态，false 是不挂起重启；

8）state：获取当前的状态，可查看实例是否 validate 并且已经可以提供服务；

9）info：获取当前实例配置信息。配置信息包括：properties，用于日志和其他一些参数输出；config，对于实例的元件进行配置；paras，osgi 更新时传入实例的参数；permit，设置的白名单列表。这些信息均可以通过该接口获取。

10）setConfig：设置实例的配置参数以修改实例元件的性状。注意该方法只是设置了参数，并没有让参数生效，需要再用 control 重启系统才能生效。这个方式与修改 config 不同的地方在于一个 config 可能被多个 eight 容器在共享使用，修改 config 等同于集群更新，而 setConfig 针对的是单个容器内的某一个实例，只对当前实例有效。（下面的例子将 config 设置为 test.documents.base#docs/，然后使用 control 重启实例，则调用的结果发生了变化）：

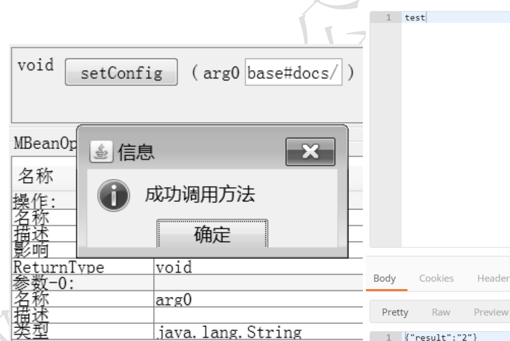


图 3.20 修改 config 并重启后实例发生变化

基于 jmx 的操作可以用 java 的 jmx client 进行，也可以使用 jolokia 来开放 web 接口，通过 restful 方式来监控、设置和管理实例。jolokia 的访问接口在/jolokia 路径下，使用/jolokia/list 可以查看到可以操作的 MBean，如下：

"net.yeeyaa.eight.osgi.runtime":{"instance=counter,type=net.yeeyaa.eight.osgi.runtime.BundleCenter":{"op":{"setTrace":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.Integer\",\"desc\":null}]","ret":"void","desc":null},"setPermit":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.String\",\"desc\":null}]","ret":"void","desc":null},"trace":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.Integer\",\"desc\":null},{\"name\":\"arg1\",\"type\":\"java.lang.Boolean\",\"desc\":null}]","ret":"java.lang.Object","desc":null},"debug":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.String\",\"desc\":null},{\"name\":\"arg1\",\"type\":\"java.lang.String\",\"desc\":null},{\"name\":\"arg2\",\"type\":\"java.lang.String\",\"desc\":null},{\"name\":\"arg3\",\"type\":\"java.lang.Object\",\"desc\":null}]","ret":"java.lang.Object","desc":null},"setLogger":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.String\",\"desc\":null}]","ret":"void","desc":null},"setClone":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.Boolean\",\"desc\":null}]","ret":"void","desc":null},"

control":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.Boolean\",\"desc\":null},{\"name\":\"arg1\",\"type\":\"java.lang.Boolean\",\"desc\":null},{\"name\":\"arg2\",\"type\":\"java.lang.Boolean\",\"desc\":null}]","ret":"java.lang.Boolean","desc":null},"state":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.Boolean\",\"desc\":null},{\"name\":\"arg1\",\"type\":\"java.lang.Boolean\",\"desc\":null}]","ret":"java.lang.Boolean","desc":null},"setConfig":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.String\",\"desc\":null}]","ret":"void","desc":null},"info":{"args":"[{\"name\":\"arg0\",\"type\":\"java.lang.Integer\",\"desc\":null},{\"name\":\"arg1\",\"type\":\"java.lang.Object\",\"desc\":null}]","ret":"java.lang.Object","desc":null}},"class":"org.apache.felix.ipojo.handlers.jmx.DynamicMBeanImpl","desc":null}

然后可以使用/jolokia/exec 来执行相应的操作，协议是：

jolokia/exec/<mbean name>/<operation name>/<arg1>/<arg2>/....。

例如在 documents 上执行 info 查询，则是：

/jolokia/exec/net.yeeyaa.eight.osgi.runtime:type=net.yeeyaa.eight.osgi.runtime.BundleCenter,instance=documents/info/0//

或采用 post 方法访问/jolokia，如：

```
{
"type":"EXEC",
"mbean":"net.yeeyaa.eight.osgi.runtime:type=net.yeeyaa.eight.osgi.runtime.BundleCenter,instance=documents",
"operation":"info",
"arguments":[0,null]
}
```

得到结果：

```
{
   "request": {
          "mbean": "net.yeeyaa.eight.osgi.runtime:instance=documents,type=net.yeeyaa.eight.osgi.runtime.BundleCenter",
          "arguments": [0, null],
          "type": "exec",
          "operation": "info"
   },
   "value": {
          "path": "OSGI-INF\/blueprint",
          "reload": true,
          "log": "log",
          "recurse": false,
          "hookid": "documents",
          "pattern": "*Context.xml",
          "holder": "beanHolder",
          "close": "close",
          "begin": "begin",
          "config": "test.documents.base#scores\/",
          "key": "service.description"
   },
```


北京漫汇为都科技有限公司（版权所有）

"timestamp": 1643641380,
"status": 200
}
可见，在 value 里面，返回了当前实例的各项配置参数。

同样的，可以使用 jolokia 的 restful 接口来设置参数和重启实例，分别是：

/jolokia/exec/net.yeeyaa.eight.osgi.runtime:type=net.yeeyaa.eight.osgi.runtime.BundleCenter,instance=documents/setConfig/test.documents.base#docs/

/jolokia/exec/net.yeeyaa.eight.osgi.runtime:type=net.yeeyaa.eight.osgi.runtime.BundleCenter,instance=documents/control/true///

这样可以达成与之前 jconsole 相同的操作效果（将 documents 扫描目录改到 docs/下并重启实例）。

综上，eight 可以很好的观测系统运行时状态，获取当前系统的结构信息，也能对系统结构中任意实例进行有效的监控、调试、修改和管理。这一切既可以通过协议实现，也可以通过基于 web 的 restful 接口调用来实现。

北京漫汇为都科技有限公司（版权所有）