相同的返回却不同的操作

Device类继承于AudioDevice与VideoDevice;

在Device中:

AudioDevice \*audioDevice() { return this; } // 接口隔离

VideoDevice \*videoDevice() { return this; } // 接口隔离

1由于都返回this指针,但实际只能访问到对应的返回值,这就是接口隔离的核心所在。

2使用基本套路

单一原则的继承;

接口的返回。

代码：

|  |
| --- |
| #include <iostream>  using namespace std;  class AudioDevice {  public:  AudioDevice() {}  void open() { cout<<"Open Audio Device."<<endl; }  void close() { cout<<"Close Audio Device."<<endl; }  };  class VideoDevice {  public:  VideoDevice() {}  void open() { cout<<"Open Video Device."<<endl; }  void close() { cout<<"Close Video Device."<<endl; }  };  class Device : public AudioDevice, public VideoDevice {  public:  Audio() {}  AudioDevice \*audioDevice() { return this; } // 接口隔离  VideoDevice \*videoDevice() { return this; } // 接口隔离  };  int main(int argc, char \*argv[])  {  Device device;  device.audioDevice()->open();  device.audioDevice()->close();  device.videoDevice()->open();  device.videoDevice()->close();  return 0;  } |

口隔离原则和单一职责都是为了提高类的内聚性、降低它们之间的耦合性，体现了封装的思想，但两者是不同的：

* 单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离。
* 单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。

**接口隔离原则的优点**

接口隔离原则是为了约束接口、降低类对接口的依赖性，遵循接口隔离原则有以下 5 个优点。

1. 将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
2. 接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。
3. 如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。
4. 使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。
5. 能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

**接口隔离原则的实现方法**

在具体应用接口隔离原则时，应该根据以下几个规则来衡量。

* 接口尽量小，但是要有限度。一个接口只服务于一个子模块或业务逻辑。
* 为依赖接口的类定制服务。只提供调用者需要的方法，屏蔽不需要的方法。
* 了解环境，拒绝盲从。每个项目或产品都有选定的环境因素，环境不同，接口拆分的标准就不同深入了解业务逻辑。
* 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。  
  下面以worker类的实现例介绍接口隔离原则的应用。

【例1】 worker类的实现

程序代码如下：

|  |
| --- |
| #include <iostream>  using namespace std;  class IWorkable  {  public:  virtual void work() = 0;  };  class IFeedable  {  public:  virtual void eat() = 0;  };  // interface segregation principle - good example  class IWorker : public IFeedable, public IWorkable  {  };  class Worker : public IWorker  {  public:  void work()  {  cout << "working" << endl;  }  void eat()  {  cout << "eating in launch break" << endl;  }  };  class SuperWorker : public IWorker  {  public:  void work()  {  cout << "working much more" << endl;  }  void eat()  {  cout << "eating in launch break" << endl;  }  };  class Robot : public IWorkable  {  public:  void work()  {  cout << "Robot working" << endl;  }  };  class Manager  {  IWorkable \*worker;  public:  void setWorker(IWorkable \*w)  {  worker = w;  }  void manage()  {  worker->work();  }  };  int main()  {  IWorkable \*w1 = new Worker();  IWorkable \*sw1 = new SuperWorker();  IWorker \*w2;  Manager m1;  m1.setWorker(w1);  m1.manage();  //When worker wants he can eat  w2 = dynamic\_cast<IWorker \*>(w1);  w2->eat();  return 0;  } |