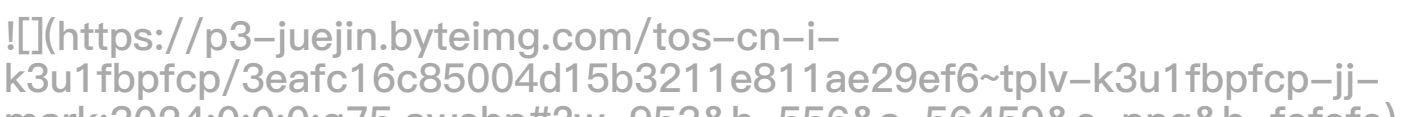
安利一个轻量级流程引擎compileflow

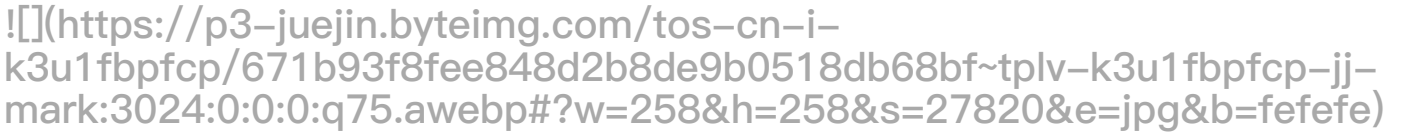
写在文章开头

今日推荐一个比较轻量级的工作流引擎——即阿里的`compileflow`，这款流程引擎算是笔者接触过流程引擎中相对轻量级、且性能和集成扩展表现都比较良好的框架，本文就会从几种常见的使用场景以及源码分析的角度介绍这款工具。



Hi，我是 **sharkChili** ，是个不断在硬核技术上作死的 **java coder** ，是 **CSDN的博客专家** ，也是开源项目 **Java Guide** 的维护者之一，熟悉 **Java** 也会一点 **Go** ，偶尔也会在 **C源码** 边缘徘徊。写过很多有意思的技术博客，也还在研究并输出技术的路上，希望我的文章对你有帮助，非常欢迎你我的公众号： **写代码的SharkChili** 。

因为近期收到很多读者的私信，所以也专门创建了一个交流群，感兴趣的读者可以通过上方的公众号获取笔者的联系方式完成好友添加，点击备注 **“加群”** 即可和笔者和笔者的朋友们进行深入交流。



为什么需要 compileflow

经评审后得出明确复杂系统的实现思路，由此绘制出流程图，但真正落地时因为每个开发不同的习惯总会导致实现会有所偏差，使得系统流程串联时会出现各种各样的问题，通过流程引擎界定业务边界后并约定开发规范，以及业务逻辑的可视化，大大降低了业务设计和开发的成本。

相比主流的几种流程引擎，`compileflow`有着如下几个优势：

1. 高性能：流程文件会直接转换成`Java`代码并编译执行，简洁且高效。
2. 集成方便：引入`compileflow`无需繁琐的配置，只需引入类库后在开发的维度完成配置即可。
3. 完善的插件：`compileflow`为`IDEA`提供的插件，在进行流程设计时可基于快速完成流程图的设计，并基于流程图的节点边界引入我们的业务代码。

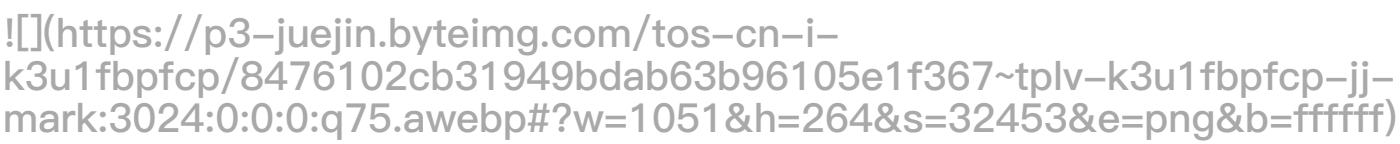
compileflow最佳实践案例

前置准备

在正式使用`compileflow`之前，我们需要在IDEA中安装一个compileflow的插件，方便我们后续的绘图工作，对应的地址如下，按需下载对应版本安装重启即可：

[compileflow-designer-upgrade](http://cxyroad.com/”https://github.com/compileflow/compileflow-designer-upgrade”)

以笔者为例，个人开发工具是`IDEA 2018`版本，所以下载的版本就是`compileflow-idea-designer-1.0.14.for.2018.up.zip`：



开平方计算

接下来笔者回基于3个例子介绍一下`compileflow`几种比较常见的使用，先来一个比较基础的平方根计算，流程比较简单，传入一个参数后，给出对应的开平方结果。

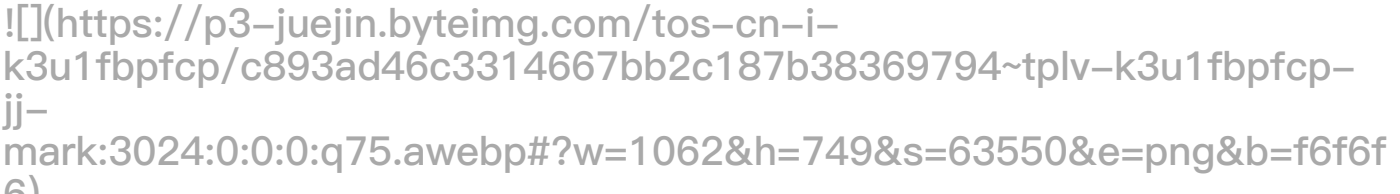
基于插件，我们在`resources`目录下创建一个名为`sqrt.bpm`的文件，通过开始、结束、自动设置3个标签完成了一张简单的开平方流程图绘制：

k3u1fbpfcp/ce45c9a4093549c5a4be2afa966097c1~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1176&h=551&s=39412&e=png&b=fcfcfc)

按照我们的设计，我们要求在求平方根这个自动节点传入一个参数`num`，让这个`num`走到我们的平方计算逻辑，基于这个约定，我们完成平方计算器的代码：

```
@Slf4j
public class SqrtCalculator {

    public double sqrt(double num) {
        double result = Math.sqrt(num);
        return result;
    }
}
```

有个这个`Java`类之后，我们就可以将这个类和图片中的自动节点绑定，如下图所示，我们双击节点后配置类的包路径以及出入参的映射即可：



自此我们的流程图就绘制并开发完成了，在`Spring`中，我们只需通过`afterPropertiesSet`这个扩展点，将该图编译转换成字节码，后续使用时就会通过反射创建并缓存起来等待用户的调用和运行：

```
@Component
@Configuration
public class BpmInitializer implements InitializingBean, ApplicationContextAware {
    @Override
    public void afterPropertiesSet() throws Exception {
        ProcessEngine processEngine = ProcessEngineFactory.getProcess
```

```
Engine();
        //编译开平方根流程图生成字节码
        processEngine.preCompile("sqrt");

    }

    @Override
    public void setApplicationContext(ApplicationContext applicationCont
ext) throws BeansException {
        SpringApplicationContextProvider.applicationContext = application
Context;
    }
}
```

为了方便测试我们写了一段`HTTP`的调用:

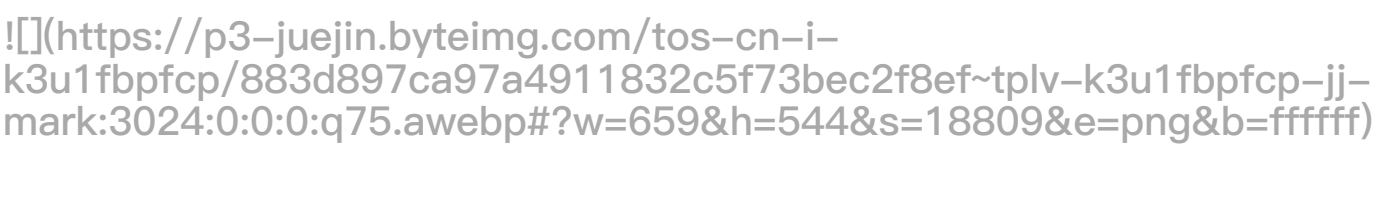
```
@RequestMapping("/sqrt")
    public double sqrt(@RequestParam Double num) {
        //code在bpm文件中定义
        String code = "sqrt";

        //执行流程的入参
        Map<String, Object> context = new HashMap<>();
        context.put("num", num);

        try {
            //从ProcessEngineFactory获取流程引擎
            ProcessEngine processEngine = ProcessEngineFactory.getProce
ssEngine();
            //拿到我们的开平方的代码并传入参数获取结果
            Map<String, Object> result = processEngine.execute(code, cont
ext);
            log.info("sqrt result:{}", result.get("result"));
        } catch (Exception e) {
            log.error("执行报错，报错原因:{}", e.getMessage(), e);
        }
        return 0;
    }
```

我们传入参数4对应的返回结果如下:

```
 INFO 14676 --- [io-18080-exec-
4] c.s.controller.CalculatorController       : sqrt result:2.0
```

两数相加

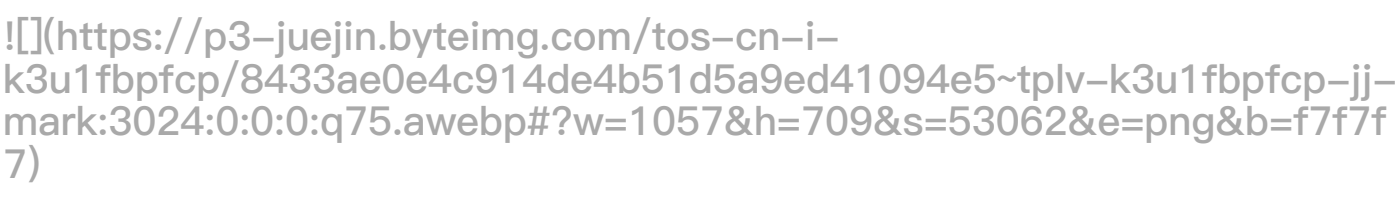
我们再拓展一个例子，要求传入两个参数`num1`、`num2`并获取这两个参数的结果`result`，同理我们快速绘制出这样一张流程图:



基于该图的约定，我们给出两数相加的代码:

```
@Slf4j
public class SumCalculator {

    public int sum(int num1, int num2) {
        int result = num1 + num2;
        log.info("{}+{}={}", num1, num2, result);
        return result;
    }
}
```

最后我们将代码和图片中的节点绑定:



同理在配置中完成该流程图的预编译:

```
 processEngine.preCompile(”sum”);
```

后续的调用也就和上文一样，这里我们也给出对应的代码:

```
@RequestMapping(”/sum”)
    public int sqrt(@RequestParam Integer num1,Integer num2) {
        //code在bpm文件中定义
        String code = ”sum”;

        //执行流程的入参
        Map<String, Object> context = new HashMap<>();
        context.put(”num1”, num1);
        context.put(”num2”, num2);

        try {
            ProcessEngine processEngine = ProcessEngineFactory.getProce
ssEngine();
            Map<String, Object> result = processEngine.execute(code, cont
ext);
            log.info(”sum result:{}”, result.get(”result”));
        } catch (Exception e) {
            log.error(”执行报错，报错原因:{}”, e.getMessage(), e);
        }
        return 0;
    }
```

团队点餐

最后我们再来一个复杂的流程图，我们会给定人数进行点餐，最后根据人数则算点餐金额，如果餐费大于100则减去10元，反之按照原价完成付款结束整个流程。 对此我们给出对应的流程图，可以看到我们在循环节点中添加了吃饭的自动节点，该节点会遍历入参并调用吃饭这个自动节点。完成吃饭的流程后通过判断节点计算金额，如果大于`100`走左边的脚本节点，反之走右边，通过脚本节点得到计算结果后，执行付款节点即可：

k3u1fbpfcp/67eccc45b22a4f3fae9f7c67d61d592c~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=498&h=699&s=11108&e=png&b=ffffff)

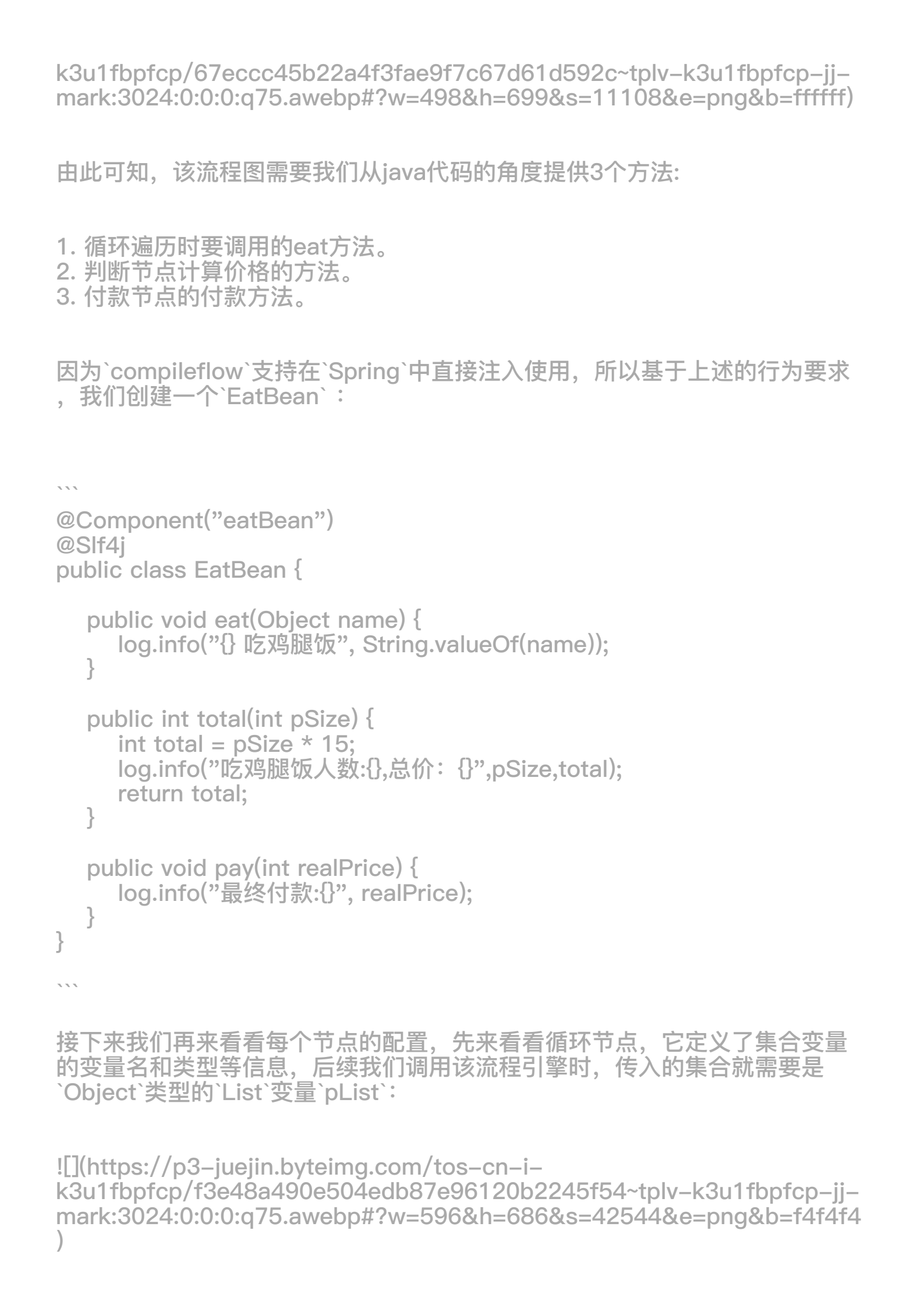
由此可知，该流程图需要我们从java代码的角度提供3个方法:

1. 循环遍历时要调用的eat方法。
2. 判断节点计算价格的方法。
3. 付款节点的付款方法。

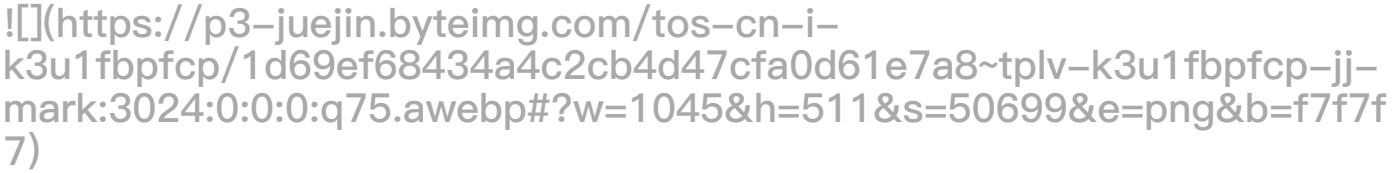
因为`compileflow`支持在`Spring`中直接注入使用，所以基于上述的行为要求，我们创建一个`EatBean`：

```
@Component(”eatBean”)
@Slf4j
public class EatBean {

    public void eat(Object name) {
        log.info(”{} 吃鸡腿饭”, String.valueOf(name));
    }

    public int total(int pSize) {
        int total = pSize * 15;
        log.info(”吃鸡腿饭人数:{},总价：{}”,pSize,total);
        return total;
    }

    public void pay(int realPrice) {
        log.info(”最终付款:{}”, realPrice);
    }
}
```

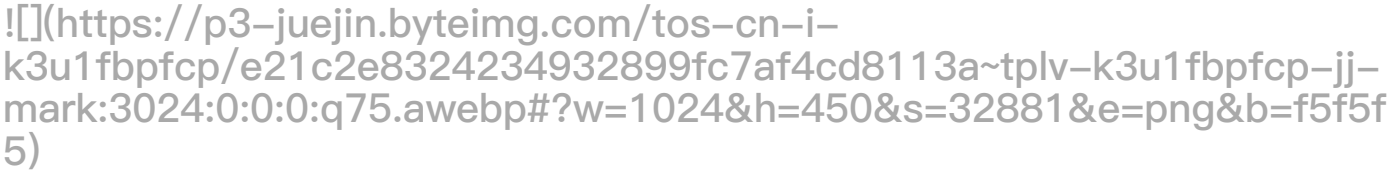
接下来我们再来看看每个节点的配置，先来看看循环节点，它定义了集合变量的变量名和类型等信息，后续我们调用该流程引擎时，传入的集合就需要是`Object`类型的`List`变量`pList`：



循环遍历的每一个节点都会调用吃饭节点，所以我们的吃饭节点就和`eat`方法绑定:



同理判断分支绑定`pList`的`size`，将返回值`total`和`return`的变量绑定：



判断分支基于计算方法得到结果后，就会走到对应的脚本节点，我们以左边减去100的为例，可以看到它会基于判断节点返回的`total`通过ql表达式将其减去`10`作为最终结果`realPrice`：

最后付款节点基于脚本节点得到的`realPrice`绑定`pay`方法付款：

最终我们给出测试代码：

```
@RequestMapping(”/eat”)
    public int eat() {
```

```
        //code在bpm文件中定义
        String code = "eat";

        //执行流程的入参
        Map<String, Object> context = new HashMap<>();
        List pList=new ArrayList();
        pList.add("user1");
        pList.add("user2");
        pList.add("user3");
        pList.add("user4");

        context.put("pList", pList);

        try {
         //从流程引擎中获取eat的代码并执行
            ProcessEngine processEngine = ProcessEngineFactory.getProce
ssEngine();
            Map<String, Object> result = processEngine.execute(code, cont
ext);
            //输出最终付款结果
            log.info("eat result:{}", result.get("realPrice"));
        } catch (Exception e) {
            log.error("执行报错，报错原因:{}", e.getMessage(), e);
        }
        return 0;
    }

```

4个人一人15，所以餐费不扣减最终付款60：

```
024-05-14 08:57:35.982  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.type.EatBean             : user1 吃鸡腿饭
2024-05-14 08:57:35.982  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.type.EatBean             : user2 吃鸡腿饭
2024-05-14 08:57:35.982  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.type.EatBean             : user3 吃鸡腿饭
2024-05-14 08:57:35.982  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.type.EatBean             : user4 吃鸡腿饭
2024-05-14 08:57:35.982  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.type.EatBean             : 吃鸡腿饭人数:4,总价：60
2024-05-14 08:57:35.998  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.type.EatBean             : 最终付款:60
2024-05-14 08:57:35.998  INFO 15416 --- [io-18080-exec-
2] com.sharkChili.controller.EatController  : eat result:60
```

```

基于源码详解compileflow工作流程
--------------------

我们直接以两数相加以为例，查看`processEngine`的预编译`preCompile`方法，其内部会基于我们的流程图将其转换为class文件并将这些内容缓存起来。

```
 ProcessEngine processEngine = ProcessEngineFactory.getProcessEngine();
 processEngine.preCompile(”sum”);
```

可以看到其内部最终会走到`AbstractProcessEngine`的`preCompile`方法，它会判断以当前`code`作为`key`查看缓存`runtimeCache`中是否有这个流程图的编译后的字节码，如果有则直接返回，反之通过当前流程图各种变量等信息生成`Java`代码并将其编译编译为字节码文件并将其缓存到`compiledClassCache`中，最后再通过`runtime`的`compile`方法用类加载器将我们编译后的字节码文件的类加载到内存中等待使用：

```
    @Override
    public void preCompile(ClassLoader classLoader, String... codes) {
        if (ArrayUtils.isEmpty(codes)) {
            throw new CompileFlowException(”No process to compile”);
        }

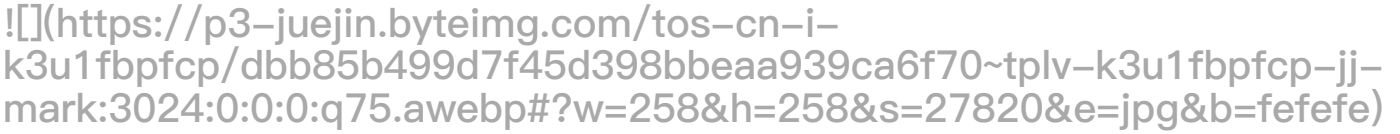
        for (String code : codes) {
            //获取运行时流程引擎，如果runtimeCache中存在则直接得到runtime ，反之基于code的信息得到流程图将其转换为Java代码并编译成字节码缓存到compiledClassCache中，再将其存入runtimeCache中并返回
            AbstractProcessRuntime runtime = getProcessRuntime(code);
            //传入类加载器，基于上文编译后的字节码文件将类加载到内存中
            runtime.compile(classLoader);
        }
    }
```

执行时我们会通过`processEngine.execute(code, context);`方法进行调用
```

，其内部逻辑就是通过我们上文的`runtimeCache`找到运行时运行时信息，基于该信息的`code`也就我们两数相加的名字`sum`调用`runtime.start`方法，其内部会从编译缓存中找到我们的字节码文件，完成反射创建并调用，最后将结果返回：

```
public Map<String, Object> execute(String code, Map<String, Object> co
ntext) {
  //基于code获取sum的运行时信息
        TbbpmProcessRuntime runtime = (TbbpmProcessRuntime)this.getP
rocessRuntime(code);
        //通过上一步得到的runtime 调用start方法，其内部会拿着runtime的
code从编译缓存compiledClassCache中拿到字节码文件完成SumFlow的反射
创建并调用
        return runtime.start(context);
    }
```

我们查看`runtime`的`start`方法就可以看到从编译缓存中拿到对应的字节码完成反射创建和调用的操作：

```
public Map<String, Object> start(Map<String, Object> context) {
  //......
        //从compiledClassCache中拿到code对应的字节码完成反射创建集合
参数context完成调用
        return this.executeProcessInstance(context);
    }

private Map<String, Object> executeProcessInstance(Map<String, Object
> context) {
        try {
         //从compiledClassCache拿到字节码完成反射创建的示例
            ProcessInstance instance = getProcessInstance();
            //基于这个示例完成调用并返回结果
            return instance.execute(context);
        } catch (CompileFlowException e) {
            throw e;
        } catch (Exception e) {
            throw new CompileFlowException("Failed to execute process, c
ode is " + code, e);
        }
    }
```

```

## 小结

本文通过3个案例并结合源码分析的形式剖析了轻量级流程引擎compileflow的工作机制，希望对你有帮助。

我是 **sharkchili** ，**CSDN Java 领域博客专家**，**开源项目—JavaGuide contributor**，我想写一些有意思的东西，希望对你有帮助，如果你想实时收到我写的硬核的文章也欢迎你我的公众号： **写代码的SharkChili** 。 因为近期收到很多读者的私信，所以也专门创建了一个交流群，感兴趣的读者可以通过上方的公众号获取笔者的联系方式完成好友添加，点击备注 **“加群”** 即可和笔者和笔者的朋友们进行深入交流。



## 参考

协议详解：[github.com/alibaba/com...](http://cxyroad.com/”https://github.com/alibaba/compileflow/wiki/%E5%8D%8F%E8%AE%AE%E8%AF%A6%E8%A7%A3#2-%E5%85%A8%E5%B1%80%E5%8F%98%E9%87%8F”)

compileflow官方文档:[github.com/alibaba/com...](http://cxyroad.com/”https://github.com/alibaba/compileflow”)

本文使用 [markdown.com.cn](http://cxyroad.com/”https://markdown.com.cn”) 排版
```