به نام خدا

**محمدجواد ماهرالنقش – 99105691**

گزارش پیاده سازی فاز 1 پروژه سیستم های بی درنگ

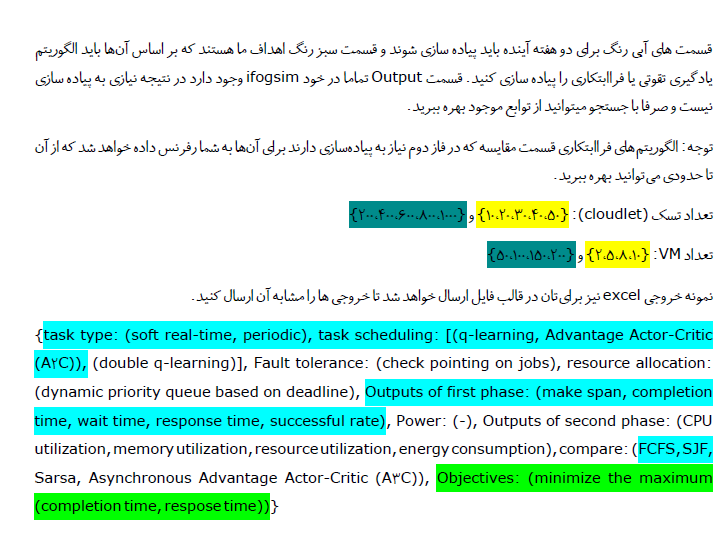
**مقدمه**

در این پروژه قصد داریم به کمک شبیه ساز iFogSim، چند الگوریتم زمانبندی را بر روی شبیه ساز iFogSim پیاده سازی کنیم.

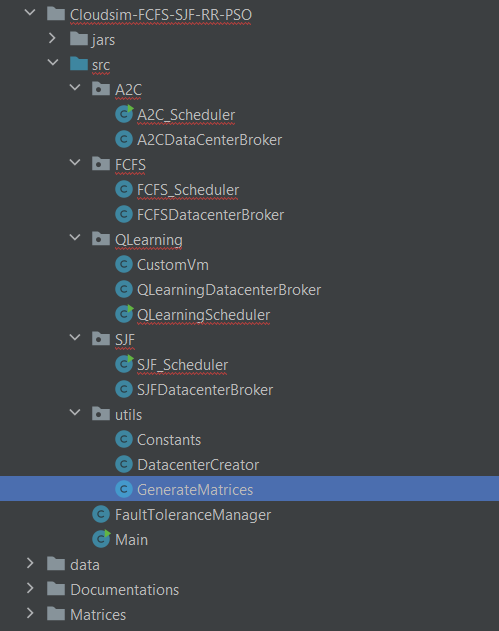
الگوریتم های FCFS، SJF، A2C و QLearning نیاز است تا در این شبیه ساز پیاده سازی شوند.

سپس این شبیه سازی به ازای تعداد مقادیر مختلف مربوط به VM و Task انجام شود.

**صورت پروژه**



**ساختار کلی پروژه**



ساختار کلی پروژه، مطابق با تصویری است که میبینید. یک فایل اصلی Main وجود دارد که الگوریتم ها را صدا زده و هر یک از 4 الگوریتم در پوشه ای جداگانه پیاده سازی شده اند. سپس هر یک از این سیاست های پیاده سازی به ازای تعداد VM و تعداد Task های مختلف اجرا شده و نتایج خروجی در فایل هایی جداگانه ذخیره می‌شوند.

در ادامه به بررسی دقیق‌تر هر یک از توابع در پروژه می‌پردازیم.

* با توجه به یکسان یا مشابه بودن بسیاری از توابع، صرفا یکی از آنها به عنوان نمونه توضیح داده میشود.
* در نسخه‌های بعدی می‌توان نسبت به واحدسازی کدهای مشابه اقدام کرد، گرچه برخی از این توابع در همین نسخه نیز واحدسازی شده‌اند.
* تقریبا همه توابع دارای چند خط کامنت و توضیحات کلی هستند که نام تابع، کارکرد تابع، ورودی (ها) و خروجی (های) تابع را نشان می‌دهد.

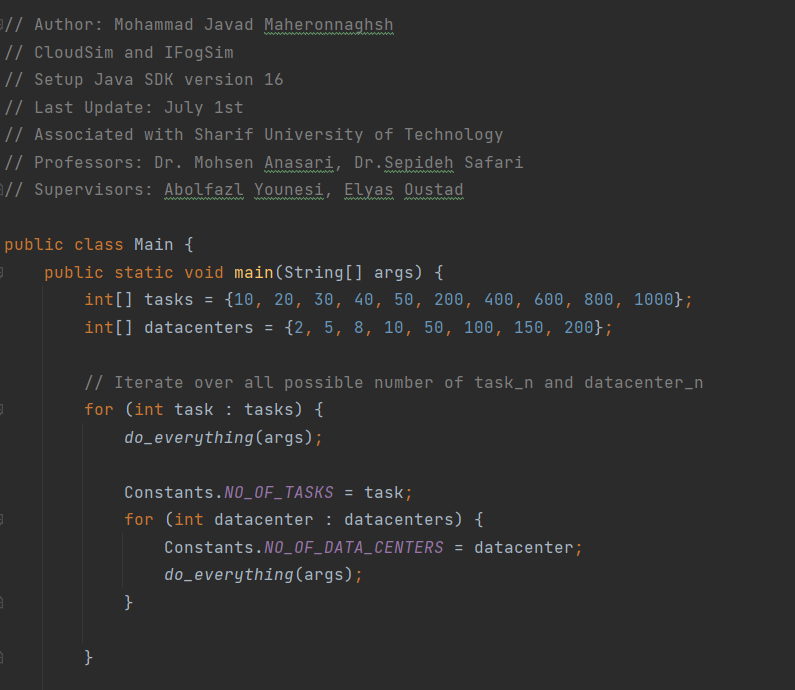
**بررسی پیاده‌سازی**

در ادامه به بررسی توایع و کدهای پیاده‌سازی‌شده می‌پردازیم.

* **تابع main در کلاس Main**

همانطور که در تصویر پایین میبینید، به ازای تعداد VM ها و Task های خواسته شده، هر بار do\_everything صدا زده شده است.

مجموعا 10\*8=80 بار این تابع صدا زده می‌شود.

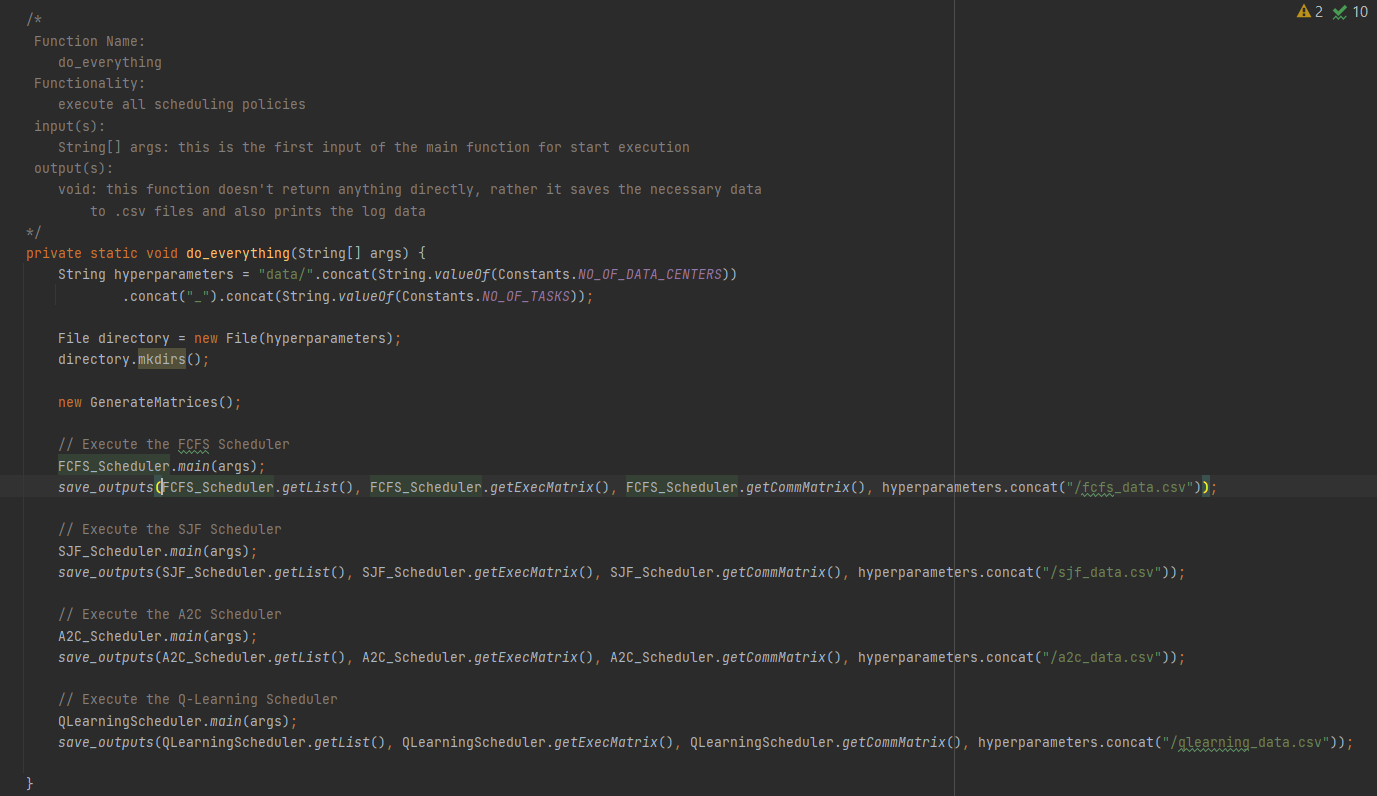


* **تابع do\_everything در کلاس Main**

همانطور که در تصویر پایین میبینیم، ابتدا به ازای ابرپارامترهای فعلی (تعداد datacenter ها و تعداد cloudlet ها)، پوشه ای با همین نام ایجاد میکنیم.

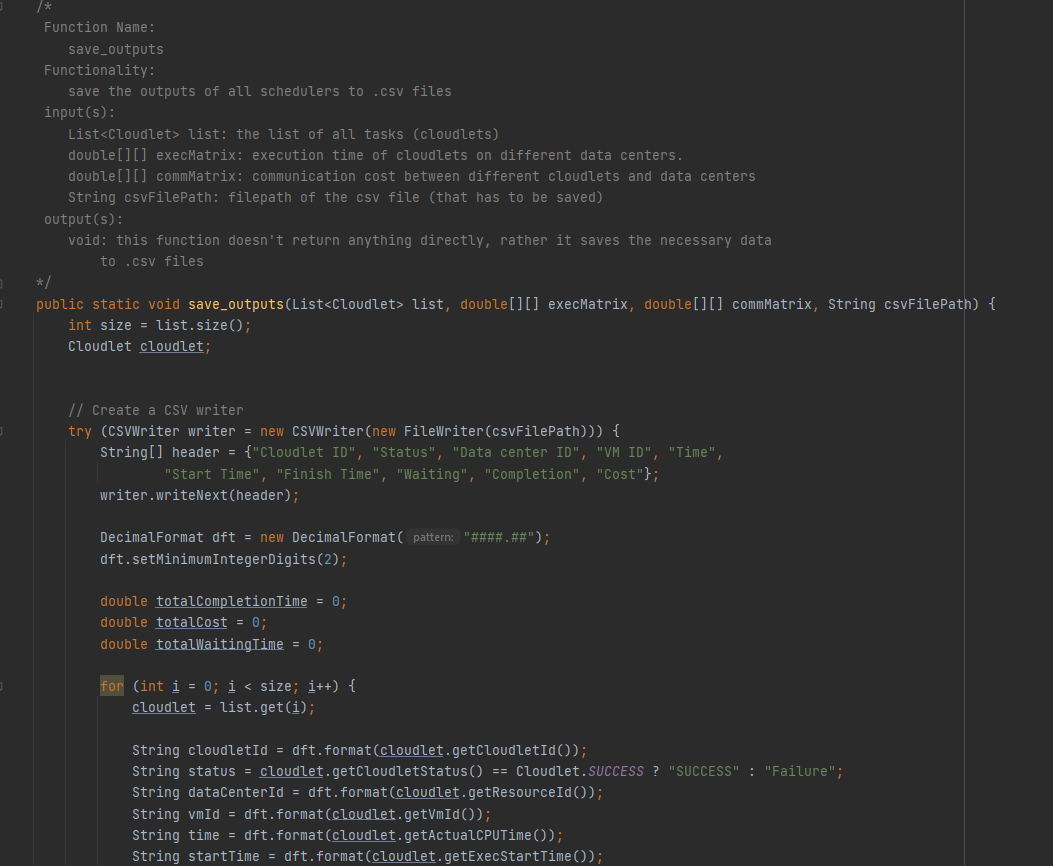
سپس GenerateMatrices را صدا زده تا دو فایل مربوط به commMatrix و execMatrix را ایجاد کند (هر یک از این ماتریس ها در پوشه ای متناسب با ابرپارامترها ذخیره میشود).

سپس 4 الگوریتم زمانبندی بر روی اینها اجرا میشوند. و هر بار پس از اجرای الگوریتم ها، در پوشه ای متناسب با ابرپارامترهای فعلی، خروجی ها و ریزالت ها ذخیره میشوند.



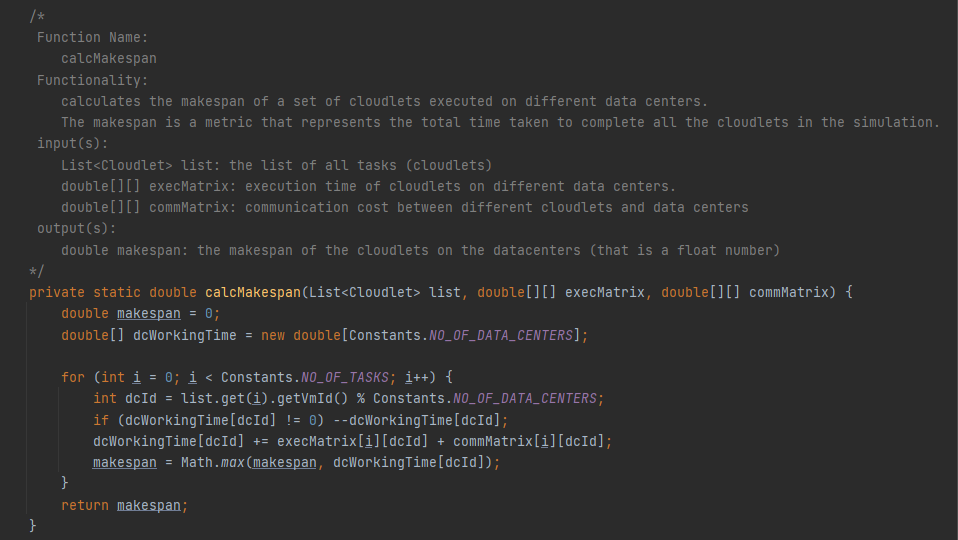
* **تابع save\_outputs در کلاس Main**

وظیفه این تابع، ذخیره خروجی های خواسته شده در فایل های csv متناظر است.



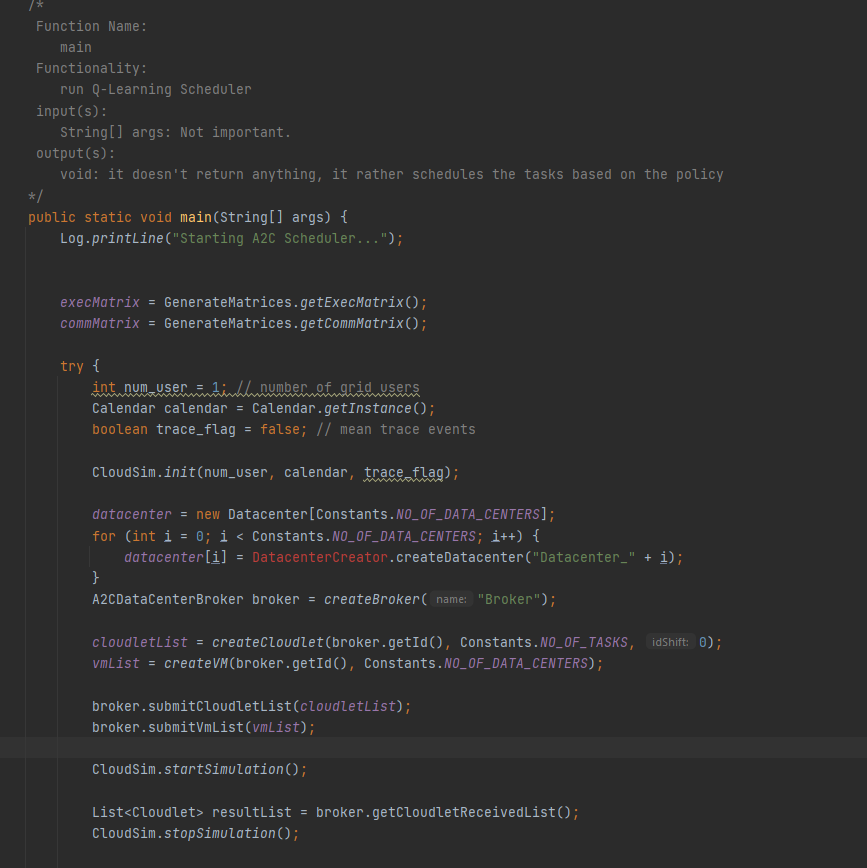
* **تابع calcMakespan در کلاس Main**

وظیفه این تابع، محاسبه مقدار عددی پارامتر makespan بوده که از مواردی است که باید در خروجی ظاهر شوند.



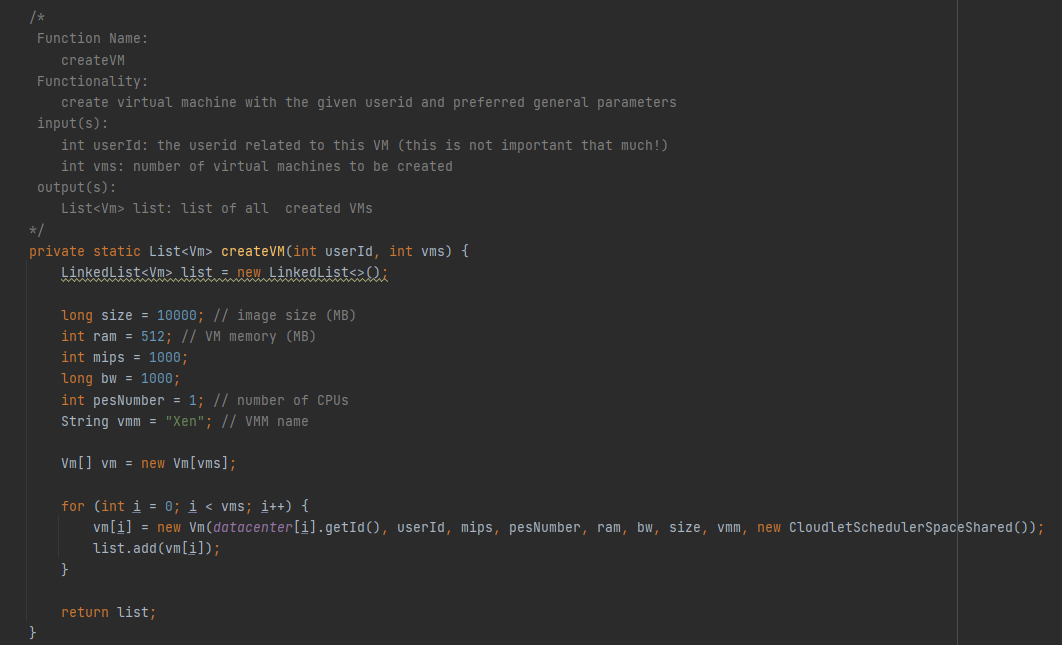
* **تابع main در کلاس A2C\_Scheduler**

در اینجا پیاده سازی تابع اصلی مربوط به زمانبند A2C را مشاهده میکنیم.



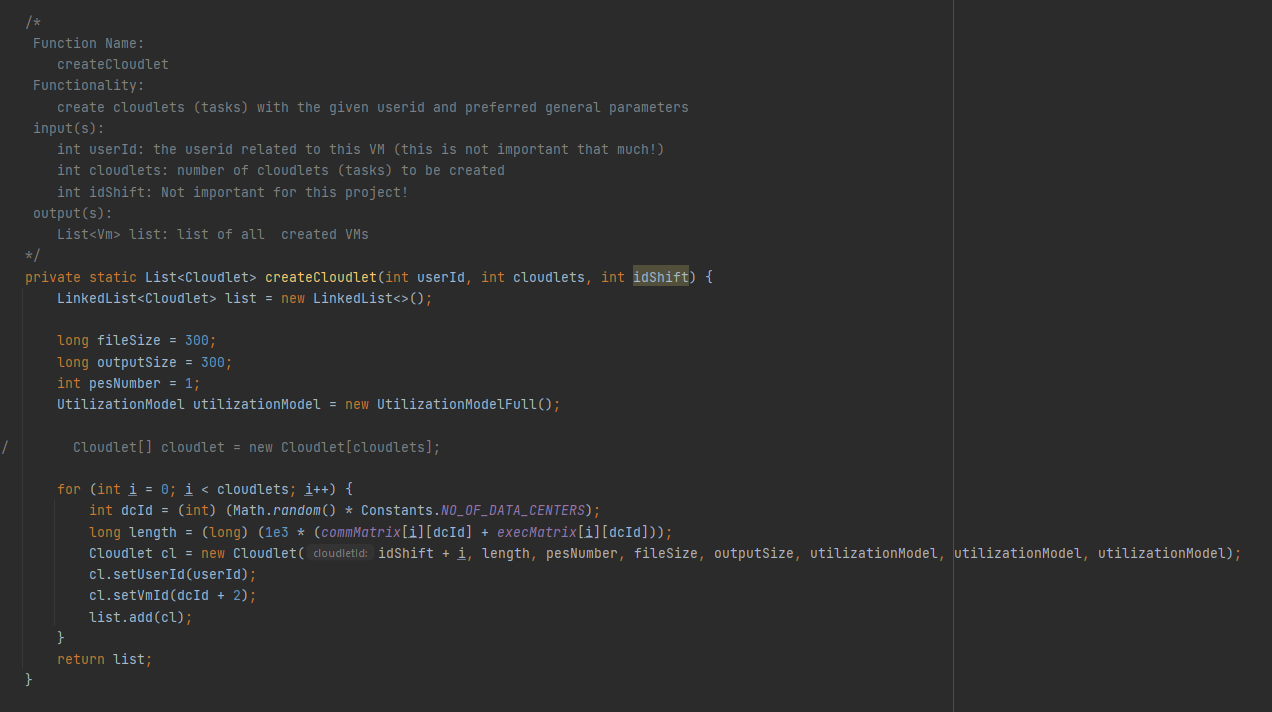
* **تابع createVM**

وظیفه این تابع، ایجاد ماشین های مجازی است. به عنوان پارامتر ورودی؛ تعداد ماشین های مجازی ای که باید تولید کند را گرفته و لیست ماشین های مجازی تولید شده را برمیگرداند.



* **تابع createCloudlet**

وظیفه این تابع، ایجاد cloudlet یا همان تسک، به تعدادی است که در پارامتر ورودی میگیرد.



* **سایر توابع پایه در کلاس های Scheduler**

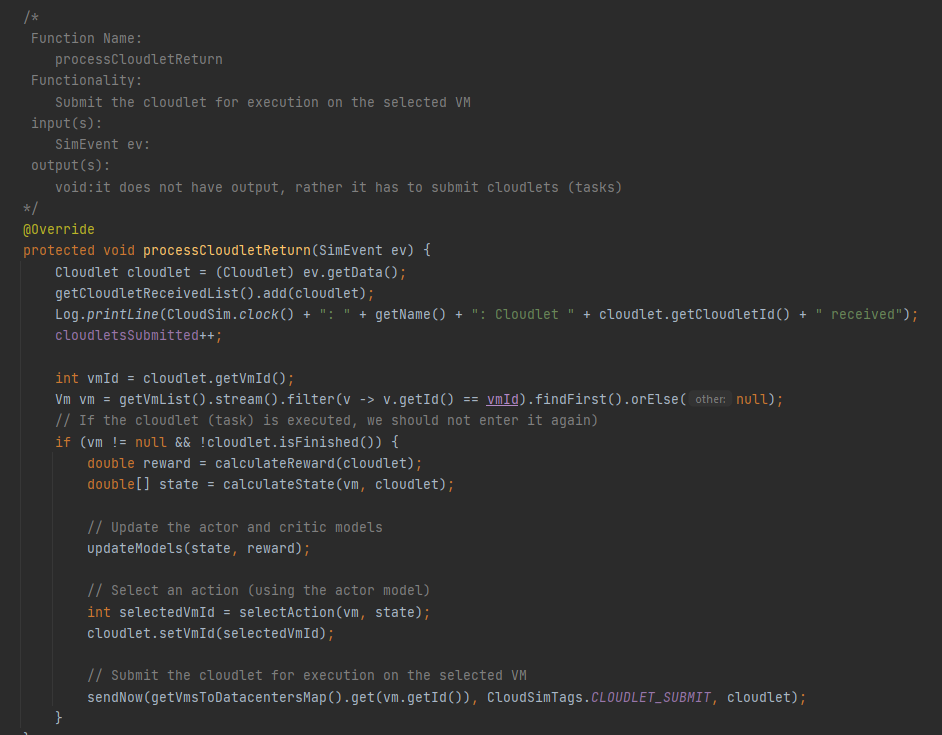
اینجا نیز 4 تابع پایه که در همه کلاس های Scheduler استفاده شده اند را میبینید. تابع اول، Broker مربوطه را ایجاد میکند، تابع دوم، لیست cloudlet ها را برمیگرداند. تابع سوم و چهارم نیز ماتریس هایی که از کلاس GenerateMatrices ایجاد شده اند را برمیگرداند.

* توجه: در به‌روزرسانی‌های بعدی، میتوان دو تابع آخر را حذف کرد و آنها را از کلاس GenerateMatrices دریافت کرد.



* **تابع processCloudletReturn در کلاس های DatacenterBroker**

این توابع وظیفه override کردن این تابع از کلاس پدر DatacenterBroker را دارند. ابتدا cloudlet را به getCloudletReceivedList اضافه کرده و سپس پردازش های مربوط به این الگوریتم را بر روی آن انجام میدهد.



* **توضیحات مربوط به A2C**

به طور کلی هر استیت به کمک 3 معیار مشخص میشوند:

1. مقدار mips مربوط به vm
2. زمان اجرای cloudlet
3. اندازه فایل cloudlet

البته این فیچرها میتوانند کم و زیاد شوند و در صورت تمایل میتوانیم آنها را در تابع calculateState تغییر دهیم.

همچنین نیاز به تعیین پاداش یا همان reward داریم که به صورت تعیین شده است. البته میتوانیم این تابع پاداش را به صورت های دیگری نیز تغییر دهیم و عوامل دیگری را نیز در نظر بگیریم (تغییرات لازم در تابع calculateReward میتوانند انجام شوند).

اکشن ها نیز به صورت vmId تعریف شده اند، یعنی اکشن وقتی x باشد یعنی vm شماره x را به این cloudlet اختصاص دهیم.

همچنین در A2C باید دومدل actor و critic داشته باشیم که میتوانند به صورت های مختلفی مثل hashmap، Q-Table و Neural Network باشند. میتوانیم مدل های دیگری از آنها را نیز برای پروژه های مشابه تست کنیم (برای این کار، باید همه جاهایی که از criticModel یا actorModel استفاده شده را به روزرسانی کنیم.

در تابع updateModels، در بخشی نیاز داریم که مقدار ارزش استیت بعدی را تخمین بزنیم. با توجه به اینکه نمیدانیم استیت بعدی چیست، به ازای تمام استیت های بعدی ممکن، میانگین میگیریم. این سیاست را نیز میتوان در این تابع (در صورت نیاز) تغییر داد.

* **توضیحات مربوط به Q-Learning**

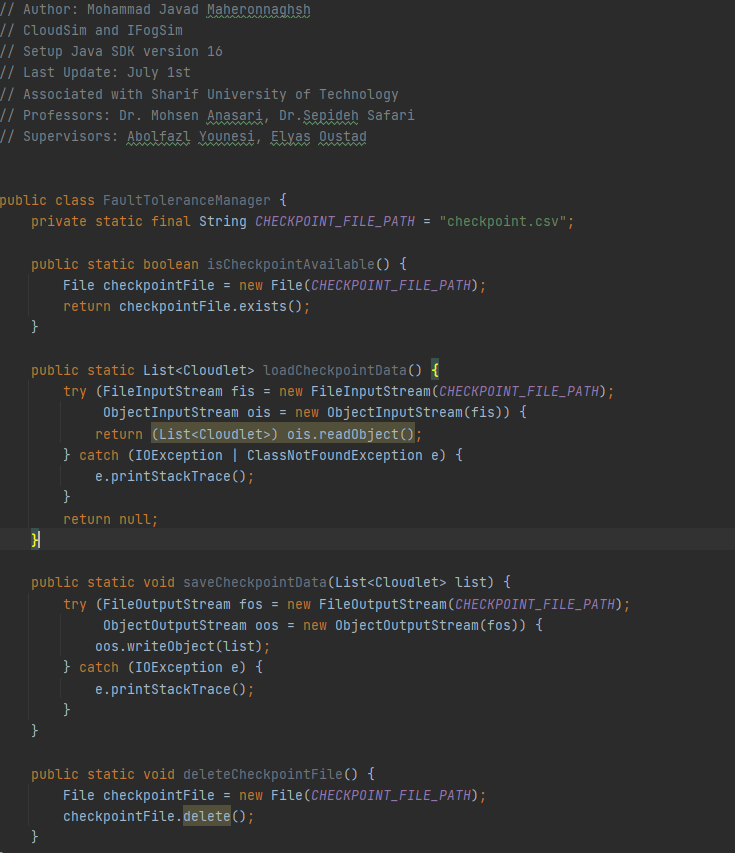
پیاده سازی مطابق با الگوریتم مرسوم Q-Learning است. استیت های ما برابر با تسک های ما میشوند و اکشن های ما نیز تسک های بعدی میشوند.

رد صورت تمایل میتوانیم اکشن ها و استیت ها را با تغییر کلاس broker مربوط به آن تغییر دهیم.

* **توضیحات مربوط به Fault Tolerance و Check Pointing**

در تصویر پایین، پیاده سازی کلی مربوط به آن را میبینید که سعی شده کلیت CheckPointing پیاده سازی شود.

البته این صرفا یک ProtoTypeاولیه است و از آن در کد استفاده نشده است و نیاز به تکمیل دارد.



**خروجی ها**

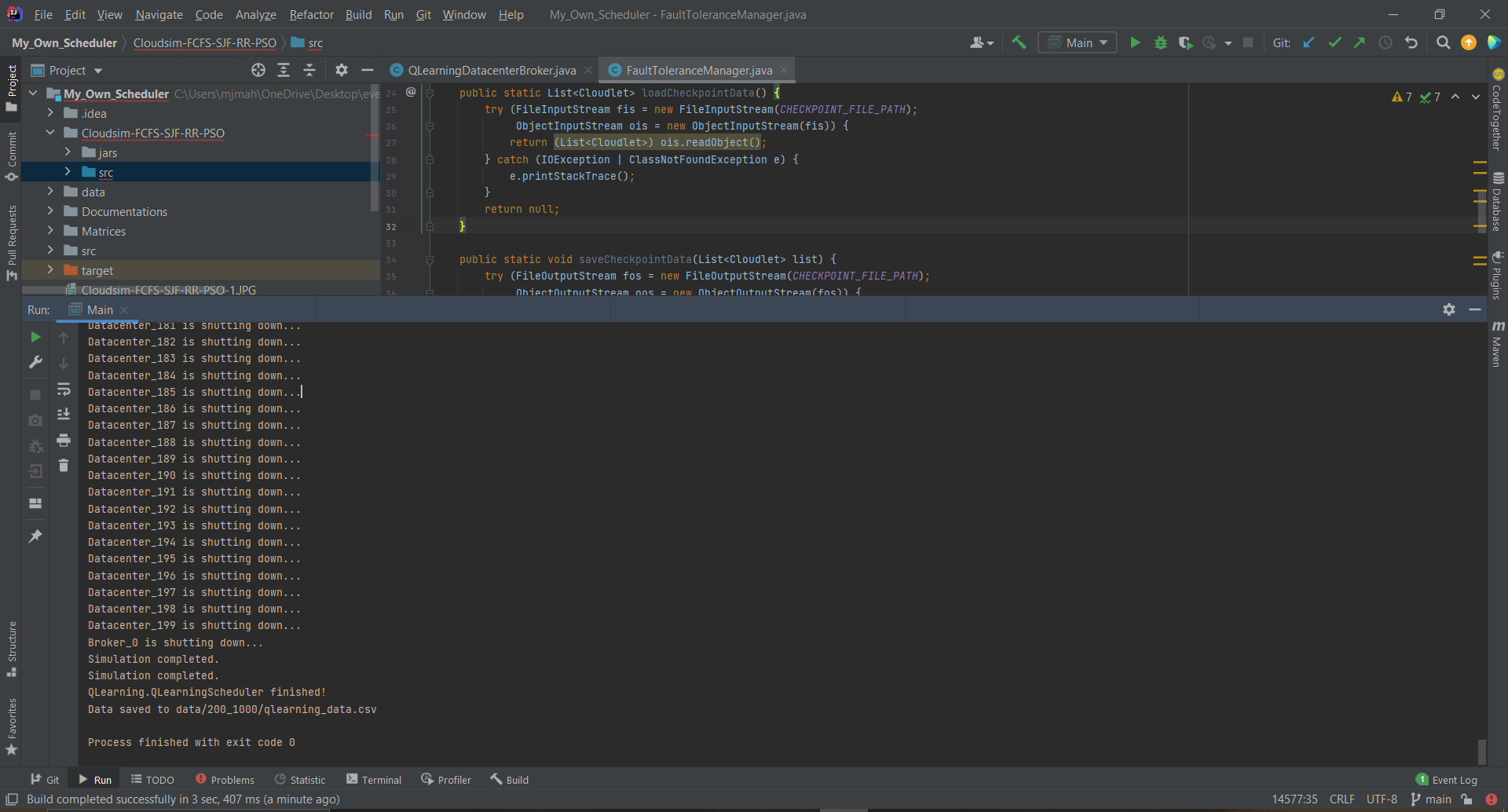
فایل های خروجی در پوشه ذخیره میشوند. میتوانید نمونه اجراهای قبلی را ببینید.

به ازای x تا Datacenter و y تا Cloudlet میتوانیم پوشه data/x\_y را بررسی کنید که در آن 4 فایل خروجی قرار داده شده است که هر کدام مربوط به یکی از 4 سیاست زمانبندی است.

همچنین commMatrix و execMatrix متناظر با اینها را نیز میتوان در پوشه Matrices دید.

**نمونه اجرا**

نمونه ای از اجرای این کد را میبینید.



در سیستم محلی ممن حدود35 ثانیه شبیه سازی و اجرای کامل این کد زمان برد. (رم16 – Windows 10).

برای بهبود زمان اجرا میتواند log ها را پرینت نکرد.

**منابع**

می توانید به سورس کد این پروژه از طریق [این لینک](https://github.com/mjmaher987/Reinforcement-Learning-Based-Fault-Tolerant-Scheduling) دسترسی داشته باشید.

البته با توجه به خصوصی بودن این ریپازیتوری، سورس کد مربوطه در فایلی کنار آن بارگذاری شده است.

با تشکر.