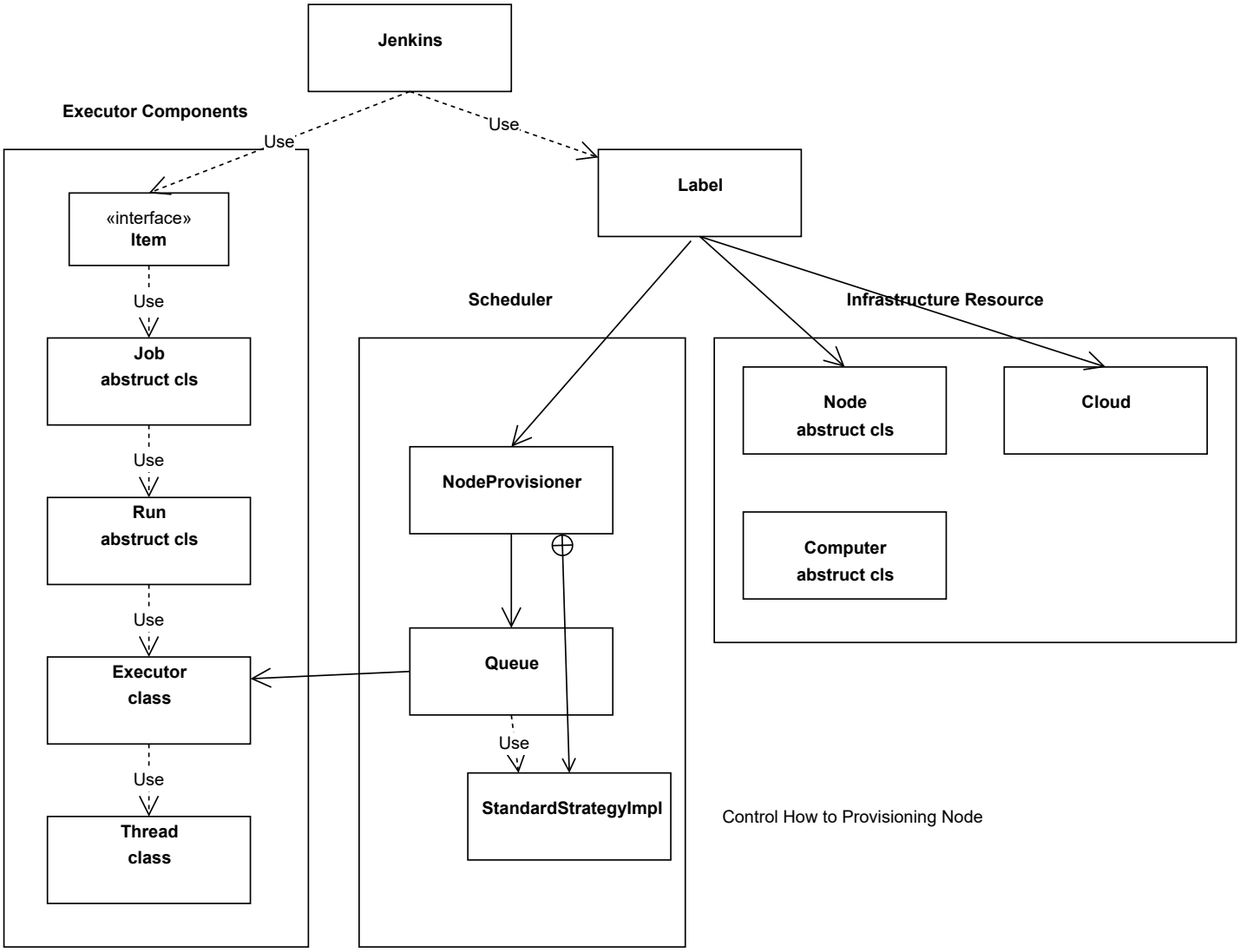
[源码解析] Jenkins 的调度过程以及相关组件的源码解读

Nov 2, 2020 • NagleZh

Jenkins Queue Invoke Chain



本文旨在记录阅读 Jenkins 源码中关于 Queue 调度机制的过程以及结论，希望能够给使用 Jenkins 的用户提供一些基本的信息，进而在大规模使用 Jenkins 的情况下，能够有一些优化的思路。
一家之言，理解不充分的地方希望批判性的去理解。
Jenkins 作为 CI/CD 的开源产品，历史比较悠久，能够存活至今，第一个可以说是社区支持，其次，代码本身肯定也是存在比较可取的优势，从而从多数软件当中脱颖而出，所以，阅读及分析代码，能够给后续设计其他的产品带来很好的思路以及见解。
历史悠久的另外一个点就是，代码量巨大，所以本文只是记录并分析其中调度的一个小部分。更多的是关于各种 class 之间的关系，具体包含以下几点：

1. `Jenkins.java` Jenkins 的主要执行对象，所有的调度，运算，任务都是绑定在该对象上。
2. 运算资源， 或者称之为 Infrastructure 层，提供 CPU， 内存，等运算资源的设备，主要用到 `Label.java`/`Node.java`/`Computer.java`/`Cloud.java`
3. 执行任务， 在 Jenkins 里面，也就是一个一个需要执行的 job，底层用 Thread 来实现，具体用到的文件有 `Item.java`/`Job.java`/`Run.java`/`Executor.java`
4. 调度逻辑， 调度如何分配需要执行的任务与运算资源的关系，主要用到的有 `Queue.java`/`NodeProvisioner.java`/`MultiStageTimeSeries.java`/`TimeSeries.java`

运算资源

在 Jenkins 的配置过程当中，我们都需要添加 Node 对象，而 Node.java & Label.java 主要负责的是基础的配置，Label 这个对象是负责一组 Node， 因为在 Build 的场景当中，一个 job ，可能是 C 写的，也可以是 java 写的，我们在 build 当中，可以通过 Label 来进行区分一组 node，而源代码里也是用的是这种方式， 一个 Label 对象，管理多个 Node 对象，而 Label 对象也引用了 NodeProvisioner 对象（调度过程当中会提到），在该 Label 一旦遇到运算资源不足，就会根据 NodeProvisioner 当中的 StandardStrategyImpl 的 Inner Class 对象进行初始化新的 Node。
以上提到的 Node 对象，主要是用于一个 Node 启动时，需要的一些静态的信息， 比如说，ExecutorNum，NodeName, Description, 也就是在 Jenkins 页面当中配置的时候需要填写的一些信息。
Jenkins 实现了另外一个对象 Computer 来管理计算节点运行时候的信息，比如说获取 offlineCause 不在线原因，getLogFile 执行的 log 文件 ，执行 connect 的操作， 等等。

执行任务

执行任务，在 Jenkins 当中分的层次比较多， 主要包含，Item -> Job -> Run -> Executor -> Thread, 下面咱们一个一个的梳理一下。

- Item： 在 Jenkins 的主页当中，其实新手一般要做的第一件事情就是， Create Item， 这个 Item ，和源码当中的 Item 也是相对应的，一个 Item ，也就是一组 Job，Item 本身也可以是一个 Job。
- Job： 一个 Job 在 Jenkins 当中，就是一个 Build 的所有静态的配置信息，包含的东西譬如，build 的代码，需要被 build 的源代码，每一次 Build 的时间以及需要，Build 的 git repo 等等。
- Run： 上面的 Job 可以总结为一个 Build 的静态配置， 那么 Run 就是每一次需要执行的 Build 运行时需要的管理器，管理的东西有 pre build stage， post build stage, build result 等等。
- Thread: 每一次 Build ，也会包装在一个 java Thread 当中进行执行。

调度逻辑

在诸多运算当中，如何进行调度运算资源，可以说是 Jenkins 里面的一个值得挖掘的点，Jenkins 需要平衡 builds（Executor） 和运算资源之间的关系，保证运算充足的情况下，不去启动多余的 Node，在运算徒增的时候，不会太快，也不会太慢的去启动新的node，这个在被无数人使用的开源产品中，相信也是一个挑战。
Jenkins 有几个类值得一提， Queue.java 中的 Queue， 以及 NodeProvisioner 当中的 NodeProvisioner 和 StandardStrategyImpl 这个 inner class。

- Queue ，顾名思义，也就是需要被执行的任务进行排队的地方，任务的调度，会在 Queue 的这个类的管理下，进行执行。
- NodeProvisioner 则是通过调用 Queue， 来管理下面的 Node 资源。
- 而具体管理的资源，也就是在 StandardStrategyImpl 这个 Inner class 当中被实现。

移动平均值的概念 EMA

在调度过程当中，使用了一个概念叫做 exponential moving average (EMA)，中文译乘移动平均值，移动平均值的概念是作用于需的 Executor 的数量。
一般来说，我们去计算一个过去一段时间需要用到 Executor 的数量，我们可以说平均每秒钟，我们需要使用 10 个，这个运算给予下面这个公式:

```
过去一分钟或者过去一个小时的执行数量 / 时间（秒为单位） = 数量/秒
```

但是这种计算方式的劣势在于，具体时间段内的运算量并不是平均的，可能上一秒数据是 10 个， 下一秒是 100 个，这个很难体现出当前的运算压力。
移动平均值就是为了解决上面的这些问题，所谓移动，就是将当前的运算量与历史的运算量的按一定的比率进行分配。比如说历史的平均值是10， 而当前的值是 100， 我们并不直接的运算 （10+100）/2 = 55. 而是，将当前的运算量的权重提高，引入权重的这样一个概念，比如说是 80%。 那么运算的方式也就变成了 历史*20% + 当前 * 80% 在前面的运算中也就变成了 ` (10*0.2) + (100*0.8) = 82， 也就更能体现出当前的运算压力了。而实现这段代码的逻辑，在 TimeSeries.java` 中进行实现的:

```
public void update(float newData) {
    float data = history[0]*decay + newData*(1-decay);

    float[] r = new float[Math.min(history.length+1,historySize)];
    System.arraycopy(history,0,r,1,Math.min(history.length,r.length-1));
    r[0] = data;
```

```
    history = r;
}
```

基于 EMA 来了解当前的运算使用量压力。在基于 EMA， Jenkins 有几个 概念实现。

1. MultiStageTimeSeries definedExecutors;
2. MultiStageTimeSeries onlineExecutors;
3. MultiStageTimeSeries connectingExecutors;
4. MultiStageTimeSeries busyExecutors;
5. MultiStageTimeSeries idleExecutors;
6. MultiStageTimeSeries availableExecutors;
7. MultiStageTimeSeries totalExecutors;

上面的几个值，也就是基于 EMA 定义的一些执行者 executor 的数量的变化处理者。而我们用来针对性的去启动新 agent 的, 是第2个，也就是 onlineExecutors. 这个也就是当前能够执行的 Executor 的数量。

那么最后计算出来的值，也就是当前需要的 Executor 的数量，最后会应用到 node / agent 生成中。

MARGIN MARGIN0 MARGIN_DECAY 参数

我们的所有的 Executor 的数量都是基于 EMA 算法来进行运算的，所有存在这样一种情况 —— 即使当前的 workload 的需求量为 1，那么 1 可能只会占去 %80 ，也就是 0.8 ，历史有可能是0， 那么，当前的 workload 只会是 0.8。 处于这种情况， Jenkins 里面实现了一个叫做 MARGIN / MARGIN0 / MARGIN_DECAY (default: 0.1/0.5/0.5)的这几个参数。
通过下面代码里面的运算，如果 workload 的需求大于 1-M 的时候（excessworload 越小，M越大）。

```
private float calcThresholdMargin(int totalSnapshot) {
    float f = (float) (MARGIN + (MARGIN0 - MARGIN) * Math.pow(MARGIN_DECAY, totalSnapshot));
    // defensively ensure that the threshold margin is in (0,1)
    f = Math.max(f, 0);
    f = Math.min(f, 1);
    return f;
}
```

totalSnapshot = 1 ; f = 0.1 + 0.4 * 0.5^1 = 1
totalSnapshot = 2 ; f = 0.1 + 0.4 * 0.5^2 = 0.75
totalSnapshot = 3 ; f = 0.1 + 0.4 * 0.5^3 = 0.625
totalSnapshot = 100; f = 0.1 + 0.4 * 0.5^100 = 0.5

总结

本文从运算资源的实现， 执行任务实现，以及调度逻辑方面，粗略的分析了 Jenkins 管理 CI 的的方式方法。更多的专注点在调用链以及几个基本的类的定义与解释。
收获方面，其一是学习到了里面的一些调度设计， 另外，在代码的整洁方面也有很大的启发， 比如说，注释方面，代码里面就会细致的区分 /**/ 与 // 的用法，这些规范让人读起来很舒服。
所以自己后面在 coding 的过程当中，也会尽力的去做到整洁，规范。
细节不多，目前对 Jenkins 实现的层次结构暂时并没有很好的去分析， 比如说配置管理，文件管理，如何分层以及各个层次之间的关系。
工作中因为经常的需要使用到 Jenkins ， 所以希望后面会再实践一些分析与总结。

Nagle's Home

- Nagle's Home
- zhang.nlage@gmail.com

- NagleZhang

Dream to be a hacker, a emacs lover.