

说明

此源码解析教程基于 `9.3.0-SNAPSHOT` 版本. 为了方便操作, 也下载 `jetty-distribution-9.2.3.v20140905.zip`, 安装在 `E:\Java\soft\jetty-distribution-9.2.3.v20140905` 目录. `jetty.home` 基于此目录.

概要

这是jetty启动的第一部分,也就是 `jetty-start` 模块部分的相关内容.

Start

从 `org.eclipse.jetty.start.Main` 的 `main()` 方法开始,这个方法也是在 `start.jar` 中的 `MANIFEST.MF` 文件中声明为 `Main-Class` 的类.在这里传递的命令行参数是:

```
1. --debug jetty.home=E:\Java\soft\jetty-distribution-9.2.3.v20140905 jetty.base=E:\Java\soft\jetty-distribution-9.2.3.v20140905\demo-base
```

也就是我们配置源码的时候添加的参数.

`main()` 主函数中主要的也就是在try中的三行代码,下面具体来看.

Main main = new Main();

此方法没必要抛出 `IOException`. 应该是代码做改动的时候没有删除.比如在 `Jetty-9.1.x` 的版本中, 此方法中实例化 `BaseHome`, 也就是进行 `jetty.home` 和 `jetty.base` 目录的操作.不过此版本中已经不再此方法中处理.

注意: `user.dir` 为 `jetty.project` 所在的目录.而不是当前目录 `jetty-start` 所在的目录.

StartArgs startArgs = main.processCommandLine(args);

`StartArgs args = new StartArgs(cmdLine);` 所做的事情就是将命令行参数存储到 `StartArgs` 类中的 `commandLine(List)` 变量中.并实例化了 `Classpath` 类.

接下来的步骤源码中写的很清楚.一共为8个步骤:

Home和Base的目录位置.

这里使用了正则表达式进行匹配. `Pattern.compile("(-D)?jetty.home=(.*)");` 这个正则不难理解. `"(-D)"` 为 `group(1)`, 可出现0次或1次. `"(.*)"` 为 `group(2)`, 也就是具体的目录.所以我们可以用两种方式指定: `jetty.home=<location>` 或者 `-Djetty.home=<location>`. 这里再次根据命令行参数处理 `jetty.home` 和 `jetty.base`. 会覆盖前面的值, 最终将其更新到系统属性中.不过这里的正则中在前面加上了 `-D`, 而这个会被作为系统参数, 也就是可用 `System.getProperty(key)` 取得.在 `BaseHome` 的构造函数中已经通过此方式处理过, 这里再次处理是否有些重复.

`jetty.home` 和 `jetty.base`.

`jetty.home`: 主要的jetty二级制和默认配置存放的位置.

`jetty.base` : 执行特定配置和获取webapps的位置.

一般来说将这两个目录设置为相同的目录即可.

启动日志记录

启动日志参数:

- `--debug` 、 `debug=true|false` 或者 `-Ddebug=true|false` 是否进行debug
- `start-log-file=<file>`

指定日志文件,这个是基于 `baseDir` 目录的,也就是位于 `baseDir` 中,可以在 `baseDir` 的子目录中.在指定的时候如果指定为 `baseDir` 的子目录,那么这个子目录必须要存在,如果不存在,jetty是不会创建的,而是抛出异常.不过jetty允许文件名为空,是否不太严格.虽然在后面会因为文件名为空抛出异常.

如果指定了日志文件,那么 `System.setOut` 和 `System.setErr` 将重定向到日志文件中.

加载 ini 文件

主要的步骤如下:

- `start.ini`
读取文件的内容,空行 或者以 `#`开头 的行将被忽略.其中 `--module=` 后可跟多个模块的名称(用","分隔).如果行的内容重复,将被忽略.
- 解析 `start.ini` 文件的每行内容
主要看 `StartArgs.parse()` 的方法,可以看出在 `start.ini` 文件中是不能出现 `--help` 或者 `?` 这样的参数的.如果存在,启动的时候会失败:

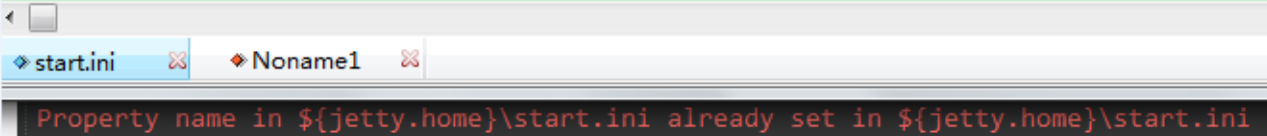
```
--help not allowed in ${jetty.home}\start.ini
```

这样的参数还有: `--stop` 、 `--dry-run` 、 `--exec-print` 、 `--add-to-startd` 、 `--add-to-start` 等.这些参数只能出现在命令行中.

注意start.ini文件中可以定义这样的参数: `-Dkey=value` (类似于系统属性).如果这里定义了 `-Djetty.home=<location>` 、 `-Djetty.base=<location>` 这样的属性,那么也会将前面定义的覆盖(再次处理 `jetty.home` 和 `jetty.base` ,不过一般谁会干这种事).

`key=value` 这样的参数 `key` 定义不能重复,重复会抛出异常.同样,这里也可以定义 `-Djetty.home=<location>` 、 `-Djetty.base=<location>` 这样的属性.不过这里定义的 `jetty.home` 和 `jetty.base` 虽然将其设置到系统属性中,但是并不会覆盖前面的值,不会对其造成影响.比如:

```
40 name=Tom
41 name=Cat
42
```



`start.ini` 文件参数总结:

- 解析 `start.d` 目录下的ini文件
默认就只有 `http.ini` ,解析过程和 `start.ini` 一样.

解析原始的命令行参数

解析过程和解析ini文件是一样的.

Module注册

所有的Module文件定义在 `{jetty.home}/modules` 目录下.这些文件可以直接使用文本编辑器打开. Jetty允许文件名为空,也就是这样的文件 `.mod` .此外Module都必须位于 `{jetty.home}/modules` 目录下.接下来的解析过程中用了很多的正则,这都是需要理解的.出现的正则在后面统一说明.注册过程主要查看 `Modules#registerAll()` 方法.

注册过程

- 注册Module
 - 一个 `.mod` 实例化一个 `Module` .实例化过程中主要关注 `init()` 和 `process()` 方法.
 - `init()` 方法很简单,取得Module名称及实例化相关的变量.
 - `process()` 方法读取文件内容,并进行相应的处理. `.mod` 文件中的以 `#` 开头的为注释内容.一般在 `[lib]` 的参数值中会出现 `${jetty.version}` .下面总结下 `.mod` 文件中可使用的参数一般格式为：

1.

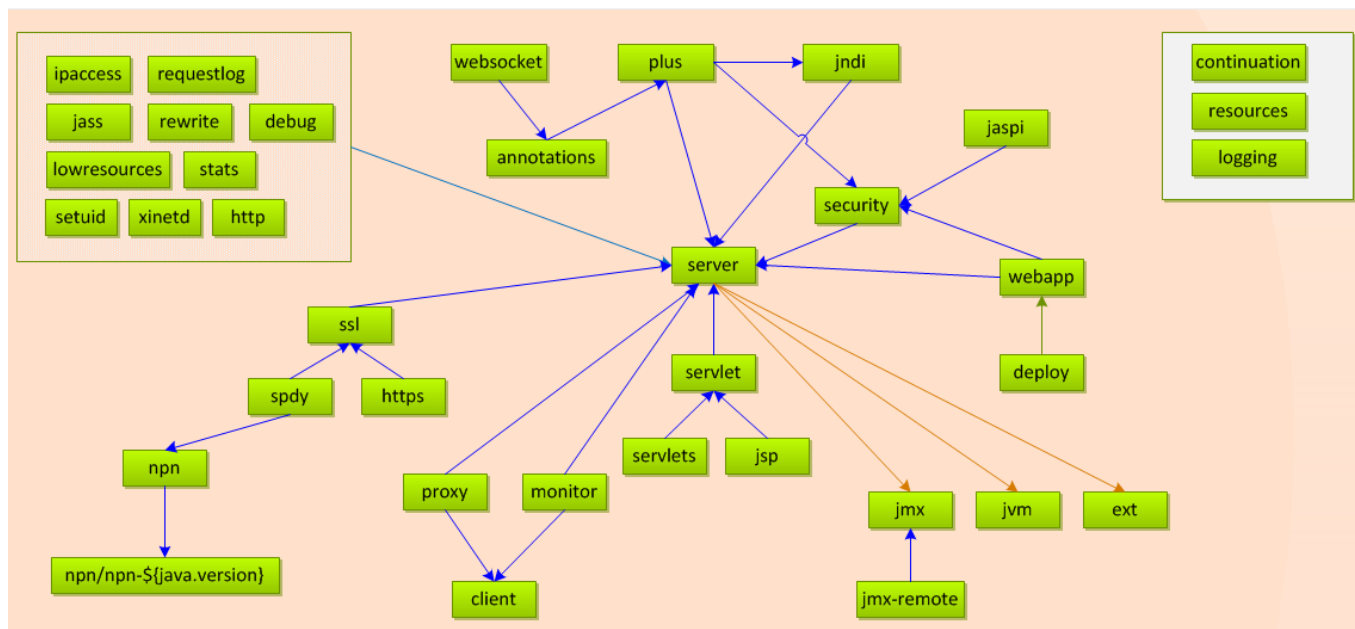
[selectionType]
2.

参数值(可多个)

参数说明：

参数名称	说明
INI-TEMPLATE	空行或#开头的内容对此是有效的
NAME	逻辑名称,会取代之前的在init方法中取得的文件的名称(<code>npn-1.7.0_9.mod</code>)
DEPEND	依赖的Module
LIB	此Module需要的lib(jar、classpath目录).可使用正则(<code>ext.mod</code>).也可用通配符(<code>jaspi.mod</code>)
XML	此Module的xml配置文件,前缀 <code>etc/</code> 可有可无
OPTIONAL	可选的依赖Module
FILES	此Module需要的files,比如 <code>deploy.mod</code>

下图为 Module 之间的关系,其中 `→` 表示此Module依赖的Module.



使用的正则说明

- 在 `Module` 类中的 `init()` 方法中,在取得mod文件名称的时候,用了一个看着有点复杂些的正则.

```
1. Pattern pat = Pattern.compile("^.*[/\\\\\\\\]{1}modules[/\\\\\\\\\\\\]{1}(.*).mod$", Pattern.CASE_INSENSITIVE);
```

这个正则的意思就是：

- `^.*` : 字符开头(可以为0个).
- `[/\\\\\\\\]` : `/` 或者 `\\` (windows下)
- `{1}` : 正好出现一次也就是前面的 `/` 或者 `\\` 正好出现一次
- `modules` : 固定出现的字符串
- `(.*)` : 表示一个group,即为文件名称,可以为空.
- `.mod$` : 以.mod结尾

```
1. this.fileRef = mat.group(1).replace('\\\\', '/');
```

这里的 `group(1)` 取得就是 `(.*)` 中的内容,即文件名称.文件名可以为空,也就是允许 `.mod` 这样的文件.不过文件名为空,也就是Module的名称为空,这样是否太宽松了?

- 在 `process()` 方法中.

```
1. Pattern section = Pattern.compile("\\s*\\\\([\\^]*\\\\)\\\\s*");
```

这个正则表达的意思是：

- `\\s*` : 0个或者多个空白
 - `\\\\[` : 对`[`进行转义
 - `([\\^]*)` : 一个group, 第一个`[`和最后一个`]`之间的`\\^`表示匹配除了`]`外的字符.
- 所以也就是要匹配 `[selectionType]` 这样的字符串格式.

- 在 `Props` 类中的 `expand` 方法中有这样一个正则：

```
1. Pattern pat = Pattern.compile("(?<=[^$]|^)(\\\\$\\\\{[\\^]*\\\\})");
```

这个正则稍微复杂些,看到 `<=`, 你可能会感到有些奇怪了.这里涉及到的是正则表达式中的 **反向预搜索**(貌似为知笔

记有 bug, 下面的某些字符不知道是不是特殊字符的原因会导致保存之后内容排版有问题, 只能将其放到 `` 块中)

1. 这里不要将 `(?<=[^$]|^)` 认为是 `group(1)`. `(\\$\\{[^}]*\\})` 才是 `group(1)`.

先看简单的后半部分, 这个比较好理解:

1. `(\\$\\{[^}]*\\})` 中 `\\$` 对 `$` 转义, `\\{` 对 `{` 转义, `[^]*`, 匹配 `}` 外的字符串(0或多个).

再看前半部分(需要了解正则中的 `正向预搜索` 和 `反向预搜索`):

1. `[^$]`: 这个很好理解, 除了 `$` 之外的字符(但必须至少有1个).
- 2.
3. `|^`: `|` 表示或, `^`, 单独的 `^` 在这里确实让人有点摸不着头脑, 不过可以想一下 `^` 的作用, 一般为两个: 1) 开头. 2) 除什么之外.
- 4.
5. 这里就得结合后面的正则, 最终表达的意思也就是以 **"后半部分匹配的字符串开头"**(这个确实有点绕人, 有可能认为这也许就是表示的字符 `^` 吧, 但请想一下, 前面的 `^$` 已经表明了除了 `$` 之外的字符, 已经包含了 `^`, 所以这里肯定不是表示 `^` 字符的意思. 而且这个是特殊字符, 至少也得转义才行. 这样也只剩下了前面所说的两种意思, 但是2)似乎也不符合这种情况, 因为这个一般用在 `[^x]` 中, 这里明显不是, 所以也只剩下了1)这种情况). 同时在后面我们可以看到, `${property}` 只能出现一次.
- 6.
7. 以 `npn/npn-${java.version}` 举例, 这个字符串. 匹配的是 `${java.version}`. 如果是字符串 `${java.version}`, 那么也可以匹配, 因为它以 `${java.version}` 开头. 如果说上面的正则中我们将 `|^` 去掉的话, `npn/npn-${java.version}` 是可以匹配到的, 但 `${java.version}` 是无法匹配的.

可以写个简单的程序测试下:

```
1. @Test
2. public void testExpandRegex() {
3.     String text;
4.     String patternStr = "(?<=[^$]|^)(\\$\\{[^}]*\\})";
5.     Pattern pattern = Pattern.compile(patternStr);
6.
7.     text = "npn/npn-${java.version}";
8.     this.match(pattern, text);
9.
10.    text = "${java.version}";
11.    this.match(pattern, text);
12. }
13.
14. @Test
15. public void testExpandRegex2() {
16.     String text;
17.     // 去掉了 |^
18.     String patternStr = "(?<=[^$])(\\$\\{[^}]*\\})";
19.     Pattern pattern = Pattern.compile(patternStr);
20.
21.     text = "npn/npn-${java.version}";
22.     this.match(pattern, text);
23.
24.     text = "${java.version}";
25.     this.match(pattern, text);
26. }
27.
28. private void match(Pattern pattern, String text) {
29.     Matcher matcher = pattern.matcher(text);
30. }
```

```

31.     System.out.println("\r\n" + text);
32.     while(matcher.find()) {
33.         System.out.println("group(1): " + matcher.group(1));
34.     }
35.     System.out.println("-----");
36. }

```

同时在此方法的 `while` 循环中我们可以看到 `${property}` 这样的值是不能出现两次的。

Jetty 对 `modules/npn` 下的诸如 `npn-1.7.0_9.mod` (名字最后一个数字小于10)文件名定义的有些问题,在我的机器上安装的 `jdk-7u9-windows-x64.exe`,最后JDK的版本显示是 `1.7.0_09`,也就是对于个位数没有进行补零。

这里插个题外话,如果在命令行下运行 Jetty 二进制包出现此问题的情况比如下图的错误:

```

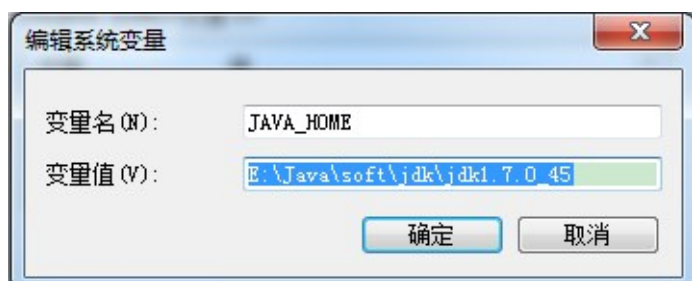
E:\Java\soft\jetty-distribution-9.1.1.v20140108>java -jar start.jar
java.io.IOException: Cannot read file: modules\npn\npn-1.8.0_20.mod
    at org.eclipse.jetty.start.Modules.registerModule(Modules.java:405)
    at org.eclipse.jetty.start.Modules.registerAll(Modules.java:395)
    at org.eclipse.jetty.start.Main.processCommandLine(Main.java:561)
    at org.eclipse.jetty.start.Main.main(Main.java:102)

Usage: java -jar start.jar [options] [properties] [configs]
       java -jar start.jar --help # for more information

E:\Java\soft\jetty-distribution-9.1.1.v20140108>java -version
java version "1.8.0_20"
Java(TM) SE Runtime Environment (build 1.8.0_20-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)

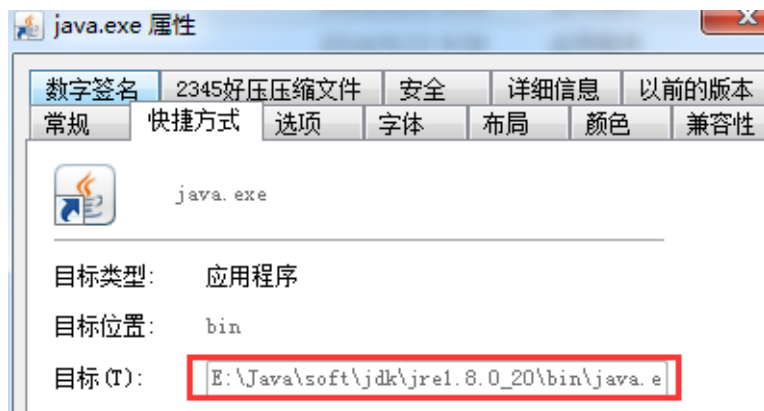
```

这个错误很容易看出来,此版本的 Jetty 还并未提供 `npn-1.8.0_20.mod`,但是上面使用 `java -version` 显示的版本是 `1.8.0_20`,这个和环境变量设置的并不一致啊.What?



简单说明下,如果是机器上有多个 Java 7 及之前的版本,那么安装完以后,在 `C:\Windows\System32` 目录下会有 `java.exe` 文件,此时如果在命令行中运行 `java` 命令,其实就是运行的此 `exe` 文件,为了防止环境变量中的 `JAVA_HOME` 和在命令下运行 `java -version` 出现不一致的情况,最好还是将 `C:\Windows\System32` 文件删除掉,但是如果安装了 Java 8,那么情况稍有不同,安装完 Java 8 之后会产生一个 `C:\ProgramData\Oracle\Java\javapath` 目录如下:

本地磁盘 (C:) > ProgramData > Oracle > Java > javapath		
工具(T) 帮助(H)		
共享 刻录 新建文件夹		
名称	修改日期	类型
java.exe	2014/9/23 9:58	应用程序
javaw.exe	2014/9/23 9:58	应用程序
javaws.exe	2014/9/23 9:58	应用程序



所以这里最好还是将此目录中文件删除或者备份改名,此时即可解决上面出现的问题。

```
C:\Users\yangkun>java -version
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)

C:\Users\yangkun>cd E:\Java\soft\jetty-distribution-9.1.1.v20140108

C:\Users\yangkun>e:

E:\Java\soft\jetty-distribution-9.1.1.v20140108>java -jar start.jar
2014-10-14 16:54:39.744:INFO:oejs.Server:main: jetty-9.1.1.v20140108
2014-10-14 16:54:39.757:INFO:oejdp.ScanningAppProvider:main: Deployment monitor [file:/E:/Java/soft/jetty-distribution-9.1.1.v20140108/webapps/] at interval 1
2014-10-14 16:54:39.780:INFO:oejs.ServerConnector:main: Started ServerConnector@7b2f2f8{HTTP/1.1}{0.0.0.0:8080}
```

激活Module

- modules.buildGraph() : 构建模块图: Module与Module之间的关系类似于父子关系
在此方法中会验证 module 之间没有循环引用.很容易理解也就是两个module之间不能相互引用.比如, annotations.mod 是依赖于 plus.mod 的修改下 plus.mod .

```
1. [depend]
2. server
3. security
4. jndi
5. # 循环引用
6. annotations
```

启动Jetty,肯定会出错,如下图:

```
java.lang.IllegalStateException: A cyclic reference in the modules has been detected: websocket -> annotations -> plus -> annotations
```

Lib & XML Expansion / Resolution

解析模块配置的libs和xmls

- Lib: 通过 org.eclipse.jetty.start.Classpath#addComponent(java.io.File) 添加,最终会添加到 classpath (自定义类加载器) 中.
- XML: 在mod文件中的 xml 中的配置的xml文件可以不用添加 ect 前缀.
- Files: 注册下载操作, 在mod文件中的 [files] 中配置的下载文件的格式为
http://repo.corp.com/maven/corp-security-policy-1.0.jar:lib/corp-security-policy.jar

解析额外的XMLs

看配置的xml文件是否存在

貌似还没全部写完 +_+.