**STAR法则是情境(situation)、目标（target）、行动(action)、结果(result)四项**

简历项目经历：

1. 云端多车信息监控与轨迹调度服务器

**情景：**在建立的单车智能自动驾驶系统之后，我们考虑使用多车来构建交通场景的方针以及在限定场景内更加高效的运行智能交通系统，包括智能调度，全局感知，轨迹优化。

**目标：**设计服务器实现与多车高校通讯，并解析车辆上传的反馈信息，转发消息给调度模块和显示模块。并在车辆请求路径的时候，根据调度模块计算的轨迹，将轨迹文件下发给车辆并执行。

**行动：**

在这个背景之下，我们需要一个云端服务器来处理多车反馈上来的信息，并反馈给调度模块，调度模块根据多车位置，速度，目标地点信息，综合计算得到理想最优化的路径轨迹，使得这个交通系统里面整体的交通等待时间最短。

上面的这个是全局智能交通调度的功能，还有一个就是可视化功能，将反馈上来的多车信息，在云端地图里面实时更新车辆位置，起到监控大平台的作用。

这里面我设计的服务器起到的作用就是与多车进行通讯，并信息数据格式的解析以及转发给调度模块和显示模块。

其中，调度模块我们还在设计进行中，目前主要是采用简单的路口锁来进行调度，没有考虑车辆速度信息以及目标点信息，不是一个全局调度，是局部调度，这一块我是了解一些，不是我具体负责实现的。

另外显示模块我们是基于ROS系统的可视化工具RVIZ来进行实现的。这个工具原本是用来显示机器人里面的消息，主要可以展示运动状态下的点，方块，车模，箭头以及图片，点云等等元素。我们的服务器通过工作进程将信息解析之后，以ROS节点的形式发布话题消息，另一边RVIZ显示节点订阅话题里面的消息并进行解析，然后在RVIZ平台进行显示。

（解释：ros全称是机器人操作系统，但是他不是一个操作系统，可以认为他是一个通讯中间件，他里面的节点一般可以被看做一个一个独立的进程（nodelet除外），然后进程之间通讯的方式是基于话题和消息的形式来进行通讯，本质上是以xml格式进行http协议的通讯，ros可以实现不同操作系统，不用主机之间的通讯，因为在机器人各个部件里面，可能是依靠不同的操作系统来运行的， ros2.0以及百度apollo自己重新设计的ros系统，底层的通讯修改为了共享内存的方式，加快了进程之间通讯的效率，也比较适合大数据比如点云和图片的通讯。）

**上面介绍了这个服务器是什么功能，以及如何与其他模块进行交互。下面介绍一下服务器具体的执行流程是什么样子的，以及涉及到了哪些技术细节。**

车辆作为客户端，与服务器端进行TCP通讯，客户端是一个自动驾驶单车职能系统，或者是一个仿真环境下的车辆，在方针环境下和在实际环境下，对服务器来说都是一样的，仿真也有自己的自动驾驶算法系统，和仿真模拟器的信息交互被包含在了算法系统里面，我们只需要将仿真自动驾驶系统反馈出来车辆信息发给云端即可。

**车辆端：**

车辆需要发给云端的车辆描述信息主要包括：车辆当前位置（车辆xyz全局坐标以及朝向四元数坐标），车辆当前速度，车辆当前所在的车道（车道被划分为了直道以及弯道，并进行了index编号），车辆id（用于区分不同车辆），车辆目标地址，目标到达时间等等。

车辆客户端以10hz的频率发送反馈信息，如果发生网络故障，比如服务器端500ms没有接受到车辆发送回来的反馈信息，那么socket就会被关闭，车辆客户端也会立即停车。

车辆在启动初始化阶段会主动与服务器建立连接，如果当前没有目标轨迹信息，就会发送轨迹请求tk\_pq\_init报文，服务器端接受到轨迹请求报文之后，就会转发给调度服务器，调度服务器然后计算将待选的多个车道片段和弯道片段连接起来，输出最优目标轨迹，并通过服务器下发给车端，车端获得了目标轨迹之后，就会等等用户输入开始指令，如果开始指令被发出，那么就开始沿着云端下发的轨迹进行前进。

**云端服务器：**

云端服务器主要是与多个车辆客户端进行通讯交互，服务器端的工作流程是：

首先读取本地的配置信息文件，包括ip地址，端口号，线程池默认开启线程数目（与电脑的cpu核心数有关）以及轨迹文件的根目录地址。

生成监听套接字，socket，设置socket adress，bind，listen

生成一个请求解析的数据结构，用于将缓冲区里面的数据进行解析，并存储在该数据结构里面。该数据结构用与监听套接字所使用。

生成epollo，生成epollo 的event数据结构，其中event.event里面存储了监听套接字的模式：ET模式和EPOLLOIN模式（读操作触发）。然后event.data里面的ptr里面存储了请求解析的数据结构。

初始化超时时间队列，就是创建一个时间节点的数组，监听套接字没有对应的计时模块，其他的连接套接字都有一个计时节点，会被加入到超时时间队列，队列里面的节点是按照堆排序进行计算的小顶堆。

然后开始进行epollo的大循环，循环开始阻塞在epollo\_wait上面（Epollo设置时间参数为-1）。如果有就绪事件产生，那么首先判断就绪时间里面是不是有超时的事件，如果有超时的事件，那么就断开连接，并通知给调度模块，专门有一个handle\_timeout来处理，体现了拓展性。

（**超时管理：**这里拓展说一下超时时间处理的逻辑，首先新创建连接套接字都会匹配一个超时时间节点的数据结构，保存在这个连接套接字对应的请求解析数据结构里面，然后这个时间数据结构在初始化的时候，会设置截止时间，一般是当前时间加上500ms的时间，作为截止时间。超时时间队列是小顶堆。如果发现前面最小时间节点的时刻小于当前时刻，那么说明已经超时了，就会调用这个时间节点的超时响应函数，会关闭这个时间节点对应的socket，并调用对应的超时处理函数，通知调度模块。车端发现连接已经关闭，那么就会停车，然后等待工作人员检查网络原因，因为在网络状态比较差的情况下，最好不要进行车端运行，因为难以及时监控车端信息，即使车端有障碍物检测模块。）

另外一个是惰性删除的原因，在一个存活的连接接受到新的数据的时候，之前的时间节点已经过时了，需要重新更新超时时间，这里没有在原来的时间节点上进行修改，而是惰性删除了原来的时间节点，添加了一个新的时间节点，这里可以进行优化，就是修改原来的时间节点内容，并进行重新的堆排序即可）。

前面说了超时管理机制，如果没有超时的情况出现，我们就正常工作。

服务器处理就绪事件的请求，因为我们是epollo,他的好处之一就是直接返回了所有已经就绪的事件，而不需要我们遍历所有的事件。（epollo底层实现，红黑树储存所有注册的事件，就绪链表储存已经就绪的事件，采用回调函数添加事件到就绪队列，不需要来回的在用户态和内核态拷贝所有的事件，提高了效率）。

我们判断如果返回事件的文件描述符如果是监听套接字，那么我们就调用accept创建新的连接套接字，并为这个套接字创建一个请求描述结构体，为这个连接创建一个时间节点，并加入到超时队列里面去。然后将这个连接套接字加入到epollo的注册时间里面去，设置模式为EPOLLO | ET | ONRESHOT , 同时套接字设置为非阻塞IO。非阻塞IO我们和ET模式来进行搭配的，我们这里使用的是reactor模式。ONRSHOT是为了避免多个线程处理一个socket，这样是危险的，我们设置ONESHOT之后，那么这个时间在被注册时候，只会被触发一次，直到重新注册才会被下一次触发。

如果发现是连接套接字的事物处理，那么就进行连接套接字事务处理。

**线程池：**

我们是使用线程池来进行任务分配的，线程池可以减少频繁的线程创建和线程取消的开销。如果使用场景是每一个连接事务处理的时间比较短，线程创建和取消的时间不可忽略，，或者频繁创建和取消任务量不大的小线程，那么我们可以考虑使用线程池。我们这里使用线程池的原因提高系统的最高并发性能，其实不使用线程池我认为也是可以的。

首先创建线程池，我们服务器初始化的过程中就已经初始化了线程池。我们创建了一个线程池一个数据结构，里面存放了一个互斥锁，一个条件变量，然后一个待处理任务队列以及一个线程数组，里面存放了多个线程。首先我们创建这些线程，并派去执行工作函数。在工作函数里面，我们设置了互斥锁和条件变量组合搭配使用，如果待处理任务队列没有任务，那么我们就将线程挂起，等待条件变量的唤醒，如果被条件变量唤醒，那么我们就从待处理任务队列上面取一个任务下来，因为这个任务是一个结构体，里面包含了函数指针，和函数参数，所以这个线程就可以执行这个任务了，执行完毕了之后就继续等待。

（这里说一下互斥锁和条件变量是如何搭配的。首先向候选任务队列里面添加新的任务是需要加锁的，因为他是生产者。加上新的任务之后就唤起条件变量。

在没有任务的时候，工作线程被阻塞在条件变量wait函数的，被唤醒之后，（只唤醒一个，避免惊群效应），就会加上互斥锁，然后判断条件变量是否满足条件，发现满足条件，那么就推出等待，开始取任务，解开锁，去执行了

如果发现条件变量不满足，那么就是进入挂起状态，然后解开锁

使用互斥锁与条件变量进行搭配是为了防止如果不加锁的，判断条件变量之后，在进入挂起状态之前，生产者添加了新的任务，并发起了唤醒，但是此刻消费者是没法收到唤醒信号，所以可能会陷入永久挂起，错过了唤醒信号。

有了互斥锁之后，那么对于添加任务以及唤醒信号的触发就是临界资源，不会有冲突事件发生，也不会错过了唤醒信号）。

线程池的结构如上所示，然后就是如何进行真正工作任务的处理。

我们主要的工作函数是do\_request.里面完成的任务就是数据格式的解析，以及对不同的车端客户端的需要进行处理。

首先，这次的请求没有超时，那么我们就需要更新超时计时器。

然后我们设置一个大循环，不断去读取socket缓冲区里面的数据，直到返回EAGAIN为止（因为我们设置的是非阻塞IO，如果是阻塞IO, 那么线程会卡在这里）说明我们这次数据都读取完毕了。

然后我们读取缓存区里面的数据，主要是依据状态机来进行逐个读取关键字的。使用类似于http报文的格式，我们修改了部分参数，其中DATA代表返回的是数据反馈报文，ROUTE代表的是轨迹请求报文。

数据与数据之间使用空格来进行间隔，下面描述各个报文的数据协议。

数据反馈报文：

DATA 车辆编号id 车辆位置X 车辆位置Y 车辆位置Z 车辆朝向四元数x 车辆朝向四元数y 车辆朝向四元数z 车辆朝向四元数w 车辆速度v 车辆所在车道编号id 车辆目标点x 车辆目标点y 车辆目标点z 车辆预计到达时间t

轨迹请求报文

ROUTE 车辆编号id 车辆位置X 车辆位置Y 车辆位置Z 车辆朝向四元数x 车辆朝向四元数y 车辆朝向四元数z 车辆朝向四元数w 车辆速度v车辆所在车道编号id 车辆目标点x 车辆目标点y 车辆目标点z 车辆预计到达时间t

在工作进程使用状态机解析这些报文，如果状态机没有到达最终状态，那么说明还没有读取到一个完整的数据，需要继续循环读取，或者等待下一次触发继续读取，这里使用缓存区的前后指针来记录当前读取所在的位置，这个变量保存在请求解析结构体里面。

当状态机到达末尾了之后，我们就获取了一个完整的报文，然后如果是DATA格式，就调用publish函数，发送ros消息给显示模块，如果是ROUTE格式，那么就调用轨迹规划函数，输出轨迹点文件.csv。轨迹点文件里面描述了一个个关键点，包括三维位置和四维四元数。

然后发送返回报文，将轨迹点发送回去给客户端。对于正常的DATA报文，服务器返回ACK报文，用作心跳包。

ACK报文很简单，里面包含了车辆id信息以及预留的保留字段，我们可以在ACK报文里面设置调度信息，比如可以设置停车信号，设置速度信号，用户车辆整体规划调度，但是目前还没有实现高级调度模块，仅仅包括了紧急停车信息。

**结果：**实现了多车并发与服务器进行通讯，并将车辆信息适时在大屏幕的RVIZ仿真软件中进行显示。

**遇到的问题以及如何进行解决的：**开始遇到的问题主要是对于一些socket接口使用的不熟悉还有一些逻辑错误导致多线程没有正常运行。

比如开始的时候发现CPU运行满附和，正常来讲应该没有多车连接的时候，没有这么多的符合，然后就是排查问题，发现注释掉一些工作线程之后，还是负荷，就怀疑是epollo的问题，最后发现是epollo的等待时间设置错误，设置为了0,导致不断的循环调用epollo，是一个忙等待的过程，而不是挂起进行，导致CPU占用率非常高。

另外一个bug发现是线程池的部分的逻辑错误，线程池关闭标识位写反，导致线程池一直没有进去分配任务，也是一个忙等待的过程，类似于死循环，发现的原因就是在于创建的六个线程都是基本相等的符合，加起来100%, 发现应该是六个线程出现了同样的问题。

有哪些改进的地方：

报文发送的信息使用字符来表示效率比较低，可以修改为使用protobuf来进行数据交换，这里我们使用自己定义的基于字符的消息传输格式过于简单，如果修改消息格式，需要修改较多的解析代码，不适合于代码后期的更新以及维护。超时传输策略中计时节点每次更新时间的时候，不能删除旧的，添加新的，可以直接在对应的时间上进行修改，然后更新堆排序。

监控模块使用的是独立的一个显示软件 ，这个软件使用起来不是很方便，主要是面向开发者调试代码用的，不是用于前端显示的，后续改进的话考虑可以在浏览器中进行前端战事，这样适用的平台会更加广泛一些。

调度模块需要算法更加高级的设计，不过这一块涉及到比较多的数学知识来研究全局调度。

**`**

**项目2： 基于线控电动底盘车的自动驾驶系统设计与实现**

**情景：**

**目标：**

**行动：**

**结果：**