# ROS软件模块分析

## 一、Roscore启动过程分析

执行命令前需要运行脚本：

source ~/ros\_catkin\_ws/install\_isolated/setup.bash

启动roscore脚本（执行命令：roscore），创建ROSLaunchParent实例并启动；

ROSLaunchParent：

功能：

启动过程：

### **配置文件处理**：

创建ROSLaunchConfig实例；

加载核心配置文件：roscore.xml；

加载用户指定的launch files文件；

加载用户指定的在内存的launch file;

### **启动进程监视器**：

start\_process\_monitor；

创建ProcessMonitor线程实例实例；

### **启动roslaunch 父XmlRPC服务**：

创建ROSLaunchParentNode实例；

创建ROSLaunchParentHandler（xmlrpc api句柄）实例；

等待父类XmlRpcNode启动线程并生成uri；这里的port随机生成，host则获取由 ROS\_IP/ROS\_HOSTNAME指定的名称或者本机的host名称；

### 若有必要，启动远程roslaunch进程；

### 初始化roslaunch runner

* 1. 创建ROSLaunchRunner；注册\_ROSLaunchListeners进程监听器；

### 运行ROSLaunchRunner

根据配置文件配置的节点列表来逐个启动节点；

1. **启动master**: 创建master进程（create\_master\_process），指定包名称（master）名称\参数，和环境变量；
2. **根据配置文件，运行核心节点的服务**（这里默认设置是rosout节点）：先利用节点的配置构建命令行参数（create\_local\_process\_args），再使用master的uri创建子进程的环境变量（设置ROS\_MASTER\_URI，子进程需要uri环境变量与master通讯），创建节点进程；

### rosmaster节点启动

启动命令：['rosmaster', '--core', '-p', '11311', '-w', '3']；

1. 设置ROSMasterHandler;
2. 在11311端口上启动XmlRpcNode节点，并生成uri: <http://test1:11311/;>

XmlRpcNode启动流程：创建新的线程，创建ThreadingXMLRPCServer实例（实际创建SimpleXMLRPCServer实例），并注册ROSMasterHandler消息处理句柄；启动循环模式处理消息；

ROSMasterHandler：

功能：实现订阅注册，发布注册，服务注册\查找，参数添加\删除， 节点查找等功能；

初始化过程：

创建线程池；创建4类注册：publishers（发布类），subscribers（订阅类），services（服务类），param\_subscribers（参数订阅类）；创建ParamDictionary（参数服务）；

**启动rosout包**（启动命令： [u'/home/cmm/ros\_catkin\_ws/install\_isolated/lib/rosout/rosout', u'\_\_name:=rosout']）

## 二、Universal Robot启动过程分析

执行命令：

roslaunch ur\_gazebo ur10.launch

这个命令中ur\_gazebo为包名，ur10.launch为launch 文件，在文件后面可以带传给文件的参数；（命令参数指定方式有2种： 1是：filename + launch args； 2是：package + relative-filename + launch args）

以下分析整个过程：

roslaunch脚本调用roslaunch.main()方法；该方法先分析命令行参数（resolve\_launch\_arguments），获取到对应的launch files文件（这里指ur10.launch全路径文件名），并传递到ROSLaunchParent实例，并调用\_\_init\_\_方法初始化后调用start方法;最后调用ROSLaunchRunner.spin()方法；

ROSLaunchParent.start:

\_start\_infrastructure:加载配置文件，开启XMLRPC服务和进程监控器；

\_load\_config：

load\_config\_default：加载roscore.xml和命令行指定的roslaunch配置文件， 详细过程看下面的“launch配置文件处理过程”；

\_start\_pm： 创建进程监视器ProcessMonitor实例，并启动；

\_start\_server： 创建launch父节点的XML-RPC服务器；

ROSLaunchParentNode.start():

调用父类XmlRpcNode的start()方法创建新的线程；然后调用子类的 run方法；

XmlRpcNode.\_Run:

\_run\_init: 创建ThreadingXMLRPCServer实例，并监听随机端 口；设置uri的值（例如： <http://test1:38317/，> 其中test1为主机名，38317为随机端口）以供其 它节点使用；

ThreadingXMLRPCServer.serve\_forever()：在线程循环调用次 方法；

\_init\_runner：创建ROSLaunchRunner实例，并根据入参初始化各成员；

ROSLaunchRunner.launch:

\_setup:

\_launch\_master:

create\_master\_process: 创建master进程；在创建进程前先设置包名： rosmaster, 根据进程类型（程序或者脚本名称：这里为 rosmaster）\端口（从环境变量ROS\_MASTER\_URI提取）等 参数组装成传递给进程的参数（第一个参数为可执行文件名 称或者脚本名称）；

\_launch\_core\_nodes: 启动核心服务进程；遍历ros\_config的nodes\_core 列表，分别执行核心节点进程；

launch\_node：根据是否为核心启动节点；在创建进程前，先判断 node.name（这是进程的执行文件或者脚本名称）是否为空； 若为空则通过node.type生成匿名名称；

create\_node\_process：通过节点属性和机器属性创建进程；其中 命令行的创建有下面的函数完成；

create\_local\_process\_args：生成子节点进程的参数；其中命 令名称通过传递node.package和node.type参数给 函数roslib.packages.find\_node得到；其它参数则 有node.remap\_args指定；

roslib.packages.find\_node：这个为包的管理类；

\_launch\_setup\_executables: 将执行在配置文件的rosparam节点指定的并 且为设置阶段的执行命令列表；

\_launch\_executable：根据命令行和参数生成子进程；

\_load\_parameters: 加载参数到参数服务器；创建与master通讯的 ServerProxy和MultiCall，并 通过它们设置参数服务器的参数值；

\_launch\_nodes: 遍历ros\_config的nodes列表，每个都执行launch\_node;

launch\_node: 过程同上，只是这里的node不具有core属性；

至此，ROSLaunchParent的初始化完成，之后进入ROSLaunchRunner.spin函数；改函数进程监控器的mainthread\_spin方法，判断是否所以进程都退出，否则一直循环执行主线程的工作；如果没有工作则不断循环；

### launch配置文件的处理：

load\_config\_default：创建ROSLaunchConfig实例和XmlLoader实例，

load\_roscore：加载coscore的launch配置文件（roscore.xml）;

loader.load: 加载解析xml文件，并把信息添加到ROSLaunchConfig；

\_parse\_launch: 解析xml文件，并返回launch节点信息；

add\_roslaunch\_file：把xml文件名信息添加到ROSLaunchConfig；

\_load\_launch：把launch的xml节点信息添加到ROSLaunchConfig；

load\_sysargs\_into\_context：把命令行参数加载到LoaderContext实例；

\_recurse\_load：递归解析launch节点的子节点（总共有以下11种节点名 称：group, node, test, param, remap, machine, rosparam, master, include, env, arg），把相关参数添加到LoaderContext实例；

group:

Node: node节点处理过程；

\_node\_tag: 处理xml的node标签或者test标签，解析节点的 名称空间；解析node节点的必要属性：包名（pkg）\ 类型(type)；还有可选属性: machine\args\output\ respawn\respawn\_delay\cwd\launch-prefix\required；添 加相应的参数到ros\_config对象；最后根据节点类型、 名称、名称空间、参数等信息创建Node实例；

ros\_config.add\_node: 根据node的类型是否为core分别添加到 相应的node列表；

Param:

\_param\_tag： 获取参数的名称和值，如果参数是私有的则保存 在LoaderContext实例中，否则保存在ROSLaunchConfig 实例ros\_config中。参数的值有4中表示形式，分别是： value,textfile,binfile,command;其中value将根据type的 类型（类型有int, double, string, bool, auto,yaml）进行 转换；textfile从指定的文件名中读取string并根据type 的类型进行转换; binfile从指定文件读取二进制流； command则执行指定的命令并把返回值根据type的类 型进行转换。

Remap:

\_remap\_tag:

Machine:

\_machine\_tag: 解析machine节点的必要属性：name\address;解 析可选属性: env-loader\ssh-port\user\password\

default\timeout; 最后根据这些属性创建Machine实例

；最后把machine值添加到ros\_config中。

Rosparam:

\_rosparam\_tag: 根据3个命令(dump\delete\load)来提取相应的 参数；其中dump\delete执行rosparam命令来获取， 这2个命令将在ros\_config.executable里边中添加对象， 等待后边执行；load则通过加载指定的yaml文件；最 后把参数添加到ros\_config对象；

Master: 本节点不作处理；

Include:

\_include\_tag: 解析Include节点的file\ns\clear\_params\

pass\_all\_args等参数；先解析file文件名称；然后解析 include文件的launch节点；添加launch file到ros\_config 文件；最后调用\_recurse\_load递归解析子参数；

Env:

\_env\_tag: 把env节点的name和value值添加到LoaderContext 实例中；

Arg:

\_arg\_tag: 解析arg节点的name\value\default\doc属性值；并添 加到LoaderContext中；

## Universal Robot节点分析

加载launch配置文件：

1. etc/ros/roscore.xml： roscore配置文件；
2. ur\_gazebo/launch/ur10.launch：命令行输入配置文件；
3. gazebo\_ros/launch/empty\_world.launch：
4. ur\_description/launch/ur10\_upload.launch：
5. ur\_gazebo/launch/controller\_utils.launch：

节点列表：（括号前面的为name, 括号里边第一个为package, 第二个为type）

|  |  |  |  |
| --- | --- | --- | --- |
| **name** | **package** | **type** | **备注** |
| arm\_controller\_spawner | controller\_manager | controller\_manager |  |
| fake\_joint\_calibration | rostopic | rostopic |  |
| gazebo | gazebo\_ros | gzserver |  |
| gazebo\_gui | gazebo\_ros | gzclient |  |
| joint\_state\_controller\_spawner | controller\_manager | controller\_manager |  |
| robot\_state\_publisher | robot\_state\_publisher | robot\_state\_publisher |  |
| spawn\_gazebo\_model | gazebo\_ros | spawn\_model |  |

### 1、gzserver节点

Gzserver脚本最终执行命令：/usr/bin/gzserver -e ode worlds/empty.world -s /home/cmm/ros\_catkin\_ws/install\_isolated/lib/libgazebo\_ros\_paths\_plugin.so -s /home/cmm/ros\_catkin\_ws/install\_isolated/lib/libgazebo\_ros\_api\_plugin.so \_\_name:=gazebo \_\_log:=./gzserver.log

其中-s参数指定gzserver的插件，gzserver启动2个插件：



#### gazebo\_ros\_paths\_plugin

本插件的功能是提供ROS的包路径给Gazebo, 配置gazebo的资源路径的环境变量，包括：GAZEBO\_MODEL\_PATH\PLUGIN\_PATH\GAZEBO\_MEDIA\_PATH; 这里配置的都为空；

#### gazebo\_ros\_api\_plugin

在插件加载的时候会向gazebo注册一些事件，当事件触发的时候做一些操作；注册事件从Load函数开始；所有注册是事件及功能如下定义：

ConnectSigInt：响应gazebo的sigint事件；触发后，停止加载插件；

ConnectWorldCreated：响应gazebo的完成虚拟世界创建的世界；触发后，创建Gazebo的传输节点Node实例（这个Node是Gazero定义的），该节点用来发布/订阅主题；其中PublisherPtr是ROS用来向Gazebo推送主题；SubscriberPtr是ROS订阅来自Gazebo的消息；

ConnectWorldUpdateBegin： 本事件响应Gazebo的World开始更新事件；

创建NodeHandle实例：用于向Master发布或者订阅主题；当订阅主题或者服务时，它接收Master过来的消息然然后设置Gazebo的world\model\link等属性或者通过Gazebo的Node推送消息；当发布主题时，它接收来自Gazebo的事件然后向Master推送消息(包括：时钟\ModelStates\LinkStates3种)；

下面列一下具体的发布和订阅的主题：

### 2、gzclient节点

gzclient 脚本最终执行命令：/usr/bin/gzclient -g /home/cmm/ros\_catkin\_ws/install\_isolated/lib/libgazebo\_ros\_paths\_plugin.so -g /home/cmm/ros\_catkin\_ws/install\_isolated/lib/libgazebo\_ros\_api\_plugin.so \_\_name:=gazebo\_gui \_\_log:=/home/cmm/.ros/log/07ff363a-4a4e-11e9-8815-0857001e626c/gazebo\_gui-3.log

其中-g参数指定gzclient的gui插件；

可执行文件gzclient是一个可视化的图形接口，它方便用户控制和设置虚拟的属性，它基于QT库；

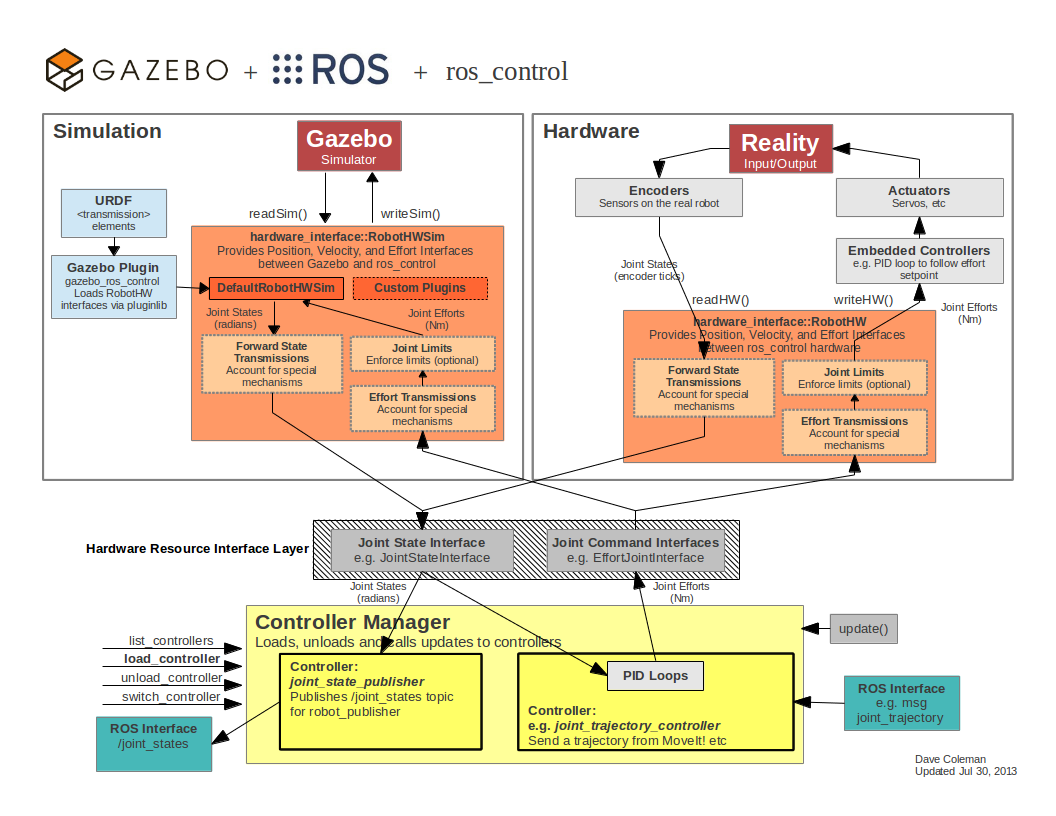
### 3、spawn\_gazebo\_model节点

本节点执行脚本：“u'/home/cmm/ros\_catkin\_ws/install\_isolated/lib/gazebo\_ros/spawn\_model', '-urdf', '-param', 'robot\_description', '-model', 'robot', '-z', '0.1', u'\_\_name:=spawn\_gazebo\_model', u'\_\_log:=/home/cmm/.ros/log/3b76c5b6-4d85-11e9-a80c-0857001e626c/spawn\_gazebo\_model-4.log']”；

rospy.init\_node函数：首先创建一个名叫“spawn\_model”的XML-RPC服务节点，然后把命令行输入的参数作为私有参数添加到参数服务器（本节点没有私有参数）；

SpawnModelNode.run()函数：先从参数服务器获取robot\_description参数的xml串，该xml串描述了robot的所有细节；然后通过gazebo\_interface.spawn\_urdf\_model\_client接口，调用gzserver提供的spawn\_urdf\_model服务（在这之前要等待服务准备好，然后把model的xml参数传给服务，包括插件的名称：libgazebo\_ros\_control.so）在gzserver创建Model；

### gazebo\_ros\_control插件



本插件由gzserver负责加载。具体过程：由spawn\_gazebo\_model节点从参数服务器获取robot\_description参数的xml串，里边指定了加载libgazebo\_ros\_control.so插件，然后调用gzserver的spawn\_urdf\_model服务，最后由gzserver在构建model的时候加载本插件。

#### 插件功能

1. 创建NodeHandle实例，用于与Master通信，发布eStopTopic主题;另外该 实例还会用在DefaultRobotHWSim与Master的通信；
2. 创建gazebo\_ros\_control::RobotHWSim的子类DefaultRobotHWSim实例，并 初始化；
3. 创建urdf::Model实例，并用于RobotHWSim实例的初始化；
4. 创建ControllerManager实例；并在初始化时，创建ControllerLoader实例， 最后发布list\_controllers、load\_controller、 reload\_controller\_libraries等主题；
5. 监听ConnectWorldUpdateBegin事件；在gazebo的每次的模拟步骤中都会调 用此事件回调函数；

每当事件触发，回调函数会做如下工作：

调用DefaultRobotHWSim实例的readSim接口，利用gazebo model的状态 更新robot模拟；调用ControllerManager的update接口，通知所有的 Controller更新；调用DefaultRobotHWSim实例的writeSim接口更新gazebo model的状态；

#### **DefaultRobotHWSim**类

提供ros\_control模块和gazebo模块之间的位置\速度\扭矩等接口功能；

主要接口如下：

1. initSim:

参数：TransmissionInfo, 模型个关节的参数信息列表；

1. 利用关节的名称，位置\速度\扭矩指针构建各个关节的JointStateHandle句柄；
2. 根据硬件接口类型（Effort\Position\Velocity）注册JointHandle句柄；
3. 根据关节名称从父模型获取gazebo的物理关节指针，用于后面的读取和更新；
4. 注册各种关节限制；
5. readSim:

从gazebo的父模型中读取各个关节的位置\速度\扭矩等参数；

1. writeSim:

设置gazebo模型的各个关节的位置\速度\扭矩等参数；

1. eStopActive：

#### **urdf::Model**类

#### **ControllerManager**类

负责发布list\_controllers、load\_controller等主题；完成Controller的动态加载功能；调用各个Controller的updateRequest接口完成更新功能；根据需要调用各个硬件接口的doSwitch开关；

### 4、joint\_state\_controller\_spawner节点

本节点由launch文件ur10.launch的include”$(find ur\_gazebo)/launch/controller\_utils.launch”指定；它由controller\_manager脚本启动，过程包括load\_controller和start\_controller2个子过程；

load\_controller：调用服务controller\_manager/load\_controller（这个服务由ControllerManager类初始化的时候发布）来加载controller: ”joint\_state\_controller”；这个过程实际是调用ControllerManager的loadController方法完成，它是通过ControllerLoaderInterface接口创建对应的Controller实例；

start\_controller: 调用服务controller\_manager/switch\_controller完成；这个过程实际调用ControllerManager的switchController方法完成；

joint\_state\_controller由配置文件./ur\_gazebo/controller/joint\_state\_controller.yaml详细定义；

### arm\_controller\_spawner节点

跟上面的joint\_state\_controller\_spawner类似，它也是通过controller\_manager脚本调用”controller\_manager/load\_controller”服务来完成启动；

arm\_controller由配置文件arm\_controller\_ur10.yaml详细定义，包括类型：position\_controllers/JointTrajectoryController；关节定义以及一些限制；

#### JointTrajectoryController类

定义在“joint\_trajectory\_controller\_impl.h”文件；

主要接口：

init(HardwareInterface\* hw, ros::NodeHandle& root\_nh, ros::NodeHandle& controller\_nh):

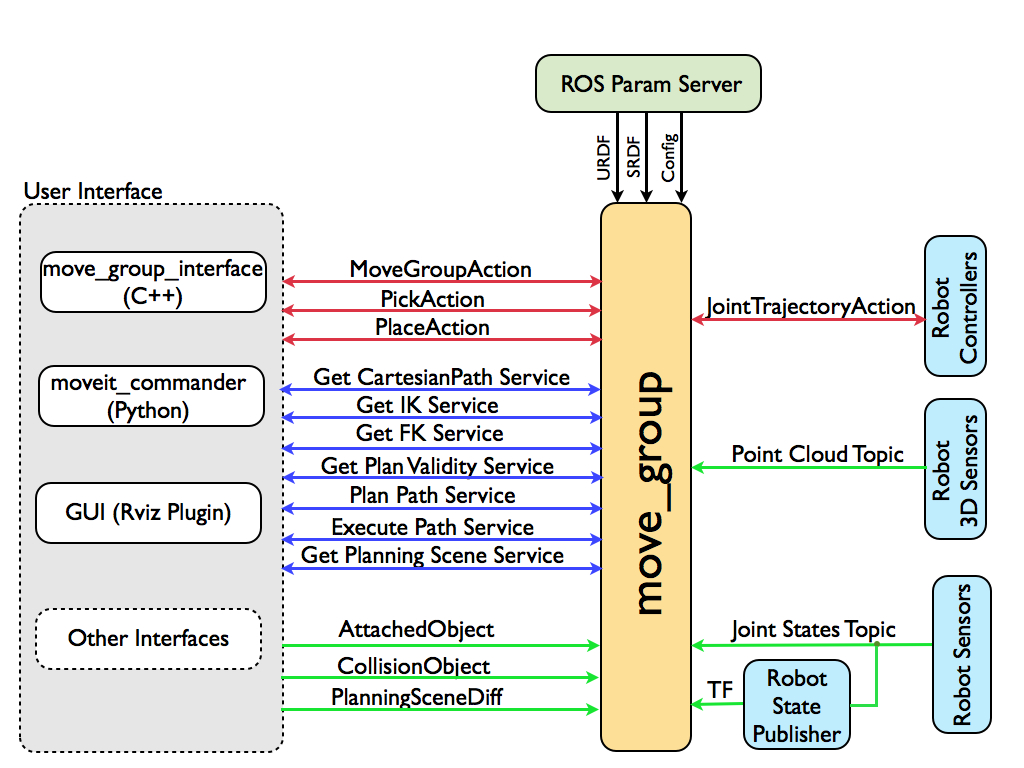
1. 通过controller\_nh从参数服务器获取一些必要的参数设置；
2. 通过root\_nh获取“robot\_description”参数，然后构建各个关节；获取各个关节对应的JointHandle实例；
3. 订阅“command”主题，响应command指令；
4. 发布“state”主题，
5. 创建ActionServer实例；
6. 发布“query\_state”服务；
7. 初始化各个关节的所有轨迹规划；

update():

#### ActionServer类

## Moveit库

参考地址：<https://moveit.ros.org/documentation/concepts/>



### moveit\_ros\_move\_group包

NodeHandle节点空闲时候CPU占用100%的bug: roscpp包的版本退回1.12.12版本。

## Moveit规划过程：

### MoveGroupMoveAction类

initialize函数:

创建SimpleActionServer实例并指定当有新的规划目标时的回调executeMoveCallback；当rviz界面点击plan按钮进行规划时，调用此回调函数；

executeMoveCallback函数：

规划回调函数，根据规划目标（MoveGroupGoal）的参数是否仅做规划调用executeMoveCallback\_PlanOnly函数或者executeMoveCallback\_PlanAndExecute函数；

executeMoveCallback\_PlanOnly函数：

根据MoveGroupGoal的request，调用PlanningPipeline类的generatePlan接口产生规划结果，即规划后的最终轨迹（RobotTrajectory实例）；

executeMoveCallback\_PlanAndExecute函数：

### PlanningPipeline类

构造函数：

根据参数服务器的参数，加载响应的Planner插件和Planner Request Adapter插件；通过Planner Request Adapter插件列表构建PlanningRequestAdapterChain实例；

generatePlan函数：

调用PlanningRequestAdapterChain实例的adaptAndPlan接口实现规划功能，并返回规划结果（包含RobotTrajectory实例的结果）；

### PlanningRequestAdapterChain类

它的功能将一系列适配器应用于运动规划。

adaptAndPlan函数：循环调用每个Planner Request Adapter实例的adaptAndPlan函数，

### PlanningRequestAdapter类

adaptAndPlan函数：

调用PlanningContext实例的solve接口返回规划结果；

### RobotTrajectory类：

维护规划轨迹的航点（waypoint）序列以及航点之间的时间间隔。

### OMPLPlannerManager类

initialize函数：

创建OMPLInterface实例，

### moveit\_planners\_ompl包

#### PlanningContextManager类

getPlanningContext函数1：

根据规划场景和规划请求，先调用下面的“getPlanningContext函数2”获取到ModelBasedPlanningContext实例，然后分别调用ModelBasedPlanningContext的setPlanningScene\setMotionPlanRequest\setCompleteInitialState\setPlanningVolume\setPathConstraints\setGoalConstraints\configure接口做配置；

getPlanningContext函数2：

根据规划配置设置，规划请求，先在Cached Context里查找是否已有缓存的Context，若没有，则根据配置构建ModelBasedStateSpaceSpecification\SimpleSetup\ModelBasedPlanningContextSpecification等实例，然后后根据ModelBasedPlanningContextSpecification实例创建ModelBasedPlanningContext实例，最后返回实例指针；

#### ModelBasedPlanningContext类

setPathConstraints函数：

创建KinematicConstraintSet实例；

setGoalConstraints函数：

创建ConstrainedGoalSampler实例；

solve函数：

调用ParallelPlan实例的solve接口完成ompl库的规划，然后调用convertPath接口把PathGeometric实例的规划路径状态转换成RobotTrajectory实例的点云数组；

#### ConstrainedGoalSampler类

### ompl包

#### ParallelPlan类

它是一个工具类，用于并行执行多个规划器，直到一个或者多个找到解决方案为止。

solve函数：

根据规划器的数量创建相应数量的线程，在线程中调用solveMore函数进行规划，规划完成后，获取结果PathGeometric，并填充到ProblemDefinition实例中。

solveMore函数：

调用Planner实例的solve接口完成规划，然后利用PathHybridization实例的computeHybridPath接口把多个规划路径混合成最后的结果。

#### RRTConnect类(Planner子类)：

solve函数：

#### ProblemDefinition类：

#### geometric：：SimpleSetup类：