**流程引擎逻辑**：

简单理解，用户给流程引擎一个任务列表，引擎按照列表顺序执行。

**Bulbasaur介绍：**

Bulbasaur是Pikachu的java版，作为jar包给用户使用，用户引入jar包，按指定格式自定xml，即可使用引擎完成任务。

**Bulbasaur 大体原理：**

**XML** ---------🡪**引擎解析**---------🡪**执行**---------🡪**结束**

1. 业务方使用，初始化引擎，引擎的核心模块的spring容器会被初始化，并且初始化时候会执行3个动作：
2. 装载所有节点类型
3. 设置解析器，就是解析xml的功能
4. 初始化引擎执行任务的线程池

而如果是持久化模块的初始化，则是该模块的spring容器初始化，里面有设置的数据源等，而且用户可以选择使用持久化的解析器，该解析器会去DB中读取xml模板，并解析。

1. 接下来就是一次流程的初始化部分：

一次用户的执行过程（一次流程实例），我们抽象为一个machine，所以用户要执行一次过程，要初始化一个machine。

一个machine 里面会读取解析一个xml模板文件，并抽象为definition，并解析xml中所有的节点，装载到该definition对象中。该对象中持有一个xml中所有节点的Map。而每个节点也会初始化好自己节点内的其他节点，并装载到本节点。所以definition就是模板。

**一个machine的初始化过程如图：**

**Machine**

NotRunnableState

123

**Definition**

**State**

**Start**

**BIzInfo**

**Event**

**State**

**Event**

**List**<Str>invokes

List<Str> invokes

**List**<Str> pre-invokes

mvelInvokable()

调用父类，持有return的节点

解析节点里面的script文本内容

解析Pojo属性，并由反射，得到要调用的业务方对象。Eg:

test.TestBean -> test，那么解析完，存成(test , testBean@123xx)

一次执行过程，初始化好的Machine，调用Machine的run方法，引擎开始执行。因为在这个machine中，已经有这次的definition对象，而该对象中装载了所有的xml中的节点，引擎开始执行，从start节点开始，然后去遍历该节点里面的paths，这样会得到去下一个节点名字，然后按名字，从process中get到下一个节点的对象，然后继续执行，直到结束。

**一次执行的过 程如图:**

Event

**State**

**State**

**Start**

**Definition**

**Event**

……

**BizInfo**

**Event**

……

……

（definition中的开始的currentState为definition创建时候（构造函数）设置的默认start。）

Start

To AAA

Get from Definition

State

To BBB

……

To BBB

……If Event, also stop

To BBB

State

End

流程引擎的细节功能：

1. Bulbasaur现在分为两个模块，CoreModule（核心执行引擎）和PersistModule（持久化相关）。后续会增加ScheduleModule（调度相关）。
2. 用户在自己的spring中配置bulbasaur的bean。并且用户想使用什么Module，也需要在自己spring中配置并且可以在配置的时候加一些可选或者必选项目。用户只要按需配置，引擎会自己计算加载逻辑。比如：

*<!-- 引擎主bean，必须配置 -->*

<bean id="bulbasaur" class="com.tmall.pokemon.bulbasaur.core.Bulbasaur">  
       *<!——此处按需配置，如果不配置，则默认加载CoreModule-->*  
        <property name="requireModule">  
            <list>  
                <ref bean="persistModule"></ref>  
            </list>  
        </property>  
    </bean>

*<!——核心模块，可以配置线程池参数，如果不配置，引擎还是会自己初始化加载该模块，只不过线程池大小为默认值，10，20-->*

<bean id="coreModule" class="com.tmall.pokemon.bulbasaur.core.CoreModule">  
        <property name="corePoolSize" value="15"/>  
        <property name="maximumPoolSize" value="30"/>  
    </bean>

*<!——持久化模块，如果需要该模块，必须要配置该模块-->*

<bean id="persistModule" class="com.tmall.pokemon.bulbasaur.persist.PersistModule">  
       *<!—用户配置数据源-->*  
        <property name="dataSource" ref="dataSource"/>  
        *<!-- 默认走DB，不走DB 加上下面属性-->*  
        <property name="usePersistParser" value="false/>

</bean>

1. 用户的容器初始化，逻辑初始化到bulbasaur的afterPropertiesSet [ 实现了InitializingBean接口 ]， 会计算出用户配置的需要加载模块的先后顺序。比如PersistModule在初始化前要先初始化CoreModule，那么就会在PersistModule中写好依赖。如:

@Override

**public** Module[] require() {  
        **return** **new** Module[]{CoreModule.getInstance()};  
    }

这样根据下面4的详细计算逻辑，把要加载模块的各个spring的容器一起初始化好 （查找模板的规则是在classpath下的与Module同名的xml文件），并将这个bulbasaur的springContext设置为用户容器的父容器。这样，用户在自己的业务逻辑里面就可以拿bulbasaur里面的bean了（在自己容器中找不到，就会去父容器里面找）。

1. 用户按所需要的Module，配置不同Module的bean，并在Bulbasaur的bean中引用需要的Module，在容器初始化时，引擎会按不同Module的依赖关系，计算出初始化这些Module的顺序，现在的算法是：

*先得到所有要加载的Module以及该模块所需加载的Module放入一个List* dependencies*。然后，按照依赖关系计算一遍，被引用需要加载一次即+1，得到最后的列表，按计数从大到小加载完成。其中有一种可选的加载，即：如果可选的加载Module在必须加载的Module的列表中，那么该Module计算时+1。如果该可选的加载Module不在必选的加载列表中，则忽略不加载*。

*此处给出一个示意图：*

*(示例代码在 ModuleTest)*

依赖

可选依赖

A：3 A：3

B：3 C：1

C：1 D：0

D：0

1. 在初始化Module时候，会将引擎的四种节点类型载入，分别是  Start ，State，Event, BizInfo此四种为引擎的四种属性。(以后可能增加)

* Start：开始节点，只出现一次。里面有向下的路径paths。

<start name="i'm start">

* State：普通节点，有execute方法。该节点是自动执行节点，不停顿。

<state name="state1">

* Event：事件节点，有execute方法和prepare方法。该节点会停顿。

<event name="event1">

* BizInfo：业务节点，支持用户在xml中设置参数，在引擎执行过程中可以用

<bizInfo >

注：*execute对应执行 List invokes ,prepare 对应 List pre-invokes*

1. 引擎中的Machine，是一个流程实例 ，与状态绑定，非单例。用户使用的时候，给定模板的名称，getInstance()得到一个machine。由Place类去控制，引擎会按照给定名称，查找相应的xml文件，并解析，按照之前初始化时候的节点类型，将xml中的所有节点的实例化。这里会new 一个Definition（流程），带有版本号，并将实例化好的所有节点（Start，State，Event）都装入Definition。而definition中的开始的currentState为definition创建时候（构造函数）设置的默认start。
2. 初始化完成，执行machine的run(...)方法，开始执行逻辑。run方法中，首先执行Start节点，执行该节点execute方法（空操作），然后计算下一个节点，解析该节点中的path，计算，得到下一个节点的名字，去之前实例化的Definition中取到，然后执行“下一个节点”（非Start）的prepare方法，返回这个“下一个节点”，并继续执行execute方法，再计算下一个，直到结束。此时引擎全程持有一个上下文Context，在上下文中持有当前执行的Definition的name，节点的name，执行成功后返回结果的name。
3. 此处需要说明：

* Start节点的execute方法和prepare方法都是空操作。
* State节点的prepare方法也是空操作，返回后作为“下一个节点”执行execute方法，解析该State配置文件里面配置的<invoke>节点，并使用Mvel来执行

注意：这种称为自动节点。当业务程序实例执行到这个节点，不会停止执行。而是会继续往下执行。如果该节点存在多个离开转向。那么，就会执行其中的第一个离开转向，在 Node 状态中，不需要外部参与者的参与，业务流程的这个部分是自动的、即时完成的

* Event的prepare方法，会解析该Event配置文件里面配置的<pre-invokes>节点，并使用Mvel来执行，然后，停止。

注意：该节点也叫手工节点，进入到这种节点，整个流程的执行就会中断。直到系统外参与者发起继续执行的命令，即重新执行该节点（重启该流程），业务程序实例的执行才能够继续下

所以，引擎执行到Event节点，会停止，但上下文保持了当前执行到的节点为该Event节点。等引擎再开始执行的时候，直接从该节点执行，即执行该节点的execute方法

1. 非停止的节点，最后会执行到end节点，这个流程就执行完了。不是特殊的end节点，只是普通的state节点，可以没有最后的end节点，因为会进行计算，如果没有path to的节点，就认为流程结束了。

<state name="end"/>

1. 因为引擎中definition是跟状态绑定的（在Place中控制），允许一个xml流程模板被执行多次，因此用版本来区别，之后的同名xml视为高版本并 +1，并且高版本会成为默认版本，也可以配合版本来使用之前的process。在引擎中用“$”做标识。如：processName + "$" + processVersion

**PersistModule介绍：**

1. 该模块提供带版本支持的数据库流程模板获取，扩展自Machine的PersistMachine提供回滚和持久化流程状态及路程的特性。
2. 使用该模块，需要在用户方建三张表，并在persistModule的bean配置一个数据源。

* bulbasaur\_d：流程模板（相当于一个流程xml）
* bulbasaur\_s：节点
* bulbasaur\_p：流程

1. PersistParser扩展自CoreModule的Parser，支持从DB中读取一个流程模板，可调用DefinitionHelper的deployDefinition方法，将流程模板存入DB。读取模板时候，可以按照版本区别读取。
2. 该模块依赖CoreModule，将CoreModule的执行动作，都会存入DB，比如：

* 一个process开始，会存入DB，并按版本区分，状态由ready到complete，并且持有当前执行的节点数据。
* 一个State开始execute之前，会在DB中存该节点，并且状态置为ready，当execute完成，则置状态为complete。计算下一个节点的时候，会将节点持久化到DB，其中，将上下文中有 “\_”开头的变量存入该条记录。

最后附上一个xml模板：

<process name="process">

<!—定义业务信息节点，用于传递业务方的信息参数等，在流程执行的时候可以取出-->

<bizInfo>  
      <info key="r" value="r1" />  
      <info key="r" value="r2" />  
      <info key="r" value="r3" />  
   </bizInfo>

<!—定义一个开始结点，名为 I’am start，指向 state1-->

<start name="i'm start">

<paths>

<path to="state1"/>

</paths>

</start>

<!—state1是一个 state 节点，可以自动执行，不用人工干预处理-->

<state name="state1">

<paths>

<path to="state2" expr="goto==2"/>

<path to="state3" expr="goto==3"/>

</paths>

</state>

<!—state2 是一个 state 节点，可以自动执行，不用人工干预处理-->

<state name="state2">

<invokes>

<script return="a" pojos="test.TestBean -> test, test.TestBean2 -> test2"> test2.testMethod(test.testMethod(i))

</script>

<script return="a" pojos="test.TestBean -> test, test.TestBean2 -> test2" async="true"> test2.testMethod(test.testMethod(2))

</script>

<script return="a" pojos="test.TestBean -> test, test.TestBean2 -> test2" async="true"> test2.testMethod(test.testMethod(4))

</script>

</invokes>

<paths>

<path to="state4"/>

</paths>

</state>

<state name="state3">

<paths>

<path to="end"/>

</paths>

</state>

<!—state4 是一个 event 类型的节点，也就是前面提到的手工节点-->

<event name="state4">

<pre-invokes>

<script return="info">

"some info"

</script>

</pre-invokes>

<paths>

<path to="end"/>

</paths>

</event>

<state name="end"/>

</process>

schedule模块：

主要提供了自动重试，在state上面配置了repeatList，失败可以自动重试。ScheduleMachine重写了persistMachine的auto\_run方法。使用tbschedule，每隔30s扫描一次job表，符合条件的拿出来，继续run。在select出来的job，检查job的有效性（该 bizId + stateName 在state表中，状态是否已经变成了 complete，如果是，删除该job）

task节点: