一 cloudstack api 调度流程

我们发往 cloudstack 的 api 命令由 management 端的 ApiServlet 的 processRequest(req, resp)处理，该函数开启一个线程进行处理。对于 login 和 logout 命令单独处理，其它命令发往 ApiServer 组件进行处理

```
final String response = _apiServer.handleRequest(params, responseType,
auditTrailSb);
```

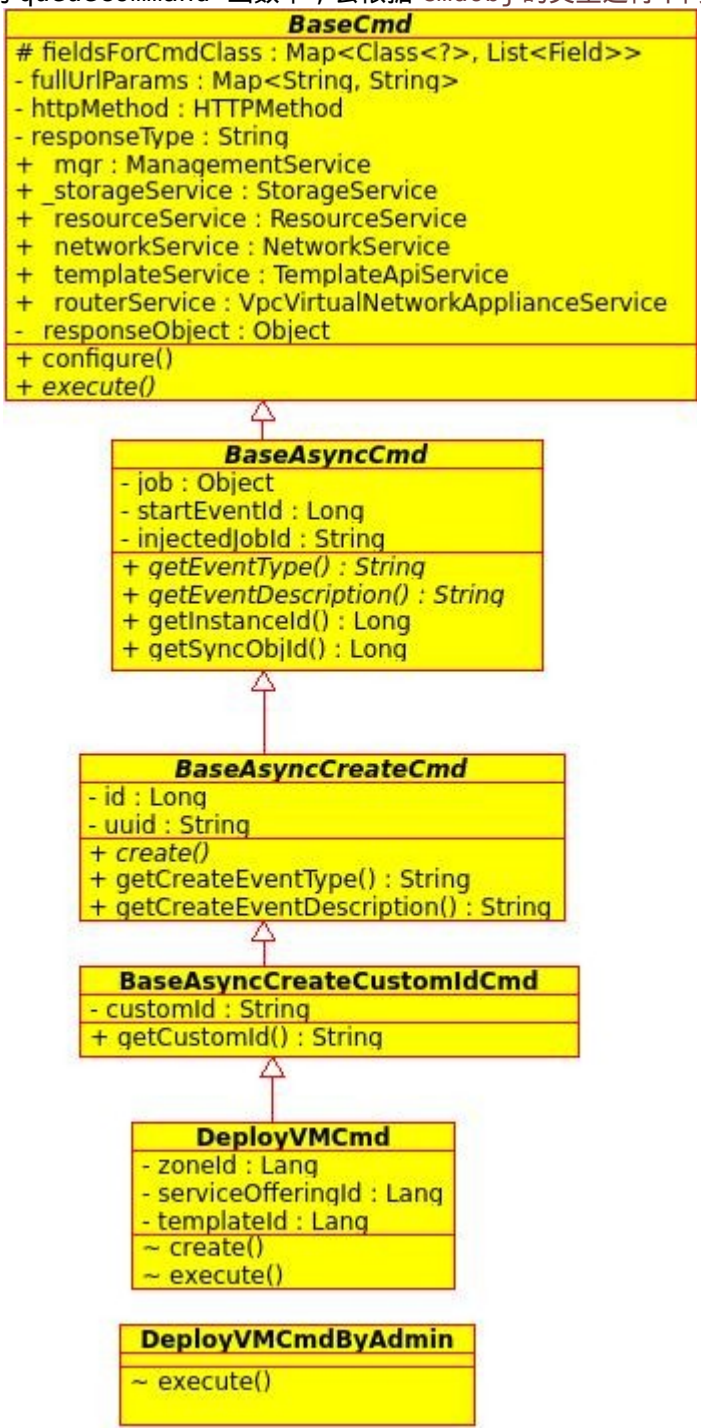
在 ApiServer 组件的 handleRequest 函数中，通过解析参数 params，构建出 cmdObj 对象，剩余参数存入 paramMap，之后进入命令的调度入口函数 queueCommand：

```
// This is where the command is either serialized, or directly dispatched
                response = queueCommand(cmdObj, paramMap);
```

在 ApiServer 组件的 queueCommand 函数中，会依据 cmdObj 的类型进行不同的调度路径

```
// Queue command
based on Cmd super
class:
// BaseCmd: cmd is
dispatched to
ApiDispatcher,
executed,
serialized and
returned.

// BaseAsyncCmd:
cmd is processed
and submitted as an
AsyncJob, job
related info is
serialized and
returned.

//
BaseAsyncCreateCmd:
cmd params are
processed and
create() is called,
then same workflow
as BaseAsyncCmd.
```

下面给出 DeployVMCmd 创建 VM 命令的类继承结构



在 queueCommand 函数中，调用逻辑关键代码如下：

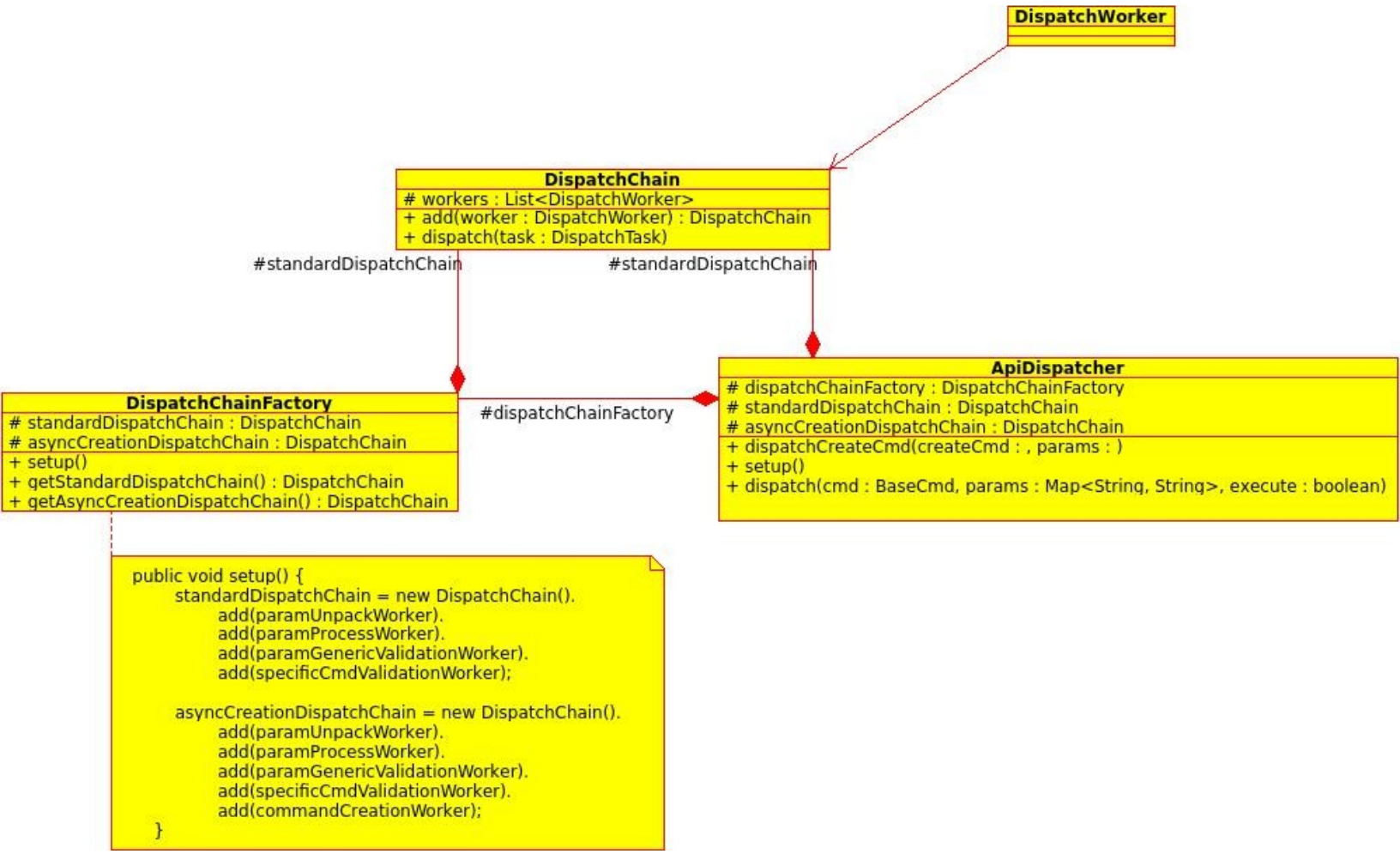
```
if (cmdObj instanceof BaseAsyncCmd) {

      if (cmdObj instanceof BaseAsyncCreateCmd) {

             _dispatcher.dispatchCreateCmd(createCmd, params);
      }
      else {
              dispatchChainFactory.getStandardDispatchChain().dispatch(new
DispatchTask(cmdObj, params));
      }
```

```
AsyncJobVO job = new AsyncJobVO(...);
job.setDispatcher(_asyncDispatcher.getName());
final long jobId = _asyncMgr.submitAsyncJob(job);

return getBaseAsyncResponse(jobId, asyncCmd);

 }
else {

_dispatcher.dispatch(cmdObj, params, false);

return ApiResponseSerializer.toSerializedString(...);
}
```

上面的调度逻辑涉及了 5 个类：

```
ApiDispatcher _dispatcher ,
DispatchChainFactory dispatchChainFactory ,

ApiAsyncJobDispatcher _asyncDispatcher ,
AsyncJobVO job ,
AsyncJobManager _asyncMgr ,
```

ApiDispatcher , DispatchChainFactory2 个类是用于直接调度 cmd 对象。
下面是 ApiDispatcher 的类图


DispatchWorker
DispatchChain
# workers : List<DispatchWorker>
+ add(worker : DispatchWorker) : DispatchChain
+ dispatch(task : DispatchTask)
#standardDispatchChain
#standardDispatchChain
ApiDispatcher
# dispatchChainFactory : DispatchChainFactory
# standardDispatchChain : DispatchChain
# asyncCreationDispatchChain : DispatchChain
+ dispatchCreateCmd(createCmd : , params : )
+ setup()
+ dispatch(cmd : BaseCmd, params : Map<String, String>, execute : boolean)
DispatchChainFactory
# standardDispatchChain : DispatchChain
# asyncCreationDispatchChain : DispatchChain
+ setup()
+ getStandardDispatchChain() : DispatchChain
+ getAsyncCreationDispatchChain() : DispatchChain
#dispatchChainFactory
public void setup() {
standardDispatchChain = new DispatchChain().
add(paramUnpackWorker).
add(paramProcessWorker).
add(paramGenericValidationWorker).
add(specificCmdValidationWorker);
asyncCreationDispatchChain = new DispatchChain().
add(paramUnpackWorker).
add(paramProcessWorker).
add(paramGenericValidationWorker).
add(specificCmdValidationWorker).
add(commandCreationWorker);
}

ApiDispatcher 的 dispatchCreateCmd(createCmd, params)会执行 Cmd 对象 createCmd 的 create()方法。

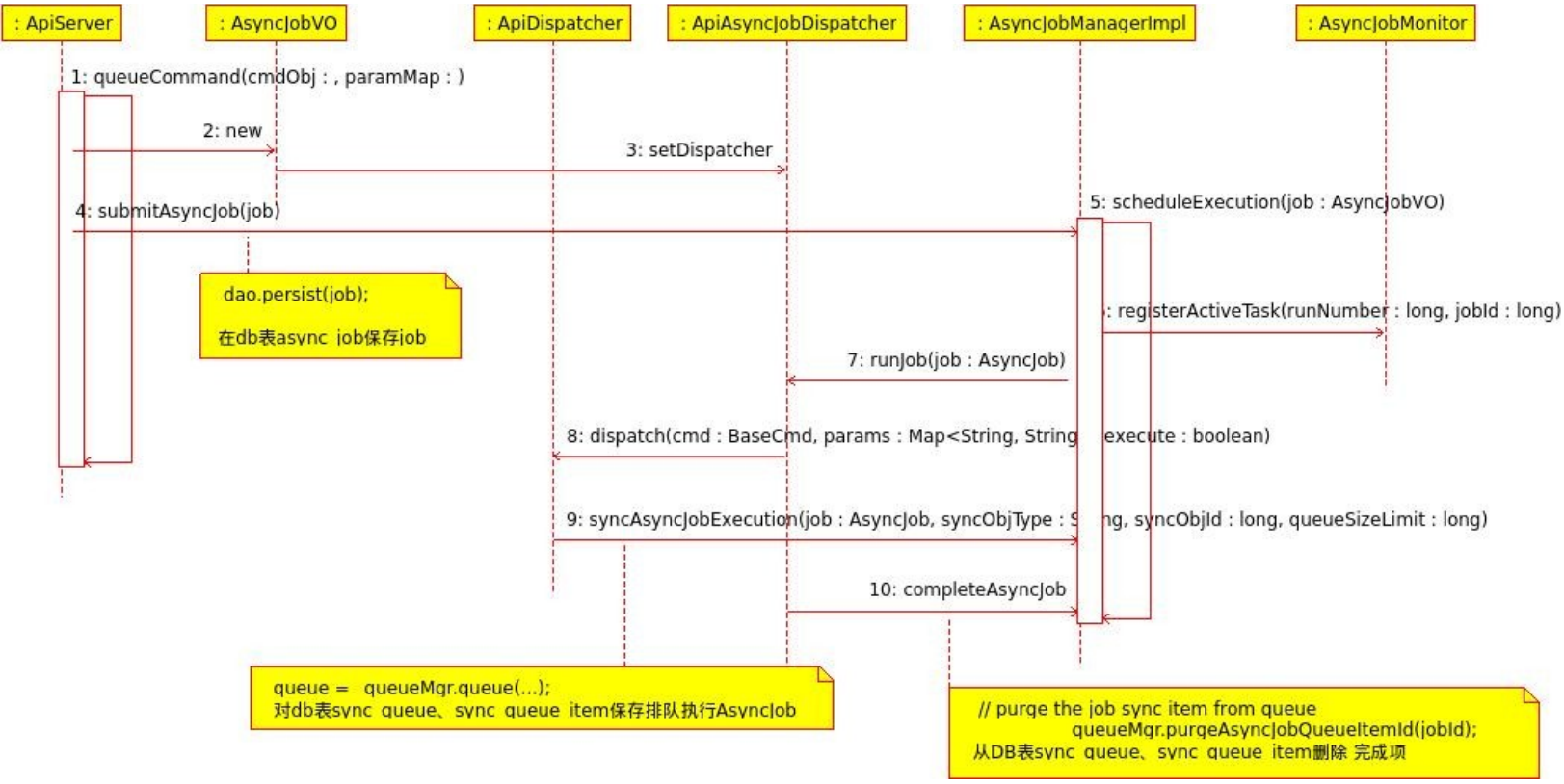
ApiDispatcher 的 dispatch(cmdObj, params, false)会执行 Cmd 对象 cmdObj 的 execute()方法。此外，在执行 cmd 对象方法 execute()前，会对 cmdObj 进行判断

```
if (cmd instanceof BaseAsyncCmd) {

.....

_asyncMgr.syncAsyncJobExecution((AsyncJob)asyncCmd.getJob(),
asyncCmd.getSyncObjType(), asyncCmd.getSyncObjId().longValue(), queueSizeLimit);

.....

}

cmd.execute();
```

异步调度框架的 ApiAsyncJobDispatcher 采用了 ApiDispatcher 的 dispatch 方法来调度 cmd 对象。

ApiAsyncJobDispatcher，AsyncJobVO，AsyncJobManager 3 个类属于异步调度，job 管理。

下面是 AsyncJob 执行流程的顺序图


: ApiServer
: AsyncJobVO
: ApiDispatcher
: ApiAsyncJobDispatcher
: AsyncJobManagerImpl
: AsyncJobMonitor
1: queueCommand(cmdObj : , paramMap : )
2: new
3: setDispatcher
4: submitAsyncJob(job)
5: scheduleExecution(job : AsyncJobVO)
dao.persist(job);
在db表async_job保存job
: registerActiveTask(runNumber : long, jobId : long)
7: runJob(job : AsyncJob)
8: dispatch(cmd : BaseCmd, params : Map<String, String
execute : boolean)
9: syncAsyncJobExecution(job : AsyncJob, syncObjType : S
ng, syncObjId : long, queueSizeLimit : long)
10: completeAsyncJob
queue = queueMgr.queue(...);
对db表sync_queue、sync_queue_item保存排队执行AsyncJob
// purge the job sync item from queue
queueMgr.purgeAsyncJobQueueItemId(jobId);
从DB表sync_queue、sync_queue_item删除 完成项

由_asyncMgr.submitAsyncJob(job)开始，调用 submitAsyncJob(job, false)函数，该函数将 job 信息存入到 db 表 async_job 中，之后
向 messageBus 发行 submit 消息
publishOnEventBus(job, "submit");

调度执行 job
scheduleExecution(job, scheduleJobExecutionInContext);

该函数体如下：

```
 private void scheduleExecution(final AsyncJob job, boolean executeInContext) {
        Runnable runnable = getExecutorRunnable(job);
        if (executeInContext) {
            runnable.run();
        } else {
            if (job.getDispatcher() == null ||
job.getDispatcher().equalsIgnoreCase("ApiAsyncJobDispatcher"))
                _apiJobExecutor.submit(runnable);
            else
                _workerJobExecutor.submit(runnable);
        }
    }
```

传入的 scheduleJobExecutionInContext 为 false，并且 job 的 Dispatcher 为 ApiAsyncJobDispatcher，最终会把 runnable 放入_apiJobExecutor 线程池中运行。

跟入 getExecutorRunnable(job)会跟踪 job 的执行流程
关键代码

```
 _jobMonitor.registerActiveTask(runNumber, job.getId());
```

对 job 执行情况进行监视。

关键代码

```
if ((getAndResetPendingSignals(job) & AsyncJob.Constants.SIGNAL_MASK_WAKEUP) != 0)
{
        AsyncJobDispatcher jobDispatcher = getWakeupDispatcher(job);
        if (jobDispatcher != null) {
                            jobDispatcher.runJob(job);
                        }
        ...
 } else {
        AsyncJobDispatcher jobDispatcher = getDispatcher(job.getDispatcher());
        if (jobDispatcher != null) {
                            jobDispatcher.runJob(job);
                        }
 }
```

job 无论是通过 Signals 被激活还是正常的执行，都会先获取 job 的 jobDispatcher 调度器，执行调度器 jobDispatcher.runJob(job)的 runJob(job)方法。

Api 命令的 job 调度器为 ApiAsyncJobDispatcher，进入 ApiAsyncJobDispatcher 的 runJob(job)方法：
在 ApiAsyncJobDispatcher 的 runJob(final AsyncJob job)方法中，关键代码为

```
.....
Class<?> cmdClass = Class.forName(job.getCmd());
            cmdObj = (BaseAsyncCmd)cmdClass.newInstance();
            cmdObj = ComponentContext.inject(cmdObj);
            cmdObj.configure();
            cmdObj.setJob(job);

.....

// dispatch could ultimately queue the job
_dispatcher.dispatch(cmdObj, params, true);

// serialize this to the async job table
_asyncJobMgr.completeAsyncJob(job.getId(), JobInfo.Status.SUCCEEDED, 0,
ApiSerializerHelper.toSerializedString(cmdObj.getResponseObject()));

......
```

由 job 构建 cmdObj 对象，通过 ApiDispatcher 的 dispatch 方法来调度 cmd 对象，调度执行完成后对 db async job 相关表进行更改。前面提到过，ApiDispatcher 的 dispatch 方法在执行 cmd 对象方法 execute()前，如果 cmd 对象为 BaseAsyncCmd 的子类，会调用 AsyncJobManagerImpl 的

```
syncAsyncJobExecution(AsyncJob job, String syncObjType, long syncObjId, long
queueSizeLimit)
```

方法。

该方法将待执行 job 的信息插入 db 的表 sync_queue 和 sync_queue_item 中。
之后执行 cmd 对象的 execute()方法。

调度完后，通过 AsyncJobManagerImpl 的 completeAsyncJob(...)方法对 db 表 async_job，sync_queue_item 和 sync_queue 修改，完成的 job 从 sync_queue_item 和 sync_queue 删除。