Front-Business-FrameWork 框架使用说明

(简称: FBF 框架)

日期	修改内容	备注	版本
20140806	修改（FBF 框架 Sample 代码、event-reactor 框架 Sample 代码）		V3
20141211			V4

项目 SVN 地址：

https://svn.58corp.com/ecat/trunk/com.bj58.fbf

1、背景

随着公司的知名度越来越高，访问量越来越大，对系统的性能和可靠性要求越来越高。随着公司的快速发展，新产品不断地被研发，这使得系统越来越多，系统的开发复杂越来越高。同时，伴随着开发人员的快速增加，对开发人员的开发规范也提出了更高的要求。

那么，如何让一般的程序员能够开发出优秀程序员一样高效、安全、稳定的系统？ 如何统一 web 开发人员的开发规范，提高开发效率和代码重用、降低沟通成本？

2、目的

1、提供统一的编程规范和流程。

2、业务分层，明确各层次的功能。

3、提供简单、方便的通用功能 API。

4、微服务，功能服务重用。

3、组成部分

主要用十一部分组成：

1、Action

职责处理器，专门处理某类请求。

2、ActionLocator

职责分发器。

3、ActionCenter

中心职责处理器。

4、DTO

数据传输对象，用于封装请求参数和操作参数。

5、VO

数据视图对象，用于封装操作处理结果数据,展示在页面。

6、Convertor

转换器，转换数据传输对象的属性信息。

7、Validator

验证器，验证数据传输对象和属性信息的正确性。

8、Business Service

操作服务，业务逻辑处理服务。功能简单的微服务。

9、Component

功能组件，业务功能服务。

主要分为两种：

A、SCF Component：

通过对 SCF 服务适配，来提供分布式服务功能和附加功能。所有的 SCF 服务都用 Component 来提供服务。

B、CACHE Component

通过配置，提供简单、方便的缓存操作。包含本地缓存、ehcache 和公司的分布式缓存。

C、EVENT Component

提供简单、方便的异步事件操作 API。是在 event-reactor 框架上进行的封装。

D、Other Component

提供通用或工具操作的类。

10、SCF

公司内部的分布式服务框架。

11、WF

公司内部的前端 MVC 框架。

4、架构图

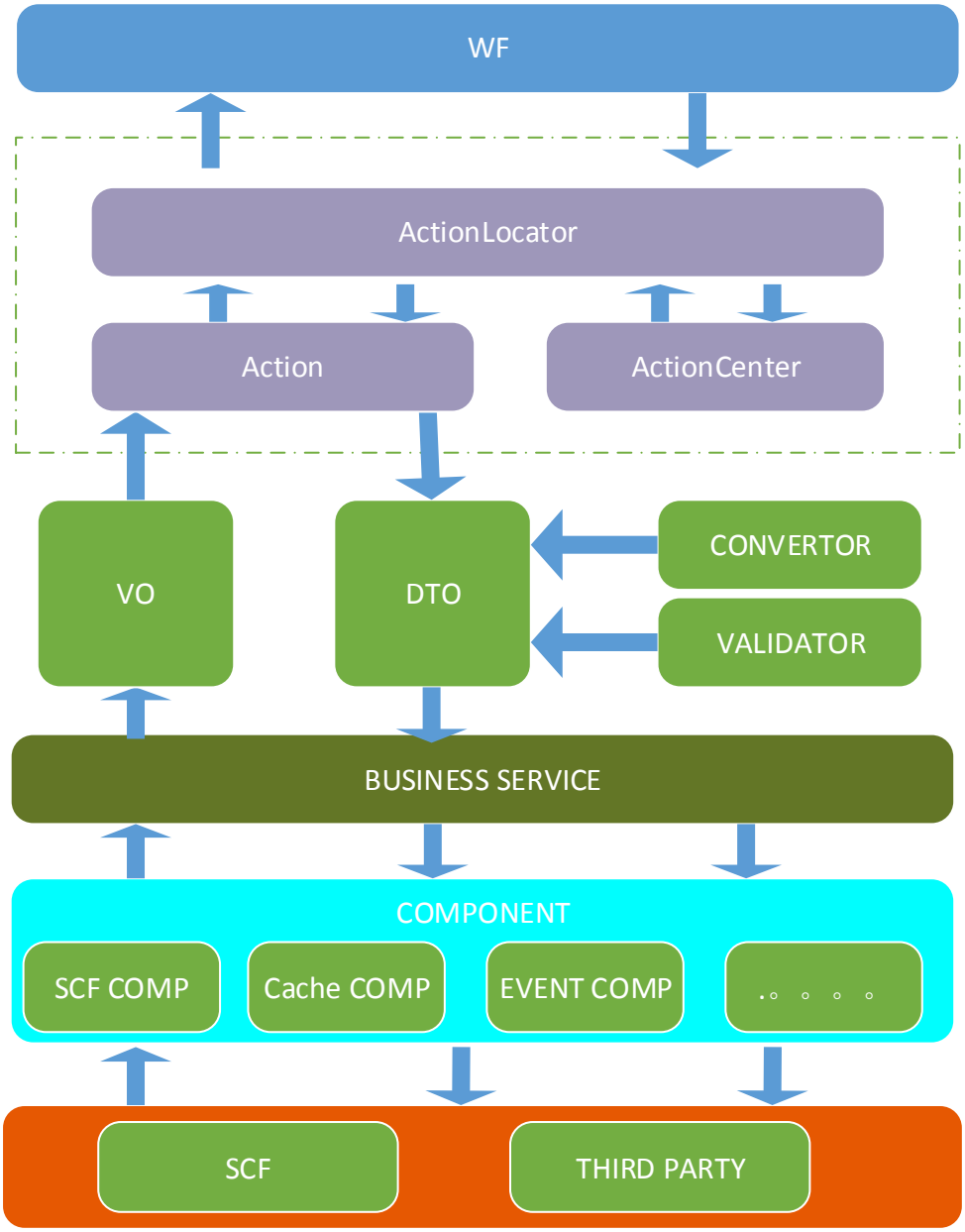

图 1 架构图

图 1 说明

1、WF 用于接收请求，初始化 DTO 和操作服务。同时封装请求参数到 DTO。

2、DTO 根据标注的 convertor 和 validator,对属性信息进行转换和验证。

3、Business Service 接收 DTO，并调用 Component 进行业务逻辑处理。

5、时序图

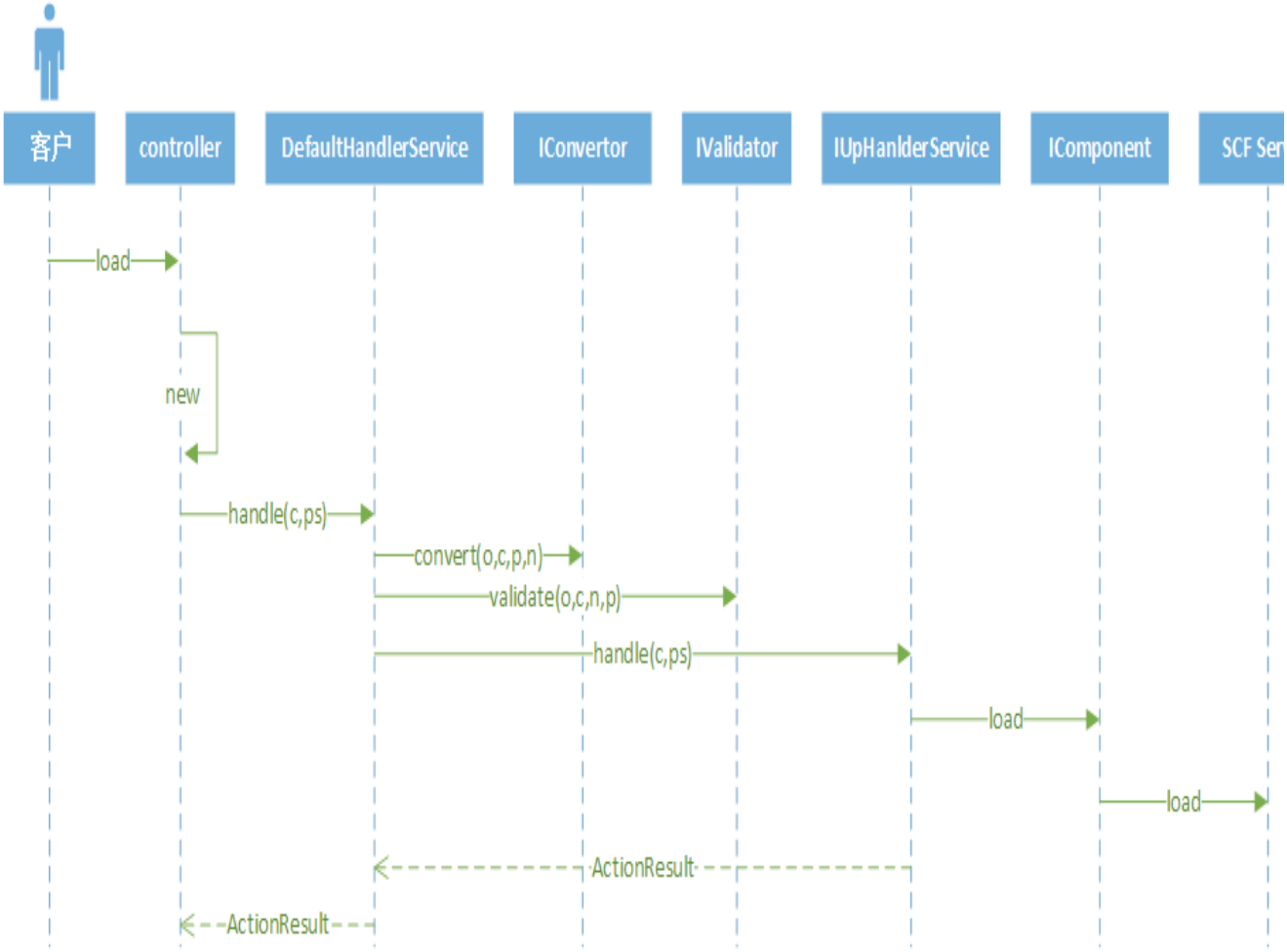

图 2 时序图

图 2 说明

1、controller 接收到客户端的查询订单请求后，初始化订单 DTO 和 DefaultHandlerService 之后，调用 DefaultHandlerService 的 handle 方法处理查询业务操作。

2、DefaultHandlerService 的 handle 方法的查询处理，主要分五步完成查询操作：

第一步、根据 DTO 标注的参数映射关系，自动封装请求参数到 DTO。使用 IConvertor 接口，支持复杂对象的转换。

第二步、根据 DTO 标注的验证器，进行属性和 DTO 的验证。确保参数的合法性、合理性。

第三步、PopulateVoHandlerService 组装 DTO 信息到 VO 里面，没有 VO，默认创建。

第四步、调用 IUpHandlerService 上行流，对业务逻辑进行处理。

第五步、调用 IDownHandlerService 下行流，对上行流处理结果再进行处理。

3、IUpHandlerService 和 IDownHandlerService 调用 SCF Component 。

4、Component 调用 SCF 服务进行数据操作。

6、服务流处理

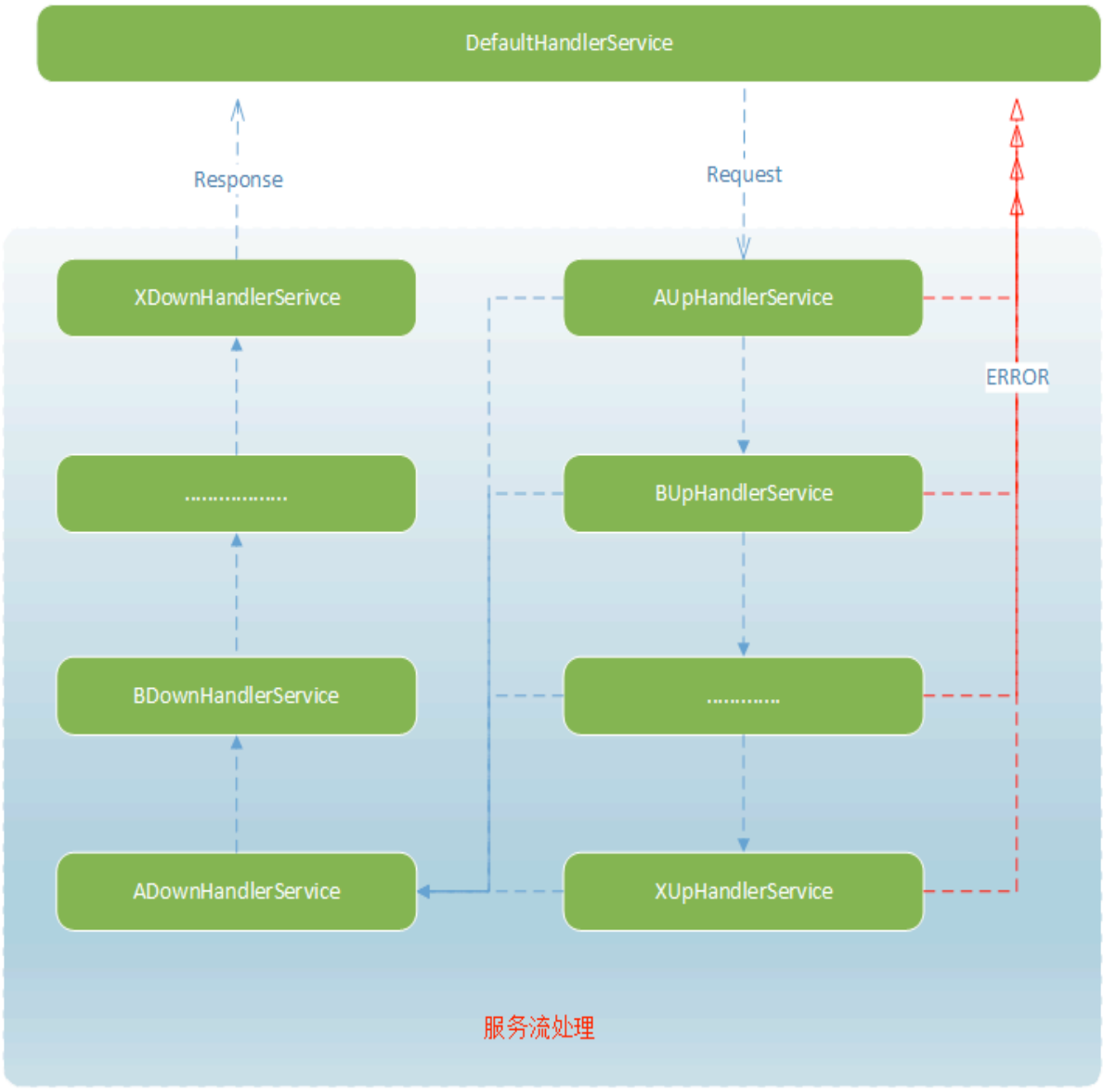

图 3 服务流处理原理图

图 3 说明

1、DefaultHandlerService 注册 IUpHandlerService 上行流服务和 IDownHandlerService 下行流服务。

2、服务流的注册顺序影响服务执行顺序。上行流一定先于下行流执行。

3、任意上行流服务有两种情况可以中断服务流处理：

第一、正常中断，上行流服务返回地址对象(ActionResult)，中断上行流执行，但还会执行下行流。

第二、异常中断，上行流服务出现异常，中断所有的服务流处理,抛出异常。

7、关键类说明

1、ExpandBeantContext.java

WF 框架中上下文环境的适配，用于提供一些方便的 API。

2、IVO.java

数据视图对象的接口，推荐视图对象实现这接口。

3、AbstractDTO.java

数据传输对象的抽象基类，开发中数据 DTO 对象必须继承该类。

4、ParamConvert.java

DTO 属性转换 annotation。

描述请求参数与 DTO 的属性映射对应关系，支持多个参数的映射和对象的转换。

DTO 的属性只有标注了 FieldConvert 才参与自动转换。

5、DTOConverts.java

DTO 转换器 annotation。

对 DTO 的属性进行复杂的转换和绑定。支持多个转换器和动态转换器。

6、AbstractConvertor.java

DTO 属性转换器的抽象基类。

7、AbstractDTOConvertor.java

DTO 对象进行复杂转换的转换器基类。可以配置动态转换器。

8、PopulateConvert.java

对 DTO 的属性进行组装的标注。

9、PopulateVO.java

标注 DTO 属性，需要组装进 VO.

10、Valids.java

DTO 及属性验证器 annotation。用于描述 DTO 及 DTO 属性需要进行验证合法性的验证器类。

DTO 只有标注了 Valids 才参与自动验证。

可以配置动态验证器。

11、AbstractParamValidator.java

DTO 属性的验证器抽象基类。

12、AbstractObjectValidator.java

DTO 对象的验证器抽象基类。

13、DefaultHandlerService.java

默认微服务实现，其他业务逻辑处理服务的入口，提供注册上行流和下行流服务的接口。

14、AbstractHandlerService.java

业务逻辑服务的抽象基类,需要现实handle(context,params)方法处理业务。

15、IUpHandlerService.java

上行流服务接口，需要现实 handle(context,params)方法，推荐继承 AbstractHandlerService.java 类。

16、IDownHandlerService.java

下行流服务接口，需要实现 down(context,action,params)方法。推荐继承 AbstractHandlerService.java 类。

17、IfElseHandlerService.java

这是一个抽象类，用于判断上行流服务的处理路线,如同 java 的条件语句(if else)。

18、FacadeComponent.java

获取 Component 组件门面模式的抽象类，用于注册所有的 Component,所有的 Component 通过这门面类来获取。项目必须实现自己的类，并且通过配置文件 fbfConfig.properties 指定。

19、Proxy.java

动态代理实例化类 annotation。默认为 Cglib 实例化类，也可以根据需要扩展。只能用于类上(final 类不能使用)。

20、Befores.java

方法前置拦截器 annotation。

用于类上。需要和 Proxy 一起使用。

21、Afters.java

方法后置拦截器 annotation。

用于类上。需要和 Proxy 一起使用。

22、Around.java

方法调用前后拦截器 annotation。

只能用于方法上。需要和 Proxy 一起使用。

23、Cache.java

方法调用结果缓存 annotation。

只能用于方法上，需要和 Proxy 一起使用。

24、ClassScanner.java

在指定的包目录下，扫描现在指定接口类的基类。包目录在项目 classpath 中的 scanner.properites 文件中以 key/value 的形式配置，key 为接口类的简单名称，如果 value 有多个，用”,”分割。参见 EventHandlerScanner.java

25、BaseController.java

提供了实例化 DefaultHandlerService 的简便 API。推荐所有 Controller 继承这基类。

26、ICacheProvider.java

统一缓存功能的接口。

目前默认实现了本地缓存(map 和 ehcache)和分布式缓存(公司的 memcache 缓存).

通过在 classpath 里配置 cache-provider.properties 来使用，本地缓存默认推荐使用 map。

27、

8、配置说明

1、框架配置文件说明

a、文件名称：fbfConfig.properties

b、key 说明：

1)scanner.file　配置缓存配置文件名称

2)cacheProvider.file 配置扫描文件名称

3)facadeComponent.call 配置 Component 的门面类

2、缓存配置说明

a、默认文件名称：cache-provider.properties.

b、存放目录:classpath 的根目录里。

c、例如：

```
cache.concurrent2.type = concurrent
cache.concurrent2.size = 2000000
cache.concurrent2.initcapacity=1024
cache.concurrent2.segments=128
Cache.concurrent2.expire=200
Cache.concurrent2.clear=60

cache.ehcache1.type = ehcache
cache.ehcache1.file = ehcache.xml
```

d、说明：

固定格式：cache+.+{名称}+.+{参数名}={值}

1）名称:整个配置中需要唯一,如 concurrent2,ehcache1。

2）参数名：

type：缓存的类型，默认支持 concurrent、ehcache 和 memcache。concurrent 类型的缓存支持多个。其他他类型只支持一个。

size: 缓存的最大个数。

segments: 分片的大小，只支持 concurrent.默认为 128.

expire:最大存活时间,单位:秒。只支持 concurrent

clear:检测间隔时间,单位:秒。只支持 concurrent

file:配置文件名称。支持 concurrent 和 memcache。文件存放在/opt/wf/{namespace}/或 usp 的配置路径下。缓存类别需要明确的配置。

3、扫描配置说明

a、默认文件名称：scanner.properties

b、存放目录:classpath 的根目录里。

c、例如

```
IEventHandler=com.bj58.comment.event,
              com.bj58.framework.event
IAjaxHandler=com.bj58.comment.ajax
```

d、说明

key: 为需要扫描的接口名称。

value: 知道扫描的包，多个用，分割。

9、FBF 框架 Sample 代码

一、场景 1

说明：显示评价信息。

代码实现：

a、 AMethodBefore 方法前置拦截器

```
public class AMethodBefore implements IMethodBefore{
```

```
    public void before(Method method, Object[] params, Object
                    target)        throws Throwable{
            .......;
    }
}
```

b、 BMethodAfter 方法后置拦截器

```
public class BMethodAfter implements IMethodAfter{
    public Object after(Object returnValue, Method method, Object[]
        args, Object target) throws Throwable{
        .......;
    }
}
```

c、 OrderServiceComponent 类

```
@Proxy
public class OrderServiceComp extends
                        AbstractServiceComponent{
    @Befores(befores={AMethodBefore.class})
    @Cache(cacheName="shop_order",provider="memcache1",expire
        =60*120)
    @Afters(afters={BMethodAfter.class})
    public OrderEntity getOrderById(Long id){
        return ...;
    }
}
```

d、 DTO 类

```
@Valids(valids={@Valid(validator=CommentValidator.class,dynamic=
                                                    true)})
```

```
public class CommentDTO extends AbstractDTO{
    @FieldConvert(value={"orderId"}, convert= OrderConvertor.class)
    OrderEntity order;
    ..............

    /***
    *实现动态验证器
    */
    public  Class<? extends IValidator>    getValidatorAdapter
           ( IBeatContext c,Class<? extends IValidator> v){
        .........
        return v;
    }
}
```

e、 订单转换器

```
public class OrderConvertor<O> extends AbstractConvertor
                                        <O,Object,OrderEntiry>{
    @Override
    public OrderEntity convert(O o, BeatContext c, String[] n, Object
                                            p)throws BaseException {
           return orderService.getOrderById((Long)p);
    }

    /**
    *是否需要转换，可以根据业务判断
    */
    public boolean isConvert(O o,IBeatContext c){
        return true;
    }
}
```

f、 评价验证器

```
public class CommentValidator extends
                        AbstractObjectValidator<CommentDTO> {
        @Override
```

```
        public void validate(CommentDTO o, ExpandBeatContext c, Object
            n, Object p)throws ValidateException {
            if(o.order==null) {
                throw new ValidateException("订单不能为空.");
            }
        }
        /**
        *是否需要验证
        */
        public boolean isValidate(CommentDTO  o, ExpandBeatContext
                                                              c){
            return true;
        }
}
```

g、 业务处理服务

```
public class AHandlerService extends AbstractHandlerService {
    @Override
    public ActionResult handle(ExpandBeatContext beat,Object...p)
                        throws Exception {
       CommentDTO  dto =beat.getProtocolDTO();
        ......
        return null;
    }
}
public class BHandlerService extends AbstractHandlerService {
    @Override
    public ActionResult handle(ExpandBeatContext beat,Object...p)
                        throws Exception {
       CommentDTO  dto =beat.getProtocolDTO();
        .....
        return new ActionResult(“...”);
    }
}
```

h、 Controller 接收请求业务逻辑处理代码块。

```
@Path({ "comment/{objectId:\\d+}"})
public ActionResult handlerRequest(Long objectId) {
    ActionResult result = null;
    try {
        ExpandBeatContext beat=newCommentBeantContext();

        result=
```

```
            this.newHandleService()
            .setDtoClass(CommentQueryDTO.class)
            .addUpSingletons(AHandleService.class,
                BHandleService.class)
            .handle(beat,objectId);
        } catch (ValidateException e) {
            return this.toErrorValidActionResult(e);
        } catch (Exception g) {
            return this.toErrorActionResult(g);
        }
        return result;
    }
```

备注：

一、在使用前后置方法拦截和缓存时，当前类需要是代理类(类上加@Proxy).

二、前后置方法拦截也可以用于类上，应用于当前类的所有共用方法上。

三、方法上，还可以使用环绕拦截器。当和缓存同时存在时，先执行缓存。

四、日志使用父类的日志。

二、场景 2

说明:显示评价信息，根据条件执行不同的逻辑处理。

代码实现：

A、AMethodBefore 方法前置拦截 同上

B、BMethodAfter 方法后置拦截器 同上

C、OrderServiceComponent 类 同上

D、DTO 类 同上

E、订单转换器 同上

F、评价验证器　同上

G、业务处理服务　同上

H、分支服务

```
public class CommentIfElseHandlerService extends IfElseHandlerService {
    @Override
    public boolean ifElseHandle(IBeatContext t, Object... extendParams)
                throws Exception {
        If(..)return true;
        return false;
    }
}
```

I、请求业务逻辑处理代码块

```
 @Path({ "comment/{objectId:\\d+}"})
    public ActionResult handlerRequest(Long objectId) {
        ActionResult result = null;
        try {
            ExpandBeatContext beat=newCommentBeantContext();

            //如If Else 代码
            CommentIfElseHandlerService ifelse=new
                CommentIfElseHandlerService(AHandleService.class);
            ifelse.addElseService(BHandleService.class);

            result=
             this.newHandleService()
            .setDtoClass(CommentQueryDTO.class)
            .addUpPrototypes(ifelse)
             .addUpSingletons(....)
            .handle(beat,objectId);
        } catch (ValidateException e) {
            return this.toErrorValidActionResult(e);
        } catch (Exception g) {
            return this.toErrorActionResult(g);
        }
        return result;
    }
```

10、Event-reactor 框架

是一个异步事件框架 ，基于开源框架 disruptor 和 spring-reactor 实现，使用方便、高效。

a、svn 地址

https://svn.58corp.com/ecat/trunk/event_reactor

b、目的

1、提高系统并发性。

2、业务逻辑与信息的解耦。

3、提供统一的、简单的 API。

c、用于场景

1、完成操作的多个服务可以异步并行执行。

2、生产者与消费者模式或观察者模式。

d、拓扑图

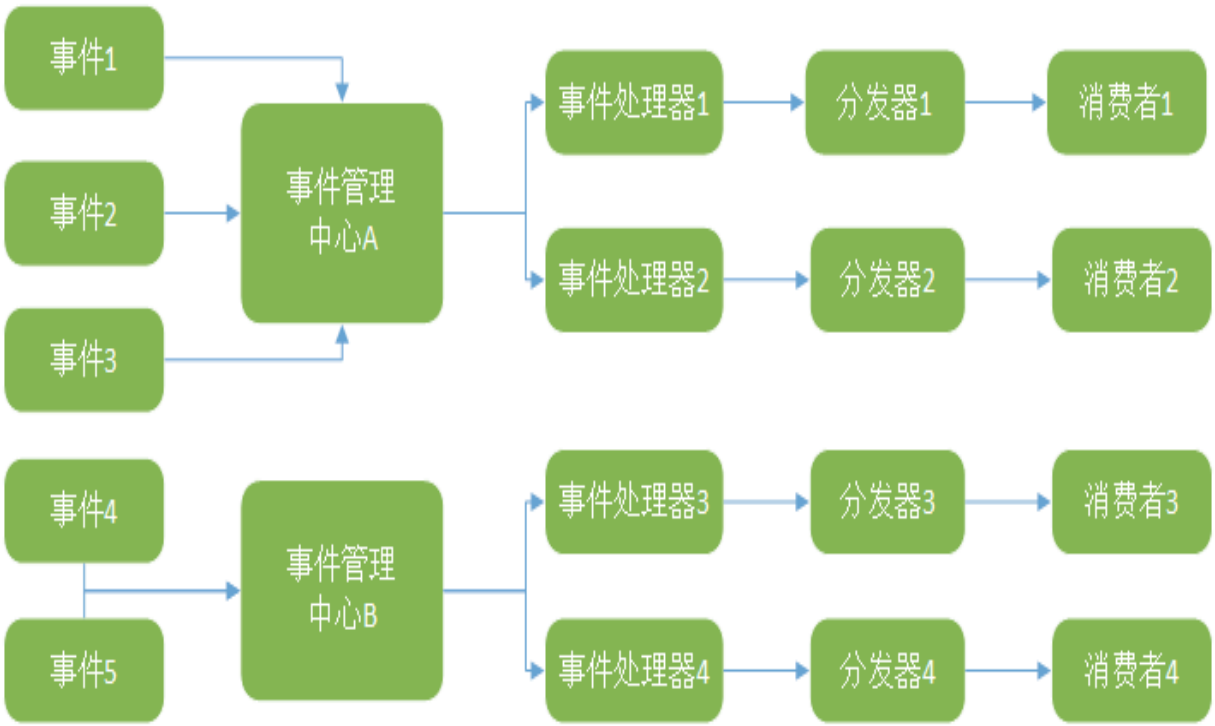

图 4 拓扑图

图 4 说明

1、事件

说明：事件可以理解为用户的操作或操作中的某一个行为。

事件(例如新增用户)触发后，相关信息封装入事件中，然后通知相应的事件管理中心。

2、事件管理中心

说明：为同一组事件提供注册、通知等管理功能。事件管理中心的个数由事件自身来决定。事件管理中心可以根据配置，包含多个事件处理器。

把事件负载均衡到事件处理器 来处理。

3、事件处理器

说明：为事件提供注册、过滤和路由等功能。

4、分发器

把事件转化为任务，并放入任务列队里。

5、消费者

说明：业务逻辑功能处理函数。

处理事件。

e、**结构关系图**

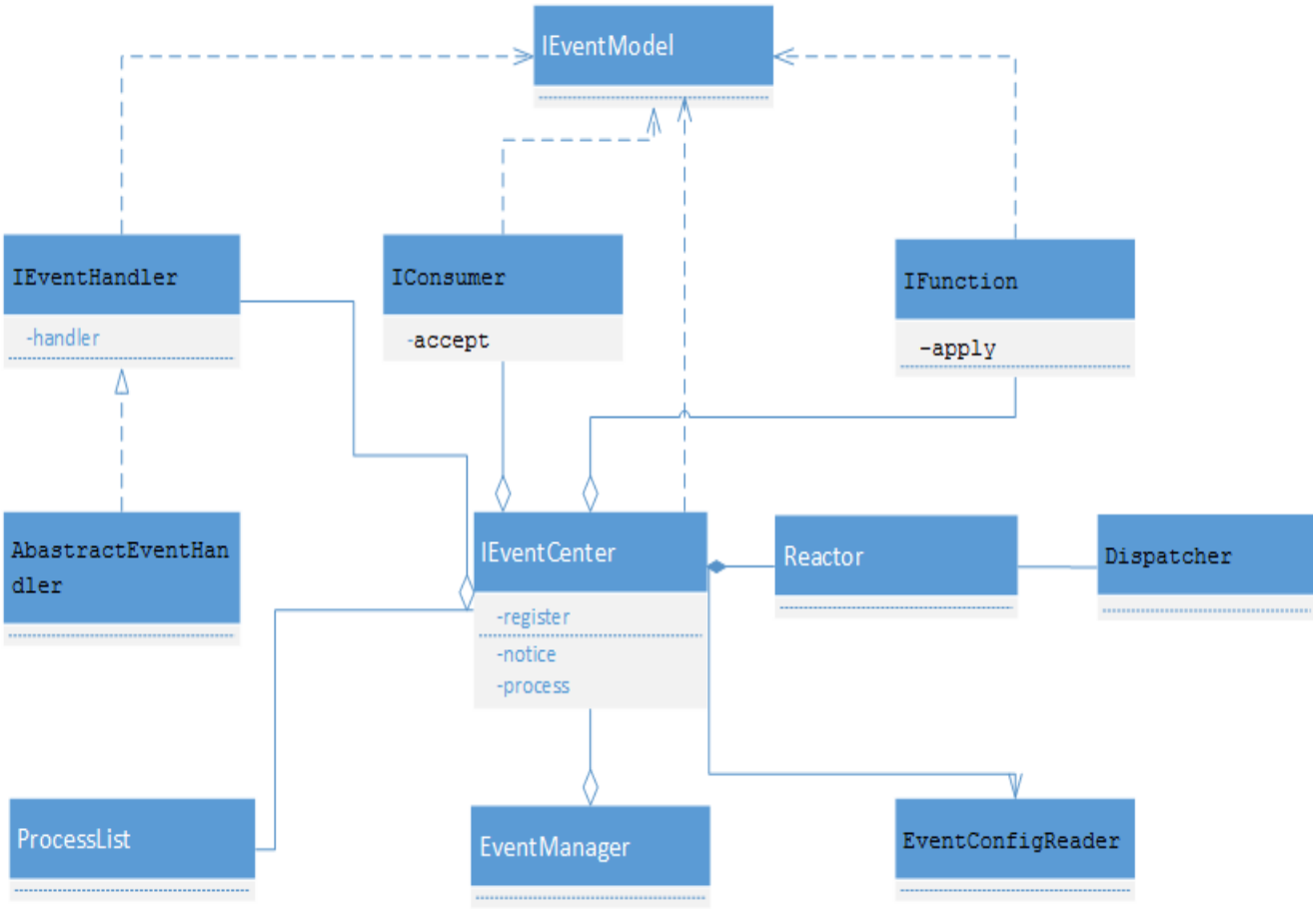

图 5 结构关系图

图 5 说明

1、Reactor 参考 spring-reactor。

2、IEventCenter 事件管理中心。管理同类的事件，提供统一的、简单的注册和通知的处理方法，包含多个 Reactor。在 spring-reactor 上进行了扩展，同类事件可以并行在多个 Reactor 上执行。

3、EventManager 管理所有的 IEventCenter。屏蔽了处理多个 IEventCenter 的复杂性，提供了查找和操作 IEventCenter 的简便方法。

4、EventConfigReader 读取指定文件,初始化 IEventCenter。

5、IEventModel 用于传递数据的事件模型。

6、IEventHandler 通知事件处理接口，需要预先注册。

7、AbstractEventHandler 是扩展实现的类。

8、IConsumer 用于处理事件，不用返回处理结果的业务处理接口。

9、IFunction 用于处理事件，并传递处理结果的业务处理接口。

10、ProcessList 用于异步事件链处理。精简了 spring-reactor 复杂的 API。

f、关键类说明

1、IEventModel

需要在类上加@Reactor,用于标示当前模型通知的 Reactor。

例如:

```
@Reactor
public TestModel implements IEventModel{
}
```

2、IEventHandler

需要在类上加@EventHandler,用于标示当前事件所注册的 Reactor 和处理的事件类型。

例如:

```
@EventHandler(key=“one”)
public EH implements IEventHandler{
```

```
    public Object handler(Object t){}
}
```

3、@Reactor

属性说明：

1)group: 事件类型的分组。

2)dispatchName: 使用 dispatch 的名称，和 group 一起定位 IEventCenter。匹配配置文件里面的值。

3)confile: 配置文件的全路径，目前为相对路径。

4、@EventHandler

属性说明:

1)model: 参见 @Reactor。

2)key: 事件监听的类型。

g、配置文件说明(区分大小写)

1、type

使用 dispatch 的类型，目前支持 ringBuffer ,ringBufferPool,workQueue,eventLoop,threadPoolExecutor

2、size

池的大小，支持 ringBufferPool。

3、queueSize

列队的大小，支持 ringBufferPool。

4、blocklog

ring 的固定大小，支持 ringBuffer,ringBufferPool。

5、parallel

reactor 的并行个数，支持 ringBuffer,ringBufferPool。

h、event-reactor 框架 sample 代码

1、依赖 maven 库配置如下

```
<dependency>
        <groupId>com.bj58.event</groupId>
        <artifactId>com.bj58.event.reactor</artifactId>
        <version>0.0.1-SNAPSHOT</version>
</dependency>
```

2、配置文件内容

```
reactor.dispatchers.ringBufferPool.type = ringBufferPool
reactor.dispatchers.ringBufferPool.size = 10
reactor.dispatchers.ringBufferPool.queueSize = 512
reactor.dispatchers.ringBufferPool.backlog = 512
reactor.dispatchers.ringBufferPool.parallel=4

reactor.dispatchers.default = ringBufferPool
```

3、通知事件 sample 代码(不能获得返回值)

一、场景 1

1)说明：

接收创建通知后，进行创建操作。

2)代码实现：

A、事件模型类

```
@Reactor
public class SModel implements  IEventModel{
    String name;
    public SModel(String name) {
        this.name  = name;
    }
}
```

B、事件处理-创建类

```
@EventHandler(key = "one")
```

```
Public class SEventHandler implements
                        IEventHandler<SModel,String>{
    public String handler(SModel t) {
        System.out.println("model name:"+t.name);
        return t.toString();
    }
}
```

C、预先注册事件处理类的代码块

```
SEventHandler handler=new SEventHandler();
EventManager.getManager().registerEventHandler(handler)
```

D、发送创建通知的代码块

```
EventManager.getManager().noticeEvent( new
              SModel("create"), "one");
```

E、打印输出信息如下

```
model name:create
```

二、场景 2

说明：

接收创建通知后进行创建操作，再根据创建信息进行其他操作

代码实现：

A、事件模型类--同上

B、事件处理--创建类

```
@EventHandler(key = "two")
Public class CreateEventHandler implements
                IEventHandler<SModel,SModel>{
    public SModel handler(SModel t) {
        System.out.println("model name:"+t.name);
        t.name ="one";
        return t;
    }
}
```

C、事件处理-其他操作类

```
@EventHandler(key = "one")
Public class SEventHandler implements
                        IEventHandler<SModel,String>{
    public String handler(SModel t) {
        System.out.println("model name:"+t.name);
        return t.toString();
    }
}
```

D、预先注册事件处理类的代码块

```
SEventHandler handler=new SEventHandler();
EventManager.getManager().registerEventHandler(handler)
;
CreateEventHandler  create=new CreateEventHandler ();
EventManager.getManager().registerEventHandler(create);
```

E、发送创建通知的代码块

```
EventManager.getManager().noticeEvent( new
              SModel("create"), "two", "one");
```

F、打印信息如下

```
model name:create
model name:one
```

4、事件链 sample 代码(可以获得返回值)

一、场景 1

说明:

根据订单 id，创建历史信息，这操作与主线程操作逻辑没有 先后依赖关系。逻辑操作成功执行以后，执行成功操作。

代码实现:

A、事件模型类

```
@Reactor
public class SModel implements  IEventModel{
    public long id;
    public Order order;
 }
```

B、事件处理类--根据定单 id 查询定单处理

```
public class FindOrderFunction implements
                        IFunction<SModel,SModel>{
    public SModel handler(SModel t) {
        System.out.println("findOrderFunction.");
        t.order=orderServer.getOrderById(t.id);
        return t;
    }
    public String getResultKey(){return “”;}
}
```

C、事件处理类--根据定单信息，创建历史信息

```
public class CreateHistoryFunction implements
                        IFunction<SModel,SModel >{
    public SModel handler(SModel t) {
        System.out.println("createHistoryFunction.");
        historyServer.add(t.order);
        return t;
    }
    public String getResultKey(){return “history”;}
}
```

D、事件处理类--成功操作

```
public class SuccessConumer implements
                        IConumser<SModel>{
    public void accept(SModel t) {
        System.out.println("success.");
    }
}
```

E、执行代码块

```
SModel  model=new SModel ();
model.id=2343;
EventManager.getManager().process(model)
.map(new FindOrderFunction())
.map(new CreateHistoryFunction() )
.then(new SuccessConumer()).execute();
```

F、打印信息如下

```
findOrderFunction.
createHistoryFunction.
success.
```

二、场景 2

说明:

根据订单 id，创建历史信息，这操作与主线程操作逻辑没有先后依赖关系。但是，主线程需要等待创建历史信息操作返回的结果。

代码实现:

A、事件模型类--同上

B、事件处理类--根据定单 id 查询定单处理 同上

C、事件处理类--根据定单信息，创建历史信息 同上

D、事件处理类--成功操作 同上

E、执行代码块

两种方式：

方式一：

```
SModel  model=new SModel ();
model.id=2343;
try{
    SModel  m=EventManager.getManager().process(model)
              .map(new FindOrderFunction())
              .map(new CreateHistoryFunction() )
              .then(new SuccessConumer()).execute(6);
}catch(Exception g){
}
```

方式二:

```
SModel  model=new SModel ();
model.id=2343;
try{
   ProcessList process=
         EventManager.getManager().process(model);
            process.map(new FindOrderFunction())
            process.map(new CreateHistoryFunction() )
```

```
        process.then(new SuccessConumer()).execute();
         //........其他处理
       SModel  m=process.get(6);
}catch(Exception g){
}
```

三、场景 3

说明：

根据订单 id，创建历史信息、发送通知、插入日志流水等等。这些操作没有先后，可以并行执行。

代码实现：

A、事件模型类--同上

B、事件处理类--根据定单 id 查询定单处理 同上

C、事件处理类--根据定单信息，创建历史信息

```
public class InsertHistoryConumer implements
                        IConumser<SModel>{
    public void accept(SModel t) {
        System.out.println("insert history.");
    }


}
```

D、事件处理类--根据定单信息，发送通知

```
public class SendSmsConumer implements
                        IConumser<SModel>{
    public void accept(SModel t) {
        System.out.println("send sms.");
    }

}
```

E、事件处理类--根据定单信息，插入日志流水

```
public class InsertLogConumer implements
                        IConumser<SModel>{
```

```
    public void accept(SModel t) {
        System.out.println("insert sms.");
    }
}
```

F、执行代码块

```
SModel  model=new SModel ();
model.id=2343;
try{
ProcessList process=
      EventManager.getManager().process(model);
           process.concurrent(new FindOrderFunction())
               .concurrent(new SendSmsConumer() )
               .concurrent(new InsertLogConumer() )
               .concurrent(new InsertHistoryConumer());
        //........其他处理
        process.execute();
}catch(Exception g){

}
```

四、场景 4

说明：

同上场景 3，但需要取得发送通知和插入日志流水操作的结果。

A、事件模型类--同上

B、事件处理类--根据定单 id 查询定单处理 同上

C、事件处理类--根据定单信息，创建历史

```
public class CreateHistoryFunction implements
                    IFunction<SModel,SModel >{
    public SModel handler(SModel t) {
        System.out.println("createHistoryFunction.");
        historyServer.add(t.order);
        return t;
    }
    public String getResultKey(){return “history”;}
}
```

D、事件处理类--根据定单信息，插入日志流水

```
public class InsertLogFunction implements
                    IFunction<SModel,Boolean>{
    public boolean handler(SModel t) {
        System.out.println("insert log.");
        return true;
    }
    public String getResultKey(){return "log";}
}
```

E、事件处理类--根据定单信息，发送通知

```
public class SendSmsFunction implements
                    IFunction<SModel,Boolean>{
    public boolean handler(SModel t) {
        System.out.println("Sendsms.");
        return true;
    }
    public String getResultKey(){return "sms";}
}
```

F、执行代码块

```
SModel  model=new SModel ();
model.id=2343;
try{
ProcessList process=
    EventManager.getManager().process(model);
        process.concurrent(new FindOrderFunction())
            .concurrent(new SendSmsConumer() )
            .concurrent(new InsertLogConumer() )
            .concurrent(new InsertHistoryConumer());
    //.........其他处理
   Map v= process.execute(50);
    System.out.println(v.get("log"));
    System.out.println(v.get("sms"));
    System.out.println(v.get("history"));


}catch(Exception g){

}
```

五、