OSPP 2022

项目申请书

项目编号：222290460

DolphinScheduler Task Plugin 扩展

增加 Java 任务类型

申请人：曹凯勋

Github ID：106umao

邮箱：578961953@qq.com

目录

1.介绍

A.个人简介

一名软件工程专业大三年级学生，热爱生活，热爱开源，积极参与各种开源社区交流，日常生活中喜欢捣鼓一些音乐软件和第三方游戏 mod。目前是 seata 项目组下 seata-go 成员之一，正在着手 seata-go 中 TCC 模式 RM 模块的资源注册，分支事务注册等功能的开发设计工作。此外还参与贡献过 apache/dolphinscheduler，apache/dubbo，apache/dubbogo，seata 和 botania 等开源项目。截至申请书提交时，共在 dolphinscheduler 中提交了 pr8 次，issue 9 次，此外积极与社区成员交流学习，对社区有一定的了解。

B.关于 Java Task Plugin

提供 Java 类型的调度任务是我本次申请的课题内容，通过与导师的沟通和社区的交流我已经比较完整掌握了本次任务的要求，大致上可以将其分为两个部分进行描述：

- 第一部分为后端模块，通过实现系统定义的任务类型接口，提供一组调用 Java 程序的方法，该类型需要支持程序的两种运行方式，即使用 java classname 命令通过用户输入的 java 源代码文件作为程序入口进行调用和通过 java -jar mainjar.jar 的方式使用 jar 文件中定义的程序入口进行程序调用。
- 第二部分为前端模块，通过在项目管理→任务→任务定义面板中绘制两种定义 Java 任务类型的 UI 界面，其需要支持手动编写 Java 代码，定义程序参数，定义虚拟机参数等，并且还需要能够通过 resource 来引入依赖 jar 包。

C.关于用到的技术

在与导师沟通且深入了解 DolphinScheduler 这个项目后，我对本次任务所需要使用到的技术有了比较完整地把握。其中大部分的技术我都比较熟悉，但是也存在一些不太熟悉的技术，不过在了解自身情况之后我对一些技术中不熟悉的地方了查漏补缺：

- Java：此次在项目中有需要用到 Java 的技术分为两部分，一是因为 DolphinScheduler 由 Java 开发，所以在编写任务类型接口代码时需要使用到 Java 知识，在这部分需要使用到的知识主要有 Java 基础，面向对象基础，SPI 机制，Shell 脚本调用机制等。二是本次开发的 Java 任务类型插件是指用户通过前端 UI 提供 Java 程序到达后端，后端通过一些机制来调用这些 Java 代码，故在此部分需要用到 javac 命令对用户提供的源文件进行编译操作，需要用到 java 命令调用编译后的 class 文件或用户提供的 main jar 包，且需要在构造 java 命令时对用户提供的 jvm 参数和程序参数 进行拼接处理。另在 jdk9 之后出现了模块路劲的概念，故在此处还要针对 jdk 版本来决定将 resource jar 包加入到类路径还是模块路径中。本人在大学学了三年 Java，对 Java 语法特性以及 JVM 虚拟机都有比较充足的了解。因此在开发过程中不会因为 Java 了解不足而受到阻碍。

- SPI 机制：是一种服务提供发现机制，可以用来启用框架扩展和替换组件， Java 的 SPI 机制可以为某个接口寻找服务实现。Java 中 SPI 机制主要思想是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要，其核心思想就是解耦。经过一段时间对 DolphinScheduler 的了解，我发现它里面对模块化的设计非常优雅，这无疑得益于 SPI 的特性，此外在学习 Dubbo，Seata 等项目时也看到过很多 SPI 的使用方式，可以说对 SPI 非常熟悉，而本次 Java 任务类型插件也是将创建 TaskChannel 的工厂接口 TaskChannelFactory 作为 SPI 来进行加载的。因此我相信自己可以在这个方面做到游刃有余。

- Shell：DolphinScheduler 项目运行不同任务类型的机制都是 Worker Server 构造 Shell 命令交由 Java 代码通过 ProcessBuilder 构造进程对象进行调用执行，Java 类型任务也不例外，也需要通过 shell 来调用 javac，java 等命令，不过在 Java 任务类型中对 shell 的使用与系统中其他任务类型差不多，所以可以在开发过程中借鉴其他任务类型对这一块的实现细节。在此处需要注意到一个问题，那就是操作系统对于 shell 命令的长度是存在限制的，故当 java 任务类型的 resource jar 太多时，还需要考虑生成临时的 shell 脚本文件进行调用。由于本人日常学习开发都是在 Linux 发行版 Manjaro Kde 桌面环境中进行的，因此对 shell 的使用操作都比较熟悉。

- Vue3.0+TypeScript：在学习项目后，了解到 dolphinscheduler-ui 模块是采用 Vue3.0 加 TypeScript 的组合进行开发的，而本人在申请此课题之前只使用过 Vue2.0+JavaScript 开发 WebUI，而并没有学习过 Vue3.0+TypeScript 的知识，不过在经过为数一周左右的摸索学习后，本人还是比较轻松地入门了 Vue3.0+TypeScript 语法基础与工程开发过程。虽然在短时间内对新技术进行深入学习是一件非常枯燥的事情，但是在这期间也有不少收获，例如本人在学习 Vue3.0+TypeScript 过程中完成了本次项目所要求的 WebUI 功能设计，并且在学习过程中发现了原有代码的一些 bug，然后提交了相关的 issue 和 pr 给到了社区最后还被 merge 了。

D.申请书内容介绍

- 在本文第 2 部分将对 DolphinScheduler 后端中有关 Task Plugin 的模块进行分析，并将描述该模块的功能组织结构。因为只有充分认识系统，理解系统后，才能更好地使用和开发系统 。

- 在本文第 3 部分将对 Java Task Plugin 后端部分的实现进行介绍，并用一些诸如图形化和伪代码的形式详细介绍自己的设计实现思路，在最后简单地设计几个单元测试用例并使用 HttpClient 设计后端接口功能测试用例。

- 在本文第 4 部分将针对 DolphinScheduler 前端有关 Task Plugin 模块的功能组织结构进行简要分析。

- 在本文第 5 部分将对 Java Task Plugin 前端的设计实现细节进行详细介绍。

- 在本文第 6 部分中对整个设计开发的时间周期进行详细规划。
- 在本文第 7 部分结合本人对项目和社区的一些理解谈一谈对未来的展望。

2.项目后端分析

A.Task Plugin 接口依赖关系分析

通过阅读源码并结合官网文档，可以比较容易理解扩展一个 Task Plugin 需要做哪些操作:

- 实现 org.apache.dolphinscheduler.plugin.task.api.TaskChannel 接口，它主要包含创建任务（任务初始化，任务运行等方法）、任务取消等方法。
- 实现 org.apache.dolphinscheduler.plugin.task.api.TaskChannelFactory 接口，他是 TaskChannel 的一个简单工厂，其他模块通过 SPI 加载这个工厂接口的实现类来实现插件扩展。
- 继承 org.apache.dolphinscheduler.plugin.task.api.ShellCommandExecutor 类，继承这个类可以获得通用的 shell 命令执行能力。
- 继承 org.apache.dolphinscheduler.plugin.task.api.AbstractTask 类和 AbstractTaskExecutor 类，可以获得通用的 Task 属性和方法
- 在实现类中组合 TaskExecutionContext 类，该类中包含 Task 前端入参和运行时所需要的一系列重要参数。

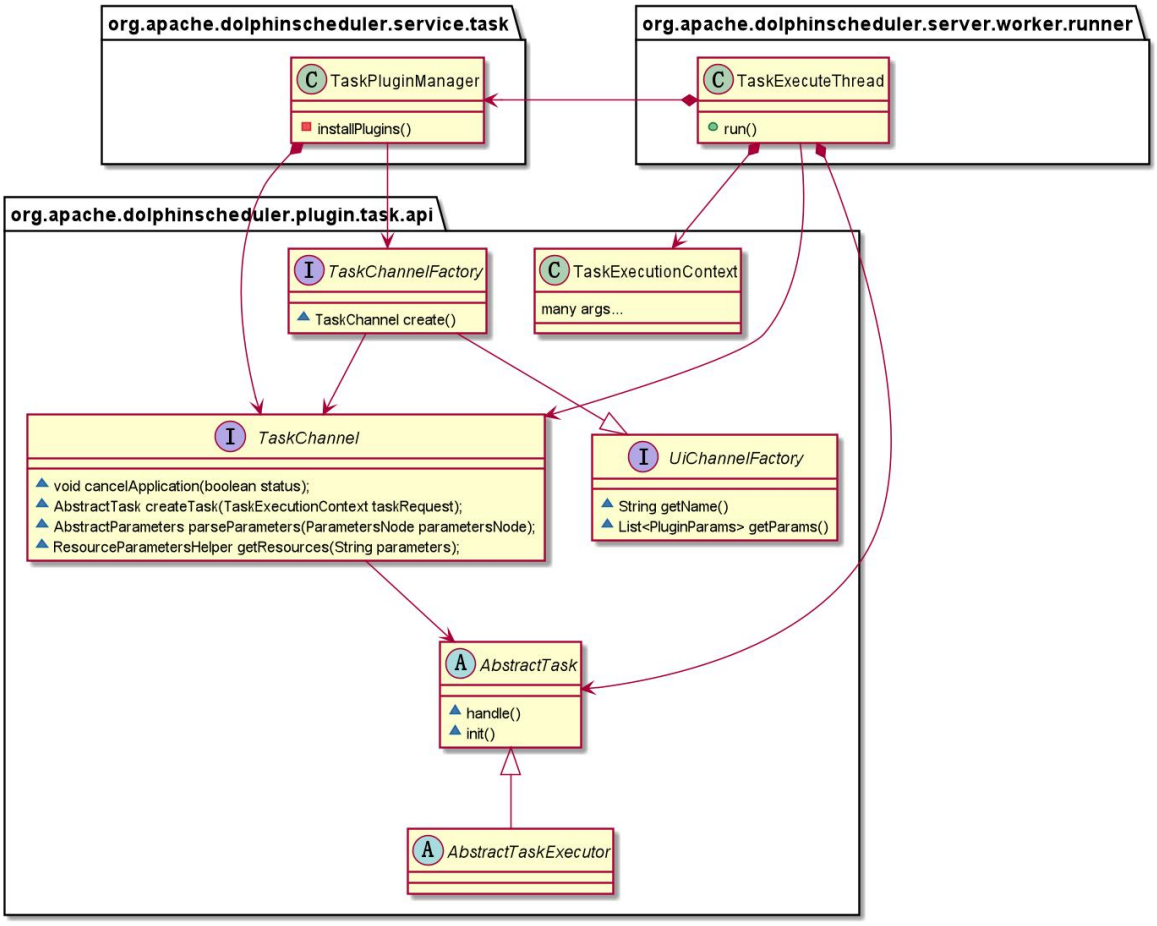


图 2-1 Task Plugin 接口依赖情况类图

如图 2-1，TaskPluginManager 用于加载 TaskPlugin 的 Channel 并对其进行缓存，以供其他模块获取，而 TaskExecuteThread 则是通过 TaskPluginManager 获取 TaskChannel 并构造 Task 然后调用 handlef 来执行 Task。其中的 TaskExecutionContext 作为任务执行上下文，存储了 Task 执行所需要的所有数据信息，包括用户定义的脚本代码，资源列表等等。

B.线程执行与插件加载流程分析

除了了解 Task Plugin 的接口结构外，了解 Task 的运行时调用过程和插件 SPI 加载过程对我们扩展新的插件的大有裨益。因此，我画了几张流程图来总结了 Task 的调用过程。

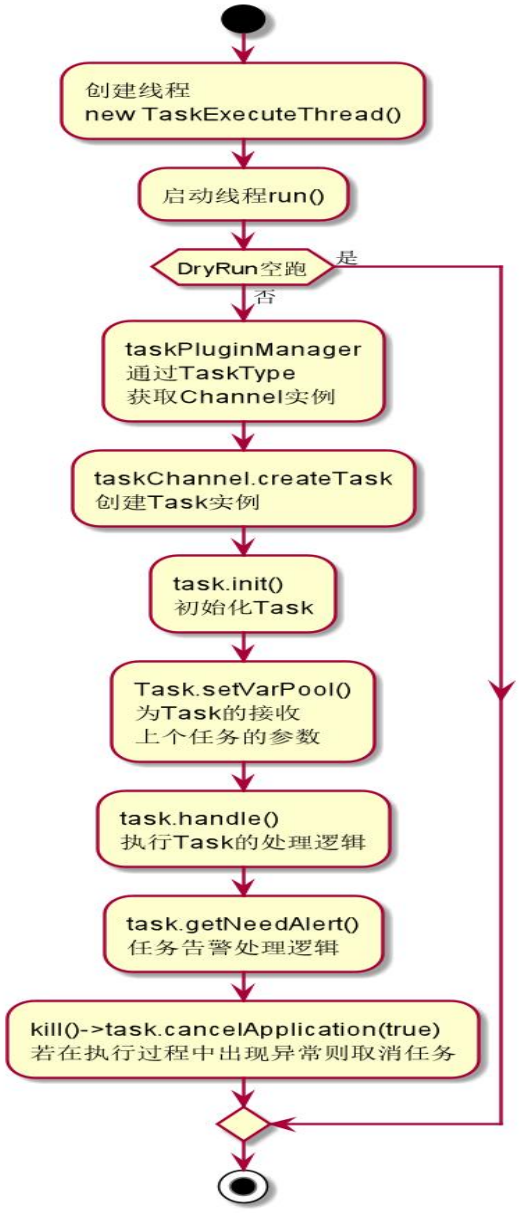


图 2-2 任务执行线程 run 方法流程图

图 2-2 展示了 org.apache.dolphinscheduler.server.worker.runner.TaskExecuteThread 线程类在启动之后 run 方法的执行逻辑，其比较核心的流程为：

- taskPluginManager.getTaskChannelMap().get(taskExecutionContext.getTaskType())方法：这个方法通过 taskPluginManager 获取缓存的 TaskChannel。
- taskChannel.createTask(taskExecutionContext)方法：该方法通过 TaskChannel 来构造一个 Task 实例。
- task.init()方法：使用该方法让 Task 进行初始化操作
- task.getParameters().setVarPool(taskExecutionContext.getVarPool())方法：该方法用于获取前置任务的传参。
- task.handle();方法调用 Task 的业务流程处理操作。

图 2-3 展示了 org.apache.dolphinscheduler.service.task.TaskPluginManager 类进行插件安装的过程，这个过程比较重要的部分在于理解 TaskChannelFactory 中 name 的作用，结合图 2-1 我们可以发现，这个 name 就是用作 TaskPluginManager 缓存 map 的 key 的，也就是 TaskType，用来标识任务类型。

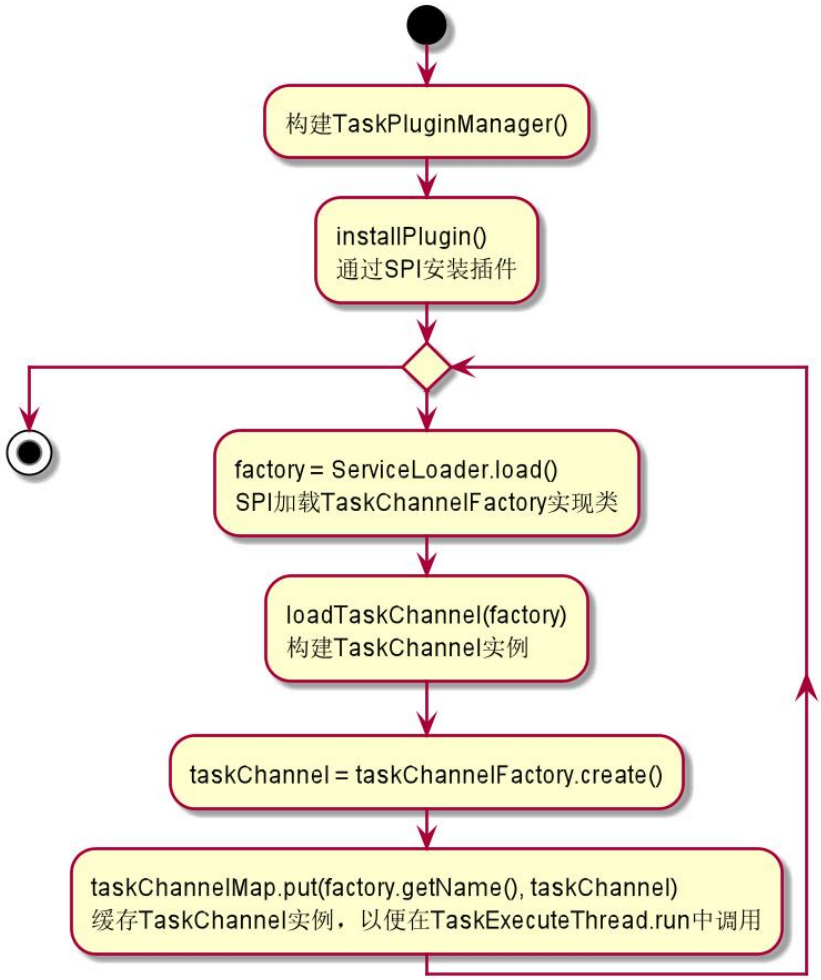
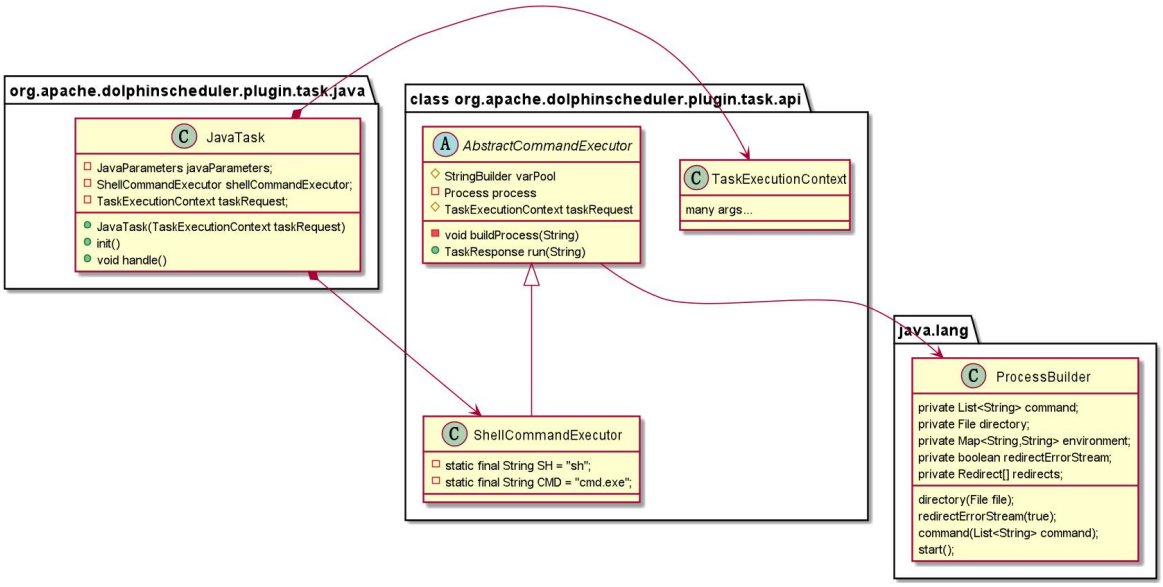


图 2-3 插件的 SPI 加载流程

C.Shell 命令执行流程分析

如下类图描述了 Shell 命令在执行过程中涉及的一些核心类。



图表 2-4 Shell 命令相关类图

如下时序图描述了 Shell 命令在执行过程中的一些核心流程。

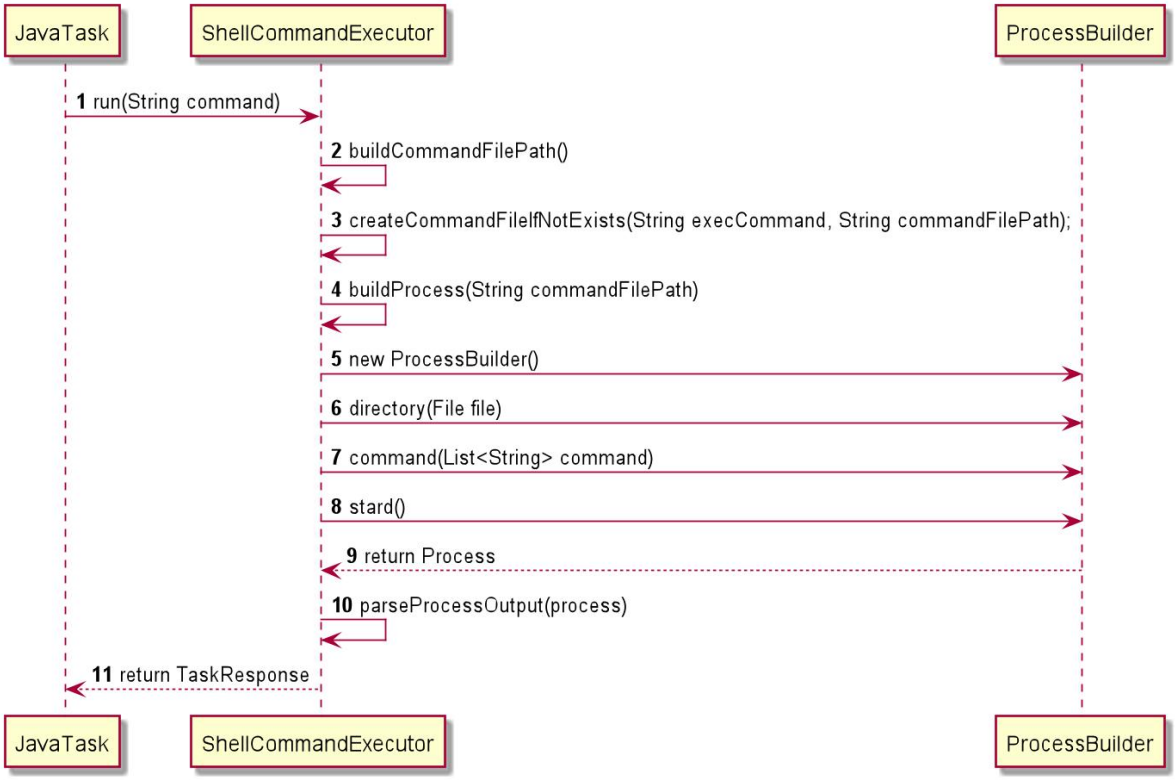


图 2-5 Shell 命令执行时序图

从图 2-4 和图 2-5 中可以很容易看出 Task 的 Shell 命令执行顺序:

1. JavaTask 的 handle 方法调用 ShellCommandExecutor 的 run 方法并传入构造好的命令字符串。
2. ShellCommandExecutor 根据操作系统的区别生成不同的命令行文件（sh 或 cmd），然后使用 createCommandFileIfNotExists 创建 sh 文件或 bat 文件，并将 run 中的参数写入脚本文件。
3. 调用 buildProcess 构造进程对象
   a) 创建 ProcessBuilder 实例，即 Process 进程的一个构造器对象。
   b) 为 Process 进程配置执行目录。
   c) 为 Process 进程配置一个 List 作为 command 命令，这里的 command 是用来执行进程的，它包含的 sudo 用来提升命令权限，它的-u tenantcode 就是使用前端指定的租户来执行命令，这里与 run 方法提供的 command 命令不能混淆。
   d) 调用构造器的 start 方法构造 Process 进程对象并执行它。
   e) start 会返回一个进程对象 Process，ShellCommandExecutor 会将这个对象赋值到自身的字段中。
4. ShellCommandExecutor 对象在 run 方法中调用 parseProcessOutput 来解析进程对象 Process 的输出结果，其内部的主要操作有获取 varPool 等，这是为了方便 JavaTask 中 handle 方法能够获取执行结果 varPool，并将这个结果传递给它的后置任务。

3.项目后端功能设计实现

A.Java Task Plugin 类功能设计

如下为 Java Task Plugin 的类图设计。

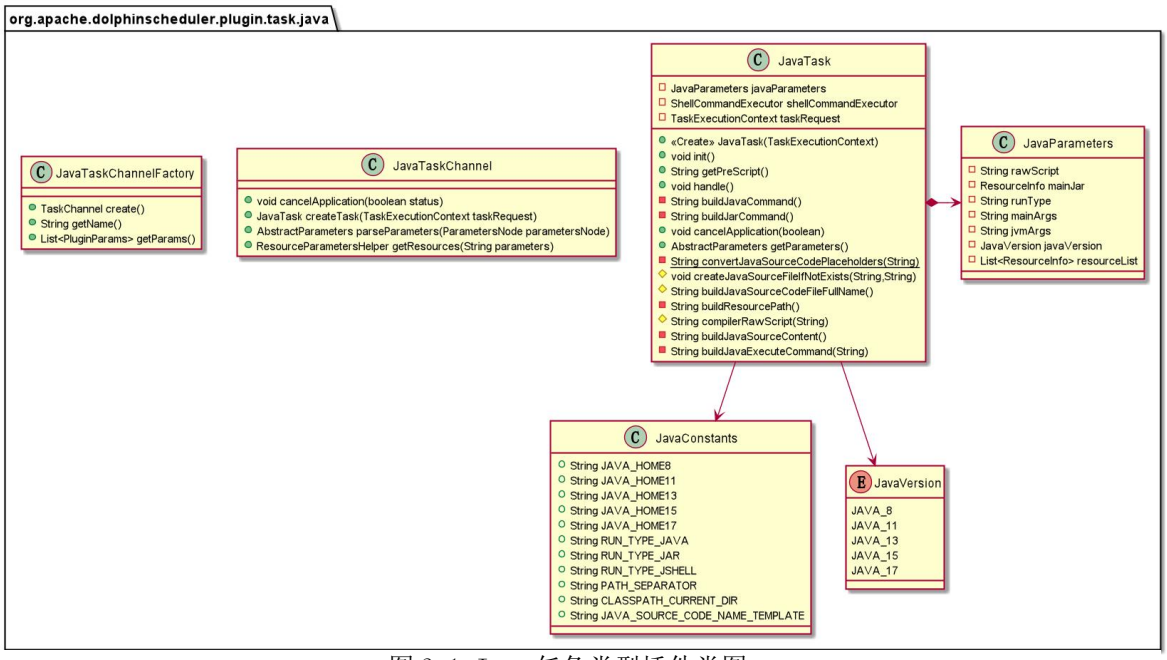


图 3-1 Java 任务类型插件类图

如下为 Java Task Plugin 及其依赖关系类图。

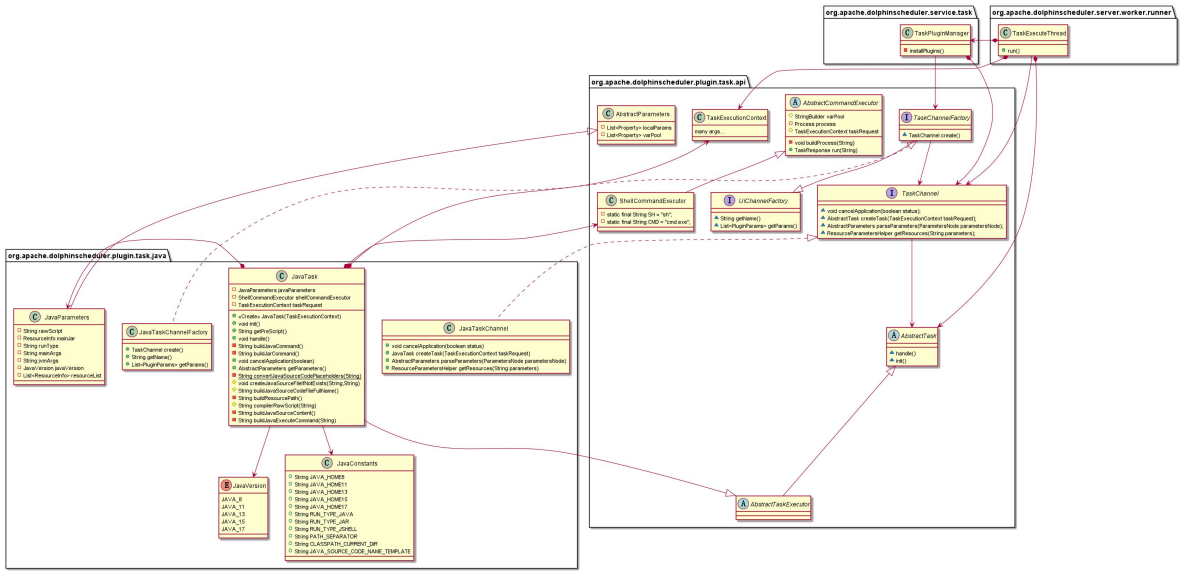


图 3-2 Java 任务类型插件及依赖关系类图

上述图 3-1 和图 3-2 比较完整地概括了我对本次课题的类关系的设计实现思路，通过类图可以很直观地发现 Task Plugin 的接口设计非常优雅，在 DolphinScheduler 中存在

很多像这样的优秀代码设计，它们很符合依赖倒置的原则，也就是依赖抽象接口而不依赖具体实现，这对后续的扩展非常友好。下面对各个类的详细设计进行介绍：

- 类 JavaConstans：这个类包含了 Java 任务类型插件在运行过程中需要使用到的常量信息，它的常量包含支持版本 JDK 的 JAVA_HOME 常量，任务运行模式常量，因 JDK9+推出了 jshell 运行模式，还预留了该常量作为未来的扩展，另外还定义了类路径或模块路径分隔符常量，当前路径常量，源代码文件全路径名称模板，常量值设计如下：

JAVA_HOME8	JAVA_HOME
JAVA_HOME11	JAVA_HOME8
JAVA_HOME13	JAVA_HOME11
JAVA_HOME15	JAVA_HOME13
JAVA_HOME17	JAVA_HOME17
RUN_TYPE_JAVA	JAVA
RUN_TYPE_JAR	JAR
RUN_TYPE_JSHELL	JSHELL
PATH_SEPARATOR	System.getProperty("path.separator")
PATH_CURRENT_DIR	. <--（这是个英文句号）
JAVA_SOURCE_CODE_NAME_TEMPLATE	%s/java_%s.java

图 3-3 常量名与常量值对照表

如图 3-3 所示，当 JavaVerion 版本为 8 时，使用默认的 JAVA_HOME 作为环境变量名。常量值显示 Java 任务类型插件将对每个长期支持的 Java 发行版进行支持

- 类 JavaPmeters：这个类继承 AbstractParameters，代表拥有 varPool 作为前置，后置任务参数传递的能力，另外我还定义了 6 个字段作为前端接收前端提交的参数，支撑 Java 任务类型的运行。这 6 个字段类型与作用如下表。

字段名	字段类型及作用
rawScript	String 类型，存储前端提交的 Java 源代码或者以后的 jshell 脚本，他是 raw 的，即存在${}的占位符需要替换。
mainJar	ResourceInfo 类型，用 jar 包用来作为 JAR 运行模式的入口。
runType	String 类型，代表此处任务的运行模式，通过匹配常量表中的值来判断是属于 JAVA, JAR 还是 JSHELL 运行模式。
mainArgs	String 类型，传递给入口 main 函数中的 String args[]参数。
jvmArgs	String 类型， JVM 参数，支持用户指定一些 JVM 的启动参数。
javaVersion	JavaVersion 类型，枚举类型，指定此次任务使用的 JDK 版本

- 枚举 JavaVersion：匹配任务执行的 JDK 版本，当前设计支持的版本为 8,11,13,15,17 等长期支持版，若日后有新版本也可进行扩展。
- 类 JavaTaskChannelFactory 实现接口 TaskChannelFactory：一个简单工厂，其 create 方法用来创建 JavaTaskChannel。getName 方法返回任务类型插件名称，该名称会与前端提交的任务类型进行匹配，该方法返回的名称还会用作 TaskChannelManager 中缓存 Map 的 key。在项目中我将返回值设计为字符串“JAVA”。
- 类 JavaTaskChannel 实现接口 TaskChannel：也属于一个工厂，核心方法 createTask 用来创建 JavaTask 实例，并在创建时为实例注入 TaskExecutionContext 任务执行上下文信息，JavaTask 后续操作都需要从上下文中来获取信息。
- 类 JavaTask 继承类 AbstractTaskExecutor：该类作为本次项目设计的核心类，属于重中之重，因此我会在接下来的部分对其中每个方法和字段进行详细介绍。通过第 2 部分的分析我们可以很清晰的知道 Task 中各处代码的执行顺序，我将按照代码调用顺序讲解 JavaTask 中的代码实现。
    - JavaTask 字段列表:

字段名	类型及作用介绍
javaParameters	JavaParameters 类型，存储前端提交的任务参数，用于在 JavaTask 中随时能够获取参数信息
shellCommandExecutor	ShellCommandExecutor 类型，用于 JavaTask 调用执行 Shell 命令
taskRequest	TaskExecutionContext 类型，用于在 JavaTask 中，可以随时获取执行任务上下文的信息

    - JavaTask 构造方法:

```
public JavaTask(TaskExecutionContext taskRequest) {
    super(taskRequest);
    this.taskRequest = taskRequest;
    this.shellCommandExecutor = new ShellCommandExecutor(this::logHandle,
            taskRequest,
            logger);
}
```

代码 3-1 JavaTask 构造方法

任务执行线程第一次与 JavaTask 交互便是通过 JavaTaskChannel 创建一个实例对象，如代码 3-1 所示，在构造方法中需要将任务执行上下文缓存在 JavaTask 对象当中，并且构造一个 Shell 命令执行器。

- JavaTask 的初始化方法 init:

```
public void init() {
    logger.info("java task params {}", taskRequest.getTaskParams());
    javaParameters = JSONUtils.parseObject(taskRequest.getTaskParams(), JavaParameters.class);
    if (javaParameters==null||!javaParameters.checkParameters()) {
        throw new TaskException("java task params is not valid");
    }
}
```

代码 3-2 JavaTask **初始化方法** init

任务执行线程在构造 JavaTask 对象之后会调用其初始化方法，在初始化方法中最重要的操作是接收前端提交的参数信息。如代码 3-2 所示，JavaTask 在初始化时将前端 JSON 格式参数序列化成 JavaParameters 对象。JavaTask 没有参数就不能执行，因此参数为空是非法的，在代码中还需要进行合法性校验。

- JavaTask 的业务逻辑方法 handle:

```
public void handle() throws Exception {
    try {
        String command = null;
        switch (javaParameters.getRunType()) {
            case JavaConstants.RUN_TYPE_JAVA:
                command = buildJavaCommand();
                break;
            case JavaConstants.RUN_TYPE_JAR:
                command = buildJarCommand();
                break;
            case JavaConstants.RUN_TYPE_JSHELL:
                // 留用扩展，构造 JSHELL 命令
        }
        TaskResponse taskResponse =
shellCommandExecutor.run(buildJavaExecuteCommand(command));
       … 设置返回值，响应代码，varPool 值传递等操作
    } catch (Exception e) {
      …异常处理操作，打印日志，设置错误响应代码等。

    }
}
```

代码 3-3 JavaTask **业务逻辑方法** handle

如代码 3-3，在核心方法 handle 中先通过判断前端提交的参数中指定哪一个运行方式，然后调用不同方法进行命令构造，（这里还预留了未来对 JSHELL 的扩展）。在初步构造命令格式后还需要通过 buildJavaExecuteCommand 方法来选定 JDK 版本，再然后才通过 shellCommandExecutor.run()来执行我们构造的命令，接下来的操作就是进行执行响应结果和异常处理了。上述出现的方法会在接下来进行详细介绍。

- buildJavaExecuteCommand 方法确定 JDK 版本

```
private String buildJavaExecuteCommand(String args) {
    String javaHome = null;
    switch (javaParameters.getJavaVersion()) {
        case JAVA_11:
            javaHome = System.getProperty(JavaConstants.JAVA_HOME11);
            break;
        case JAVA_13:
            javaHome = System.getProperty(JavaConstants.JAVA_HOME13);
            break;
        case JAVA_15:
            javaHome = System.getProperty(JavaConstants.JAVA_HOME15);
            break;
        case JAVA_17:
            javaHome = System.getProperty(JavaConstants.JAVA_HOME17);
            break;
        case JAVA_8:
        default:
            javaHome = System.getProperty(JavaConstants.JAVA_HOME8);
    }
    return javaHome + System.getProperty("file.separator") + args;
}
```

代码 3-4 buildJavaExecuteCommand 方法确定 JDK 版本

如代码 3-4，在构造完命令基本结构后，通过该方法来确定一个 JDK 版本，当用户没有提供 JavaVersion 时，此处默认使用 JDK8 作为默认的 JDK。方法通过 System 提供的 API 查询系统环境变量得到 JDK 路径，并将 JDK 路径与构造好的命令进行拼接。

- buildJavaCommand 方法构造 JAVA 运行模式的 shell 命令

```
private String buildJavaCommand() throws Exception {
    String sourceCode = buildJavaSourceContent();
    String className = compilerRawScript(sourceCode);
    StringBuilder builder = new StringBuilder();
    builder.append("java").append(" ")
            .append(className).append(" ")
            .append(javaParameters.getMainArgs().trim()).append(" ")
            .append(javaParameters.getJvmArgs().trim()).append(" ")
            .append(buildResourcePath());
    return builder.toString();
}
```

代码 3-5 buildJavaCommand 方法构造 JAVA 运行模式的 shell 命令

如代码 3-5，在该方法中将会针对 JAVA 运行模式构造 shell 命令。首先使用 buildJavaSourceContent 将 rawScript 中的${}占位符替换成相关变量或参数的值，然后使用 compilerRawScript 进行编译操作并得到编译后的类名，最后使用 StringBuilder 构造一个 java 命令的 shell 语句，在构造语句的最后一段落使用 buildResourcePath 方法将资源列表添加到类路径或模块路径上。

- buildJarCommand 方法构造 JAR 运行模式的 shell 命令

```
private String buildJarCommand() {
    String fullName = javaParameters.getMainJar().getResourceName();
    String mainJarName = fullName.substring(0, fullName.lastIndexOf('.'));
    mainJarName = mainJarName.substring(mainJarName.lastIndexOf('.') + 1) + ".jar";
    StringBuilder builder = new StringBuilder();
    builder.append("java").append(" ")
            .append("-jar").append(" ")
            .append(mainJarName).append(" ")
            .append(javaParameters.getMainArgs().trim()).append(" ")
            .append(javaParameters.getJvmArgs().trim()).append(" ")
            .append(buildResourcePath());
    return builder.toString();
}
```

代码 3-6 buildJarCommand 方法构造 JAR 运行模式命令

如代码 3-6，该方法构造命令的逻辑与代码 3-5 中非常类似，不同的地方在于 JAR 模式不需要编译 jar 包，且构造 JAR 运行模式的命令需要提供 -jar 选项，需要提供文件名后缀.jar，而 JAVA 运行模式不需要提供.class 后缀，否则会产生错误。

- buildResourcePath 方法生成类路径或模块路径

```
private String buildResourcePath() {
    StringBuilder builder = new StringBuilder();
    if (javaParameters.getJavaVersion() == JavaVersion.JAVA_8) builder.append("-class-path");
    else builder.append("-module-path");
    builder.append(" ").append(JavaConstants.PATH_CURRENT_DIR)
            .append(JavaConstants.PATH_SEPARATOR)
            .append(taskRequest.getExecutePath());
    for (ResourceInfo info : javaParameters.getResourceFilesList()) {
        builder.append(JavaConstants.PATH_SEPARATOR);
        builder.append(info.getResourceName());
    }
    return builder.toString();
}
```

代码 3-7 buildResourcePath 方法生成类路径或模块路径

如代码 3-7，构造类路径或模块路径的思路其实就是遍历 ResourceInfo List，然后将 ResourceInfo 的全路径名称加入到路径当中，此处需要注意到 Unix 和 Windows 的路径分割不是不同的，Unix 是“:”符号，而 Windows 下是“;”符号，所以我在 JavaConstants 中定义了常量来解决该问题。另外需要提到的一点是使用 taskRequest.getExecutePath 方法得到文件所在的位置，并将其也作为了构造路径的一部分，这样做的好处就是在使用 java classname 或 java -jar jarname 时不需要提供全限定类名或绝对路径名了。最后需要注意的是，当用户要求的 JDK 版本不是 8 时我们默认使用到 JDK9+的模块化特性。

- buildJavaSourceContent 和 convertJavaSourceCodePlaceholders 方法替换变量

```
private String buildJavaSourceContent(){
    String rawJavaScript = javaParameters.getRawScript().replaceAll("\\r\\n", "\n");
    Map<String, Property> paramsMap = ParamUtils.convert(taskRequest, javaParameters);
    if (MapUtils.isEmpty(paramsMap)) {
        paramsMap = new HashMap<>();
    }
    if (MapUtils.isNotEmpty(taskRequest.getParamsMap())) {
        paramsMap.putAll(taskRequest.getParamsMap());
    }
    rawJavaScript = ParameterUtils.convertParameterPlaceholders(rawJavaScript,
ParamUtils.convert(paramsMap));
    logger.info("raw java script : {}", javaParameters.getRawScript());
    return rawJavaScript;
}
```

代码 3-8 替换变量

如代码 3-8，在该方法中，主要进行的操作是将用户定义脚本中的${}占位符替换为 Task 定义的 LocalParam， Workflow 定义的 GlobalParam 和前置任务传递的 VarPool 中对应 key 的值。该方法其实大部分代码都是借鉴的其他 Task Plugin 如 Shell, Python 等的实现。

- compilerRawScript 方法编译源代码

```
protected String compilerRawScript(String sourceCode) throws IOException, InterruptedException {
    String fileName =  buildJavaSourceCodeFileFullName();
    createJavaSourceFileIfNotExists(sourceCode, fileName);
    String className = fileName.substring(0 ,fileName.lastIndexOf('.'));
    className = className.substring(className.lastIndexOf('.') + 1);
    StringBuilder compilerCommand = new StringBuilder().append("javac").append(" ")
            .append(className + ".java").append(" ")
            .append(buildResourcePath());
    try{
        shellCommandExecutor.run(compilerCommand.toString());
    }finally {
        Files.delete(Paths.get(fileName));
    }
    return className;
}
```

代码 3-9 compilerRawScript 方法编译源代码

如代码 3-9，这个方法的方法名其实设计得并不好，叫做 compilerSourceCode 会更加清晰明了，后续将会更正。该方法进行可以整体描述为创建源代码文件，构造编译命令，执行编译命令，删除源代码文件四个步骤。其中构造编译命令的过程与构造执行命令没有太大差异。这里需要注意的是执行编译命令的步骤，因为他可能会抛出一个异常，如果此处没有进行异常处理的话就可能出现临时文件不被删除，最后导致系统资源泄漏。

- buildJavaSourceCodeFileFullName 方法构建源文件路径

```
protected String buildJavaSourceCodeFileFullName() {
    return String.format(JavaConstants.JAVA_SOURCE_CODE_NAME_TEMPLATE,
taskRequest.getExecutePath(), taskRequest.getTaskAppId());
}
```

代码 3-10 buildJavaSourceCodeFileFullName 方法构建源文件路径

如代码 3-10， 这个方法只有短短一行代码，它的作用就是使用模板常量来构造一个源代码文件的绝对路径。

B.Java Task Plugin 运行流程描述

通过上述对 Java Task Plugin 中类设计实现的描述，以及对核心类 JavaTask 各个方法的详细描述，Java 任务类型的运行流程各个实现细节已经非常清晰明了，那么我将在这一小节提供一个流程图，它可以让我们更加直观地理解 Java 任务类型的执行过程。

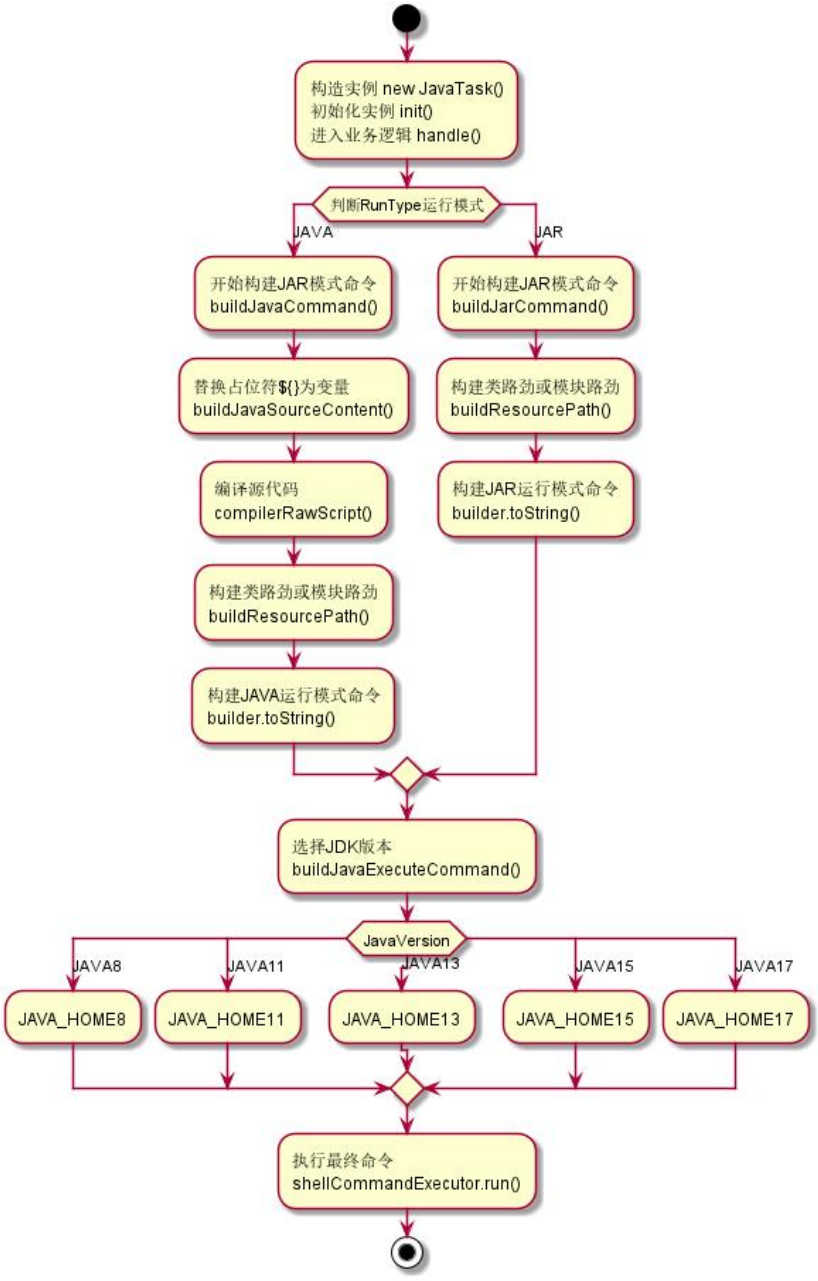


图 3-4 Java Task Plugin 运行流程图

图 3-4 很直观地展示了 Java 任务类型地执行逻辑，当进入 handle 方法后，首先判断 RunType 运行模式，然后对不同模式来构造 shell 命令，再判断 JavaVersion 确定 JDK 版本地 JAVA_HOME 环境变量，最后使用 shellCommandExecutor 执行命令。

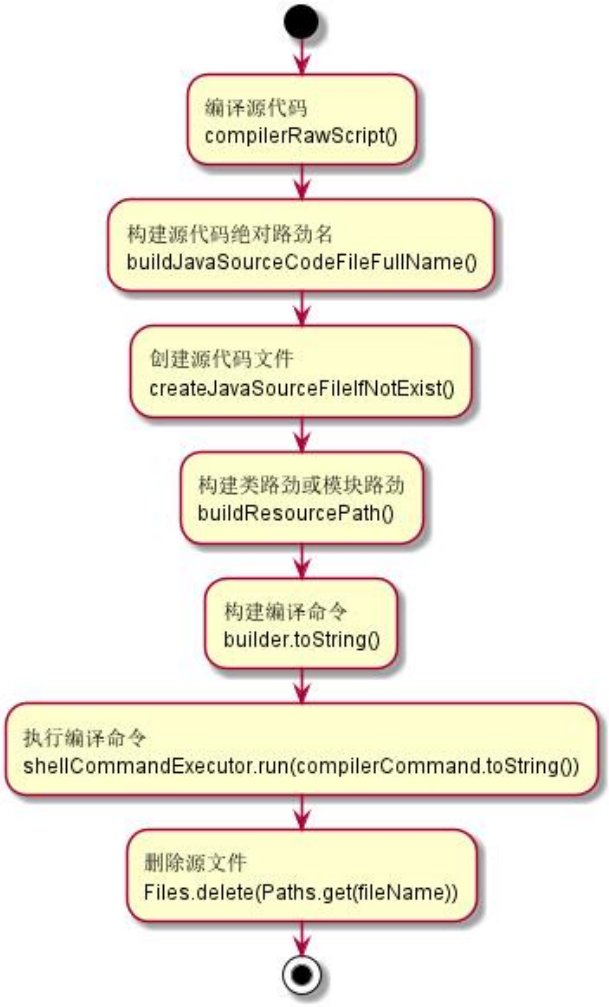


图 3-5 源代码编译流程图

因为一张图太大了，图 3-4 中省略了编译过程，所以在此页附上源代码编译流程图。

图 3-5 展示了进入 compilerRawScript 方法后源代码地编译流程图，该方法中首先会构建源文件地绝对路径名，然后创建该文件并将源代码输入到该文件，再然后就是构造类路劲或者模块路径并且构造出一条编译命令了。在该方法中还需要执行编译命令，并且为了不造成系统资源泄露，还要删除源文件。

C.单元测试

测试用例表

测试用例	输入	期望输出	功能
testJavaCommand	rawScript,runType,List<ResourceInfo>	正确命令	测试 JAVA 模式
testJarCommand	mainJar, List<ResourceInfo>	正确命令	测试 JAR 模式
testJavaVersion	JavaVersion, java 命令	正确命令	测试 JDK 版本

D.使用 HttpClient 进行接口功能测试

在进行简单的单元测试后其实并不能保证系统能够正常运行，功能能够正常实现，因此在进行单元测试之外还需要对系统的 API 进行功能性测试。

在此处模拟的操作为：项目管理面板下任务定义和流程执行操作。

- 前置条件：登录用户，即配置 Session 和 Cookie 信息。
- 定义任务：
    1. 生成任务编号：/dolphinscheduler/projects/{projectCode}/task-definition/gen-task-codes?genNum=1
    2. 保存任务：/dolphinscheduler/projects/{projectCode}/task-definition/save-single

       参数：{processDefinitionCode}，{taskDefinitionJson}
- 执行任务： /dolphinscheduler/projects/{processDefinitionCode}/executors/start-process-instance 通过启动流程来启动任务
- 查询任务： /dolphinscheduler/projects/{projectCode}/task-definition?pageSize=10&pageNo=1&searchTaskName=&searchWorkflowName=&taskType=JAVA

4.项目前端功能分析

A.Java Task Plugin 扩展相关文件分析

如下是进行扩展时主要涉及到的 task 目录结构。

```
task
├──components
│   └──node
│       ├──fields
│       └──tasks
├──constants
├──definition
│   └──components
└──instance
```

代码 4-1 task 目录结构

在代码 4-1 中，task 是 src/views/project 下的子目录，存放的 task 相关的视图组件。

想要进行 Java 任务类型扩展，则必须在原有代码上进行修改，新增一些文件。

- task/components 及其子目录分析
  - 其中 task/components 用于存放任务节点的通用组件，
  - task/components/node 下存放节点详情组件，需要修改 format-data.ts 文件，将自定义参数写入 reqeust 请求的任务参数对象 obj 中。
  - task/components/node/fields 用于存放字段组件，需要添加一个 use-java.ts 文件来在定义任务时自定义一些 java 任务类型的相关字段，此外还需要修改 use-main-jar.ts 文件来达到 main Jar 字段的复用的目的。
  - task/components/node/tasks 用于存放任务类型组件。需要添加 use-java.ts 定义 java 任务类型，并修改 index.ts 文件来导出 java 任务类型。
- task/comstants 存放 task 相关的常量信息，需要修改 task-type.ts 文件来增加 java 任务类型的常量值。

- task/definition 用于存放任务定义面板组件，task/definition/components 用于存放版本号等组件。
- task/instance 存放任务实例面板组件
- 除此之外，想要新增 java 任务类型的扩展还需要修改国际化文件 locales/modules/en_US.ts 和 locales/modules/zh_CN.ts，因为在自定义字段时，不可避免要进行一些名称的定义，故需要做国际化处理。

小结：综上分析，可以得出进行扩展时需要修改或新增的文件如下：

1. task/comstants/task-type.ts 添加常量。
2. task/components/node/tasks/use-java.ts 添加 java 任务类型组件。
3. task/components/node/tasks/index.ts 导出 java 任务类型。
4. task/components/node/fields/use-java.ts 新增自定义字段组件。
5. task/components/node/fields/use-main-jar.ts 修改并复用组件。
6. task/components/node/format-data.ts 修改文件以导出自定义字段。
7. locales/modules/en_US.ts, locales/modules/zh_CN.ts 进行国际化处理。

5.项目前端功能设计实现

A.task/comstants/task-type.ts 添加常量

```
在 TASK_TYPES_MAP 中添加如下内容
 JAVA: {
  alias: 'JAVA'
 }
为 TaskType 类型追加如下内容
  | 'JAVA'
```

代码 5-1 添加常量

代码 5-1 中描述的代码功能就是为 task 添加一个 Java 类型的常量，其中任务定义面板中选择任务类型的选项框中的值就是通过 TASK_TYPES_MAP 加载的。

B.task/components/node/tasks/use-java.ts 添加 java 任务类型组件

```
{代码块 1
    taskType: 'JAVA',
    mainJar:null,
    runType:'JAVA',
    mainArgs:'',
    jvmArgs:'',
    programType: 'JAVA'
}
{代码块 2
    ...Fields.useJava(model),
}
```

代码 5-2 添加 java 任务类型组件

由于该文件中的代码实在太长，并且我在设计实现时还参考了已有的功能实现，所以在这里就简单描述一下它和已有代码的区别，修改了那些内容:

- 代码块 1 复用了 use-pyhon.ts 的 model 对象，并删除了 taskType:'PYHON'字段，该处代码块定义的字段主要用来进行表单数据的双向绑定。
- 代码块 2 复用了 use-mr.ts 的 return json 数组，并删除了...Fields.useMr(model)，添加了自定义...Fields.useJava(model)，该处代码块定义的内容就是表单中显示的字段。

C.task/components/node/tasks/index.ts 导出 java 任务类型

```
import { useJava } from './use-java'
export default {
-  MLFLOW: useMlflow
+  MLFLOW: useMlflow,
+  JAVA:useJava
 }
```

代码 5-3 导出 java 任务类型

在代码 5-3 中的操作主要分为两步，先是将自定义任务类型 use-java 组件导入到 index.ts 中，然后再将其挂载到 JAVA 字段上进行导出。

D.task/components/node/fields/use-java.ts 新增自定义字段组件

```
  const rawScriptSpan = computed(() => (model.runType === 'JAR' ? 0 : 24))  //第 1 处
    type: 'select',  //第 2 处
    field: 'runType',
    name: t('project.node.run_type'),
    options: RUN_TYPES,
    value: model.runType
    …
    type: 'input', //第 3 处
    field: 'mainArgs',
    name: t('project.node.main_arguments'),
    …    placeholder: t('project.node.main_arguments_tips')
    …
    type: 'input', //第 4 处
    field: 'jvmArgs',
    name: t('project.node.jvm_args'),
    …    placeholder: t('project.node.jvm_args_tips')
    …
   type: 'editor', //第 5 处
   field: 'rawScript',
   span: rawScriptSpan,
    …
    useMainJar(model),   //第 6 处
    useResources(), //第 7 处
  ...useCustomParams({ model, field: 'localParams', isSimple: false }) //第 8 处
export const RUN_TYPES =  //第 9 处
[{label: 'JAVA',
   value: 'JAVA'},
  {label: 'JAR',
   value: 'JAR'} ]
```

代码 5-4 新增自定义字段组件 use-java.ts

这个组件的内容也非常多，因此我在代码 5-4 做了简化处理，下面将对这些伪代码进行详细介绍：

- 第 1 处代码定义了一个计算变量，当切换 JAVA 和 JAR 运行模式时，用于在控制面板中控制 rawScript 和 mainJar 字段表单组件的显隐状态。
- 第 2 处到 8 处代码都是在定义面板字段的详细规则，它们分别定义了 runType，mainArgs，jvmArgs，rawScript 等字段，还通过 useMainJar，useResources，useCustomParams 字段等复用了已有的一些组件。
- 第 9 处代码的作用是导出两个常量提供给 runType 的选项框进行选择。

E.task/components/node/fields/use-main-jar.ts 修改并复用组件

```
const mainJarSpan = computed(() => (model.programType === 'SQL' ? 0 : 24)) //第 1 处

const mainJarSpan = computed(() => (model.programType === 'SQL'||model.runType === 'JAVA' ? 0 : 24)) //第 2 处
```

代码 5-5 复用 use-main-jar.ts 字段组件

代码 5-5 中将 use-main-jar.ts 文件的第 1 处代码修改为第 2 处代码，作用是增加了对于 runType 的判断，通过这样到的方式轻松地复用 use-main-jar.ts 字段组件。

F.task/components/node/format-data.ts 修改文件以导出自定义字段

```
 if(data.taskType === 'JAVA'){
  taskParams.runType = data.runType
  taskParams.mainArgs = data.mainArgs
  taskParams.jvmArgs = data.jvmArgs
  if(data.runType === 'JAR'){
   taskParams.mainJar = data.mainJar
  }
 }
```

代码 5-6 为 request 请求导出自定义字段

将代码 5-6 中的代码添加到 format-data.ts 文件的 formatParams 方法中，作用是使得 Java 任务类型自定义的字段可以成为 http request 请求中请求参数的一部分。

G.locales/modules/en_US.ts, locales/modules/zh_CN.ts 进行国际化处理

```
 project.node.run_type: '运行方式' || 'Operation Mode'

 project.node.main_arguments: '虚拟机参数'|| 'JVM Args'

 project.node.main_arguments _tips: '请输入虚拟机参数'|| 'Please type the jvm args'
```

代码 5-7 进行国际化处理

代码 5-7 中使用伪代码的方式表示了需要国际化处理的几个字段以及它们的值。

6.时间规划

在报名开源之夏之前，我已经找到了一份暑期实习工作但是暂时没有入职，因为我认为一段开源经历要比实习经历重要得多，因为社区的同学们都非常友好，在这里能够结识到更多志同道合的朋友，此外还能迅速提升自己的技能。再者说入职实习想留下转正不一定能行，但在社区中只要愿意贡献，就肯定有一方净土！因此如果有幸能够参与到 DolphinScheduler 的这个项目来，我将会拒绝拿到的实习 Offer，并将暑假的时间全额投入到开源社区中。所以有幸中标的话，我将有充足且高效的时间来完成这个项目的任务的各种工作。也许您认为我目前正在参与其他开源社区的工作，会无暇顾及本次开源之夏的任务，其实很容易发现这种想法站不住脚，因为开源社区的同学不会给贡献者们施加太多生活上的压力，如果我能同时参与到多个社区中来，其他社区的同学也会考虑到我的情况而不会给我分配太沉重的任务。

我认为在项目开发过程中与导师的沟通是非常重要，因此我计划每周与导师进行一次汇报沟通，内容就是汇报本周工作的完成情况以及下周工作如何准备等。具体的沟通方式和时间需要在中标后与导师商量。在项目开发过程中可能会出现很多其他因素导致任务开发与任务计划不能达到完全统一，因此我在时间规划中分配了一些弹性时间来应对这样的情况，以此让整个项目开发过程能够顺利地进行。

以下是我制定的一个简短的时间规划，在后期开发过程中或与导师沟通之后可能会有一些小的改动。

我将整个项目开发周期总共划分为 3 阶段，共有 11 周，累计 396 小时。

A.第一阶段：后端功能实现（7.1 – 8.1）144 小时

- 第一周 36 小时
  - 设计 Java Task Plugin 功能实现细节（35 小时）
  - 向导师汇报进度与安排（1 小时）
- 第二周 36 小时
  - 设计 Java Task Plugin 功能实现细节（25 小时）
  - 优化 Shell 命令实现细节（10 小时）
  - 向导师汇报进度与安排（1 小时）
- 第三周 36 小时
  - 设计单元测试用例 （20 小时）
  - 使用 HttpCilent 进行接口功能测试（15 小时）
  - 向导师汇报进度与安排（1 小时）

- 第四周 36 小时
  - 进行系统集成测试（10 小时）
  - Code Review 优化代码（10 小时）
  - 完善接口文档（5 小时）
  - 弹性时间（10 小时）
  - 向导师汇报进度与安排（1 小时）

B.第二阶段：前端功能实现（8.1 – 9.1）144 小时

- 第一周 36 小时
  - 绘制前端 UI 界面（35 小时）
  - 向导师汇报进度与安排（1 小时）
- 第二周 36 小时
  - 优化前端 UI，思考如何使用户操作更加方便（10 小时）
  - 实现前端 API 调用功能（25 小时）
  - 向导师汇报进度与安排（1 小时）
- 第三周 36 小时
  - 测试前端 API 调用功能性（10 小时）
  - 前端 UI 与 API 调用集成测试（10 小时）
  - 编写使用文档（15 小时）
  - 向导师汇报进度与安排（1 小时）
- 第四周 36 小时
  - 进行系统集成测试（10 小时）
  - Code Review 优化代码（10 小时）
  - 弹性时间安排（15 小时）
  - 向导师汇报进度与安排（1 小时）

C.第三阶段：文档编写与项目 Review（8.1 – 8.23）108 小时

- 第一周 36 小时
  - 编写官网使用文档（20 小时）

- 编写开发者文档（15 小时）
- 向导师汇报进度与安排（1 小时）

- 第二周 36 小时
  - 对项目进行 review 与维护，并思考优化方案（35 小时）
  - 向导师汇报进度与安排（1 小时）

- 第三周 36 小时
  - 进行结项报告文档编写（35 小时）
  - 向导师汇报进度与安排（1 小时）

7.展望未来（9.31 – …）

在编写项目方案书的过程中，我思考了很多关于项目的优化和扩展方案，例如在正文中多次提到的多版本 JDK 支持，JSHELL 脚本支持等。除此之外我还思考了很多想在以后进行的扩展东西，我并没有把它们写在正文当中，它们包括但不限于 Dokcer 容器执行，GraalVM native 编译热启动，OSGI 实现 Worker Server JVM 部署隔离 Java 任务代码和提供用户提前编译代码选项等。总的来说我对开源之夏这次的项目充满了憧憬，并且付诸了长达数十天的辛苦实践，无论最后是否申请成功，都不可否认这次脚踏实地的编写项目方案书，设计系统分析和项目实现图纸和设计实现代码给我带来了很大的成长。其实在此之前我对编写文档这件事是非常抗拒的，因为它是一件非常枯燥无聊的工作，总是需要做一些如字体更正等重复性极强操作。但是这次我把自己认为不可能的事做到了，并且逐渐喜欢上了这种沉稳和严谨的感觉。我相信在这之后能够克服更多自己认为的不可能!

无论无何，我都要感谢本次项目的导师，没有你的悉心帮助，我不一定能做出这么一份完善的项目方案书。其次还要感谢社区的其他同学，没有你们建立的社区就没有今天参加开源之夏的我们，就没有一个能让我们立足的肩膀。能够站在巨人的肩膀上是一种莫大的幸福!

如果有幸能够申请成功，我将会优化自己提交给社区的成果并且积极回馈社区带给我的成长，在结项后持续为社区做出自己力所能及的贡献。阶段的结束并不意味着开源生涯的结束，它恰恰相反，是一个新的起点，我相信自己一定能从这个小小的起点走向开源世界的星辰大海!