

机器人视觉系统标定

1.环境配置

本系统中使用的 Kinect V2，在使用 Kinect 之前，首先需要对 Kinect 的驱动进行安装，下面详细介绍 Kinect 的驱动安装步骤。在 ROS 环境里使用 Kinect2，主要依靠 iai-kinect2 这个包。

Github 地址：https://github.com/code-iai/iai_kinect2。

首先，需要安装其中的 libfreenect2，详细步骤如下（以下默认以 ubuntu14.04 系统版本，其它版本和系统下的安装参见原始网站说明。）

1.1 libfreenect2 安装步骤

(1) 下载 libfreenect2 源码

```
git clone https://github.com/OpenKinect/libfreenect2.git
cd libfreenect2
```

(2) 下载 upgrade deb 文件

```
cd depends; ./download_debs_trusty.sh
```

(3) 安装编译工具

```
sudo apt-get install build-essential cmake pkg-config
```

(4) 安装 libusb. 版本需求 >= 1.0.20.

```
sudo dpkg -i debs/libusb*deb
```

(5) 安装 TurboJPEG

```
sudo apt-get install libturbojpeg libjpeg-turbo8-dev
```

(6) 安装 OpenGL

```
sudo dpkg -i debs/libglfw3*deb
```

```
sudo apt-get install -f
```

```
sudo apt-get install libgl1-mesa-dri-lts-vivid
```

(If the last command conflicts with other packages, don't do it.) 此处会出现这样的提示，在安装的时候，第三条指令确实出现了错误，就直接忽略第三条指令了。

以下部分为可选的安装部分，不安装以下部分不影响本系统的标定和检测功能，但是如果系统中涉及较多的图形编程和 GPU 加速模块，建议安装以下部分。

(1) 安装 OpenCL

Intel GPU:

```
sudo apt-add-repository ppa:floe/beignet
```

```
sudo apt-get update
```

```
sudo apt-get install beignet-dev
```

```
sudo dpkg -i debs/ocl-icd*deb12341234
```

```
AMD GPU: apt-get install opencl-headers
```

```
验证安装: clinfo
```

(2) 安装 CUDA:

(Ubuntu 14.04 only) Download cuda-repo-ubuntu1404...*.deb (“deb (network)”) from Nvidia website, follow their installation instructions, including apt-get install cuda which installs Nvidia graphics driver.

(Jetson TK1) It is preloaded.

(Nvidia/Intel dual GPUs) After apt-get install cuda, use sudo prime-select intel to use Intel GPU for desktop.

(Other) Follow Nvidia website' s instructions.

(3) 安装 VAAPI

```
sudo dpkg -i debs/{libva,i965}*deb; sudo apt-get install
```

(4) 安装 OpenNI2

```
sudo apt-add-repository ppa:deb-rob/ros-trusty && sudo apt-get update
```

```
sudo apt-get install libopenni2-dev1212
```

安装完成后需要对其进行编译，编译步骤如下：

```
Build
```

```
cd ..
```

```
mkdir build && cd build
```

```
cmake .. -DCMAKE_INSTALL_PREFIX=$HOME/freenect2 -DENABLE_CXX11=ON
```

```
make
```

```
make install1234512345
```

编译说明：针对上面 cmake 命令， 第一个参数， 是特别指定安装的位置，可以指定其他地址， 但一般标准的路径是上述示例路径或者/usr/local，尽量不要安装在 home 下，后面使用显得比较乱。 第二个参数是增加 C++11的支持（需要注意的是，不增加对 C++11的支持的话，默认支持的是 C++10）。

设定 udev rules: sudo cp ../platform/linux/udev/90-kinect2.rules /etc/udev/rules.d/, 然后重新插拔 Kinect2.

然后可以尝试运行 Demo 程序: ./bin/Protonect, 安装正确的话应该能够看到如下效

果:



1.2 iai-kinect2安装步骤

安装好 libfreenect2后，可以开始安装 iai-kinect2，下面详细介绍 iai-kinect2的安装和编译步骤。

利用命令行从 Github 上面下载工程源码到工作空间内 src 文件夹内：

```
cd ~/catkin_ws/src/  
git clone https://github.com/code-iai/iai_kinect2.git  
cd iai_kinect2  
rosdep install -r --from-paths .  
cd ~/catkin_ws
```

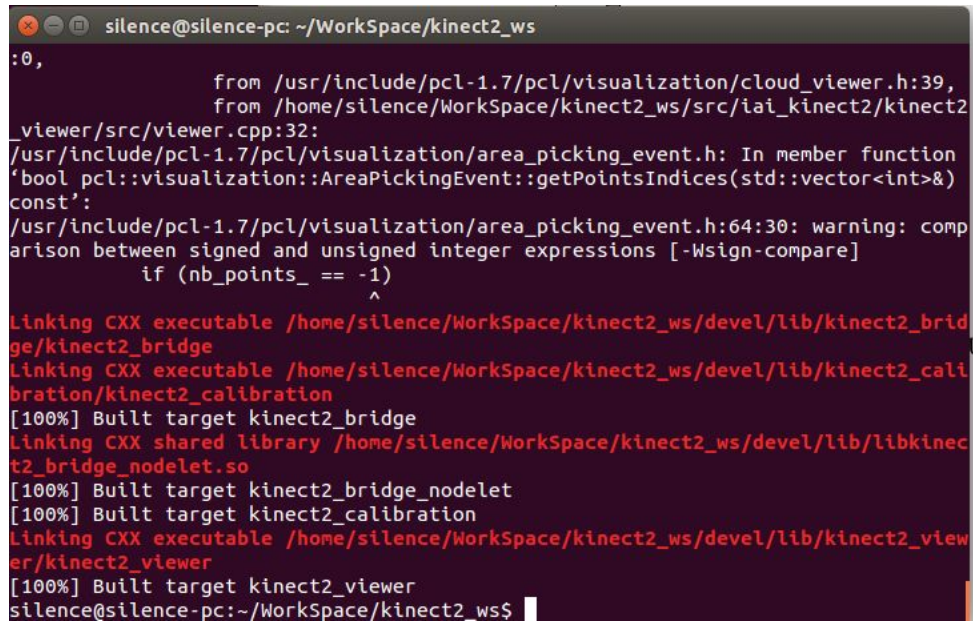
```
catkin_make -DCMAKE_BUILD_TYPE="Release"123456123456
```

编译说明：上述命令中最后一行指令，需要说明的是，如果前面 libfreenect2安装的位置不是标准的两个路径下，需要提供参数指定 libfreenect2所在路径：

```
catkin_make -Dfreenect2_DIR=path_to_freenect2/lib/cmake/freenect2 -
```

DCMAKE_BUILD_TYPE="Release"11

编译结束， 会看到如下提示：

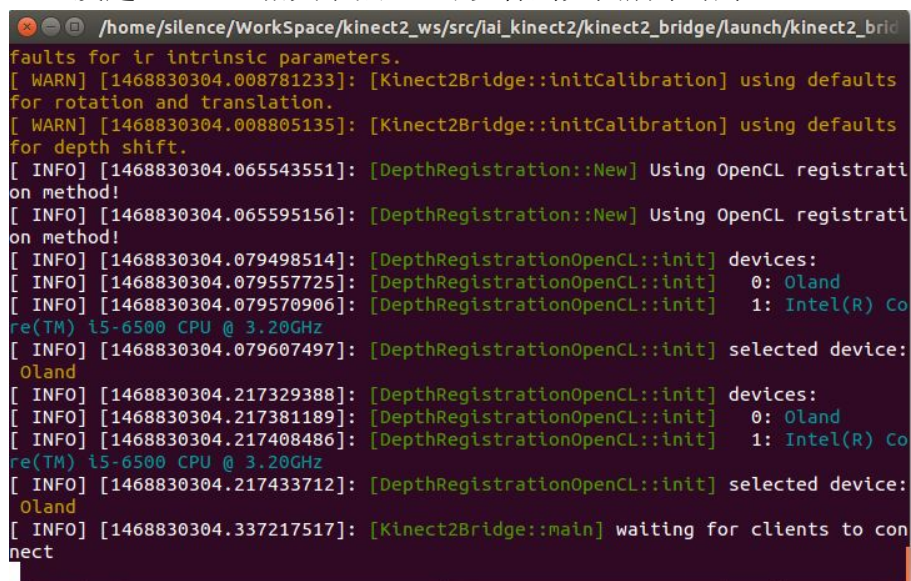


```
silence@silence-pc: ~/WorkSpace/kinect2_ws
:0,
      from /usr/include/pcl-1.7/pcl/visualization/cloud_viewer.h:39,
      from /home/silence/WorkSpace/kinect2_ws/src/iai_kinect2/kinect2
viewer/src/viewer.cpp:32:
/usr/include/pcl-1.7/pcl/visualization/area_picking_event.h: In member function
'bool pcl::visualization::AreaPickingEvent::getPointsIndices(std::vector<int>&)
const':
/usr/include/pcl-1.7/pcl/visualization/area_picking_event.h:64:30: warning: comp
arison between signed and unsigned integer expressions [-Wsign-compare]
      if (nb_points_ == -1)
                          ^
Linking CXX executable /home/silence/WorkSpace/kinect2_ws/devel/lib/kinect2_brid
ge/kinect2_bridge
Linking CXX executable /home/silence/WorkSpace/kinect2_ws/devel/lib/kinect2_cal
ibration/kinect2_calibration
[100%] Built target kinect2_bridge
Linking CXX shared library /home/silence/WorkSpace/kinect2_ws/devel/lib/libkinect
2_bridge_nodelet.so
[100%] Built target kinect2_bridge_nodelet
[100%] Built target kinect2_calibration
Linking CXX executable /home/silence/WorkSpace/kinect2_ws/devel/lib/kinect2_view
er/kinect2_viewer
[100%] Built target kinect2_viewer
silence@silence-pc:~/WorkSpace/kinect2_ws$
```

安装完成后，Kinect2的驱动基本建立， 在 ROS 中运行 Kinect2的节点就可以获取 Kinect2的数据。

roslaunch kinect2_bridge kinect2_bridge.launch

使用 roslaunch 发起 kinect2相关节点， 可以看到如图所示结果，

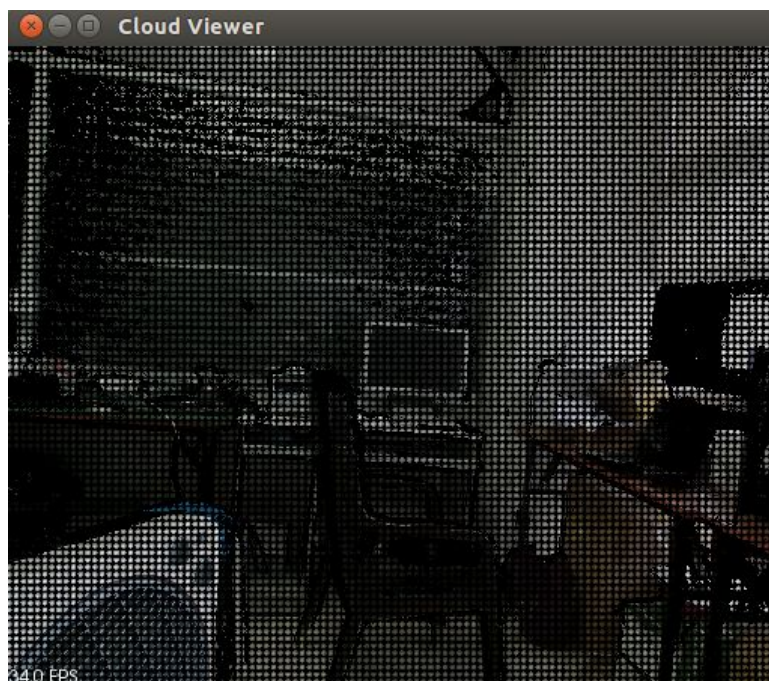


```
/home/silence/WorkSpace/kinect2_ws/src/iai_kinect2/kinect2_bridge/launch/kinect2_brid
faults for ir intrinsic parameters.
[ WARN] [1468830304.008781233]: [Kinect2Bridge::initCalibration] using defaults
for rotation and translation.
[ WARN] [1468830304.008805135]: [Kinect2Bridge::initCalibration] using defaults
for depth shift.
[ INFO] [1468830304.065543551]: [DepthRegistration::New] Using OpenCL registrati
on method!
[ INFO] [1468830304.065595156]: [DepthRegistration::New] Using OpenCL registrati
on method!
[ INFO] [1468830304.079498514]: [DepthRegistrationOpenCL::init] devices:
[ INFO] [1468830304.079557725]: [DepthRegistrationOpenCL::init] 0: Oland
[ INFO] [1468830304.079570906]: [DepthRegistrationOpenCL::init] 1: Intel(R) Co
re(TM) i5-6500 CPU @ 3.20GHz
[ INFO] [1468830304.079607497]: [DepthRegistrationOpenCL::init] selected device:
Oland
[ INFO] [1468830304.217329388]: [DepthRegistrationOpenCL::init] devices:
[ INFO] [1468830304.217381189]: [DepthRegistrationOpenCL::init] 0: Oland
[ INFO] [1468830304.217408486]: [DepthRegistrationOpenCL::init] 1: Intel(R) Co
re(TM) i5-6500 CPU @ 3.20GHz
[ INFO] [1468830304.217433712]: [DepthRegistrationOpenCL::init] selected device:
Oland
[ INFO] [1468830304.337217517]: [Kinect2Bridge::main] waiting for clients to con
nect
```

在另外一个命令行中， 输入 rostopic list， 可以查看到该节点发布出来的 Topic， 还可以输入

roslaunch kinect2_viewer kinect2_viewer sd cloud,

来开启一个 Viewer 查看数据。 结果如下图所示



可能出现的问题:在启动 Kinect 时,运行

```
roslaunch kinect2_bridge kinect2_bridge.launch,
```

出现该文件不是一个 launch 文件的错误提示信息,造成该错误的原因是环境变量没有配置进 .bashrc 文件,

运行 `gedit .bashrc`,加入 `source ~/catkin_ws/devel/setup.bash`

1.3 Kinect 内外参标定

Kinect 的驱动安装完毕后,尽管在运行时,Kinect 驱动内提供了默认的参数,但由于环境和安装过程中的影响,为了提高后续识别的精度,有必要对 Kinect 的内参和外参进行重新标定.

标定板的选择,在安装好 iai-kinect 驱动后,选择驱动文件内的 `kinect2_calibration/patterns` 文件夹,其中有三个标定板的图像,在这里选择 `chess5x7x0.03.pdf`,因为这个图像可以正好打印在 A4 纸上,方便标定。将该文件打印后固定在一个平板上,就可以开始标定了。

标定需要注意的几点:

(1) 标定板必须尽可能平整,使用前必须检测标定板的尺寸,因为在打印标定板标准图像时可能会存在放缩。

(2) 标定时尽量采用两个三角架,一个固定标定板,一个固定 Kinect,这样在标定时可以实现稳定采集图像,在标定时可以只移动标定板进行标定。

(3) 采集标定图像时,尽量多变换位置距离采集,实际操作中可以选择在 1m 和 1.2m 处,分别选取左、中、右三个位置,在这三个位置处利用三角架变换标定板相对于 Kinect 的位置,每次标定采集约 100 幅图像。

标定过程如图所示



Kinect 标定原理图

具体标定步骤

(1) 启动 Kinect, 运行 `roslaunch kinect2_bridge kinect2_bridge.launch`, 如果在 PC 端运行 Kinect 采集图像产生严重卡顿, 可以采用降低帧率的方式, 完成图像采集, 命令为

```
roslaunch kinect2_bridge kinect2_bridge _fps_limit:=2
```

(2) 创建存储标定数据的文件夹.

```
mkdir ~/kinect_cal_data; cd ~/kinect_cal_data
```

(3) 使用彩色相机采集图像 `roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 record color`

(4) 标定彩色相机的内参 `roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 calibrate color`

(5) 使用 ir 相机采集图像 `roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 record ir`

(6) 标定 ir 相机内参 `roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 calibrate ir`

(7) 使用彩色相机和 ir 相机同时采集图像 `roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 record sync`

(8) 标定彩色相机和 ir 相机之间的位置关系, 即求取外参 `roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 calibrate sync`

(9) 校准采集的深度值

```
roslaunch kinect2_calibration kinect2_calibration chess5x7x0.03 calibrate depth
```

(10) 查看 Kinect 的设备序列号, 序列号为运行 kinect2_bridge 时显示的第一行

例: device serial: 012526541941

(11) 在 kinect2_bridge/data/\$serial 目录中创建存储标定结果的文件夹

```
roscd kinect2_bridge/data; mkdir 012526541941
```

(12) 把以下文件从标定目录(~/.kinect_cal_data)下拷到上一步创建的文件夹下

(13) 重新启动 kinect_bridge

实际标定后, 对彩色图和深度图配准时的畸变问题有改善, 效果清晰可见, 所以在项目精度要求较高时不建议采用默认参数, 相机标定步骤不可缺少。

1.4 calib 包的建立

首先需要在 ROS 环境下创建一个包, 具体创建包的步骤如下:

创建一个工作空间:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
```

通过上面两条命令, 就可以创建一个工作空间, 并转到已创建好的工作空间之下, 尽管这个空间是空的, 仍然可以构建 (build) 它:

```
cd ~/catkin_ws/
catkin_make
```

当时用 catkin 工作空间时, catkin_make 是一个非常方便的命令行工具。在工作空间里面多了两个文件夹 “build” 和 “devel”。在 devel 文件夹下, 可以看到很多 setup.*sh 文件。输入如下命令配置工作空间:

```
source devel/setup.bash
```

建过一个空的工作空间: catkin_ws 之后, 下面我们来看一下如何在一个工作空间中创建一个包。在创建一个 catkin 包时需要使用 catkin_create_pkg 命令。

首先进入到目录 ~/catkin_ws/src 下, 使用如下命令:

```
cd ~/catkin_ws/src
```

接着创建一个名字为 Calib 的包, 使用如下命令:

```
catkin_create_pkg Calib std_msgs rospy roscpp
```

在创建的 Calib 文件夹下可以看到 package.xml 和 CMakeLists.txt。catkin_create_pkg 要求给出包的名字, 及选择性的给出所创建的包依赖于哪一个包。这里我们提供了几个程序包作为依赖包, : std_msgs, rospy 以及 roscpp

使用方法如下:

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

这样, 我们的一个包就创建好了, 我们可能会需要对包之间的依赖性做一下解释。我们可以使用 rospack 命令来查看包之间的依赖关系。查看直接依赖关系:

```
rospack depends1 Calib
```

可以看到, 返回的结果正是我们使用 catkin_create_pkg 时, 所使用的参数。我们还可以直接在 Calib 包下的 package.xml 中查看包的依赖关系。使用命令:

```
roscd Calib
cat package.xml
```

结果如下:

```
<package>
```

```
...
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
...
</package>
```

在通常情况下，一个包所依赖的包又会依赖许多其它的包，这称为间接依赖。我们使用如下命令来查看 rospy 依赖的包：

```
rospack depends1 rospy
```

返回结果如下：

```
genpy
rosgraph
rosgraph_msgs
roslib
std_msgs
```

一个包可以有很多的间接依赖关系，我们可以使用命令：

```
rospack depends Calib
```

来进行查看。返回结果如下：

```
cpp_common
rostime
roscpp_traits
roscpp_serialization
genmsg
genpy
message_runtime
roscconsole
std_msgs
rosgraph_msgs
xmlrpcpp
roscpp
rosgraph
catkin
rospack
roslib
rospy
```

接下来配置的 package.xml 包，我们将会一个标签一个标签的分析这个 xml 文件。首先是 description 标签：

```
<description>The Calib package</description>
```

在这个标签中的内容可以改变为任何的内容，不过一般是对这个包的一个简述，尽量简单就行。接下来是 maintainer 标签：

```
<!-- One maintainer tag required, multiple allowed, one person per tag -->
<!-- Example: -->
<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
<maintainer email="user@todo.todo">user</maintainer>
```

这是一个很重要的标签，因为可以根据这个标签知道它的维护者，另外标签的 email 属性也是必须的，可有多个 maintainer 标签。接下来是 license 标签，

```
<!-- One license tag required, multiple allowed, one license per tag -->
<!-- Commonly used license strings: -->
<!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
<license>TODO</license>
```


在后面的使用中，一般我们将 license 修改为 BSD。接下来是依赖性的标签：

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use build_depend for packages you need at compile time: -->
<!-- <build_depend>genmsg</build_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!-- <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use run_depend for packages you need at runtime: -->
<!-- <run_depend>python-yaml</run_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!-- <test_depend>gtest</test_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
```

我们向里面添加了 run_depend:

```
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
```

经过以上的步骤，我们的 package.xml 已经配置好了。

1.4 CMakeLists 配置

在移植时，需要把 calib 包拷到工作空间的 src 目录下，进行编译，calib 包的 CMakeLists 配置如下。（部分需要注意的地方已注释）

```
cmake_minimum_required(VERSION 2.8.3)
project(calib)
```

```
## Find catkin macros and libraries
```

```
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
```

```
## is used, also find other catkin packages
```

```
find_package(catkin REQUIRED COMPONENTS//这里是 Calib 包需要依赖的包
```

```
  cv_bridge                                //opencv 需要依赖的包
```

```
  image_transport                          //图像转化需要依赖的包
```

```
  kinect2_bridge                           //Kinect 图像转化依赖包
```

```
  roscpp
```

```
  sensor_msgs
```

```
  std_msgs
```

```
  message_filters
```

```
  geometry_msgs
```

```
)
```

```
## System dependencies are found with CMake's conventions
```

```
# find_package(Boost REQUIRED COMPONENTS system)
```

```
## Uncomment this if the package has a setup.py. This macro ensures
```

```
## modules and global scripts declared therein get installed
```

```
## See http://ros.org/doc/api/catkin/html/user\_guide/setup\_dot\_py.html
```

```
# catkin_python_setup()
```

```
#####
```

```

## Declare ROS messages, services and actions ##
#####

## To declare and build messages, services or actions from within this
## package, follow these steps:
## * Let MSG_DEP_SET be the set of packages whose message types you use in
##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
## * In the file package.xml:
##   * add a build_depend tag for "message_generation"
##   * add a build_depend and a run_depend tag for each package in MSG_DEP_SET
##   * If MSG_DEP_SET isn't empty the following dependency has been pulled in
##     but can be declared for certainty nonetheless:
##     * add a run_depend tag for "message_runtime"
## * In this file (CMakeLists.txt):
##   * add "message_generation" and every package in MSG_DEP_SET to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * add "message_runtime" and every package in MSG_DEP_SET to
##     catkin_package(CATKIN_DEPENDS ...)
##   * uncomment the add_*_files sections below as needed
##     and list every .msg/.srv/.action file to be processed
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   sensor_msgs# std_msgs
# )

#####
## Declare ROS dynamic reconfigure parameters ##

```

```
#####

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
## * In the file package.xml:
## * add a build_depend and a run_depend tag for "dynamic_reconfigure"
## * In this file (CMakeLists.txt):
## * add "dynamic_reconfigure" to
##   find_package(catkin REQUIRED COMPONENTS ...)
## * uncomment the "generate_dynamic_reconfigure_options" section below
##   and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
#   cfg/DynReconf2.cfg
# )

#####
## catkin specific configuration ##
#####
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if you package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES calib
#  CATKIN_DEPENDS cv_bridge image_transport roscpp sensor_msgs std_msgs
#  DEPENDS system_lib
)

#####
## Build ##
#####

## Specify additional locations of header files
## Your package locations should be listed before other locations
# include_directories(include)
include_directories(
  ${catkin_INCLUDE_DIRS}
)

## Declare a C++ library
# add_library(calib
#   src/${PROJECT_NAME}/calib.cpp
# )

## Add cmake target dependencies of the library
## as an example, code may need to be generated before libraries
```

```

## either from message generation or dynamic reconfigure
# add_dependencies(calib ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

## Declare a C++ executable
# add_executable(calib_node src/calib_node.cpp)

## Add cmake target dependencies of the executable
## same as for the library above
# add_dependencies(calib_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

## Specify libraries to link a library or executable target against
# target_link_libraries(calib_node
#   ${catkin_LIBRARIES}
# )

#####
## Install ##
#####

# all install targets should use catkin DESTINATION variables
# See http://ros.org/doc/api/catkin/html/adv\_user\_guide/variables.html

## Mark executable scripts (Python etc.) for installation
## in contrast to setup.py, you can choose the destination
# install(PROGRAMS
#   scripts/my_python_script
#   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark executables and/or libraries for installation
# install(TARGETS calib calib_node
#   ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
#   RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# )

## Mark cpp header files for installation
# install(DIRECTORY include/${PROJECT_NAME}/
#   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
#   FILES_MATCHING PATTERN "*.h"
#   PATTERN ".svn" EXCLUDE
# )

## Mark other files for installation (e.g. launch and bag files, etc.)
# install(FILES
#   # myfile1
#   # myfile2
#   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
# )

```

```
#####
## Testing ##
#####

## Add gtest based cpp test target and link libraries
# catkin_add_gtest(${PROJECT_NAME}-test test/test_calib.cpp)
# if(TARGET ${PROJECT_NAME}-test)
#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# endif()

## Add folders to be run by python nosetests
# catkin_add_nosetests(test)
add_executable(calib src/calib.cpp) #//添加可执行程序
target_link_libraries(calib ${catkin_LIBRARIES}) #//添加链接库
add_dependencies(calib calib_generate_messages_cpp) #//添加依赖项
```

2. 文件结构

标定部分文件在 calib 包中，源文件 calib.cpp 实现以下功能：

- (1) 彩色图和深度图同步获取
- (2) 标定同心圆的位置检测
- (3) Kinect 坐标系和机械臂坐标系关系转换
- (4) 点云数据的获取

标定结果文件：

arm2cam.yml：机械臂坐标系到 Kinect 坐标系下的转化矩阵

cam2arm.yml：Kinect 坐标系到机械臂坐标系下的转化矩阵

3. 开发说明

整个代码实现的功能包括定义了彩色图和深度图同步获取、标定同心圆的位置检测、Kinect 坐标系和机械臂坐标系关系转换和点云的数据获取。

标定的步骤是首先获取同步采集的彩色图和深度图，通过在彩色图中检测标定同心圆的圆心位置，求解出标定圆心在 Kinect 坐标系下的位置，同时读取机械臂坐标系下的位置，求解转换矩阵，得到 Kinect 于机械臂之间的转换关系。采集图像时，可以同时采集点云图像。

下面针对相关代码进行详细讲解，以下源代码在 calib.cpp 中。

3.1 彩色图和深度图同步获取

这里在话题订阅时采用的同步采集的 Kinect 话题，从 topicColor 和 topicDepth 话题中取得的图像为同步采集得到的图像。

同步采集类型定义：

```
typedef message_filters::sync_policies::ExactTime<sensor_msgs::Image, sensor_msgs::Image,
sensor_msgs::CameraInfo, sensor_msgs::CameraInfo> ExactSyncPolicy;
typedef message_filters::sync_policies::ApproximateTime<sensor_msgs::Image,
sensor_msgs::Image, sensor_msgs::CameraInfo, sensor_msgs::CameraInfo>
ApproximateSyncPolicy;
//订阅的图像话题
std::string topicCameraInfoColor = topicColor.substr(0, topicColor.rfind('/') + "/camera_info";
std::string topicCameraInfoDepth = topicDepth.substr(0, topicDepth.rfind('/') + "/camera_info";
image_transport::TransportHints hints(useCompressed ? "compressed" : "raw");
subImageColor = new image_transport::SubscriberFilter(it, topicColor, queueSize, hints);
subImageDepth = new image_transport::SubscriberFilter(it, topicDepth, queueSize, hints);
subCameraInfoColor = new message_filters::Subscriber<sensor_msgs::CameraInfo>(nh,
topicCameraInfoColor, queueSize);
subCameraInfoDepth = new message_filters::Subscriber<sensor_msgs::CameraInfo>(nh,
topicCameraInfoDepth, queueSize);
```

3.2 标定圆圆心检测

标定圆圆心检测功能在函数 image_process 中实现。部分代码如下：

图像预处理：对图像去噪，用 canny 算子得到边缘，用 HoughCircles 函数检测图像中的圆，由于存在多个同心圆，通过设置阈值分别保存检测出的圆心，最后求取平均值作为检测的标定圆圆心。以下代码为预处理部分和检测最大圆的部分，其余检测圆的原理与此相同。

```
// Gaussian blur and extract edge
cv::cvtColor(img, img, CV_RGB2GRAY);
cv::GaussianBlur( img, img, cv::Size(9, 9), 2, 2 );
cv::Canny(img, edge, 10, 200, 3, false);

//Find circles and show centers
cv::vector<cv::Vec3f> circles0;
cv::HoughCircles(img, circles0, CV_HOUGH_GRADIENT, 1, 10000, 200, 20, 10, 20);

for( size_t i = 0; i < circles0.size(); i++ )
{
    cv::Point center0(cvRound(circles0[i][0]), cvRound(circles0[i][1]));
    int radius0 = cvRound(circles0[i][2]);
    // draw the circle center
    cv::circle( img_origin, center0, 3, cv::Scalar(0,255,0), -1, 8, 0 );//color -BGR
    // draw the circle outline
    cv::circle( img_origin, center0, radius0, cv::Scalar(0,0,255), 1.5, 8, 0 );

    avg_u += cvRound(circles0[0][0]);
    avg_v += cvRound(circles0[0][1]);
    circle_count++;
}
```

3.3 Kinect 和机械臂坐标系转换关系求取

从深度图中得到圆心处的深度位置，即 Kinect 坐标系下的 Z 坐标

```
float get_depth(cv::Mat& depth_image, int u, int v)
{
    float depth = depth_image.at<unsigned short>(v, u) / 1000.0f;
    return depth;
}
```

根据相机的投影模型得到 Kinect 坐标系下圆心的 X、Y 坐标

```
void UVtoXY(float *x, float *y, int u, int v, float depthValue)
{
    float fx = 505.565470964, fy = 506.534289972;
    float cx = 461.185698933, cy = 278.003533135;

    *x = (u - cx) * depthValue / fx;
    *y = (v - cy) * depthValue / fy;
}
```

通过 ros 消息回调得到机械臂末端坐标

```
void pointCallback(const geometry_msgs::Point::ConstPtr &msg)
{
    pointLock.lock();
    armPoint.x = msg->x;
    armPoint.y = msg->y;
    armPoint.z = msg->z;
    pointLock.unlock();
}
```

得到圆心在 Kinect 和机械臂坐标系下的坐标后，采用 PnP 原理求解 Kinect 和机械臂之间的转换关系，通过以下函数实现

```
void solveCameraPose()
{
    solvePnP(Points3D, Points2D, cameraIntrinsicMatrix, cameraDistortionCoefficients,
rvec, tvec, false, CV_EPNP); //该方法可以用于 N 点位姿估计
    cv::Rodrigues(rvec, rotM);

    // 将机械臂坐标系下的点转换到相机坐标系下的[R|t]
    // 旋转矩阵
    double r11 = rotM.ptr<double>(0)[0];
    double r12 = rotM.ptr<double>(0)[1];
    double r13 = rotM.ptr<double>(0)[2];
    double r21 = rotM.ptr<double>(1)[0];
    double r22 = rotM.ptr<double>(1)[1];
    double r23 = rotM.ptr<double>(1)[2];
    double r31 = rotM.ptr<double>(2)[0];
    double r32 = rotM.ptr<double>(2)[1];
    double r33 = rotM.ptr<double>(2)[2];

    // 平移向量
```

```

double tx = tvec.ptr<double>(0)[0];
double ty = tvec.ptr<double>(0)[1];
double tz = tvec.ptr<double>(0)[2];

// 整理成齐次矩阵, 求逆, 得到相机坐标系转换到机械臂坐标系下的[R|t]
arm2cam.at<double>(0,0) = rotM.ptr<double>(0)[0];
arm2cam.at<double>(0,1) = rotM.ptr<double>(0)[1];
arm2cam.at<double>(0,2) = rotM.ptr<double>(0)[2];
arm2cam.at<double>(1,0) = rotM.ptr<double>(1)[0];
arm2cam.at<double>(1,1) = rotM.ptr<double>(1)[1];
arm2cam.at<double>(1,2) = rotM.ptr<double>(1)[2];
arm2cam.at<double>(2,0) = rotM.ptr<double>(2)[0];
arm2cam.at<double>(2,1) = rotM.ptr<double>(2)[1];
arm2cam.at<double>(2,2) = rotM.ptr<double>(2)[2];

arm2cam.at<double>(0,3) = tvec.ptr<double>(0)[0];
arm2cam.at<double>(1,3) = tvec.ptr<double>(0)[1];
arm2cam.at<double>(2,3) = tvec.ptr<double>(0)[2];

arm2cam.at<double>(3,3) = 1.0;

cam2arm = arm2cam.inv();

std::cout<<"arm2cam:"<<std::endl<<arm2cam<<std::endl;
std::cout<<"cam2arm:"<<std::endl<<cam2arm<<std::endl;

cv::FileStorage fs1("arm2cam.yml", cv::FileStorage::WRITE);
fs1<<"R_t"<<arm2cam;
fs1.release();

cv::FileStorage fs2("cam2arm.yml", cv::FileStorage::WRITE);
fs2<<"R_t"<<cam2arm;
fs2.release();
}

```

采集图像点时, 定义按键进行多个目标点的采集:

s 键, 程序记录三维坐标和图像二维坐标, 至少需要4组对应坐标

o 键, 程序开始求解坐标变换参数

v 键, 程序输出转换结果

```

switch(key & 0xFF)
{
case 27:
case 'q':
    running = false;
    break;
case ' ':
case 's':
    Points2D.push_back(cv::Point2f(u, v));

```

```

        pointLock.lock();
        Points3D.push_back(armPoint);
        pointLock.unlock();
        count++;

        std::cout<<"save point count: "<<count<<std::endl;
        std::cout<<"point2d: "<<cv::Point2f(u, v)<<std::endl;
        std::cout<<"point3d: "<<armPoint<<std::endl;

        break;
    case 'o':
        solveCameraPose();
        break;
    case 'v':
        kinect_point.at<double>(0,0) = X;
        kinect_point.at<double>(1,0) = Y;
        kinect_point.at<double>(2,0) = Z;
        kinect_point.at<double>(3,0) = 1.0;

        arm_point = cam2arm * kinect_point;
        std::cout<<"cam point: "<<kinect_point<<std::endl;
        std::cout<<"arm point: "<<arm_point<<std::endl;
        std::cout<<"Real arm point: "<<armPoint<<std::endl;

        break;
}

```

在求解 Kinect 和机械臂坐标系之间的转换关系时，使用的是 PnP 求解方法，这里有必要先简单介绍 PnP 问题的求解原理。

我们用 Kinect 检测得到标定圆的圆心位置，同时可以通过机械臂得到圆心的位置，这种 2D 点和 3D 点之间的对应转换关系在实际应用中，可以当做是一个 PnP 问题的简化，也就是相机位姿估计的基本问题，关于 PNP 问题就是指通过世界中的 N 个特征点与图像成像中的 N 个像点，计算出其投影关系，从而获得相机或物体位姿的问题。

以下讨论中设相机位于点 O_c ， P_1 、 P_2 、 P_3 ……为特征点。



PnP 问题求解原理图

(1) 当 N=1时

当只有一个特征点 P1，我们假设它就在图像的正中央，那么显然向量 OcP1就是相机坐标系中的 Z 轴，此事相机永远是面对 P1，于是相机可能的位置就是在以 P1为球心的球面上，再一个就是球的半径也无法确定，于是有无数个解。

(2) 当 N=2时

现在多了一个约束条件，显然 OcP1P2形成一个三角形，由于 P1、P2两点位置确定，三角形的变 P1P2确定，再加上向量 OcP1，OcP2从 Oc 点射线特征点的方向角也能确定，于是能够计算出 OcP1的长度=r1，OcP2的长度=r2。于是这种情况下得到两个球：以 P1为球心，半径为 r1的球 A；以 P2为球心，半径为 r2的球 B。显然，相机位于球 A，球 B 的相交处，依旧是无数个解。

(3) 当 N=3时

与上述相似，这次又多了一个以 P3为球心的球 C，相机这次位于 ABC 三个球面的相交处，终于不再是无数个解了，这次应该会有4个解，其中一个就是我们需要的真解了。

(4) 当 N 大于3时

N=3时求出4组解，好像再加一个点就能解决这个问题了，事实上也几乎如此。说几乎是因为还有其他一些特殊情况，这些特殊情况就不再讨论了。N>3后，能够求出正解了，但为了一个正解就又要多加一个球 D 显然太占用资源，为了更快更节省计算机资源地解决问题，先用3个点计算出4组解获得四个旋转矩阵、平移矩阵。根据公式：

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

将第四个点的世界坐标代入公式，获得其在图像中的四个投影（一个解对应一个投影），取出其中投影误差最小的那个解，就是我们所需要的正解。

传统的 PnP 问题就是指通过世界中的 N 个特征点与图像成像中的 N 个像点，计算出其投影关系，从而获得相机或物体位姿的问题。在使用 Kinect 的情况下,可以得到特征点的深度信息,所以说该问题为 PnP 问题的一个简化. 相机位姿估计就是通过几个已知坐标的特征点，以及他们在相机照片中的成像，求解出相机位于坐标系内的坐标与旋转角度，其核心问题就在于对 PNP 问题的求解

对 pnp 问题的求解直接调用了 OpenCV 的库函数“solvePnP”，其函数原型为：

```
bool solvePnP(InputArray objectPoints, InputArray imagePoints,
InputArray cameraMatrix, InputArray distCoeffs, OutputArray rvec,
OutputArray tvec, bool useExtrinsicGuess=false, int flags=ITERATIVE )
```


第一个输入 `objectPoints` 为特征点的世界坐标，坐标值需为 `float` 型，不能为 `double` 型，可以输入 `mat` 类型，也可以直接输入 `vector<point3f>`。

第二个输入 `imagePoints` 为特征点在图像中的坐标，需要与前面的输入一一对应。同样可以输入 `mat` 类型，也可以直接输入 `vector<point3f>`。

第三个输入 `cameraMatrix` 为相机内参数矩阵，大小为 3×3 ，形式为：

$$\begin{bmatrix} 0 & & \\ 0 & & \\ 0 & 0 & 1 \end{bmatrix}$$

第四个输入 `distCoeffs` 输入为相机的畸变参数，为 1×5 的矩阵。

第五个 `rvec` 为输出矩阵，输出解得的旋转向量。

第六个 `tvec` 为输出平移向量。

第七个设置为 `true` 后似乎会对输出进行优化。

最后的输入参数有三个可选项：

`CV_ITERATIVE`，默认值，它通过迭代求出重投影误差最小的解作为问题的最优解。

`CV_P3P` 则是使用非常经典的 Gao 的 P3P 问题求解算法。

`CV_EPNP` 使用文章《EPnP: Efficient Perspective-n-Point Camera Pose Estimation》中的方法求解

在本项目中解决 PnP 问题选择的是 EPNP 求解算法。

3.4 点云的获取和保存

这里采集的点云是在彩色图和深度图配准的情况下采集的点云图，如果 Kinect 标定参数效果不好，配准的彩色图和深度图将出现不匹配现象，此时采集的点云图在后续的点云识别等问题上将出现错误，所以在保存点云时应先确定深度图和彩色图是同步采集的。

//点云创建和保存

```
void createCloud(const cv::Mat &depth, const cv::Mat &color,
pcl::PointCloud<pcl::PointXYZRGBA>::Ptr &cloud) const
{
    const float badPoint = std::numeric_limits<float>::quiet_NaN();

    #pragma omp parallel for
    for(int r = 0; r < depth.rows; ++r)
    {
        pcl::PointXYZRGBA *itP = &cloud->points[r * depth.cols];
        const uint16_t *itD = depth.ptr<uint16_t>(r);
        const cv::Vec3b *itC = color.ptr<cv::Vec3b>(r);
        const float y = lookupY.at<float>(0, r);
        const float *itX = lookupX.ptr<float>();

        for(size_t c = 0; c < (size_t)depth.cols; ++c, ++itP, ++itD, ++itC, ++itX)
        {
            register const float depthValue = *itD / 1000.0f;
```

```

// Check for invalid measurements
if(*itD == 0)
{
    // not valid
    itP->x = itP->y = itP->z = badPoint;
    itP->rgba = 0;
    continue;
}
itP->z = depthValue;
itP->x = *itX * depthValue;
itP->y = y * depthValue;
itP->b = itC->val[0];
itP->g = itC->val[1];
itP->r = itC->val[2];
itP->a = 255;
}
}
}
//同时保存点云和图像
void saveCloudAndImages(const pcl::PointCloud<pcl::PointXYZRGBA>::ConstPtr cloud,
const cv::Mat &color, const cv::Mat &depth, const cv::Mat &depthColored)
{
    oss.str("");
    oss << "/" << std::setfill('0') << std::setw(4) << frame;
    const std::string baseName = oss.str();
    const std::string cloudName = baseName + "_cloud.pcd";
    const std::string colorName = baseName + "_color.jpg";
    const std::string depthName = baseName + "_depth.png";
    const std::string depthColoredName = baseName + "_depth_colored.png";

    OUT_INFO("saving cloud: " << cloudName);
    writer.writeBinary(cloudName, *cloud);
    OUT_INFO("saving color: " << colorName);
    cv::imwrite(colorName, color, params);
    OUT_INFO("saving depth: " << depthName);
    cv::imwrite(depthName, depth, params);
    OUT_INFO("saving depth: " << depthColoredName);
    cv::imwrite(depthColoredName, depthColored, params);
    OUT_INFO("saving complete!");
    ++frame;
}

```

4.操作说明

4.1 节点启动

(1) 启动 Kinect

```
roslaunch kinect2_bridge kinect2_bridge.launch
```

(2) 启动 calib 节点

```
roslaunch calib calib
```

(3) 启动机械臂节点（获取机械臂末端位置）

```
roslaunch jaco_moveit_control main.sh ;
```

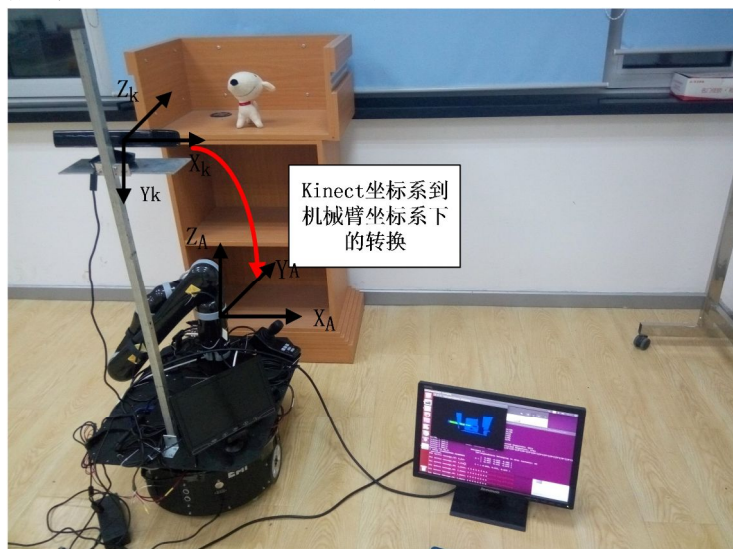
4.2 标定操作

机械臂和 Kinect 传感器之间在安装后存在着固定的位置关系, 协同工作之前需要标定这个位置关系. 解决这个问题的关键在于得到机械臂坐标系下的坐标点对应在 Kinect 坐标系下的坐标. 由于标定板的尺寸较大, 无法安装在机械臂末端, 特征点的位置在机械臂坐标系下无法准确得知. 所以使用一个打印的同心标定圆, 如图所示.



标定同心圆

标定过程中, 将标定圆固定在机械手的中心位置, 该位置在机械臂坐标系下的坐标可以在机械臂配套的 API 中获取. Kinect 得到配准后的彩色和深度图像, 如图所示, 在配准的图像中利用霍夫变换检测标定圆的圆心, 计算出在 Kinect 坐标系下的坐标, 这样同一个点在机械臂坐标系下坐标和 Kinect 坐标系下坐标可以同时获得, 两个坐标系之间的转换矩阵可以求出, 这样就标定了机械臂和 Kinect 之间的位置关系.

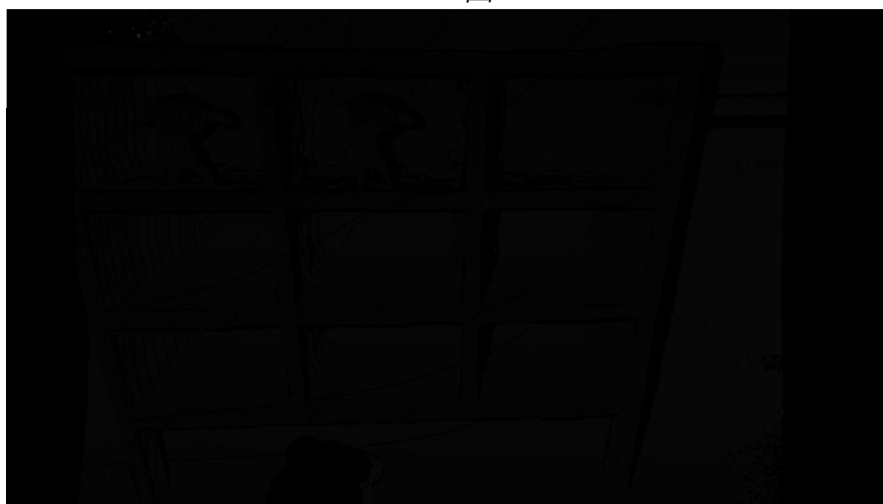


标定原理图

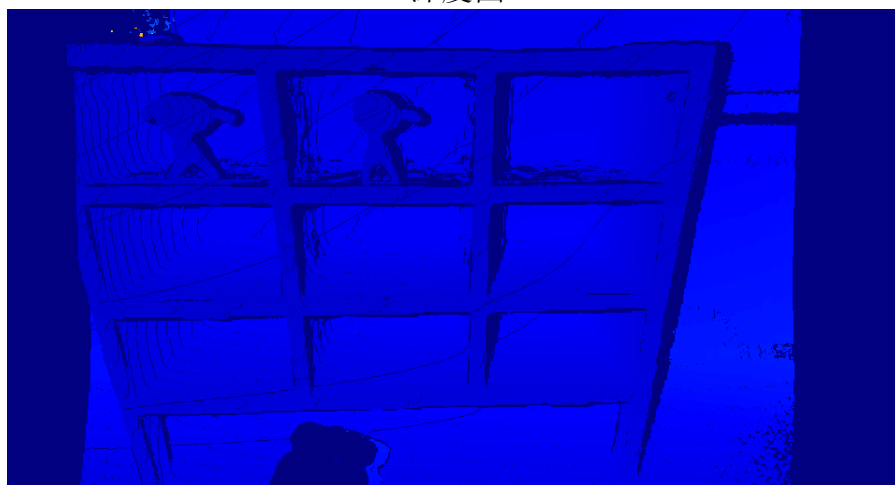
在进行位置关系标定之前, 可以通过 `rostopic` 查看 Kinect 发布的图像话题, 确定彩色图和深度图是同时采集, 并且可以查看配准后的图像, 如下所示.



RGB 图



深度图



配准图

标定步骤如下：

- (1) 将标定同心圆粘贴在机械臂末端
 - (2) 机械臂切换至手动控制模式，使标定圆可以在 Kinect 视野中出现，并且标定圆的圆心位置可以被稳定检测出
 - (3) 按“s”键，同时保存标定圆圆心在 Kinect 坐标系下的坐标和机械臂下的坐标
 - (4) 重复(2) — (3)，保存多组数据点(10个点左右)
 - (5) 按“o”键，求解 Kinect 坐标系和机械臂坐标系之间的转换矩阵，
 - (6) 按“v”键，可输出标定结果并保存成 yml 文件
- 以上就完成了 Kinect 和机械臂之间的位置关系标定。