Execution in the Kingdom of Nouns

翻译自Steve Yegge的大作《Execution in the Kingdom of Nouns》原文在 这里

中文翻译：名词王国的死刑

另外第一次翻译，很多地方不准确或根本翻译不出来，见谅~

翻译正文

Hello，world！今天我给大家讲一个关于Java魔鬼国王和他在全国范围内驱逐动词的故事。

注意：这个故事并没有什么圆满结局。如果你心灵脆弱或者吹毛求疵的话，这个故事不适合你。 如果你易于动怒或喜欢在别人的博客上妄加评论， 那么请立即停止阅读。

在我们开始这个故事之前，先让我们熟悉一下背景：

溢出的垃圾

所有使用Java的人都喜欢“用例”，所以让我们以一个用例开始吧：倒垃圾。就像这样：“Johnny，快去倒垃圾，他都快溢出来了！”

如何表达倒垃圾这一活动呢？如果你是一个正常说英语的人，你可以粗略地用以下几句话来描述它

```
在水池下取出垃圾袋
带着垃圾袋去车库
把它扔到垃圾桶里
走回来
洗手
坐回沙发上
继续玩你的电视游戏（或者干其他的事）
```

即使你不用英语思考，你也会想象出一系列类似的动作。 不考虑你选择的语言，或者采取的具体步骤，取决于你采取的行动，倒垃圾是一系列终止于垃圾在外面，你回到屋子里的动作。

我们的思想充斥着各种或勇敢的，或暴躁的，或激昂的动作。 我们生活，我们呼吸，我们走路，我们谈话，我们笑，我们哭，我们希望，我们害怕，我们吃，我们喝， 我们走，我们听，我们倒垃圾。 我们能自由地“做”和“行动”。假如我们只是石头，生活没准还算好，但是我们不会自由。因为我们可以“做”事，所以才会自由。

我们的生活也同时充斥着各种“名词”。我们吃“名词”（食物），我们从商店买“名词”（商品），我们坐在“名词”（凳子）上。 “名词”（石头）可能会忽然砸到你头上，在你的“名词”（头）上弄一个“名词”（大包）。 名词即事物，想想没有了事物我们会怎样？ 但他们仅仅只是事物，比如： 意味着结束或者结束本身，或者一些贵重物品，或者我们周围经常看到的事物的名字。 这是一座建筑，那是一个石头。任何一个小孩子都能指出名

词，仅此而已。 发生在名词身上的 “变化”才是最有趣的事情。

变化需要动作。 动作是生活的调料。 动作甚至给了调料以调料！ 毕竟除非你“吃”它，你是不会感到香这种味道的。 名词也许无处不在，但是生活一直在变并一直有趣的功劳还是在于动词、

当然，除了名词和动词，我们还有形容词，介词，代词，冠词，连词，语气词，和许许多多其他让我们构造有趣语言的词汇。 它们都在语言中扮演着自己的角色，而且每一个都很重要。 如果它们哪一个不存在了的话，那是挺遗憾的事情。

那么，如果有一天我们不再用动词了，你是不是感到很奇怪呢？

在下面我要给大家讲的故事里，这件事情真的发生了……

名词王国

在Java王国中，国王Java以铁腕统治着他的国家，而他子民的思考方式也并不和你我一样。 在这里，你可以看到，名词是十分重要的并直接服从国王的命令。 名词是最重要的居民，它们身穿艳丽的服装显得高贵而优雅，而这些衣服是由形容词提供的。 而形容词哪，也很满意它们的生活，当然，他们不可能像名词那么高贵， 不过相比于动词来讲却幸运得很多。

因为，动词在Java王国里的日子，相当，相当的糟糕。

奉国王Java的法令，动词是隶属于名词的，但他们不仅仅是宠物而已。 或者说连宠物都不是，在整个国家，动词负担起所有的劳力工作。 实际上，他们是王国的奴隶，至少是 农奴或者契约奴之类的。 Java王国的居民对自己的生活还是比较满意的，他们从来没有想到会发生什么变化。

动词负责王国里的所有工作，但是仍然获取不到任何尊重，甚至都不允许单独出来。 如果一个名词被发现在公共场合出现，它会立即被名词逮捕。

当然“逮捕”也是一个动词，他也从不被允许单独行动；你必须创造一个“动词逮捕着”来协助逮捕。 但是“创造”和“协助”哪？这样的话，“创造者”和“协助者”也各自在这个工作上 伴随“创造”和“协助”起到了重要的作用。

国王Java，在他的上帝Sun（现在是Oracle了吧...[译者注]）的指引下，时不时地威胁要将所有动词驱逐出王国。 如果那一天到来了，他们当然需要至少一个动词来做 各种工作，而从国王残忍的幽默感上猜测，这个动词很可能就是“执行”。

动词“执行”(execute)，和它的亲戚“运行”，“开始”，“走你”，“做”，“就这样做” 或者相似的什么词可以通过找到合适的“执行者”来替代任何其他的动词。 想等(wait)一下？ Waiter.execute().刷(brush)牙(teeth)?ToothBrusher(myTeeth).go().扔(take out)垃圾(garbage)?TrashDisposalPlanExecutor.doIt()。 没有任何 一个动词是安全的，一切动词都会被执行的名词而取代。

在这种精神更加泛滥的角落，名词已经把动词驱逐干净。 不仔细看的话，你会觉得仍然有动词存在，比如耕种或倒茶壶，但是一旦仔细观察， 真相便浮出了水面：名词可以随意 命名紧跟在它们后面的动词“执行”，而不改变自身的角色。 所以，当你看到“耕地者”在“耕地”，“倒茶壶者”在“倒”或者说“注册管理者”在“注册”， 你真正看到的是魔鬼国王

Java的 “执行者”大军，只不过他们披着所有者的外衣而已。

在其他王国里的动词

在其他编程语言的王国中，倒垃圾是一件很直白的事情，和我们用英语表述的十分相似。 在Java王国中，数据实体是名词而函数是动词，而在其他王国中却不然：王国的居民是混在一起的， 而且在能顺利完成工作的前提下，只要他们愿意，既可以是名词也可以是动词。

比如在附近的C的领域，JavaScript的地盘，Perl的地盘和Ruby的地盘， 他们可能会把倒垃圾这件事分解成一系列的动作（或者叫做动词或者函数）。 如果他们将这些 动作以适当的顺序应用于适当的事物（拿垃圾，把它带出去，扔到垃圾桶里等等），倒垃圾的任务就圆满成功了。 在这个过程中根本不需要执行者或其他的伴随者这出现。

在这些王国里，真的没有必要创造这么多的包裹器来包裹动词。 他们没有“垃圾倾倒策略”之类的名词，或者“垃圾倾倒地点定位者”来只是你倒垃圾的路径， 也没有“倒完垃圾后的回调”来保证你倒垃圾后回到自己的沙发上。 他们只是写一些动词来操作名词，并创建一个主要的名词，例如,提出垃圾（take_out_garbage()）并把一些需要做的子动作放在里面。

在这些王国中，当需求提升的时候，也通常有一种机制来生成比较重要的名词。 如果这些精明的创造者创造出了一个全新的名词，比如房子，马车，或者耕起地来比人还快的机器， 他们会被给予一个统一的概念：类。 而人们会给类一个名称，一个描述，一些状态和一些操作建议。

这些王国与Java的不同之处在于，动词是允许单独出现的，你没必要创造新的名词去束缚他们。

Java王国的人一种轻视的态度看待他们的邻居；而这也是程序诸王国的现状。

如果你挖个足够深的洞...

在世界的另一边，有一篇贫瘠的居住地。 在那里，动词居民的地位十分之高。 这就是函数式王国，包括Haskellia,Ocamlica,Schemeria和一些其它的国家。 因为附近的国家很少，他们几乎不与Java王国何其附近的国家有接触。 也正因为这样，函数式诸国们相互轻视，并有事没事的时候打一仗以排遣寂寞。

在函数式王国里，名词和动词一般被看做同样等级的居民。 但是，名词，对是名词，基本上整天无所事事。 他们的出现在做事或者执行任务的时候并没有多大意义，因为 活跃的动词们基本把能做的事情都做了。 这里也没有什么奇怪的法律说要创造各种“帮助者”来帮助动词做事，因此在这些王国中，名词的数量和实际上存在事物的数量是相同的。

这样做的结果是，动词在这片土地上为所欲为（请原谅我的用词）。 如果你是一个外来人，你很容易产生名词（函数）是这里最重要的居民的印象。 顺便提一句， 这也是为什么这里被叫做函数式诸国还不是事物诸国的原因。

在最为遥远的地方，远离函数式诸国，存在着一块传说中的土地， “Lamda the Ultimate”(终极lamda？霸气~[译者注])。传说中在那里，没有名词，只有动词。 那里有事

物，但事物由动词组成。如果传说不虚，甚至数字，那里最为流行的货币，也是动词！ 数字0被表示为lamda()，数字1是lamda(lamda())，2是lamda(lamda(lamda())),以此类推。

在这片神奇的土地上，一切事物，别管你是名词，动词，还是其他什么，都是由最基本的动词lamda组成的。

老实说，Java王国中幸福生活着的居民并没有意识到另外一个世界的存在。 你能想象得知此事之后的文化震动么？他们可能会发明一个新的名词（比如“憎恶”）来表达自己新的感受。

Java王国中的居民真的快乐么？

你可能觉得Java王国中的生活有点奇怪，如果糟糕的话还效率还会变得十分低下。 不过，你能从一个地方的童谣中看出他们的幸福程度，而Java王国的童谣，是一群古怪的诗。 比如，这里的儿童经常朗诵的寓言：（这就不翻了[译者注]）

```
For the lack of a nail,
    throw new HorseshoeNailNotFoundException("no nails!");

For the lack of a horseshoe,
    EquestrianDoctor.getLocalInstance().getHorseDispatcher().shoot();

For the lack of a horse,
    RidersGuild.getRiderNotificationSubscriberList().getBroadcaster().run(
      new BroadcastMessage(StableFactory.getNullHorseInstance()));

For the lack of a rider,
    MessageDeliverySubsystem.getLogger().logDeliveryFailure(
      MessageFactory.getAbstractMessageInstance(
        new MessageMedium(MessageType.VERBAL),
        new MessageTransport(MessageTransportType.MOUNTED_RIDER),
        new MessageSessionDestination(BattleManager.getRoutingInfo(
                                        BattleLocation.NEAREST))),
      MessageFailureReasonCode.UNKNOWN_RIDER_FAILURE);

For the lack of a message,
    ((BattleNotificationSender)
      BattleResourceMediator.getMediatorInstance().getResource(
        BattleParticipant.PROXY_PARTICIPANT,
        BattleResource.BATTLE_NOTIFICATION_SENDER)).sendNotification(
          ((BattleNotificationBuilder)
            (BattleResourceMediator.getMediatorInstance().getResource(
            BattleOrganizer.getBattleParticipant(Battle.Participant.GOOD_GUYS),
            BattleResource.BATTLE_NOTIFICATION_BUILDER))).buildNotification(
              BattleOrganizer.getBattleState(BattleResult.BATTLE_LOST),
              BattleManager.getChainOfCommand().getCommandChainNotifier()));

For the lack of a battle,
```

```
    try {
        synchronized(BattleInformationRouterLock.getLockInstance()) {
          BattleInformationRouterLock.getLockInstance().wait();
        }
    } catch (InterruptedException ix) {
      if (BattleSessionManager.getBattleStatus(
           BattleResource.getLocalizedBattleResource(Locale.getDefault()),
           BattleContext.createContext(
             Kingdom.getMasterBattleCoordinatorInstance(
               new TweedleBeetlePuddlePaddleBattle()).populate(
                 RegionManager.getArmpitProvince(Armpit.LEFTMOST)))) ==
          BattleStatus.LOST) {
        if (LOGGER.isLoggable(Level.TOTALLY_SCREWED)) {
          LOGGER.logScrewage(BattleLogger.createBattleLogMessage(
            BattleStatusFormatter.format(BattleStatus.LOST_WAR,
                                         Locale.getDefault())));
        }
      }
    }

For the lack of a war,
    new ServiceExecutionJoinPoint(
      DistributedQueryAnalyzer.forwardQueryResult(
        NotificationSchemaManager.getAbstractSchemaMapper(
          new PublishSubscribeNotificationSchema()).getSchemaProxy().
            executePublishSubscribeQueryPlan(
              NotificationSchema.ALERT,
              new NotificationSchemaPriority(SchemaPriority.MAX_PRIORITY),
              new PublisherMessage(MessageFactory.getAbstractMessage(
                MessageType.WRITTEN,
                new MessageTransport(MessageTransportType.WOUNDED_SURVIVOR),
                new MessageSessionDestination(
                  DestinationManager.getNullDestinationForQueryPlan()))),
              DistributedWarMachine.getPartyRoleManager().getRegisteredParties(
                PartyRoleManager.PARTY_KING ||
                PartyRoleManager.PARTY_GENERAL ||
                PartyRoleManager.PARTY_AMBASSADOR)).getQueryResult(),
        PriorityMessageDispatcher.getPriorityDispatchInstance())).
      waitForService();

All for the lack of a horseshoe nail.
```

直到今天，这仍然是美好的建议。

尽管在Java王国的叙述方式和本.富兰克林的原作大有不同，但是这里的居民觉得他们的重新编排还是有一种不同的魅力在里面。

而最大的魅力在于“架构”，是所有人都能看见的。 架构被国王Java授予了之高无上的地位，因为，架构全部是由名词构成的。 正如我们所知的，在Java王国，名词即事物，事物

的荣耀高于一切的动词。 架构由无数事物构成：你可以看或触摸的事物，给你留下壮观印象的事物，用棍子刮擦发出美妙声音的事物。 国王Java，十分喜欢刮擦的噪音； 每当他新换车夫的时候，踢轮子发出的美妙声音让他觉得很满意。 不管上面的故事有什么瑕疵，“事物”总是不缺少的。

作为人类，我们的第一本能总是寻找由各宗物体构成的庇护。 庇护越坚固，我们感觉越安全。 在Java王国，有很过坚固的东西让居民们感到安心。 他们感慨如此庞大的架构 建造之神奇并认为它是“最为坚固的设计”。 而且每当结构变化时，他们就越坚信这点。 接着，架构的力量也变得强的令人生畏以至于没有人认为可以摧毁他。

除了坚固的架构之外，在Java王国中的所有东西很有调理地组织着：你会发现任何名词都会呆在适当的地方。 这里每个故事都有一个固定的模式：实例构造在故事的表述中占了主要的篇幅， 因为每个抽象都会有一个管理者（Manager）， 而且每个管理者都有一个run()方法。 Java居民们觉得他们可以用这种模型表述任何事情。 这是一种“名词计算”，只要你愿意， 它可以满足任何抽象，任何计算。 你需要的仅仅是足够的名词，名词的构造器，获取器方法，和重要的execute()函数来实现你的计划。

Java王国的居民活的不仅仅是幸福，简直是迸发出强烈的自豪感。

面向对象

StateManager.getConsiderationSetter("Noun Oriented Thinking", State.HARMFUL).run()或者， 正如外面的世界所说，“面向名词的思考是有害的”

面向对象的编程把名词放到首位，但是我们为什么非得把名词捧上神坛以至于让语句变的如此啰嗦哪？ 为什么一种语句成分的低位非得高于另外一种？这并不是好像面向对象的编程突然使得动词的低位降低，正如我们认为的那样。 这是一种奇怪的认识的扭曲。 正如我的朋友Jacob Gabrielso一次说到，提倡面向对象的编程好比提倡面向裤子的穿衣方式。

Java的静态类型系统，像起他任何类似的语言一样，有着共同的问题。 但是过分强调面向名词的编程思想给人带来很大的困扰。 任何类型系统都会要求你重新梳理思路来配合它，但是清除独立的动词看起来十分不合情理。

C++并没有这个问题。因为C++作为C语言的超集允许你定义单独的函数。 此外，C++提供了独立的命名空间的概念。 Java的类承载了太多的内容：命名空间，用户自定义类型，句法委托机制，可见性和作用域机制，等等。

不要误解了我的意思。 我并没有说C++“好”，我只是赞美它至少相比于Java来讲灵活的类型系统。 C++出现问题会让听众抓狂并且想杀了你（比如，意想不到的段错误和其他难以发现的隐患）。 并且在C++你很难找到一个能描述你的想法的咒语。 但是它灵活地可表述的思想的范围却远远超出了Java。 因为C++提供给你了动词，谁想用一个没有动词的语言说话哪？

类是Java中唯一提供的建模的工具。 所以当一个新的想法出现在你脑海的时候，你不得不重塑它，包装它，甚至弄碎它直到它变成一个名词， 即使它开始是一个动作，过程，或者任何其他不是“物”的概念。

我似乎回到了8,9年前一帮搞Perl的家伙对我说的： “伙计，并不是所有的东西都是对象的。”

很奇怪，Java似乎是主流面向对象语言中唯一一个完全以名词为中心的语言。 在Python或者Ruby中，你不会找到AbstractProxyMediator，NotificationStrategyFactory或者其他类似的东西。 为什么在Java中它们满地都是？ 我敢打赌这是原因出在了动词的身上。 Python，Ruby，JavaScript，Perl当然，还有所有的函数式编程语言允许你声明并传递函数而不用用类包装它。

很显然，动态类型语言的使用更容易； 你可以仅仅传递一个引用给函数，函数可以用名字获取它，而函数的调用者仅仅用合适的参数调用函数并正确地使用返回的值就可以了。

但是很多静态类型的语言同样也有第一类的函数。 这包括固定类型的语言比如C和C++，还有类型自动推断的语言比如Haskell和ML。 这些语言仅仅需要一些语法来建立，传递和调用函数的内容就可以了。

Java没有理由不简单地添加第一类函数并最终实现一个成熟的，没有扭曲的可以让人自由运用动词来实现他们想法的世界。 实际上，有一个基于JVM叫做 The Nice programming language 的语言实现了一个非常类似Java的语法， 并包含了一个非常具有表现力的实现了使用动词方式：独立函数。 而Java强制你用Callback，Ruunable或其他匿名接口来包装它为一个类以便于调用。

Sun公司甚至没有打破他们一切函数都必须被类拥有的信条。 任何匿名的函数都会具有一个隐式的this指针指向定义它的类；问题解决了。

我不知道为什么Sun公司坚持Java矗立在名词王国。 我怀疑这是低估了他们的民众；他们添加了泛型，一个更加复杂的概念，所以他们不再关心如何保持他们语法的简练。 并且添加动词并不是一件坏事，这是因为Java现今所建立的：为一个Java程序员提供工具让他们按自己的想法编程更有意义。

我真心希望Java能修复这个缺陷，以便我可以把垃圾带出去并回来玩游戏或者一切当时在做的事情。