



Join Yuque to read better

Want to follow this author to read more or favorite this article? Please [Sign Up](#) or

[Log In](#) to Yuque

Join now



PCL第二次分享——PCL基础定义与语法

自我介绍 汪寿安 中国矿业大学（北京）研一

需要讲解的内容

PCL基础：命名规范、设计结构、点类型，增加自己定义点类型，编写新的PCL类，异常处理机制

前期资料准备

PCD格式资料（github） <https://github.com/MNewBie/PCL-Notes/blob/master/chapter2.md>

<<https://github.com/MNewBie/PCL-Notes/blob/master/chapter2.md>>

点云数据结构（黑马） https://robot.czxy.com/docs/pcl/chapter01/intro/#_2

<https://robot.czxy.com/docs/pcl/chapter01/intro/#_2>

PCL中可用的PointT类型 <https://www.cnblogs.com/li-yao7758258/p/6433445.html>

<<https://www.cnblogs.com/li-yao7758258/p/6433445.html>>

分享过程中，请多多交流！

1.1 命名规范

（翻的官网英文，这里简单介绍）

1. 文件命名

- (1) 所有的文件名单词之间应该用下划线隔开,例如 `unordered_map.hpp`。
- (2) 头文件的扩展名为 `.h`。
- (3) 模板类实现文件的扩展名是 `.hpp`。
- (4) 源文件的扩展名是 `.cpp`。

2. 目录命名

所有的目录及其子目录命名应该符合,如果由多个单词组成,其之间用下划线隔开,PCL 中各个目录遵循以下规则:

- (1) 头文件都应放在源码目录树中的 `include/`下。
- (2) 模板类实现文件都应放在目录树中的 `include/impl/`下。
- (3) 源文件都应放在目录树中的 `src/`下。

3. Include 语句

当文件在同一目录下时 Include 指示语句用双引号,在其他情况下则用尖括号,例如:

```
#include <pcl/module_name/file_name.h>
#include <pcl/module_name/impl/file_name.hpp>
#include "file_name.cpp" //在同一目录下
```

4. 宏定义命名

宏定义中字母都采用大写格式,为头文件所定义的宏最后面还需要加上下划线,并且名称从 `include` 下目录开始,例如 `pcl/filters/bilateral.h` 对应 `PCL_FILTERS_BILATERAL_H_`。 `#ifndef` 和 `#define` 定义放在 BSD 协议后面代码前面。 `#endif` 定义一直在文件结尾,并且加上一句注释掉的宏对应头文件的宏定义,例如:

```
// the BSD license
#ifndef PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_ //为避免重复包含头文件而定义的宏
#define PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_
// the code
#endif // PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_
```

5. 命名空间命名

命名空间多于一个单词的,单词之间应该用下划线连接,例如:

```
namespace pcl_io
{
    ...
}
```

6. 类/结构命名

类名(和其他自定义类型的名称)应该是 CamelCased(驼峰命名)命名规范,也就是连写单词组成命名,每个单词首字母大写。但是有例外:如果类名包含一个缩写,这个缩写应该全部大写,类名和结构名最好是名词组成的名字,例如 PFHEstimation 代替了 EstimatePFH,下面是正确的命名代码例子:

```
class ExampleClass;  
class PFHEstimation;
```

7. 函数/成员函数命名

函数和类的成员函数的命名应该采用 camelCased,也就是连写单词组成命名,除了首个单词首字母小写其他单词首字母大写,它们的参数命名单词之间用下划线

隔开,函数和类的成员函数命名最好采用动词,应该确保这些名字能清楚的表达函数和类成员函数的功能,例如,checkForErrors()而不是 errorCheck(), dumpDataToFile()而不是 dataFileDump(),正确的用法:

```
int  
applyExample (int example_arg);
```

8. 变量命名

变量的命名应该单词之间用下划线隔开例如:

```
int my_variable;
```

(1) 迭代子变量命名。迭代子变量应该反应出它们迭代的对象,例如:

```
std::list<int> pid_list;  
std::list<int>::iterator pid_it; //指示迭代的对象为点的索引
```

(2) 常量命名。常量的名字应该是全大写,例如:

```
const static int MY_CONSTANT = 1000;
```

(3) 成员变量命名。类的成员变量命名单词之间用下划线隔开并且以下划线结尾,例如:

```
int example_int_; //对阅读 PCL 源码很有帮助,可明显区分成员变量与局部变量
```

9. Return 语句

return 语句需要在圆括号中设返回值,即规定 return 语句必须有返回值,大家知道,return 如果没有返回值也会编译,例如:

```
int  
main()  
{  
    return (0);  
}
```

1.2 缩进与空格

1. 命名空间缩进格式

在头文件里,命名空间的内容应该缩进两个空格,例如:

```
namespace pcl
{
class Foo
{
    ...
};
}
```

在一个实现文件里,对每一个类成员函数或函数的命名必须添加命名空间限定,例如:

```
void
pcl::Foo::bar ()
{
    ...
}
```

2. 类格式

一个模板类的模板参数必须与类定义在不同行,例如:

```
template<typename T>
class Foo
{
    ...
}
```

3. 函数/类成员函数格式

每一个函数的返回类型声明必须与函数声明放在不同的行,例如:

```
void
bar ();
```

在函数实现的时候也一样,返回类型声明必须与函数声明放在不同的行,例如:

```
void
bar ()
{
    ...
}
```

或者:

```
void
Foo::bar ()
```



```
{  
  ...  
}
```

或者：

```
template<typename T> void  
Foo<T>::bar ()  
{  
  ...  
}
```

4. 花括号

花括号成对出现，与上一句代码另起一行定义，必须闭合才组成合理的程序块，例如：

```
if (a<b)  
{  
  ...  
}  
else  
{  
  ...  
}
```

下面的情况花括号可以省略，例如：

```
if (a<b)  
  x = 2 * a;
```

5. 空格格式

让我们再来强调一次，在 PCL 中的每一个代码块的标准缩进是两个空格，这里用单个空格来隔开函数/类成员函数名字与其参数列表，例如：

```
int  
exampleMethod (int example_arg);
```

如果在头文件内嵌套应用了命名空间名，需要将其缩进两个空格，例如：

```
namespace foo  
{  
  namespace bar  
  {
```

```
void
method (int my_var);
}
}
```

类和结构成员采用两个空格进行缩进,访问权限限定(public, private and protected)与类成员一级,而在其限定下的成员则需要缩进两个空格。例如:

```
namespace foo
{
class Bar
{
int i;
public:
int j;
protected:
void
baz ();
}
}
```

6. 自动格式化代码

PCL 提供下面一套规则文件通过多种不同的集成开发环境、编辑器等可以自动格式化编码。

(1) Emacs, 可以利用 PCL C/C++ 配置文件(<http://dev.pointclouds.org/attachments/download/748/pcl-c-style.el>), 下载并存储此文件, 再按如下操作进行:

打开 emacs 编辑器在 C/C++ hook 下添加下面的代码:

```
(load-file "/location/to/pcl-c-style.el")
(add-hook c-mode-common-hook pcl-set-c-style)
```

(2) Uncrustify 等其他配置。PCL 在快速发展和更新阶段, 笔者测试其他 IDE 上的配置文件不稳定, 关于其 IDE 的配置文件, 读者可以去网站看实时的帮助文件更新。

1.3 设计结构

1. 类和应用程序接口

对于 PCL 的大多数类而言, 调用接口(所有 public 成员)是不含公开成员变量的而只有采用两种成员方法(不排除有部分类有公开成员):

- (1) 固定的类型,它允许通过 `get/set` 修改或添加参数以及输入数据。
- (2) 实际实现功能的函数,例如运算、滤波、分割、配准等处理功能。

2. 参数传递

`get/set` 类型的方式遵循下面的规则:

(1) 如果大量的数据需要传送(常见的例子是在 PCL 中输入数据)优先采用 `boost` 共享指针,而不是传送实际的数据。

(2) 成对的 `get` 与 `set` 类型成员函数总是需要采用一致的数据类型。

(3) 对于 `get` 类型成员函数而言,如果只有一个参数需要被传递则会通过返回值,如果是两个或两个以上的参数需要传递,则通过引用方式进行传递。

对于运算、滤波、分割等类型的参数遵循以下规则:

(1) 无论传递数据的大小,返回参数最好是非指针型参数。

(2) 总是通过引用方式来传递输出参数。

1.4 编写新的PCL类

1.4.1 加入开源社区的好处

- (1) 别人以用户的代码为基础建立新的项目。
- (2) 学习其他人新的用法(例如,设计的时候没有考虑的非常有用的设计);
- (3) 无忧无虑的维护者身份(例如,可以休假一段时间,回来看到自己的代码还在更新维护中。其他贡献者会配置它以适应最新的平台、最新的编译器等);
- (4) 在社区的名声会提高——人人都喜欢受人敬仰。

1.4.2 建立文件结构

有**两种不同的方法**来建立文件结构：①分别编写代码，作为独立的 PCL 类在 PCL 代码树之外；②直接把文件建立在 PCL 代码目录树中，我们来阐述后者的操作方式，因为后者是最终结果有利于 PCL 库发展壮大，也是因为它有一点复杂（也就是，它包含几个附加的步骤）。对于前者，可以同样操作，只是不需要在 PCL 代码目录树中建立对应的文件组织形式，也不需要了解 CMake 的使用。

假设我们想要新的算法成为 PCL 滤波库的一部分，我们开始先在代码树目录 filters 下新建 3 个不同的文件：

- (1) include/pcl/filters/bilateral.h——包含所有的定义和声明；
- (2) include/pcl/filters/impl/bilateral.hpp——包含模板类的具体实现；
- (3) src/bilateral.cpp——包含具体的不同点类型的模板类实例化。

我们需要给新的类命名，把它称做 BilateralFilter，PCL 滤波器接口规定每个算法必须有两个声明和实现可供使用：一个操作 `PointCloud<T>`，一个操作 `PointCloud2`。本小节只讲前者操作 `PointCloud<T>` 的实现。

采取的是**先建立再填充**的思路

bilateral.h

bilateral.h头文件包含与BilateralFilter类相关的所有定义，这是一个最小的框架：

```
1  #ifndef PCL_FILTERS_BILATERAL_H_
2  #define PCL_FILTERS_BILATERAL_H_
3
4  #include <pcl/filters/filter.h>
5
6  namespace pcl
7  {
8      template<typename PointT>
9      class BilateralFilter : public Filter<PointT> //以滤波库为例，成为其一部分
10     {
11     };
12 }
13
14 #endif // PCL_FILTERS_BILATERAL_H_
```

C++ | [Copy Code](#)

bilateral.hpp

新建两个文件**bilateral.hpp**和**bilateral.cpp**。首先是**bilateral.hpp**：

```
1  #ifndef PCL_FILTERS_BILATERAL_IMPL_H_
2  #define PCL_FILTERS_BILATERAL_IMPL_H_
3
4  #include <pcl/filters/bilateral.h>
5
6  #endif // PCL_FILTERS_BILATERAL_IMPL_H_
```

这应该很简单。我们还没有为*BilateralFilter*声明任何方法，因此没有实现。

bilateral.cpp

下面我们也写下*bilateral.cpp*:

```
1  #include <pcl/filters/bilateral.h>
2  #include <pcl/filters/impl/bilateral.hpp>
```

官方说法：因为我们正在用PCL (1.x)编写模板化代码，其中模板参数是一个点类型(请参阅添加您自己的自定义PointT类型)，所以**我们希望显式地在bilateral.cpp中实例化最常见的用例**。这样用户在编译使用我们的BilateralFilter的代码时就不必花费额外的周期。为此，我们需要同时访问头文件(bilateral.h)和实现(bilateral.hpp)。

CMakeLists.txt

让我们将所有文件添加到PCL滤波类CMakeLists.txt文件中，以便启用构建。

```

1  # Find "set (srcs", and add a new entry there, e.g.,
2  set (srcs
3      src/conditional_removal.cpp
4      # ...
5      src/bilateral.cpp)
6  )
7
8  # Find "set (incs", and add a new entry there, e.g.,
9  set (incs
10     include pcl/${SUBSYS_NAME}/conditional_removal.h
11     # ...
12     include pcl/${SUBSYS_NAME}/bilateral.h
13     )
14
15  # Find "set (impl_incs", and add a new entry there, e.g.,
16  set (impl_incs
17     include/pcl/${SUBSYS_NAME}/impl/conditional_removal.hpp
18     # ...
19     include/pcl/${SUBSYS_NAME}/impl/bilateral.hpp
20     )

```

1.4.3 填写类的内容

填充类结构

如果您正确地编辑了上面的所有文件，那么使用新的filter类重新编译PCL应该没有问题。在本节中，我们将开始在每个文件中填充实际代码。让我们从***bilateral.cpp***文件开始，因为它的内容最短。

bilatera.cpp

如前所述，我们将**显式实例化和预编译*BilateralFilter*类的一些模板化专门化**。虽然这可能会增加PCL过滤库的编译时间，但是当用户在编写的代码中使用该类时，这将为他们省去处理和编译模板的麻烦。**最简单的方法是在*bilateral.cpp*文件中声明我们希望手工预编译的每个实例**，如下：

```

1  #include <pcl/point_types.h>
2  #include <pcl/filters/bilateral.h>
3  #include <pcl/filters/impl/bilateral.hpp>
4
5  template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZ>;
6  template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZI>;
7  template class PCL_EXPORTS pcl::BilateralFilter<pcl::PointXYZRGB>;
8  // ...

```

然而，随着PCL支持的点类型数量的增长，这变得非常麻烦。在PCL的多个文件中更新这个列表也是很痛苦的。因此，我们将使用一个名为**PCL_INSTANTIATE**的特殊宏，并改变上面的代码如下：

C++ |  Copy Code

```

1  #include <pcl/point_types.h>
2  #include <pcl/impl/instantiate.hpp>
3  #include <pcl/filters/bilateral.h>
4  #include <pcl/filters/impl/bilateral.hpp>
5
6  PCL_INSTANTIATE(BilateralFilter, PCL_XYZ_POINT_TYPES);

```

这个例子将为`point_types.h`文件中定义的所有XYZ点类型实例化一个`BilateralFilter`(有关更多信息, 请参见:[pcl: ' PCL_XYZ_POINT_TYPES<PCL_XYZ_POINT_TYPES >'](https://blog.csdn.net/u013235582/article/details/100060247#mulu3))。

<https://blog.csdn.net/u013235582/article/details/100060247#mulu3>)。

通过仔细查看示例中的代码:[双边过滤器](https://blog.csdn.net/u013235582/article/details/100530293#mulu3)

<https://blog.csdn.net/u013235582/article/details/100530293#mulu3> , 我们注意到一些结构, 比如 `cloud->points[point_id].intensity`。这表明我们的过滤器期望在`point`类型中存在一个**强度**字段。因此, 使用**PCL_XYZ_POINT_TYPES**将不起作用, 因为并不是所有定义的类型都有强度数据。事实上, 很容易注意到只有两种类型包含强度, 即:[pcl: ' PointXYZI](https://blog.csdn.net/u013235582/article/details/100060247#mulu3)

<https://blog.csdn.net/u013235582/article/details/100060247#mulu3> `pcl::PointXYZI` '和:[pcl: ' PointXYZINormal](https://blog.csdn.net/u013235582/article/details/100060247#mulu3) <https://blog.csdn.net/u013235582/article/details/100060247#mulu3>

`pcl::PointXYZINormal` '。因此, 我们替换**PCL_XYZ_POINT_TYPES**, 最终的`bilateral.cpp`文件变成:

C++ |  Copy Code

```

1  #include <pcl/point_types.h>
2  #include <pcl/impl/instantiate.hpp>
3  #include <pcl/filters/bilateral.h>
4  #include <pcl/filters/impl/bilateral.hpp>
5
6  PCL_INSTANTIATE(BilateralFilter, (pcl::PointXYZI)(pcl::PointXYZINormal));

```

注意, 目前我们还没有为`BilateralFilter`声明**PCL_INSTANTIATE**模板, 也没有实际实现抽象类中的纯虚函数:[pcl: ' pcl::Filter](http://www.pointclouds.org/documentation/tutorials/writing_new_classes.php#id10)

http://www.pointclouds.org/documentation/tutorials/writing_new_classes.php#id10 `pcl::Filter` ', 所以尝试编译代码将会导致以下错误:

C++ |  Copy Code

```

1  filters/src/bilateral.cpp:6:32: error: expected constructor, destructor, or type conversion
    before '(' token

```

(上述内容演讲者不太懂, 希望讨论; 下述内容具体内容见链接)

(个人感觉学习给开源社区共享PCL类的方法意义不大)

bilateral.h

首先声明构造函数及其成员变量，然后填充BilateralFilter类。由于双边滤波算法有两个参数，我们将把它们作为类成员存储，并为它们实现赋值和查询功能，以便与PCL 1.x的API模式兼容。

bilateral.hpp

这里我们需要实现两个方法，即`applyFilter`和`computePointWeight`。

详细的代码见：[【译】PCL官网教程翻译\(12\)：编写一个新的PCL类 - Writing a new PCL class](https://blog.csdn.net/u013235582/article/details/100530293)
<<https://blog.csdn.net/u013235582/article/details/100530293>>

1.5 PointT点类型

PCL中可用的PointT类型：

※※PointXYZ——成员变量：float x,y,z

PointXYZ是使用最常见的一个点数据类型，因为他之包含三维XYZ坐标信息，这三个浮点数附加一个浮点数来满足存储对齐，可以通过`points[i].data[0]`或`points[i].x`访问点X的坐标值
(由此推测y是`points[i].data[1]` z是`points[i].data[2]`)

```
1  union
2  {
3      float data[4];
4      struct
5      {
6          float x;
7          float y;
8          float z;
9      };
10 };
```

C++ |  Copy Code

PointXYZI——成员变量：float x,y,z,intensity

PointXYZI是一个简单的X Y Z坐标加intensity的point类型，是一个单独的结构体，并且满足存储对齐，由于point的大部分操作会把`data[4]`元素设置成0或1（用于变换）

不能让intensity与XYZ在同一个结构体中，如果这样的话其内容将会被覆盖，例如：两个点的点积会把第四个元素设置为0，否则点积没有意义

提问1：为何俩个点的点积要把第四个元素设置成0？

提问2：XYZI相比于XYZ多的这个intensity的具体作用是啥？


```
1  union
2  {
3      float data[4];
4      struct
5      {
6          float x;
7          float y;
8          float z;
9      };
10 };
11 union
12 {
13     struct
14     {
15         float intensity;
16     };
17     float data_c[4];
18 };
```

PointXYZRGBA——成员变量：float x,y,z;uint32_t rgba

除了RGBA信息被包含在一个整型变量中，其他的和PointXYZI类似
(不介绍PointXYZRGB了，rgb信息在一个浮点型中，将淘汰)

```
1  union
2  {
3      float data[4];
4      struct
5      {
6          float x;
7          float y;
8          float z;
9      };
10 };
11 union
12 {
13     struct
14     {
15         uint32_t rgba;
16     }
17     float data_c[4];
18 };
```

PointXY——成员变量：float x,y

简单的二维x-y结构代码

```
1  struct
2  {
3      float x;
4      float y;
5  };
```

InterestPoint——成员变量：float x,y,z,strength

除了strength表示关键点的强度测量值，其他的和PointXYZI一样

```
1  union
2  {
3      float data[4];
4      struct
5      {
6          float x;
7          float y;
8          float z;
9      };
10 };
11 union
12 {
13     struct
14     {
15         float strength;
16     };
17     float data_c[4];
18 };
```

※※Normal——成员变量：float data_n[3],normal[3],curvature

另一个常用的数据类型，Normal结构体表示给定点所在样本表面上的法线方向，以及对应曲率的测量值，例如访问法向量的第一个坐标可以通过points[i].data_n[0]或者points[i].normal[0]或者points[i]

再一次强调，曲率不能被存储在同一个结构体中，因为他会被普通的数据操作给覆盖掉

```
1  union
2  {
3      float data_n[4];
4      float normal[3];
5      struct
6      {
7          float normal_x;
8          float normal_y;
9          float normal_z;
10     };
11 }
12 union
13 {
14     struct
15     {
16         float curvature;
17     };
18     float data_c[4];
19 };
```

PointNormal——成员变量：float x,y,z; float normal[3],curvature

PointNormal是存储XYZ数据的point结构体，并且包括了采样点的法线和曲率

```
1  union
2  {
3      float data[4];
4      struct
5      {
6          float x;
7          float y;
8          float z;
9      };
10 };
11 union
12 {
13     float data_n[4];
14     float normal[3];
15     struct
16     {
17         float normal_x;
18         float normal_y;
19         float normal_z;
20     };
21 };
22 union
23 {
24     struct
25     {
26         float curvature;
27     };
28     float data_c[4];
29 };
```

后续的PointT类型不一一介绍，遇到&&需要的时候再去查询

1. 点云PCL库从入门到精通 P47-54



PCL1.8.1

2. 点云PCL学习教程 p60-67



PCL1.6.0

3. 更多点类型笔记: https://blog.csdn.net/qq_27806947/article/details/101067997
<https://blog.csdn.net/qq_27806947/article/details/101067997>

1.6 增加自己定义的点类型

要添加新的点类型，首先要定义它

```
1 struct MyPointType
2 {
3     float test;
4 };
```

C++ | [Copy Code](#)

然后，需要确保你的代码包含PCL中你希望你的新点类型MyPointType使用的特定类/算法的模板头实现。例如，假设你想使用pcl::PassThrough。你要做的就是

```
1 #define PCL_NO_PRECOMPILE
2 #include <pcl/filters/passthrough.h>
3 #include <pcl/filters/impl/passthrough.hpp>
4
5 // the rest of the code goes here
```

C++ | [Copy Code](#)

如果你的代码是库的一部分，它被其他人使用，那么尝试为你的MyPointType类型使用显式实例化也可能是有意义的，对于你公开的任何类(从PCL我们的外部PCL)。

实例

下面的代码片段示例创建一个新的point类型,其中包含XYZ数据(SSE衬垫),连同同一个test浮点数据型


```
1  #define PCL_NO_PRECOMPILE
2  #include <pcl/pcl_macros.h>
3  #include <pcl/point_types.h>
4  #include <pcl/point_cloud.h>
5  #include <pcl/io/pcd_io.h>
6
7  struct MyPointType
8  {
9      PCL_ADD_POINT4D;           // preferred way of adding a XYZ+padding
10     float test;
11     PCL_MAKE_ALIGNED_OPERATOR_NEW // make sure our new allocators are aligned
12 } EIGEN_ALIGN16;               // enforce SSE padding for correct memory alignment
13
14 POINT_CLOUD_REGISTER_POINT_STRUCT (MyPointType,           // here we assume a XYZ + "test"
    (as fields)
15                                     (float, x, x)
16                                     (float, y, y)
17                                     (float, z, z)
18                                     (float, test, test)
19 )
20
21
22 int
23 main (int argc, char** argv)
24 {
25     pcl::PointCloud<MyPointType> cloud;
26     cloud.points.resize (2);
27     cloud.width = 2;
28     cloud.height = 1;
29
30     cloud.points[0].test = 1;
31     cloud.points[1].test = 2;
32     cloud.points[0].x = cloud.points[0].y = cloud.points[0].z = 0;
33     cloud.points[1].x = cloud.points[1].y = cloud.points[1].z = 3;
34
35     pcl::io::savePCDFile ("test.pcd", cloud);
36 }
```

1.7 异常处理机制

本节我们主要讨论PCL在编写和应用过程中如何利用PCL的异常机制，提高程序的健壮性，首先从PCL开发者角度，解释如何定义和抛出自己的异常，最后从PCL使用者角度出发，解释用户如何捕获异常及处理异常。

1.7.1 开发者如何增加一个新的异常类

为了增强程序的健壮性，PCL提供了异常处理机制，作为PCL的开发者，需要通过自定义异常以及抛出异常，告诉调用者，在出现什么错误，并提示其如何处理，在PCL中，**规定任何一个新定义的PCL异常类都需要继承于PCLException** <http://docs.pointclouds.org/trunk/classpcl_1_1_p_c_l_exception.html>类，其具体定义在文件pcl/exceptions.h中，这样才能够使用PCL中其他和异常处理相关的机制和宏定义等。

C++ |  Copy Code

```

1  /** \class MyException
2      * \brief An exception that is thrown when I want it.
3      */
4  class PCL_EXPORTS MyException :public PCLException
5  {
6  public:
7      MyException (const std::string& error_description,
8      const std::string& file_name = "",
9      const std::string& function_name = "",
10     unsigned line_number =0) throw ()
11     : pcl::PCLException (error_description, file_name, function_name, line_number) { }
12 };

```

上面是一个最简单的自定义异常类，只定义了空的重构函数，但也足以可以完成对一般异常信息的抛出等功能了。

1.7.2 如何使用自定义的异常

在PCL中，为了方便开发者使用自定义的异常，定义下面宏定义：

C++ |  Copy Code

```

1  #define PCL_THROW_EXCEPTION (ExceptionName, message)
2  {
3      std::ostringstream s;
4      s << message;
5      throw ExceptionName (s.str (), __FILE__, "", __LINE__);
6  }

```

在异常抛出时使用就相当简单，添加下面的代码即可完成对异常的抛出：

C++ |  Copy Code

```

1  if (my_requirements != the_parameters_used_)
2      PCL_THROW_EXCEPTION (MyException, "my requirements are not met "<< the_parameters_used);

```

如此，通过宏调用，就可以实现对异常的抛出，此处抛出的异常包含异常信息、发生异常的文件名、以及异常发生的行号，当然这里异常信息可以包含很多信息，主要因为在宏定义中通过使用ostringstream的对象，开发者可以任意自定义自己的异常信息，例如添加运行过程当中一些重要的参

数名或变量名以及其值等，这样就给异常捕获者更多有用的信息，方便异常处理。这里需要说明的另一个问题是，以以下代码为例：

```
1  /** Function that does cool stuff
2      * \param nb number of points
3      * \throws MyException
4      */    //Doxygen格式的注释，在进行API文档生成时，会把该注释作为帮助信息，与函数说明放在一起。
5  void
6  myFunction (int nb);
```

C++

 Copy Code

PCL开发者在自定义函数中，如果使用了异常抛出，则需要添加Doxygen格式的注释，这样可以在最终的API文档中产生帮助信息，使用者通过文档可以知道，在调用该函数时需要捕获异常和进行异常处理，本例中，在用户调用myFunction(int nb)函数时，就需要捕获处理MyException异常。

1.7.3 异常的处理

作为PCL的使用者来说，为了能更好的处理异常你需要使用try... catch程序块。此处和其他异常处理基本一样，例如下面实例：

```
1  //在这里调用 myFunction时，可以捕获异常
2  try
3  {
4      myObject.myFunction (some_number);
5      //可以添加更多的其他异常捕获语句
6  }
7  // 针对try块捕获的MyException异常进行相应的处理
8  catch (pcl::MyException& e)
9  {
10     //MyException异常处理代码
11 }
12 //下面一段代码是对任何异常进行捕获处理的
13 #if 0
14 catch (exception& e)
15 {
16     // 发生异常的处理代码
17 }
18 #endif
```

C++

 Copy Code

异常的处理与其自身的上下文关系很大，并没有一般的规律可循，此处**列举一些最常用的处理方式**：

- 如果捕获的异常很关键那就终止运行
- 修改异常抛出函数的当前调用参数，在此重新调用该函数
- 抛出明确而对用户有意义的异常消息
- 采取继续运行该程序，这种选择慎用

