校企项目班推广稿

在场的同学们大家下午好吖！今天很荣幸能够在这里向大家介绍我自己在企业日常工作中的一些内容，给大家分享一些经验；以及借此机会，通过提问交流找到我们相互认可的同学，能够深入沟通成为同事。

在介绍之前，我想先分享一下关于测试这个定语的理解，由于我们部门一直以来提供的岗位就是：测试工程师、测试开发工程师，在我开始进入工作之前，对这两个岗位也有偏见，测试？我可是一个励志投身于 IT 行业的程序员诶，完全不想去做测试相关的工作，唯有开发比天高，不都有话说男怕入错行嘛，等到后来入职一段时间我才发现，原来我之前的理解有偏差。一个产品，从开发完成，直到上线运营之前这段期间，就是我们负责测试的同学开始接入的时间，而测试的目的，就是保证产品的质量不出现故障。

我们也都知道，天下没有不透风的墙，也不会有不存在 bug 的程序，这样的话那我们不就是无论如何都无法避免问题走在没有终点的道路上吗？不是的，举个例子，在进行测试的时候，一个服务器一天宕机一次跟一个月宕机一次是两种情况，同样是这个服务器，下午四点宕机跟凌晨四点宕机也是两种情况，在面对以上的情况，测试的责职是什么？

当然，无论什么情况，宕机都不是我们可以接受的，所以我们首先就要定位服务器宕机的问题，保障它不会再次宕机，因为只要是宕机了，就已经是重大事故了，我们必须要避免事故发生，同时总结引发宕机的原因，预防后续再次发生类似的问题。

除了宕机之外，比较常见的问题还有客户端闪退的问题，有时是机型不兼容导致的，有时是打跨平台的包时出现差异，有时是某个场景、某个技能特效所占的内存峰值较高，又或者是玩法逻辑本身存在缺陷，等等等等，都需要从各方各面进行测试。

归结于我们部门在项目中扮演的角色，我们的岗位名称都与测试有关，与代码接触的多的就是测试开发工程师，与代码接触的少就是测试工程师。

机器人

平时我在公司的任务主要是负责游戏服务器所能承载并发数的压力测试，属于测试开发中的一个小类，所以自然也是测试开发的一员，接下来，我准备笼统的介绍

下我的日常工作内容，以及我之前是如何从新手来上手这个任务，同学们如果遇到什么问题，或者有什么建设性想法，可以及时提出，大家互相探讨一番。

由来

通常，当一个项目开发完成，各个流程都已经能够顺利走通没出现问题，这还不能够说明项目已经完成，有一个问题还需要进行确认，服务器在只进入少数测试角色的情况下能够完成所有流程，那当测试角色达到一百万的情况下能不能够承担住，在并发数达到多少的时候会开始会变得不流畅，又或是时延时、回退、丢包等等不正常现象，这时候就需要专门的人来负责服务器压力测试相关内容。

模型（引入代理、安全性协议测试）

压力测试，就是测试服务器在承载一定用户量的时候，通过观察服务器的各项指标，来评估服务器的承载能力，实质做法是模拟多个玩家对服务器进行访问。

针对服务器，一般有几种方式来达到这个目的，一种是在现有客户端上利用 GM 指令接口，构造多个自定义 GM 角色进行相关模块测试，这种做法的优点是能够利用已有客户端接口，开发效率高，但是一个角色操作在客户端多多少少会消耗资源，在一个客户端上构造得越多自然消耗得越快，普通的机器无法满足较高（成千上万）并发需求，所以这种方式只适合需要快速完成测试方案，对并发要求不高的测试。

还有一种是，完全脱离现有客户端资源，重新构造一个只有网络数据处理的“机器人”。这里我们先暂停一下，大家一起思考下，对于服务器来说，玩家的实质是什么？客户端的实质又是什么？玩家与客户端的差别在哪？还有回过头来，服务器的主要工作又是做什么的？（服务器分很多种，这里指部署在远端机器上的服务器程序）

先举个例子，大家在计算机设备上下过棋吗？一般情况下会有对战电脑跟搜索在线对手这两种对战方式，这两种方式从我们玩家的角度来看，可能就是对手不同的差别，电脑会比一般新手玩家厉害一些。但大家仔细想过吗，这两种方式从逻辑来看实现方式是否存在差异。跟电脑对战，我们即使脱机不连服务器也能够在本机上顺利进行，而与在线对手对战的话就一定得连上服务器才可以，只有连上服务器我们才能得到来自服务器的响应，而这个响应可能则是同样连接在该服务器上的另一个用户，从而实现交互通信进行对战。在这个过程中，客户端是用来将我们下棋的动作转换成服务器能理解的指令（因为服务器上可能没有棋盘这个概念，只是存着一个棋盘大小的二维数组，然后将各棋子对应的值放入其中），然后服务器接受这个指令后返回给对手玩家，返回实时棋盘布局，让玩家知道对方相互之间的下棋操作。从这里我们可以看出，客户端就是玩家与服务器交互的一个中间者，当然有时也不单单这样，客户端还可以限制玩家的输入操作，不让玩家作出某个动作，从而避免接收来自玩家的错误指令，比如，“马”向前前进一单位这个指令可能就直接在客户端层面限制住，无法发送至服务器（这方面是一个安全性相关测试的拓展）。

现在我们再回过头来看，想要实现一个“网络机器人”，其实就是，用代码实现客户端网络层处理包的逻辑，相当于脱离图像展示的客户端，把现有的操作指令都封装好在机器人的代码中进行动态组装，从而实现自动的“机器人”。由于没有 UI 占用系统资源，大大降低了单个测试角色对系统资源的要求，从而能够在有限的资源中运行更多的机器人，实现压力测试的目的（单核 300 的并发）。

问题：当你在一游戏场景中点下鼠标使角色移动到你指定的地方，你旁边的玩家是以怎样的过程来接收到你这个鼠标操作的信息?

网络模型

在知道机器人核心就是网络数据交互之后，接下来我们继续了解网络模型。

在计算机中，我们通常是通过流的形式来进行输入输出，比如控制台、文本读写，网络通信也是如此，通过一个称为 socket 的对象，从一端的输入流写入数据，接着在另一端的输出流读取之前写入的数据，从而实现数据交互。

有一点与往常不同的是，这里的流对象的读写不是采用普通阻塞的方式进行（**IO Multiplexing**），不够灵活还影响性能，但我们还是先按顺序性的阻塞模型来理解，然后在这个模型上再加上一个特性，可以在我们所实现的机器人生命周期的任意位置灵活地控制协议逻辑处理（不用专注阻塞的问题）。

接下来再进一步看这个数据的处理方式，前面说了，数据是以流的形式来进行传输的，即是不同请求之间是没有边界的，我们有可能一次读取只收到半个协议的数据，也有可能一次性接收了三个协议之后还有第四个协议一半的数据，这时候我们会引入一个数据包的概念，将单个请求看成是一个协议包，在刚从输出流读取数据时，还未进行处理的数据我们可以将其看成是“粘包的”，需要通过一个“割包”的操作解析出单个完整的请求协议，从而进行处理。

问题：这个**“割包”**的实现思路该如何进行?

网络嗅探工具

在开发过程中，我们时常容易遇到复杂的网络问题，有可能是服务器开发完成度不高没有覆盖所有逻辑出现的问题，有可能是我们开发的机器人代码存在 bug，最明显的表现就是 socket 连接直接断开了，遇到这种情况时就需要进行问题定位。

比如，当开发一个 pk 模块，机器人在进行搜索排队时，突然间断开了连接，我们知道，机器人发起了一个 pk 请求，得到了成功的响应（所以可以等待排队），紧接着等待一个匹配成功的响应从而进入 pk 场，这时候，无论是身边角色的动作同步协议有多少，不论世界频道的聊天讨论有多热烈，机器人只是专注地在等待一个排队匹配成功的响应，因为机器人并没有对其他接收到的响应进行处理。

由于我们不知道服务器发生了什么情况，推送给我们巨量的协议各自是什么含义，而且我们机器人所做的动作也少的可怜（机器人发送心跳包这种协议一定是实现了的），凭空进行问题定位难度很大，所以这时就需要工具来协助定位问题。

先来看网络中的数据传输，并不是像我们想象的那般，发送一个数据包的时候函数一返回就表示对方接收到，举一个具体的例子，我们从一端成功发送数据到另外一端，但对方并不接收，这时算成功还是失败呢？其实 `socket` 数据的收发是操作系统层实现的，我们只是负责将数据放入 `socket` 输入缓冲区，由操作系统将其数据发送到另一端的输出缓冲区，如果对方一直不读取数据清空输出缓冲区的话，也会因为无法接收新数据而造成 `socket` 异常断开连接，而此时双方并没有做出什么不合理的操作，仅仅只是接受端没有及时接收数据导致连接断开。

除开 socket 本身限制导致的错误，我们可以通过 wireshark、tcpdump 之类的工具，嗅探具体的 tcp 报文，观察异常断开连接的报文时由服务端还是客户端发起的，再定位反向的第一个数据包，分析可能导致断开连接的协议内容，这样就能定位平时一些常见的问题。

抽象

除了项目的内容之外，我们来谈一下抽象的概念，举一个例子，当在一个没有自增运算符的语言中，突然出现一个 “++i” 的表达式，你们会怎么理解它？或者，在一个语言中，同时存在 “i++” 与 “++i”，你们会怎么区分它们？抽象的一个做法，就是将一个非基本原子性的事物进行分解，在上述问题中提取相同的部分，对剩下存在差异的部分进行比较单独对比，就能够明了的得出对比结果，我们可以得出这俩个表达式的差异就是返回值不同，相同的是在这个表达式执行过后 i 都会自增。

项目中的抽象

在我们上手一个全新的项目的时候，通常不会一上来直接事无巨细地读整个项目中的代码，也不会接手一份机器人代码的时候一开始就看击败 boss 之后奖励处理模块的逻辑是如何实现的。这样做不仅容易过度消耗个人精力，而且还会造成事倍功半的效果。通常上手一个项目时，我们会先找出程序入口，而且会区分出每个相同层级的代码模块，就像在熟悉战斗模块的代码时，肯定会自动跳过商店操作的相关代码，因为它们属于两个平行没什么交集的模块，而在做协议逻辑处理的时候，也肯定会将协议封装写成一个通用接口，这样每个协议数据也不用重新处理，读代码的时候也不用每个协议都读到具体是如何将这个协议发送至服务器，这些都是关于抽象的一些应用思路。

编程语言的抽象

回到语言上来，我们一开始常常会以为不同语言是各自独立不相交的，对于没了解过的语言会比较抵触。但大家有没有想过，语言的本质是什么？我们先来看计算机，计算机的本质，是一堆硬件，而我们是通过用 CPU 解析操作系统的指令来对其进行管理和计算的，这种 CPU 能解读的指令我们称为机器代码。所以不管我们使用什么语言开发出什么样的工具，最后都会通过编译器或解释器将代码转换成 CPU 可以解读的指令。一种高等级语言的出现，往往是对低一级语言的抽象封装，使其更容易让人理解与使用，而开发出另外一种高级语言，其目的很大可能只是

因另一种高级语言在某个方面做得不够好，针对该目的，舍弃其他一些影响不大的特性来全面优化该领域的特性。因此我们日常接触的高级语言的入门使用教程都是及其相似的，毕竟这些都是通过封装低一级语言而来的，不同语言差异主要体现在各自领域擅长的方面。

术业方向的抽象

同理，既然语言最后都会走到机器语言的结果，那么，在职业选择上也可以类似这样的抽象来进行分析，每个方向最后都能找到一个核心的点，这个点就是我们抽象开始的地方，这里就暂由大家自我发挥了。

抽象的思想

回过头来，我们来看之前抽象的过程，是先通过找到核心的点–CPU 能解析的机器语言，然后再往下拓展成其他各种各样的高级语言，这种先通过抓取核心点，然后进行举一反三拓展问题的思路一般是用于更加全面地认识一个事物。与其相反，还有一种思路，比如我们上手一个项目，就拿最前面那个网络机器人来说，现在需要实现 pk、副本、跑商等模块，在开发之前我们会先进行功能分析，把各个模块的子功能找出，将其接近的可重用的功能尝试进行抽象封装，做成通用库，这种自底向上的思路通常是用于总结多个复杂问题中的相同点，便于归纳。