人工智能产品测试方法探索

自我介绍

大家好，我叫孙高飞，是来自第四范式的测试开发工程师。 今天我想跟大家讨论一下要如何测试人工智能相关的产品。我们知道人工智能是出了名的难测试，不论是我们以前在互联网就经常看到的内容推荐系统，还是金融领域常用的信息分类系统。测试介入的都不多。 我们最常做的测试其实就是直接看模型效果， 效果不好就慢慢调。你也说不清是不是有bug。 为了改变这种现状，我们希望测试人员也能做一些事情，参与到人工智能服务的质量保证当中来。 所以我们在日常工作中总结了一些经验跟大家讨论一下。

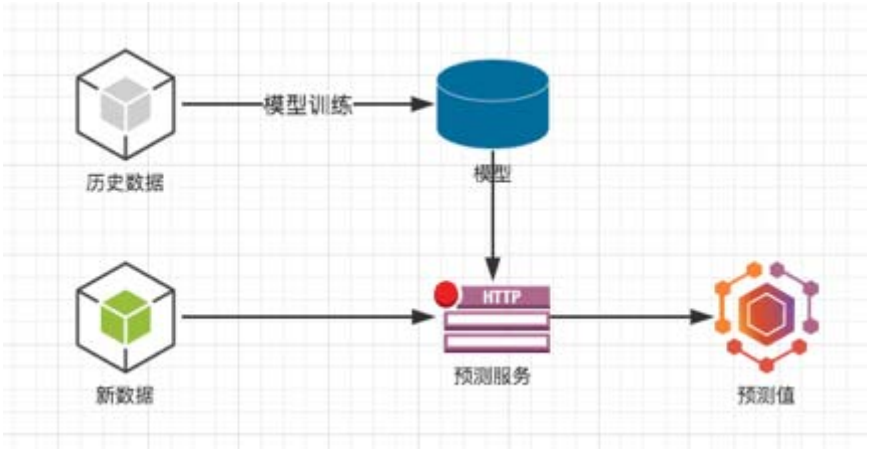
什么是人工智能

再讲如何测试人工智能产品之前，我觉得我要先跟不熟悉人工智能的同学们解释一下什么是人工智能。毕竟想要测试一个东西，就要先了解它么。用一句话来概括现阶段的人工智能就是：人工智能=大数据+机器学习。

我理解的现阶段的人工智能是使用机器学习算法在大量的历史数据下进行训练，从历史数据中找到一定的规律并对未来做出的预测行为。这么说有点拗口。我举个例子，我们曾经给银行做过反欺诈项目。 以前在银行里有一群专家，他们的工作就是根据经验向系统中输入一些规则。例如某一张卡在一个城市有了一笔交易，之后1小时内在另一个城市又有了一笔交易。这些专家根据以前的经验判断这种情况是有盗刷的风险的。他们在系统中输入了几千条这样的规则，组成了一个专家系统。 这个专家系统是建立在人类对过往的数据所总结出的经验下建立的。 后来我们引入了机器学习算法，对过往所有的历史数据进行训练，最后在25亿个特征中抽取出8000万条有效特征。大家可以把这些特征就暂且当成是专家系统中的规则。8000万对不到1万，效果是可以预想的。当时对第一版模型上线后的数据做统计，反欺诈效果提升了7倍， 这就是二分类算法典型的业务场景。

为什么叫机器学习呢，因为它给人一种感觉，机器能像人类一样从过去的数据中学习到经验，只不过机器的能力更强。 如果想再稍微深究一下机器学习训练出来的模型到底是什么，那大家可以暂且理解为一个二分类的模型主要是就是一个key，value的数据库。key就是在数据中抽取出来的特征，value就是这个特征的权重。 当我们想要预测一个用户的行为的时候，就会从用户的数据中提取特征并在模型中查找对应的权重。 最后根据这些特征的权重算出一个分，也可以说是一个概率。 如果大家看过最强大脑这个节目的话。应该记得第一次人机大战项目是人脸识别，第三回合的时候机器给出了两个答案。因为当时做人脸识别项目的志愿者中有一对双胞胎。所以机器计算出的这两个人的概率只差了0.1%，最后为了谨慎起见机器输出两个答案出来。以至于吴恩达先生也很纠结到底该选哪一个答案。

再之后的人机对战项目里我们也能看到虽然小度一路高歌，毫无败绩。但是其中是有选错过几次的。所以大家可以看到我们得到的答案其实是一个概率，是一个根据以往的数据进行总结并作出预测的一个行为。并不是100%准确的。即便是阿尔法狗当初也是输过一局的。所以其实我们测试的痛点也就来了，你怎么测试一个概率呢。 接下来我们慢慢的来说这个问题。



这个图就是一个人工智能服务的略缩图。 在历史数据上训练出模型，并发布一个预测服务，这个预测服务可能就是一个http的接口。 然后新的数据过来以后，根据模型算出一个预测值。经过刚才的说明，我们看到数据是人工智能的根本。拥有的数据越多，越丰富，越真实，那么训练出的模型效果越好。

测试思路

- 数据测试
- 分层测试
- 训练集与测试集对比

数据测试

根据我们之前对人工智能的定义，我们发现数据是人工智能的根据。 保证数据的正确定是非常必要的。 而且几乎所有的机器学习算法对数据的容错能力都很强，即便数据稍有偏差，它们也能通过一次一次的迭代和对比来减少误差。 所以即便我们的数据有一点问题最后得出的模型效果可能还不差， 但是这个时候我们不能认为模型就没问题了。 因为很可能在某些特定场景下就会出现雪崩效应。

再举个例子。当初阿尔法狗与李世石一战成名。如果说前三盘的结果令各路专家大跌眼镜的话。那第四盘可能是让所有人都大跌眼镜了。阿尔法狗连出昏招，几乎是将这一局拱手相让。那阿尔法狗出bug了？DeepMind团队说，这是一个系统问题。那我们来看看这个系统到底有什么问题。根据当时公布出来的数据我们发现阿尔法狗的养成方法是这样的。

阿尔法狗如何养成

1. 整理过去人类对弈的80多万盘棋局

2. 拿1的棋谱，训练一只狗狗，能够预测人类下一步会下在哪里

3. 拿2得到的狗狗，每天自己和自己下100万盘棋

4. 拿1和3的棋谱，再训练一只狗狗，这就是阿尔法狗。

阿尔法狗是基于1亿+盘机器棋局和80万人类棋局训练出来的狗狗。问题出在哪？我们看到，训练阿尔法狗所用的棋谱，只有80万是人类棋局，总数上亿的棋局是机器对弈的。它下的每一步，都是将局面导向历史上(也就是80万人类棋局和1亿自己对弈的棋局)胜率最大的局面。 问题就恰恰出在这里，80万和1亿。相差甚多。那么阿尔法狗选择的所谓胜率最大，一定是赢自己的概率最大，而不是赢人类的概率最大。这样的标准在顺丰棋尚且不容易出问题，一旦遇到逆风局，它的选择就变成了，选择对手犯错概率最大的棋，而这个对手恰恰就是它自己。这就是为什么当初阿尔法狗在逆风局中下出一些匪夷所思的棋。当然这些都只是行业内的人的猜测。但从这个例子就可以看到数据的重要性。数据出现偏差会直接导致不可估计的后果。

所以我们总结出的第一个突破口就是数据测试。保证我们用来建模的数据是正确的。在这里的数据测试会跟我们以往的测试有些不一样。

1. 首先是数据规模的增长。数据大了里面的有些数据可能会发生一些很诡异的事情。

2. 然后是存储介质和技术栈的变化，我们用来训练模型的数据都会存放在hadoop集群上，所以一般都会用spark或者Hbase来编写测试脚本。

3. 最后是验证方式的变化，之前我们测试的时候都是精确的测试每条数据的正确性。但是现在我们需要根据业务设定每一个字段的规则。然后扫描每一行数据，如果这条数据超出了这个字段的规定范围，会发出报警。 假如用户年龄这个字段，正常值可能是0~100岁。如果突然出现个300岁，那么这条数据肯定是有问题的。我们之前曾经碰见过这样的数据。本来这条数据应该是30岁的，但不知道抽了什么风多了个0.

针对这方面的测试，一般都是使用spark或者Hbase这种大数据处理框架把任务提交到集群上去扫描每一张表。 这方面不多说了，脚本其实蛮好写的，只是需要十分熟悉业务，才能制定出相应的规则去扫表。

分层测试

虽然我们能测试用来做训练的数据是正确的。但是一个人工智能系统是很复杂的， 我们只是保证了作为源头的数据貌似是不够的。以前我们测试的时候都知道把系统拆解开做分层测试，所以我们的第二个突破口就是基于这个思路。我们先来了解一下，一个模型的诞生都经理了哪些步骤。

模型诞生的步骤：

- 数据引入

- 数据处理(清洗，拆分，拼接等)
- 特征工程
- 模型训练
- 模型上线

数据引入：将历史数据引入到系统中，并做第一步的预处理。例如处理一些明显的异常的行。

数据处理： 其中又包含很多种操作。 例如对数据进行清洗，拆分。 把两张表进行拼接等。

特征工程：我们从数据中按一定规律提取一些特征出来。之后需要将提取的样本表传递给机器学习算法进行训练

可以看到我们把数据引入到系统后，先是用SQL算子对数据做了拼接，然后清洗一些无效数据。 再把数据拆分为训练集和测试集。分别对两个数据集做特征抽取。之后训练集传递给逻辑回归这个机器学习算法进行训练。训练之后使用测试集来测试一下模型的效果。大家看到这个图了，这是创建一个模型比较常见的流程。图中的逻辑归回就是一种机器学习算法，也是一种最简单的二分类算法，其他的算法诸如GBDT，SVM，DNN等算法的模型都是这个流程。我们可以看到算法上面的流程。全部与机器学习无关。他们都属于大数据处理范畴。而且一个成型的系统在每一个模块都会提供一些固定的接口。

例如我们公司在特征抽取算法上就提供了近百个特征抽取的接口，可以根据不同的情况使用不同的方式提取数据中的特征。 数据拆分也有很多种不同的拆分方法，按随机拆分，分层拆分，规则拆分。 每个子模块都会提供一些接口供上层调用。 所以既然提到接

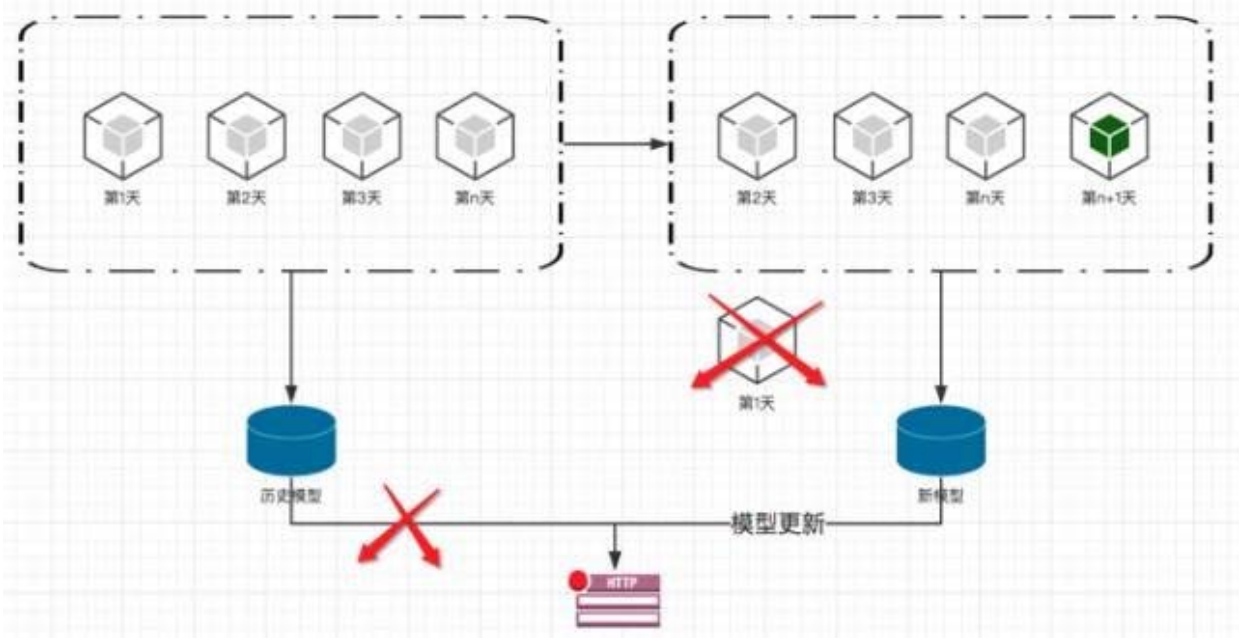
口层面的东西了，大家应该都知道怎么测了吧。 只不过有些接口并不是http或者RPC协议的。 有时候需要我们在产品的repo里写测试用例。

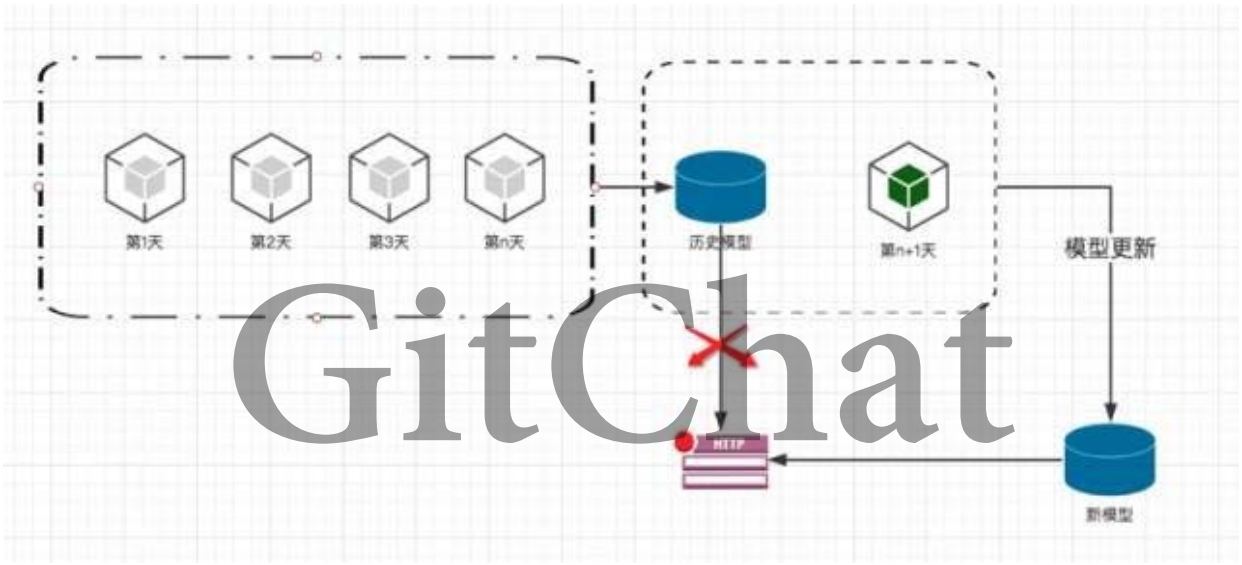
训练集与测试集对比

这是我们的第三种测试思路。 我们刚才一直用来举例的分类算法是一种监督学习。 什么是监督学习呢，就是我们的历史数据中是有答案的。还拿刚才的反欺诈的例子说，就是我们的数据中都有一个字段标明了这条数据是否是欺诈场景。 所以我们完全可以把历史数据拆分为训练集和测试集。将测试集输入到模型中以评价模型预测出的结果的正确率如何。所以每次版本迭代都使用同样的数据，同样的参数配置。 统计模型效果并进行对比。当然这种测试方式是一种模糊的方式。就如我再刚开始说的一样，这种方式无法判断问题出在哪里。是bug，还是参数设置错了？我们无法判断。

常见的测试场景

自学习





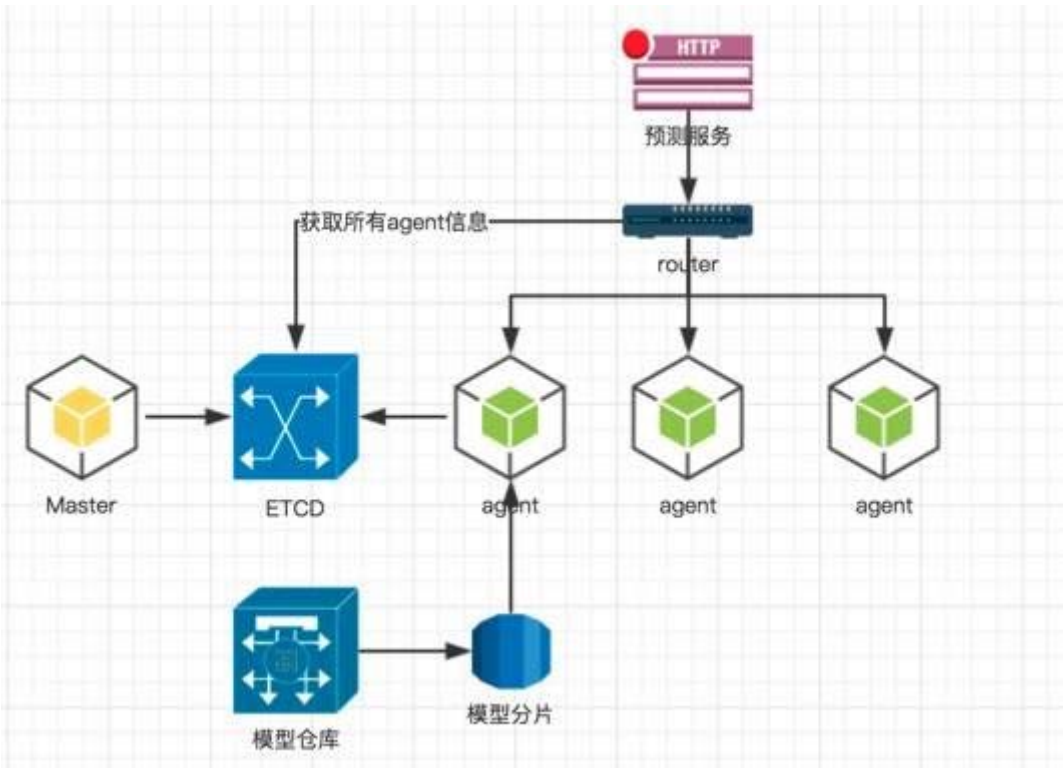
几乎所有的人工智能服务都必须要支持自学习场景。就像阿尔法狗一样，它输了一局，就会从输的这一局中学习到经验，以后他就不会那么下了，这也是机器学习恐怖的地方，它会变的越来越无懈可击，以前人类还能赢上一局，但是未来可能人类再也赢不了阿尔法狗了。 做法就是我们的数据每天都是在更新的，用户行为也是一直在变化的。所以我们的模型要有从最新的数据中进行学习的能力。

上面是常见的自学习场景流程图。假如我们用历史上n天的数据训练出一个模型并发布成了一个预测的服务。 那么到了隔天的时候。我们抛弃之前第一天的数据，使用第二天到第n+1天的数据重新训练一个模型并代替之前的模型发布一个预测服务。这样不停的循环，每一天都收集到最新的数据参与模型训练。 这时候大家应该明白该测试什么了。每天收集到的新数据，就是测试重点。就是我们刚才说的第一种测试思路，使用spark，Hbase这些技术，根据业务指定规则，扫描这些数据。一旦有异常就要报警。

预测服务

下面一个场景是预测服务的。预测服务的架构一般都满复杂的，为了实现高可用，负载均衡等目的，所以一般都是标准的服务发现架构。以etcd这种分布式存储机制为载体。

所有的预测服务分别以自注册的方式来提供服务。



上面的一个图是一个比较流行的预测服务的架构。当然我做了相应的简化，隐去了一些细节。所有的部署任务由master写入ETCD。所有agent以自注册的方式将自己的信息写入ETCD以接受master的管理并执行部署任务。而router也同样读取etcd获取所有agent提供的预测服务的信息并负责负载均衡。有些公司为了做高可用和弹性伸缩甚至将agent纳入了kubernetes的HPA中进行管理。由此我们需要测试这套机制能实现他该有的功能。例如：

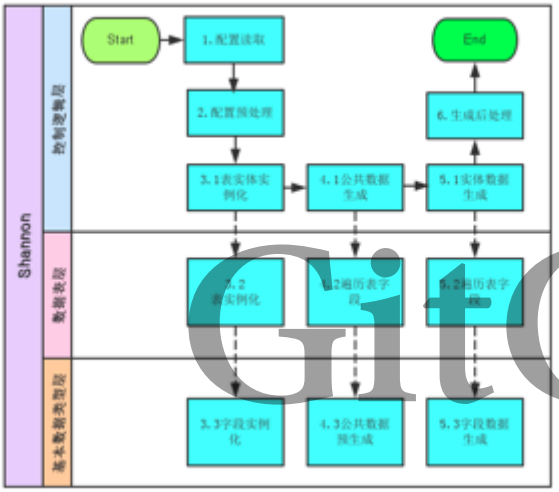
- router会按规则把压力分发到各个agent上。
- 把某个agent的预测服务被kill掉后，router会自动切换。
- 预估服务挂掉，agent会自动感知并重新拉起服务。
- agent被kill掉后，也会被自动拉起。
- 如果做了弹性伸缩，需要将预测服务压到临界点后观察系统是否做了扩容等等。

性能测试

我们要接触的性能测试跟互联网的不太一样。我们知道预测服务仍然还是访问密集型业务。但是模型调研的过程是属于计算密集型业务。我们要模拟的情况不再是高并发。而是不同的数据规模，数据分布和数据类型。我们日常的性能测试都是需要在各种不同的数据下跑各种不同的算子和参数。所以我们首先需要一种造数机制，能帮助我们按需求生成大规模的数据。我们选择的是spark，利用分布式计算在hadoop集群上生成大量的数据。

原理也很简单，接触过spark的同学肯定都知道在spark中生成一个RDD有两种方式， 一种是从文件中读取，另一种是从内存中的一个list种解析。第一种方式肯定不是我们想要的， 所以从内存中的list解析就是我们选择的方式。假如我们想生成一个10亿行的数据。就可以先使用python 的xrange造一个生成器以防止内存被撑爆。然后用这个生成器初始化一个有着10亿行的空的RDD，定义并操作RDD的每一行去生成我们想要的数据，然后设置RDD的分片以及消耗的container，内存，cpu等参数。提交到集群上利用集群庞大的计算资源帮助我们在段时间内生成我们需要的数据。 前两天我再一个3个节点的集群上造过一个1.5T的数据，大概用了5个小时。这样一开始的时候我们是写spark脚本来完成这些事。后来需求越来越多，我们发现可以造数做成一个工具。把表和字段都提取到配置文件中进行定义。就这样我们成立了shannon这个项目。慢慢的从造数脚本到造数工具再到造数平台。

它的架构特别简单，就是对原生spark的应用，这里我就不展示spark的架构是什么样了。就贴一下造数工具的设计图吧。



简单来说shannon分了3层。最底层是基本数据类型层。负责字段实例化，定义并实现了shannon支持的所有数据类型。例如，随机，枚举，主键，unique key，控制分布，大小，范围等等。

测试环境管理

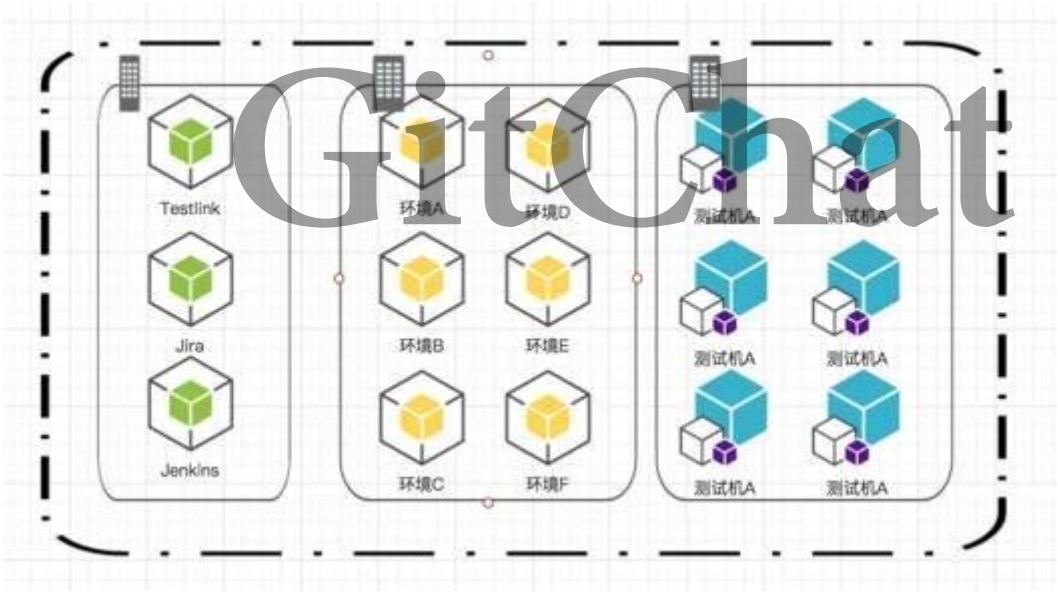
常见的测试场景我们基本上都说完了。 我们再说一说测试环境管理的问题。 为了能够保证研发和测试效率，一个能够支撑大规模测试环境的基础设施是十分必要的。为什么这么说呢？

- 首先但凡是涉及到机器学习的业务，运行时间都非常慢。 有时候做测试的时候跑一个模型要几个小时甚至一天都是有的。也就是说，我们运行测试的成本比较高。如果在运行测试的途中环境出了什么问题那么损失还是很大的。多人共用一套环境难免会有互相踩踏的情况，例如一个RD在测试自己的模块，另一个人上来把服务重启了。这时候我们心里一般就是一万头某种动物飘过。所以我们一般希望每个人都能拥有一套独立的环境甚至一个人多套环境。这就增加了测试环境的数量。尤其是团队越来越大的时候，测试环境的数量已经到达了一个恐怖的量级。

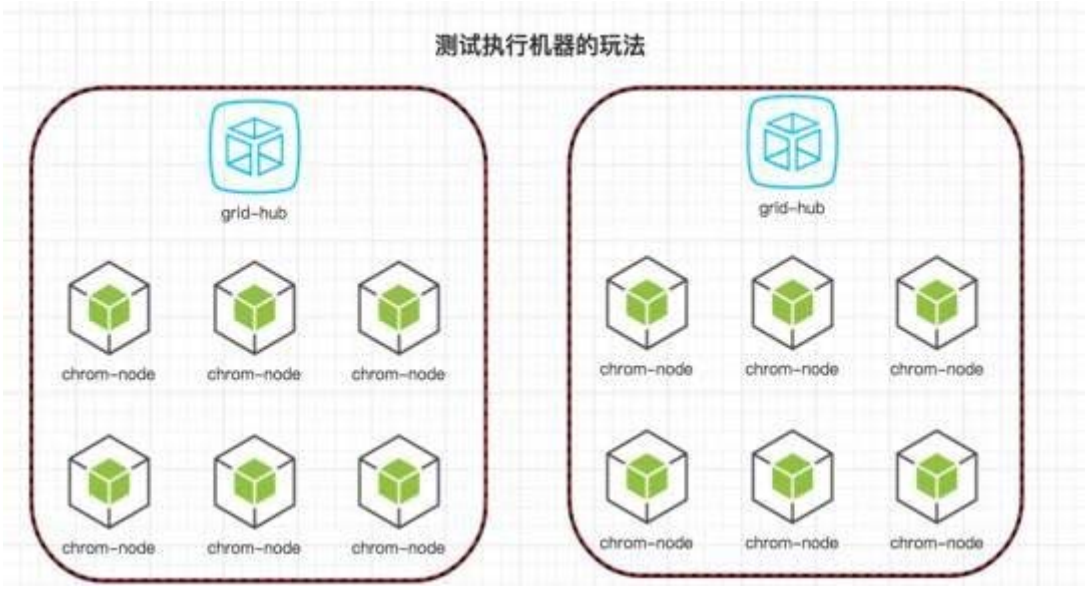
- 其次如果各位所在的公司也像我们一样做TO B的业务，那么我们的测试环境就需要多版本管理，要有能力随时快速的搭建起特定版本的产品环境供开发，产品，测试，以及技术支持人员使用。所以这无疑又增加了环境管理设置的复杂度。

- 再有就是随着环境数量的扩张，我们的环境从单节点走向集群，这时候我们对环境调度能力的要求会比较高，例如我们要对环境的资源进行计算和限制，保证最大化利用资源的同时不会撑爆系统。例如我们要保证系统有足够的冗余，在某些环境出现故障的时候能够自动检测出来并在冗余节点进行恢复。例如我们需要能够实现多租户管理，执行资源管控，限制超售行为. 例如我们希望系统有一定能力的无人值守运维能力等等。

所以我们经过一段时间的讨论和实验，引入了k8s+docker来完成这个目标。docker的优势大家应该都知道，快速，标准化，隔离性，可迁移性都不错。 通过镜像我们可以迅速的将测试环境的数量提升一个量级，镜像的版本管理正适合TO B业务的多版本管理。 之所以选择k8s，是因为k8s相较于swarm和mesos 都拥有着更强大的功能和更简单的部署方式。刚才说的预测服务需要部署很多个agent，使用k8s的话只需要设置一下replica set的数量，k8s就会自动帮我们维护好这个数量的实例了，很方便 。k8s的调度机制能很轻松的满足我们刚才说的对于灾难恢复，冗余，多租户，高可用，负载均衡，资源管理等要求。所以我们当初怀揣着对google莫名的憧憬走上了k8s的踩坑之旅。

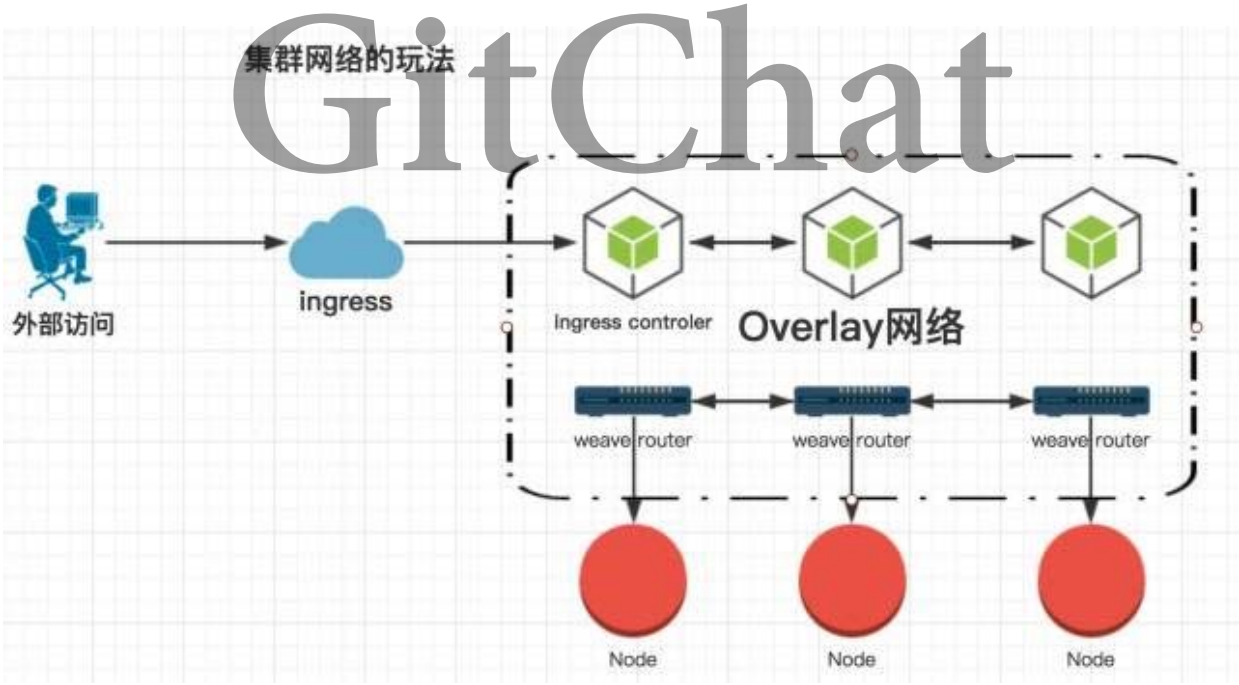


首先说一下我们都用容器做什么。主要三大类，第一种是诸如testlink，jenkins这种基础服务。第二种是产品的测试环境，这是占比最多的。然后就是我们的测试执行机器了。例如UI自动化，我们采取的是分别将selenium hub和node docker化的做法。如下图：



当UI自动化的case增多的时候，分布式运行往往是最好的解决方案。 目前我们通过这种方式容器化了20个浏览器进行并发测试。这些镜像都有官方的版本，使用起来还是蛮方面的。

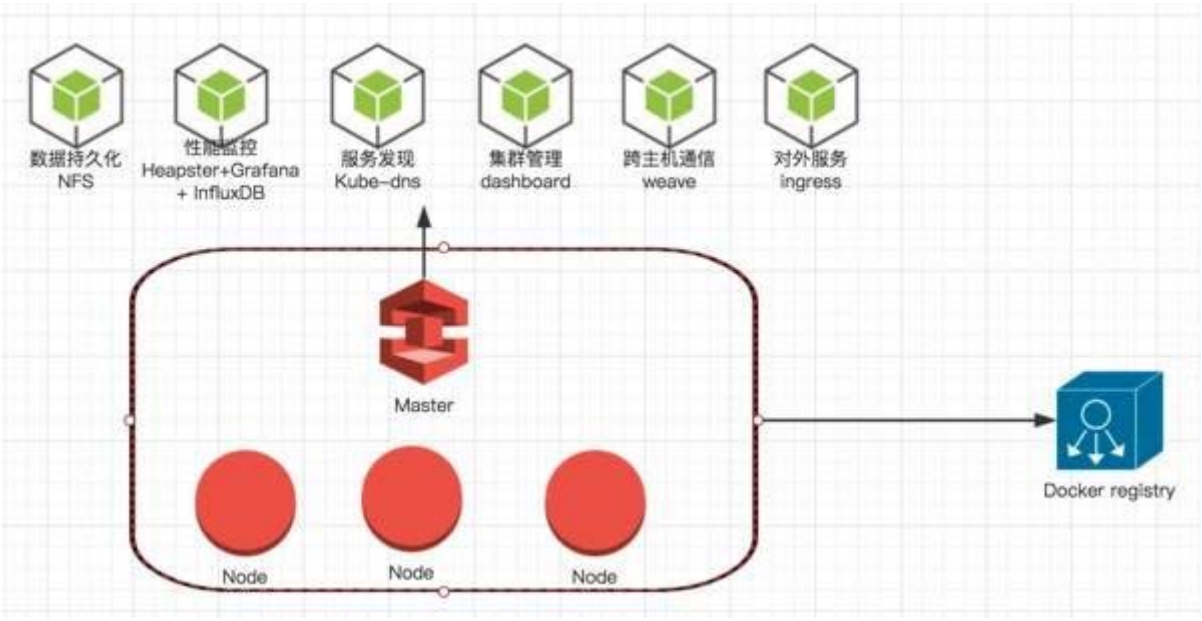
然后说一下比较关键的网络解决方案，我们从单机到集群，中途历经了集中网络模型的变化。从一开始的端口映射，到利用路由规则给容器分配真实的ip，再到给每个容器在DHCP和DNS服务器上注册和续租。到最后我们演进出了下面这个k8s的网络模型。



我们知道每个docker宿主机都会自己维护一个私有网络。如果想让容器跨主机通讯或者外部访问容器。一般就是通过三种方式： 端口映射，路由规则以及overlay网络。我们选择在k8s中引入的overlay网络是weave，以解决夸主机通信问题。安装kube-dns实现服务发现。之后为了能让外部访问容器服务， 使用了k8s提供的ingress机制来实现。这个ingress网络其实就是在集群中启动一个容器，这个容器既能访问容器网络的同是还监听了宿主机的80端口。容器里是一个nginx，它会负责帮忙转发请求。nginx负责转发的有servicename和path，这里我们是无法使用路径进行转发的。所以我们在公司内部的DNS上做了泛域名解析。所有testenv为后缀的域名都会解析成集群的master节点的ip。这样我们的请求就能命中nginx中固定的servicename并做转发了。通过这种机制我们就可以

很方面的访问容器提供的服务。当然ingress的缺点是暂时还无法做4层转发。如果要访问4层协议的服务暂时还是只能暴露node port。

我们这个测试环境的管理平台主要的架构是这样的：



集群中所有的镜像都过公司内部搭建的镜像仓库进行共享，我们在集群之上安装了各种服务来满足测试环境的需要。例如使用NFS做数据持久化，Heapster+Grafana+InfluxDB做性能监控，kube-DNS做服务发现，dashboard提供web管理界面，weave做集群网络，ingress做服务的转发。并且我们在这个整体上针对k8s的APIserver做了一层cli的封装。我们尝试过脚本管理，web服务管理，但是发现大家对这些方式的接受度都不高。我们面对的大多数都是一帮做梦都在写代码的人，所以我们换做提供一个cli的方式可以让使用者更灵活来定制自己需要的服务。通过这种形式，我们在公司内部搭建了一个可以提供测试资源的私有云，配合jenkins我们可以很方便的一键部署我们需要的环境并执行UT,接口,UI自动化测试等等，并提供一个详细的测试报告。下面是我们的部署一个环境后所提供的测试报告。

好了，关于k8s+docker的内容我暂时就讲到这，其实这块内容比较多，完全可以单独拉出来当做一个topic来讲了。但是今天由于时间限制我们先讲这么多吧。我今天的分享也

就到此结束了。关于人工智能类产品的测试我们也属于摸着石头过河，还有很多不足的地方。